Xcode 12, Swift 5.3 and iOS 14 Ready

Mastering SwiftUI

SIMON NG

APPCODA

Table of Contents

Xcode 12, Swift 5.3 and iOS 14 Ready
Mastering
SwiftUI
SIMON NG
APPCODA

Published by AppCoda Limited

Preface

Frankly, I didn't expect Apple would announce anything big in WWDC 2019 that would completely change the way we build UI for Apple platforms. A year ago, Apple released a brand new framework called *SwiftUI*, along with the release of Xcode 11. The debut of SwiftUI was huge, really huge for existing iOS developers or someone who is going to learn iOS app building. It was unarguably the biggest change in iOS app development in recent years.

I have been doing iOS programming for nearly 10 years and already get used to developing UIs with UIKit. I love to use a mix of storyboards and Swift code for building UIs. However, whether you prefer to use Interface Builder or create UI entirely using code, the approach of UI development on iOS doesn't change much. Everything is still relying on the UIKit framework.

To me, SwiftUI is not merely a new framework. It's a paradigm shift that fundamentally changes the way you think about UI development on iOS and other Apple platforms. Instead of using the imperative programming style, Apple now advocates the declarative/functional programming style. Instead of specifying exactly how a UI component should be laid out and function, you focus on describing what elements you need in building the UI and what the actions should perform when programming in declarative style.

If you have worked with React Native or Flutter before, you will find some similarities between the programming styles and probably find it easier to build UIs in SwiftUI. That said, even if you haven't developed in any functional programming languages before, it would just take you some time to get used to the syntax. Once you manage the basics, you will love the simplicity of coding complex layouts and animations in SwiftUI.

This year, Apple has packed even more features and UI components into the SwiftUI framework, which comes alongside with Xcode 12. It just takes UI development on iOS, iPadOS, and macOS to the next level. You can develop some fancy animations with way less code, as compared to UIKit. Most importantly, the latest version of the SwiftUI framework makes it easier for developers to develop apps for Apple platforms. You will understand what I mean after you go through the book.

The release of SwiftUI doesn't mean that Interface Builder and UIKit are deprecated right away. They will still stay for many years to come. However, SwiftUI is the future of app development on Apple's platforms. To stay at the forefront of technological innovations, it's time to prepare yourself for this new way of UI development. And I hope this book will help you get started with SwiftUI development and build some amazing UIs.

Simon Ng
Founder of AppCoda

What You Will Learn in This Book

We will dive deep into the SwiftUI framework, teaching you how to work with various UI elements, and build different types of UIs. After going through the basics and understanding the usage of common components, we will put together with all the materials you've learned and build a complete app.

As always, we will explore SwiftUI with you by using the "Learn by doing" approach. This new book features a lot of hands-on exercises and projects. Don't expect you can just read the book and understand everything. You need to get prepared to write code and debug.

Audience

This book is written for both beginners and developers with some iOS programming experience. Even if you have developed an iOS app before, this book will help you understand this brand-new framework and the new way to develop UI. You will also learn how to integrate UIKit with SwiftUI.

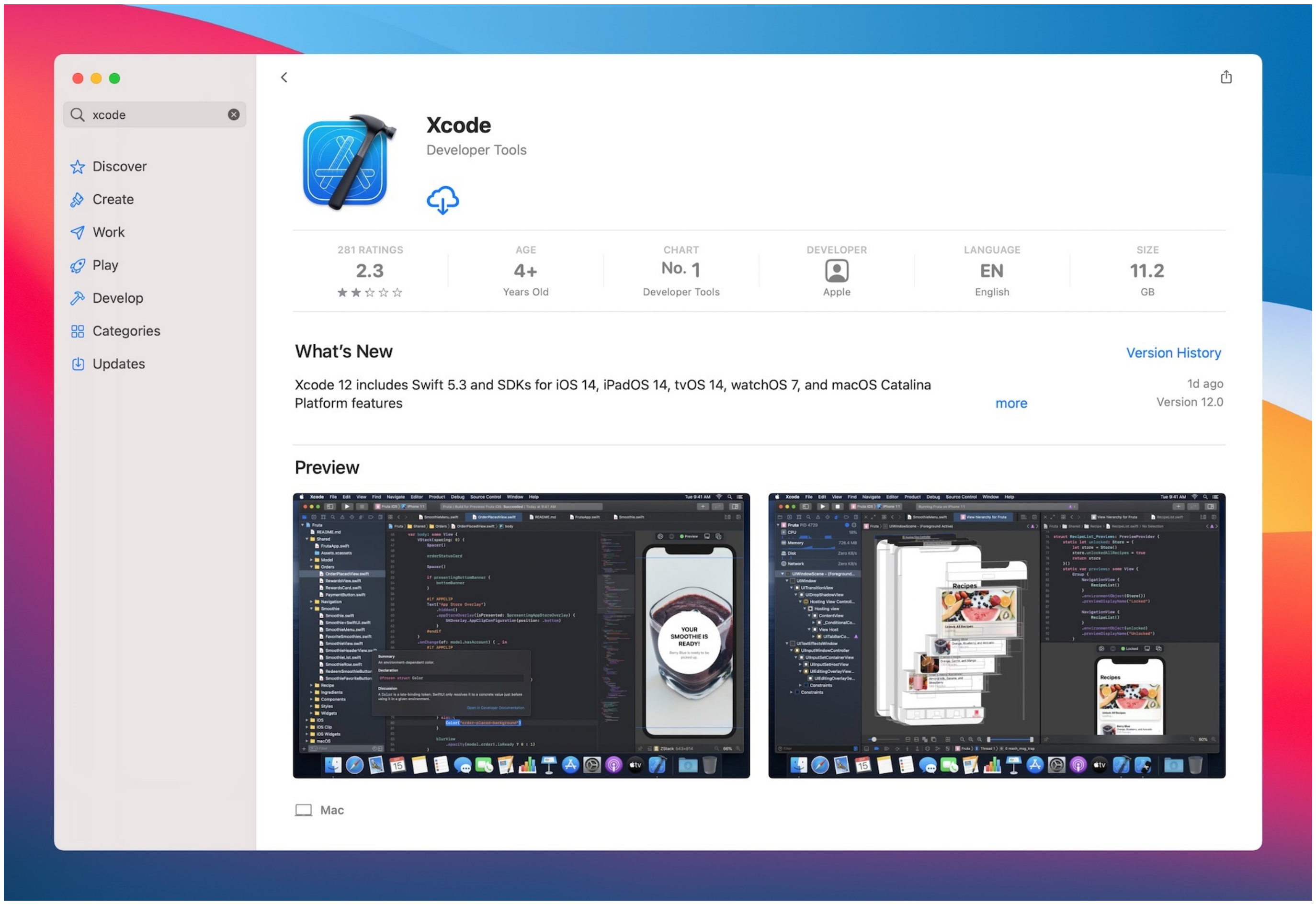
What You Need to Develop Apps with SwiftUI

Having a Mac is the basic requirement for iOS development. To use SwiftUI, you need to have a Mac installed with macOS Catalina (v10.15 or up) and Xcode 11 (or up). However, to properly follow the content of this book, you are required to have Xcode 12 installed.

If you are new to iOS app development, Xcode is an integrated development environment (IDE) provided by Apple. Xcode provides everything you need to kick start your app development. It already bundles the latest version of the iOS SDK (short for Software Development Kit), a built-in source code editor, graphic user interface (UI) editor, debugging tools and much more. Most importantly, Xcode comes with an iPhone (and iPad) simulator so you can test your app without the real devices. With Xcode 12, you can instantly preview the result of your SwiftUI code.

Installing Xcode

To install Xcode 12, go up to the Mac App Store and download it. Simply search "Xcode" and click the "Get" button to download it. At the time of this writing, the latest official version of Xcode is 12.0. Once you complete the installation process, you will find Xcode in the Launchpad.


xcode
Discover
Create
Work
Play
Develop
Categories
Updates
Xcode
Developer Tools
281 RATINGS
2.3
AGE
4+
Years Old
CHART
No. 1
Developer Tools
DEVELOPER
Apple
LANGUAGE
EN
English
SIZE
11.2
GB
What's New
Version History
Xcode 12 includes Swift 5.3 and SDKs for iOS 14, iPadOS 14, tvOS 14, watchOS 7, and macOS Catalina Platform features
more
1d ago
Version 12.0
Preview
Mac

Frequestly Asked Questions about SwiftUI

I got quite a lot of questions from new comers when the SwiftUI framework was first announced. These questions are some of the common ones that I want to share with you. And I hope the answers will give you a better idea about SwiftUI.

1. ***Do I need to learn Swift before learning SwiftUI?***

   Yes, you still need to know the Swift programming language before using SwiftUI. SwiftUI is just a UI framework written in Swift. Here, the keyword is UI, meaning that the framework is designed for building user interfaces. However, for a complete application, other than UI, there are many other components such as network components for connecting to remote server, data components for loading data from internal database, business logic component for handling the flow of data, etc. All these components are not built using SwiftUI. So, you should be knowledgeable about Swift and SwiftUI, as well as, other built-in frameworks (e.g. Map) in order to build an app.

2. ***Should I learn SwiftUI or UIKit?***

   The short answer is Both. That said, it all depends on your goals. If you target to become a professional iOS developer and apply for a job in iOS development, you better equip yourself with knowledge of SwiftUI and UIKit. Over 99% of the apps published on the App Store were built using UIKit. To be considered for hire, you should be very knowledgeable with UIKit because most companies are still using the framework to build the app UI. However, like any technological advancement, companies will gradually adopt SwiftUI in new projects. This is why you need to learn both to increase your employment opportunities.

   On the other hand, if you just want to develop an app for your personal or side project, you can develop it entirely using SwiftUI. However, since SwiftUI is very new, it doesn't cover all the UI components that you can find in UIKit. In some cases, you may also need to integrate UIKit with SwiftUI.

3. ***Do I need to learn auto layout?***

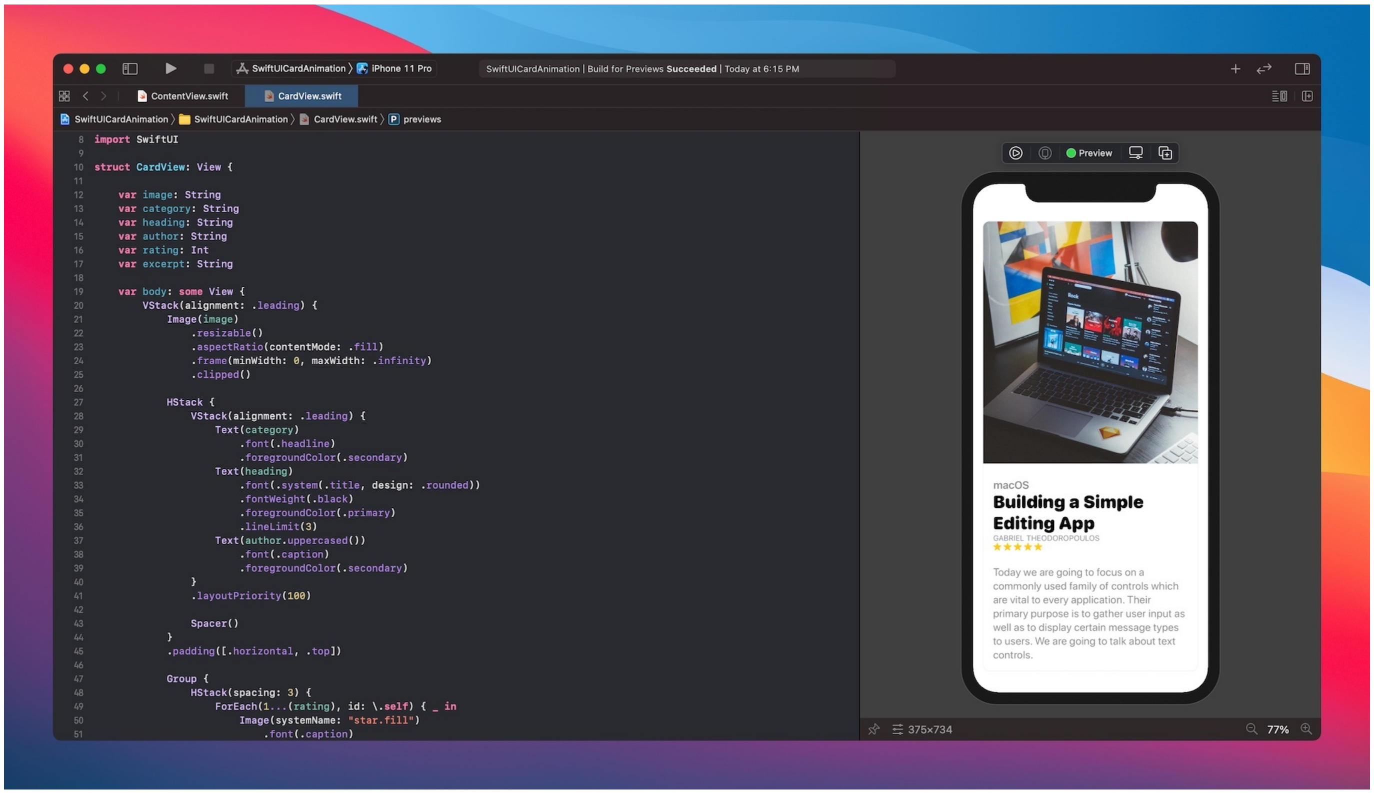
This may be a good news to some of you. Many beginners find it hard to work with auto layout. With SwiftUI, you no longer need to define layout constraints. Instead, you use stacks, spacers, and padding to arrange the layout.

Chapter 1
Introduction to SwiftUI

In WWDC 2019, Apple surprised every developer by announcing a completely new framework called *SwiftUI*. It doesn't just change the way you develop iOS apps. This is the biggest shift in the Apple developer's ecosystem (including iPadOS, macOS, tvOS, and watchOS) since the debut of Swift.

> SwiftUI is an innovative, exceptionally simple way to build user interfaces across all Apple platforms with the power of Swift. Build user interfaces for any Apple device using just one set of tools and APIs.
>
> - Apple (https://developer.apple.com/xcode/swiftui/)

Developers have been debating for a long time whether we should use Storyboards or build the app UI programmatically. The introduction of SwiftUI is Apple's answer. With this brand new framework, Apple offers developers a new way to create user interfaces. Take a look at the figure below and have a glance at the code.


SwiftUICardAnimation 〉 iPhone 11 Pro
SwiftUICardAnimation | Build for Previews Succeeded | Today at 6:15 PM
ContentView.swift
CardView.swift
SwiftUICardAnimation 〉 SwiftUICardAnimation 〉 CardView.swift 〉 previews
Preview
375×734
77%

```swift
import SwiftUI

struct CardView: View {

    var image: String
    var category: String
    var heading: String
    var author: String
    var rating: Int
    var excerpt: String

    var body: some View {
        VStack(alignment: .leading) {
            Image(image)
                .resizable()
                .aspectRatio(contentMode: .fill)
                .frame(minWidth: 0, maxWidth: .infinity)
                .clipped()

            HStack {
                VStack(alignment: .leading) {
                    Text(category)
                        .font(.headline)
                        .foregroundColor(.secondary)
                    Text(heading)
                        .font(.system(.title, design: .rounded))
                        .fontWeight(.black)
                        .foregroundColor(.primary)
                        .lineLimit(3)
                    Text(author.uppercased())
                        .font(.caption)
                        .foregroundColor(.secondary)
                }
                .layoutPriority(100)

                Spacer()
            }
            .padding([.horizontal, .top])

            Group {
                HStack(spacing: 3) {
                    ForEach(1...(rating), id: \.self) { _ in
                        Image(systemName: "star.fill")
                            .font(.caption)
```

macOS
Building a Simple Editing App
GABRIEL THEODOROPOULOS
★★★★★
Today we are going to focus on a commonly used family of controls which are vital to every application. Their primary purpose is to gather user input as well as to display certain message types to users. We are going to talk about text controls.

Figure 1. Programming in SwiftUI

With the release of SwiftUI, which is bundled in Xcode 11 (or later), you can now develop the app's UI with a declarative Swift syntax. What that means to you is that the UI code is easier and more natural to write. Compared with the existing UI frameworks like UIKit, you can create the same UI with way less code.

The preview function has always been a weak point of Xcode. While you can preview simple layouts in Interface Builder, you usually can't preview the complete UI until the app is loaded onto the simulators. With SwiftUI, you get immediate feedback of the UI you are coding. For example, you add a new record to a table, Xcode renders the UI change on the fly in a preview canvas. If you want to preview how your UI looks in dark mode, you just need to change an option. This instant preview feature simply makes UI development a breeze and iteration much faster.

Not only does it allow you to preview the UI, the new canvas also lets you design the user interface visually using drag and drop. What's great is that Xcode automatically generates the SwiftUI code as you add the UI component visually. The code and the UI are always in sync. This is a feature Apple developers anticipated for a long time.

In this book, you will dive deep into SwiftUI, learn how to layout the built-in components, and create complex UIs with the framework. I know some of you may already have experience in iOS development. Let me first walk you through the major differences between the existing framework that you're using (e.g. UIKit) and SwiftUI. If you are completely new to iOS development or even have no programming experience, you can use the information as a reference or even skip the following sections. I don't want to scare you away from learning SwiftUI, it is an awesome framework for beginners.

Declarative vs Imperative Programming

Like Java, C++, PHP, and C#, Swift is an imperative programming language. SwiftUI, however, is proudly claimed as a declarative UI framework that lets developers create UI in a declarative way. What does the term "declarative" mean? How does it differ from imperative programming? Most importantly, how does this change affect the way you code?

If you are new to programming, you probably don't need to care about the difference because everything is new to you. However, if you have some experience in Object-oriented programming or have developed with UIKit before, this paradigm shift affects how you think about building user interfaces. You may need to unlearn some old concepts and relearn new ones.

So, what's the difference between imperative and declarative programming? If you go to Wikipedia and search for the terms, you will find these definitions:

> In computer science, **imperative programming** is a programming paradigm that uses statements that change a program's state. In much the same way that the imperative mood in natural languages expresses commands, an imperative program consists of commands for the computer to perform.
>
> In computer science, **declarative programming** is a programming paradigm—a style of building the structure and elements of computer programs—that expresses the logic of a computation without describing its control flow.

It's pretty hard to understand the actual difference if you haven't studied Computer Science. Let me explain the difference this way.

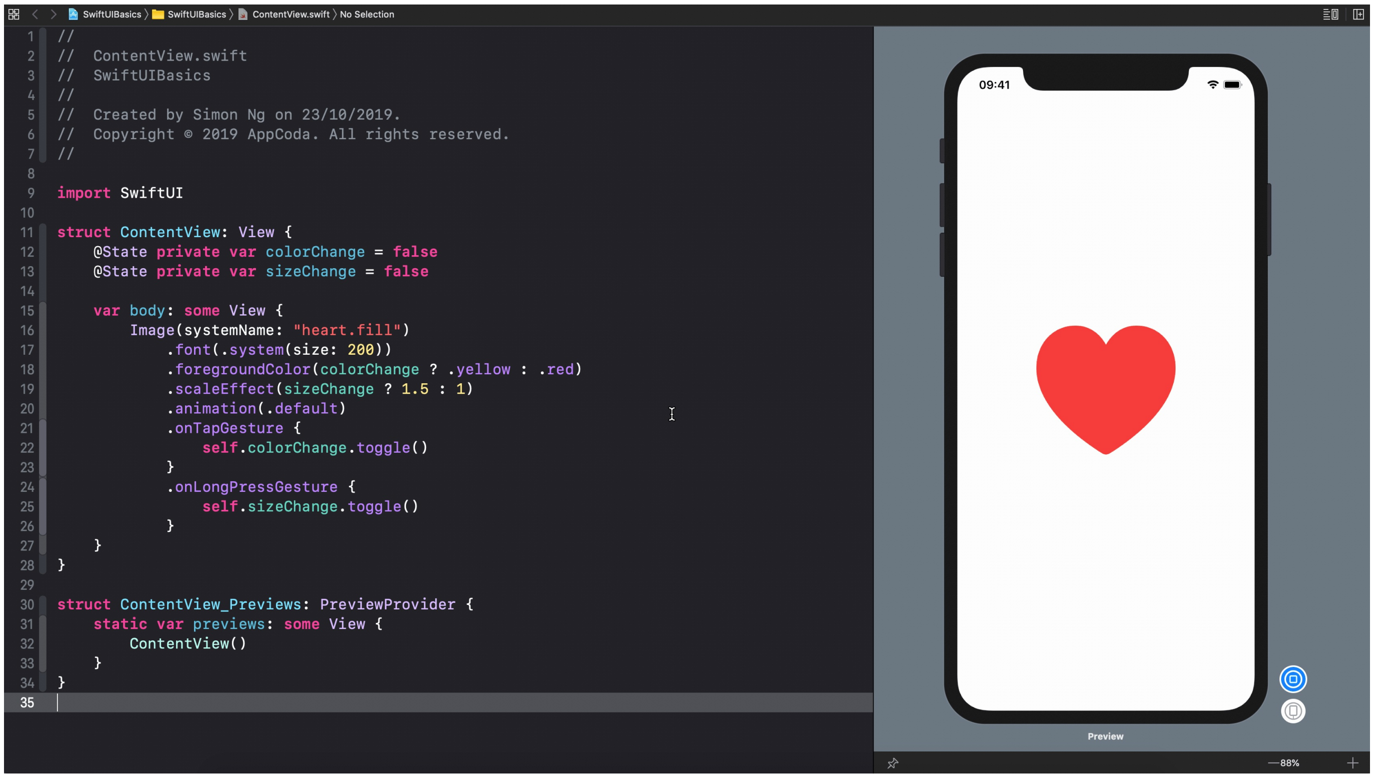
Instead of focusing on programming, let's talk about cooking a pizza (or any dishes you like). Let's assume you are instructing someone else (a helper) to prepare the pizza, you can either do it *imperatively* or *declaratively*. To cook the pizza imperatively, you tell your helper each of the instructions clearly like a recipe:

1. Heat the over to 550°F or higher for at least 30 minutes
2. Prepare one-pound of dough
3. Roll out the dough to make a 10-inch circle
4. Spoon the tomato sauce onto the center of the pizza and spread it out to the edges
5. Place toppings (including onions, sliced mushrooms, pepperoni, cooked sausage, cooked bacon, diced peppers and cheese) on top of the sauce
6. Bake the pizza for 5 minutes

On the other hand, if you cook it in a declarative way, you do not need to specify the step by step instructions but just describe how you would like the pizza cooked. Thick or thin crust? Pepperoni and bacon, or just a classic Margherita with tomato sauce? 10-inch or 16-inch? The helper will figure out the rest and cook the pizza for you.

That's the core difference between the term imperative and declarative. Now back to UI programming. Imperative UI programming requires developers to write detailed instructions to layout the UI and control its states. Conversely, declarative UI programming lets developers describe what the UI looks like and what you want to respond when a state changes.

The declarative way of coding would make the code much easier to read and understand. Most importantly, the SwiftUI framework allows you to write way less code to create a user interface. Say, for example, you are going to build a heart button in an app. This button should be positioned at the center of the screen and is able to detect touches. If a user taps the heart button, its color is changed from red to yellow. When a user taps and holds the heart, it scales up with an animation.


SwiftUIBasics 〉 SwiftUIBasics 〉 ContentView.swift 〉 No Selection
09:41
Preview
88%

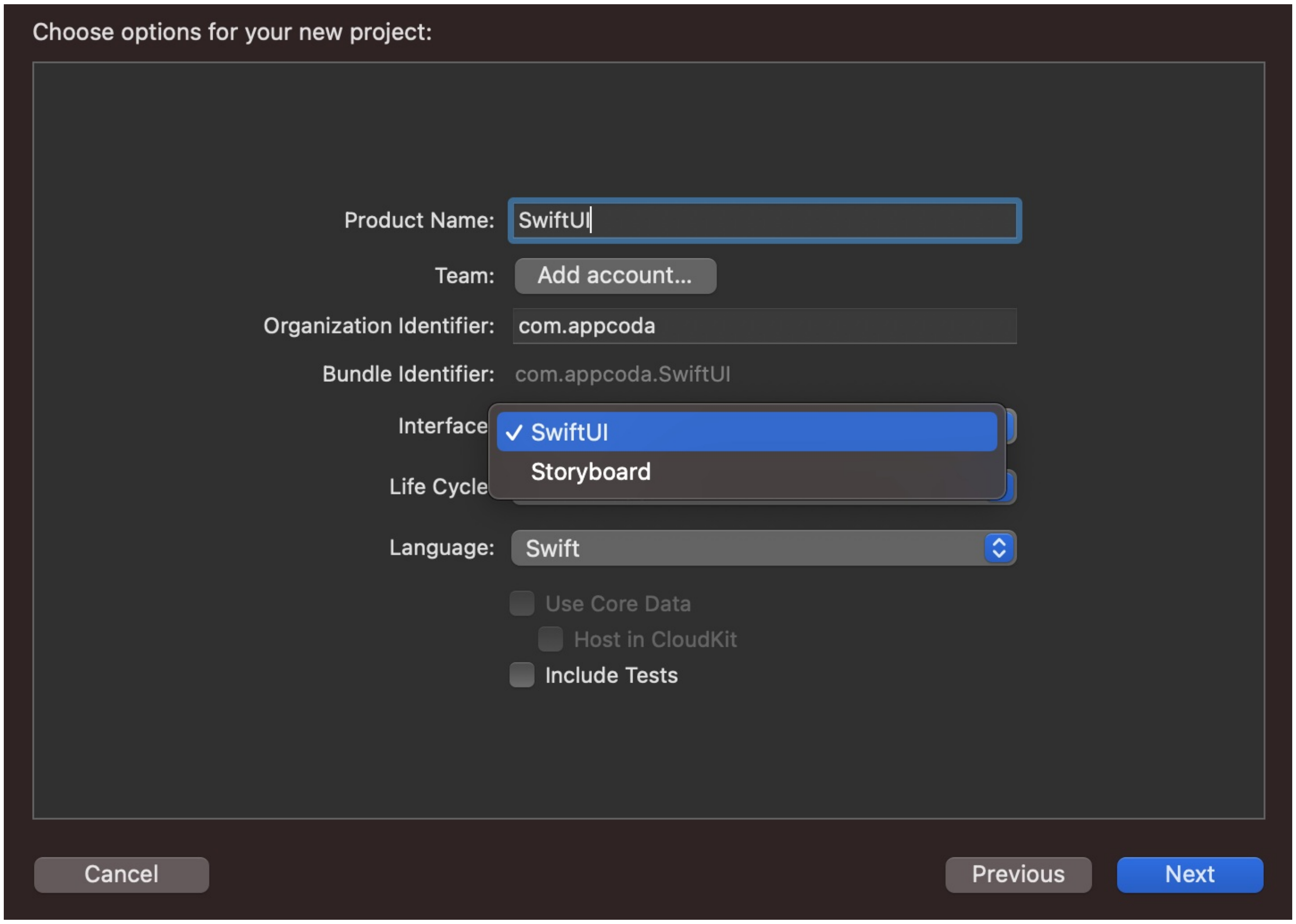
```swift
//
//  ContentView.swift
//  SwiftUIBasics
//
//  Created by Simon Ng on 23/10/2019.
//  Copyright © 2019 AppCoda. All rights reserved.
//

import SwiftUI

struct ContentView: View {
    @State private var colorChange = false
    @State private var sizeChange = false

    var body: some View {
        Image(systemName: "heart.fill")
            .font(.system(size: 200))
            .foregroundColor(colorChange ? .yellow : .red)
            .scaleEffect(sizeChange ? 1.5 : 1)
            .animation(.default)
            .onTapGesture {
                self.colorChange.toggle()
            }
            .onLongPressGesture {
                self.sizeChange.toggle()
            }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 2. The implementation of an interactive heart button

Take a look at figure 2. That's the code you need to implement the heart button. In around 20 lines of code, you create an interactive button with a scale animation. This is the power of the SwiftUI declarative UI framework.

No more Interface Builder and Auto Layout

In Xcode 11/12, you can choose between SwiftUI and Storyboard to build the user interface. If you have built an app before, you may use Interface Builder to layout the UI on the storyboard. With SwiftUI, Interface Builder and storyboards are completely gone. It's replaced by a code editor and a preview canvas like the one shown in figure 2. You write the code in the code editor. Xcode then renders the user interface in real time and displays it in the canvas.


Choose options for your new project:

Product Name: SwiftUI

Team: Add account...

Organization Identifier: com.appcoda

Bundle Identifier: com.appcoda.SwiftUI

Interface: ✓ SwiftUI / Storyboard

Life Cycle

Language: Swift

Use Core Data

Host in CloudKit

Include Tests

Figure 3. User interface option in Xcode

Auto layout has always been one of the hard topics when learning iOS development. With SwiftUI, you no longer need to learn how to define layout constraints and resolve the conflicts. Now you compose the desired UI by using stacks, spacers, and padding. We will discuss this concept in detail in later chapters.

The Combine Approach

Other than storyboards, the view controller is gone too. For new comers, you can ignore what a view controller is. But if you are an experienced developer, you may find it strange that SwiftUI doesn't use a view controller as a central building block for talking to the view and the model.

Communications and data sharing between views are now done via another brand new framework called Combine. This new approach completely replaces the role of the view controller in UIKit. In this book, we will also cover the basics of Combine and how to use it to handle UI events.

Learn Once, Apply Anywhere

While this book focuses on building UIs for iOS, everything you learn here is applicable to other Apple platforms such as watchOS. Prior to the launch of SwiftUI, you used platform-specific UI frameworks to develop the user interface. You used AppKit to write UIs for macOS apps. To develop tvOS apps, you relied on TVUIKit. And, for watchOS apps, you used WatchKit.

With SwiftUI, Apple offers developers a unified UI framework for building user interfaces on all types of Apple devices. The UI code written for iOS can be easily ported to your watchOS/macOS/watchOS app without modifications or with very minimal modifications. This is made possible thanks to the declarative UI framework.

Your code describes how the user interface looks. Depending on the platform, the same piece of code in SwiftUI can result in different UI controls. For example, the code below declares a toggle switch:

```
Toggle(isOn: $isOn) {
    Text("Wifi")
        .font(.system(.title))
        .bold()
}.padding()
```

For iOS and iPadOS, the toggle is rendered as a switch. On the other hand, SwiftUI renders the control as a checkbox for macOS.

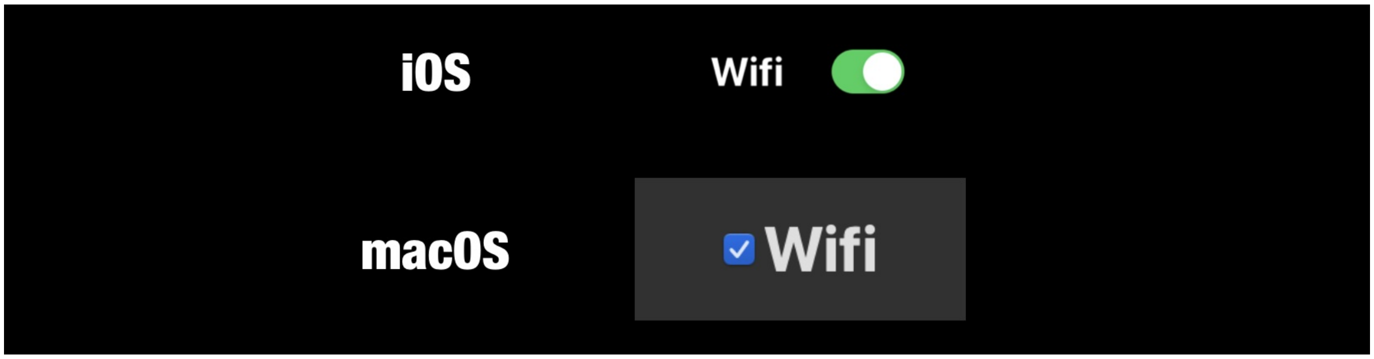


Figure 4. Toggle on macOS and iOS

The beauty of this unified framework is that you can reuse most of the code on all Apple platforms without making any changes. SwiftUI does the heavy lifting to render the corresponding controls and layout.

However, don't consider SwiftUI as a "Write once, run anywhere" solution. As Apple stressed in a WWDC talk, that's not the goal of SwiftUI. So, don't expect you can turn a beautiful app for iOS into a tvOS app without any modifications.

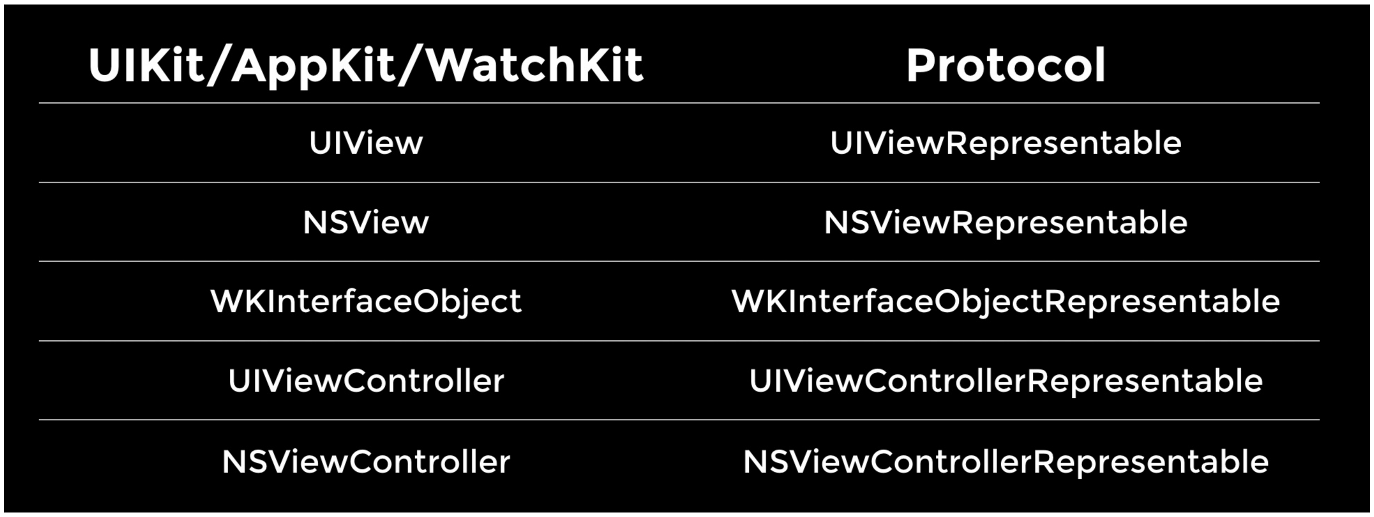
> There are definitely going to be opportunities to share code along the way, just where it makes sense. And so we think it's kind of important to think about SwiftUI less as write once and run anywhere and more like learn once and apply anywhere.
>
> - WWDC Talk (SwiftUI On All Devices)

While the UI code is portable across Apple platforms, you still need to provide specialization that targets for a particular type of device. You should always review each edition of your app to make sure the design is right for the platform. That said, SwiftUI already saves you a lot of time from learning another platform-specific framework, plus you should be able to reuse most of the code.

Interfacing with UIKit/AppKit/WatchKit

Can I use SwiftUI on my existing projects? I don't want to rewrite the entire app which was built on UIKit.

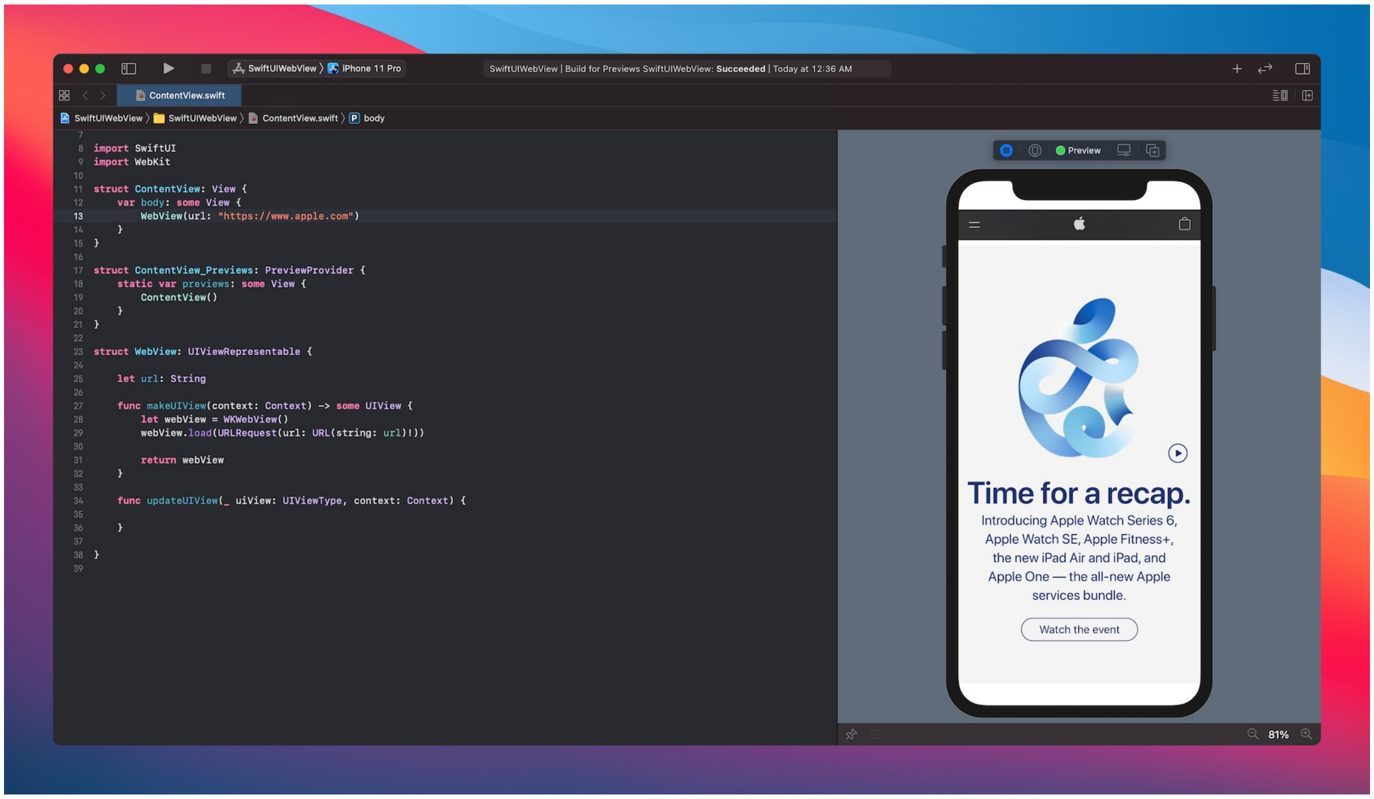
SwiftUI is designed to work with the existing frameworks like UIKit for iOS and AppKit for macOS. Apple provides several representable protocols for you to adopt in order to wrap a view or controller into SwiftUI.



UIKit/AppKit/WatchKit	Protocol
UIView	UIViewRepresentable
NSView	NSViewRepresentable
WKInterfaceObject	WKInterfaceObjectRepresentable
UIViewController	UIViewControllerRepresentable
NSViewController	NSViewControllerRepresentable

Figure 5. The Representable protocols for existing UI frameworks

Say, you have a custom view developed using UIKit, you can adopt the `UIViewRepresentable` protocol for that view and make it into SwiftUI. Figure 6 shows the sample code of using `WKWebView` in SwiftUI.


SwiftUIWebView 〉 iPhone 11 Pro
SwiftUIWebView | Build for Previews SwiftUIWebView: Succeeded | Today at 12:36 AM
ContentView.swift
SwiftUIWebView 〉 SwiftUIWebView 〉 ContentView.swift 〉 body
Preview
Time for a recap.
Introducing Apple Watch Series 6, Apple Watch SE, Apple Fitness+, the new iPad Air and iPad, and Apple One — the all-new Apple services bundle.
Watch the event
81%

```swift
import SwiftUI
import WebKit

struct ContentView: View {
    var body: some View {
        WebView(url: "https://www.apple.com")
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}

struct WebView: UIViewRepresentable {

    let url: String

    func makeUIView(context: Context) -> some UIView {
        let webView = WKWebView()
        webView.load(URLRequest(url: URL(string: url)!))

        return webView
    }

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

    }

}
```

Figure 6. Porting MKMapView to SwiftUI

Use SwiftUI for Your Next Project

Every time when a new framework is released, people usually ask, "Is the framework ready for my next project? Should I wait a little bit longer?"

Though SwiftUI is still new to most developers, Now is the right time to learn and incorporate the framework into your new project. Along with the release of Xcode 12, Apple has made the SwiftUI framework more stable and feature-rich. If you have some personal projects or side projects for personal use or at work, there is no reason why you shouldn't try out SwiftUI.

Having said that, you need to consider carefully whether you should apply SwiftUI to your commercial projects. One major drawback of SwiftUI is that the device must run at a minimum on iOS 13, macOS 10.15, tvOS 13, or watchOS 6. If your app requires support for lower versions of the platform (e.g. iOS 12), you may need to wait at least a year before adopting SwiftUI.

At the time of this writing, SwiftUI has been officially released for more than a year. The debut of Xcode 12 has brought us more UI controls and new APIs for SwiftUI. In terms of features, you can't compare it with the existing UI frameworks (e.g. UIKit), which has been available for years. Some features (e.g. changing the separator style in table views) which are present in the old framework may not be available in SwiftUI. You may need to develop some solutions to work around the issue. This is something you have to take into account when adopting SwiftUI in production projects.

SwiftUI is very new. It will take time to grow into a mature framework, but what's clear is that SwiftUI is the future of application development for Apple platforms. Even though it may not yet be applicable to your production projects, I recommended you start a side project and explore the framework. Once you try out SwiftUI and master its use, you will enjoy developing UIs in a declarative way.

Chapter 2
Getting Started with SwiftUI and Working with Text

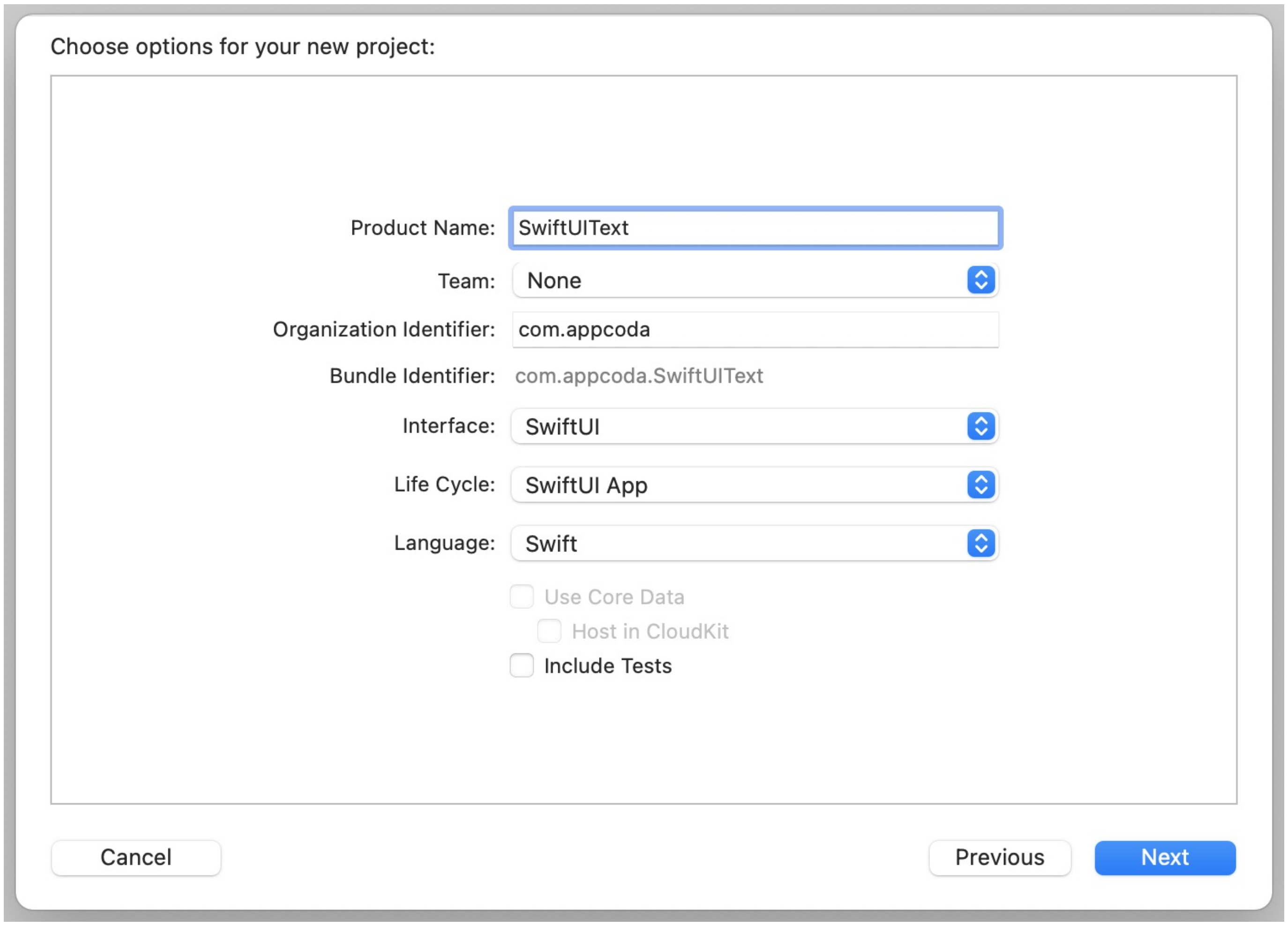
If you've worked with UIKit before, the `Text` control in SwiftUI is very similar to `UILabel` in UIKit. It's a view for you to display one or multiple lines of text. This `Text` control is non-editable but is useful for presenting read-only information on screen. For example, you want to present an on-screen message, you can use `Text` to implement it.

In this chapter, I'll show you how to work with `Text` to present information. You'll also learn how to customize the text with different colors, fonts, backgrounds and apply rotation effects.

Creating a New Project for Playing with SwiftUI

First, fire up Xcode 12 and create a new project using the *App* template under the iOS category. Apple has revamped some of the project templates. If you have used the older version of Xcode before, the *Single Application* template is now replaced with the *App* template.

Choose *Next* to proceed to the next screen and type the name of the project. I set it to *SwiftUIText* but you're free to use any other name. For the organization name, you can set it to your company or organization. The organization identifier is a unique identifier of your app. Here I use *com.appcoda* but you should set it to your own value. If you have a website, set it to your domain in reverse domain name notation.


Choose options for your new project:

Product Name: SwiftUIText

Team: None

Organization Identifier: com.appcoda

Bundle Identifier: com.appcoda.SwiftUIText

Interface: SwiftUI

Life Cycle: SwiftUI App

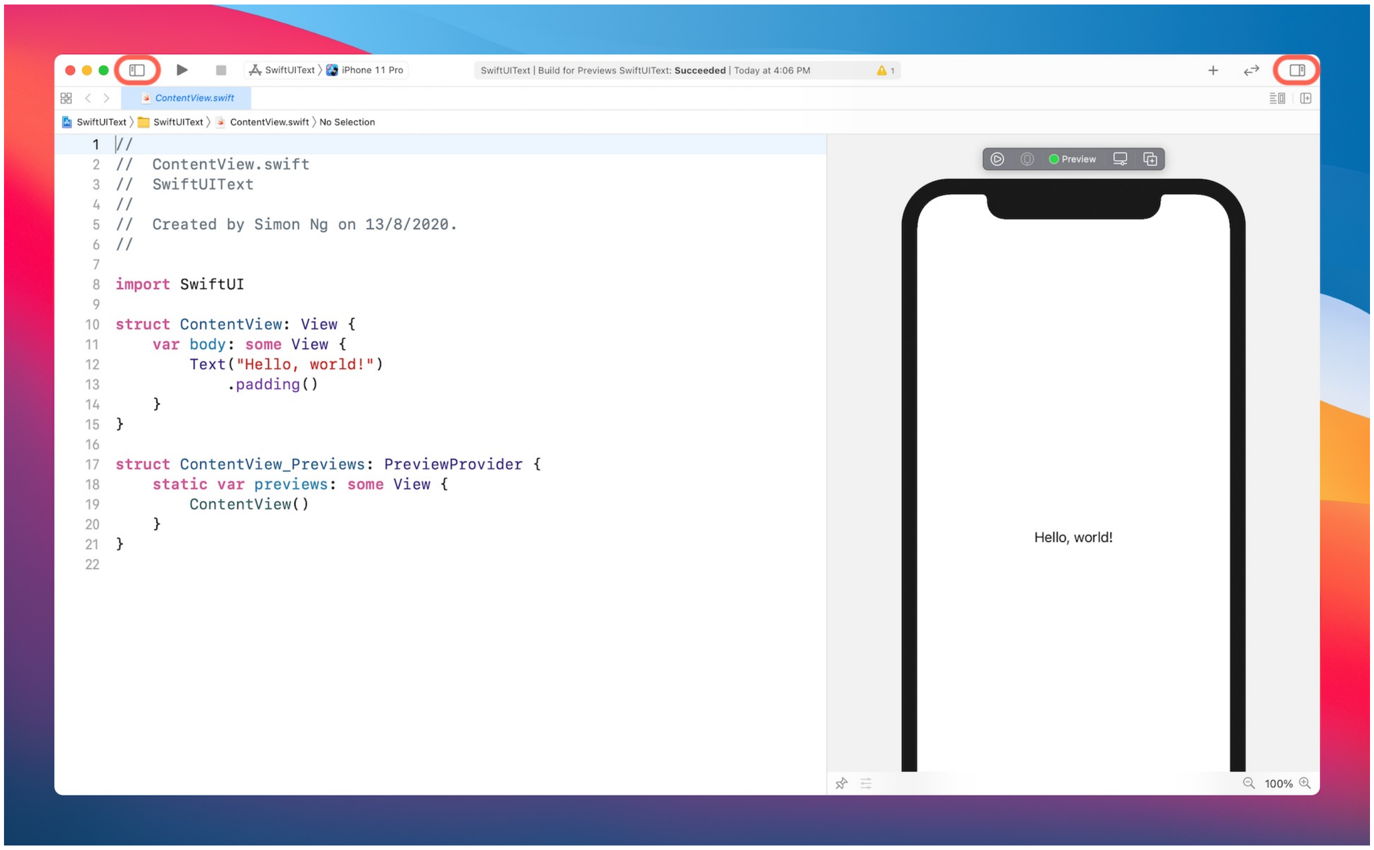
Language: Swift

Figure 1. Creating a new project

To use SwiftUI, you have to choose *SwiftUI* in the User Interface option. Xcode 12 introduces the *Life Cycle* option that you can choose between *SwiftUI App* and *UIKit App Delegate*. By default, it's set to *SwiftUI App*. This allows you to build the entire app using SwiftUI, which was not possible in Xcode 11. For this book, we will use this option for our demo apps. However, in case that your app needs to support iOS 13, you can fallback to the *UIKit App Delegate*.

Click *Next* and choose a folder to create the project. Once you save the project, Xcode should load the `ContentView.swift` file and display a design/preview canvas. If you can't see the design canvas, you can go up to the Xcode menu and choose *Editor* > *Canvas* to enable it. To give yourself more space for writing code, you can hide both the project navigator and the inspector (see figure 2).

By default, Xcode generates some SwiftUI code for `ContentView.swift`. However, the preview canvas doesn't render the app preview. You have to click the *Resume* button in order to see the preview. After you click the button, Xcode renders the preview in a simulator that you choose in the simulator selection (e.g. iPhone 11 Pro).



SwiftUIText 〉 iPhone 11 Pro
SwiftUIText | Build for Previews SwiftUIText: Succeeded | Today at 4:06 PM
ContentView.swift
SwiftUIText 〉 SwiftUIText 〉 ContentView.swift 〉 No Selection
Preview
Hello, world!
100%

```
//
//  ContentView.swift
//  SwiftUIText
//
//  Created by Simon Ng on 13/8/2020.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, world!")
            .padding()
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

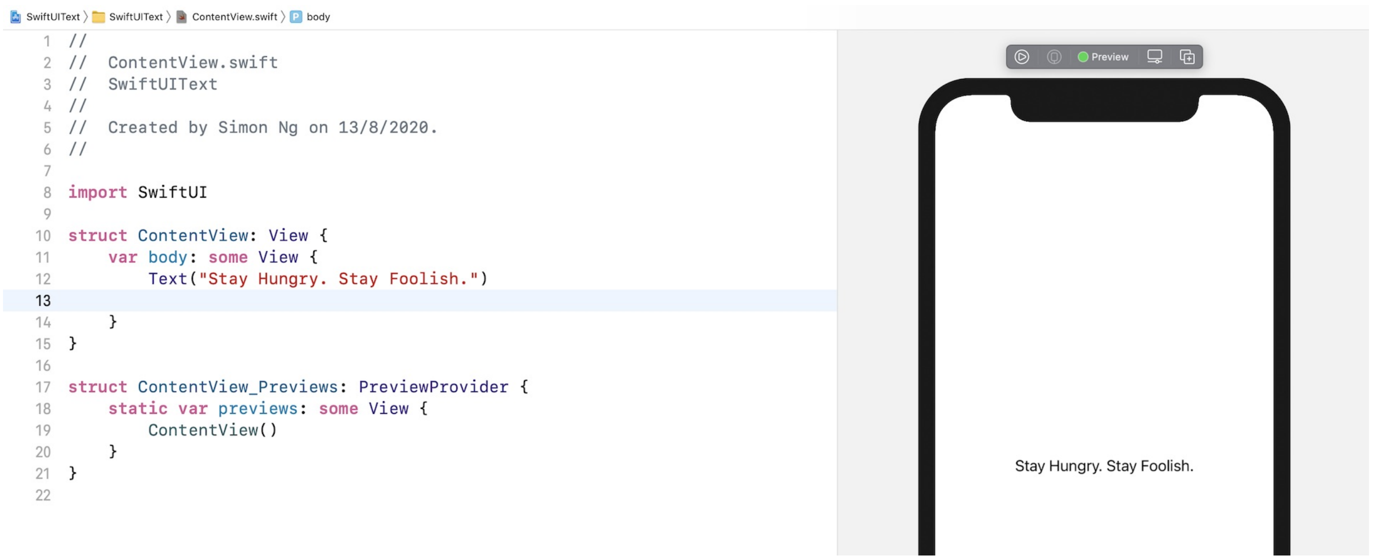
Figure 2. The code editor and the canvas

Displaying Simple Text

The sample code generated in `ContentView` already shows you how to display a single line of text. You initialize a `Text` object and pass to it the text (e.g. *Hello World*) to display like this:

```
Text("Hello World")
```

The preview canvas should display *Hello World* on screen. This is the basic syntax for creating a text view. You're free to change the text to whatever value you want and the canvas should show you the change instantaneously.


SwiftUIText 〉 SwiftUIText 〉 ContentView.swift 〉 body
Preview
Stay Hungry. Stay Foolish.

```
//
//  ContentView.swift
//  SwiftUIText
//
//  Created by Simon Ng on 13/8/2020.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Stay Hungry. Stay Foolish.")

    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}

```

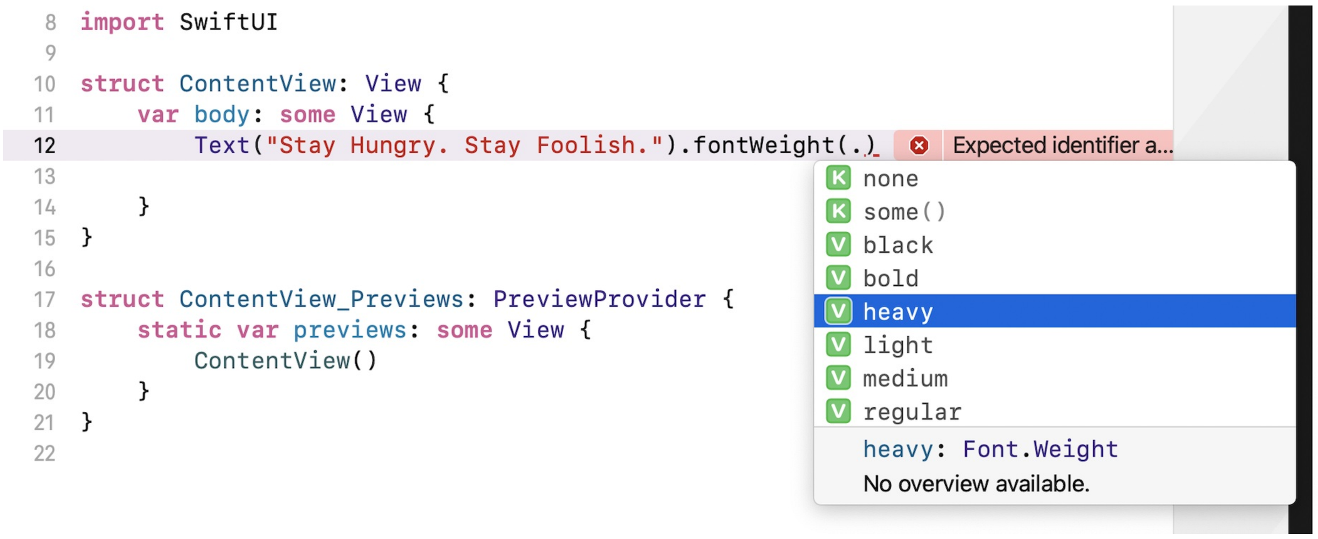
Figure 3. Changing the text

Changing the Font Type and Color

In SwiftUI, you can change the properties (e.g. color, font, weight) of a control by calling methods that are known as *Modifiers*. Let's say, you want to bold the text. You can use the modifier `fontWeight` and specify your preferred font weight (e.g. `.bold`) like this:

```
Text("Stay Hungry. Stay Foolish.").fontWeight(.bold)
```

You access the modifier by using the dot syntax. Whenever you type a dot, Xcode will show you the possible modifiers or values you can use. For example, you will see various font weight options when you type a dot in the `fontWeight` modifier. You can choose `bold` to bold the text. If you want to make it even bolder, use `heavy` or `black`.



```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Stay Hungry. Stay Foolish.").fontWeight(.)
        
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Expected identifier a...

```
K none
K some()
V black
V bold
V heavy
V light
V medium
V regular
heavy: Font.Weight
No overview available.
```

Figure 4. Choosing your preferred font weight

By calling `fontWeight` with the value `.bold`, it actually returns to you a new view that has the bolded text. What is interesting in SwiftUI is that you can further chain this new view with other modifiers. Say, you want to make the bolded text a little bit bigger, you write the code like this:

```
Text("Stay Hungry. Stay Foolish.").fontWeight(.bold).font(.title)
```

Since we may chain multiple modifiers together, we usually write the code above in the following format:

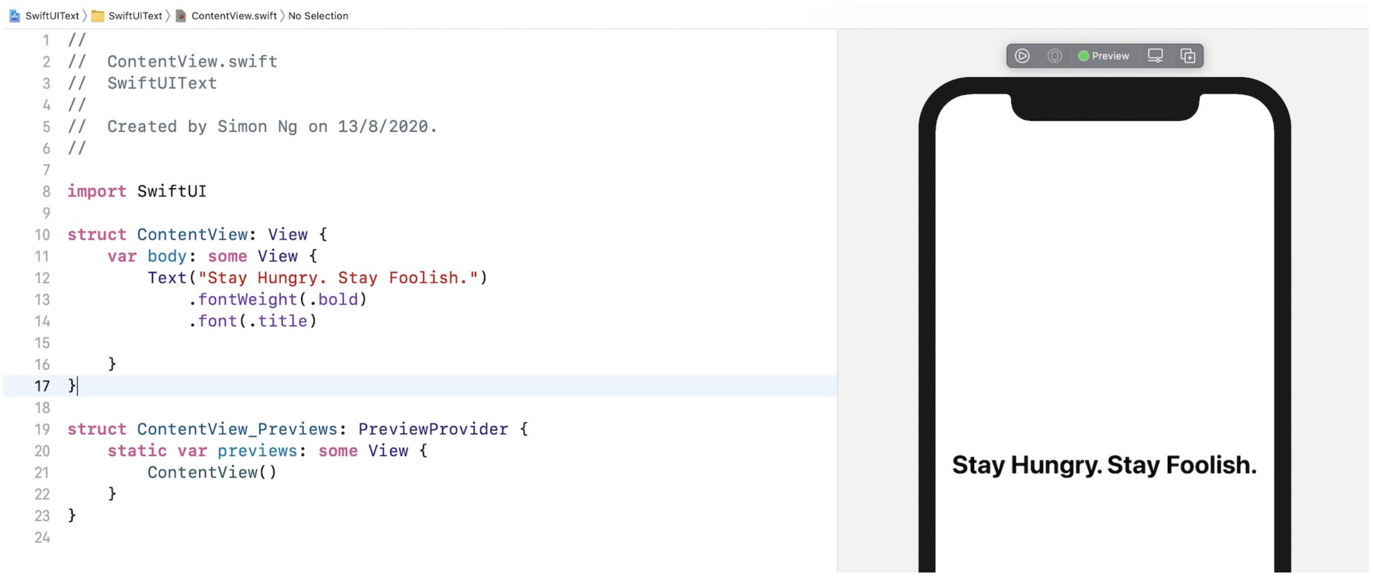
```
Text("Stay Hungry. Stay Foolish.")
    .fontWeight(.bold)
    .font(.title)
```

The functionality is the same but I believe you'll find the code above more easy to read. We will continue to use this coding convention for the rest of this book.

The `font` modifier lets you change the font properties. In the code above, we specify the *title* font type in order to enlarge the text. SwiftUI comes with several built-in text styles including *title*, *largeTitle*, *body*, etc. If you want to further increase the font size, replace

`.title` with `.largeTitle`.

Note: You can always to refer the documentation (https://developer.apple.com/documentation/swiftui/font) to find out all the supported values of the `font` modifier.


SwiftUIText 〉 SwiftUIText 〉 ContentView.swift 〉 No Selection
Preview

```
//
//  ContentView.swift
//  SwiftUIText
//
//  Created by Simon Ng on 13/8/2020.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Stay Hungry. Stay Foolish.")
            .fontWeight(.bold)
            .font(.title)

    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Stay Hungry. Stay Foolish.

Figure 5. Changing the font type

You can also use the `font` modifier to specify the font design. Let's say, you want the font to be rounded. You can write the `font` modifier like this:

```
.font(.system(.title, design: .rounded))
```

Here you specify to use the system font with `title` text style and `rounded` design. The preview canvas should immediately respond to the change and show you the rounded text.

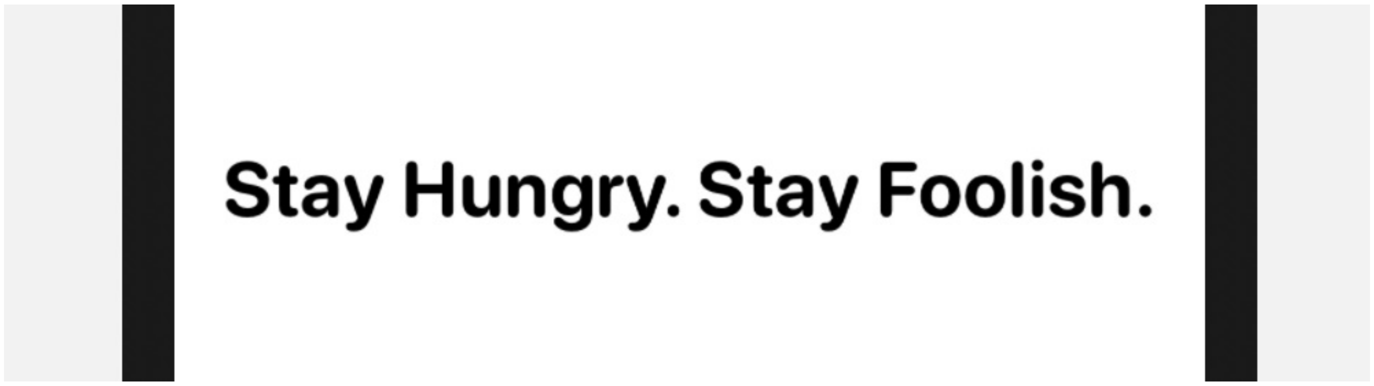


Figure 6. Using the rounded font design

Dynamic Type is a feature of iOS that automatically adjusts the font size in reference to the user's setting (Settings > Display & Brightness > Text Size). In other words, when you use text styles (e.g. `.title`), the font size will be varied and your app will scale the text automatically, depending on the user's preference.

To use a fixed-size font, write the code like this:

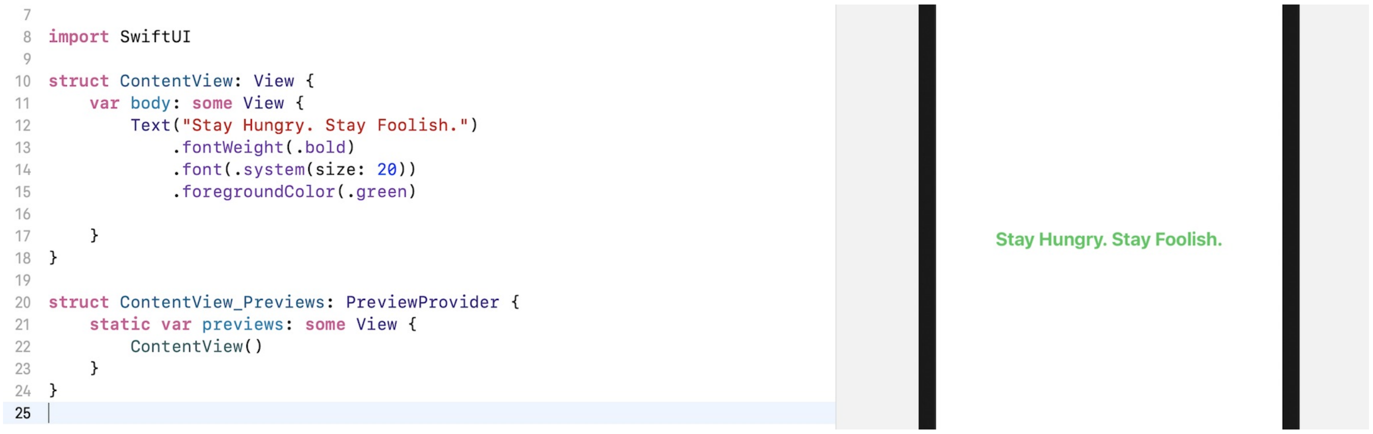
```
.font(.system(size: 20))
```

This tells the system to use a fixed font size of 20 points.

You can chain other modifiers to further customize the text. Let's change the font color. To do that, you use the `foregroundColor` modifier like this:

```
.foregroundColor(.green)
```

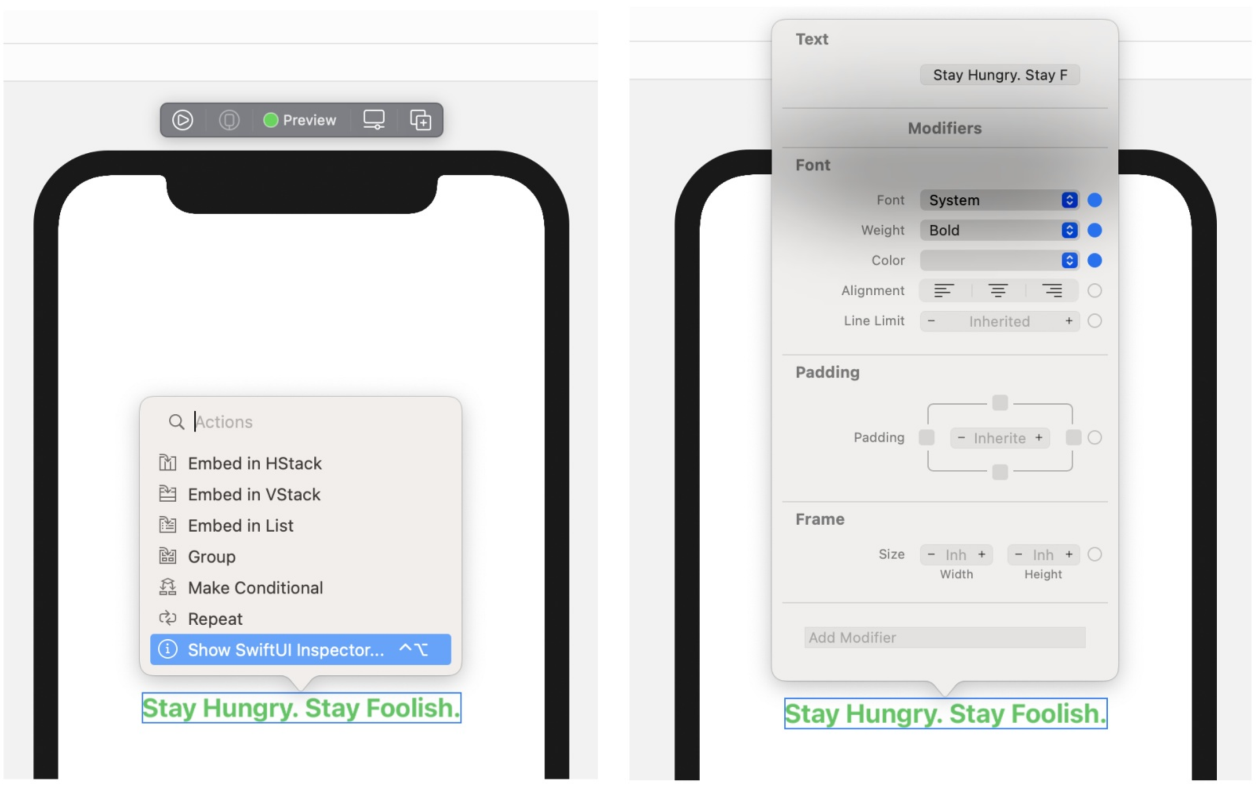
The `foregroundColor` modifier accepts a value of `Color`. Here we specify `.green`, which is a built-in color. You may use other built-in values like `.red`, `.purple`, etc.



```swift
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Stay Hungry. Stay Foolish.")
            .fontWeight(.bold)
            .font(.system(size: 20))
            .foregroundColor(.green)

    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 7. Changing the font color

While I prefer to customize the properties of a control using code, you can also use the design canvas to edit them. Hold the command key and click the text to bring up a pop-over menu. Choose *Show SwiftUI Inspector* and then you can edit the text/font properties. What is great is that the code will update automatically when you make changes to the font properties.



Text: Stay Hungry. Stay F

Font
Font: System
Weight: Bold
Color
Alignment
Line Limit: Inherited

Padding
Padding: Inherite

Frame
Size: Inh (Width), Inh (Height)

Figure 8. Using the Inspect feature to edit the properties of the text

Using Custom Fonts

By default, all text is displayed using the system font. If you want to use other fonts, you can replace the following line of code:

```
.font(.system(size: 20))
```

With:

```
.font(.custom("Helvetica Neue", size: 25))
```

Instead of using `.system`, the code above uses `.custom` and specifies the preferred font name. Font names can be found in the application "Font Book". You can open Finder > Application and click *Font Book* to launch the app.

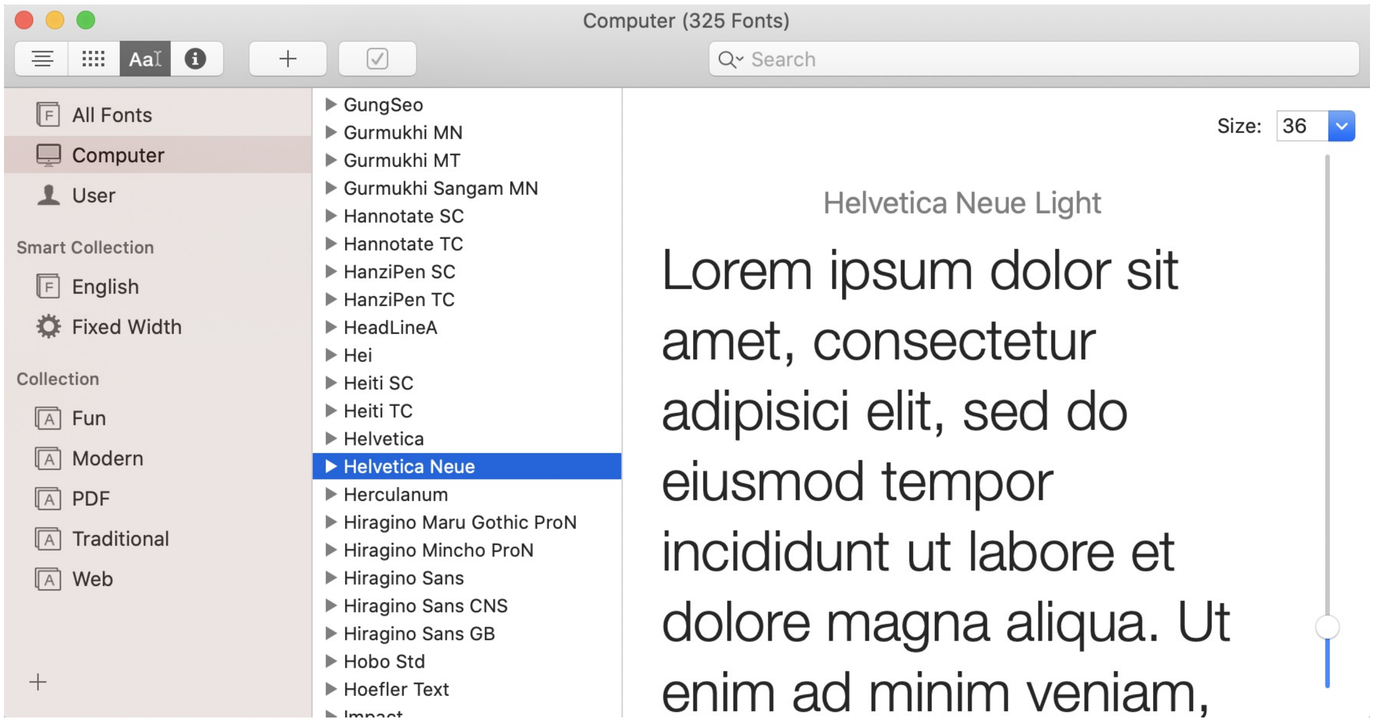


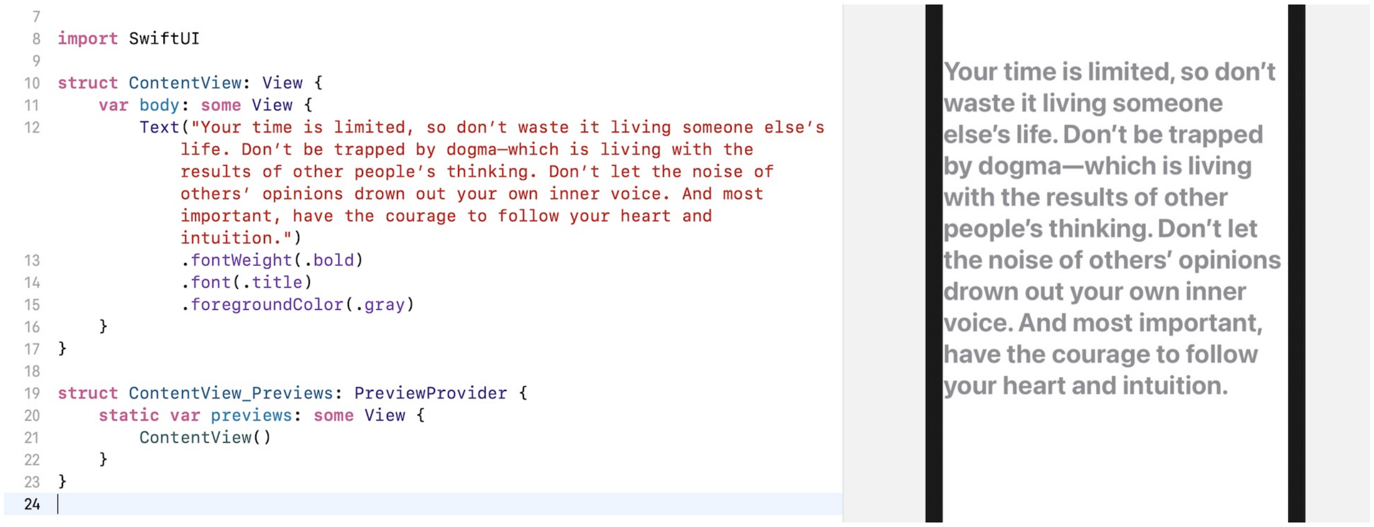
Figure 9. Font Book

Working with Multiline Text

`Text` supports multiple lines by default, so it can display a paragraph of text without using any additional modifiers. Replace your current code with the following:

```
Text("Your time is limited, so don’t waste it living someone else’s life. Don’t be
 trapped by dogma—which is living with the results of other people’s thinking. Don
’t let the noise of others’ opinions drown out your own inner voice. And most impo
rtant, have the courage to follow your heart and intuition.")
    .fontWeight(.bold)
    .font(.title)
    .foregroundColor(.gray)
```

You're free to replace the paragraph of text with your own text. Just make sure it's long enough. Once you have made the change, the design canvas will render a multiline text label.



```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Your time is limited, so don't waste it living someone else's
            life. Don't be trapped by dogma—which is living with the
            results of other people's thinking. Don't let the noise of
            others' opinions drown out your own inner voice. And most
            important, have the courage to follow your heart and
            intuition.")
            .fontWeight(.bold)
            .font(.title)
            .foregroundColor(.gray)
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Your time is limited, so don't waste it living someone else's life. Don't be trapped by dogma—which is living with the results of other people's thinking. Don't let the noise of others' opinions drown out your own inner voice. And most important, have the courage to follow your heart and intuition.

Figure 10. Display multiline text

To center align the text, insert the `multilineTextAlignment` modifier after the `.foreground` modifier and set its value to `.center` like this:

```
.multilineTextAlignment(.center)
```

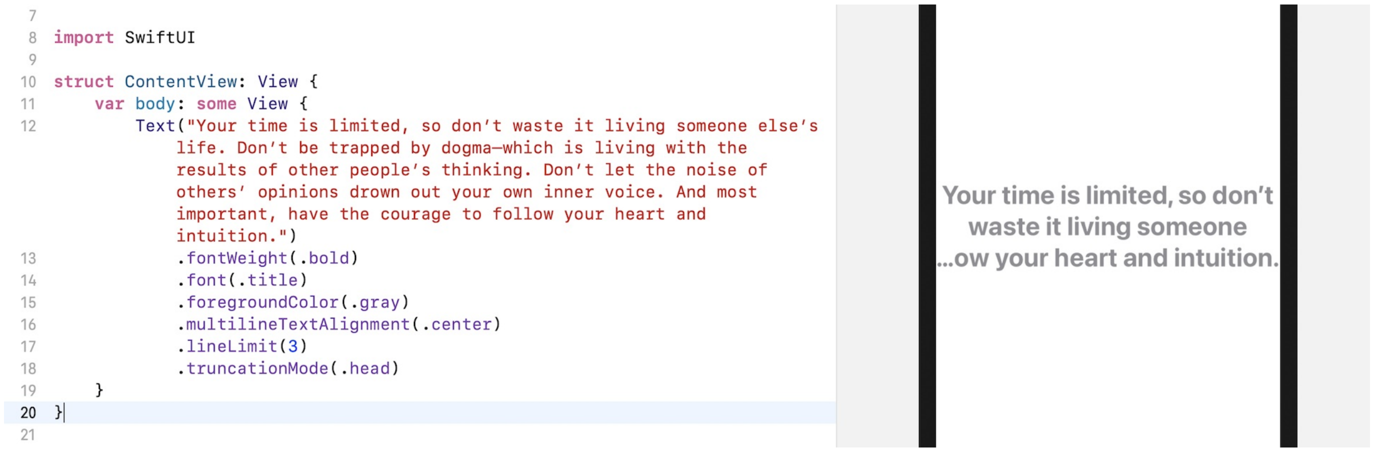
In some cases, you may want to limit the number of lines to a certain number. You use the `lineLimit` modifier to control it. Here is an example:

```
.lineLimit(3)
```

Another modifier, `truncationMode` specifies where to truncate the text within the text view. You can truncate at the beginning, middle, or end of the text view. By default, the system is set to use tail truncation. To modify the truncation mode of the text, you use the `truncationMode` modifier and set its value to `.head` or `.middle` like this:

```
.truncationMode(.head)
```

After the change, your text should look like the figure below.



```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Your time is limited, so don’t waste it living someone else’s
            life. Don’t be trapped by dogma—which is living with the
            results of other people’s thinking. Don’t let the noise of
            others’ opinions drown out your own inner voice. And most
            important, have the courage to follow your heart and
            intuition.")
            .fontWeight(.bold)
            .font(.title)
            .foregroundColor(.gray)
            .multilineTextAlignment(.center)
            .lineLimit(3)
            .truncationMode(.head)
    }
}
```

Your time is limited, so don’t
waste it living someone
...ow your heart and intuition.

Figure 11. Using the .head truncation mode

Earlier, I mentioned that the `Text` control displays multiple lines by default. The reason is that the SwiftUI framework has set a default value of `nil` for the `lineLimit` modifier. You can change the value of `.lineLimit` to `nil` and see the result:

```
.lineLimit(nil)
```

Setting the Padding and Line Spacing

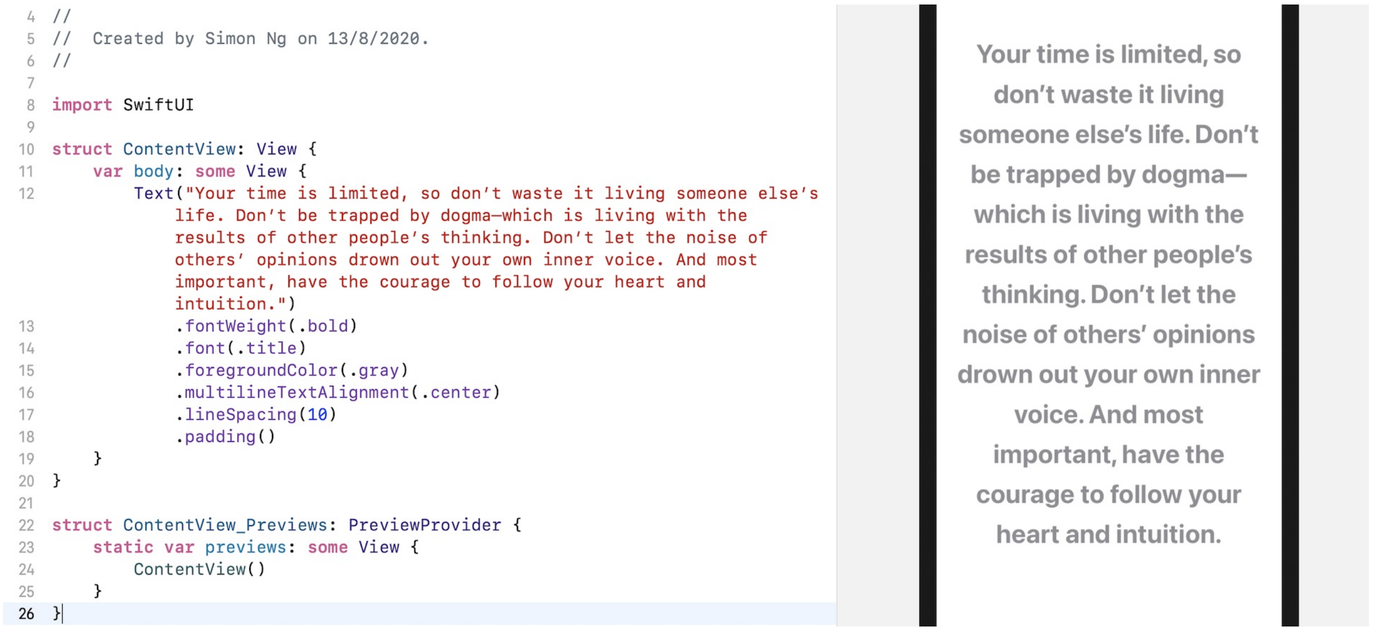
Normally the default line spacing is good enough for most situations. To alter the default setting, you adjust the line spacing by using the `lineSpacing` modifier.

```
.lineSpacing(10)
```

As you see, the text is too close to the left and right side of the edges. To give it some more space, you can use the `padding` modifier, which adds some extra space to each side of the text. Insert the following line of code after the `lineSpacing` modifier:

```
.padding()
```

Your design canvas should now look like this:



```
//
//  Created by Simon Ng on 13/8/2020.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Your time is limited, so don’t waste it living someone else’s
            life. Don’t be trapped by dogma—which is living with the
            results of other people’s thinking. Don’t let the noise of
            others’ opinions drown out your own inner voice. And most
            important, have the courage to follow your heart and
            intuition.")
            .fontWeight(.bold)
            .font(.title)
            .foregroundColor(.gray)
            .multilineTextAlignment(.center)
            .lineSpacing(10)
            .padding()
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Your time is limited, so don’t waste it living someone else’s life. Don’t be trapped by dogma—which is living with the results of other people’s thinking. Don’t let the noise of others’ opinions drown out your own inner voice. And most important, have the courage to follow your heart and intuition.

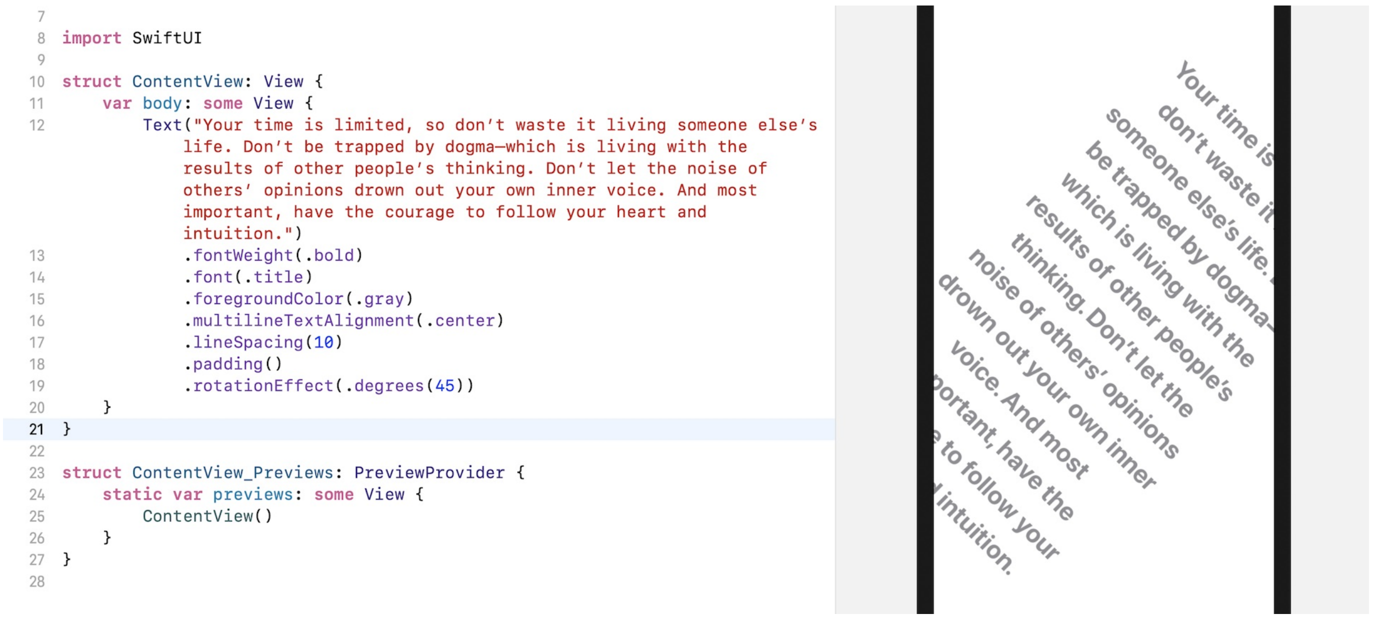
Figure 12. Setting the padding and line spacing of the text

Rotating the Text

The SwiftUI framework provides a modifier to let you easily rotate the text. You use the `rotateEffect` modifier and pass the degree of rotation like this:

```
.rotationEffect(.degrees(45))
```

If you insert the above line of code after `padding()`, you will see the text is rotated by 45 degrees.



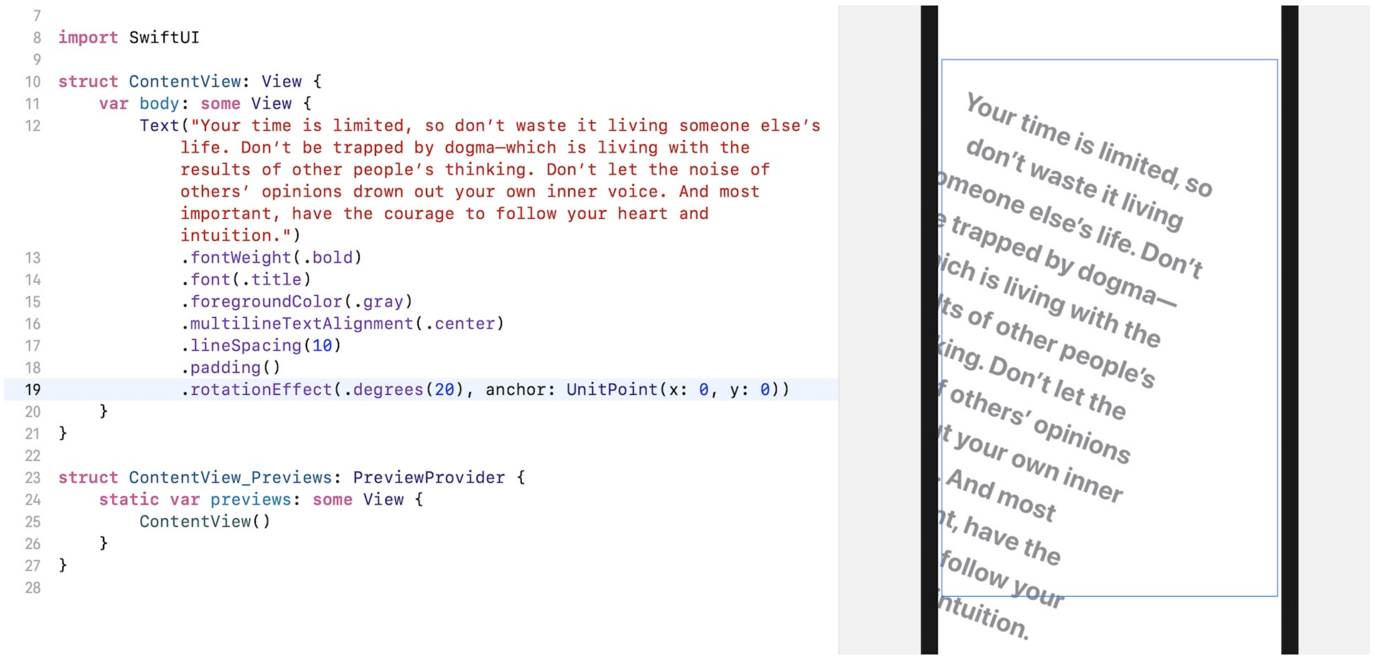
```swift

import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Your time is limited, so don't waste it living someone else's
            life. Don't be trapped by dogma—which is living with the
            results of other people's thinking. Don't let the noise of
            others' opinions drown out your own inner voice. And most
            important, have the courage to follow your heart and
            intuition.")
            .fontWeight(.bold)
            .font(.title)
            .foregroundColor(.gray)
            .multilineTextAlignment(.center)
            .lineSpacing(10)
            .padding()
            .rotationEffect(.degrees(45))
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}

```

Figure 13. Rotate the text

By default, the rotation happens around the center of the text view. If you want to rotate the text around a specific point (say, the top-left corner), you write the code like this:

```swift
.rotationEffect(.degrees(20), anchor: UnitPoint(x: 0, y: 0))
```

We pass an extra parameter `anchor` to specify the point of the rotation.



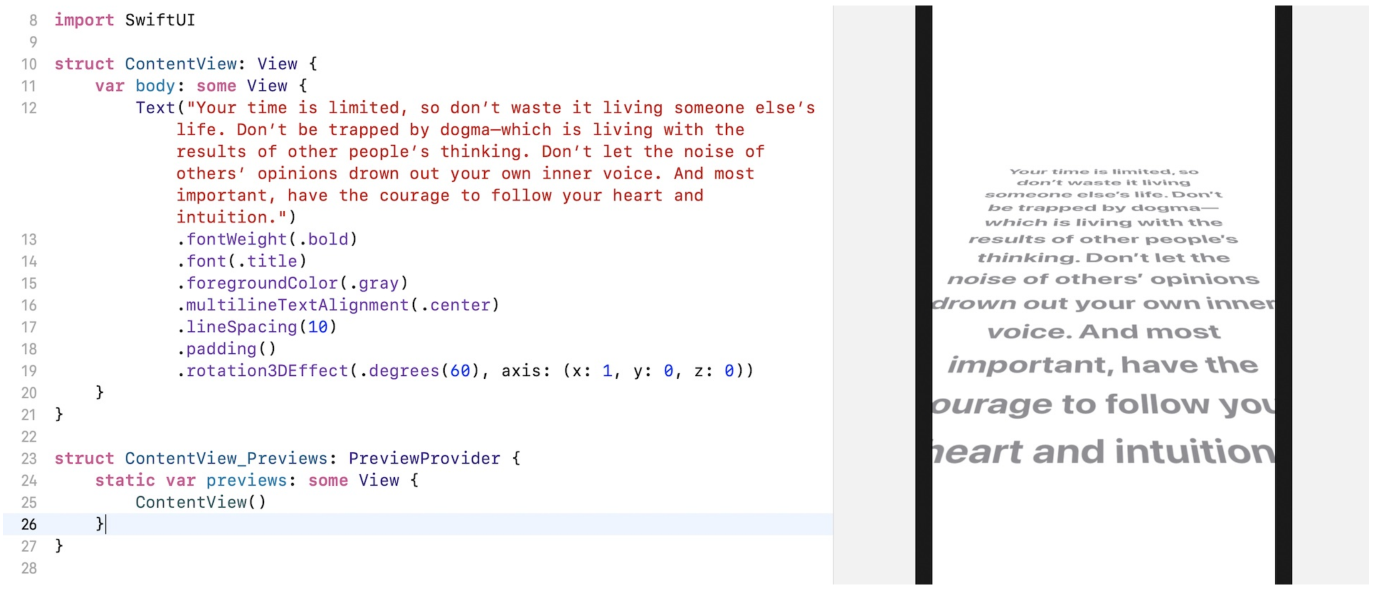
```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Your time is limited, so don't waste it living someone else's
            life. Don't be trapped by dogma—which is living with the
            results of other people's thinking. Don't let the noise of
            others' opinions drown out your own inner voice. And most
            important, have the courage to follow your heart and
            intuition.")
            .fontWeight(.bold)
            .font(.title)
            .foregroundColor(.gray)
            .multilineTextAlignment(.center)
            .lineSpacing(10)
            .padding()
            .rotationEffect(.degrees(20), anchor: UnitPoint(x: 0, y: 0))
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 14. Rotate the text around the top-left of the text view

Not only can you rotate the text in 2D, SwiftUI provides a modifier called `rotation3DEffect` that allows you to create some amazing 3D effects. The modifier takes two parameters: *rotation angle and the axis of the rotation*. Say, you want to create a perspective text effect, you write the code like this:

```
.rotation3DEffect(.degrees(60), axis: (x: 1, y: 0, z: 0))
```

With just a line of code, you have created the Star Wars perspective text!



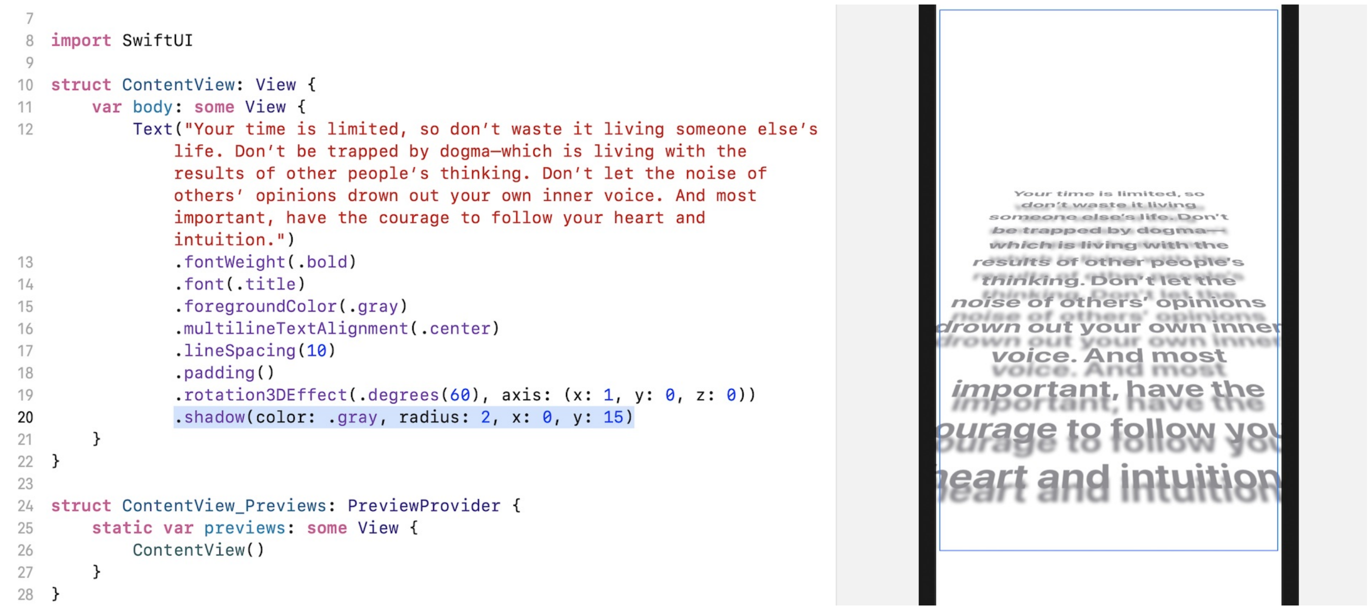
```swift
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Your time is limited, so don’t waste it living someone else’s
            life. Don’t be trapped by dogma—which is living with the
            results of other people’s thinking. Don’t let the noise of
            others’ opinions drown out your own inner voice. And most
            important, have the courage to follow your heart and
            intuition.")
            .fontWeight(.bold)
            .font(.title)
            .foregroundColor(.gray)
            .multilineTextAlignment(.center)
            .lineSpacing(10)
            .padding()
            .rotation3DEffect(.degrees(60), axis: (x: 1, y: 0, z: 0))
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 15. Create amazing text effect by using 3D rotation

You can further insert the following line of code to create a drop shadow effect for the perspective text:

```swift
.shadow(color: .gray, radius: 2, x: 0, y: 15)
```

The `shadow` modifier will apply the shadow effect to the text. All you need to do is specify the color and radius of the shadow. Optionally, you can tell the system the position of the shadow by specifying the `x` and `y` values.



```swift

import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Your time is limited, so don't waste it living someone else's
            life. Don't be trapped by dogma—which is living with the
            results of other people's thinking. Don't let the noise of
            others' opinions drown out your own inner voice. And most
            important, have the courage to follow your heart and
            intuition.")
            .fontWeight(.bold)
            .font(.title)
            .foregroundColor(.gray)
            .multilineTextAlignment(.center)
            .lineSpacing(10)
            .padding()
            .rotation3DEffect(.degrees(60), axis: (x: 1, y: 0, z: 0))
            .shadow(color: .gray, radius: 2, x: 0, y: 15)
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 16. Applying the drop shadow effect

Summary

Do you enjoy creating user interfaces with SwiftUI? I hope so. The declarative syntax of SwiftUI makes the code more readable and easier to understand. As you have experienced, it only takes a few lines of code in SwiftUI to create fancy text in 3D style.

For reference, you can download the complete text project here:

- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUIText.zip)

Chapter 3
Working with Images and Labels

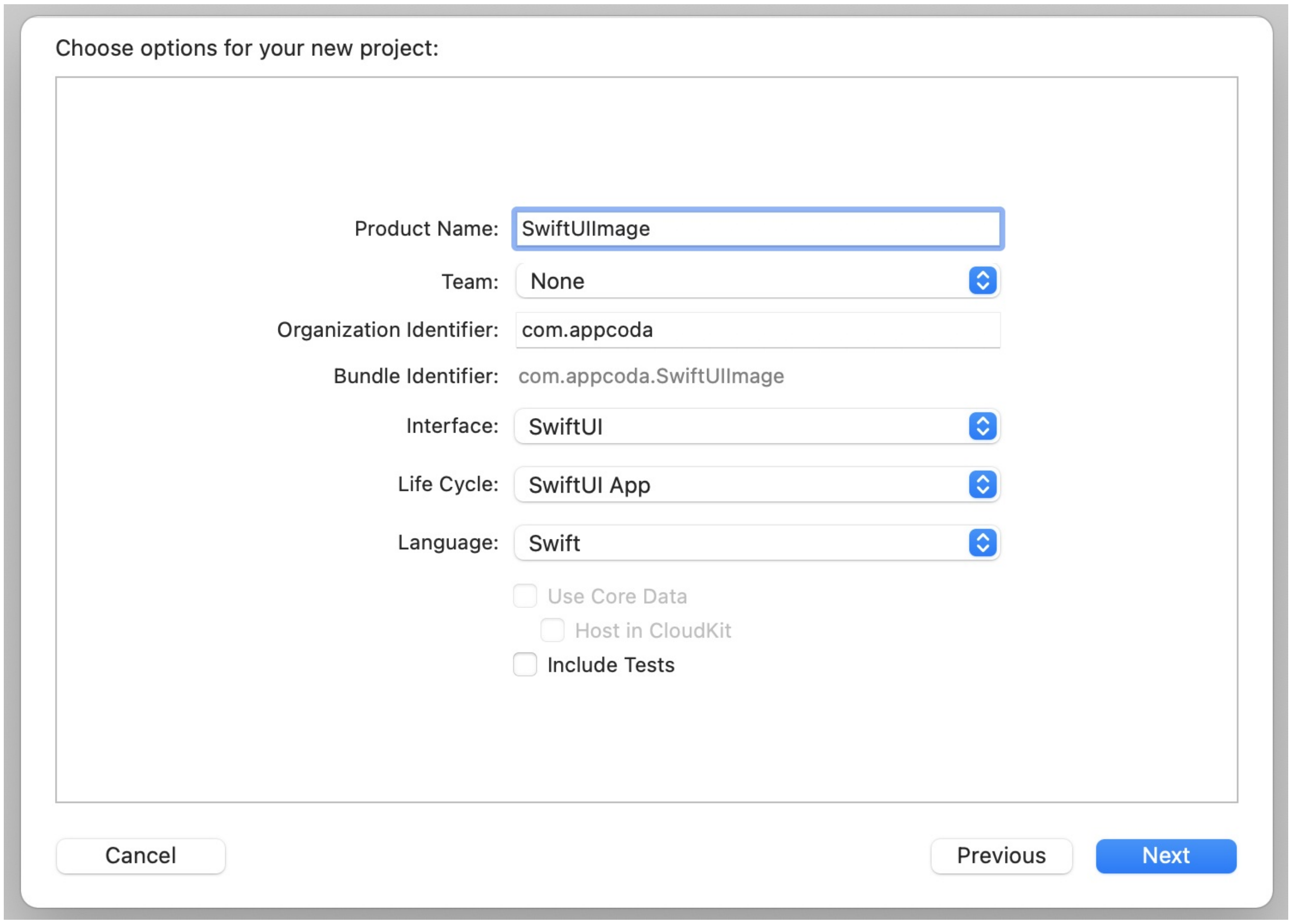
Now that you have a basic introduction to SwiftUI and understand how to display textual content, let's learn how to display images in this chapter. We will also explore the usage of `Label`, a new UI component introduced in iOS 14.

In addition to text, images are another basic element that you'll use in iOS app development. SwiftUI provides a view called `Image` for developers to render and draw images on screen. Similar to what we've done in the previous chapter, I'll show you how to work with `Image` by building a simple demo. In brief, this chapter covers the following topics:

- What's SF Symbols and how to display a system image
- How to display our own images
- How to resize an image
- How to display a full screen image using `edgesIgnoringSafeArea`
- How to create a circular image
- How to apply an overlay to an image

Creating a New Project for Playing with Images

First, fire up Xcode and create a new project using the *App* template (under iOS). Enter *SwiftUIImage* as the name of the project. For the organization name, you can set it to your company or organization. Again, here I use *com.appcoda* but you should set to your own value. To use SwiftUI, please make sure you select *SwiftUI* for the *Interface* option. Click *Next* and choose a folder to create the project.

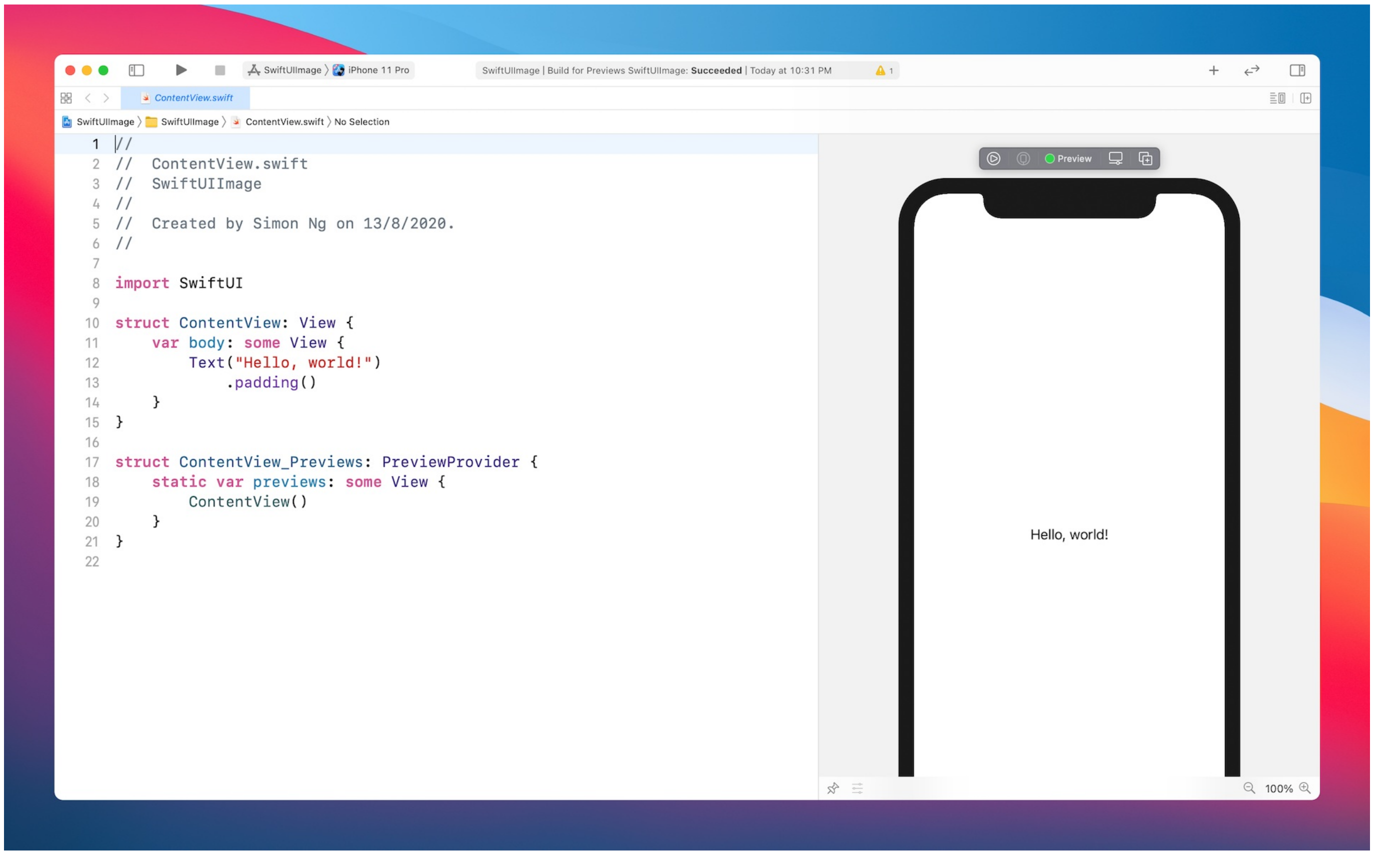

Choose options for your new project:

Product Name:	SwiftUIImage
Team:	None
Organization Identifier:	com.appcoda
Bundle Identifier:	com.appcoda.SwiftUIImage
Interface:	SwiftUI
Life Cycle:	SwiftUI App
Language:	Swift

☐ Use Core Data
☐ Host in CloudKit
☐ Include Tests

Figure 1. Creating a new project

Once you save the project, Xcode should load the `ContentView.swift` file and display a design/preview canvas. If you can't see the preview, make sure you click the *Resume* button.



SwiftUIImage 〉 iPhone 11 Pro
SwiftUIImage | Build for Previews SwiftUIImage: Succeeded | Today at 10:31 PM
ContentView.swift
SwiftUIImage 〉 SwiftUIImage 〉 ContentView.swift 〉 No Selection
Preview
Hello, world!
100%

```
//
//  ContentView.swift
//  SwiftUIImage
//
//  Created by Simon Ng on 13/8/2020.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, world!")
            .padding()
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 2. Previewing the generated code

Understanding SF Symbols

> With over 2,400 configurable symbols, SF Symbols is designed to integrate seamlessly with San Francisco, the system font for Apple platforms. Each symbol comes in a wide range of weights and scales that automatically align with text labels, and supports Dynamic Type and the Bold Text accessibility feature. You can also export symbols and edit them in vector graphics editing tools to create custom symbols with shared design characteristics and accessibility features.

Before I show you how to display an image on screen, let's first talk about where the images I use come from. Needless to say, you can provide your own images for use in the app. Starting from iOS 13, Apple introduced a large set of system images called SF

Symbols that allow developers to use them in any app. Along with the release of iOS 14, Apple further improved the image set by releasing SF Symbols 2. It features over 750 new symbols and adds over 150 preconfigured, multicolor symbols.

These images are referred as symbols since it's integrated with the built-in San Francisco font. To use these symbols, no extra installation is required. As long as your app is deployed to a device running iOS 13 (or later), you can access these symbols directly.

To use the symbols, all you need is the name of the symbol. With over 2,400 symbols available for your use, Apple has released an app called SF Symbols (https://developer.apple.com/sf-symbols/), so that you can easily explore the symbols and locate the one that fits your need. I highly recommend you install the app before proceeding to the next section.

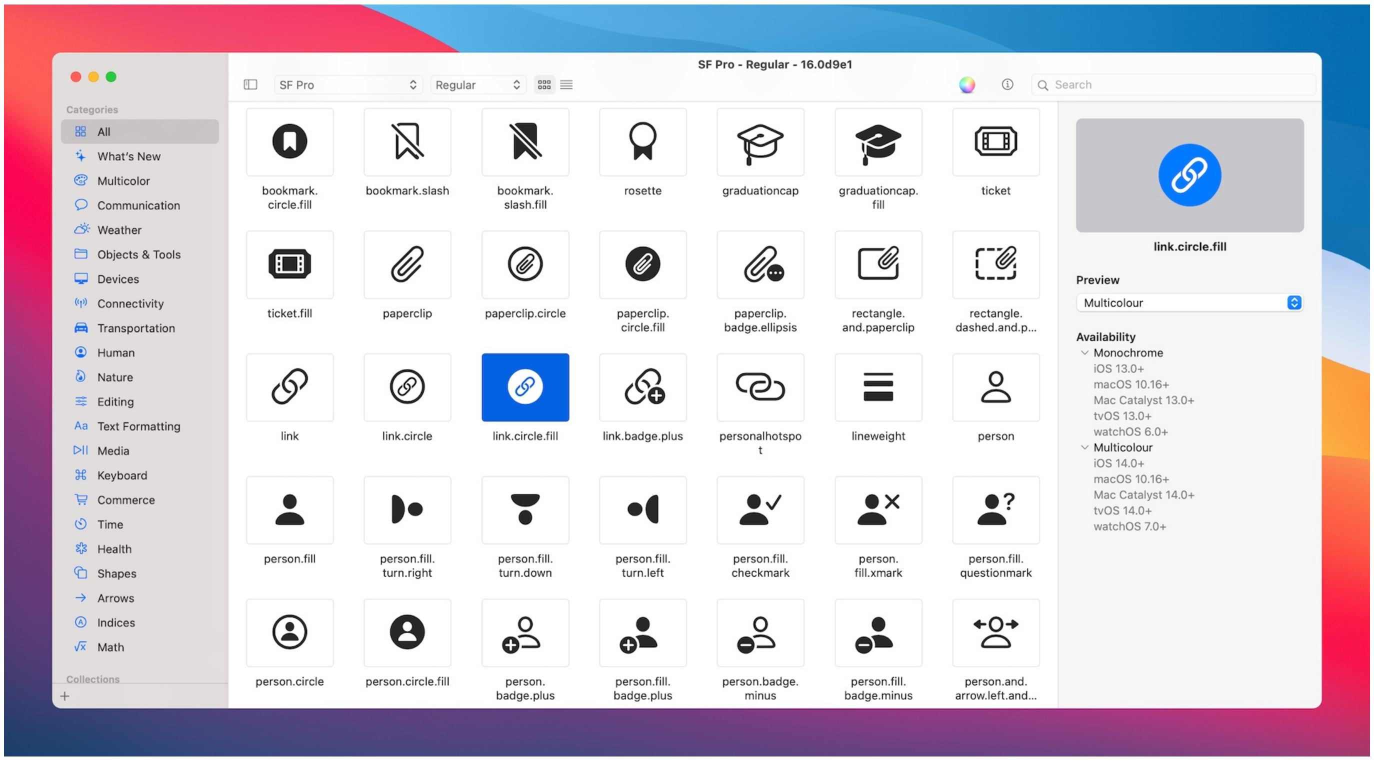


Figure 3. SF Symbols App

Displaying a System Image

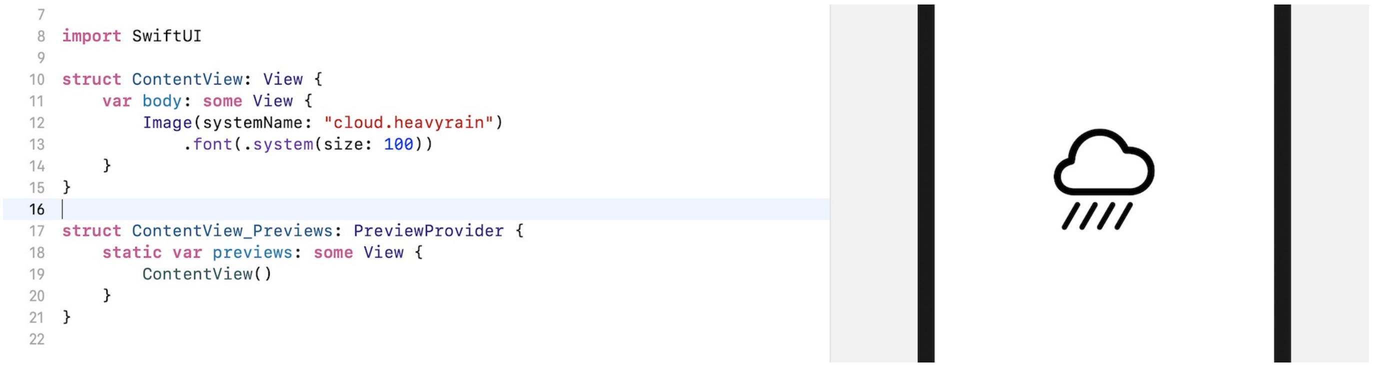
To display a system image (symbol) on screen, you initialize an `Image` view with the `systemName` parameter like this:

```
Image(systemName: "cloud.heavyrain")
```

This will create an image view and load the specified system image. As mentioned before, SF symbols are seamlessly integrated with the San Francisco font. You can easily scale the image by applying the `font` modifier:

```
Image(systemName: "cloud.heavyrain")
    .font(.system(size: 100))
```

Given that the image is part of a font family, you can vary the font size using the `size` parameter, as we did in the previous chapter.



```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Image(systemName: "cloud.heavyrain")
            .font(.system(size: 100))
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 4. Display a system image

Again, since this system image is actually a font, you can apply other modifiers such as `foregroundColor` that you learned in the previous chapter, to change its appearance.

For example, to change the symbol's color to blue, you write the code like this:

```
Image(systemName: "cloud.heavyrain")
    .font(.system(size: 100))
    .foregroundColor(.blue)
```

To add a drop shadow effect, you use the `shadow` modifier:

```
Image(systemName: "cloud.heavyrain")
    .font(.system(size: 100))
    .foregroundColor(.blue)
    .shadow(color: .gray, radius: 10, x: 0, y: 10)
```

Using Your Own Images

We just used the built-in images provided by Apple. You will have your own images to use in your app. Let's see how can you load your images using the `Image` view.

***Note**: You're free to use your own image. In case you don't have an appropriate image to use, you can download this image (https://unsplash.com/photos/Q0-fOL2nqZc) from unsplash.com to follow the rest of the material. After downloading the photo, please make sure you change the filename to "paris.jpg".*

Before you can use an image in your project, the first step is to import the images into the asset catalog (`Assets.xcassets`). Assuming you already prepared the image (`paris.jpg`), press command+0 to reveal the project navigator and then choose `Assets.xcassets` . Open Finder and drag the image to the outline view.

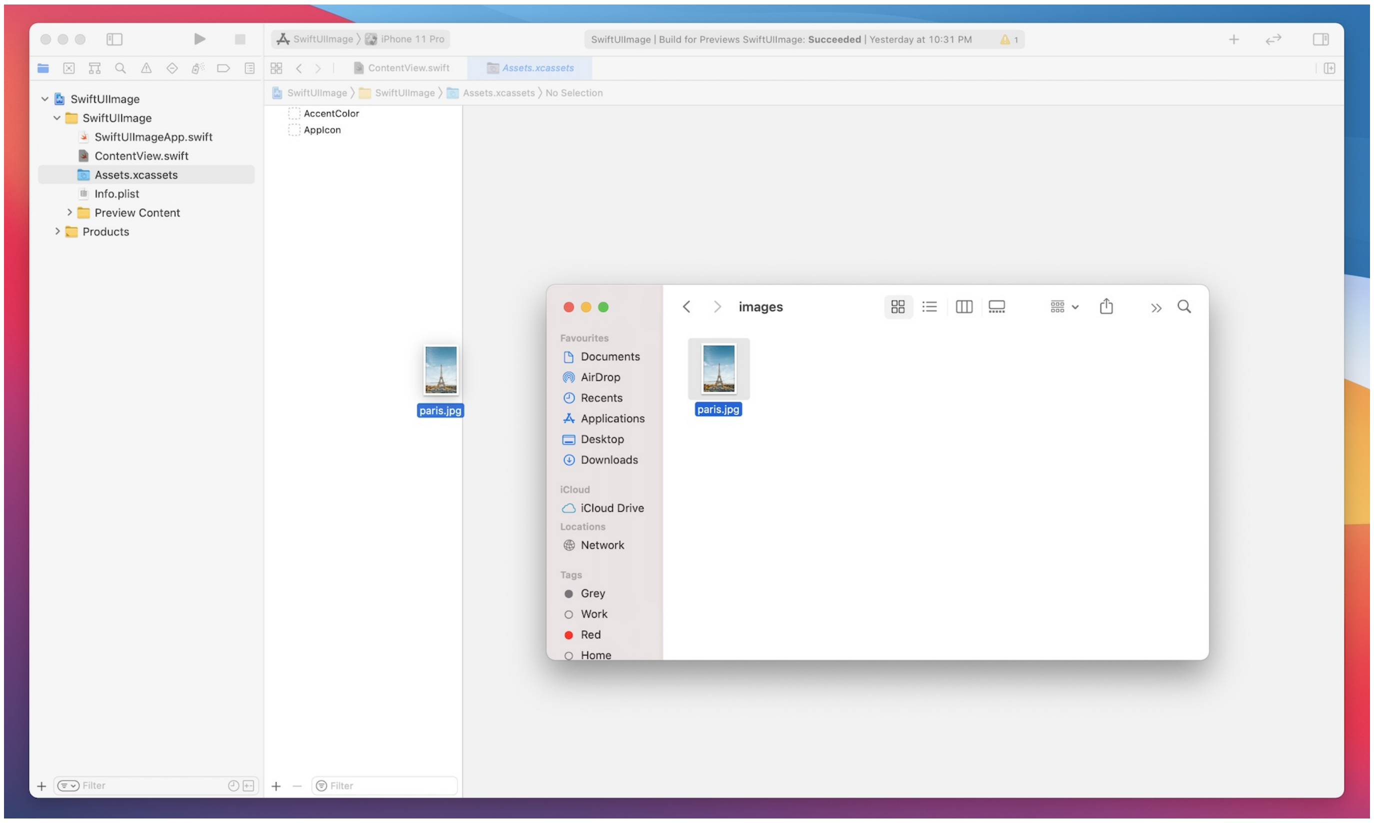


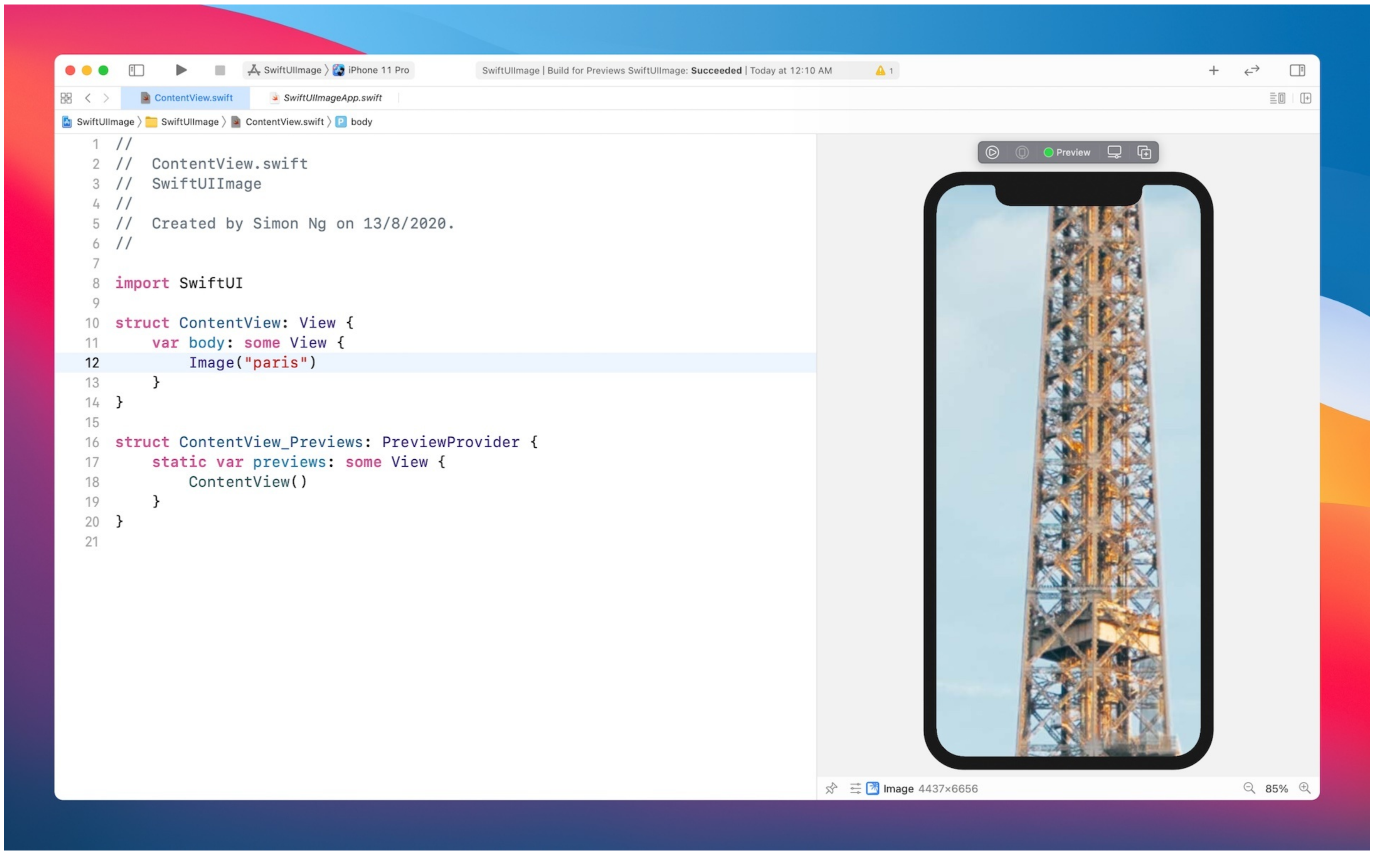
Figure 5. Drag theimage to the asset catalog

If you're new to iOS app development, this asset catalog is where you store application resources like images, color, and data. Once you put the image in the asset catalog, you can load the image by referring to its name. Additionally, you can configure on which device the image can be loaded (e.g. iPhone only).

To display the image on screen, you write the code like this (see figure 6):

```
Image("paris")
```

All you need to do is specify the name of the image and you should see the image in the preview canvas. However, since the image is a high resolution image (4437x6656 pixels), you only see a part of the image.


SwiftUIImage
iPhone 11 Pro
SwiftUIImage | Build for Previews SwiftUIImage: Succeeded | Today at 12:10 AM
ContentView.swift
SwiftUIImageApp.swift
SwiftUIImage 〉 SwiftUIImage 〉 ContentView.swift 〉 body
Preview
Image 4437×6656
85%

```
//
//  ContentView.swift
//  SwiftUIImage
//
//  Created by Simon Ng on 13/8/2020.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        Image("paris")
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

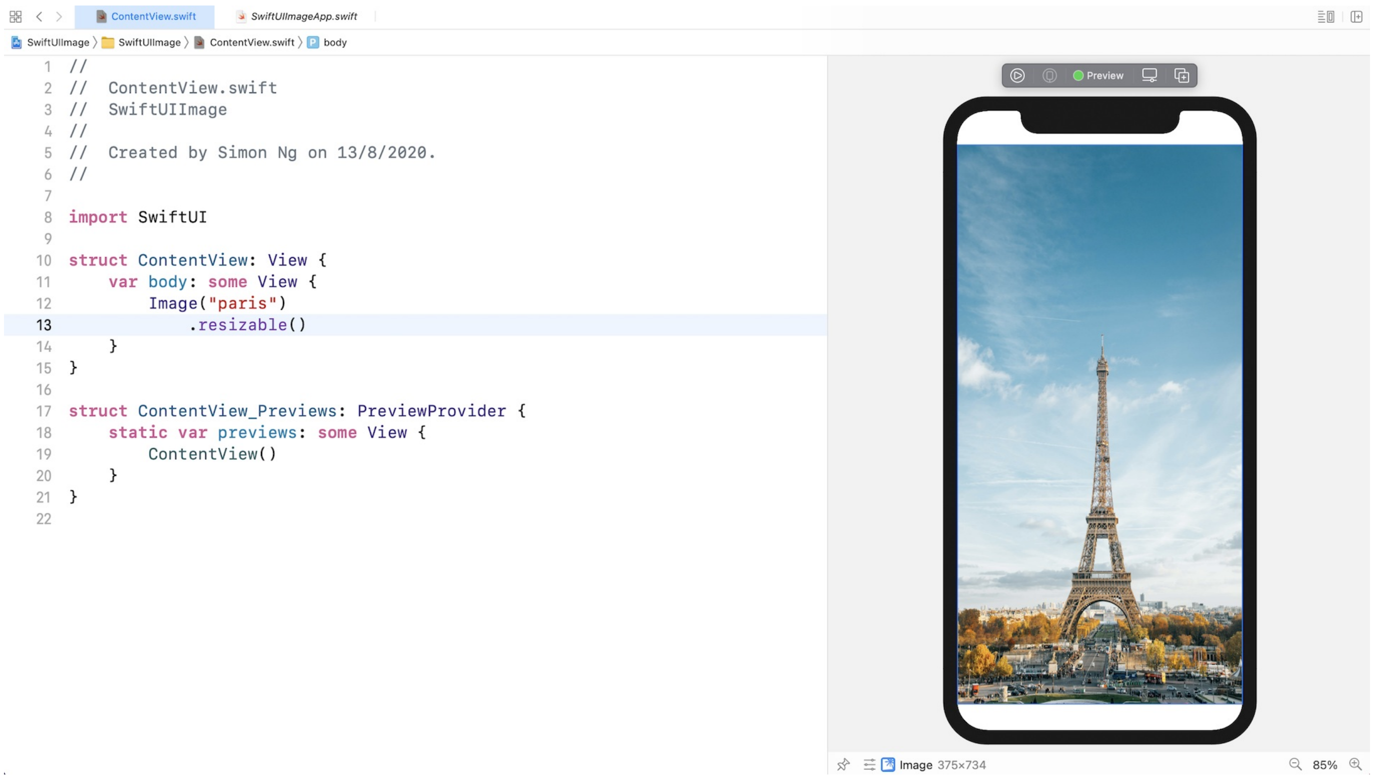
Figure 6. Loading a custom image

Resizing an Image

To resize the image, the `resizable` modifier is used:

```
Image("paris")
    .resizable()
```

By default, the image resizes the image using the *stretch* mode. This means the original image will be scaled to fill the whole screen (except the top and bottom area).

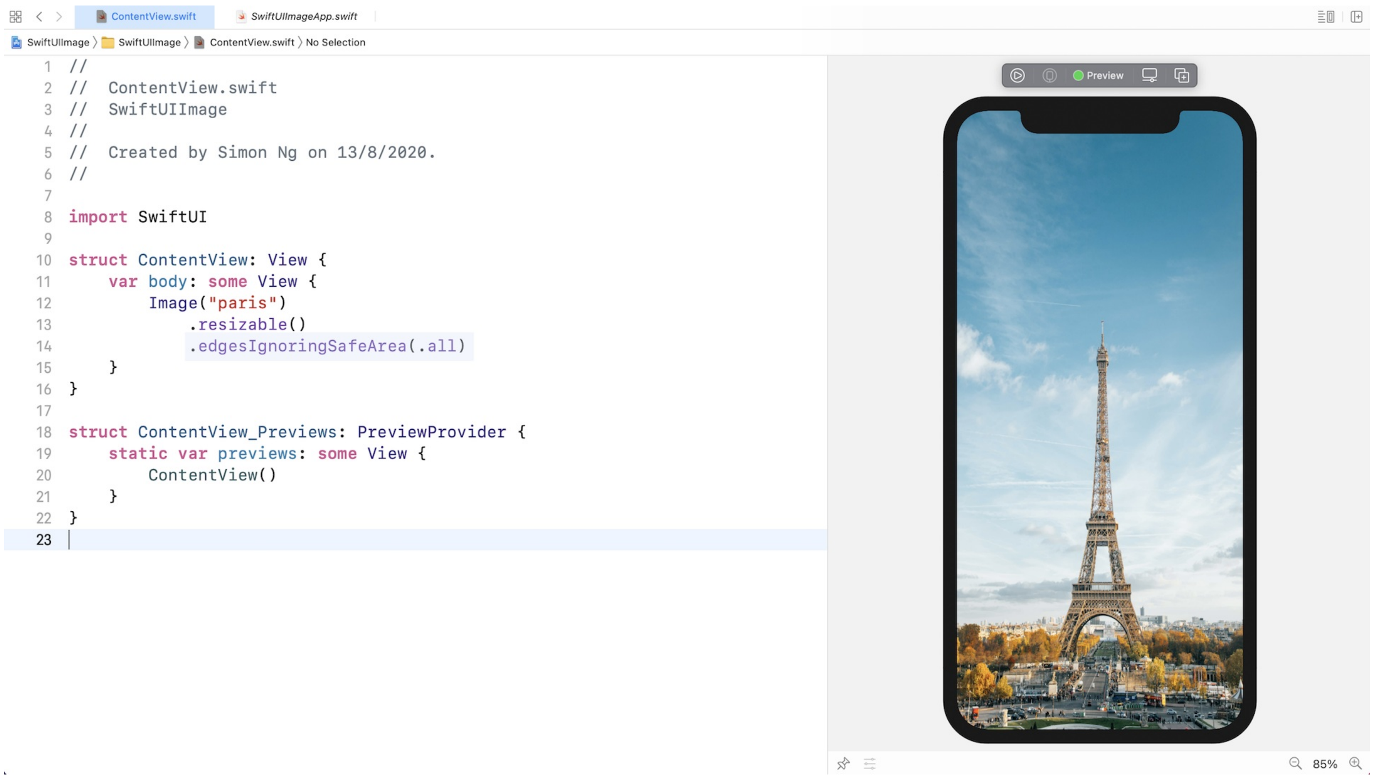


```
//
//  ContentView.swift
//  SwiftUIImage
//
//  Created by Simon Ng on 13/8/2020.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        Image("paris")
            .resizable()
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 7. Resizing the image with the resizable modifier

Technically speaking, the image fills the whole safe area as defined by iOS. The concept of safe area has been around for quite a long time. The safe area is defined as the view area that is safe to lay out our UI component. For example, as you can see from the figure, the safe area is the view area that excludes the top bar (i.e. status bar) and the bottom bar. The safe area will prevent you from accidentally hiding (by over lapping) system UI components like the status bar, navigation bar, and tab bar.

If you want to display a full-screen image, you can ignore the safe area by setting the `edgesIgnoringSafeArea` modifier.



ContentView.swift
SwiftUIImageApp.swift
SwiftUIImage 〉 SwiftUIImage 〉 ContentView.swift 〉 No Selection
Preview
85%

```
//
//  ContentView.swift
//  SwiftUIImage
//
//  Created by Simon Ng on 13/8/2020.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        Image("paris")
            .resizable()
            .edgesIgnoringSafeArea(.all)
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}

```

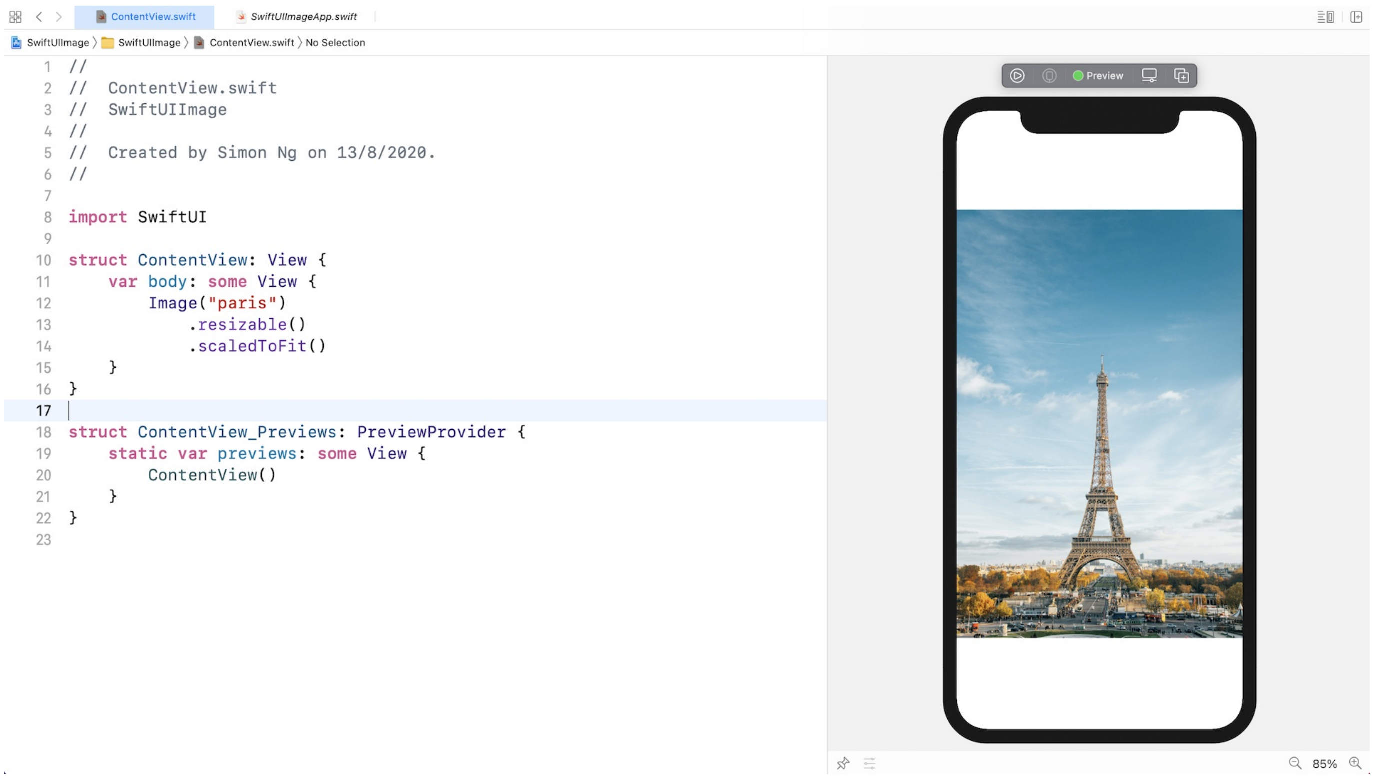
Figure 8. Ignoring the safe area

You can also choose to ignore the safe area for a specific edge. To ignore the safe area for the top edge, you can specify the parameter `.top`. In our code example, we specify `.all`, which means to ignore the safe area for all edges.

Aspect Fit and Aspect Fill

If you look into both images in the previous section and compare it with the original image, you will find that the aspect ratio is a bit distorted. The *stretch* mode doesn't take into account the aspect ratio of the original image. It stretches each side to fit the view area. To keep the original aspect ratio, you can apply the modifier `scaledToFit` like this:

```
Image("paris")
    .resizable()
    .scaledToFit()
```


ContentView.swift
SwiftUIImageApp.swift
SwiftUIImage 〉 SwiftUIImage 〉 ContentView.swift 〉 No Selection
Preview
85%

```
//
//  ContentView.swift
//  SwiftUIImage
//
//  Created by Simon Ng on 13/8/2020.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        Image("paris")
            .resizable()
            .scaledToFit()
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}

```

Figure 9. Scaling the image and keep the original aspect ratio

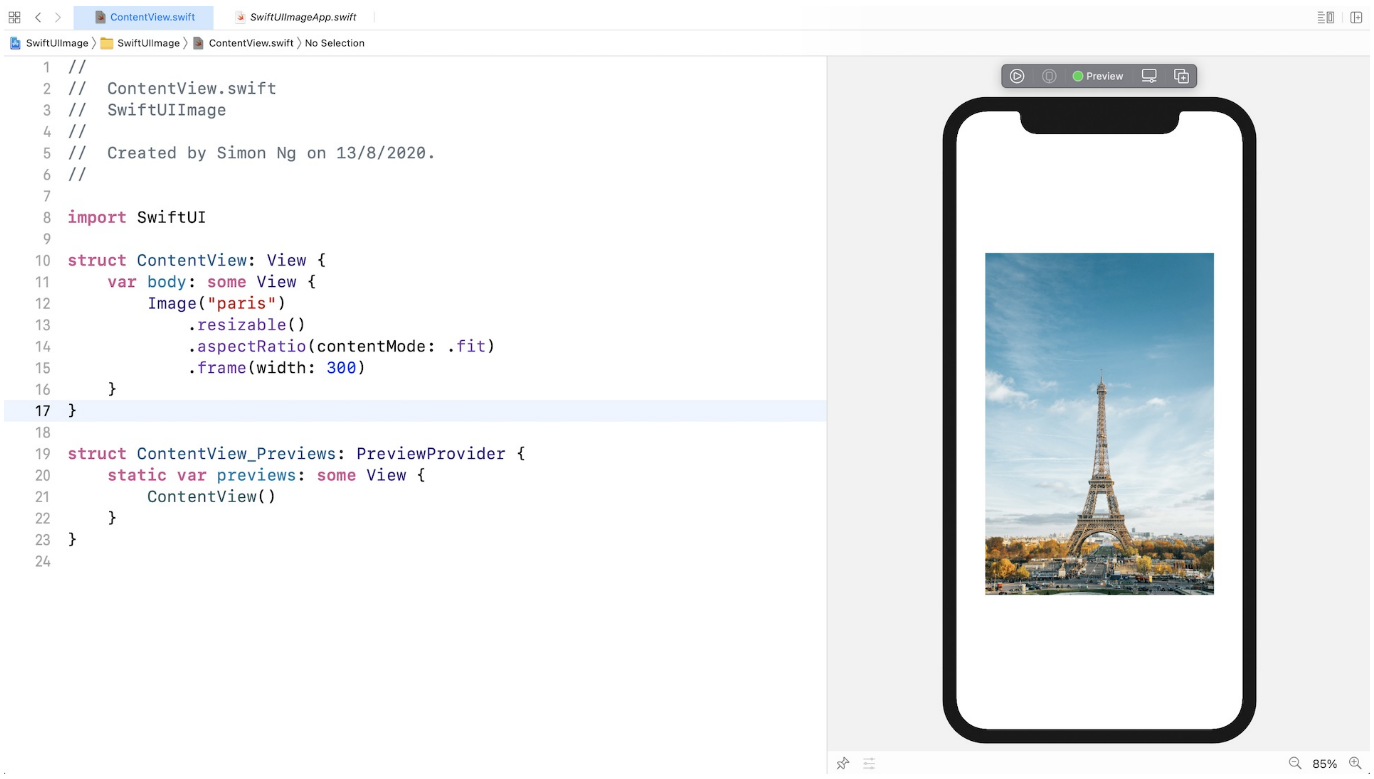
Alternatively, you can use the `aspectRatio` modifier and set the content mode to `.fit`. This will achieve the same result.

```
Image("paris")
    .resizable()
    .aspectRatio(contentMode: .fit)
```

In some cases you may want to keep the aspect ratio of the image but stretch the image to as large as possible, to do this, apply the `.fill` content mode:

```
Image("paris")
    .resizable()
    .aspectRatio(contentMode: .fill)
```

To get a better understanding of the difference between these two modes, Let's limit the size of the image. The `frame` modifier allows you to control the size of a view. By setting the frame's width to 300 points, the image's width will be limited to 300 points.



ContentView.swift
SwiftUIImageApp.swift
SwiftUIImage 〉 SwiftUIImage 〉 ContentView.swift 〉 No Selection
Preview
85%

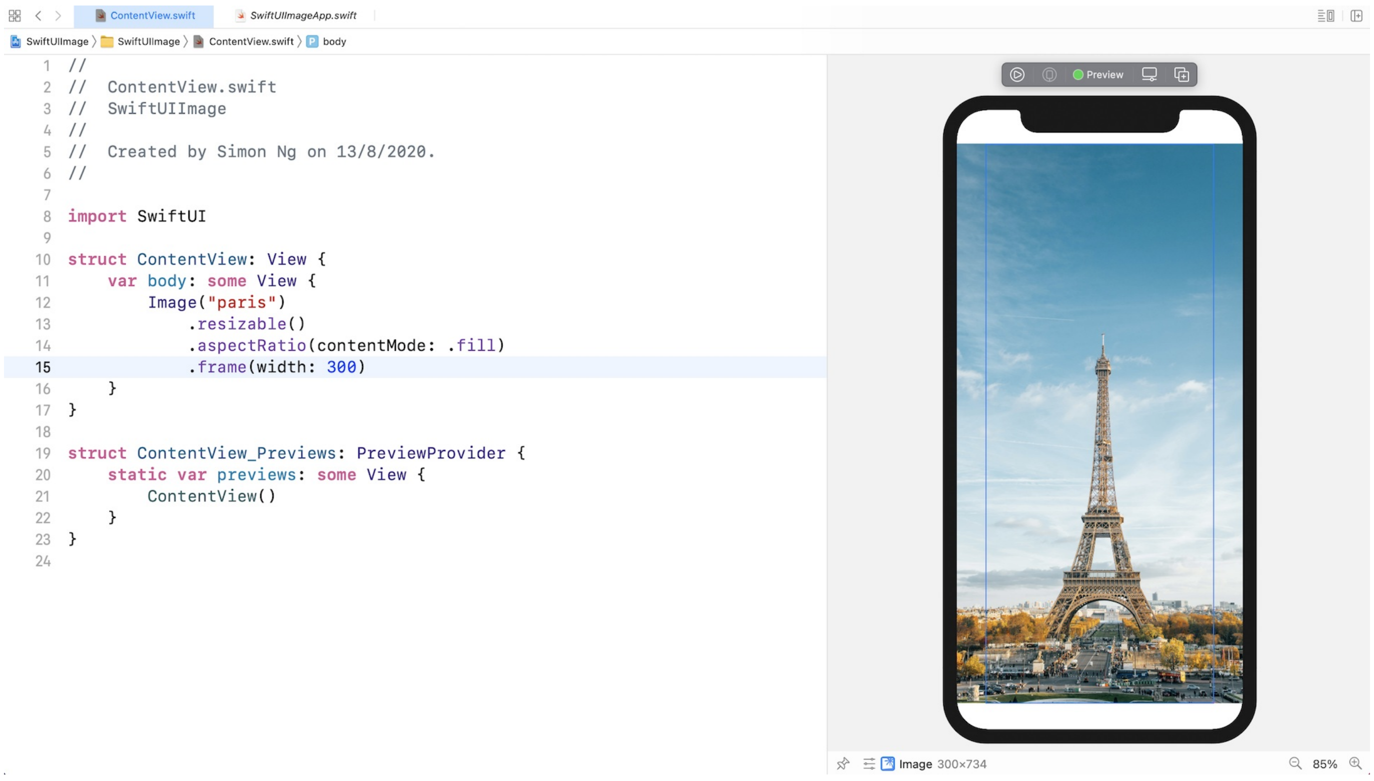
```
//
//  ContentView.swift
//  SwiftUIImage
//
//  Created by Simon Ng on 13/8/2020.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        Image("paris")
            .resizable()
            .aspectRatio(contentMode: .fit)
            .frame(width: 300)
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 10. Scaling down the image and keep the original aspect ratio

Now replace the `Image` code with the following:

```
Image("paris")
    .resizable()
    .aspectRatio(contentMode: .fit)
    .frame(width: 300)
```

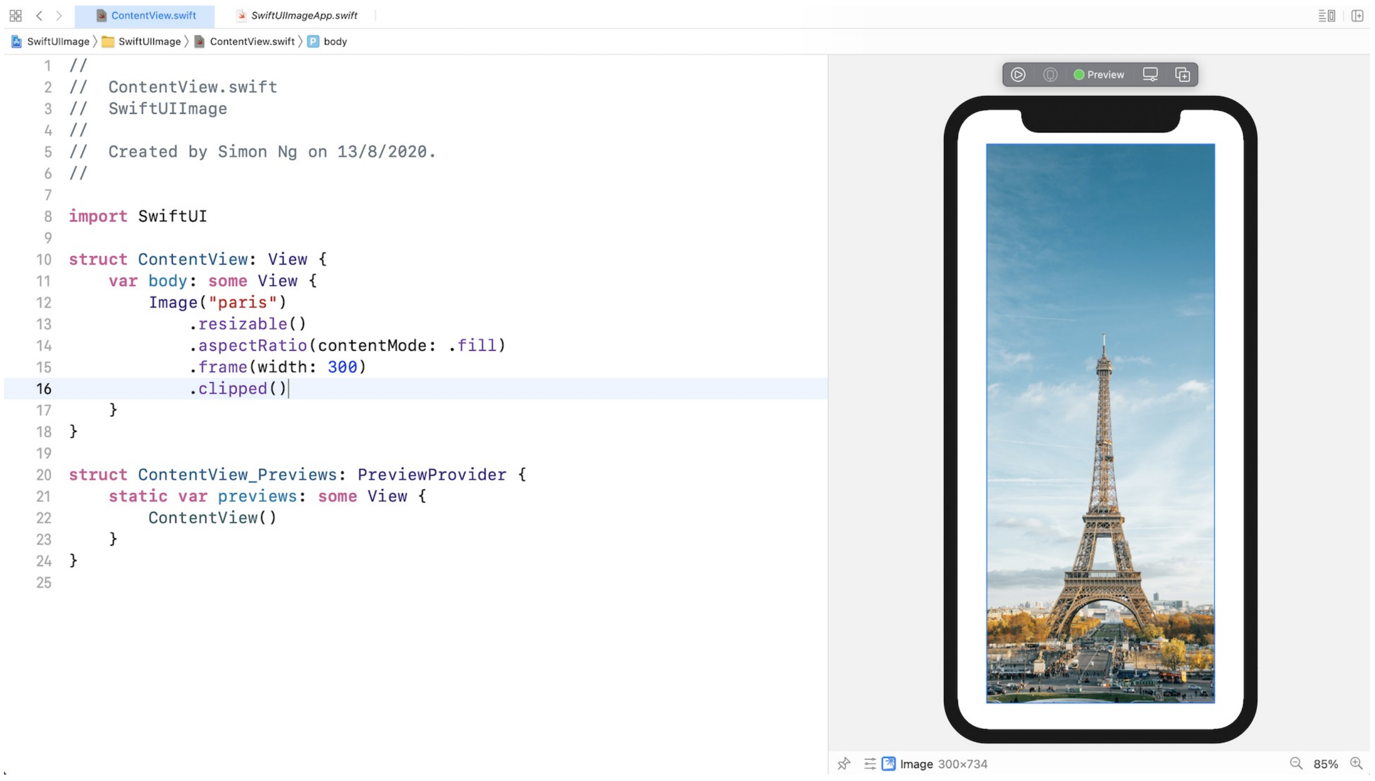
The image will be scaled down in size but the original aspect ratio is kept. If you change the content mode to `.fill`, the image looks pretty much the same as figure 7. However, if you look at the image carefully, the aspect ratio of the orignial image is maintained.



```swift
//
//  ContentView.swift
//  SwiftUIImage
//
//  Created by Simon Ng on 13/8/2020.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        Image("paris")
            .resizable()
            .aspectRatio(contentMode: .fill)
            .frame(width: 300)
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 11. Using .fill content mode

One thing you may notice is that the image's width still takes up the whole screen width. To make it scale correctly, you use the `clipped` modifier to eliminate extra parts of the view (the left and right edges).



```
//
//  ContentView.swift
//  SwiftUIImage
//
//  Created by Simon Ng on 13/8/2020.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        Image("paris")
            .resizable()
            .aspectRatio(contentMode: .fill)
            .frame(width: 300)
            .clipped()
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 12. Use .clipped to clip the view

Creating a Circular Image

In addition to clipping the image in rectangle shape, SwiftUI provides other modifiers for you to clip the image into various shapes (circle, elliose, and capsule). For example, if you want to create a circular image, you use the `clipShape` modifier like this:

```
Image("paris")
    .resizable()
    .aspectRatio(contentMode: .fill)
    .frame(width: 300)
    .clipShape(Circle())
```

Here we specify to clip the image into a circular shape. You can pass different parameters to create an image with a different shape. Figure 13 shows you some examples.

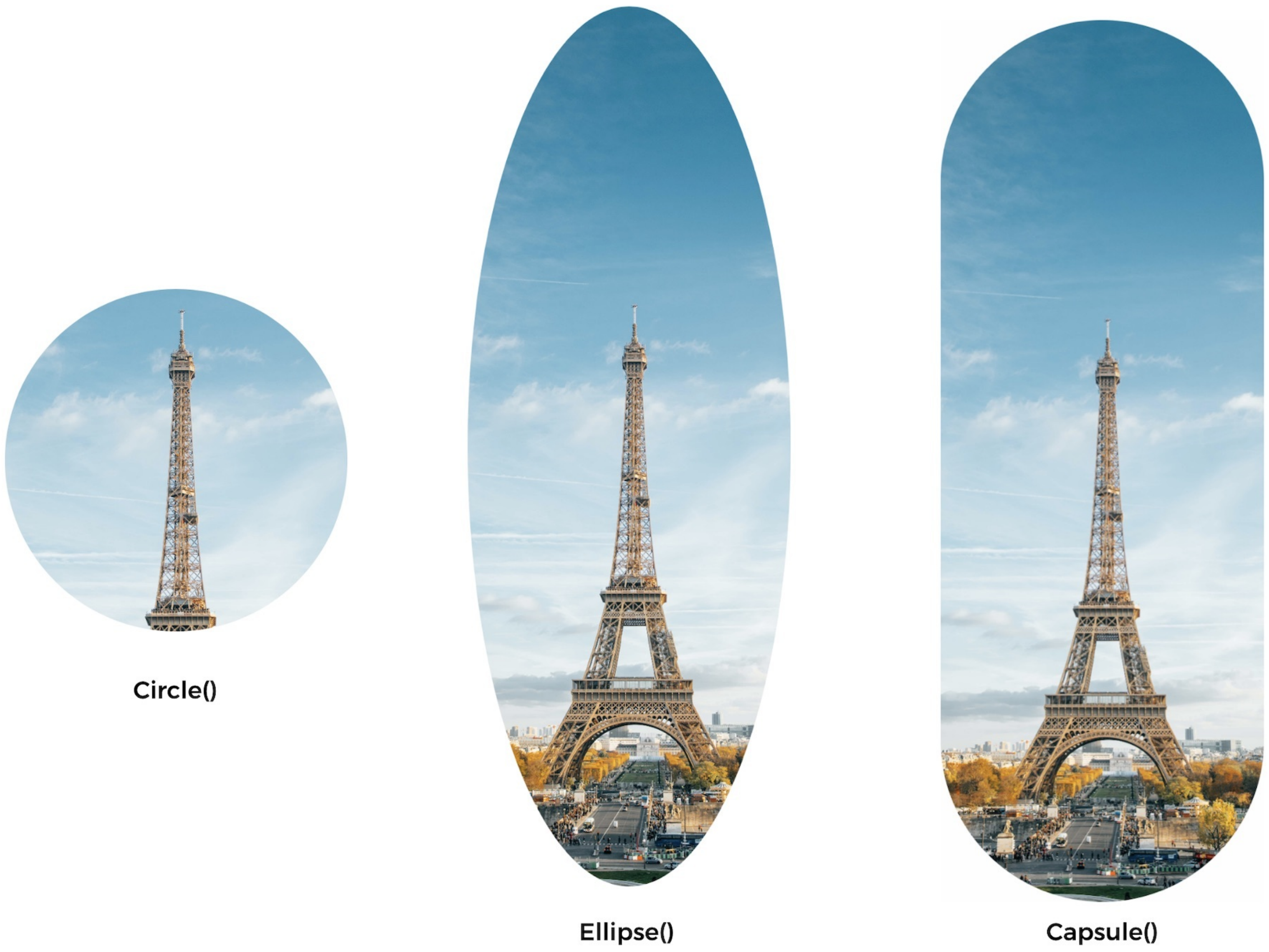


Figure 13. Use the .clipShape modifier to create image with different shape

Adjusting the Opacity

SwiftUI comes with a modifier named `opacity` that you can use to control the opacity of an image (or any view). You pass a value between 0 and 1 to indicate the opacity of the image. Zero means that the view is completely invisible. A value of 1 indicates the image is fully opaque.

For example, if you apply the `opacity` modifier to the image view and set its value to 0.5, the image will become partially transparent.

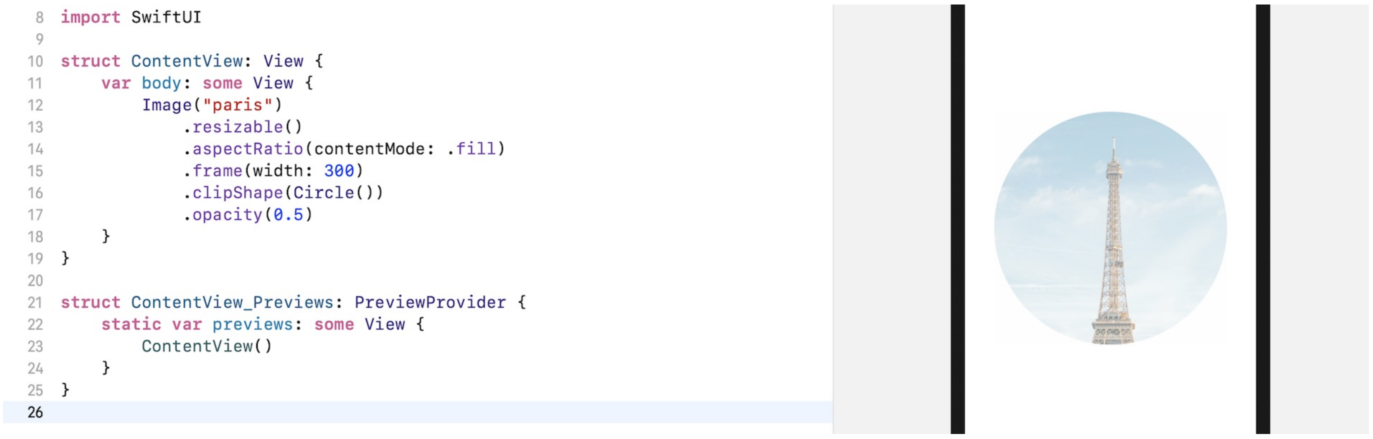

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Image("paris")
            .resizable()
            .aspectRatio(contentMode: .fill)
            .frame(width: 300)
            .clipShape(Circle())
            .opacity(0.5)
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

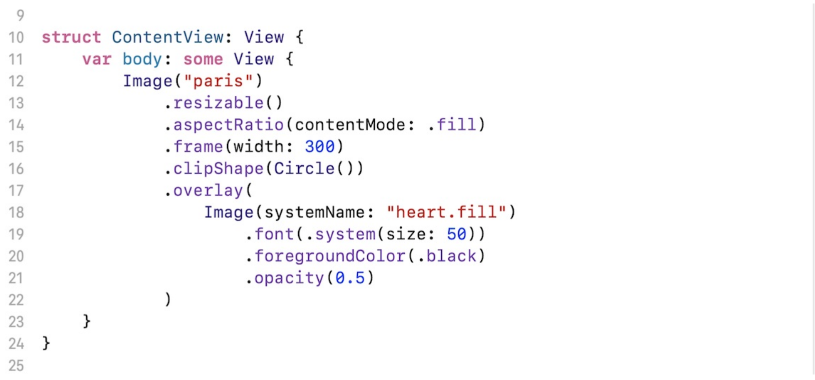
Figure 14. Adjusting the opacity to 50%

Applying an Overlay to an Image

When designing your app, you may need to layer another image or text on top of an image view. The SwiftUI framework provides a modifier named `overlay` for developers to apply an overlay to an image. Let's say, you want to overlay a system image (i.e. heart.fill) on top of the existing image. You write the code like this:

```
Image("paris")
    .resizable()
    .aspectRatio(contentMode: .fill)
    .frame(width: 300)
    .clipShape(Circle())
    .overlay(
        Image(systemName: "heart.fill")
            .font(.system(size: 50))
            .foregroundColor(.black)
            .opacity(0.5)
    )
```

The `.overlay` modifier takes in a `View` as parameter. In the code above, we create another image (i.e. heart.fill) and lay it over the existing image (i.e. Paris).



```
struct ContentView: View {
    var body: some View {
        Image("paris")
            .resizable()
            .aspectRatio(contentMode: .fill)
            .frame(width: 300)
            .clipShape(Circle())
            .overlay(
                Image(systemName: "heart.fill")
                    .font(.system(size: 50))
                    .foregroundColor(.black)
                    .opacity(0.5)
            )
    }
}
```

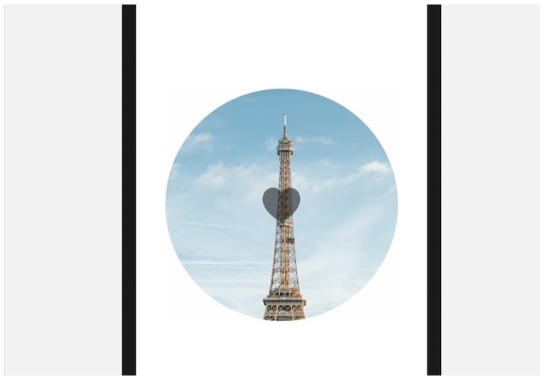
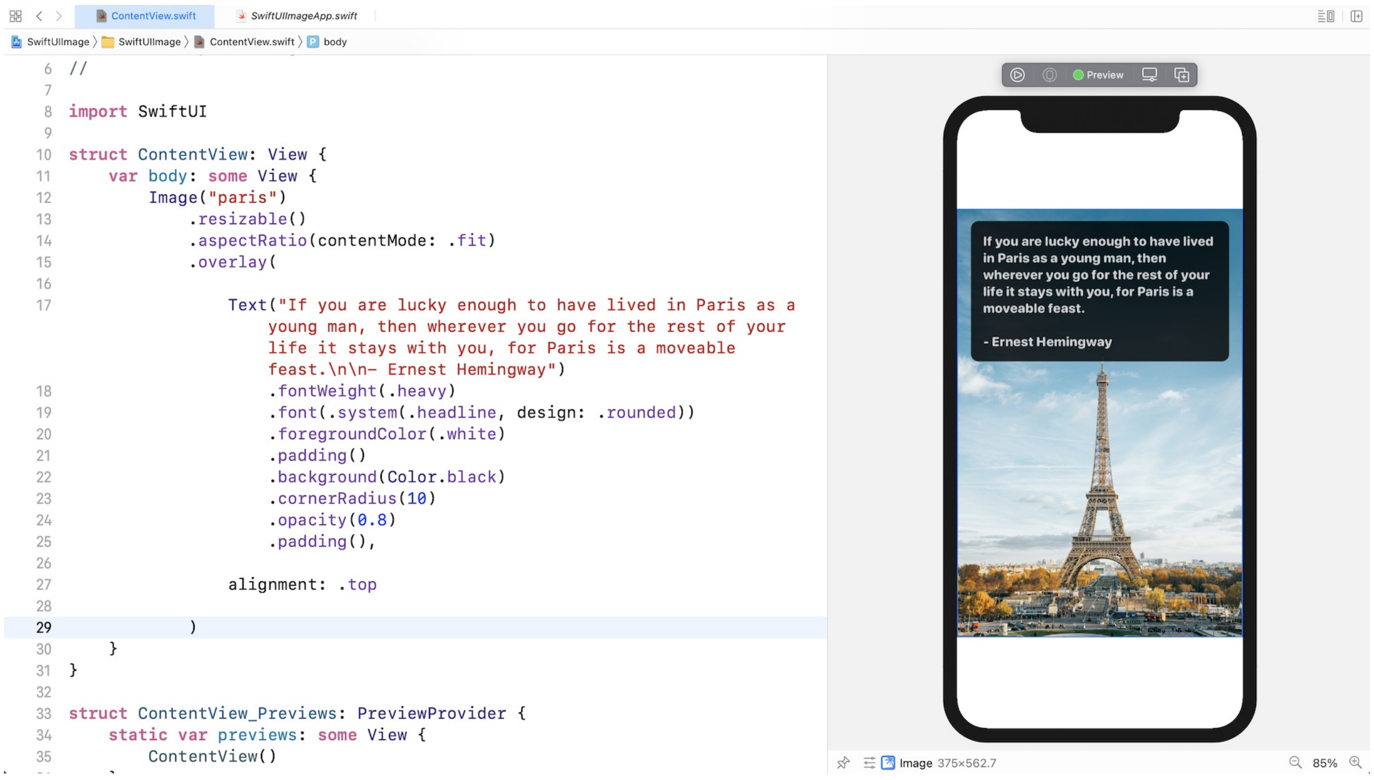


Figure 15. Applying an overlay to the existing image

In fact, you can apply any view as an overlay. For example, you can overlay a `Text` view on the image, like this:

```
Image("paris")
    .resizable()
    .aspectRatio(contentMode: .fit)
    .overlay(

        Text("If you are lucky enough to have lived in Paris as a young man, then 
wherever you go for the rest of your life it stays with you, for Paris is a moveab
le feast.\n\n- Ernest Hemingway")
            .fontWeight(.heavy)
            .font(.system(.headline, design: .rounded))
            .foregroundColor(.white)
            .padding()
            .background(Color.black)
            .cornerRadius(10)
            .opacity(0.8)
            .padding(),

        alignment: .top

    )
```

In the `overlay` modifier, you create a `Text` view and this text view will be applied as an overlay to the image. You should be familiar with the modifiers of the `Text` view as we have discussed in the previous chapter. To change the text, we simply change the font and its color. Aadditionly we can add some padding and apply a background color. One thing I'd like to highlight is the `alignment` parameter. For the `overlay` modifier, you can provide an optional value to adjust the alignment of the view. By default, it's set to center. In this case, we want to position the text overlay to the top part of the image. Change the value from `.center` to `.top` to see how it works.



```swift
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        Image("paris")
            .resizable()
            .aspectRatio(contentMode: .fit)
            .overlay(

                Text("If you are lucky enough to have lived in Paris as a
                    young man, then wherever you go for the rest of your
                    life it stays with you, for Paris is a moveable
                    feast.\n\n- Ernest Hemingway")
                    .fontWeight(.heavy)
                    .font(.system(.headline, design: .rounded))
                    .foregroundColor(.white)
                    .padding()
                    .background(Color.black)
                    .cornerRadius(10)
                    .opacity(0.8)
                    .padding(),

                alignment: .top

            )
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
```

If you are lucky enough to have lived in Paris as a young man, then wherever you go for the rest of your life it stays with you, for Paris is a moveable feast.

- Ernest Hemingway

Figure 16. Applying an overlay to the existing image

Darken an Image Using Overlay

Not only can you overlay an image or text on another image, you can apply an overlay to darken an image. Replace the `Image` code with the following to see the effect:

```
Image("paris")
    .resizable()
    .aspectRatio(contentMode: .fit)
    .overlay(
        Rectangle()
            .foregroundColor(.black)
            .opacity(0.4)
    )
```

We draw a `Rectangle` over the image and set its foreground color to *black*. In order to apply a darkening effect, we set the opacity to 0.4, giving it a 40% opacity. The image should now be darkened.

Alternatively, you may rewrite the code like this to achieve the same effect:

```
Image("paris")
    .resizable()
    .aspectRatio(contentMode: .fit)
    .overlay(
        Color.black
            .opacity(0.4)
    )
```

In SwiftUI, `Color` is also a view. This is why we can use `Color.black` as the top layer to darken the image underneath.

This technique is very useful if you want to over lay some light-colored text on a bright image to make the text more legible. Replace the `Image` code like this:

```
Image("paris")
    .resizable()
    .aspectRatio(contentMode: .fit)
    .frame(width: 300)
    .overlay(
        Color.black
            .opacity(0.4)
            .overlay(
                Text("Paris")
                    .font(.largeTitle)
                    .fontWeight(.black)
                    .foregroundColor(.white)
                    .frame(width: 200)
            )
    )
```

As mentioned before, the `overlay` modifier is not limited to `Image`. You can apply it to any other view. In the code above, we use `Color.black` to darken the image. On top of that, we apply an overlay and place a `Text` over it. If you've made the change correctly, you should see the word "Paris" in bold white, placed over the darkened image.

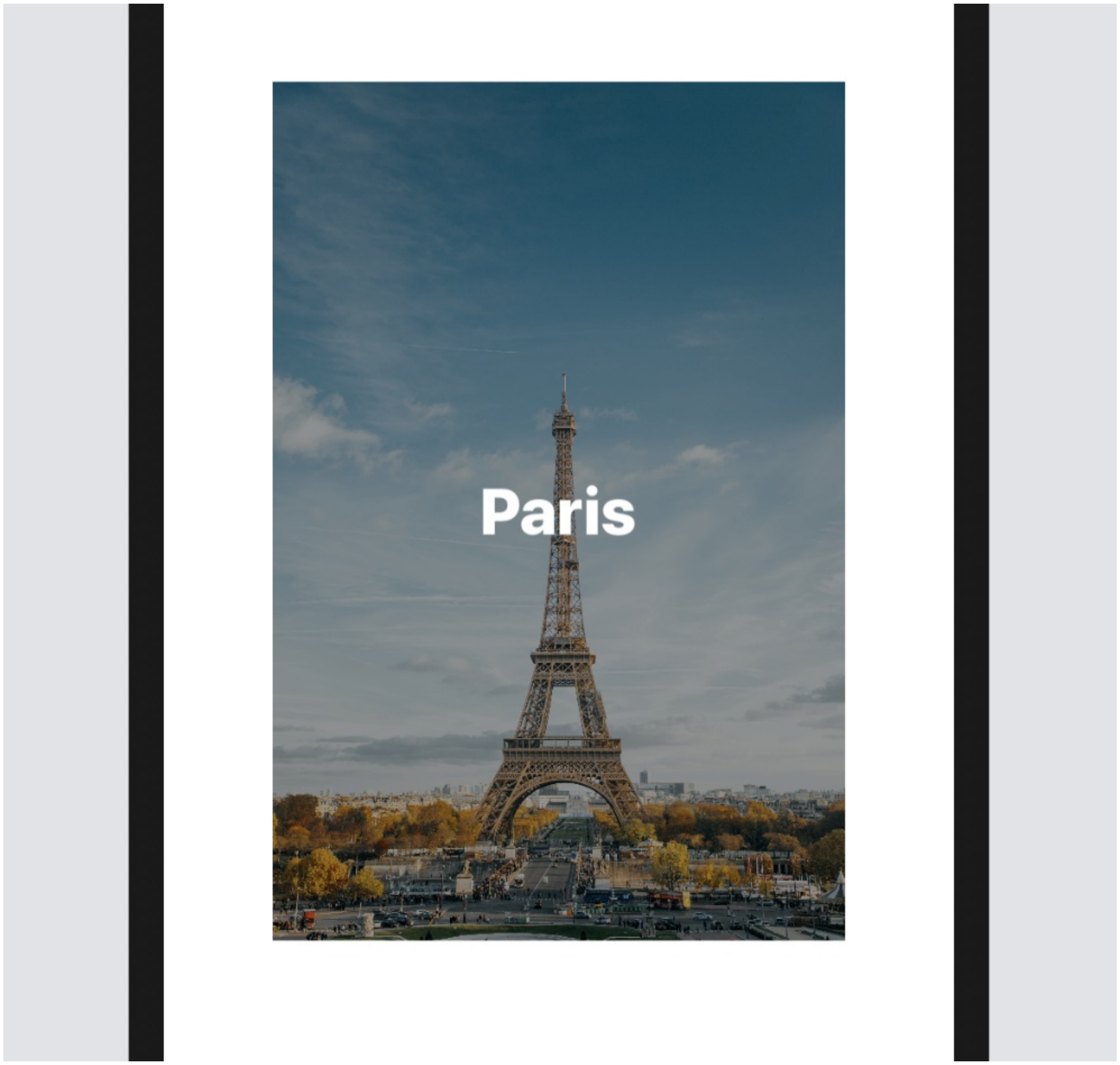


Figure 17. Darken an image and apply a text overlay

Wrap Up

In this chapter, I showed you how to work with images. SwiftUI has made it very easy for developers to display images and use different modifiers to apply various image effects. If you're an indie developer, the newly introduced SF Symbols will save you a lot of time from searching third-party icons!

For reference, you can download the complete images project here:

- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUIImage.zip)

Chapter 4
Layout User Interface with Stacks

Stacks in SwiftUI is similar to the stack views in UIKit. By combining views in horizontal and vertical stacks, you can construct complex user interfaces for your apps. For UIKit, it's inevitable to use auto layout in order to build interfaces that fit all screen sizes. To some beginners, auto layout is a complicated subject and hard to learn. The good news is that you no longer need to use auto layout in SwiftUI. Everything is stacks including VStack, HStack, and ZStack.

In this chapter, I will walk you through all types of stacks and build a grid layout using stacks. So, what project will you work on? Take a look at the figure below. We'll lay out a simple grid interfaces step by step. After going over this chapter, you will be able to combine views with stacks and build the UI you want.

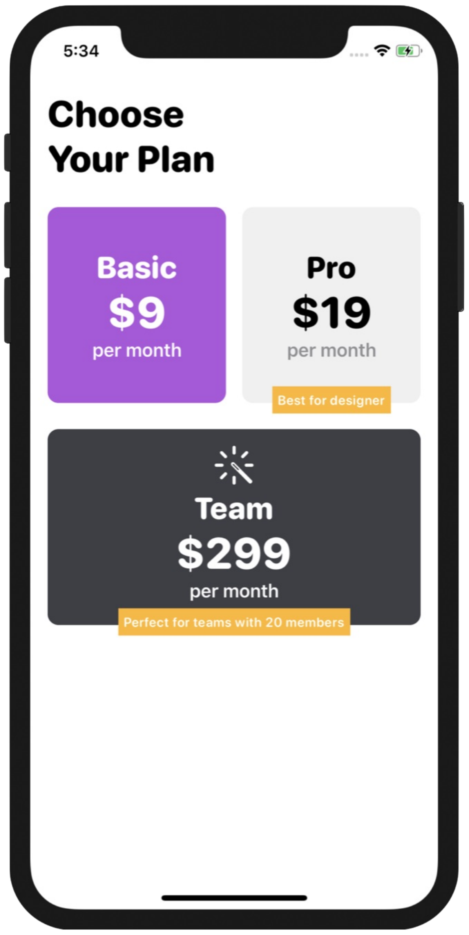


Figure 1. The demo app

Understanding VStack, HStack, and ZStack

SwiftUI provides three different types of stacks for developers to combine views in various orientations. Depending on how you're going to arrange the views, you can either use:

- **HStack** - arranges the views horizontally
- **VStack** - arranges the views vertically
- **ZStack** - overlays one view on top of another

The figure below shows you how these stacks can be used to organize views.

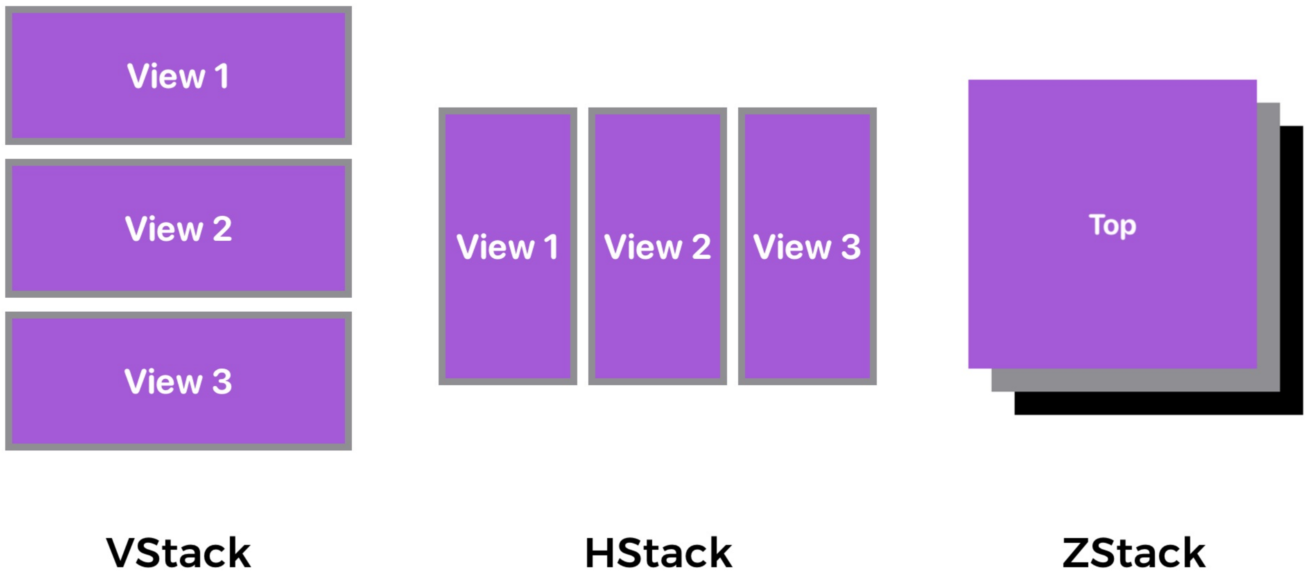
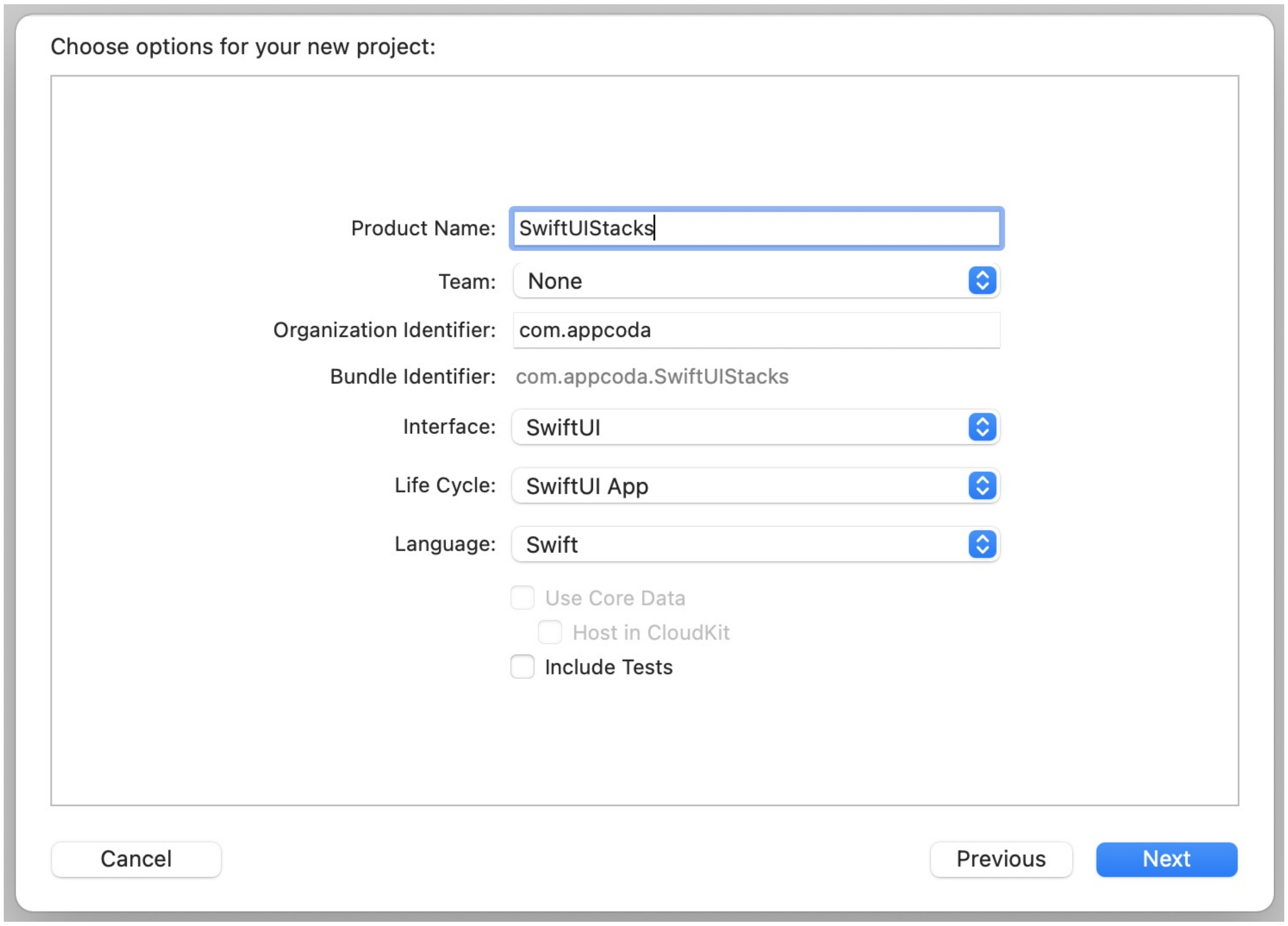


Figure 2. Different types of stack view

Creating a New Project with SwiftUI enabled

First, fire up Xcode and create a new project using the *App* template under the iOS tab. In the next screen, type the name of the project. I set it to *SwiftUIStacks* but you're free to use any other name. Be sure to select the *SwiftUI* option for Interface.


Choose options for your new project:

Product Name: SwiftUIStacks
Team: None
Organization Identifier: com.appcoda
Bundle Identifier: com.appcoda.SwiftUIStacks
Interface: SwiftUI
Life Cycle: SwiftUI App
Language: Swift

Figure 3. Creating a new project

Once you save the project, Xcode will load the `ContentView.swift` file and display a preview in the design canvas. If the preview is not displayed, click the *Resume* button in the canvas.

Using VStack

We're going to build the UI as displayed in figure 1, but first, let's break down the UI into small parts. We'll begin with the heading as shown below.

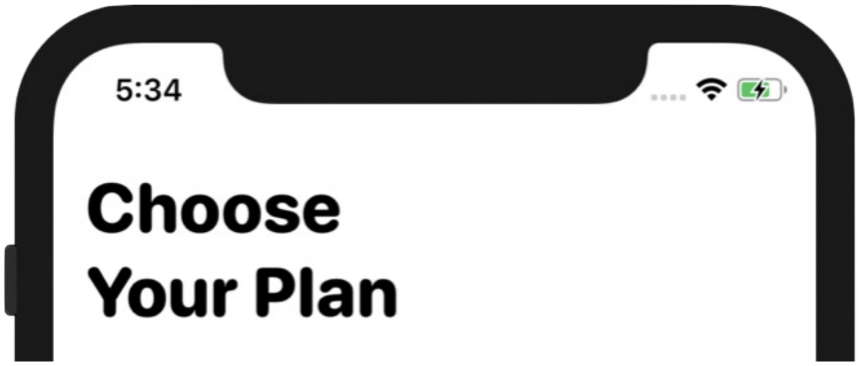


Figure 4. The heading

Presently, Xcode should have already generated the following code to display the "Hello World" label:

```
struct ContentView: View {
    var body: some View {
        Text("Hello World")
    }
}
```

To display the text as shown in figure 4, we will combine two `Text` views within a `VStack` like this:

```
struct ContentView: View {
    var body: some View {
        VStack {
            Text("Choose")
                .font(.system(.largeTitle, design: .rounded))
                .fontWeight(.black)
            Text("Your Plan")
                .font(.system(.largeTitle, design: .rounded))
                .fontWeight(.black)
        }
    }
}
```

When you embed views in a `VStack`, the views will be arranged vertically like this:

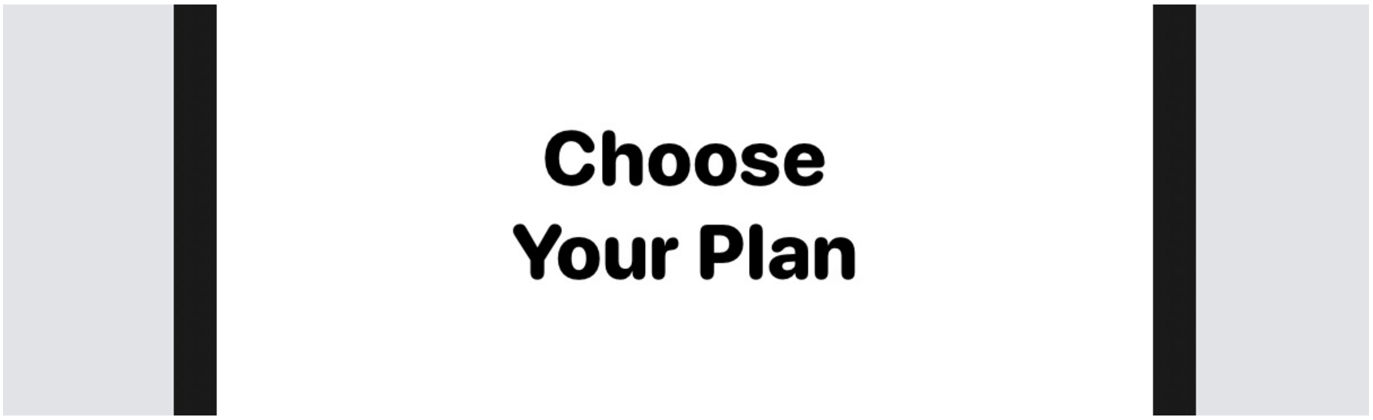
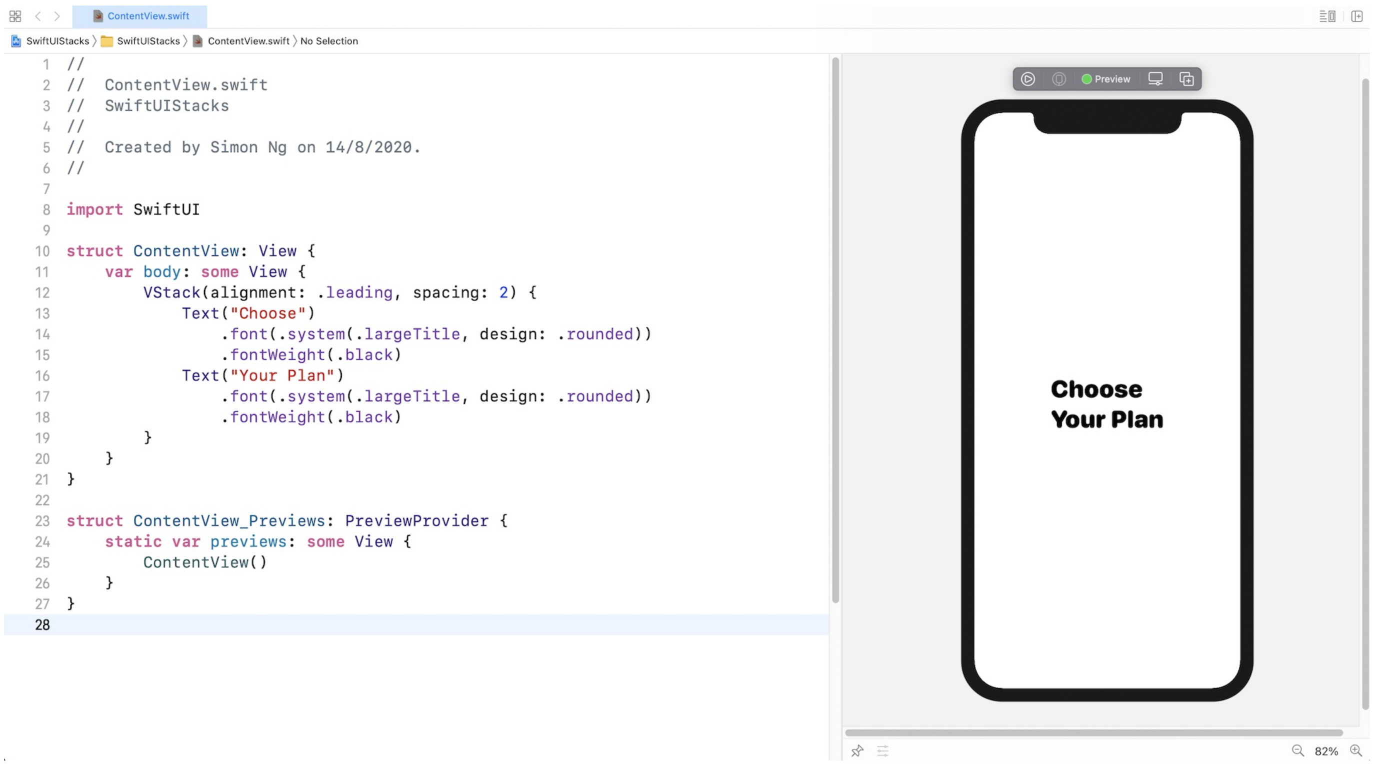


Figure 5. Combining two texts using VStack

By default, the views embedded in the stack are aligned in center position. To align both views to the left, you can specify the `alignment` parameter and set its value to `.leading` like this:

```
VStack(alignment: .leading, spacing: 2) {
    Text("Choose")
        .font(.system(.largeTitle, design: .rounded))
        .fontWeight(.black)
    Text("Your Plan")
        .font(.system(.largeTitle, design: .rounded))
        .fontWeight(.black)
}
```

Additionally, you can adjust the space of the embedded views by using the `space` parameter. We'll add the additional parameter `spacing: 2` to the VStack. The figure below shows the resulting view.



```swift
//
//  ContentView.swift
//  SwiftUIStacks
//
//  Created by Simon Ng on 14/8/2020.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        VStack(alignment: .leading, spacing: 2) {
            Text("Choose")
                .font(.system(.largeTitle, design: .rounded))
                .fontWeight(.black)
            Text("Your Plan")
                .font(.system(.largeTitle, design: .rounded))
                .fontWeight(.black)
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

ContentView.swift
SwiftUIStacks 〉 SwiftUIStacks 〉 ContentView.swift 〉 No Selection
Preview
Choose
Your Plan
82%

Figure 6. Changing the alignment of VStack

Using HStack

Next, let's layout the first two pricing plans. If you look at the *Basic* and *Pro* plans, the look & feel of these two components are very similar. Let's take the *Basic* plan as an example, to achieve the desired layout, you can use `VStack` to combine three text views.

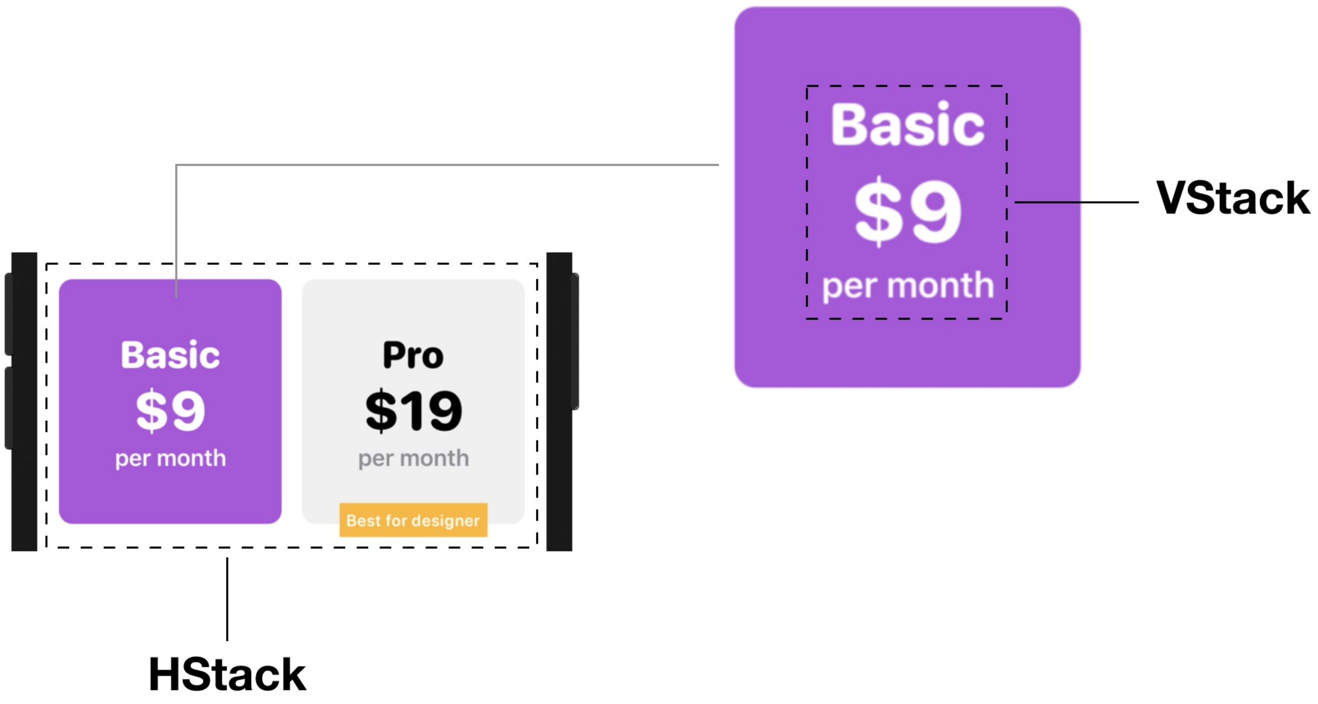


Figure 7. Layout the pricing plans

Both the *Basic* and *Pro* components are arranged side by side. By using `HStack`, you can lay out views horizontally. Stacks can be nested meaning that you can nest stack views within other stack views. Since the pricing plan block sits right below the heading view, which is a `VStack`, we will use another `VStack` to embed a vertical stack (i.e. Choose Your Plan) and a horizontal stack (i.e. the pricing plan block).

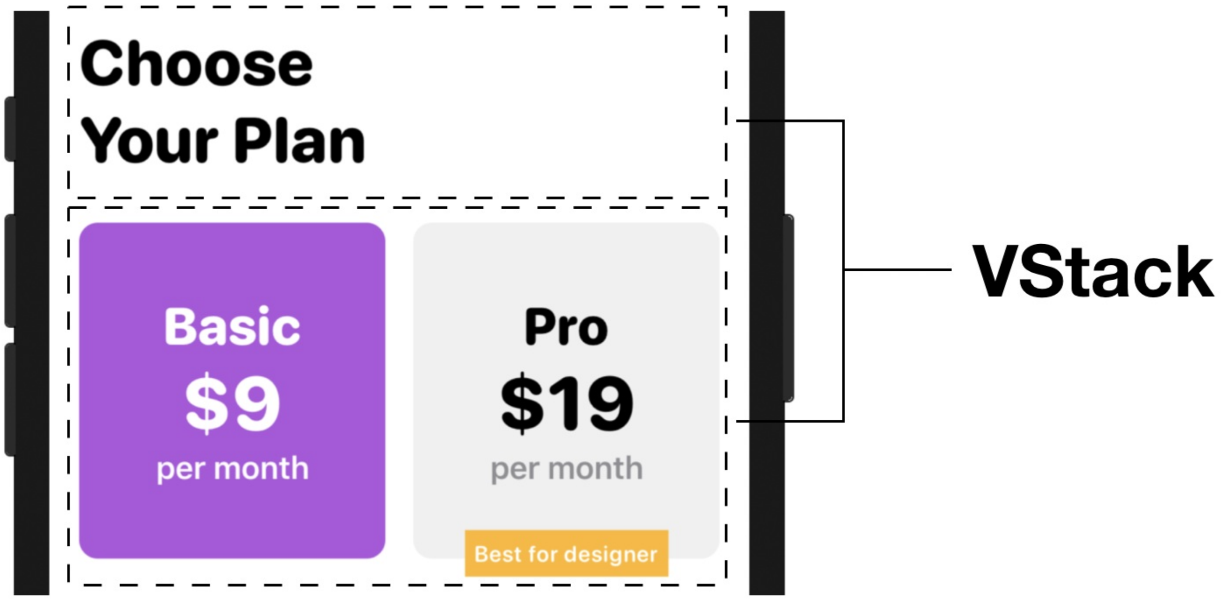
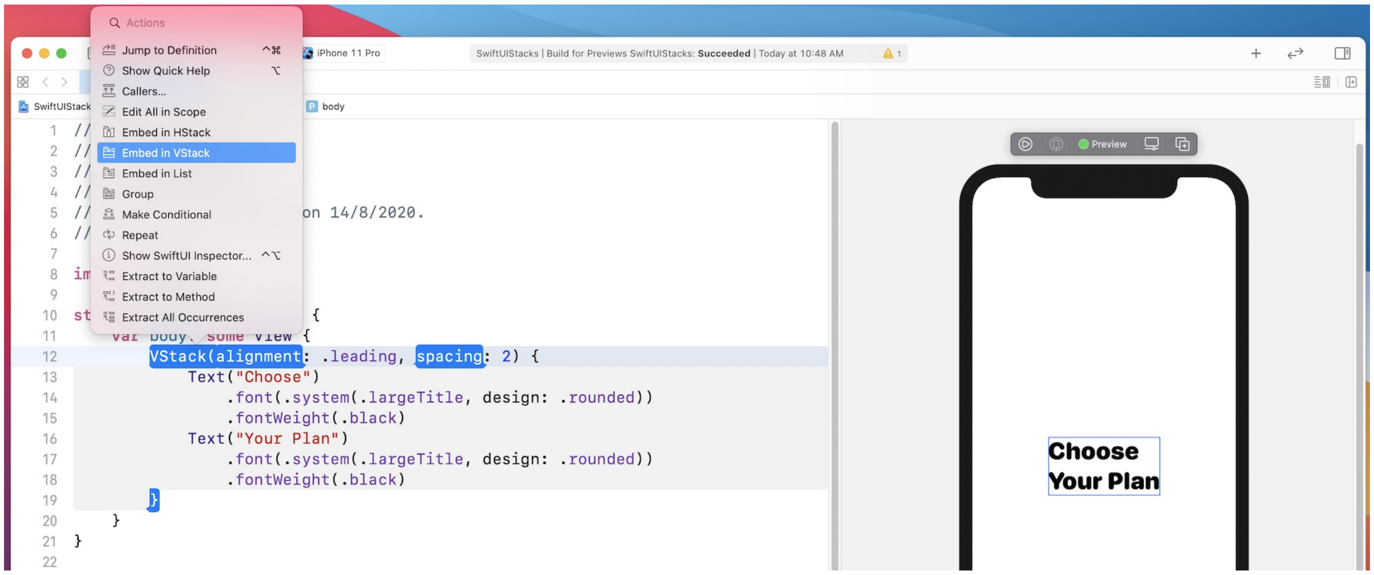


Figure 8. Using a VStack to embed other stack views

Now that you have some basic idea how we're going to use `VStack` and `HStack` for implementing the UI, let's jump right into the code.

To embed the existing `VStack` in another `VStack`, you hold the command key and then click the `VStack` keyword. This will bring up a context menu showing all the available options. Choose *Embed in VStack* to embed the `VStack`.



```
//
//
//
//
//              on 14/8/2020.
//

im

st                      {
    var body: some View {
        VStack(alignment: .leading, spacing: 2) {
            Text("Choose")
                .font(.system(.largeTitle, design: .rounded))
                .fontWeight(.black)
            Text("Your Plan")
                .font(.system(.largeTitle, design: .rounded))
                .fontWeight(.black)
        }
    }
}
```

Figure 9. Embed in VStack

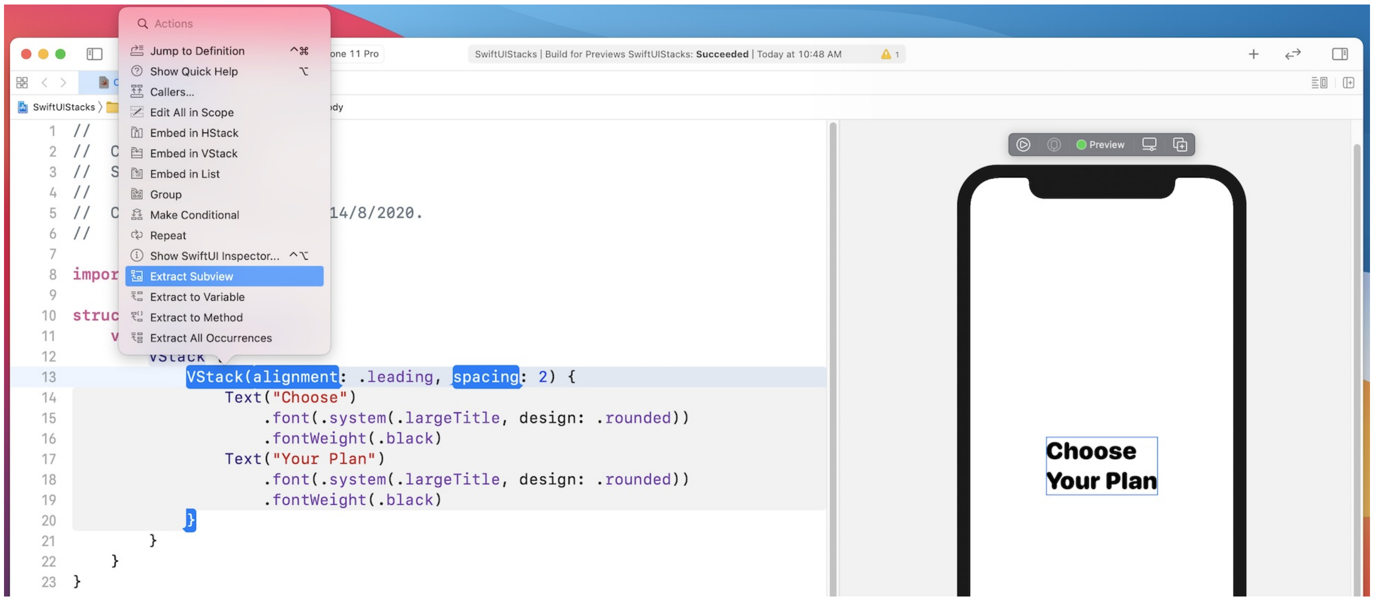
Xcode will then generate the required code to embed the stack. Your code should look like the following:

```
struct ContentView: View {
    var body: some View {
        VStack {
            VStack(alignment: .leading, spacing: 2) {
                Text("Choose")
                    .font(.system(.largeTitle, design: .rounded))
                    .fontWeight(.black)
                Text("Your Plan")
                    .font(.system(.largeTitle, design: .rounded))
                    .fontWeight(.black)
            }
        }
    }
}
```

Extracting a View

Before we continue to lay out the UI, let me show you a trick to better organize the code. As you're going to build a more complex UI that involves several components, the code inside `ContentView` will eventually become a giant code block that is hard to review and debug. It's always a good practice to break large blocks of code into smaller blocks so the code is easier to read and maintain.

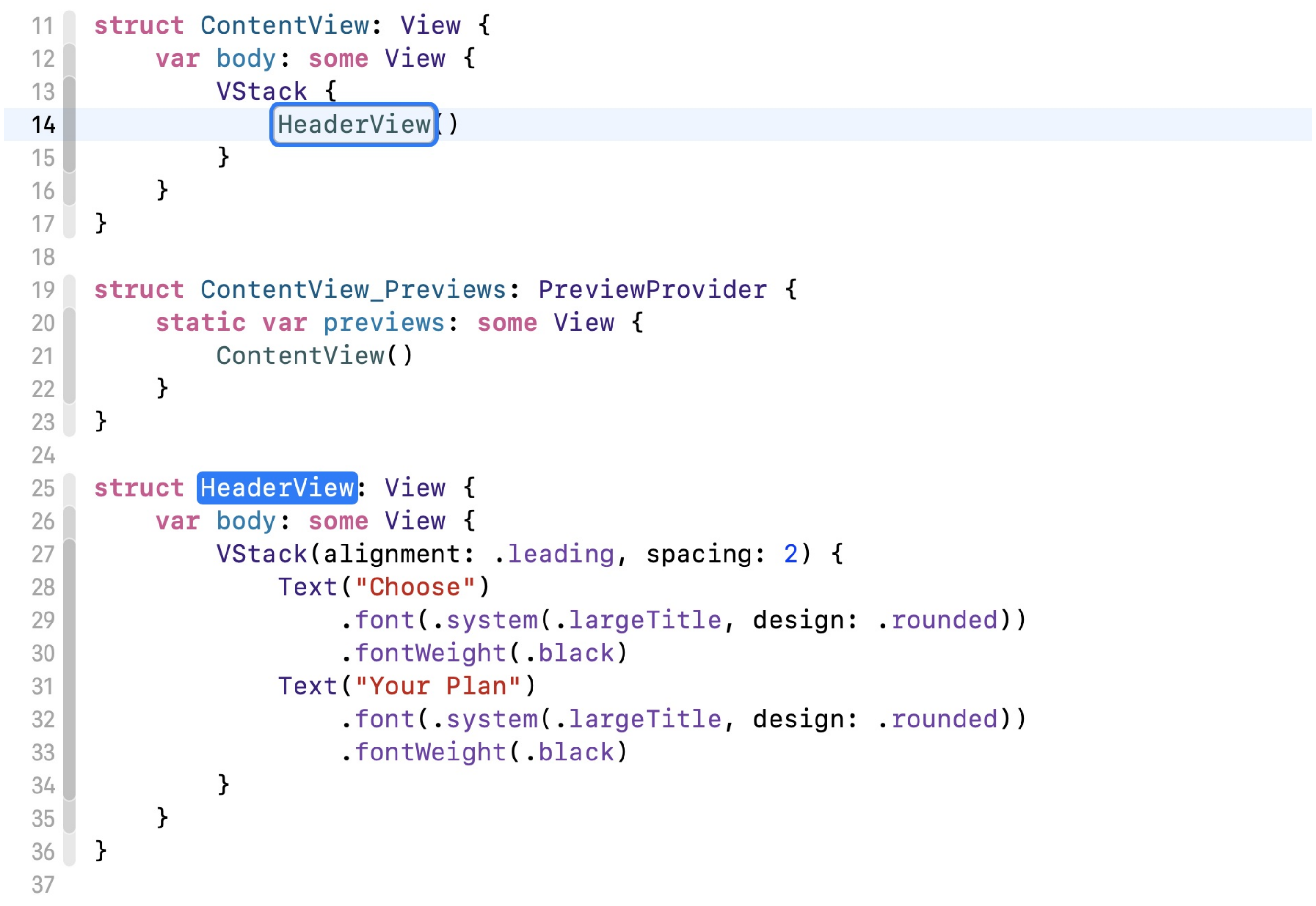
Xcode has a built-in feature to refactor the SwiftUI code. Hold the command key and click the `VStack` that holds the text views. Select *Extract Subview* to extract the code.



```
//
//  C
//  S
//
//  C                       14/8/2020.
//

impor

struc
    v
        VStack
            VStack(alignment: .leading, spacing: 2) {
                Text("Choose")
                    .font(.system(.largeTitle, design: .rounded))
                    .fontWeight(.black)
                Text("Your Plan")
                    .font(.system(.largeTitle, design: .rounded))
                    .fontWeight(.black)
            }
        }
    }
}
```

Figure 10. Extract subview

Xcode extracts the code block and creates a default struct named `ExtractedView`. Rename `ExtractedView` to `HeaderView` to give it a more meaningful name (see the figure below for details).



```
struct ContentView: View {
    var body: some View {
        VStack {
            HeaderView()
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}

struct HeaderView: View {
    var body: some View {
        VStack(alignment: .leading, spacing: 2) {
            Text("Choose")
                .font(.system(.largeTitle, design: .rounded))
                .fontWeight(.black)
            Text("Your Plan")
                .font(.system(.largeTitle, design: .rounded))
                .fontWeight(.black)
        }
    }
}
```

Figure 11. Extract subview

The UI is still the same. However, look at the code block in `ContentView`. It's now much cleaner and easier to read.

Let's continue to implement the UI of the pricing plans. We'll first create the UI for the *Basic* plan. Update `ContentView` like this:

```
struct ContentView: View {
    var body: some View {
        VStack {
            HeaderView()

            VStack {
                Text("Basic")
                    .font(.system(.title, design: .rounded))
                    .fontWeight(.black)
                    .foregroundColor(.white)
                Text("$9")
                    .font(.system(size: 40, weight: .heavy, design: .rounded))
                    .foregroundColor(.white)
                Text("per month")
                    .font(.headline)
                    .foregroundColor(.white)
            }
            .padding(40)
            .background(Color.purple)
            .cornerRadius(10)
        }
    }
}
```

Here we add another `VStack` under `HeaderView`. This `VStack` is used to hold three text views for showing the *Basic* plan. I'll not go into the details of `padding`, `background`, and `cornerRadius` because we have already discussed these modifiers in earlier chapters.

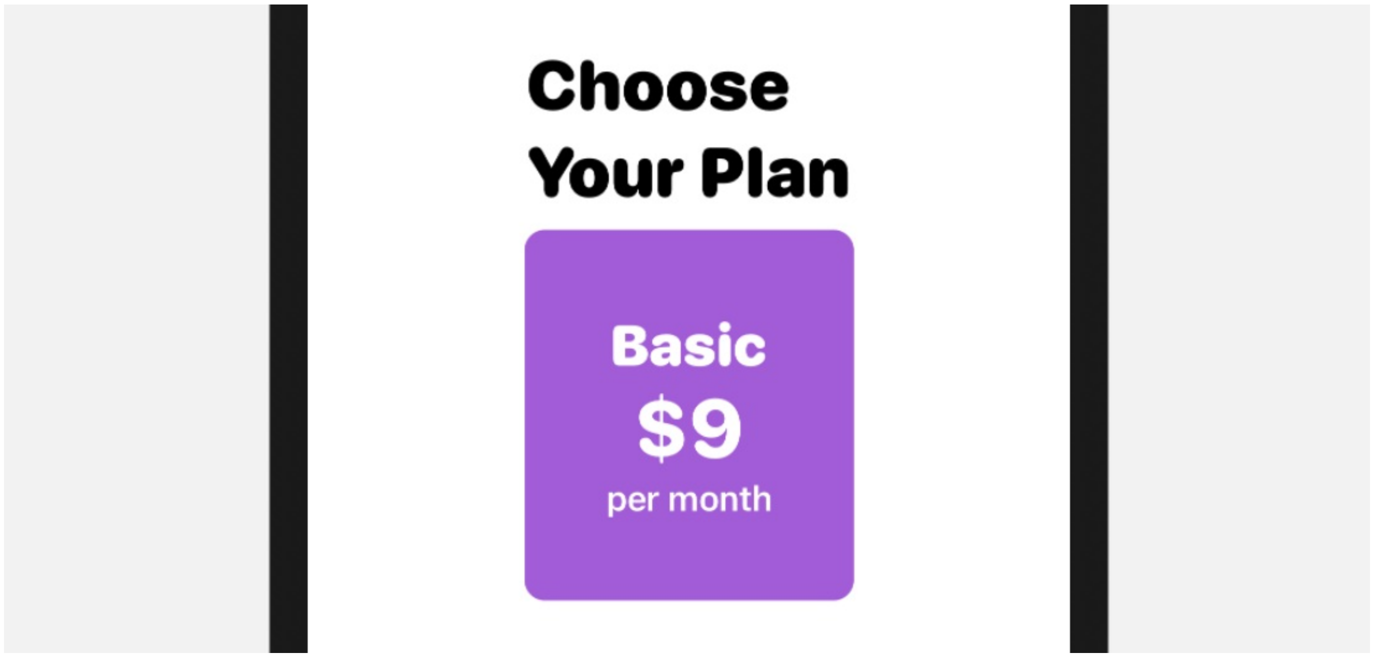
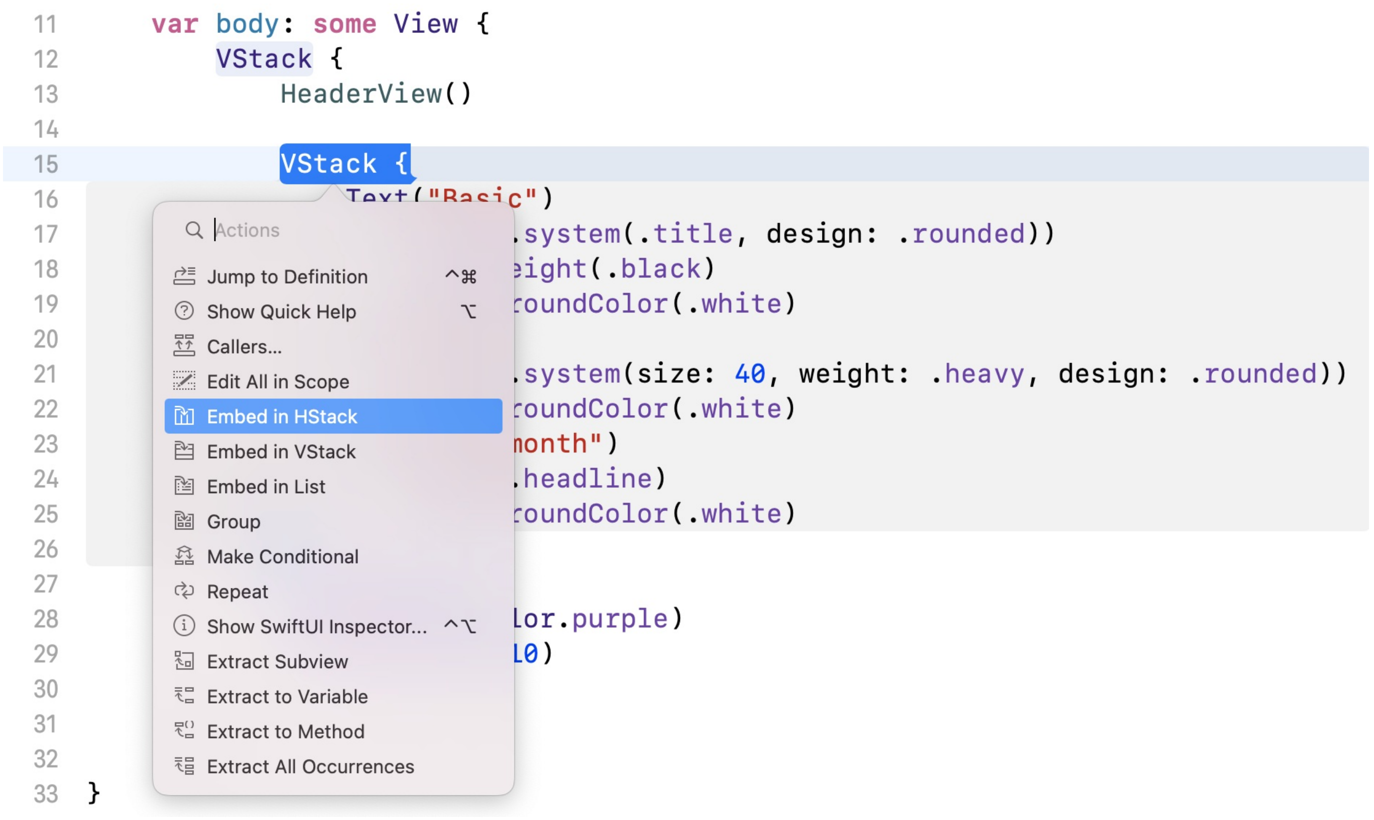


Figure 12. The Basic Plan

Next, we're going to implement the UI of the *Pro* plan. This *Pro* plan should be placed right next the *Basic* plan. In order to do that, you need to embed the `VStack` of the *Basic* plan in a `HStack`. Hold the command key and click the `VStack` keyword. Choose *Embed in HStack*.



```
    var body: some View {
        VStack {
            HeaderView()

            VStack {
                Text("Basic")
                          .system(.title, design: .rounded))
                          eight(.black)
                          roundColor(.white)

                          .system(size: 40, weight: .heavy, design: .rounded))
                          roundColor(.white)
                          month")
                          .headline)
                          roundColor(.white)


                          lor.purple)
                          10)




}
```

Figure 13. Embed in HStack

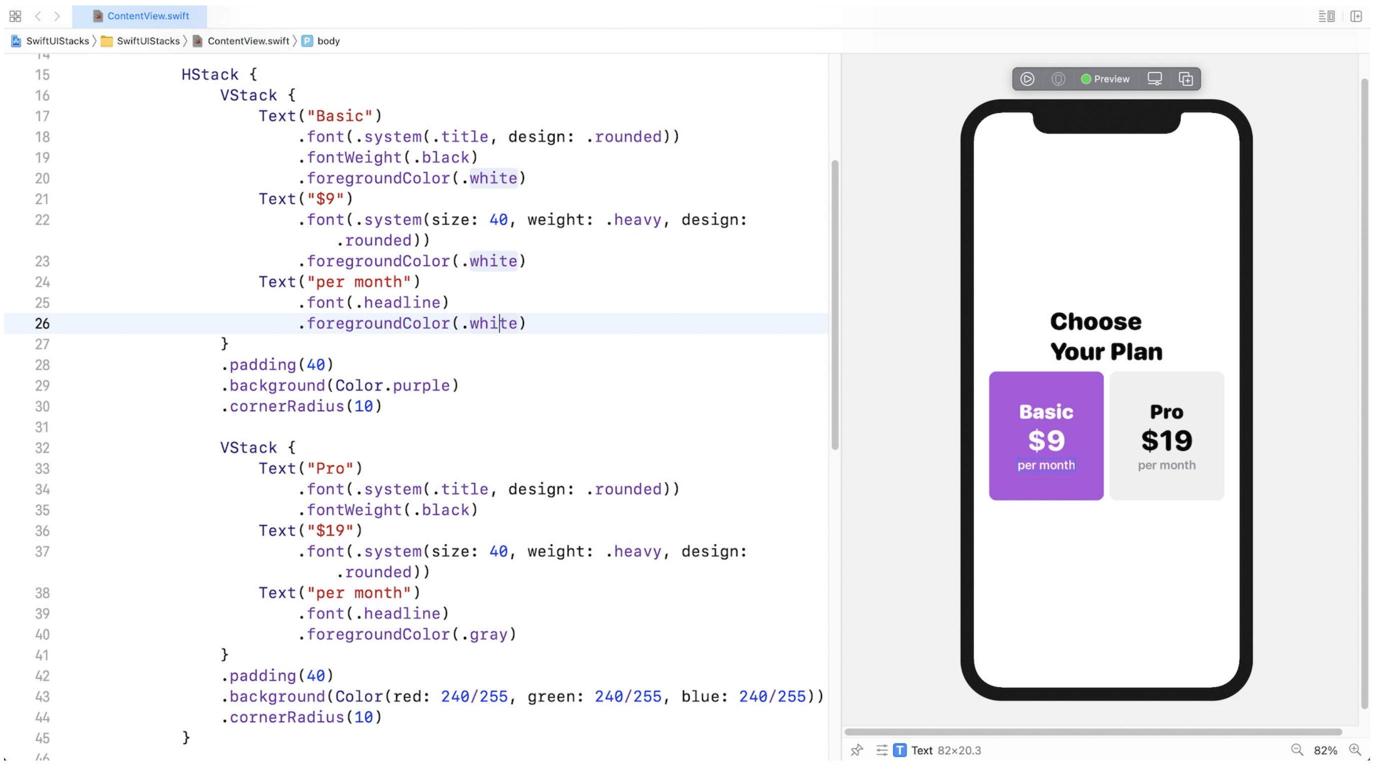
Xcode should insert the code for `HStack` and embed the selected `VStack` in the horizontal stack like this:

```
HStack {
    VStack {
        Text("Basic")
            .font(.system(.title, design: .rounded))
            .fontWeight(.black)
            .foregroundColor(.white)
        Text("$9")
            .font(.system(size: 40, weight: .heavy, design: .rounded))
            .foregroundColor(.white)
        Text("per month")
            .font(.headline)
            .foregroundColor(.white)
    }
    .padding(40)
    .background(Color.purple)
    .cornerRadius(10)
}
```

Now we're ready to create the UI of the *Pro* plan. The code is very similar to that of the *Basic* plan except for the background and text colors. Insert the following code right below `cornerRadius(10)` :

```
VStack {
    Text("Pro")
        .font(.system(.title, design: .rounded))
        .fontWeight(.black)
    Text("$19")
        .font(.system(size: 40, weight: .heavy, design: .rounded))
    Text("per month")
        .font(.headline)
        .foregroundColor(.gray)
}
.padding(40)
.background(Color(red: 240/255, green: 240/255, blue: 240/255))
.cornerRadius(10)
```

As soon as you insert the code, you should see the layout below in the canvas.



```swift
            HStack {
                VStack {
                    Text("Basic")
                        .font(.system(.title, design: .rounded))
                        .fontWeight(.black)
                        .foregroundColor(.white)
                    Text("$9")
                        .font(.system(size: 40, weight: .heavy, design:
                            .rounded))
                        .foregroundColor(.white)
                    Text("per month")
                        .font(.headline)
                        .foregroundColor(.white)
                }
                .padding(40)
                .background(Color.purple)
                .cornerRadius(10)

                VStack {
                    Text("Pro")
                        .font(.system(.title, design: .rounded))
                        .fontWeight(.black)
                    Text("$19")
                        .font(.system(size: 40, weight: .heavy, design:
                            .rounded))
                    Text("per month")
                        .font(.headline)
                        .foregroundColor(.gray)
                }
                .padding(40)
                .background(Color(red: 240/255, green: 240/255, blue: 240/255))
                .cornerRadius(10)
            }
```

Figure 14. Using HStack to layout two views horizontally

The current size of the pricing blocks look similar, but actually they vary depending on the length of the text. Let's say, you change the word "Pro" to "Professional". The gray area will expand to accomodate the change. In short, the view defines its own size and its size is just big enough to fit the content.

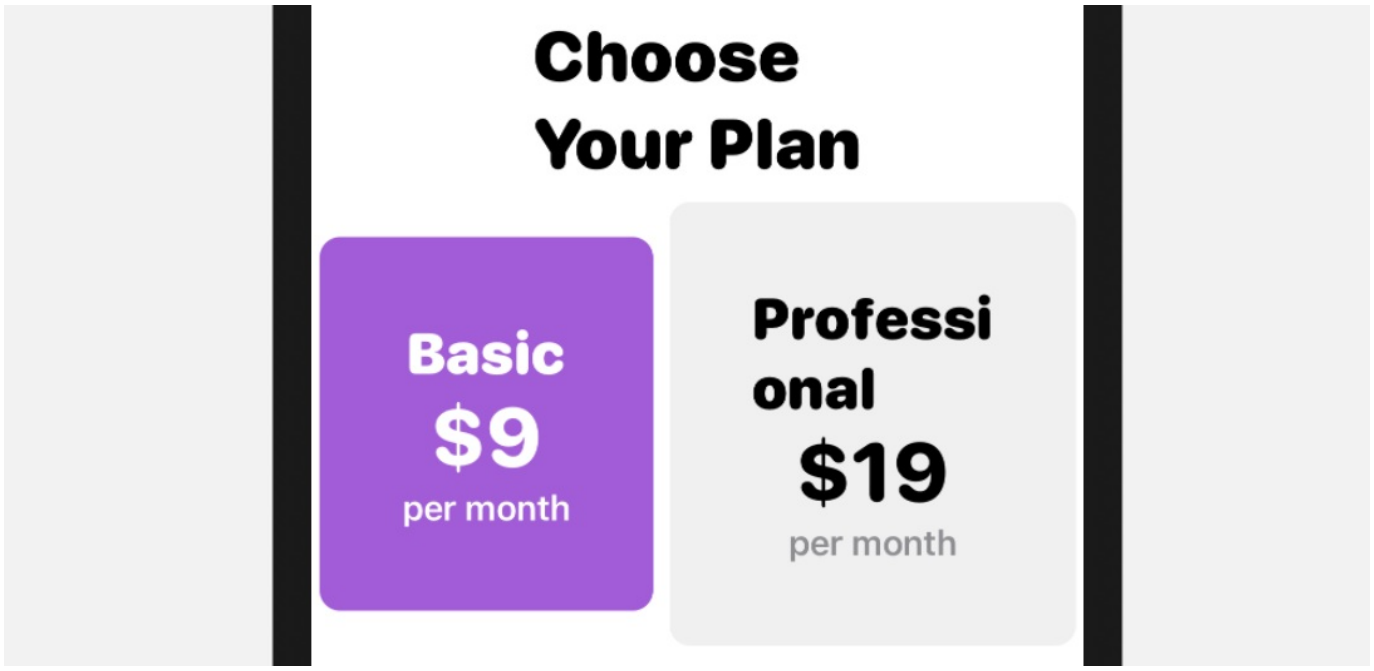


Figure 15. The size of the Pro block becomes wider

If you refer to figure 1 again, both pricing blocks have the same size. To adjust both blocks to have the same size, you can use the `.frame` modifier to set the `maxWidth` to `.infinity` like this:

```
.frame(minWidth: 0, maxWidth: .infinity, minHeight: 100)
```

The `.frame` modifier allows you to define the frame size. You can specify the size as a fixed value. For example, in the code above, we set the `minHeight` to 100 points. When you set the `maxWidth` to `.infinity`, the view will adjust itself to fill the maximum width. For example, if there is only one pricing block, it will take up the whole screen width.

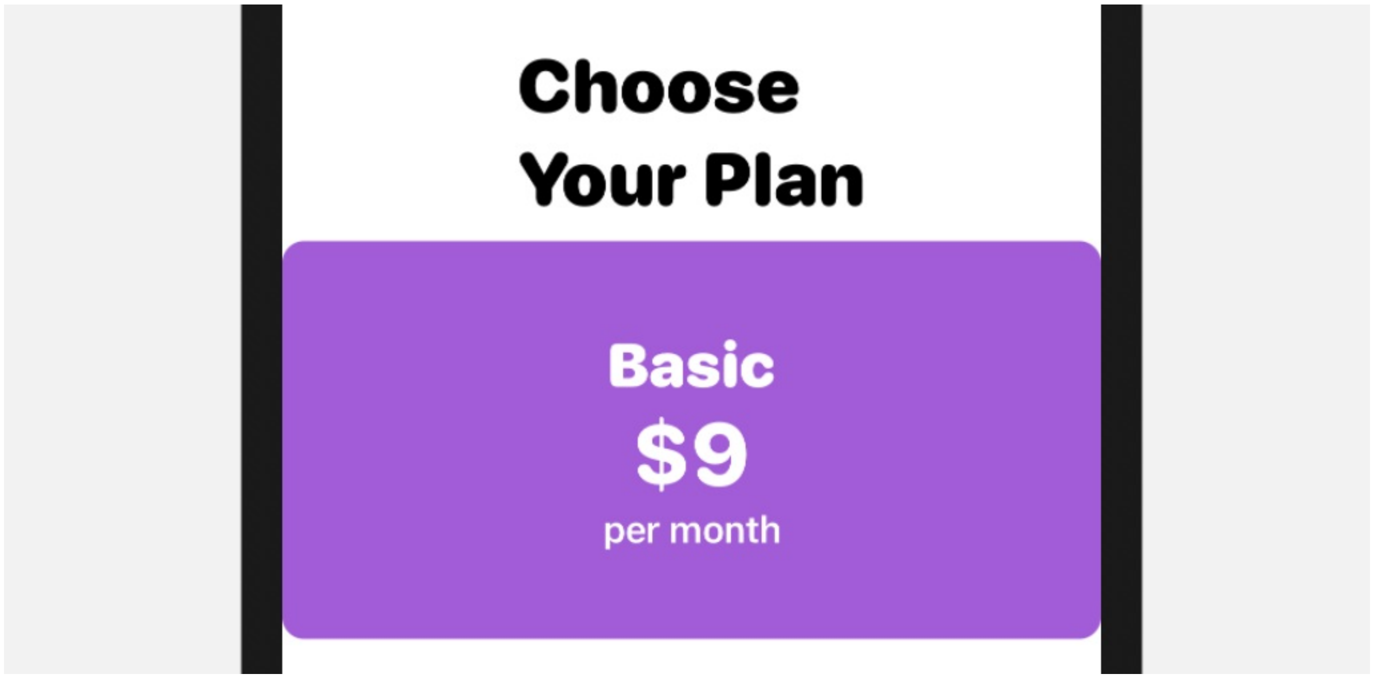
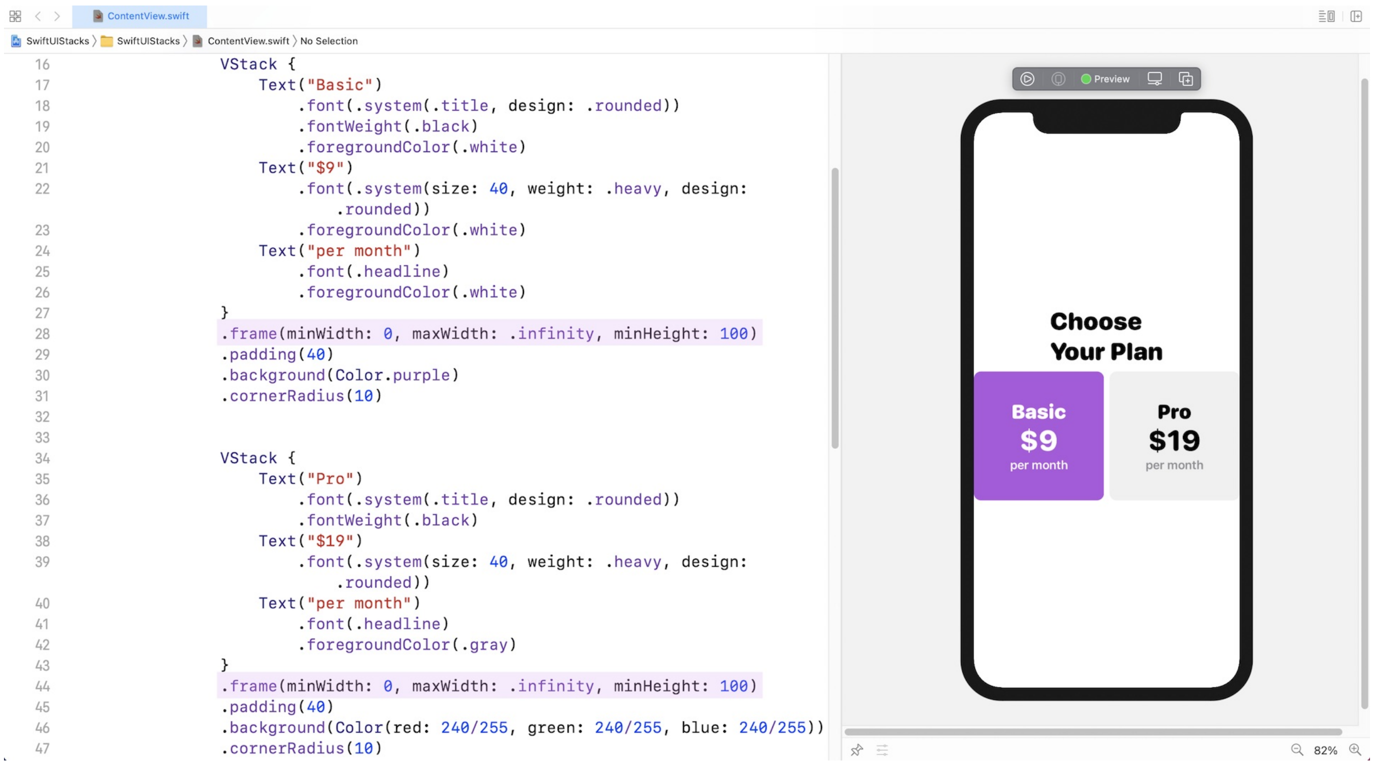


Figure 16. Setting the maxWidth to .infinity

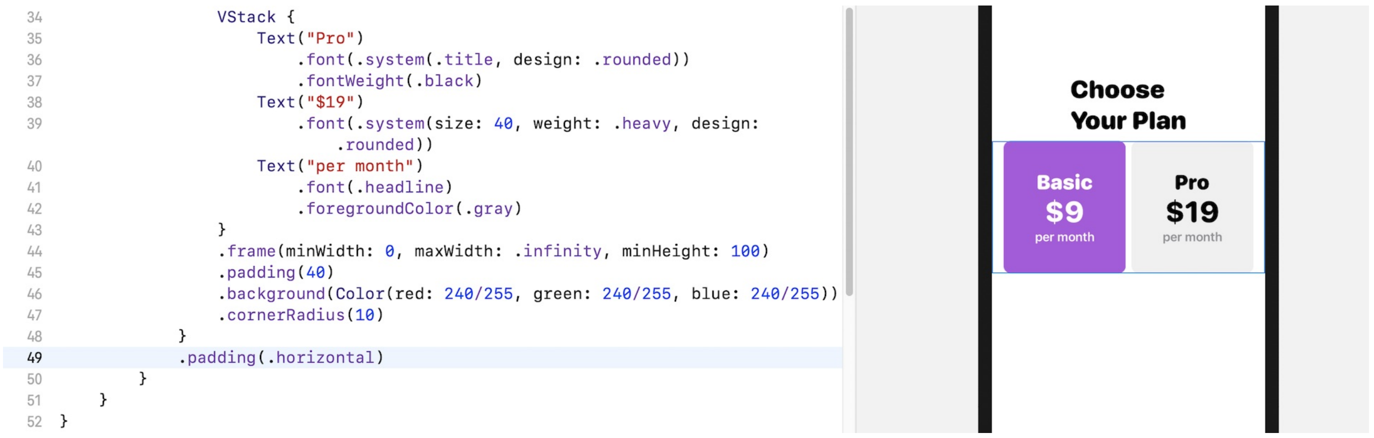
For two pricing blocks, iOS will fill the block equally when `maxWidth` is set to `.infinity`. Now insert the above line of code into each of the pricing blocks. Your result should look like figure 17.



```
            VStack {
                Text("Basic")
                    .font(.system(.title, design: .rounded))
                    .fontWeight(.black)
                    .foregroundColor(.white)
                Text("$9")
                    .font(.system(size: 40, weight: .heavy, design:
                        .rounded))
                    .foregroundColor(.white)
                Text("per month")
                    .font(.headline)
                    .foregroundColor(.white)
            }
            .frame(minWidth: 0, maxWidth: .infinity, minHeight: 100)
            .padding(40)
            .background(Color.purple)
            .cornerRadius(10)


            VStack {
                Text("Pro")
                    .font(.system(.title, design: .rounded))
                    .fontWeight(.black)
                Text("$19")
                    .font(.system(size: 40, weight: .heavy, design:
                        .rounded))
                Text("per month")
                    .font(.headline)
                    .foregroundColor(.gray)
            }
            .frame(minWidth: 0, maxWidth: .infinity, minHeight: 100)
            .padding(40)
            .background(Color(red: 240/255, green: 240/255, blue: 240/255))
            .cornerRadius(10)
```

Figure 17. Arranging both pricing blocks with equal width

To give the horizontal stack some spacing, you can add a `.padding` modifier like this:



```
            VStack {
                Text("Pro")
                    .font(.system(.title, design: .rounded))
                    .fontWeight(.black)
                Text("$19")
                    .font(.system(size: 40, weight: .heavy, design:
                        .rounded))
                Text("per month")
                    .font(.headline)
                    .foregroundColor(.gray)
            }
            .frame(minWidth: 0, maxWidth: .infinity, minHeight: 100)
            .padding(40)
            .background(Color(red: 240/255, green: 240/255, blue: 240/255))
            .cornerRadius(10)
        }
        .padding(.horizontal)
    }
}
}
```

Figure 18. Adding some paddings for the stack view

The `.horizontal` parameter means we want to add some padding for both leading and trailing sides of the `HStack`.

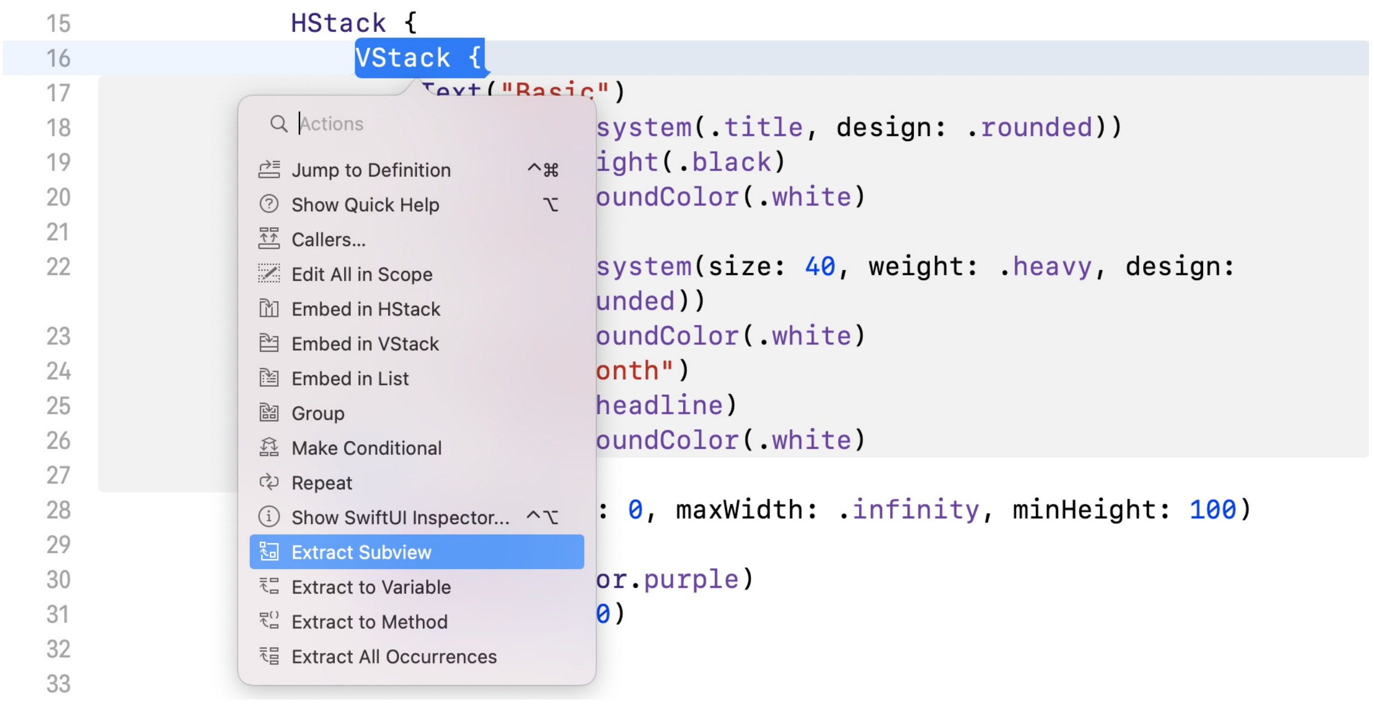
Organizing the Code

Again, before we lay out the rest of the UI components, let's refactor the current code to make it more organized. If you look at both stacks that are used to lay out the *Basic* and *Pro* pricing plan, the code is very similar except the following items:

- the name of the pricing plan
- the price
- the text color
- the background color of the pricing block

To streamline the code and improve reusability, we can extract the `VStack` code block and make it adaptable to different values of the pricing plan.

Go back to the code editor. Hold the command key and click the `VStack` of the *Basic* plan. Once Xcode extracts the code, rename the subview from `ExtractedView` to `PricingView`.



```
            HStack {
                VStack {
                    Text("Basic")
                                  system(.title, design: .rounded))
                                  ight(.black)
                                  oundColor(.white)

                                  system(size: 40, weight: .heavy, design:
                                  unded))
                                  oundColor(.white)
                                  onth")
                                  headline)
                                  oundColor(.white)

                                  : 0, maxWidth: .infinity, minHeight: 100)

                                  or.purple)
                                  0)
```

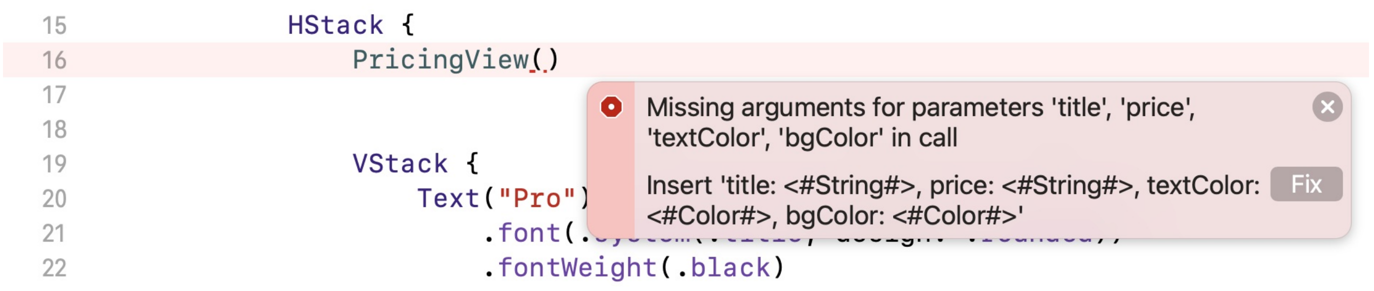
Figure 19. Extracting the subview

As we mentioned earlier, the `PricingView` should be flexible to display different pricing plans. We will add four variables in the `PricingView` struct. Update `PricingView` like this:

```
struct PricingView: View {

    var title: String
    var price: String
    var textColor: Color
    var bgColor: Color

    var body: some View {
        VStack {
            Text(title)
                .font(.system(.title, design: .rounded))
                .fontWeight(.black)
                .foregroundColor(textColor)
            Text(price)
                .font(.system(size: 40, weight: .heavy, design: .rounded))
                .foregroundColor(textColor)
            Text("per month")
                .font(.headline)
                .foregroundColor(textColor)
        }
        .frame(minWidth: 0, maxWidth: .infinity, minHeight: 100)
        .padding(40)
        .background(bgColor)
        .cornerRadius(10)
    }
}
```

We added variables for the title, price, text, and background color of the pricing block. Furthermore, we make use of these variables in the code to update the title, price, text and background color accordingly.

Once you make the changes, you'll see an error telling you that there are some missing arguments for the `PricingView`.



```
HStack {
    PricingView()


    VStack {
        Text("Pro")
            .font(.system(.title, design: .rounded))
            .fontWeight(.black)
```

Missing arguments for parameters 'title', 'price', 'textColor', 'bgColor' in call

Insert 'title: <#String#>, price: <#String#>, textColor: <#Color#>, bgColor: <#Color#>'

Figure 20. Xcode indicates an error on the PricingView

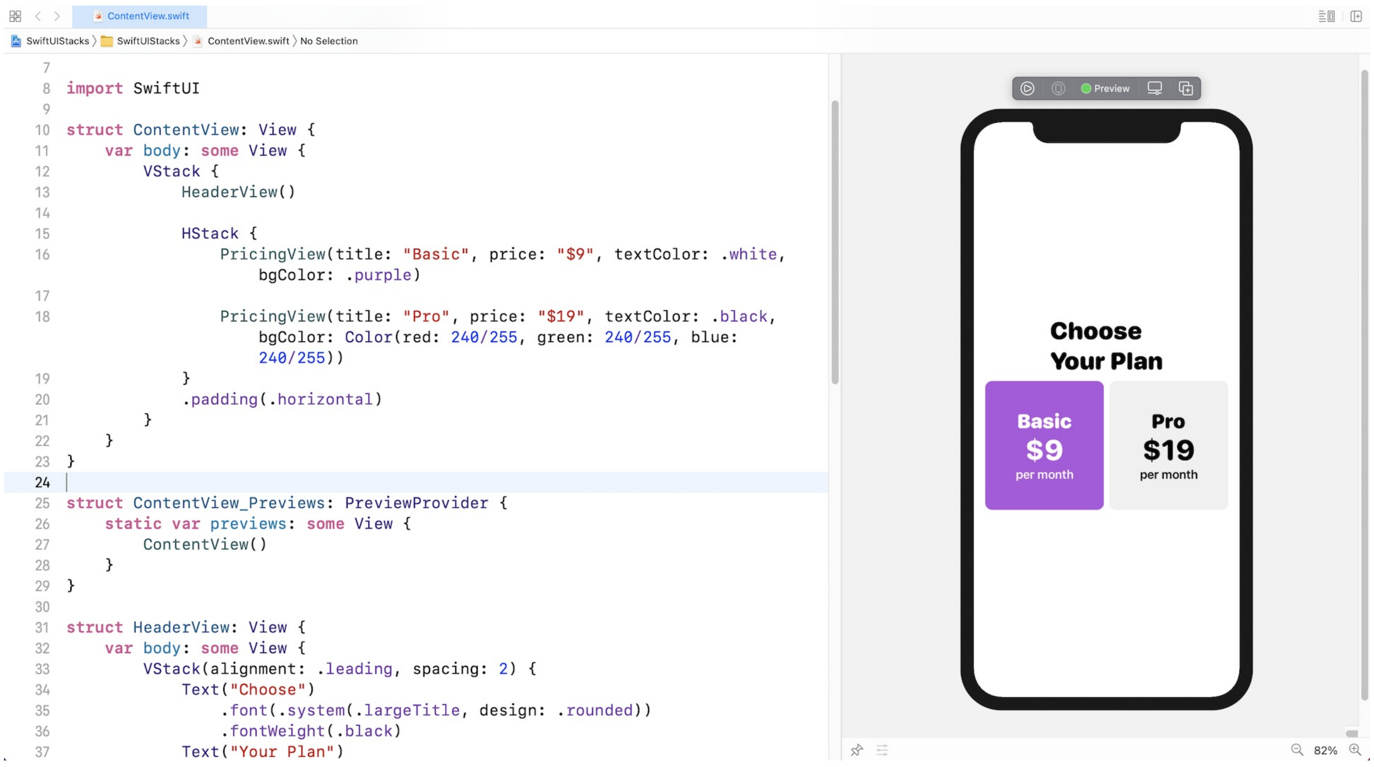
Earlier, we introduced four variables in the view. When calling `PricingView`, we must now provide the values for these parameters. So, change `PricingView()` to add the values:

```
PricingView(title: "Basic", price: "$9", textColor: .white, bgColor: .purple)
```

Also, you can replace the `VStack` of the *Pro* plan using `PricingView` like this:

```
PricingView(title: "Pro", price: "$19", textColor: .black, bgColor: Color(red: 240/
255, green: 240/255, blue: 240/255))
```

The layout of the pricing blocks is the same but the underlying code, as you can see, is much cleaner and easier to read.



```swift
import SwiftUI

struct ContentView: View {
    var body: some View {
        VStack {
            HeaderView()

            HStack {
                PricingView(title: "Basic", price: "$9", textColor: .white,
                    bgColor: .purple)

                PricingView(title: "Pro", price: "$19", textColor: .black,
                    bgColor: Color(red: 240/255, green: 240/255, blue:
                    240/255))
            }
            .padding(.horizontal)
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}

struct HeaderView: View {
    var body: some View {
        VStack(alignment: .leading, spacing: 2) {
            Text("Choose")
                .font(.system(.largeTitle, design: .rounded))
                .fontWeight(.black)
            Text("Your Plan")
```

Figure 21. ContentView after refactoring the code

Using ZStack

Now that you've laid out the pricing blocks and refactored the code, there is still one thing missing for the *Pro* pricing. We want to overlay a message in yellow on the pricing block. To do that, we can use the `ZStack` that allows you to overlay a view on top of an existing view.

Embed the `PricingView` of the *Pro* plan within a `ZStack` and add the `Text` view, like this:

```
ZStack {
    PricingView(title: "Pro", price: "$19", textColor: .black, bgColor: Color(red:
240/255, green: 240/255, blue: 240/255))

    Text("Best for designer")
        .font(.system(.caption, design: .rounded))
        .fontWeight(.bold)
        .foregroundColor(.white)
        .padding(5)
        .background(Color(red: 255/255, green: 183/255, blue: 37/255))
}
```

The order of the views embedded in the `ZStack` determine how the views are overlaid with each other. For the code above, the `Text` view will overlay on top of the pricing view. In the canvas, you should see the pricing layout like this:

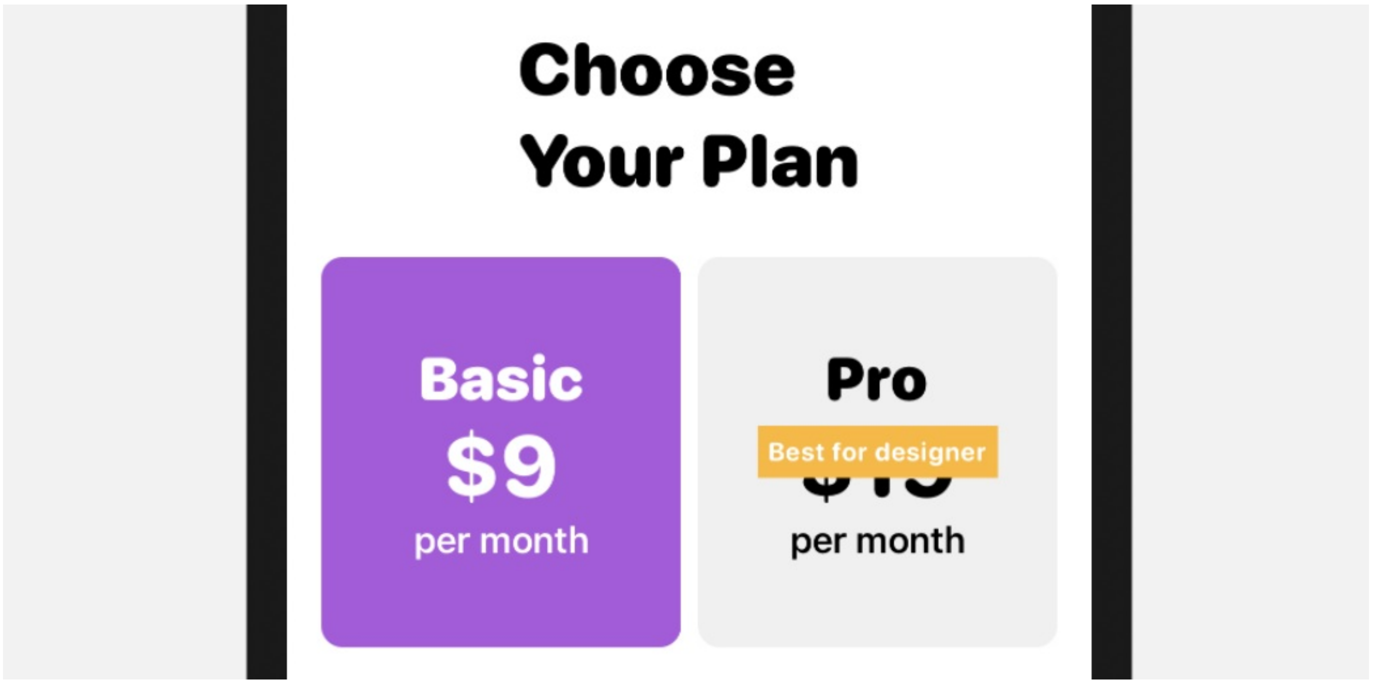
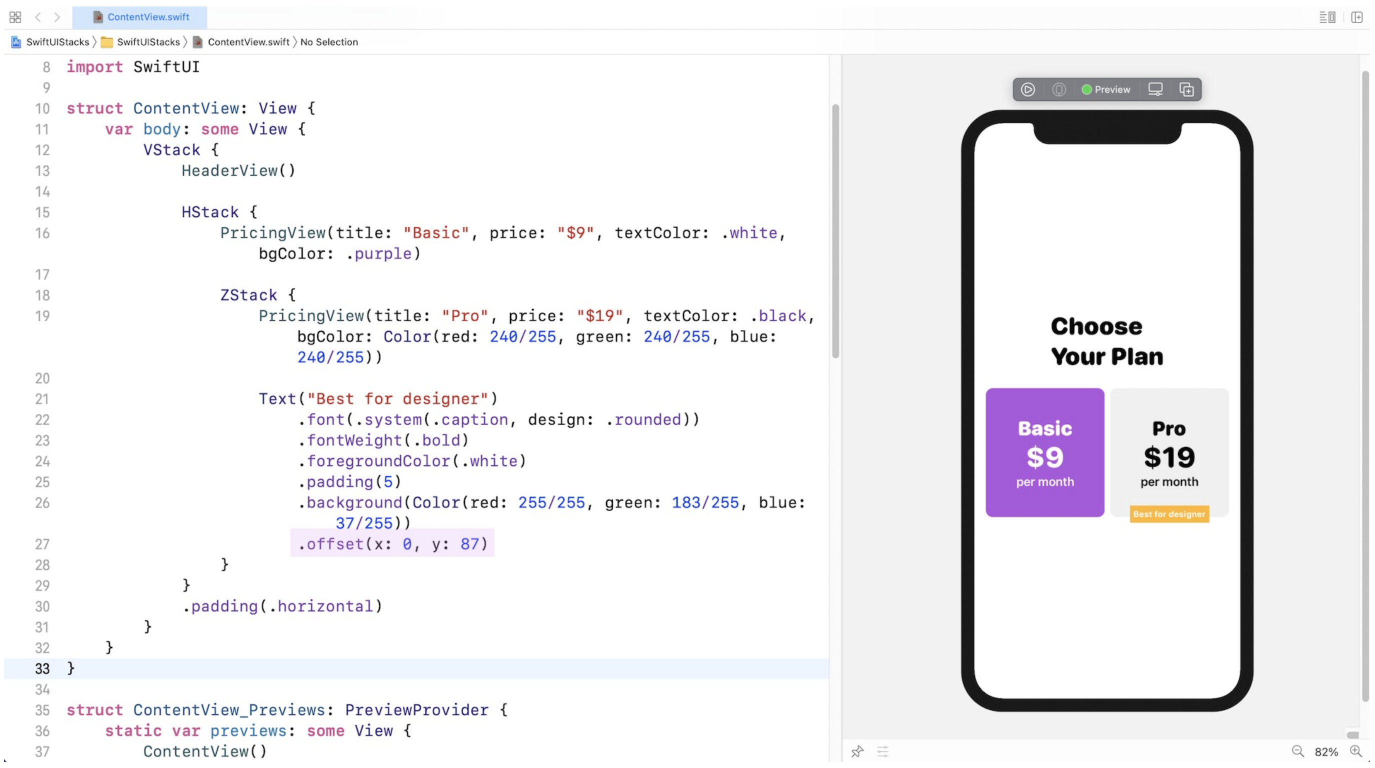


Figure 22. ContentView after refactoring the code

To adjust the position of the text, you can use the `.offset` modifier. Insert the following line of code at the end of the `Text` view:

```
.offset(x: 0, y: 87)
```

The *Best for designer* label will move to the bottom of the block. A negative value of `y` will move the label to the top part if you want to re-position it.



```
import SwiftUI

struct ContentView: View {
    var body: some View {
        VStack {
            HeaderView()

            HStack {
                PricingView(title: "Basic", price: "$9", textColor: .white,
                    bgColor: .purple)

                ZStack {
                    PricingView(title: "Pro", price: "$19", textColor: .black,
                        bgColor: Color(red: 240/255, green: 240/255, blue:
                        240/255))

                    Text("Best for designer")
                        .font(.system(.caption, design: .rounded))
                        .fontWeight(.bold)
                        .foregroundColor(.white)
                        .padding(5)
                        .background(Color(red: 255/255, green: 183/255, blue:
                            37/255))
                        .offset(x: 0, y: 87)
                }
            }
            .padding(.horizontal)
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
```

Figure 23. Position the text view using .offset

Optionally, if you want to adjust the spacing between the *Basic* and *Pro* pricing block, you can specify the `spacing` parameter within a `HStack` like this:

```
HStack(spacing: 15) {
  ...
}
```

Exercise #1

We haven't finished yet. I want to discuss how we handle optionals in SwiftUI and introduce another view component called Spacer. However, before we continue, let's have a simple exercise. Your task is to lay out the *Team* pricing plan as shown in figure 24. The image I used is a system image with the name "wand.and.rays" from SF Symbols.

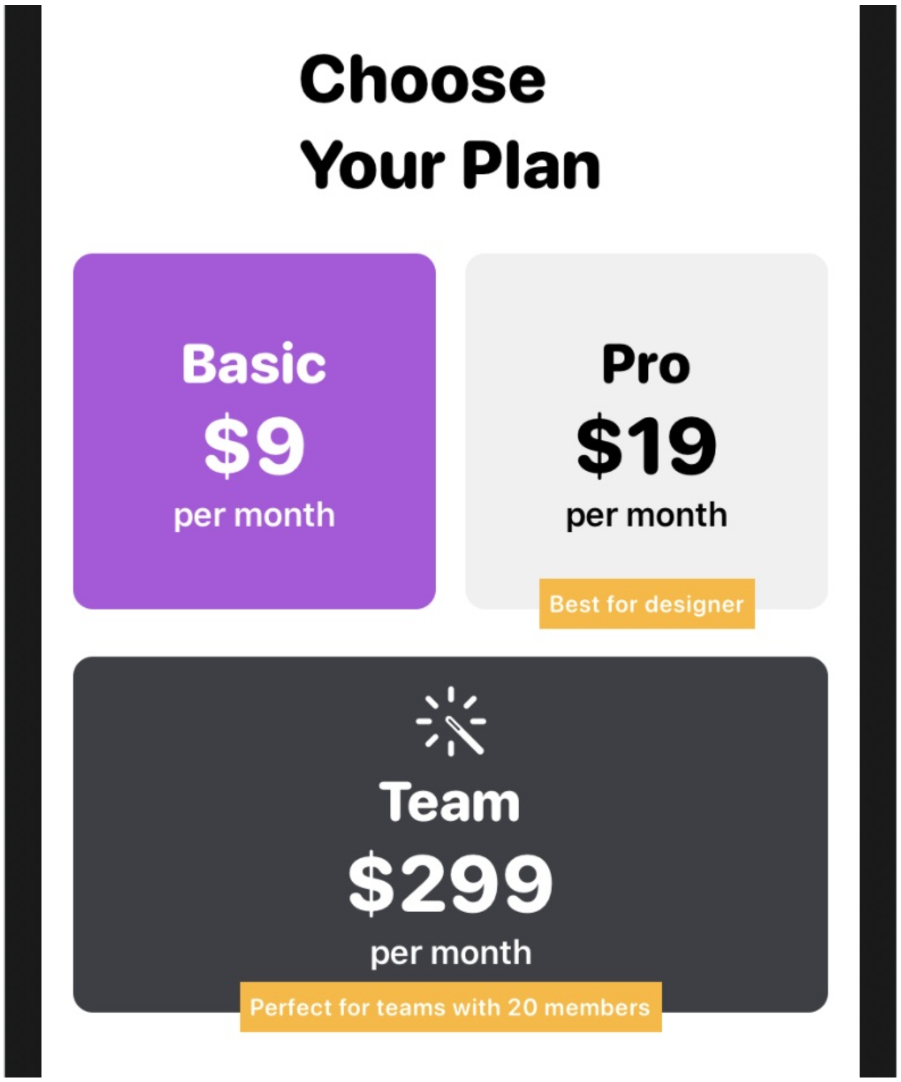


Figure 24. Adding the Team plan

Please don't look at the solution, try to develop your own solution.

Handling Optionals in SwiftUI

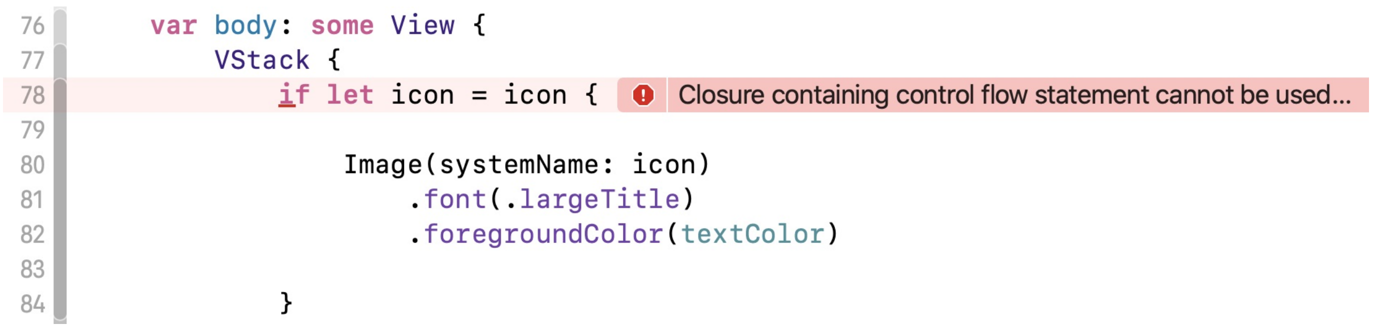
Have you tried the exercise and come up with your own solution? The layout of the *Team* plan is very similar to the *Basic* & *Pro* plans. You could replicate the `VStack` of these two plans and create the *Team* plan. however, let me show you a more elegant solution.

We can reuse the `PricingView` to create the *Team* plan. However, as you are aware, the *Team* plan has an icon that sits above the title. In order to lay out this icon, we need to modify `PricingView` to accomodate an icon. Since the icon is not mandatory for a pricing plan, we declare an optional in `PricingView`:

```
var icon: String?
```

If you're new to Swift, an optional means that the variable may or may not have a value. In the example above, we define a variable named `icon` of the type `String`. The call to the method is expected to pass the image name if the pricing plan is required to display an icon. Otherwise, this variable is set to `nil` by default.

So, how do you handle an optional in SwiftUI? In Swift, you usually use `if let` to check if an optional has a value and unwrap it. If you are still using Xcode 11, you can't use `if let` in SwiftUI. You will end up with the following error:



```
    var body: some View {
        VStack {
            if let icon = icon {    Closure containing control flow statement cannot be used...

                Image(systemName: icon)
                    .font(.largeTitle)
                    .foregroundColor(textColor)

            }
```

Figure 25. Using if let in Xcode 11

One way to work with optionals in Xcode 11 is to check if the optional has a non-nil value. For example, we need to check if `icon` has a value before displaying an image. We can write the code like this:

```
if icon != nil {

    Image(systemName: icon!)
        .font(.largeTitle)
        .foregroundColor(textColor)

}
```

In Xcode 12, the SwiftUI framework supports the usage of `if let`. The code can be rewritten like this:

```
if let icon = icon {

    Image(systemName: icon)
        .font(.largeTitle)
        .foregroundColor(textColor)

}
```

To support the rendering of an icon, the final code of `PricingView` should be updated as below:

```
struct PricingView: View {

    var title: String
    var price: String
    var textColor: Color
    var bgColor: Color
    var icon: String?

    var body: some View {
        VStack {

            icon.map({
                Image(systemName: $0)
                    .font(.largeTitle)
                    .foregroundColor(textColor)
            })

            Text(title)
                .font(.system(.title, design: .rounded))
                .fontWeight(.black)
                .foregroundColor(textColor)
            Text(price)
                .font(.system(size: 40, weight: .heavy, design: .rounded))
                .foregroundColor(textColor)
            Text("per month")
                .font(.headline)
                .foregroundColor(textColor)
        }
        .frame(minWidth: 0, maxWidth: .infinity, minHeight: 100)
        .padding(40)
        .background(bgColor)
        .cornerRadius(10)
    }
}
```

Once you make this change, you can create the *Team* plan by using `ZStack` and `PricingView`. You put the code in `ContentView` and insert it after `.padding(.horiontal)`:

```
ZStack {
    PricingView(title: "Team", price: "$299", textColor: .white, bgColor: Color(re
d: 62/255, green: 63/255, blue: 70/255), icon: "wand.and.rays")
        .padding()

    Text("Perfect for teams with 20 members")
        .font(.system(.caption, design: .rounded))
        .fontWeight(.bold)
        .foregroundColor(.white)
        .padding(5)
        .background(Color(red: 255/255, green: 183/255, blue: 37/255))
        .offset(x: 0, y: 87)
}
```

Using Spacer

When comparing your current UI with that of figure 1, do you see any difference? There are a couple of differences you may notice:

1. The *Choose Your Plan* label is not left aligned.
2. The *Choose Your Plan* label and the pricing plans should be aligned to the top of the screen.

In UIKit, you would define auto layout constraints to position the views. SwiftUI doesn't have auto layout. Instead, it provides a view called `Spacer` for you to create complex layouts.

> A flexible space that expands along the major axis of its containing stack layout, or on both axes if not contained in a stack.
>
> - SwiftUI documentation (https://developer.apple.com/documentation/swiftui/spacer)

To fix the left alignment, let's update the `HeaderView` like this:

```
struct HeaderView: View {
    var body: some View {
        HStack {
            VStack(alignment: .leading, spacing: 2) {
                Text("Choose")
                    .font(.system(.largeTitle, design: .rounded))
                    .fontWeight(.black)
                Text("Your Plan")
                    .font(.system(.largeTitle, design: .rounded))
                    .fontWeight(.black)
            }

            Spacer()
        }
        .padding()
    }
}
```

Here we embed the original `VStack` and a `Spacer` within a `HStack`. By using a `Spacer`, we push the `VStack` to the left. Figure 26 illustrates how the spacer works.

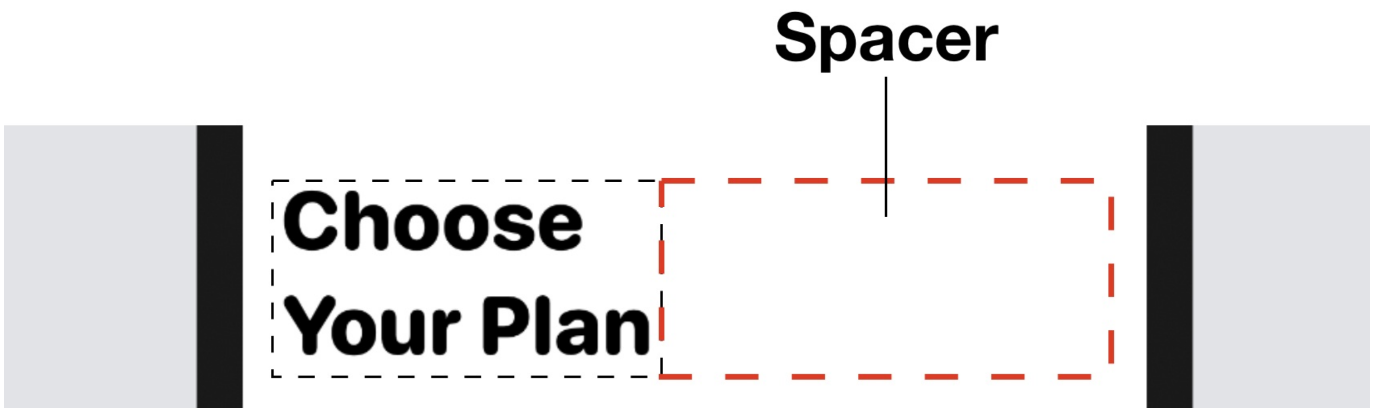
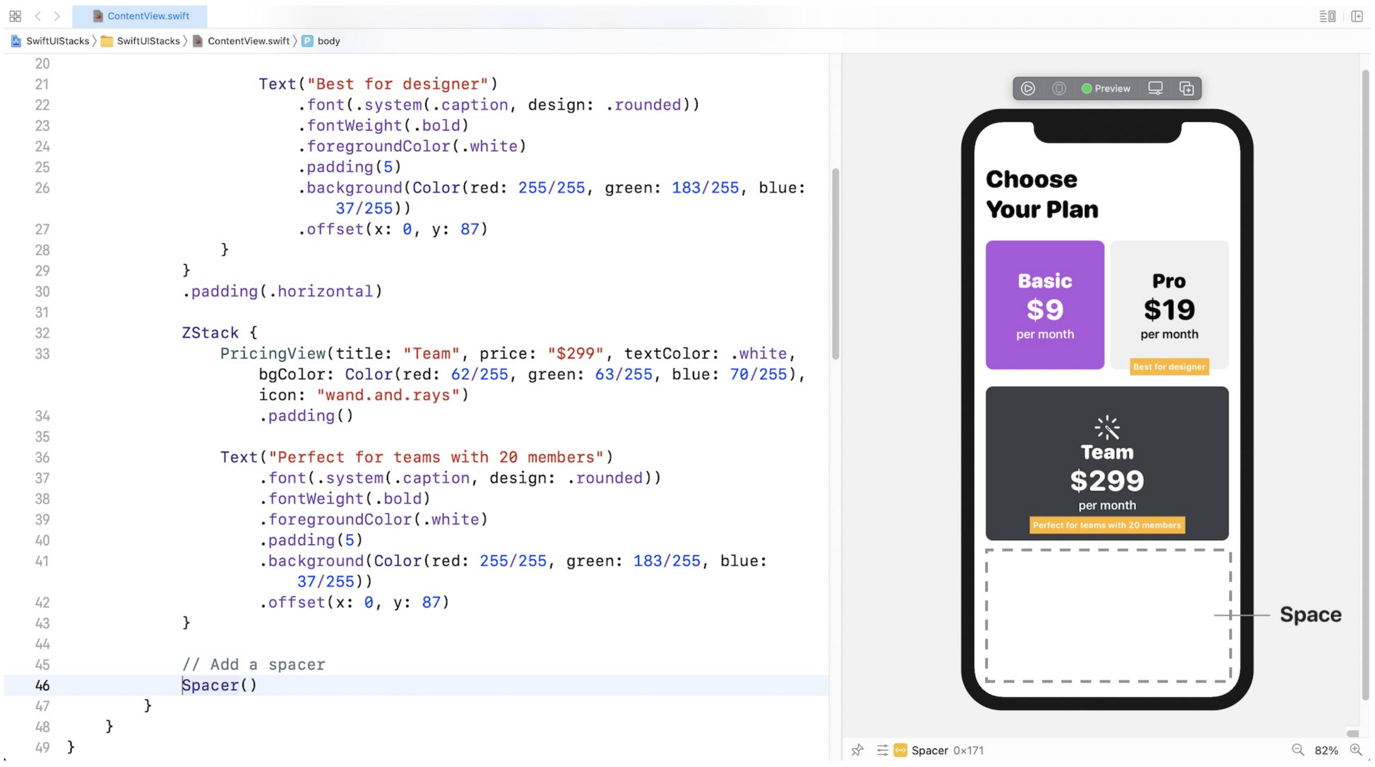


Figure 26. Using Spacer in HStack

You may now know how to fix the second difference. The solution is to add a spacer at the end of the `VStack` of `ContentView` like this:

```
struct ContentView: View {
    var body: some View {
        VStack {
            HeaderView()

            HStack(spacing: 15) {
                ...
            }
            .padding(.horizontal)

            ZStack {
                ...
            }

              // Add a spacer
            Spacer()
        }
    }
}
```

Figure 27 illustrates how the spacer works visually.



```

                    Text("Best for designer")
                        .font(.system(.caption, design: .rounded))
                        .fontWeight(.bold)
                        .foregroundColor(.white)
                        .padding(5)
                        .background(Color(red: 255/255, green: 183/255, blue: 37/255))
                        .offset(x: 0, y: 87)
                }
            }
            .padding(.horizontal)

            ZStack {
                PricingView(title: "Team", price: "$299", textColor: .white, bgColor: Color(red: 62/255, green: 63/255, blue: 70/255), icon: "wand.and.rays")
                    .padding()

                Text("Perfect for teams with 20 members")
                    .font(.system(.caption, design: .rounded))
                    .fontWeight(.bold)
                    .foregroundColor(.white)
                    .padding(5)
                    .background(Color(red: 255/255, green: 183/255, blue: 37/255))
                    .offset(x: 0, y: 87)
            }

            // Add a spacer
            Spacer()
        }
    }
}
```

Figure 27. Using spacer in VStack

Exercise #2

Now that you know how `VStack`, `HStack`, and `ZStack` work, your final exercise is to create a layout as shown in figure 28. For the icons, I use system images from SF Symbols. You're free to pick any of the images instead of following mine. As a hint, you can use the modifier `.scale` to scale up/down a view. For example, if you attach `.scale(0.5)` to a view, it automatically resizes the view to half of its original size.

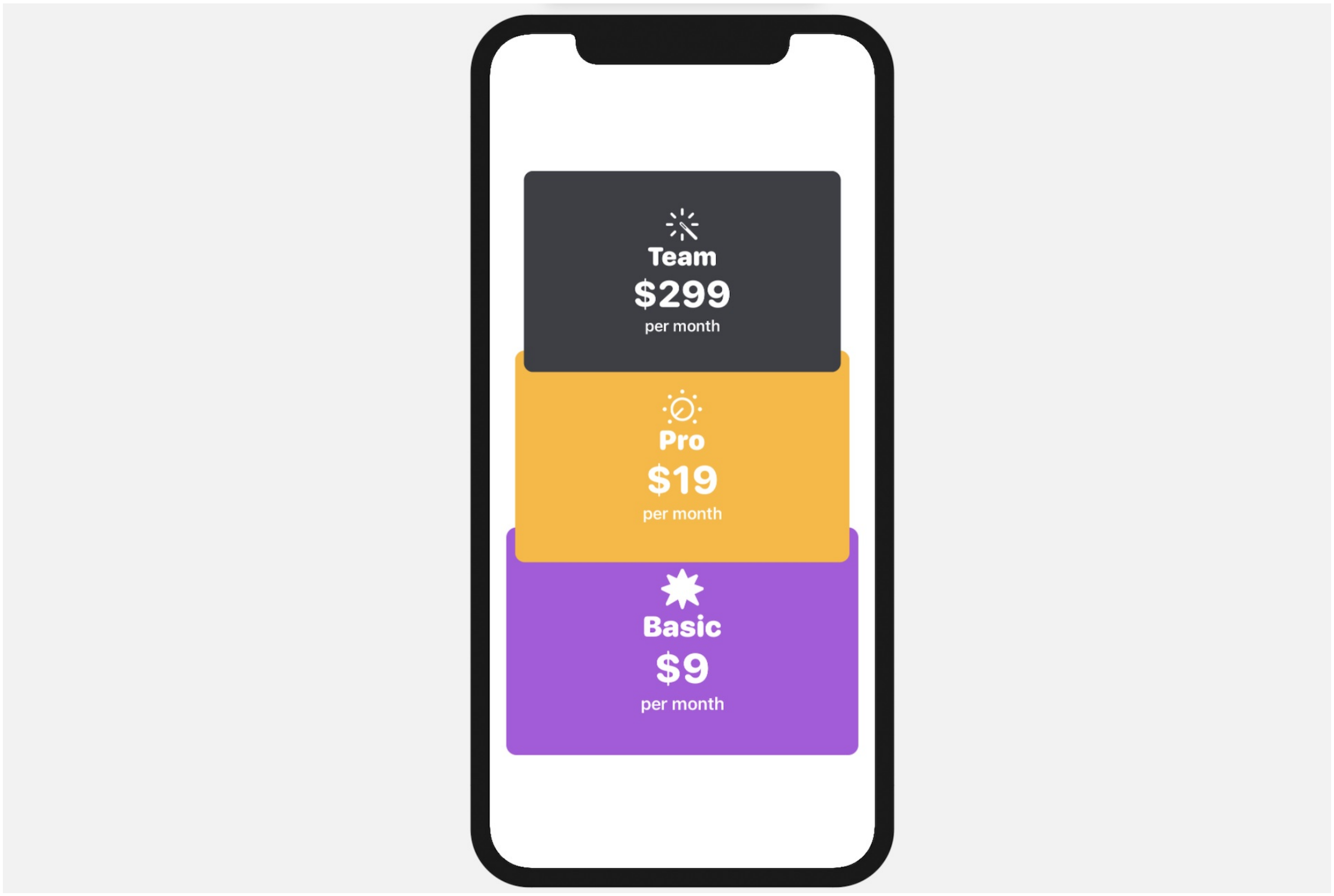


Figure 28. Your exercise

For reference, you can download the complete stack projects here:

- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUIStacks.zip)
- Solution to exercise #2 (https://www.appcoda.com/resources/swiftui2/SwiftUIStacksExercise.zip)

Chapter 5
Understanding ScrollView and Building a Carousel UI

After going through the previous chapter, I believe you should now understand how to build a complex UI using stacks. Of course, it will take you a lot of practice before you can master SwiftUI. Therefore, before we dive deep into ScrollView to make the views scrollable, let's begin this chapter with a challenge. Your task is to create a card view like that shown in figure 1.

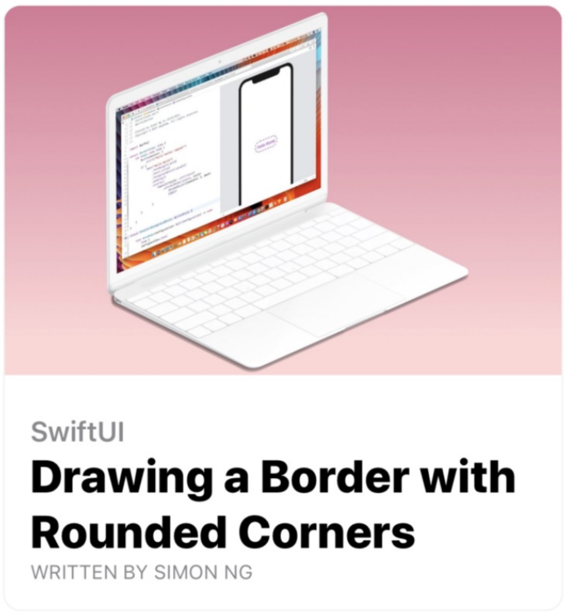


Figure 1. The card view

By using stacks, image, and text views, you should be able to create the UI. While I will go through the implementation step by step with you later, please take some time to work on the exercise and figure out your own solution.

Once you create the card view, I will discuss `ScrollView` with you and build a scrollable interface using the card view. Figure 2 shows you the complete UIs.

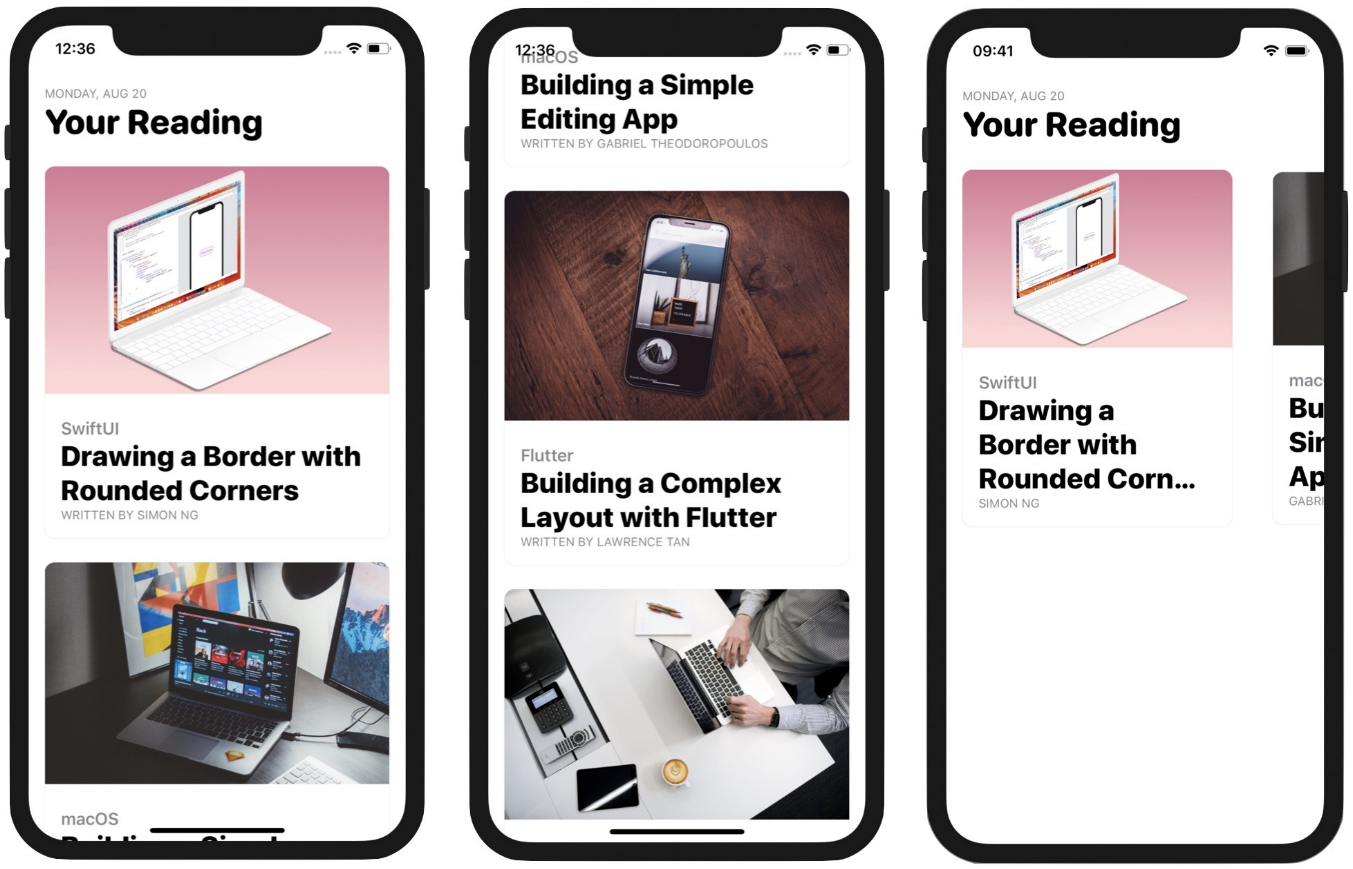

12:36
MONDAY, AUG 20
Your Reading
SwiftUI
Drawing a Border with Rounded Corners
WRITTEN BY SIMON NG
macOS
Building a Simple Editing App
WRITTEN BY GABRIEL THEODOROPOULOS
Flutter
Building a Complex Layout with Flutter
WRITTEN BY LAWRENCE TAN
09:41
MONDAY, AUG 20
Your Reading
SwiftUI
Drawing a Border with Rounded Corn...
SIMON NG

Figure 2. Building a scrollable UI with ScrollView

Creating a Card-like UI

If you haven't opened Xcode, fire it up and create a new project using the *App* template (under iOS). In the next screen, set the product name to `SwiftUIScrollView` (or whatever name you like) and fill in all the required values. Make sure you select `SwiftUI` for the *Interface* option.

So far, we have coded the user interface in the `ContentView.swift` file. It's very likely you wrote your solution code there. That's completely fine, but I want to show you a better way to organize your code. For the implementation of the card view, let's create a separate file. In the project navigator, right click `SwiftUIScrollView` and choose *New File...*

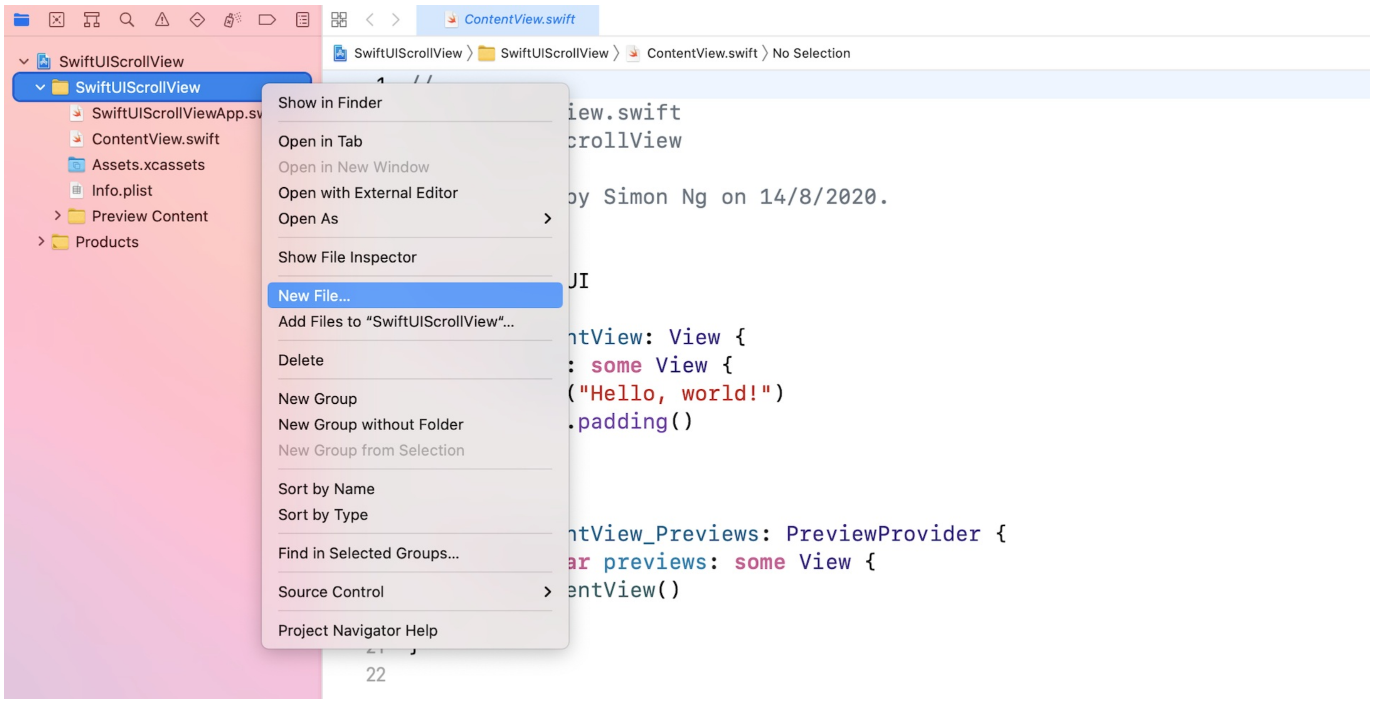


Figure 3. Creating a new file

In the *User Interface* section, choose the *SwiftUI View* template and click *Next* to create the file. Name the file `CardView` and save it in the project folder.

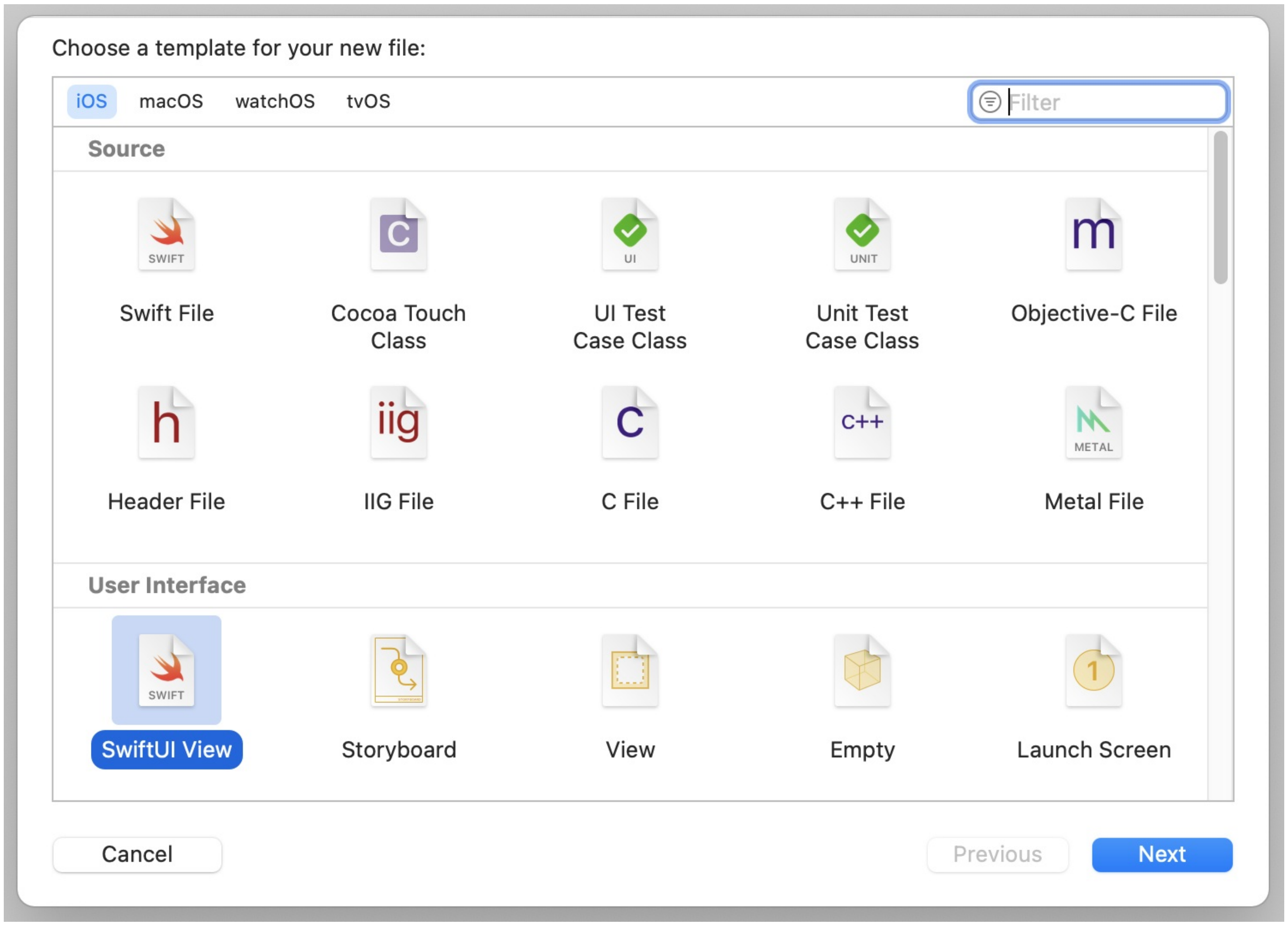
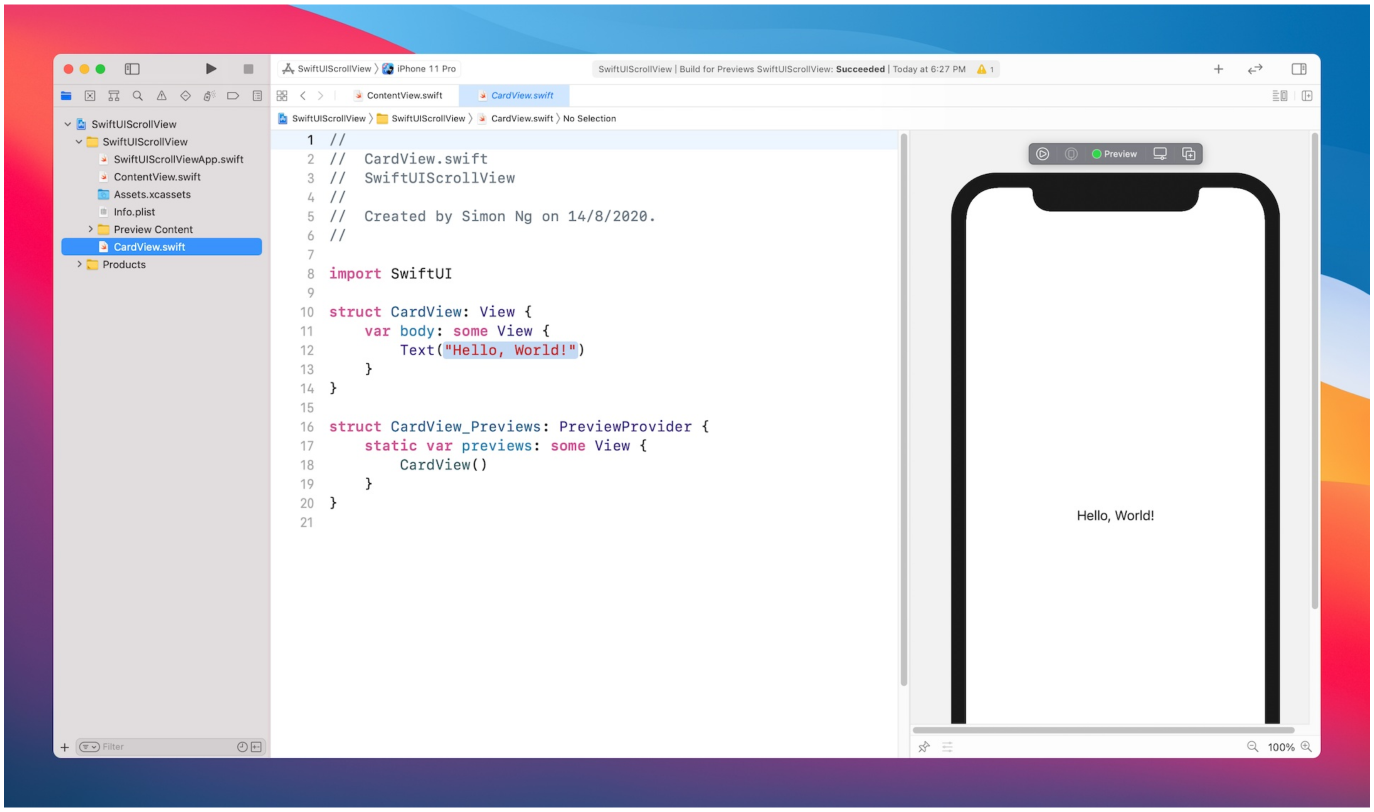


Figure 4. Choose the SwiftUI View template

The code in `CardView.swift` looks very similar to that of `ContentView.swift`. Similarly, you can preview the UI in the canvas.



```swift
//
//  CardView.swift
//  SwiftUIScrollView
//
//  Created by Simon Ng on 14/8/2020.
//

import SwiftUI

struct CardView: View {
    var body: some View {
        Text("Hello, World!")
    }
}

struct CardView_Previews: PreviewProvider {
    static var previews: some View {
        CardView()
    }
}
```

Figure 5. Just like ContentView.swift, you can preview CardView.swift in the canvas

Preparing the Image Files

Now we're ready to code the card view. But first, you need to prepare the image files and import them in the asset catalog. If you don't want to prepare your own images, you can download the sample images from https://www.appcoda.com/resources/swiftui/SwiftUIScrollViewImages.zip. Once you unzip the image archive, select `Assets.xcassets` and drag all the images to the asset catalog.

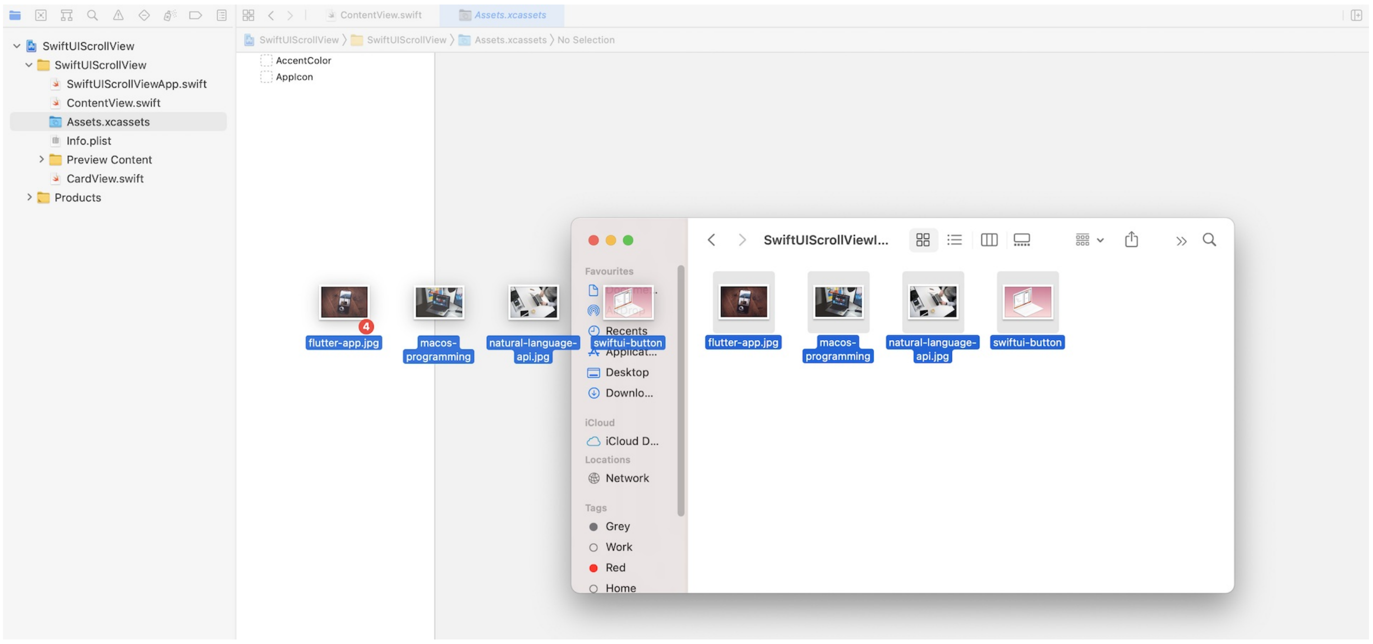


Figure 6. Adding the image files to the asset catalog

Implementing the Card View

Now switch back to the `CardView.swift` file. If you look at figure 1 again, the card view is composed of two parts: the upper part is the image and the lower part is the text description.

Let's start with the image. I'll make the image resizable and scale it to fit the screen while retaining the aspect ratio. You write the code like this:

```
struct CardView: View {
    var body: some View {
        Image("swiftui-button")
            .resizable()
            .aspectRatio(contentMode: .fit)
    }
}
```

If you forgot what these two modifiers do, go back and read the chapter about the `Image` object. Next, let's implement the text description. You may write the code like this:

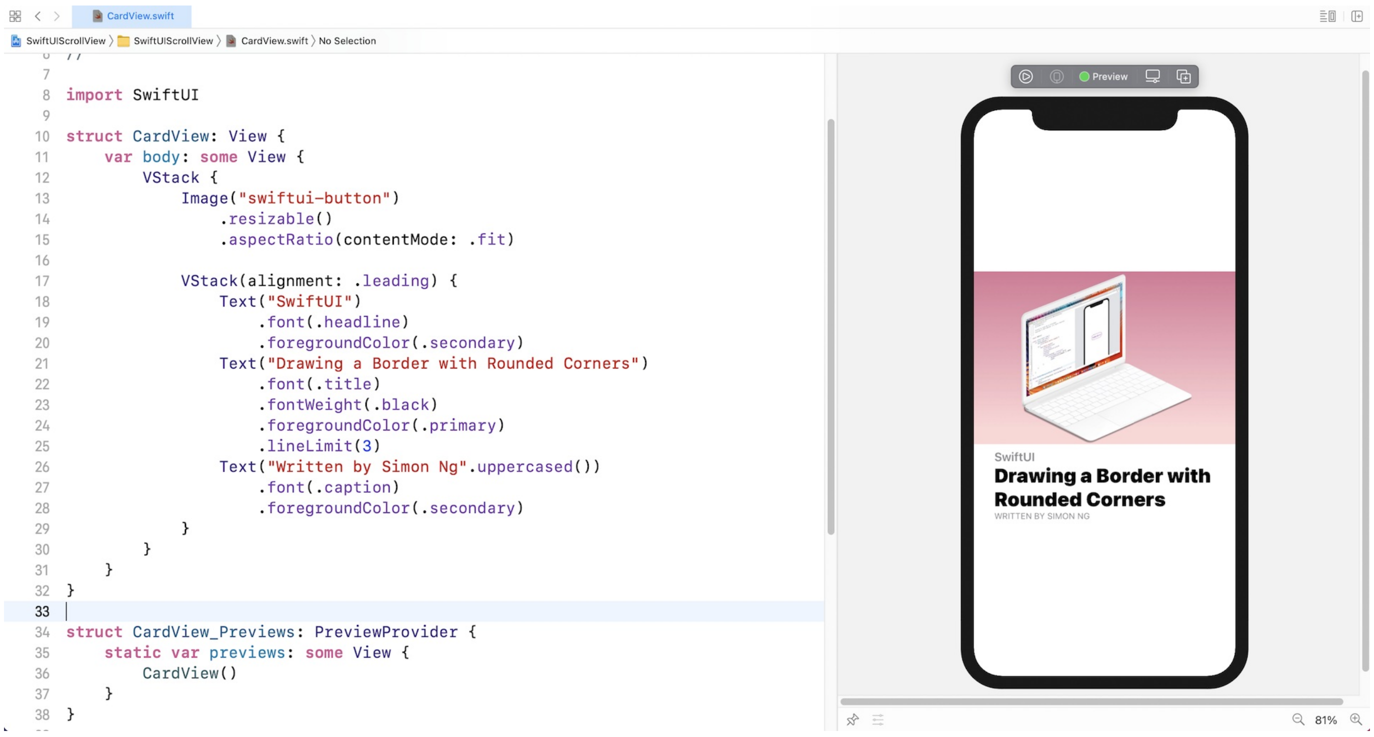
```
VStack(alignment: .leading) {
    Text("SwiftUI")
        .font(.headline)
        .foregroundColor(.secondary)
    Text("Drawing a Border with Rounded Corners")
        .font(.title)
        .fontWeight(.black)
        .foregroundColor(.primary)
        .lineLimit(3)
    Text("Written by Simon Ng".uppercased())
        .font(.caption)
        .foregroundColor(.secondary)
}
```

You need to use `Text` to create the text view. Since we actually have three text views in the description, that are vertically aligned, we use a `VStack` to embed them. For the `VStack`, we specify the alignment as `.leading`. This will align the text view to the left of the stack view.

The modifiers of `Text` are all discussed in the chapter about the `Text` object. You can refer to it if you find any of the modifiers are confusing. But one topic about the `.primary` and `.secondary` colors should be highlighted.

While you can specify a standard color like `.black` and `.purple` in the `foregroundColor` modifier, iOS 13 introduces a set of system colors that contain primary, secondary, and tertiary variants. By using these color variants, your app can easily support both light and dark modes. For example, the primary color of the text view is set to black in light mode by default. When the app is switched over to dark mode, the primary color will be adjusted to white. This is automatically arranged by iOS, so you don't have to write extra code to support the dark mode. We will discuss dark mode in depth in a later chapter.

To arrange the image and these text views vertically, we use a `VStack` to embed them. The current layout is shown in the figure below.


CardView.swift
SwiftUIScrollView 〉 SwiftUIScrollView 〉 CardView.swift 〉 No Selection
Preview
SwiftUI
Drawing a Border with Rounded Corners
WRITTEN BY SIMON NG
81%

```swift
import SwiftUI

struct CardView: View {
    var body: some View {
        VStack {
            Image("swiftui-button")
                .resizable()
                .aspectRatio(contentMode: .fit)

            VStack(alignment: .leading) {
                Text("SwiftUI")
                    .font(.headline)
                    .foregroundColor(.secondary)
                Text("Drawing a Border with Rounded Corners")
                    .font(.title)
                    .fontWeight(.black)
                    .foregroundColor(.primary)
                    .lineLimit(3)
                Text("Written by Simon Ng".uppercased())
                    .font(.caption)
                    .foregroundColor(.secondary)
            }
        }
    }
}

struct CardView_Previews: PreviewProvider {
    static var previews: some View {
        CardView()
    }
}
```

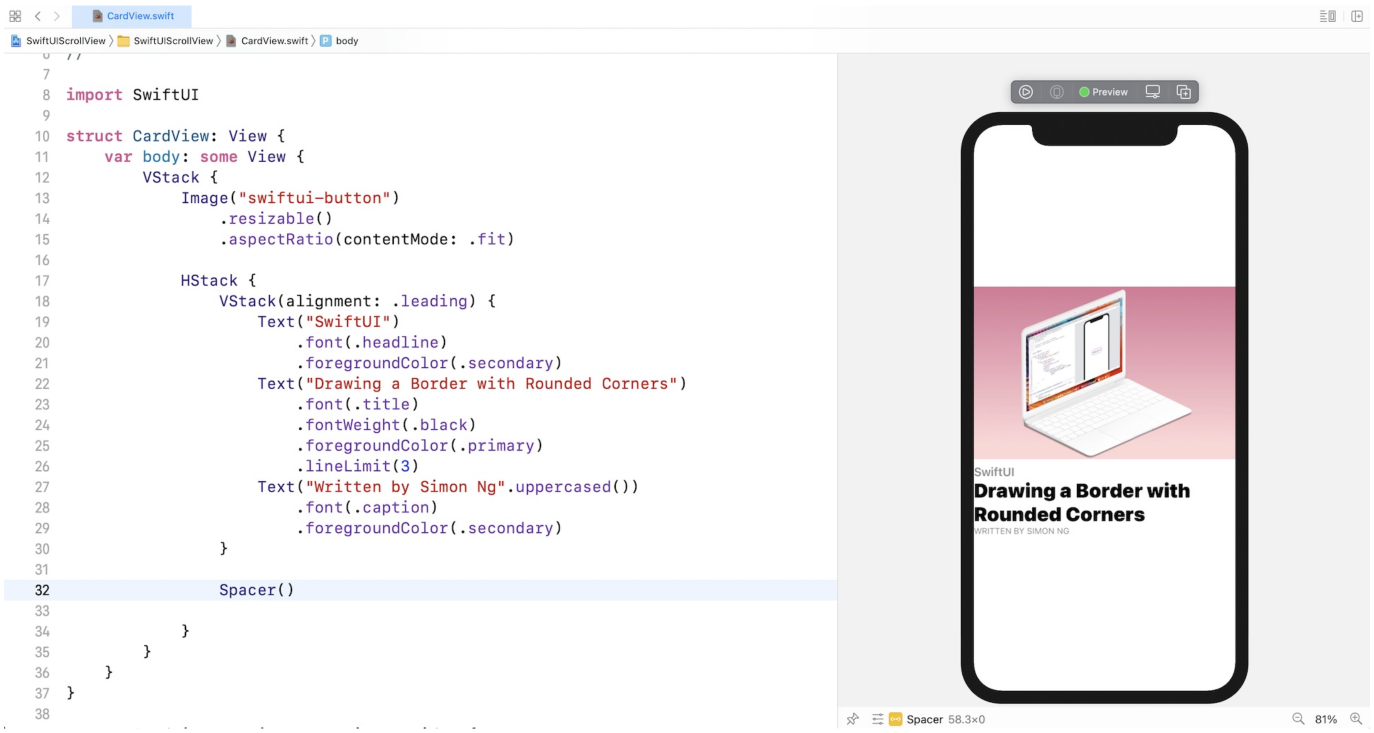
Figure 7. Embed the image and text views in a VStack

We are not done yet! There are still a couple of things we need to implement. First, the text description block should be left aligned to the edge of the image.

How do you do that?

Based on what we've learned, we can embed the `VStack` of the text views in a `HStack`. And then, we will use a `Spacer` to push the `VStack` to the left. Let's see if this works.

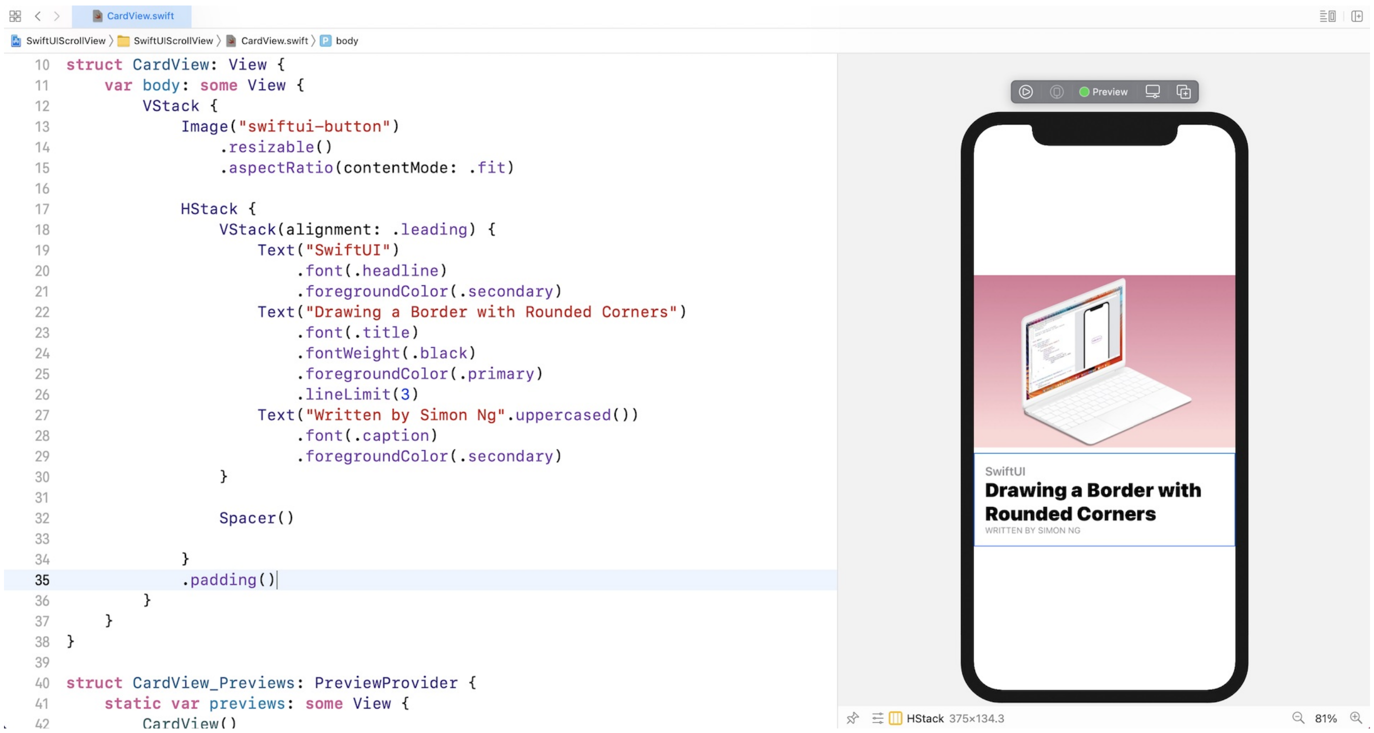
If you've changed the code to match the one shown in figure 8, the `VStack` of the text views are aligned to the left of the screen.



```swift
import SwiftUI

struct CardView: View {
    var body: some View {
        VStack {
            Image("swiftui-button")
                .resizable()
                .aspectRatio(contentMode: .fit)

            HStack {
                VStack(alignment: .leading) {
                    Text("SwiftUI")
                        .font(.headline)
                        .foregroundColor(.secondary)
                    Text("Drawing a Border with Rounded Corners")
                        .font(.title)
                        .fontWeight(.black)
                        .foregroundColor(.primary)
                        .lineLimit(3)
                    Text("Written by Simon Ng".uppercased())
                        .font(.caption)
                        .foregroundColor(.secondary)
                }

                Spacer()

            }
        }
    }
}
```

Figure 8. Aligning the text description

It would be better to add some padding around the `HStack`. Insert the `padding` modifier like this:



CardView.swift
SwiftUIScrollView 〉 SwiftUIScrollView 〉 CardView.swift 〉 body
Preview
HStack 375×134.3
81%

```
struct CardView: View {
    var body: some View {
        VStack {
            Image("swiftui-button")
                .resizable()
                .aspectRatio(contentMode: .fit)

            HStack {
                VStack(alignment: .leading) {
                    Text("SwiftUI")
                        .font(.headline)
                        .foregroundColor(.secondary)
                    Text("Drawing a Border with Rounded Corners")
                        .font(.title)
                        .fontWeight(.black)
                        .foregroundColor(.primary)
                        .lineLimit(3)
                    Text("Written by Simon Ng".uppercased())
                        .font(.caption)
                        .foregroundColor(.secondary)
                }

                Spacer()

            }
            .padding()
        }
    }
}

struct CardView_Previews: PreviewProvider {
    static var previews: some View {
        CardView()
```

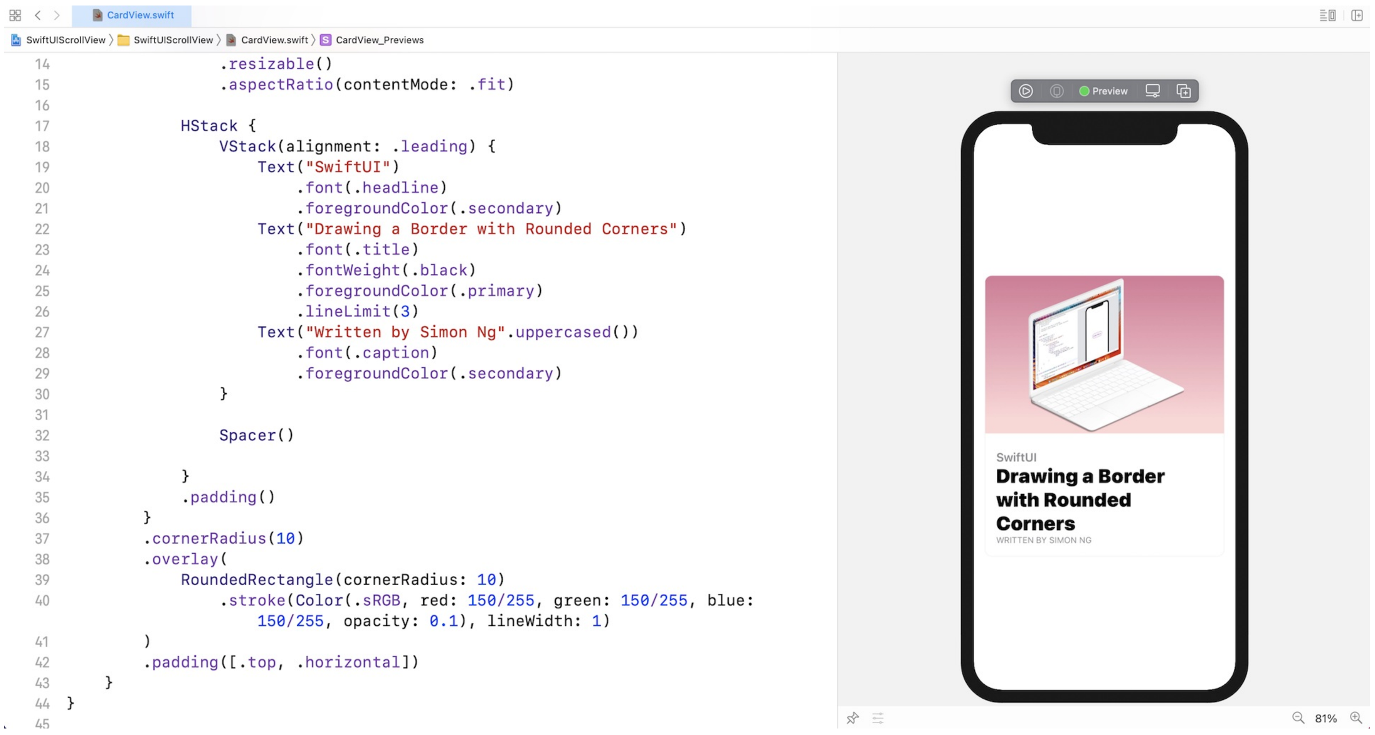
SwiftUI
Drawing a Border with Rounded Corners
WRITTEN BY SIMON NG

Figure 9. Adding some paddings for the text description

Lastly, the border. We have discussed how to draw a border with rounded corners in an earlier chapter. We use the `overlay` modifier and draw the border using `RoundedRectangle`. Here is the complete code:

```
struct CardView: View {
    var body: some View {
        VStack {
            Image("swiftui-button")
                .resizable()
                .aspectRatio(contentMode: .fit)

            HStack {
                VStack(alignment: .leading) {
                    Text("SwiftUI")
                        .font(.headline)
                        .foregroundColor(.secondary)
                    Text("Drawing a Border with Rounded Corners")
                        .font(.title)
                        .fontWeight(.black)
                        .foregroundColor(.primary)
                        .lineLimit(3)
                    Text("Written by Simon Ng".uppercased())
                        .font(.caption)
                        .foregroundColor(.secondary)
                }

                Spacer()

            }
            .padding()
        }
        .cornerRadius(10)
        .overlay(
            RoundedRectangle(cornerRadius: 10)
                .stroke(Color(.sRGB, red: 150/255, green: 150/255, blue: 150/255,
opacity: 0.1), lineWidth: 1)
        )
        .padding([.top, .horizontal])
    }
}
```

In addition to the border, we also add some padding for the top, left, and right sides. Now you have completed the card view layout.


CardView.swift
SwiftUIScrollView 〉 SwiftUIScrollView 〉 CardView.swift 〉 CardView_Previews
Preview
SwiftUI
Drawing a Border with Rounded Corners
WRITTEN BY SIMON NG
81%

```
                .resizable()
                .aspectRatio(contentMode: .fit)

            HStack {
                VStack(alignment: .leading) {
                    Text("SwiftUI")
                        .font(.headline)
                        .foregroundColor(.secondary)
                    Text("Drawing a Border with Rounded Corners")
                        .font(.title)
                        .fontWeight(.black)
                        .foregroundColor(.primary)
                        .lineLimit(3)
                    Text("Written by Simon Ng".uppercased())
                        .font(.caption)
                        .foregroundColor(.secondary)
                }

                Spacer()

            }
            .padding()
        }
        .cornerRadius(10)
        .overlay(
            RoundedRectangle(cornerRadius: 10)
                .stroke(Color(.sRGB, red: 150/255, green: 150/255, blue:
                    150/255, opacity: 0.1), lineWidth: 1)
        )
        .padding([.top, .horizontal])
    }
}
```

Figure 10. Adding a border and rounded corners

Make the Card View more Flexible

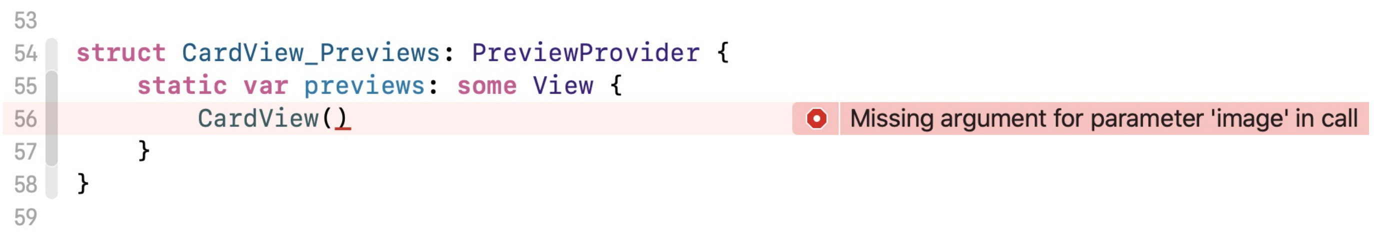
While the card view works, we've hard-coded the image and text. To make it more flexible, let's refactor the code. First, declare these variables for the image, category, heading, and author in `CardView` :

```
var image: String
var category: String
var heading: String
var author: String
```

Next, replace the values of the `Image` and `Text` views with our variables like this:

```
VStack {
    Image(image)
        .resizable()
        .aspectRatio(contentMode: .fit)

    HStack {
        VStack(alignment: .leading) {
            Text(category)
                .font(.headline)
                .foregroundColor(.secondary)
            Text(heading)
                .font(.title)
                .fontWeight(.black)
                .foregroundColor(.primary)
                .lineLimit(3)
            Text("Written by \(author)".uppercased())
                .font(.caption)
                .foregroundColor(.secondary)
        }

        Spacer()
    }
    .padding()
}
```

Once you made the changes, you will see an error in the `CardView_Previews` struct. This is because we've introduced some variables in `CardView`. We have to specify the parameters when using it.



```
struct CardView_Previews: PreviewProvider {
    static var previews: some View {
        CardView()                          Missing argument for parameter 'image' in call
    }
}
```

Figure 11. Missing parameters when calling the CardView

Modify the code like this:

```
struct CardView_Previews: PreviewProvider {
    static var previews: some View {
        CardView(image: "swiftui-button", category: "SwiftUI", heading: "Drawing a
 Border with Rounded Corners", author: "Simon Ng")
    }
}
```

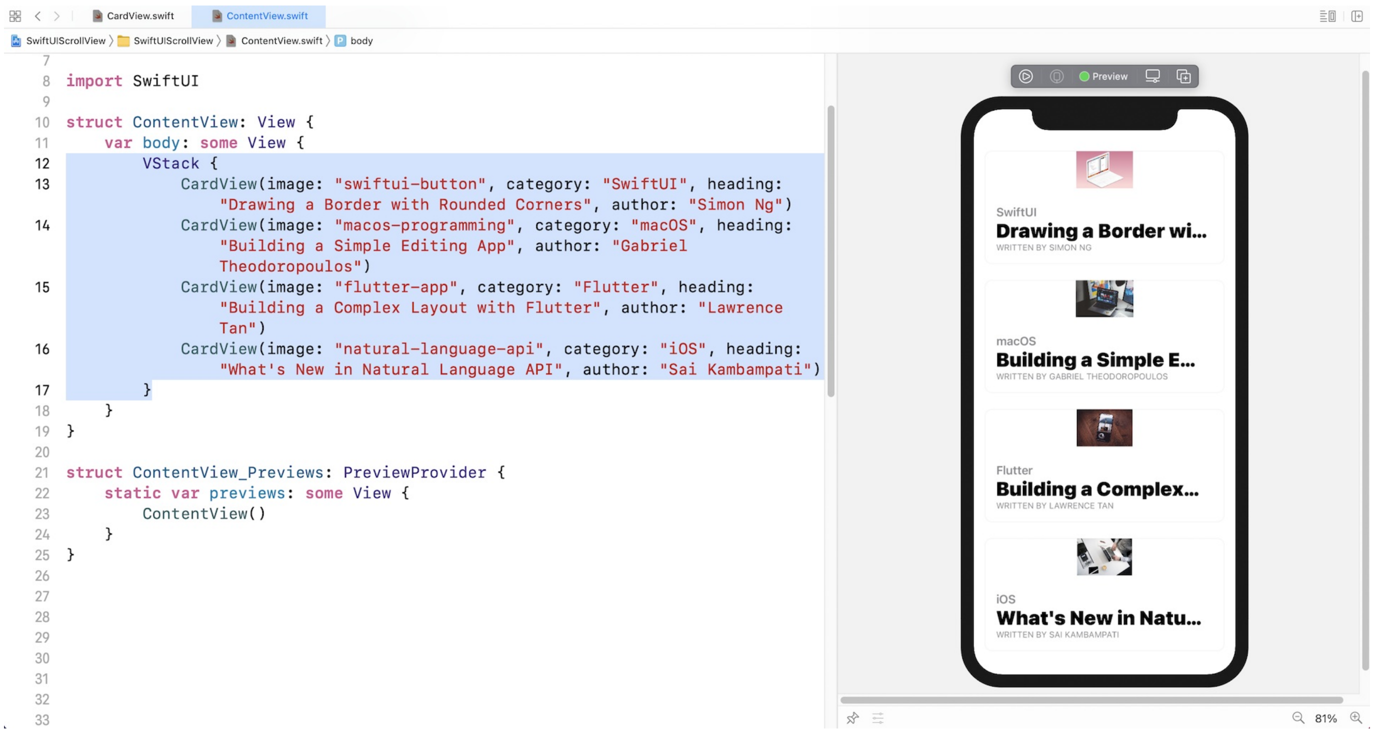
This will fix the error. And, you now have built a flexible `CardView` that accepts different images and text.

Introducing ScrollView

Take a look at figure 2 again. That's the user interface we're going to implement. At first, you may think we can embed four card views using a `VStack`. You can switch over to `ContentView.swift` and insert the following code:

```
VStack {
    CardView(image: "swiftui-button", category: "SwiftUI", heading: "Drawing a Bor
der with Rounded Corners", author: "Simon Ng")
    CardView(image: "macos-programming", category: "macOS", heading: "Building a S
imple Editing App", author: "Gabriel Theodoropoulos")
    CardView(image: "flutter-app", category: "Flutter", heading: "Building a Compl
ex Layout with Flutter", author: "Lawrence Tan")
    CardView(image: "natural-language-api", category: "iOS", heading: "What's New
in Natural Language API", author: "Sai Kambampati")
}
```

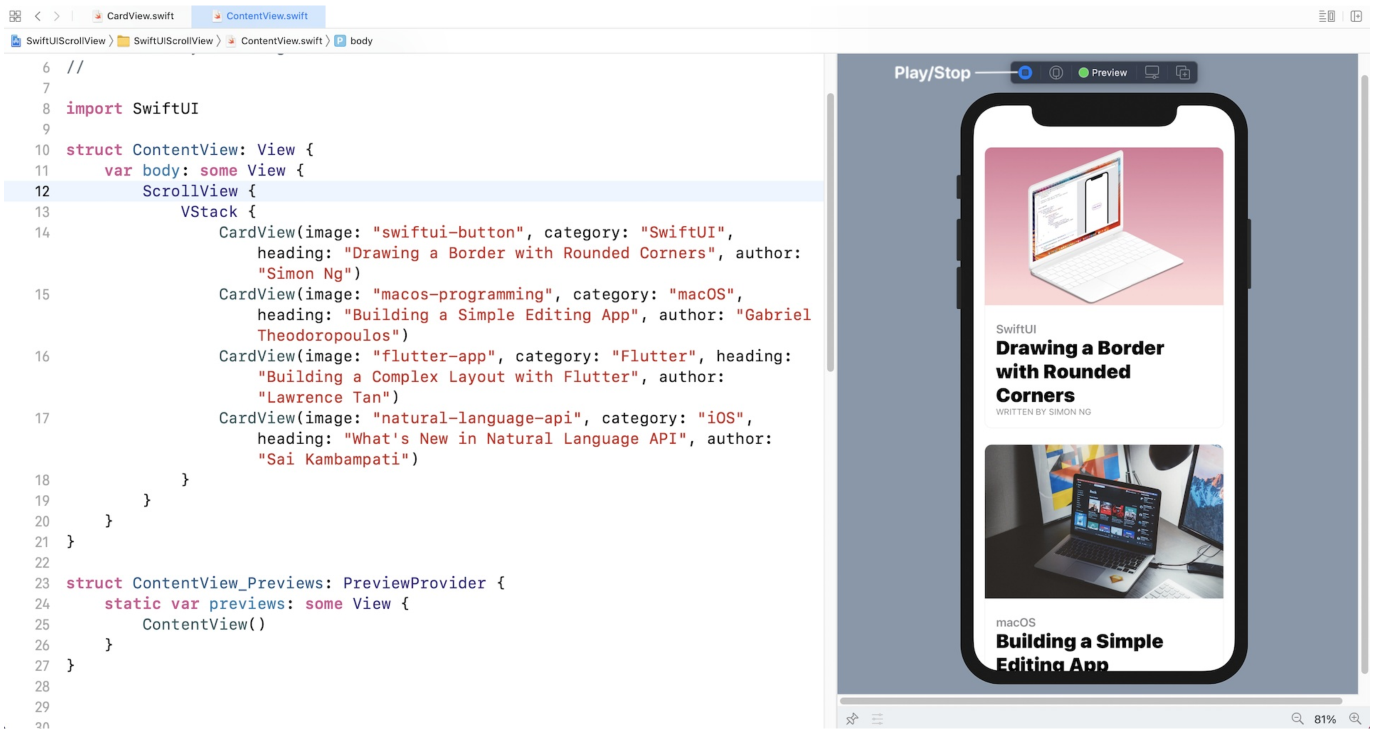
If you did that, the card views will be squeezed to fit the screen because `VStack` is non-scrollable, just like that shown in figure 12.



```
import SwiftUI

struct ContentView: View {
    var body: some View {
        VStack {
            CardView(image: "swiftui-button", category: "SwiftUI", heading:
                "Drawing a Border with Rounded Corners", author: "Simon Ng")
            CardView(image: "macos-programming", category: "macOS", heading:
                "Building a Simple Editing App", author: "Gabriel
                Theodoropoulos")
            CardView(image: "flutter-app", category: "Flutter", heading:
                "Building a Complex Layout with Flutter", author: "Lawrence
                Tan")
            CardView(image: "natural-language-api", category: "iOS", heading:
                "What's New in Natural Language API", author: "Sai Kambampati")
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 12. Embedding the card views in a VStack

To support scrollable content, SwiftUI provides a view called `ScrollView`. When the content is embedded in a `ScrollView`, it becomes scrollable. What you need to do is to enclose the `VStack` within a `ScrollView` to make the views scrollable. In the preview canvas, you can click the *Play* button and drag the views to scroll the content.



CardView.swift
ContentView.swift
SwiftUIScrollView 〉 SwiftUIScrollView 〉 ContentView.swift 〉 body
Play/Stop
Preview
SwiftUI
Drawing a Border with Rounded Corners
WRITTEN BY SIMON NG
macOS
Building a Simple Editing App
81%

```
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        ScrollView {
            VStack {
                CardView(image: "swiftui-button", category: "SwiftUI",
                    heading: "Drawing a Border with Rounded Corners", author:
                    "Simon Ng")
                CardView(image: "macos-programming", category: "macOS",
                    heading: "Building a Simple Editing App", author: "Gabriel
                    Theodoropoulos")
                CardView(image: "flutter-app", category: "Flutter", heading:
                    "Building a Complex Layout with Flutter", author:
                    "Lawrence Tan")
                CardView(image: "natural-language-api", category: "iOS",
                    heading: "What's New in Natural Language API", author:
                    "Sai Kambampati")
            }
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 13. Using ScrollView

Exercise #1

Your task is to add a header to the existing scroll view. The result is displayed in figure 14. If you understand `VStack` and `HStack` thoroughly, you should be able to create the layout.

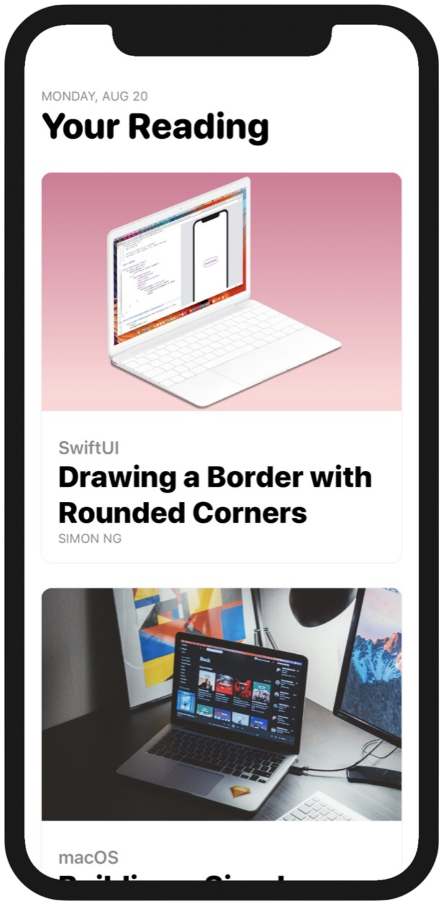


Figure 14. Exercise #1

Creating a Carousel UI with Horizontal ScrollView

By default, the `ScrollView` allows you to scroll the content in vertical orientation. Alternatively, it also supports scrollable content in horizontal orientation. Let's see how to convert the current layout into a carousel UI with a few changes.

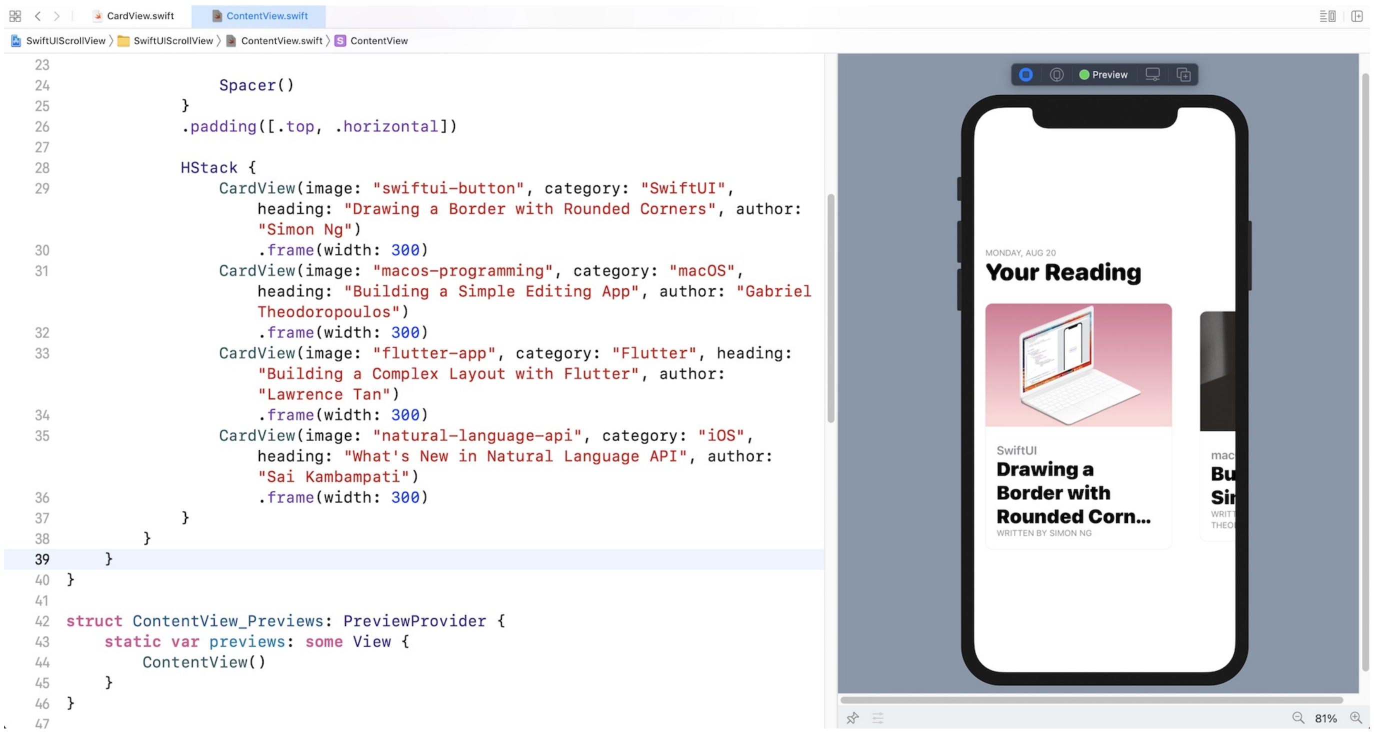
Update the `ContentView` like this:

```
struct ContentView: View {
    var body: some View {

        ScrollView(.horizontal) {

            // Your code for exercise #1

            HStack {
                CardView(image: "swiftui-button", category: "SwiftUI", heading: "D
rawing a Border with Rounded Corners", author: "Simon Ng")
                    .frame(width: 300)
                CardView(image: "macos-programming", category: "macOS", heading: "
Building a Simple Editing App", author: "Gabriel Theodoropoulos")
                    .frame(width: 300)
                CardView(image: "flutter-app", category: "Flutter", heading: "Buil
ding a Complex Layout with Flutter", author: "Lawrence Tan")
                    .frame(width: 300)
                CardView(image: "natural-language-api", category: "iOS", heading:
"What's New in Natural Language API", author: "Sai Kambampati")
                    .frame(width: 300)
            }
        }

    }
}
```

We've made three changes in the code above:

1. We specify in `ScrollView` to use a horizontal scroll view by passing it a `.horizontal` value.
2. Since we use a horizontal scroll view, we also need to change the stack view from `VStack` to `HStack`.
3. For each card view, we set the frame's width to 300 points. This is required because the image is too wide to display.

After changing the code, you'll see the card views are arranged horizontally and they are scrollable.


CardView.swift
ContentView.swift
SwiftUIScrollView 〉 SwiftUIScrollView 〉 ContentView.swift 〉 ContentView
Preview
MONDAY, AUG 20
Your Reading
SwiftUI
Drawing a Border with Rounded Corn...
WRITTEN BY SIMON NG
81%

```swift

                Spacer()
            }
            .padding([.top, .horizontal])

            HStack {
                CardView(image: "swiftui-button", category: "SwiftUI",
                    heading: "Drawing a Border with Rounded Corners", author:
                    "Simon Ng")
                    .frame(width: 300)
                CardView(image: "macos-programming", category: "macOS",
                    heading: "Building a Simple Editing App", author: "Gabriel
                    Theodoropoulos")
                    .frame(width: 300)
                CardView(image: "flutter-app", category: "Flutter", heading:
                    "Building a Complex Layout with Flutter", author:
                    "Lawrence Tan")
                    .frame(width: 300)
                CardView(image: "natural-language-api", category: "iOS",
                    heading: "What's New in Natural Language API", author:
                    "Sai Kambampati")
                    .frame(width: 300)
            }
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}

```

Figure 15. Carousel UI

Hiding the Scroll Indicator

While you're scrolling the views, did you notice there is a scroll indicator near the bottom of the screen? This indicator is displayed by default. If you want to hide it, you can change the `ScrollView` by adding `showsIndicators: false` to it:

```swift
ScrollView(.horizontal, showsIndicators: false)
```

By setting `showIndicators` to `false`, iOS will no longer show the indicator.

Grouping View Content

If you look at the code again, you will see that all the `CardView`s have a `.frame` modifier to limit their width to 300 points. Is there any way we can simplify that and remove the duplicated code? The SwiftUI framework provides a `Group` view for developers to group related content. More importantly, you can attach modifiers to the group to apply the changes to each of the views embedded in the group.

For example, you can rewrite the code in `HStack` like this to achieve the same result:

```
HStack {
    Group {
        CardView(image: "swiftui-button", category: "SwiftUI", heading: "Drawing a
 Border with Rounded Corners", author: "Simon Ng")
        CardView(image: "macos-programming", category: "macOS", heading: "Building
 a Simple Editing App", author: "Gabriel Theodoropoulos")
        CardView(image: "flutter-app", category: "Flutter", heading: "Building a C
omplex Layout with Flutter", author: "Lawrence Tan")
        CardView(image: "natural-language-api", category: "iOS", heading: "What's
New in Natural Language API", author: "Sai Kambampati")
    }
    .frame(width: 300)
}
```

Resize the Text Automatically

As you can see in figure 15, the title of the first card is truncated. How do you fix this? In SwiftUI, you can use the `.minimumScaleFactor` modifier to automatically downscale text. Switch over to `CardView.swift` and attach the following modifier to `Text(heading)`:

```
.minimumScaleFactor(0.5)
```

SwiftUI will automatically scale down the text to fit the available space. The value sets the minimum amount of scaling that the view permits. In this case, SwiftUI can draw the text in a font size as small as 50% of the original font size.

Exercise #2

Here comes to the final exercise. Modify the current code and re-arrange it like that shown in figure 16.

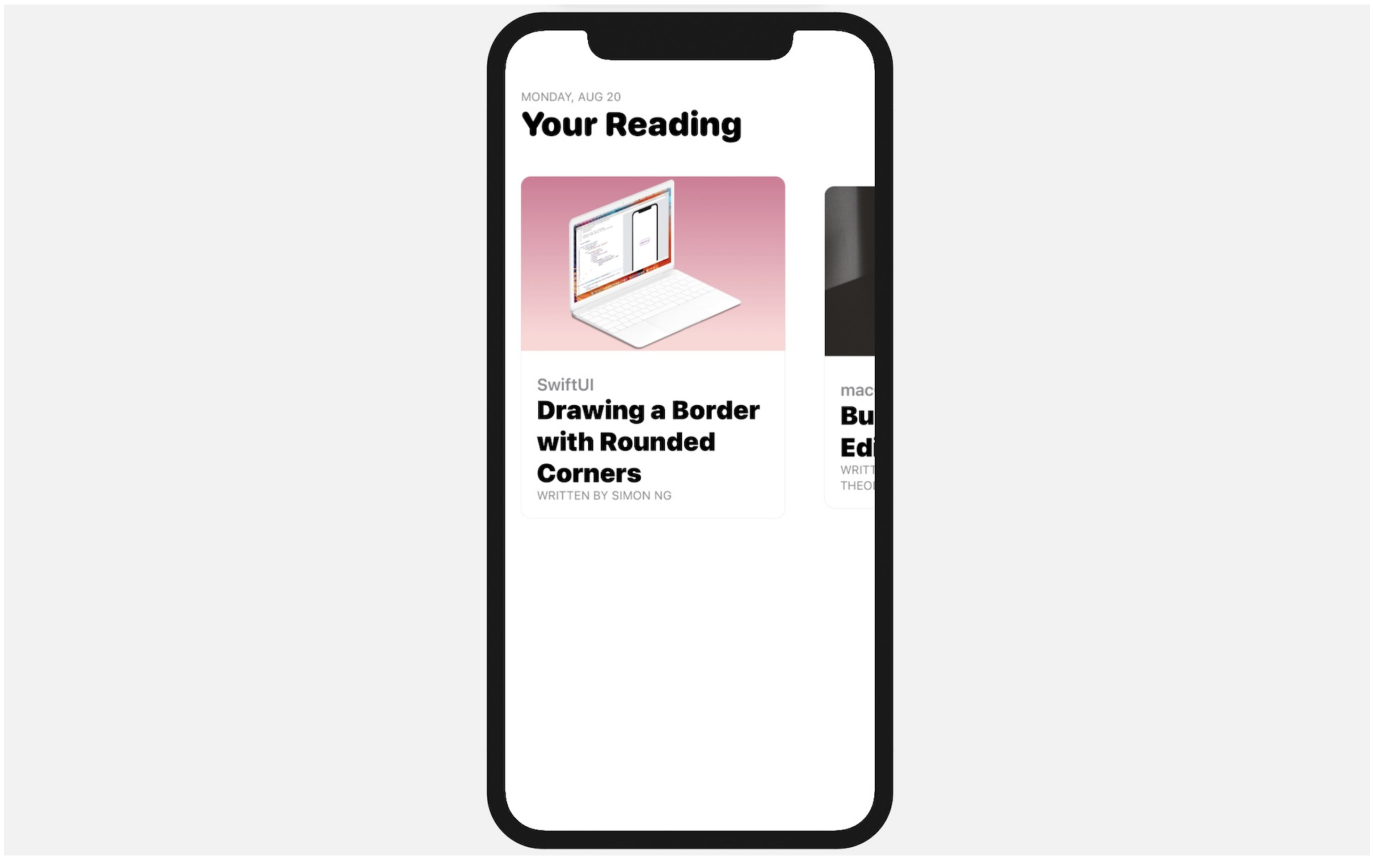


Figure 16. Aligning the views to the top

For reference, you can download the complete scrollview project here:

- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUIScrollView.zip)
- Solution to exercise (https://www.appcoda.com/resources/swiftui2/SwiftUIScrollViewExercise.zip)

Chapter 6
Working with SwiftUI Buttons, Labels and Gradient

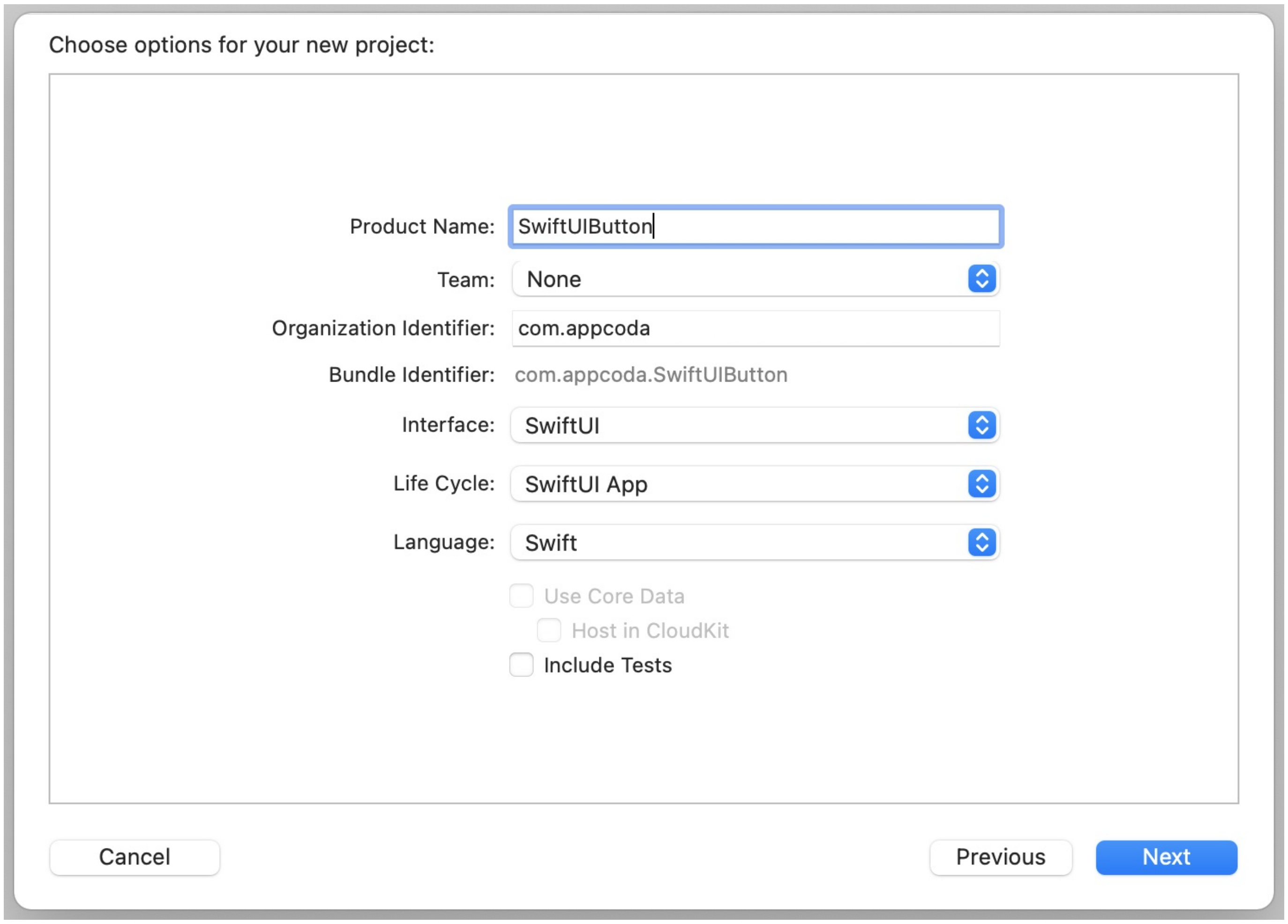
> Buttons initiate app-specific actions, have customizable backgrounds, and can include a title or an icon. The system provides a number of predefined button styles for most use cases. You can also design fully custom buttons.
>
> - Apple's documentation (https://developer.apple.com/design/human-interface-guidelines/ios/controls/buttons/)

I don't think I need to explain what a button is. It's a very basic UI control that you can find in all apps and has the ability to handle users' touch, and trigger a certain action. If you have learned iOS programming before, `Button` in SwiftUI is very similar to `UIButton` in UIKit. It's just more flexible and customizable. You will understand what I mean in a while. In this chapter, I will go through this SwiftUI control and you will learn the following techniques:

1. How to create a simple button and handle the user's selection
2. How to customize the button's background, padding and font
3. How to add borders to a button
4. How to create a button with both image and text
5. How to create a button with a gradient background and shadows
6. How to create a full-width button
7. How to create a reusable button style
8. How to add a tap animation

Creating a New Project with SwiftUI enabled

Okay, let's start with the basics and create a simple button using SwiftUI. First, fire up Xcode and create a new project using the *App* template. In the next screen, type the name of the project. I set it to *SwiftUIButton* but you're free to use any other name. You need to make sure you select *SwiftUI* for the *Interface* option.

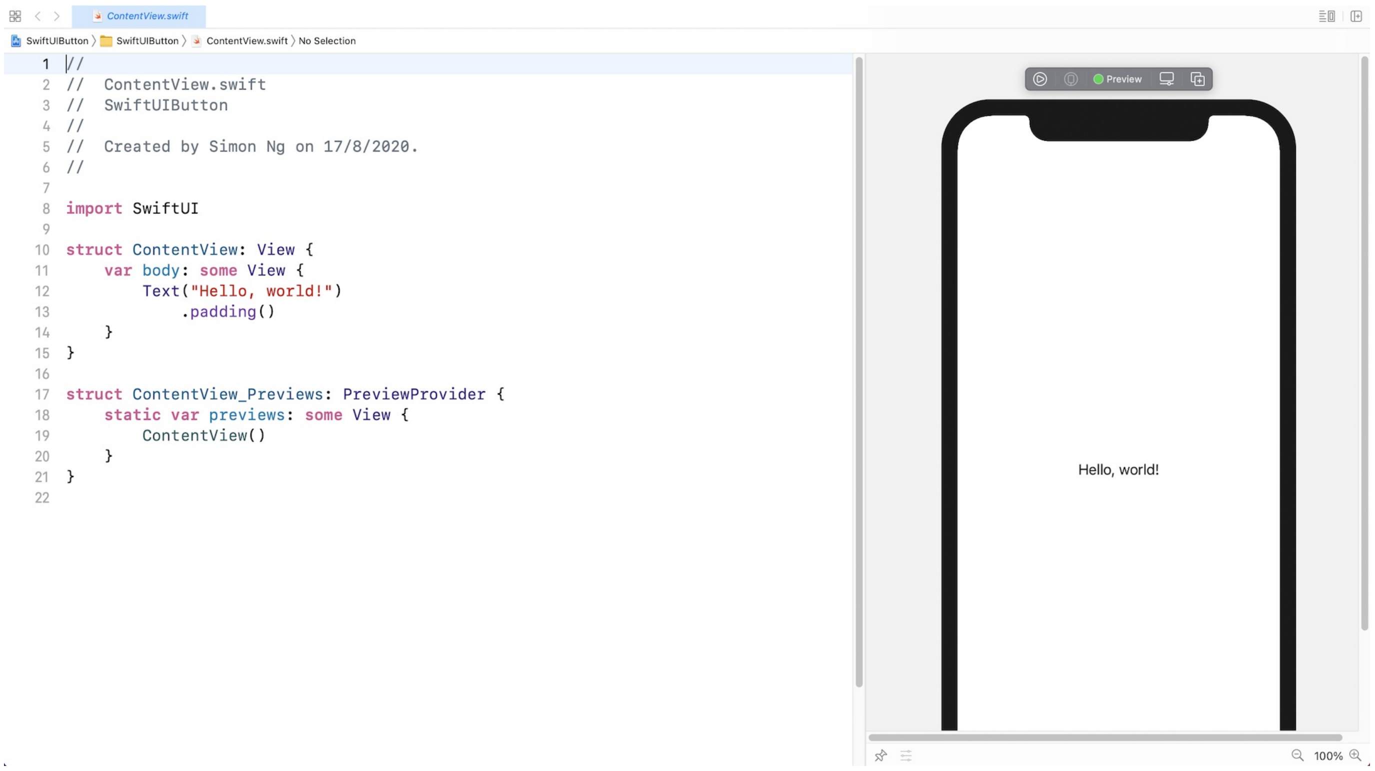

Choose options for your new project:

Product Name:	SwiftUIButton
Team:	None
Organization Identifier:	com.appcoda
Bundle Identifier:	com.appcoda.SwiftUIButton
Interface:	SwiftUI
Life Cycle:	SwiftUI App
Language:	Swift

- [ ] Use Core Data
  - [ ] Host in CloudKit
- [ ] Include Tests

Figure 1. Creating a new project

Once you save the project, Xcode should load the `ContentView.swift` file and display a preview in the design canvas. In case the preview is not displayed, click the *Resume* button in the canvas.


ContentView.swift
SwiftUIButton 〉 SwiftUIButton 〉 ContentView.swift 〉 No Selection
Preview
Hello, world!
100%

```
//
//  ContentView.swift
//  SwiftUIButton
//
//  Created by Simon Ng on 17/8/2020.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, world!")
            .padding()
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 2. Previewing the default content view

It's very easy to create a button using SwiftUI. Basically, you use the code snippet below to create a button:

```
Button(action: {
    // What to perform
}) {
    // How the button looks like
}
```

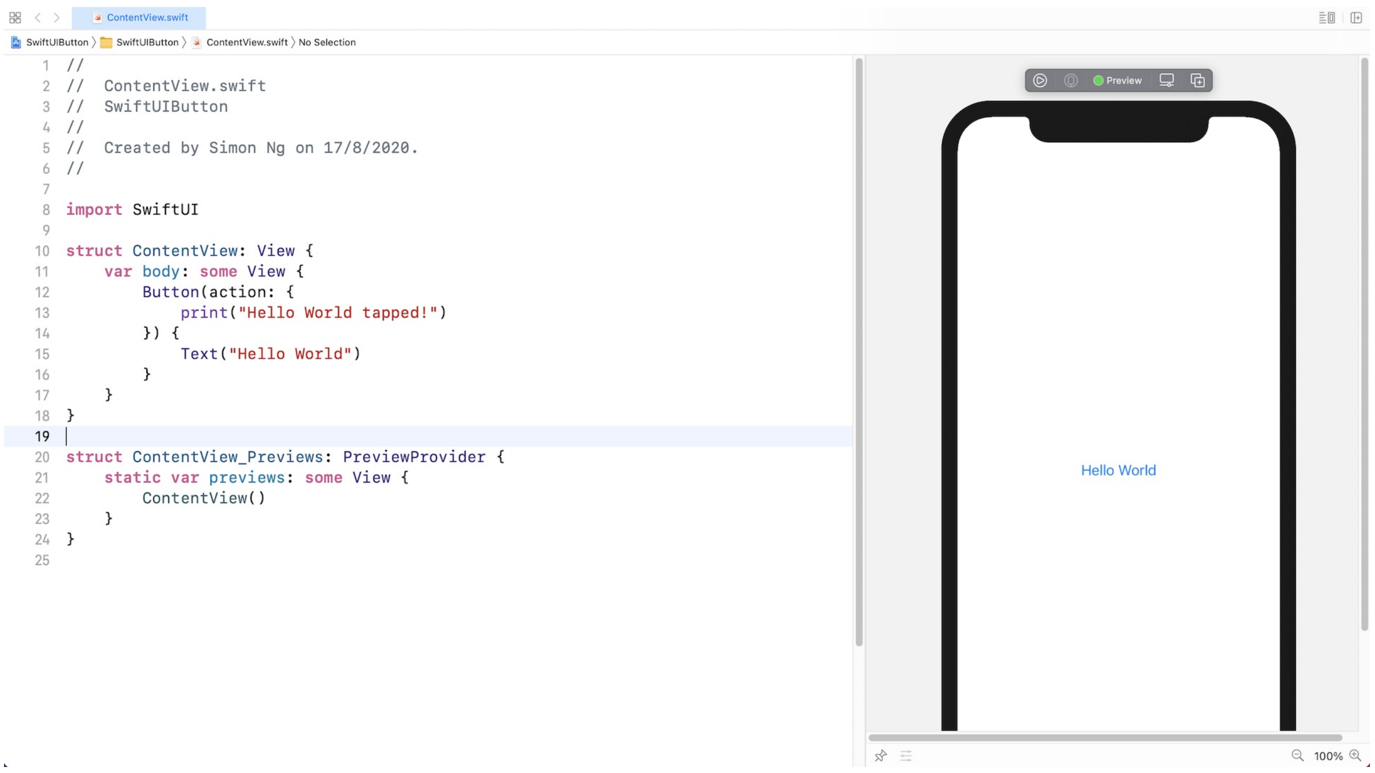
When creating a button, you need to provide two code blocks:

1. **What action to perform** - the code to perform after the button is tapped or selected by the user.
2. **How the button looks** - the code block that describes the look & feel of the button.

For example, if you just want to turn the *Hello World* label into a button, you can update the code like this:

```
struct ContentView: View {
    var body: some View {
        Button(action: {
            print("Hello World tapped!")
        }) {
            Text("Hello World")
        }
    }
}
```

Now the *Hello World* text becomes a tappable button as you see it in the canvas.



```
//
//  ContentView.swift
//  SwiftUIButton
//
//  Created by Simon Ng on 17/8/2020.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        Button(action: {
            print("Hello World tapped!")
        }) {
            Text("Hello World")
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

ContentView.swift
SwiftUIButton 〉 SwiftUIButton 〉 ContentView.swift 〉 No Selection
Preview
Hello World
100%

Figure 3. Creating a simple button

The button is non-tappable in the design canvas. To test it, click the *Play* button to run the app. However, in order to view the *Hello World tapped* message, you have to right-click the *Play* button and then choose *Debug Preview*. You will see the *Hello World* *tapped* message on the console when you tap the button. If you can't see the console, go up to the Xcode menu and choose *View > Debug Area > Activate Console*.

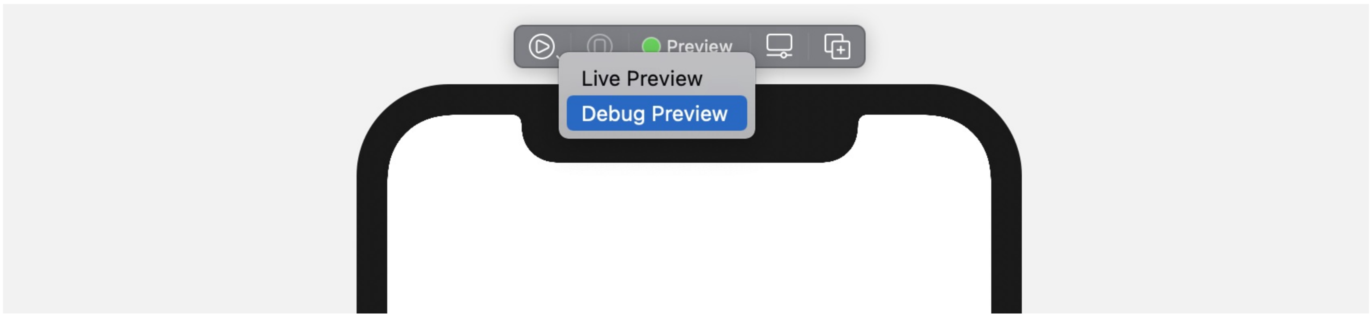


Figure 4. The console message can only be displayed in debug mode

Customizing the Button's Font and Background

Now that you know how to create a simple button, let's customize its look & feel with the built-in modifiers. To change the background and text color, you can use the `background` and `foregroundColor` modifiers like this:

```
Text("Hello World")
    .background(Color.purple)
    .foregroundColor(.white)
```

If you want to change the font type, you use the `font` modifier and specify the font type (e.g. `.title`) like this:

```
Text("Hello World")
    .background(Color.purple)
    .foregroundColor(.white)
    .font(.title)
```

After the change, your button should look like the figure below.

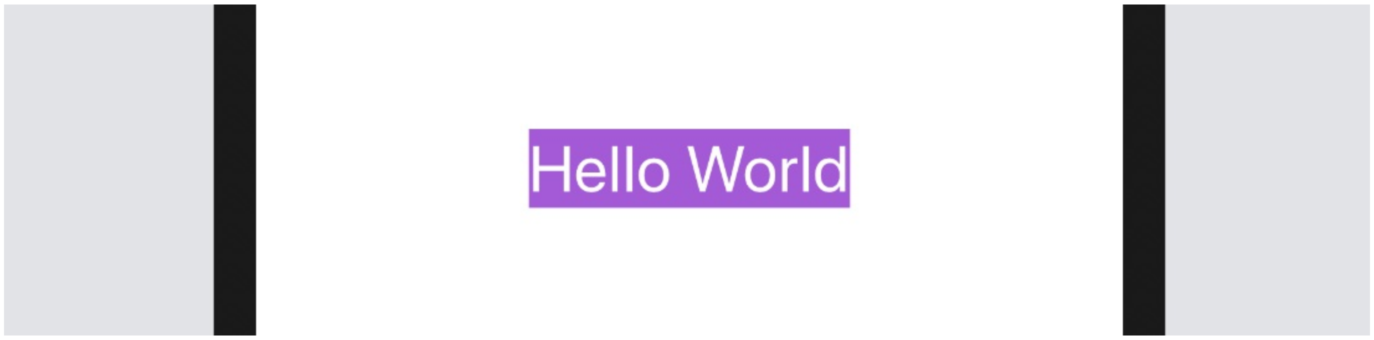


Figure 5. Customizing the background and foreground color of a button

As you can see, the button doesn't look very good. Wouldn't it be great to add some space around the text? To do that, you can use the `padding` modifier like this:

```
Text("Hello World")
    .padding()
    .background(Color.purple)
    .foregroundColor(.white)
    .font(.title)
```

After you make the change, the canvas will update the button accordingly. The button should now look much better.

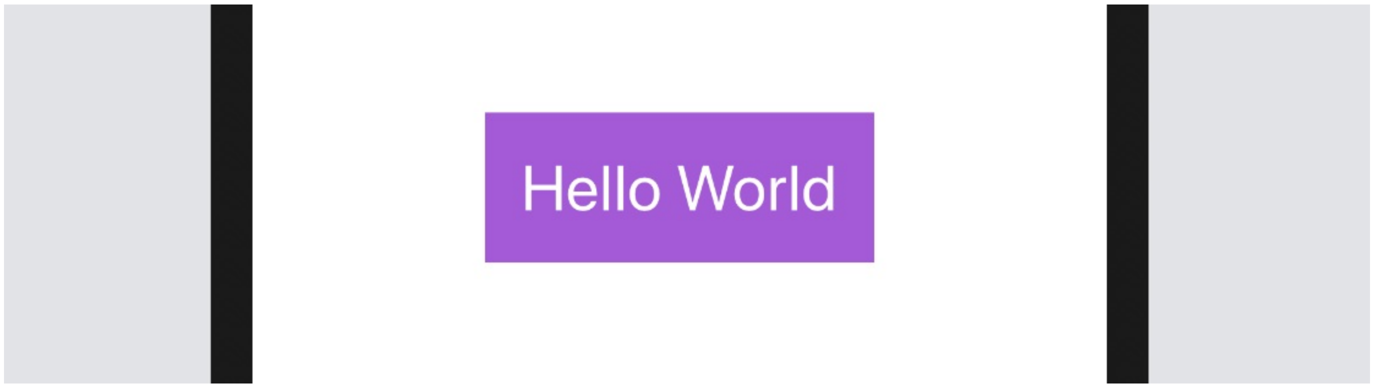
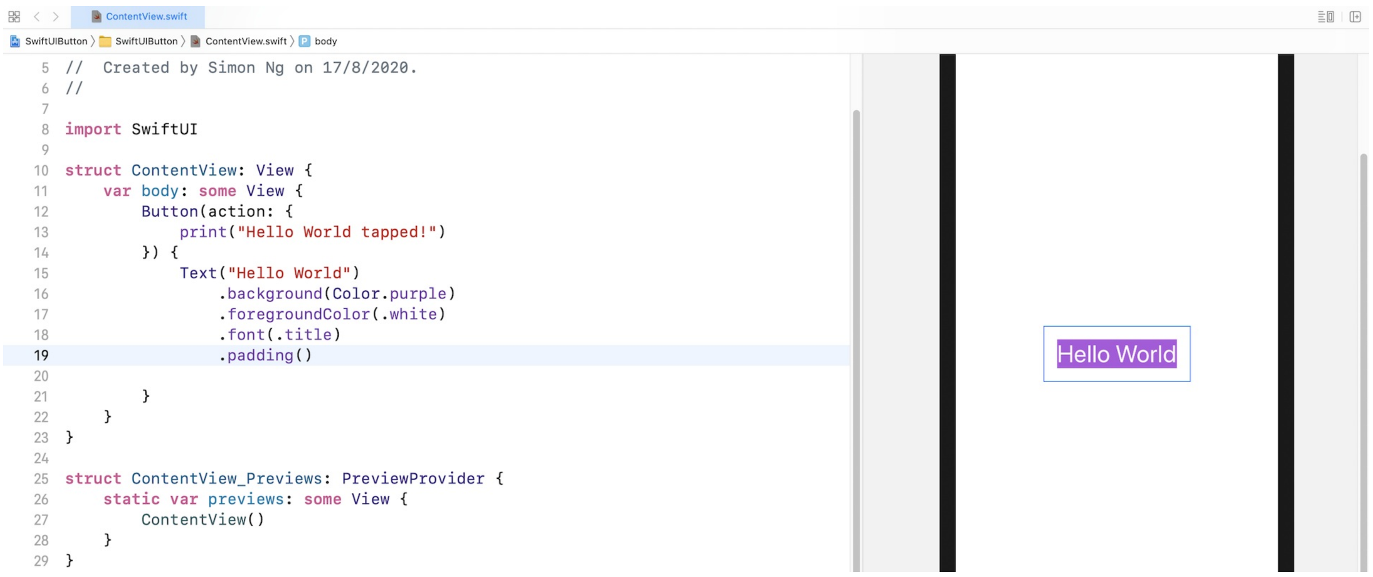


Figure 6. Adding padding to the button

The Order of Modifiers is Important

One thing I want to highlight is that the `padding` modifier should be placed before the `background` modifier. If you write the code as illustrated below, the end result will be different.



```
//  Created by Simon Ng on 17/8/2020.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        Button(action: {
            print("Hello World tapped!")
        }) {
            Text("Hello World")
                .background(Color.purple)
                .foregroundColor(.white)
                .font(.title)
                .padding()

        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 7. Placing the padding modifier after the background modifier

If you place the `padding` modifier after the `background` modifier, you can still add some padding to the button but without the preferred background color. If you wonder why, the modifiers like this:

```
Text("Hello World")
    .background(Color.purple) // 1. Change the background color to purple
    .foregroundColor(.white)  // 2. Set the foreground/font color to white
    .font(.title)             // 3. Change the font type
    .padding()                // 4. Add the paddings with the primary color (i.e.
white)
```

Conversely, the modifiers will work like this if the `padding` modifier is placed before the `background` modifier:

```
Text("Hello World")
    .padding()                 // 1. Add the paddings
    .background(Color.purple) // 2. Change the background color to purple includin
g the padding
    .foregroundColor(.white)  // 3. Set the foreground/font color to white
    .font(.title)              // 4. Change the font type
```

Adding Borders to the Button

This doesn't mean the `padding` modifier should always be placed at the very beginning. It just depends on your button design. Let's say, you want to create a button with borders like this:

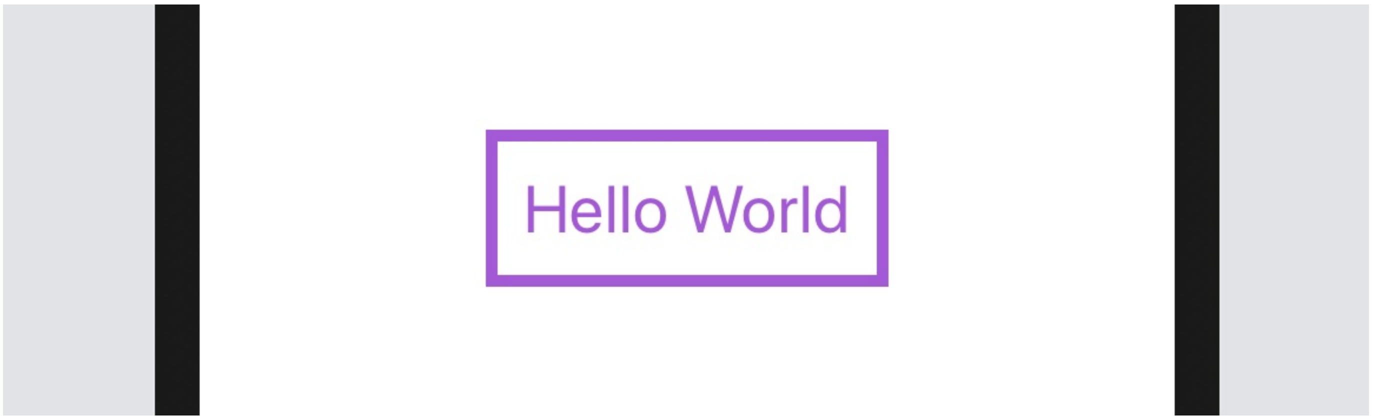


Figure 8. A button with borders

You can change the code of the `Text` control like below:

```
Text("Hello World")
    .foregroundColor(.purple)
    .font(.title)
    .padding()
    .border(Color.purple, width: 5)
```

Here we set the foreground color to purple and then add some empty padding around the text. The `border` modifier is used to define the border's width and color. Please alter the value of the `width` parameter to see how it works.

Let me give you another example. Let's say, a designer shows you the following button design. How are you going to implement it with what you've learned? Before you read the next paragraph, Take a few minutes to figure out the solution.

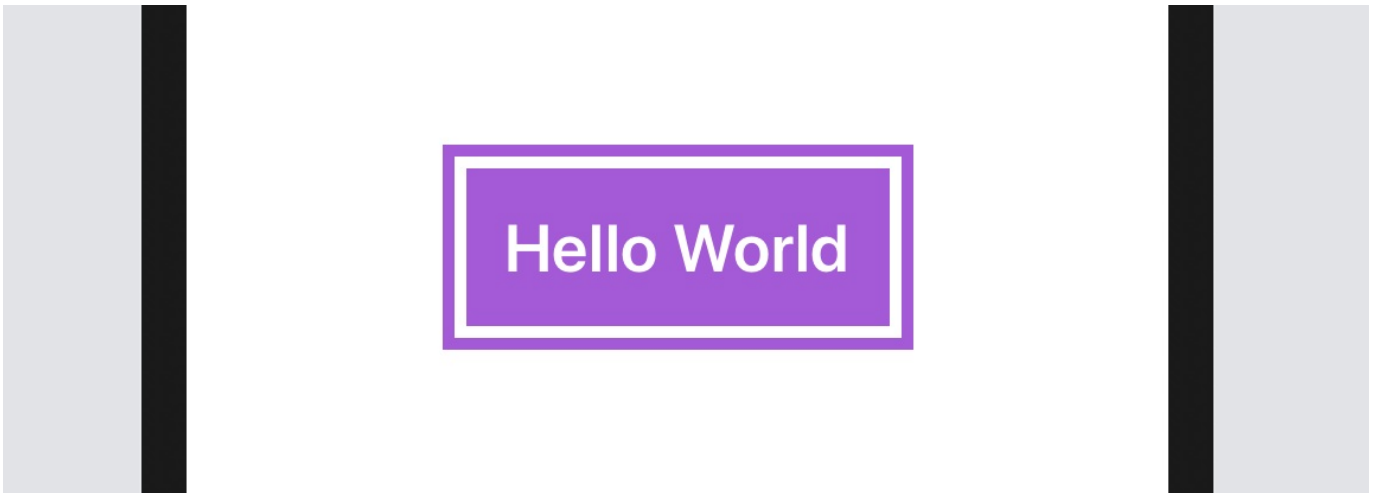


Figure 9. A button with both background and border

Okay, here is the solution:

```
Text("Hello World")
    .fontWeight(.bold)
    .font(.title)
    .padding()
    .background(Color.purple)
    .foregroundColor(.white)
    .padding(10)
    .border(Color.purple, width: 5)
```

We use two `padding` modifiers to create the button design. The first `padding`, together with the `background` modifier, is for creating a button with padding and a purple background. The `padding(10)` modifier adds extra padding around the button and the `border` modifier specifies a rounded border in purple.

Let's look at a more complex example. What if you wanted a button with rounded borders like this?

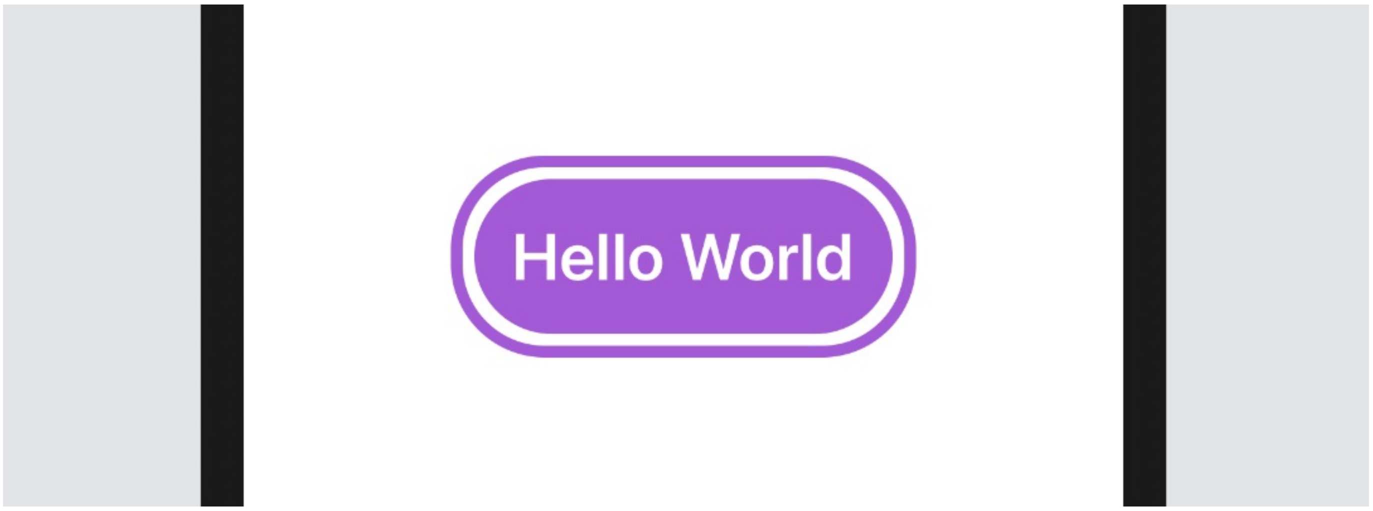


Figure 10. A button with a rounded border

SwiftUI comes with a modifier named `cornerRadius` that lets you easily create rounded corners. To render the button's background with rounded corners, you simply use the modifier and specify the corner radius:

```
.cornerRadius(40)
```

For the border with rounded corners, it'll take a little bit of work since the `border` modifier doesn't allow you to create rounded corners. What we need to do is to draw a border and overlay it on the button. Here is the final code:

```
Text("Hello World")
    .fontWeight(.bold)
    .font(.title)
    .padding()
    .background(Color.purple)
    .cornerRadius(40)
    .foregroundColor(.white)
    .padding(10)
    .overlay(
        RoundedRectangle(cornerRadius: 40)
            .stroke(Color.purple, lineWidth: 5)
    )
```

The `overlay` modifier lets you overlay another view on top of the current view. In the code, we draw a border with rounded corners using the `stroke` modifier of the `RoundedRectangle` object. The `stroke` modifier allows you to configure the color and line width of the stroke.

Creating a Button with Images and Text

So far, we have only worked with text buttons. In a real world project, you or your designer may want to display a button with an image. The syntax of creating an image button is exactly the same except that you use the `Image` control instead of the `Text` control like this:

```
Button(action: {
    print("Delete button tapped!")
}) {
    Image(systemName: "trash")
        .font(.largeTitle)
        .foregroundColor(.red)
}
```

For convenience, we use the built-in SF Symbols (i.e. trash) to create the image button. We specify `.largeTitle` in the `font` modifier to make the image a bit larger. Your button should look like this:

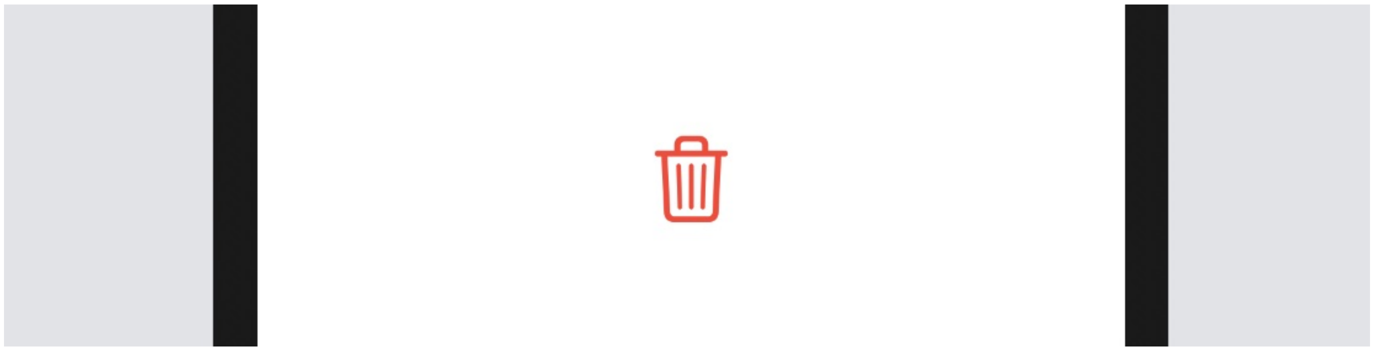
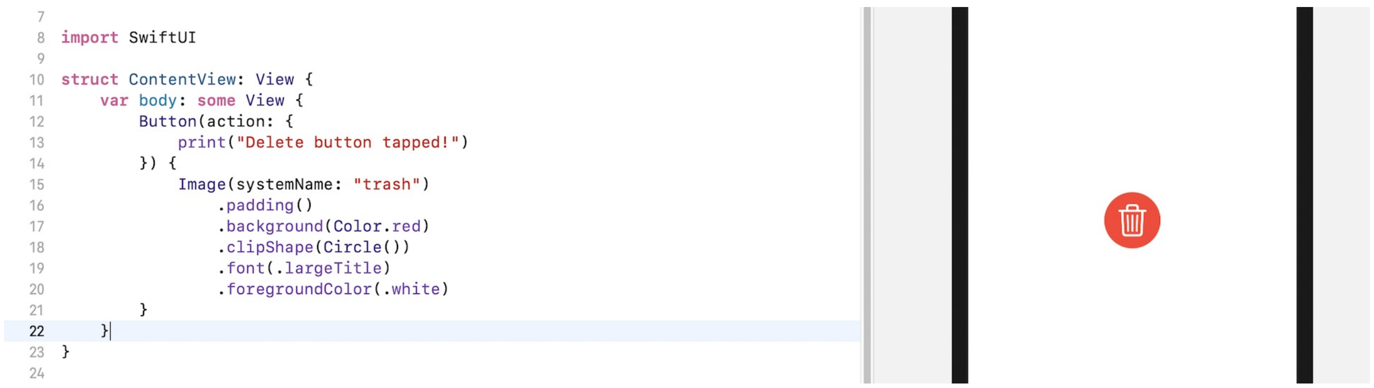


Figure 11. An image button

Similarly, if you want to create a circular image button with a solid background color, you can apply the modifiers we discussed earlier. Figure 12 shows you an example.



```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Button(action: {
            print("Delete button tapped!")
        }) {
            Image(systemName: "trash")
                .padding()
                .background(Color.red)
                .clipShape(Circle())
                .font(.largeTitle)
                .foregroundColor(.white)
        }
    }
}
```

Figure 12. A circular image button

You can use both text and image to create a button. Say, you want to put the word "Delete" next to the icon. Replace the code like this:

```
Button(action: {
    print("Delete tapped!")
}) {
    HStack {
        Image(systemName: "trash")
            .font(.title)
        Text("Delete")
            .fontWeight(.semibold)
            .font(.title)
    }
    .padding()
    .foregroundColor(.white)
    .background(Color.red)
    .cornerRadius(40)
}
```

Here we embed both the image and the text control in a horizontal stack. This will lay out the *trash* icon and the *Delete* text side by side. The modifiers applied to the `HStack` set the background color, padding, and round the button's corners. Figure 13 shows the resulting button.

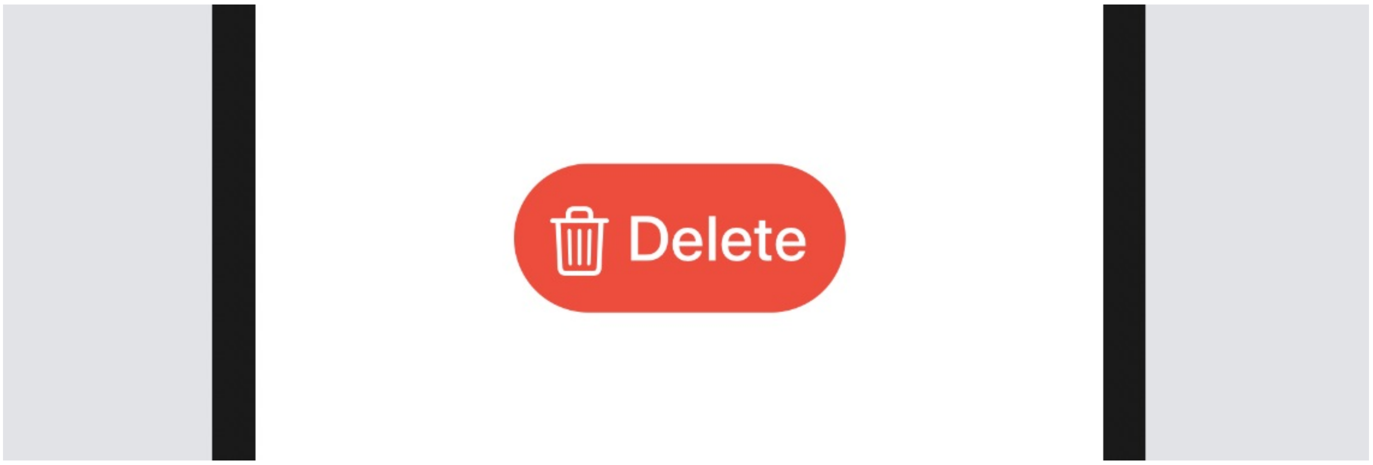


Figure 13. A button with both image and text

Using Label

In iOS 14, the SwiftUI framework introduces a new view called `Label` that lets you place an image and text side by side. Thus, instead of using `HStack`, you can use `Label` to create the same layout.

```
Button(action: {
    print("Delete tapped!")
}) {
    Label(
        title: { Text("Delete")
            .fontWeight(.semibold)
            .font(.title)
        },
        icon: { Image(systemName: "trash")
            .font(.title)
        }
    )
    .padding()
    .foregroundColor(.white)
    .background(Color.red)
    .cornerRadius(40)
}
```

Creating a Button with Gradient Background and Shadow

With SwiftUI, you can easily style the button with a gradient background. Not only can you define a specific color to the `background` modifier, you can easily apply a gradient effect to any button. All you need to do is to replace the following line of code:

```
.background(Color.red)
```

With:

```
.background(LinearGradient(gradient: Gradient(colors: [Color.red, Color.blue]), st
artPoint: .leading, endPoint: .trailing))
```

The SwiftUI framework comes with several built-in gradient effects. The code above applies a linear gradient from left (`.leading`) to right (`.trailing`). It begins with red on the left and ends with blue on the right.

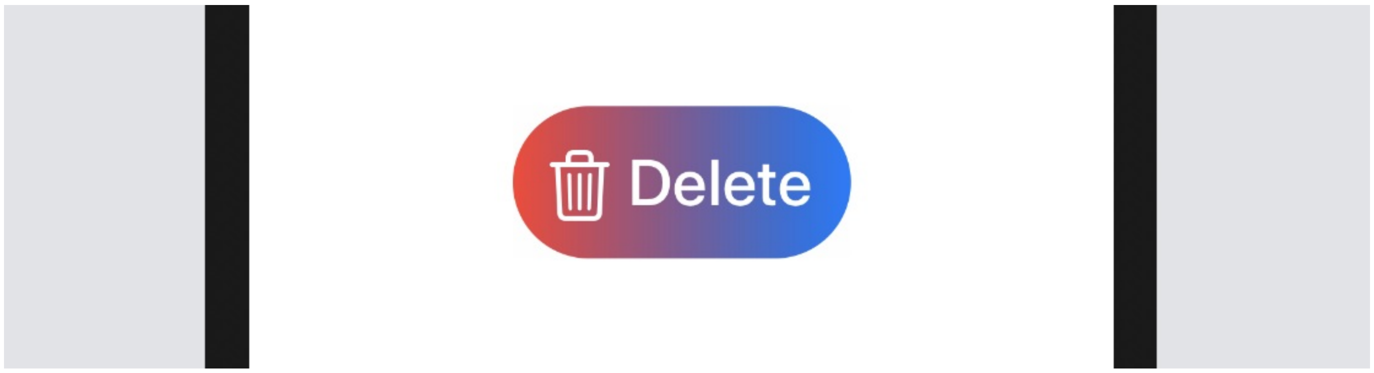


Figure 14. A button with gradient background

If you want to apply the gradient from top to bottom, you replace the `.leading` with `.top` and the `.trailing` with `.bottom` like this:

```
.background(LinearGradient(gradient: Gradient(colors: [Color.red, Color.blue]), st
artPoint: .top, endPoint: .bottom))
```

You're free to use your own colors to render the gradient effect. Let's say, your designer tells you to use the following gradient:

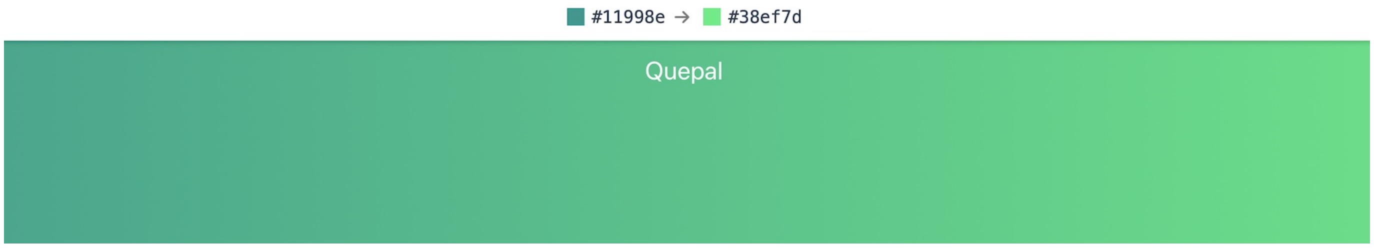


Figure 15. A sample gradient from uigradients.com

There are multiple ways to convert the color code from hex to the compatible format in Swift. Here is one approach. In the project navigator, choose the asset catalog (`Assets.xcassets`). Right click the blank area (under AppIcon) and select *Color Set*.

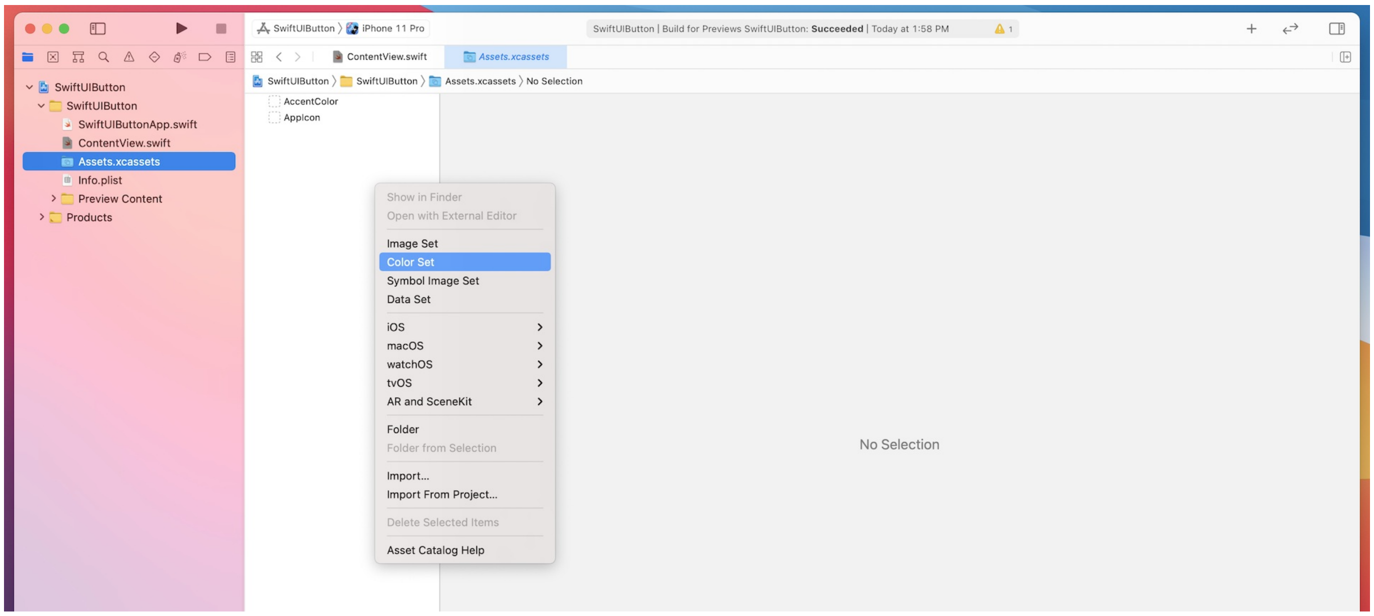


Figure 16. Define a new color set in the asset catalog

Next, choose the color well for *Any Appearance* and click the *Show inspector* button. Then click the *Attributes inspector* icon to reveal the attributes of a color set. In the name field, set the name to *DarkGreen*. In the *Color* section, change the input method to *8-bit Hexadecimal*.

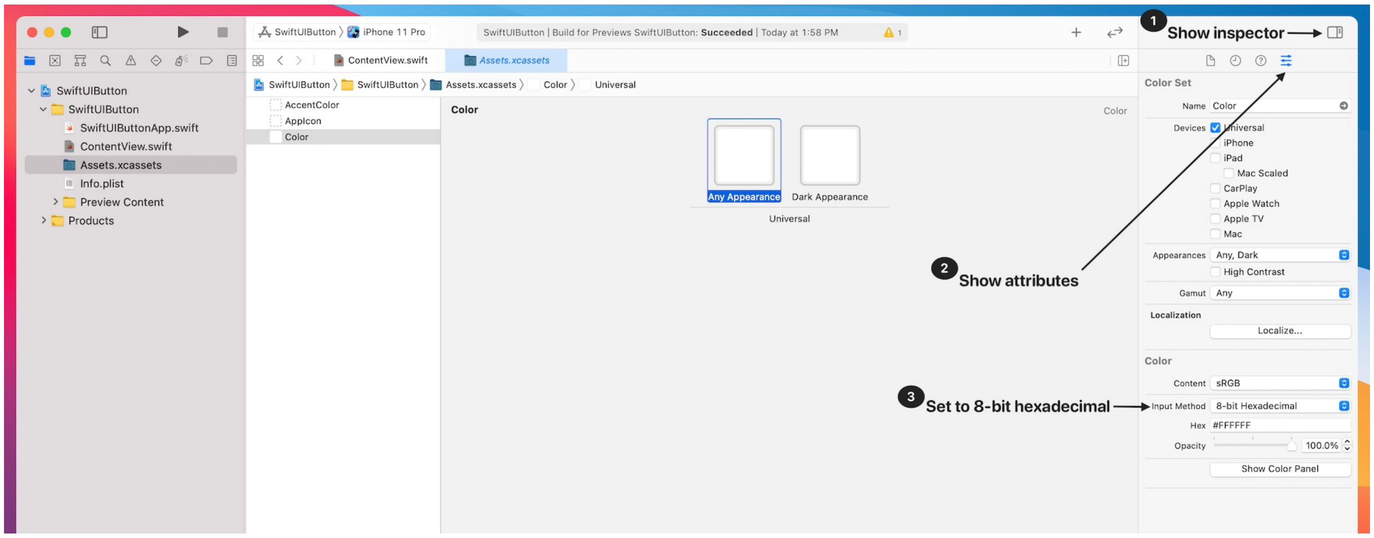


Figure 17. Editing the attributes of a color set

Now you can set the color code in the *Hex* field. For this example, enter `#11998e` to define the color. Name the color set *LightGreen*. Repeat the same procedure to define another color set. Enter `#38ef7d` for the additional color. Name this color *DarkGreen*.

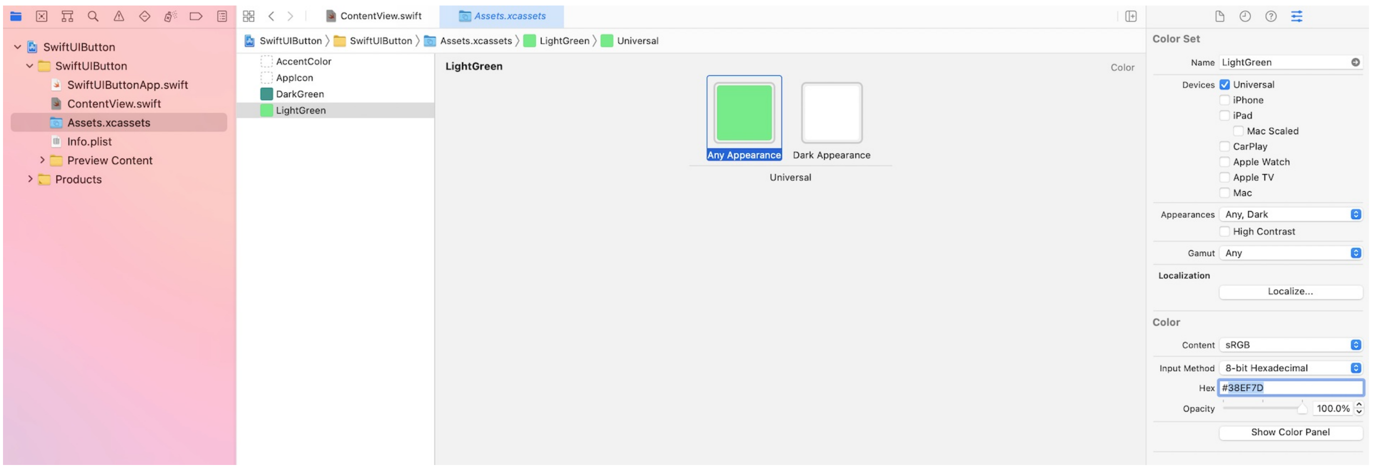


Figure 18. Define two color sets

Now that you've defined two color sets, let's go back to `ContentView.swift` and update the code. To use the color set, you just need to specify the name of the color set like this:

```
Color("DarkGreen")
Color("LightGreen")
```

To render the gradient with the *DarkGreen* and *LightGreen* color sets, all you need is to update the `background` modifier like this:

```
.background(LinearGradient(gradient: Gradient(colors: [Color("DarkGreen"), Color("
LightGreen")]), startPoint: .leading, endPoint: .trailing))
```

If you've made the change correctly, your button should have a nice gradient background as shown in figure 19.

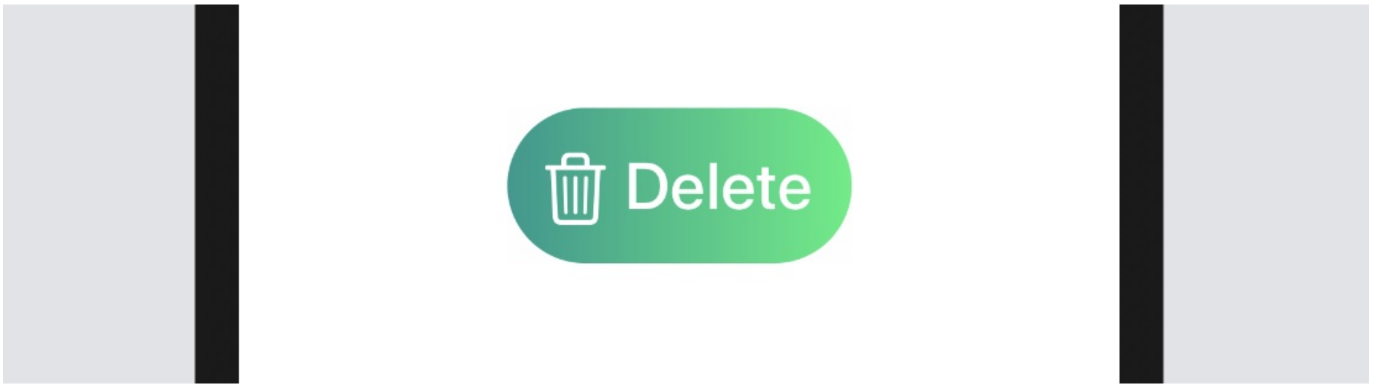


Figure 19. Generating a gradient with our own colors

There is one more modifier I want to show you in this section. The `shadow` modifier allows you to draw a shadow around the button (or any view). Just add this line of code after the `cornerRadius` modifier to see the shadow:

```
.shadow(radius: 5.0)
```

Optionally, you can control the color, radius, and position of the shadow. Here is an example:

```
.shadow(color: .gray, radius: 20.0, x: 20, y: 10)
```

Creating a Full-width Button

Bigger buttons usually grab user's attention. Sometimes, you may need to create a full-width button that takes up the width of the screen. The `frame` modifier is designed to control the size of a view. Whether you want to create a fixed size button or a button with variable width, you use this modifier. To create a full-width button, you can change the `Button` code like this:

```
Button(action: {
    print("Delete tapped!")
}) {
    HStack {
        Image(systemName: "trash")
            .font(.title)
        Text("Delete")
            .fontWeight(.semibold)
            .font(.title)
    }
    .frame(minWidth: 0, maxWidth: .infinity)
    .padding()
    .foregroundColor(.white)
    .background(LinearGradient(gradient: Gradient(colors: [Color("DarkGreen"), Col
or("LightGreen")]), startPoint: .leading, endPoint: .trailing))
    .cornerRadius(40)
}
```

This is very similar to the code we just wrote, except that we added the `frame` modifier before `padding`. Here we define a flexible width for the button. We set the `maxWidth` parameter to `.infinity`. This will result in the button filling the width of the container view. You should now see a full-width button in the canvas.

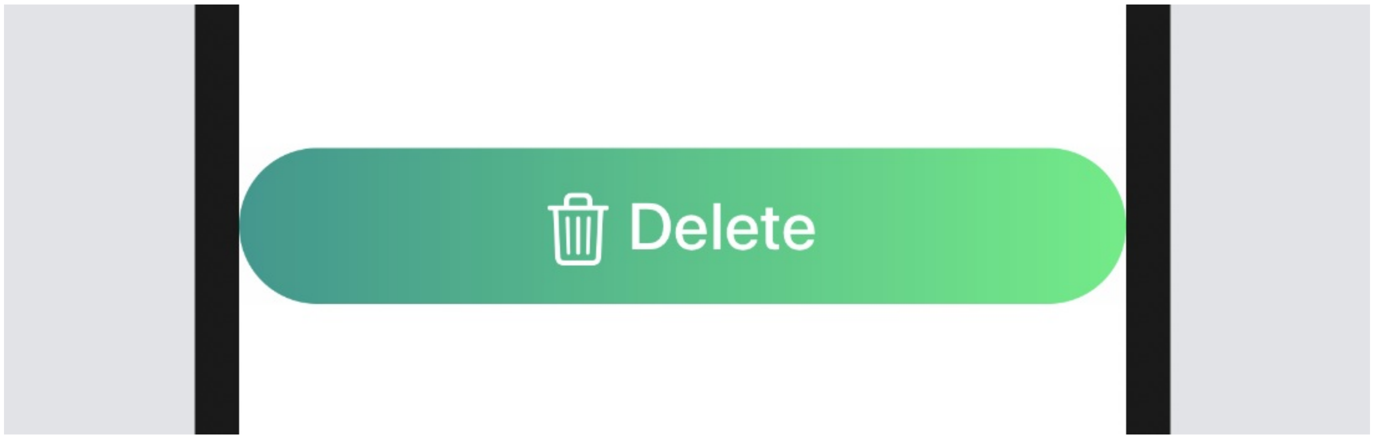


Figure 20. A full-width button

By defining `maxWidth` to `.infinity`, the width of the button will be adjusted automatically depending on the screen width of the device. If you want to give the button some more horizontal space, insert a `padding` modifier after `.cornerRadius(40)`:

```
.padding(.horizontal, 20)
```

Styling Buttons with ButtonStyle

In a real world app, the same button design will be utilised in multiple buttons. Let's say, you're creating three buttons: *Delete*, *Edit*, and *Share* that all have the same button style like this:

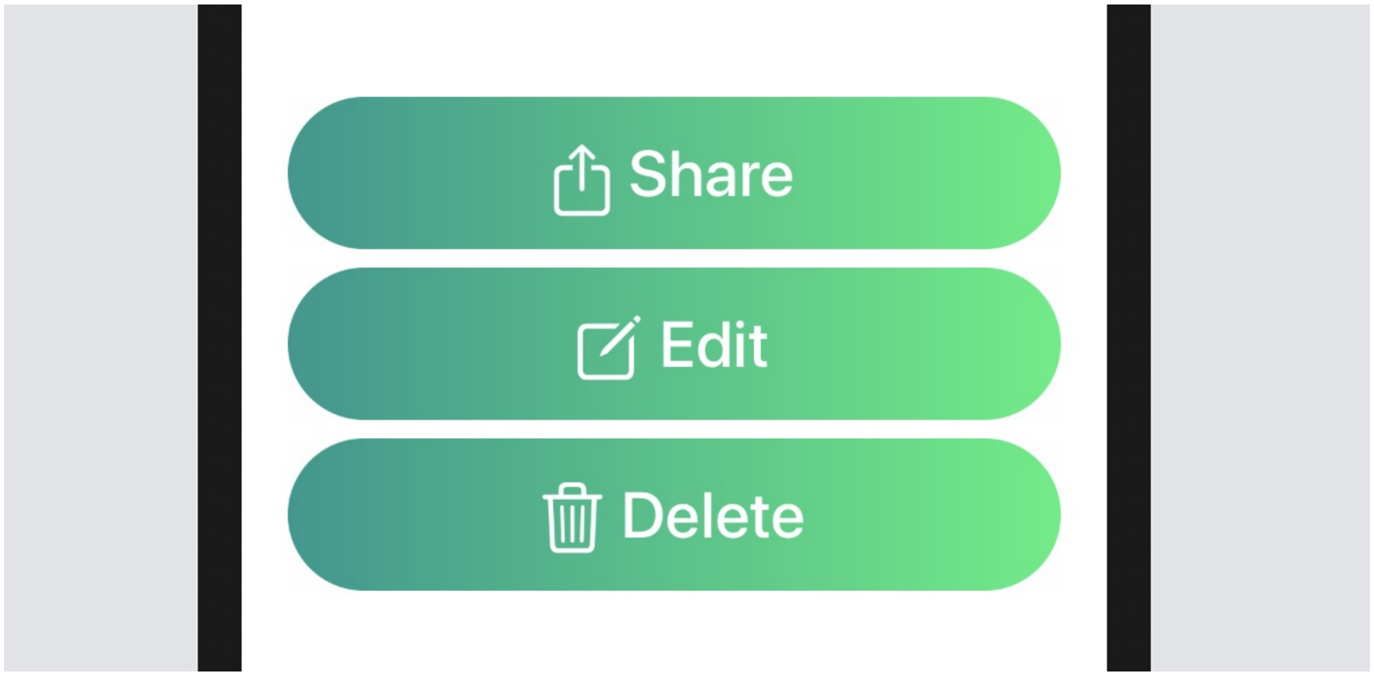


Figure 21. A full-width button

You'll probably write the code like this:

```
struct ContentView: View {

    var body: some View {

        VStack {

            Button(action: {
                print("Share tapped!")
            }) {
                HStack {
                    Image(systemName: "square.and.arrow.up")
                        .font(.title)
                    Text("Share")
                        .fontWeight(.semibold)
                        .font(.title)
                }
                .frame(minWidth: 0, maxWidth: .infinity)
                .padding()
                .foregroundColor(.white)
                .background(LinearGradient(gradient: Gradient(colors: [Color("Dark
Green"), Color("LightGreen")]), startPoint: .leading, endPoint: .trailing))
                .cornerRadius(40)
                .padding(.horizontal, 20)
            }

            Button(action: {
                print("Edit tapped!")
            }) {
                HStack {
                    Image(systemName: "square.and.pencil")
                        .font(.title)
                    Text("Edit")
                        .fontWeight(.semibold)
                        .font(.title)
                }
                .frame(minWidth: 0, maxWidth: .infinity)
                .padding()
                .foregroundColor(.white)
                .background(LinearGradient(gradient: Gradient(colors: [Color("Dark
Green"), Color("LightGreen")]), startPoint: .leading, endPoint: .trailing))
                .cornerRadius(40)
                .padding(.horizontal, 20)
            }

            Button(action: {
                print("Delete tapped!")
            }) {
                HStack {
                    Image(systemName: "trash")
                        .font(.title)
                    Text("Delete")
                        .fontWeight(.semibold)
                        .font(.title)
                }
                .frame(minWidth: 0, maxWidth: .infinity)
                .padding()
                .foregroundColor(.white)
                .background(LinearGradient(gradient: Gradient(colors: [Color("Dark
Green"), Color("LightGreen")]), startPoint: .leading, endPoint: .trailing))
                .cornerRadius(40)
                .padding(.horizontal, 20)
            }

        }
    }
}
```

As you can see from the code above, you need to replicate all modifiers for each of the buttons. What if you or your designer want to modify the button style? You'll need to modify all the modifiers. That's quite a tedious task and not good coding practice. How can you generalize the style and make it reusable?

SwiftUI provides a protocol called `ButtonStyle` for you to create your own button style. To create a reusable style for our buttons, Create a new struct called `GradientBackgroundStyle` that conforms to the `ButtonStyle` protocol. Insert the following code snippet and put it right above `struct ContentPreview_Previews` :

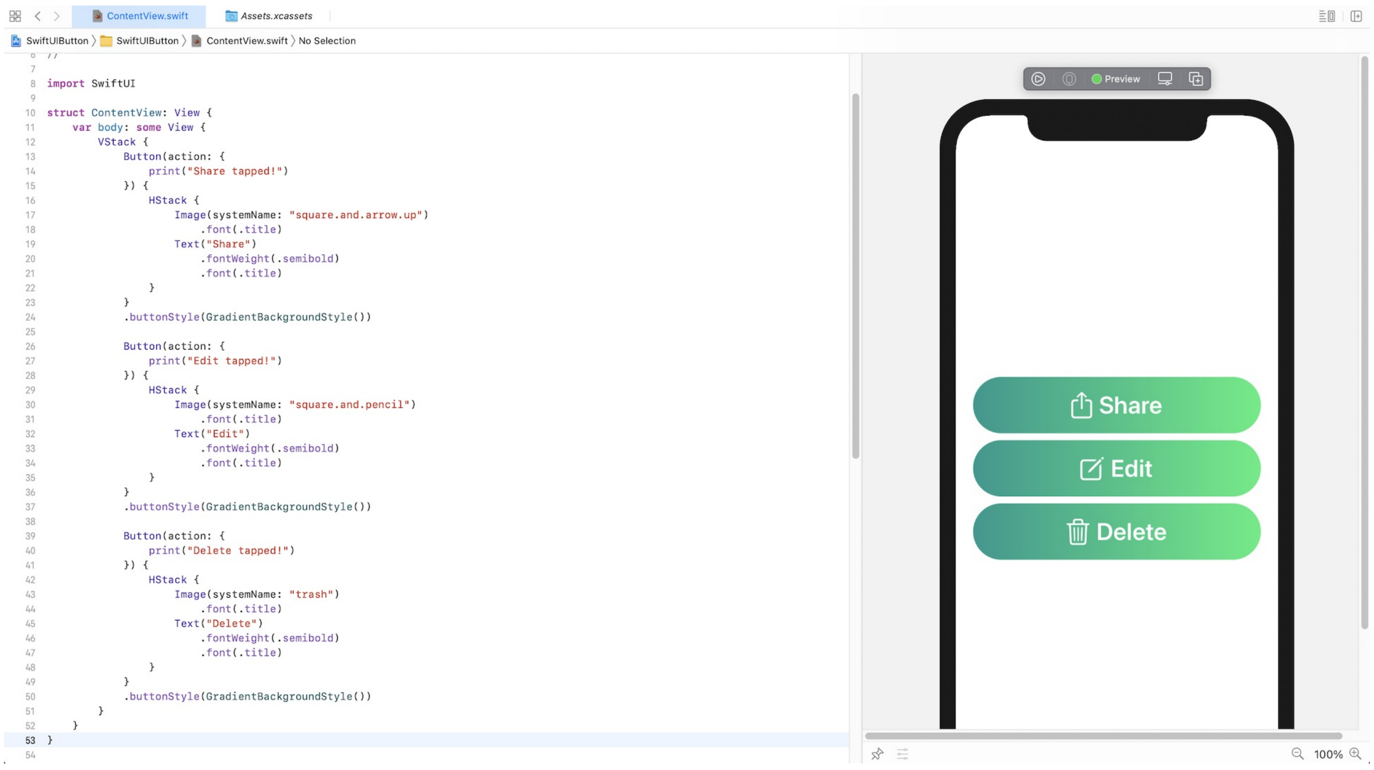
```
struct GradientBackgroundStyle: ButtonStyle {

    func makeBody(configuration: Self.Configuration) -> some View {
        configuration.label
            .frame(minWidth: 0, maxWidth: .infinity)
            .padding()
            .foregroundColor(.white)
            .background(LinearGradient(gradient: Gradient(colors: [Color("DarkGree
n"), Color("LightGreen")]), startPoint: .leading, endPoint: .trailing))
            .cornerRadius(40)
            .padding(.horizontal, 20)
    }
}
```

The protocol requires us to provide the implementation of the `makeBody` function that accepts a `configuration` parameter. The `configuration` parameter includes a `label` property applies modifiers to change the button's style. In the code above, we apply the same set of modifiers that we used before.

So, how do you apply the custom style to a button? SwiftUI provides a modifier called `.buttonStyle` for you to apply the button style like this:

```
Button(action: {
    print("Delete tapped!")
}) {
    HStack {
        Image(systemName: "trash")
            .font(.title)
        Text("Delete")
            .fontWeight(.semibold)
            .font(.title)
    }
}
.buttonStyle(GradientBackgroundStyle())
```

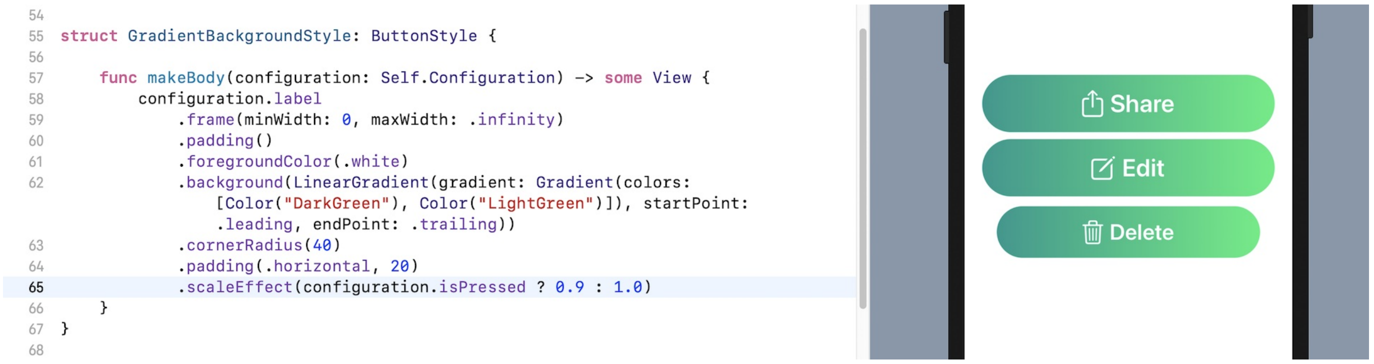
Cool, right? The code is now simplified and you can easily apply the button style to any button with just one line of code.


ContentView.swift
Assets.xcassets
SwiftUIButton 〉 SwiftUIButton 〉 ContentView.swift 〉 No Selection
Preview
Share
Edit
Delete
100%

```
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        VStack {
            Button(action: {
                print("Share tapped!")
            }) {
                HStack {
                    Image(systemName: "square.and.arrow.up")
                        .font(.title)
                    Text("Share")
                        .fontWeight(.semibold)
                        .font(.title)
                }
            }
            .buttonStyle(GradientBackgroundStyle())

            Button(action: {
                print("Edit tapped!")
            }) {
                HStack {
                    Image(systemName: "square.and.pencil")
                        .font(.title)
                    Text("Edit")
                        .fontWeight(.semibold)
                        .font(.title)
                }
            }
            .buttonStyle(GradientBackgroundStyle())

            Button(action: {
                print("Delete tapped!")
            }) {
                HStack {
                    Image(systemName: "trash")
                        .font(.title)
                    Text("Delete")
                        .fontWeight(.semibold)
                        .font(.title)
                }
            }
            .buttonStyle(GradientBackgroundStyle())
        }
    }
}
```

Figure 22. Applying the custom style using .buttonStyle modifier

You can also determine if the button is pressed by accessing the `isPressed` property. This allows you to alter the style of the button when the user taps on it. For example, let's say we want to make the button a bit smaller when someone presses the button. You add a line of code like this:



```
struct GradientBackgroundStyle: ButtonStyle {

    func makeBody(configuration: Self.Configuration) -> some View {
        configuration.label
            .frame(minWidth: 0, maxWidth: .infinity)
            .padding()
            .foregroundColor(.white)
            .background(LinearGradient(gradient: Gradient(colors:
                [Color("DarkGreen"), Color("LightGreen")]), startPoint:
                .leading, endPoint: .trailing))
            .cornerRadius(40)
            .padding(.horizontal, 20)
            .scaleEffect(configuration.isPressed ? 0.9 : 1.0)
    }
}
```

Figure 23. Applying the custom style using .buttonStyle modifier

The `scaleEffect` modifier lets you scale up or down a button (and any view). To scale up the button, you provide a value greater than 1.0. To make the button smaller, enter a value less than 1.0.

```
.scaleEffect(configuration.isPressed ? 0.9 : 1.0)
```

So, what the line of code does is scale down the button (i.e. `0.9`) when the button is pressed and scales back to its original size (i.e. `1.0`) when the user lifts their finger. Run the app, you should see a nice animation when the button is scaled up and down. This is the power of SwiftUI. You do not need to write any extra lines of code and it comes with built-in animation.

Exercise

Your exercise is to create an animated button which shows a plus icon. When a user presses the button, the plus icon will rotate (clockwise/counterclockwise) to become a cross icon.

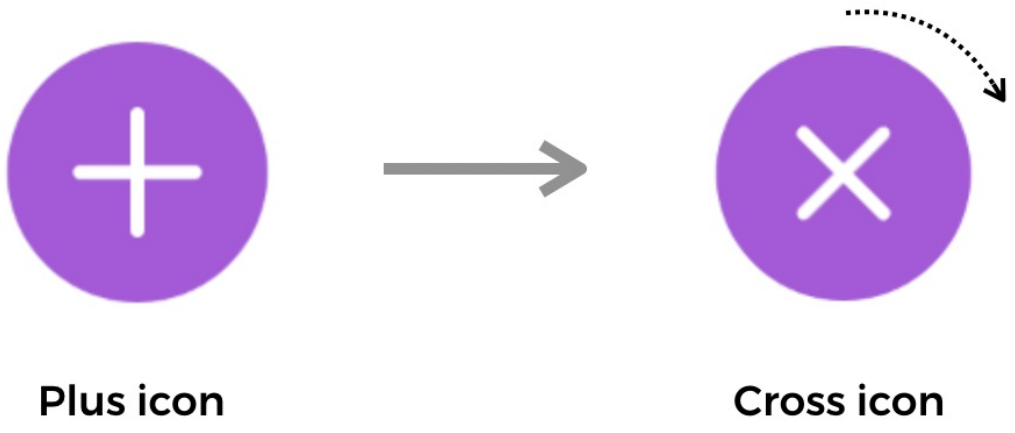


Figure 24. Rotate the icon when a user presses it

As a hint, the modifier `rotationEffect` may be used to rotate the button (or other view).

Summary

In this chapter, we covered the basics of creating buttons in SwiftUI. Buttons play a key role in any application UI. Well designed buttons, not only make your UI more appealing, but bring the user experience of your app to the next level. As you have learned, by mixing SF Symbols, gradients, and animations together, you can easily build attractive and useful buttons.

For reference, you can download the complete button project here:

- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUIButton.zip)

Chapter 7
Understanding State and Binding

State management is something every developer has to deal with in application development. Imagine that you are developing a music player app. When a user taps the *Play* button, the button will change itself to a *Stop* button. In your implementation, there must be some way to keep track of the application's state so that you know when to change the button's appearance.

Figure 1. Stop and Play buttons

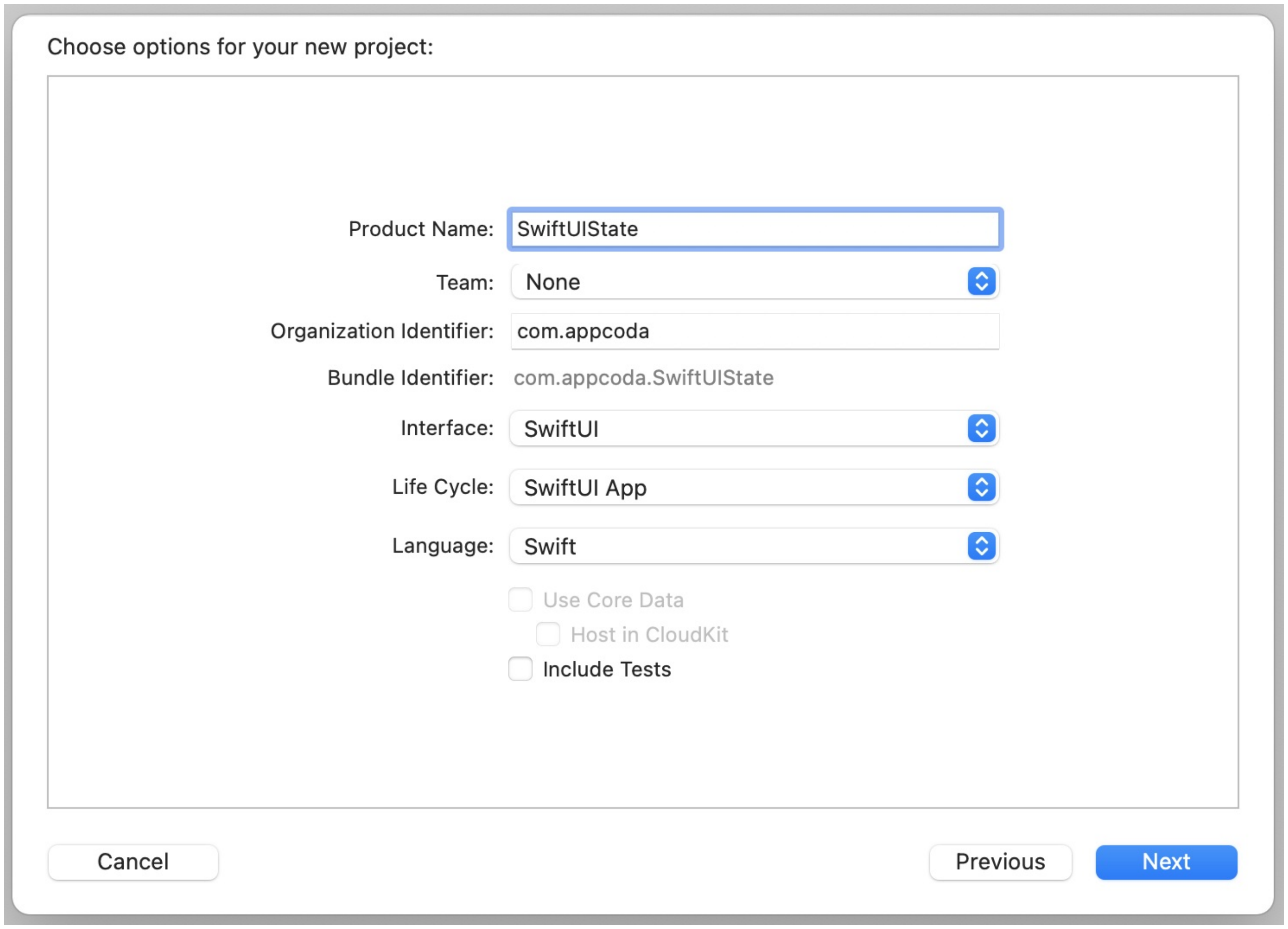
SwiftUI comes with a few built-in features for state management. In particular, it introduces a property wrapper named `@State`. When you annotate a property with `@State`, SwiftUI automatically stores it somewhere in your application. What's more, views that make use of that property automatically listen to the value change of the property. When the state changes, SwiftUI will recompute those views and update the application's appearance.

Doesn't it sound great? Or are you a bit confused with state management?

You will get a better understanding of state and binding after going through the coding examples in this chapter and a couple of exercises I've prepared for you. Please take some time to work on these. It will help you master this important concept of SwiftUI.

Creating a New Project with SwiftUI enabled

Let's start with a simple example that I just described earlier to see how to switch between a *Play* button and a *Stop* button, by keeping track of the application's state. First, fire up Xcode and create a new project using the *App* template. Set the name of the project to *SwiftUIState* but you're free to use any other name. Please make sure *SwiftUI* is selected as the *Interface* option.


Choose options for your new project:

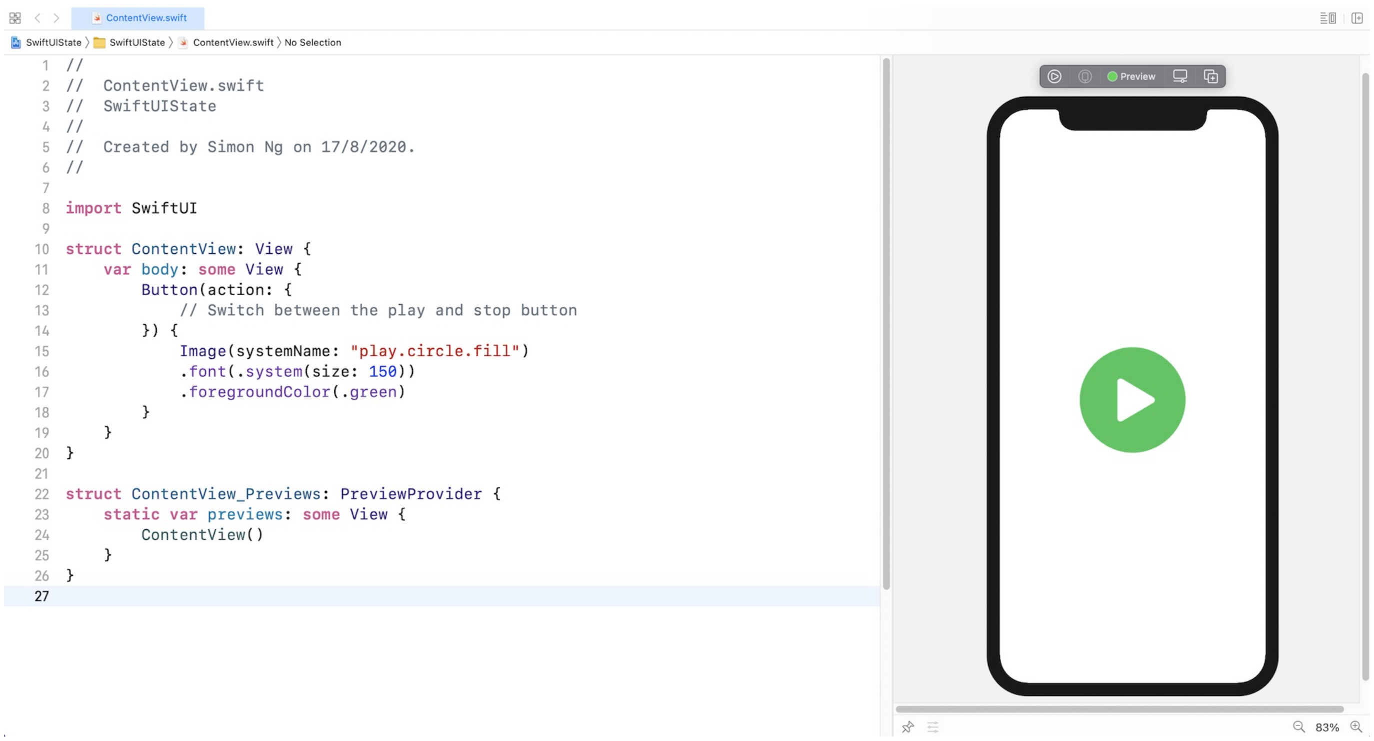
Product Name: SwiftUIState
Team: None
Organization Identifier: com.appcoda
Bundle Identifier: com.appcoda.SwiftUIState
Interface: SwiftUI
Life Cycle: SwiftUI App
Language: Swift
Use Core Data
Host in CloudKit
Include Tests

Figure 2. Creating a new project

Once you save the project, Xcode will load the `ContentView.swift` file and display a preview in the design canvas. Create the *Play* button like this:

```
Button(action: {
    // Switch between the play and stop button
}) {
    Image(systemName: "play.circle.fill")
    .font(.system(size: 150))
    .foregroundColor(.green)
}
```

We make use of the system image `play.circle.fill` and color the button green.


ContentView.swift
SwiftUIState 〉 SwiftUIState 〉 ContentView.swift 〉 No Selection
Preview
83%

```
//
//  ContentView.swift
//  SwiftUIState
//
//  Created by Simon Ng on 17/8/2020.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        Button(action: {
            // Switch between the play and stop button
        }) {
            Image(systemName: "play.circle.fill")
            .font(.system(size: 150))
            .foregroundColor(.green)
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 3. Previewing the play button

Controlling the Button's State

The button's action is empty. We want to change its appearance from *Play* to *Stop* when someone taps the button. The color of the button should also be changed to red when the stop button is displayed.

So, how can we implement that? Obviously, we need a variable to keep track of the button's state. Let's name it `isPlaying`. It's a boolean variable indicating whether the app is in the *Playing* state or not. If `isPlaying` is set to `true`, the app should show a *Stop* button. If `isPlaying` is set to `false`, the app shows a *Play* button. The code is written like this:

```
struct ContentView: View {

    private var isPlaying = false

    var body: some View {
        Button(action: {
            // Switch between play and stop button
        }) {
            Image(systemName: isPlaying ? "stop.circle.fill" : "play.circle.fill")
            .font(.system(size: 150))
            .foregroundColor(isPlaying ? .red : .green)
        }

    }
}
```

We change the image's name and color by referring the value of the `isPlaying` variable. If you update the code in your project, you should see a *Play* button in the preview canvas. However, if you set the default value of `isPlaying` to `true`, you would see a *Stop* button.

Now the question is how can the app monitor the change of the state (i.e. `isPlaying`) and update the button automatically? With SwiftUI, all you need to do is prefix the `isPlaying` property with `@State`:

```
@State private var isPlaying = false
```

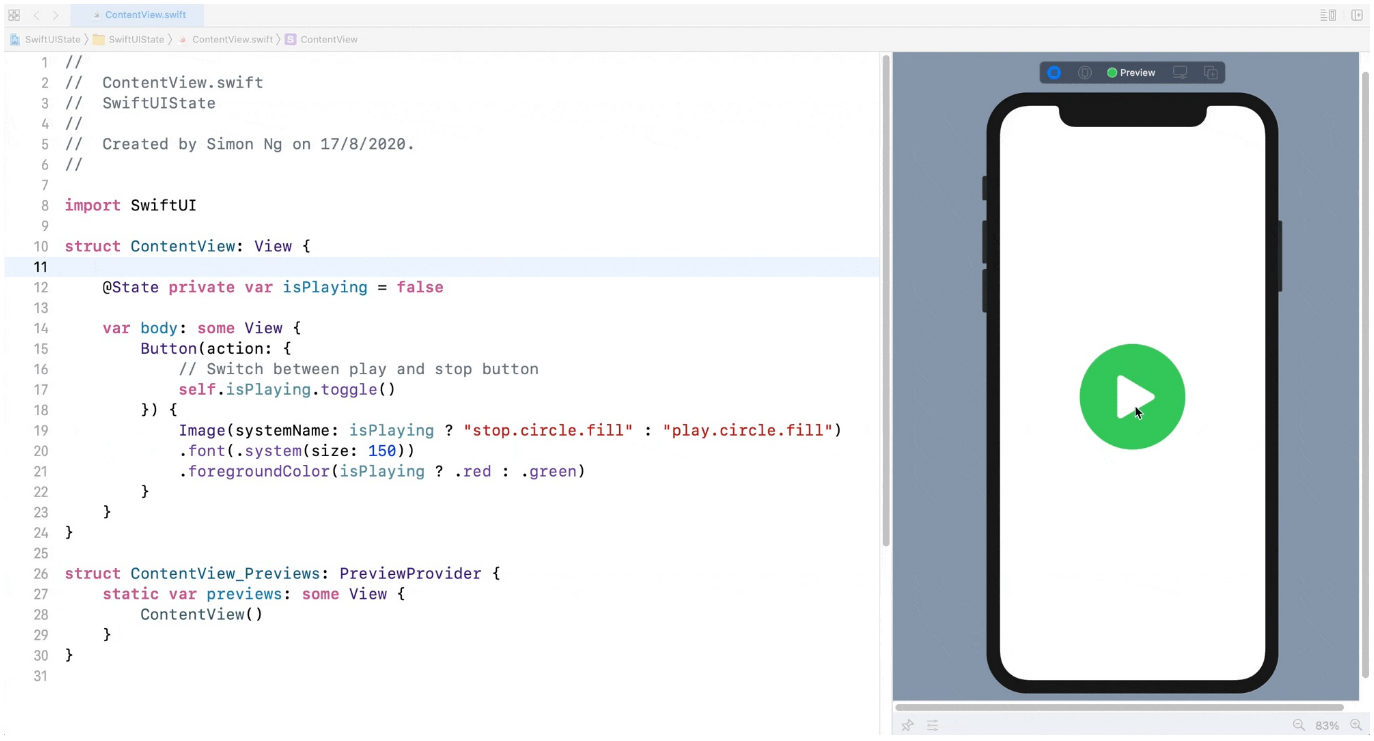
Once we declare the property as a state variable, SwiftUI manages the storage of `isPlaying` and monitors its value change. When the value of `isPlaying` changes, SwiftUI automatically recomputes the views that are referencing the `isPlaying` state. In our sample code, it's the Button that changes

> Only access a state property from inside the view's *body* (or from functions called by it). For this reason, you should declare your state properties as `private`, to prevent clients of your view from accessing it
>
> - Apple's official documentation (https://developer.apple.com/documentation/swiftui/state)

We still haven't implemented the button's action. So, let's do that now:

```
Button(action: {
    // Switch between play and stop button
    self.isPlaying.toggle()
}) {
    Image(systemName: isPlaying ? "stop.circle.fill" : "play.circle.fill")
    .font(.system(size: 150))
    .foregroundColor(isPlaying ? .red : .green)
}
```

In the `action` closure, we call the `toggle()` method to toggle the Boolean value from `false` to `true` or from `true` to `false`. Run the app by clicking the play icon in the preview canvas and toggle between the *Play* and *Stop* button.


ContentView.swift
SwiftUIState 〉 SwiftUIState 〉 ContentView.swift 〉 ContentView
Preview
83%

```
//
//  ContentView.swift
//  SwiftUIState
//
//  Created by Simon Ng on 17/8/2020.
//

import SwiftUI

struct ContentView: View {

    @State private var isPlaying = false

    var body: some View {
        Button(action: {
            // Switch between play and stop button
            self.isPlaying.toggle()
        }) {
            Image(systemName: isPlaying ? "stop.circle.fill" : "play.circle.fill")
            .font(.system(size: 150))
            .foregroundColor(isPlaying ? .red : .green)
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 4. Toggle between the Play and Stop button

Did you notice that SwiftUI renders a fade animation when you toggle between the buttons? This animation is built-in and automatically generated for you. We will talk more about animations in later chapters of the book, but as you can see, SwiftUI makes UI animation more approachable for all developers.

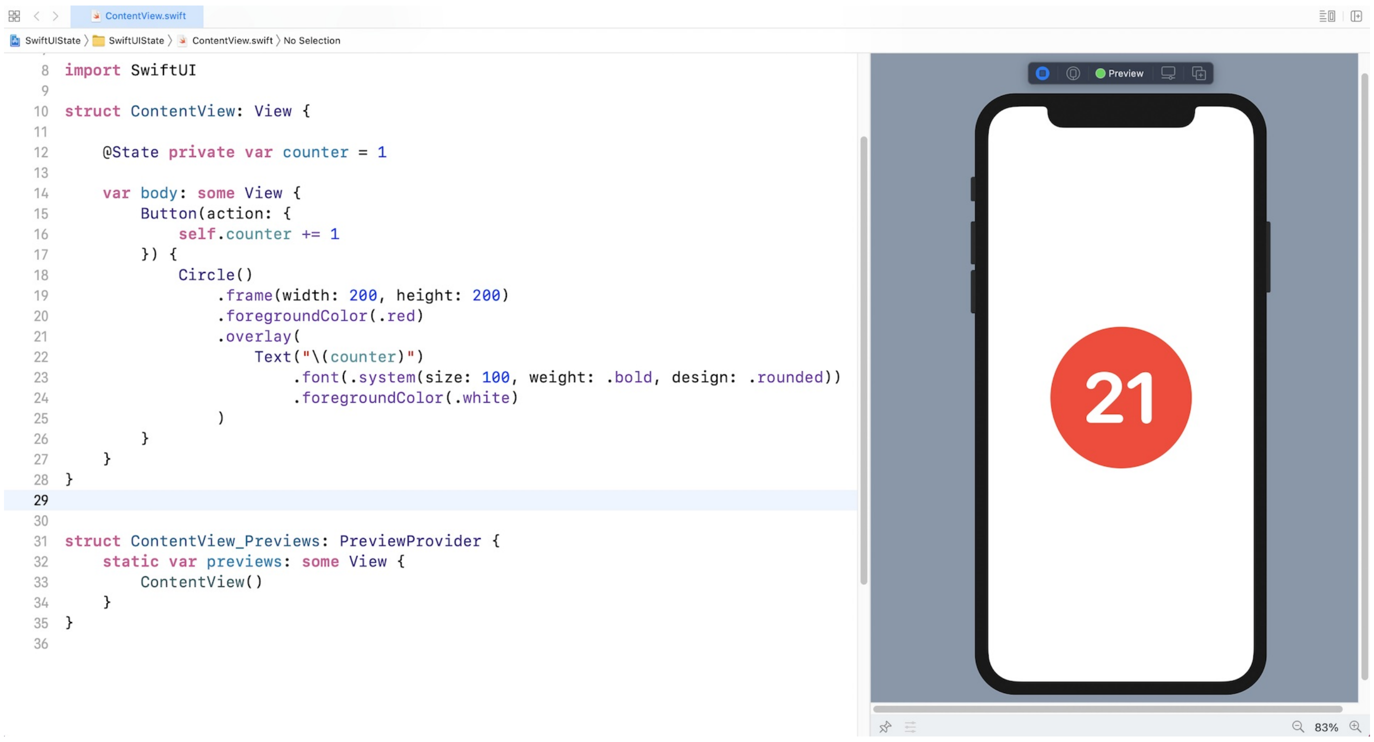
Exercise #1

Your exercise is to create a counter button which shows the number of taps. When a user taps the button, the counter will increase by one and display the total number of taps.

Figure 5. Toggle between the Play and Stop button

Working with Binding

Were you successful in creating the counter button? Instead of declaring a boolean variable as a state, we use an integer state variable to keep track of the count. When the button is tapped, the counter will increase by 1. Figure 6 shows the code snippet to achieve this.


ContentView.swift
SwiftUIState 〉 SwiftUIState 〉 ContentView.swift 〉 No Selection
Preview
21
83%

```swift
import SwiftUI

struct ContentView: View {

    @State private var counter = 1

    var body: some View {
        Button(action: {
            self.counter += 1
        }) {
            Circle()
                .frame(width: 200, height: 200)
                .foregroundColor(.red)
                .overlay(
                    Text("\(counter)")
                        .font(.system(size: 100, weight: .bold, design: .rounded))
                        .foregroundColor(.white)
                )
        }
    }
}


struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 6. A counter button

Now we will further modify the code to display three counter buttons (see figure 7). All three buttons share the same counter. No matter which button is tapped, the counter will increase by 1 and all the buttons will be invalidated to display the updated count.

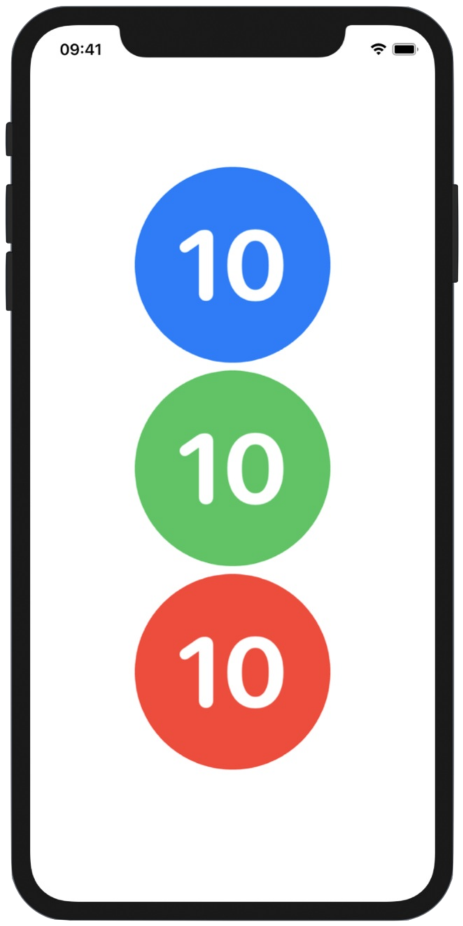


Figure 7. Three counter buttons

As you can see, all the buttons share the same look & feel. As I've explained in earlier chapters, rather than duplicating the code, it's always a good practice to extract a common view into a reusable subview. We can extract the Button to create an independent subview like this:

```
struct CounterButton: View {

    @Binding var counter: Int

    var color: Color

    var body: some View {
        Button(action: {
            self.counter += 1
        }) {
            Circle()
                .frame(width: 200, height: 200)
                .foregroundColor(color)
                .overlay(
                    Text("\(counter)")
                        .font(.system(size: 100, weight: .bold, design: .rounded))
                        .foregroundColor(.white)
                )
        }
    }
}
```

The `CounterButton` view accepts two parameters: *counter* and *color*. You can create a button colored red like this:

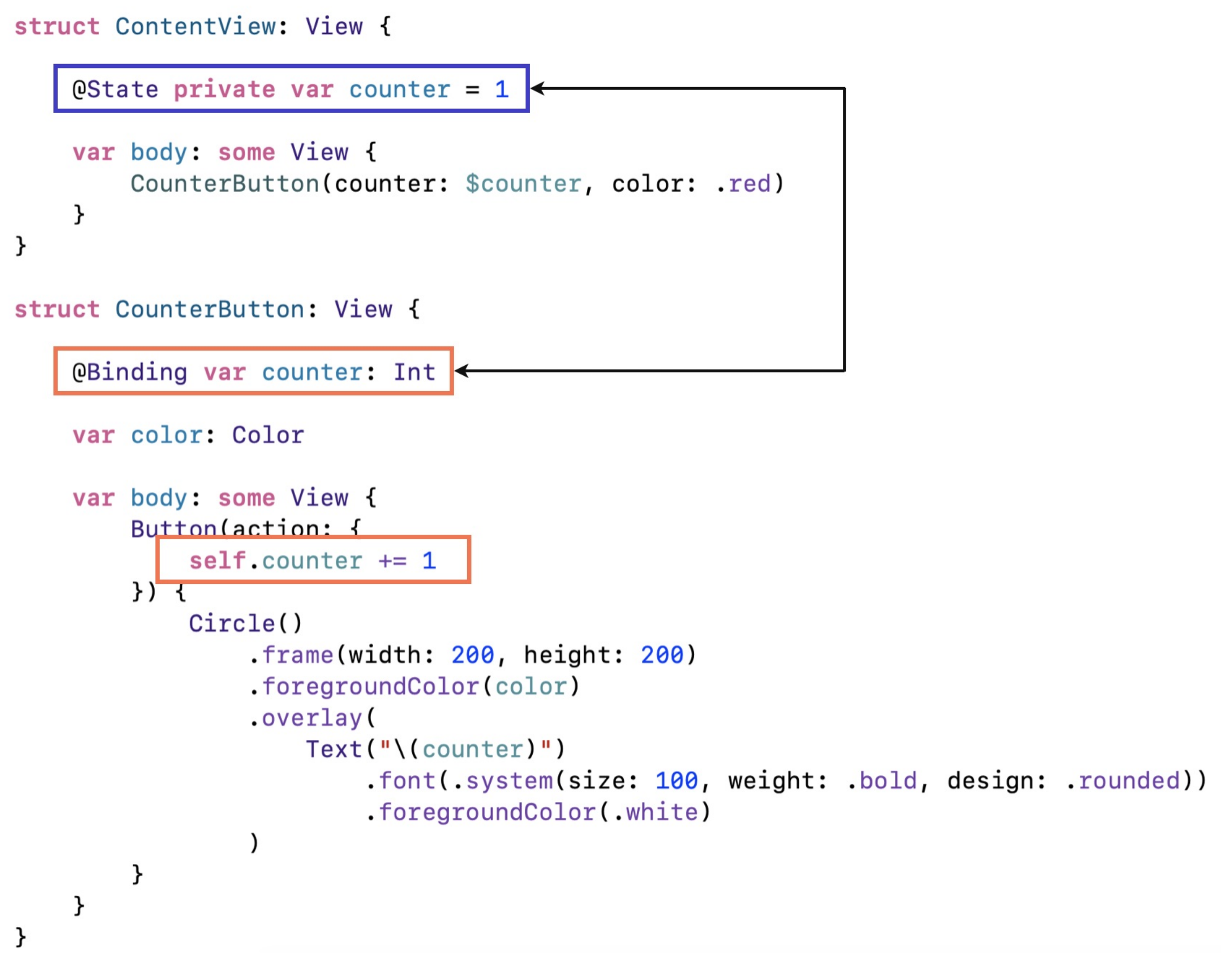
```
CounterButton(counter: $counter, color: .red)
```

You should notice that the `counter` variable is annotated with `@Binding`. When you create a `CounterButton` instance, the `counter` is prefixed by a $ sign.

What do they mean?

After we extract the button into a separate view, `CounterButton` becomes a subview of `ContentView`. The counter increment is now done in the `CounterButton` view instead of the `ContentView`. The `CounterButton` must have a way to manage the state variable in the `ContentView`.

The `@Binding` keyword indicates that the caller must provide the binding of the state variable. It's just like creating the two-way connection between the `counter` in the `ContentView` and the `counter` in the `CounterButton`. Updating `counter` in the `CounterButton` view propagates its value back to the `counter` state in the `ContentView`.



```
struct ContentView: View {

    @State private var counter = 1

    var body: some View {
        CounterButton(counter: $counter, color: .red)
    }
}

struct CounterButton: View {

    @Binding var counter: Int

    var color: Color

    var body: some View {
        Button(action: {
            self.counter += 1
        }) {
            Circle()
                .frame(width: 200, height: 200)
                .foregroundColor(color)
                .overlay(
                    Text("\(counter)")
                        .font(.system(size: 100, weight: .bold, design: .rounded))
                        .foregroundColor(.white)
                )
        }
    }
}
```

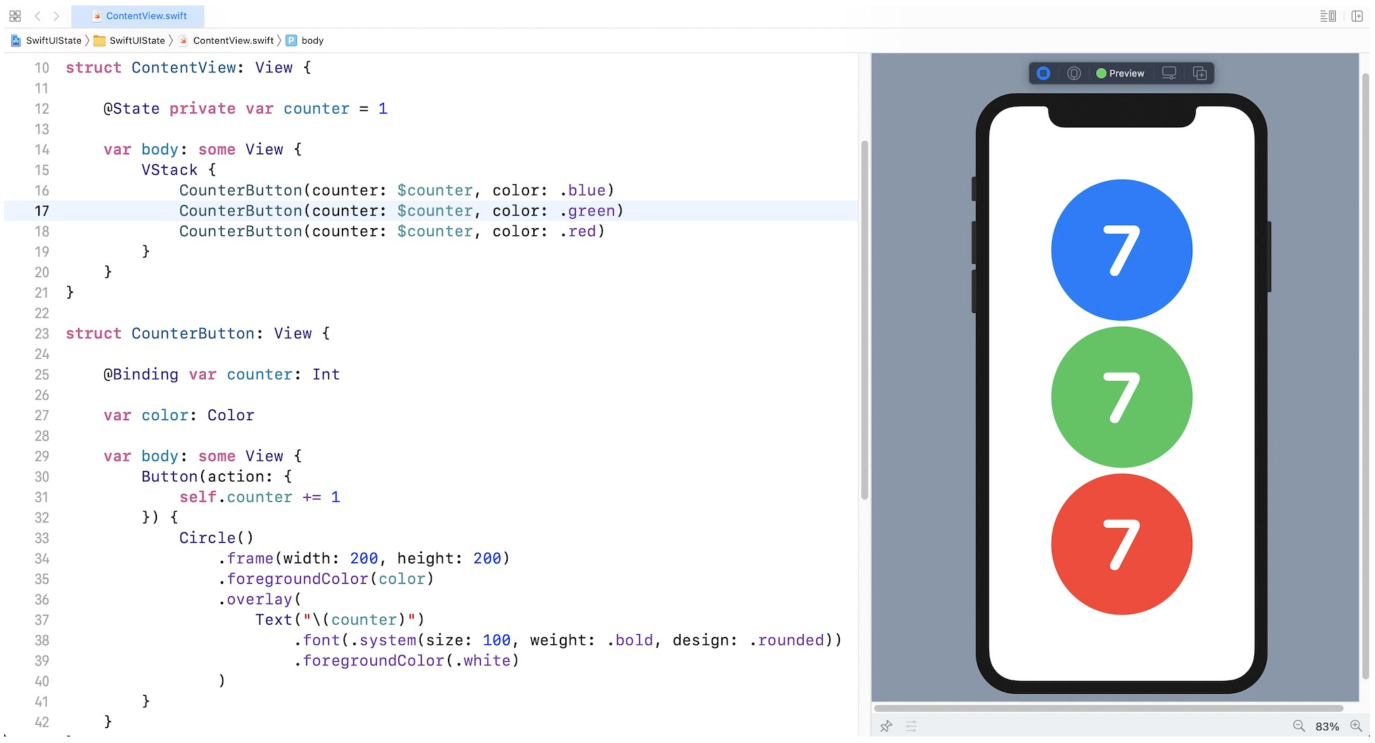
Figure 8. Understanding Binding

So, what's the `$` sign? In SwiftUI, you use the $ prefix operator to get the binding from a state variable.

Now that you understand how binding works, you can continue to create the other two buttons and align them vertically using `VStack` like this:

```
struct ContentView: View {

    @State private var counter = 1

    var body: some View {
        VStack {
            CounterButton(counter: $counter, color: .blue)
            CounterButton(counter: $counter, color: .green)
            CounterButton(counter: $counter, color: .red)
        }
    }
}
```

After the changes, Run the app to test it. Tapping any of the buttons will increase the count by one.



```
struct ContentView: View {

    @State private var counter = 1

    var body: some View {
        VStack {
            CounterButton(counter: $counter, color: .blue)
            CounterButton(counter: $counter, color: .green)
            CounterButton(counter: $counter, color: .red)
        }
    }
}

struct CounterButton: View {

    @Binding var counter: Int

    var color: Color

    var body: some View {
        Button(action: {
            self.counter += 1
        }) {
            Circle()
                .frame(width: 200, height: 200)
                .foregroundColor(color)
                .overlay(
                    Text("\(counter)")
                        .font(.system(size: 100, weight: .bold, design: .rounded))
                        .foregroundColor(.white)
                )
        }
    }
```

Figure 9. Testing the three counter buttons

Exercise #2

Presently, all the buttons share the same count. For this exercise, you are required to modify the code such that each of the button has its own counter. When the user taps the blue button, the app only increases the counter of the blue button by 1. In addition, you will need to provide a master counter that sums up the counter of all buttons. Figure 10 shows the sample layout for the exercise..

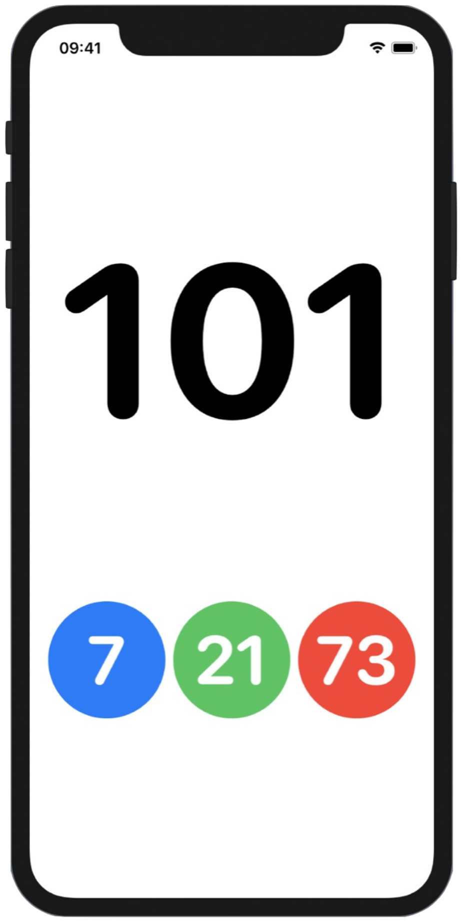


Figure 10. Each button has its own counter

Summary

The support of State in SwiftUI simplifies state management in application development. It's important you understand what `@State` and `@Binding` mean because they play a big part in SwiftUI for managing states and UI updates. This chapter kicks off the basics of state management in SwiftUI. Later, you will learn more about how we can utilize `@State` in view animation and how to manage shared states among multiple views.

For reference, you can download the sample state project below:

- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUICounter.zip)
- Exercise (https://www.appcoda.com/resources/swiftui2/SwiftUIMasterCounter.zip)

Chapter 8
Implementing Path and Shape for Line Drawing and Pie Charts

For experienced developers, you probably have used the Core Graphics APIs to draw shapes and objects. It's a very powerful framework for you to create vector-based drawings. SwiftUI also provides several vector drawing APIs for developers to draw lines and shapes.

In this chapter, you will learn how to draw lines, arcs, pie charts, and donut charts using `Path` and the built-in `Shape` such as `Circle` and `RoundedRectangle`. Here are the topics we'll cover:

- Understanding Path and how to draw a line
- What is the `Shape` protocol and how to draw a custom shape by conforming to the protocol
- How to draw a pie chart
- How to create a progress indicator with an open circle
- How to draw a donut chart

Figure 1 shows you some of the shapes and charts that we will create in this chapter.

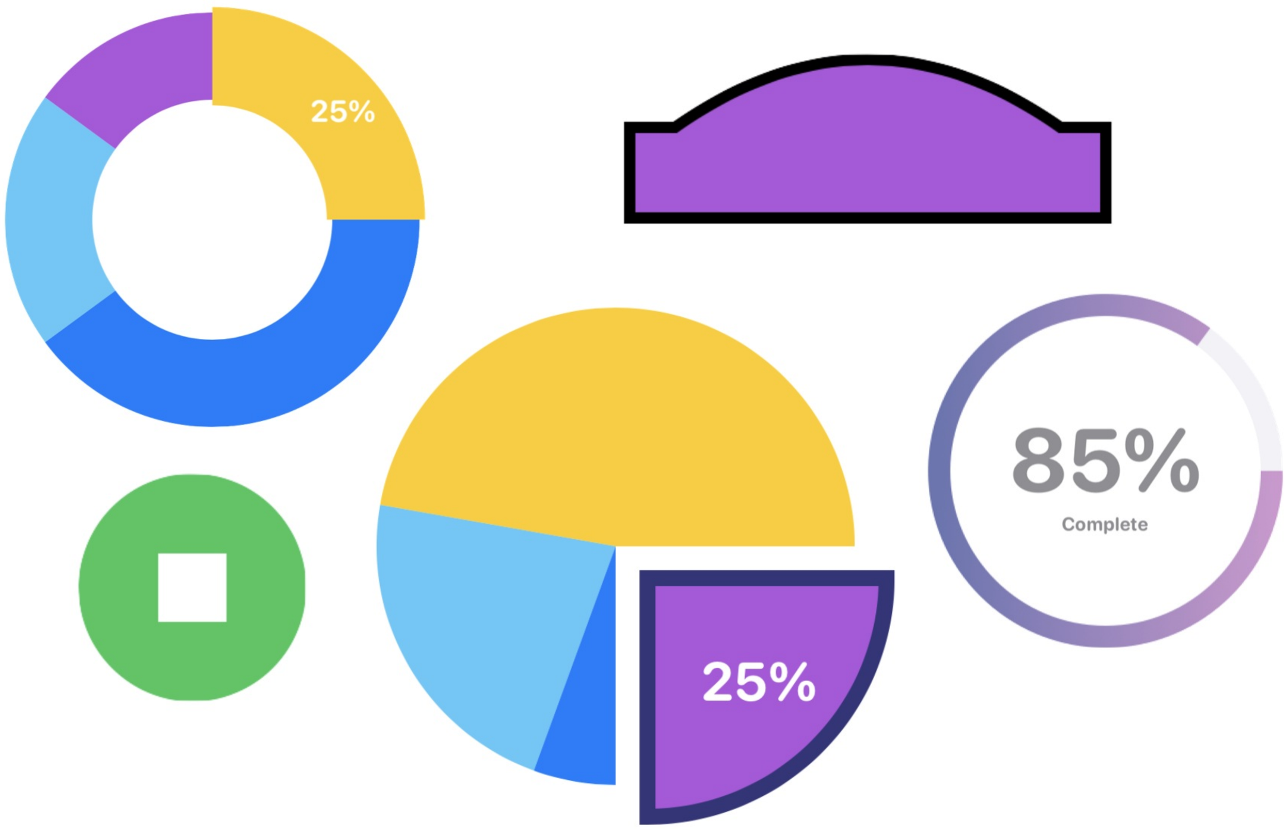


Figure 1. Sample shapes and charts

Understanding Path

In SwiftUI, you draw lines and shapes using `Path`. If you refer to Apple's documentation (https://developer.apple.com/documentation/swiftui/path), `Path` is a struct containing the outline of a 2D shape. Basically, a path is the setting of a point of origin, then drawning lines from point to point. Let me give you an example. Take a look at figure 2. We will walk thorugh how this rectangle is drawn.

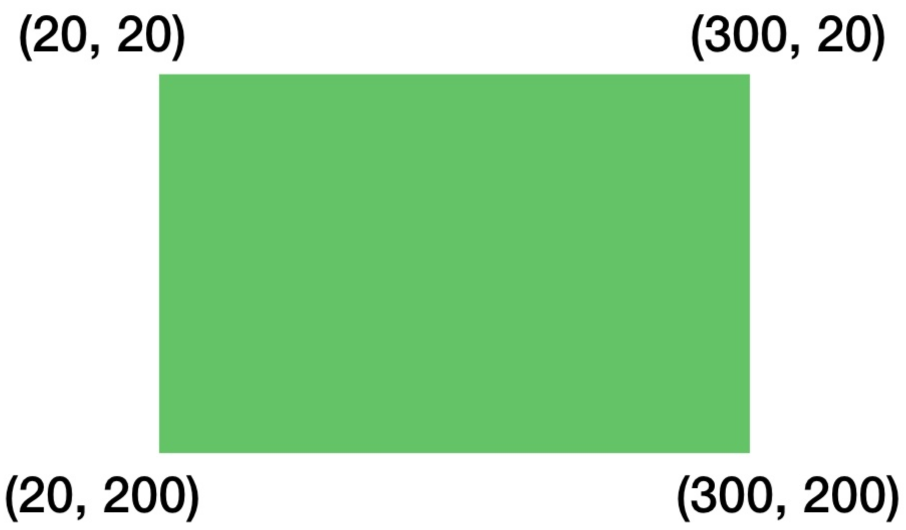


Figure 2. A rectange with coordinates

If you were to verbally tell me how you would draw the rectangle step by step, you would probably provide the following description:

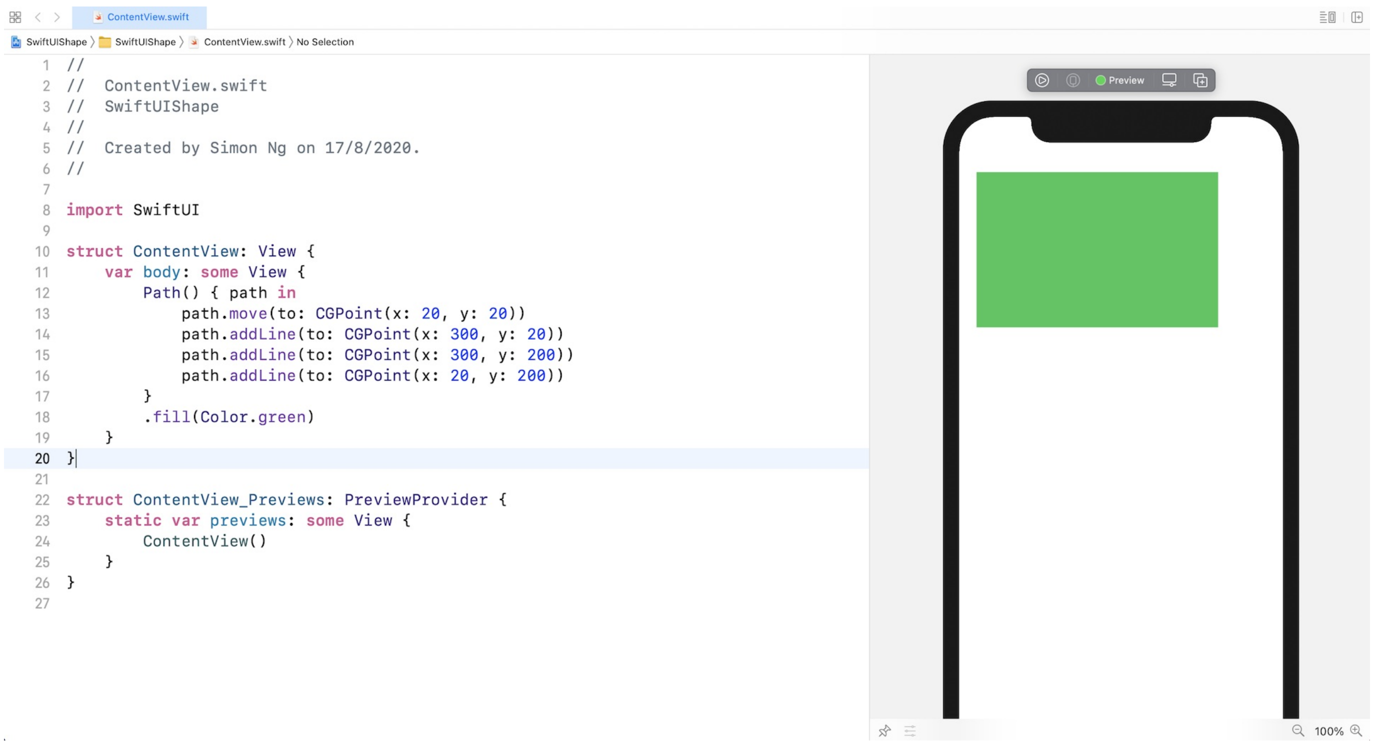
1. Move the point (20, 20)
2. Draw a line from (20, 20) to (300, 20)
3. Draw a line from (300, 20) to (300, 200)
4. Draw a line from (300, 200) to (20, 200)
5. Fill the whole area in green

That's how `Path` is works! Let's write your verbal description in code:

```
Path() { path in
    path.move(to: CGPoint(x: 20, y: 20))
    path.addLine(to: CGPoint(x: 300, y: 20))
    path.addLine(to: CGPoint(x: 300, y: 200))
    path.addLine(to: CGPoint(x: 20, y: 200))
}
.fill(Color.green)
```

You initialize a `Path` and provide detailed instructions in the closure. You call the `move(to:)` method to move to a particular coordinate. To draw a line from the current point to a specific point, you call the `addLine(to:)` method. By default, iOS fills the path with the default foreground color, which is black. To fill it with a different color, you can use the `.fill` modifier and set a different color.

Test the code by creating a new project using the *App* template. Name the project `SwiftUIShape` (or whatever name you like) and then type the above code snippet in the `body`. The preview canvas should display a rectangle in green.



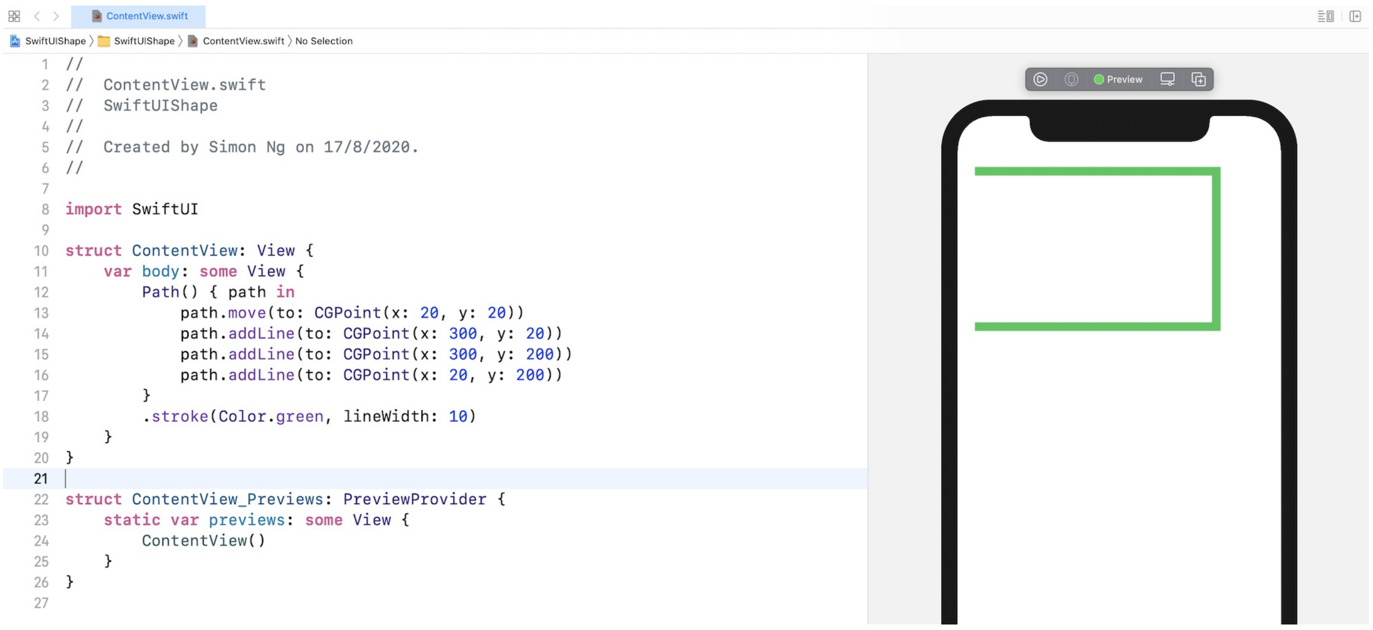
ContentView.swift
SwiftUIShape 〉 SwiftUIShape 〉 ContentView.swift 〉 No Selection
Preview
100%

```
//
//  ContentView.swift
//  SwiftUIShape
//
//  Created by Simon Ng on 17/8/2020.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        Path() { path in
            path.move(to: CGPoint(x: 20, y: 20))
            path.addLine(to: CGPoint(x: 300, y: 20))
            path.addLine(to: CGPoint(x: 300, y: 200))
            path.addLine(to: CGPoint(x: 20, y: 200))
        }
        .fill(Color.green)
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 3. Drawing a rectangle using Path

Using Stroke to Draw Borders

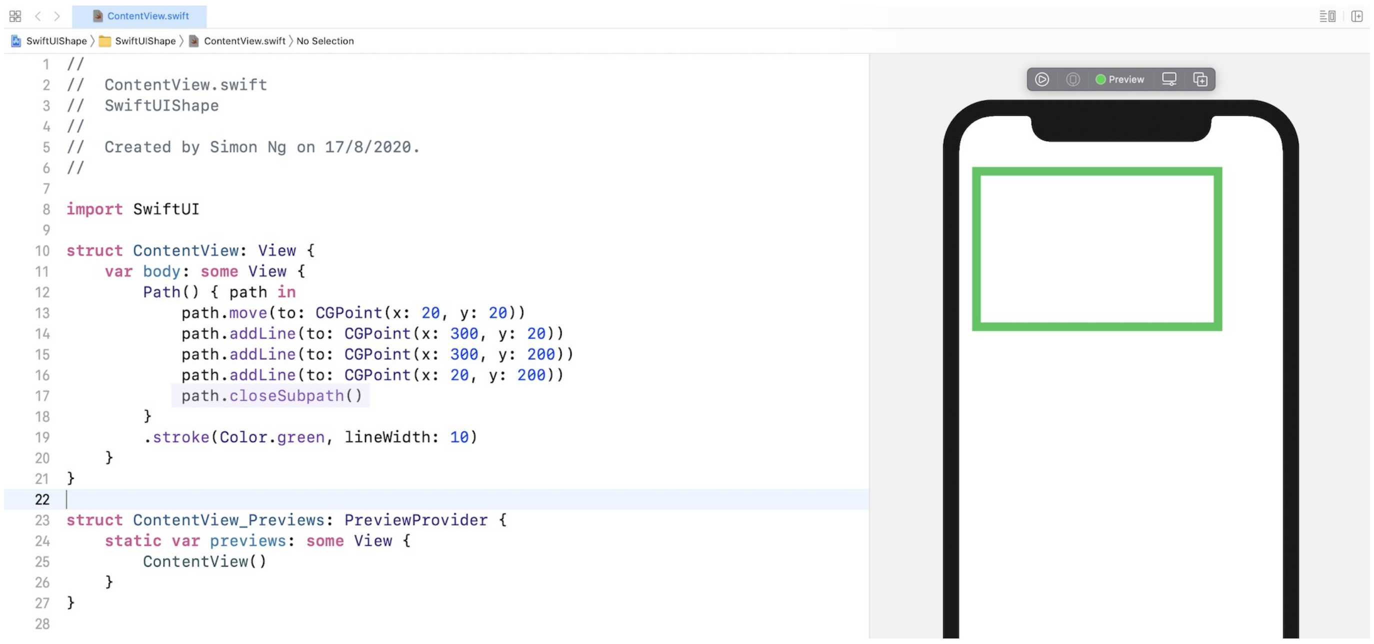
You're not required to fill the whole area with color. If you just want to draw the lines, you can use the `.stroke` modifier and specify the line width and color. Figure 4 shows the result.


ContentView.swift
SwiftUIShape 〉 SwiftUIShape 〉 ContentView.swift 〉 No Selection
Preview

```swift
//
//  ContentView.swift
//  SwiftUIShape
//
//  Created by Simon Ng on 17/8/2020.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        Path() { path in
            path.move(to: CGPoint(x: 20, y: 20))
            path.addLine(to: CGPoint(x: 300, y: 20))
            path.addLine(to: CGPoint(x: 300, y: 200))
            path.addLine(to: CGPoint(x: 20, y: 200))
        }
        .stroke(Color.green, lineWidth: 10)
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 4. Drawing the lines with stroke

Because we didn't specify a step to draw the line to the point of origin, it shows an open-ended path. To close the path, you can call the `closeSubpath()` method at the end of the `Path` closure, that will automatically connect the current point with the point of origin.


ContentView.swift
SwiftUIShape 〉 SwiftUIShape 〉 ContentView.swift 〉 No Selection
Preview

```swift
//
//  ContentView.swift
//  SwiftUIShape
//
//  Created by Simon Ng on 17/8/2020.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        Path() { path in
            path.move(to: CGPoint(x: 20, y: 20))
            path.addLine(to: CGPoint(x: 300, y: 20))
            path.addLine(to: CGPoint(x: 300, y: 200))
            path.addLine(to: CGPoint(x: 20, y: 200))
            path.closeSubpath()
        }
        .stroke(Color.green, lineWidth: 10)
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 5. Closing the path with closeSubpath()

Drawing Curves

`Path` provides several built-in APIs to help you draw different shapes. You are not limited to drawing straight lines. The `addQuadCurve`, `addCurve`, and `addArc` allow you to create curves and arcs. Let's say, you want to draw a dome on top of a rectangle like that shown in figure 6.

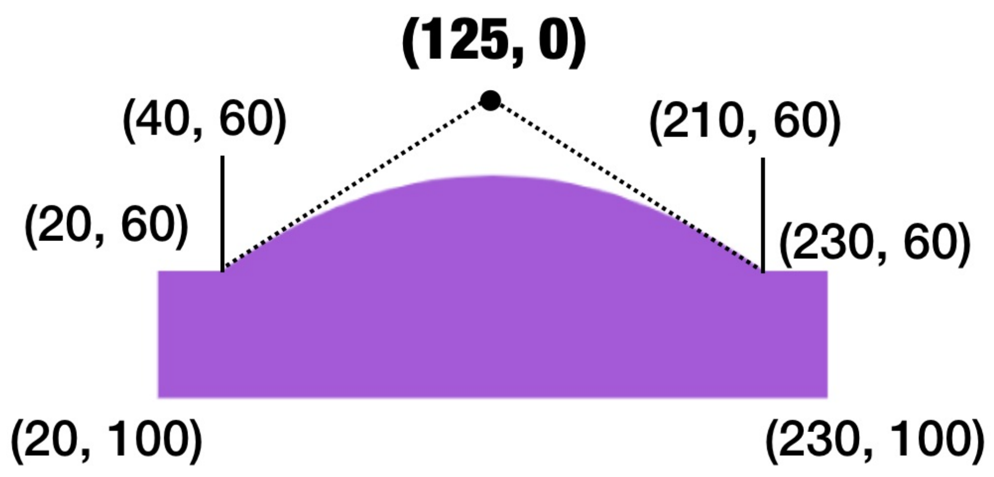


Figure 6. A dome with a rectangle bottom

You write the code like this:

```
Path() { path in
    path.move(to: CGPoint(x: 20, y: 60))
    path.addLine(to: CGPoint(x: 40, y: 60))
    path.addQuadCurve(to: CGPoint(x: 210, y: 60), control: CGPoint(x: 125, y: 0))
    path.addLine(to: CGPoint(x: 230, y: 60))
    path.addLine(to: CGPoint(x: 230, y: 100))
    path.addLine(to: CGPoint(x: 20, y: 100))
}
.fill(Color.purple)
```

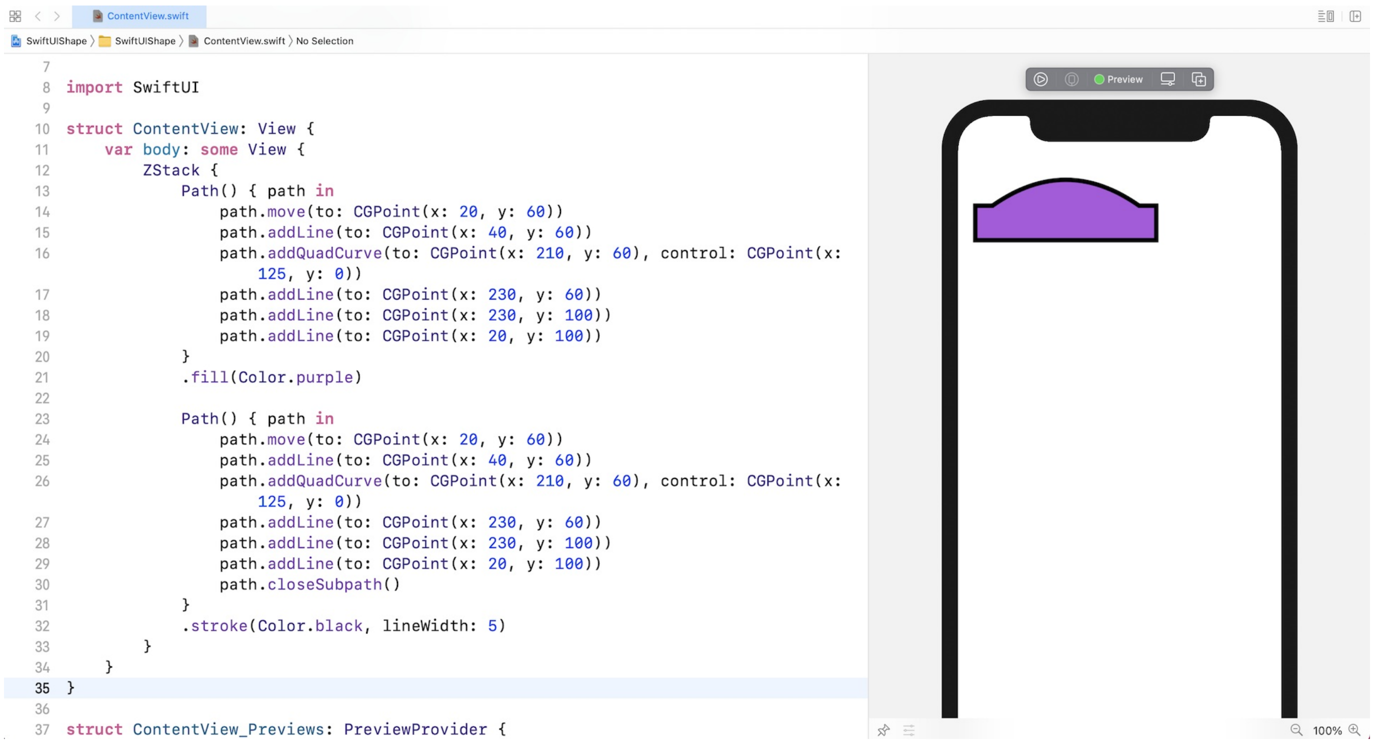
The `addQuadCurve` method lets you draw a curve by defining a control point. Referring to figure 6, (40, 60) and (210, 60) are known as anchor points. (125, 0) is the control point, which is calculated to create the dome shape. I'm not going to discuss the mathematics involved in drawing the curve, however, try to change the value of the control point to see its effect. In brief, this control point controls how the curve is drawn. If it's placed closer to the top of the rectangle (e.g. 125, 30), you will create a less rounded appearance.

Fill and Stroke

What if you want to draw the border of the shape and fill the shape with color at the same time? The `fill` and `stroke` modifiers cannot be used in parallel. You can make use of `ZStack` to achieve the same effect. Here is the code:

```
ZStack {
    Path() { path in
        path.move(to: CGPoint(x: 20, y: 60))
        path.addLine(to: CGPoint(x: 40, y: 60))
        path.addQuadCurve(to: CGPoint(x: 210, y: 60), control: CGPoint(x: 125, y: 0
))
        path.addLine(to: CGPoint(x: 230, y: 60))
        path.addLine(to: CGPoint(x: 230, y: 100))
        path.addLine(to: CGPoint(x: 20, y: 100))
    }
    .fill(Color.purple)

    Path() { path in
        path.move(to: CGPoint(x: 20, y: 60))
        path.addLine(to: CGPoint(x: 40, y: 60))
        path.addQuadCurve(to: CGPoint(x: 210, y: 60), control: CGPoint(x: 125, y: 0
))
        path.addLine(to: CGPoint(x: 230, y: 60))
        path.addLine(to: CGPoint(x: 230, y: 100))
        path.addLine(to: CGPoint(x: 20, y: 100))
        path.closeSubpath()
    }
    .stroke(Color.black, lineWidth: 5)
}
```

We create two `Path` objects with the same path and overlay one on top of the other using `ZStack`. The one underneath uses `fill` to fill the dome rectangle with purple color. The one overlayed on top only draws the borders with black color. Figure 7 shows the result.


ContentView.swift
SwiftUIShape 〉 SwiftUIShape 〉 ContentView.swift 〉 No Selection
Preview
100%

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        ZStack {
            Path() { path in
                path.move(to: CGPoint(x: 20, y: 60))
                path.addLine(to: CGPoint(x: 40, y: 60))
                path.addQuadCurve(to: CGPoint(x: 210, y: 60), control: CGPoint(x:
                    125, y: 0))
                path.addLine(to: CGPoint(x: 230, y: 60))
                path.addLine(to: CGPoint(x: 230, y: 100))
                path.addLine(to: CGPoint(x: 20, y: 100))
            }
            .fill(Color.purple)

            Path() { path in
                path.move(to: CGPoint(x: 20, y: 60))
                path.addLine(to: CGPoint(x: 40, y: 60))
                path.addQuadCurve(to: CGPoint(x: 210, y: 60), control: CGPoint(x:
                    125, y: 0))
                path.addLine(to: CGPoint(x: 230, y: 60))
                path.addLine(to: CGPoint(x: 230, y: 100))
                path.addLine(to: CGPoint(x: 20, y: 100))
                path.closeSubpath()
            }
            .stroke(Color.black, lineWidth: 5)
        }
    }
}

struct ContentView_Previews: PreviewProvider {
```

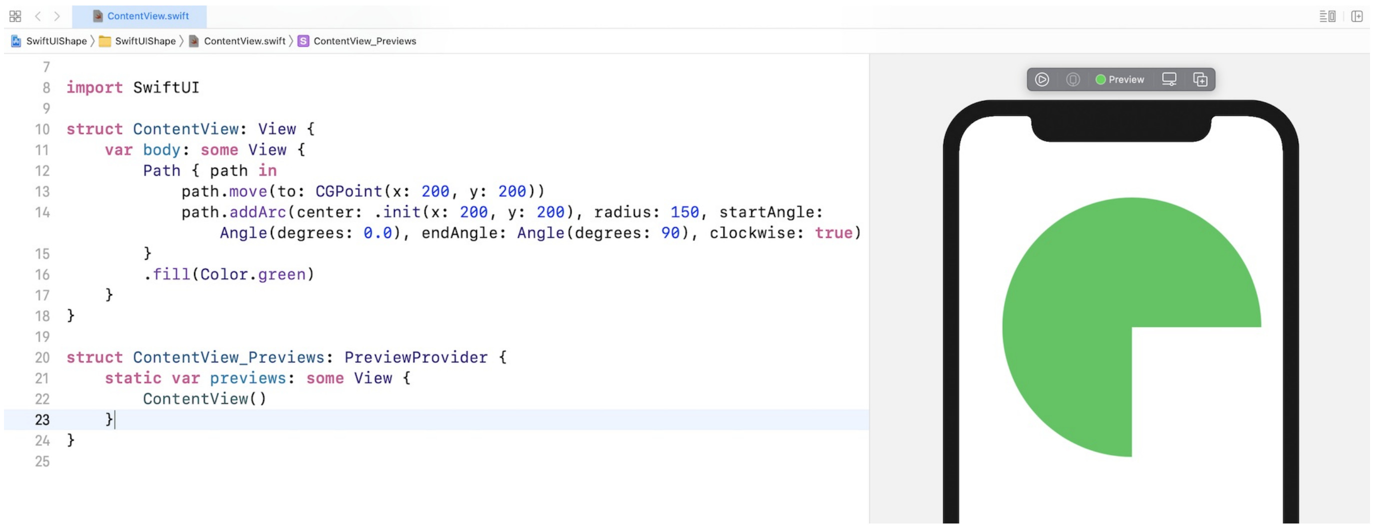
Figure 7. A dome rectange with borders

Drawing Arcs and Pie Charts

SwiftUI provides a convenient API for developers to draw arcs. This API is incredibly useful to compose various shapes and objects including pie charts. To draw an arc, you write the code like this:

```
Path { path in
    path.move(to: CGPoint(x: 200, y: 200))
    path.addArc(center: .init(x: 200, y: 200), radius: 150, startAngle: Angle(degr
ees: 0.0), endAngle: Angle(degrees: 90), clockwise: true)
}
.fill(Color.green)
```

Enter this code in the body, you will see an arc that fills with green color in the preview canvas.


ContentView.swift
SwiftUIShape 〉 SwiftUIShape 〉 ContentView.swift 〉 ContentView_Previews
Preview

```swift
import SwiftUI

struct ContentView: View {
    var body: some View {
        Path { path in
            path.move(to: CGPoint(x: 200, y: 200))
            path.addArc(center: .init(x: 200, y: 200), radius: 150, startAngle:
                Angle(degrees: 0.0), endAngle: Angle(degrees: 90), clockwise: true)
        }
        .fill(Color.green)
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 8. A sample arc

In the code, we first move to the starting point (200, 200). Then we call `addArc` to create the arc. The `addArc` method accepts several parameters:

- **center** - the center point of the circle
- **radius** - the radius of the circle for creating the arc
- **startAngle** - the starting angle of the arc
- **endAngle** - the ending angle of the arc
- **clockwise** - the direction to draw the arc

If you just look at the name of the *startAngle* and *endAngle* parameters, you might be a bit confused with their meaning. Figure 9 will give you a better idea of how these parameters work.

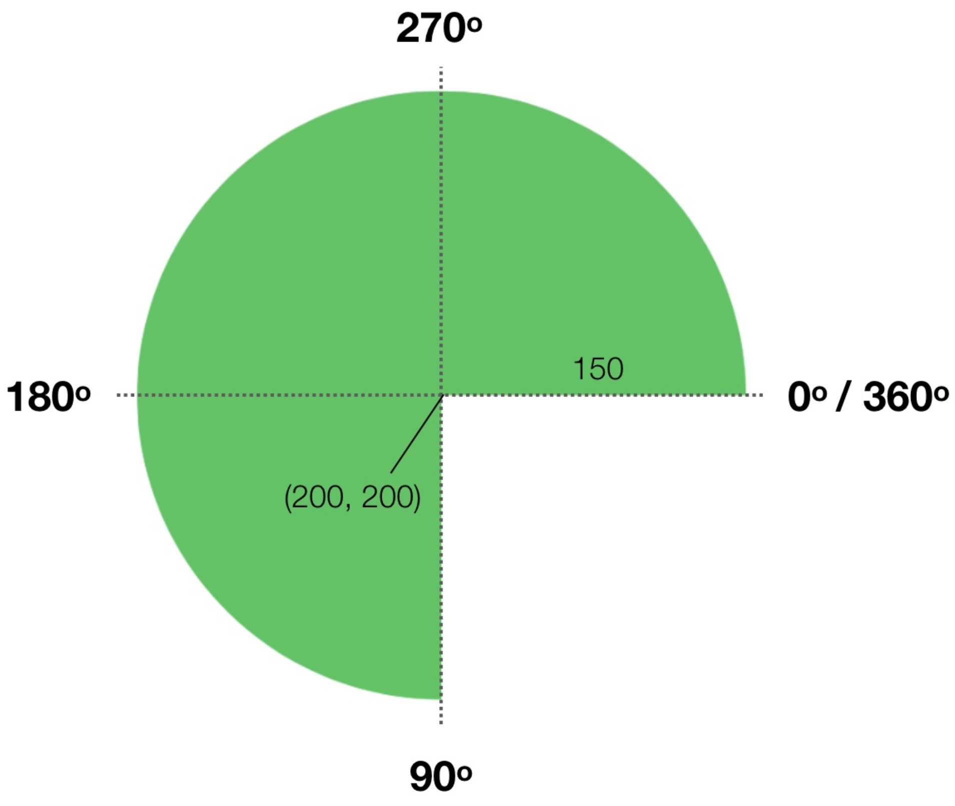


Figure 9. Understanding starting and end angle

By using `addArc`, you can easily create a pie chart with different colored segments. All you need to do is overlay different pie segments with `ZStack`. Each segment has different values for `startAngle` and `endAngle` to compose the chart. Here is an example:

```
ZStack {
    Path { path in
        path.move(to: CGPoint(x: 187, y: 187))
        path.addArc(center: .init(x: 187, y: 187), radius: 150, startAngle: Angle(
degrees: 0.0), endAngle: Angle(degrees: 190), clockwise: true)
    }
    .fill(Color(.systemYellow))

    Path { path in
        path.move(to: CGPoint(x: 187, y: 187))
        path.addArc(center: .init(x: 187, y: 187), radius: 150, startAngle: Angle(
degrees: 190), endAngle: Angle(degrees: 110), clockwise: true)
    }
    .fill(Color(.systemTeal))

    Path { path in
        path.move(to: CGPoint(x: 187, y: 187))
        path.addArc(center: .init(x: 187, y: 187), radius: 150, startAngle: Angle(
degrees: 110), endAngle: Angle(degrees: 90), clockwise: true)
    }
    .fill(Color(.systemBlue))

    Path { path in
        path.move(to: CGPoint(x: 187, y: 187))
        path.addArc(center: .init(x: 187, y: 187), radius: 150, startAngle: Angle(
degrees: 90.0), endAngle: Angle(degrees: 360), clockwise: true)
    }
    .fill(Color(.systemPurple))

}
```

This will render a pie chart with 4 segments. If you want to have more segments, just create additional path objects with different angle values. As a side note, the color I used comes from the standard color objects provided in iOS 13 (or later). You can check out the full set of color objects at
https://developer.apple.com/documentation/uikit/uicolor/standard_colors.

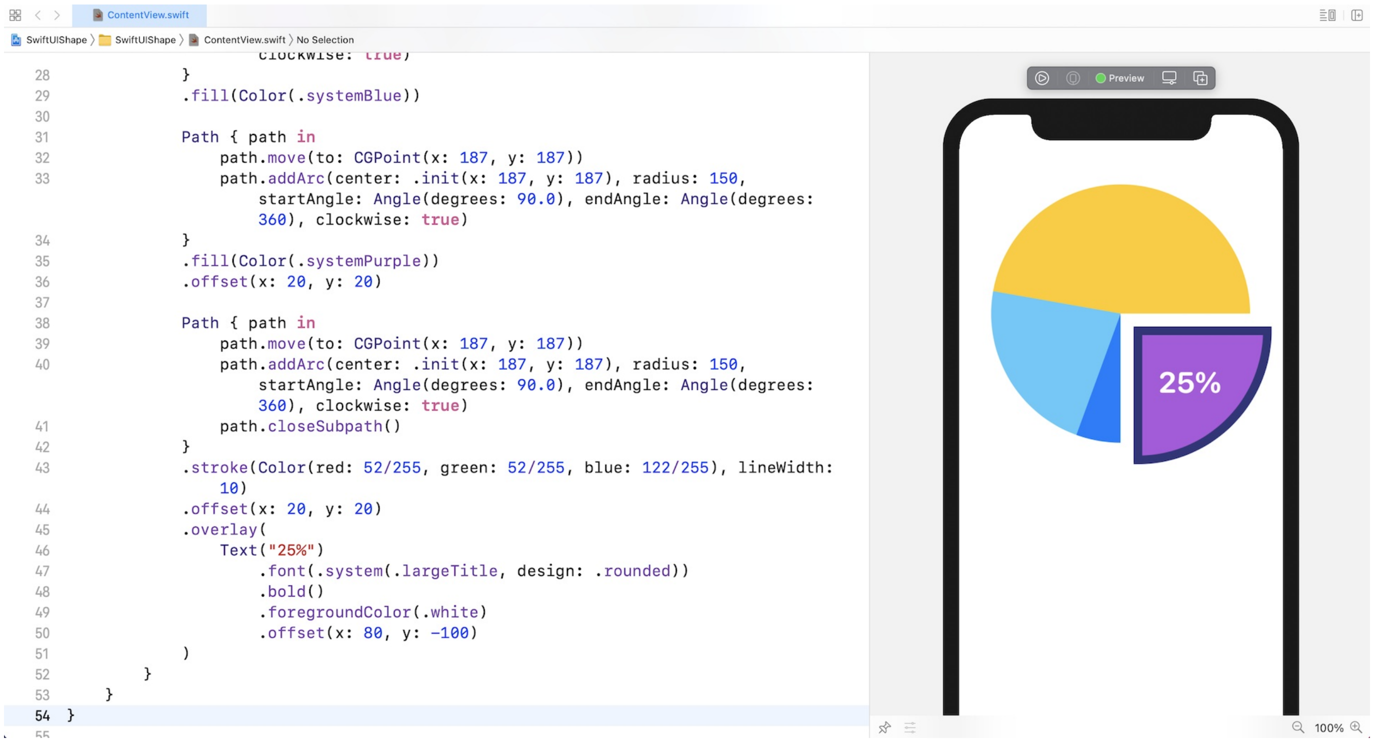
Sometimes, you may want to highlight a particular segment by splitting it from the pie chart. Say, to highlight the segment in purple, you can apply the `offset` modifier to re-position the segment:

```
Path { path in
    path.move(to: CGPoint(x: 187, y: 187))
    path.addArc(center: .init(x: 187, y: 187), radius: 150, startAngle: Angle(degr
ees: 90.0), endAngle: Angle(degrees: 360), clockwise: true)
}
.fill(Color(.systemPurple))
.offset(x: 20, y: 20)
```

Optionally, you can overlay a border to further catch people's attention. If you want to add a label to the highlighted segment, you can also overlay a `Text` view like this:

```
Path { path in
    path.move(to: CGPoint(x: 187, y: 187))
    path.addArc(center: .init(x: 187, y: 187), radius: 150, startAngle: Angle(degr
ees: 90.0), endAngle: Angle(degrees: 360), clockwise: true)
    path.closeSubpath()
}
.stroke(Color(red: 52/255, green: 52/255, blue: 122/255), lineWidth: 10)
.offset(x: 20, y: 20)
.overlay(
    Text("25%")
        .font(.system(.largeTitle, design: .rounded))
        .bold()
        .foregroundColor(.white)
        .offset(x: 80, y: -100)
)
```

This path has the same starting and end angle as the purple segment, however; it only draws the border and adds a text view in order to make the segment stand out. Figure 10 shows the end result.


ContentView.swift
SwiftUIShape 〉 SwiftUIShape 〉 ContentView.swift 〉 No Selection
Preview
25%
100%

```
                    clockwise: true)
            }
            .fill(Color(.systemBlue))

            Path { path in
                path.move(to: CGPoint(x: 187, y: 187))
                path.addArc(center: .init(x: 187, y: 187), radius: 150,
                    startAngle: Angle(degrees: 90.0), endAngle: Angle(degrees:
                    360), clockwise: true)
            }
            .fill(Color(.systemPurple))
            .offset(x: 20, y: 20)

            Path { path in
                path.move(to: CGPoint(x: 187, y: 187))
                path.addArc(center: .init(x: 187, y: 187), radius: 150,
                    startAngle: Angle(degrees: 90.0), endAngle: Angle(degrees:
                    360), clockwise: true)
                path.closeSubpath()
            }
            .stroke(Color(red: 52/255, green: 52/255, blue: 122/255), lineWidth:
                10)
            .offset(x: 20, y: 20)
            .overlay(
                Text("25%")
                    .font(.system(.largeTitle, design: .rounded))
                    .bold()
                    .foregroundColor(.white)
                    .offset(x: 80, y: -100)
            )
        }
    }
}
```

Figure 10. A pie chart with a highlighted segment

Understanding the Shape Protocol

Before we look into the `Shape` protocol, let's begin with a simple exercise. Based on what you have learned, draw the following shape with `Path`.

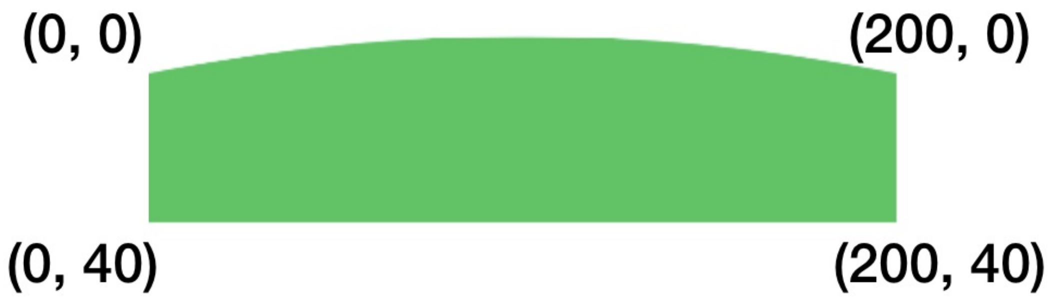


Figure 11. Your exercise

Don't look at the solution yet. Try to build one by yourself.

Okay, to build this shape, you create a `Path` using `addLine` and `addQuadCurve`:

```
Path() { path in
    path.move(to: CGPoint(x: 0, y: 0))
    path.addQuadCurve(to: CGPoint(x: 200, y: 0), control: CGPoint(x: 100, y: -20))
    path.addLine(to: CGPoint(x: 200, y: 40))
    path.addLine(to: CGPoint(x: 200, y: 40))
    path.addLine(to: CGPoint(x: 0, y: 40))
}
.fill(Color.green)
```

If you've read the documentation for `Path`, you may find another function called `addRect`, which lets you draw a rectangle with a specific width and height. Let's use it to create the same shape:

```
Path() { path in
    path.move(to: CGPoint(x: 0, y: 0))
    path.addQuadCurve(to: CGPoint(x: 200, y: 0), control: CGPoint(x: 100, y: -20))
    path.addRect(CGRect(x: 0, y: 0, width: 200, height: 40))
}
.fill(Color.green)
```

Let's talk about the `Shape` protocol. The protocol is very simple with only one requirement. To adopt it, you must implement the following function:

```
func path(in rect: CGRect) -> Path
```

When is it useful to adopt the `Shape` protocol? To answer this, let's say you want to create a button with the dome shape but flexible size. Is it possible to reuse the `Path` that you have just created?

Take a look at the code above again. You created a path with absolute coordinates and size. In order to create the same shape but with variable size, you can create a struct to adopt the `Shape` protocol and implement the `path(in:)` function. When the `path(in:)`

function is called by the framework, you will be given the `rect` size. You can then draw the path within that `rect`.

In the code that follows we create the `Dome` shape using a `path(in:)` function.

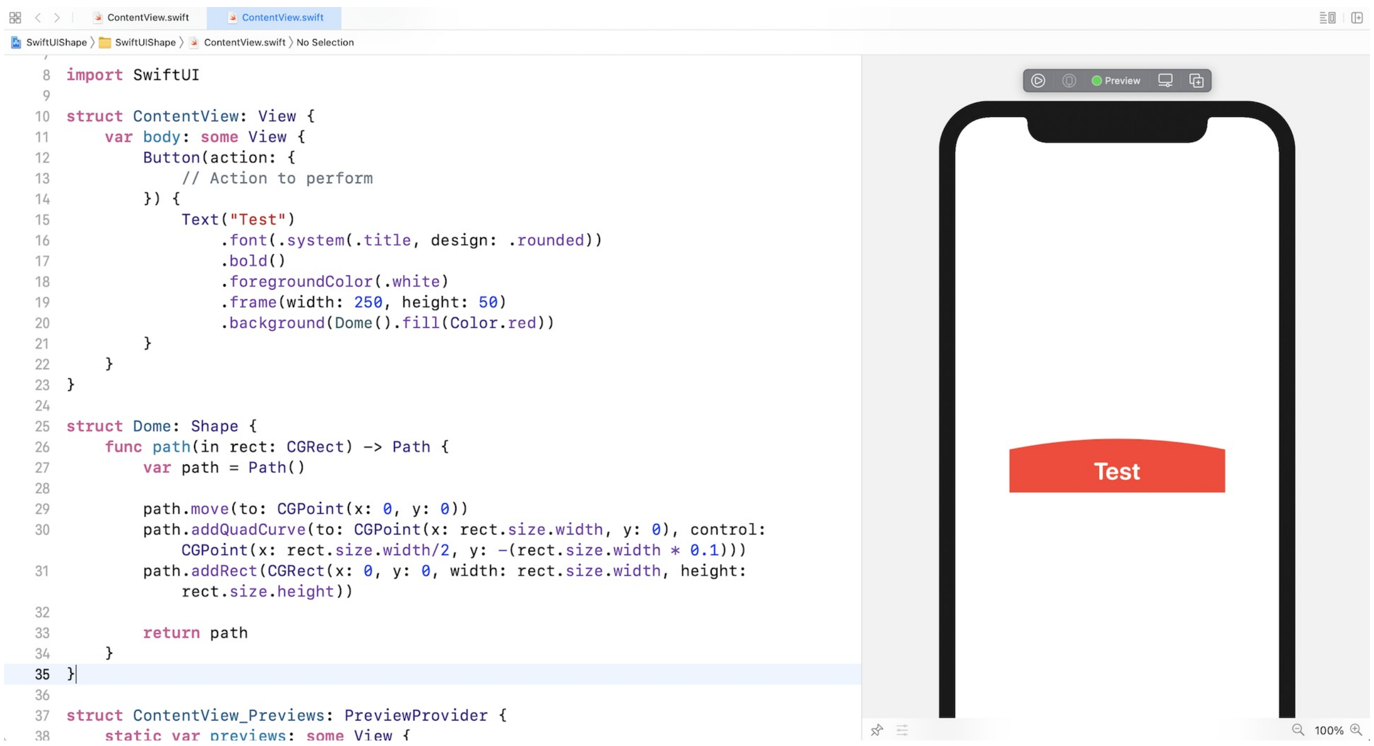
```
struct Dome: Shape {
    func path(in rect: CGRect) -> Path {
        var path = Path()

        path.move(to: CGPoint(x: 0, y: 0))
        path.addQuadCurve(to: CGPoint(x: rect.size.width, y: 0), control: CGPoint(
x: rect.size.width/2, y: -(rect.size.width * 0.1)))
        path.addRect(CGRect(x: 0, y: 0, width: rect.size.width, height: rect.size.
height))

        return path
    }
}
```

By adopting the protocol, we are given the rectangular area for drawing the path. From the `rect`, we get the width and height of the rectangular area to compute the control point and draw the rectangle base.

With the dynamic shape, you can create various SwiftUI controls. For example, you can create a button with the `Dome` shape like this:

```
Button(action: {
    // Action to perform
}) {
    Text("Test")
        .font(.system(.title, design: .rounded))
        .bold()
        .foregroundColor(.white)
        .frame(width: 250, height: 50)
        .background(Dome().fill(Color.red))
}
```

We apply the `Dome` shape as the background of the button. Its width and height are based on the specified frame size.



```swift
import SwiftUI

struct ContentView: View {
    var body: some View {
        Button(action: {
            // Action to perform
        }) {
            Text("Test")
                .font(.system(.title, design: .rounded))
                .bold()
                .foregroundColor(.white)
                .frame(width: 250, height: 50)
                .background(Dome().fill(Color.red))
        }
    }
}

struct Dome: Shape {
    func path(in rect: CGRect) -> Path {
        var path = Path()

        path.move(to: CGPoint(x: 0, y: 0))
        path.addQuadCurve(to: CGPoint(x: rect.size.width, y: 0), control:
            CGPoint(x: rect.size.width/2, y: -(rect.size.width * 0.1)))
        path.addRect(CGRect(x: 0, y: 0, width: rect.size.width, height:
            rect.size.height))

        return path
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
```

Figure 12. Creating a button with the Dome shape

Using the Built-in Shapes

Earlier, we built a custom shape by using the `Shape` protocol. SwiftUI actually comes with several built-in shapes including `Circle` , `Rectangle` , `RoundedRectangle` , `Ellipse` , etc. If you don't need anything fancy, these shapes are good enough for you to create common objects.

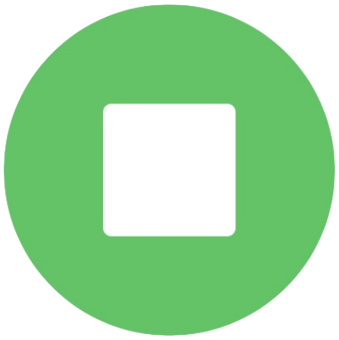


Figure 13. A stop button

Let's say, you want to create a stop button like the one shown in figure 13. It's composed of a rounded rectangle and a circle. You can write the code like this:

```
Circle()
    .foregroundColor(.green)
    .frame(width: 200, height: 200)
    .overlay(
        RoundedRectangle(cornerRadius: 5)
            .frame(width: 80, height: 80)
            .foregroundColor(.white)
    )
```

Here, we initialize a `Circle` view and then overlay a `RoundedRectangle` view on it.

Creating a Progress Indicator Using Shapes

By mixing and matching the built-in shapes, you can create various types of vector-based UI controls for your applications. Let me show you another example. Figure 14 shows you a progress indicator that can be built by using `Circle` .

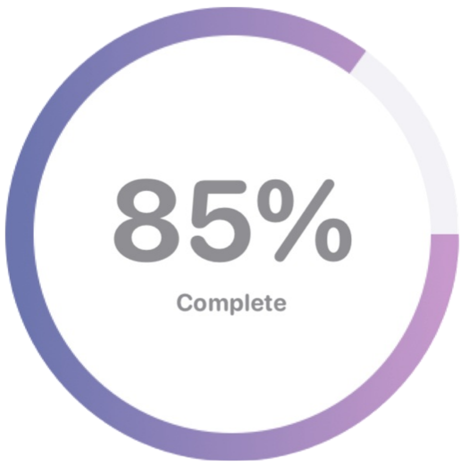
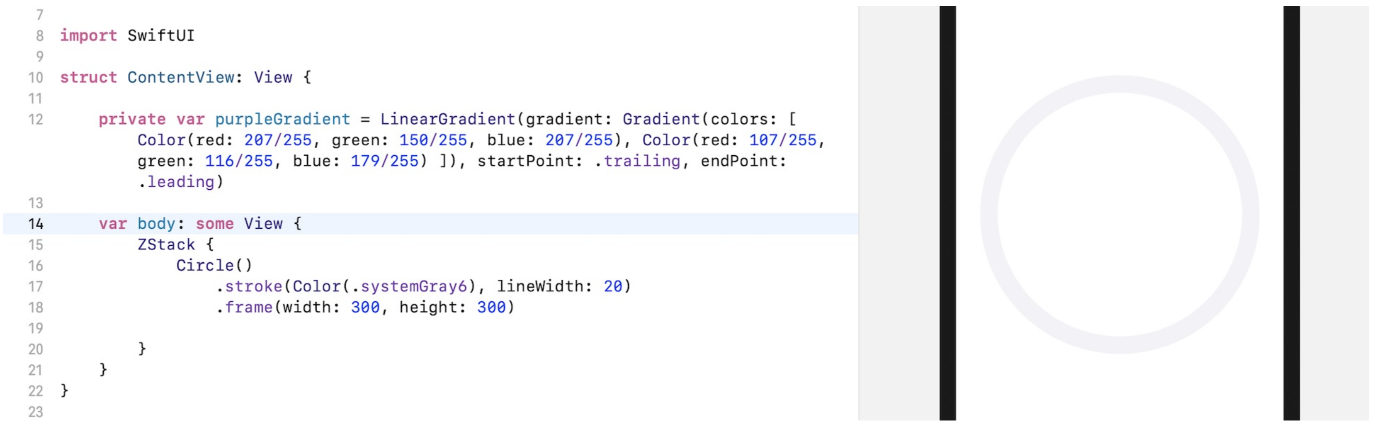


Figure 14. A progress indicator

This progress indicator is actually composed of two circles. We have a gray outline of a circle underneath. On top of the grey circle, is an open outline of a circle indicating the completion progress. In your project, write the code in `ContentView` like this:

```
struct ContentView: View {

    private var purpleGradient = LinearGradient(gradient: Gradient(colors: [ Color
(red: 207/255, green: 150/255, blue: 207/255), Color(red: 107/255, green: 116/255,
 blue: 179/255) ]), startPoint: .trailing, endPoint: .leading)

    var body: some View {

        ZStack {
            Circle()
                .stroke(Color(.systemGray6), lineWidth: 20)
                .frame(width: 300, height: 300)

        }
    }
}
```

We use the `stroke` modifier to draw the outline of the grey circle. You may adjust the value of the `lineWidth` parameter if you prefer thicker (or thinner) lines. The `purpleGradient` property defines the purple gradient that we will use later in drawing the open circle.



```
import SwiftUI

struct ContentView: View {

    private var purpleGradient = LinearGradient(gradient: Gradient(colors: [
        Color(red: 207/255, green: 150/255, blue: 207/255), Color(red: 107/255,
        green: 116/255, blue: 179/255) ]), startPoint: .trailing, endPoint:
        .leading)

    var body: some View {
        ZStack {
            Circle()
                .stroke(Color(.systemGray6), lineWidth: 20)
                .frame(width: 300, height: 300)

        }
    }
}
```

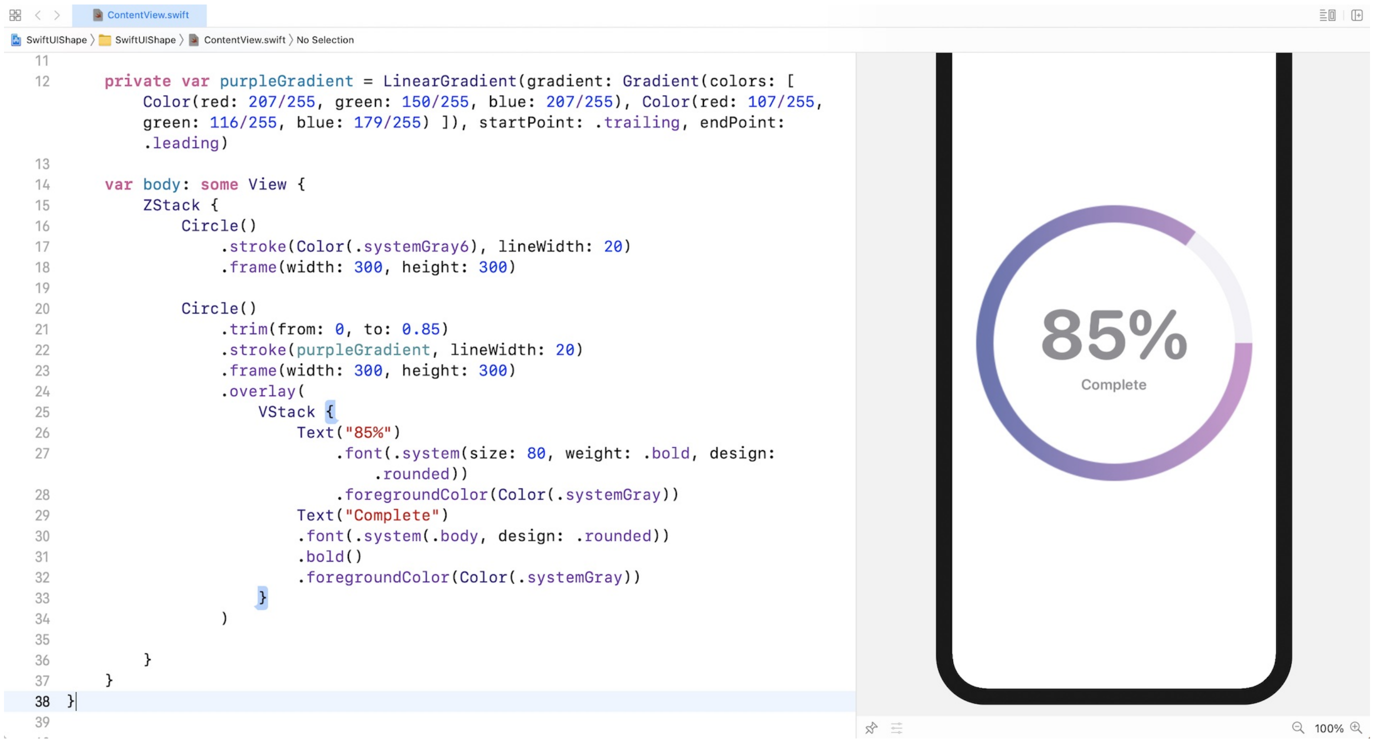
Figure 15. Drawing a gray circle

Now, insert the following code in `ZStack` to create the open circle:

```
Circle()
    .trim(from: 0, to: 0.85)
    .stroke(purpleGradient, lineWidth: 20)
    .frame(width: 300, height: 300)
    .overlay(
        VStack {
            Text("85%")
                .font(.system(size: 80, weight: .bold, design: .rounded))
                .foregroundColor(Color(.systemGray))
            Text("Complete")
            .font(.system(.body, design: .rounded))
            .bold()
            .foregroundColor(Color(.systemGray))
        }
    )
```

To create an open circle, add the `trim` modifier. You specify a `from` value and a `to` value to indicate which segment of the circle should be shown. In this case, we want to show progress of 85%. So, we set the `from` value to 0 and the `to` value to 0.85.

To display the completion percentage, we overlay a text view in the middle of the circle.


ContentView.swift
SwiftUIShape 〉 SwiftUIShape 〉 ContentView.swift 〉 No Selection
85%
Complete
100%

```swift

    private var purpleGradient = LinearGradient(gradient: Gradient(colors: [
        Color(red: 207/255, green: 150/255, blue: 207/255), Color(red: 107/255,
        green: 116/255, blue: 179/255) ]), startPoint: .trailing, endPoint:
        .leading)

    var body: some View {
        ZStack {
            Circle()
                .stroke(Color(.systemGray6), lineWidth: 20)
                .frame(width: 300, height: 300)

            Circle()
                .trim(from: 0, to: 0.85)
                .stroke(purpleGradient, lineWidth: 20)
                .frame(width: 300, height: 300)
                .overlay(
                    VStack {
                        Text("85%")
                            .font(.system(size: 80, weight: .bold, design:
                                .rounded))
                            .foregroundColor(Color(.systemGray))
                        Text("Complete")
                        .font(.system(.body, design: .rounded))
                        .bold()
                        .foregroundColor(Color(.systemGray))
                    }
                )

        }
    }
}

```

Figure 16. Drawing the progress view

Drawing a Donut Chart

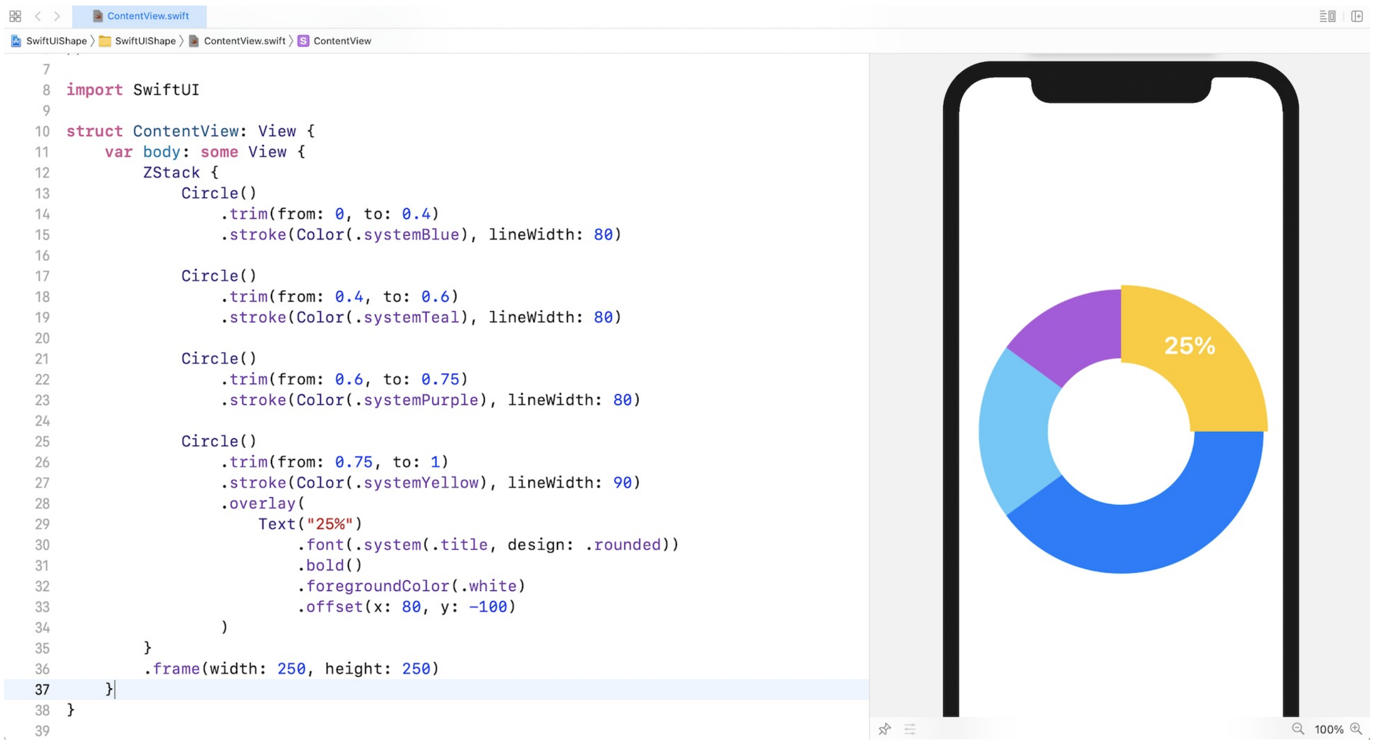
The last example I want to show you is a donut chart. If you fully understand how the `trim` modifier works, you may already know how we are going to implement the donut chart. By playing around with the values of the `trim` modifier, we can break a circle into multiple segments.

That's the technique we use to create a donut chart and here is the code:

```
ZStack {
    Circle()
        .trim(from: 0, to: 0.4)
        .stroke(Color(.systemBlue), lineWidth: 80)

    Circle()
        .trim(from: 0.4, to: 0.6)
        .stroke(Color(.systemTeal), lineWidth: 80)

    Circle()
        .trim(from: 0.6, to: 0.75)
        .stroke(Color(.systemPurple), lineWidth: 80)

    Circle()
        .trim(from: 0.75, to: 1)
        .stroke(Color(.systemYellow), lineWidth: 90)
        .overlay(
            Text("25%")
                .font(.system(.title, design: .rounded))
                .bold()
                .foregroundColor(.white)
                .offset(x: 80, y: -100)
        )
}
.frame(width: 250, height: 250)
```

The first segment represents 40% of the circle. The second segment 20% of the circle, but note that the `from` value is 0.4 instead of 0. This starts the second segment at the end of the first segment.

For the last segment, I intentionally set the line width to a larger value so that this segment stands out from the others. If you don't like that, you can change the value of `lineWidth` from `90` to `80`.



ContentView.swift
SwiftUIShape 〉 SwiftUIShape 〉 ContentView.swift 〉 ContentView
25%
100%

```swift
import SwiftUI

struct ContentView: View {
    var body: some View {
        ZStack {
            Circle()
                .trim(from: 0, to: 0.4)
                .stroke(Color(.systemBlue), lineWidth: 80)

            Circle()
                .trim(from: 0.4, to: 0.6)
                .stroke(Color(.systemTeal), lineWidth: 80)

            Circle()
                .trim(from: 0.6, to: 0.75)
                .stroke(Color(.systemPurple), lineWidth: 80)

            Circle()
                .trim(from: 0.75, to: 1)
                .stroke(Color(.systemYellow), lineWidth: 90)
                .overlay(
                    Text("25%")
                        .font(.system(.title, design: .rounded))
                        .bold()
                        .foregroundColor(.white)
                        .offset(x: 80, y: -100)
                )
        }
        .frame(width: 250, height: 250)
    }
}
```

Figure 17. Drawing the donut chart

Summary

I hope you enjoyed reading this chapter and coding the demo projects. With these drawing APIs, provided by the framework, you can easily create custom shapes for your application. There is a lot you can do with `Path` and `Shape`. I have covered just a few of these in this chapter, try to apply what you've learned and further explore these powerful APIs, they are magical!

For reference, you can download the shapes project files below:

- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUIShape.zip)

Chapter 9
Basic Animations and Transitions

Have you ever used the magic move animation in Keynote? With magic move, you can easily create slick animation between slides. Keynote automatically analyzes the objects between slides and renders the animations automatically. To me, SwiftUI has brought Magic Move to app development. Animations using the framework are automatic and magical. You define two states of a view and SwiftUI will figure out the rest, animating the changes between the two states.

SwiftUI empowers you to animate changes for individual views and transitions between views. The framework comes with a number of built-in animations to create different effects.

In this chapter, you will learn how to animate views using implicit and explicit animations, provided by SwiftUI. As usual, we'll work on a few demo projects and learn the programming technique along the way.

Implicit and Explicit Animations

SwiftUI provides two types of animations: *implicit* and *explicit*. Both approaches allow you to animate views and view transitions. For implementing implicit animations, the framework provides a modifier called `animation`. You attach this modifier to the views you want to animate and specify your preferred animation type. Optionally, you can define the animation duration and delay. SwiftUI will then automatically render the animation based on the state changes of the views.

Explicit animations offer more finite control over the animations you want to present. Instead of attaching a modifier to the view, you tell SwiftUI what state changes you want to animate inside the `withAnimation()` block.

A bit confused? That's fine. You will have a better idea after going through a couple of examples.

Implicit Animations

Let's begin with implicit animations. Create a new project, name it `SwiftUIAnimation` (or whatever name you like). Be sure to select SwiftUI for the interface!

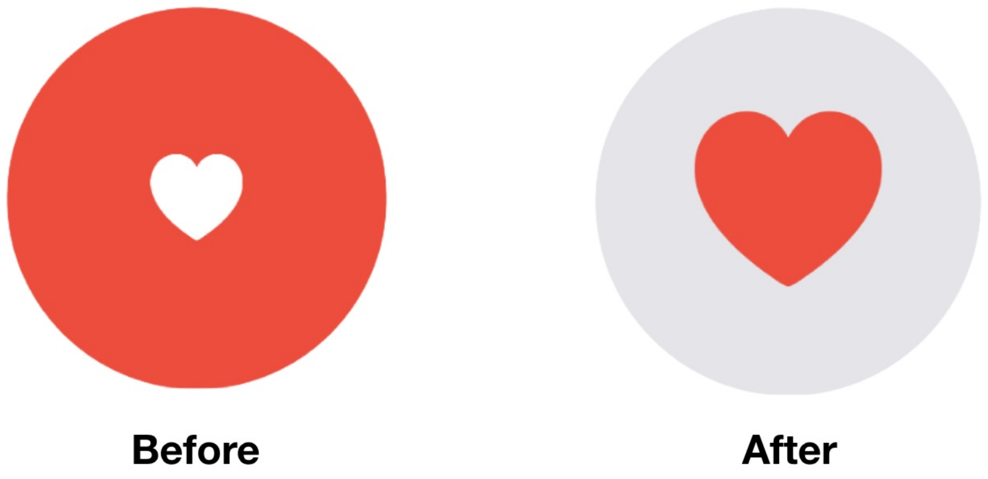


Figure 1. Animate a button's state change

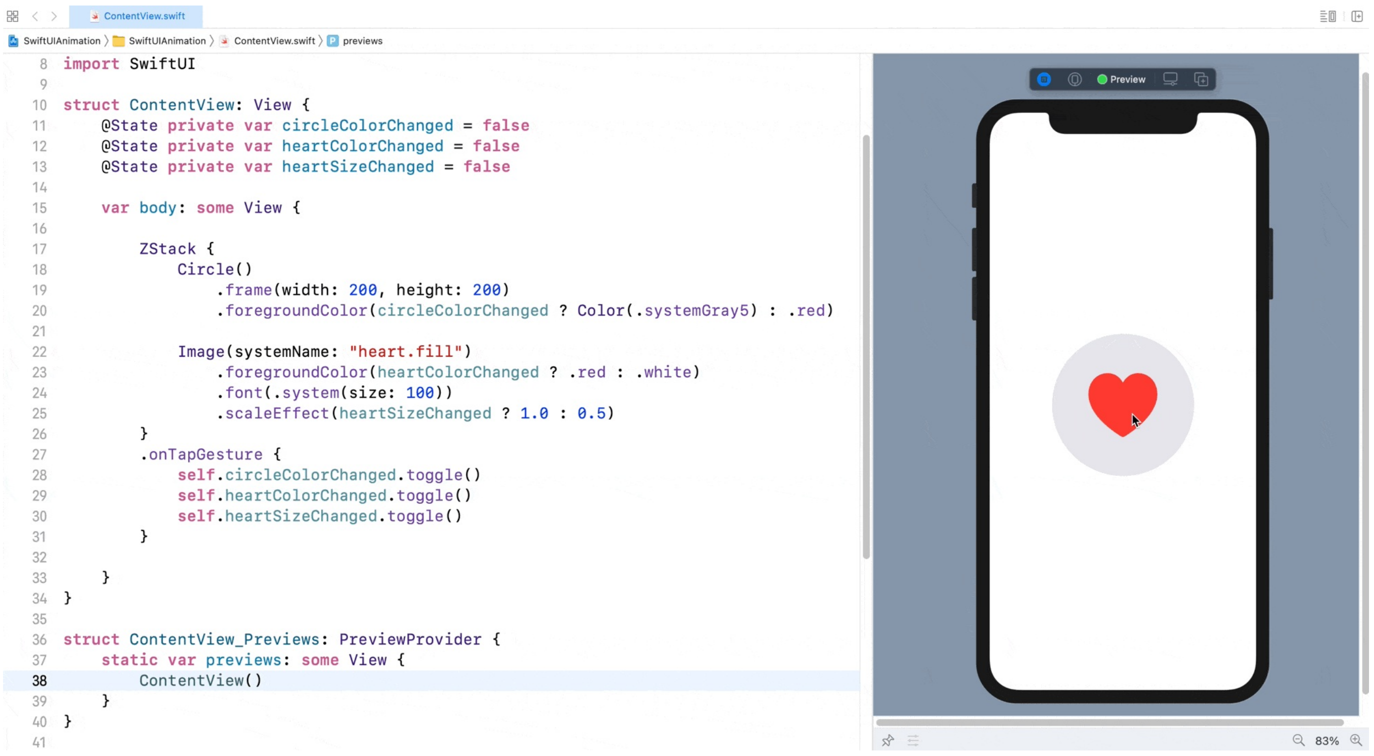
Take a look at figure 1. It's a simple tappable view that is composed of a red circle and a heart. When a user taps the heart or circle, the circle's color will be changed to light gray and the heart's color to red. At the same time, the size of the heart icon grows bigger. We have various state changes here:

1. The color of the circle changes from red to light gray.
2. The color of the heart icon changes from white to red.
3. The heart icon doubles its original size.

To implement the tappable circle using SwiftUI, add the following code to `ContentView.swift`:

```
struct ContentView: View {
    @State private var circleColorChanged = false
    @State private var heartColorChanged = false
    @State private var heartSizeChanged = false

    var body: some View {

        ZStack {
            Circle()
                .frame(width: 200, height: 200)
                .foregroundColor(circleColorChanged ? Color(.systemGray5) : .red)

            Image(systemName: "heart.fill")
                .foregroundColor(heartColorChanged ? .red : .white)
                .font(.system(size: 100))
                .scaleEffect(heartSizeChanged ? 1.0 : 0.5)
        }
        .onTapGesture {
            self.circleColorChanged.toggle()
            self.heartColorChanged.toggle()
            self.heartSizeChanged.toggle()
        }

    }
}
```

We define three state variables to model the states of the circle color, heart color and heart size, with the inital value set to false. To create the circle and heart, we use `ZStack` to overlay the heart image on top of the circle. SwiftUI comes with the `onTapGesture` modifier to detect the tap gesture. You can attach it to any view to make it tappable. In the `onTapGesture` closure, we toggle the states to change the view's appearance.



```
import SwiftUI

struct ContentView: View {
    @State private var circleColorChanged = false
    @State private var heartColorChanged = false
    @State private var heartSizeChanged = false

    var body: some View {

        ZStack {
            Circle()
                .frame(width: 200, height: 200)
                .foregroundColor(circleColorChanged ? Color(.systemGray5) : .red)

            Image(systemName: "heart.fill")
                .foregroundColor(heartColorChanged ? .red : .white)
                .font(.system(size: 100))
                .scaleEffect(heartSizeChanged ? 1.0 : 0.5)
        }
        .onTapGesture {
            self.circleColorChanged.toggle()
            self.heartColorChanged.toggle()
            self.heartSizeChanged.toggle()
        }

    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 2. Implementing the circle and heart views

If you run the app in the canvas, the color of the circle and heart icon change when you tap the view. However, these changes are not animated.

To animate the changes, you need to attach the `animation` modifier to both `Circle` and `Image` views:

```
Circle()
    .frame(width: 200, height: 200)
    .foregroundColor(circleColorChanged ? Color(.systemGray5) : .red)
    .animation(.default)

Image(systemName: "heart.fill")
    .foregroundColor(heartColorChanged ? .red : .white)
    .font(.system(size: 100))
    .scaleEffect(heartSizeChanged ? 1.0 : 0.5)
    .animation(.default)
```

SwiftUI automatically computes and renders the animation that allows the views to go smoothly from one state to another state. Tap the heart again and you should see a slick animation.

Not only can you apply the `animation` modifier to a single view, it is applicable to a group of views. For example, you can rewrite the code above by attaching the `animation` modifier to `ZStack` like this:

```
ZStack {
    Circle()
        .frame(width: 200, height: 200)
        .foregroundColor(circleColorChanged ? Color(.systemGray5) : .red)

    Image(systemName: "heart.fill")
        .foregroundColor(heartColorChanged ? .red : .white)
        .font(.system(size: 100))
        .scaleEffect(heartSizeChanged ? 1.0 : 0.5)
}
.animation(.default)
.onTapGesture {
    self.circleColorChanged.toggle()
    self.heartColorChanged.toggle()
    self.heartSizeChanged.toggle()
}
```

It works exactly same. SwiftUI looks for all the state changes embedded in `ZStack` and creates the animations.

In the example, we use the default animation. SwiftUI provides a number of built-in animations for you to choose including `linear` , `easeIn` , `easeOut` , `easeInOut` , and `spring` . The `linear` animation animates the changes in linear speed, while other easing animations have various speed. For details, you can check out www.easings.net to see the difference between each of the easing functions.

To use an alternate animation, you just need to set the specific animation in the `animation` modifier. Let's say, you want to use the `spring` animation, you can change `.default` to the following:

```
.animation(.spring(response: 0.3, dampingFraction: 0.3, blendDuration: 0.3))
```

This renders a spring-based animation that gives the heart a bumpy effect. You can adjust the damping and blend values to achieve a different effect.

Explicit Animations

That's how you animate views using implicit animation. Let's see how we can achieve the same result using explicit animation. As explained before, you need to wrap the state changes in a `withAnimation` block. To create the same animated effect, you can write the code like this:

```
ZStack {
    Circle()
        .frame(width: 200, height: 200)
        .foregroundColor(circleColorChanged ? Color(.systemGray5) : .red)

    Image(systemName: "heart.fill")
        .foregroundColor(heartColorChanged ? .red : .white)
        .font(.system(size: 100))
        .scaleEffect(heartSizeChanged ? 1.0 : 0.5)
}
.onTapGesture {
    withAnimation(.default) {
        self.circleColorChanged.toggle()
        self.heartColorChanged.toggle()
        self.heartSizeChanged.toggle()
    }
}
```

We no longer use the `animation` modifier, instead we wrap the code in `onTapGesture` with `withAnimation`. The `withAnimation` call takes in an animation parameter. Here we specify to use the default animation.

Of course, you can change it to spring animation by updating `withAnimation` like this:

```
withAnimation(.spring(response: 0.3, dampingFraction: 0.3, blendDuration: 0.3)) {
    self.circleColorChanged.toggle()
    self.heartColorChanged.toggle()
    self.heartSizeChanged.toggle()
}
```

With explicit animation, you can easily control which state you want to animate. For example, if you don't want to animate the size change of the heart icon, you can exclude that line of code from `withAnimation` like this:

```
.onTapGesture {
    withAnimation(.spring(response: 0.3, dampingFraction: 0.3, blendDuration: 0.3)
) {
        self.circleColorChanged.toggle()
        self.heartColorChanged.toggle()
    }

    self.heartSizeChanged.toggle()
}
```

In this case, SwiftUI will only animate the color change of both circle and heart. You will no longer see the animated growing effect of the heart icon.

You may wonder if we can disable the scale animation by using implicit animation. You can! You can reorder the `.animation` modifier to prevent SwiftUI from animating a certain state change. Here is the code that achieves the same effect:

```
ZStack {
    Circle()
        .frame(width: 200, height: 200)
        .foregroundColor(circleColorChanged ? Color(.systemGray5) : .red)
        .animation(.spring(response: 0.3, dampingFraction: 0.3, blendDuration: 0.3
))

    Image(systemName: "heart.fill")
        .foregroundColor(heartColorChanged ? .red : .white)
        .font(.system(size: 100))
        .animation(.spring(response: 0.3, dampingFraction: 0.3, blendDuration: 0.3
))
        .scaleEffect(heartSizeChanged ? 1.0 : 0.5)
}
.onTapGesture {
    self.circleColorChanged.toggle()
    self.heartColorChanged.toggle()
    self.heartSizeChanged.toggle()
}
```

For the `Image` view, we place the `animation` modifier right before `scaleEffect`. This will cancel the animation. The state change of the `scaleEffect` modifier will not be animated.

While you can create the same animation using implicit animation, in my opinion, it's more convenient to use explicit animation in this case.

Creating a Loading Indicator Using RotationEffect

The power of SwiftUI animation is that you don't need to care how the views are animated. All you need is to provide the start and end state. SwiftUI will then figure out the rest. Utilizing this concept, you can create various types of animation.

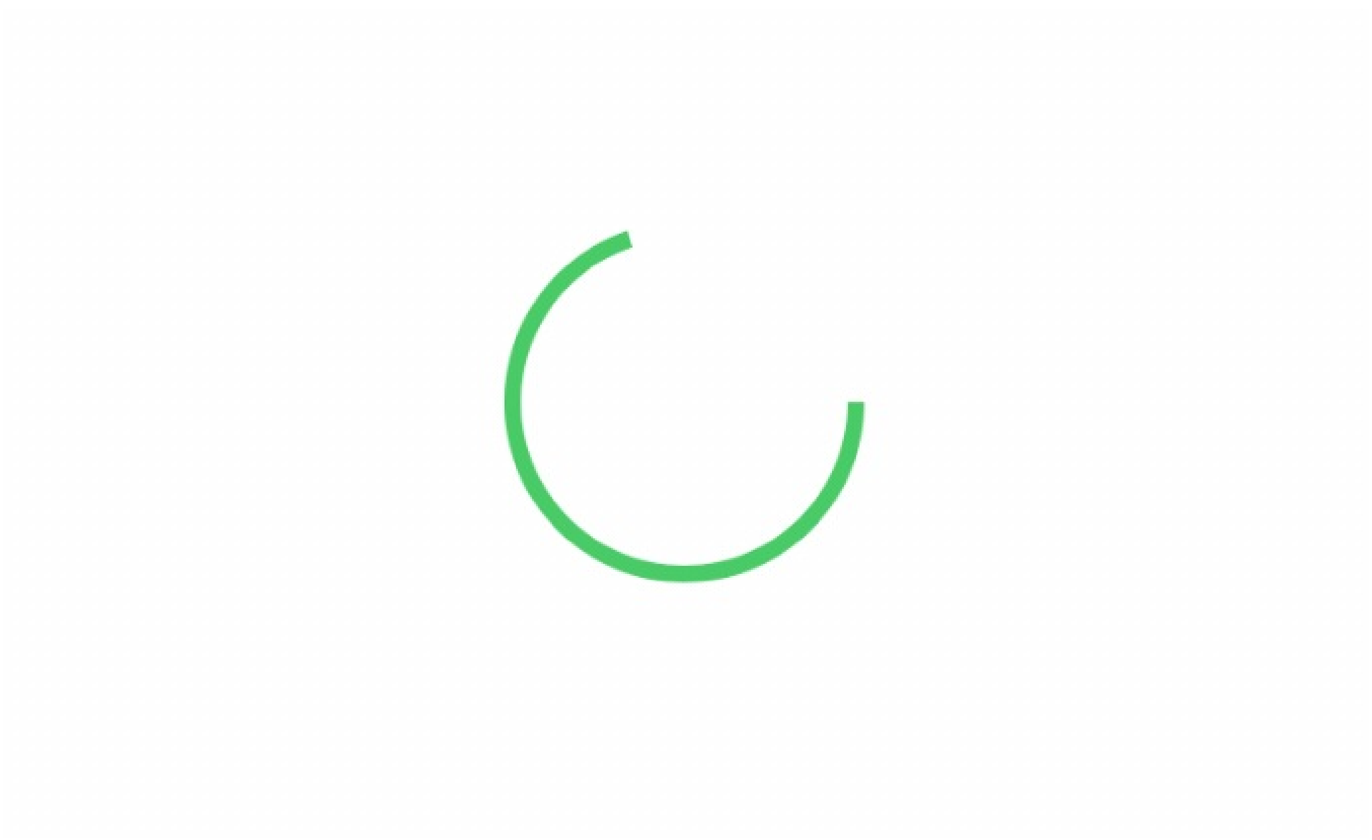


Figure 3. A sample loading indicator

For example, let's create a simple loading indicator that you can commonly find in a real-world application like "Medium". To create a loading indicator like that shown in figure 3, we start with an open ended circle like this:

```
Circle()
    .trim(from: 0, to: 0.7)
    .stroke(Color.green, lineWidth: 5)
    .frame(width: 100, height: 100)
```

How do we rotate the circle? We make use of the `rotationEffect` and `animation` modifiers. The trick is to keep rotating the circle by 360 degrees. Here is the code:

```
struct ContentView: View {
    @State private var isLoading = false

    var body: some View {
        Circle()
            .trim(from: 0, to: 0.7)
            .stroke(Color.green, lineWidth: 5)
            .frame(width: 100, height: 100)
            .rotationEffect(Angle(degrees: isLoading ? 360 : 0))
            .animation(Animation.default.repeatForever(autoreverses: false))
            .onAppear() {
                self.isLoading = true
            }
    }
}
```

The `rotationEffect` modifier takes in the rotation degree (360 degrees). In the code above, we have a state variable to control the loading status. When it's set to true, the rotation degree will be set to 360 to rotate the circle. In the `animation` modifier, we specify to use the `.default` animation, but there is a difference. We tell SwiftUI to repeat the same animation again and again. This is the trick that creates the loading animation.

If you want to change the speed of the animation, you can use the linear animation and specify a duration like this:

```
.animation(Animation.linear(duration: 5).repeatForever(autoreverses: false))
```

The greater the duration value the slower the animation (rotation).

The `onAppear` modifier may be new to you. If you have some knowledge of UIKit, this modifier is very similar to `viewDidAppear`. It's automatically called when the view appears on screen. In the code, we change the loading status to true in order to start the animation when the view is loaded up.

Once you manage this technique, you can tweak the design and develop various versions of loading indicator. For example, you can overlay an arc on a circle to create a fancy loading indicator.

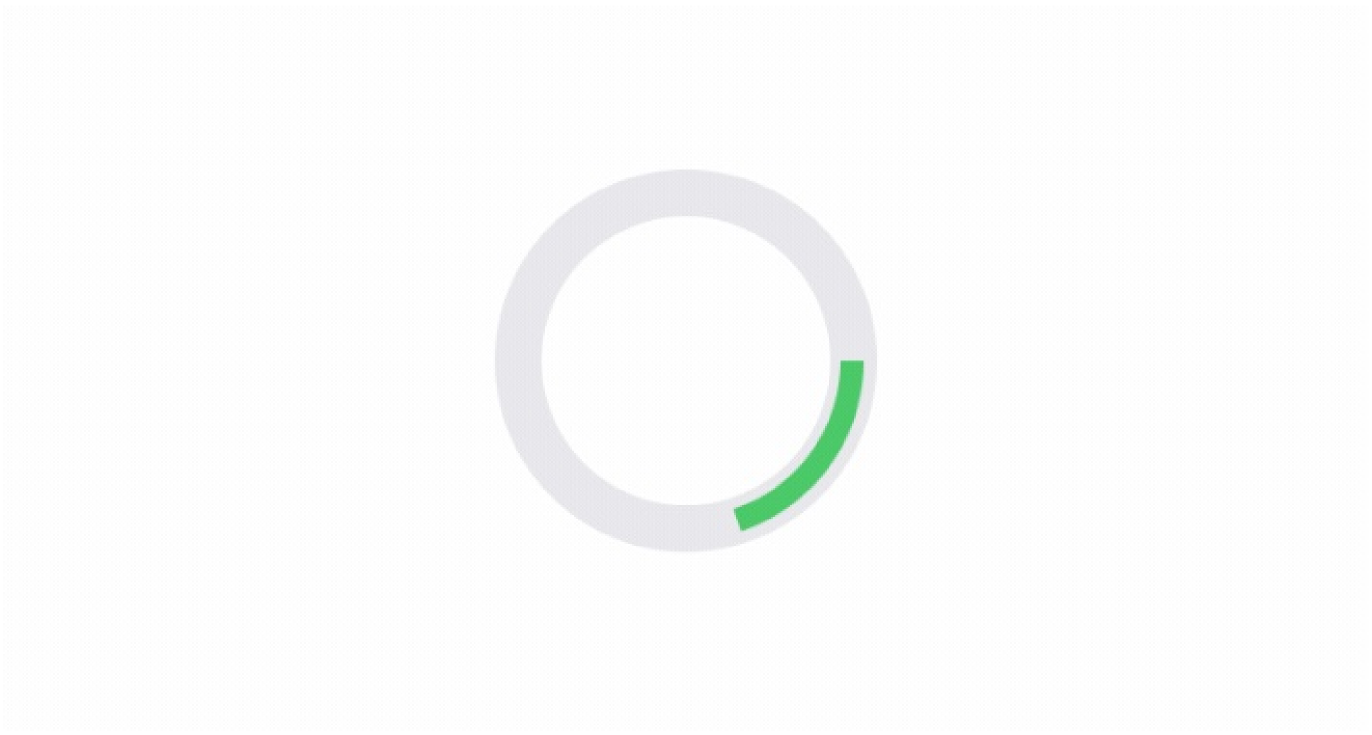


Figure 4. A sample loading indicator

```
struct ContentView: View {

    @State private var isLoading = false

    var body: some View {
        ZStack {

            Circle()
                .stroke(Color(.systemGray5), lineWidth: 14)
                .frame(width: 100, height: 100)

            Circle()
                .trim(from: 0, to: 0.2)
                .stroke(Color.green, lineWidth: 7)
                .frame(width: 100, height: 100)
                .rotationEffect(Angle(degrees: isLoading ? 360 : 0))
                .animation(Animation.linear(duration: 1).repeatForever(autoreverse
s: false))
                .onAppear() {
                    self.isLoading = true
            }
        }
    }
}
```

The loading indicator doesn't need to be circular. You can also use `Rectangle` or `RoundedRectangle` to create the indicator. Instead of changing the rotation angle, you modify the value of the offset to create an animation like this.

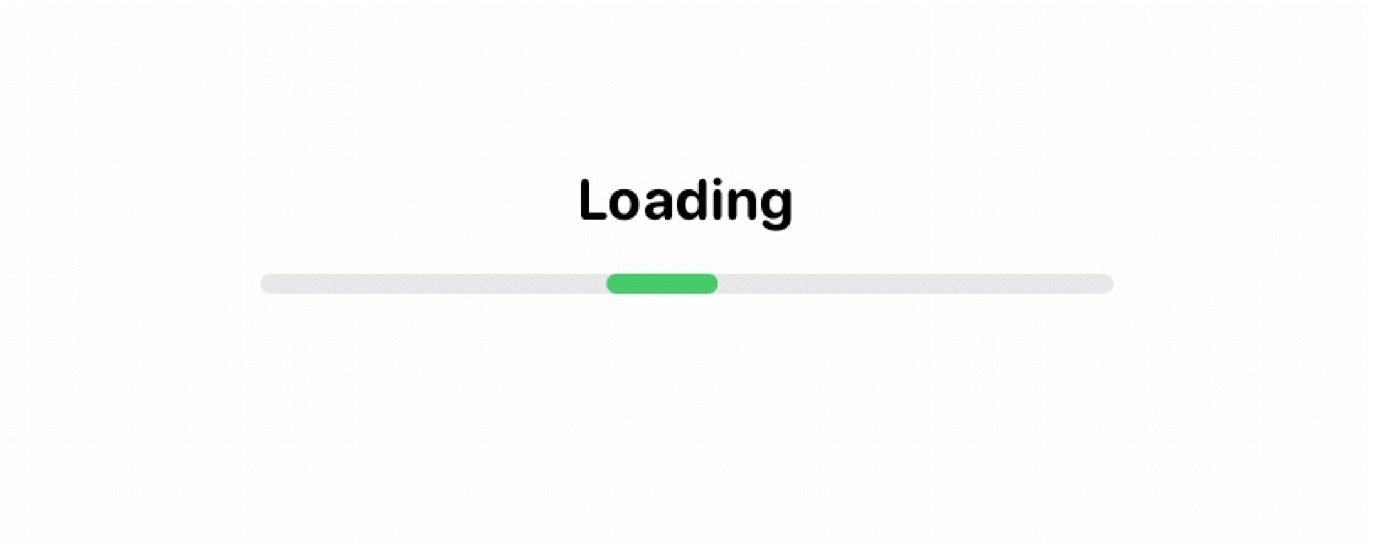


Figure 5. Another example of the loading indicator

To create the animation, we overlay two rounded rectangles together. The rectangle on top is much shorter than the one below. When the loading begins, we update its offset value from -110 to 110.

```
struct ContentView: View {

    @State private var isLoading = false

    var body: some View {
        ZStack {

            Text("Loading")
                .font(.system(.body, design: .rounded))
                .bold()
                .offset(x: 0, y: -25)

            RoundedRectangle(cornerRadius: 3)
                .stroke(Color(.systemGray5), lineWidth: 3)
                .frame(width: 250, height: 3)

            RoundedRectangle(cornerRadius: 3)
                .stroke(Color.green, lineWidth: 3)
                .frame(width: 30, height: 3)
                .offset(x: isLoading ? 110 : -110, y: 0)
                .animation(Animation.linear(duration: 1).repeatForever(autoreverse
s: false))
        }
        .onAppear() {
            self.isLoading = true
        }
    }
}
```

This moves the green rectangle along the line. When you repeat the same animation over and over, it becomes a loading animation. Figure 6 illustrates the offset values.

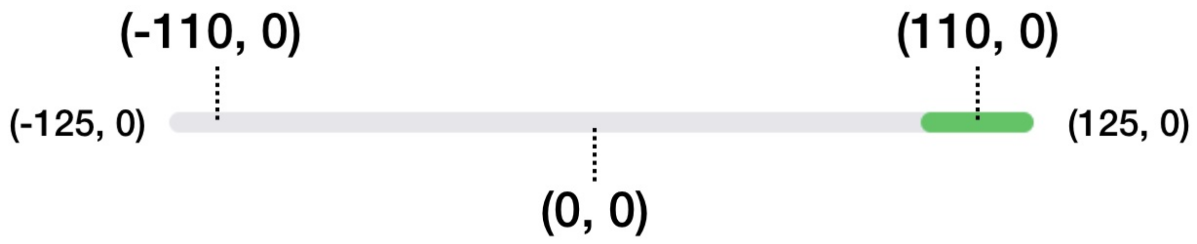


Figure 6. Another example of the loading indicator

Creating a Progress Indicator

The loading indicator provides feedback to the user that the app is working on something. However, it doesn't show the actual progress. If you need to give users more information about the progress of a task, you may want to build a progress indicator.

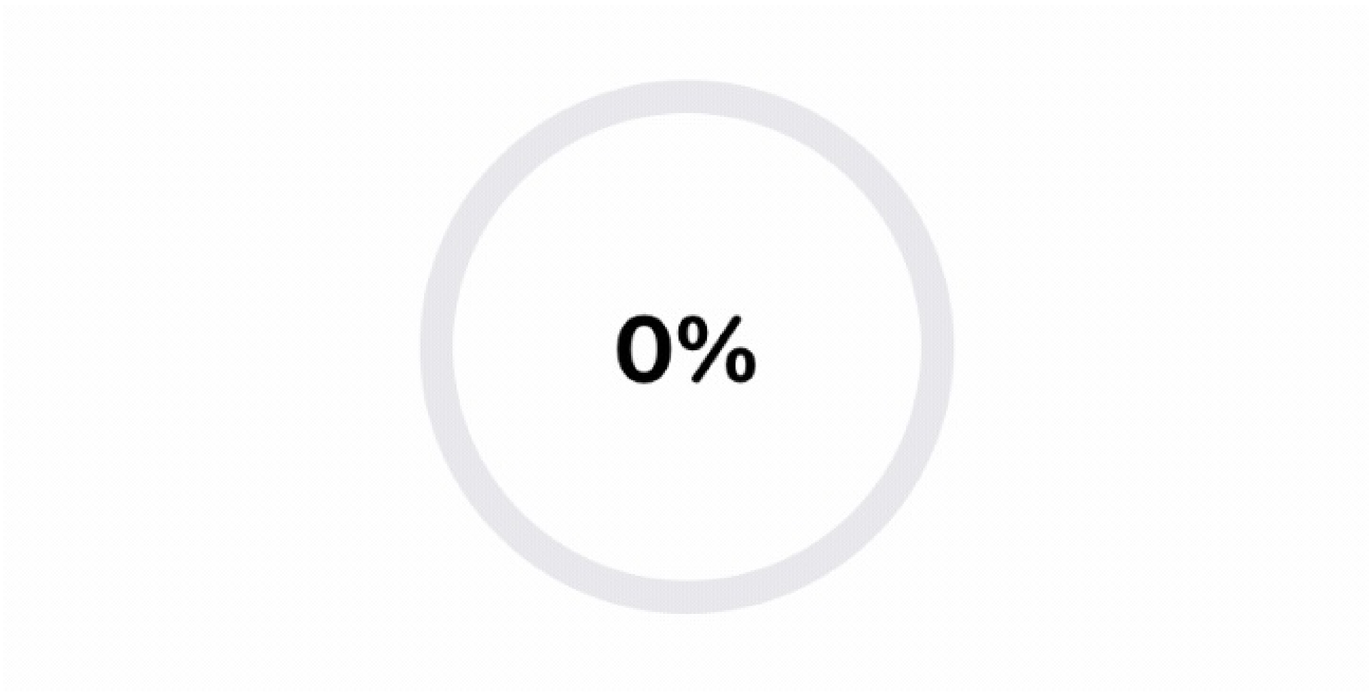


Figure 7. A progress indicator

Building a progress indicator is very similar to that of the loading indicator. But you need a state variable to keep track of the progress. Here is a code snippet for creating the indicator:

```
struct ContentView: View {
    @State private var progress: CGFloat = 0.0

    var body: some View {

        ZStack {
            Text("\(Int(progress * 100))%")
                .font(.system(.title, design: .rounded))
                .bold()

            Circle()
                .stroke(Color(.systemGray5), lineWidth: 10)
                .frame(width: 150, height: 150)

            Circle()
                .trim(from: 0, to: progress)
                .stroke(Color.green, lineWidth: 10)
                .frame(width: 150, height: 150)
                .rotationEffect(Angle(degrees: -90))
        }
        .onAppear() {
            Timer.scheduledTimer(withTimeInterval: 0.5, repeats: true) { timer in
                self.progress += 0.05
                print(self.progress)
                if self.progress >= 1.0 {
                    timer.invalidate()
                }
            }
        }
    }
}
```

Instead of a boolean state variable, we use a floating point number to store the status. To display progress, we set the `trim` modifier with the progress value. In a real world application, you can update the value of the `progress` value to show the actual progress of the operation. For this demo, we used a timer which updates the progress every half second.

Delaying an Animation

Not only does the SwiftUI framework allow you to control the duration of an animation, you can also delay an animation through the `delay` function like this:

```
Animation.default.delay(1.0)
```

This will delay the start of the animation by 1 second. The `delay` function is applicable to other animations.

By mixing and matching the values of duration and delay, you can achieve some interesting animations like the dot loading indicator below.

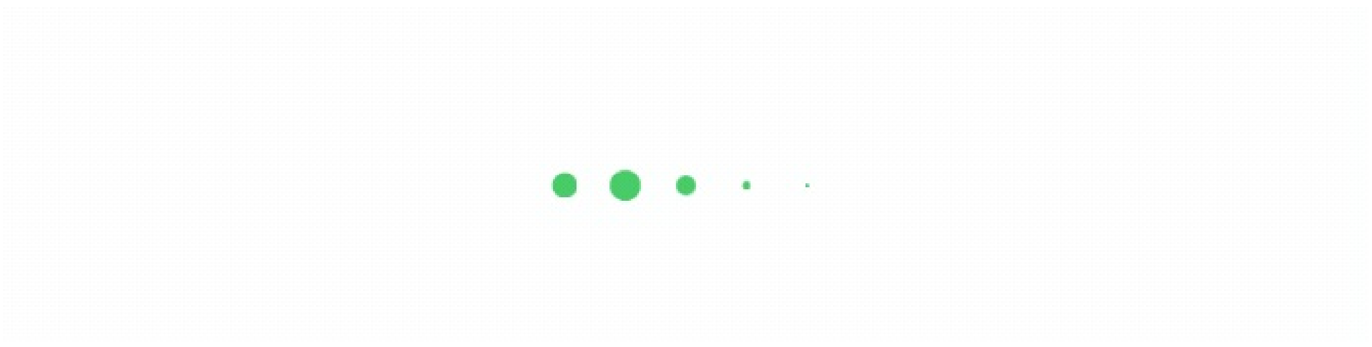


Figure 8. A dot loading indicator

This indicator is composed of five dots. Each dot is animated to scale up and down, but with different time delays. Here is how it's implemented in code.

```
struct ContentView: View {
    @State private var isLoading = false

    var body: some View {
        HStack {
            ForEach(0...4, id: \.self) { index in
                Circle()
                    .frame(width: 10, height: 10)
                    .foregroundColor(.green)
                    .scaleEffect(self.isLoading ? 0 : 1)
                    .animation(Animation.linear(duration: 0.6).repeatForever().del
ay(0.2 * Double(index)))
            }
        }
        .onAppear() {
            self.isLoading = true
        }
    }
}
```

We first use a `HStack` to layout the circles horizontally. Since all five circles (dots) are the same size and color, we use `ForEach` to create the circles. The `scaleEffect` modifier is used to scale the circle's size. By default, it's set to 1, which is its original size. When the loading starts, the value is updated to 0. This will minimize the dot.

The line of code for rendering the animation looks a bit complicated. Let's break it down and look at it step by step:

```
Animation.linear(duration: 0.6).repeatForever().delay(0.2 * Double(index))
```

The first part creates a linear animation with a duration of 0.6 seconds. This animation is expected to run repeatedly, so we call the `repeatForever` function.

If you run the animation without calling the `delay` function, all the dots scales up and down simultaneously. However, this is not what we want. Instead of scaling up/down all at once, each dot should resize itself independently. This is why we call the `delay` function and use a different delay value for each dot (based on its order in the row).

You may vary the value of duration and delay to tweak the animation.

Transforming a Rectangle into Circle

Sometimes, you probably need to smoothly transform one shape (e.g. rectangle) into another (e.g. circle). With the built-in shape and animation, you can easily create this transformation as shown in figure 9.

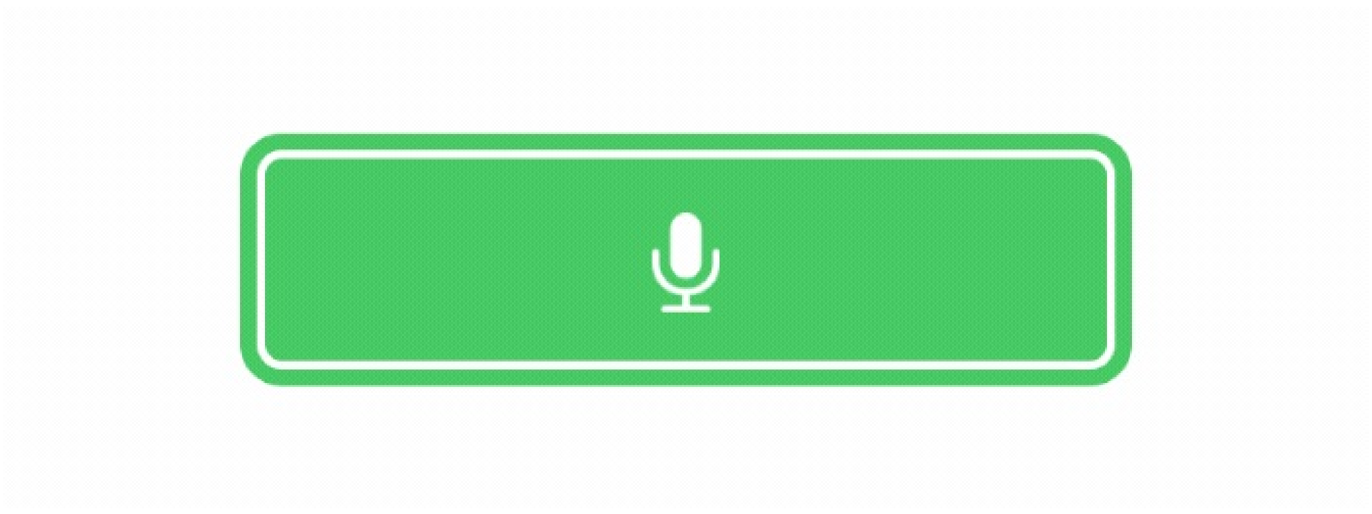


Figure 9. Morphing a rectangle into a circle

The trick of morphing a rectangle into a circle is to use the `RoundedRectangle` shape and animate the change of the corner radius. Assuming the width and height of the rectangle are the same, it becomes a circle when its corner radius is set to half of its width. Here is the implementation of the morphing button:

```swift
struct ContentView: View {
    @State private var recordBegin = false
    @State private var recording = false

    var body: some View {
        ZStack {

            RoundedRectangle(cornerRadius: recordBegin ? 30 : 5)
                .frame(width: recordBegin ? 60 : 250, height: 60)
                .foregroundColor(recordBegin ? .red : .green)
                .overlay(
                    Image(systemName: "mic.fill")
                        .font(.system(.title))
                        .foregroundColor(.white)
                        .scaleEffect(recording ? 0.7 : 1)
                )

            RoundedRectangle(cornerRadius: recordBegin ? 35 : 10)
                .trim(from: 0, to: recordBegin ? 0.0001 : 1)
                .stroke(lineWidth: 5)
                .frame(width: recordBegin ? 70 : 260, height: 70)
                .foregroundColor(.green)

        }
        .onTapGesture {
            withAnimation(Animation.spring()) {
                self.recordBegin.toggle()
            }

            withAnimation(Animation.spring().repeatForever().delay(0.5)) {
                self.recording.toggle()
            }
        }
    }
}
```

We have two state variables here: `recordBegin` and `recording` to control two separate animations. The first variable controls the morphing of the button. As explained before, we make use of the corner radius for the transformation. The width of the rectangle is originally set to 250 points. When a user taps the rectangle to trigger the transformation, the frame's width is changed to 60 points. Alongside with the change, the corner radius is changed to 30 points, which is half of the width.

This is how we transform a rectangle into a circle. SwiftUI automatically renders the animation of this transformation.

The `recording` state variable, handles the scaling of the mic image. We change the scaling ratio from 1 to 0.7 when it's in the recording state. By running the same animation repeatedly, it creates the pulsing animation.

Note that the code above uses the explicit approach to animate the views. This is not mandatory. If you prefer, you can use the implicit animation approach to achieve the same result.

Understanding Transitions

What we have discussed so far is animating a view that already exists in the view hierarchy. We animate the view's size by scaling it up and down.

SwiftUI allows developers to do more than that. You can define how a view is inserted or removed from the view hierarchy. In SwiftUI, this is known as transition. By default, the framework uses fade in and fade out transition. However, it comes with several ready-to-use transitions such as slide, move, opacity, etc. Of course, you are allowed to develop your own or simply mix and match various types of transitions together to create your desired transition.

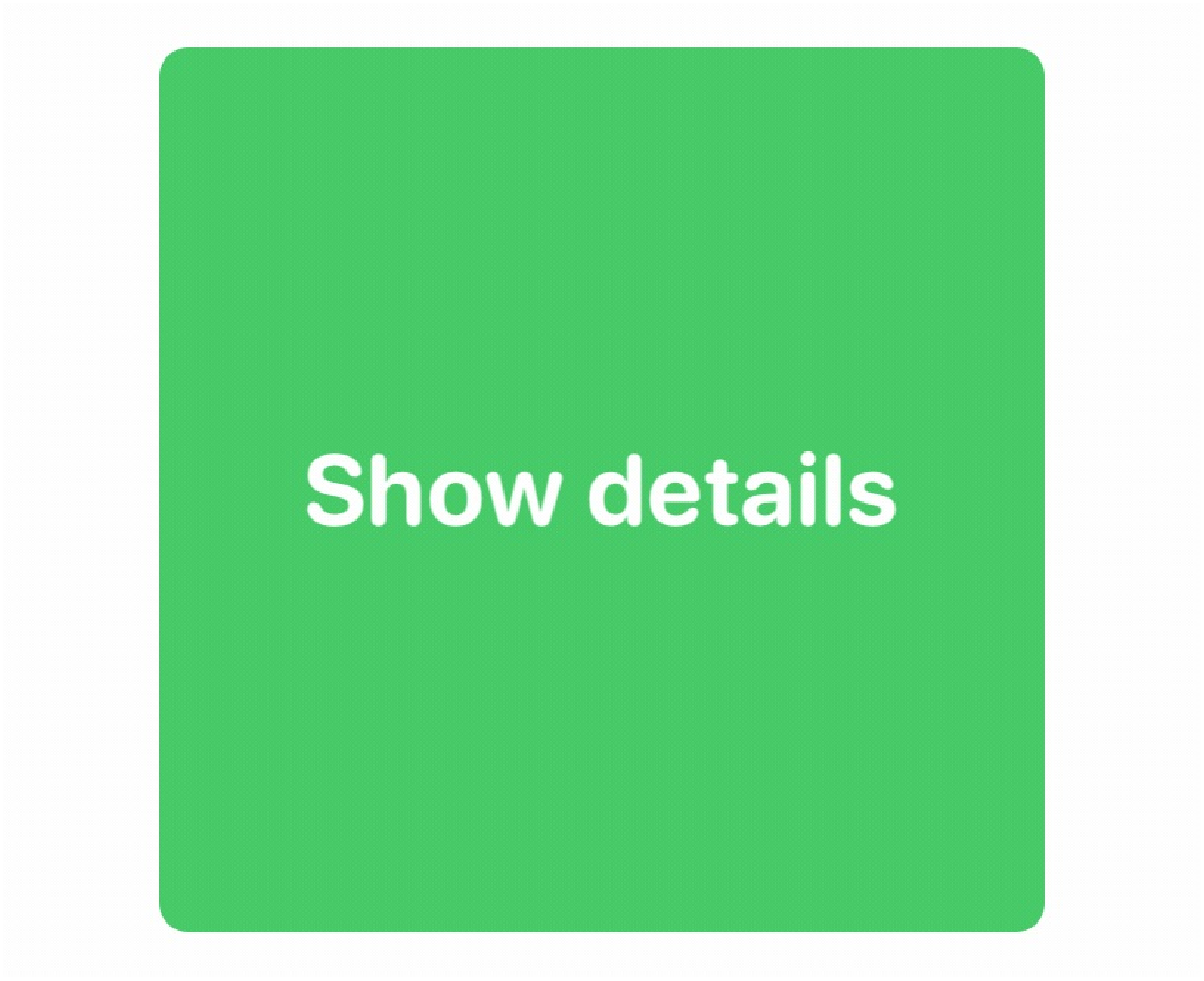


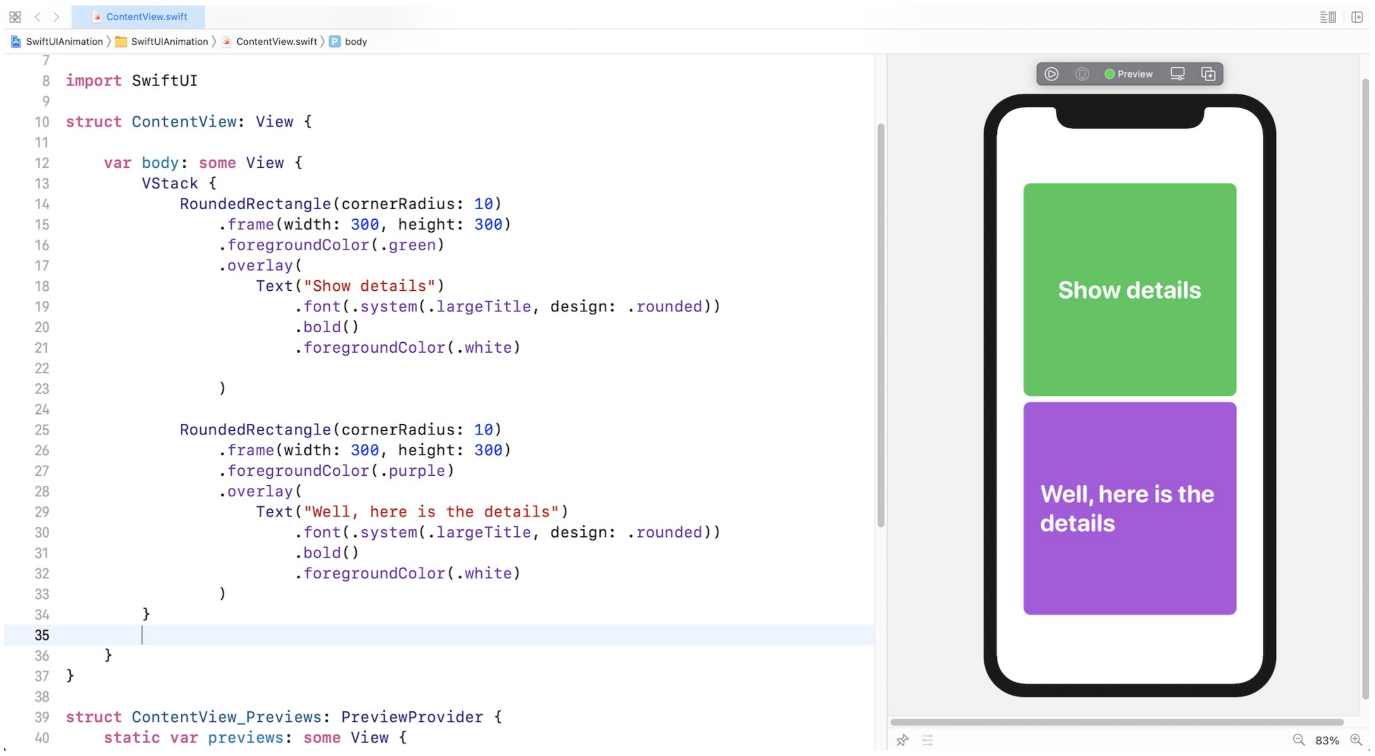
Figure 10. A sample transition created using SwiftUI

Building a Simple Transition

Let's take a look at a simple example to better understand what a transition is and how it works with animations. Create a new project named `SwiftUITransition` and update the `ContentView` like this:

```
struct ContentView: View {

    var body: some View {
        VStack {
            RoundedRectangle(cornerRadius: 10)
                .frame(width: 300, height: 300)
                .foregroundColor(.green)
                .overlay(
                    Text("Show details")
                        .font(.system(.largeTitle, design: .rounded))
                        .bold()
                        .foregroundColor(.white)

            )

            RoundedRectangle(cornerRadius: 10)
                .frame(width: 300, height: 300)
                .foregroundColor(.purple)
                .overlay(
                    Text("Well, here is the details")
                        .font(.system(.largeTitle, design: .rounded))
                        .bold()
                        .foregroundColor(.white)
                )
        }
    }
}
```

In the code above, we lay out two squares vertically using `VStack`. At first, the purple rectangle should be hidden. It's displayed only when a user taps the green rectangle (i.e. Show details). In order to show the purple square, we need to make the green square tappable.



```
import SwiftUI

struct ContentView: View {

    var body: some View {
        VStack {
            RoundedRectangle(cornerRadius: 10)
                .frame(width: 300, height: 300)
                .foregroundColor(.green)
                .overlay(
                    Text("Show details")
                        .font(.system(.largeTitle, design: .rounded))
                        .bold()
                        .foregroundColor(.white)

                )

            RoundedRectangle(cornerRadius: 10)
                .frame(width: 300, height: 300)
                .foregroundColor(.purple)
                .overlay(
                    Text("Well, here is the details")
                        .font(.system(.largeTitle, design: .rounded))
                        .bold()
                        .foregroundColor(.white)
                )
        }

    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
```

ContentView.swift
SwiftUIAnimation 〉 SwiftUIAnimation 〉 ContentView.swift 〉 body
Preview
Show details
Well, here is the details
83%

Figure 11. Layout two rectangles vertically

To do that, we need to declare a state variable to determine whether the purple square is shown or not. Insert this line of code in `ContentView`:

```
@State private var show = false
```

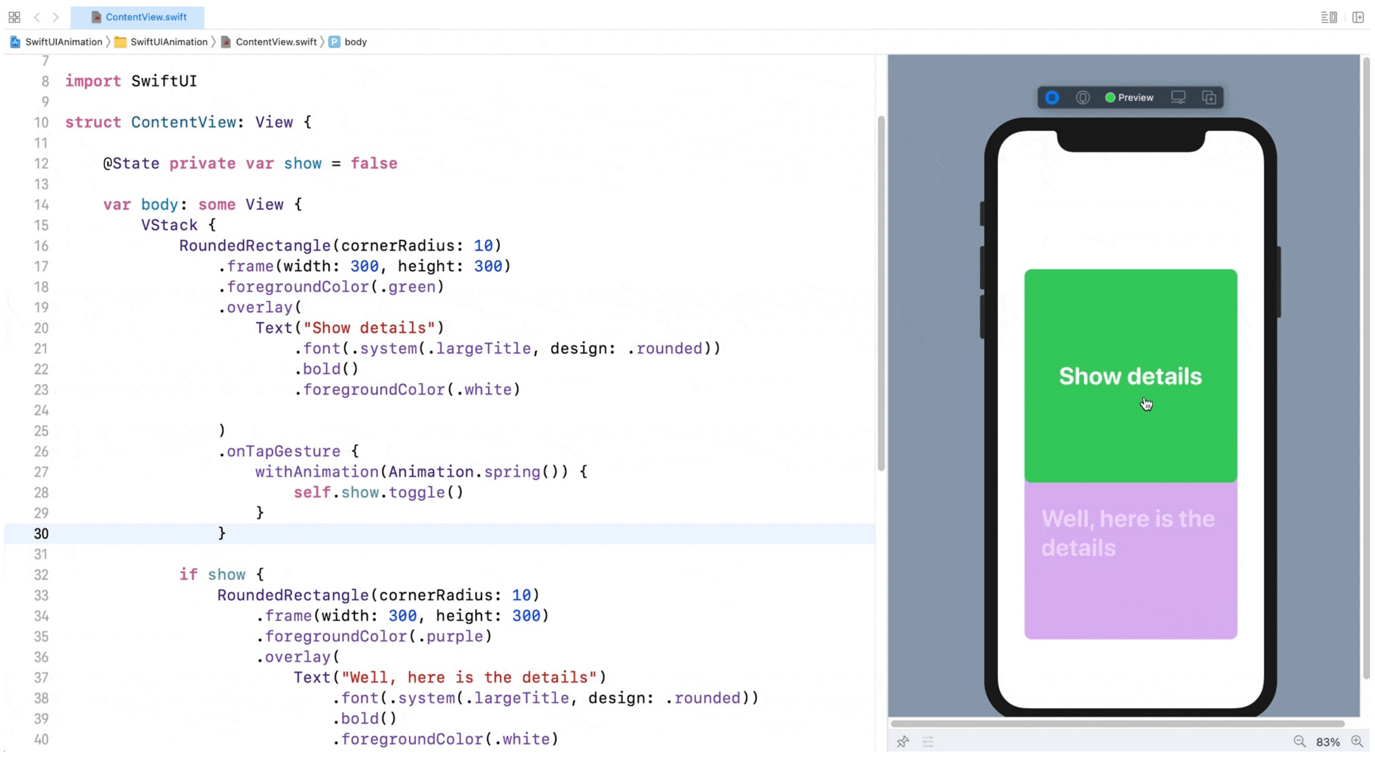
Next, to hide the purple square, we wrap the purple square within a `if` clause like this:

```
if show {
    RoundedRectangle(cornerRadius: 10)
        .frame(width: 300, height: 300)
        .foregroundColor(.purple)
        .overlay(
            Text("Well, here is the details")
                .font(.system(.largeTitle, design: .rounded))
                .bold()
                .foregroundColor(.white)
        )
}
```

For the `VStack`, we attach the `onTapGesture` function to detect a tap and create an animation for the state change. Note that the transition should be associated with an animation, otherwise, it won't work on its own.

```
.onTapGesture {
    withAnimation(Animation.spring()) {
        self.show.toggle()
    }
}
```

Once a user taps the stack, we toggle the `show` variable to display the purple square. If you run the app in the simulator or the preview canvas, you should only see the green square. Tapping it will display the purple rectangle with a smooth fade in/out transition.


ContentView.swift
SwiftUIAnimation 〉 SwiftUIAnimation 〉 ContentView.swift 〉 body
Preview
Show details
Well, here is the details
83%

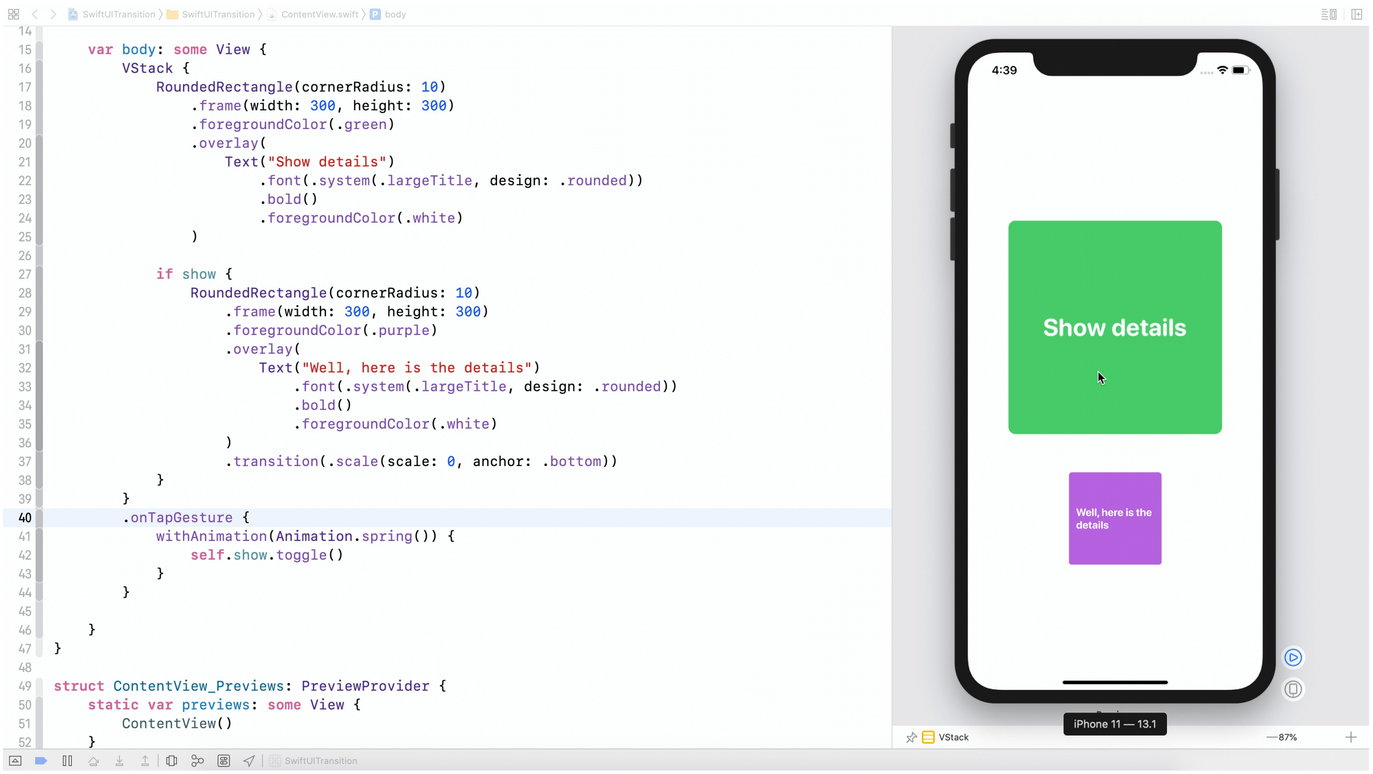
```
import SwiftUI

struct ContentView: View {

    @State private var show = false

    var body: some View {
        VStack {
            RoundedRectangle(cornerRadius: 10)
                .frame(width: 300, height: 300)
                .foregroundColor(.green)
                .overlay(
                    Text("Show details")
                        .font(.system(.largeTitle, design: .rounded))
                        .bold()
                        .foregroundColor(.white)

                )
                .onTapGesture {
                    withAnimation(Animation.spring()) {
                        self.show.toggle()
                    }
                }

            if show {
                RoundedRectangle(cornerRadius: 10)
                    .frame(width: 300, height: 300)
                    .foregroundColor(.purple)
                    .overlay(
                        Text("Well, here is the details")
                            .font(.system(.largeTitle, design: .rounded))
                            .bold()
                            .foregroundColor(.white)
```

Figure 12. The fade transition

As mentioned, if you do not specify the transition you want to use, SwiftUI renders the fade in and out transition. To use an alternative transition, attach the `transition` modifier to the purple square like this:

```
if show {
    RoundedRectangle(cornerRadius: 10)
        .frame(width: 300, height: 300)
        .foregroundColor(.purple)
        .overlay(
            Text("Well, here is the details")
                .font(.system(.largeTitle, design: .rounded))
                .bold()
                .foregroundColor(.white)
        )
        .transition(.scale(scale: 0, anchor: .bottom))
}
```

The `transition` modifier takes in a parameter of the type `AnyTransition`. Here we use the `scale` transition with the anchor set to `.bottom`. That's all you need to do to modify the transition. Run the app in simulator. You should see a pop animation when the app reveals the purple square. I suggest testing animations using the built-in simulator instead of running the app in preview because the preview canvas may not render the transition correctly.



```
    var body: some View {
        VStack {
            RoundedRectangle(cornerRadius: 10)
                .frame(width: 300, height: 300)
                .foregroundColor(.green)
                .overlay(
                    Text("Show details")
                        .font(.system(.largeTitle, design: .rounded))
                        .bold()
                        .foregroundColor(.white)
                )

            if show {
                RoundedRectangle(cornerRadius: 10)
                    .frame(width: 300, height: 300)
                    .foregroundColor(.purple)
                    .overlay(
                        Text("Well, here is the details")
                            .font(.system(.largeTitle, design: .rounded))
                            .bold()
                            .foregroundColor(.white)
                    )
                    .transition(.scale(scale: 0, anchor: .bottom))
            }
        }
        .onTapGesture {
            withAnimation(Animation.spring()) {
                self.show.toggle()
            }
        }

    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
```

SwiftUITransition 〉 SwiftUITransition 〉 ContentView.swift 〉 body
4:39
Show details
Well, here is the details
VStack
iPhone 11 — 13.1
87%
SwiftUITransition

Figure 13. Scaling transition

In addition to `.scale`, the SwiftUI framework comes with several built-in transitions including `.opaque`, `.offset`, `.move`, and `.slide`. Replace the `.scale` transition with the `.offset` transition like this:

```
.transition(.offset(x: -600, y: 0))
```

This time, the purple square slides in from the left when it's inserted into the `VStack`.

Combining Transitions

You can combine two or more transitions together by calling the `combined(with:)` method to create an even more slick transition. For example, to combine the offset and scale animation, you write the code like this:

```
.transition(AnyTransition.offset(x: -600, y: 0).combined(with: .scale))
```

Here is another example that combines three transitions:

```
.transition(AnyTransition.offset(x: -600, y: 0).combined(with: .scale).combined(wi
th: .opacity))
```

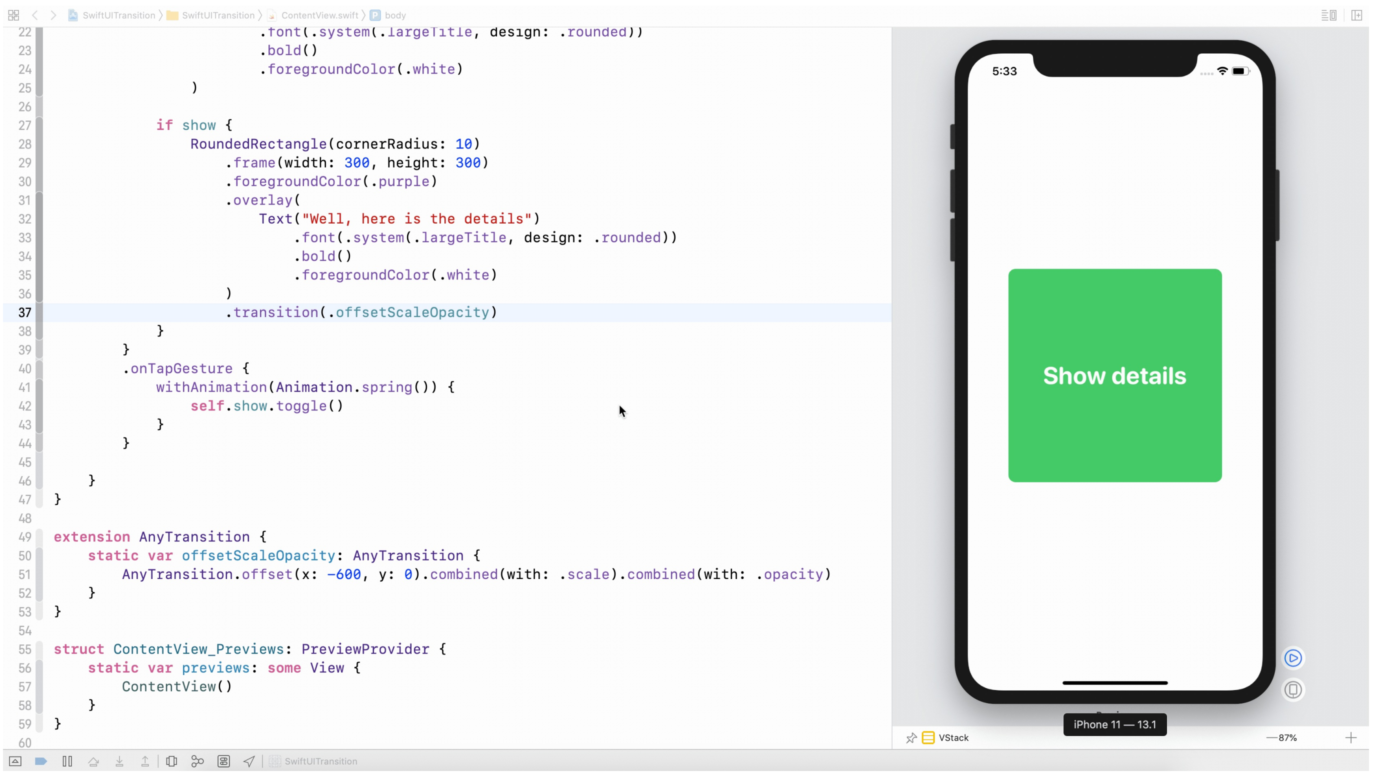
Sometimes you need to define a reusable animation. You can define an extension on `AnyTransition` like this:

```
extension AnyTransition {
    static var offsetScaleOpacity: AnyTransition {
        AnyTransition.offset(x: -600, y: 0).combined(with: .scale).combined(with:
.opacity)
    }
}
```

Then you can use the `offsetScaleOpacity` animation in the `transition` modifier directly:

```
.transition(.offsetScaleOpacity)
```

Run the app and test the transition again. Does it look great?


SwiftUITransition 〉 SwiftUITransition 〉 ContentView.swift 〉 body
5:33
Show details
iPhone 11 — 13.1
VStack
87%
SwiftUITransition

```
                        .font(.system(.largeTitle, design: .rounded))
                        .bold()
                        .foregroundColor(.white)
                )

            if show {
                RoundedRectangle(cornerRadius: 10)
                    .frame(width: 300, height: 300)
                    .foregroundColor(.purple)
                    .overlay(
                        Text("Well, here is the details")
                            .font(.system(.largeTitle, design: .rounded))
                            .bold()
                            .foregroundColor(.white)
                    )
                    .transition(.offsetScaleOpacity)
            }
        }
        .onTapGesture {
            withAnimation(Animation.spring()) {
                self.show.toggle()
            }
        }

    }
}

extension AnyTransition {
    static var offsetScaleOpacity: AnyTransition {
        AnyTransition.offset(x: -600, y: 0).combined(with: .scale).combined(with: .opacity)
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 14. Combining the scale, offset, and opacity transition

Asymmetric Transitions

The transitions that we just discussed are all symmetric, meaning that the insertion and removal of the view use the same transition. For example, if you apply the scale transition to a view, SwiftUI scales up the view when it's inserted in the view hierarchy. When it's removed, the framework scales it back down to the original size.

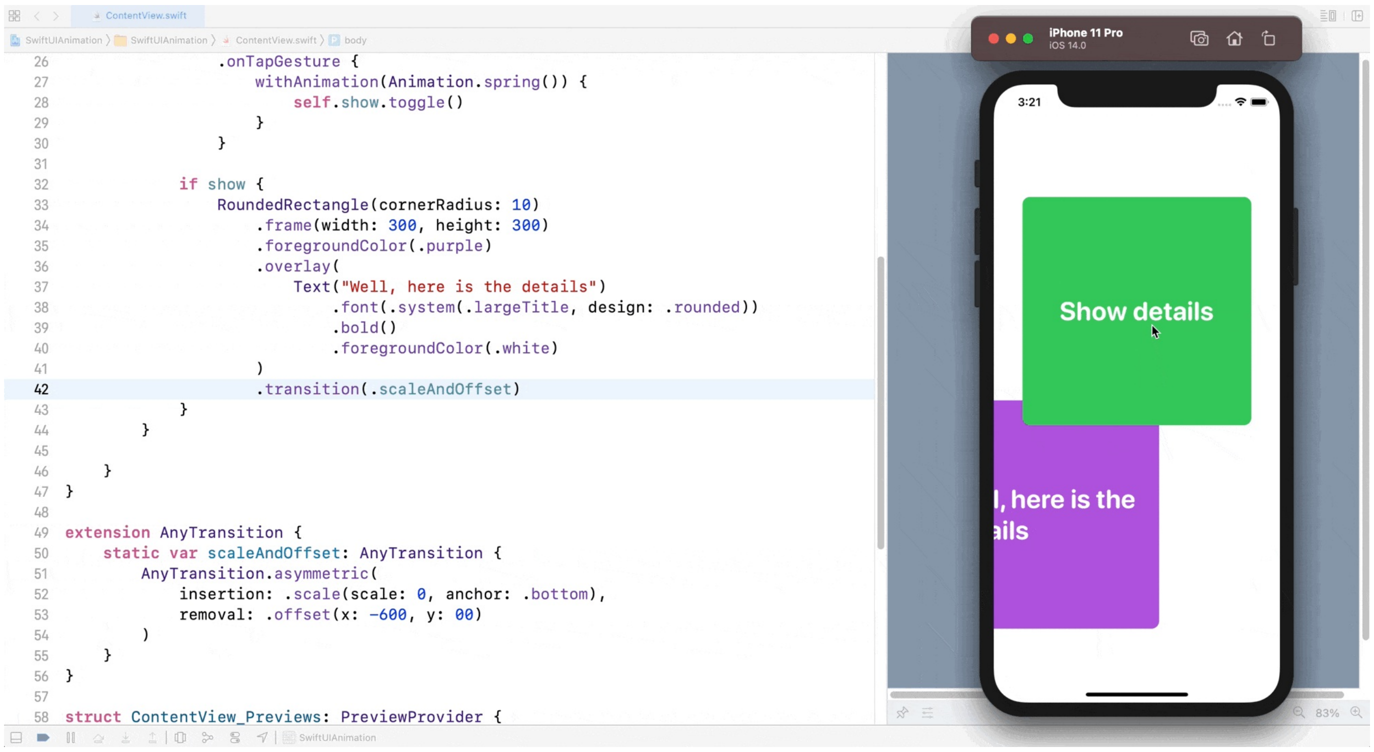
So, what if you want to use a *scale* transition when the view is inserted and an *offset* transition when the view is removed? This is known as *Assymetric Transitions* in SwiftUI. It's very simple to use this type of transition. You just need to call the `.assymetric` method and specify both the insertion & removal transitions. Here is the sample code:

```
.transition(.asymmetric(insertion: .scale(scale: 0, anchor: .bottom), removal: .of
fset(x: -600, y: 0)))
```

Again, if you need to reuse the transition, you can define an extension on `AnyTransition` like this:

```
extension AnyTransition {
    static var scaleAndOffset: AnyTransition {
        AnyTransition.asymmetric(
            insertion: .scale(scale: 0, anchor: .bottom),
            removal: .offset(x: -600, y: 00)
        )
    }
}
```

Add this code after the ContentView block and before the ContentView_Previews block. Run the app using the built-in simulator. You should see the *scale* transition when the purple square appears on screen. When you tap the rectangles again, the purple rectangle will slide off the screen.



```
                .onTapGesture {
                    withAnimation(Animation.spring()) {
                        self.show.toggle()
                    }
                }

            if show {
                RoundedRectangle(cornerRadius: 10)
                    .frame(width: 300, height: 300)
                    .foregroundColor(.purple)
                    .overlay(
                        Text("Well, here is the details")
                            .font(.system(.largeTitle, design: .rounded))
                            .bold()
                            .foregroundColor(.white)
                    )
                    .transition(.scaleAndOffset)
            }
        }

    }
}

extension AnyTransition {
    static var scaleAndOffset: AnyTransition {
        AnyTransition.asymmetric(
            insertion: .scale(scale: 0, anchor: .bottom),
            removal: .offset(x: -600, y: 00)
        )
    }
}

struct ContentView_Previews: PreviewProvider {
```

Figure 15. Assymetric transition demo

Exercise #1: Using Animation and Transition to Build a Fancy Button

Now that you have learned transitions and animations, let me challenge you to build a fancy button that displays the current state of an operation. If you can't see the animation below, please click this link (https://www.appcoda.com/wp-content/uploads/2019/10/swiftui-animation-16.gif) to see the animation.

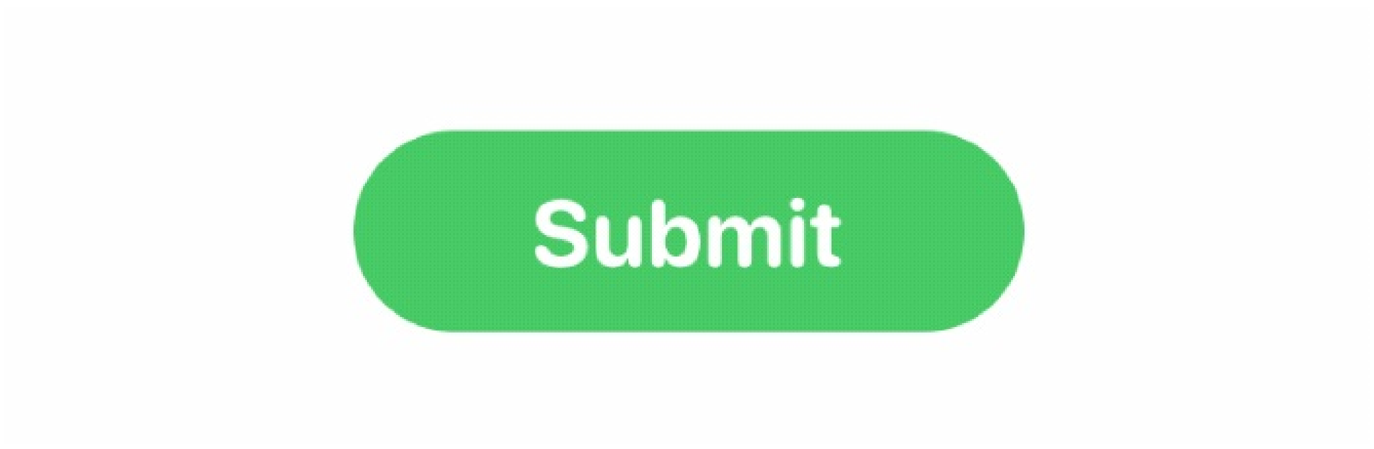


Figure 16. A fancy button

This button has three states:

- The original state: it shows a *Submit* button in green.
- The processing state: it displays a rotating circle and updates its label to *Processing*.
- The complete state: it displays the *Done* button in red.

It's quite a challenging project that will test your knowledge of SwiftUI animation and transition. You will need to combine everything you've learned so far to work out the solution.

In the demo button shown in figure 16, the processing takes around 4 seconds. You do not need to perform a real operation. To help you with this exercise, I use the following code to simulate an operation.

```
private func startProcessing() {
    self.loading = true

    // Simulate an operation by using DispatchQueue.main.asyncAfter
    // In a real world project, you will perform a task here.
    // When the task finishes, you set the completed status to true
    DispatchQueue.main.asyncAfter(deadline: .now() + 4) {
        self.completed = true
    }
}
```

Exercise #2: Animated View Transitions

You've learned how to implement view transitions. Try to integrate a transition with the card view project that you built in chapter 5 and create a view transition like below. When a user taps the card, the current view will scale down and fade away. The next view will be brought to the front with a scale-up animation.

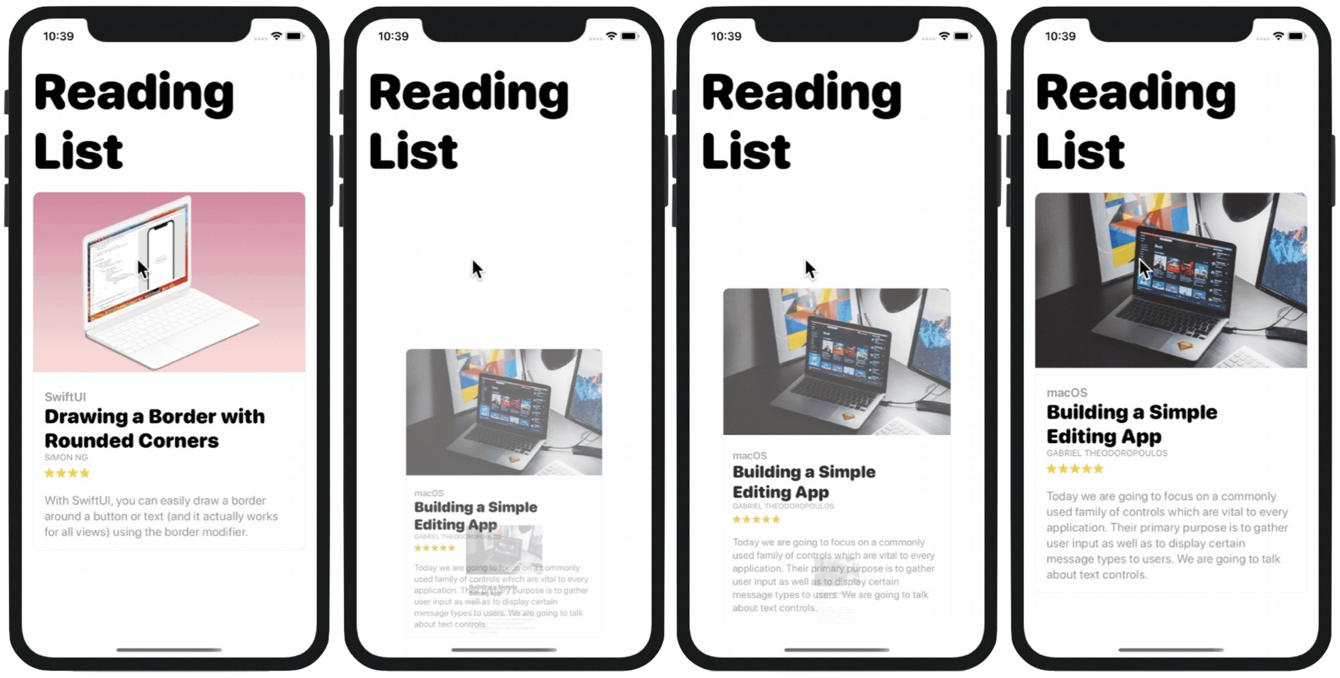

10:39
Reading List
SwiftUI
Drawing a Border with Rounded Corners
SIMON NG
With SwiftUI, you can easily draw a border around a button or text (and it actually works for all views) using the border modifier.
macOS
Building a Simple Editing App
GABRIEL THEODOROPOULOS
Today we are going to focus on a commonly used family of controls which are vital to every application. Their primary purpose is to gather user input as well as to display certain message types to users. We are going to talk about text controls.

Figure 17. Animated view transition

If you can't understand the animation above, you can click this link (https://www.appcoda.com/wp-content/uploads/2019/10/swiftui-view-animation.gif) to see the desired result.

Summary

Animation has a special role in mobile UI design. Well thought out animation improves user experience and brings meaning to UI interaction. A smooth and effortless transition between two views will delight and impress your users. With more than 2 million apps on the App Store, it's not easy to make your app stand out. However, a well-designed UI with animation will definitely make a difference!

Even for experienced developers, it's not an easy task to code slick animations. Fortunately, the SwiftUI framework has simplified the development of UI animation and transition. You tell the framework how the view should look at the beginning and the end. SwiftUI figures out the rest, rendering a smooth and nice animation.

In this chapter, I've walked you through the basics. But as you can see, you've already built some delightful animations and transitions. Most importantly, it needed just a few lines of code.

I hope you enjoyed reading this chapter and find the techniques useful. For reference, you can download the sample projects and solutions to exercises below:

- Demo projects & Exercise #1 (https://www.appcoda.com/resources/swiftui2/SwiftUIAnimation.zip)
- Exercise #2 (https://www.appcoda.com/resources/swiftui2/SwiftUICardAnimation.zip)

Chapter 10
Understanding List, ForEach and Identifiable

In UIKit, `UITableView` is one of the most common UI controls in iOS. If you've developed apps with UIKit before, you know that a table view can be used for presenting a list of data. This UI control is commonly found in content-based app such as newspaper apps. Figure 1 shows you some list/table views that you can find in popular apps like Instagram, Twitter, Airbnb, and Apple News.

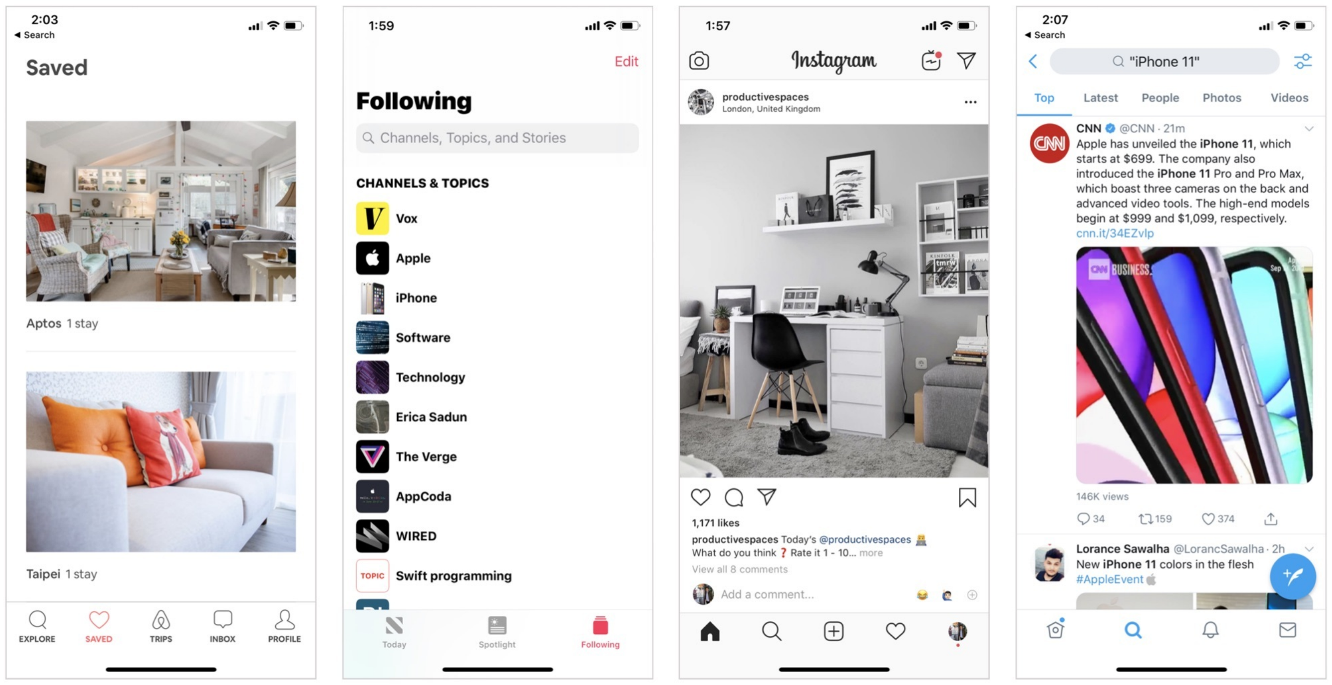


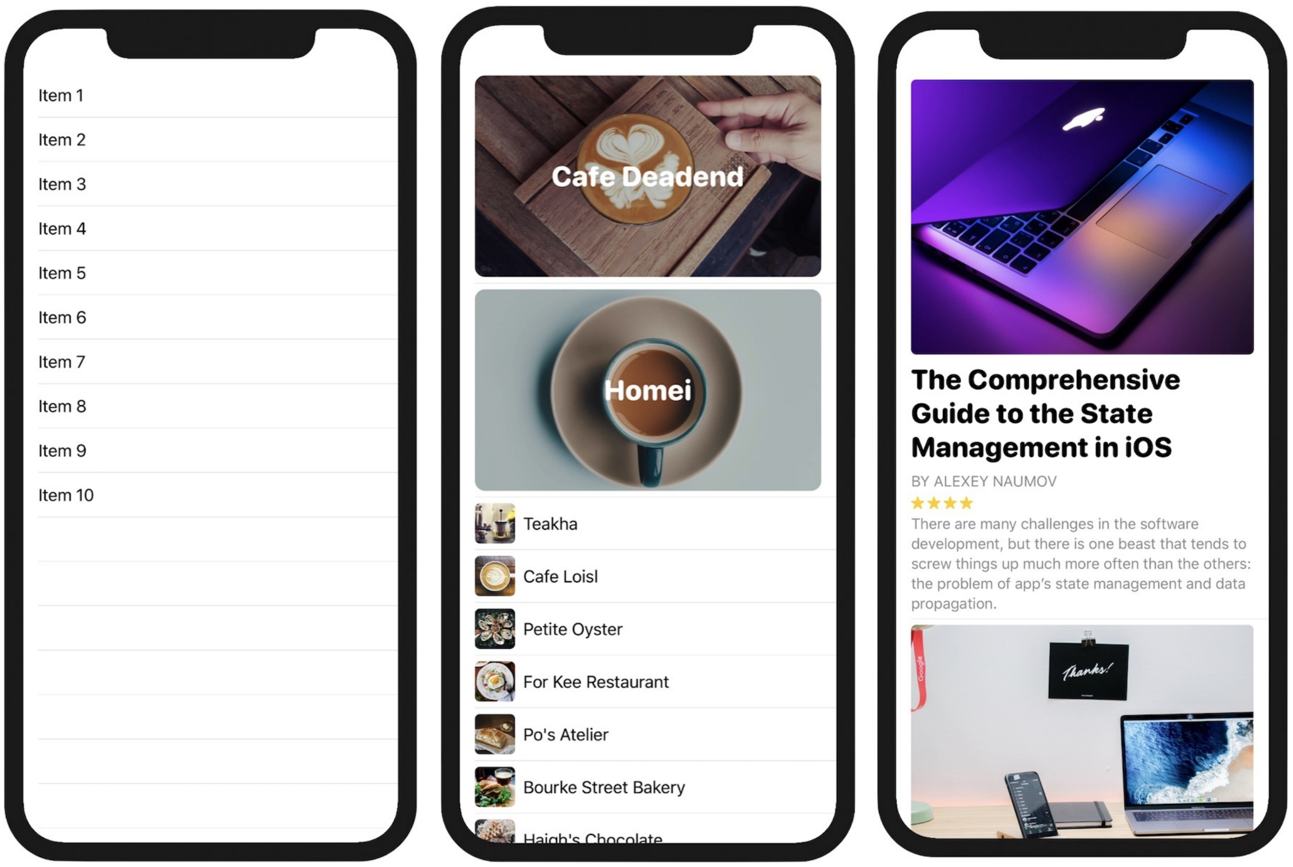
Figure 1. Sample list views

Instead of using `UITableView`, we use `List` in Swift UI to present rows of data. If you've built a table view with UIKit before, you know it'll take you a bit of work to implement a simple table view. It'll take even more effort to build a table view with custom cell layout.

SwiftUI simplifies this whole process. With just a few lines of code, you will be able to list data in table form. Even if you need to customize the layout of the rows, it only requires minimal effort.

Feeling confused? No worries. You'll understand what I mean in a while.

In this chapter, we will start with a simple list. Once you understand the basics, I will show you how to present a list of data with a more complex layout as shown in figure 2.



There are many challenges in the software development, but there is one beast that tends to screw things up much more often than the others: the problem of app's state management and data propagation.

Figure 2. Building a simple and complex list

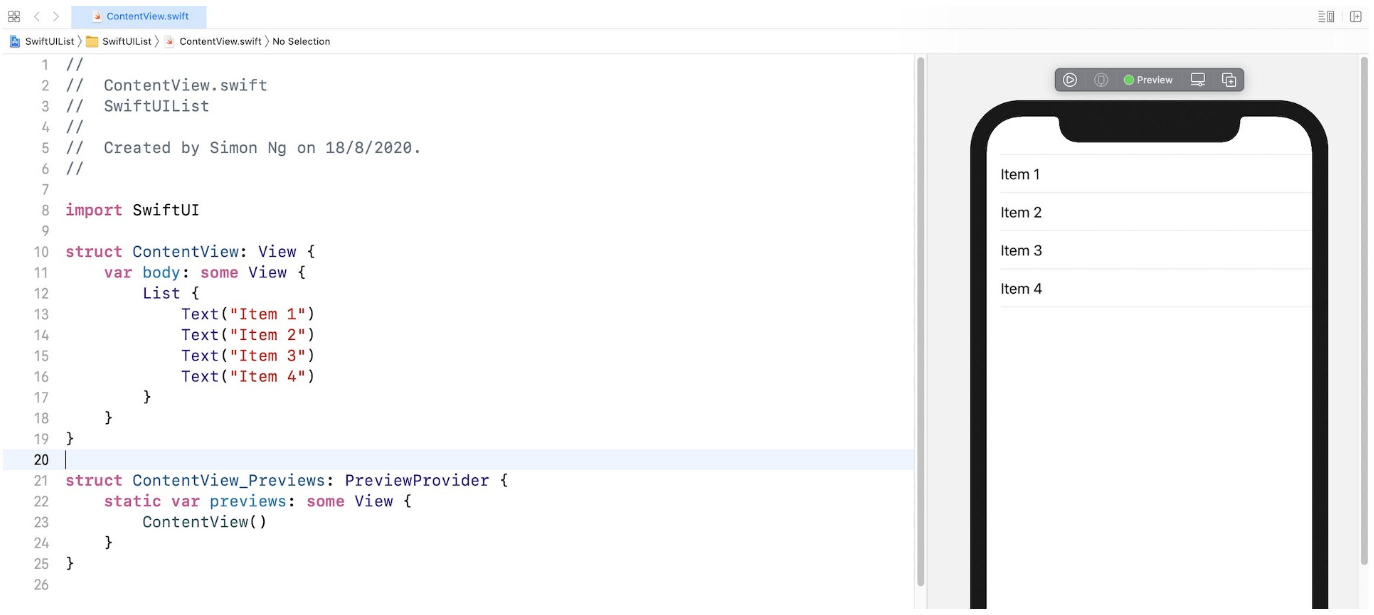
Creating a Simple List

Let's begin with a simple list. First, fire up Xcode and create a new project using the *App* template. In the next screen, set the product name to `SwiftUIList` (or whatever name you like) and fill in all the required values. Make sure you select `SwiftUI` for the *Interface* option.

Xcode will generate the "Hello World" code in the `ContentView.swift` file. Replace the "Hello World" text object with the following:

```
struct ContentView: View {
    var body: some View {
        List {
            Text("Item 1")
            Text("Item 2")
            Text("Item 3")
            Text("Item 4")
        }
    }
}
```

That's all the code you need to build a simple list or table. When you embed the text views in a `List`, the list view will present the data in rows. Here, each row shows a text view with different description.


ContentView.swift
SwiftUIList 〉 SwiftUIList 〉 ContentView.swift 〉 No Selection
Preview

```
//
//  ContentView.swift
//  SwiftUIList
//
//  Created by Simon Ng on 18/8/2020.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        List {
            Text("Item 1")
            Text("Item 2")
            Text("Item 3")
            Text("Item 4")
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Item 1
Item 2
Item 3
Item 4

Figure 3. Creating a simple list

The same code snippet can be written like this using `ForEach` :

```
struct ContentView: View {
    var body: some View {
        List {
            ForEach(1...4, id: \.self) { index in
                Text("Item \(index)")
            }
        }
    }
}
```

Since the text views are very similar, you can use `ForEach` in SwiftUI to create views in a loop.

> A structure that computes views on demand from an underlying collection of of identified data.
>
> \- Apple's official documentation (https://developer.apple.com/documentation/swiftui/foreach)

You can provide `ForEach` with a collection of data or a range. But one thing you have to take note of is that you need to tell `ForEach` how to identify each of the items in the collection. The parameter `id` is for this purpose. Why does `ForEach` need to identify the items uniquely? SwiftUI is powerful enough to update the UI automatically when some/all items in the collection are changed. To make this possible, it needs an identifier to uniquely identify the item when it's updated or removed.

In the code above, we pass `ForEach` a range of values to loop through. The identifier is set to the value itself (i.e. 1, 2, 3, or 4). The `index` parameter stores the current value of the loop. Say, for example, it starts with the value of 1. The `index` parameter will have a value of 1.

Within the closure, is the code you need to render the views. Here, we create the text view. Its description will change depending on the value of `index` in the loop. That's how you create 4 items in the list with different titles.

Let me show you one more technique. The same code snippet can be further rewritten like this:

```
struct ContentView: View {
    var body: some View {
        List {
            ForEach(1...4, id: \.self) {
                Text("Item \($0)")
            }
        }
    }
}
```

You can omit the `index` parameter and use the shorthand `$0`, which refers the first parameter of the closure.

Let's further rewrite the code to make it even more simple. You can pass the collection of data to the `List` view directly. Here is the code:

```
struct ContentView: View {
    var body: some View {
        List(1...4, id: \.self) {
            Text("Item \($0)")
        }
    }
}
```

As you can see, you only need a couple lines of code to build a simple list/table.

Creating a List View with Text and Images

Now that you know how to create a simple list, let's see how to work with a more complex layout. In most cases, the items of a list view contain both text and images. How do you implement that? If you know how `Image`, `Text`, `VStack`, and `HStack` work, you should have some ideas about how to create a complex list.

If you've read our book, Beginning iOS Programming with Swift, this example should be very familiar to you. Let's use it as an example and see how easy it is to build the same table with SwiftUI.

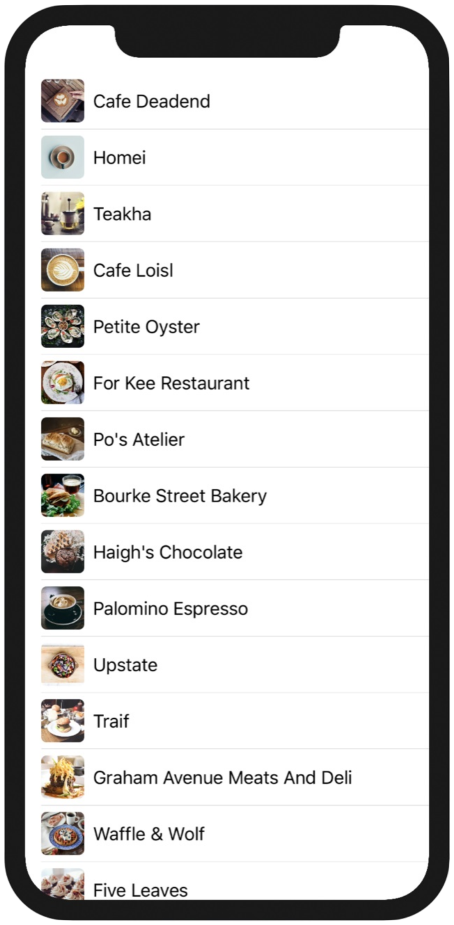


Figure 4. A simple table view showing rows of restaurants

To build the table using UIKit, you'll need to create a table view or table view controller and then customize the prototype cell. Furthermore, you'll have to code the table view data source to provide the data. That's quite a lot of steps to build a table UI. Let's see how the same table view is implemented in SwiftUI.

First, download the image pack from https://www.appcoda.com/resources/swiftui/SwiftUISimpleTableImages.zip. Unpack the zip file and import all the images to the asset catalog.

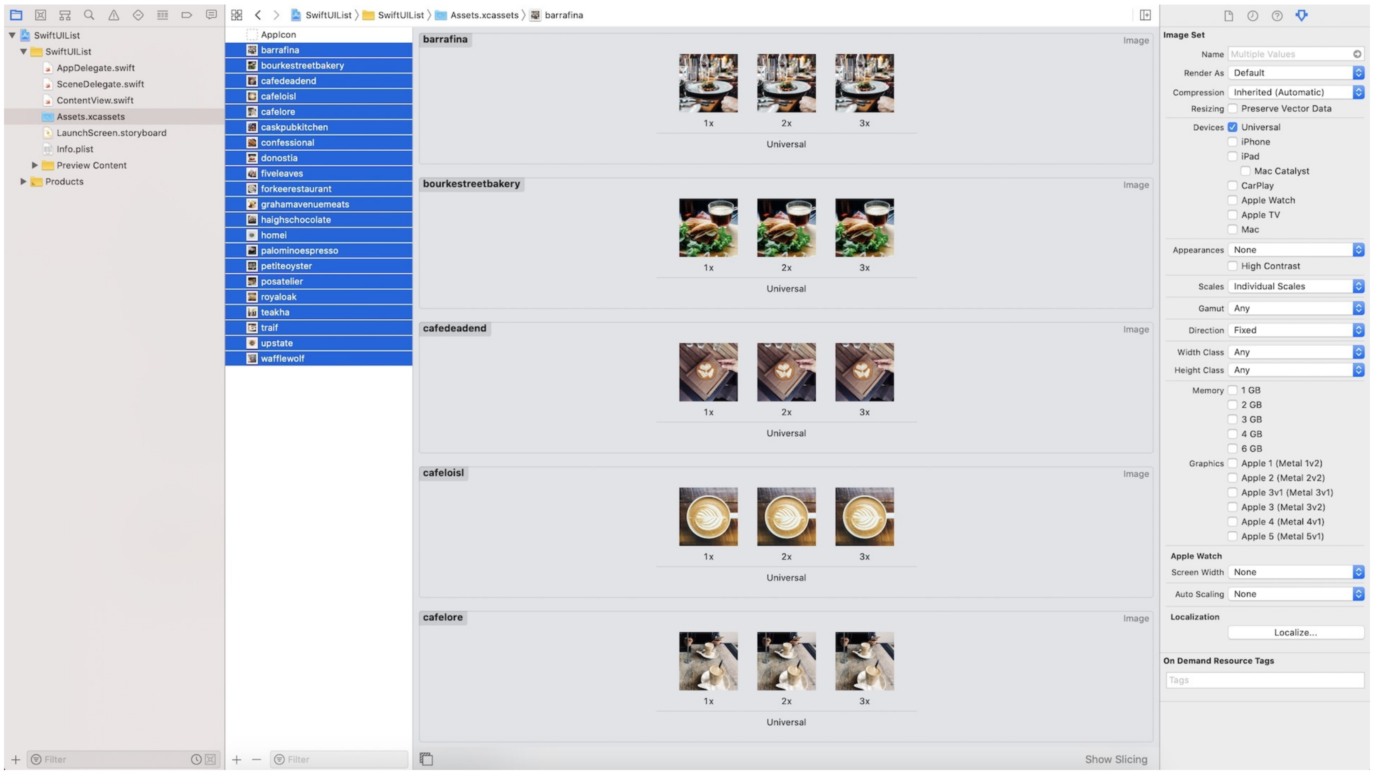


Figure 5. Import images to the asset catalog

Now switch over to `ContentView.swift` to code the UI. First, let's declare two arrays in `ContentView`. These arrays are for storing restaurant names and images. Here is the complete code:

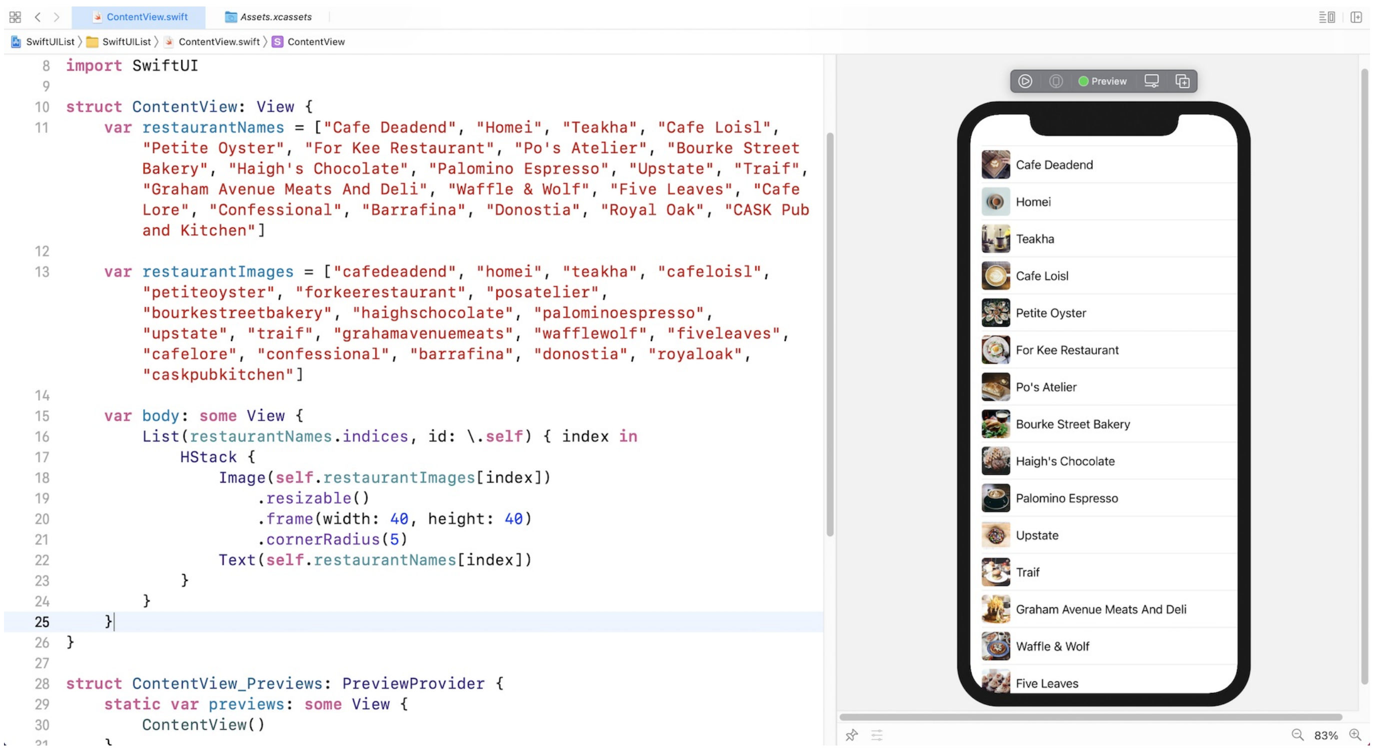
```
struct ContentView: View {

    var restaurantNames = ["Cafe Deadend", "Homei", "Teakha", "Cafe Loisl", "Petit
e Oyster", "For Kee Restaurant", "Po's Atelier", "Bourke Street Bakery", "Haigh's
Chocolate", "Palomino Espresso", "Upstate", "Traif", "Graham Avenue Meats And Deli"
, "Waffle & Wolf", "Five Leaves", "Cafe Lore", "Confessional", "Barrafina", "Donos
tia", "Royal Oak", "CASK Pub and Kitchen"]

    var restaurantImages = ["cafedeadend", "homei", "teakha", "cafeloisl", "petite
oyster", "forkeerestaurant", "posatelier", "bourkestreetbakery", "haighschocolate"
, "palominoespresso", "upstate", "traif", "grahamavenuemeats", "wafflewolf", "five
leaves", "cafelore", "confessional", "barrafina", "donostia", "royaloak", "caskpub
kitchen"]

    var body: some View {
        List(1...4, id: \.self) {
            Text("Item \($0)")
        }
    }
}
```

Both arrays have the same number of items. The `restaurantNames` array stores the name of the restaurants, the `restaurantImages` array stores the name of the images you just imported. To create a list view like that shown in figure 4, all you need to do is update the `body` variable like this:

```
var body: some View {
    List(restaurantNames.indices, id: \.self) { index in
        HStack {
            Image(self.restaurantImages[index])
                .resizable()
                .frame(width: 40, height: 40)
                .cornerRadius(5)
            Text(self.restaurantNames[index])
        }
    }
}
```

We've made a couple of changes. First, instead of a fixed range, we pass the array of restaurant names (i.e. `restaurantNames.indices`) to the `List` view. The `restaurantNames` array has 21 items so we'll have a range from 0 to 20 (arrays are 0 indexed). This only works when both arrays are of the same size as the index of one is used as an index for the other array.

In the closure, the code was updated to create the row layout. I'll not go into the details as the code is similar to previous stack views we've created. With less than 10 lines of code, we have created a list (or table) view with a custom layout.



```
import SwiftUI

struct ContentView: View {
    var restaurantNames = ["Cafe Deadend", "Homei", "Teakha", "Cafe Loisl",
        "Petite Oyster", "For Kee Restaurant", "Po's Atelier", "Bourke Street
        Bakery", "Haigh's Chocolate", "Palomino Espresso", "Upstate", "Traif",
        "Graham Avenue Meats And Deli", "Waffle & Wolf", "Five Leaves", "Cafe
        Lore", "Confessional", "Barrafina", "Donostia", "Royal Oak", "CASK Pub
        and Kitchen"]

    var restaurantImages = ["cafedeadend", "homei", "teakha", "cafeloisl",
        "petiteoyster", "forkeerestaurant", "posatelier",
        "bourkestreetbakery", "haighschocolate", "palominoespresso",
        "upstate", "traif", "grahamavenuemeats", "wafflewolf", "fiveleaves",
        "cafelore", "confessional", "barrafina", "donostia", "royaloak",
        "caskpubkitchen"]

    var body: some View {
        List(restaurantNames.indices, id: \.self) { index in
            HStack {
                Image(self.restaurantImages[index])
                    .resizable()
                    .frame(width: 40, height: 40)
                    .cornerRadius(5)
                Text(self.restaurantNames[index])
            }
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
```

Figure 6. A list view with custom row layout

Working with a Collection of Data

As mentioned before, `List` can take in a range or a collection of data. You've learned how to work with range. Let's see how to use `List` with an array of restaurant objects.

Instead of holding the restaurant data in two separate arrays, we'll create a `Restaurant` struct to better organize the data. This struct has two properties: *name* and *image*. Insert the following code at the end of the `ContentView.swift` file:

```
struct Restaurant {
    var name: String
    var image: String
}
```

With this struct, we can combine both `restaurantNames` and `restaurantImages` arrays into a single array. Delete the `restaurantNames` and `restaurantImages` variables and replace them with this variable in `ContentView` :

```
var restaurants = [ Restaurant(name: "Cafe Deadend", image: "cafedeadend"),
               Restaurant(name: "Homei", image: "homei"),
               Restaurant(name: "Teakha", image: "teakha"),
               Restaurant(name: "Cafe Loisl", image: "cafeloisl"),
               Restaurant(name: "Petite Oyster", image: "petiteoyster"),
               Restaurant(name: "For Kee Restaurant", image: "forkeerestaurant"),
               Restaurant(name: "Po's Atelier", image: "posatelier"),
               Restaurant(name: "Bourke Street Bakery", image: "bourkestreetbakery"
),
               Restaurant(name: "Haigh's Chocolate", image: "haighschocolate"),
               Restaurant(name: "Palomino Espresso", image: "palominoespresso"),
               Restaurant(name: "Upstate", image: "upstate"),
               Restaurant(name: "Traif", image: "traif"),
               Restaurant(name: "Graham Avenue Meats And Deli", image: "grahamaven
uemeats"),
               Restaurant(name: "Waffle & Wolf", image: "wafflewolf"),
               Restaurant(name: "Five Leaves", image: "fiveleaves"),
               Restaurant(name: "Cafe Lore", image: "cafelore"),
               Restaurant(name: "Confessional", image: "confessional"),
               Restaurant(name: "Barrafina", image: "barrafina"),
               Restaurant(name: "Donostia", image: "donostia"),
               Restaurant(name: "Royal Oak", image: "royaloak"),
               Restaurant(name: "CASK Pub and Kitchen", image: "caskpubkitchen")
]
```

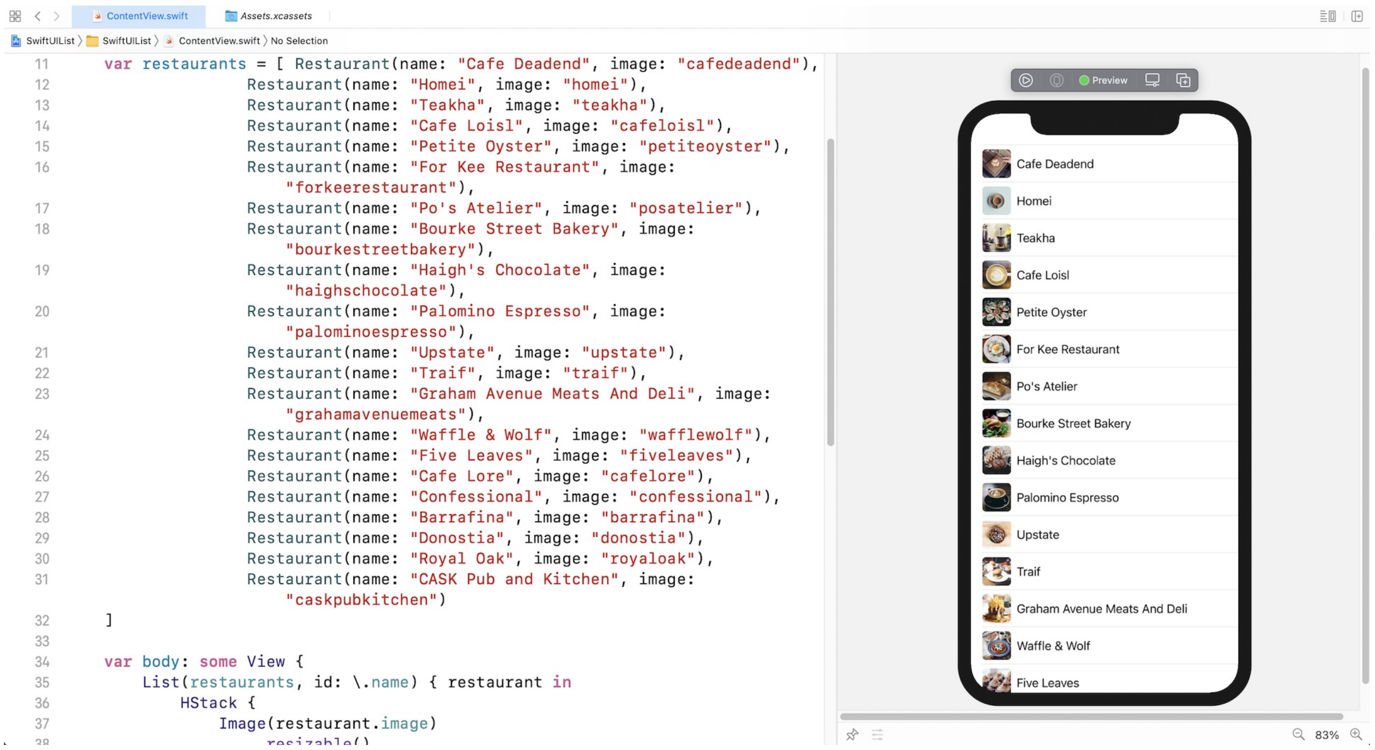
If you're new to Swift, each item of the array represents restaurant object containing both the name and image for each restauant. Once you have replaced the array, you'll see an error in Xcode, complaining that the `restaurantNames` variable is missing. That's expected because we've just removed it.

Now update the `body` variable like this:

```
var body: some View {
    List(restaurants, id: \.name) { restaurant in
        HStack {
            Image(restaurant.image)
                .resizable()
                .frame(width: 40, height: 40)
                .cornerRadius(5)
            Text(restaurant.name)
        }
    }
}
```

Take a look at the parameters we pass into `List`. Instead of passing the range, we pass the `restaurants` array and tell the `List` to use its `name` property as the identifier. The `List` will loop through the array and let us know the current `restaurant` it's handling in the closure. So, in the closure, we tell the list how we want to present the restaurant row. Here, we simply present both the restaurant image and name in a `HStack`.

The resultant UI is still the same but the underlying code was modified to utilize `List` with a collection of data.



```
    var restaurants = [ Restaurant(name: "Cafe Deadend", image: "cafedeadend"),
                        Restaurant(name: "Homei", image: "homei"),
                        Restaurant(name: "Teakha", image: "teakha"),
                        Restaurant(name: "Cafe Loisl", image: "cafeloisl"),
                        Restaurant(name: "Petite Oyster", image: "petiteoyster"),
                        Restaurant(name: "For Kee Restaurant", image:
                            "forkeerestaurant"),
                        Restaurant(name: "Po's Atelier", image: "posatelier"),
                        Restaurant(name: "Bourke Street Bakery", image:
                            "bourkestreetbakery"),
                        Restaurant(name: "Haigh's Chocolate", image:
                            "haighschocolate"),
                        Restaurant(name: "Palomino Espresso", image:
                            "palominoespresso"),
                        Restaurant(name: "Upstate", image: "upstate"),
                        Restaurant(name: "Traif", image: "traif"),
                        Restaurant(name: "Graham Avenue Meats And Deli", image:
                            "grahamavenuemeats"),
                        Restaurant(name: "Waffle & Wolf", image: "wafflewolf"),
                        Restaurant(name: "Five Leaves", image: "fiveleaves"),
                        Restaurant(name: "Cafe Lore", image: "cafelore"),
                        Restaurant(name: "Confessional", image: "confessional"),
                        Restaurant(name: "Barrafina", image: "barrafina"),
                        Restaurant(name: "Donostia", image: "donostia"),
                        Restaurant(name: "Royal Oak", image: "royaloak"),
                        Restaurant(name: "CASK Pub and Kitchen", image:
                            "caskpubkitchen")
    ]

    var body: some View {
        List(restaurants, id: \.name) { restaurant in
            HStack {
                Image(restaurant.image)
                     resizable()
```

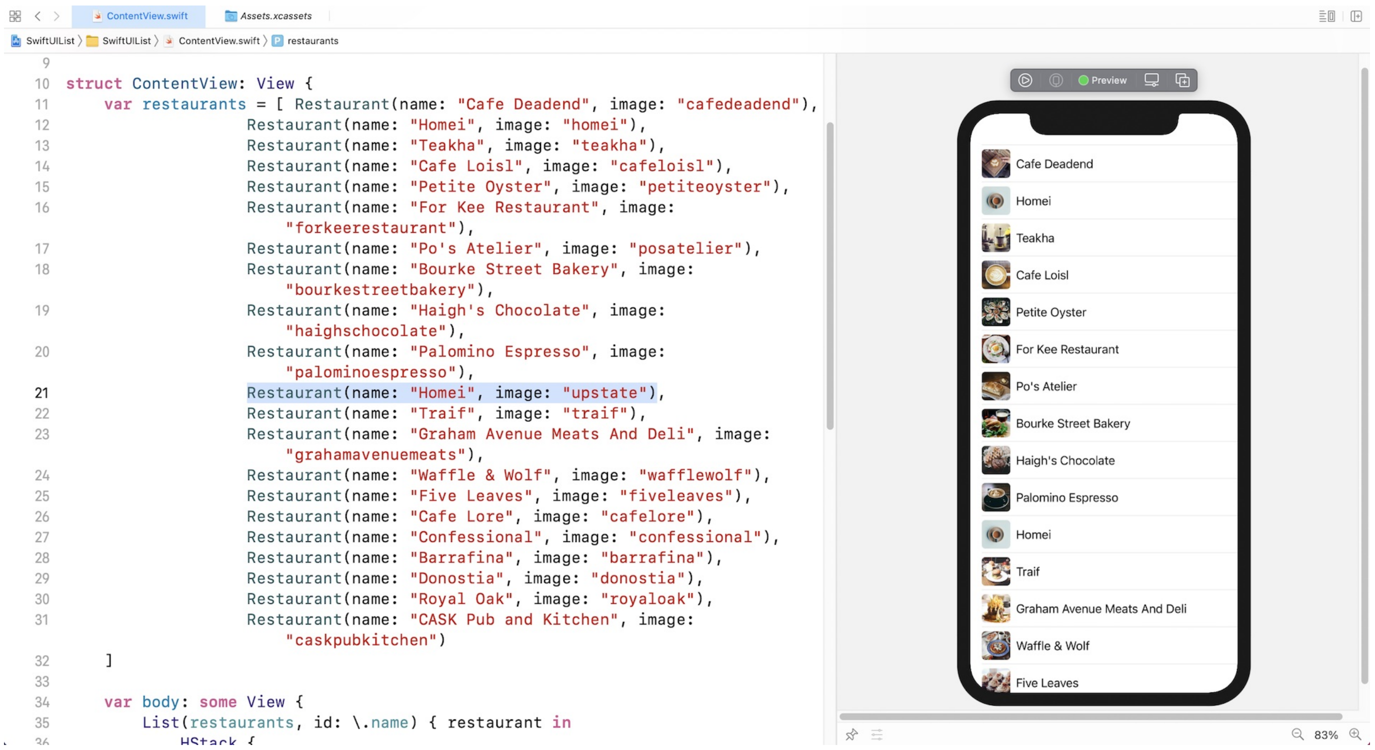
Figure 7. Same UI as figure 6

Working with the Identifiable Protocol

To help you better understand the purpose of the `id` parameter in `List`, let's make a minor change to the `restaurants` array. Currently, we use the name of the restaurant as an identifier. What happens when we have two records with the same restaurant name? Change *Upstate* (the 11th item in the array) to *Homei* in the `restaurants` array like this:

```
Restaurant(name: "Homei", image: "upstate")
```

Take note that we are only changing the value of the *name* property and keeping the image to `upstate`. The preview canvas should render the view automatically. If you see the message "Automatic preview updating paused", click the *Resume* button to reload the preview.



```
struct ContentView: View {
    var restaurants = [ Restaurant(name: "Cafe Deadend", image: "cafedeadend"),
                   Restaurant(name: "Homei", image: "homei"),
                   Restaurant(name: "Teakha", image: "teakha"),
                   Restaurant(name: "Cafe Loisl", image: "cafeloisl"),
                   Restaurant(name: "Petite Oyster", image: "petiteoyster"),
                   Restaurant(name: "For Kee Restaurant", image:
                       "forkeerestaurant"),
                   Restaurant(name: "Po's Atelier", image: "posatelier"),
                   Restaurant(name: "Bourke Street Bakery", image:
                       "bourkestreetbakery"),
                   Restaurant(name: "Haigh's Chocolate", image:
                       "haighschocolate"),
                   Restaurant(name: "Palomino Espresso", image:
                       "palominoespresso"),
                   Restaurant(name: "Homei", image: "upstate"),
                   Restaurant(name: "Traif", image: "traif"),
                   Restaurant(name: "Graham Avenue Meats And Deli", image:
                       "grahamavenuemeats"),
                   Restaurant(name: "Waffle & Wolf", image: "wafflewolf"),
                   Restaurant(name: "Five Leaves", image: "fiveleaves"),
                   Restaurant(name: "Cafe Lore", image: "cafelore"),
                   Restaurant(name: "Confessional", image: "confessional"),
                   Restaurant(name: "Barrafina", image: "barrafina"),
                   Restaurant(name: "Donostia", image: "donostia"),
                   Restaurant(name: "Royal Oak", image: "royaloak"),
                   Restaurant(name: "CASK Pub and Kitchen", image:
                       "caskpubkitchen")
    ]

    var body: some View {
        List(restaurants, id: \.name) { restaurant in
            HStack {
```

Cafe Deadend
Homei
Teakha
Cafe Loisl
Petite Oyster
For Kee Restaurant
Po's Atelier
Bourke Street Bakery
Haigh's Chocolate
Palomino Espresso
Homei
Traif
Graham Avenue Meats And Deli
Waffle & Wolf
Five Leaves

Figure 8. Two restaurants have the same name

Do you see the issue (in figure 8)? We now have two records with the name *Homei*. You might expect the second *Homei* record to show the *upstate* image, but iOS renders two records with the same text and image. In the code, we told the `List` to use the restaurant's name as the unique identifier. When two restaurants have the same name, iOS considers both restaurants to be the same restaurant. Thus, it reuses the same view and renders the same image.

So, how do you fix this issue?

That's pretty easy. Instead of using the name as the identifier (ID), you should give each restaurant a unique identifier. Update the `Restaurant` struct like this:

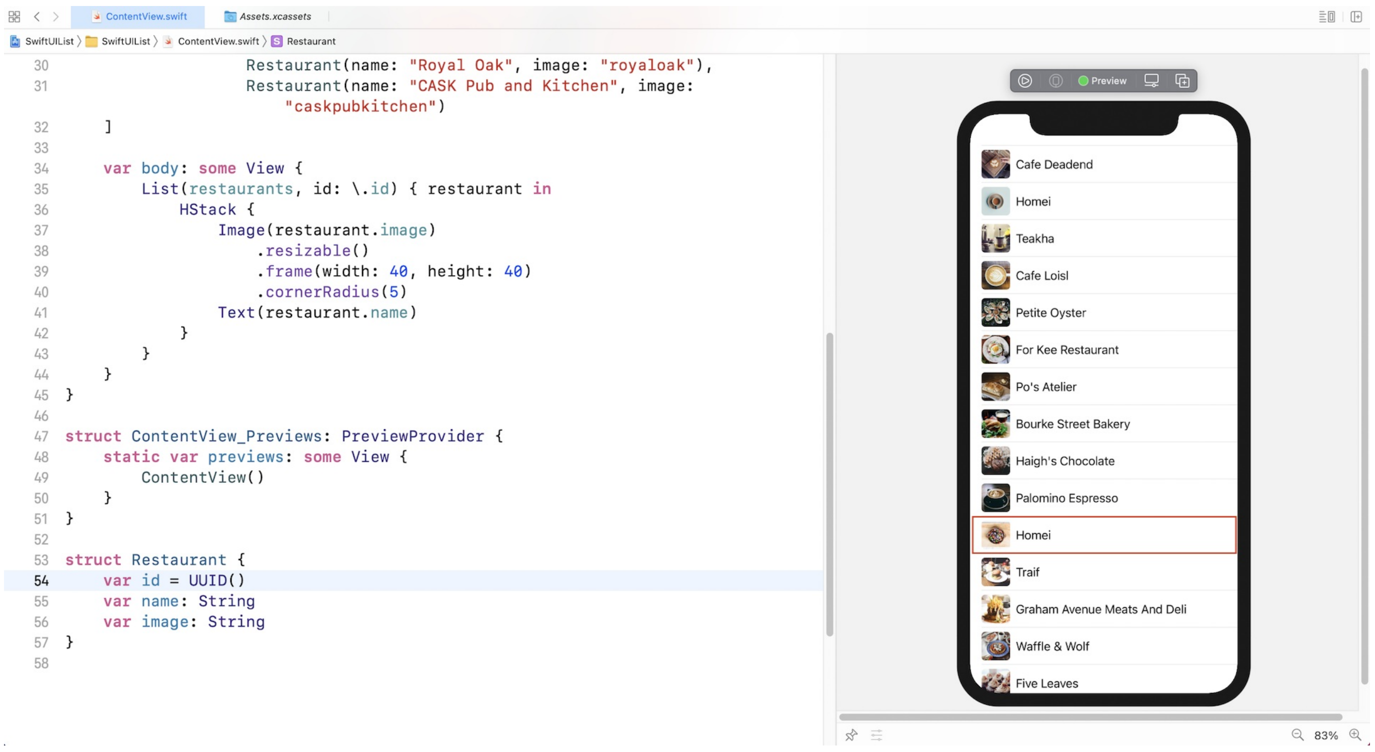
```
struct Restaurant {
    var id = UUID()
    var name: String
    var image: String
}
```

In the code, we added an `id` property and initialized it with a unique identifier. The `UUID()` function is designed to generate a random identifier that is universally unique. A UUID is composed of 128-bit number, so theoretically the chance of having two same indentifers is almost zero.

Now each restaurant has a unique ID, but we still have to make one more change for things to work. For the `List`, change the value of the `id` parameter from `\.name` to `\.id`:

```
List(restaurants, id: \.id)
```

This tells the `List` view to use the `id` property of the restaurants as the unique identifier. Take a look at the preview, the second *Homei* record now shows the *upstate* image.



```
                    Restaurant(name: "Royal Oak", image: "royaloak"),
                    Restaurant(name: "CASK Pub and Kitchen", image:
                        "caskpubkitchen")
    ]

    var body: some View {
        List(restaurants, id: \.id) { restaurant in
            HStack {
                Image(restaurant.image)
                    .resizable()
                    .frame(width: 40, height: 40)
                    .cornerRadius(5)
                Text(restaurant.name)
            }
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}

struct Restaurant {
    var id = UUID()
    var name: String
    var image: String
}
```

Figure 9. The bug is now fixed showing the correct image

We can further simplify the code by making the `Restaurant` struct conform to the `Identifiable` protocol. This protocol has only one requirement, that the type implementing the protocol should have some sort of `id` as a unique identifier. Update `Restaurant` to implement the `Identifiable` protocol like this:

```
struct Restaurant: Identifiable {
    var id = UUID()
    var name: String
    var image: String
}
```

Since `Restaurant` already provides a unique `id` property, this conforms to the protocol requirement.

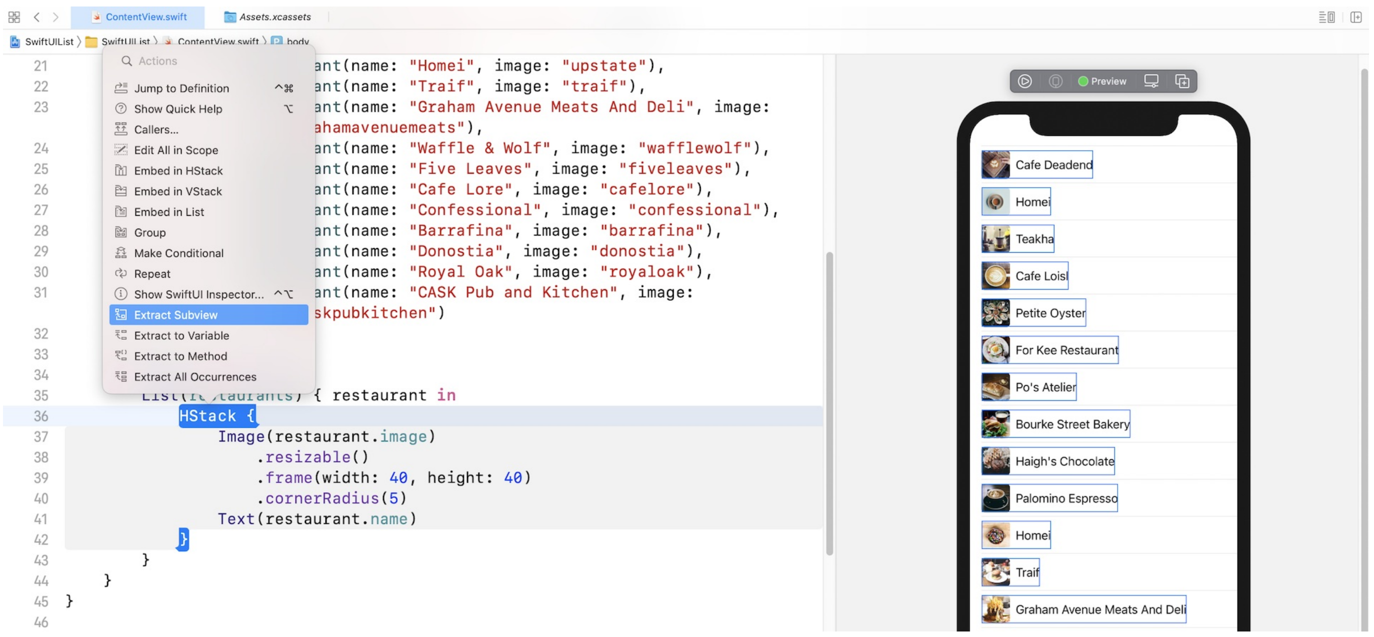
What's the purpose of implementing the `Identifiable` protocol here? With the `Restaurant` struct conforming to the `Identifiable` protocol, you can initialize the `List` without the `id` parameter. You just simplified the code! Here is the updated code for the list view:

```
List(restaurants) { restaurant in
    HStack {
        Image(restaurant.image)
            .resizable()
            .frame(width: 40, height: 40)
            .cornerRadius(5)
        Text(restaurant.name)
    }
}
```

That's how you use `List` to present a collection of data.

Refactoring the Code

The code works but it's always good coding practice to refactor the code to make it even better. You've learned how to extract a view. Let's extract the `HStack` into a separate struct. Hold the command key and click `HStack`. Select *Extract subview* to extract the code. Rename the struct to `BasicImageRow` .



```
ant(name: "Homei", image: "upstate"),
ant(name: "Traif", image: "traif"),
ant(name: "Graham Avenue Meats And Deli", image:
ahamavenuemeats"),
ant(name: "Waffle & Wolf", image: "wafflewolf"),
ant(name: "Five Leaves", image: "fiveleaves"),
ant(name: "Cafe Lore", image: "cafelore"),
ant(name: "Confessional", image: "confessional"),
ant(name: "Barrafina", image: "barrafina"),
ant(name: "Donostia", image: "donostia"),
ant(name: "Royal Oak", image: "royaloak"),
ant(name: "CASK Pub and Kitchen", image:
skpubkitchen")



        List(restaurants) { restaurant in
            HStack {
                Image(restaurant.image)
                    .resizable()
                    .frame(width: 40, height: 40)
                    .cornerRadius(5)
                Text(restaurant.name)
            }
        }
    }
}
```

Figure 10. Extracting subview

Xcode immediately shows you an error once you made the change. Since the extracted subview doesn't have a `restaurant` property, update the `BasicImageRow` struct like this to declare the `restaurant` property:

```
struct BasicImageRow: View {
    var restaurant: Restaurant

    var body: some View {
        HStack {
            Image(restaurant.image)
                .resizable()
                .frame(width: 40, height: 40)
                .cornerRadius(5)
            Text(restaurant.name)
        }
    }
}
```

Next, update the `List` view to pass the `restaurant` parameter:

```
List(restaurants) { restaurant in
    BasicImageRow(restaurant: restaurant)
}
```

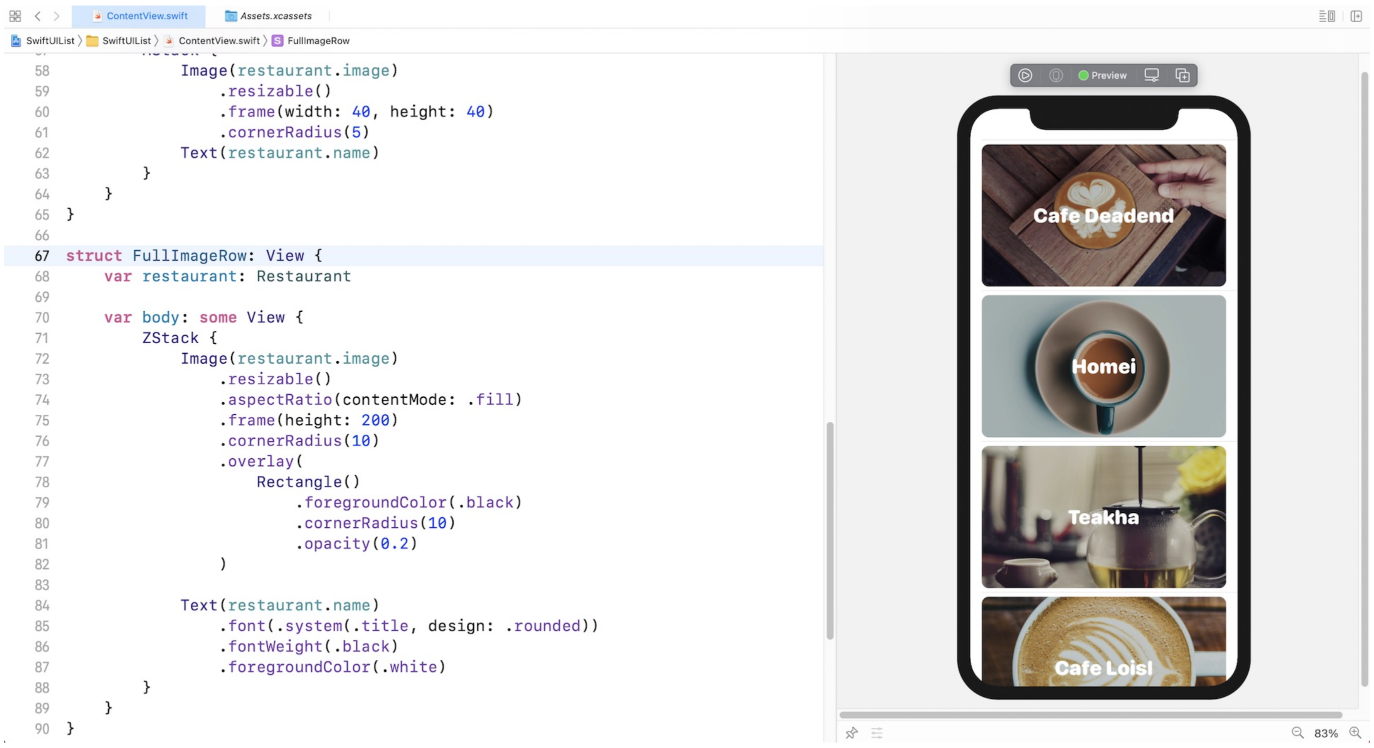
Now everything should work without errors. The list view still looks the same but the underlying code is more readable and organized. It's also more adaptable to code change. Let's say, you create another layout for the row like this:

```
struct FullImageRow: View {
    var restaurant: Restaurant

    var body: some View {
        ZStack {
            Image(restaurant.image)
                .resizable()
                .aspectRatio(contentMode: .fill)
                .frame(height: 200)
                .cornerRadius(10)
                .overlay(
                    Rectangle()
                        .foregroundColor(.black)
                        .cornerRadius(10)
                        .opacity(0.2)
                )

            Text(restaurant.name)
                .font(.system(.title, design: .rounded))
                .fontWeight(.black)
                .foregroundColor(.white)
        }
    }
}
```

This row layout is designed to show a larger restaurant with the restaurant name overlayed on top. Since we've refactored our code, it's very easy to change the app to use the new layout. All you need to do is replace `BasicImageRow` with `FullImageRow` in the closure of `List` :

```
List(restaurants) { restaurant in
    FullImageRow(restaurant: restaurant)
}
```

By changing one line of code, the app instantly switches to another layout.



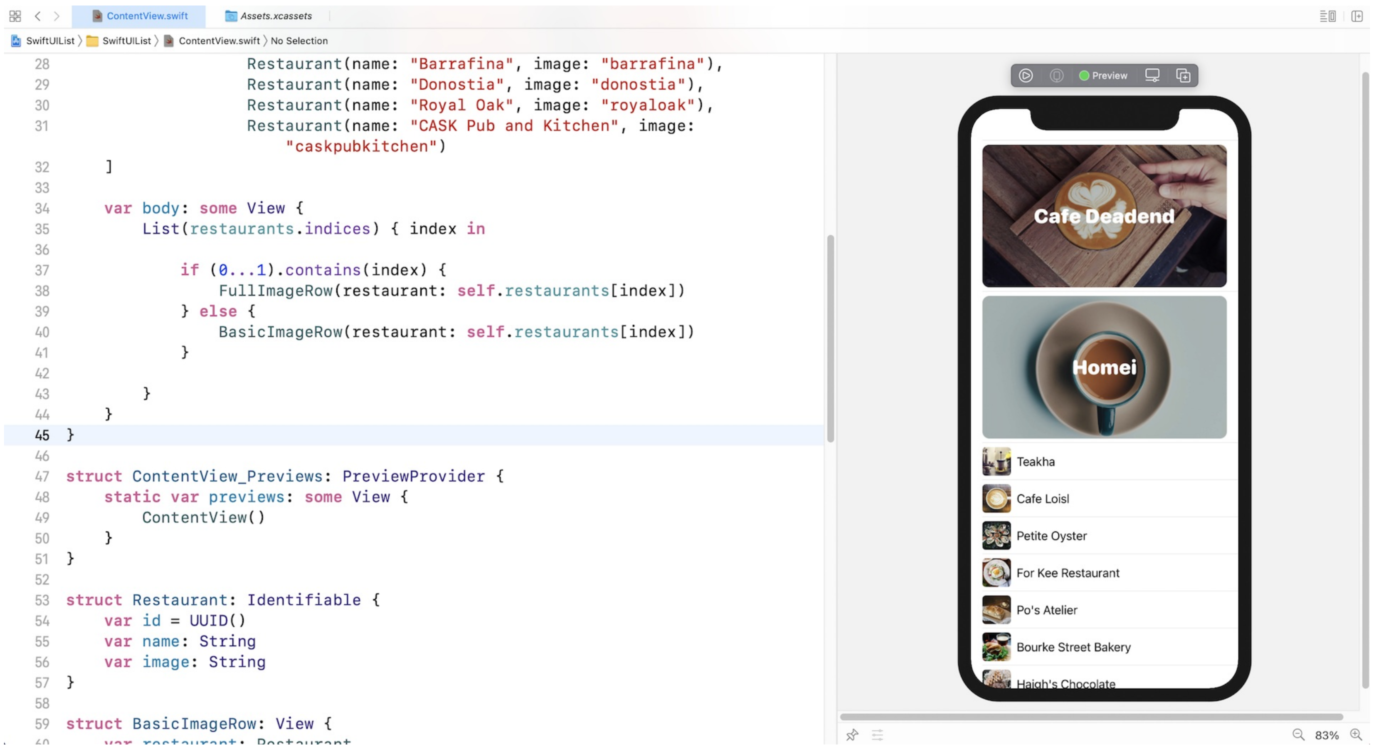
```
            Image(restaurant.image)
                .resizable()
                .frame(width: 40, height: 40)
                .cornerRadius(5)
            Text(restaurant.name)
        }
    }
}

struct FullImageRow: View {
    var restaurant: Restaurant

    var body: some View {
        ZStack {
            Image(restaurant.image)
                .resizable()
                .aspectRatio(contentMode: .fill)
                .frame(height: 200)
                .cornerRadius(10)
                .overlay(
                    Rectangle()
                        .foregroundColor(.black)
                        .cornerRadius(10)
                        .opacity(0.2)
                )

            Text(restaurant.name)
                .font(.system(.title, design: .rounded))
                .fontWeight(.black)
                .foregroundColor(.white)
        }
    }
}
```

Figure 11. Changing the row layout

You can further mix the row layouts to build a more interesting UI. For example, our list is to use `FullImageRow` for the first two rows of data and the rest of the rows will utilize the `BasicImageRow` . To do this, you update `List` like this:

```
List(restaurants.indices) { index in

    if (0...1).contains(index) {
        FullImageRow(restaurant: self.restaurants[index])
    } else {
        BasicImageRow(restaurant: self.restaurants[index])
    }

}
```

Since we need to retrieve the index of the rows, we pass the `List` the index range of the restaurant data. In the closure, we check the value of `index` to determine which row layout to use.



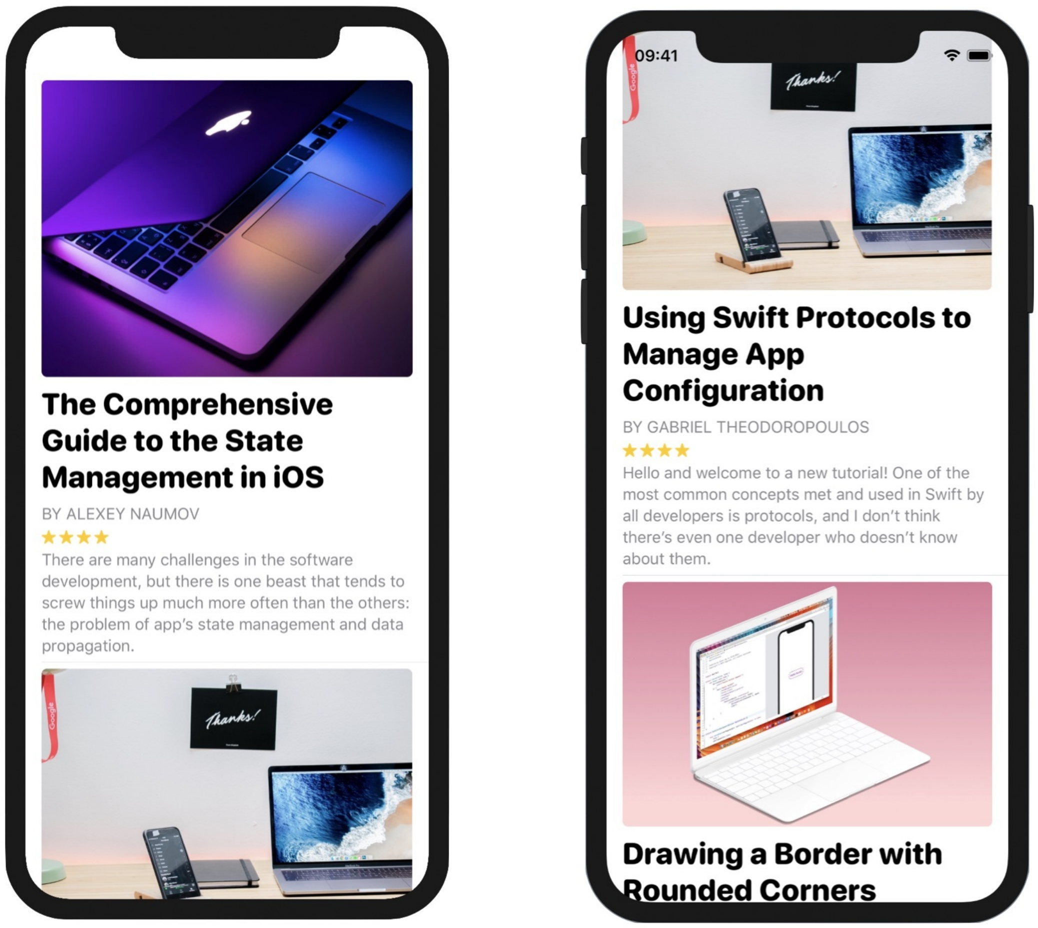
```
                    Restaurant(name: "Barrafina", image: "barrafina"),
                    Restaurant(name: "Donostia", image: "donostia"),
                    Restaurant(name: "Royal Oak", image: "royaloak"),
                    Restaurant(name: "CASK Pub and Kitchen", image:
                        "caskpubkitchen")
    ]

    var body: some View {
        List(restaurants.indices) { index in

            if (0...1).contains(index) {
                FullImageRow(restaurant: self.restaurants[index])
            } else {
                BasicImageRow(restaurant: self.restaurants[index])
            }

        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}

struct Restaurant: Identifiable {
    var id = UUID()
    var name: String
    var image: String
}

struct BasicImageRow: View {
    var restaurant: Restaurant
```

Figure 12. Building a list view with two different row layouts

Exercise

Before you move on to the next chapter, challenge yourself by building the list view shown in figure 13. It looks complicated but if you fully understand this chapter, you should be able to build the UI. Take some time to work on this exercise. I guarantee you'll learn a lot!

To save you time finding your own images, you can download the image pack for this exercise from https://www.appcoda.com/resources/swiftui/SwiftUIArticleImages.zip.



The Comprehensive Guide to the State Management in iOS

BY ALEXEY NAUMOV

★★★★

There are many challenges in the software development, but there is one beast that tends to screw things up much more often than the others: the problem of app's state management and data propagation.

09:41

Using Swift Protocols to Manage App Configuration

BY GABRIEL THEODOROPOULOS

★★★★

Hello and welcome to a new tutorial! One of the most common concepts met and used in Swift by all developers is protocols, and I don't think there's even one developer who doesn't know about them.

Drawing a Border with Rounded Corners

Figure 13. Building a list view with complex row layout

For reference, you can download the complete list project and solution to the exercise here:

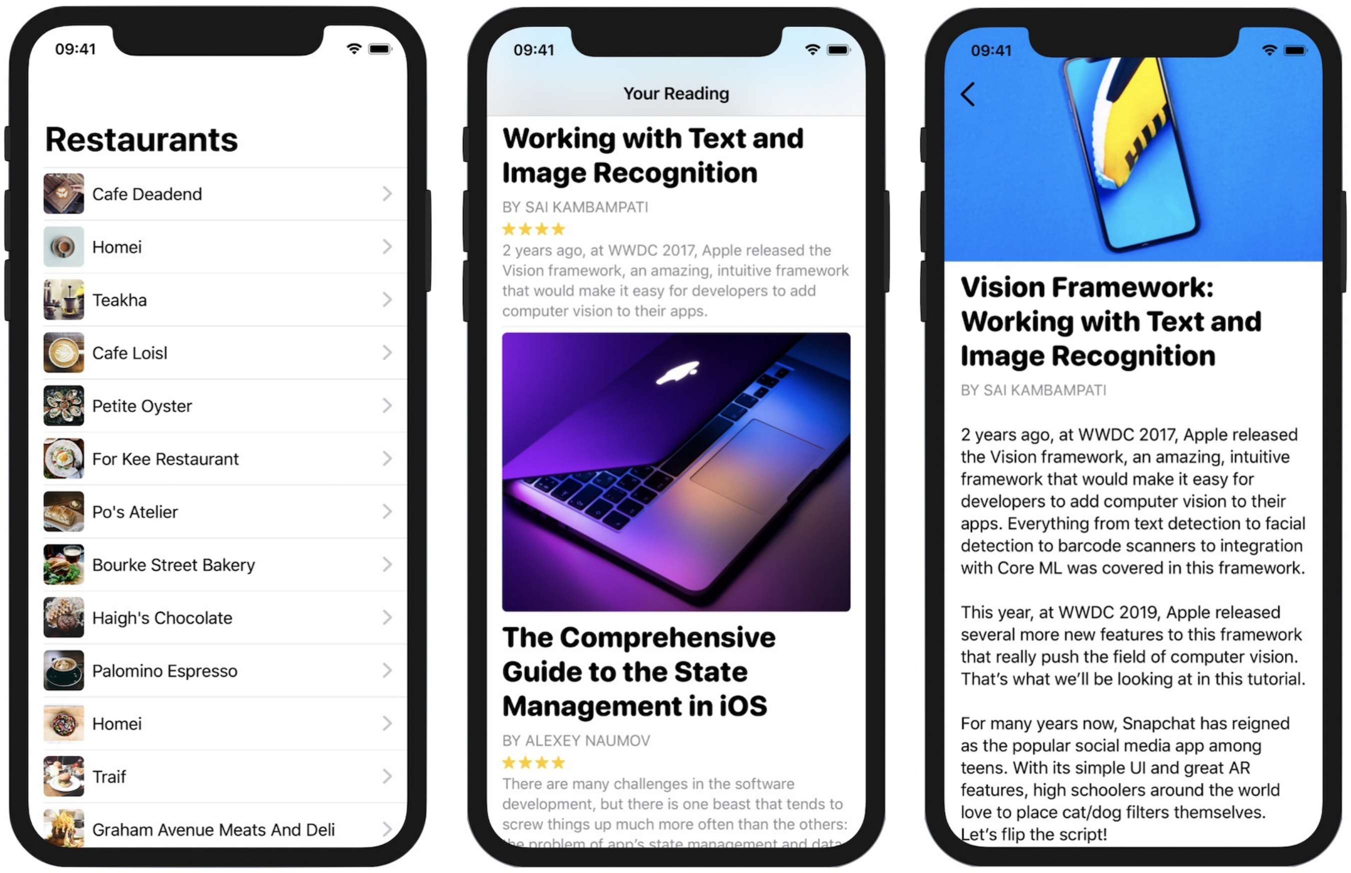
- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUIList.zip)
- Solution to exercise

(https://www.appcoda.com/resources/swiftui2/SwiftUIListExercise.zip)

Chapter 11
Working with Navigation UI and Navigation Bar Customization

In most apps, you will have experienced a navigational interface. This kind of UI ty[ically has a navigation bar and a list of data. It allows users navigate to a detail view when tapping the content.

In UIKit, we implement this type of interface using UINavigationController. For SwiftUI, Apple calls it NavigationView. In this chapter, I will walk you through the implementation of NavigationView and show you how to perform some customizations. As usual, we will work on a couple of demo projects so you'll get some hands on experience with NavigationView.


09:41

Restaurants

Cafe Deadend
Homei
Teakha
Cafe Loisl
Petite Oyster
For Kee Restaurant
Po's Atelier
Bourke Street Bakery
Haigh's Chocolate
Palomino Espresso
Homei
Traif
Graham Avenue Meats And Deli

09:41

Your Reading

Working with Text and Image Recognition

BY SAI KAMBAMPATI

★★★★

2 years ago, at WWDC 2017, Apple released the Vision framework, an amazing, intuitive framework that would make it easy for developers to add computer vision to their apps.

The Comprehensive Guide to the State Management in iOS

BY ALEXEY NAUMOV

★★★★

There are many challenges in the software development, but there is one beast that tends to screw things up much more often than the others: the problem of app's state management and data

09:41

Vision Framework: Working with Text and Image Recognition

BY SAI KAMBAMPATI

2 years ago, at WWDC 2017, Apple released the Vision framework, an amazing, intuitive framework that would make it easy for developers to add computer vision to their apps. Everything from text detection to facial detection to barcode scanners to integration with Core ML was covered in this framework.

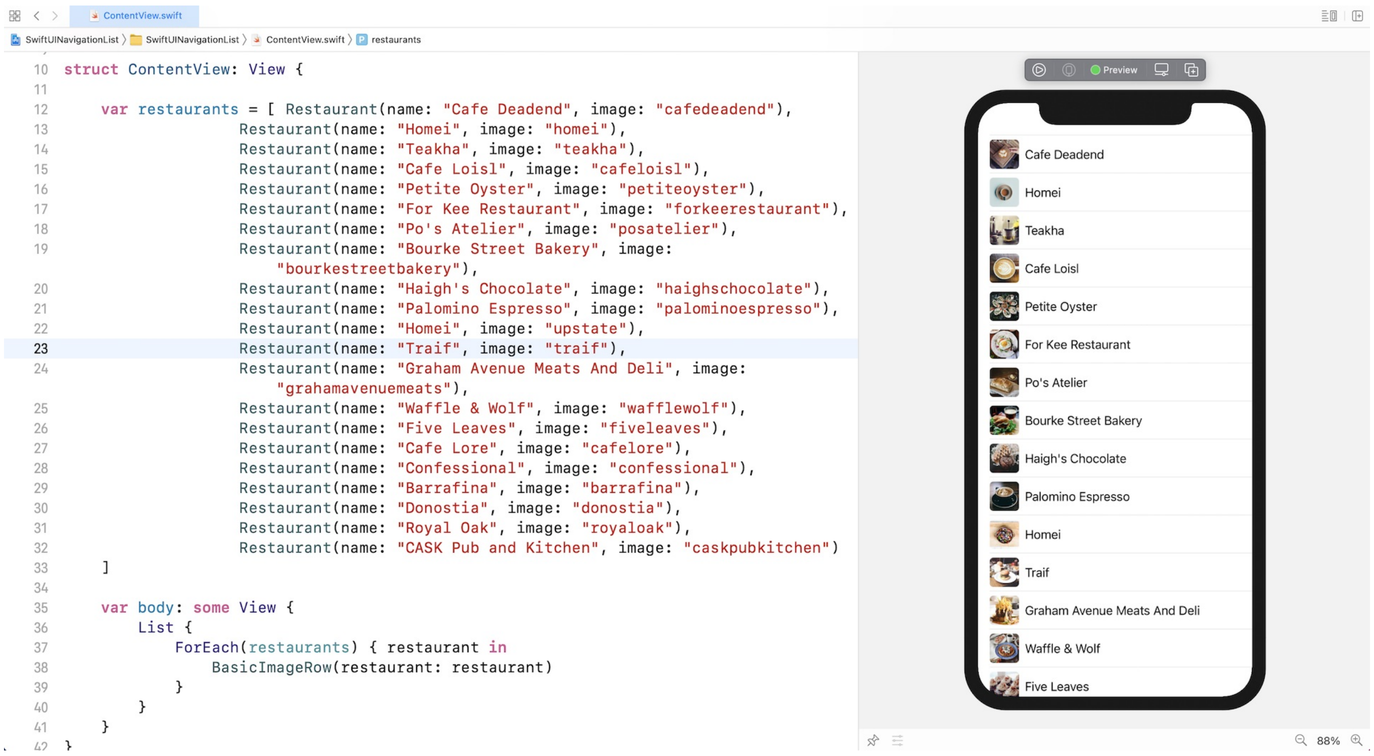
This year, at WWDC 2019, Apple released several more new features to this framework that really push the field of computer vision. That's what we'll be looking at in this tutorial.

For many years now, Snapchat has reigned as the popular social media app among teens. With its simple UI and great AR features, high schoolers around the world love to place cat/dog filters themselves. Let's flip the script!

Figure 1. Sample navigation interface for our demo projects

Preparing the Starter Project

Let's get started and implement a demo project that we have built earlier with a navigation UI. So, first download the starter project from https://www.appcoda.com/resources/swiftui2/SwiftUINavigationListStarter.zip. Once downloaded, open the project and check out the preview. You should be very familiar with this demo app. It just displays a list of restaurants.



```
struct ContentView: View {

    var restaurants = [ Restaurant(name: "Cafe Deadend", image: "cafedeadend"),
                        Restaurant(name: "Homei", image: "homei"),
                        Restaurant(name: "Teakha", image: "teakha"),
                        Restaurant(name: "Cafe Loisl", image: "cafeloisl"),
                        Restaurant(name: "Petite Oyster", image: "petiteoyster"),
                        Restaurant(name: "For Kee Restaurant", image: "forkeerestaurant"),
                        Restaurant(name: "Po's Atelier", image: "posatelier"),
                        Restaurant(name: "Bourke Street Bakery", image:
                            "bourkestreetbakery"),
                        Restaurant(name: "Haigh's Chocolate", image: "haighschocolate"),
                        Restaurant(name: "Palomino Espresso", image: "palominoespresso"),
                        Restaurant(name: "Homei", image: "upstate"),
                        Restaurant(name: "Traif", image: "traif"),
                        Restaurant(name: "Graham Avenue Meats And Deli", image:
                            "grahamavenuemeats"),
                        Restaurant(name: "Waffle & Wolf", image: "wafflewolf"),
                        Restaurant(name: "Five Leaves", image: "fiveleaves"),
                        Restaurant(name: "Cafe Lore", image: "cafelore"),
                        Restaurant(name: "Confessional", image: "confessional"),
                        Restaurant(name: "Barrafina", image: "barrafina"),
                        Restaurant(name: "Donostia", image: "donostia"),
                        Restaurant(name: "Royal Oak", image: "royaloak"),
                        Restaurant(name: "CASK Pub and Kitchen", image: "caskpubkitchen")
    ]

    var body: some View {
        List {
            ForEach(restaurants) { restaurant in
                BasicImageRow(restaurant: restaurant)
            }
        }
    }
}
```

Figure 2. The starter project should display a simple list view

What we're going to do is embed this list view in a navigation view.

Implementing a Navigation View

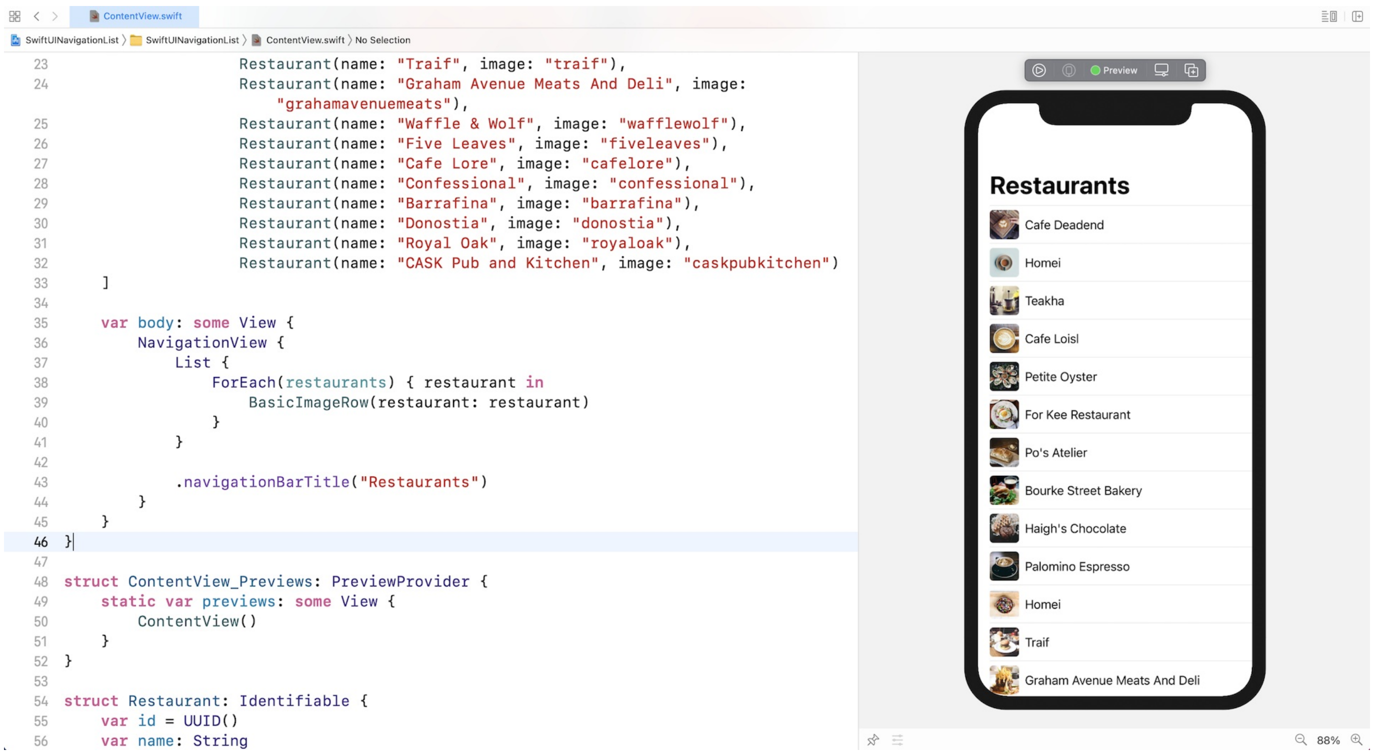
The SwiftUI framework provides a view called `NavigationView` for you to create a navigation UI. To embed the list view in a `NavigationView`, all you need to do is wrap the `List` with a `NavigationView` like this:

```
NavigationView {
    List {
        ForEach(restaurants) { restaurant in
            BasicImageRow(restaurant: restaurant)
        }
    }
}
```

Once you have made the change, you should see an empty navigation bar. To assign a title to the bar, insert the `navigationBarTitle` modifier like below:

```
NavigationView {
    List {
        ForEach(restaurants) { restaurant in
            BasicImageRow(restaurant: restaurant)
        }
    }

    .navigationBarTitle("Restaurants")
}
```

Now the app has a navigation bar with a large title.



```
                        Restaurant(name: "Traif", image: "traif"),
                        Restaurant(name: "Graham Avenue Meats And Deli", image:
                            "grahamavenuemeats"),
                        Restaurant(name: "Waffle & Wolf", image: "wafflewolf"),
                        Restaurant(name: "Five Leaves", image: "fiveleaves"),
                        Restaurant(name: "Cafe Lore", image: "cafelore"),
                        Restaurant(name: "Confessional", image: "confessional"),
                        Restaurant(name: "Barrafina", image: "barrafina"),
                        Restaurant(name: "Donostia", image: "donostia"),
                        Restaurant(name: "Royal Oak", image: "royaloak"),
                        Restaurant(name: "CASK Pub and Kitchen", image: "caskpubkitchen")
    ]

    var body: some View {
        NavigationView {
            List {
                ForEach(restaurants) { restaurant in
                    BasicImageRow(restaurant: restaurant)
                }
            }

            .navigationBarTitle("Restaurants")
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}

struct Restaurant: Identifiable {
    var id = UUID()
    var name: String
```

Figure 3. A basic navigation UI

Passing Data to a Detail View Using NavigationLink

So far, we have added a navigation bar to the list view. We usually use a navigation interface for the user to navigate to a detail view, showing the details of the selected item. For this demo, we will build a simple detail view showing a bigger image of the restaurant.

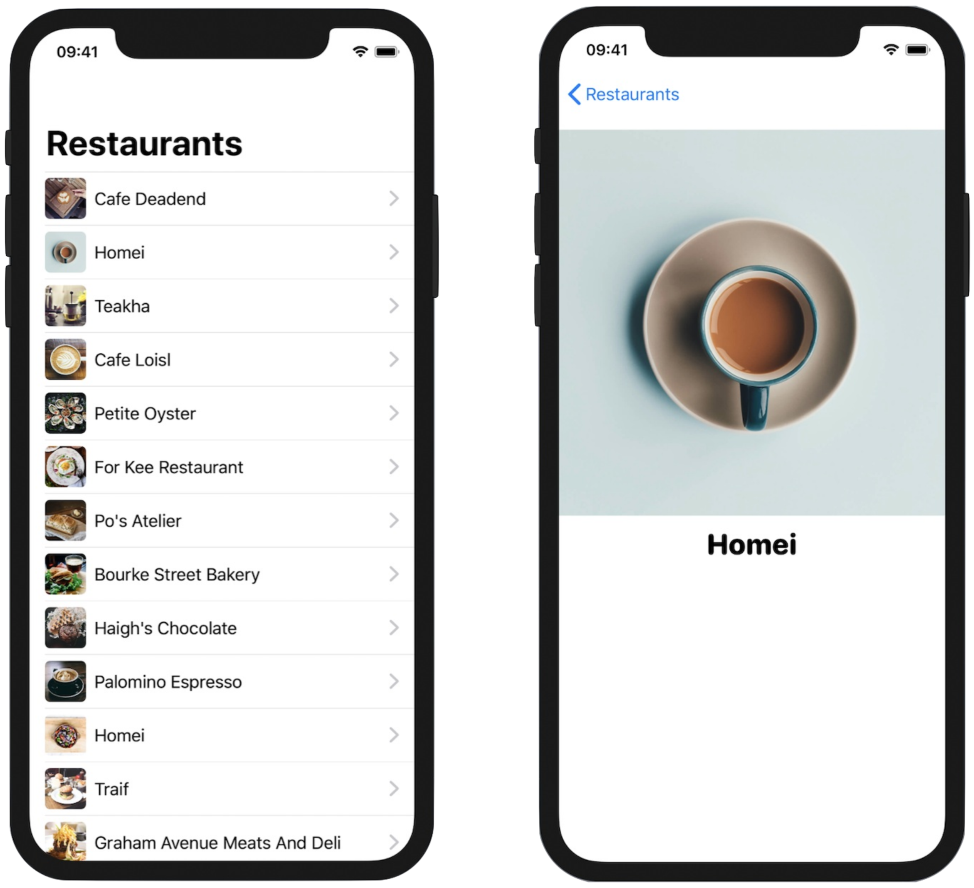


Figure 4. The content view and detail view

Let's start with the detail view. Insert the following code at the end of the `ContentView.swift` file to create the detail view:

```
struct RestaurantDetailView: View {
    var restaurant: Restaurant

    var body: some View {
        VStack {
            Image(restaurant.image)
                .resizable()
                .aspectRatio(contentMode: .fit)

            Text(restaurant.name)
                .font(.system(.title, design: .rounded))
                .fontWeight(.black)

            Spacer()
        }
    }
}
```

The detail view is just like other SwiftUI views of the type `View`. Its layout is very simple in that it only displays the restaurant image and name. The `RestaurantDetailView` struct also takes in a `Restaurant` object in order to retrieve the image and name of the restaurant.

With the detail view now ready, the question is how you can pass the selected restaurant in the content view to this detail view?

SwiftUI provides a special button called `NavigationLink`, which is able to detect users' touches and triggers the navigation presentation. The basic usage of `NavigationLink` is like this:

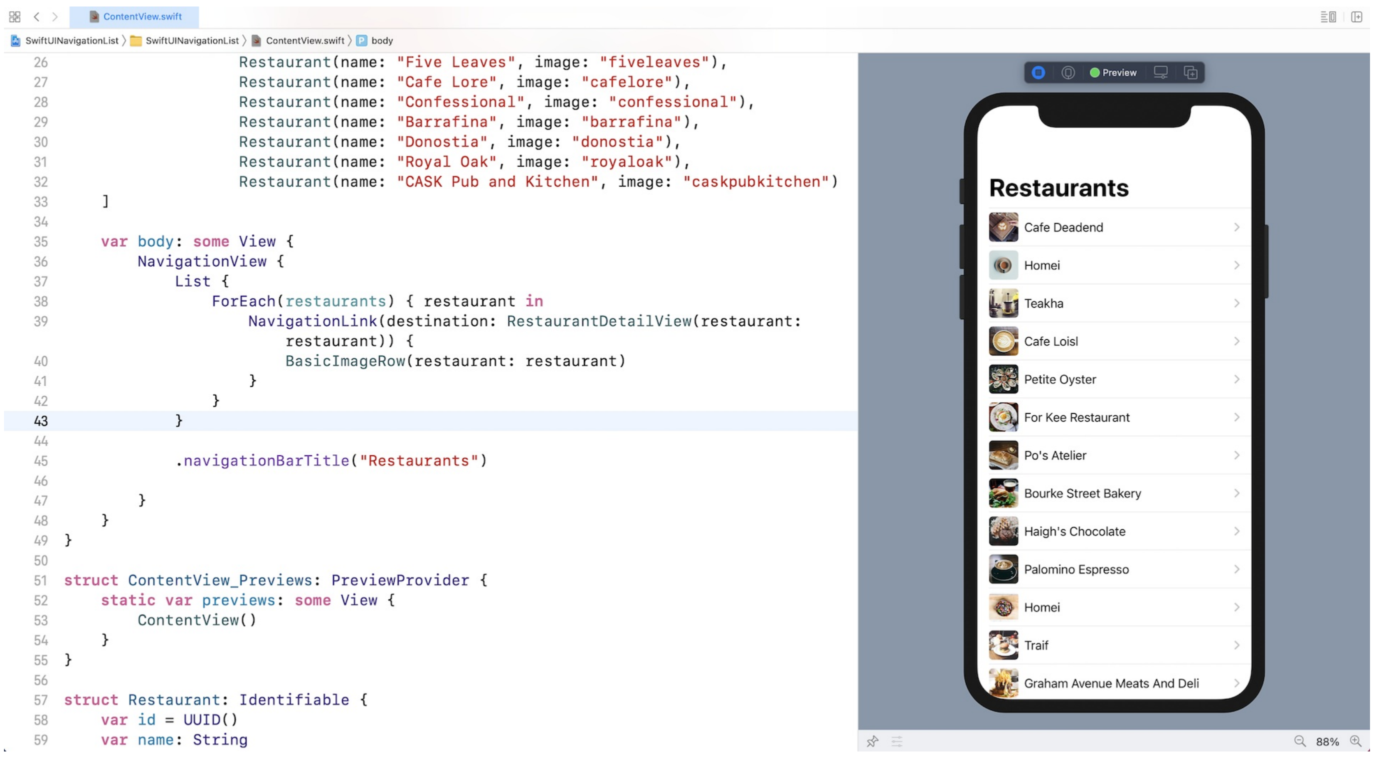
```
NavigationLink(destination: DetailView()) {
    Text("Press me for details")
}
```

You specify the destination view in the `destination` parameter and implement its look in the closure. For the demo app, it should navigate to the `RestaurantDetailView` when any of the restaurants is tapped. In this case, we can apply `NavigationLink` to each of the rows.

Update the `List` view like this:

```
List {
    ForEach(restaurants) { restaurant in
        NavigationLink(destination: RestaurantDetailView(restaurant: restaurant))
{
            BasicImageRow(restaurant: restaurant)
        }
    }
}
```

In the code above, we tell `NavigationLink` to navigate to the `RestaurantDetailView` when users select a restaurant. We also pass the selected restaurant to the detail view for display. That's all you need to build a navigation interface and perform data passing.



```
                    Restaurant(name: "Five Leaves", image: "fiveleaves"),
                    Restaurant(name: "Cafe Lore", image: "cafelore"),
                    Restaurant(name: "Confessional", image: "confessional"),
                    Restaurant(name: "Barrafina", image: "barrafina"),
                    Restaurant(name: "Donostia", image: "donostia"),
                    Restaurant(name: "Royal Oak", image: "royaloak"),
                    Restaurant(name: "CASK Pub and Kitchen", image: "caskpubkitchen")
    ]

    var body: some View {
        NavigationView {
            List {
                ForEach(restaurants) { restaurant in
                    NavigationLink(destination: RestaurantDetailView(restaurant:
                        restaurant)) {
                        BasicImageRow(restaurant: restaurant)
                    }
                }
            }

            .navigationBarTitle("Restaurants")

        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}

struct Restaurant: Identifiable {
    var id = UUID()
    var name: String
```

Figure 5. Run the app to test the navigation

In the canvas, you should notice that each row of data has been added with a disclosure icon. Click the Run button to execute the project. You should be able to navigate to the detail view after selecting one of the restaurants. Furthermore, you can navigate back to content view by clicking the back button. The whole navigation is automatically rendered by `NavigationView`.

Customizing the Navigation Bar

Starting from iOS 13, Apple added a new API called `UINavigationBarAppearance` for navigation bar customization. Its usage is very similar to the old API but offers you more granularity. You are allowed to configure the following for a navigation bar:

1. **Standard Appearance** (`.standardAppearance`) - the appearance of a standard-height navigation bar (e.g. the navigation bar appears in iPhone portrait mode)
2. **Compact Appearance** (`.compactAppearance`) - the appearance of a compact-height navigation bar (e.g. the navigation bar appears in iPhone landscape mode)
3. **Scroll Edge Appearance** (`.scrollEdgeAppearance`) - this is the appearance when the edge of the scrolled content reaches the navigation bar

In any given app, you are not required to implement all three of these navigation bar appearances. You can apply the same settings for all instances of a navigation bar (see section "Configuring Font and Color".

What customizations can you apply to the navigation bar? Actually, quite a lot. You can change the navigation bar's font, color, background, etc. I'll cover some of the attributes; However, for the full details, you may refer to Apple's official documentation (https://developer.apple.com/documentation/uikit/uinavigationbarappearance).

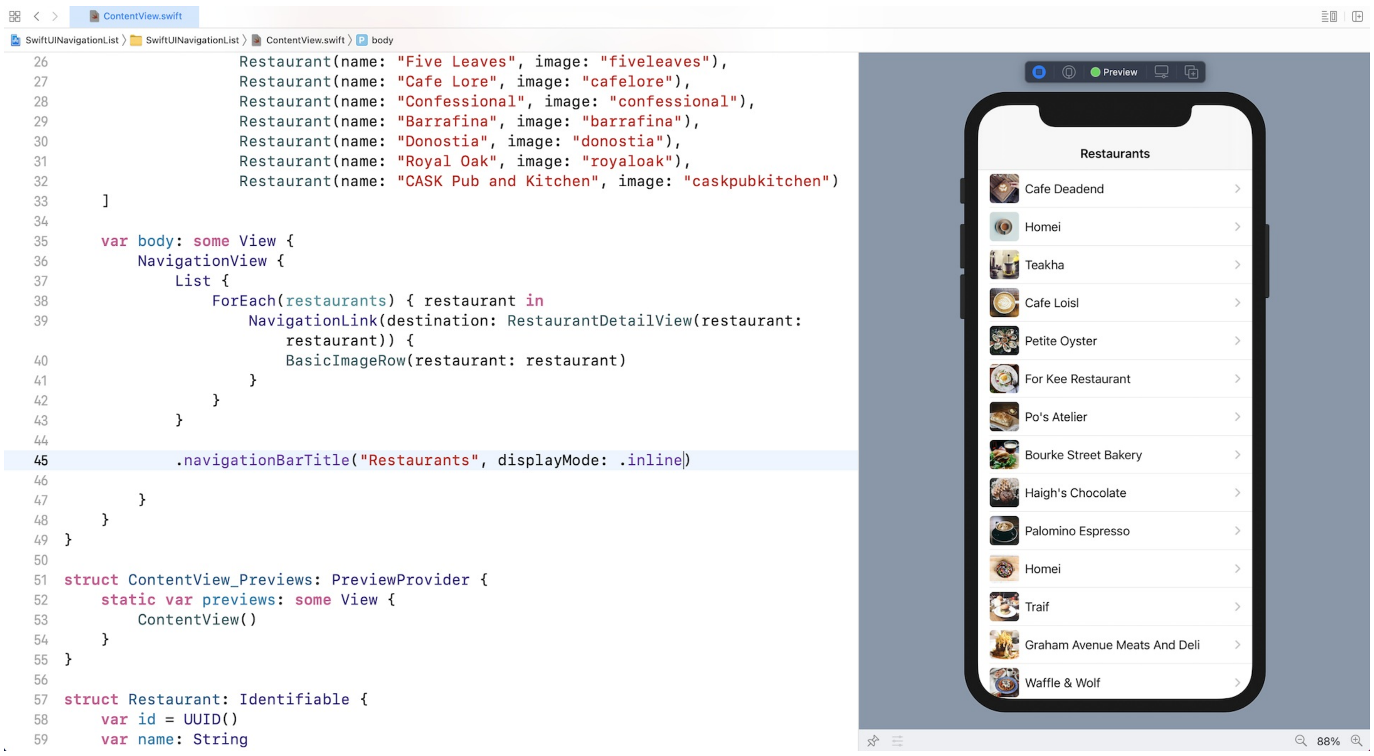
Display Mode

First, let's talk about the display mode of the navigation bar. By default, the navigation bar is set to appear as a large title. But when you scroll up the list, the navigation bar will become smaller. This became the default behaviour when Apple introduced the "Large Title" navigation bar.

If you want to keep the navigation bar compact and disable the use of the large title, you can change the `navigationBarTitle` modifier like this:

```
.navigationBarTitle("Restaurants", displayMode: .inline)
```

The `displayMode` parameter controls the appearance of the navigation bar, whether it should appear as a large title bar or compact title. By default, it's set to `.automatic`, which means large title is used. In the code above, we set it to `.inline`. This instructs iOS to use a compact bar.



```
                    Restaurant(name: "Five Leaves", image: "fiveleaves"),
                    Restaurant(name: "Cafe Lore", image: "cafelore"),
                    Restaurant(name: "Confessional", image: "confessional"),
                    Restaurant(name: "Barrafina", image: "barrafina"),
                    Restaurant(name: "Donostia", image: "donostia"),
                    Restaurant(name: "Royal Oak", image: "royaloak"),
                    Restaurant(name: "CASK Pub and Kitchen", image: "caskpubkitchen")
    ]

    var body: some View {
        NavigationView {
            List {
                ForEach(restaurants) { restaurant in
                    NavigationLink(destination: RestaurantDetailView(restaurant:
                        restaurant)) {
                        BasicImageRow(restaurant: restaurant)
                    }
                }
            }

            .navigationBarTitle("Restaurants", displayMode: .inline)

        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}

struct Restaurant: Identifiable {
    var id = UUID()
    var name: String
```

Figure 6. Setting the display mode to .inline to use the compact bar

Change the display mode to `.automatic` and the navigation bar will become a large title bar again.

```
.navigationBarTitle("Restaurants", displayMode: .automatic)
```

Configuring Font and Color

Next, let's change the title's font and color. At the time of this writing, there is no modifier in SwiftUI for developers to configure the navigation bar's font and color. Instead, we need to use the API named `UINavigationBarAppearance` provided by UIKit.

Say we want to change the title color to red and the font to *Arial Rounded MT Bold*. We create a `UINavigationBarAppearance` object in the `init()` function and configure the attributes accordingly. Insert the following function in `ContentView` :

```
init() {
    let navBarAppearance = UINavigationBarAppearance()
    navBarAppearance.largeTitleTextAttributes = [.foregroundColor: UIColor.systemR
ed, .font: UIFont(name: "ArialRoundedMTBold", size: 35)!]
    navBarAppearance.titleTextAttributes = [.foregroundColor: UIColor.systemRed, .
font: UIFont(name: "ArialRoundedMTBold", size: 20)!]

    UINavigationBar.appearance().standardAppearance = navBarAppearance
    UINavigationBar.appearance().scrollEdgeAppearance = navBarAppearance
    UINavigationBar.appearance().compactAppearance = navBarAppearance
}
```

The `largeTitleTextAttributes` property is used to configuring the text attributes of the large-size title, while the `titleTextAttributes` property is used for setting the text attributes of the standard-size title. Once we configure the `navBarAppearance` , we assign it to the three appearance properties including `standardAppearance` , `scrollEdgeAppearance` , and `compactAppearance` . If you want, you can create and assign a separate appearance object for `scrollEdgeAppearance` , and `compactAppearance` .

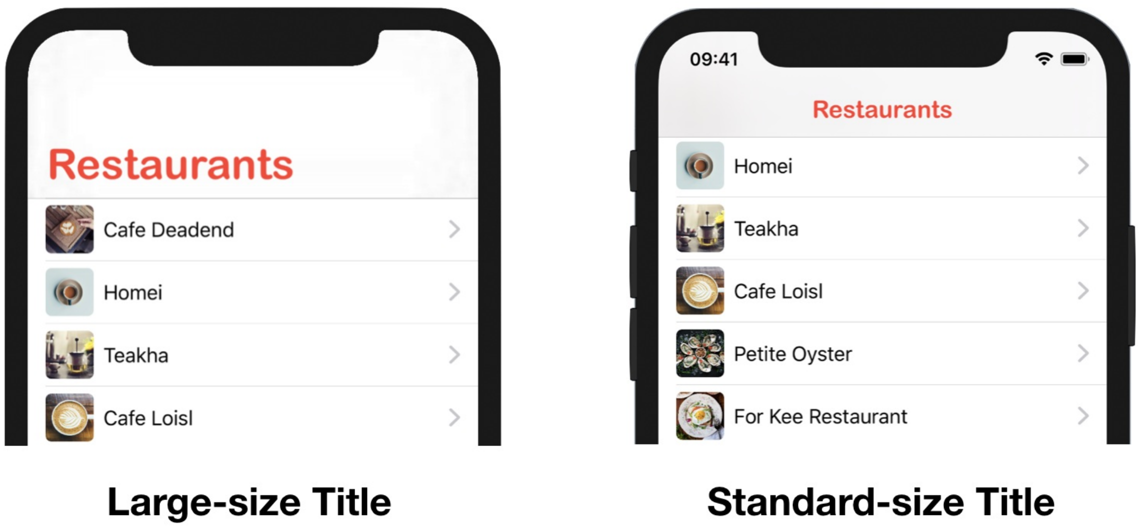


Figure 7. Changing the font type and color for both large-size and standard-size titles

Back Button Image and Color

The back button of the navigation view is set to blue by default and it uses a chevron icon to indicate "Go back." By using the `UINavigationBarAppearance` API, you can also customize the color and even the indicator image of the back button.

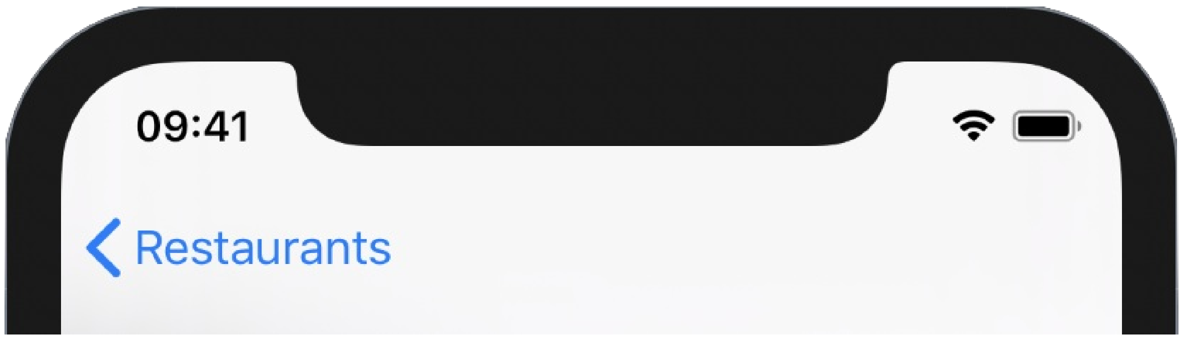


Figure 8. A standard back button

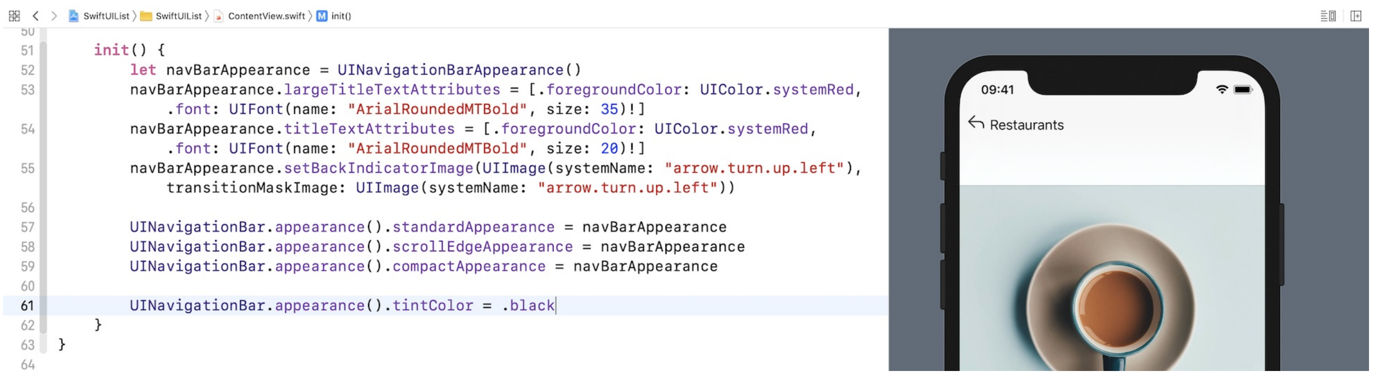
Let's see how this customization works. To change the indicator image, you can call the `setBackIndicatorImage` method and provide your own `UIImage`. Here I set it to the system image `arrow.turn.up.left`.

```
navBarAppearance.setBackIndicatorImage(UIImage(systemName: "arrow.turn.up.left"),
transitionMaskImage: UIImage(systemName: "arrow.turn.up.left"))
```

For the back button color, you can change it by setting the `tintColor` property:

```
UINavigationBar.appearance().tintColor = .black
```

Run the app. The back button should be like that shown in figure 9.



```
    init() {
        let navBarAppearance = UINavigationBarAppearance()
        navBarAppearance.largeTitleTextAttributes = [.foregroundColor: UIColor.systemRed,
            .font: UIFont(name: "ArialRoundedMTBold", size: 35)!]
        navBarAppearance.titleTextAttributes = [.foregroundColor: UIColor.systemRed,
            .font: UIFont(name: "ArialRoundedMTBold", size: 20)!]
        navBarAppearance.setBackIndicatorImage(UIImage(systemName: "arrow.turn.up.left"),
            transitionMaskImage: UIImage(systemName: "arrow.turn.up.left"))

        UINavigationBar.appearance().standardAppearance = navBarAppearance
        UINavigationBar.appearance().scrollEdgeAppearance = navBarAppearance
        UINavigationBar.appearance().compactAppearance = navBarAppearance

        UINavigationBar.appearance().tintColor = .black
    }
}
```

Figure 9. Customizing the appearance of the back button

Custom Back Button

Instead of using the APIs of UIKit to customize the back button, an alternative approach is to hide the default back button and create our own back button in SwiftUI. To hide the back button, you can use the modifier `.navigationBarBackButtonHidden` and set its value to `true` like this:

```
.navigationBarBackButtonHidden(true)
```

SwiftUI also provides a modifier called `navigationBarItems` for creating your own navigation bar items. For example, you can create a back button with the name of the selected restaurant like this:

```
.navigationBarItems(leading: Button(action : {
    // action
}){
    Text("\(Image(systemName: "chevron.left")) \(restaurant.name)")
        .foregroundColor(.red)
})
```

To put the following code into action and update `RestaurantDetailView` like below:

```
struct RestaurantDetailView: View {
    @Environment(\.presentationMode) var mode

    var restaurant: Restaurant

    var body: some View {
        VStack {
            Image(restaurant.image)
                .resizable()
                .aspectRatio(contentMode: .fit)

            Text(restaurant.name)
                .font(.system(.title, design: .rounded))
                .fontWeight(.black)

            Spacer()
        }

        .navigationBarBackButtonHidden(true)
        .navigationBarItems(leading: Button(action : {
            self.mode.wrappedValue.dismiss()
        }){
            Text("\(Image(systemName: "chevron.left")) \(restaurant.name)")
                .foregroundColor(.red)
        })
    }
}
```

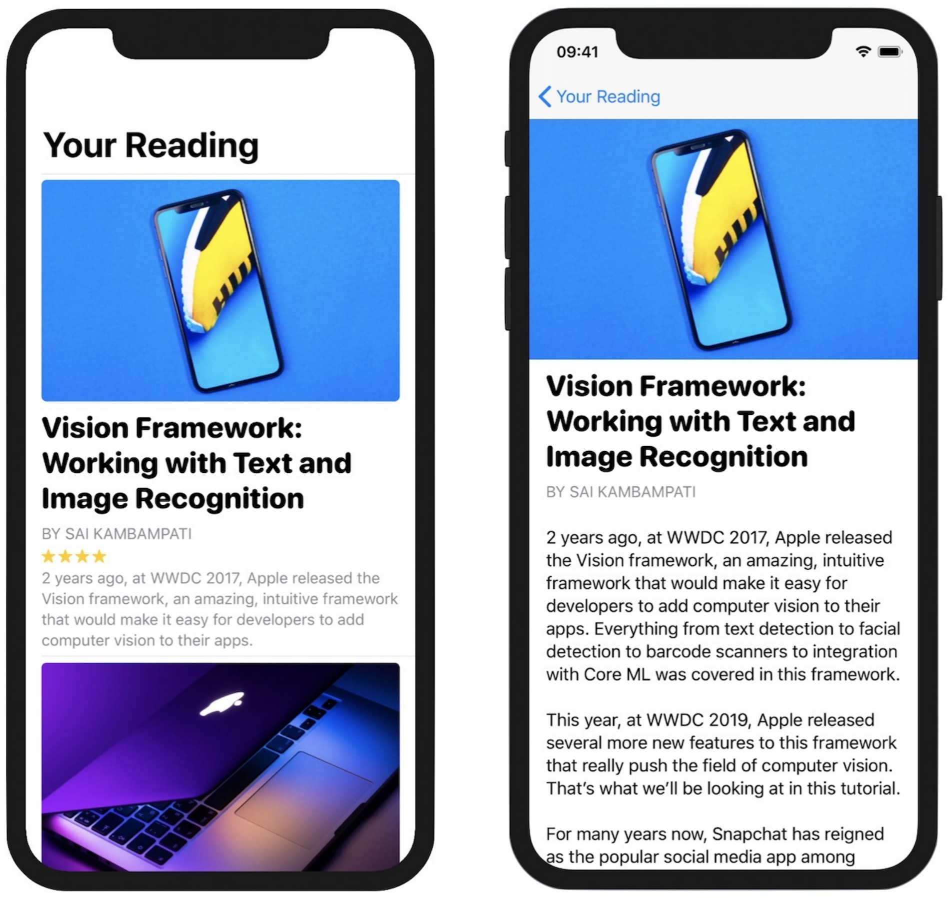
SwiftUI offers a wide range of built-in environment values. To dismiss the current view and go back to the previous view, we retrieve the environment value `.presentationMode` and then call its `dismiss()` function. If you run the app in the preview canvas and select any of the restaurants, you will see a back button with the restaurant name. Tapping the back button will navigate back to the main screen.

Exercise

To make sure understand how to build a navigation UI, here is an exercise for you. First, download this starter project from https://www.appcoda.com/resources/swiftui2/SwiftUINavigationStarter.zip. Open the project and you will see a demo app showing a list of articles.

This project is very similar to the one you've built before. The main difference is the introduction of `Article.swift`. This file stores the `articles` array, which contains sample data. If you look at the `Article` struct closely, it now has the `content` property for storing a full article.

Your task is to embed the list in a navigation view and create the detail view. When a user taps one of the articles in the content view, it'll navigate to the detail view showing the full article. I'll present the solution to you in the next section, but please try your best to figure out your own solution.



Vision Framework: Working with Text and Image Recognition

BY SAI KAMBAMPATI

★★★★

2 years ago, at WWDC 2017, Apple released the Vision framework, an amazing, intuitive framework that would make it easy for developers to add computer vision to their apps.

Vision Framework: Working with Text and Image Recognition

BY SAI KAMBAMPATI

2 years ago, at WWDC 2017, Apple released the Vision framework, an amazing, intuitive framework that would make it easy for developers to add computer vision to their apps. Everything from text detection to facial detection to barcode scanners to integration with Core ML was covered in this framework.

This year, at WWDC 2019, Apple released several more new features to this framework that really push the field of computer vision. That's what we'll be looking at in this tutorial.

For many years now, Snapchat has reigned as the popular social media app among

Figure 10. Building a navigation UI for a Reading app

Building the Detail View

Have you completed the exercise? The detail view is more complicated than the one we built earlier. Let's see how to create it.

To better organize the code, instead of creating the detail view in the `ContentView.swift` file, we will create a separate file for it. In the project navigator, right-click the `SwiftUINavigation` folder and select *New File...* Choose the *SwiftUI View* template and name the file *ArticleDetailView.swift*.

Since the detail view is going to display the full article , we need to have this property for the caller to pass the article. So, declare an `article` property in `ArticleDetailView` :

```
var article: Article
```

Next, update the `body` like this to lay out the detail view:

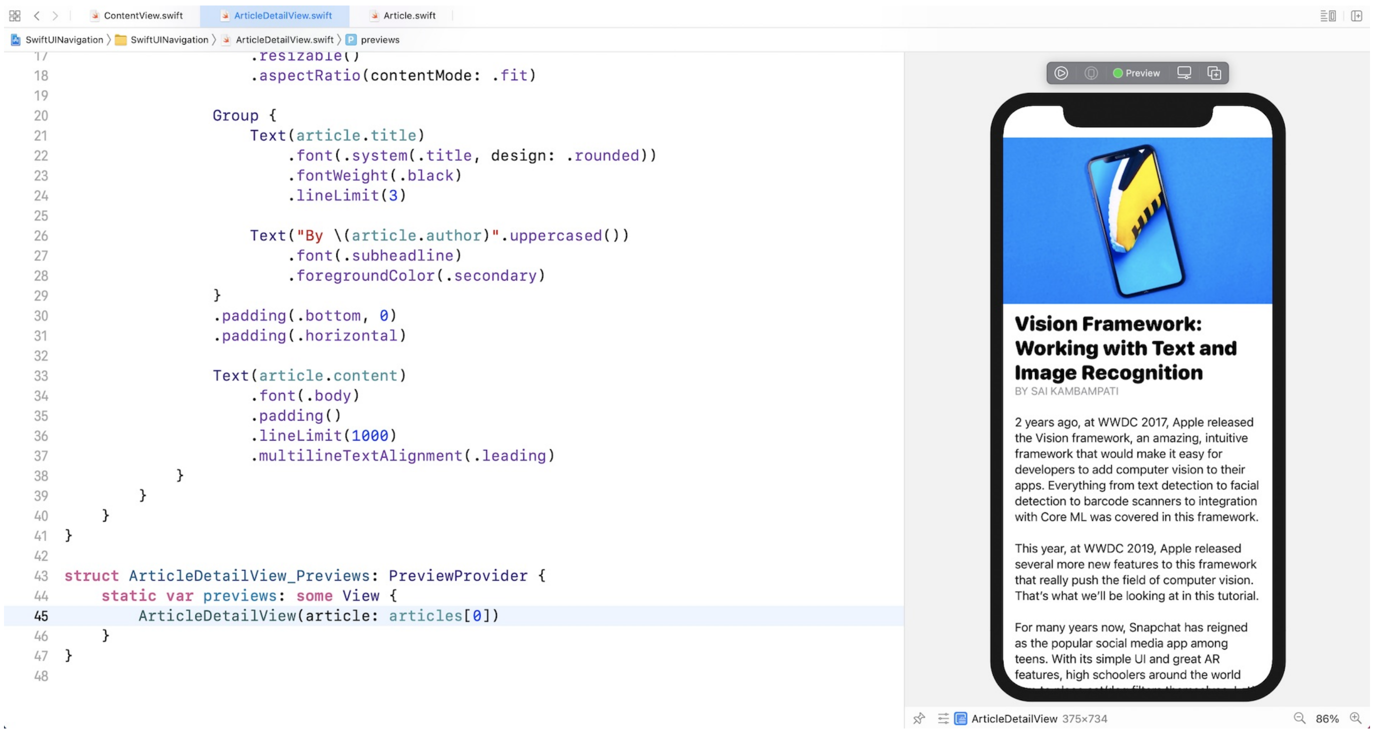
```
var body: some View {
    ScrollView {
        VStack(alignment: .leading) {
            Image(article.image)
                .resizable()
                .aspectRatio(contentMode: .fit)

            Group {
                Text(article.title)
                    .font(.system(.title, design: .rounded))
                    .fontWeight(.black)
                    .lineLimit(3)

                Text("By \(article.author)".uppercased())
                    .font(.subheadline)
                    .foregroundColor(.secondary)
            }
            .padding(.bottom, 0)
            .padding(.horizontal)

            Text(article.content)
                .font(.body)
                .padding()
                .lineLimit(1000)
                .multilineTextAlignment(.leading)
        }
    }
}
```

We use a `ScrollView` to wrap all the views to enable scrollable content. I'll not go over the code line by line as you understand how `Text` , `Image` , and `VStack` work. But one modifier that I want to highlight is `Group` . This modifier allows you to group multiple views together and apply a configuration to the group. In the code above, we need to apply padding to both `Text` views. To avoid code duplication, we group both views together and apply the padding.

Now that we have completed the layout of the detail view, you will see an error in Xcode complaining about the `ArticleDetailView_Previews`. The preview doesn't work because we've added the property `article` in `ArticleDetailView`. Therefore, you need to pass a sample article in the preview. Update `ArticleDetailView_Previews` like this to fix the error:

```
struct ArticleDetailView_Previews: PreviewProvider {
    static var previews: some View {
        ArticleDetailView(article: articles[0])
    }
}
```

Here we simply pick the first article of the `articles` array for preview. You can change it to a different value if you want to preview other articles. Once you have made this change, the preview canvas should render the detail view properly.


ContentView.swift
ArticleDetailView.swift
Article.swift
SwiftUINavigation 〉 SwiftUINavigation 〉 ArticleDetailView.swift 〉 previews
Preview

```
                    .resizable()
                    .aspectRatio(contentMode: .fit)

                Group {
                    Text(article.title)
                        .font(.system(.title, design: .rounded))
                        .fontWeight(.black)
                        .lineLimit(3)

                    Text("By \(article.author)".uppercased())
                        .font(.subheadline)
                        .foregroundColor(.secondary)
                }
                .padding(.bottom, 0)
                .padding(.horizontal)

                Text(article.content)
                    .font(.body)
                    .padding()
                    .lineLimit(1000)
                    .multilineTextAlignment(.leading)
            }
        }
    }
}

struct ArticleDetailView_Previews: PreviewProvider {
    static var previews: some View {
        ArticleDetailView(article: articles[0])
    }
}
```

Vision Framework: Working with Text and Image Recognition

BY SAI KAMBAMPATI

2 years ago, at WWDC 2017, Apple released the Vision framework, an amazing, intuitive framework that would make it easy for developers to add computer vision to their apps. Everything from text detection to facial detection to barcode scanners to integration with Core ML was covered in this framework.

This year, at WWDC 2019, Apple released several more new features to this framework that really push the field of computer vision. That's what we'll be looking at in this tutorial.

For many years now, Snapchat has reigned as the popular social media app among teens. With its simple UI and great AR features, high schoolers around the world

Figure 11. The detail view for showing the article

Let's try one more thing. Since this view is going to be embed in a `NavigationView`, you can modify the preview code to preview how it looks in a navigation view:

```
struct ArticleDetailView_Previews: PreviewProvider {
    static var previews: some View {
        NavigationView {
            ArticleDetailView(article: articles[0])
        }
    }
}
```

By updating the code, you will see a blank navigation bar in the preview canvas.

Now that we've completed the layout of the detail view, it's time to go back to `ContentView.swift` to implement the navigation. Update the `ContentView` struct like this:

```
struct ContentView: View {

    var body: some View {

        NavigationView {
            List(articles) { article in
                NavigationLink(destination: ArticleDetailView(article: article)) {
                    ArticleRow(article: article)
                }
            }

            .navigationBarTitle("Your Reading")
        }

    }
}
```

In the code above, we embed the `List` view in a `NavigationView` and apply a `NavigationLink` to each of the rows. The destination of the navigation link is set to the detail view we just created. In your preview, you should be able to test the app by clicking the *Play* button and navigate to the detail view when selecting an article.

Removing the Disclosure Indicator

The app works perfectly but there are two issues that you may want to fine tune. First, it's the disclosure indicator in the content view. It looks a bit weird to display the disclosure indicator. We will disable it. The second issue is the empty space appearing right above the featured image in the detail view. Let's discuss the issues one at a time.

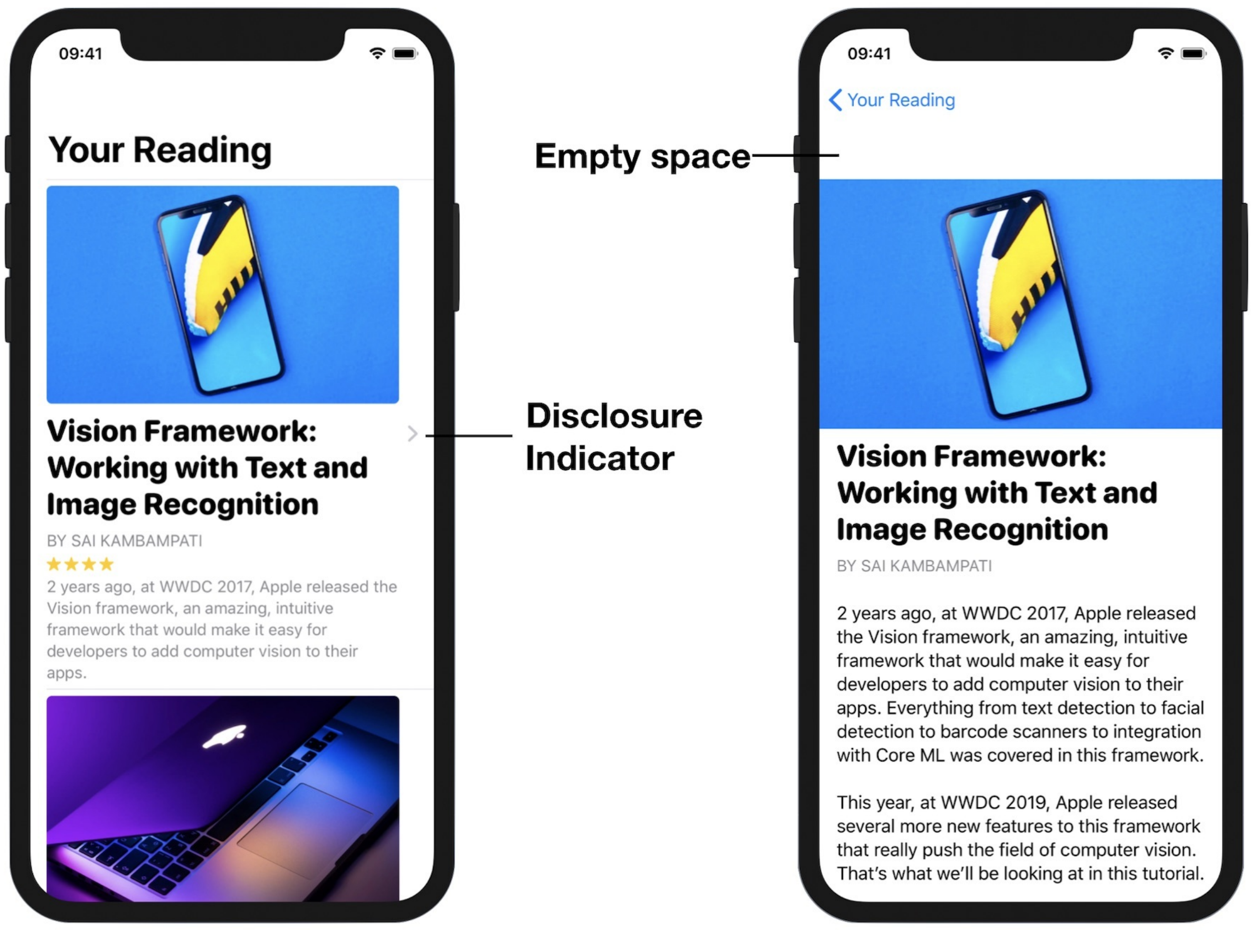

09:41
Your Reading
Vision Framework: Working with Text and Image Recognition
BY SAI KAMBAMPATI
★★★★
2 years ago, at WWDC 2017, Apple released the Vision framework, an amazing, intuitive framework that would make it easy for developers to add computer vision to their apps.
Empty space
Disclosure Indicator
09:41
Your Reading
Vision Framework: Working with Text and Image Recognition
BY SAI KAMBAMPATI
2 years ago, at WWDC 2017, Apple released the Vision framework, an amazing, intuitive framework that would make it easy for developers to add computer vision to their apps. Everything from text detection to facial detection to barcode scanners to integration with Core ML was covered in this framework.
This year, at WWDC 2019, Apple released several more new features to this framework that really push the field of computer vision. That's what we'll be looking at in this tutorial.

Figure 12. Two issues in the current design

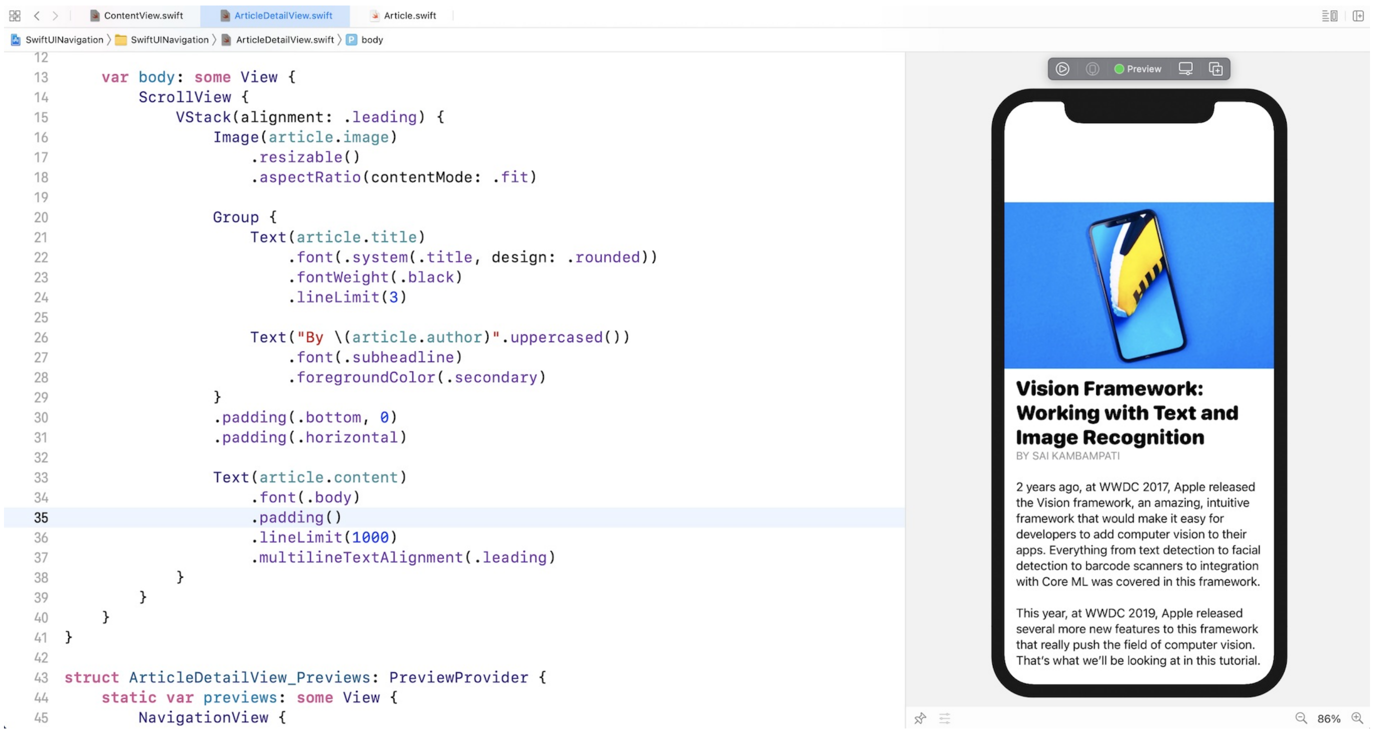
SwiftUI doesn't provide an option for developers to disable or hide the disclosure indicator. To work around the issue, we are not going to apply `NavigationLink` to the article row directly. Instead, we create a `ZStack` with two layers. Update the `NavigationView` of the `ContentView` like this:

```
NavigationView {
    List(articles) { article in
        ZStack {

            ArticleRow(article: article)

            NavigationLink(destination: ArticleDetailView(article: article)) {
                EmptyView()
            }
        }
    }

    .navigationBarTitle("Your Reading")
}
```

The lower layer is the article row, while the upper layer is an empty view. The `NavigationLink` now applies to the empty view, preventing iOS from rendering the disclosure button. Once you have made the change, the disclosure indicator vanishes but you can still navigate to the detail view.

Now let's see the root cause of the second issue.

Switch over to `ArticleDetailView.swift`. I didn't mention the issue when we were designing the detail view. But actually from the preview, you should spot the issue (see figure 13).


ContentView.swift
ArticleDetailView.swift
Article.swift
SwiftUINavigation 〉 SwiftUINavigation 〉 ArticleDetailView.swift 〉 body
Preview
86%

```
    var body: some View {
        ScrollView {
            VStack(alignment: .leading) {
                Image(article.image)
                    .resizable()
                    .aspectRatio(contentMode: .fit)

                Group {
                    Text(article.title)
                        .font(.system(.title, design: .rounded))
                        .fontWeight(.black)
                        .lineLimit(3)

                    Text("By \(article.author)".uppercased())
                        .font(.subheadline)
                        .foregroundColor(.secondary)
                }
                .padding(.bottom, 0)
                .padding(.horizontal)

                Text(article.content)
                    .font(.body)
                    .padding()
                    .lineLimit(1000)
                    .multilineTextAlignment(.leading)
            }
        }
    }
}

struct ArticleDetailView_Previews: PreviewProvider {
    static var previews: some View {
        NavigationView {
```

Vision Framework: Working with Text and Image Recognition

BY SAI KAMBAMPATI

2 years ago, at WWDC 2017, Apple released the Vision framework, an amazing, intuitive framework that would make it easy for developers to add computer vision to their apps. Everything from text detection to facial detection to barcode scanners to integration with Core ML was covered in this framework.

This year, at WWDC 2019, Apple released several more new features to this framework that really push the field of computer vision. That's what we'll be looking at in this tutorial.

Figure 13. Empty space in the header

The reason why we have that empty space right above the image is due to the navigation bar. This empty space is actually a large-size navigation bar with a blank title. When the app navigates from the content view to the detail view, the navigation bar becomes a standard-size bar. So, to fix the issue, all we need to do is explicitly specify to use the standard-size navigation bar.

Insert this line of code after the closing bracket of `ScrollView`:

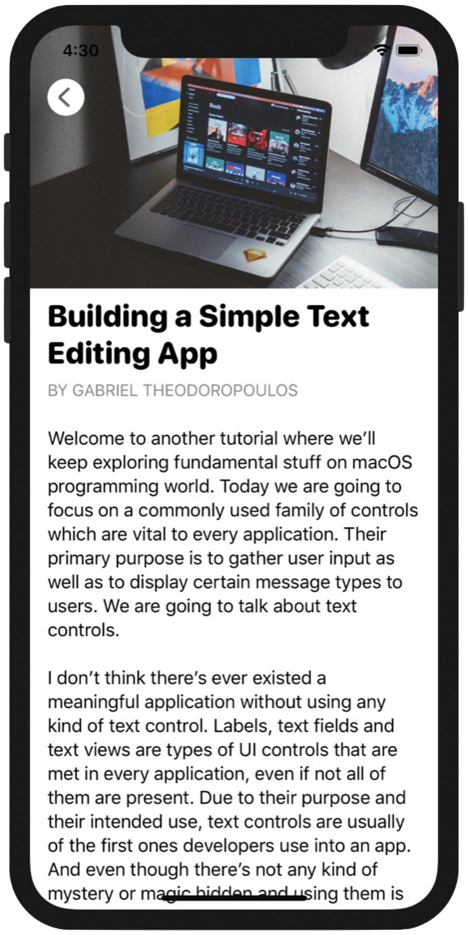
```
.navigationBarTitle("", displayMode: .inline)
```

By setting the navigation bar to the `inline` mode, the empty space will be minimized. You can now go back to `ContentView.swift` and test the app again. The detail view now looks much better.

An even more Elegant UI with a Custom Back Button

Though you can customize the back button indicator image using a built-in property, sometimes you may want to build a custom back button that navigates back to the content view. The question is how can it be done programmatically?

In this last section, I want to show you how to build an even more elegant detailed view by hiding the navigation bar and building your own back button. First, let's check out the final design displayed in figure 14. Doesn't it look great?



Building a Simple Text Editing App

BY GABRIEL THEODOROPOULOS

Welcome to another tutorial where we'll keep exploring fundamental stuff on macOS programming world. Today we are going to focus on a commonly used family of controls which are vital to every application. Their primary purpose is to gather user input as well as to display certain message types to users. We are going to talk about text controls.

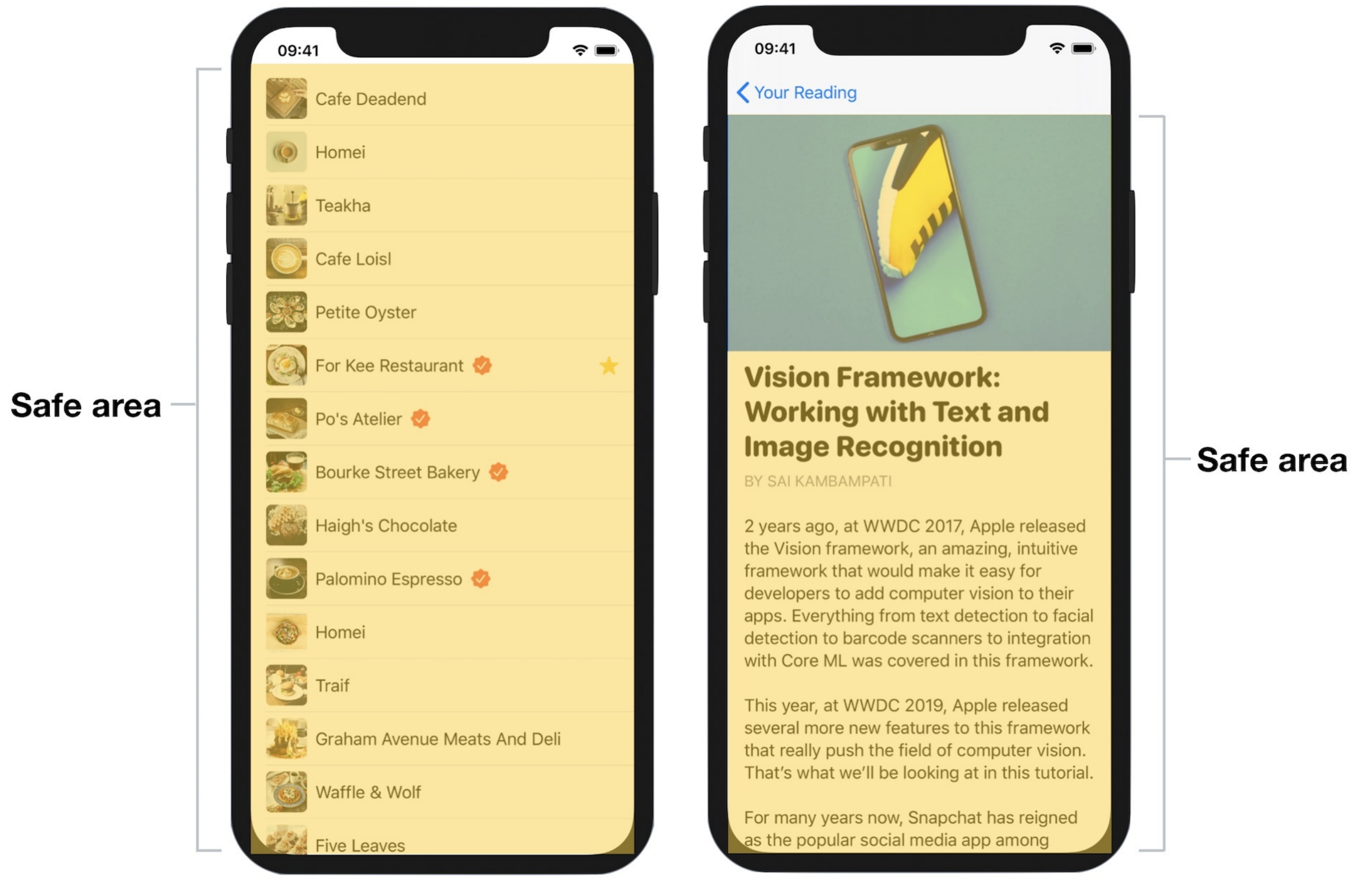
I don't think there's ever existed a meaningful application without using any kind of text control. Labels, text fields and text views are types of UI controls that are met in every application, even if not all of them are present. Due to their purpose and their intended use, text controls are usually of the first ones developers use into an app. And even though there's not any kind of mystery or magic hidden and using them is

Figure 14. The revised design of the detail view

To lay out this screen, we have to tackle two issues:

1. Extend the scroll view to the very top of the screen
2. Create a custom back button and trigger the navigation programmatically

iOS has a concept known as *safe areas* for aiding the layout of views. Safe areas help you place the views within the visible portion of the interface. For example, safe areas prevent the views from hiding the status bar. If your UI has a navigation bar, the safe area will automatically be adjusted to prevent you from positioning views that hide the navigation bar.



Vision Framework: Working with Text and Image Recognition

BY SAI KAMBAMPATI

2 years ago, at WWDC 2017, Apple released the Vision framework, an amazing, intuitive framework that would make it easy for developers to add computer vision to their apps. Everything from text detection to facial detection to barcode scanners to integration with Core ML was covered in this framework.

This year, at WWDC 2019, Apple released several more new features to this framework that really push the field of computer vision. That's what we'll be looking at in this tutorial.

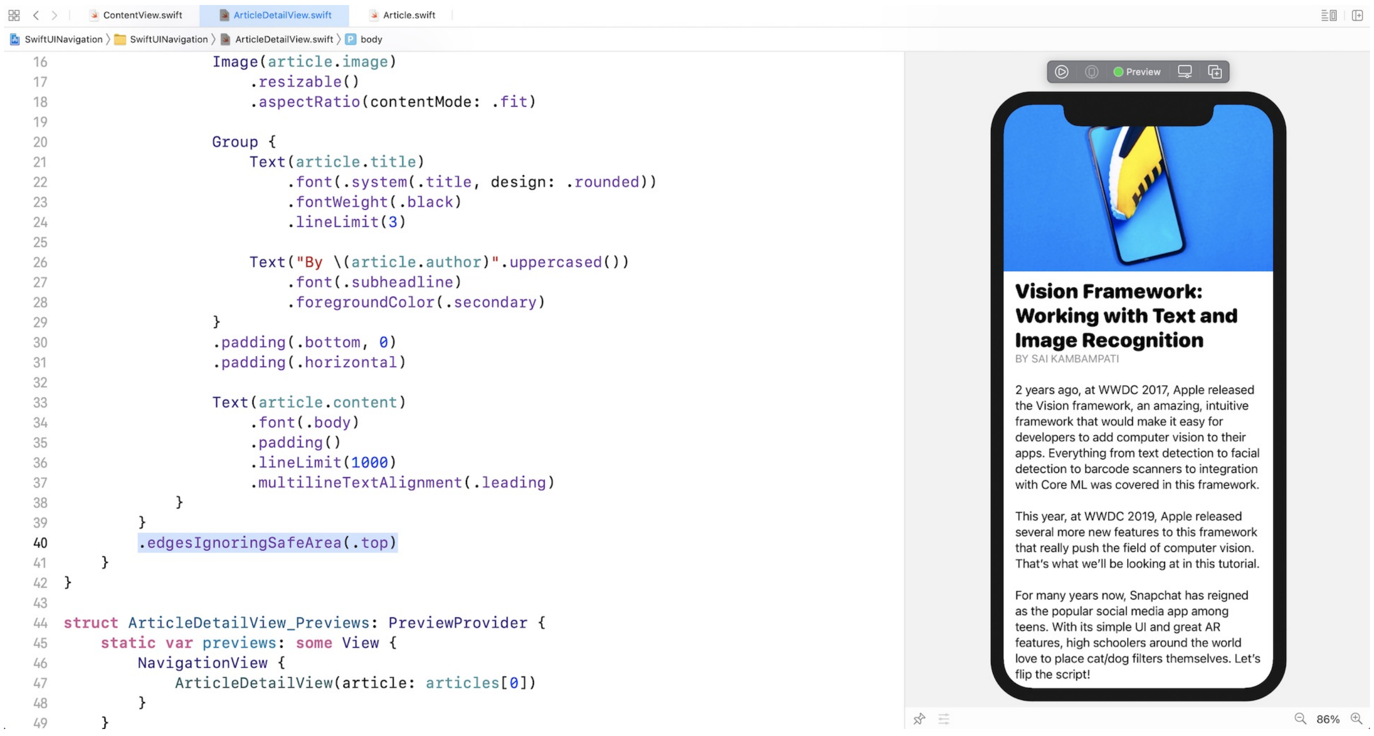
For many years now, Snapchat has reigned as the popular social media app among

![Figure 15. Safe areas](images/navigation/swiftui-navigation-15.jpg)

To place content that extends outside the safe areas, you use a modifier named `edgesIgnoringSafeArea`. For our project, we want the scroll view to go beyond the top edge of the safe area, To accomplish this, we write the modifier like this:

```
.edgesIgnoringSafeArea(.top)
```

This modifiers accepts other values like `.bottom` and `.leading`. If you want to ignore the whole safe area, you can pass it a `.all` value. By attaching this modifier to the `ScrollView`, we can hide the navigation bar and achieve a visually pleasing detail view.


ContentView.swift
ArticleDetailView.swift
Article.swift
SwiftUINavigation 〉 SwiftUINavigation 〉 ArticleDetailView.swift 〉 body
Preview
86%

```
                Image(article.image)
                    .resizable()
                    .aspectRatio(contentMode: .fit)

                Group {
                    Text(article.title)
                        .font(.system(.title, design: .rounded))
                        .fontWeight(.black)
                        .lineLimit(3)

                    Text("By \(article.author)".uppercased())
                        .font(.subheadline)
                        .foregroundColor(.secondary)
                }
                .padding(.bottom, 0)
                .padding(.horizontal)

                Text(article.content)
                    .font(.body)
                    .padding()
                    .lineLimit(1000)
                    .multilineTextAlignment(.leading)
            }
        }
        .edgesIgnoringSafeArea(.top)
    }
}

struct ArticleDetailView_Previews: PreviewProvider {
    static var previews: some View {
        NavigationView {
            ArticleDetailView(article: articles[0])
        }
    }
```

Vision Framework: Working with Text and Image Recognition

BY SAI KAMBAMPATI

2 years ago, at WWDC 2017, Apple released the Vision framework, an amazing, intuitive framework that would make it easy for developers to add computer vision to their apps. Everything from text detection to facial detection to barcode scanners to integration with Core ML was covered in this framework.

This year, at WWDC 2019, Apple released several more new features to this framework that really push the field of computer vision. That's what we'll be looking at in this tutorial.

For many years now, Snapchat has reigned as the popular social media app among teens. With its simple UI and great AR features, high schoolers around the world love to place cat/dog filters themselves. Let's flip the script!

Figure 16. Applying the modifiers to the scroll view

Now comes the second issue of creating our own back button. This issue is more tricky than the first one. Here is what we're going to implement:

1. Hide the original back button
2. Create a normal button and then assign it as the left button of the navigation bar

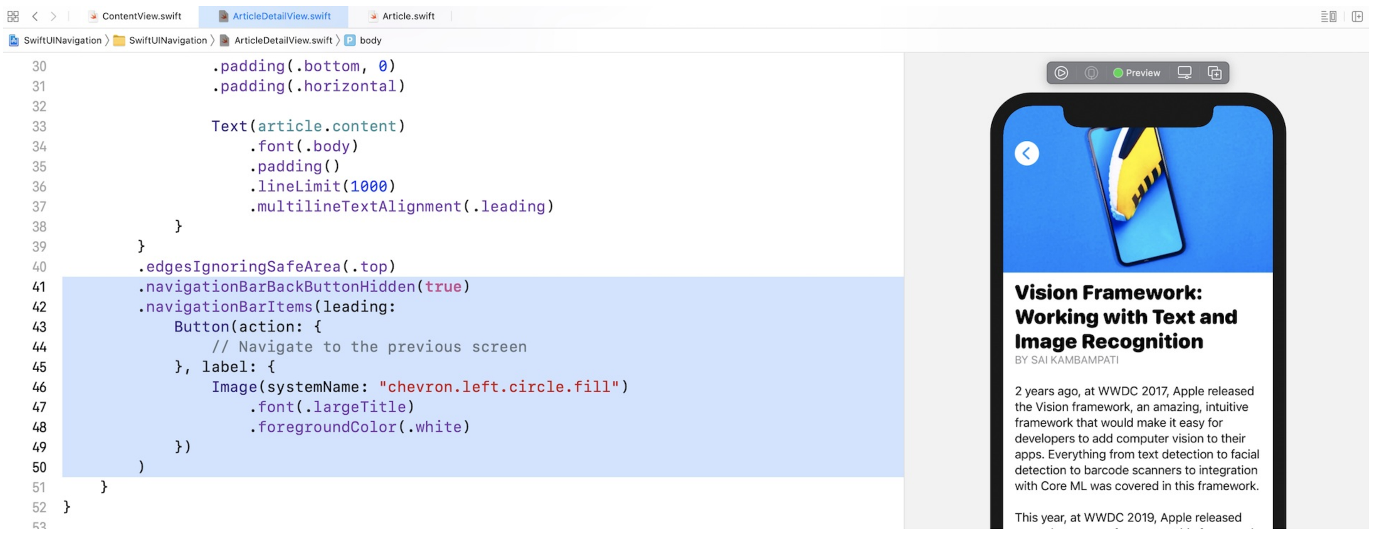
To hide the back button, SwiftUI provides a modifier called `navigationBarBackButtonHidden`. You just need to set its value to `true` to hide the back button:

```
.navigationBarBackButtonHidden(true)
```

Once the back button is hidden, you can replace it with your own button. The `navigationBarItems` modifier allows you to configure the navigation bar items. We can make use of it to assign the button as the left button of the navigation bar. Here is the code:

```
.navigationBarItems(leading:
    Button(action: {
        // Navigate to the previous screen
    }, label: {
        Image(systemName: "chevron.left.circle.fill")
            .font(.largeTitle)
            .foregroundColor(.white)
    })
)
```

You can attach the above modifiers to the `ScrollView`. Once the change is applied, you should see our custom back button in the preview canvas.



```
                .padding(.bottom, 0)
                .padding(.horizontal)

                Text(article.content)
                    .font(.body)
                    .padding()
                    .lineLimit(1000)
                    .multilineTextAlignment(.leading)
            }
        }
        .edgesIgnoringSafeArea(.top)
        .navigationBarBackButtonHidden(true)
        .navigationBarItems(leading:
            Button(action: {
                // Navigate to the previous screen
            }, label: {
                Image(systemName: "chevron.left.circle.fill")
                    .font(.largeTitle)
                    .foregroundColor(.white)
            })
        )
    }
}
```

Vision Framework: Working with Text and Image Recognition

BY SAI KAMBAMPATI

2 years ago, at WWDC 2017, Apple released the Vision framework, an amazing, intuitive framework that would make it easy for developers to add computer vision to their apps. Everything from text detection to facial detection to barcode scanners to integration with Core ML was covered in this framework.

This year, at WWDC 2019, Apple released

Figure 17. Creating our own back button

You may have noticed that the `action` closure of the button was left empty. The back button has been laid out nicely but the problem is that it doesn't function!

The original back button rendered by `NavigationView` can automatically navigate back to the previous screen. We need to programmatically navigate back. Thanks to the environment values built into the SwiftUI framework. You can refer to an environment binding named `presentationMode` to get the current presentation mode of the view. Most importantly, you can make use of it to dismiss a presented view (in this case, the detail view) to go back to the previous view.

Now declare a `presentationMode` variable in `ArticleDetailView` to capture the environment value:

```
@Environment(\.presentationMode) var presentationMode
```

Next, in the `action` of our custom back button, insert this line of code:

```
self.presentationMode.wrappedValue.dismiss()
```

Here we call the `dismiss` method to dismiss the detail view when the back button is tapped. Run the app and test it again. You should be able to navigate between the content view and the detail view.

Summary

Navigation UI is very common in mobile apps. It's crucial you understand this key concept. With this understanding, you are capable of building a simple content-based app, although the data is static.

For reference, you can download the complete project here:

- Demo project for the first project (https://www.appcoda.com/resources/swiftui2/SwiftUINavigationList.zip)
- Demo project for the second project (https://www.appcoda.com/resources/swiftui2/SwiftUINavigation.zip)

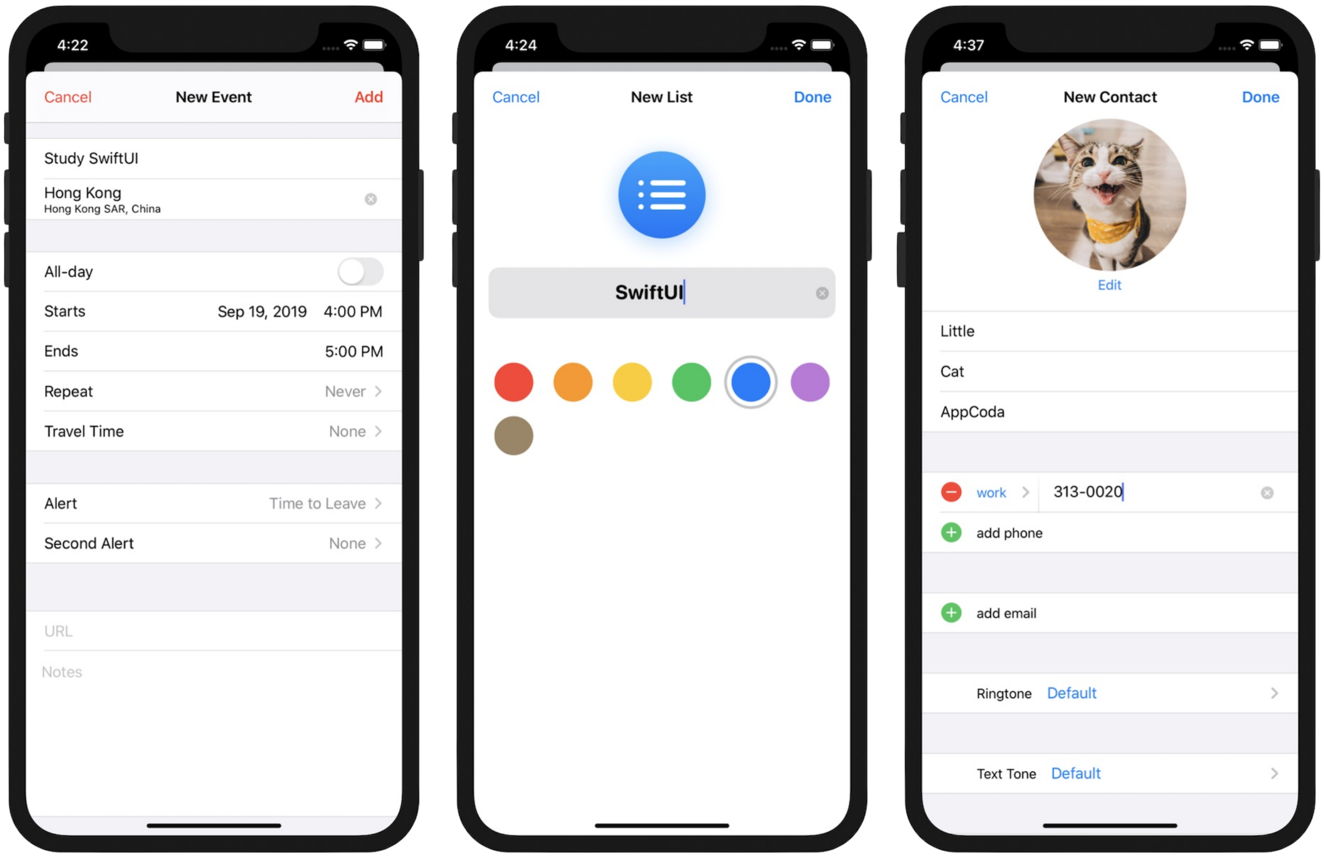
To further study navigation view, you can also refer to the documentation provided by Apple:

- https://developer.apple.com/tutorials/swiftui/building-lists-and-navigation
- https://developer.apple.com/documentation/swiftui/navigationview

Chapter 12
Playing with Modal Views, Floating Buttons and Alerts

Earlier, we built a navigation interface that lets users navigate from the content view to the detail view. The view transition is nicely animated and completely taken care by iOS. When a user triggers the transition, the detail view slides from right to left fluidly. Navigation UI is just one of the commonly-used UI patterns. In this chapter, I'll introduce to you another design technique to present content modally.

For iPhone users, you should be very familiar with modal views. One common use of modal views is for presenting a form for input. For example, the Calendar app presents a modal view for users to create a new event. The built-in Reminders and Contact apps also use modal views to ask for user input.



Study SwiftUI
Hong Kong
Hong Kong SAR, China
All-day
Starts Sep 19, 2019 4:00 PM
Ends 5:00 PM
Repeat Never
Travel Time None
Alert Time to Leave
Second Alert None
URL
Notes

SwiftUI

Little
Cat
AppCoda
work 313-0020
add phone
add email
Ringtone Default
Text Tone Default

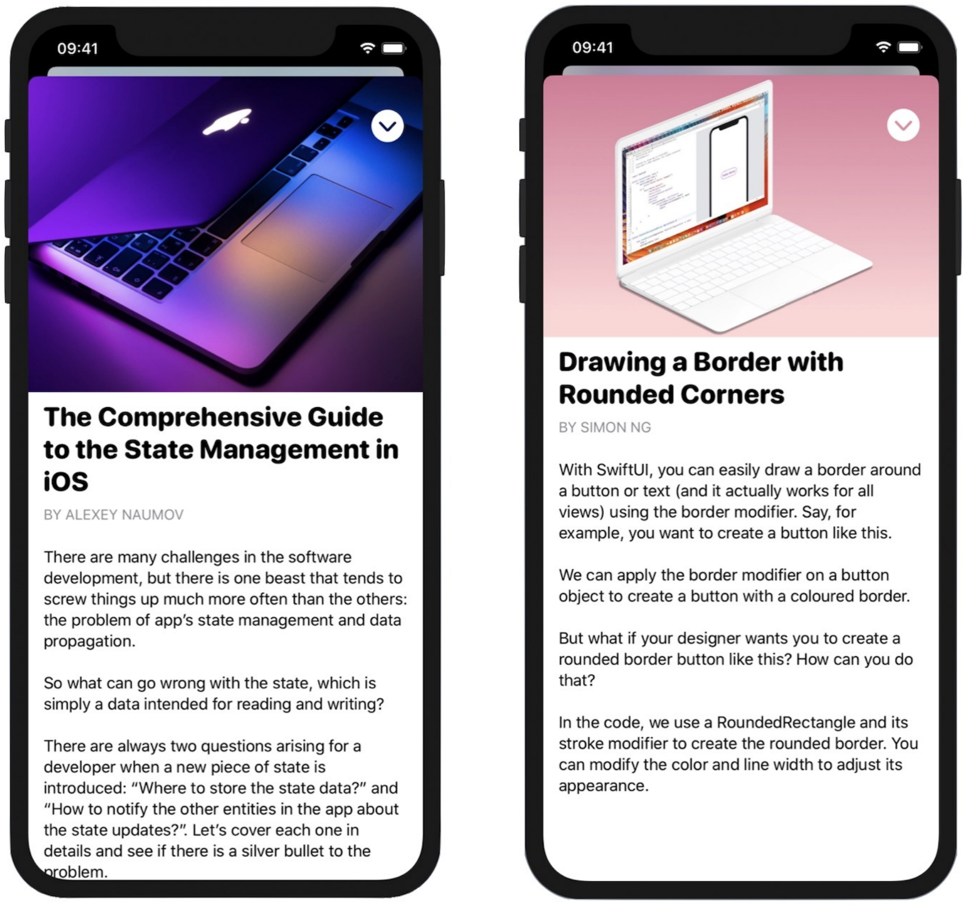
Figure 1. Sample modal views in Calendar, Reminders, and Contact apps

From the user experience point of view, a modal view is usually triggered by tapping a button. Again, the transition animation of the modal view is handled by iOS. When presenting a full-screen modal view, it slides up fluidly from the bottom of the screen.

If you're a long-time iOS user, you may find the look & feel of the modal views displayed in figure 1 are not the same as the traditional ones. Prior to iOS 13, the presentation of modal views covered the entire screen. Starting with iOS 13, modal views are displayed in card-like format by default. The modal view doesn't cover the whole screen but partially covers the underlying content view. You can still see the top edge of the content/parent view. On top of the visual change, the modal view can now be dismissed by swiping down from anywhere on the screen. You do not need to write a line of code to enable this gesture. It's completely built-in and generated by iOS. Of course, if you want to dismiss a modal view via a button, you can still do that.

Okay, so what are we going to work on in this chapter?

I will show you how to present the same detail view that we implemented in the previous chapter using a modal view. While modal views are commonly used for presenting a form, it doesn't mean you can't use them for presenting other information. In addition to modal views, you will also learn how to create a floating button in the detail view. While the modal views can be dismissed through the swipe gesture, I want to provide a *Close* button for users to dismiss the detail view. Furthermore, we will also look into Alerts, which is another kind of modal view.



The Comprehensive Guide to the State Management in iOS

BY ALEXEY NAUMOV

There are many challenges in the software development, but there is one beast that tends to screw things up much more often than the others: the problem of app's state management and data propagation.

So what can go wrong with the state, which is simply a data intended for reading and writing?

There are always two questions arising for a developer when a new piece of state is introduced: "Where to store the state data?" and "How to notify the other entities in the app about the state updates?". Let's cover each one in details and see if there is a silver bullet to the problem.

Drawing a Border with Rounded Corners

BY SIMON NG

With SwiftUI, you can easily draw a border around a button or text (and it actually works for all views) using the border modifier. Say, for example, you want to create a button like this.

We can apply the border modifier on a button object to create a button with a coloured border.

But what if your designer wants you to create a rounded border button like this? How can you do that?

In the code, we use a RoundedRectangle and its stroke modifier to create the rounded border. You can modify the color and line width to adjust its appearance.

Figure 2. Presenting the detail screen using modal views

We got a lot to discuss in this chapter. Let's get started.

Understanding Sheet in SwiftUI

> The sheet presentation style appears as a *card* that partially covers the underlying content and dims all uncovered areas to prevent interaction with them. The top edge of the parent view or a previous card is visible behind the current card to help people remember the task they suspended when they opened the card.
>
> - Apple's official documentation (https://developer.apple.com/design/human-interface-guidelines/ios/app-architecture/modality/)

Before we dive into the implementation, let me give you a quick introduction to the card-like presentation of modal views. The card presentation is achieved in SwiftUI using the sheet presentation style. It's the default presentation style for modal views.

Basically, to present a modal view, you apply the `sheet` modifier like this:

```
.sheet(isPresented: $showModal) {
    DetailView()
}
```

It takes in a boolean value to indicate whether the modal view is presented. If `isPresented` is set to `true`, the modal view will be automatically presented in the form of card.

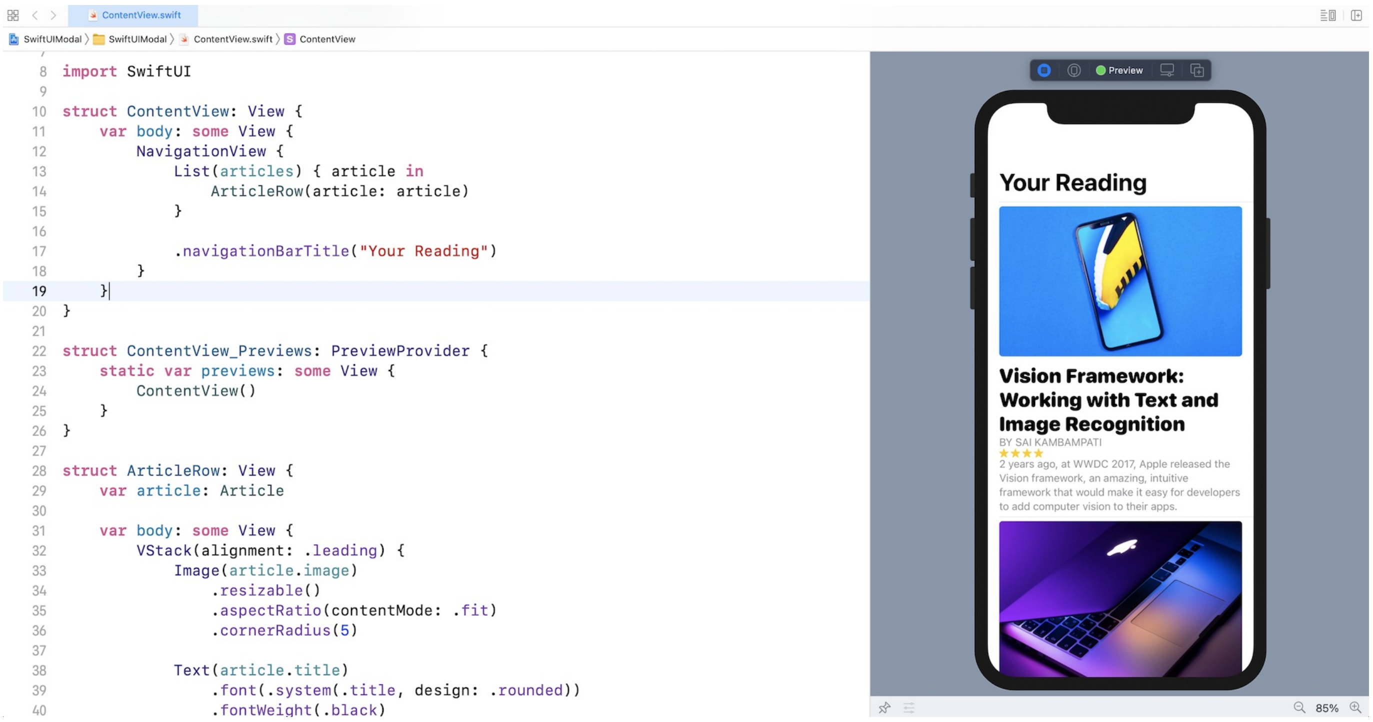
Another way to present the modal view is like this:

```
.sheet(item: $itemToDisplay) {
    DetailView()
}
```

The `sheet` modifier also allows you to trigger the display of modal views by passing an optional binding. If the optional has a value, iOS will bring up the modal view. If you remember our discussion on `actionSheet` in an earlier chapter, you will find that the usage of `sheet` is very similar to `actionSheet`.

Preparing the Starter Project

That's enough background information. Let's move onto the actual implementation of our demo project. To begin, please download the starter project from https://www.appcoda.com/resources/swiftui2/SwiftUIModalStarter.zip. Once downloaded, open the project and check out the preview. You should be very familiar with this demo app. The app still has a navigation bar but the navigation link has been removed.


ContentView.swift
SwiftUIModal 〉 SwiftUIModal 〉 ContentView.swift 〉 ContentView
Preview
85%

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        NavigationView {
            List(articles) { article in
                ArticleRow(article: article)
            }

            .navigationBarTitle("Your Reading")
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}

struct ArticleRow: View {
    var article: Article

    var body: some View {
        VStack(alignment: .leading) {
            Image(article.image)
                .resizable()
                .aspectRatio(contentMode: .fit)
                .cornerRadius(5)

            Text(article.title)
                .font(.system(.title, design: .rounded))
                .fontWeight(.black)
```

Your Reading

Vision Framework: Working with Text and Image Recognition

BY SAI KAMBAMPATI

★★★★

2 years ago, at WWDC 2017, Apple released the Vision framework, an amazing, intuitive framework that would make it easy for developers to add computer vision to their apps.

Figure 3. Starter project

Implementing the Modal View Using isPresented

As discussed earlier, the `sheet` modifier provides us two ways to present a modal. I'll show you how both approaches work. Let's start with the `isPresented` approach. For this approach, we need a state variable of the type `Bool` to keep track of the status of the modal view. Declare this variable in `ContentView` :

```
@State var showDetailView = false
```

By default, it's set to `false`. The value of this variable will be set to `true` when one of the rows is clicked. Later, we will make this change in the code.

When presenting the detail view, the view requires us to pass the selected article. So, we also need to declare a state variable to store the user's selection. In `ContentView`, declare another state variable for this purpose:

```
@State var selectedArticle: Article?
```

To implement the modal view, we attach the `sheet` modifier to the `List` like this:

```
NavigationView {
    List(articles) { article in
        ArticleRow(article: article)
    }
    .sheet(isPresented: self.$showDetailView) {

        if let selectedArticle = self.selectedArticle {
            ArticleDetailView(article: selectedArticle)
        }
    }

    .navigationBarTitle("Your Reading")
}
```

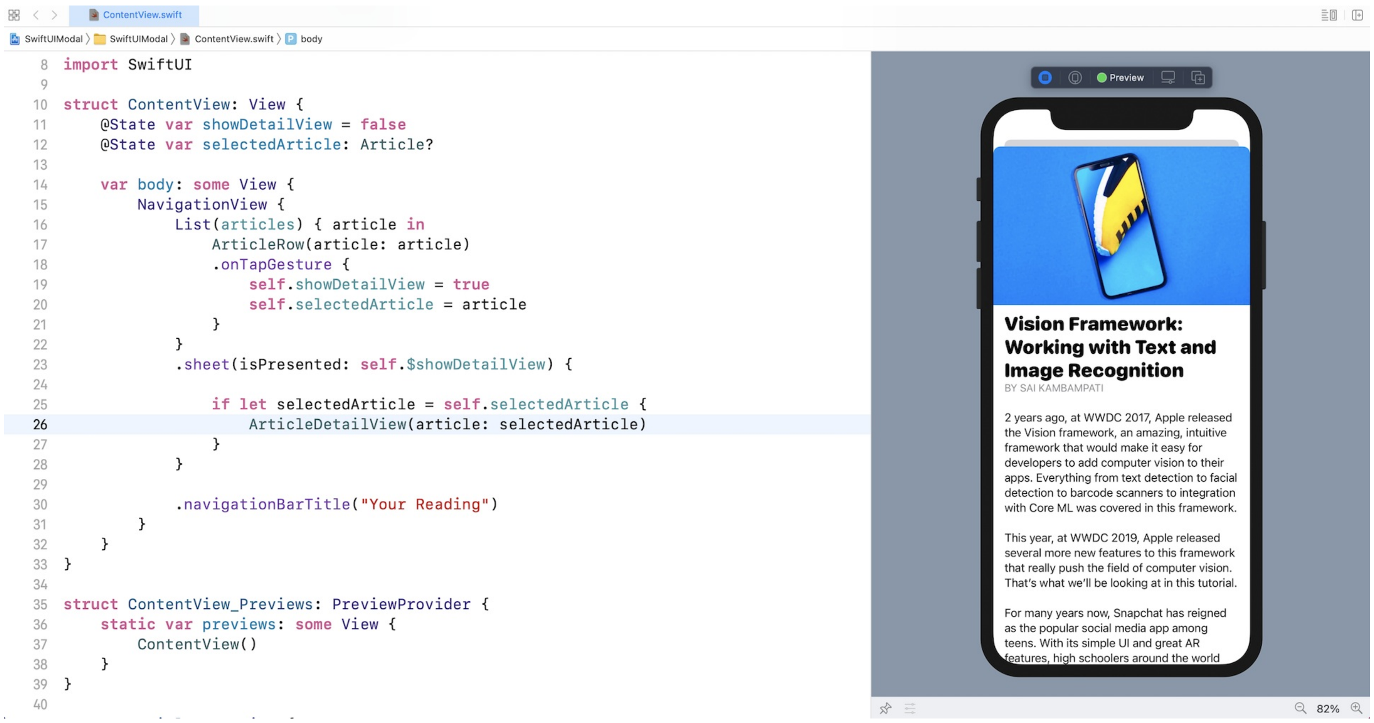
The presentation of the modal view depends on the value of the `showDetailView` property. This is why we specify it in the `isPresented` parameter. The closure of the `sheet` modifier describes the layout of the view to be presented. Here we will present the `ArticleDetailView`.

The remaining item is to detect touch. When building the navigation UI, we utilize `NavigationLink` to handle touch. However, this special button is designed for the navigation interface. In SwiftUI, there is a handler called `onTapGesture` which can be used to recognize a tap gesture. You can attach this handler to each of the `ArticleRow` to detect the users' touch. Modify the `NavigationView` in the `body` variable like this:

```
NavigationView {
    List(articles) { article in
        ArticleRow(article: article)
        .onTapGesture {
            self.showDetailView = true
            self.selectedArticle = article
        }
    }
    .sheet(isPresented: self.$showDetailView) {

        if let selectedArticle = self.selectedArticle {
            ArticleDetailView(article: selectedArticle)
        }
    }

    .navigationBarTitle("Your Reading")
}
```

In the closure of `onTapGesture`, we set the `showDetailView` to `true`. This is used to trigger the presentation of the modal view. We also store the selected article in the `selectedArticle` variable.

Run the app in the preview canvas, by clicking the play button. You should be able to bring up the detail view modally. **Note: It is better to run this demo in the simulator. If you run this in the preview, you may get a blank dialog. Wipe down the dialog to dismiss it, select another article (not the same article) and you should get the correct rendering.**



```swift
import SwiftUI

struct ContentView: View {
    @State var showDetailView = false
    @State var selectedArticle: Article?

    var body: some View {
        NavigationView {
            List(articles) { article in
                ArticleRow(article: article)
                .onTapGesture {
                    self.showDetailView = true
                    self.selectedArticle = article
                }
            }
            .sheet(isPresented: self.$showDetailView) {

                if let selectedArticle = self.selectedArticle {
                    ArticleDetailView(article: selectedArticle)
                }
            }

            .navigationBarTitle("Your Reading")
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Vision Framework: Working with Text and Image Recognition

BY SAI KAMBAMPATI

2 years ago, at WWDC 2017, Apple released the Vision framework, an amazing, intuitive framework that would make it easy for developers to add computer vision to their apps. Everything from text detection to facial detection to barcode scanners to integration with Core ML was covered in this framework.

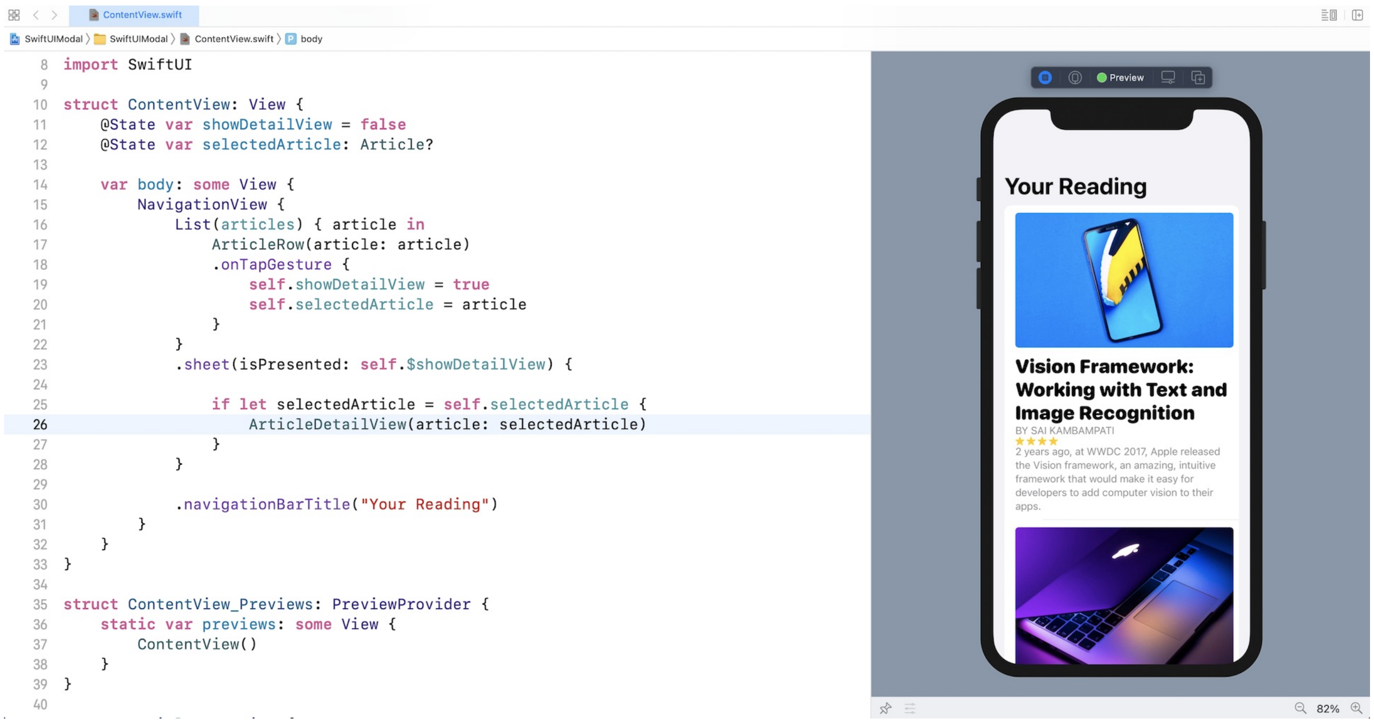
This year, at WWDC 2019, Apple released several more new features to this framework that really push the field of computer vision. That's what we'll be looking at in this tutorial.

For many years now, Snapchat has reigned as the popular social media app among teens. With its simple UI and great AR features, high schoolers around the world

Figure 4. Presenting the detail view modally

Changing the Navigation View Style

There was a change after we applied the `.sheet` modifier to the `List` view. Did you notice the change? Look at the following figure and compare it with the original list view shown in figure 3. Do you see the difference?

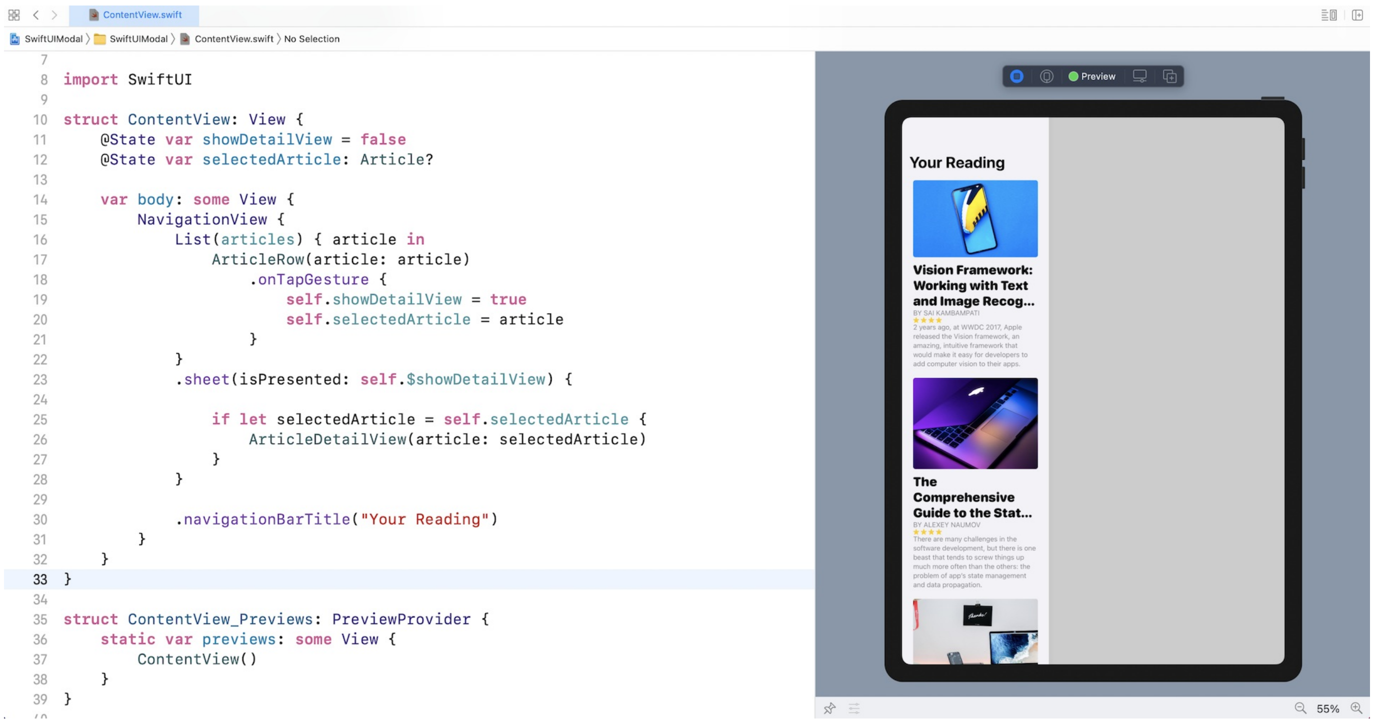


```swift
import SwiftUI

struct ContentView: View {
    @State var showDetailView = false
    @State var selectedArticle: Article?

    var body: some View {
        NavigationView {
            List(articles) { article in
                ArticleRow(article: article)
                .onTapGesture {
                    self.showDetailView = true
                    self.selectedArticle = article
                }
            }
            .sheet(isPresented: self.$showDetailView) {

                if let selectedArticle = self.selectedArticle {
                    ArticleDetailView(article: selectedArticle)
                }
            }

            .navigationBarTitle("Your Reading")
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Your Reading
Vision Framework: Working with Text and Image Recognition
BY SAI KAMBAMPATI
★★★★
2 years ago, at WWDC 2017, Apple released the Vision framework, an amazing, intuitive framework that would make it easy for developers to add computer vision to their apps.

Figure 5. The list view appears like an inset grouped list

Once we attach the `.sheet` modifer to the `List` view, SwiftUI automatically changes the list style such that the list view appears like an inset grouped list. The root cause for this is due to a change in the `NavigationView` style. In Xcode 12, SwiftUI changes the navigation view's style to `DoubleColumnNavigationViewStyle` when the `.sheet` modifier is used. Run the app on iPadOS, you will get a better idea how this style affects the layout.



```
import SwiftUI

struct ContentView: View {
    @State var showDetailView = false
    @State var selectedArticle: Article?

    var body: some View {
        NavigationView {
            List(articles) { article in
                ArticleRow(article: article)
                    .onTapGesture {
                        self.showDetailView = true
                        self.selectedArticle = article
                    }
            }
            .sheet(isPresented: self.$showDetailView) {

                if let selectedArticle = self.selectedArticle {
                    ArticleDetailView(article: selectedArticle)
                }
            }

            .navigationBarTitle("Your Reading")
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

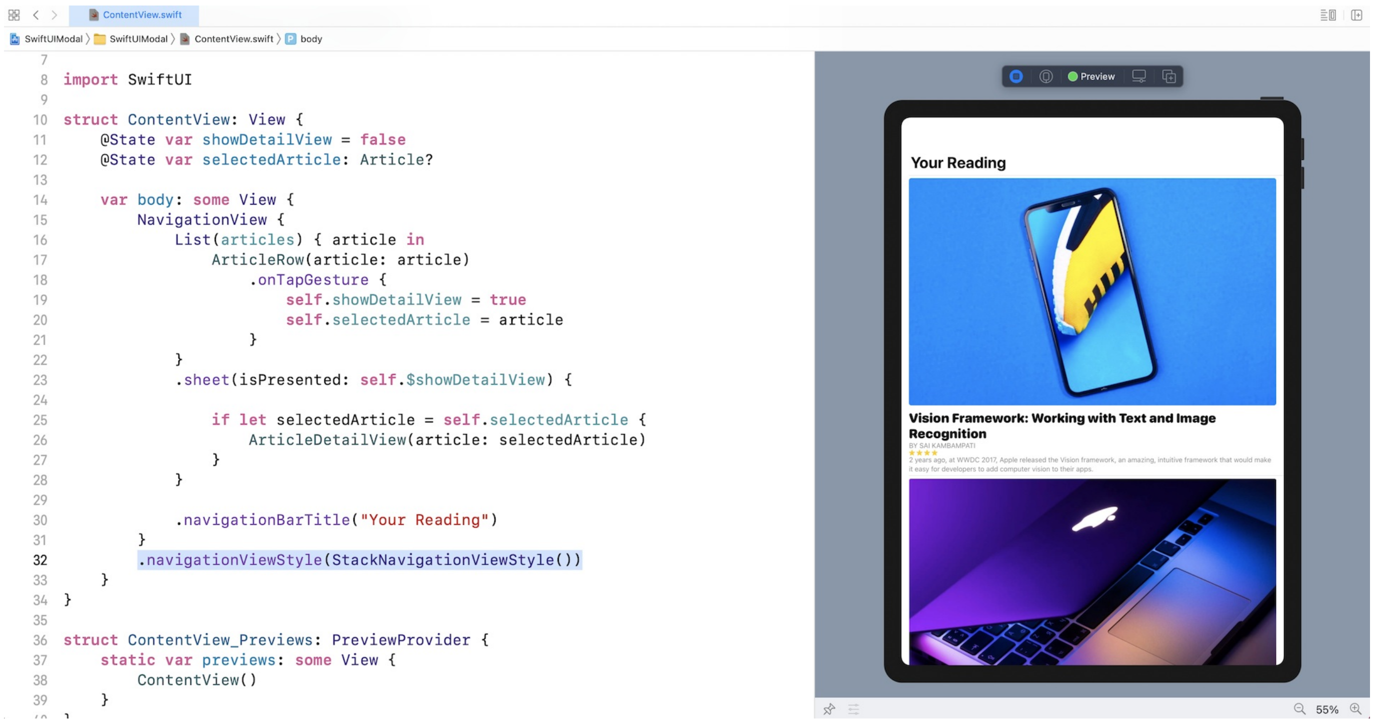
ContentView.swift
SwiftUIModal 〉 SwiftUIModal 〉 ContentView.swift 〉 No Selection
Preview
Your Reading
Vision Framework: Working with Text and Image Recog...
The Comprehensive Guide to the Stat...
55%

Figure 6. The list view appears like a sidebar menu

To force the navigation view to use the original non inset grouped liststyle, you attach the `navigationViewStyle` modifier to the navigation view and set the style to `StackNavigationViewStyle`.

```
.navigationViewStyle(StackNavigationViewStyle())
```

Once you make the change, this will result a standard list view even when the app is run on iPadOS.


ContentView.swift
SwiftUIModal 〉 SwiftUIModal 〉 ContentView.swift 〉 body
Preview
Your Reading
Vision Framework: Working with Text and Image Recognition
BY SAI KAMBAMPATI
55%

```
import SwiftUI

struct ContentView: View {
    @State var showDetailView = false
    @State var selectedArticle: Article?

    var body: some View {
        NavigationView {
            List(articles) { article in
                ArticleRow(article: article)
                    .onTapGesture {
                        self.showDetailView = true
                        self.selectedArticle = article
                    }
            }
            .sheet(isPresented: self.$showDetailView) {

                if let selectedArticle = self.selectedArticle {
                    ArticleDetailView(article: selectedArticle)
                }
            }

            .navigationBarTitle("Your Reading")
        }
        .navigationViewStyle(StackNavigationViewStyle())
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
```

Figure 7. Change the navigation view to StackNavigationViewStyle

Implementing the Modal View with Optional Binding

The `sheet` modifier also provides another way for you to present the modal view. Instead of having a boolean value to control the appearance of the modal view, the modifier lets you use an optional binding to achieve the same goal.

You can replace the `sheet` modifier like this:

```
.sheet(item: self.$selectedArticle) { article in
    ArticleDetailView(article: article)
}
```

In this case, the `sheet` modifier requires you to pass an optional binding. Here we specify the binding of the `selectedArticle`. What this means is that iOS will bring up the modal view only if the selected article has a value. The code in the closure specifies how the modal view looks, but it's slightly different than the code we wrote earlier.

For this approach, the `sheet` modifier will pass the selected article in the closure. The `article` parameter contains the selected article which is guaranteed to have a value. This is why we can use it to initiate an `ArticleDetailView` directly.

Since we no longer use the `showDetailView` variable, you can remove this line of code:

```
@State var showDetailView = false
```

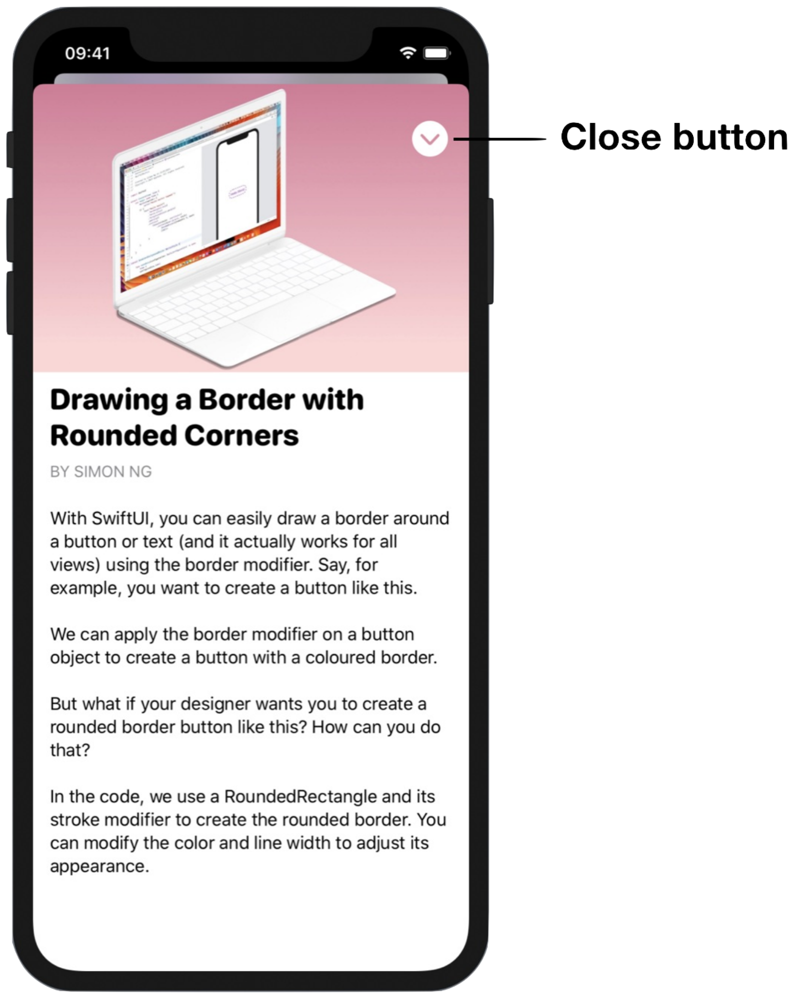
And remove the `self.showDetailView = true` from the `.onTapGesture` closure.

```
.onTapGesture {
    self.showDetailView = true
    ...
}
```

After changing the code, you can test the app again. Everything should work like the first version but the underlying code is cleaner than the original code.

Creating a Floating Button for Dismissing the Modal View

The modal view has built-in support for the swipe-down gesture. Currently, you can swipe down the modal view to close it. I guess this works pretty naturally for long-time iPhone users because apps like Facebook have used this type of gesture for dismissing a view. However, new comers may not know about this. It's better for us to develop a *Close* button as an alternative way of dismissing the modal view.



Drawing a Border with Rounded Corners

BY SIMON NG

With SwiftUI, you can easily draw a border around a button or text (and it actually works for all views) using the border modifier. Say, for example, you want to create a button like this.

We can apply the border modifier on a button object to create a button with a coloured border.

But what if your designer wants you to create a rounded border button like this? How can you do that?

In the code, we use a RoundedRectangle and its stroke modifier to create the rounded border. You can modify the color and line width to adjust its appearance.

Figure 8. The close button for dismissing the modal view

Switch over to `ArticleDetailView.swift`. We'll add the close button to the view as shown in figure 8.

Do you know how to position the button at the top-right corner? Try not to peek at my code and come up with your own implementation.

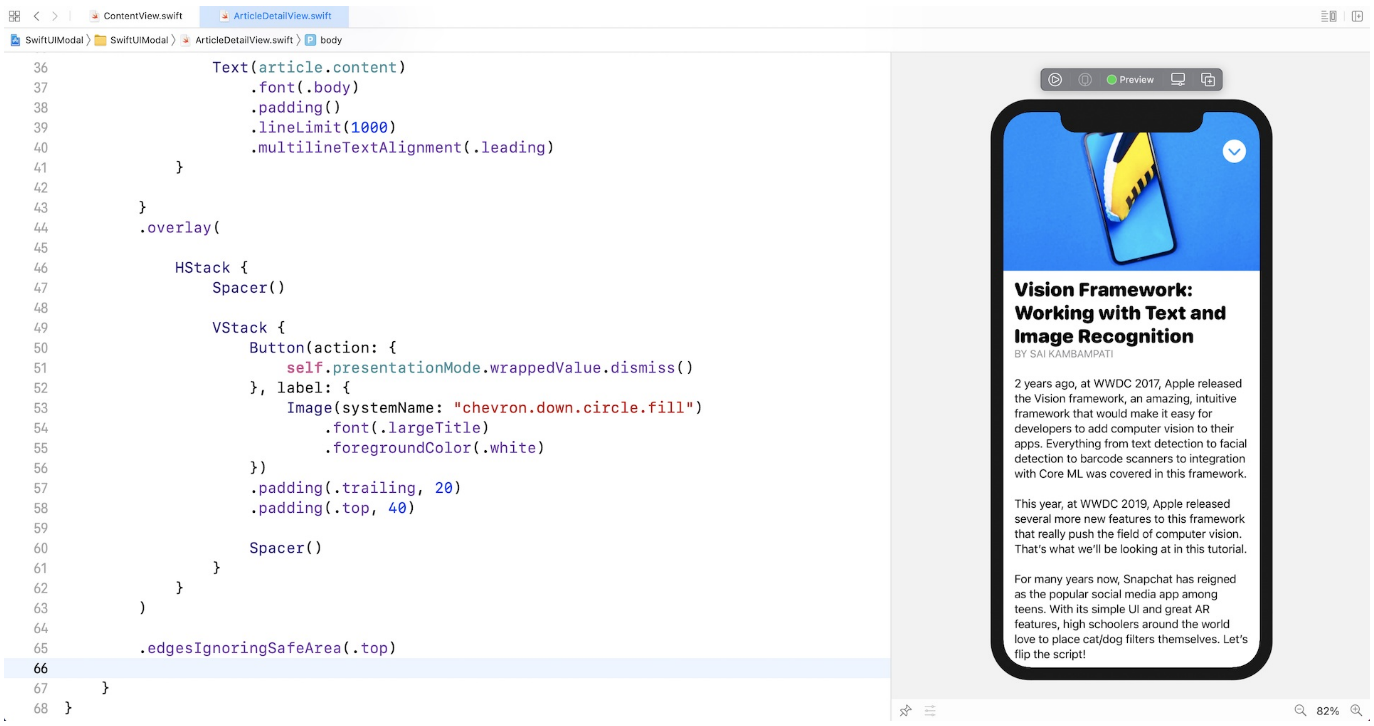
Similar to `NavigationView`, we can dismiss the modal view by using the `presentationMode` environment value. So, first declare the following variable in `ArticleDetailView`:

```
@Environment(\.presentationMode) var presentationMode
```

For the close button, we can attach the `overlay` modifier to the scroll view like this:

```
.overlay(

    HStack {
        Spacer()

        VStack {
            Button(action: {
                self.presentationMode.wrappedValue.dismiss()
            }, label: {
                Image(systemName: "chevron.down.circle.fill")
                    .font(.largeTitle)
                    .foregroundColor(.white)
            })
            .padding(.trailing, 20)
            .padding(.top, 40)

            Spacer()
        }
    }
)
```

The button will be overlayed on top of the scroll view so that it appears as a floating button. Even if you scroll down the view, the button will be stuck at the same position. To place the button at the top-right corner, here we use a `HStack` and a `VStack`, together with the help of `Spacer`. To dismiss the view, you call the `dismiss()` function of `presentationMode`.



```
                Text(article.content)
                    .font(.body)
                    .padding()
                    .lineLimit(1000)
                    .multilineTextAlignment(.leading)
            }

        }
        .overlay(

            HStack {
                Spacer()

                VStack {
                    Button(action: {
                        self.presentationMode.wrappedValue.dismiss()
                    }, label: {
                        Image(systemName: "chevron.down.circle.fill")
                            .font(.largeTitle)
                            .foregroundColor(.white)
                    })
                    .padding(.trailing, 20)
                    .padding(.top, 40)

                    Spacer()
                }
            }
        )

        .edgesIgnoringSafeArea(.top)

    }
}
```

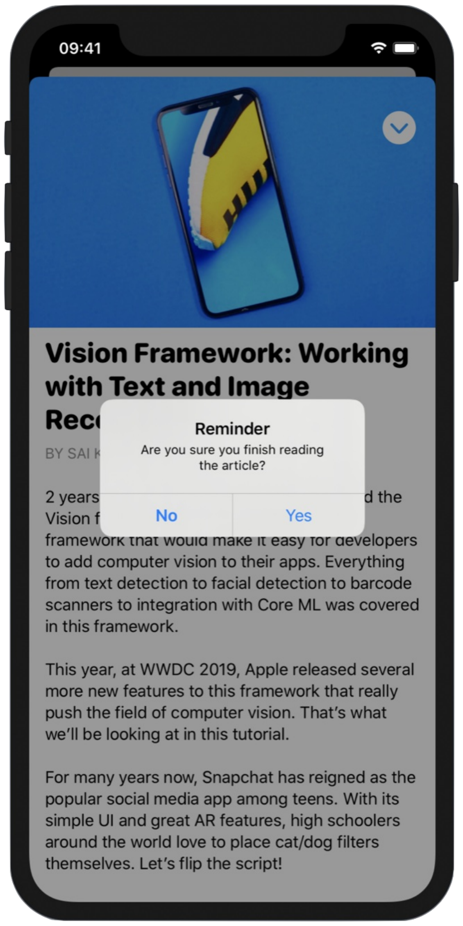
ContentView.swift
ArticleDetailView.swift
SwiftUIModal 〉 SwiftUIModal 〉 ArticleDetailView.swift 〉 body
Preview
Vision Framework: Working with Text and Image Recognition
BY SAI KAMBAMPATI
2 years ago, at WWDC 2017, Apple released the Vision framework, an amazing, intuitive framework that would make it easy for developers to add computer vision to their apps. Everything from text detection to facial detection to barcode scanners to integration with Core ML was covered in this framework.
This year, at WWDC 2019, Apple released several more new features to this framework that really push the field of computer vision. That's what we'll be looking at in this tutorial.
For many years now, Snapchat has reigned as the popular social media app among teens. With its simple UI and great AR features, high schoolers around the world love to place cat/dog filters themselves. Let's flip the script!
82%

Figure 9. Implementing the close button

Run the app in a simulator or switch over to `ContentView` and run it in the canvas. You should be able to dismiss the modal view by clicking the close button.

Using Alerts

In addition to the card-like modal views, *Alerts* are another kind of modal view. When it's presented, the entire screen is blocked. You can't dismiss the dialog without choosing one of the options. Figure 10 shows a sample alert that we're going to implement in our demo project. What we're going to display an alert after a user taps the close button.



Vision Framework: Working with Text and Image Rec

BY SAI K

Reminder

Are you sure you finish reading the article?

2 years d the
Vision f
framework that would make it easy for developers to add computer vision to their apps. Everything from text detection to facial detection to barcode scanners to integration with Core ML was covered in this framework.

This year, at WWDC 2019, Apple released several more new features to this framework that really push the field of computer vision. That's what we'll be looking at in this tutorial.

For many years now, Snapchat has reigned as the popular social media app among teens. With its simple UI and great AR features, high schoolers around the world love to place cat/dog filters themselves. Let's flip the script!

Figure 10. Displaying an alert

In SwiftUI, you create an alert using the `Alert` struct. Here is an example of `Alert` :

```
Alert(title: Text("Warning"), message: Text("Are you sure you want to leave?"), pr
imaryButton: .default(Text("Confirm")), secondaryButton: .cancel())
```

The sample code initiates an alert view with the title "Warning". The alert prompt also displays the message, "Are you sure you want to leave" to the user. There are two buttons in the alert view: *Confirm* and *Cancel*.

Here is the code to create the alert as shown in figure 10:

```
Alert(title: Text("Reminder"), message: Text("Are you sure you are finished readin
g the article?"), primaryButton: .default(Text("Yes"), action: { self.presentation
Mode.wrappedValue.dismiss() }), secondaryButton: .cancel(Text("No")))
```

It's similar to the previous code snippet except that the primary button has the `action` parameter. This alert asks the user whether he/she has finished reading the article. If the user chooses *Yes*, the modal view will be closed. Otherwise, the modal view will stay open.

Now that we have the code for creating the alert, the question is how can we trigger the display of the alert? SwiftUI provides the `alert` modifier that you can attach it to any view. Again, you use a boolean variable to control the display of the alert. So, declare a state variable in `ArticleDetailView` :

```
@State private var showAlert = false
```

Next, attach the `alert` modifier to the `ScrollView` :

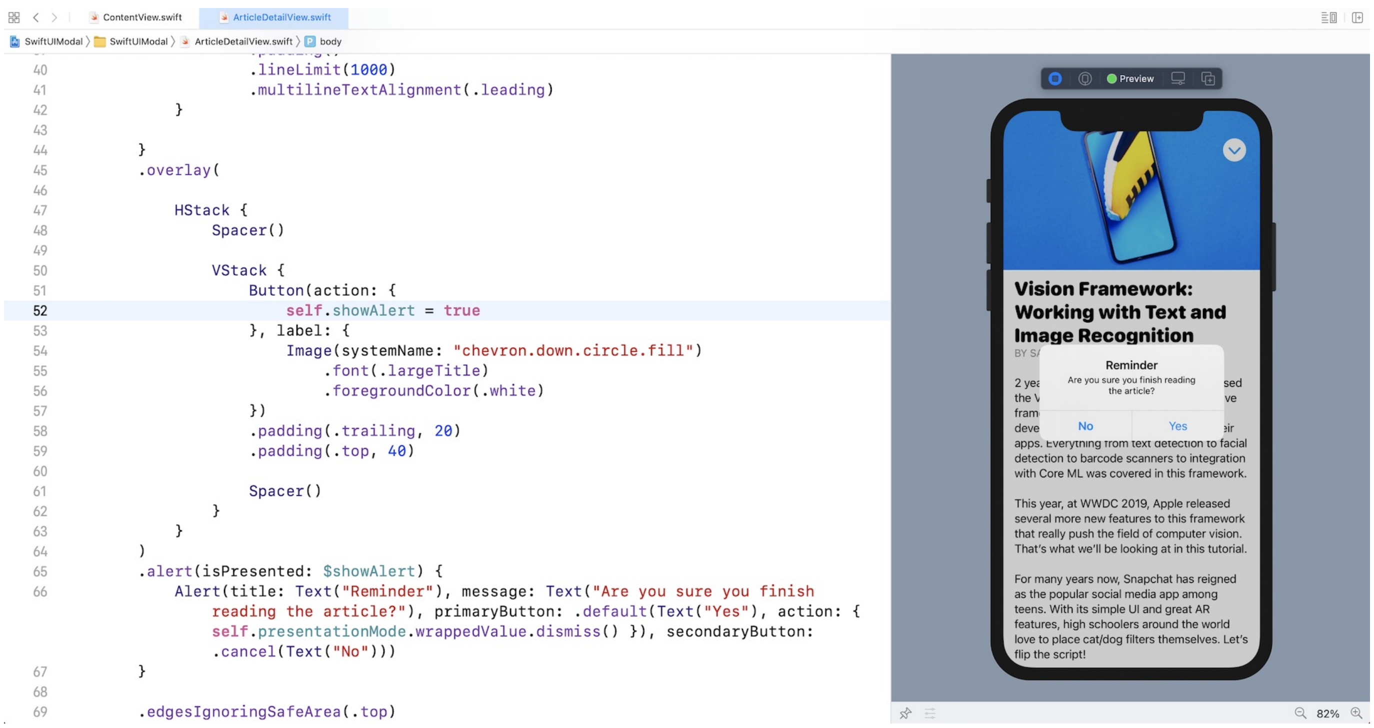
```
.alert(isPresented: $showAlert) {
    Alert(title: Text("Reminder"), message: Text("Are you sure you are finished re
ading the article?"), primaryButton: .default(Text("Yes"), action: { self.presenta
tionMode.wrappedValue.dismiss() }), secondaryButton: .cancel(Text("No")))
}
```

There is still one thing left. When should we trigger this alert? In other words, when should we set `showAlert` to `true` ?

Obviously, the app should display the alert when someone taps the close button. So, replace the button's action like this:

```
Button(action: {
    self.showAlert = true
}, label: {
    Image(systemName: "chevron.down.circle.fill")
        .font(.largeTitle)
        .foregroundColor(.white)
})
```

Instead of dismissing the modal view directly, we instruct iOS to show the alert by setting `showAlert` to `true`. You're now ready to test the app. When you tap the close button, you'll see the alert. The modal view will be dismissed if you choose "Yes."


ContentView.swift
ArticleDetailView.swift
SwiftUIModal 〉 SwiftUIModal 〉 ArticleDetailView.swift 〉 body
Preview
82%

```
                    .lineLimit(1000)
                    .multilineTextAlignment(.leading)
            }

        }
        .overlay(

            HStack {
                Spacer()

                VStack {
                    Button(action: {
                        self.showAlert = true
                    }, label: {
                        Image(systemName: "chevron.down.circle.fill")
                            .font(.largeTitle)
                            .foregroundColor(.white)
                    })
                    .padding(.trailing, 20)
                    .padding(.top, 40)

                    Spacer()
                }
            }
        )
        .alert(isPresented: $showAlert) {
            Alert(title: Text("Reminder"), message: Text("Are you sure you finish
                reading the article?"), primaryButton: .default(Text("Yes"), action: {
                self.presentationMode.wrappedValue.dismiss() }), secondaryButton:
                .cancel(Text("No")))
        }

        .edgesIgnoringSafeArea(.top)
```

Vision Framework: Working with Text and Image Recognition

BY SA

Reminder

Are you sure you finish reading the article?

No Yes

2 yea sed the V ve fram deve eir apps. Everything from text detection to facial detection to barcode scanners to integration with Core ML was covered in this framework.

This year, at WWDC 2019, Apple released several more new features to this framework that really push the field of computer vision. That's what we'll be looking at in this tutorial.

For many years now, Snapchat has reigned as the popular social media app among teens. With its simple UI and great AR features, high schoolers around the world love to place cat/dog filters themselves. Let's flip the script!

Figure 11. Tapping the close button will show you the alert

Displaying a Full Screen Modal View

Starting with iOS 13, the modal view doesn't cover the whole screen by default. If you want to present a full screen modal view, you can use the `.fullScreenCover` modifier introduced in iOS 14. Instead of using `.sheet` to bring up a modal view, you can apply the `.fullScreenCover` modifier like this:

```
.fullScreenCover(item: self.$selectedArticle) { article in
    ArticleDetailView(article: article)
}
```

Summary

You've learned how to present a modal view, implement a floating button, and show an alert. The latest release of iOS continues to encourage people interact with the device using gestures and provides built-in support for common gestures. Without writing a line of code, you can let users swipe down the screen to dismiss a modal view.

The API design of both modal view and alert is very similar. It monitors a state variable to determine whether the modal view (or alert) should be triggered. Once you understand this technique, the implementation shouldn't be difficult for you.

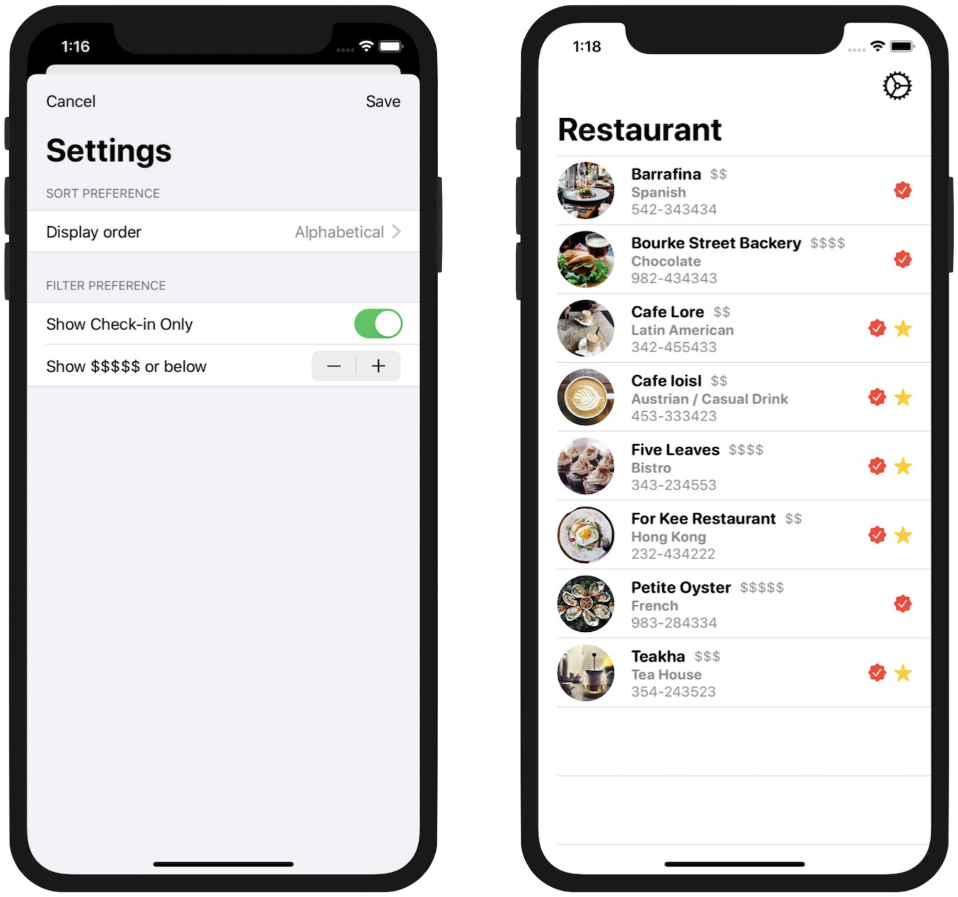
For reference, you can download the complete modal project here:

- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUIModal.zip)

Chapter 13
Building a Form with Picker, Toggle and Stepper

Mobile apps use forms to interact with users and solicit required data from them. Every day, when using your iPhone, it's very likely you will come across a mobile form. For example, a calendar app may present you a form to fill in the information for a new event. A shopping app asks you to provide the shipping and payment information by showing you a form. As a user, I can't deny that I hate filling out forms. That said, as a developer, these forms help us interact with users and ask for information to complete certain operations. Developing a form is definitely an essential skill you need to grasp.

In the SwiftUI framework, there is a special UI control called *Form*. With this new control, you can easily build a form. I will show you how to build a form using this *Form* component. While building out a form, you will also learn how to work with common controls like picker, toggle, and stepper.



Settings

SORT PREFERENCE

Display order Alphabetical

FILTER PREFERENCE

Show Check-in Only

Show $$$$$ or below

Restaurant

Barrafina	$$	Spanish	542-343434
Bourke Street Backery	$$$$	Chocolate	982-434343
Cafe Lore	$$	Latin American	342-455433
Cafe loisl	$$	Austrian / Casual Drink	453-333423
Five Leaves	$$$$	Bistro	343-234553
For Kee Restaurant	$$	Hong Kong	232-434222
Petite Oyster	$$$$$	French	983-284334
Teakha	$$$	Tea House	354-243523

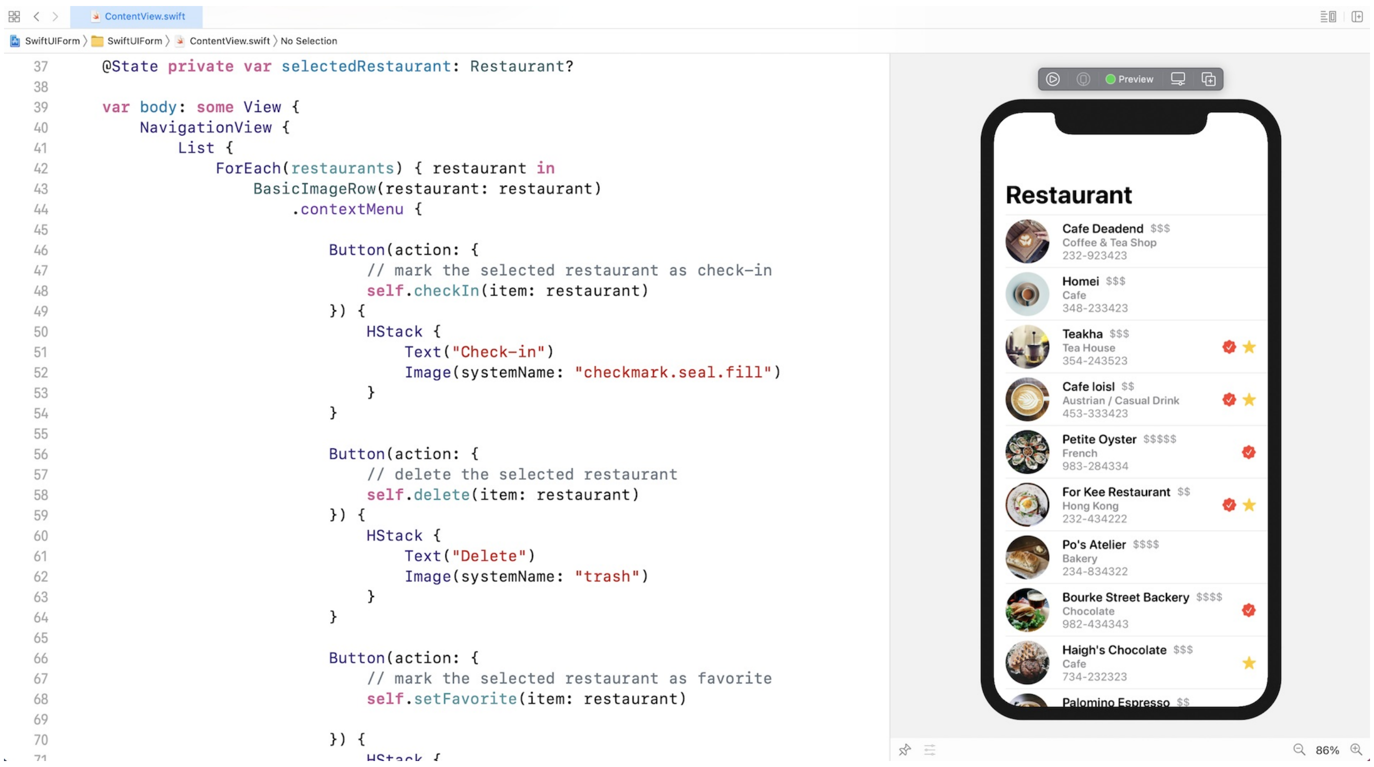
Figure 1. Building a Setting screen

Okay, what project are we going to work on? Take a look at figure 1. We're going to build a Setting screen for the Restaurant app we have been working on in earlier chapters. The screen provides users with the options to configure the order and filter preferences. This type of form is very common in real-life projects. Once you understand how it works, you will be able to create your own form in your app projects.

In this chapter, we will focus on implementing the form layout. You will understand how to use the *Form* component to lay out a setting screen. We will also implement a picker for selecting a sort preference. We'll also create a toggle and a stepper for indicating filter preferences. Once you understand how to lay out a form, in the next chapter, I will show you how to make the app fully functional by updating the list in accordance with the user's preferences. You'll learn how to store user preferences, share data between views and monitor data update with `@EnvironmentObject`.

Preparing the Starter Project

To save you time from building the restaurant list again, I have created a starter project for you. Download it from https://www.appcoda.com/resources/swiftui2/SwiftUIFormStarter.zip. Once downloaded, open the `SwiftUIForm.xcodeproj` file with Xcode. Preview `ContentView.swift` in the canvas and you'll see a familiar UI except that it incorporates more detailed information for a restaurant.


ContentView.swift
SwiftUIForm 〉 SwiftUIForm 〉 ContentView.swift 〉 No Selection
Preview
86%

```
    @State private var selectedRestaurant: Restaurant?

    var body: some View {
        NavigationView {
            List {
                ForEach(restaurants) { restaurant in
                    BasicImageRow(restaurant: restaurant)
                        .contextMenu {

                            Button(action: {
                                // mark the selected restaurant as check-in
                                self.checkIn(item: restaurant)
                            }) {
                                HStack {
                                    Text("Check-in")
                                    Image(systemName: "checkmark.seal.fill")
                                }
                            }

                            Button(action: {
                                // delete the selected restaurant
                                self.delete(item: restaurant)
                            }) {
                                HStack {
                                    Text("Delete")
                                    Image(systemName: "trash")
                                }
                            }

                            Button(action: {
                                // mark the selected restaurant as favorite
                                self.setFavorite(item: restaurant)

                            }) {
                                HStack {
```

```
Restaurant
Cafe Deadend $$$
Coffee & Tea Shop
232-923423
Homei $$$
Cafe
348-233423
Teakha $$$
Tea House
354-243523
Cafe loisl $$
Austrian / Casual Drink
453-333423
Petite Oyster $$$$$
French
983-284334
For Kee Restaurant $$
Hong Kong
232-434222
Po's Atelier $$$$
Bakery
234-834322
Bourke Street Backery $$$$
Chocolate
982-434343
Haigh's Chocolate $$$
Cafe
734-232323
Palomino Espresso $$
```

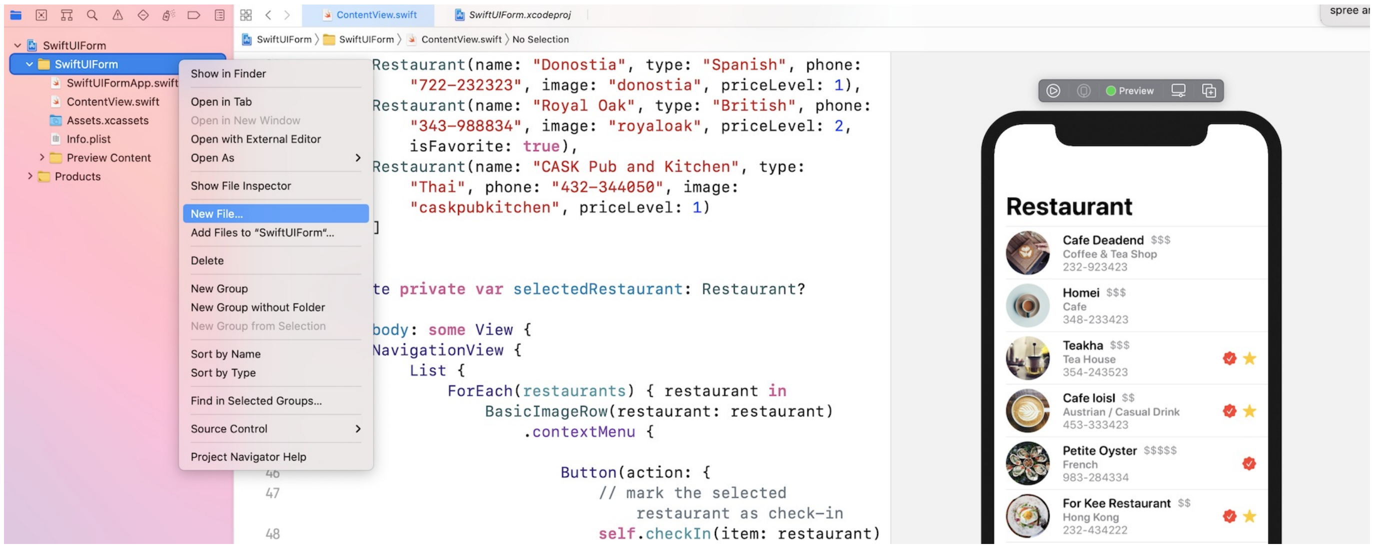
Figure 2. The restaurant list view

The `Restaurant` struct now has three more properties: *type*, *phone*, and *priceLevel*. I think both type and phone are self explanatory. Price level stores an integer of range 1 to 5 reflecting the average cost of the restaurant. The `restaurants` array has been prepopulated with some sample data. For later testing, some of the restaurants have `isFavorite` and `isCheckIn` set to `true`. This is why you see some check-in and favorite indicators displayed in the preview.

Building the Form UI

As mentioned, SwiftUI provides a UI component called `Form` for building the form UI. It's a container for holding and grouping controls (e.g. toggle) for data entry. Rather than explaining its usage to you, it's better to jump right into the implementation. You will understand how to use the component along the way.

Since we will build a separate screen for Settings, let's create a new file for the form. In the project navigator, right click the *SwiftUIForm* folder and choose "New File...." Next, select to use *SwiftUI View* as the template and name the file *SettingView.swift*.



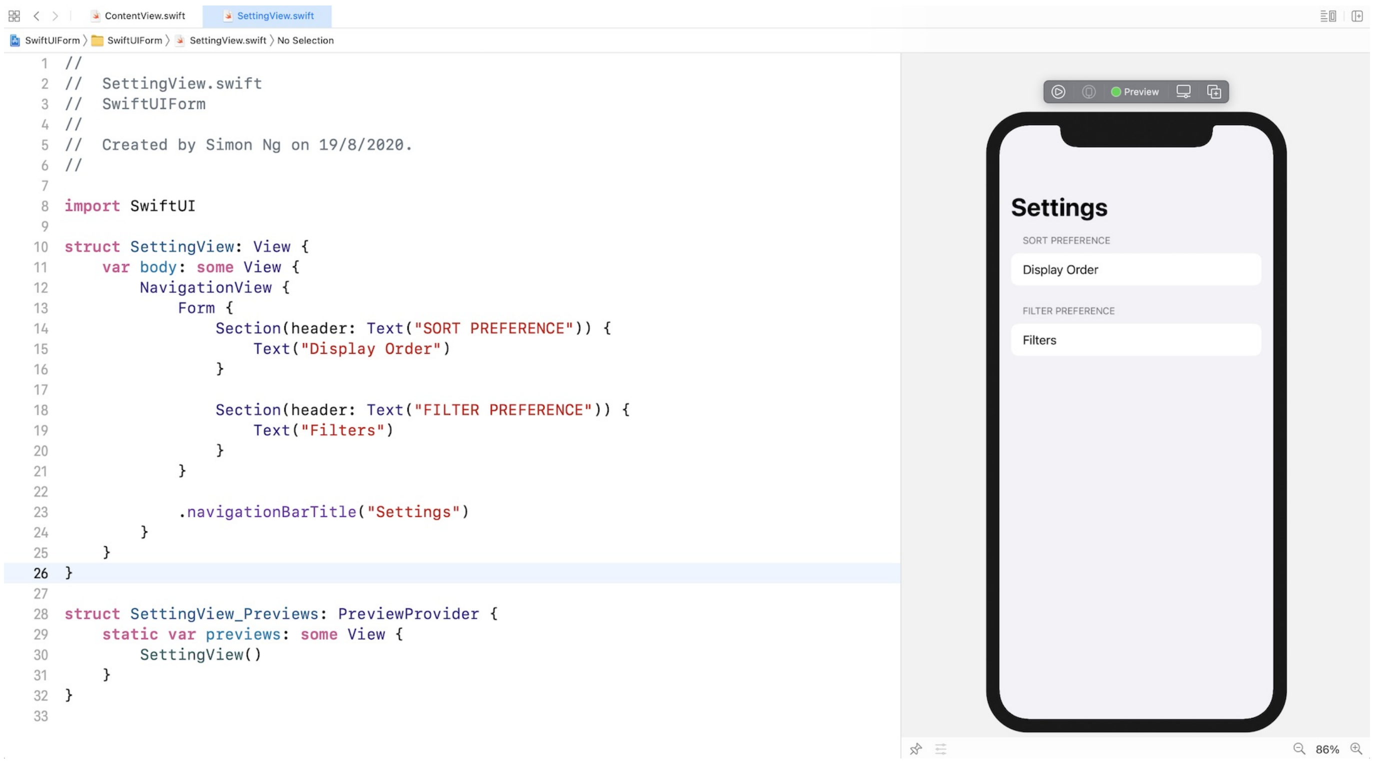
```
Restaurant(name: "Donostia", type: "Spanish", phone:
    "722-232323", image: "donostia", priceLevel: 1),
Restaurant(name: "Royal Oak", type: "British", phone:
    "343-988834", image: "royaloak", priceLevel: 2,
    isFavorite: true),
Restaurant(name: "CASK Pub and Kitchen", type:
    "Thai", phone: "432-344050", image:
    "caskpubkitchen", priceLevel: 1)
]

te private var selectedRestaurant: Restaurant?

body: some View {
NavigationView {
    List {
        ForEach(restaurants) { restaurant in
            BasicImageRow(restaurant: restaurant)
                .contextMenu {

                    Button(action: {
                        // mark the selected
                            restaurant as check-in
                        self.checkIn(item: restaurant)
```

Figure 3. Creating a new SwiftUI file

Now, let's start by creating the form. Replace `SettingView` with this:

```
struct SettingView: View {
    var body: some View {
        NavigationView {
            Form {
                Section(header: Text("SORT PREFERENCE")) {
                    Text("Display Order")
                }

                Section(header: Text("FILTER PREFERENCE")) {
                    Text("Filters")
                }
            }

            .navigationBarTitle("Settings")
        }
    }
}
```

To lay out a form, you use the `Form` container. Inside it, you add sections and form components (text field, picker, toggle etc.). In the code above, we create two sections: *Sort Preference* and *Filter Preference*. For each section, we have a text view. Your canvas should display a preview like that shown in figure 4.



```swift
//
//  SettingView.swift
//  SwiftUIForm
//
//  Created by Simon Ng on 19/8/2020.
//

import SwiftUI

struct SettingView: View {
    var body: some View {
        NavigationView {
            Form {
                Section(header: Text("SORT PREFERENCE")) {
                    Text("Display Order")
                }

                Section(header: Text("FILTER PREFERENCE")) {
                    Text("Filters")
                }
            }

            .navigationBarTitle("Settings")
        }
    }
}

struct SettingView_Previews: PreviewProvider {
    static var previews: some View {
        SettingView()
    }
}
```

Figure 4. Create a simple form with two sections

Creating a Picker View

When presenting a form, you certainly want to secure some information. It's useless if we just present a Text component. In the actual form, we use three types of UI controls for user input including a picker view, a toggle, and a stepper. Let's begin with the sort preference. For that, we will implement a picker view.

For the sort preference, users are allowed to choose the display order of the restaurant list, in which we offer three options for them to choose:

1. Alphabetically
2. Show Favorite First
3. Show Check-in First

A `Picker` control is very suitable for handling this kind of input. First, You represent each of the options above in an array. Let's declare an array named `displayOrders` in `SettingView` :

```
private var displayOrders = [ "Alphabetical", "Show Favorite First", "Show Check-i
n First"]
```

To use a picker, you also need to declare a state variable to store the user's selected option. In `SettingView`, declare the variable like this:

```
@State private var selectedOrder = 0
```

Here, `0` means the first item of `displayOrders`. Now replace the *SORT PREFERENCE* section like this:

```
Section(header: Text("SORT PREFERENCE")) {
    Picker(selection: $selectedOrder, label: Text("Display order")) {
        ForEach(0 ..< displayOrders.count, id: \.self) {
            Text(self.displayOrders[$0])
        }
    }
}
```

This is how you create a picker container in SwiftUI. You have to provide two values; the binding of the selection (i.e. `$selectedOrder`) and the text label describing what the option is for. In the closure, you display the available options using `Text`.

In the canvas, you should see that the *Display Order* is set to *Alphabetical*. This is because `selectedOrder` is default to `0`. If you click the *Play* button to text the view, tapping the option will bring you to the next screen, showing you all the available options. You can pick any of the options (e.g. Show Favorite First) for testing. When you go back to the Setting screen, the *Display Order* will become your selection. This is the power of the `@State` keyword. It automatically monitors the changes and helps you store the state of the selection.

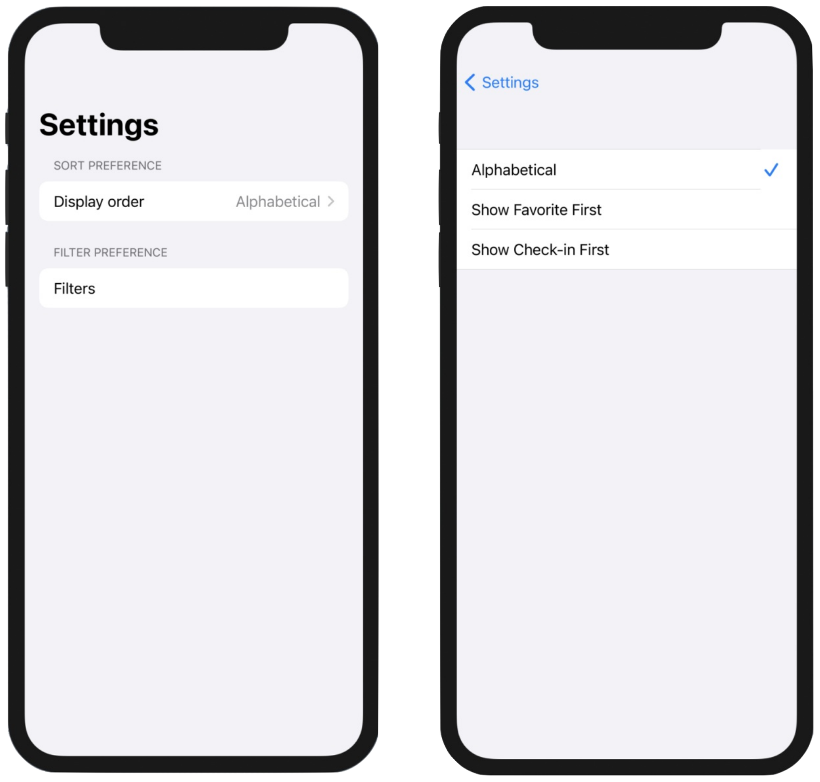


Figure 5. Using Picker view for display order selection

Working with Toggle Switches

Next, let's move onto the input for setting the filter preference. First, we will implement a toggle (or a switch) to enable/disable the "Show Check-in Only" filter. A toggle has only two states: *ON* or *OFF*. THis control is useful for prompting users to choose between two mutually exclusive options.

Creating a toggle switch using SwiftUI is quite straightforward. Similar to `Picker`, we have to declare a state variable to store the current setting of the toggle. So, declare the following variable in `SettingView`:

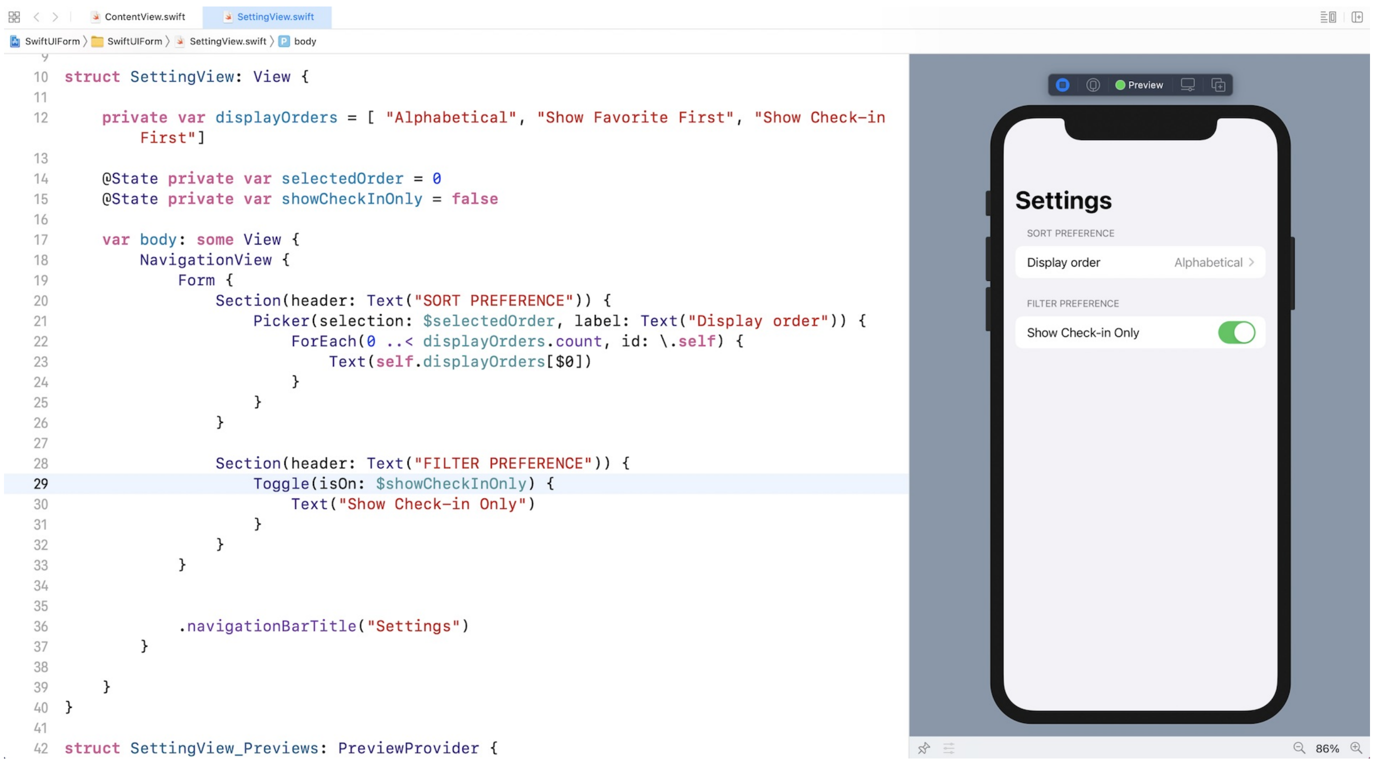
```
@State private var showCheckInOnly = false
```

Then, update the *FILTER PREFERENCE* section like this:

```
Section(header: Text("FILTER PREFERENCE")) {
    Toggle(isOn: $showCheckInOnly) {
        Text("Show Check-in Only")
    }
}
```

You use `Toggle` to create a toggle switch and pass it the current state of the toggle. In the closure, you present the description of the toggle. Here, we simply use a `Text` view.

The canvas should show a toggle switch under the Filter Preference section. If you run the app, you can switch it between the ON and OFF states. Similarly, the state variable `showCheckInOnly` will always keep track of the user selection.



```
struct SettingView: View {

    private var displayOrders = [ "Alphabetical", "Show Favorite First", "Show Check-in
        First"]

    @State private var selectedOrder = 0
    @State private var showCheckInOnly = false

    var body: some View {
        NavigationView {
            Form {
                Section(header: Text("SORT PREFERENCE")) {
                    Picker(selection: $selectedOrder, label: Text("Display order")) {
                        ForEach(0 ..< displayOrders.count, id: \.self) {
                            Text(self.displayOrders[$0])
                        }
                    }
                }

                Section(header: Text("FILTER PREFERENCE")) {
                    Toggle(isOn: $showCheckInOnly) {
                        Text("Show Check-in Only")
                    }
                }
            }


            .navigationBarTitle("Settings")
        }

    }
}

struct SettingView_Previews: PreviewProvider {
```

Figure 6. Showing a toggle switch

Using Steppers

The last UI control in the setting form is a *Stepper*. Again, referring to figure 1, users can filter the restaurants by setting the pricing level. Each of the restaurants has a pricing indicator with a range of 1 to 5. Users can adjust the price level to narrow down the number of restaurants displayed in the list view.

In the setting form, we will implement a stepper for users to adjust this setting. Basically, a Stepper in iOS shows a text field and *plus* and *minus* buttons to perform increment and decrement actions on the text field.

To implement a stepper in SwiftUI, we first need a state variable to hold the current value of the stepper. In this case, this variable stores the user's price level filter. Declare the state variable in `SettingView` like this:

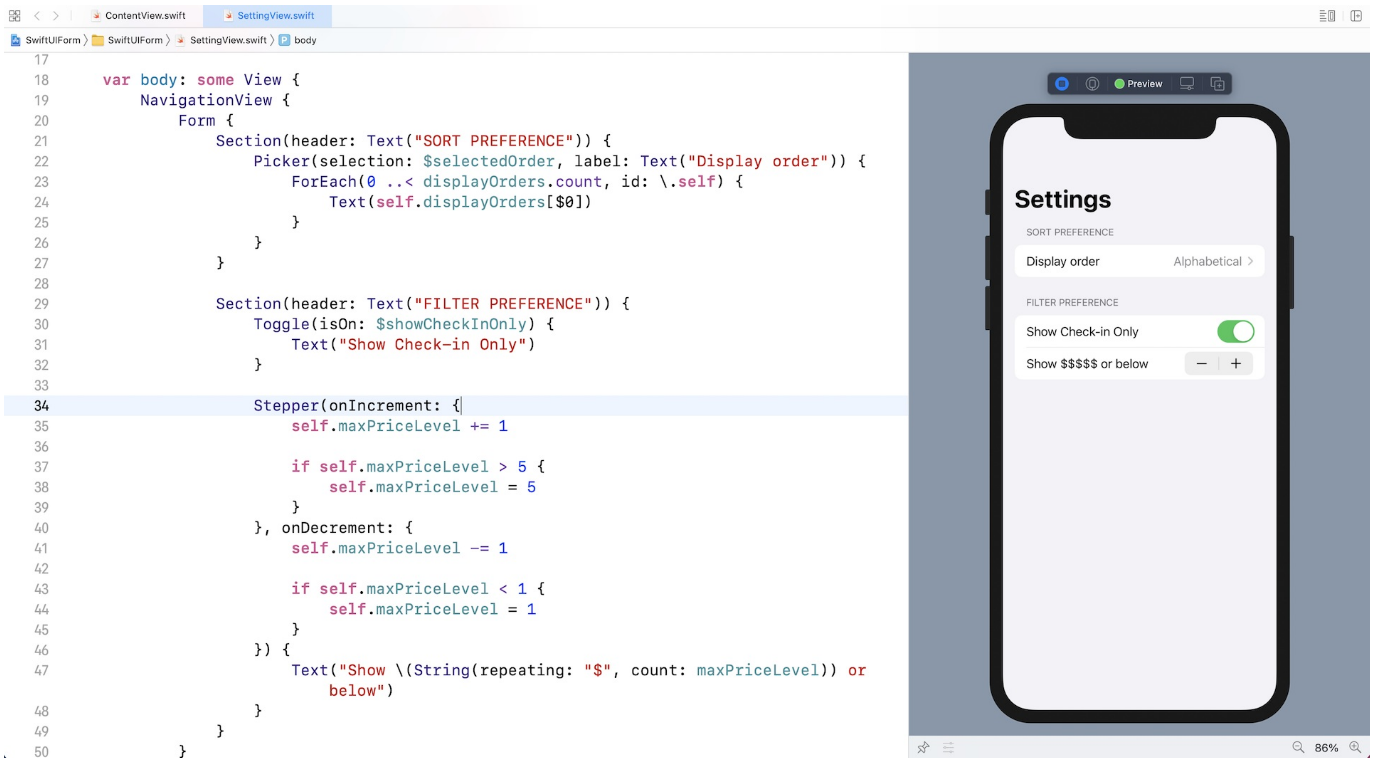
```
@State private var maxPriceLevel = 5
```

By default, we set the `maxPriceLevel` to `5`. Update the *FILTER PREFERENCE* section like this:

```
Section(header: Text("FILTER PREFERENCE")) {
    Toggle(isOn: $showCheckInOnly) {
        Text("Show Check-in Only")
    }

    Stepper(onIncrement: {
        self.maxPriceLevel += 1

        if self.maxPriceLevel > 5 {
            self.maxPriceLevel = 5
        }
    }, onDecrement: {
        self.maxPriceLevel -= 1

        if self.maxPriceLevel < 1 {
            self.maxPriceLevel = 1
        }
    }) {
        Text("Show \(String(repeating: "$", count: maxPriceLevel)) or below")
    }
}
```

You create a stepper by initiating a `Stepper` component. For the `onIncrement` parameter, you specify the action to perform when the `+` button is clicked. In the code, we simply increase `maxPriceLevel` by 1. Conversely, the code specified in the `onDecrement` parameter will be executed when the `-` button is clicked.

Since the price level is in the range of 1 to 5, we perform a check to make sure the value of `maxPriceLevel` is between the value of 1 and 5. In the closure, we display the text description of the filter preference. The maximum price level is indicated by dollar signs.



```
    var body: some View {
        NavigationView {
            Form {
                Section(header: Text("SORT PREFERENCE")) {
                    Picker(selection: $selectedOrder, label: Text("Display order")) {
                        ForEach(0 ..< displayOrders.count, id: \.self) {
                            Text(self.displayOrders[$0])
                        }
                    }
                }

                Section(header: Text("FILTER PREFERENCE")) {
                    Toggle(isOn: $showCheckInOnly) {
                        Text("Show Check-in Only")
                    }

                    Stepper(onIncrement: {
                        self.maxPriceLevel += 1

                        if self.maxPriceLevel > 5 {
                            self.maxPriceLevel = 5
                        }
                    }, onDecrement: {
                        self.maxPriceLevel -= 1

                        if self.maxPriceLevel < 1 {
                            self.maxPriceLevel = 1
                        }
                    }) {
                        Text("Show \(String(repeating: "$", count: maxPriceLevel)) or
                            below")
                    }
                }
            }
```

Figure 7. Implementing a stepper

Click the *Play* button to run the app. The number of $ signs will be adjusted when you click the `+` / `-` button.

Presenting the Form

Now that you've completed the form UI, the next step is to present the form to users. For the demo, we will present this form as a modal view. In the content view, we will add a *Setting* button in the navigation bar to trigger the setting view.

Switch over to `ContentView.swift`. I assume you've read the modal view chapter, so I will not explain the code in depth. First, we need a variable to keep track of the state (i.e. shown or not shown) of the modal view. Insert the following line of code to declare the state variable:

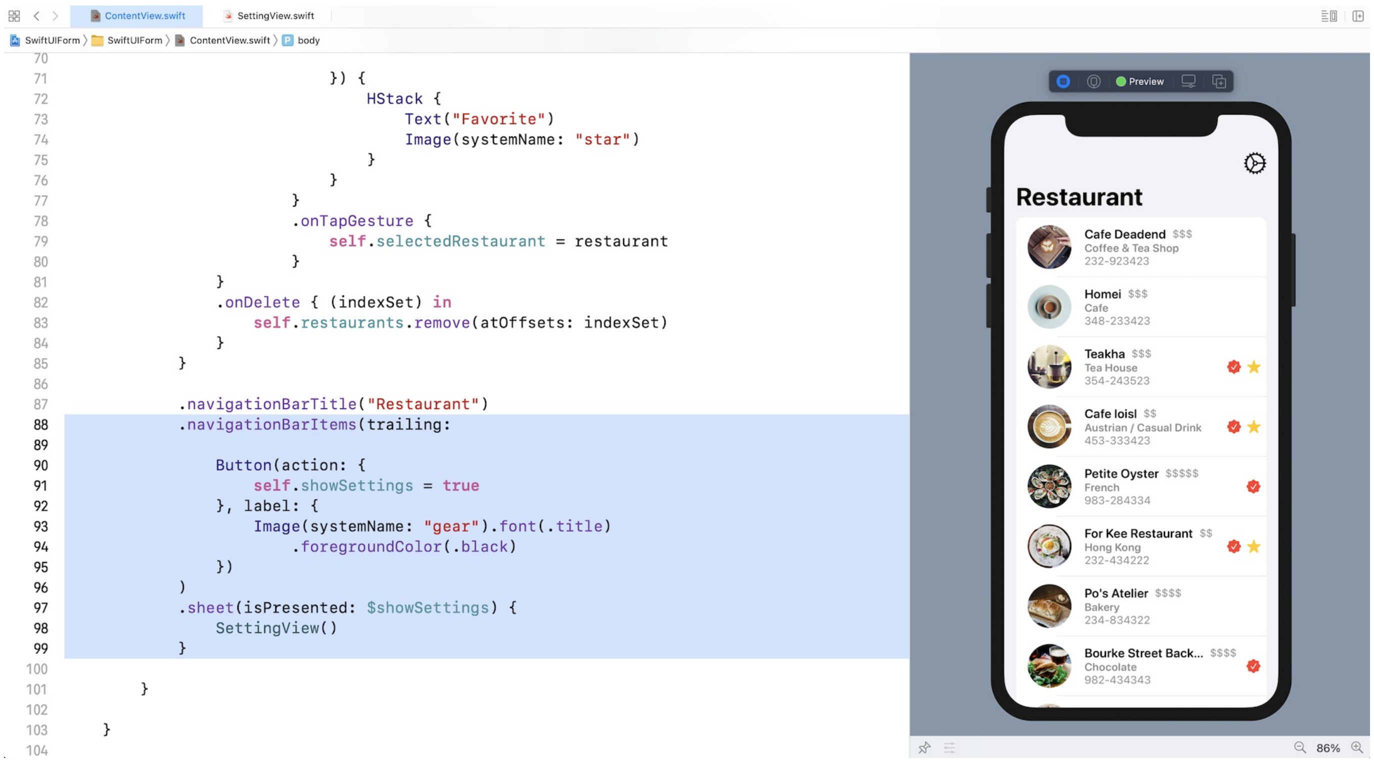
```
@State private var showSettings: Bool = false
```

Next, insert the following modifiers to the `NavigationView`:

```
.navigationBarItems(trailing:

    Button(action: {
        self.showSettings = true
    }, label: {
        Image(systemName: "gear").font(.title)
            .foregroundColor(.black)
    })
)
.sheet(isPresented: $showSettings) {
    SettingView()
}
```

The `navigationBarItems` modifier let you add a button in the navigation bar. You're allowed to create a button at the leading or trailing position of the navigation bar. Since we want to display the button at the top-right corner, we use the `trailing` parameter. The `sheet` modifier is used for presenting the `SettingView` as a modal view.

In the canvas, you should see a *gear* icon in the navigation bar. Run the app and click the *gear* icon, it should bring up the *Setting* view.



```
                        }) {
                            HStack {
                                Text("Favorite")
                                Image(systemName: "star")
                            }
                        }
                    }
                    .onTapGesture {
                        self.selectedRestaurant = restaurant
                    }
                }
                .onDelete { (indexSet) in
                    self.restaurants.remove(atOffsets: indexSet)
                }
            }

            .navigationBarTitle("Restaurant")
            .navigationBarItems(trailing:

                Button(action: {
                    self.showSettings = true
                }, label: {
                    Image(systemName: "gear").font(.title)
                        .foregroundColor(.black)
                })
            )
            .sheet(isPresented: $showSettings) {
                SettingView()
            }

        }

    }
```

Restaurant

Cafe Deadend $$$
Coffee & Tea Shop
232-923423

Homei $$$
Cafe
348-233423

Teakha $$$
Tea House
354-243523

Cafe loisl $$
Austrian / Casual Drink
453-333423

Petite Oyster $$$$$
French
983-284334

For Kee Restaurant $$
Hong Kong
232-434222

Po's Atelier $$$$
Bakery
234-834322

Bourke Street Back... $$$$
Chocolate
982-434343

Figure 8. Creating the navigation bar button

Similar to what we have experienced in the previous chapter, you should see an extra padding around the list view after attaching the `.sheet` modifier. To revert to the original style of the list view, you can attach the `.navigationViewStyle` modifier to `NavigationView` and set its style to `StackNavigationViewStyle` like this:

```
.navigationViewStyle(StackNavigationViewStyle())
```

Exercise

The only way to dismiss the Setting view is by using the swipe-down gesture. In the modal view chapter, you learned how to dismiss a modal view programmatically. As a refresher exercise, please create two buttons (*Save* & *Cancel*) in the navigation bar. You are not required to implement these button. When a user taps any of the buttons, just dismiss the setting view.

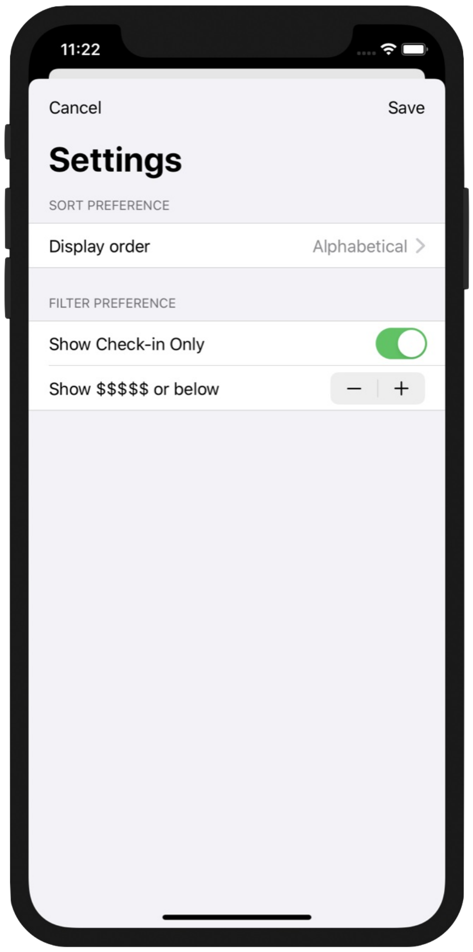


Figure 9. Adding two buttons (Save & Cancel) in the navigation bar

What's Coming Next

I hope you understand how the *Form* component works and that you know how to build a form UI with components like Picker and Stepper. Currently, the app can't store the user preferences permanently. Every time you launch the app, the settings are reset to the original settings. In the next chapter, I will show you how to save these settings in local storage. More importantly, we will update the list view in accordance with the user's preferences.

For reference, you can download the complete form project here:

- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUIForm.zip)

Chapter 14
Data Sharing with Combine and Environment Objects

In the previous chapter, you learned how to lay out a form using the *Form* component. However, the form is not functional yet. No matter what options you select, the list view doesn't change to reflect the user's preference. This is what we're going to discuss and implement in this chapter. We will continue to develop the settings screen and make the app fully functional by updating the restaurant list in reference to the user's personal preference.

Specifically, there are a few topics we will discuss in later sections:

1. How to use enum to better organize our code
2. How to store the user's preference permanently using UserDefaults
3. How to share data using Combine and @EnvironmentObject

If you haven't finished the exercise in the previous chapter, I encourage you to spend some time on it. That said, if you can't wait to read this chapter, you can download the project from https://www.appcoda.com/resources/swiftui2/SwiftUIForm.zip.

Refactoring the Code with Enum

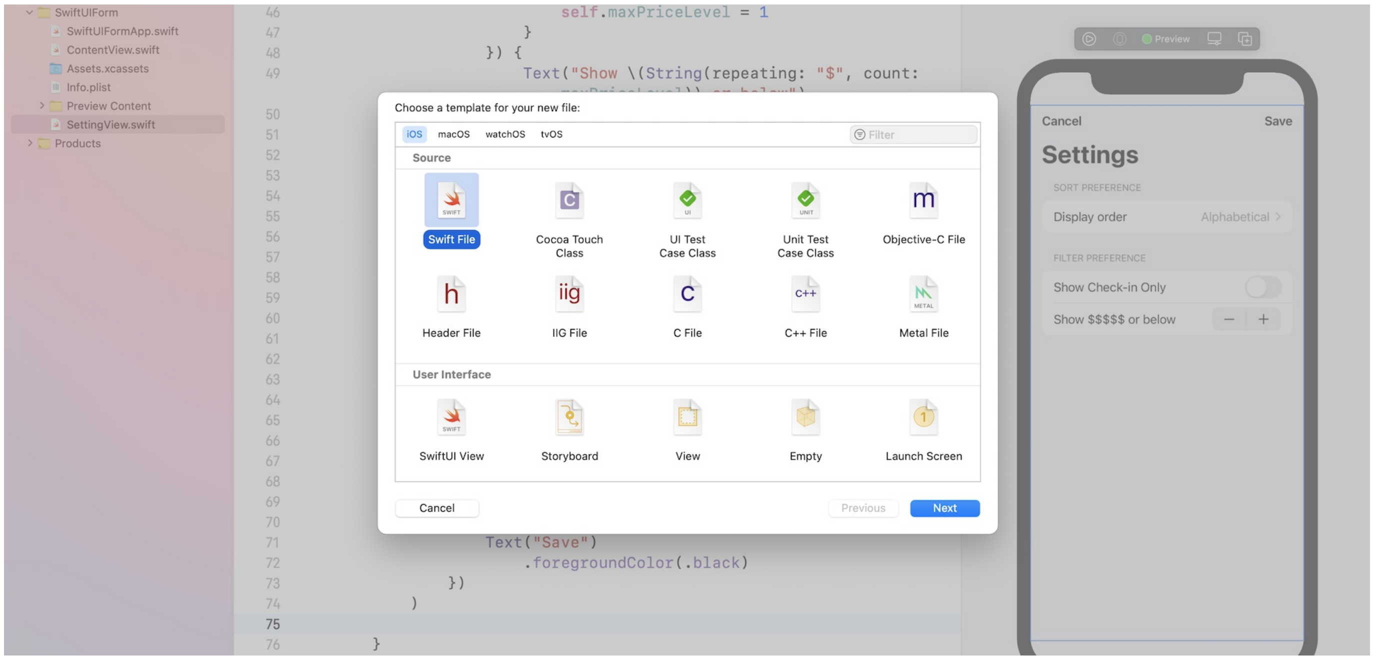
We currently use an array to store the three options of the display order. It works but there is a better way to improve the code.

> An enumeration defines a common type for a group of related values and enables you to work with those values in a type-safe way within your code.
>
> - Apple's official documentation (https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html)

Since this group of fixed values is related to display order, we can use an `Enum` to hold them and each case can be assigned with an integer value like this:

```
enum DisplayOrderType: Int, CaseIterable {
    case alphabetical = 0
    case favoriteFirst = 1
    case checkInFirst = 2

    init(type: Int) {
        switch type {
        case 0: self = .alphabetical
        case 1: self = .favoriteFirst
        case 2: self = .checkInFirst
        default: self = .alphabetical
        }
    }

    var text: String {
        switch self {
        case .alphabetical: return "Alphabetical"
        case .favoriteFirst: return "Show Favorite First"
        case .checkInFirst: return "Show Check-in First"
        }
    }
}
```

What makes `Enum` great is that we can work with these values in a type-safe way within our code. Additionally, `Enum` in Swift is a first-class type in its own right. That means you can create instance methods to provide additional functionality related to the values. Later, we will add a function for handling the filtering. Meanwhile, let's create a new Swift file named `SettingStore.swift` to store the `Enum`. You can right click `SwiftUIForm` in the project navigation and choose *New File...* to create the file.



```
                self.maxPriceLevel = 1
            }
        }) {
            Text("Show \(String(repeating: "$", count:
...
        Text("Save")
            .foregroundColor(.black)
    })
)
}
```

Display order: Alphabetical
Show Check-in Only
Show $$$$$ or below

Figure 1. Creating a new Swift file

After creating `SettingStore.swift`, insert the code snippet above in the file. Next, go back to `SettingView.swift`. We will update the code to use the `DisplayOrder` enumeration instead of the `displayOrders` array.

First, delete this line of code from `SettingView`:

```
private var displayOrders = [ "Alphabetical", "Show Favorite First", "Show Check-i
n First"]
```

Next, update the default value of `selectedOrder` to `DisplayOrderType.alphabetical` like this:

```
@State private var selectedOrder = DisplayOrderType.alphabetical
```

Here, we set the default display order to *alphabetical*. Comparing this to the previous value, of 0, the code is more readable after switching to use an enumeration. Next, you also need to change the code in the *Sort Preference* section. Specifically, we update the code in the `ForEach` loop:

```
Section(header: Text("SORT PREFERENCE")) {
    Picker(selection: $selectedOrder, label: Text("Display order")) {
        ForEach(DisplayOrderType.allCases, id: \.self) {
            orderType in
            Text(orderType.text)
        }
    }
}
```

Since we have adopted the `CaseIterable` protocol in the `DisplayOrder` enum, we can obtain all the display orders by accessing the `allCases` property, which contains an array of all the enum's cases.

Now you can test the *Settings* screen again. It should work and look the same. However, the underlying code is more manageable and readable.

Saving the User Preferences in UserDefaults

Right now, the app can't save the user's preference permanently. Whenever you restart the app, the Settings screen resets to its default settings.

There are multiple ways to store the settings. For saving small amounts of data like user settings on iOS, the built-in "defaults" database is a good option. This "defaults" system allows an app to store user's preferences in key-value pairs. To interact with this defaults database, you use a programmatic interface called `UserDefaults`.

In the `SettingStore.swift` file, we will create a `SettingStore` class to provide some convenience methods for saving and loading the user's preferences. Insert the following code snippet in `SettingStore.swift`:

```
final class SettingStore: ObservableObject {

    init() {
        UserDefaults.standard.register(defaults: [
            "view.preferences.showCheckInOnly" : false,
            "view.preferences.displayOrder" : 0,
            "view.preferences.maxPriceLevel" : 5
        ])
    }

    var showCheckInOnly: Bool = UserDefaults.standard.bool(forKey: "view.preferences.showCheckInOnly") {
        didSet {
            UserDefaults.standard.set(showCheckInOnly, forKey: "view.preferences.showCheckInOnly")
        }
    }

    var displayOrder: DisplayOrderType = DisplayOrderType(type: UserDefaults.standard.integer(forKey: "view.preferences.displayOrder")) {
        didSet {
            UserDefaults.standard.set(displayOrder.rawValue, forKey: "view.preferences.displayOrder")
        }
    }

    var maxPriceLevel: Int = UserDefaults.standard.integer(forKey: "view.preferences.maxPriceLevel") {
        didSet {
            UserDefaults.standard.set(maxPriceLevel, forKey: "view.preferences.maxPriceLevel")
        }
    }

}
```

Let me briefly explain the code. In the `init` method, we initialize the defaults system with some default values. These values will only be used if the user's preferences are not found in the database.

In the code above, we declare three properties (showCheckInOnly, displayOrder, and maxPriceLevel) that will be saved in key-value pairs with `UserDefaults`. The default value is loaded from the default system for the specific key. In the `didSet`, we use the `set` method of `UserDefaults` (`UserDefaults.standard.set()`) to save the value in the user default. All the three properties are marked with `@Published` so that they will notify all their subscribers when its value is updated.

With the `SettingStore` ready, let's switch over to the `SettingView.swift` file to implement the *Save* operation. First, declare a property in `SettingView` for the `SettingStore`:

```
var settingStore: SettingStore
```

For the *Save* button, find the *Save* button code (in the .navigationBarItems trailing block) and replace the existing code with this:

```
Button(action: {
    self.settingStore.showCheckInOnly = self.showCheckInOnly
    self.settingStore.displayOrder = self.selectedOrder
    self.settingStore.maxPriceLevel = self.maxPriceLevel
    self.presentationMode.wrappedValue.dismiss()

}, label: {
    Text("Save")
        .foregroundColor(.black)
})
```

We added three lines of code to the exiting save button to save the user's preference. To load the user's preferences when the *Settings* view is brought up, you can add a `onAppear` modifier to the `NavigationView` like this:

```
.onAppear {
    self.selectedOrder = self.settingStore.displayOrder
    self.showCheckInOnly = self.settingStore.showCheckInOnly
    self.maxPriceLevel = self.settingStore.maxPriceLevel
}
```

The `onAppear` modifier will be called when the view appears. We load the user's settings from the defaults system in its closure.

Before you can test the changes, you have to update `SettingView_Previews` like this:

```
struct SettingView_Previews: PreviewProvider {
    static var previews: some View {
        SettingView(settingStore: SettingStore())
    }
}
```

Now, switch over to `ContentView.swift` and declare the `settingStore` property:

```
var settingStore: SettingStore
```

And then update the `sheet` modifier like this:

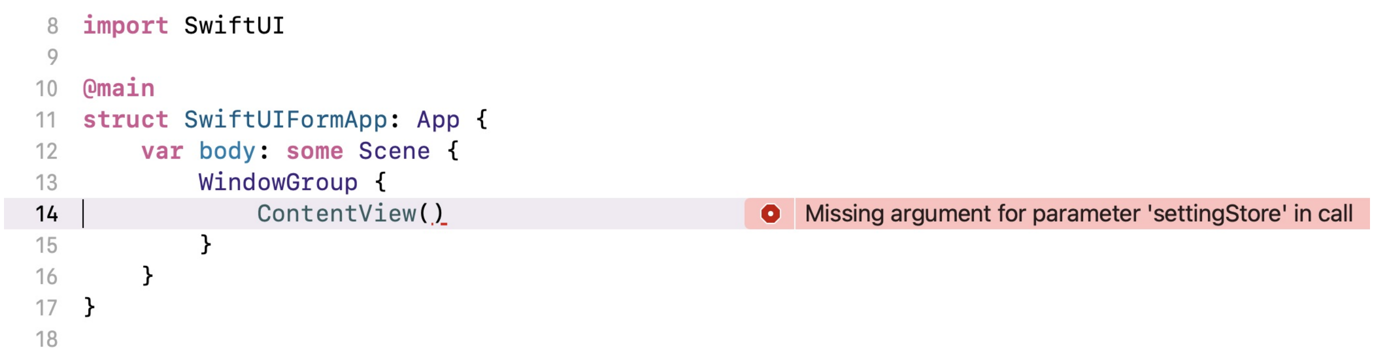
```
.sheet(isPresented: $showSettings) {
    SettingView(settingStore: self.settingStore)
}
```

Lastly, update `ContentView_Previews` like this:

```
struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView(settingStore: SettingStore())
    }
}
```

We initialize a `SettingStore` and pass it to `SettingView`. This is required because we've added the `settingStore` property in `SettingView`.

If you compile and run the app now, Xcode will show you an error. There is one more change we need to make before the app can run properly.



```
import SwiftUI

@main
struct SwiftUIFormApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

Missing argument for parameter 'settingStore' in call

Figure 2. An error in SwiftUIFormApp.swift

Go to `SwiftUIFormApp.swift` and add this property to create a `SettingStore` instance:

```
var settingStore = SettingStore()
```

Next, change the line code in the WindowGroup block to the following to fix the error:

```
ContentView(settingStore: settingStore)
```

You should now be able to execute app and play around with the settings. Once you save the settings, they are stored permanently in the local defaults system. You can stop the app and launch it again. The saved settings should be loaded in the Setting screen.

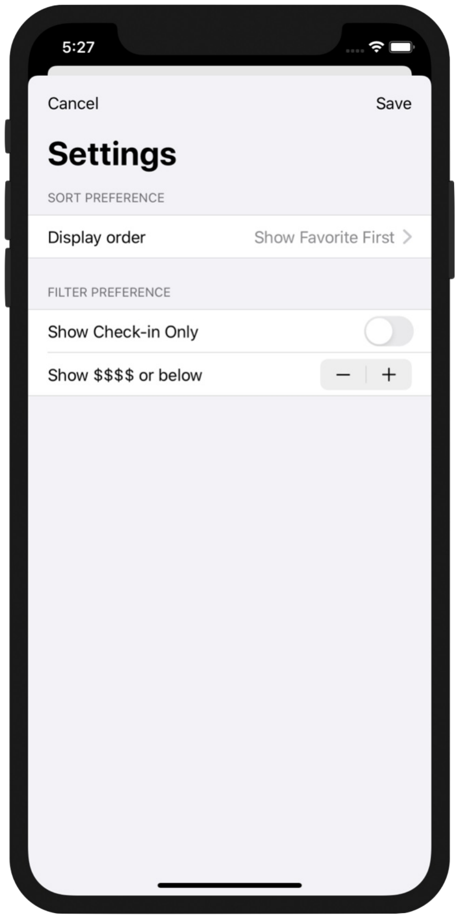


Figure 3. The Setting screen should load your user preference

Sharing Data Between Views Using @EnvironmentObject

Now that the user's preferences are saved in the local defaults system, the list view doesn't change in accordance to the user's settings. Again, there are various ways to solve this problem.

Let's recap what we have right now. When a user taps the *Save* button in the Setting screen, we save the selected options in the local defaults system. The *Settings* screen is then dismissed and the app will bring the user back to the list view. So, either we instruct the list view to reload the settings or the list view must be capable of monitoring the changes of the defaults system and trigger un update of the list.

Along with the introduction of SwiftUI, Apple also released a new framework called *Combine*. According to Apple, this framework provides a declarative API for processing values over time. In the context of this demo, Combine lets you easily monitor a single object and get notified of changes. Working along with SwiftUI, we can trigger an update of a view without writing a line of code. Everything is handled behind the scenes by SwiftUI and Combine.

So, how can the list view know the user's preference is modified and trigger the update itself?

Let me introduce three keywords:

1. **@EnvironmentObject** - Technically, this is known as a property wrapper, but you may consider this keyword as a special marker. When you declare a property as an environment object, SwiftUI monitors the value of the property and invalidates the corresponding view whenever there are changes. @EnvironmentObject works pretty much the same as @State. But when a property is declared as an environment object, it will be made accessible to all views in the entire app. For example, if your app has a lot of views that share the same piece of data (e.g. user settings), environment objects work great for this. You do not need to pass the property between views but instead you can access it automatically.
2. **ObservableObject** - this is a protocol of the Combine framework. When you declare a property as an environment object, the type of that property must implement this protocol. Back to our question: how can we let the list view know the user's preferences are changed? By implementing this protocol, the object can serve as a publisher that emits the changed value(s). The subscribers that monitor the value change will get notified.
3. **@Published** - is a property wrapper that works along with `ObservableObject`. When a property is prefixed with `@Publisher`, this indicates that the publisher should inform all subscribers whenever the property's value is changed.

I know it's a bit confusing. You will have a better understanding once we go through the code.

Let's start with `SettingStore.swift` . Since both the settings view and the list view need to monitor the change of user preferences, `SettingStore` should implement the `ObservableObject` protocol and announce the change of the `defaults` property. In the beginning of the `SettingStore.swift` file, we have to first import the Combine framework:

```
import Combine
```

The `SettingStore` class should adopt the `ObservableObject` protocol. Update the class declaration like this:

```
final class SettingStore: ObservableObject {
```

Next, insert the `@Published` annotation for all the properties like this:

```
@Published var showCheckInOnly: Bool = UserDefaults.standard.bool(forKey: "view.pr
eferences.showCheckInOnly") {
    didSet {
        UserDefaults.standard.set(showCheckInOnly, forKey: "view.preferences.showC
heckInOnly")
    }
}

@Published var displayOrder: DisplayOrderType = DisplayOrderType(type: UserDefaults
.standard.integer(forKey: "view.preferences.displayOrder")) {
    didSet {
        UserDefaults.standard.set(displayOrder.rawValue, forKey: "view.preferences
.displayOrder")
    }
}

@Published var maxPriceLevel: Int = UserDefaults.standard.integer(forKey: "view.pr
eferences.maxPriceLevel") {
    didSet {
        UserDefaults.standard.set(maxPriceLevel, forKey: "view.preferences.maxPric
eLevel")
    }
}
```

By using the `@Published` property wrapper, the publisher will let subscribers know whenever there is a value change of the property (e.g. an update of `displayOrder`).

As you can see, it's pretty easy to inform a changed value with Combine. Actually we haven't written any new code but simply adopted a required protocol and inserted a marker.

Now let's switch over to `SettingView.swift`. The `settingStore` should now declared as an environment object so that we share the data with other views. Update the `settingStore` variable like this:

```
@EnvironmentObject var settingStore: SettingStore
```

You do not need to update any code related to the *Save* button. However, when you set a new value for the setting store (e.g. update `showCheckInOnly` from true to false), this update will be published and let all subscribers know.

Because of the change, we need to update `SettingView_Previews` to the following:

```
struct SettingView_Previews: PreviewProvider {
    static var previews: some View {
        SettingView().environmentObject(SettingStore())
    }
}
```

Here, we inject an instance of `SettingStore` into the environment for the preview.

Okay, all our work has been on the *Publisher* side. What about the *Subscriber*? How can we monitor the change of `defaults` and update the UI accordingly?

In the demo project, the list view is the *Subscriber* side. It needs to monitor the changes of the setting store and re-render the list view to reflect the user's setting. Now let's open `ContentView.swift` to make some changes. Similar to what we've just done, the `settingStore` should now declared as an environment object:

```
@EnvironmentObject var settingStore: SettingStore
```

Due to the change, the code in the `sheet` modifier should be modified to grab this environment object:

```
.sheet(isPresented: $showSettings) {
    SettingView().environmentObject(self.settingStore)
}
```

Also, for testing purposes, the preview code should be updated accordingly to inject the environment object:

```
struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView().environmentObject(SettingStore())
    }
}
```

Lastly, open `SwiftUIFormApp.swift` and update the line of code inside `WindowGroup` like this:

```
struct SwiftUIFormApp: App {

    var settingStore = SettingStore()

    var body: some Scene {
        WindowGroup {
            ContentView().environmentObject(settingStore)
        }
    }
}
```

Here, we inject the setting store into the environment by calling the `environmentObject` method. Now the instance of setting store is available to all views within the app. In other words, both the *Setting* and *List* views can access it automatically.

Implementing the Filtering Options

Now we have implemented a common setting store that can be accessed by all views. What's great is that for any change in the setting store, it automatically notifies the views that monitor for updates. Though you don't experience any visual difference, the setting store does notify the changes to the list view when you update the options in the setting screen.

Our final task is to implement the filtering and sort options to display only the restaurants that match the user preferences. Let's start with the implementation of these two filtering options:

- Show check-in only
- Show restaurants below a certain price level

In `ContentView.swift`, we will create a new function called `showShowItem` to handle the filtering:

```
private func shouldShowItem(restaurant: Restaurant) -> Bool {
    return (!self.settingStore.showCheckInOnly || restaurant.isCheckIn) && (restau
rant.priceLevel <= self.settingStore.maxPriceLevel)
}
```

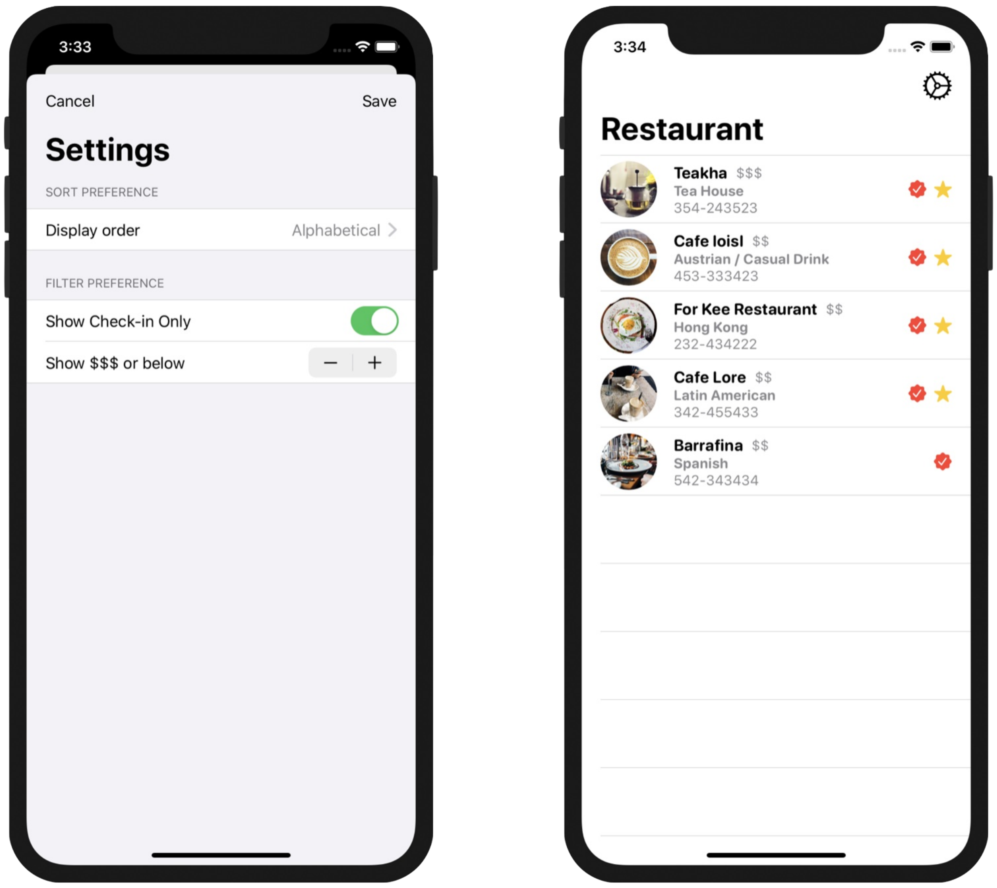
This function takes in a restaurant object and tells the caller if the restaurant should be displayed. In the code above, we check if the "Show Check-in Only" option is selected and verify the price level of the given restaurant.

Next, wrap the `BasicImageRow` with a `if` clause like this:

```
if self.shouldShowItem(restaurant: restaurant) {
        BasicImageRow(restaurant: restaurant)
            .contextMenu {

                 ...

            }
}
```

Here we first call the `shouldShowItem` function we just implemented to check if the restaurant should be displayed.

Now run the app and have a quick test. In the setting screen, set the *Show Check-in Only* option to ON and configure the price level option to show restaurants that are with price level 3 (i.e. $$$) or below. Once you tap the *Save* button, the list view should be automatically refreshed (with animation) and shows you the filtered records.


3:33

Cancel Save

Settings

SORT PREFERENCE

Display order Alphabetical

FILTER PREFERENCE

Show Check-in Only

Show $$$ or below

3:34

Restaurant

Teakha $$$
Tea House
354-243523

Cafe loisl $$
Austrian / Casual Drink
453-333423

For Kee Restaurant $$
Hong Kong
232-434222

Cafe Lore $$
Latin American
342-455433

Barrafina $$
Spanish
542-343434

Figure 4. The list view now refreshes its items when you change the filter preference

Implementing the Sort Option

Now that we've completed the implementation of the filtering options, let's work on the sort option. In Swift, you can sort a sequence of elements by using the `sort(by:)` method. When you use this method, you need to provide a predicate to it that returns `true` when the first element should be ordered before the second.

For example, to sort the `restaurants` array in alphabetical order. You can use the `sort(by:)` method like this:

```
restaurants.sorted(by: { $0.name < $1.name })
```

Here, $0 is the first element and $1 is the second element. In this case, a restaurant with the name "Upstate" is larger than a restaurant with the name "Homei". So, "Homei" will be put in front of "Upstate" in the sequence.

Conversely, if you want to sort the restaurants in alphabetical descending order, you can write the code like this:

```
restaurants.sorted(by: { $0.name > $1.name })
```

How can we sort the array to show "check-in" first or show "favorite" first? We can use the same method but provide a different predictate like this:

```
restaurants.sorted(by: { $0.isFavorite && !$1.isFavorite })
restaurants.sorted(by: { $0.isCheckIn && !$1.isCheckIn })
```

To better organize our code, we can put these predicates in the `DisplayOrderType` enum. In `SettingStore.swift`, add a new function in `DisplayOrderType` like this:

```
func predicate() -> ((Restaurant, Restaurant) -> Bool) {
    switch self {
    case .alphabetical: return { $0.name < $1.name }
    case .favoriteFirst: return { $0.isFavorite && !$1.isFavorite }
    case .checkInFirst: return { $0.isCheckIn && !$1.isCheckIn }
    }
}
```

This function simply returns the predicate, which is a closure, for the corresponding display order. Now we are ready to make the final change. Go back to `ContentView.swift` and change the `ForEach` statement from:

```
ForEach(restaurants) {
  ...
}
```

To:

```
ForEach(restaurants.sorted(by: self.settingStore.displayOrder.predicate())) {
  ...
}
```

That's it! Test the app and change the sort preference. When you update the sort option, the list view will get notified and re-orders the restaurants accordingly.

What's Coming Next

Are you aware that SwiftUI and Combine work together to help us write better code? In the last two sections of this chapter, we didn't write a lot of code to implement the filtering and sort options. Combine handles the heavy lifting of event processing. When pairing it with SwiftUI, it's even more powerful and saves you from developing your own implementation to monitor the state changes of objects and trigger UI updates. Everything is nearly automatic and taken care of by these two new frameworks.

In the next chapter, we will continue to explore Combine by building a registration screen. You will further understand how Combine can help you write cleaner and more modular code.

For reference, you can download the complete project here:

- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUIFormData.zip)

Chapter 15
Building a Registration Form with Combine and View Model

Now that you have some basic idea about Combine, let's explore how Combine can make SwiftUI really shine. When developing a real-world app, it's very common to have a user registration page for people to sign up and create an account. In this chapter, we will build a simple registration screen with three text fields. Our focus is on form validation, so we will not perform an actual sign up. You'll learn how we can leverage the power of Combine to validate each of the input fields and organize our code in a view model.

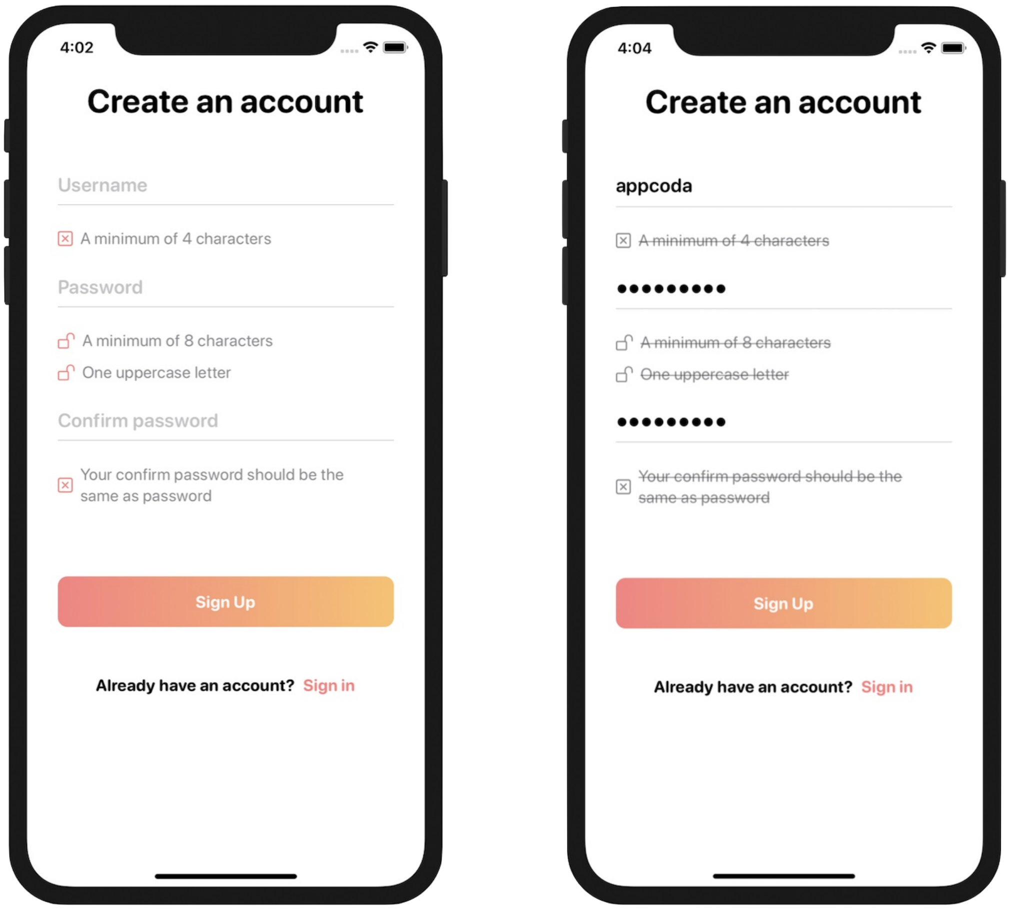

4:02
Create an account
Username
A minimum of 4 characters
Password
A minimum of 8 characters
One uppercase letter
Confirm password
Your confirm password should be the same as password
Sign Up
Already have an account? Sign in
4:04
Create an account
appcoda
~~A minimum of 4 characters~~
~~A minimum of 8 characters~~
~~One uppercase letter~~
~~Your confirm password should be the same as password~~
Sign Up
Already have an account? Sign in

Figure 1. User registration demo

Before we dive into the code, take a look at figure 1. That is the user registration screen we're going to build. Under each of the input fields, it lists out the requirements. As soon as the user fills in the information, the app validates the input in real-time and crosses out the requirement if it's been fulfilled. The sign up button is disabled until all the requirements are matched.

If you have experience in Swift and UIKit, you know there are various types of implementation to handle the form validation. In this chapter, however, we're going to explore how you can utilize the Combine framework to perform form validation.

Layout the Form using SwiftUI

Let's begin this chapter with an exercise, use what you've learned so far and layout the form UI shown in figure 1. To create a text field in SwiftUI, you can use the `TextField` component. For the password fields, SwiftUI provides a secure text field called `SecureField`.

To create a text field, you initiate a `TextField` with a field name and a binding. This renders an editable text field with the user's input stored in your given binding. Similar to other form fields, you can modify its look & feel by applying the associated modifiers. Here is a sample code snippet:

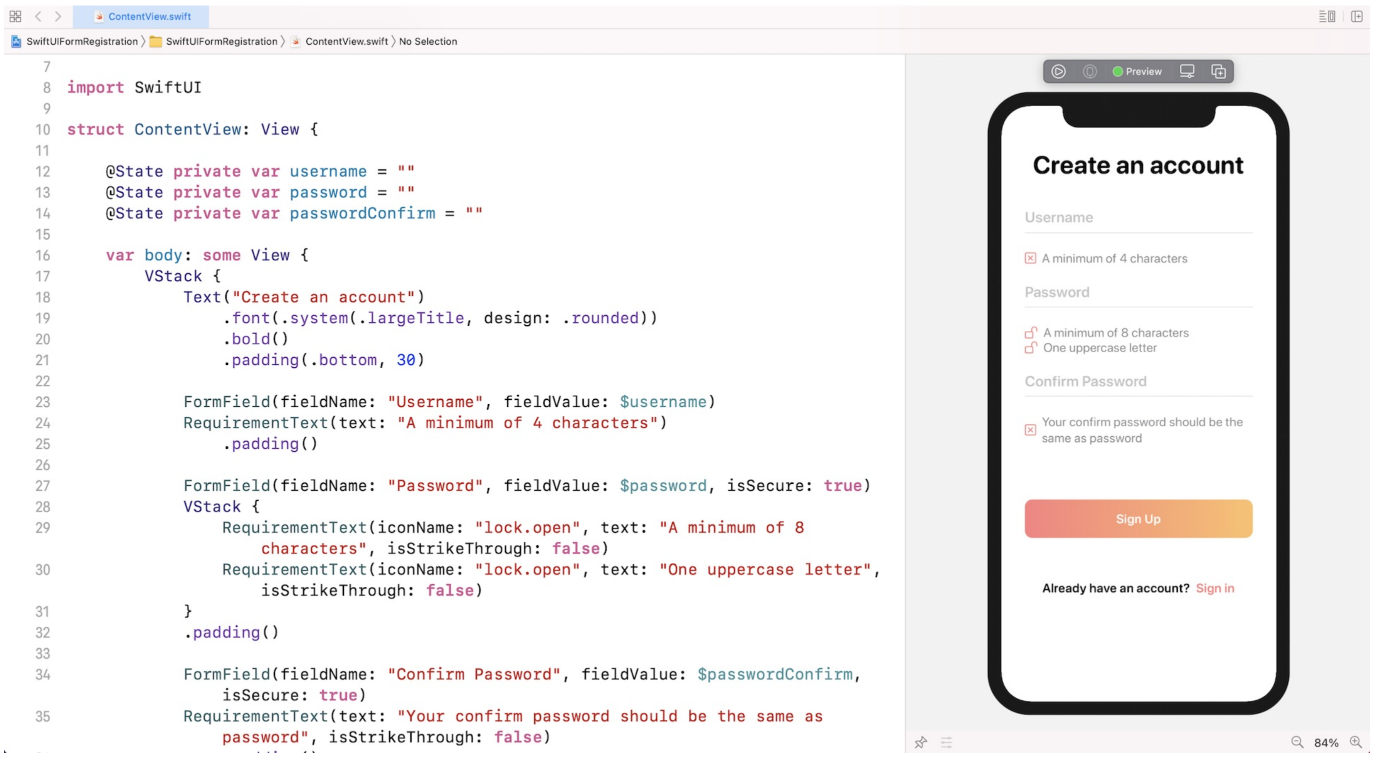
```
TextField("Username", text: $username)
    .font(.system(size: 20, weight: .semibold, design: .rounded))
    .padding(.horizontal)
```

The usage of these two components are very similar except that the secure field automatically masks the user's input:

```
SecureField("Password", text: $password)
    .font(.system(size: 20, weight: .semibold, design: .rounded))
    .padding(.horizontal)
```

I know these two components are new to you, but try your best to build the form before looking at the solution.

OWre you able to create the form? Even if you can't finish the exercise, that's completely fine. Download this project from https://www.appcoda.com/resources/swiftui2/SwiftUIFormRegistrationUI.zip. I will go through my solution with you.


ContentView.swift
SwiftUIFormRegistration 〉 SwiftUIFormRegistration 〉 ContentView.swift 〉 No Selection
Preview
Create an account
Username
A minimum of 4 characters
Password
A minimum of 8 characters
One uppercase letter
Confirm Password
Your confirm password should be the same as password
Sign Up
Already have an account? Sign in
84%

```swift
import SwiftUI

struct ContentView: View {

    @State private var username = ""
    @State private var password = ""
    @State private var passwordConfirm = ""

    var body: some View {
        VStack {
            Text("Create an account")
                .font(.system(.largeTitle, design: .rounded))
                .bold()
                .padding(.bottom, 30)

            FormField(fieldName: "Username", fieldValue: $username)
            RequirementText(text: "A minimum of 4 characters")
                .padding()

            FormField(fieldName: "Password", fieldValue: $password, isSecure: true)
            VStack {
                RequirementText(iconName: "lock.open", text: "A minimum of 8
                    characters", isStrikeThrough: false)
                RequirementText(iconName: "lock.open", text: "One uppercase letter",
                    isStrikeThrough: false)
            }
            .padding()

            FormField(fieldName: "Confirm Password", fieldValue: $passwordConfirm,
                isSecure: true)
            RequirementText(text: "Your confirm password should be the same as
                password", isStrikeThrough: false)
```

Figure 2. The starter project

Open the `ContentView.swift` file and preview the layout in the canvas. Your rendered view should look like that shown in figure 2. Now, let's briefly go over the code. Let's start with the `RequirementText` view.

```
struct RequirementText: View {

    var iconName = "xmark.square"
    var iconColor = Color(red: 251/255, green: 128/255, blue: 128/255)

    var text = ""
    var isStrikeThrough = false

    var body: some View {
        HStack {
            Image(systemName: iconName)
                .foregroundColor(iconColor)
            Text(text)
                .font(.system(.body, design: .rounded))
                .foregroundColor(.secondary)
                .strikethrough(isStrikeThrough)
            Spacer()
        }
    }
}
```

First, why do I create a separate view for the requirements text (see figure 3)? If you look at all of the requirements text, each requirement has an icon and a description. Instead of creating each of the requirements text from scratch, we can generalize the code and build a generic view for it.

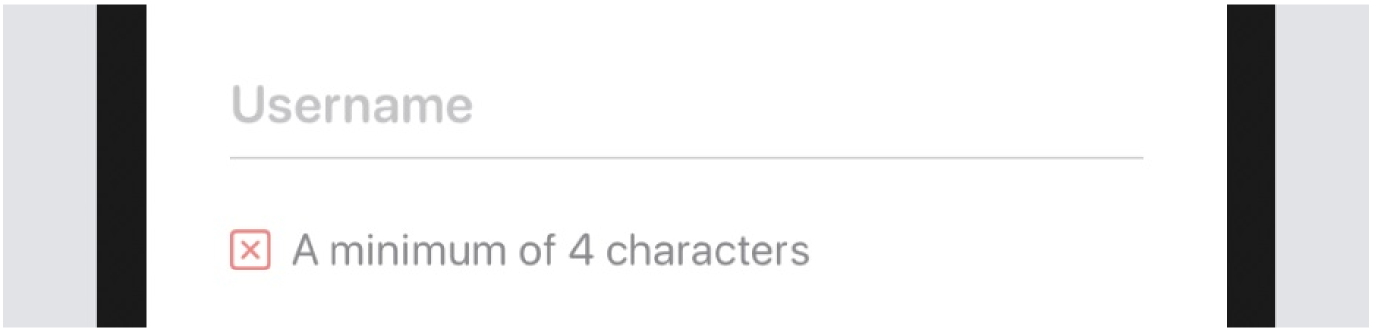


Figure 3. A sample text field and its requirement text

The `RequirementText` view has four properties including `iconName`, `iconColor`, `text`, and `isStrikeThrough`. It's flexible enough to support different styles of requirements text. If you accept the default icon and color, you can simply create a requirement text like this:

```
RequirementText(text: "A minimum of 4 characters")
```

This will render the square with an x in it (xmark.square) and the text as shown in figure 3. In some cases, the requirement text should be crossed out and display a different icon/color. The code can be written like this:

```
RequirementText(iconName: "lock.open", iconColor: Color.secondary, text: "A minimu
m of 8 characters", isStrikeThrough: true)
```

You specify a different system icon name, color, and set the `isStrikeThrough` option to `true`. This will allow you to create a requirement text like that displayed in figure 4.

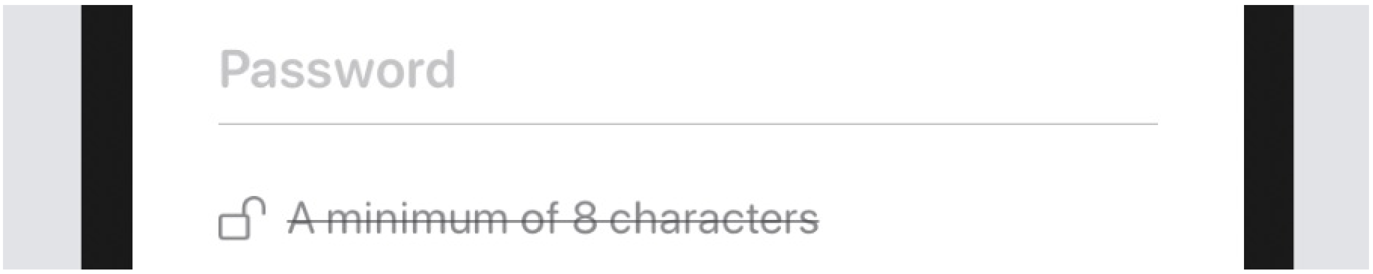


Figure 4. The requirement text is crossed out

Now that you understand how the `RequirementText` view works and why I created that, let's take a look at the `FormField` view. Again, if you look at all the text fields, they all have a common style - a text field with rounded font style. This is the reason why I extracted the common code and created a `FormField` view.

```
struct FormField: View {
    var fieldName = ""
    @Binding var fieldValue: String

    var isSecure = false

    var body: some View {

        VStack {
            if isSecure {
                SecureField(fieldName, text: $fieldValue)
                    .font(.system(size: 20, weight: .semibold, design: .rounded))
                    .padding(.horizontal)

            } else {
                TextField(fieldName, text: $fieldValue)
                    .font(.system(size: 20, weight: .semibold, design: .rounded))
                    .padding(.horizontal)
            }

            Divider()
                .frame(height: 1)
                .background(Color(red: 240/255, green: 240/255, blue: 240/255))
                .padding(.horizontal)

        }
    }
}
```

Since this generic `FormField` needs to take care of both text fields and secure fields, it has a property named `isSecure`. If it's set to `true`, the form field will be created as a secure field. In SwiftUI, you can make use of the `Divider` component to create a line. In the code, we use the `frame` modifier to change its height to 1 point.

To create the username field, you write the code like this:

```
FormField(fieldName: "Username", fieldValue: $username)
```

For the password field, the code is very similar except that the `isSecure` parameter is set to true:

```
FormField(fieldName: "Password", fieldValue: $password, isSecure: true)
```

Okay, let's head back to the `ContentView` struct and see how the form is laid out.

```
struct ContentView: View {

    @State private var username = ""
    @State private var password = ""
    @State private var passwordConfirm = ""

    var body: some View {
        VStack {
            Text("Create an account")
                .font(.system(.largeTitle, design: .rounded))
                .bold()
                .padding(.bottom, 30)

            FormField(fieldName: "Username", fieldValue: $username)
            RequirementText(text: "A minimum of 4 characters")
                .padding()

            FormField(fieldName: "Password", fieldValue: $password, isSecure: true
)
            VStack {
                RequirementText(iconName: "lock.open", iconColor: Color.secondary,
 text: "A minimum of 8 characters", isStrikeThrough: true)
                RequirementText(iconName: "lock.open", text: "One uppercase letter"
, isStrikeThrough: false)
            }
            .padding()

            FormField(fieldName: "Confirm Password", fieldValue: $passwordConfirm,
 isSecure: true)
            RequirementText(text: "Your confirm password should be the same as the
 password", isStrikeThrough: false)
                .padding()
                .padding(.bottom, 50)
            Button(action: {
                // Proceed to the next screen
            }) {
                Text("Sign Up")
                    .font(.system(.body, design: .rounded))
                    .foregroundColor(.white)
                    .bold()
                    .padding()
                    .frame(minWidth: 0, maxWidth: .infinity)
                    .background(LinearGradient(gradient: Gradient(colors: [Color(r
ed: 251/255, green: 128/255, blue: 128/255), Color(red: 253/255, green: 193/255, b
lue: 104/255)]), startPoint: .leading, endPoint: .trailing))
                    .cornerRadius(10)
                    .padding(.horizontal)

            }

            HStack {
                Text("Already have an account?")
                    .font(.system(.body, design: .rounded))
                    .bold()

                Button(action: {
                    // Proceed to Sign in screen
                }) {
                    Text("Sign in")
                        .font(.system(.body, design: .rounded))
                        .bold()
                        .foregroundColor(Color(red: 251/255, green: 128/255, blue:
128/255))
                }
            }.padding(.top, 50)

            Spacer()
        }
        .padding()
    }

}
```

First, we have a `VStack` to hold all the form elements. It begins with the heading, followed by all the form fields and requirement text. I have already explained how the form fields and requirement text are created, so I will not go through them again. What I added to the fields is the `padding` modifier. This is used to add some space between the text fields.

The *Sign up* button is created using the `Button` component and has an empty action. I intend to leave the action closure blank because our focus is on form validation. Again, I believe you should know how a button can be customized, so I will not go into it in detail. You can always refer to the Button chapter.

Last, is the description text *Already have an account*. This text and the *Sign in* button are completely optional. I'm mimicing the layout of a common sign up form.

That's how I laid out the user registration screen. If you tried out the exercise, you may have come up with a different solution. That's completely fine. Here I just wanted to show you one of the approaches to building the form. You can use it as a reference and come up with an even better implementation.

Understanding Combine

Before we dive into the code for form validation, it's better for me, first, to give you some more background information of the Combine framework. As mentioned in the previous chapter, this new framework provides a declarative API for processing values over time.

What does it mean by "processing values over time"? What are these values?

Let's use the registration form as an example. The app continues to generate UI events when it interacts with users. Each keystroke a user enters in the text field triggers an event. This becomes a stream of values as illustrated in figure 5.

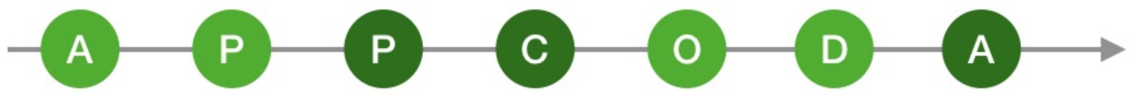


Figure 5. A stream of data input

These UI events are one type of "values" the framework refers to. Another example of these values is network events (e.g. downloading a file from a remote server).

> The Combine framework provides a declarative approach for how your app processes events. Rather than potentially implementing multiple delegate callbacks or completion handler closures, you can create a single processing chain for a given event source. Each part of the chain is a Combine operator that performs a distinct action on the elements received from the previous step.
>
> - Apple's official documentation (https://developer.apple.com/documentation/combine/receiving_and_handling_events_with_combine)

Publisher and *Subscriber* are the two core elements of the framework. With Combine, Publisher sends events and Subscriber subscribes to receive values from that Publisher. Again, let's use the text field as an example. By using Combine, each keystroke the user inputs in the text field triggers a value change event. The subscriber, which is interested in monitoring these values, can subscribe to receive these events and perform further operations (e.g. validation).

For example, you are writing a form validator which has a property to indicate if the form is ready to submit. In this case, you can mark that property with the `@Published` annotation like this:

```
class FormValidator: ObservableObject {
    @Published var isReadySubmit: Bool = false
}
```

Every time you change the value of `isReadySubmit`, it publishes an event to the subscriber. The subscriber receives the updated value and continues the processing. Let's say, the subscriber uses that value to determine if the submit button should be enabled or not.

You may think `@Published` works pretty much like `@State` in SwiftUI. While it works pretty much the same for this example, `@State` only applies to properties that belong to a specific SwiftUI view. If you want to create a custom type that doesn't belong to a specific view or that can be used among multiple views, you need to create a class that conforms to `ObservableObject` and mark those properties with the `@Published` annotation.

Combine and MVVM

Now that you have a basic concept of Combine, let's begin to implement the form validation using the framework. Here is what we are going to do:

1. Create a view model to represent the user registration form
2. Implement form validation in the view model

I know you may have a few questions in mind. First, why do we need to create a view model? Can't we add the properties of the form and perform the form validation in the ContentView?

Absolutely, you can do that. But as your project grows or the view becomes more complex, it's a good practice to break a complex component into multiple layers.

"Separation of concerns" is a fundamental principle of writing good software. Instead of putting everything in a single view, we can separate a view into two components: the view and its view model. The view itself is responsible for the UI layout, while the view model holds the states and data to be displayed in the view. The view model also handles the data validation and conversion. For experienced developers, we are applying a well known design pattern called MVVM (short for Model-View-ViewModel).

So, what data will this view model hold?

Take a look at the registration form again. We have three text fields including:

- Username
- Password
- Password confirm

On top of that, this view model will hold the states of the requirements text, indicating whether they should be crossed out or not:

- A minimum of 4 characters (username)
- A minimum of 8 characters (password)
- One uppercase letter (password)
- Your confirm password should the same as the password (password confirm)

Therefore, the view model will have seven properties and each of these properties publishes its value change to those which are interested in receiving the value. The basic skeleton of the view model can be defined like this:

```
class UserRegistrationViewModel: ObservableObject {
    // Input
    @Published var username = ""
    @Published var password = ""
    @Published var passwordConfirm = ""

    // Output
    @Published var isUsernameLengthValid = false
    @Published var isPasswordLengthValid = false
    @Published var isPasswordCapitalLetter = false
    @Published var isPasswordConfirmValid = false
}
```

That's the data model for the form view. The `username`, `password`, and `passwordConfirm` properties hold the value of the username, password, and password confirm text fields respectively. This class should conform to `ObservableObject`. All these properties are annotated with `@Published` because we want to notify the subscribers whenever there is a value change and perform the validation accordingly.

Validating the Username with Combine

Okay, that's the data model. But we still haven't dealt with the form validation. How do we validate the username, password, and passwordConfirm in accordance to the requirements?

With Combine, you have to develop a publisher/subscriber mindset to answer the question. Consider the username, we actually have two publishers here: *username* and *isUsernameLengthValid*. The `username` publisher emits value changes whenever the user enters in a keystroke in the username field. The `isUsernameLengthValid` publisher informs the subscriber about the validation status of the user input. Nearly all controls in SwiftUI are subscribers, so the requirements text view will listen to the change of validation result and update its style (i.e. strikethrough or not) accordingly. Figure 6 illustrates how we use Combine to validate the username input.

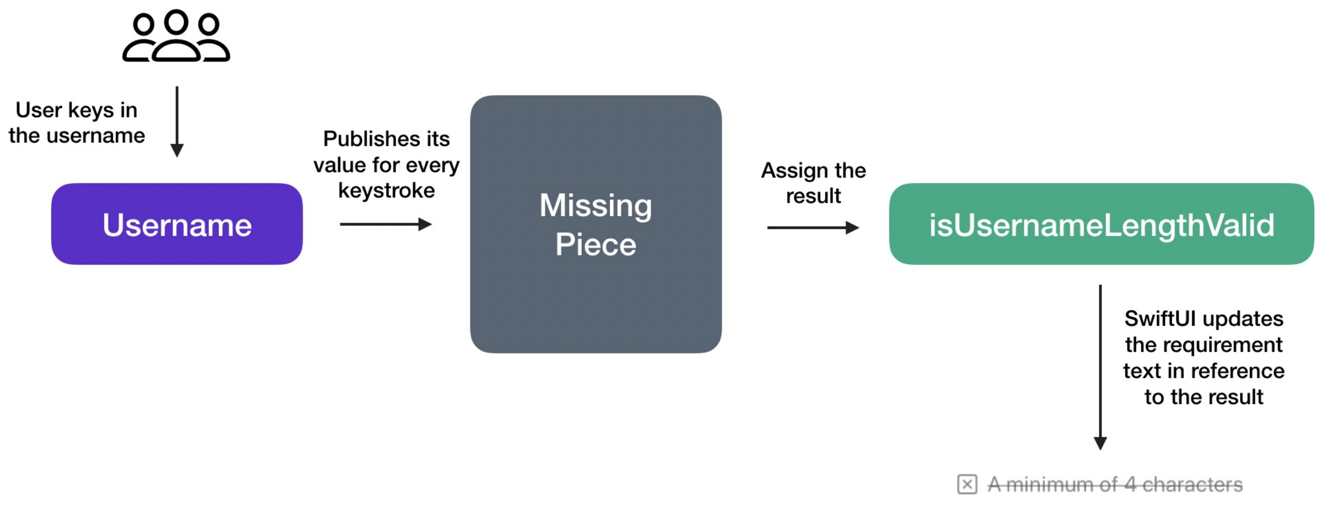

User keys in the username
Username
Publishes its value for every keystroke
Missing Piece
Assign the result
isUsernameLengthValid
SwiftUI updates the requirement text in reference to the result
~~A minimum of 4 characters~~

Figure 6. The username and isUsernameValid publishers

What's missing here is something that connects between these two publishers. And, this "something" should handle the following tasks:

- Listen to the `username` change
- Validate the username and return the validation result (true/false)
- Assign the result to `isUsernameLengthValid`

If you transform the above requirements into code, here is what the code snippet looks like:

```
$username
    .receive(on: RunLoop.main)
    .map { username in
        return username.count >= 4
    }
    .assign(to: \.isUsernameLengthValid, on: self)
```

The Combine framework provides two built-in subscribers: *sink* and *assign*. For `sink`, it creates a general purpose subscriber to receive values. `assign` allows you to create another type of subscriber that can update a specific property of an object. For example, it assigns the validation result (true/false) to `isUsernameLengthValid` directly.

Let me dive deeper through the code above line by line. `$username` is the source of value change that we want to listen to. Since we're subscribing to the change of UI events, we call the `receive(on:)` function to ensure the subscriber receives values on the main thread (i.e. `RunLoop.main`).

The value sent by the publisher is the username input by the user. But what the subscriber is interested in is whether the length of the username meets the minimum requirement. Here, the `map` function is an operator in Combine that takes an input, processes it, and transforms the input into something that the subscriber expects. So, what we did in the code above is:

1. We take the username as input.
2. Then we validate the username and verify if it has at least 4 characters.
3. Lastly, we return the validation result as a boolean (true/false) to the subscriber.

With the validation result, the subscriber simply sets the result to the `isUsernameLengthValid` property. Recall that `isUsernameLengthValid` is also a publisher, we can then update the `RequirementText` control like this to subscribe to the change and update the UI accordingly:

```
RequirementText(iconColor: userRegistrationViewModel.isUsernameLengthValid ? Color
.secondary : Color(red: 251/255, green: 128/255, blue: 128/255), text: "A minimum
of 4 characters", isStrikeThrough: userRegistrationViewModel.isUsernameLengthValid
)
```

Both the icon color and the status of strike through depend on the validation result (i.e. `isUsernameLengthValid`).

This is how we use Combine to validate a form field. We still haven't put the code change into our project, but I want you to understand the concept of publisher/subscriber and how we perform validation using this approach. In later section, we will apply what we learned and make the code change.

Validate the Passwords with Combine

Now that you understand how the validation of the username field is done, we will apply a similar implementation for the password and password confirm validation.

The password field has two requirements:

1. The length of password should have at least 8 characters.
2. It should contain at least one uppercase letter.

To meet these requirements, we set up two subscribers like this:

```
$password
    .receive(on: RunLoop.main)
    .map { password in
        return password.count >= 8
    }
    .assign(to: \.isPasswordLengthValid, on: self)

$password
    .receive(on: RunLoop.main)
    .map { password in
        let pattern = "[A-Z]"
        if let _ = password.range(of: pattern, options: .regularExpression) {
            return true
        } else {
            return false
        }
    }
    .assign(to: \.isPasswordCapitalLetter, on: self)
```

The first subscriber subscribes the verification result of password length and assigns it to the `isPasswordLengthValid` property. The second subscriber hands the validation of the uppercase letter. We use the `range` method to test if the password has at least one uppercase letter. Again, the subscriber assigns the validation result the `isPasswordCapitalLetter` property directly.

Okay, what's left is the validation of the password confirm field. For this field, the input requirement is that the password confirm should be equal to that of the password field. Both `password` and `passwordConfirm` are publishers. To verify if both publishers have the same value, we use `Publisher.combineLatest` to receive and combine the latest values from the publishers. We can then verify if the two values are the same. Here is the code snippet:

```
Publishers.CombineLatest($password, $passwordConfirm)
    .receive(on: RunLoop.main)
    .map { (password, passwordConfirm) in
        return !passwordConfirm.isEmpty && (passwordConfirm == password)
    }
    .assign(to: \.isPasswordConfirmValid, on: self)
```

Similarly, we assign the validation result to the `isPasswordConfirmValid` property.

Implementing the UserRegistrationViewModel

Now that I've explained the implementation, let's put everything together into the project. First, create a new Swift file named `UserRegistrationViewModel.swift` using the *Swift File* template. Replace the whole file's content with the following code:

```
import Foundation
import Combine

class UserRegistrationViewModel: ObservableObject {
    // Input
    @Published var username = ""
    @Published var password = ""
    @Published var passwordConfirm = ""

    // Output
    @Published var isUsernameLengthValid = false
    @Published var isPasswordLengthValid = false
    @Published var isPasswordCapitalLetter = false
    @Published var isPasswordConfirmValid = false

    private var cancellableSet: Set<AnyCancellable> = []

    init() {
        $username
            .receive(on: RunLoop.main)
            .map { username in
                return username.count >= 4
            }
            .assign(to: \.isUsernameLengthValid, on: self)
            .store(in: &cancellableSet)

        $password
            .receive(on: RunLoop.main)
            .map { password in
                return password.count >= 8
            }
            .assign(to: \.isPasswordLengthValid, on: self)
            .store(in: &cancellableSet)

        $password
            .receive(on: RunLoop.main)
            .map { password in
                let pattern = "[A-Z]"
                if let _ = password.range(of: pattern, options: .regularExpression
) {
                    return true
                } else {
                    return false
                }
            }
            .assign(to: \.isPasswordCapitalLetter, on: self)
            .store(in: &cancellableSet)

        Publishers.CombineLatest($password, $passwordConfirm)
            .receive(on: RunLoop.main)
            .map { (password, passwordConfirm) in
                return !passwordConfirm.isEmpty && (passwordConfirm == password)
            }
            .assign(to: \.isPasswordConfirmValid, on: self)
            .store(in: &cancellableSet)
    }
}
```

The code is nearly the same as what we went through in the earlier sections. To use Combine, you first need to import the *Combine* framework. In the `init()` method, we initialize all the subscribers to listen to the value change of the text fields and perform the corresponding validations.

The code is nearly the same as the code snippets we discussed earlier. One thing you may notice is the `cancellableSet` variable. Additionally, for each of the subscribers, we call the `store` function at the very end.

What does the `store` function and `cancellableSet` variable do?

The `assign` function, which creates the subscriber, returns you with a cancellable instance. You can use this instance to cancel the subscription at the appropriate time. The `store` function lets us save the cancellable reference into a set for later cleanup. If you do not store the reference, the app may end up with memory leak issues.

So, when will the clean up happen for this demo? Because `cancellableSet` is defined as a property of the class, the cleanup and cancellation of the subscription will happen when the class is deinitialized.

Now switch back to `ContentView.swift` and update the UI controls. First, replace the following state variables:

```
@State private var username = ""
@State private var password = ""
@State private var passwordConfirm = ""
```

with a view model and name it `userRegistrationViewModel`:

```
@ObservedObject private var userRegistrationViewModel = UserRegistrationViewModel(
)
```

Next, update the text field and the requirement text of username like this:

```
FormField(fieldName: "Username", fieldValue: $userRegistrationViewModel.username)
RequirementText(iconColor: userRegistrationViewModel.isUsernameLengthValid ? Color
.secondary : Color(red: 251/255, green: 128/255, blue: 128/255), text: "A minimum
of 4 characters", isStrikeThrough: userRegistrationViewModel.isUsernameLengthValid
)
    .padding()
```

The `fieldValue` parameter is now changed to `$userRegistrationViewModel.username`. For the requirement text, SwiftUI monitors the `userRegistrationViewModel.isUsernameLengthValid` property and updates the requirement text accordingly.

Similarly, update the UI code for the password and password confirm fields like this:

```
FormField(fieldName: "Password", fieldValue: $userRegistrationViewModel.password,
isSecure: true)
VStack {
    RequirementText(iconName: "lock.open", iconColor: userRegistrationViewModel.is
PasswordLengthValid ? Color.secondary : Color(red: 251/255, green: 128/255, blue:
128/255), text: "A minimum of 8 characters", isStrikeThrough: userRegistrationView
Model.isPasswordLengthValid)
    RequirementText(iconName: "lock.open", iconColor: userRegistrationViewModel.is
PasswordCapitalLetter ? Color.secondary : Color(red: 251/255, green: 128/255, blue
: 128/255), text: "One uppercase letter", isStrikeThrough: userRegistrationViewMod
el.isPasswordCapitalLetter)
}
.padding()

FormField(fieldName: "Confirm Password", fieldValue: $userRegistrationViewModel.pa
sswordConfirm, isSecure: true)
RequirementText(iconColor: userRegistrationViewModel.isPasswordConfirmValid ? Color
.secondary : Color(red: 251/255, green: 128/255, blue: 128/255), text: "Your confi
rm password should be the same as password", isStrikeThrough: userRegistrationView
Model.isPasswordConfirmValid)
    .padding()
    .padding(.bottom, 50)
```

That's it! You're now ready to test the app. If you've made all the changes correctly, the app should now validate the user input.

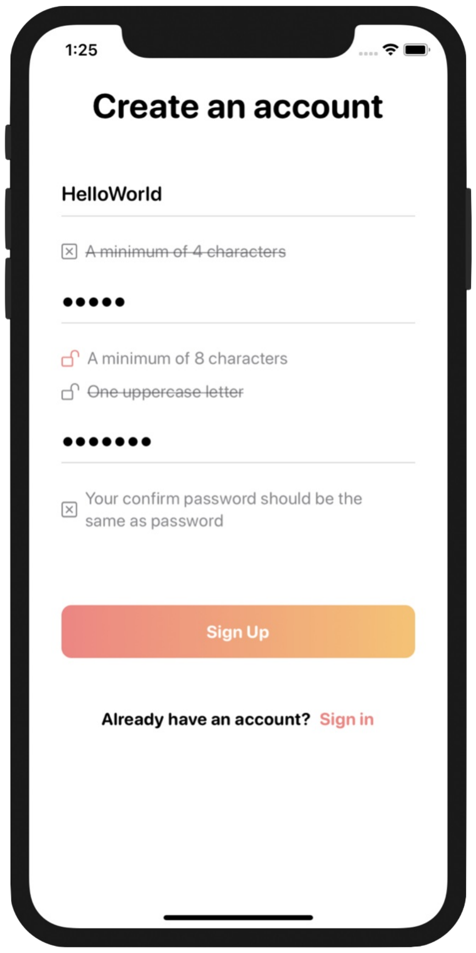

1:25
Create an account
HelloWorld
~~A minimum of 4 characters~~
●●●●●
A minimum of 8 characters
~~One uppercase letter~~
●●●●●●●
Your confirm password should be the same as password
Sign Up
Already have an account? Sign in

Figure 7. The registration form now validates the user input

Summary

I hope you now have gained some basic knowledge of the Combine framework. The introduction of SwiftUI and Combine completely change the way you build apps. Functional Reactive Programming (FRP) has become more and more popular in recent years, This is the first time Apple has released their own functional reactive framework. To me, it's a major paradigm shift. The company finally took position on FRP and recommends Apple developers embrace this new programming methodology.

Like the introduction of any new technology, there will be a learning curve. Even if you've been programming in iOS, it will take some time to move from the programming methodology of delegates to publishers and subscribers.

However, once you get comfortable with the Combine framework, you will be very glad as it will help you achieve more maintainable and modular code. As you can now see, together with SwiftUI, communication between a view and a view model is a breeze.

For reference, you can download the complete form validation project here:

- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUIFormRegistration.zip)

Chapter 16
Working with Swipe-to-Delete, Context Menu and Action Sheets

Previously, you learned how to present rows of data using list. In this chapter, we will dive a little bit deeper and see how to let users interact with the list view including:

- Use swipe to delete a row
- Tap a row to invoke an action sheet
- Touch and hold a row to bring up a context menu

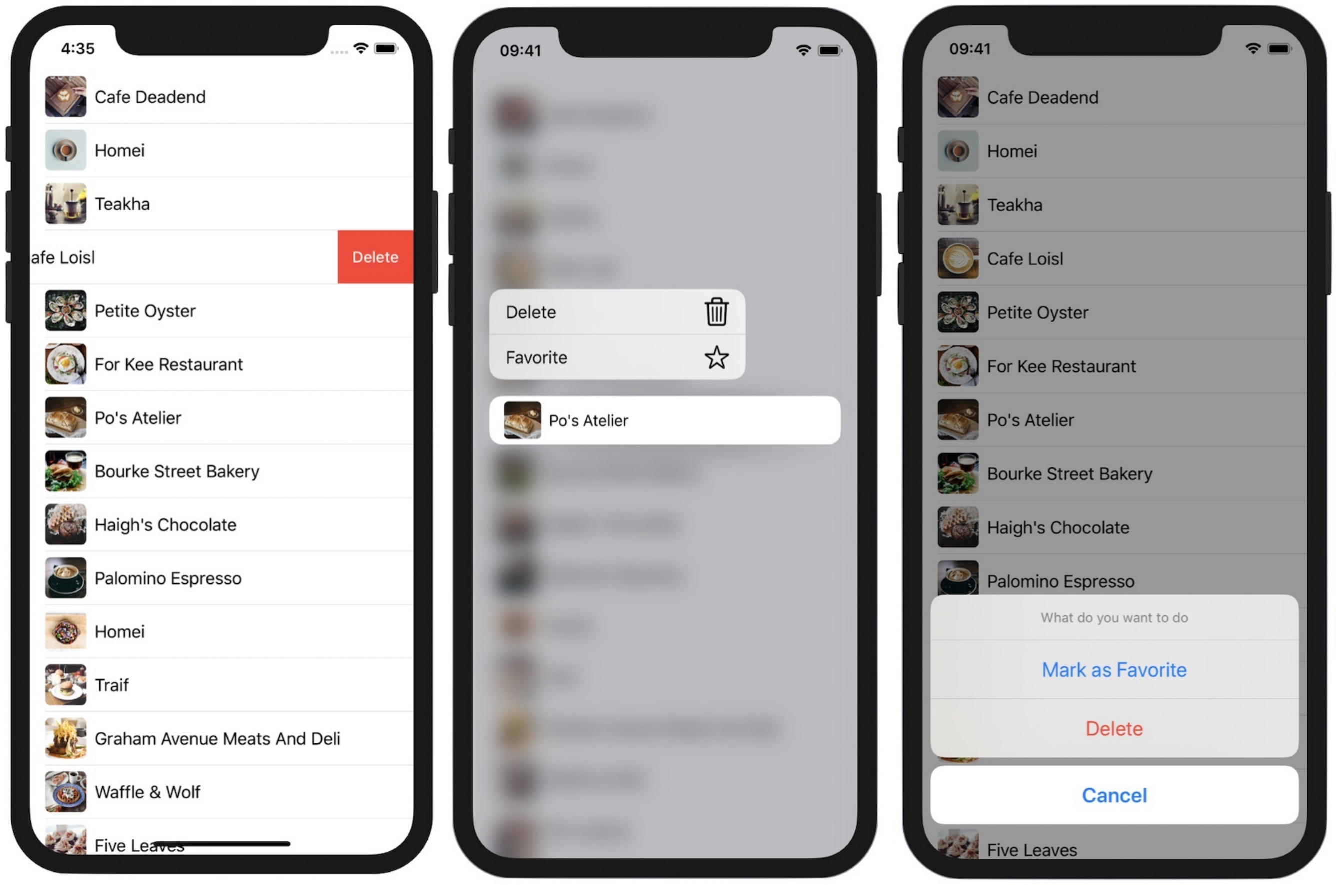


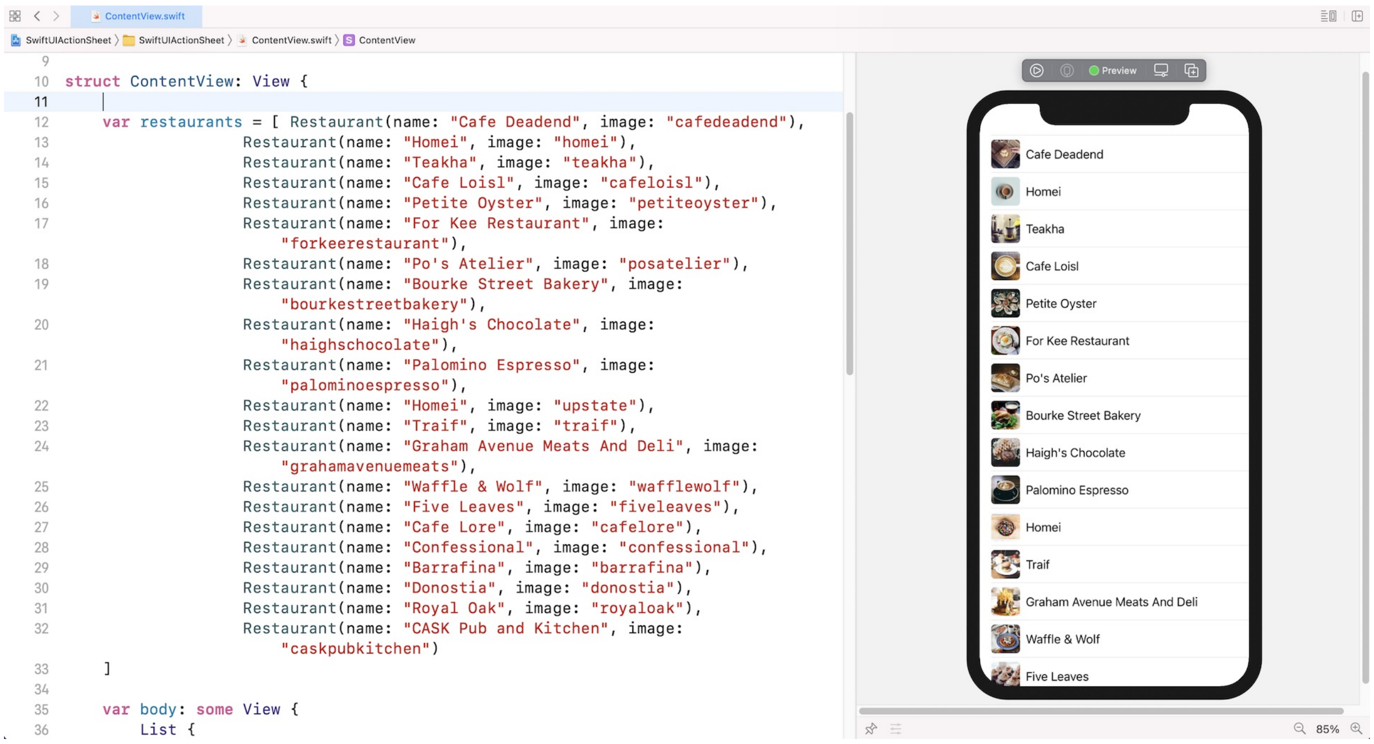
Figure 1. Swipe to delete (left), context menu, and action sheet (right)

Referring to figure 1, I believe you should be very familiar with swipe-to-delete and action sheet. These two UI elements have existed in iOS for several years. Context menus are recently introduced in iOS 13, though they look similar to peek and pop of 3D Touch. For any views (e.g. button) implemented with the context menu, iOS will bring up a popover menu whenever a user force touches on the view. For developers, it's your responsibility to configure the action items displayed in the menu.

While this chapter focuses on the interaction of a list, the techniques that I'm going to show you can also be applied to other UI controls such as buttons.

Preparing the Starter Project

Let's get started and create the demo. We will build an interactive list based on the restaurant list app. You can download the starter project from https://www.appcoda.com/resources/swiftui2/SwiftUIActionSheetStarter.zip. Once downloaded, open the project and check out the preview. It should display a simple list with text and images. Later, we will add the swipe-to-delete feature, an action sheet, and a context menu to this demo app.



```
struct ContentView: View {

    var restaurants = [ Restaurant(name: "Cafe Deadend", image: "cafedeadend"),
                       Restaurant(name: "Homei", image: "homei"),
                       Restaurant(name: "Teakha", image: "teakha"),
                       Restaurant(name: "Cafe Loisl", image: "cafeloisl"),
                       Restaurant(name: "Petite Oyster", image: "petiteoyster"),
                       Restaurant(name: "For Kee Restaurant", image:
                           "forkeerestaurant"),
                       Restaurant(name: "Po's Atelier", image: "posatelier"),
                       Restaurant(name: "Bourke Street Bakery", image:
                           "bourkestreetbakery"),
                       Restaurant(name: "Haigh's Chocolate", image:
                           "haighschocolate"),
                       Restaurant(name: "Palomino Espresso", image:
                           "palominoespresso"),
                       Restaurant(name: "Homei", image: "upstate"),
                       Restaurant(name: "Traif", image: "traif"),
                       Restaurant(name: "Graham Avenue Meats And Deli", image:
                           "grahamavenuemeats"),
                       Restaurant(name: "Waffle & Wolf", image: "wafflewolf"),
                       Restaurant(name: "Five Leaves", image: "fiveleaves"),
                       Restaurant(name: "Cafe Lore", image: "cafelore"),
                       Restaurant(name: "Confessional", image: "confessional"),
                       Restaurant(name: "Barrafina", image: "barrafina"),
                       Restaurant(name: "Donostia", image: "donostia"),
                       Restaurant(name: "Royal Oak", image: "royaloak"),
                       Restaurant(name: "CASK Pub and Kitchen", image:
                           "caskpubkitchen")

    ]

    var body: some View {
        List {
```

Figure 2. The starter project should display a simple list view

If you have a sharp eye, you may spot that the starter project used `ForEach` to implement the list. Why did I change it back to `ForEach` instead of passing the collection of data to `List`? The main reason is that the `onDelete` handler that I'm going to walk you through *only* works with `ForEach`.

Implementing Swipe-to-delete

Assuming you have the starter project ready, let's begin implementing the swipe-to-delete feature. I've briefly mentioned the `onDelete` handler. To activate swipe-to-delete for all rows in a list, you just need to attach this handler to all the row data. So, update the `List` like this:

```
List {
    ForEach(restaurants) { restaurant in
        BasicImageRow(restaurant: restaurant)
    }
    .onDelete { (indexSet) in
        self.restaurants.remove(atOffsets: indexSet)
    }
}
```

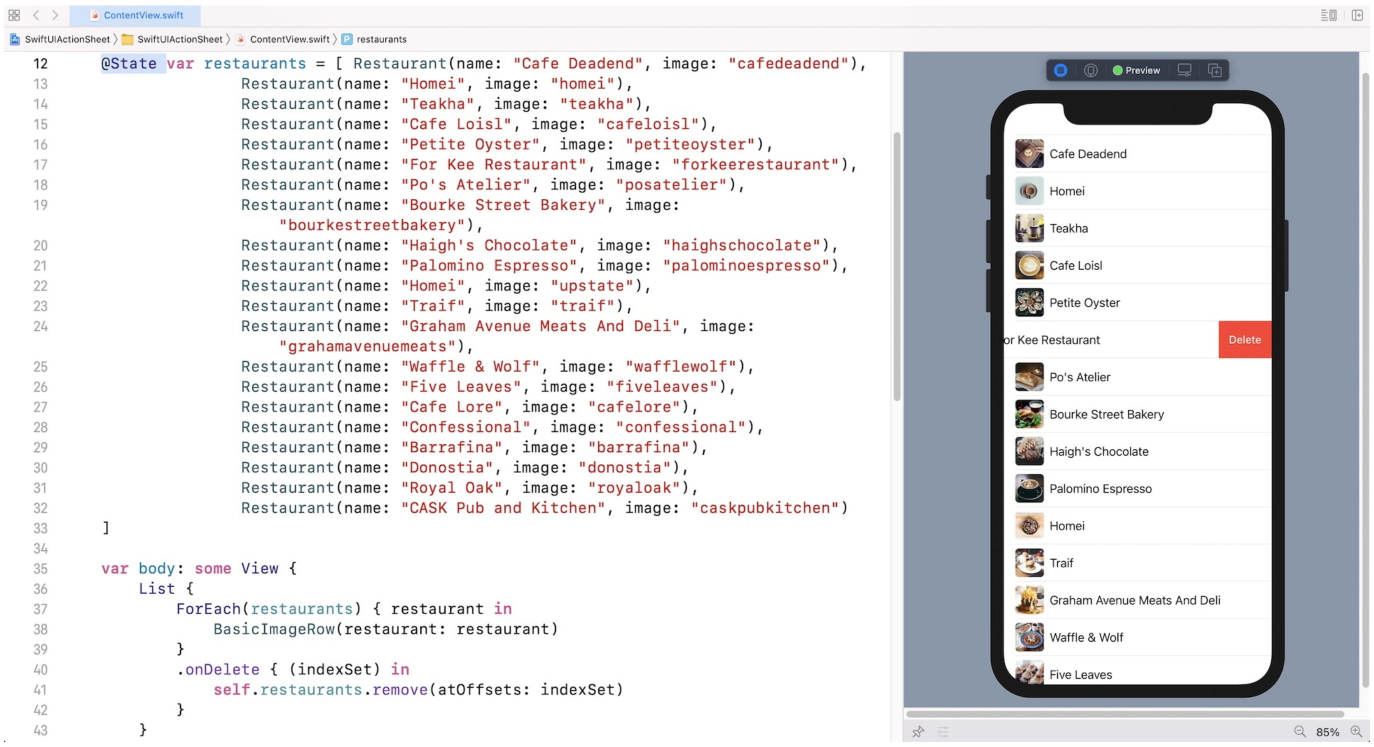
In the closure of `onDelete`, we pass an `indexSet` storing the index of the rows to be deleted. We then call the `remove` method with the `indexSet` to delete the specific items in the `restaurants` array.

There is still one thing left before the swipe-to-delete feature works. Whenever a user removes a row from the list, the UI should be updated accordingly. As discussed in earlier chapters, SwiftUI has come with a very powerful feature to manage the application's state. In our code, the value of the `restaurants` array will be changed when a user chooses to delete a record. We have to ask SwiftUI to monitor the property and update the UI whenever the value of the property changes.

To do that, insert the `@State` keyword to the `restaurants` variable:

```
@State var restaurants = [ ... ]
```

Once you have made the change, run the app (click the *Play* button) in the canvas. Swipe any of the rows to the left to reveal the *Delete* button. Tap it and that row will be removed from the list. By the way, do you notice the nice animation while the row is being removed? You don't need to write any extra code. This animation is automatically generated by SwiftUI. Cool, right?



```
@State var restaurants = [ Restaurant(name: "Cafe Deadend", image: "cafedeadend"),
                Restaurant(name: "Homei", image: "homei"),
                Restaurant(name: "Teakha", image: "teakha"),
                Restaurant(name: "Cafe Loisl", image: "cafeloisl"),
                Restaurant(name: "Petite Oyster", image: "petiteoyster"),
                Restaurant(name: "For Kee Restaurant", image: "forkeerestaurant"),
                Restaurant(name: "Po's Atelier", image: "posatelier"),
                Restaurant(name: "Bourke Street Bakery", image:
                    "bourkestreetbakery"),
                Restaurant(name: "Haigh's Chocolate", image: "haighschocolate"),
                Restaurant(name: "Palomino Espresso", image: "palominoespresso"),
                Restaurant(name: "Homei", image: "upstate"),
                Restaurant(name: "Traif", image: "traif"),
                Restaurant(name: "Graham Avenue Meats And Deli", image:
                    "grahamavenuemeats"),
                Restaurant(name: "Waffle & Wolf", image: "wafflewolf"),
                Restaurant(name: "Five Leaves", image: "fiveleaves"),
                Restaurant(name: "Cafe Lore", image: "cafelore"),
                Restaurant(name: "Confessional", image: "confessional"),
                Restaurant(name: "Barrafina", image: "barrafina"),
                Restaurant(name: "Donostia", image: "donostia"),
                Restaurant(name: "Royal Oak", image: "royaloak"),
                Restaurant(name: "CASK Pub and Kitchen", image: "caskpubkitchen")
]

var body: some View {
    List {
        ForEach(restaurants) { restaurant in
            BasicImageRow(restaurant: restaurant)
        }
        .onDelete { (indexSet) in
            self.restaurants.remove(atOffsets: indexSet)
        }
    }
```

Figure 3. Deleting an item from the list

If you've written the same feature using UIKit, I'm sure you are amazed by SwiftUI. With just a few lines of code and a keyword, you implemented the swipe-to-delete feature.

Creating a Context Menu

Next, let's talk about context menus, a new feature which was introduced in iOS 13. As said, a context menu is similar to peek and pop in 3D Touch. One noticeable difference is that this feature works on all devices running iOS 13 and later, even if the device doesn't support 3D Touch. To bring up a context menu, you use the touch and hold gesture or force touch if the device is powered with 3D Touch.

SwiftUI has made it very simple to implement a context menu. All you do is attach the `contextMenu` container to the view and configure its menu items.

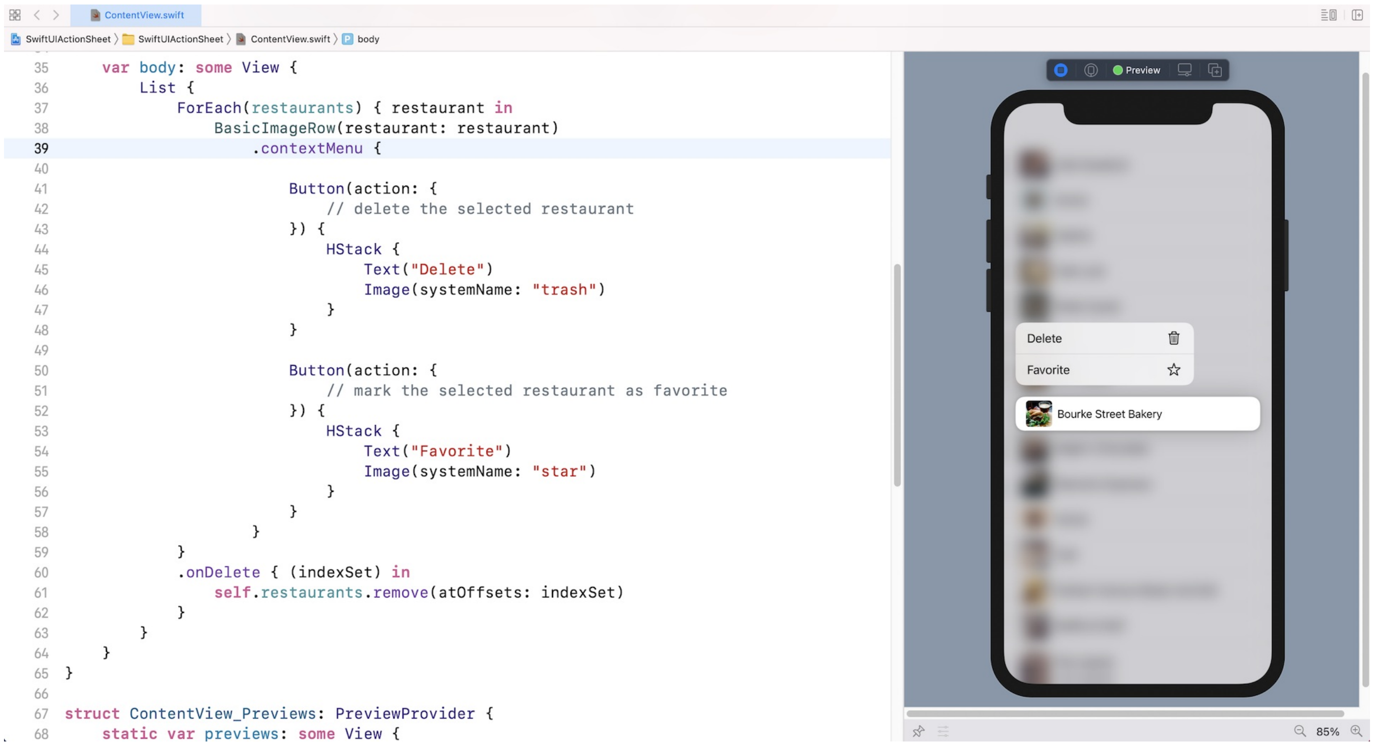
For our demo app, we want to trigger the context menu when people touch and hold any of the rows. The menu provides two action buttons for users to choose: *Delete* and *Favorite*. When selected, the *Delete* button will remove the row from the list. The

Favorite button will mark the selected row with a star indicator.

To present these two items in the context menu, we attach the `contextMenu` to each of the rows in the list like this:

```
List {
    ForEach(restaurants) { restaurant in
        BasicImageRow(restaurant: restaurant)
            .contextMenu {

                Button(action: {
                    // delete the selected restaurant
                }) {
                    HStack {
                        Text("Delete")
                        Image(systemName: "trash")
                    }
                }

                Button(action: {
                    // mark the selected restaurant as favorite
                }) {
                    HStack {
                        Text("Favorite")
                        Image(systemName: "star")
                    }
                }
            }
    }
    .onDelete { (indexSet) in
        self.restaurants.remove(atOffsets: indexSet)
    }
}
```

We haven't implemented any of the button actions yet. However, if you execute the app, the app will bring up the context menu when you touch and hold one of the rows.


ContentView.swift
SwiftUIActionSheet 〉 SwiftUIActionSheet 〉 ContentView.swift 〉 body
Preview
Delete
Favorite
Bourke Street Bakery
85%

```
    var body: some View {
        List {
            ForEach(restaurants) { restaurant in
                BasicImageRow(restaurant: restaurant)
                    .contextMenu {

                        Button(action: {
                            // delete the selected restaurant
                        }) {
                            HStack {
                                Text("Delete")
                                Image(systemName: "trash")
                            }
                        }

                        Button(action: {
                            // mark the selected restaurant as favorite
                        }) {
                            HStack {
                                Text("Favorite")
                                Image(systemName: "star")
                            }
                        }
                    }
            }
            .onDelete { (indexSet) in
                self.restaurants.remove(atOffsets: indexSet)
            }
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
```

Figure 4. Deleting an item from the list

Let's continue by implementing the delete action. Unlike the `onDelete` handler, the `contextMenu` doesn't give us the index of the selected restaurant. To figure it out, it would require a little bit of work. Create a new function in `ContentView`:

```
private func delete(item restaurant: Restaurant) {
    if let index = self.restaurants.firstIndex(where: { $0.id == restaurant.id })
{
        self.restaurants.remove(at: index)
    }
}
```

This `delete` function takes in a restaurant object and searches for its index in the `restaurants` array. To find the index, we call the `firstIndex` function and specify the search criteria. The function loops through the array and compares the id of the given restaurant with those in the array. If there is a match, the `firstIndex` function returns the index of the given restaurant. Once we have the index, we can remove the restaurant from the `restaurants` array by calling `remove(at:)`.

Next, insert the following line of code under `// delete the selected restaurant` :

```
self.delete(item: restaurant)
```

We simply call the `delete` function when the user selects the *Delete* button. Now you're ready to test the app. Click the *Play* button in the canvas to run the app. Press and hold one of the rows to bring up the context menu. Choose *Delete* and you should see your selected restaurant removed from the list.

Let's move onto the implementation of the *Favorite* button. When this button is selected, the app will place a star in the selected restaurant's row. To implement this feature, we first need to modify the `Restaurant` struct and add a new property named `isFavorite` like this:

```
struct Restaurant: Identifiable {
    var id = UUID()
    var name: String
    var image: String
    var isFavorite: Bool = false
}
```

This `isFavorite` property indicates whether the restaurant is marked as a favorite. By default, it's set to `false` .

Similar to the *Delete* feature, we'll create a separate function in `ContentView` for setting a favorite restaurant. Insert the following code to create the new function:

```
private func setFavorite(item restaurant: Restaurant) {
    if let index = self.restaurants.firstIndex(where: { $0.id == restaurant.id })
{
        self.restaurants[index].isFavorite.toggle()
    }
}
```

The code is very similar to that of the `delete` function. We first find out the index of the given restaurant. Once we have the index, we change the value of its `isFavorite` property. Here we invoke the `toggle` function to toggle the value. For example, if the original value of `isFavorite` is set to `false`, the value will change to `true` after calling `toggle()`.

Next, we have to handle the UI for the row. Whenever the restaurant's `isFavorite` property is set to `true`, the row should present a star indicator. Update the `BasicImageRow` struct like this:

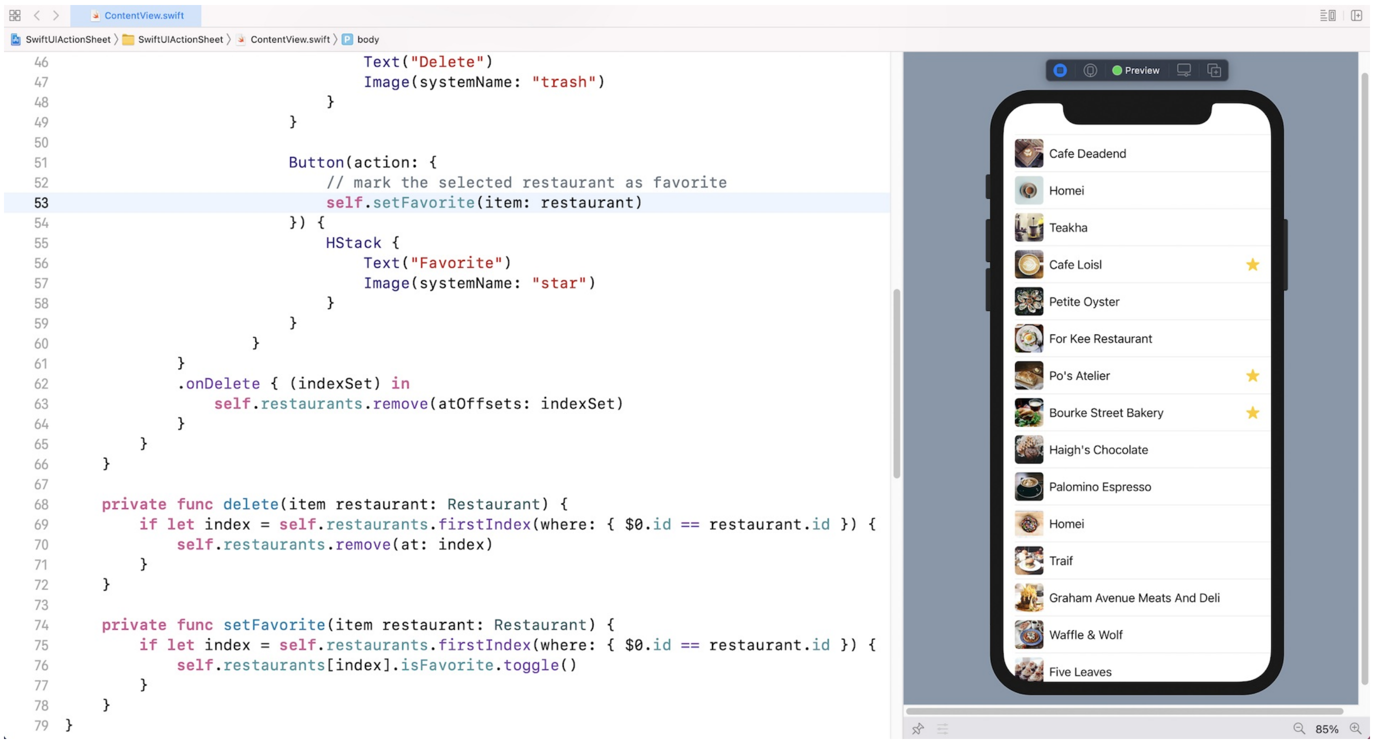
```
struct BasicImageRow: View {
    var restaurant: Restaurant

    var body: some View {
        HStack {
            Image(restaurant.image)
                .resizable()
                .frame(width: 40, height: 40)
                .cornerRadius(5)
            Text(restaurant.name)

            if restaurant.isFavorite {
                Spacer()

                Image(systemName: "star.fill")
                    .foregroundColor(.yellow)
            }
        }
    }
}
```

In the code above, we just add a code snippet in the `HStack`. If the `isFavorite` property of the given restaurant is set to `true`, we add a spacer and a system image to the row.

That's how we implement the *Favorite* feature. Lastly, insert the following line of code under `// mark the selected restaurant as favorite` to invoke the `setFavorite` function:

```
self.setFavorite(item: restaurant)
```

Now it's time to test. Execute the app in the canvas. Press and hold one of the rows (e.g. Petite Oyster), and then choose *Favorite*. You should see a star app appeared at the end of the row.



```
                            Text("Delete")
                            Image(systemName: "trash")
                        }
                    }

                    Button(action: {
                        // mark the selected restaurant as favorite
                        self.setFavorite(item: restaurant)
                    }) {
                        HStack {
                            Text("Favorite")
                            Image(systemName: "star")
                        }
                    }
                }
            }
            .onDelete { (indexSet) in
                self.restaurants.remove(atOffsets: indexSet)
            }
        }
    }

    private func delete(item restaurant: Restaurant) {
        if let index = self.restaurants.firstIndex(where: { $0.id == restaurant.id }) {
            self.restaurants.remove(at: index)
        }
    }

    private func setFavorite(item restaurant: Restaurant) {
        if let index = self.restaurants.firstIndex(where: { $0.id == restaurant.id }) {
            self.restaurants[index].isFavorite.toggle()
        }
    }
}
```

Figure 5. Using the Favorite function

Working with Action Sheets

That is how you implement context menus. Lastly, let's see how to create an action sheet in SwiftUI. The action sheet, that we are going to build, provides the same options as the context menu. If you forgot what the action sheet looks like, please refer to figure 1 again.

The SwiftUI framework comes with an `ActionSheet` view for you to create an action sheet. Basically, you can create an action sheet like this:

```
ActionSheet(title: Text("What do you want to do"), message: nil, buttons: [.default
(Text("Delete"))]
```

You initialize an action sheet with a title and an option message. The `buttons` parameter accepts an array of buttons. In the sample code above, it provides a default button titled *Delete*.

To activate an action sheet, you attach the `actionSheet` modifier to a button or any view. If you look into SwiftUI's documentation, you have two ways to bring up an action sheet.

You can control the appearance of an action sheet by using the `isPresented` parameter:

```
func actionSheet(isPresented: Binding<Bool>, content: () -> ActionSheet) -> some V
iew
```

Or through an optional binding:

```
func actionSheet<T>(item: Binding<T?>, content: (T) -> ActionSheet) -> some View w
here T : Identifiable
```

We will use both approaches to present the action sheet, so you'll understand when to use which approach.

For the first approach, we need a Boolean variable to represent the status of the action and also a variable of the type `Restaurant` to store the selected restaurant. So, declare these two variables in `ContentView` :

```
@State private var showActionSheet = false

@State private var selectedRestaurant: Restaurant?
```

By default, the `showActionSheet` variable is set to `false` , meaning that the action sheet is not shown. We will toggle this variable to `true` when a user selects a row. The `selectedRestaurant` variable, as its name suggests, is designed to hold the chosen restaurant. Both variables have the `@State` keyword because we want SwiftUI to monitor their changes and update the UI accordingly.

Next, add the `onTapGesture` and `actionSheet` modifiers to the `List` view like this:

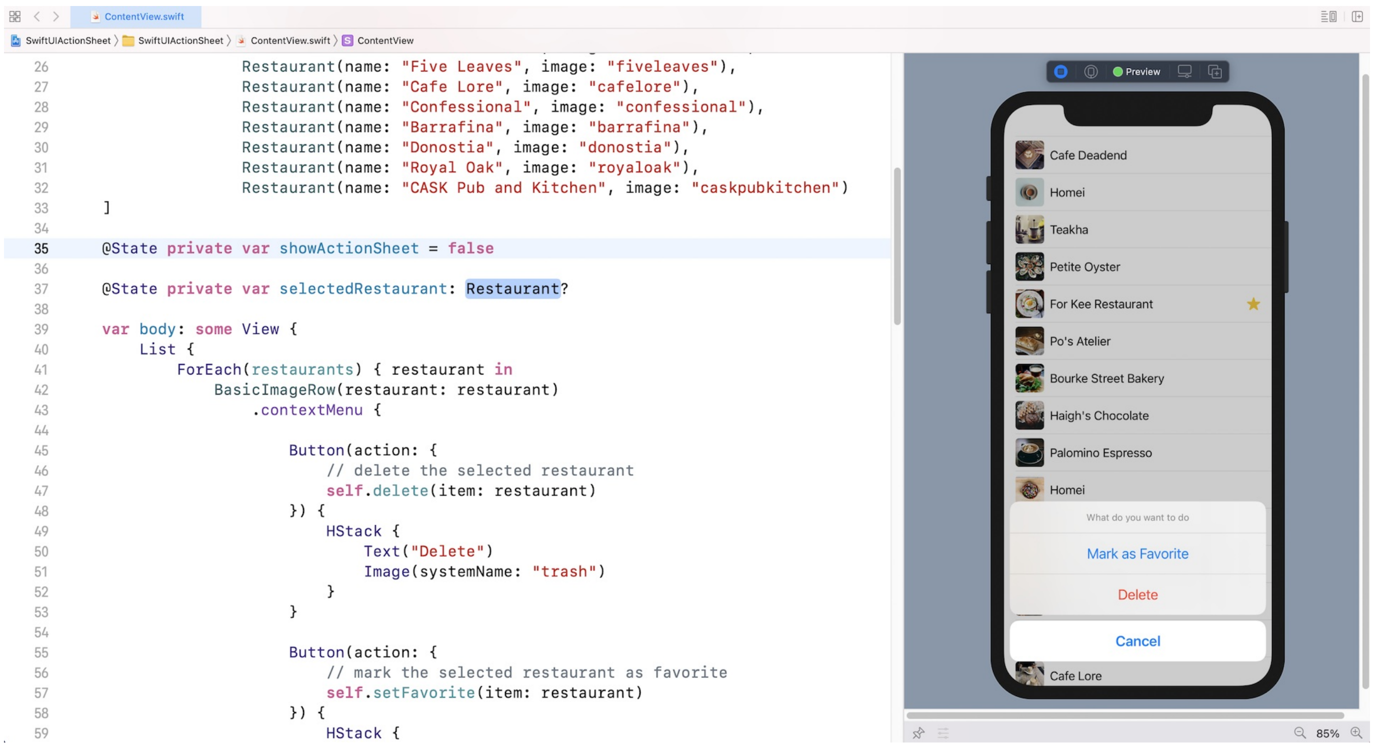
```
List {
    ForEach(restaurants) { restaurant in
        BasicImageRow(restaurant: restaurant)
            .contextMenu {

                ...
            }
            .onTapGesture {
                self.showActionSheet.toggle()
                self.selectedRestaurant = restaurant
            }
            .actionSheet(isPresented: self.$showActionSheet) {

                ActionSheet(title: Text("What do you want to do"), message: nil, b
uttons: [

                    .default(Text("Mark as Favorite"), action: {
                        if let selectedRestaurant = self.selectedRestaurant {
                            self.setFavorite(item: selectedRestaurant)
                        }
                    }),

                    .destructive(Text("Delete"), action: {
                        if let selectedRestaurant = self.selectedRestaurant {
                            self.delete(item: selectedRestaurant)
                        }
                    }),

                    .cancel()
                ])
            }
    }
    .onDelete { (indexSet) in
        self.restaurants.remove(atOffsets: indexSet)
    }
}
```

The `onTapGesture` modifier, attached to each row, is used to detect users' touch. When a row is tapped, the block of code in `onTapGesture` will be run. Here, we toggle the `showActionSheet` variable and set the `selectedRestaurant`.

Earlier, I explained the usage of the `actionSheet` modifier. In the code above, we pass the `isPresented` parameter with the binding of `showActionSheet`. When `showActionSheet` is set to `true`, the block of code will be executed. We initiate an `ActionSheet` with three buttons: *Mark as Favorite*, *Delete*, and *Cancel*. Action sheet comes with three types of buttons including default, destructive, and cancel. You usually use the *default* button type for ordinary actions. A destructive button is very similar to a default button but the font color is set to red to indicate destructive actions such as delete. The *cancel* button is a special type for dismissing the action sheet.

The *Mark as Favorite* button, is our default button. In the `action` closure, we call the `setFavorite` function to add the *star*. For the destructive button we used *Delete*. Similar to the *Delete* button of the context menu, we call the `delete` function to remove the selected restaurant.

If you've made the changes correctly, you should be able to bring up the action sheet when you tap one of the rows in the list view. Selecting the *Delete* button will remove the row. If you choose the *Mark as Favorite* option, you will mark the row with a yellow star.



```
                    Restaurant(name: "Five Leaves", image: "fiveleaves"),
                    Restaurant(name: "Cafe Lore", image: "cafelore"),
                    Restaurant(name: "Confessional", image: "confessional"),
                    Restaurant(name: "Barrafina", image: "barrafina"),
                    Restaurant(name: "Donostia", image: "donostia"),
                    Restaurant(name: "Royal Oak", image: "royaloak"),
                    Restaurant(name: "CASK Pub and Kitchen", image: "caskpubkitchen")
    ]

    @State private var showActionSheet = false

    @State private var selectedRestaurant: Restaurant?

    var body: some View {
        List {
            ForEach(restaurants) { restaurant in
                BasicImageRow(restaurant: restaurant)
                    .contextMenu {

                        Button(action: {
                            // delete the selected restaurant
                            self.delete(item: restaurant)
                        }) {
                            HStack {
                                Text("Delete")
                                Image(systemName: "trash")
                            }
                        }

                        Button(action: {
                            // mark the selected restaurant as favorite
                            self.setFavorite(item: restaurant)
                        }) {
                            HStack {
```

Figure 6. Tapping a row to reveal the action sheet

Everything works great, but I promised you to walk you through the second approach of using the `actionSheet` modifier. The first approach, which we have covered, relies on a Boolean value (i.e. `showActionSheet`) to indicate whether the action sheet should be displayed.

The second approach triggers the action sheet through an optional *Identifiable* binding:

```
func actionSheet<T>(item: Binding<T?>, content: (T) -> ActionSheet) -> some View w
here T : Identifiable
```

In plain English, this means the action sheet will be shown when the item you pass has a value. For our case, the `selectedRestaurant` variable is an optional that conforms to the `Identifiable` protocol. To use the second approach, you just need to pass the `selectedRestaurant` binding to the `actionSheet` modifier like this:

```
.actionSheet(item: self.$selectedRestaurant) { restaurant in

    ActionSheet(title: Text("What do you want to do"), message: nil, buttons: [

        .default(Text("Mark as Favorite"), action: {
            self.setFavorite(item: restaurant)
        }),

        .destructive(Text("Delete"), action: {
            self.delete(item: restaurant)
        }),

        .cancel()
    ])
}
```

If the `selectedRestaurant` has a value, the app will bring up the action sheet. From the closure's parameter, you can retrieve the selected restaurant and perform the operations accordingly.

When you use this approach, you no longer need the boolean variable `shownActionSheet`. You can remove it from the code:

```
@State private var showActionSheet = false
```

Also, in the `tapGesture` modifier, remove the line of the code that toggles the `showActionSheet` variable:

```
self.showActionSheet.toggle()
```

Test the app again. The action sheet looks still the same, but you implemented the action sheet with a different approach.

Exercise

Now that you have some idea how to build a context menu, let's have an exercise to test your knowledge. Your task is to add a *Check-in* item in the context menu. When a user selects the option, the app will add a check-in indicator in the selected restaurant. You can refer to figure 7 for the sample UI. For the sample, I used the system image named `checkmark.seal.fill` for the check-in indicator. However, you are free to choose your own image.

Please take some time to work on the exercise before checking out the solution. Have fun!

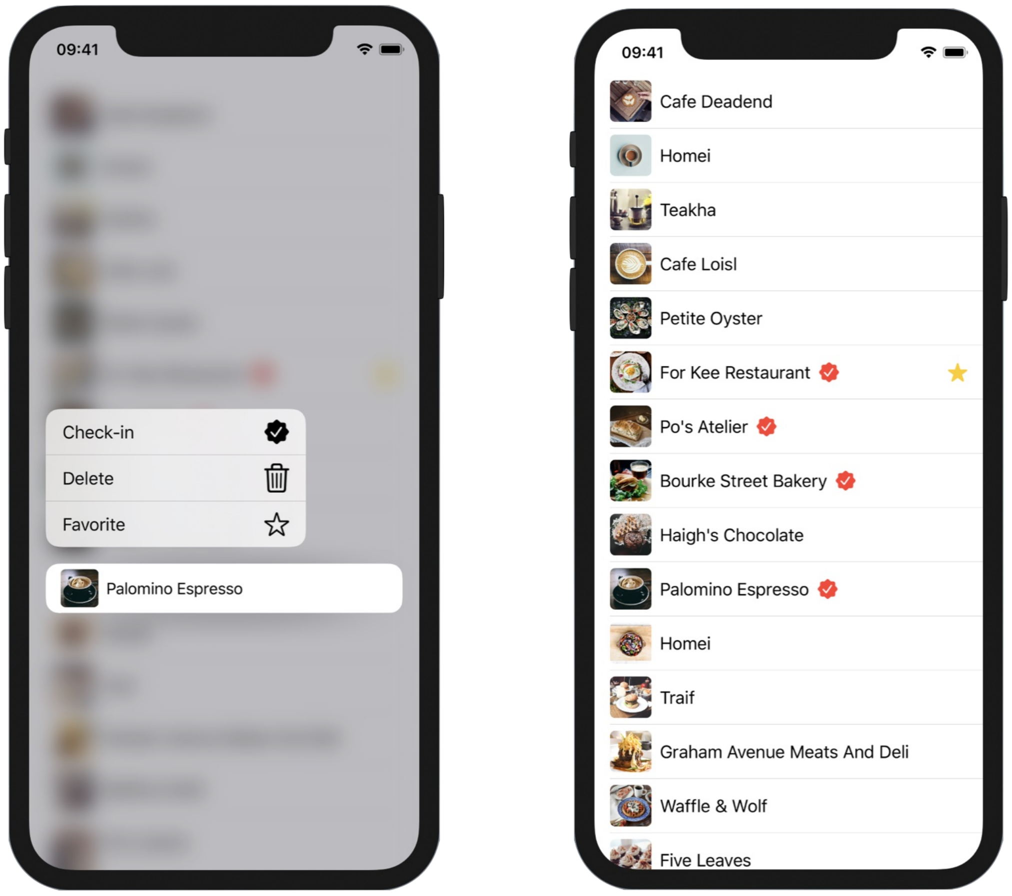


Figure 7. Adding a check-in feature

For reference, you can download the complete project here:

- Demo project and solution to exercise (https://www.appcoda.com/resources/swiftui2/SwiftUIActionSheet.zip)

Chapter 17
Understanding Gestures

In earlier chapters, you got a taste of building gestures with SwiftUI. We used the `onTapGesture` modifier to handle a user's touch and provide a corresponding response. In this chapter, let's dive deeper to see how we work with various types of gestures in SwiftUI.

The framework provides several built-in gestures such as the tap gesture we have used before. additionally, *DragGesture*, *MagnificationGesture*, and *LongPressGesture* are some of the ready-to-use gestures. We will be looking at a couple of them and seeing how to work with gestures in SwiftUI. On top of that, you will learn how to build a generic view that supports the drag gesture.

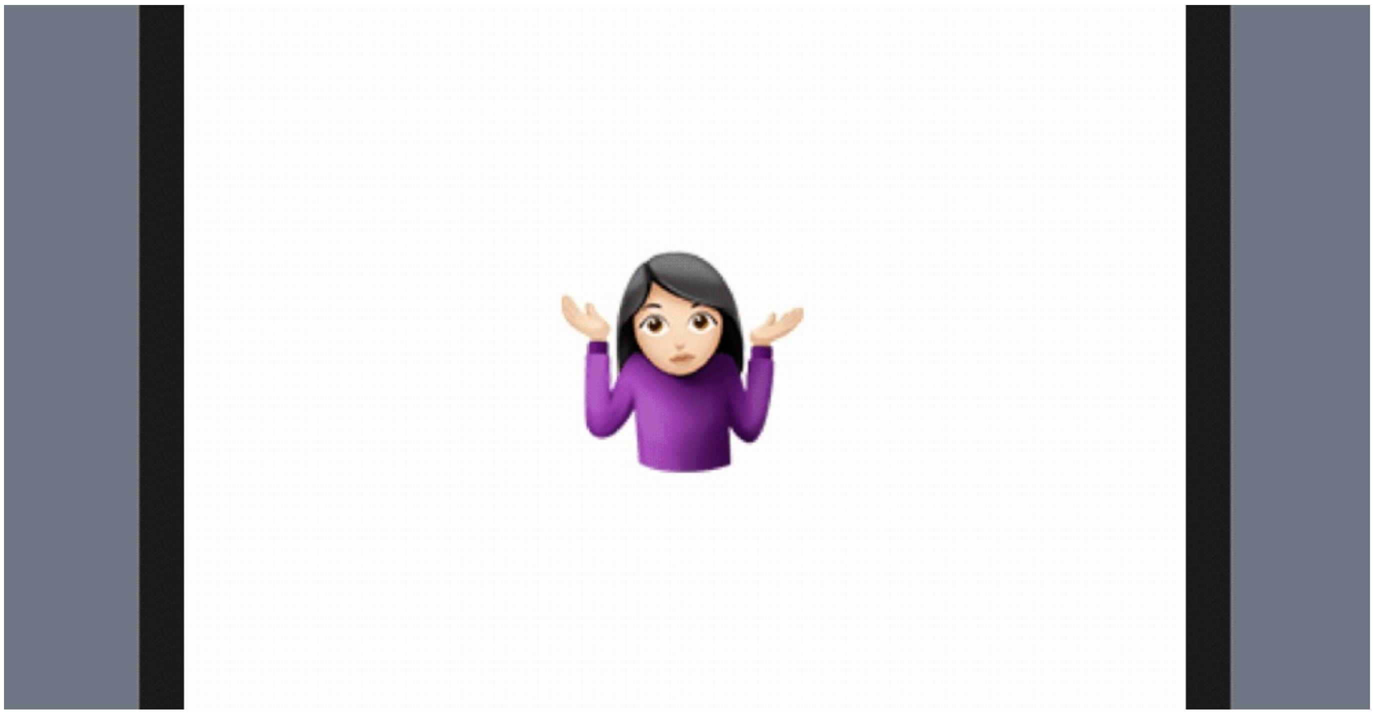


Figure 1. A demo showing the draggable view

Using the Gesture Modifier

To recognize a particular gesture using the SwiftUI framework, all you need to do is attach a gesture recognizer to a view using the `.gesture` modifier. Here is a sample code snippet which attaches the `TapGesture` using the `.gesture` modifier:

```
var body: some View {
    Image(systemName: "star.circle.fill")
        .font(.system(size: 200))
        .foregroundColor(.green)
        .gesture(
            TapGesture()
                .onEnded({
                    print("Tapped!")
                })
        )
}
```

If you want to try out the code, create a new project using the *App* template and make sure you select *SwiftUI* for the *Interface* option. Then paste the code in `ContentView.swift`.

By modifying the code above a bit and introducing a state variable, we can create a simple scale animation when the star image is tapped. Here is the updated code:

```
struct ContentView: View {
    @State private var isPressed = false

    var body: some View {
        Image(systemName: "star.circle.fill")
            .font(.system(size: 200))
            .scaleEffect(isPressed ? 0.5 : 1.0)
            .animation(.easeInOut)
            .foregroundColor(.green)
            .gesture(
                TapGesture()
                    .onEnded({
                        self.isPressed.toggle()
                    })
            )
    }
}
```

When you run the code in the canvas or simulator, you should see a scaling effect. This is how you use the `.gesture` modifier to detect and respond to certain touch events. If you forget how animation works, please go back to read chapter 9.

Figure 2. A simple scaling effect

Using Long Press Gesture

One of the built-in gesture is `LongPressGesture`. This gesture recognizer allows you to detect a long-press event. For example, if you want to resize the star image only when the user presses and holds it for at least 1 second, you can use the `LongPressGesture` to detect the touch event.

Modify the code in the `.gesture` modifier like this to implement the `LongPressGesture`:

```
.gesture(
    LongPressGesture(minimumDuration: 1.0)
        .onEnded({ _ in
            self.isPressed.toggle()
        })
)
```

Run the project in the preview canvas to see what it does. Now you have to press and hold the star image for at least a second before it toggles its size.

The @GestureState Property Wrapper

When you press and hold the star image, the image doesn't give the user any response until the long press event is detected. Obviously, there is something we can do to improve the user experience. What I want to do is to give the user immediate feedback when he/she taps the image. Any kind of feedback will help to improve the situation. Let's dim the image a bit when the user taps it. This just lets the user know that our app captures the touch and is doing work. Figure 3 illustrates how the animation works.

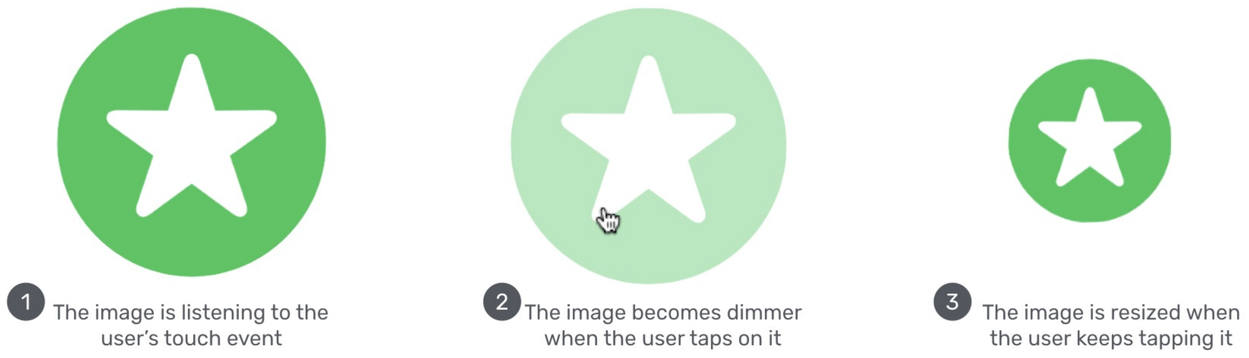

1 The image is listening to the user's touch event
2 The image becomes dimmer when the user taps on it
3 The image is resized when the user keeps tapping it

Figure 3. Applying a dimming effect when the image is tapped

To implement the animation, you need to keep track of the state of gestures. During the performance of the long press gesture, we have to differentiate between tap and long press events. So, how do we do that?

SwiftUI provides a property wrapper called `@GestureState` which conveniently tracks the state change of a gesture and lets developers decide the corresponding action. To implement the animation we just described, we can declare a property using `@GestureState` like this:

```
@GestureState private var longPressTap = false
```

This gesture state variable indicates whether a tap event is detected during the performance of the long press gesture. Once you have the variable defined, you can modify the code of the `Image` view like this:

```
Image(systemName: "star.circle.fill")
    .font(.system(size: 200))
    .opacity(longPressTap ? 0.4 : 1.0)
    .scaleEffect(isPressed ? 0.5 : 1.0)
    .animation(.easeInOut)
    .foregroundColor(.green)
    .gesture(
        LongPressGesture(minimumDuration: 1.0)
            .updating($longPressTap, body: { (currentState, state, transaction) in
                state = currentState
            })
            .onEnded({ _ in
                self.isPressed.toggle()
            })
    )
```

We only made a couple of changes in the code above. First, we added the `.opacity` modifier. When the tap event is detected, we set the opacity value to `0.4` so that the image becomes dimmer.

Second, we adddded the `updating` method of the `LongPressGesture`. During the performance of the long press gesture, this method will be called. It accepts three parameters: *value*, *state*, and *transaction*:

- The *value* parameter is the current state of the gesture. This value varies from gesture to gesture, but for the long press gesture, a `true` value indicates that a tap is detected.
- The *state* parameter is actually an in-out parameter that lets you update the value of the `longPressTap` property. In the code above, we set the value of `state` to `currentState`. In other words, the `longPressTap` property always keeps track of the latest state of the long press gesture.
- The `transaction` parameter stores the context of the current state-processing update.

After you make the code change, run the project in the preview canvas to test it. The image immediately becomes dimmer when you tap it. Keep holding it for one second and then the image resizes itself.

The opacity of the image is automatically reset to normal when the user releases the long press. Do you wonder why? This is an advantage of `@GestureState`. When the gesture ends, it automatically sets the value of the gesture state property to its initial value, `false` in our case.

Using Drag Gesture

Now that you understand how to use the `.gesture` modifier and `@GestureState`, let's look into another common gesture: *Drag*. What we are going to do is modify the existing code to support the drag gesture, allowing a user to drag the star image to move it around.

Replace the `ContentView` struct like this:

```
struct ContentView: View {
    @GestureState private var dragOffset = CGSize.zero

    var body: some View {
        Image(systemName: "star.circle.fill")
            .font(.system(size: 100))
            .offset(x: dragOffset.width, y: dragOffset.height)
            .animation(.easeInOut)
            .foregroundColor(.green)
            .gesture(
                DragGesture()
                    .updating($dragOffset, body: { (value, state, transaction) in

                        state = value.translation
                    })
            )
    }
}
```

To recognize a drag gesture, you initialize a `DragGesture` instance and listen for an update. In the `update` function, we pass a gesture state property to keep track of the drag event. Similar to the long press gesture, the closure of the `update` function accepts three parameters. In this case, the *value* parameter stores the current data of the drag including the translation. This is why we set the `state` variable, which is actually the `dragOffset`, to `value.translation`.

Run the project in the preview canvas, you can drag the image around. but, when you release it, the image returns to its original position.

Do you know why the image returns to its starting point? As explained in the previous section, one advantage of using `@GestureState` is that it will reset the value of the property to its original value when the gesture ends. So, when you end the drag and release the press, the `dragOffset` is reset to `.zero`, which is its original position.

But what if you want the image to stay at the end point of the drag? How do you do that? Give yourself a few minutes to think about how to implement it.

Since the `@GestureState` property wrapper will reset the property to its original value, we need another state property to save the final position. Therefore, let's declare a new state property like this:

```
@State private var position = CGSize.zero
```

Next, update the `body` variable like this:

```
var body: some View {
    Image(systemName: "star.circle.fill")
        .font(.system(size: 100))
        .offset(x: position.width + dragOffset.width, y: position.height + dragOff
set.height)
        .animation(.easeInOut)
        .foregroundColor(.green)
        .gesture(
            DragGesture()
                .updating($dragOffset, body: { (value, state, transaction) in

                    state = value.translation
                })
                .onEnded({ (value) in
                    self.position.height += value.translation.height
                    self.position.width += value.translation.width
                })
        )
}
```

We have made a couple of changes to the code:

1. We implemented the `onEnded` function which is called when the drag gesture ends. In the closure, we compute the new position of the image by adding the drag offset.
2. The `.offset` modifier was also updated, such that we take the current position into account.

Now when you run the project and drag the image, the image stays where it is even after the drag ends.

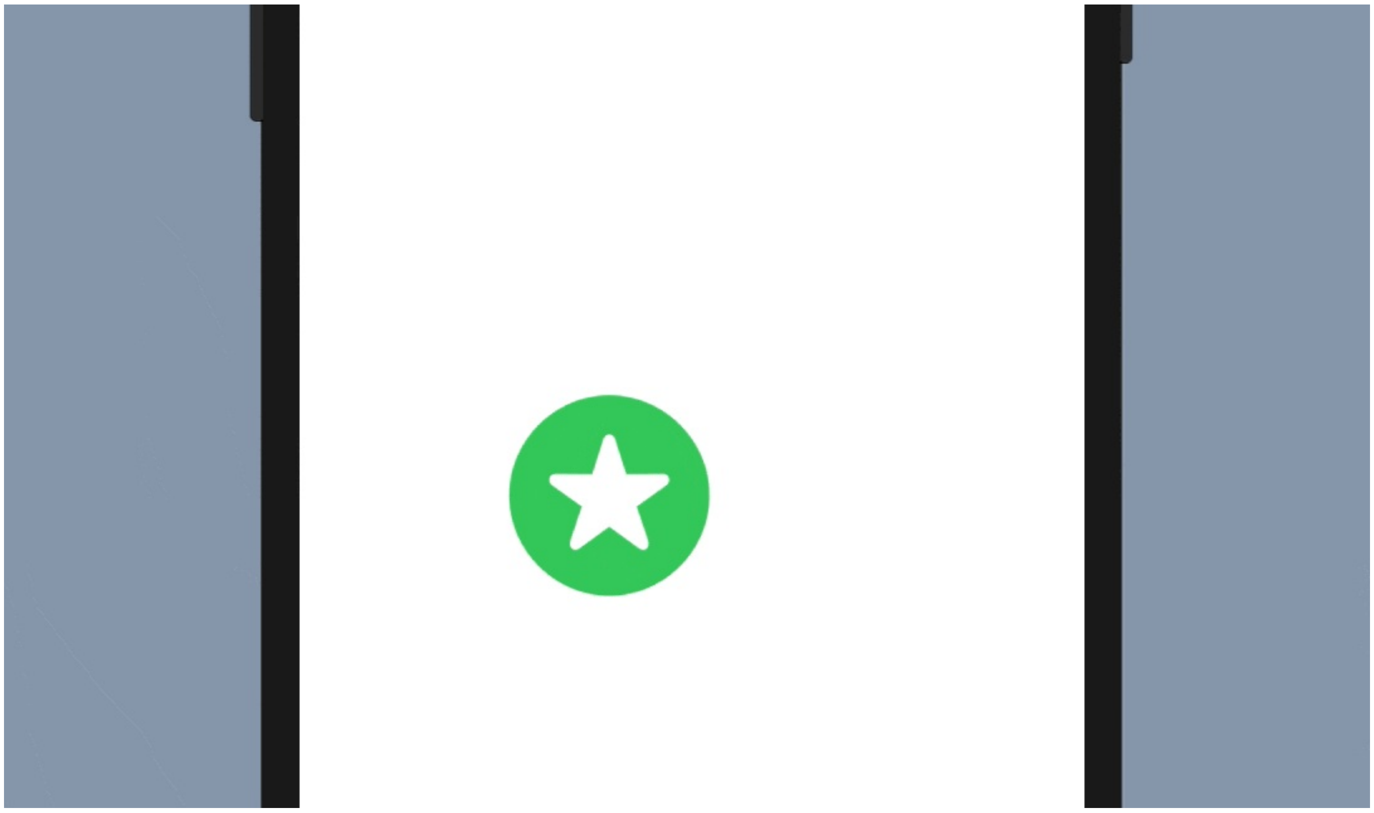


Figure 4. Drag the image around

Combining Gestures

In some cases, you need to use multiple gesture recognizers in the same view. Let's say, we want the user to press and hold the image before starting the drag, we have to combine both long press and drag gestures. SwiftUI allows you to easily combine gestures to perform more complex interactions. It provides three gesture composition types including *simultaneous*, *sequenced*, and *exclusive*.

When you need to detect multiple gestures at the same time, you use the *simultaneous* composition type. When you combine gestures using the *exclusive* composition type, SwiftUI recognizes all the gestures you specify but it will ignore the rest when one of the gestures is detected.

As the name suggests, if you combine multiple gestures using the *sequenced* composition type, SwiftUI recognizes the gestures in a specific order. This is the type of the composition that we will use to sequence the long press and drag gestures.

To work with multiple gestures, you update the code like this:

```
struct ContentView: View {
    // For long press gesture
    @GestureState private var isPressed = false

    // For drag gesture
    @GestureState private var dragOffset = CGSize.zero
    @State private var position = CGSize.zero

    var body: some View {
        Image(systemName: "star.circle.fill")
            .font(.system(size: 100))
            .opacity(isPressed ? 0.5 : 1.0)
            .offset(x: position.width + dragOffset.width, y: position.height + dra
gOffset.height)
            .animation(.easeInOut)
            .foregroundColor(.green)
            .gesture(
                LongPressGesture(minimumDuration: 1.0)
                .updating($isPressed, body: { (currentState, state, transaction) in

                    state = currentState
                })
                .sequenced(before: DragGesture())
                .updating($dragOffset, body: { (value, state, transaction) in

                    switch value {
                    case .first(true):
                        print("Tapping")
                    case .second(true, let drag):
                        state = drag?.translation ?? .zero
                    default:
                        break
                    }

                })
                .onEnded({ (value) in

                    guard case .second(true, let drag?) = value else {
                        return
                    }

                    self.position.height += drag.translation.height
                    self.position.width += drag.translation.width
                })
            )
    }
}
```

You should be very familiar with part of the code snippet because we are combining the long press gesture that we have built with the drag gesture.

Let me explain the code in the `.gesture` modifier line by line. We require the user to press and hold the image for at least one second before he/she can begin the dragging. So, we start by creating the `LongPressGesture`. Similar to what we have implemented before, we have a `isPressed` gesture state property. When someone taps the image, we will alter the opacity of the image.

The `sequenced` keyword is how we link the long press and drag gestures together. We tell SwiftUI that the `LongPressGesture` should happen before the `DragGesture`.

The code in both `updating` and `onEnded` functions looks pretty similar, but the `value` parameter now actually contains two gestures (i.e. long press and drag). We have the `switch` statement to differentiate between the gestures. You can use the `.first` and `.second` cases to find out which gesture to handle. Since the long press gesture should be recognized before the drag gesture, the *first* gesture here is the long press gesture. In the code, we do nothing but just print the *Tapping* message for your reference.

When the long press is confirmed, we will reach the `.second` case. Here, we pick up the *drag* data and update the `dragOffset` with the corresponding translation.

When the drag ends, the `onEnded` function will be called. Similarly, we update the final position by figuring out the drag data (i.e. `.second` case).

Now you're ready to test the gesture combination. Run the app in the preview canvas using the debug preview, so you can see the message in the console. You can't drag the image until holding the star image for at least one second.

Figure 5. Dragging only happens when a user presses and holds the image for at least one second

Refactoring the Code Using Enum

A better way to organize the drag state is by using Enum. This allows you to combine the `isPressed` and `dragOffset` state into a single property. Let's declare an enumeration called `DragState`.

```
enum DragState {
    case inactive
    case pressing
    case dragging(translation: CGSize)

    var translation: CGSize {
        switch self {
        case .inactive, .pressing:
            return .zero
        case .dragging(let translation):
            return translation
        }
    }

    var isPressing: Bool {
        switch self {
        case .pressing, .dragging:
            return true
        case .inactive:
            return false
        }
    }
}
```

We have three states here: *inactive*, *pressing*, and *dragging*. These states are good enough to represent the states during the performance of the long press and drag gestures. For the *dragging* state, we associate it with the translation of the drag.

With the `DragState` enum, we can modify the original code like this:

```
struct ContentView: View {
    @GestureState private var dragState = DragState.inactive
    @State private var position = CGSize.zero

    var body: some View {
        Image(systemName: "star.circle.fill")
            .font(.system(size: 100))
            .opacity(dragState.isPressing ? 0.5 : 1.0)
            .offset(x: position.width + dragState.translation.width, y: position.h
eight + dragState.translation.height)
            .animation(.easeInOut)
            .foregroundColor(.green)
            .gesture(
                LongPressGesture(minimumDuration: 1.0)
                .sequenced(before: DragGesture())
                .updating($dragState, body: { (value, state, transaction) in

                    switch value {
                    case .first(true):
                        state = .pressing
                    case .second(true, let drag):
                        state = .dragging(translation: drag?.translation ?? .zero)
                    default:
                        break
                    }

                })
                .onEnded({ (value) in

                    guard case .second(true, let drag?) = value else {
                        return
                    }

                    self.position.height += drag.translation.height
                    self.position.width += drag.translation.width
                })
            )
    }
}
```

We now declare a `dragState` property to track the drag state. By default, it's set to `DragState.inactive`. The code is nearly the same as the previous code except that it's modified to work with `dragState` instead of `isPressed` and `dragOffset`. For example, for the `.offset` modifier, we retrieve the drag offset from the associated value of the dragging state.

The result of the code is the same. However, it's always good practice to use Enum to track complicated states of gestures.

Building a Generic Draggable View

So far, we have built a draggable image view. What if we want to build a draggable text view? Or what if we want to create a draggable circle? Should you copy and paste all the code to create the text view or circle?

There is a better way to implement that. Let's see how we can build a generic draggable view.

In the project navigator, right click the `SwiftUIGesture` folder and choose *New File...*. Select the *SwiftUI View* template and name the file `DraggableView`.

Declare the `DragState` enum and update the `DraggableView` struct like this:

```
enum DraggableState {
    case inactive
    case pressing
    case dragging(translation: CGSize)

    var translation: CGSize {
        switch self {
        case .inactive, .pressing:
            return .zero
        case .dragging(let translation):
            return translation
        }
    }

    var isPressing: Bool {
        switch self {
```

```
        case .pressing, .dragging:
            return true
        case .inactive:
            return false
        }
    }
}

struct DraggableView<Content>: View where Content: View {
    @GestureState private var dragState = DraggableState.inactive
    @State private var position = CGSize.zero

    var content: () -> Content

    var body: some View {
        content()
            .opacity(dragState.isPressing ? 0.5 : 1.0)
            .offset(x: position.width + dragState.translation.width, y: position.h
eight + dragState.translation.height)
            .animation(.easeInOut)
            .gesture(
                LongPressGesture(minimumDuration: 1.0)
                .sequenced(before: DragGesture())
                .updating($dragState, body: { (value, state, transaction) in

                    switch value {
                    case .first(true):
                        state = .pressing
                    case .second(true, let drag):
                        state = .dragging(translation: drag?.translation ?? .zero)
                    default:
                        break
                    }

                })
                .onEnded({ (value) in

                    guard case .second(true, let drag?) = value else {
                        return
                    }

                    self.position.height += drag.translation.height
                    self.position.width += drag.translation.width
                })
            )
    }
}
```

All of the code is very similar to what you've written before. The tricks are to declare the `DraggableView` as a generic view and create a `content` property. This property accepts any `View`. We power this `content` view with the long press and drag gestures.

Now you can test this generic view by replacing the `DraggableView_Previews` like this:

```
struct DraggableView_Previews: PreviewProvider {
    static var previews: some View {
        DraggableView() {
            Image(systemName: "star.circle.fill")
                .font(.system(size: 100))
                .foregroundColor(.green)
        }
    }
}
```

In the code, we initalize a `DraggableView` and provide our own content, which is the star image. In this case, you should achieve the same star image which supports the long press and drag gestures.

So, what if we want to build a draggable text view? You can replace the code snippet with the following code:

```
struct DraggableView_Previews: PreviewProvider {
    static var previews: some View {
        DraggableView() {
            Text("Swift")
                .font(.system(size: 50, weight: .bold, design: .rounded))
                .bold()
                .foregroundColor(.red)
        }
    }
}
```

In the closure, we create a text view instead of the image view. If you run the project in the preview canvas, you can drag the text view to move it around (remember to long press for 1 second). Isn't it cool?

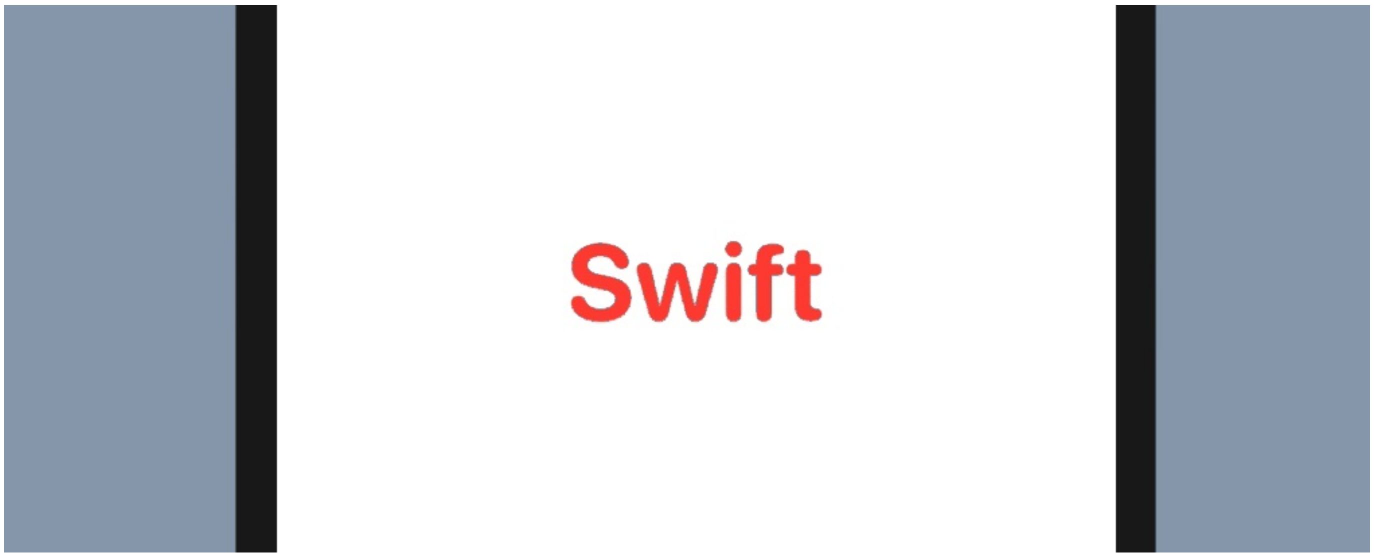


Figure 6. A draggable text view

If you want to create a draggable circle, you can replace the code like this:

```
struct DraggableView_Previews: PreviewProvider {
    static var previews: some View {
        DraggableView() {
            Circle()
                .frame(width: 100, height: 100)
                .foregroundColor(.purple)
        }
    }
}
```

That's how you create a generic draggable. Try to replace the circle with other views to make your own draggable view and have fun!

Exercise

We've explored three built-in gestures including tap, drag, and long press in this chapter. However, there are a couple of them we haven't checked out. As an exercise, try to create a generic scalable view that can recognize the `MagnificationGesture` and scale any given view accordingly. Figure 7 shows you a sample result.

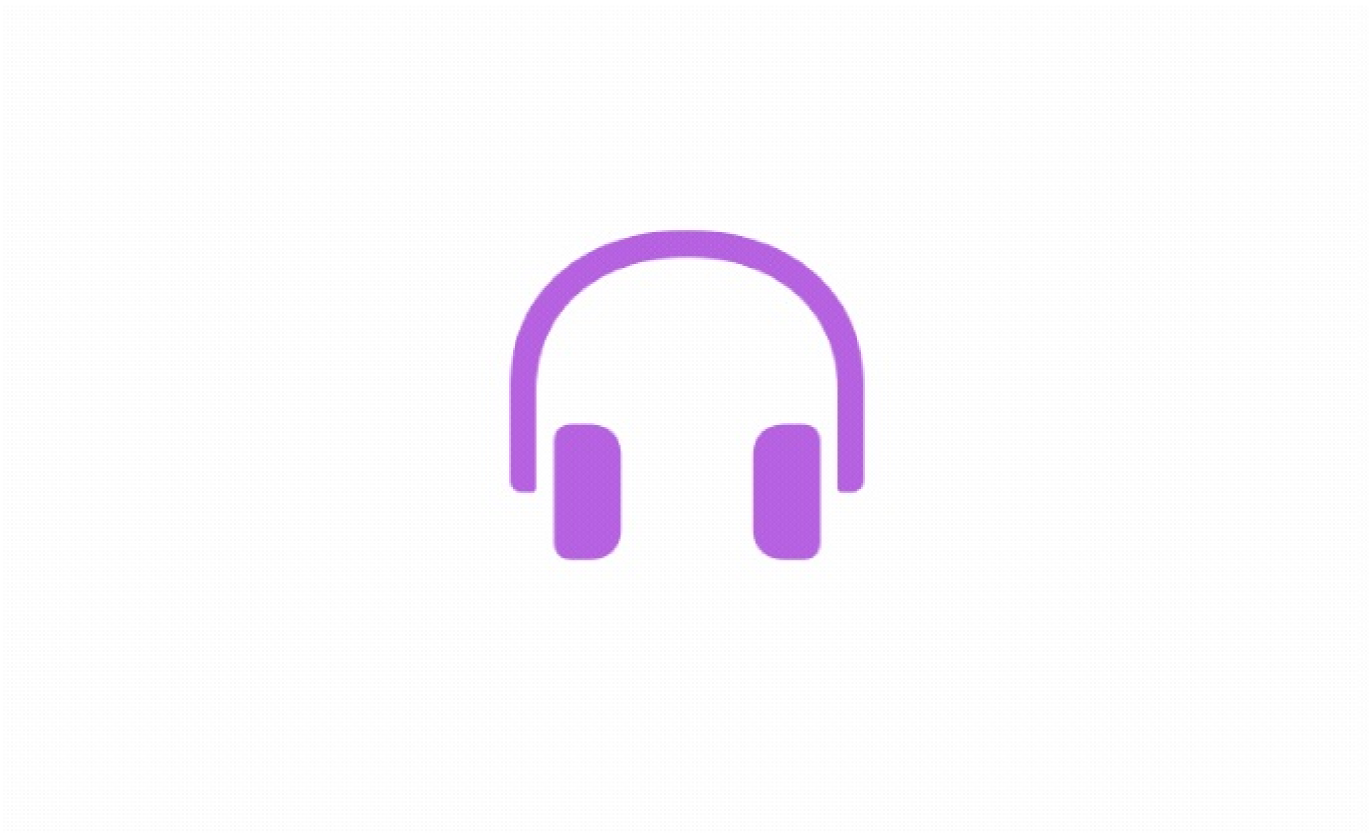


Figure 7. A scalable image view

Summary

The SwiftUI framework has made gesture handling very easy. As you've learned in this chapter, the framework has provided several ready to use gesture recognizers. To enable a view to support a certain type of gesture, all you need to do is attach to it the `.gesture` modifier. Composing multiple gestures has never been so simple.

It's a growing trend to build gesture-driven user interfaces for mobile apps. With the easy to use API, try to power your apps with some useful gestures to delight your users.

For reference, you can download the complete gesture project here:

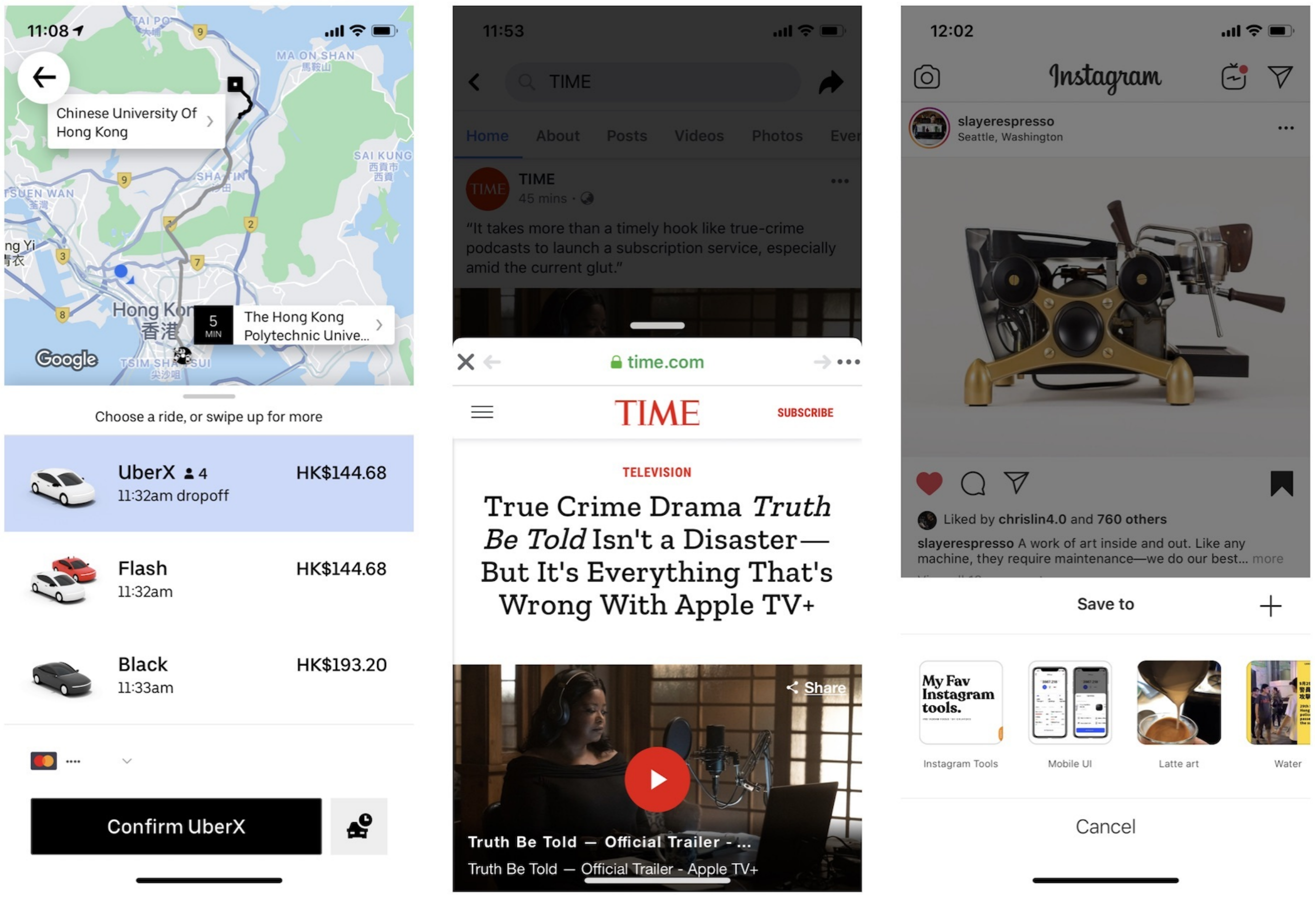
- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUIGesture.zip)

Chapter 18
Building an Expandable Bottom Sheet with SwiftUI Gestures and GeometryReader

Now that you have a basic understanding of SwiftUI gestures, let's see how you can apply the technique you learned to build a feature which is commonly used in real world apps.

Bottom sheets have increased in popularity lately! You can easily find them in famous apps like Facebook and Uber. Bottom sheets are like an enhanced version of an action sheet that slides up from the bottom of screen and overlays on top of the original view to provide contextual information or additional user options. For instance, when you save a photo to a collection in Instagram, the app shows you a bottom sheet to choose a collection. The Facebook app displays the sheet with additonal action items when you click the ellipsis button of a post. The Uber app also makes use of bottom sheets to display the pricing of your chosen trip.

The size of bottom sheets varies depending on the contextual information you want to display. In some cases, bottom sheets tend to be bigger (which is also known as backdrops) and take up 80-90% of the screen. Usually, users are allowed to interact with the sheet with the drag gesture. You can slide it up to expand its size or slide it down to minimize or dismiss the sheet.



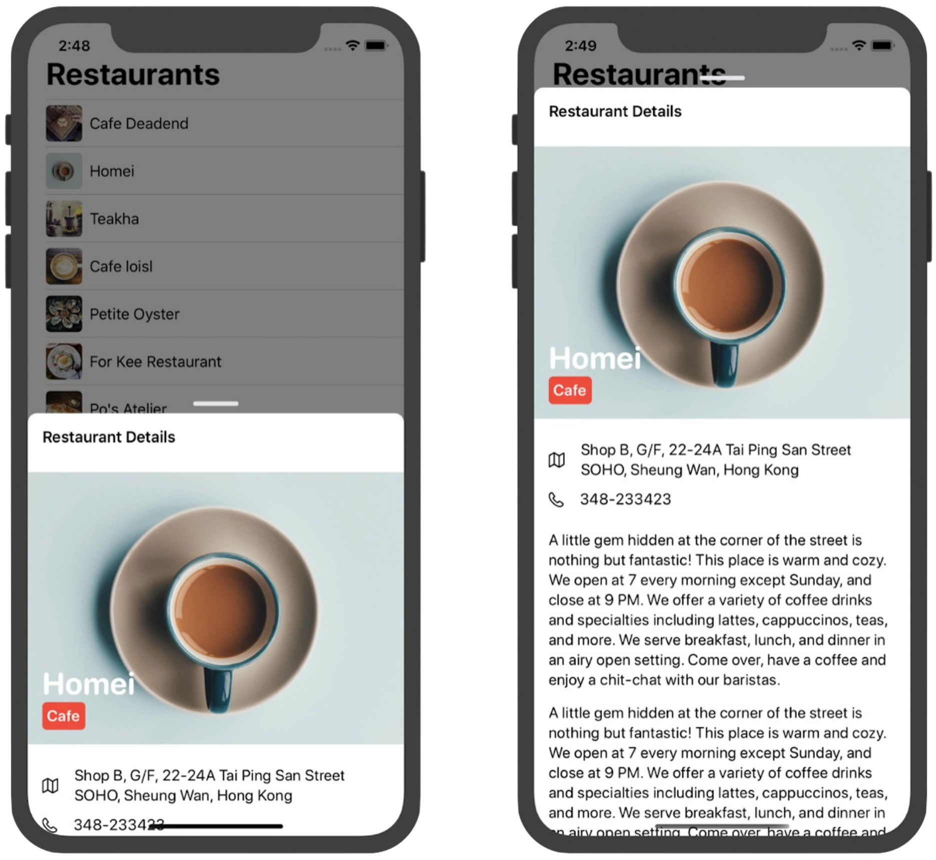
"It takes more than a timely hook like true-crime podcasts to launch a subscription service, especially amid the current glut."

True Crime Drama *Truth Be Told* Isn't a Disaster—But It's Everything That's Wrong With Apple TV+

slayerespresso A work of art inside and out. Like any machine, they require maintenance—we do our best... more

Figure 1. Uber, Facebook and Instagram all use bottom sheets in their apps

In this chapter, we will build a similar expandable bottom sheet using SwiftUI gestures. The demo app shows a list of restaurants in the main view. When a user taps one of the restaurant records, the app brings up a bottom sheet to display the restaurant details. You can expand the sheet by sliding it up. To dismiss the sheet, you can slide it down.



Shop B, G/F, 22-24A Tai Ping San Street SOHO, Sheung Wan, Hong Kong

348-233423

A little gem hidden at the corner of the street is nothing but fantastic! This place is warm and cozy. We open at 7 every morning except Sunday, and close at 9 PM. We offer a variety of coffee drinks and specialties including lattes, cappuccinos, teas, and more. We serve breakfast, lunch, and dinner in an airy open setting. Come over, have a coffee and enjoy a chit-chat with our baristas.

A little gem hidden at the corner of the street is nothing but fantastic! This place is warm and cozy. We open at 7 every morning except Sunday, and close at 9 PM. We offer a variety of coffee drinks and specialties including lattes, cappuccinos, teas, and more. We serve breakfast, lunch, and dinner in an airy open setting. Come over, have a coffee and

Figure 2. Building a expandable bottom sheet

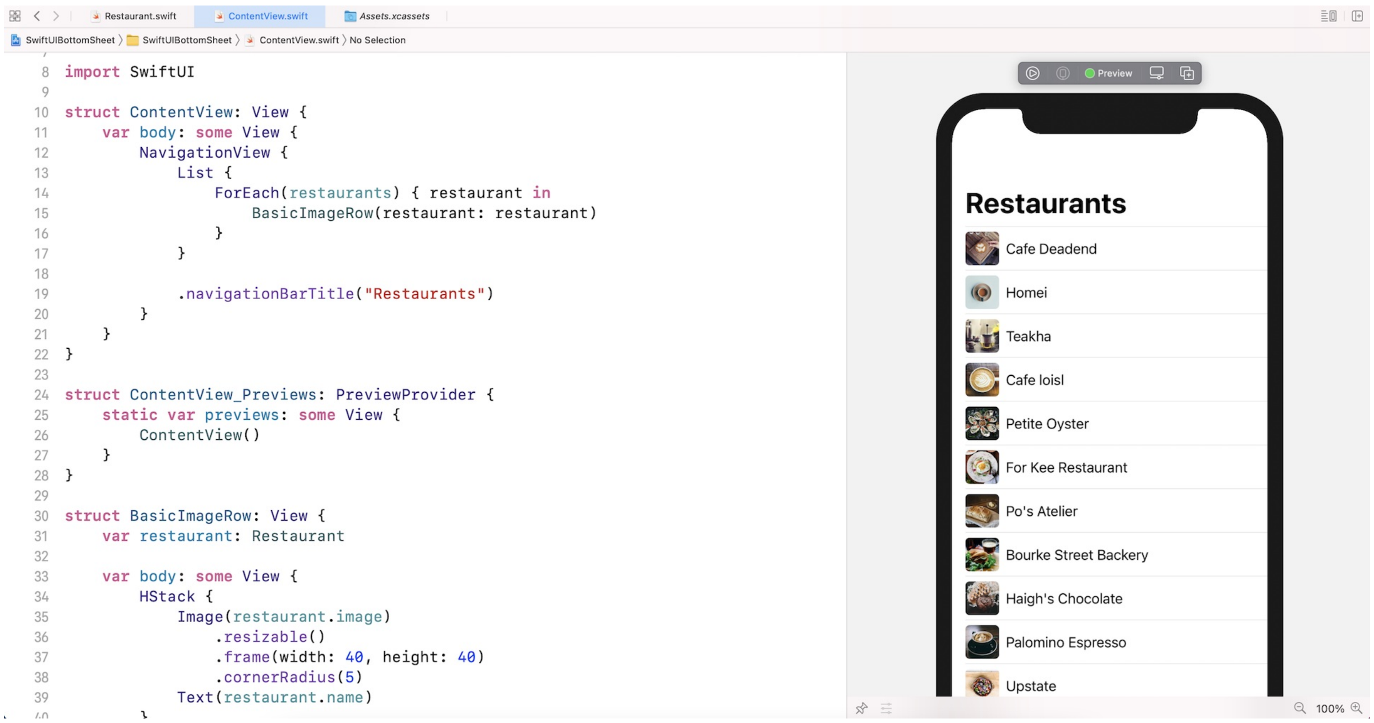
Understanding the Starter Project

To save you some time building the demo app from the ground up, I've prepared a starter project for you. You can download it from https://www.appcoda.com/resources/swiftui2/SwiftUIBottomSheetStarter.zip. Unzip the file and open `SwiftUIBottomSheet.xcodeproj` to get started.

The starter project comes with a set of restaurant images and the restaurant data. If you look in the *Model* folder in the project navigator, you should find a file named `Restaurant.swift`. This file contains the `Restaurant` struct and the set of sample restaurant data.

```
struct Restaurant: Identifiable {
    var id: UUID = UUID()
    var name: String
    var type: String
    var location: String
    var phone: String
    var description: String
    var image: String
    var isVisited: Bool

    init(name: String, type: String, location: String, phone: String, description:
String, image: String, isVisited: Bool) {
        self.name = name
        self.type = type
        self.location = location
        self.phone = phone
        self.description = description
        self.image = image
        self.isVisited = isVisited
    }

    init() {
        self.init(name: "", type: "", location: "", phone: "", description: "", im
age: "", isVisited: false)
    }
}
```

I've created the main view for you that displays a list of restaurants. You can open the `ContentView.swift` file to check out the code. I am not going to explain the code in details as we have gone through the implementation of list in chapter 10.


Restaurant.swift
ContentView.swift
Assets.xcassets
SwiftUIBottomSheet 〉 SwiftUIBottomSheet 〉 ContentView.swift 〉 No Selection
Preview
100%

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        NavigationView {
            List {
                ForEach(restaurants) { restaurant in
                    BasicImageRow(restaurant: restaurant)
                }
            }

            .navigationBarTitle("Restaurants")
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}

struct BasicImageRow: View {
    var restaurant: Restaurant

    var body: some View {
        HStack {
            Image(restaurant.image)
                .resizable()
                .frame(width: 40, height: 40)
                .cornerRadius(5)
            Text(restaurant.name)
```

Restaurants

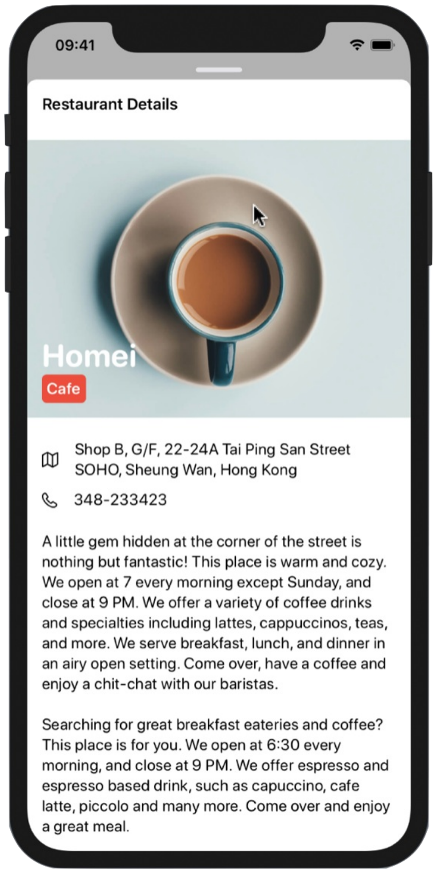
Cafe Deadend
Homei
Teakha
Cafe loisl
Petite Oyster
For Kee Restaurant
Po's Atelier
Bourke Street Backery
Haigh's Chocolate
Palomino Espresso
Upstate

Figure 3. The list view

When you run the code in the canvas or simulator, you should see a scaling effect. This is how you can use the `.gesture` modifier to detect and respond to certain touch events. If you forget how the animation works, please go back to read chapter 9.

Creating the Restaurant Detail View

The bottom sheet will contain the restaurant details with a small handlebar. So, the very first thing we have to do is to create the restaurant detail view like that shown in figure 4.



Shop B, G/F, 22-24A Tai Ping San Street
SOHO, Sheung Wan, Hong Kong

348-233423

A little gem hidden at the corner of the street is nothing but fantastic! This place is warm and cozy. We open at 7 every morning except Sunday, and close at 9 PM. We offer a variety of coffee drinks and specialties including lattes, cappuccinos, teas, and more. We serve breakfast, lunch, and dinner in an airy open setting. Come over, have a coffee and enjoy a chit-chat with our baristas.

Searching for great breakfast eateries and coffee? This place is for you. We open at 6:30 every morning, and close at 9 PM. We offer espresso and espresso based drink, such as capuccino, cafe latte, piccolo and many more. Come over and enjoy a great meal.

Figure 4. The restaurant detail view with a small handlebar

Before you follow me to implement the view, I suggest you consider it as an exercise and create the detail view on your own. As you can see, the detail view is composed of UI components including *Image*, *Text*, and *ScrollView*. We have already covered all these components, so give it a try and provide your own implementation.

Okay, let me show you how to build the detail view. If you have already built the detail view on your own, you can use my implementation as a reference.

The layout of the detail view is a bit complicated, so it's better to break it into multiple parts for easier implementation:

- The handlebar, which is a small rounded rectangle
- The title bar containing the title of the detail view
- The header view containing the featured image, restaurant name, and type

- The detail info view containing the restaurant data, which includes address, phone, and description.

We will implement each of the above using a separate `struct` to better organize our code. Now create a new file using the *SwiftUI View* template and name it `RestaurantDetailView.swift`. All the code discussed below will be put in this new file.

Handlebar

First, the handlebar. The handlebar is actually a small rectangle with rounded corners. To create it, all we need to do is to create a `Rectangle` and give it rounded corners. In the `RestaurantDetailView.swift` file, insert the following code:

```
struct HandleBar: View {

    var body: some View {
        Rectangle()
            .frame(width: 50, height: 5)
            .foregroundColor(Color(.systemGray5))
            .cornerRadius(10)
    }
}
```

Title Bar

Next, the title bar. The implementation is simple since it's just a *Text* view. Let's create another struct for it:

```
struct TitleBar: View {

    var body: some View {
        HStack {
            Text("Restaurant Details")
                .font(.headline)
                .foregroundColor(.primary)

            Spacer()
        }
        .padding()
    }
}
```

The spacer here is used to align the text to the left.

Header View

The header view consists of an image view and two text views. The text views are overlayed on top of the image view. Again, we will use a separate struct to implement the header view:

```
struct HeaderView: View {
    let restaurant: Restaurant

    var body: some View {
        Image(restaurant.image)
            .resizable()
            .scaledToFill()
            .frame(height: 300)
            .clipped()
            .overlay(
                HStack {
                    VStack(alignment: .leading) {
                        Spacer()
                        Text(restaurant.name)
                            .foregroundColor(.white)
                            .font(.system(.largeTitle, design: .rounded))
                            .bold()

                        Text(restaurant.type)
                            .font(.system(.headline, design: .rounded))
                            .padding(5)
                            .foregroundColor(.white)
                            .background(Color.red)
                            .cornerRadius(5)

                    }
                    Spacer()
                }
                .padding()
            )
    }
}
```

Since we need to display the restaurant data, the `HeaderView` has the `restaurant` property. For the layout, we created an `Image` view and set the content mode to `scaleToFill`. The height of the image is fixed at 300 points. Since we use the `scaleToFill` mode, we need to attach the `.clipped()` modifier to hide any content that extends beyond the edges of the image frame.

For the two labels, we use the `.overlay` modifier to overlay two `Text` views.

Detail Info View

Lastly, the information view. If you look at the address, phone, and description fields carefully, you should notice that they are pretty similar. Both address and phone fields have an icon right next to the textual information, while the description field contains text only.

So, wouldn't it be great to build a view which is flexible to handle both field types? Here is the code snippet:

```
struct DetailInfoView: View {
    let icon: String?
    let info: String

    var body: some View  {
        HStack {
            if icon != nil {
                Image(systemName: icon!)
                    .padding(.trailing, 10)
            }
            Text(info)
                .font(.system(.body, design: .rounded))

            Spacer()
        }.padding(.horizontal)
    }
}
```

The `DetailInfoView` takes in two parameters: `icon` and `info`. The `icon` parameter is an optional, meaning that it can either have a value or nil.

When you need to present a data field with an icon, you use the `DetailInfoView` like this:

```
DetailInfoView(icon: "map", info: self.restaurant.location)
```

Alternatively, if you only need to present a text-only field like the description field, you use the `DetailInfoView` like this:

```
DetailInfoView(icon: nil, info: self.restaurant.description)
```

As you can see, by building a generic view to handle similar layout, you make the code more modular and reusable.

Using VStack to Glue Them All Together

Now that we have built all components, we can combine them by using `VStack` like this:

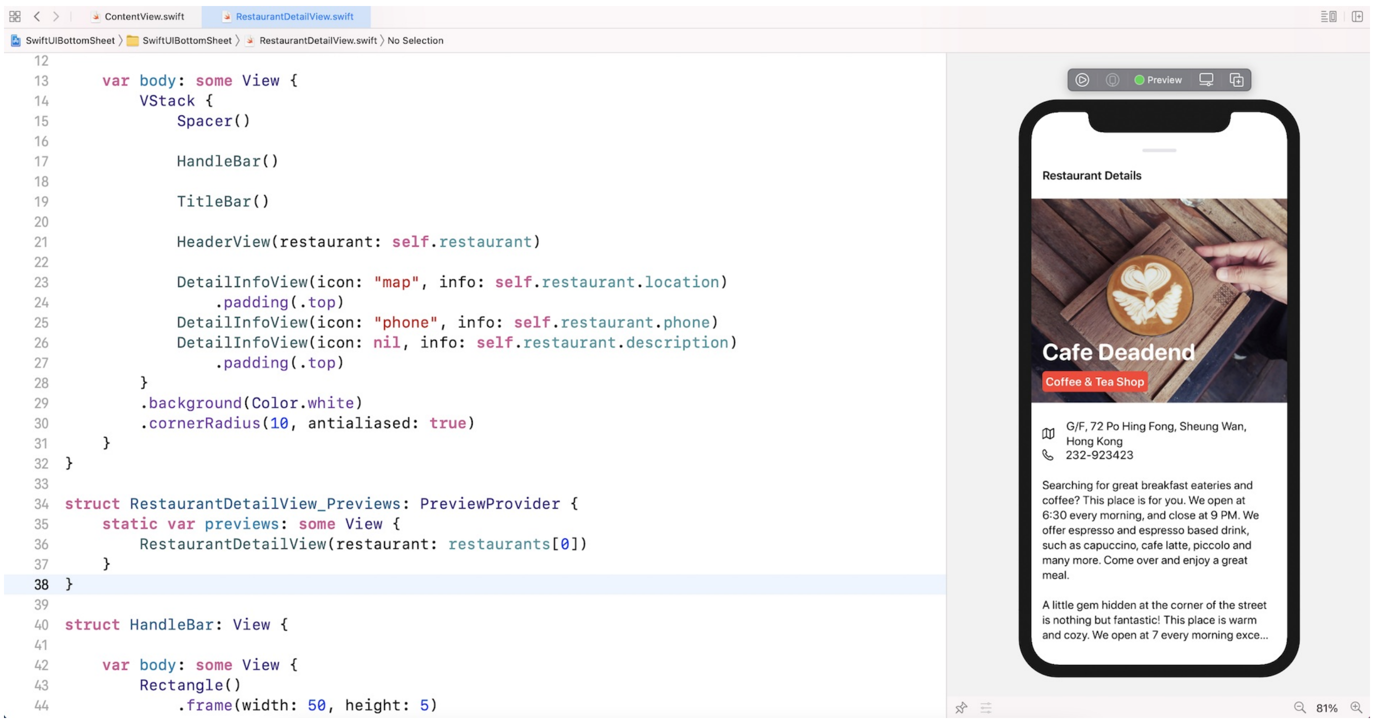
```
struct RestaurantDetailView: View {
    let restaurant: Restaurant

    var body: some View {
        VStack {
            Spacer()

            HandleBar()

            TitleBar()

            HeaderView(restaurant: self.restaurant)

            DetailInfoView(icon: "map", info: self.restaurant.location)
                .padding(.top)
            DetailInfoView(icon: "phone", info: self.restaurant.phone)
            DetailInfoView(icon: nil, info: self.restaurant.description)
                .padding(.top)
        }
        .background(Color.white)
        .cornerRadius(10, antialiased: true)
    }
}
```

The code above is self explanatory. We use the components that were built in the earlier sections and embed them in a vertical stack. Originally, the `VStack` has a transparent background. To ensure that the detail view has a white background, we attach the `background` modifier.

Before you can test the detail view, you have to modify the code of `RestaurantDetailView_Previews` like this:

```
struct RestaurantDetailView_Previews: PreviewProvider {
    static var previews: some View {
        RestaurantDetailView(restaurant: restaurants[0])
    }
}
```

In the code, we pass a sample restaurant (i.e. `restaurants[0]`) for testing. If you've followed everything correctly, Xcode should show you a similar detail view in the preview canvas to figure 5.


ContentView.swift
RestaurantDetailView.swift
SwiftUIBottomSheet 〉 SwiftUIBottomSheet 〉 RestaurantDetailView.swift 〉 No Selection
Preview
81%

```
    var body: some View {
        VStack {
            Spacer()

            HandleBar()

            TitleBar()

            HeaderView(restaurant: self.restaurant)

            DetailInfoView(icon: "map", info: self.restaurant.location)
                .padding(.top)
            DetailInfoView(icon: "phone", info: self.restaurant.phone)
            DetailInfoView(icon: nil, info: self.restaurant.description)
                .padding(.top)
        }
        .background(Color.white)
        .cornerRadius(10, antialiased: true)
    }
}

struct RestaurantDetailView_Previews: PreviewProvider {
    static var previews: some View {
        RestaurantDetailView(restaurant: restaurants[0])
    }
}

struct HandleBar: View {

    var body: some View {
        Rectangle()
            .frame(width: 50, height: 5)
```

Restaurant Details

Cafe Deadend

Coffee & Tea Shop

G/F, 72 Po Hing Fong, Sheung Wan, Hong Kong

232-923423

Searching for great breakfast eateries and coffee? This place is for you. We open at 6:30 every morning, and close at 9 PM. We offer espresso and espresso based drink, such as capuccino, cafe latte, piccolo and many more. Come over and enjoy a great meal.

A little gem hidden at the corner of the street is nothing but fantastic! This place is warm and cozy. We open at 7 every morning exce...

Figure 5. The restaurant detail view

Make It Scrollable

Do you notice that the detail view can't display the full description? To fix the issue, we have to make the detail view scrollable by embedding the content in a `ScrollView` like this:

```
struct RestaurantDetailView: View {
    let restaurant: Restaurant

    var body: some View {
        VStack {
            Spacer()

            HandleBar()

            ScrollView(.vertical) {
                TitleBar()

                HeaderView(restaurant: self.restaurant)

                DetailInfoView(icon: "map", info: self.restaurant.location)
                    .padding(.top)
                DetailInfoView(icon: "phone", info: self.restaurant.phone)
                DetailInfoView(icon: nil, info: self.restaurant.description)
                    .padding(.top);
            }
            .background(Color.white)
            .cornerRadius(10, antialiased: true)
        }
    }
}
```

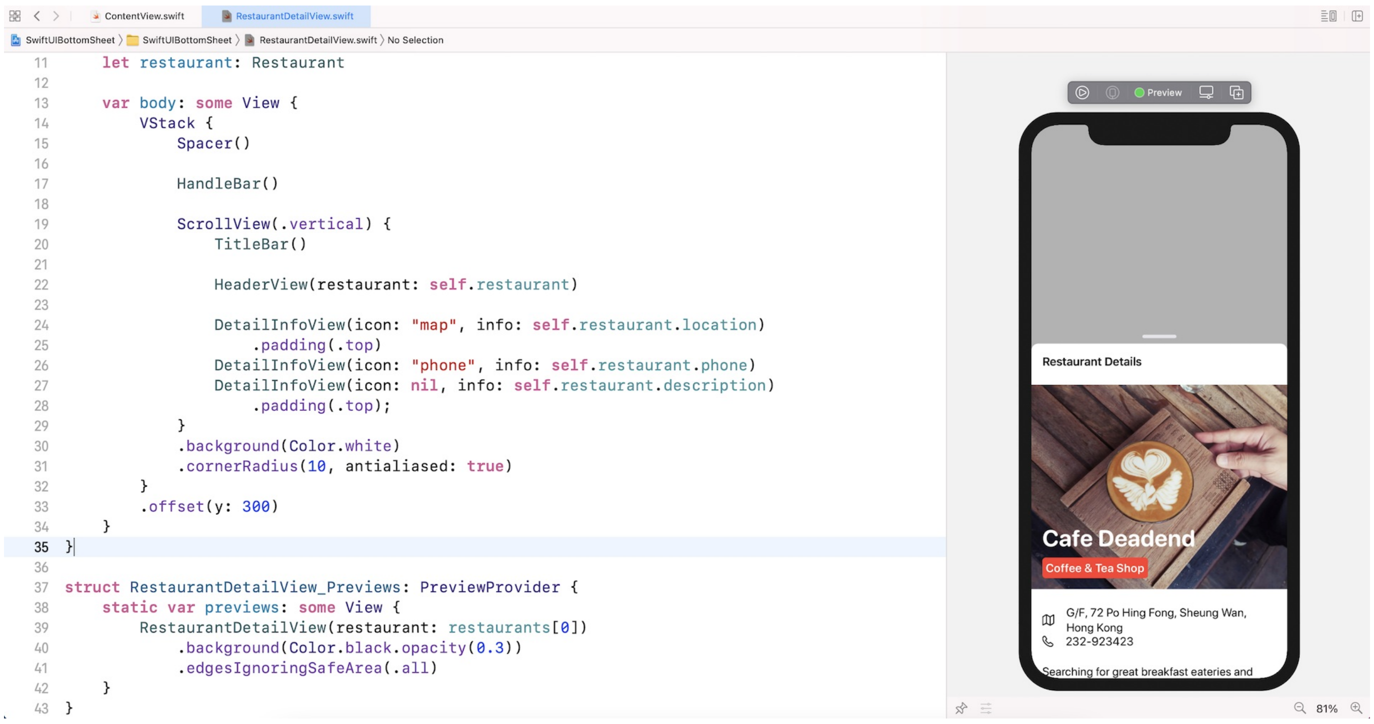
Except the handlebar, the rest of the views are wrapped within the scroll view. If you run the app in the preview canvas again, the detail view is now scrollable.

Adjusting the Offset

A bottom sheet is overlaid on top of the original content but usually only covers part of it. Therefore, we have to adjust the detail view's offset so that it only covers part of the screen. To achieve that, we can attach the `offset` modifier to the `VStack` like this:

```
.offset(y: 300)
```

This moves the detail view downward by 300 points. If you test the code in the preview canvas, the detail view should be shifted to the lower part of the screen.


ContentView.swift
RestaurantDetailView.swift
SwiftUIBottomSheet 〉 SwiftUIBottomSheet 〉 RestaurantDetailView.swift 〉 No Selection
Preview
Restaurant Details
Cafe Deadend
Coffee & Tea Shop
G/F, 72 Po Hing Fong, Sheung Wan, Hong Kong
232-923423
Searching for great breakfast eateries and
81%

```
    let restaurant: Restaurant

    var body: some View {
        VStack {
            Spacer()

            HandleBar()

            ScrollView(.vertical) {
                TitleBar()

                HeaderView(restaurant: self.restaurant)

                DetailInfoView(icon: "map", info: self.restaurant.location)
                    .padding(.top)
                DetailInfoView(icon: "phone", info: self.restaurant.phone)
                DetailInfoView(icon: nil, info: self.restaurant.description)
                    .padding(.top);
            }
            .background(Color.white)
            .cornerRadius(10, antialiased: true)
        }
        .offset(y: 300)
    }
}

struct RestaurantDetailView_Previews: PreviewProvider {
    static var previews: some View {
        RestaurantDetailView(restaurant: restaurants[0])
            .background(Color.black.opacity(0.3))
            .edgesIgnoringSafeArea(.all)
    }
}
```

Figure 6. Adjusting the offset of the detail view

To make it look more like the final result, you can also change the background color of `RestaurantDetailView_Previews` like this:

```
struct RestaurantDetailView_Previews: PreviewProvider {
    static var previews: some View {
        RestaurantDetailView(restaurant: restaurants[0])
            .background(Color.black.opacity(0.3))
            .edgesIgnoringSafeArea(.all)
    }
}
```

The detail view looks pretty good right now. However, one problem is that the offset is set to a fixed value. As the app is going to support multiple devices or screen sizes, the offset value should be able to adjust itself automatically.

The offset value should be set to half of the screen height. So, how can you find out the screen size of a device? SwiftUI provides a container view called *GeometryReader* that gives you access to the size and position of the parent view. Therefore, to get the screen height, all you need to do is wrap the `VStack` with a `GeometryReader` like this:

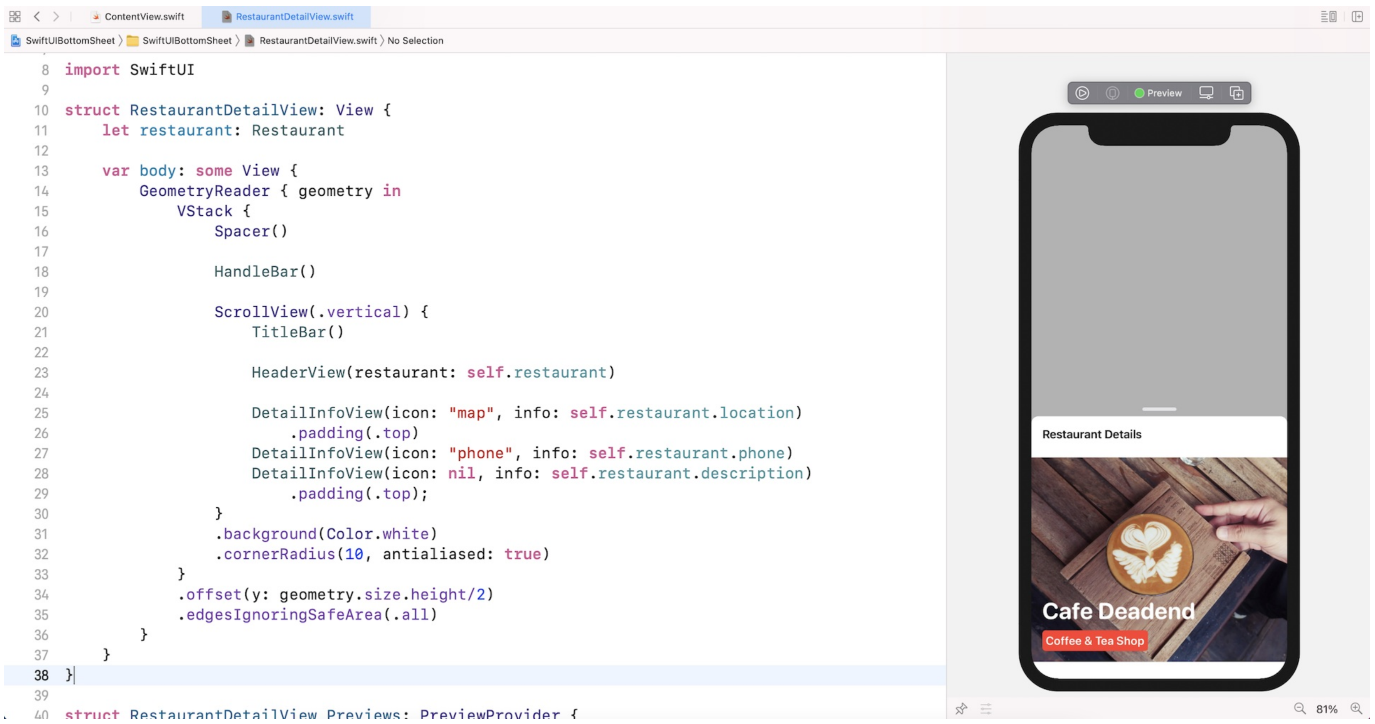
```
struct RestaurantDetailView: View {
    let restaurant: Restaurant

    var body: some View {
        GeometryReader { geometry in
            VStack {
                Spacer()

                HandleBar()

                ScrollView(.vertical) {
                    TitleBar()

                    HeaderView(restaurant: self.restaurant)

                    DetailInfoView(icon: "map", info: self.restaurant.location)
                        .padding(.top)
                    DetailInfoView(icon: "phone", info: self.restaurant.phone)
                    DetailInfoView(icon: nil, info: self.restaurant.description)
                        .padding(.top)
                }
                .background(Color.white)
                .cornerRadius(10, antialiased: true)
            }
            .offset(y: geometry.size.height/2)
            .edgesIgnoringSafeArea(.all)
        }
    }
}
```

In the closure, we can access the size of the parent view using the `geometry` parameter. This is why we set the `offset` modifier like this:

```
.offset(y: geometry.size.height/2)
```

To correctly compute the full screen size, we added the `edgesIgnoringSafeArea` modifier and set its parameter to `.all` to completely ignore the safe area.

Now run the app again in the preview canvas. You should have a bottom sheet which takes up half of the screen size.


ContentView.swift
RestaurantDetailView.swift
SwiftUIBottomSheet 〉 SwiftUIBottomSheet 〉 RestaurantDetailView.swift 〉 No Selection
Preview
Restaurant Details
Cafe Deadend
Coffee & Tea Shop
81%

```swift
import SwiftUI

struct RestaurantDetailView: View {
    let restaurant: Restaurant

    var body: some View {
        GeometryReader { geometry in
            VStack {
                Spacer()

                HandleBar()

                ScrollView(.vertical) {
                    TitleBar()

                    HeaderView(restaurant: self.restaurant)

                    DetailInfoView(icon: "map", info: self.restaurant.location)
                        .padding(.top)
                    DetailInfoView(icon: "phone", info: self.restaurant.phone)
                    DetailInfoView(icon: nil, info: self.restaurant.description)
                        .padding(.top);
                }
                .background(Color.white)
                .cornerRadius(10, antialiased: true)
            }
            .offset(y: geometry.size.height/2)
            .edgesIgnoringSafeArea(.all)
        }
    }
}

struct RestaurantDetailView_Previews: PreviewProvider {
```

Figure 7. Adjusting the offset of the detail view

Bring Up the Detail View

Now that the detail view is pretty much done. Let's go back to the list view (i.e. `ContentView.swift`) to bring it up whenever a user selects a restaurant.

In the `ContentView` struct, declare two state variables:

```swift
@State private var showDetail = false
@State private var selectedRestaurant: Restaurant?
```

The `showDetail` variable indicates whether the detail view is shown, while the `selectedRestaurant` variable stores the user's chosen restaurant.

As you've learned in an earlier chapter, you can attach the `onTapGesture` modifier to detect the tap gesture. So, when a tap is recognized, we toggle the value of `showDetail` and update the value of `selectedRestaurant` like this:

```
List {
    ForEach(restaurants) { restaurant in
        BasicImageRow(restaurant: restaurant)
            .onTapGesture {
                self.showDetail = true
                self.selectedRestaurant = restaurant
            }
    }
}
```

The detail view, which is the bottom sheet, is expected to overlay on top of the list view. To achieve that, embed the navigation view in a `ZStack`. Right below the navigation view, we will check if the detail view is enabled and initialize it like this:

```
var body: some View {
    ZStack {
        NavigationView {
            List {
                ForEach(restaurants) { restaurant in
                    BasicImageRow(restaurant: restaurant)
                        .onTapGesture {
                            self.showDetail = true
                            self.selectedRestaurant = restaurant
                        }
                }
            }

            .navigationBarTitle("Restaurants")
        }

        if showDetail {
            if let selectedRestaurant = selectedRestaurant {
                RestaurantDetailView(restaurant: selectedRestaurant)
                    .transition(.move(edge: .bottom))
            }
        }
    }
}
```

We attach the `transition` modifier to the detail view such that it uses the `move` transition type. The `selectedRestaurant` property is defined as an optional. This means it can either have a value or nil. Before accessing the value of the property, it's required to check if `selectedRestaurant` has a value or not. Therefore, we use `if let` to perform the verification. As a side note, this `if let` operator is only supported in iOS 14 (or up).

If you run the app in the preview canvas, it will bring up the detail view when you select a restaurant. However, the implementation of the bottom sheet is far from completion.

First, the list view is not blocked from user interaction when the bottom sheet is active. In fact, the list view should be dimmed to indicate it's the back layer.

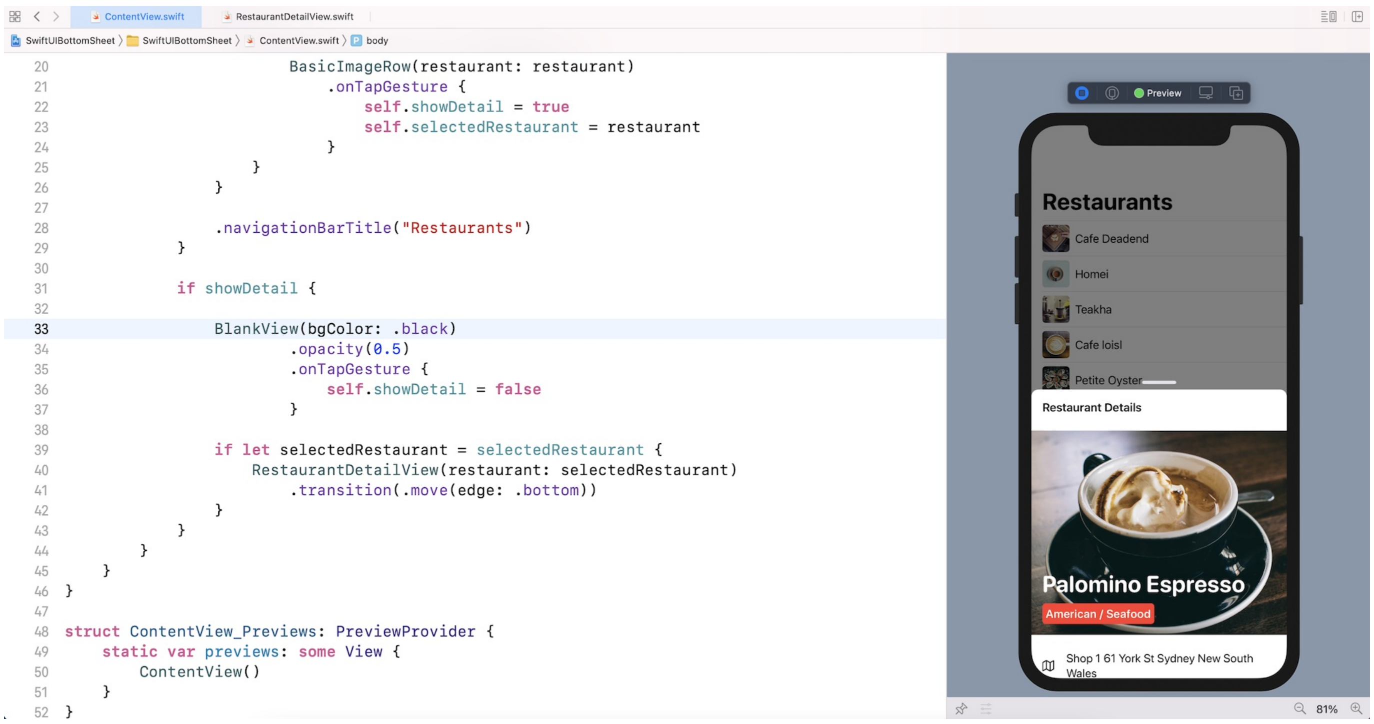
To implement this, we create an empty view and place it between the list view and the detail view. In the `ContentView.swift` file, insert the following code to create the empty view:

```
struct BlankView : View {

    var bgColor: Color

    var body: some View {
        VStack {
            Spacer()
        }
        .frame(minWidth: 0, maxWidth: .infinity, minHeight: 0, maxHeight: .infinit
y)
        .background(bgColor)
        .edgesIgnoringSafeArea(.all)
    }
}
```

Next, update the `if` clause like this:

```
if showDetail {

    BlankView(bgColor: .black)
            .opacity(0.5)
            .onTapGesture {
                self.showDetail = false
            }

    if let selectedRestaurant = selectedRestaurant {
        RestaurantDetailView(restaurant: selectedRestaurant)
            .transition(.move(edge: .bottom))
    }
}
```

When the detail view is displayed, we place an empty view right below it. The empty view is filled in black and semi opaque. This will block users from interacting with the list view but still keep them in the original context. We also attach a tap gesture recognizer to the empty view, so that the detail view will be dismissed whenever the user taps the empty area.



```
                        BasicImageRow(restaurant: restaurant)
                            .onTapGesture {
                                self.showDetail = true
                                self.selectedRestaurant = restaurant
                            }
                    }
                }

                .navigationBarTitle("Restaurants")
            }

            if showDetail {

                BlankView(bgColor: .black)
                        .opacity(0.5)
                        .onTapGesture {
                            self.showDetail = false
                        }

                if let selectedRestaurant = selectedRestaurant {
                    RestaurantDetailView(restaurant: selectedRestaurant)
                        .transition(.move(edge: .bottom))
                }
            }
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 8. Dimming the list view

Now run the app and try out the changes. When you tap the dimmed area, you can close the detail view.

Adding Animations

We are a little bit closer to the final product, but there are a few things we still need to take care of. Were you aware that the transition of the detail view was not animated? While we have the `.transition` modifier, the transition will only be animated when we pair it with an animation.

So, go back to `RestaurantDetailView.swift` and attach the `.animation` modifier to the `VStack` like this:

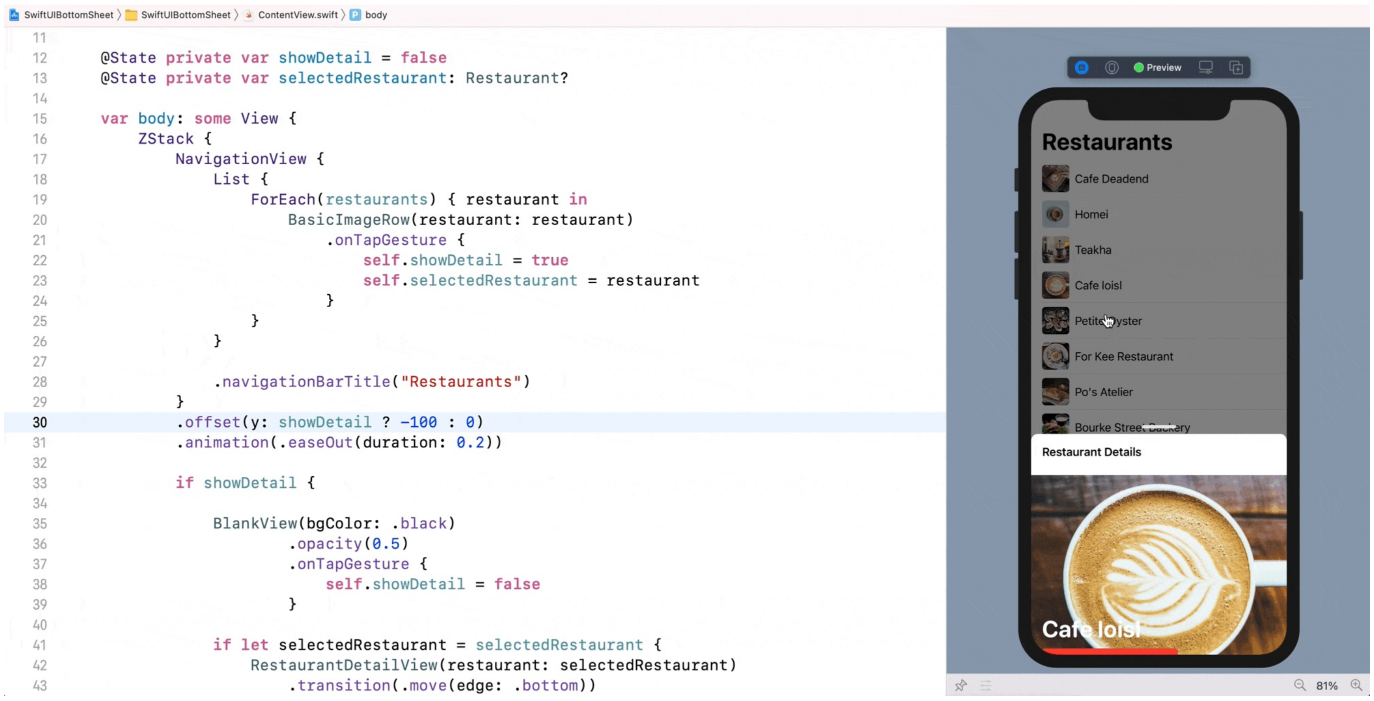
```
VStack {
    ...
}
.offset(y: geometry.size.height/2)
.animation(.interpolatingSpring(stiffness: 200.0, damping: 25.0, initialVelocity: 
10.0))
.edgesIgnoringSafeArea(.all)
```

In the code, we use the `interpolatingSpring` animation. The values of *stiffness*, *damping*, and *initialVelocity* can be changed. You can play around with the values to find out the best animation for your app.

I also want to add a subtle animation to the list view, such that it shifts a little bit upward when we bring up the detail view. Go back to `ContentView.swift`. Attach an `.offset` and `.animation` modifier to the navigation view:

```
NavigationView {
    List {
        ...
    }

    .navigationBarTitle("Restaurants")
}
.offset(y: showDetail ? -100 : 0)
.animation(.easeOut(duration: 0.2))
```

Now run the app in the preview canvas again. You should see a nice animated effect when the detail view is displayed.


SwiftUIBottomSheet 〉 SwiftUIBottomSheet 〉 ContentView.swift 〉 body

```swift
    @State private var showDetail = false
    @State private var selectedRestaurant: Restaurant?

    var body: some View {
        ZStack {
            NavigationView {
                List {
                    ForEach(restaurants) { restaurant in
                        BasicImageRow(restaurant: restaurant)
                            .onTapGesture {
                                self.showDetail = true
                                self.selectedRestaurant = restaurant
                            }
                    }
                }

                .navigationBarTitle("Restaurants")
            }
            .offset(y: showDetail ? -100 : 0)
            .animation(.easeOut(duration: 0.2))

            if showDetail {

                BlankView(bgColor: .black)
                    .opacity(0.5)
                    .onTapGesture {
                        self.showDetail = false
                    }

                if let selectedRestaurant = selectedRestaurant {
                    RestaurantDetailView(restaurant: selectedRestaurant)
                        .transition(.move(edge: .bottom))
```

Figure 9. Adding animations to the bottom sheet

Adding Gesture Support

Now that we have a half-baked bottom sheet, the next step is to make it expandable with gesture support. As mentioned at the very beginning of the chapter, users can slide the view up to expand its size. Or slide it down to minimize or dismiss it.

Since you've learned how drag gesture works in the previous chapter, we will apply a similar technique to create the expandable detail view. The implementation of the dragging gesture of this expandable bottom sheet is more complicated than before.

In `RestaurantDetailView.swift`, first define an Enum to represent the drag state:

```
enum DragState {
    case inactive
    case pressing
    case dragging(translation: CGSize)

    var translation: CGSize {
        switch self {
        case .inactive, .pressing:
            return .zero
        case .dragging(let translation):
            return translation
        }
    }

    var isDragging: Bool {
        switch self {
        case .pressing, .dragging:
            return true
        case .inactive:
            return false
        }
    }

}
```

Furthermore, declare a gesture state variable to keep track of the drag and a state variable to store the position offset of the bottom sheet:

```
@GestureState private var dragState = DragState.inactive
@State private var positionOffset: CGFloat = 0.0
```

To recognize the drag, we can attach the `.gesture` modifier to the `VStack` :

```
.gesture(DragGesture()
    .updating(self.$dragState, body: { (value, state, transaction) in
        state = .dragging(translation: value.translation)
        })
)
```

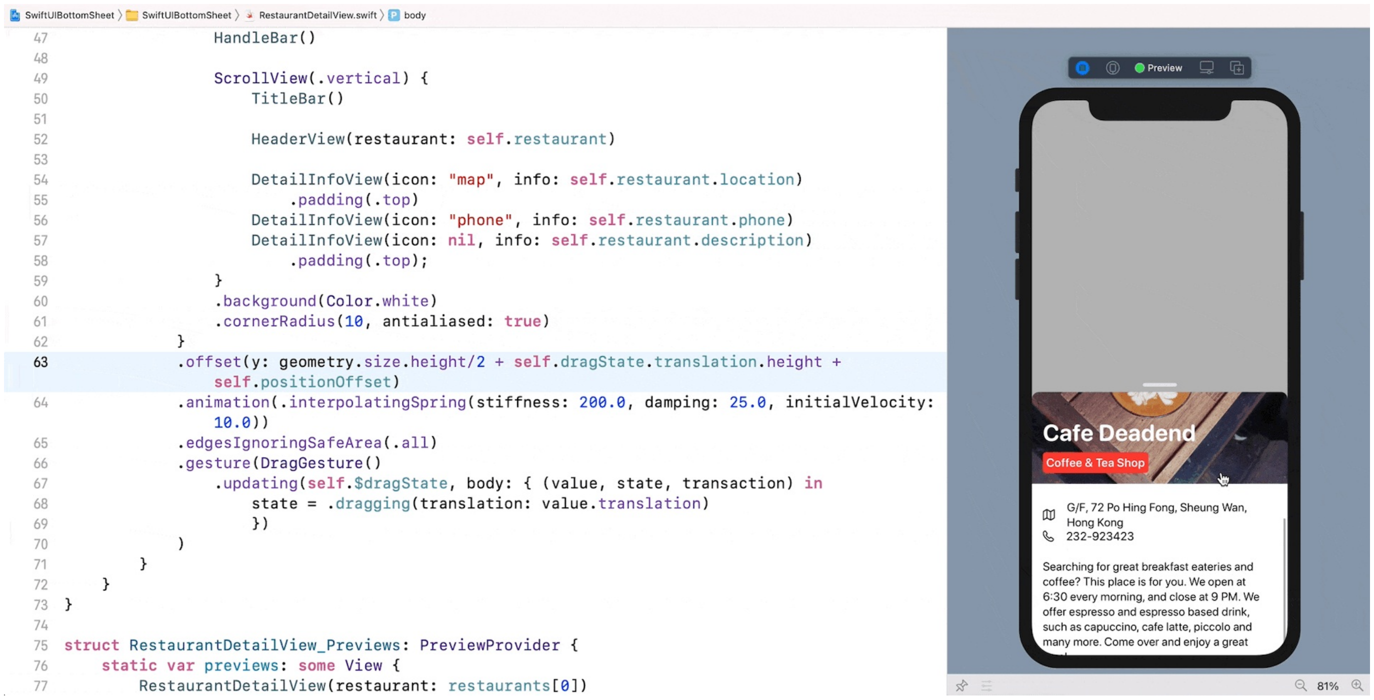
In the `updating` function, we simply update the `dragState` property with the latest drag data.

Finally, modify the `.offset` modifier of the `VStack` like this to move the detail view:

```
.offset(y: geometry.size.height/2 + self.dragState.translation.height + self.posit
ionOffset)
```

Instead of a fixed value, the drag's translation and the position offset will be taken into account for calculating the offset. This is how we enable the detail view to support the drag gesture.

If you test the detail view in the preview canvas, you should be able to slide the view by dragging the handlebar. Select the image (content view) and drag upwards, it's nearly impossible to slide up/down the view by dragging the content view.


SwiftUIBottomSheet 〉 SwiftUIBottomSheet 〉 RestaurantDetailView.swift 〉 body
Preview
81%

```
                HandleBar()

                ScrollView(.vertical) {
                    TitleBar()

                    HeaderView(restaurant: self.restaurant)

                    DetailInfoView(icon: "map", info: self.restaurant.location)
                        .padding(.top)
                    DetailInfoView(icon: "phone", info: self.restaurant.phone)
                    DetailInfoView(icon: nil, info: self.restaurant.description)
                        .padding(.top);
                }
                .background(Color.white)
                .cornerRadius(10, antialiased: true)
            }
            .offset(y: geometry.size.height/2 + self.dragState.translation.height +
                self.positionOffset)
            .animation(.interpolatingSpring(stiffness: 200.0, damping: 25.0, initialVelocity:
                10.0))
            .edgesIgnoringSafeArea(.all)
            .gesture(DragGesture()
                .updating(self.$dragState, body: { (value, state, transaction) in
                    state = .dragging(translation: value.translation)
                })
            )
        }
    }
}

struct RestaurantDetailView_Previews: PreviewProvider {
    static var previews: some View {
        RestaurantDetailView(restaurant: restaurants[0])
```

Cafe Deadend
Coffee & Tea Shop
G/F, 72 Po Hing Fong, Sheung Wan, Hong Kong
232-923423
Searching for great breakfast eateries and coffee? This place is for you. We open at 6:30 every morning, and close at 9 PM. We offer espresso and espresso based drink, such as capuccino, cafe latte, piccolo and many more. Come over and enjoy a great

Figure 10. Dragging the content part of the detail view won't slide up the view

Why couldn't we drag the content view to expand the detail view?

Don't forget, the content part of the detail view is embedded in a scroll view. Therefore, we actually have two gesture recognizers here: the one built into the scroll view and the drag gesture we added.

So, how do we fix it? One way is to disable the user interaction with the scroll view. You can add the `.disabled` modifier to the scroll view and set its value to `true` :

```
.disabled(true)
```

Once you attach this modifier to `ScrollView` , you will be able to slide up/down the detail view by dragging the content part.

But here comes to the next question. Users can't interact with the scroll view when it's disabled. That means the user can't view the full content of the restaurant. Therefore, we still need to make the content part scrollable.

Obviously, we need to figure out a way to control the enablement of the scroll view. A simple solution is to disable the scroll view when it's in half open state. After the detail view is fully opened, we enable the scrolling again.

The other problem with the current implementation is that the detail view won't stay fully open even if the user slides the view all the way up to the status bar. It just bounces back to the half-open state when the drag ends. Additionally, you can't dismiss the detail view when you slide it down to the end of the screen.

How do we tackle these problems?

Handling the Half-opened State

Let's first handle the issue of the half-opened state. This is the default state when the detail view is brought up. In this state, there are two scenarios that can happen:

1. The user chooses to slide up the view to make it fully open. But the user may drag the view a bit upward and then drag it down to cancel the action.

2. Alternatively, the user can choose to slide the view down to dismiss it. Again, the user may drag it back to up to cancel the dismissal.

As you can see from these scenarios, other than keeping track of the drag offset, we need some kind of threshold to control the opening and dismissal of the view.

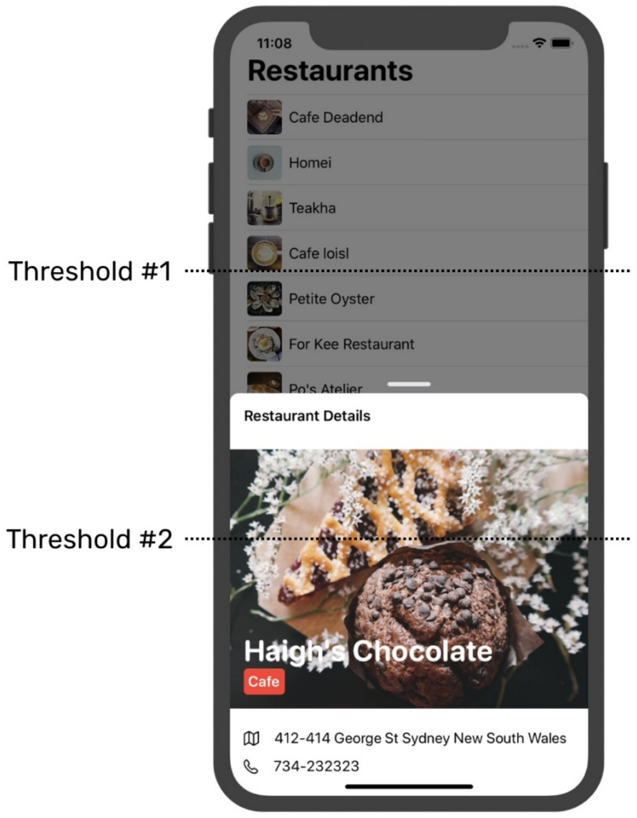


Figure 11. Adding thresholds to control the dragging and view state

The figure above shows you the thresholds we are going to define for the half-opened state:

- **Threshold #1** - once the drag goes beyond this threshold, the detail view becomes fully open.
- **Threshold #2** - when the drag moves lower than this threshold, the detail view will be dismissed.

Now that you understand how the half opened view should work, let's move onto the coding part. First, we declare another Enum in `RestaurantDetailView.swift` to represent these two view states:

```
enum ViewState {
    case full
    case half
}
```

In `RestaurantDetailView`, declare another state property to store the current view state. By default, it's set to half open:

```
@State private var viewState = ViewState.half
```

Also, in order to dismiss the view itself, we need the `ContentView` to pass us the binding of its state variable. So, declare the `isShow` binding variable like this:

```
@Binding var isShow: Bool
```

> Obviously, we need to figure out a way to control the enablement of the scroll view. A simple solution is to disable the scroll view when it's in half open state. After the detail view is fully opened, we enable the scrolling again.

Let's fix the scrolling issue of the scroll view. Attach the `.disabled` modifier to the `ScrollView` like this:

```
.disabled(self.viewState == .half)
```

Here, we disable the user interaction with the scroll view when the detail view is in half opened state.

Now, let's implement the thresholds that we just discussed. As you've learned before, when the drag ends, SwiftUI automatically calls the `onEnded` function. So, we will handle the thresholds in this function. Now update the `.gesture` modifier like this:

```
.gesture(DragGesture()
    .updating(self.$dragState, body: { (value, state, transaction) in
        state = .dragging(translation: value.translation)
        })
    .onEnded({ (value) in

        if self.viewState == .half {
            // Threshold #1
            // Slide up and when it goes beyond the threshold
            // change the view state to fully opened by updating
            // the position offset
            if value.translation.height < -geometry.size.height * 0.25 {
                self.positionOffset = -geometry.size.height/2 + 50
                self.viewState = .full
            }

            // Threshold #2
            // Slide down and when it goes pass the threshold
            // dismiss the view by setting isShow to false
            if value.translation.height > geometry.size.height * 0.3 {
                self.isShow = false
            }
        }

    })
)
```

I compute the threshold by using the screen height. For instance, threshold #2 is set to one-third of the screen height. This is just a sample value. You may alter it you desire.

Take a look at the code comments if you need further explanation about how the code works.

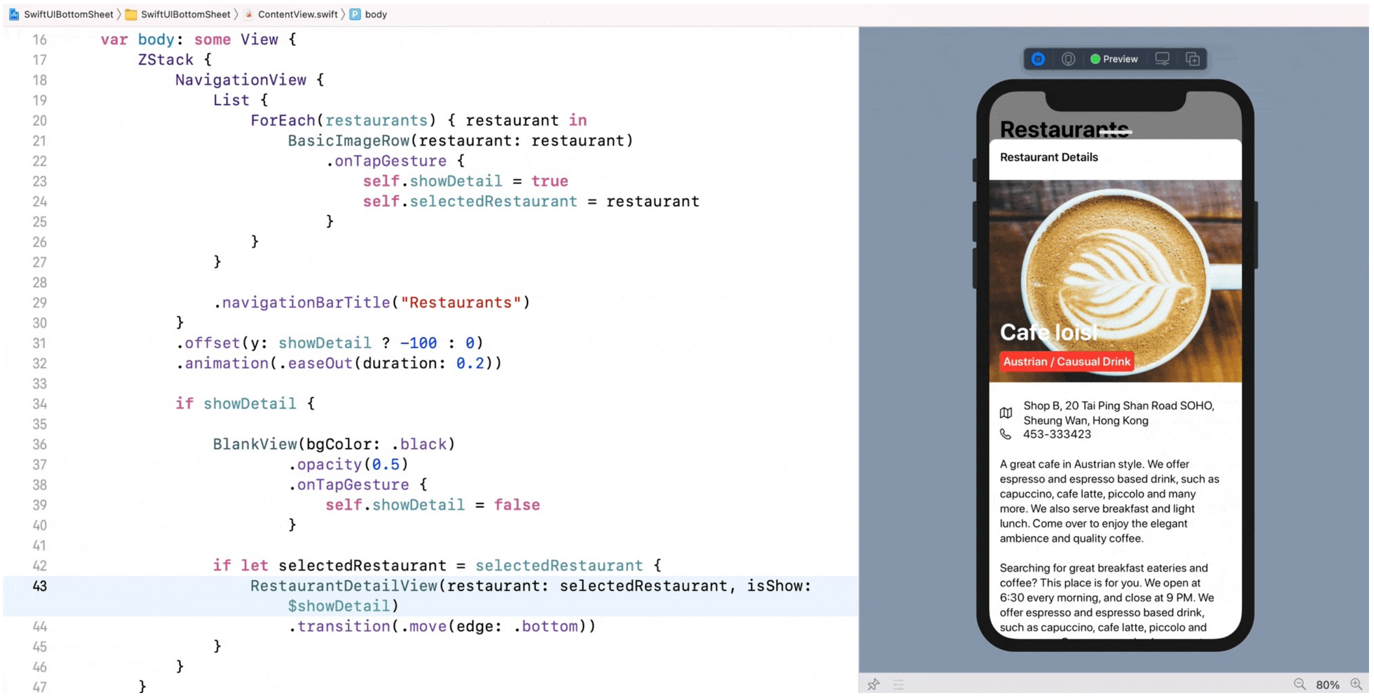
Since, we added a binding variable in `RestaurantDetailView`, we have to update the code of `RestaurantDetailView_Previews`:

```swift
struct RestaurantDetailView_Previews: PreviewProvider {
    static var previews: some View {
        RestaurantDetailView(restaurant: restaurants[0], isShow: .constant(true))
            .background(Color.black.opacity(0.3))
            .edgesIgnoringSafeArea(.all)
    }
}
```

Similarly, you need to go back to `ContentView.swift` to make the change according:

```swift
if let selectedRestaurant = selectedRestaurant {
    RestaurantDetailView(restaurant: selectedRestaurant, isShow: $showDetail)
        .transition(.move(edge: .bottom))
}
```

After all this hard work, it's time to test out the expandable detail view. Run the app in the simulator or preview canvas, you should be able to drag down the detailed view to dismiss it. Or drag it up to open it fully.


SwiftUIBottomSheet 〉 SwiftUIBottomSheet 〉 ContentView.swift 〉 body
Preview
80%

```swift
    var body: some View {
        ZStack {
            NavigationView {
                List {
                    ForEach(restaurants) { restaurant in
                        BasicImageRow(restaurant: restaurant)
                            .onTapGesture {
                                self.showDetail = true
                                self.selectedRestaurant = restaurant
                            }
                    }
                }

                .navigationBarTitle("Restaurants")
            }
            .offset(y: showDetail ? -100 : 0)
            .animation(.easeOut(duration: 0.2))

            if showDetail {

                BlankView(bgColor: .black)
                    .opacity(0.5)
                    .onTapGesture {
                        self.showDetail = false
                    }

                if let selectedRestaurant = selectedRestaurant {
                    RestaurantDetailView(restaurant: selectedRestaurant, isShow:
                        $showDetail)
                        .transition(.move(edge: .bottom))
                }
            }
        }
```

Restaurants

Restaurant Details

Cafe loisl

Austrian / Causual Drink

Shop B, 20 Tai Ping Shan Road SOHO, Sheung Wan, Hong Kong
453-333423

A great cafe in Austrian style. We offer espresso and espresso based drink, such as capuccino, cafe latte, piccolo and many more. We also serve breakfast and light lunch. Come over to enjoy the elegant ambience and quality coffee.

Searching for great breakfast eateries and coffee? This place is for you. We open at 6:30 every morning, and close at 9 PM. We offer espresso and espresso based drink, such as capuccino, cafe latte, piccolo and

Figure 12. Drag the detail view upward to fully open it

Handling the Fully Open State

Now that you are able to drag down the detailed view to dismiss it or drag the view up to open it fully. It works pretty well but there is an issue that the scroll view overrides our drag gesture when the view is fully open. In other words, you can scroll through the content but can't revert the view back to the half opened state.

Before we fix this scrolling issue, let's first fix a minor issue with the detailed view. You may notice that when the detailed view is in the fully open state, you can't scroll to the end of the description. To resolve this issue, open `RestaurantDetailView.swift` and update the `DetailInfoView` for displaying the restaurant description like this:

```
DetailInfoView(icon: nil, info: self.restaurant.description)
    .padding(.top)
    .padding(.bottom, 100)
```

We simply attach an additional `.padding` modifier to increase the empty space. This will allow you to scroll through the description properly.

Now let's go back to the scrolling issue. How can we let users scroll through the content and revert the view to half open by using the drag gesture? The user drags the view up to reveal the content. Conversely, the user will drag the view down when he/she wants to dismiss the detailed view. Right now, you can drag down the scroll view but the problem is that it will bounce back to the top when you release your finger.

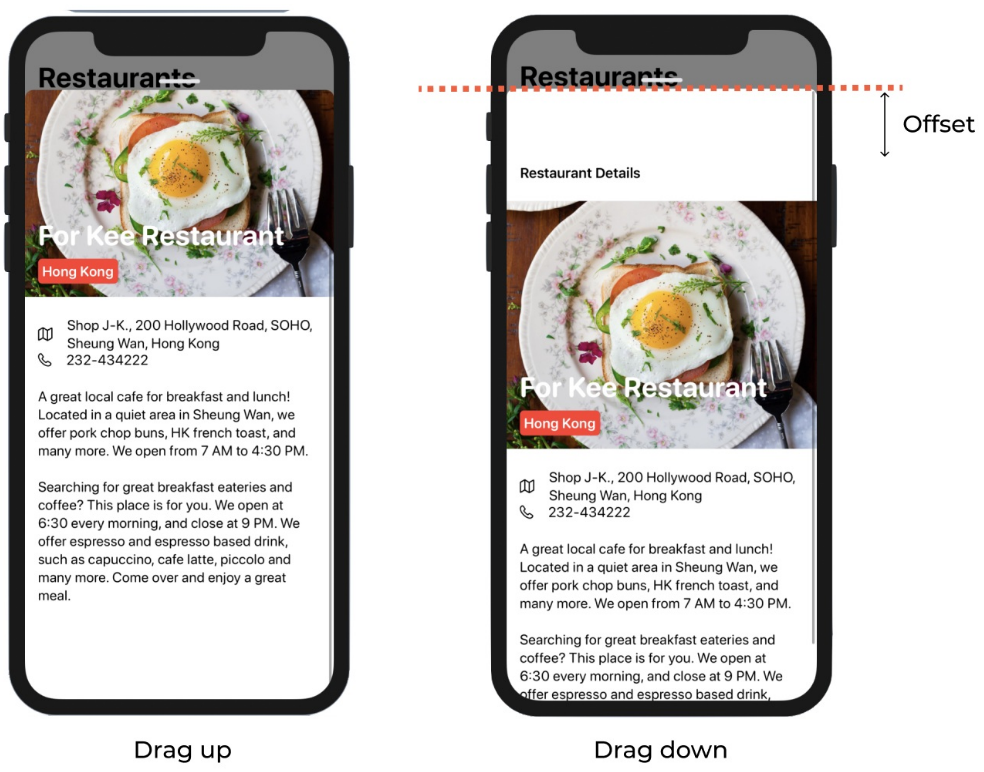

Restaurants
For Kee Restaurant
Hong Kong
Shop J-K., 200 Hollywood Road, SOHO, Sheung Wan, Hong Kong
232-434222
A great local cafe for breakfast and lunch! Located in a quiet area in Sheung Wan, we offer pork chop buns, HK french toast, and many more. We open from 7 AM to 4:30 PM.
Searching for great breakfast eateries and coffee? This place is for you. We open at 6:30 every morning, and close at 9 PM. We offer espresso and espresso based drink, such as capuccino, cafe latte, piccolo and many more. Come over and enjoy a great meal.
Restaurant Details
Offset
Drag up
Drag down

Figure 13. Dragging the scroll view in both directions

Since the scroll view blocks the drag gesture we attached, we have to find an alternate way to detect this "drag down" gesture. You've learned how to use `GeometryReader` to measure the size of a view. It can also be used to find out the scroll offset as indicated in figure 13.

In `RestaurandDetailView`, add the following code inside `ScrollView` and place it right above `TitleBar()`:

```
GeometryReader { scrollViewProxy in
    Text("\(scrollViewProxy.frame(in: .named("scrollview")).minY)")
}
.frame(height: 0)
```

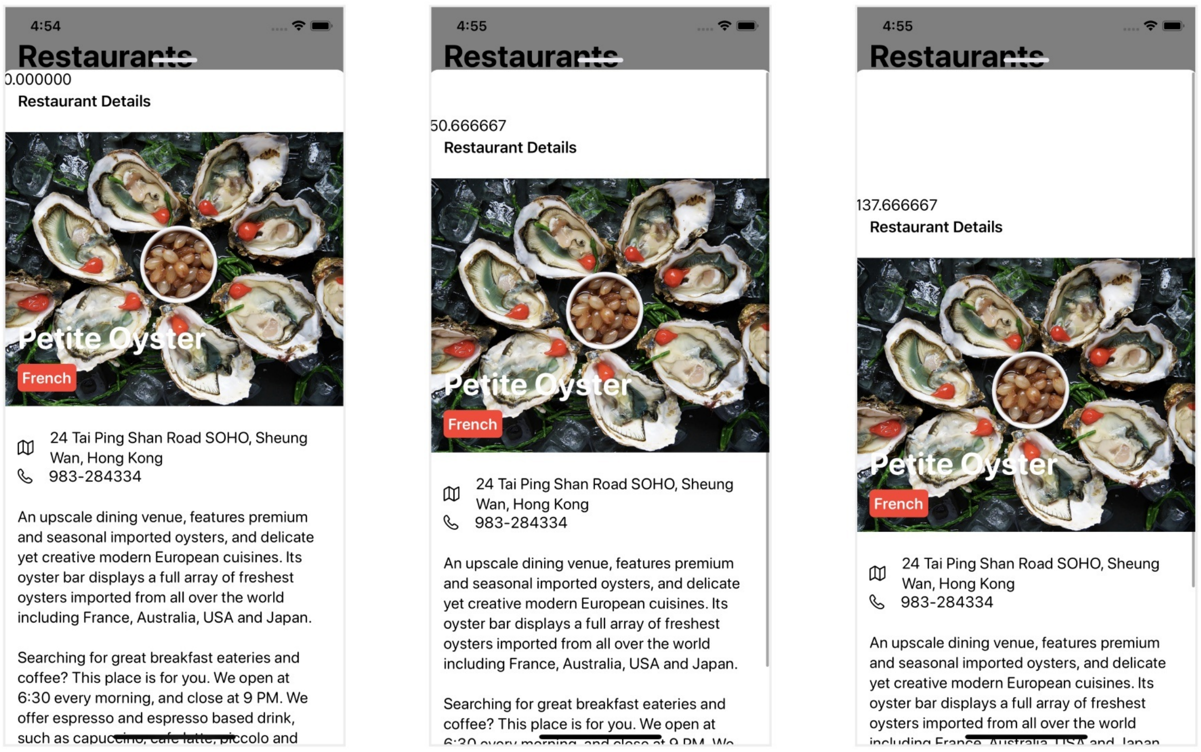
This is not the final code. I just want to show you how to use `GeometryReader` to read the scroll offset. In the closure of `GeometryReader`, we use the `scrollViewProxy` to figure out the offset by calling the `frame` function and retrieve the value of `minY`. When calling the `frame` function, you have to pass it your preferred coordinate space. Here, we define our own coordinate space, which is confined to the scroll view.

In order to define our own coordinate space, attach the `.coordinateSpace` modifer to the `ScrollView`:

```
.coordinateSpace(name: "scrollview")
```

You may wonder why we have to use our own coordinate space instead of `.global`. Take a look at figure 13 again. The dotted red line is the reference point, which should have an offset value of zero. By defining the coordinate space confined to the scroll view, we can make that possible.

Now run the app in a simulator. Bring up the detail view and make it fully open. When you drag the view down, you should see the drag offset. As you can see in figure 14, the offset increases as you drag farther from the top bar.



24 Tai Ping Shan Road SOHO, Sheung Wan, Hong Kong
983-284334

An upscale dining venue, features premium and seasonal imported oysters, and delicate yet creative modern European cuisines. Its oyster bar displays a full array of freshest oysters imported from all over the world including France, Australia, USA and Japan.

Searching for great breakfast eateries and coffee? This place is for you. We open at 6:30 every morning, and close at 9 PM. We offer espresso and espresso based drink, such as capuccino, cafe latte, piccolo and

24 Tai Ping Shan Road SOHO, Sheung Wan, Hong Kong
983-284334

An upscale dining venue, features premium and seasonal imported oysters, and delicate yet creative modern European cuisines. Its oyster bar displays a full array of freshest oysters imported from all over the world including France, Australia, USA and Japan.

Searching for great breakfast eateries and coffee? This place is for you. We open at

24 Tai Ping Shan Road SOHO, Sheung Wan, Hong Kong
983-284334

An upscale dining venue, features premium and seasonal imported oysters, and delicate yet creative modern European cuisines. Its oyster bar displays a full array of freshest oysters imported from all over the world

Figure 14. Dragging the scroll view in both directions

Introducing PreferenceKey

Now that we have figured out a way to retrieve the scroll offset, the next question is how to let its parent views know about the offset for further processing. The SwiftUI framework provides a protocol called `PreferenceKey` which allows you to easily pass data from child views to its ancestors.

In order to pass the scroll offset using preferences, we have to create a struct conforming to the `PreferenceKey` protocol like below. Insert the following code snippet in `RestaurantDetailView.swift`:

```
struct ScrollOffsetKey: PreferenceKey {
    typealias Value = CGFloat

    static var defaultValue = CGFloat.zero

    static func reduce(value: inout Value, nextValue: () -> Value) {
        value += nextValue()
    }
}
```

The protocol has two requirements. First, you have to define the default value, which is zero for our implementation. Second, you need to implement the `reduce` function to combine the offset values into one.

Next, modify the `GeometryReader` created in the previous section like this:

```
GeometryReader { scrollViewProxy in
    Color.clear.preference(key: ScrollOffsetKey.self, value: scrollViewProxy.frame(
in: .named("scrollview")).minY)
}
.frame(height: 0)
```

In the code, we retrieve the scroll offset and save it into the preference key. We use the `Color.clear` view to make the view invisible to the users.

The next question is how can you retrieve the scroll offset from the preference?

One simple way is to use the `.onPreferenceChange` modifier to observe the value change. You can attach the modifier to the `VStack` and place it under the `.gesture` modifier like this:

```
.onPreferenceChange(ScrollOffsetKey.self) { value in
    print("\(value)")
}
```

Here we simply print the value of the offset. When you run the app in a simulator, you should see the offset value in the console while dragging the detail view.

Now that you have some basic idea how PreferenceKey works, let's finish the implementation and make the detail view return to the half open state.

Declare a new state variable to keep track of the scroll offset:

```
@State private var scrollOffset: CGFloat = 0.0
```

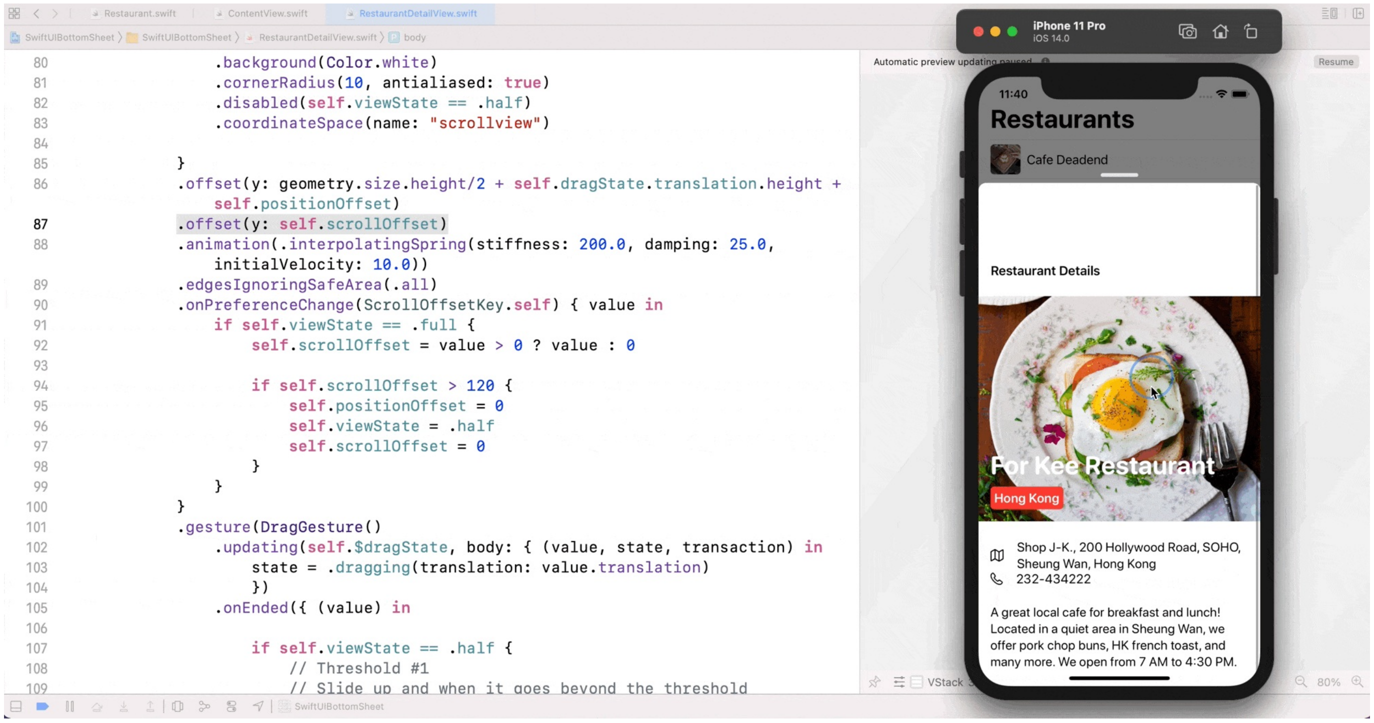
Next, update the `.onPreferenceChange` modifer like this:

```
.onPreferenceChange(ScrollOffsetKey.self) { value in
    if self.viewState == .full {
        self.scrollOffset = value > 0 ? value : 0

        if self.scrollOffset > 120 {
            self.positionOffset = 0
            self.viewState = .half
            self.scrollOffset = 0
        }
    }
}
```

When there is a change of the scroll offset, we check if it exceeds our preset threshold (i.e. 120). If it's true, we set the view state back to `.half`. Now attach another `.offset` modifier to the `VStack`:

```
.offset(y: self.scrollOffset)
```

The purpose of this additional `.offset` modifier is to move the detail view down. If you run the app on a simulator, you should be able to drag the fully open view down to revert it back to the half open state.



```
                .background(Color.white)
                .cornerRadius(10, antialiased: true)
                .disabled(self.viewState == .half)
                .coordinateSpace(name: "scrollview")

            }
            .offset(y: geometry.size.height/2 + self.dragState.translation.height +
                self.positionOffset)
            .offset(y: self.scrollOffset)
            .animation(.interpolatingSpring(stiffness: 200.0, damping: 25.0,
                initialVelocity: 10.0))
            .edgesIgnoringSafeArea(.all)
            .onPreferenceChange(ScrollOffsetKey.self) { value in
                if self.viewState == .full {
                    self.scrollOffset = value > 0 ? value : 0

                    if self.scrollOffset > 120 {
                        self.positionOffset = 0
                        self.viewState = .half
                        self.scrollOffset = 0
                    }
                }
            }
            .gesture(DragGesture()
                .updating(self.$dragState, body: { (value, state, transaction) in
                    state = .dragging(translation: value.translation)
                    })
                .onEnded({ (value) in

                    if self.viewState == .half {
                        // Threshold #1
                        // Slide up and when it goes beyond the threshold
```

Shop J-K., 200 Hollywood Road, SOHO, Sheung Wan, Hong Kong
232-434222

A great local cafe for breakfast and lunch! Located in a quiet area in Sheung Wan, we offer pork chop buns, HK french toast, and many more. We open from 7 AM to 4:30 PM.

Figure 15. Sliding down the view when it's in fully open state

You may notice that there is a minor bounce effect when you drag the scroll view down. To resolve the issue, you can wrap the subviews inside the scroll view with a vertical stack and attach an `.offset` modifier to it.

```
VStack {
    TitleBar()

    HeaderView(restaurant: self.restaurant)

    DetailInfoView(icon: "map", info: self.restaurant.location)
        .padding(.top)
    DetailInfoView(icon: "phone", info: self.restaurant.phone)
    DetailInfoView(icon: nil, info: self.restaurant.description)
        .padding(.top)
        .padding(.bottom, 100)
}
.offset(y: -self.scrollOffset)
.animation(nil)
```

Summary

This is a huge chapter and I hope you enjoyed it. In this chapter, we utilized everything you have learned so far to build a expandable bottom sheet. I tried my best to document my thought process on tackling issues I discovered when I built the sheet. I really hope this helps you understand my solution and code.

One of the advantages of SwiftUI is that it encourages you to build modular UI components. We haven't done it yet. But, as you may discover, it is pretty easy to turn this restaurant detail view into a generic bottom sheet that supports various types of content by using the techniques that I covered in the previous chapter.

Try to build the generic bottom sheet by yourself and make it a reusable component for your projects.

For reference, you can download the complete bottom sheet project here:

- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUIBottomSheet.zip)

Chapter 19
Creating a Tinder-like UI with Gestures and Animations

Wasn't it fun to build an expandable bottom sheet? Let's continue to apply what we learned about gestures to a real-world project. I'm not sure if you've used the Tinder app before. But you've probably come across a Tinder-like user interface in other apps. The swiping motion is central to Tinder's UI design and has become one of the most popular mobile UI patterns. Users swipe right to like a photo or swipe left to dislike it.

What we are going to do in this chapter is to build a simple app with a Tinder-like UI. The app presents users with a deck of travel cards and allows them to use the swipe gesture to like/dislike a card.

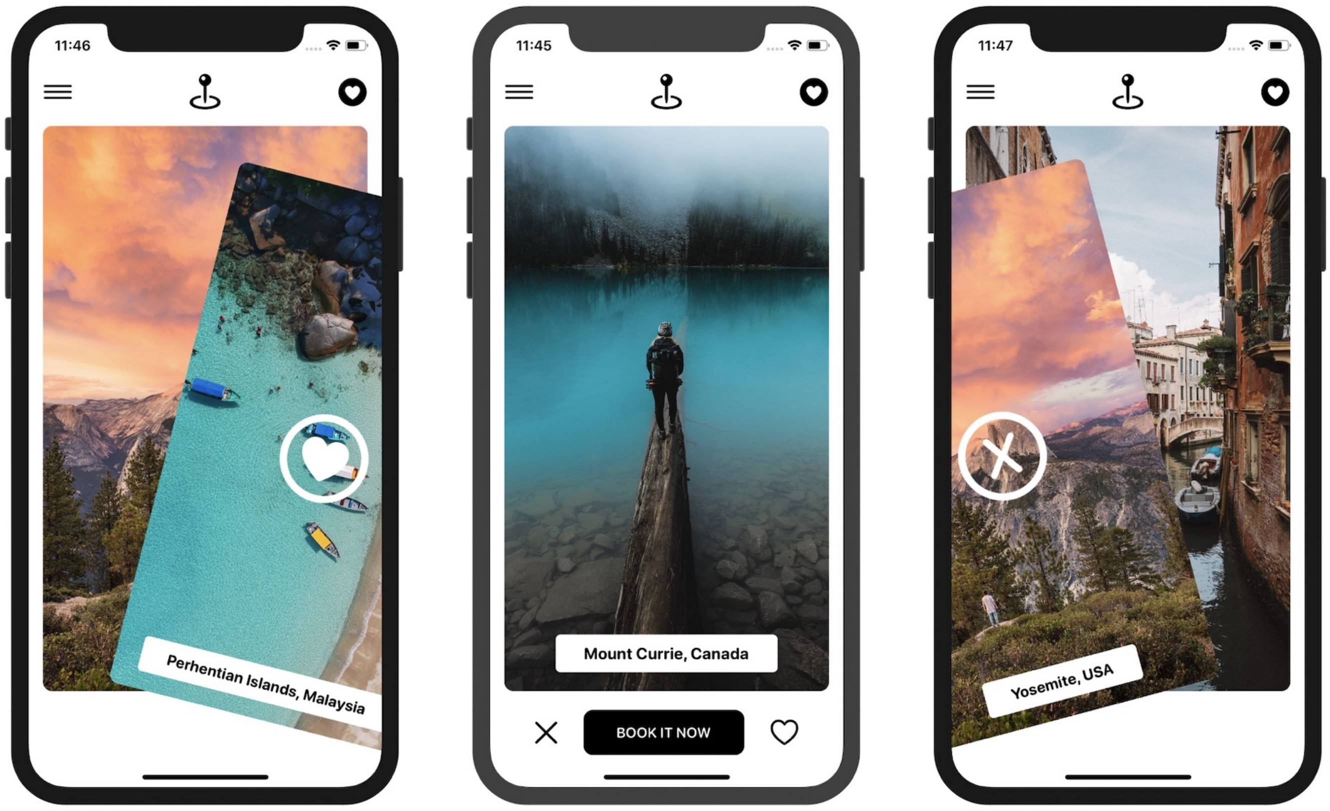
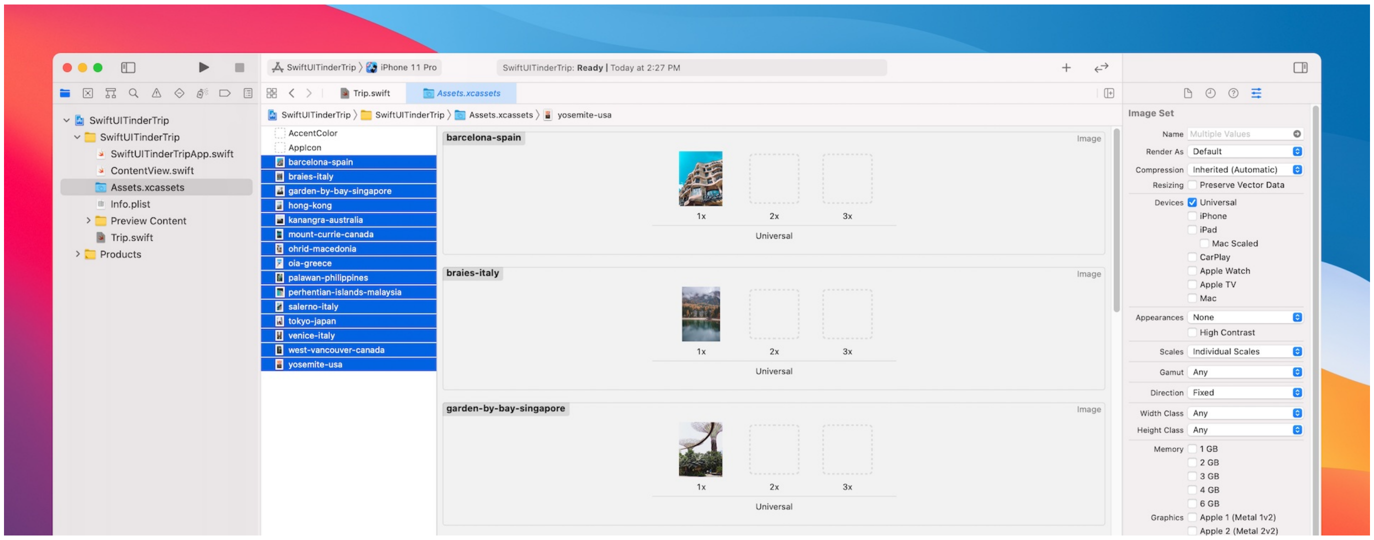


Figure 1. Building a tinder-like user interface

Note that we are not going to build a fully functional app but focus only on the Tinder-like UI.

Project Preparation

It would be great if you want to use your own images. But to save you time preparing trip images, I have created a starter project for you. You can download it from https://www.appcoda.com/resources/swiftui2/SwiftUITinderTripStarter.zip. This project comes with a set of photos for the travel cards.



Name: Multiple Values
Render As: Default
Compression: Inherited (Automatic)
Resizing: Preserve Vector Data
Devices: ✓ Universal, iPhone, iPad, Mac Scaled, CarPlay, Apple Watch, Apple TV, Mac
Appearances: None
High Contrast
Scales: Individual Scales
Gamut: Any
Direction: Fixed
Width Class: Any
Height Class: Any
Memory: 1 GB, 2 GB, 3 GB, 4 GB, 6 GB
Graphics: Apple 1 (Metal 1v2), Apple 2 (Metal 2v2)

Figure 2. Preloaded with a set of travel photos

I have also prepared the test data for the demo app and created the `Trip.swift` file to represent a trip:

```
struct Trip {
    var destination: String
    var image: String
}

#if DEBUG
var trips = [ Trip(destination: "Yosemite, USA", image: "yosemite-usa"),
              Trip(destination: "Venice, Italy", image: "venice-italy"),
              Trip(destination: "Hong Kong", image: "hong-kong"),
              Trip(destination: "Barcelona, Spain", image: "barcelona-spain"),
              Trip(destination: "Braies, Italy", image: "braies-italy"),
              Trip(destination: "Kanangra, Australia", image: "kanangra-australia"
),
              Trip(destination: "Mount Currie, Canada", image: "mount-currie-canad
a"),
              Trip(destination: "Ohrid, Macedonia", image: "ohrid-macedonia"),
              Trip(destination: "Oia, Greece", image: "oia-greece"),
              Trip(destination: "Palawan, Philippines", image: "palawan-philippine
s"),
              Trip(destination: "Salerno, Italy", image: "salerno-italy"),
              Trip(destination: "Tokyo, Japan", image: "tokyo-japan"),
              Trip(destination: "West Vancouver, Canada", image: "west-vancouver-c
anada"),
              Trip(destination: "Singapore", image: "garden-by-bay-singapore"),
              Trip(destination: "Perhentian Islands, Malaysia", image: "perhentian
-islands-malaysia")
            ]
#endif
```

In case you prefer to use your own images and data, simply replace the images in the asset catalog and update `Trip.swift`.

Building the Card Views and Menu Bars

Before implementing the swipe feature, let's start by creating the main UI. I will break the main screen into three parts:

1. The top menu bar
2. The card view

3. The bottom menu bar

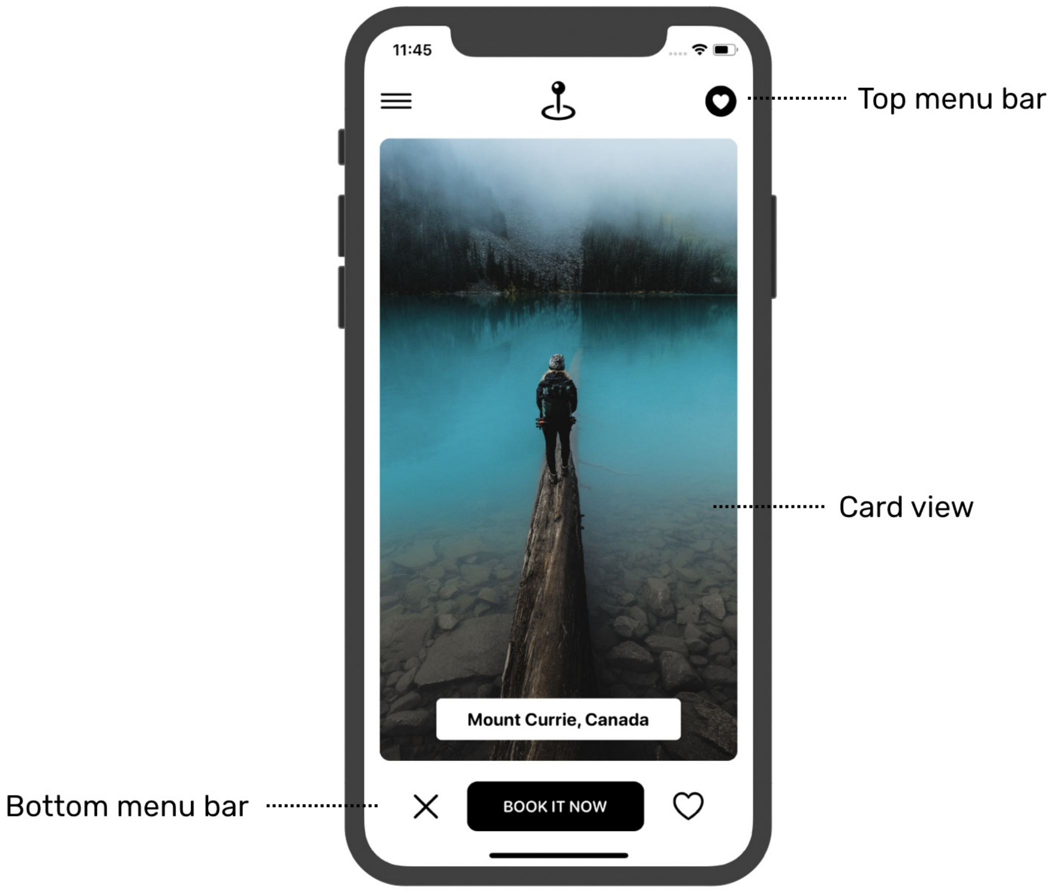


Figure 3. The main screen

Card View

First, let's create a card view. *If you want to challenge yourself, I highly recommend you stop here and implement it without following this section. Otherwise, keep reading.*

To better organize the code, we will implement the card view in a separate file. In the project navigator, create a new file using the *SwiftUI View* template and name it `CardView.swift`.

The `CardView` is designed to display different photos and titles. So, declare two variables for storing these data:

```
let image: String
let title: String
```

The main screen is going to display a deck of card views. Later, we will use `ForEach` to loop through an array of card views and present them. If you still remember the usage of `ForEach`, SwiftUI needs to know how to uniquely identify each item in the array. Therefore, we will make `CardView` conform to the `Identifiable` protocol and introduce an `id` variable like this:

```
struct CardView: View, Identifiable {
    let id = UUID()
    let image: String
    let title: String

    .
    .
    .
}
```

In case you forgot what the `Identifiable` protocol is, please refer to chapter 10.

Now let's continue to implement the card view and update the `body` variable like this:

```
var body: some View {
    Image(image)
        .resizable()
        .scaledToFill()
        .frame(minWidth: 0, maxWidth: .infinity)
        .cornerRadius(10)
        .padding(.horizontal, 15)
        .overlay(

            VStack {

                Text(title)
                    .font(.system(.headline, design: .rounded))
                    .fontWeight(.bold)
                    .padding(.horizontal, 30)
                    .padding(.vertical, 10)
                    .background(Color.white)
                    .cornerRadius(5)
            }
            .padding([.bottom], 20)

        , alignment: .bottom)

}
```

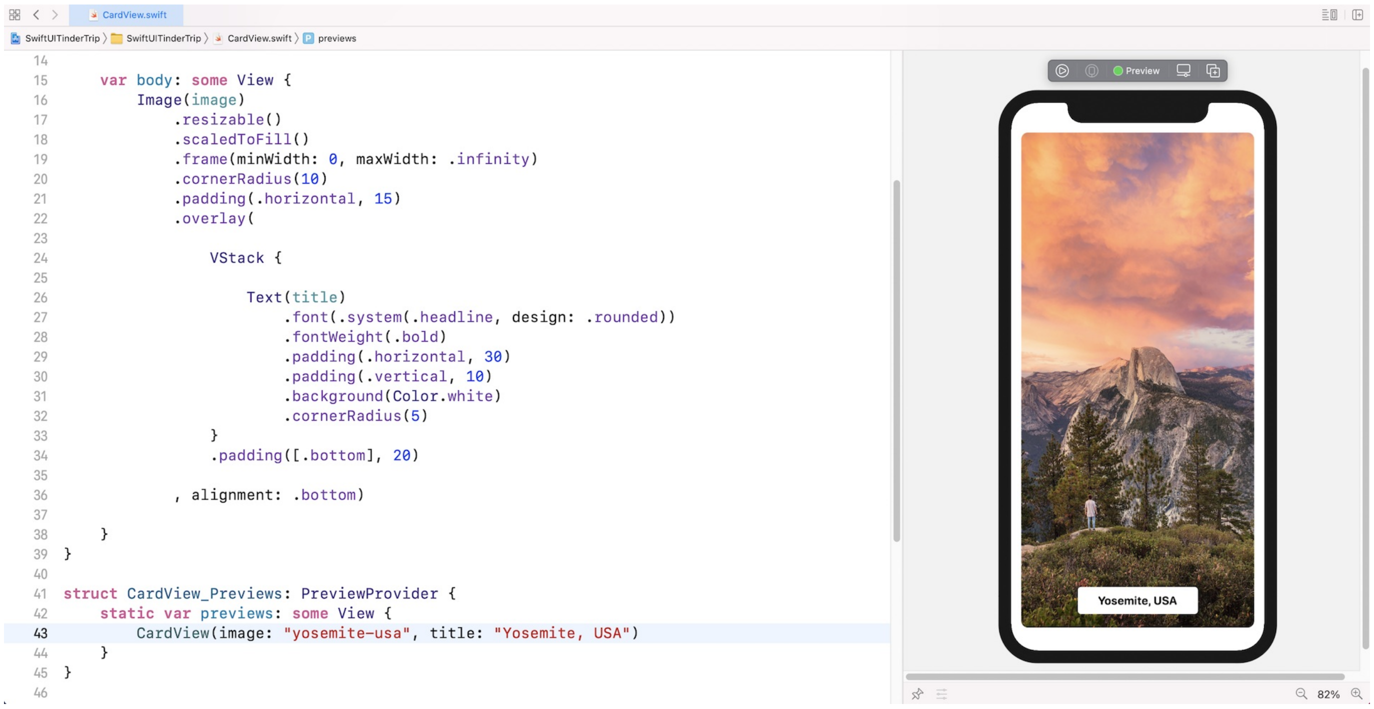
The card view is composed of an image and a text component, which is overlayed on top of the image. We set the image to the `scaleToFill` mode and round the corners by using the `cornerRadius` modifier. The text component is used to display the destination of the trip.

We have an in-depth discussion about a similar implementation of the card view in chapter 5. If you don't fully understand the code, please check out that chapter again.

You can't preview the card view yet because you have to provide the values of both `image` and `title` in the `CardView_Previews`. Therefore, update the `CardView_Previews` struct like this:

```
struct CardView_Previews: PreviewProvider {
    static var previews: some View {
        CardView(image: "yosemite-usa", title: "Yosemite, USA")
    }
}
```

I simply use one of the images in the asset catalog for preview purposes. You are free to alter the image and title to fit your own needs. In the preview canvas, you should now see the card view similar to figure 4.


CardView.swift
SwiftUITinderTrip 〉 SwiftUITinderTrip 〉 CardView.swift 〉 previews
Preview
Yosemite, USA
82%

```
    var body: some View {
        Image(image)
            .resizable()
            .scaledToFill()
            .frame(minWidth: 0, maxWidth: .infinity)
            .cornerRadius(10)
            .padding(.horizontal, 15)
            .overlay(

                VStack {

                    Text(title)
                        .font(.system(.headline, design: .rounded))
                        .fontWeight(.bold)
                        .padding(.horizontal, 30)
                        .padding(.vertical, 10)
                        .background(Color.white)
                        .cornerRadius(5)
                }
                .padding([.bottom], 20)

            , alignment: .bottom)

    }
}

struct CardView_Previews: PreviewProvider {
    static var previews: some View {
        CardView(image: "yosemite-usa", title: "Yosemite, USA")
    }
}
```

Figure 4. Previewing the card view

Menu Bars and Main UI

With the card view ready, we can move on to implementing the main UI. The main UI has the card and two menu bars. For both menu bars, I will create a separate `struct` for each of them.

Now open `ContentView.swift` and start the implementation. For the top bar menu, create a new `struct` like this:

```
struct TopBarMenu: View {
    var body: some View {
        HStack {
            Image(systemName: "line.horizontal.3")
                .font(.system(size: 30))
            Spacer()
            Image(systemName: "mappin.and.ellipse")
            .font(.system(size: 35))
            Spacer()
            Image(systemName: "heart.circle.fill")
            .font(.system(size: 30))
        }
        .padding()
    }
}
```

The three icons are arranged using a horizontal stack with equal spacing. For the bottom bar menu, the implementation is pretty much the same. Insert the following code in `ContentView.swift` to create the menu bar:

```
struct BottomBarMenu: View {
    var body: some View {
        HStack {
            Image(systemName: "xmark")
                .font(.system(size: 30))
                .foregroundColor(.black)

            Button(action: {
                // Book the trip
            }) {
                Text("BOOK IT NOW")
                    .font(.system(.subheadline, design: .rounded))
                .bold()
                    .foregroundColor(.white)
                    .padding(.horizontal, 35)
                    .padding(.vertical, 15)
                    .background(Color.black)
                    .cornerRadius(10)

            }
            .padding(.horizontal, 20)

            Image(systemName: "heart")
                .font(.system(size: 30))
                .foregroundColor(.black)
        }

    }
}
```

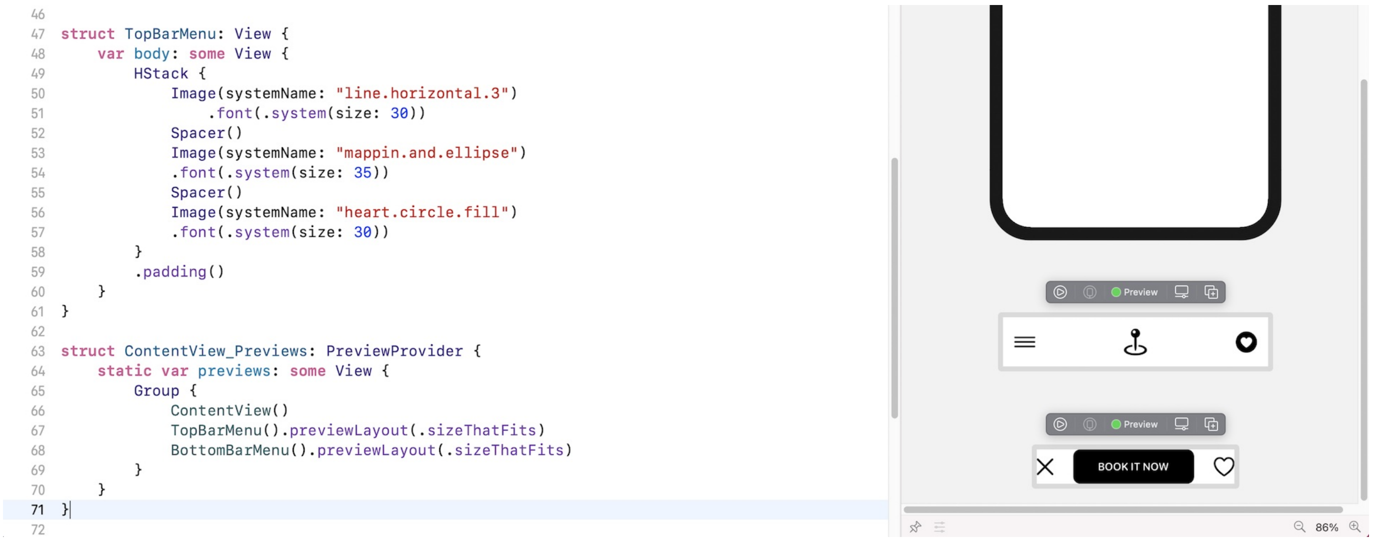
We are not going to implement the "Book Trip" feature, so the action block is left blank. The rest of the code should be self explanatory assuming you understand how stacks and images work.

Before building the main UI, let me show you a trick to preview these two menu bars. It's not mandatory to put these bars in the `ContentView` in order to preview their look and feel.

Now update the `ContentView_Previews` struct like this:

```
struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        Group {
            ContentView()
            TopBarMenu().previewLayout(.sizeThatFits)
            BottomBarMenu().previewLayout(.sizeThatFits)
        }
    }
}
```

Here we use `Group` to group the preview of multiple components. Without specifying any preview option (like `ContentView`), Xcode displays the preview on the current simulator. For both `TopBarMenu` and `BottomBarMenu`, we tell Xcode to preview the layout in a container view. Figure 5 gives you a better idea what the preview looks like.



```

struct TopBarMenu: View {
    var body: some View {
        HStack {
            Image(systemName: "line.horizontal.3")
                .font(.system(size: 30))
            Spacer()
            Image(systemName: "mappin.and.ellipse")
            .font(.system(size: 35))
            Spacer()
            Image(systemName: "heart.circle.fill")
            .font(.system(size: 30))
        }
        .padding()
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        Group {
            ContentView()
            TopBarMenu().previewLayout(.sizeThatFits)
            BottomBarMenu().previewLayout(.sizeThatFits)
        }
    }
}

```

Figure 5. Previewing the menu bars

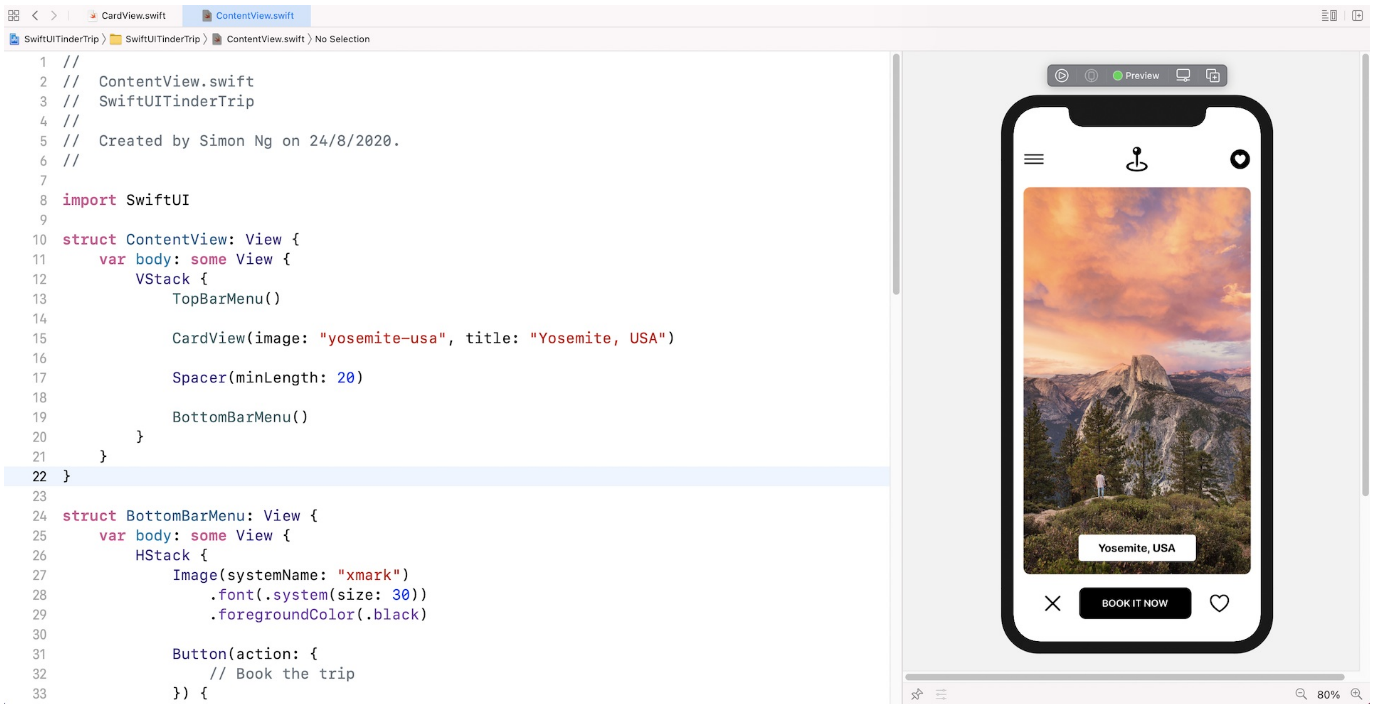
The `.sizeThatFits` option instructs Xcode to resize the container view to fit the content. Alternatively, you can update the preview code like below to create a fixed size container view:

```
TopBarMenu().previewLayout(.fixed(width: 375, height: 60))
BottomBarMenu().previewLayout(.fixed(width: 375, height: 60))
```

Okay, let's continue to lay out the main UI. Update the `ContentView` like this:

```
struct ContentView: View {
    var body: some View {
        VStack {
            TopBarMenu()

            CardView(image: "yosemite-usa", title: "Yosemite, USA")

            Spacer(minLength: 20)

            BottomBarMenu()
        }
    }
}
```

In the code, we simply arrange the UI components we have built using a `VStack`. Your preview should now show you the main screen.


CardView.swift
ContentView.swift
SwiftUITinderTrip 〉 SwiftUITinderTrip 〉 ContentView.swift 〉 No Selection
Preview
Yosemite, USA
BOOK IT NOW
80%

```swift
//
//  ContentView.swift
//  SwiftUITinderTrip
//
//  Created by Simon Ng on 24/8/2020.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        VStack {
            TopBarMenu()

            CardView(image: "yosemite-usa", title: "Yosemite, USA")

            Spacer(minLength: 20)

            BottomBarMenu()
        }
    }
}

struct BottomBarMenu: View {
    var body: some View {
        HStack {
            Image(systemName: "xmark")
                .font(.system(size: 30))
                .foregroundColor(.black)

            Button(action: {
                // Book the trip
            }) {
```

Figure 6. Previewing the main UI

Implementing the Card Deck

With all the preparation, we finally comes to the implementation of the Tinder-like UI. For those who haven't used the Tinder app before, let me first explain how a Tinder-like UI works.

You can imagine a Tinder-like UI as a deck of piled photo cards. For our demo app, the photo is a destination of a trip. Swiping the topmost card (i.e. the first trip) slightly to the left or right unveils the next card (i.e. the next trip) underneath. If the user releases the card, the app brings the card to the original position. But, when the user swipes hard enough, he/she can throw away the card and the app will bring the second card forward to become the topmost card.

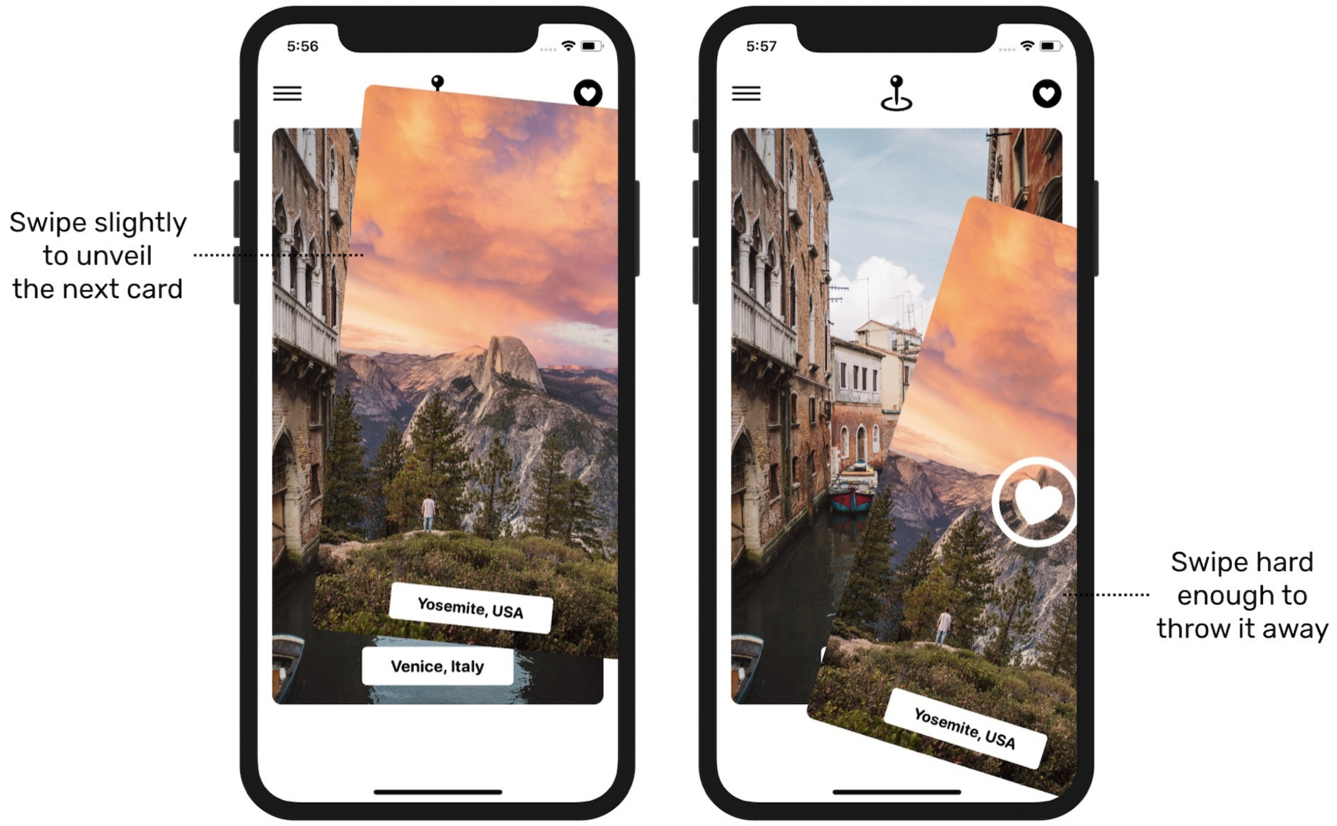


Figure 7. How the Tinder-like UI works

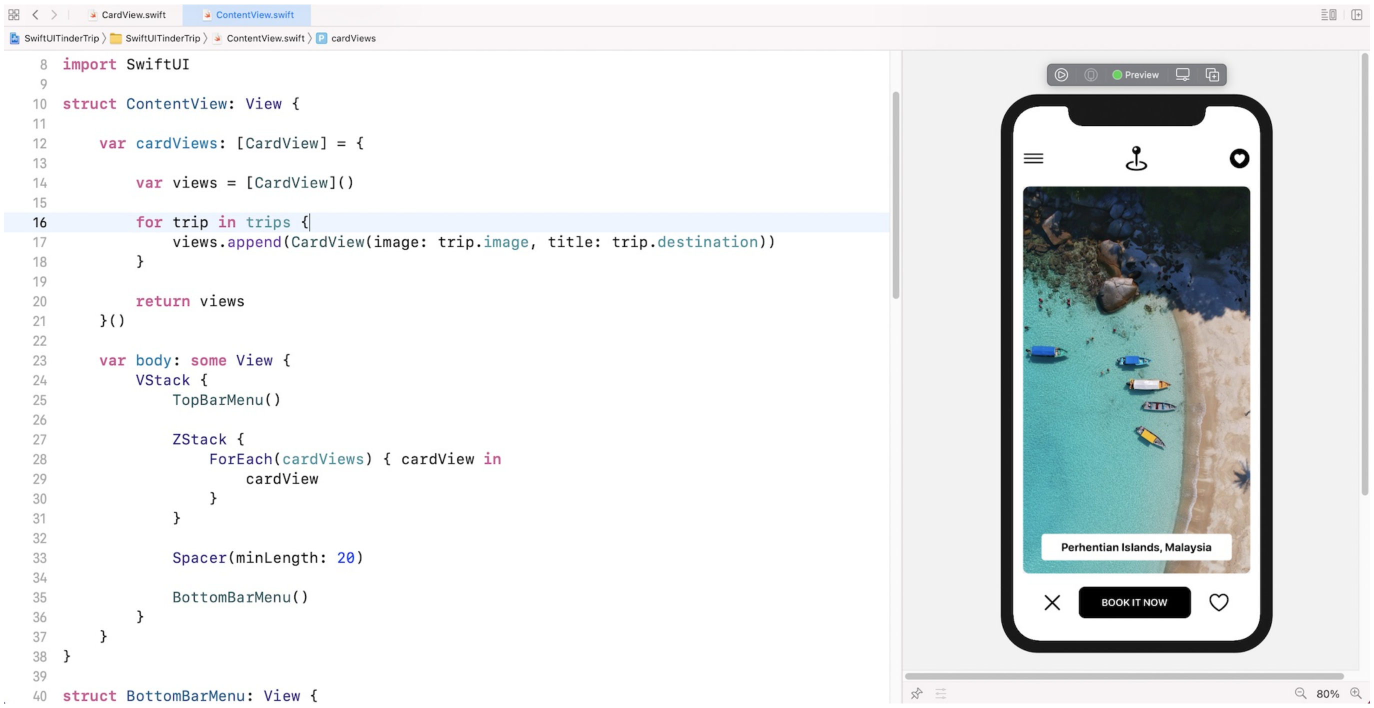
The main screen we have implemented only contains a single card view. So, how can we implement the pile of card views?

The most straightforward way is to overlay each of the card views on top of each other using a `ZStack`. Let's try to do this. Update the `ContentView` struct like this:

```
struct ContentView: View {

    var cardViews: [CardView] = {

        var views = [CardView]()

        for trip in trips {
            views.append(CardView(image: trip.image, title: trip.destination))
        }

        return views
    }()

    var body: some View {
        VStack {
            TopBarMenu()

            ZStack {
                ForEach(cardViews) { cardView in
                    cardView
                }
            }

            Spacer(minLength: 20)

            BottomBarMenu()
        }
    }
}
```

In the code above, we initialize an array of `cardViews` containing all the trips, which was defined in the `Trip.swift` file. In the `body` variable, we loop through all the card views and overlay one with another by wrapping them in a `ZStack`.

The preview canvas should show you the same UI but with another image.

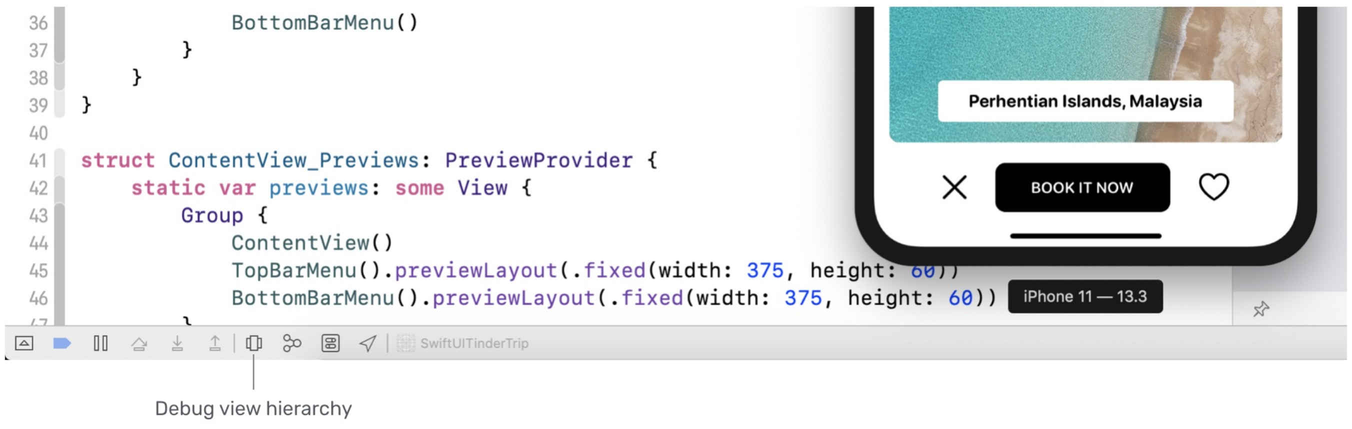


```swift
import SwiftUI

struct ContentView: View {

    var cardViews: [CardView] = {

        var views = [CardView]()

        for trip in trips {
            views.append(CardView(image: trip.image, title: trip.destination))
        }

        return views
    }()

    var body: some View {
        VStack {
            TopBarMenu()

            ZStack {
                ForEach(cardViews) { cardView in
                    cardView
                }
            }

            Spacer(minLength: 20)

            BottomBarMenu()
        }
    }
}

struct BottomBarMenu: View {
```

Figure 8. Building the deck of card views

Why did it display another image? If you refer to the `trips` array defined in `Trip.swift`, the image is the last element of the array. In the `ForEach` block, the first trip is placed at the lowermost part of the deck. Thus, the last trip becomes the topmost photo of the deck.

How do you make sure all the images are laid out since we can only see the last image? Instead of using the preview canvas, try to run the project in a simulator. After the app is launched, click the *Debug View Hierarchy* button.



```
            BottomBarMenu()
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        Group {
            ContentView()
            TopBarMenu().previewLayout(.fixed(width: 375, height: 60))
            BottomBarMenu().previewLayout(.fixed(width: 375, height: 60))
```

Figure 9. Click the Debug View Hierarchy button

Xcode then shows you a 3D rendering of the view hierarchy. You can rotate the rendering to inspect the views. As you can see in figure 10, you can reveal all the layers of the card deck.

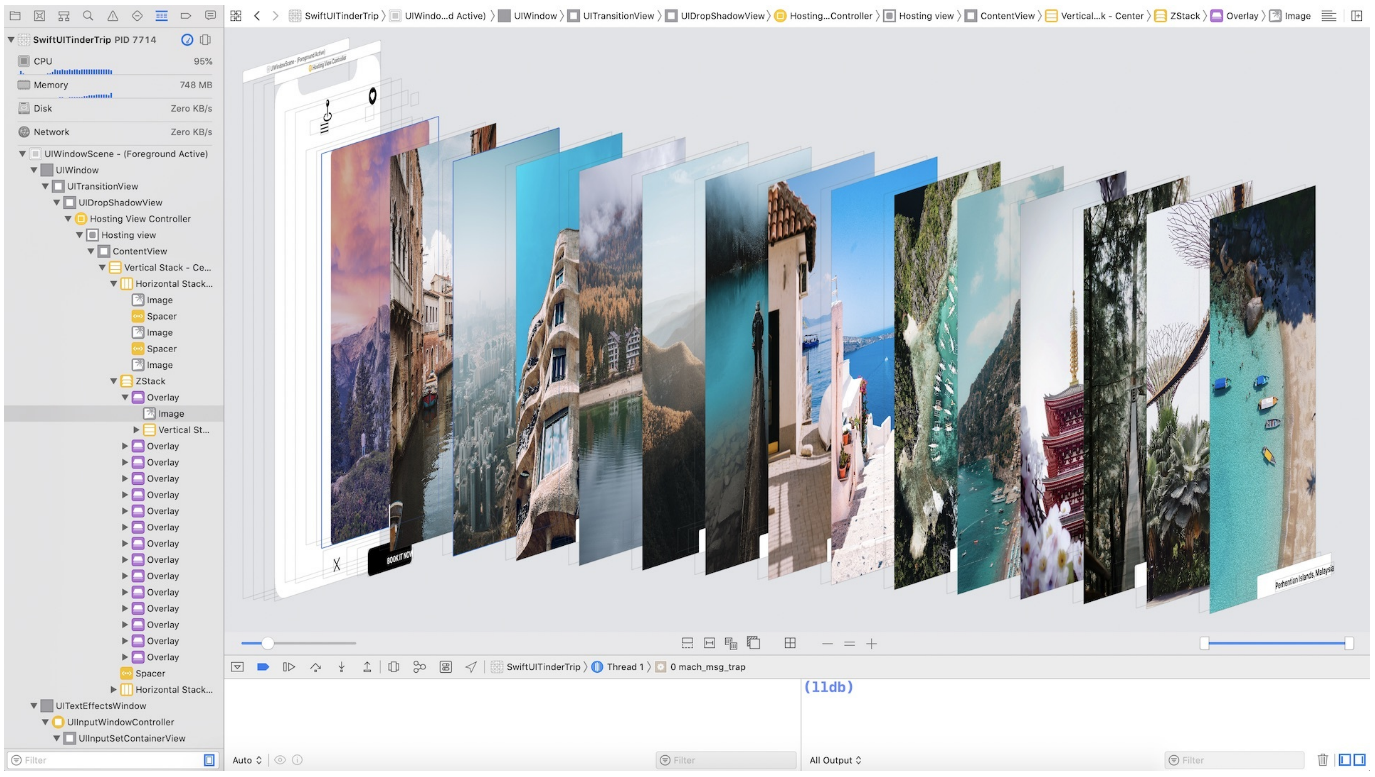


Figure 10. Click the Debug View Hierarchy button

Our card deck has two issues:

1. The first trip of the `trips` array is supposed to be the topmost card, however, it's now the lowermost card.
2. We rendered 15 card views for 15 trips. What if we have 10,000 trips or even more in the future? Should we create one card view for each of the trips? Is there a resource efficient way to implement the card deck?

Let's first fix the card order issue. SwiftUI provides the `zIndex` modifier for you to indicate the order of the views in a ZStack. A view with a higher value of `zIndex` is placed on top of those with a lower value. So, the topmost card should have the largest value of `zIndex`.

With this in mind, we create the following new function in `ContentView`:

```
private func isTopCard(cardView: CardView) -> Bool {

    guard let index = cardViews.firstIndex(where: { $0.id == cardView.id }) else {
        return false
    }

    return index == 0
}
```

While looping through the card views, we have to figure out a way to identify the topmost card. The function above takes in a card view, find out its index, and tells you if the card view is the topmost one.

Next, update the code block of `ZStack` like this:

```
ZStack {
    ForEach(cardViews) { cardView in
        cardView
            .zIndex(self.isTopCard(cardView: cardView) ? 1 : 0)
    }
}
```

We added the `zIndex` modifier for each of the card views. The topmost card is assigned a higher value of `zIndex`. In the preview canvas, you should now see the photo of the first trip (i.e. Yosemite, USA).

For the second issue, it's more complicated. Our goal is to make sure the card deck can support tens of thousands of card views but without becoming resource intensive.

Let's take a deeper look at the card deck. Do we actually need to initiate an individual card view for each trip photo? To create this card deck UI, we can just create two card views and overlay them with each other.

When the topmost card view is thrown away, the card view underneath becomes the topmost card. And, at the same time, we immediately initiate a new card view with a different photo and put it behind the topmost card. No matter how many photos you need to display in the card deck, the app has only two card views at all times. However, from a user point of view, the UI is composed of a pile of cards.

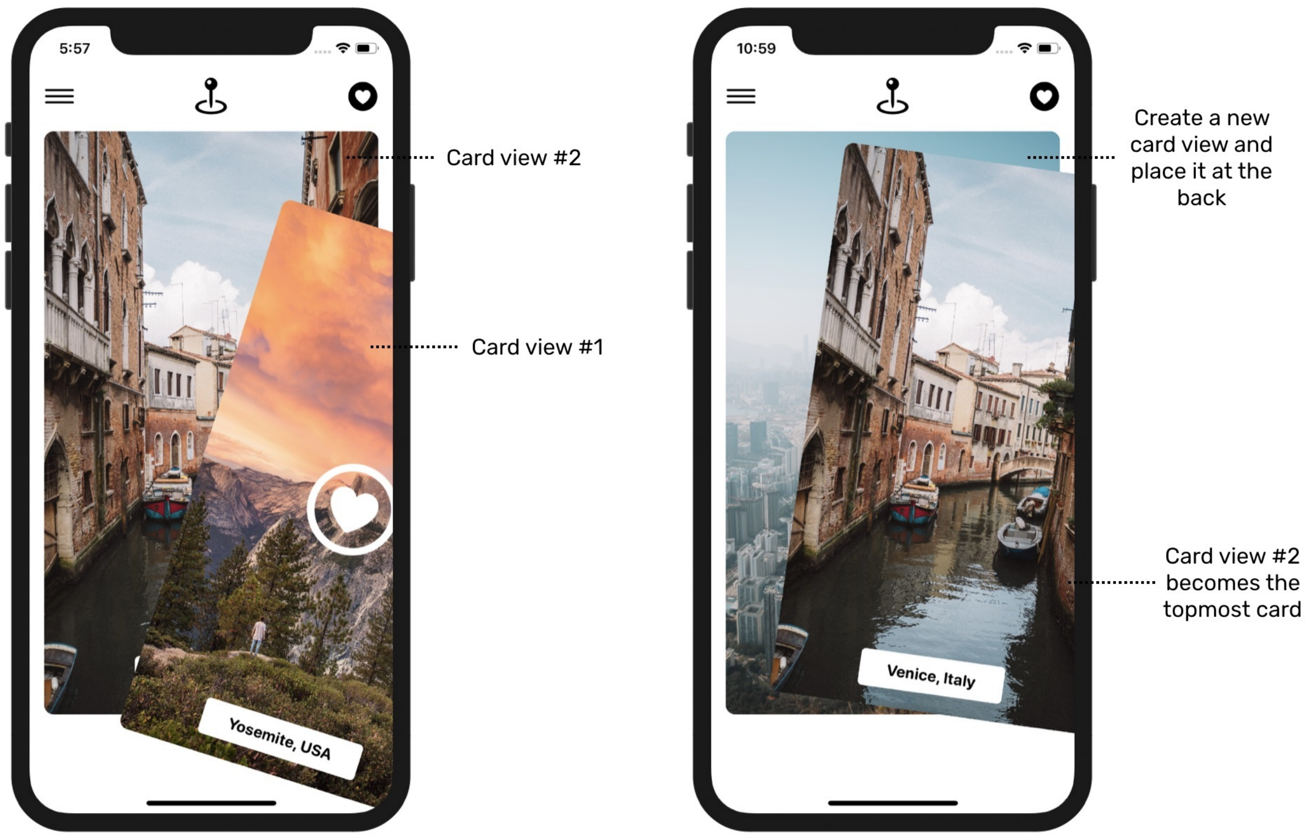

5:57
Card view #2
Card view #1
Yosemite, USA
10:59
Create a new card view and place it at the back
Card view #2 becomes the topmost card
Venice, Italy

Figure 11. How we use two card views to create a deck

Now that you understand how we are going to construct the card deck, let's move onto the implementation.

First, update the `cardViews` array, we no longer need to initialize all the trips but only the first two. Later, when the first trip (i.e. the first card) is thrown away, we will add another one to it.

```
var cardViews: [CardView] = {

    var views = [CardView]()

    for index in 0..<2 {
        views.append(CardView(image: trips[index].image, title: trips[index].desti
nation))
    }

    return views
}()
```

After the code change, the UI should look exactly the same. Run it in a simulator and debug the view hierarchy. You should only see two card views in the deck.

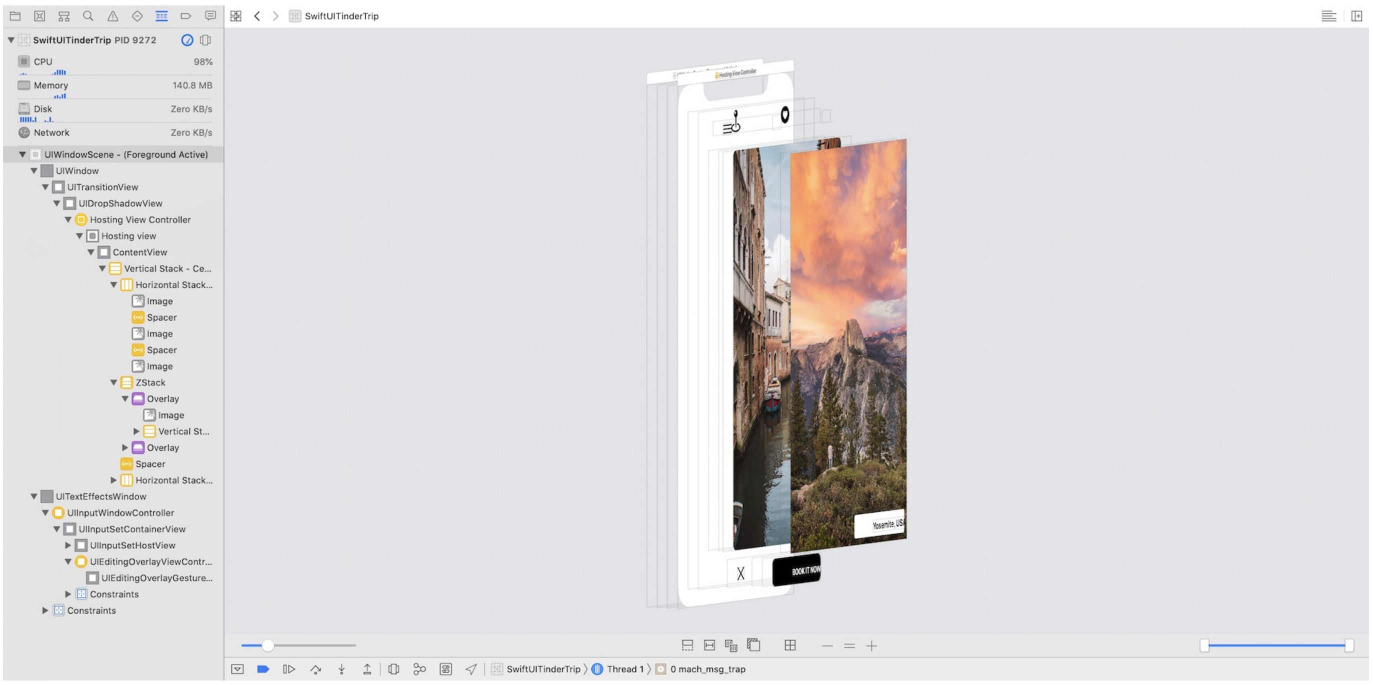


Figure 12. Using debug view hierarchy to view the card views

Implementing the Swiping Motion

Before we dynamically create a new card view, we have to implement the swipe feature first. If you forgot how to work with gestures, read chapters 17 and 18 again. We will reuse some of the code discussed before.

First, define the `DragState` enum in `ContentView`, which represents the possible drag states:

```
enum DragState {
    case inactive
    case pressing
    case dragging(translation: CGSize)

    var translation: CGSize {
        switch self {
        case .inactive, .pressing:
            return .zero
        case .dragging(let translation):
            return translation
        }
    }

    var isDragging: Bool {
        switch self {
        case .dragging:
            return true
        case .pressing, .inactive:
            return false
        }
    }

    var isPressing: Bool {
        switch self {
        case .pressing, .dragging:
            return true
        case .inactive:
            return false
        }
    }

}
```

Once again, if you don't understand what an enum is used for, stop here and review the chapters on gestures. Next, let's define a `@GestureState` variable to store the drag state, which is set to *inactive* by default:

```
@GestureState private var dragState = DragState.inactive
```

Now, update the `body` part like this:

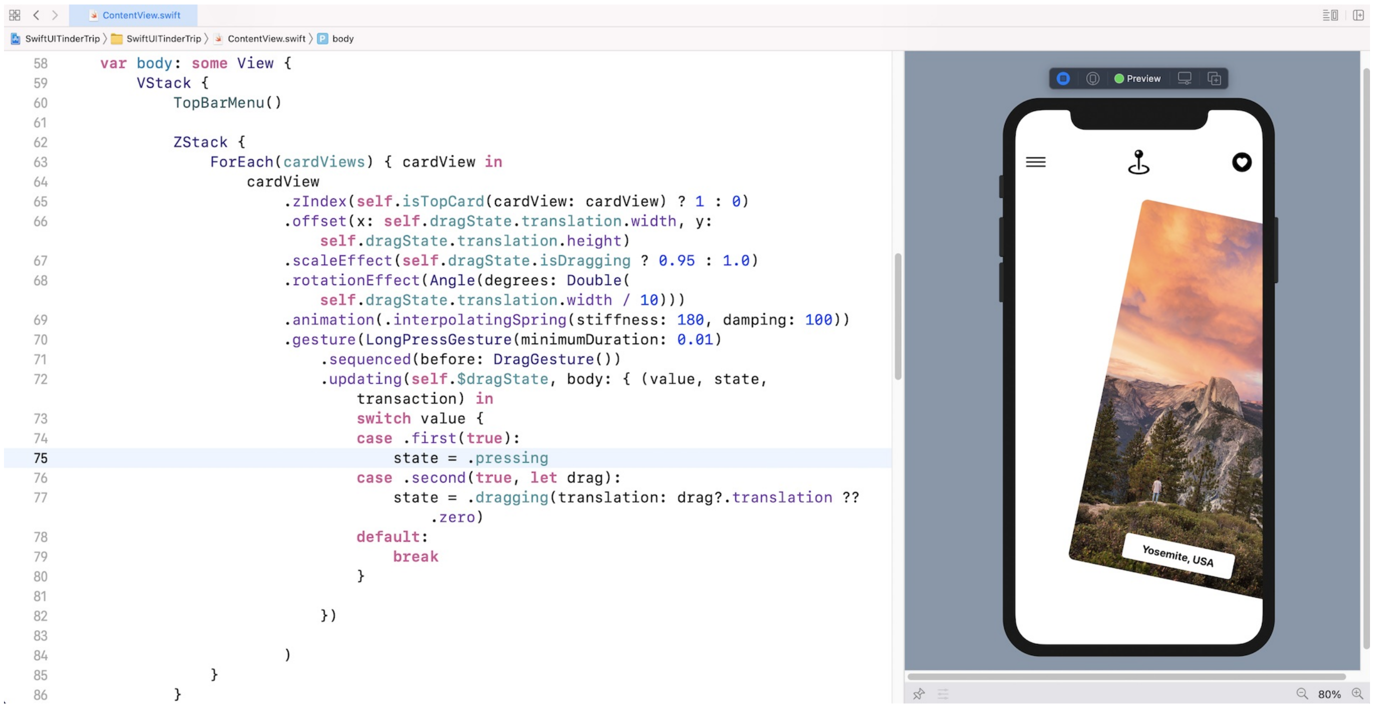
```
var body: some View {
    VStack {
        TopBarMenu()

        ZStack {
            ForEach(cardViews) { cardView in
                cardView
                    .zIndex(self.isTopCard(cardView: cardView) ? 1 : 0)
                    .offset(x: self.dragState.translation.width, y:  self.dragStat
e.translation.height)
                    .scaleEffect(self.dragState.isDragging ? 0.95 : 1.0)
                    .rotationEffect(Angle(degrees: Double( self.dragState.translat
ion.width / 10)))
                    .animation(.interpolatingSpring(stiffness: 180, damping: 100))
                    .gesture(LongPressGesture(minimumDuration: 0.01)
                        .sequenced(before: DragGesture())
                        .updating(self.$dragState, body: { (value, state, transact
ion) in
                            switch value {
                            case .first(true):
                                state = .pressing
                            case .second(true, let drag):
                                state = .dragging(translation: drag?.translation ?
? .zero)
                            default:
                                break
                            }

                        })

                    )
            }
        }

        Spacer(minLength: 20)

        BottomBarMenu()
            .opacity(dragState.isDragging ? 0.0 : 1.0)
            .animation(.default)
    }
}
```

Basically, we apply what we learned in the gesture chapter to implement the dragging. The `.gesture` modifier has two gesture recognizers: long press and drag. When the drag gesture is detected, we update the `dragState` variable and store the translation of the drag.

The combination of the `offset`, `scaleEffect`, `rotationEffect`, and `animation` modifiers create the drag effect. The drag is made possible by updating the `offset` of the card view. When the card view is in the dragging state, we will scale it down a little bit by using `scaleEffect` and rotate it at a certain angle by applying the `rotationEffect` modifier. The animation is set to `interpolatingSpring`, but you are free to try out other animations.

We also made some code changes to the `BottomBarMenu`. While a user is dragging the card view, I want to hide the bottom bar. Thus, we apply the `.opacity` modifier and set its value to zero when it's in the dragging state.

After you make the change, run the project in the preview canvas to test it. You should be able to drag the card and move around. And, when you release the card, it returns to its original position.



ContentView.swift
SwiftUITinderTrip 〉 SwiftUITinderTrip 〉 ContentView.swift 〉 body
Preview
Yosemite, USA
80%

```
var body: some View {
    VStack {
        TopBarMenu()

        ZStack {
            ForEach(cardViews) { cardView in
                cardView
                    .zIndex(self.isTopCard(cardView: cardView) ? 1 : 0)
                    .offset(x: self.dragState.translation.width, y:
                        self.dragState.translation.height)
                    .scaleEffect(self.dragState.isDragging ? 0.95 : 1.0)
                    .rotationEffect(Angle(degrees: Double(
                        self.dragState.translation.width / 10)))
                    .animation(.interpolatingSpring(stiffness: 180, damping: 100))
                    .gesture(LongPressGesture(minimumDuration: 0.01)
                        .sequenced(before: DragGesture())
                        .updating(self.$dragState, body: { (value, state,
                            transaction) in
                            switch value {
                            case .first(true):
                                state = .pressing
                            case .second(true, let drag):
                                state = .dragging(translation: drag?.translation ??
                                    .zero)
                            default:
                                break
                            }

                        })

                    )
            }
        }
```

Figure 13. Dragging the card view

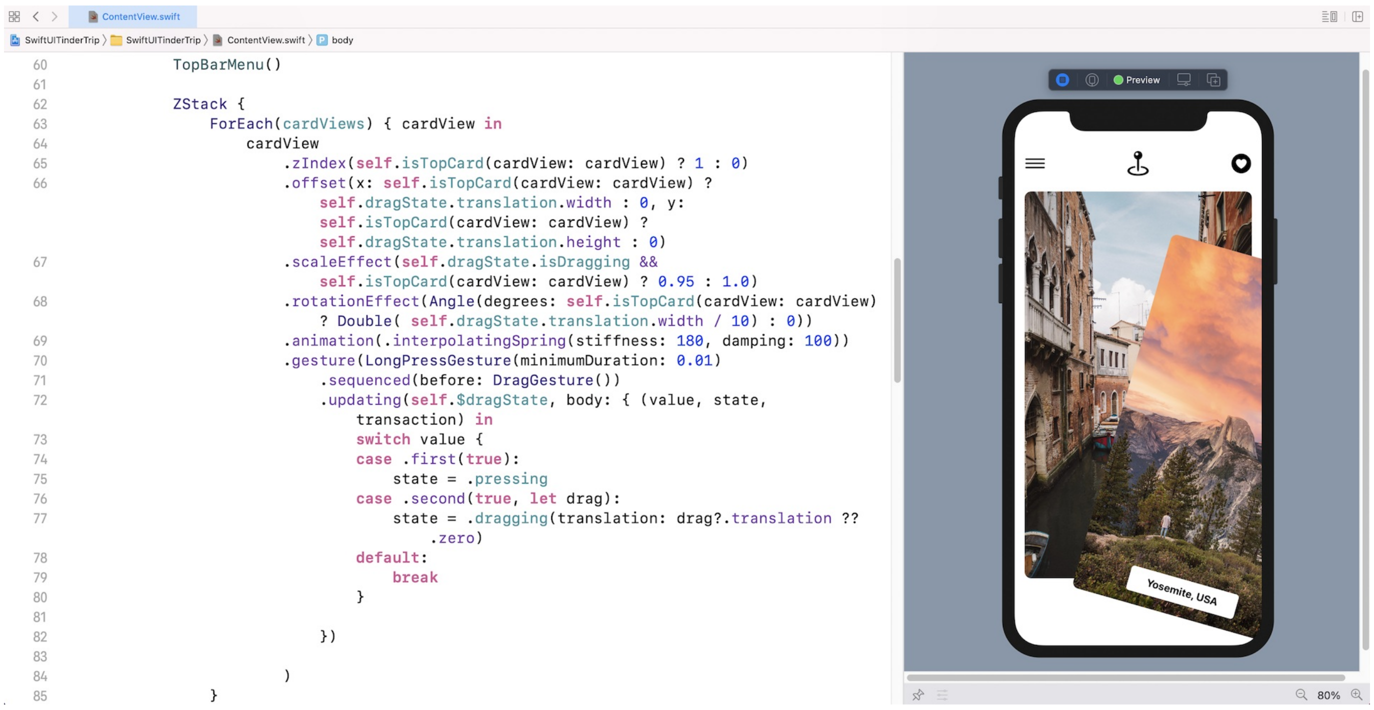
Do you notice a problem here? While the drag is working, you're actually dragging the whole card deck! It's supposed to only drag the topmost card and the card underneath should stay unchanged. Also, the scaling effect should only apply to the topmost card.

To fix the issues, we need to modify the code of the `offset`, `scaleEffect`, and `rotationEffect` modifiers such that the dragging only happens for the topmost card view.

```
ZStack {
    ForEach(cardViews) { cardView in
        cardView
            .zIndex(self.isTopCard(cardView: cardView) ? 1 : 0)
            .offset(x: self.isTopCard(cardView: cardView) ? self.dragState.transla
tion.width : 0, y: self.isTopCard(cardView: cardView) ? self.dragState.translation
.height : 0)
            .scaleEffect(self.dragState.isDragging && self.isTopCard(cardView: car
dView) ? 0.95 : 1.0)
            .rotationEffect(Angle(degrees: self.isTopCard(cardView: cardView) ? Do
uble( self.dragState.translation.width / 10) : 0))
            .animation(.interpolatingSpring(stiffness: 180, damping: 100))
            .gesture(LongPressGesture(minimumDuration: 0.01)
                .sequenced(before: DragGesture())
                .updating(self.$dragState, body: { (value, state, transaction) in
                    switch value {
                    case .first(true):
                        state = .pressing
                    case .second(true, let drag):
                        state = .dragging(translation: drag?.translation ?? .zero)
                    default:
                        break
                    }

                })

            )
    }
}
```

Just focus on the changes to the `offset`, `scaleEffect`, and `rotationEffect` modifiers. The rest of the code was kept intact. For those modifiers, we introduce an additional check such that the effects are only applied to the topmost card.

Now if you run the app again, you should see the card underneath and drag the topmost card.


ContentView.swift
SwiftUITinderTrip 〉 SwiftUITinderTrip 〉 ContentView.swift 〉 body
Preview
Yosemite, USA
80%

```
        TopBarMenu()

        ZStack {
            ForEach(cardViews) { cardView in
                cardView
                    .zIndex(self.isTopCard(cardView: cardView) ? 1 : 0)
                    .offset(x: self.isTopCard(cardView: cardView) ?
                        self.dragState.translation.width : 0, y:
                        self.isTopCard(cardView: cardView) ?
                        self.dragState.translation.height : 0)
                    .scaleEffect(self.dragState.isDragging &&
                        self.isTopCard(cardView: cardView) ? 0.95 : 1.0)
                    .rotationEffect(Angle(degrees: self.isTopCard(cardView: cardView)
                        ? Double( self.dragState.translation.width / 10) : 0))
                    .animation(.interpolatingSpring(stiffness: 180, damping: 100))
                    .gesture(LongPressGesture(minimumDuration: 0.01)
                        .sequenced(before: DragGesture())
                        .updating(self.$dragState, body: { (value, state,
                            transaction) in
                            switch value {
                            case .first(true):
                                state = .pressing
                            case .second(true, let drag):
                                state = .dragging(translation: drag?.translation ??
                                    .zero)
                            default:
                                break
                            }

                        })

                    )
            }
```

Figure 14. The dragging effect only applies to the topmost card

Displaying the Heart and xMark icons

Cool! The drag is now working. However, it's not done yet. The user should be able to swipe right/left to throw away the topmost card. And, there should be an icon (heart or xmark) shown on the card depending on the swiping direction.

First, let's declare a drag threshold in `ContentView`:

```
private let dragThreshold: CGFloat = 80.0
```

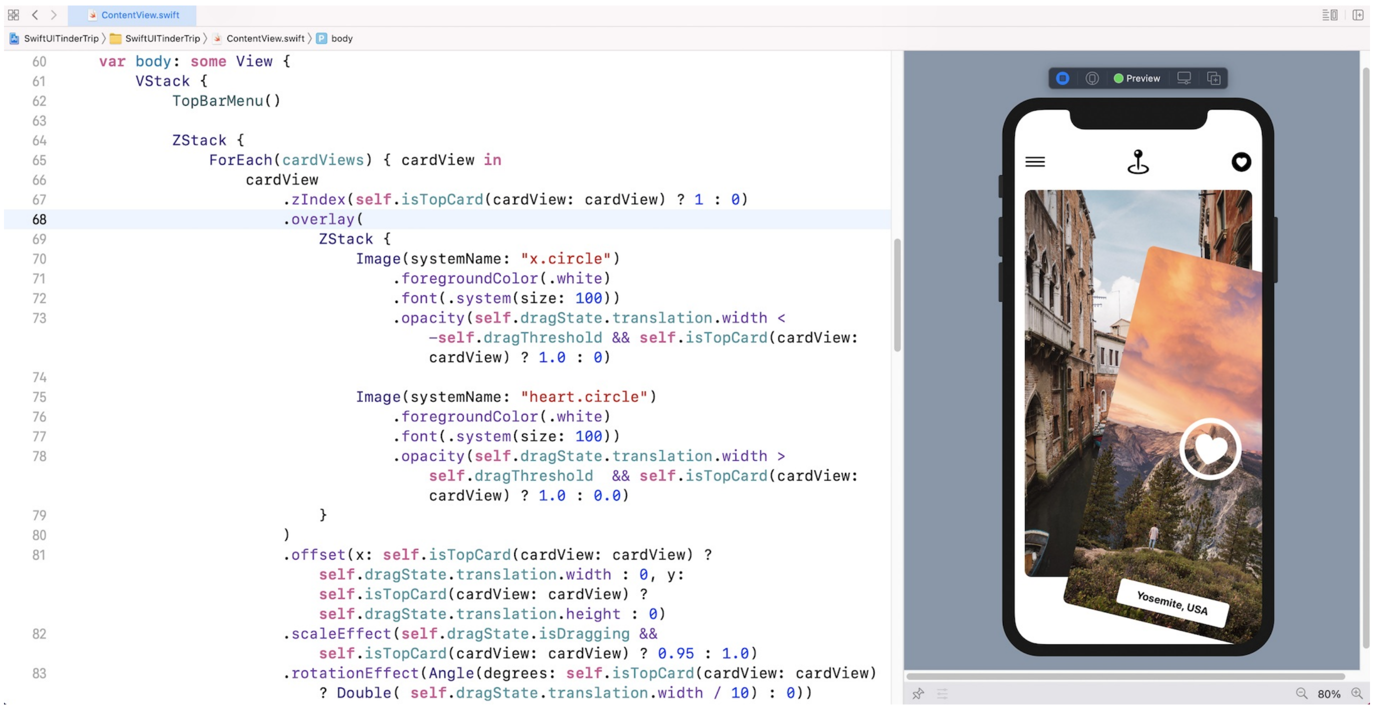
Once the translation of a drag passes the threshold, we will overlay an icon (either heart or xmark) on the card. Furthermore, if the user releases the card, the app will remove it from the deck, create a new one, and place the new card to the back of the deck.

To overlay the icon, add an `overlay` modifier to the `cardViews`. You can insert the following code under the `.zIndex` modifier:

```
.overlay(
    ZStack {
        Image(systemName: "x.circle")
            .foregroundColor(.white)
            .font(.system(size: 100))
            .opacity(self.dragState.translation.width < -self.dragThreshold && self
.isTopCard(cardView: cardView) ? 1.0 : 0)

        Image(systemName: "heart.circle")
            .foregroundColor(.white)
            .font(.system(size: 100))
            .opacity(self.dragState.translation.width > self.dragThreshold  && self
.isTopCard(cardView: cardView) ? 1.0 : 0.0)
    }
)
```

By default, both images are hidden by setting its opacity to zero. The translation's width has a positive value if the drag is to the right. Otherwise, it's a negative value. Depending on the drag direction, the app will unveil one of the images when the drag's translation exceeds the threshold.

You can run the project to have a quick test. When your drag exceeds the threshold, the heart/xmark icon will appear.


ContentView.swift
SwiftUITinderTrip 〉 SwiftUITinderTrip 〉 ContentView.swift 〉 body
Preview
Yosemite, USA
80%

```
    var body: some View {
        VStack {
            TopBarMenu()

            ZStack {
                ForEach(cardViews) { cardView in
                    cardView
                        .zIndex(self.isTopCard(cardView: cardView) ? 1 : 0)
                        .overlay(
                            ZStack {
                                Image(systemName: "x.circle")
                                    .foregroundColor(.white)
                                    .font(.system(size: 100))
                                    .opacity(self.dragState.translation.width <
                                        -self.dragThreshold && self.isTopCard(cardView:
                                        cardView) ? 1.0 : 0)

                                Image(systemName: "heart.circle")
                                    .foregroundColor(.white)
                                    .font(.system(size: 100))
                                    .opacity(self.dragState.translation.width >
                                        self.dragThreshold  && self.isTopCard(cardView:
                                        cardView) ? 1.0 : 0.0)
                            }
                        )
                        .offset(x: self.isTopCard(cardView: cardView) ?
                            self.dragState.translation.width : 0, y:
                            self.isTopCard(cardView: cardView) ?
                            self.dragState.translation.height : 0)
                        .scaleEffect(self.dragState.isDragging &&
                            self.isTopCard(cardView: cardView) ? 0.95 : 1.0)
                        .rotationEffect(Angle(degrees: self.isTopCard(cardView: cardView)
                            ? Double( self.dragState.translation.width / 10) : 0))
```

Figure 15. The heart icon appears

Removing/Inserting the Cards

Now when you release the card, it will still return to its original position. How do we remove the topmost card and add a new card at the same time?

First, let's mark the `cardViews` array with `@State` so that we can update its value and refresh the UI:

```
@State var cardViews: [CardView] = {

    var views = [CardView]()

    for index in 0..<2 {
        views.append(CardView(image: trips[index].image, title: trips[index].desti
nation))
    }

    return views
}()
```

Next, declare another state variable to keep track of the last index of the trip. Say, when the card deck is first initialized, we display the first two trips stored in the `trips` array. The last index is set to `1`.

```
@State private var lastIndex = 1
```

Okay, here comes the core function for removing and inserting the card views. Define a new function called `moveCard`:

```
private func moveCard() {
    cardViews.removeFirst()

    self.lastIndex += 1
    let trip = trips[lastIndex % trips.count]

    let newCardView = CardView(image: trip.image, title: trip.destination)

    cardViews.append(newCardView)
}
```

This function first removes the topmost card from the `cardViews` array, then it instantiates a new card view with the subsequent trip's image. Since `cardViews` is defined as a state property, SwiftUI will render the card views again once the array's value is changed. This is how we remove the topmost card and insert a new one to the deck.

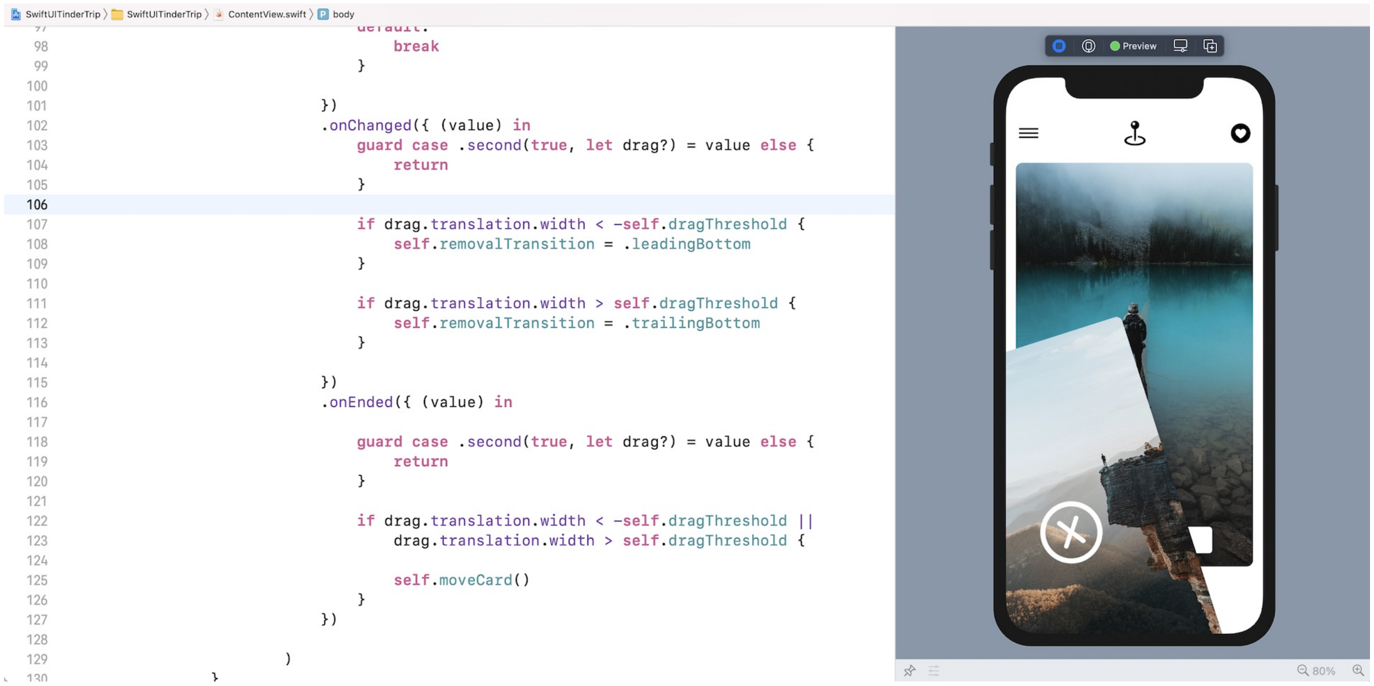
For this demo, I want the card deck to keep showing a trip. After the last photo of the `trips` array is displayed, the app will revert back to the first element (note the modulus operator % in the code above).

Next, update the `.gesture` modifier and insert the `.onEnded` function:

```
.gesture(LongPressGesture(minimumDuration: 0.01)
    .sequenced(before: DragGesture())
    .updating(self.$dragState, body: { (value, state, transaction) in
        .
        .
        .
    })
    .onEnded({ (value) in

        guard case .second(true, let drag?) = value else {
            return
        }

        if drag.translation.width < -self.dragThreshold ||
            drag.translation.width > self.dragThreshold {

            self.moveCard()
        }
    })
)
```

When the drag gesture ends, we check if the drag's translation exceeds the threshold and call the `moveCard()` accordingly.

Now run the project in the preview canvas. Drag the image to the right/left until the icon appears. Release the drag and the topmost card should be replaced by the next card.


SwiftUITinderTrip 〉 SwiftUITinderTrip 〉 ContentView.swift 〉 body
Preview
80%

```
                    break
                }

            })
            .onChanged({ (value) in
                guard case .second(true, let drag?) = value else {
                    return
                }

                if drag.translation.width < -self.dragThreshold {
                    self.removalTransition = .leadingBottom
                }

                if drag.translation.width > self.dragThreshold {
                    self.removalTransition = .trailingBottom
                }

            })
            .onEnded({ (value) in

                guard case .second(true, let drag?) = value else {
                    return
                }

                if drag.translation.width < -self.dragThreshold ||
                    drag.translation.width > self.dragThreshold {

                    self.moveCard()
                }
            })

        )
```

Figure 16. Removing the topmost card

Fine Tuning the Animations

The app almost works but the animation falls short of expectations. Instead of having the card view disappear abruptly, the card should fall out of the screen gradually when it's thrown away.

To fine tune the animation effect, we will attach the `transition` modifier and apply an asymmetric transition to the card views.

Add the extension, `AnyTransition` to the bottom of ContentView.swift and define two transition effects:

```
extension AnyTransition {
    static var trailingBottom: AnyTransition {
        AnyTransition.asymmetric(
            insertion: .identity,
            removal: AnyTransition.move(edge: .trailing).combined(with: .move(edge
: .bottom))
        )

    }

    static var leadingBottom: AnyTransition {
        AnyTransition.asymmetric(
            insertion: .identity,
            removal: AnyTransition.move(edge: .leading).combined(with: .move(edge:
 .bottom))
        )
    }
}
```

The reason why we use asymmetric transitions is that we only want to animate the transition when the card view is removed. When a new card view is inserted in the deck, there should be no animation.

The `trailingBottom` transition is used when the card view is thrown away to the right of the screen, while we apply the `leadingBottom` transition when the card view is thrown away to the left.

Next, declare a state property that holds the transition type. It's set to `trailingBottom` by default.

```
@State private var removalTransition  = AnyTransition.trailingBottom
```

Now attach the `.transition` modifier to the card view. You can place it after the `.animation` modifier:

```
.transition(self.removalTransition)
```

Finally, update the code of the `.gesture` modifier with the `onChanged` function like this:

```
.gesture(LongPressGesture(minimumDuration: 0.01)
    .sequenced(before: DragGesture())
    .updating(self.$dragState, body: { (value, state, transaction) in
        switch value {
        case .first(true):
            state = .pressing
        case .second(true, let drag):
            state = .dragging(translation: drag?.translation ?? .zero)
        default:
            break
        }

    })
    .onChanged({ (value) in
        guard case .second(true, let drag?) = value else {
            return
        }

        if drag.translation.width < -self.dragThreshold {
            self.removalTransition = .leadingBottom
        }

        if drag.translation.width > self.dragThreshold {
            self.removalTransition = .trailingBottom
        }

    })
    .onEnded({ (value) in

        guard case .second(true, let drag?) = value else {
            return
        }

        if drag.translation.width < -self.dragThreshold ||
            drag.translation.width > self.dragThreshold {

            self.moveCard()
        }
    })
)
```

The code sets the `removalTransition`. The transition type is updated according to the swipe direction. Now you're ready to run the app again. You should now see an improved animation when the card is thrown away.

Summary

With SwiftUI, you can easily build some cool animations and mobile UI patterns. This Tinder-like UI is an examples.

I hope you fully understand what I covered in this chapter so you can adapt the code to fit your own project. It's quite a huge chapter. I wanted to document my thought process instead of just presenting you with the final solution. Just like you and many other developers, I am still studying this new framework and exploring its best practices. This is one of many approaches to create this kind of UI. If you come up with a better approach, I am happy to discuss it with you. Feel free to send me email at simonng@appcoda.com.

For reference, you can download the complete tinder project here:

- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUITinderTrip.zip)

Chapter 20
Creating an Apple Wallet like Animation and View Transition

Do you use Apple's Wallet app? In the previous chapter, we built a simple app with a Tinder-like UI. What we're going to do in this chapter is to create an animated UI similar to the one you see in the Wallet app. When you tap and hold a credit card in the wallet app, you can use the drag gesture to rearrange the cards. If you haven't used the app, open Wallet and take a quick look. Alternatively, you can visit this URL (https://link.appcoda.com/swiftui-wallet) to check out the animation we're going to build.

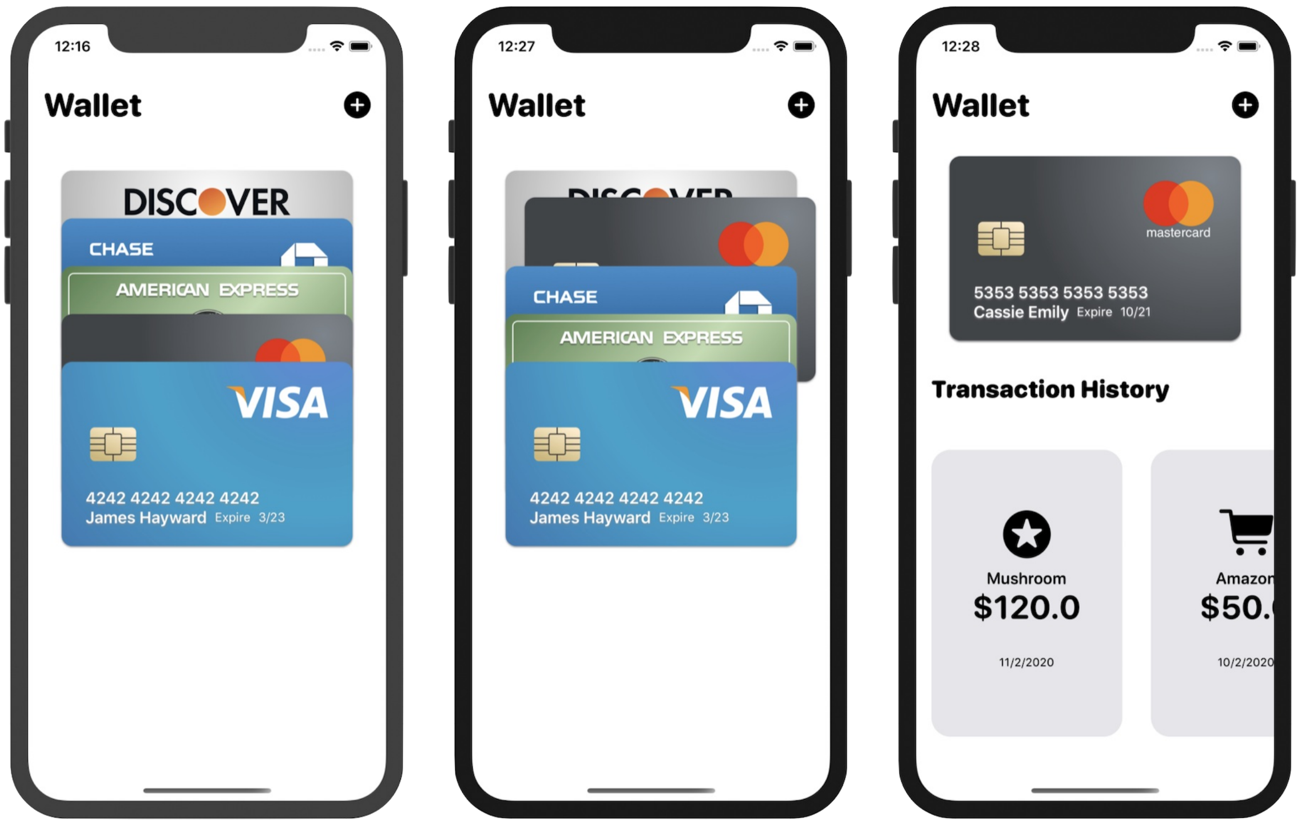
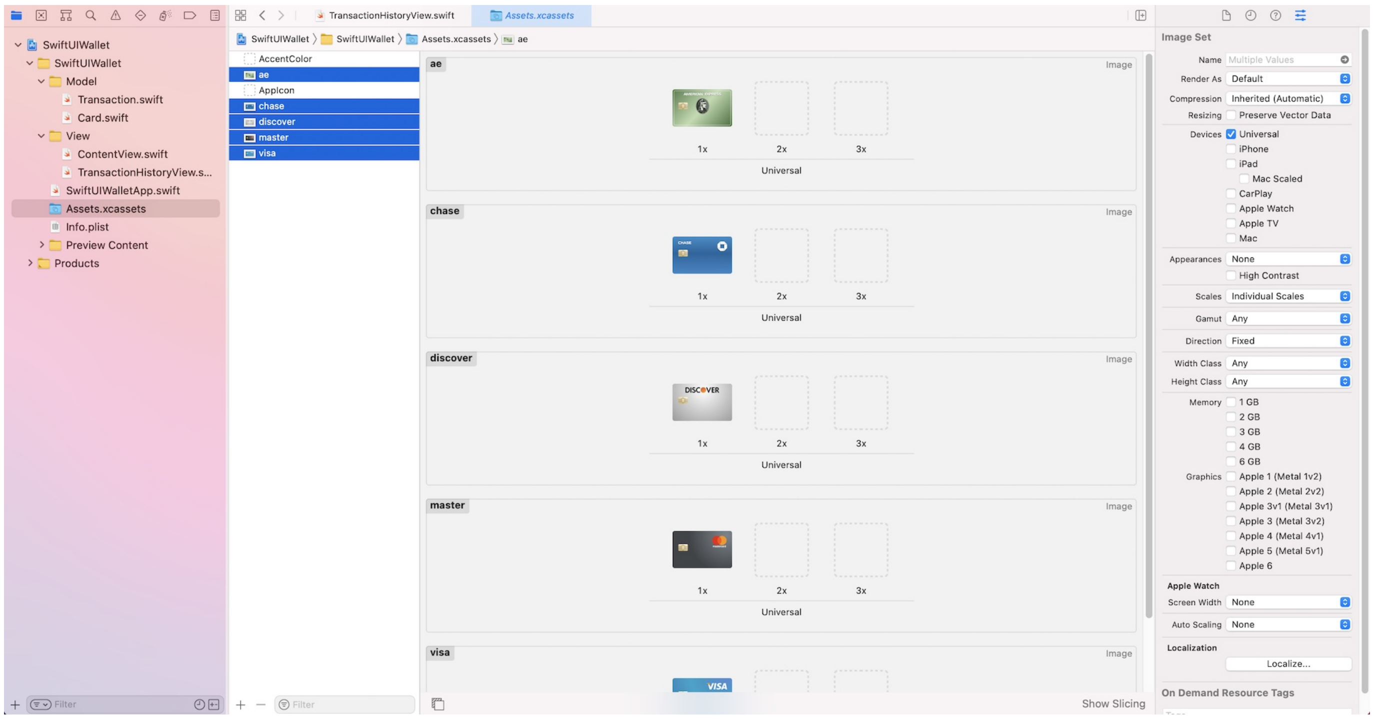


Figure 1. Building a Wallet-like animations and view transitions

In the Wallet app, tapping one of the cards will bring up its transaction history. We will also create a similar animation, so you'll better understand view transitions and horizontal scroll view.

Project Preparation

To keep you focused on learning animations and view transitions, begin with this starter project (https://www.appcoda.com/resources/swiftui2/SwiftUIWalletStarter.zip). The starter project already bundles the required credit card images and comes with a built-in transaction history view. If you want to use your own images, please replace them in the asset catalog.



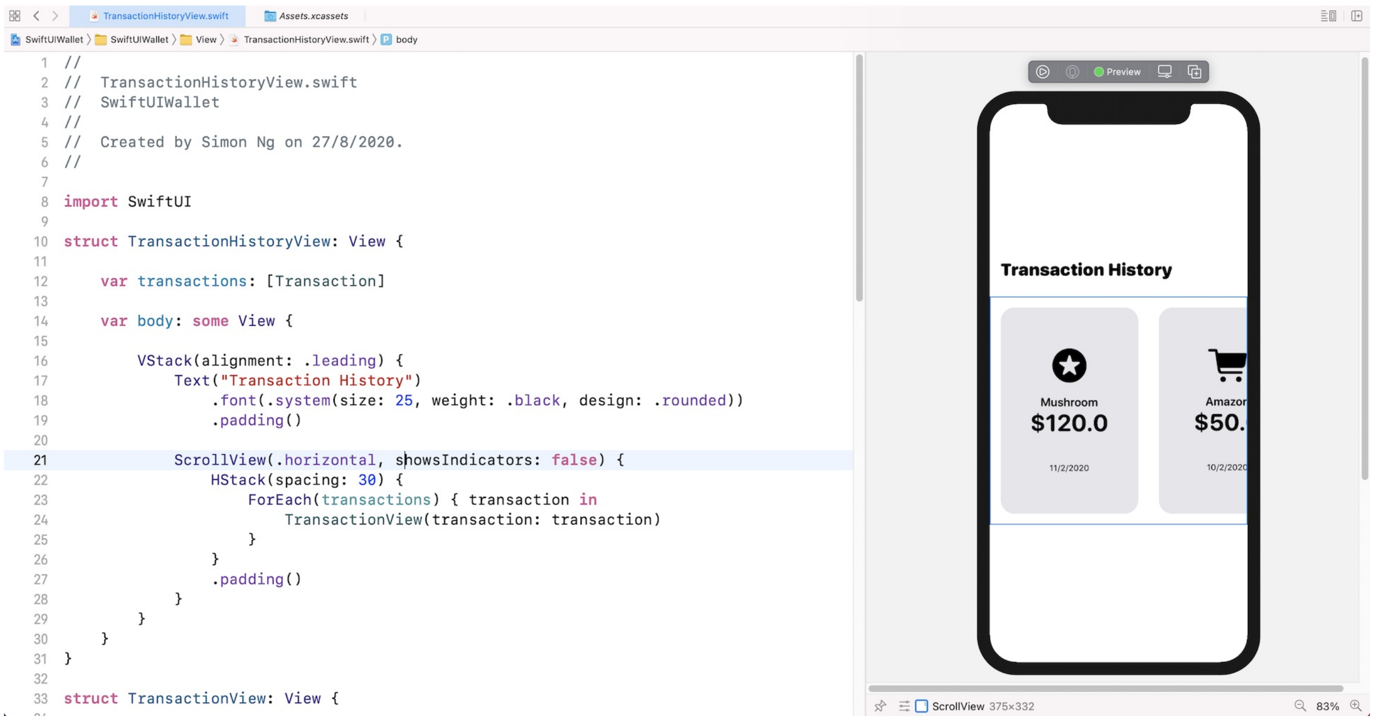
Name: Multiple Values
Render As: Default
Compression: Inherited (Automatic)
Resizing: Preserve Vector Data
Devices: Universal ✓, iPhone, iPad, Mac Scaled, CarPlay, Apple Watch, Apple TV, Mac
Appearances: None
High Contrast
Scales: Individual Scales
Gamut: Any
Direction: Fixed
Width Class: Any
Height Class: Any
Memory: 1 GB, 2 GB, 3 GB, 4 GB, 6 GB
Graphics: Apple 1 (Metal 1v2), Apple 2 (Metal 2v2), Apple 3v1 (Metal 3v1), Apple 3 (Metal 3v2), Apple 4 (Metal 4v1), Apple 5 (Metal 5v1), Apple 6
Screen Width: None
Auto Scaling: None

Figure 2. The starter project bundles the credit card images

In the project navigator, you should find a number of `.swift` files:

- **Transaction.swift** - the `Transaction` struct represents a transaction in the wallet app. Each transaction has an unique ID, merchant, amount, date, and icon. In addition to the `Transaction` struct, we also declare an array of test transactions for demo purposes.

- **Card.swift** - this file contains the struct of `Card`. A `Card` represents the data of a credit card including the card number, type, expiry date, image, and the customer's name. Additionally, there is an array of test credit cards in the file. One point to note is that the card image doesn't include any personal information, only the card brand (e.g. Visa). Later, we will create a view for a credit card.
- **TransactionHistoryView.swift** - this is the transaction history view displayed in figure 1. The starter project comes with an implementation of the transaction history view. We display the transactions in a horizontal scroll view. You've worked with vertical scroll views before. The trick of creating a horizontal view is to pass a value of `.horizontal` during the initialization of a scroll view. Take a look at figure 3 or simply look at the Swift file for details.
- **ContentView.swift** - this is the default SwiftUI view generated by Xcode.



```swift
//
//  TransactionHistoryView.swift
//  SwiftUIWallet
//
//  Created by Simon Ng on 27/8/2020.
//

import SwiftUI

struct TransactionHistoryView: View {

    var transactions: [Transaction]

    var body: some View {

        VStack(alignment: .leading) {
            Text("Transaction History")
                .font(.system(size: 25, weight: .black, design: .rounded))
                .padding()

            ScrollView(.horizontal, showsIndicators: false) {
                HStack(spacing: 30) {
                    ForEach(transactions) { transaction in
                        TransactionView(transaction: transaction)
                    }
                }
                .padding()
            }
        }
    }
}

struct TransactionView: View {
```

Figure 3. Using .horizontal to create a horizontal scroll view

Building a Card View

As mentioned in the previous section, all the card's images do not include any personal information and card number. Open the asset catalog again and take a look at the images. Each of the card images only has the card logo. We will soon create a card view to lay out the personal information and card number, as shown in figure 4.

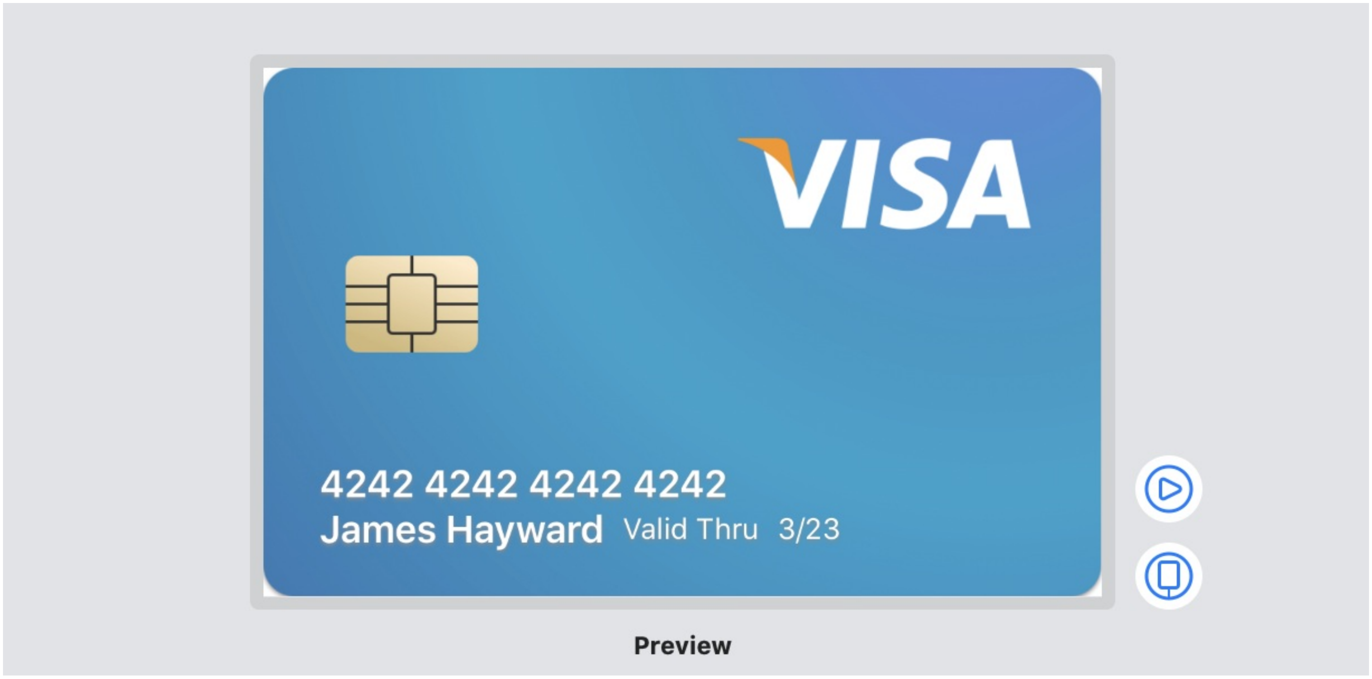


Figure 4. A sample card

To create the card view, right click the `View` group in the project navigator and create a new file. Choose the *SwiftUI View* template and name the file `CardView.swift`. Next, update the code like this:

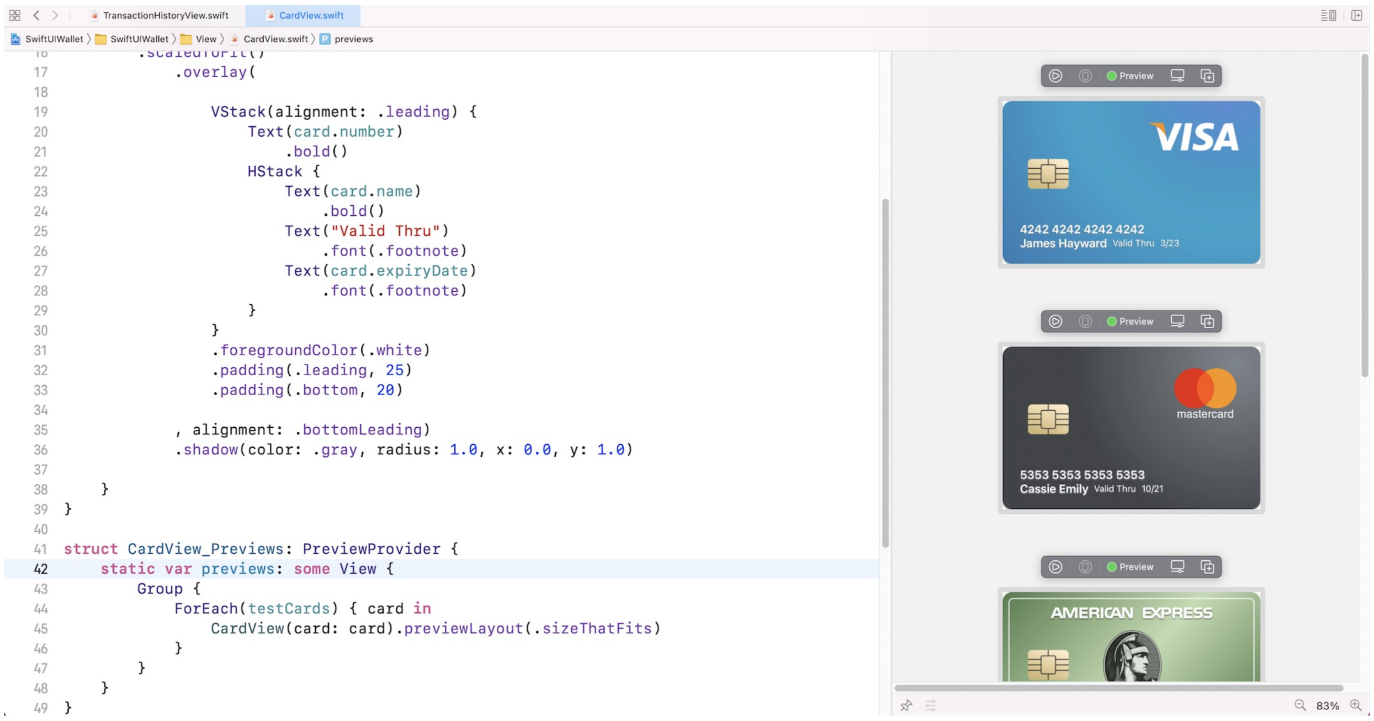
```
struct CardView: View {
    var card: Card

    var body: some View {
        Image(card.image)
        .resizable()
        .scaledToFit()
            .overlay(

                VStack(alignment: .leading) {
                    Text(card.number)
                        .bold()
                    HStack {
                        Text(card.name)
                            .bold()
                        Text("Valid Thru")
                            .font(.footnote)
                        Text(card.expiryDate)
                            .font(.footnote)
                    }
                }
                .foregroundColor(.white)
                .padding(.leading, 25)
                .padding(.bottom, 20)

            , alignment: .bottomLeading)
            .shadow(color: .gray, radius: 1.0, x: 0.0, y: 1.0)

    }
}
```

We declare a `card` property to take in the card data. To display the personal information and card number on the card image, we use the `overlay` modifier and layout the text components with a vertical stack view and a horizontal stack view.

To preview the cards, update the `CardView_Previews` struct like this:

```
struct CardView_Previews: PreviewProvider {
    static var previews: some View {
        Group {
            ForEach(testCards) { card in
                CardView(card: card).previewLayout(.sizeThatFits)
            }
        }
    }
}
```

The `testCards` variable was defined in `Card.swift`. Therefore, we use `ForEach` to loop through the cards and preview each card by calling `previewLayout`. Since we passed `previewLayout` with `.sizeThatFits`, Xcode will layout the card like that shown in figure 5, instead of displaying it in a simulated device.


TransactionHistoryView.swift
CardView.swift
SwiftUIWallet 〉 SwiftUIWallet 〉 View 〉 CardView.swift 〉 previews
Preview
VISA
4242 4242 4242 4242
James Hayward Valid Thru 3/23
Preview
mastercard
5353 5353 5353 5353
Cassie Emily Valid Thru 10/21
Preview
AMERICAN EXPRESS
83%

```
        .scaledToFit()
            .overlay(

                VStack(alignment: .leading) {
                    Text(card.number)
                        .bold()
                    HStack {
                        Text(card.name)
                            .bold()
                        Text("Valid Thru")
                            .font(.footnote)
                        Text(card.expiryDate)
                            .font(.footnote)
                    }
                }
                .foregroundColor(.white)
                .padding(.leading, 25)
                .padding(.bottom, 20)

            , alignment: .bottomLeading)
            .shadow(color: .gray, radius: 1.0, x: 0.0, y: 1.0)

    }
}

struct CardView_Previews: PreviewProvider {
    static var previews: some View {
        Group {
            ForEach(testCards) { card in
                CardView(card: card).previewLayout(.sizeThatFits)
            }
        }
    }
}
```

Figure 5. Previewing the card views

Building the Wallet View and Card Deck

Now that we have implemented the card view, let's start to build the wallet view. If you forgot what the wallet view looks like, take a look at figure 6. We will first layout the card deck before working on the gestures and animations.

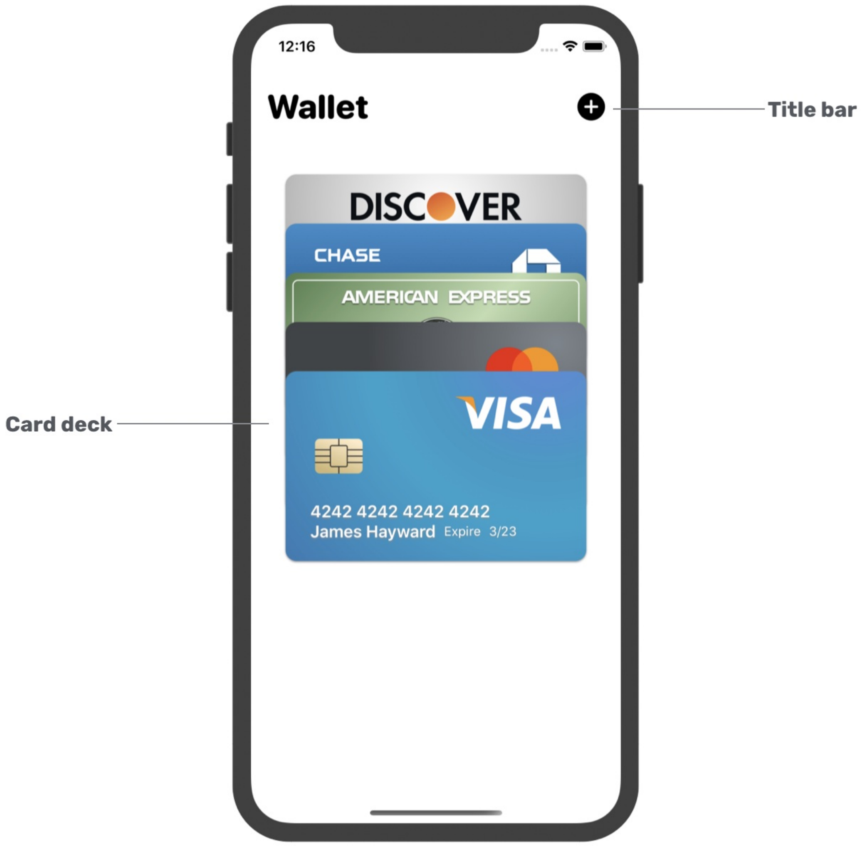


Figure 6. The wallet view

In the project navigator, you should see the `ContentView.swift` file. Delete it and then right click the `View` folder to create a new one. In the dialog, choose *SwiftUI View* as the template and name the file `WalletView.swift`.

If you preview the `WalletView` or run the app on simulator, Xcode should display an error because the `ContentView` is set to the initial view and it was deleted. To fix the error, open `SwiftUIWalletApp.swift` and change the following line of code in `WindowGroup` from:

```
ContentView()
```

To:

```
WalletView()
```

Switch back to `WalletView.swift`. The compilation error will be fixed once you make the change. Now let's continue to layout the wallet view. First, we'll start with the title bar. In the `WalletView.swift` file, insert a new struct for the bar:

```
struct TopNavBar: View {

    var body: some View {
        HStack {
            Text("Wallet")
                .font(.system(.largeTitle, design: .rounded))
                .fontWeight(.heavy)

            Spacer()

            Image(systemName: "plus.circle.fill")
                .font(.system(.title))
        }
        .padding(.horizontal)
        .padding(.top, 20)
    }
}
```

The code is very straightforward. We laid out the title and the plus image using a horizontal stack.

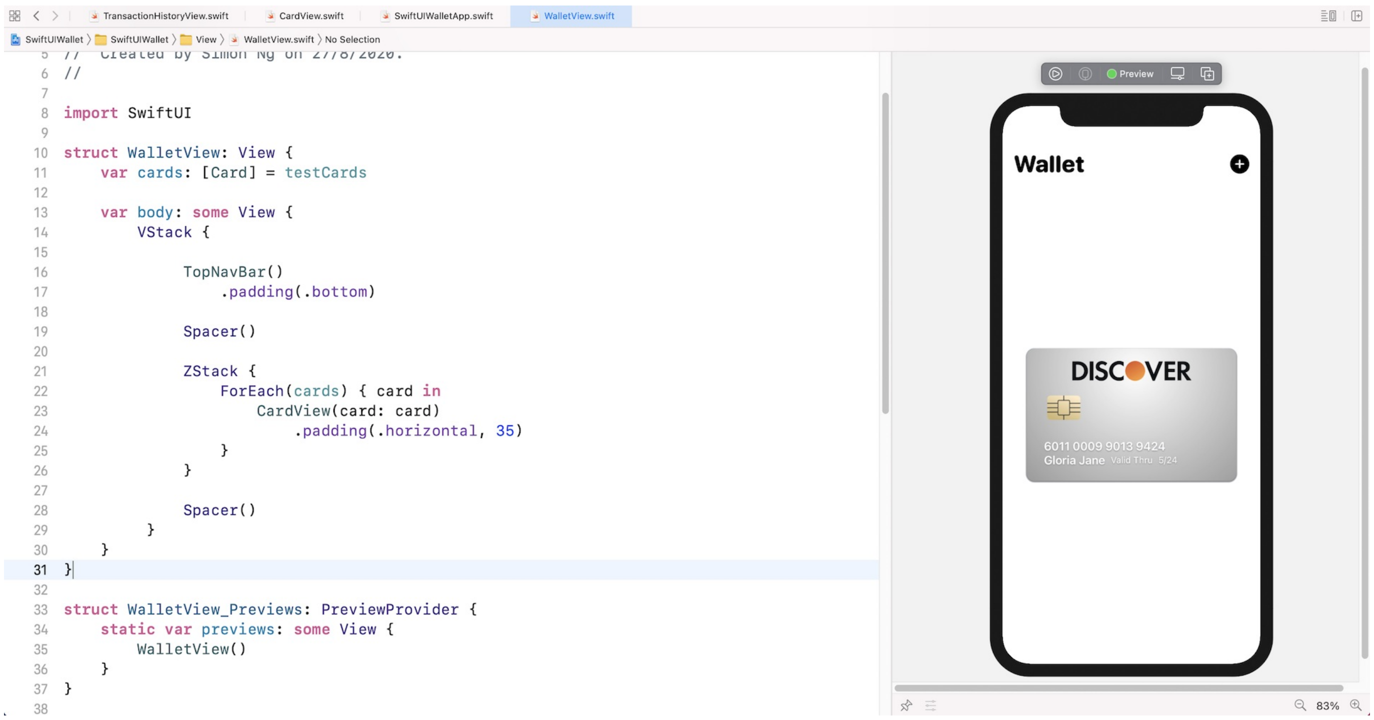
Next, we create the card deck. First, declare a property in the `WalletView` struct for the array of credit cards:

```
var cards: [Card] = testCards
```

For demo purpose, we simply set the default value to `testCards` which was defined in the `Card.swift` file. To lay out the wallet view, we use both a `VStack` and `ZStack`. Update the `body` variable like this:

```
var body: some View {
   VStack {

        TopNavBar()
            .padding(.bottom)

        Spacer()

        ZStack {
            ForEach(cards) { card in
                CardView(card: card)
                    .padding(.horizontal, 35)
            }
        }

        Spacer()
    }
}
```

If you run the app on simulator or preview the UI directly, you should only see the last card in the card deck like that shown in figure 7.



```
//  Created by Simon Ng on 27/8/2020.
//

import SwiftUI

struct WalletView: View {
    var cards: [Card] = testCards

    var body: some View {
        VStack {

            TopNavBar()
                .padding(.bottom)

            Spacer()

            ZStack {
                ForEach(cards) { card in
                    CardView(card: card)
                        .padding(.horizontal, 35)
                }
            }

            Spacer()
        }
    }
}

struct WalletView_Previews: PreviewProvider {
    static var previews: some View {
        WalletView()
    }
}
```

TransactionHistoryView.swift
CardView.swift
SwiftUIWalletApp.swift
WalletView.swift
SwiftUIWallet 〉 SwiftUIWallet 〉 View 〉 WalletView.swift 〉 No Selection
Preview
Wallet
DISCOVER
6011 0009 9013 9424
Gloria Jane Valid Thru 5/24
83%

Figure 7. Trying to display the card deck

There are two issues with the current implementation:

1. *The cards are now overlapped with each other* - we need to figure out a way to spread out the deck of cards.
2. *The Discover card is supposed to be the last card* - In a ZStack, the items stack on top of each other. The first item being put into the ZStack becomes the lowermost layer, while the last item is the uppermost layer. If you look at the `testCards` array in `Card.swift`, the first card is the Visa card, while the last card is the Discover card.

Okay, so how are we going to fix these issues? For the first issue, we can make use of the `offset` modifier to spread out the deck of cards. For the second issue, obviously we can alter the `zIndex` for each card in the `CardView` to change the order of the cards. Figure 8 illustrates how the solution works.

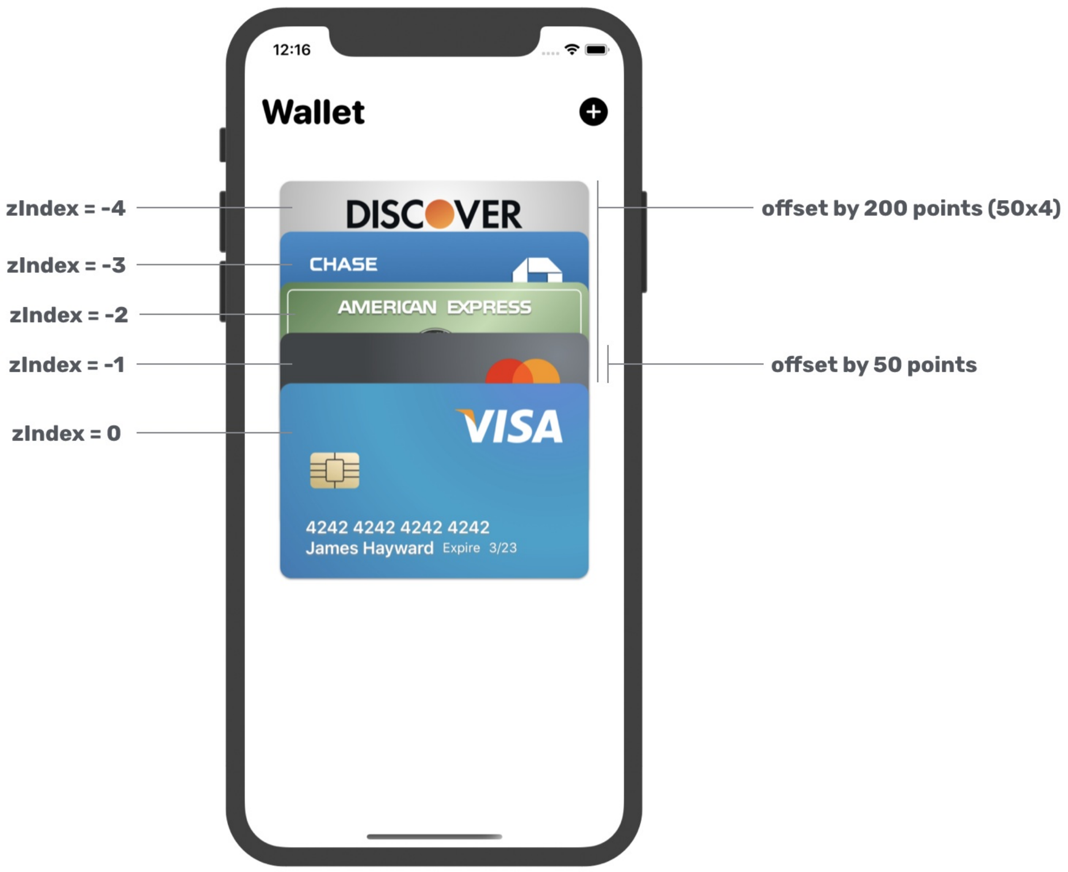


Figure 8. Understanding zIndex and offset

Let's first talk about the z-index. Each card's z-index is the negative value of its index in the `cards` array. The last item with the largest array index will have the smallest z-index. For this implementation, we will create an individual function to handle the computation of z-index. In the `WalletView`, insert the following code:

```
private func zIndex(for card: Card) -> Double {
    guard let cardIndex = index(for: card) else {
        return 0.0
    }

    return -Double(cardIndex)
}

private func index(for card: Card) -> Int? {
    guard let index = cards.firstIndex(where: { $0.id == card.id }) else {
        return nil
    }

    return index
}
```

Both functions work together to figure out the correct z-index of a given card. To compute a correct z-index, the first thing we need is the index of the card in the `cards` array. The `index(for:)` function is designed to get the array index of the given card. Once we have the index, we can turn it into a negative value. This is what the `zIndex(for:)` function does.

Now, you can attach the `zIndex` modifier to the `CardView` like this:

```
CardView(card: card)
    .padding(.horizontal, 35)
    .zIndex(self.zIndex(for: card))
```

Once you make the change, the Visa card should move to the top of the deck.

Next, let's fix the first issue to spread out the cards. Each of the cards should be offset by a certain vertical distance. That distance is computed by using the card's index. Say, we set the default vertical offset to 50 points. The last card (with the index #4) will be offset by 200 points (50*4).

Now that you should understand how we are going to spread the cards, let's write the code. Declare the default vertical offset in `WalletView`:

```
private static let cardOffset: CGFloat = 50.0
```

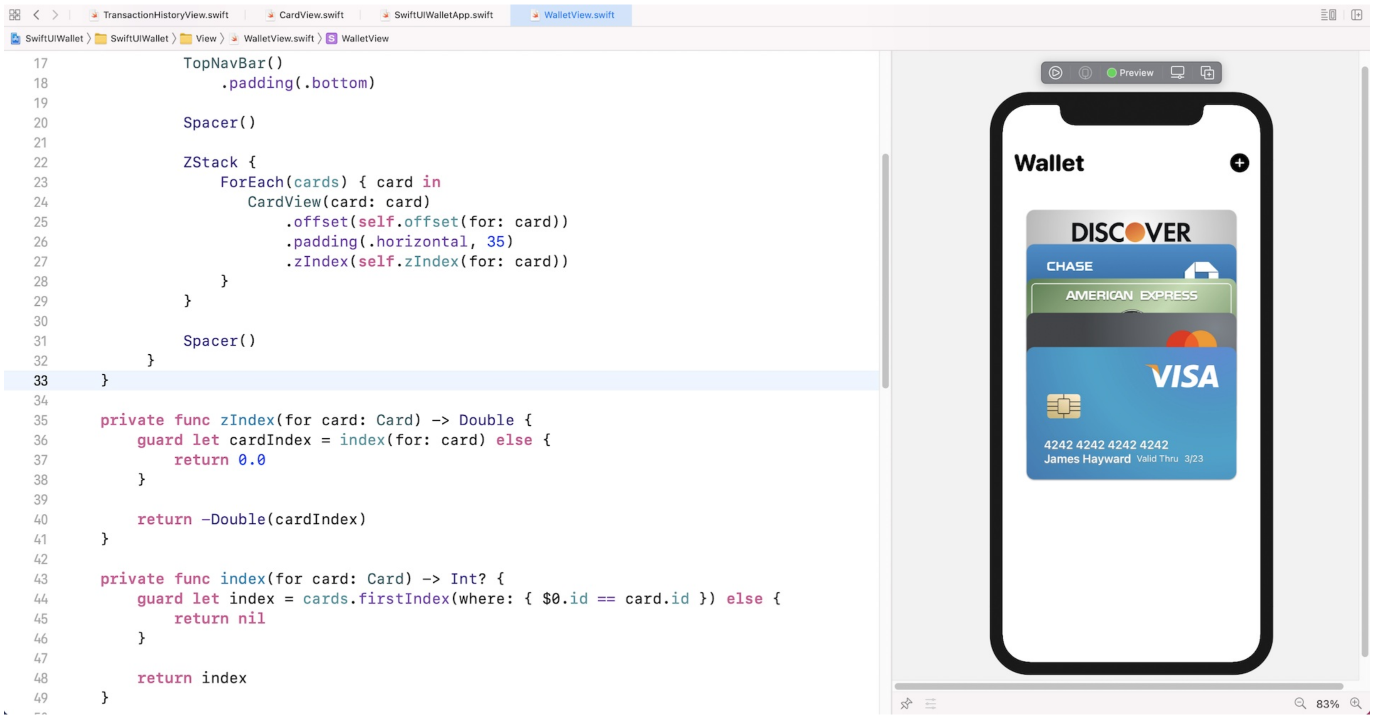
Next, create a new function called `offset(for:)` that is used to compute the vertical offset of the given card:

```
private func offset(for card: Card) -> CGSize {

    guard let cardIndex = index(for: card) else {
        return CGSize()
    }

    return CGSize(width: 0, height: -50 * CGFloat(cardIndex))
}
```

Finally, attach the `offset` modifier to the `CardView` :

```
CardView(card: card)
    .offset(self.offset(for: card))
    .padding(.horizontal, 35)
    .zIndex(self.zIndex(for: card))
```

That's how we spread the card using the `offset` modifier. If everything is correct, you should see a preview like that shown in figure 9.



```
            TopNavBar()
                .padding(.bottom)

            Spacer()

            ZStack {
                ForEach(cards) { card in
                    CardView(card: card)
                        .offset(self.offset(for: card))
                        .padding(.horizontal, 35)
                        .zIndex(self.zIndex(for: card))
                }
            }

            Spacer()
        }
    }

    private func zIndex(for card: Card) -> Double {
        guard let cardIndex = index(for: card) else {
            return 0.0
        }

        return -Double(cardIndex)
    }

    private func index(for card: Card) -> Int? {
        guard let index = cards.firstIndex(where: { $0.id == card.id }) else {
            return nil
        }

        return index
    }
```

TransactionHistoryView.swift
CardView.swift
SwiftUIWalletApp.swift
WalletView.swift
SwiftUIWallet 〉 SwiftUIWallet 〉 View 〉 WalletView.swift 〉 WalletView
Preview
Wallet
DISCOVER
CHASE
AMERICAN EXPRESS
VISA
4242 4242 4242 4242
James Hayward Valid Thru 3/23
83%

Figure 9. Spreading the cards

Adding a Slide-in Animation

Now that we have completed the layout of the wallet view, it's time to add some animations. The first animation I want to add is a slide-in animation. When the app is first launched, each of the cards slides from the far left of the screen. You may think that this animation is unnecessary but I want to take this opportunity to show you how to create an animation and view transition at the app launch.

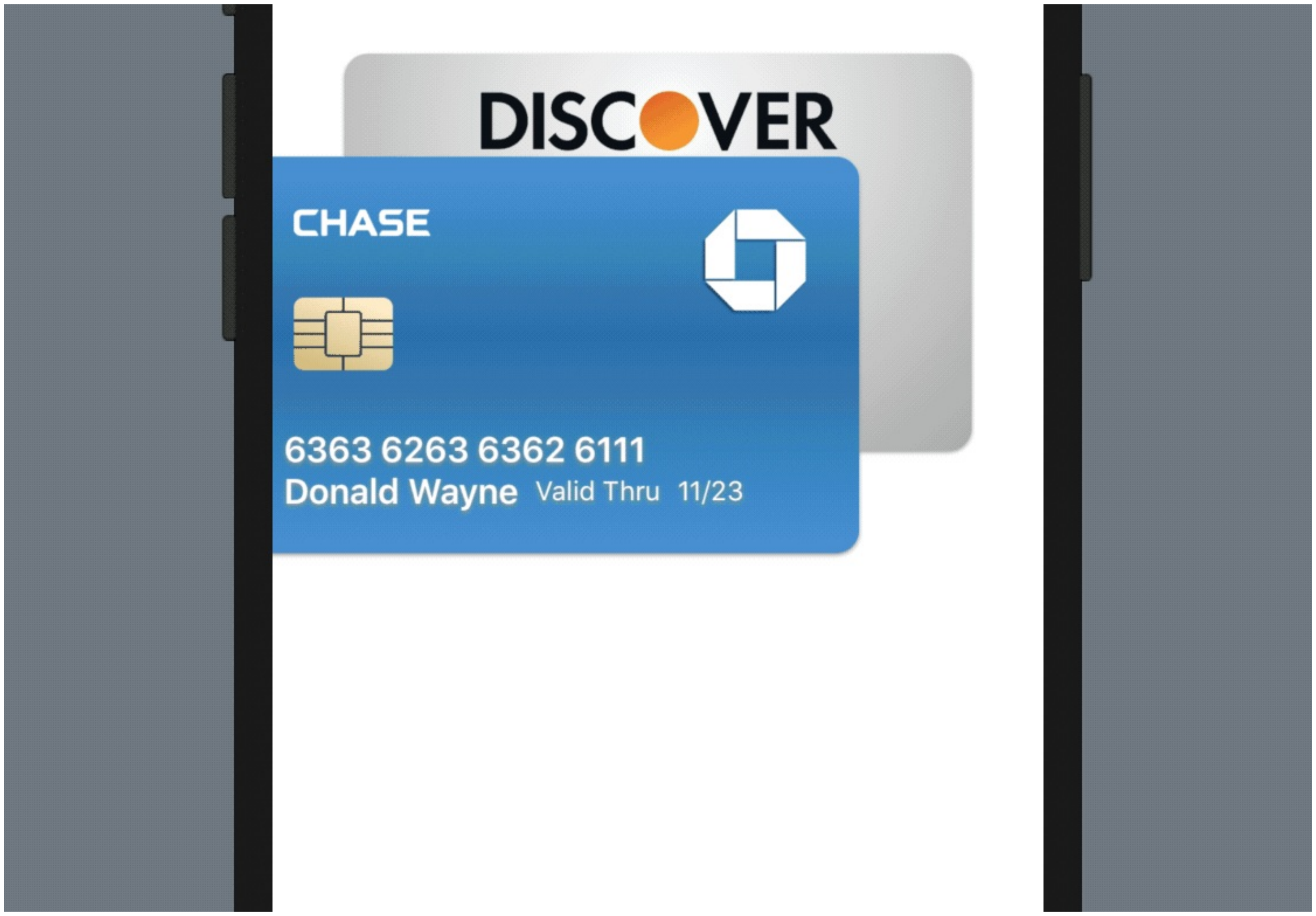


Figure 10. The slide-in animation

Each of the cards is a view. To implement an animation like that displayed in figure 10, we need to attach both the `transition` and `animation` modifiers to the `CardView` like this:

```
CardView(card: card)
    .offset(self.offset(for: card))
    .padding(.horizontal, 35)
    .zIndex(self.zIndex(for: card))
    .transition(AnyTransition.slide.combined(with: .move(edge: .leading)).combined
(with: .opacity))
    .animation(self.transitionAnimation(for: card))
```

For the transition, we combine the default slide transition with the move transition. As mentioned before, the transition will not be animated without the `animation` modifier. This is why we also attach the `animation` modifier. Since each card has its own animation, we create a function called `transitionAnimation(for:)` to compute the animation. Insert the following code to create the function:

```
private func transitionAnimation(for card: Card) -> Animation {
    var delay = 0.0

    if let index = index(for: card) {
        delay = Double(cards.count - index) * 0.1
    }

    return Animation.spring(response: 0.1, dampingFraction: 0.8, blendDuration: 0.
02).delay(delay)
}
```

In fact, all the cards have a similar animation, which is a spring animation. The difference is in the delay. The last card of the deck will appear first, thus the value of the delay should be the smallest. The formula below is how we compute the delay for each of the cards. The smaller the index, the longer the delay.

```
delay = Double(cards.count - index) * 0.1
```

The view transition still doesn't work because we need some way to trigger the transition. Let's declare a state variable at the beginning of `CardView` :

```
@State var isCardPresented = false
```

This variable indicates whether the cards should be presented on screen. By default, it's set to `false` . Next, modify the code of the `ZStack` like this:

```
ZStack {
    if isCardPresented {
        ForEach(cards) { card in
            CardView(card: card)
                .offset(self.offset(for: card))
                .padding(.horizontal, 35)
                .zIndex(self.zIndex(for: card))
                .transition(AnyTransition.slide.combined(with: .move(edge: .leadin
g)).combined(with: .opacity))
                .animation(self.transitionAnimation(for: card))
        }
    }
}
```

We wrap the `ForEach` loop with a `if` clause. Since the value of `isCardPresented` is set to `false`, all the card views are not displayed by default.

So, how can we trigger the view transition of the card view at the app launch? The trick is to use `onAppear` and attach it to the `ZStack`:

```
.onAppear {
    self.isCardPresented.toggle()
}
```

When the `ZStack` appears, we change the value of `isCardPresented` from `false` to `true`. This will trigger the view animation of the cards. After applying the changes, hit the *Play* button in the preview canvas to see the result.

Handling the Tap Gesture and Displaying the Transaction History

When a user taps a card, the app moves the selected card upward and brings up the transaction history. For those non-selected cards, they are moved off the screen.

To implement this feature, we need two more state variables. Declare these variables in `WalletView`:

```
@State var isCardPressed = false
@State var selectedCard: Card?
```

The `isCardPressed` variable indicates if a card is selected, while the `selectedCard` variable stores the card selected by the user.

```
.gesture(
    TapGesture()
        .onEnded({ _ in
            withAnimation {
                self.isCardPressed.toggle()
                self.selectedCard = self.isCardPressed ? card : nil
            }
        })
)
```

To handle the tap gesture, we attach the above `gesture` modifier to the `CardView` (just below `.animation(self.transitionAnimation(for: card))`) and use the built-in `TapGesture` to capture the tap event. In the code block, we simply toggle the state of `isCardPressed` and set the current card to the `selectedCard` variable.

To move the selected card (and those underneath) upward and the rest of the cards move off the screen, update the `offset(for:)` function like this:

```
private func offset(for card: Card) -> CGSize {

    guard let cardIndex = index(for: card) else {
        return CGSize()
    }

    if isCardPressed {
        guard let selectedCard = self.selectedCard,
            let selectedCardIndex = index(for: selectedCard) else {
                return .zero
        }

        if cardIndex >= selectedCardIndex {
            return .zero
        }

        let offset = CGSize(width: 0, height: 1400)

        return offset
    }

    return CGSize(width: 0, height: -50 * CGFloat(cardIndex))
}
```

We added an `if` clause to check if a card is selected. If the given card is the card selected by the user, we set the offset to `.zero`. For those cards that are right below the selected card, we will also move them upward. This is why we set the offset to `.zero`. For the rest of the cards, we move them off the screen. Therefore, the vertical offset is set to `1400` points.

Now we are ready to write the code for bringing up the transaction history view. As mentioned at the very beginning, the starter project comes with this transaction history view. Therefore, you do not need to build it yourself.

We can use the `isCardPressed` state variable to determine if the transaction history view is shown or not. Insert the following right before `Spacer()`:

```
if isCardPressed {
    TransactionHistoryView(transactions: testTransactions)
        .padding(.top, 10)
        .transition(.move(edge: .bottom))
        .animation(Animation.easeOut(duration: 0.15).delay(0.1))
}
```

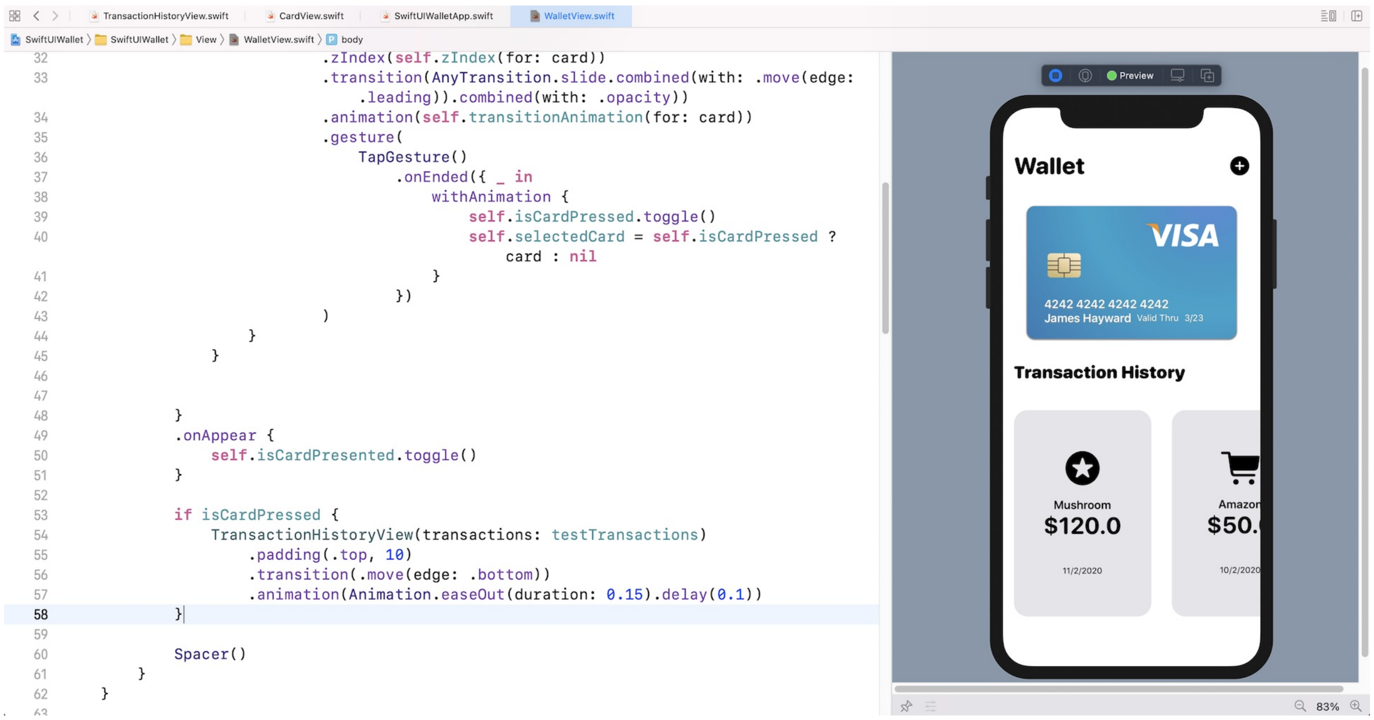
In the code above, we set the transition to `.move` to bring the view up from the bottom of the screen. Feel free to change it to suit your preference.

The app should now work if you test it in the preview pane or run it in a simulator. However, you may find the animation is a bit laggy after you tap a card. This is due to the transition animation that we set in the previous section. For each of the cards (except the first card), we introduced a delay to the animation. So, when a card is selected, the same series of animations is executed.

To make the animation more fluid, update the modifiers of the `CardView` by inserting the `animation` and `scaleEffect` modifiers after `offset` like this:

```
CardView(card: card)
    .offset(self.offset(for: card))
    .animation(.default)
    .scaleEffect(1.0)
    .padding(.horizontal, 35)
    .
    .
    .
```

SwiftUI allows you to apply more than one animation to a view. In the code above, we instruct SwiftUI to use the default animation to animate the change of an offset. For the view transition, we keep using the original animation created earlier.



```
                            .zIndex(self.zIndex(for: card))
                            .transition(AnyTransition.slide.combined(with: .move(edge:
                                .leading)).combined(with: .opacity))
                            .animation(self.transitionAnimation(for: card))
                            .gesture(
                                TapGesture()
                                    .onEnded({ _ in
                                        withAnimation {
                                            self.isCardPressed.toggle()
                                            self.selectedCard = self.isCardPressed ?
                                                card : nil
                                        }
                                    })
                            )
                    }
                }


            }
            .onAppear {
                self.isCardPresented.toggle()
            }

            if isCardPressed {
                TransactionHistoryView(transactions: testTransactions)
                    .padding(.top, 10)
                    .transition(.move(edge: .bottom))
                    .animation(Animation.easeOut(duration: 0.15).delay(0.1))
            }

            Spacer()
        }
    }
```

Figure 11. Displaying the transaction history

Rearranging the Cards Using the Drag Gesture

Now comes the core part of this chapter. Let's see how to rearrange the card deck with the drag gesture. First, let me describe how this feature works in detail:

1. To initiate the dragging action, the user must tap and hold the card. A simple tap will only bring up the transaction history view.
2. Once the user successfully holds a card, the app will move it a little upward. This is the feedback that we want to give to users, telling them we are ready to drag the card around.
3. As the user drags the card, the user should be able to move it across the deck.
4. After the user releases the card at a certain position, the app will update the position of all the cards in the card deck.

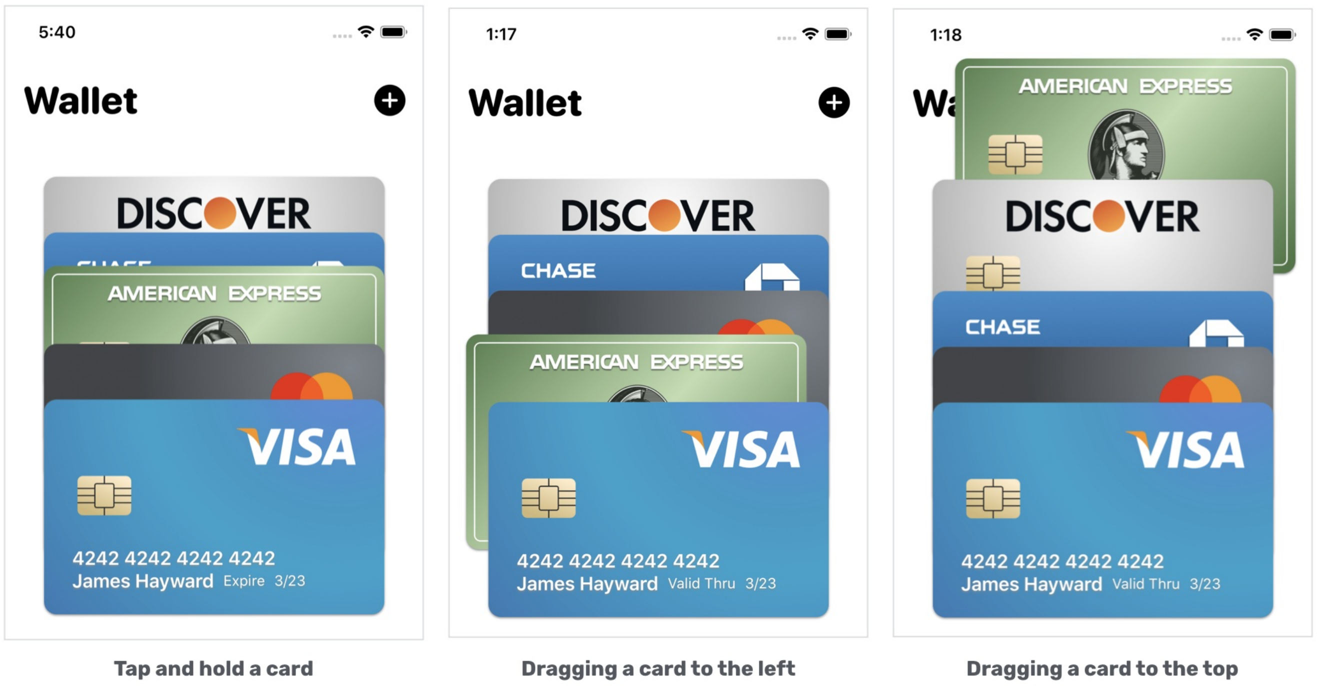


Figure 12. Moving a card across the deck using the drag gesture

Handling the Long Press and Drag Gestures

Now that you understand what we are going to do, let's move onto the implementation. If you forgot how SwiftUI handles gestures, please go back and read chapter 17. Most of the techniques that we will use have been discussed in that chapter.

To begin, insert the following code in `WalletView.swift` to create the `DragState` enum so that we can easily keep track of the drag state:

```
enum DragState {
    case inactive
    case pressing(index: Int? = nil)
    case dragging(index: Int? = nil, translation: CGSize)

    var index: Int? {
        switch self {
        case .pressing(let index), .dragging(let index, _):
            return index
        case .inactive:
            return nil
        }
    }
    var translation: CGSize {
        switch self {
        case .inactive, .pressing:
            return .zero
        case .dragging(_, let translation):
            return translation
        }
    }

    var isPressing: Bool {
        switch self {
        case .pressing, .dragging:
            return true
        case .inactive:
            return false
        }
    }

    var isDragging: Bool {
        switch self {
        case .dragging:
            return true
        case .inactive, .pressing:
            return false
        }
    }
}
```

Next, declare a state variable in `WalletView` to keep track of the drag state:

```
@GestureState private var dragState = DragState.inactive
```

If you've read the chapter about SwiftUI gestures, you should already know how to detect a long press and drag gesture. However, this time it will be a bit different. We need to handle the tap gesture, the drag, and the long press gesture at the same time. Additionally, the app should ignore the tap gesture if the long press gesture is detected.

Now update the `gesture` modifier of the `CardView` like this:

```
.gesture(
    TapGesture()
        .onEnded({ _ in
            withAnimation(.default) {
                self.isCardPressed.toggle()
                self.selectedCard = self.isCardPressed ? card : nil
            }
        })
        .exclusively(before: LongPressGesture(minimumDuration: 0.05)
        .sequenced(before: DragGesture())
        .updating(self.$dragState, body: { (value, state, transaction) in
            switch value {
            case .first(true):
                state = .pressing(index: self.index(for: card))
            case .second(true, let drag):
                state = .dragging(index: self.index(for: card), translation: drag?
.translation ?? .zero)
            default:
                break
            }

        })
        .onEnded({ (value) in

            guard case .second(true, let drag?) = value else {
                return
            }

            // Rearrange the cards
        })

    )
)
```

SwiftUI allows you to combine multiple gestures exclusively. In the code above, we tell SwiftUI to either capture the tap gesture or the long press gesture. In other words, SwiftUI will ignore the long press gesture once the tap gesture is detected.

The code for the tap gesture is exactly the same as our previous code. The drag gesture is sequenced after the long press gesture. In the `updating` function, we set the state of the drag, the translation, and the card's index to the `dragState` variable defined earlier. I'm not going to explain the code in detail as it was covered in chapter 17.

Before you can drag the card, you have to update the `offset(for:)` function like this:

```
private func offset(for card: Card) -> CGSize {

    guard let cardIndex = index(for: card) else {
        return CGSize()
    }

    if isCardPressed {
        guard let selectedCard = self.selectedCard,
            let selectedCardIndex = index(for: selectedCard) else {
                return .zero
        }

        if cardIndex >= selectedCardIndex {
            return .zero
        }

        let offset = CGSize(width: 0, height: 1400)

        return offset
    }

    // Handle dragging
    var pressedOffset = CGSize.zero
    var dragOffsetY: CGFloat = 0.0

    if let draggingIndex = dragState.index,
        cardIndex == draggingIndex {
        pressedOffset.height = dragState.isPressing ? -20 : 0

        switch dragState.translation.width {
        case let width where width < -10: pressedOffset.width = -20
        case let width where width > 10: pressedOffset.width = 20
        default: break
        }
```

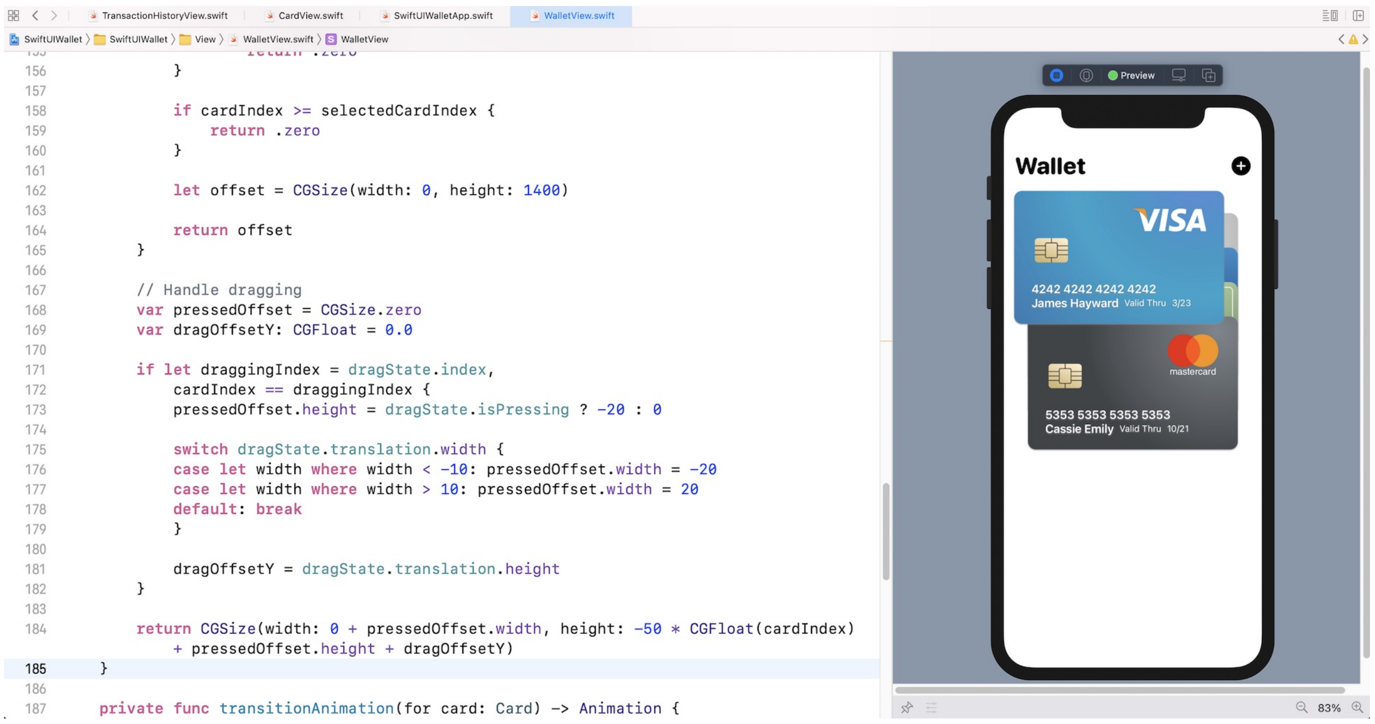
```
            dragOffsetY = dragState.translation.height
        }

        return CGSize(width: 0 + pressedOffset.width, height: -50 * CGFloat(cardIndex)
     + pressedOffset.height + dragOffsetY)
    }
```

We added a block of code to handle the dragging. Please bear in the mind that only the selected card is draggable. Therefore, we need to check if the given card is the one being dragged by the user before making the offset change.

Earlier, we stored the card's index in the `dragState` variable. So, we can easily compare the given card index with the one stored in `dragState` to figure out which card to drag.

For the dragging card, we add an additional offset both horizontally and vertically.

Run the app to test it out, tap & hold a card and then drag it around.



TransactionHistoryView.swift
CardView.swift
SwiftUIWalletApp.swift
WalletView.swift
SwiftUIWallet 〉 SwiftUIWallet 〉 View 〉 WalletView.swift 〉 WalletView
Preview
Wallet
VISA
4242 4242 4242 4242
James Hayward Valid Thru 3/23
mastercard
5353 5353 5353 5353
Cassie Emily Valid Thru 10/21
83%

```
            }

            if cardIndex >= selectedCardIndex {
                return .zero
            }

            let offset = CGSize(width: 0, height: 1400)

            return offset
        }

        // Handle dragging
        var pressedOffset = CGSize.zero
        var dragOffsetY: CGFloat = 0.0

        if let draggingIndex = dragState.index,
            cardIndex == draggingIndex {
            pressedOffset.height = dragState.isPressing ? -20 : 0

            switch dragState.translation.width {
            case let width where width < -10: pressedOffset.width = -20
            case let width where width > 10: pressedOffset.width = 20
            default: break
            }

            dragOffsetY = dragState.translation.height
        }

        return CGSize(width: 0 + pressedOffset.width, height: -50 * CGFloat(cardIndex)
            + pressedOffset.height + dragOffsetY)
    }

    private func transitionAnimation(for card: Card) -> Animation {
```

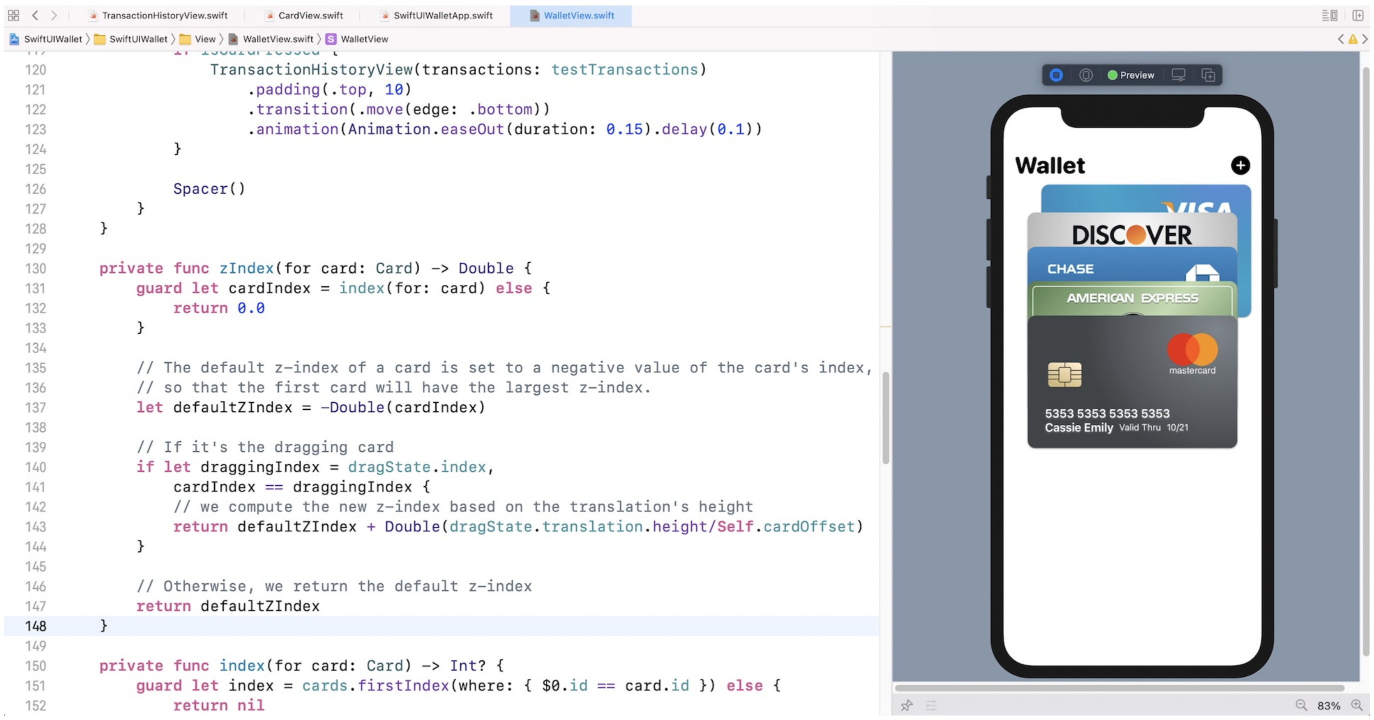
Figure 13. Dragging a card

Currently, you should be able to drag the card, however, the card's z-index doesn't change accordingly. For example, if you drag the Visa card, it always stays on the top of the deck. Let's fix it by updating the `zIndex(for:)` function:

```
private func zIndex(for card: Card) -> Double {
    guard let cardIndex = index(for: card) else {
        return 0.0
    }

    // The default z-index of a card is set to a negative value of the card's inde
x,
    // so that the first card will have the largest z-index.
    let defaultZIndex = -Double(cardIndex)

    // If it's the dragging card
    if let draggingIndex = dragState.index,
        cardIndex == draggingIndex {
        // we compute the new z-index based on the translation's height
        return defaultZIndex + Double(dragState.translation.height/Self.cardOffset
)
    }

    // Otherwise, we return the default z-index
    return defaultZIndex
}
```

The default z-index is still set to the negative value of the card's index. For the dragging card, we need to compute a new z-index as the user drags across the deck. The updated z-index is calculated based on the translation's height and the default offset of the card (i.e. 50 points).

Run the app and drag the Visa card again. Now the z-index is continuously updated as you drag the card.



```
                TransactionHistoryView(transactions: testTransactions)
                    .padding(.top, 10)
                    .transition(.move(edge: .bottom))
                    .animation(Animation.easeOut(duration: 0.15).delay(0.1))
            }

            Spacer()
        }
    }

    private func zIndex(for card: Card) -> Double {
        guard let cardIndex = index(for: card) else {
            return 0.0
        }

        // The default z-index of a card is set to a negative value of the card's index,
        // so that the first card will have the largest z-index.
        let defaultZIndex = -Double(cardIndex)

        // If it's the dragging card
        if let draggingIndex = dragState.index,
            cardIndex == draggingIndex {
            // we compute the new z-index based on the translation's height
            return defaultZIndex + Double(dragState.translation.height/Self.cardOffset)
        }

        // Otherwise, we return the default z-index
        return defaultZIndex
    }

    private func index(for card: Card) -> Int? {
        guard let index = cards.firstIndex(where: { $0.id == card.id }) else {
            return nil
```

Figure 14. Moving the Visa card to the back

Updating the Card Deck

When you release the card, it returns to its original position. So, how can we reorder the cards' after the drag?

The trick here is to update the items of the `cards` array, so as to trigger a UI update. First, we need to mark the `cards` variable as a state variable like this:

```
@State var cards: [Card] = testCards
```

Next, let's create another new function for rearranging the cards:

```
private func rearrangeCards(with card: Card, dragOffset: CGSize) {
    guard let draggingCardIndex = index(for: card) else {
        return
    }

    var newIndex = draggingCardIndex + Int(-dragOffset.height / Self.cardOffset)
    newIndex = newIndex >= cards.count ? cards.count - 1 : newIndex
    newIndex = newIndex < 0 ? 0 : newIndex

    let removedCard = cards.remove(at: draggingCardIndex)
    cards.insert(removedCard, at: newIndex)

}
```

When you drag the card over the adjacent cards, we need to update the z-index once the drag's translation is greater than the default offset. Figure 15 shows the expected behaviour of the drag.

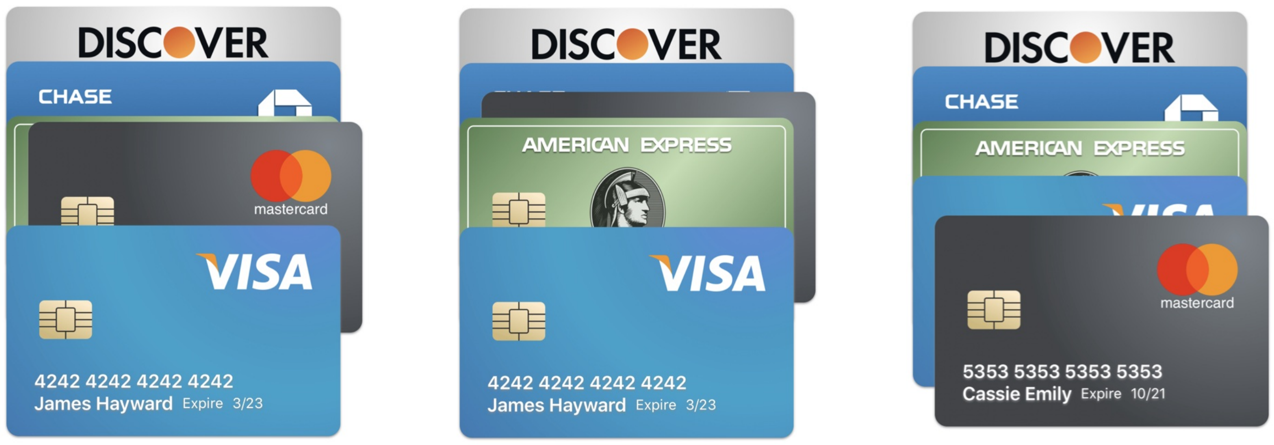


Figure 15. Dragging the mastercard between the adjacent cards

This is the formula we use to compute the updated z-index:

```
var newIndex = draggingCardIndex + Int(-dragOffset.height / Self.cardOffset)
```

Once we have the updated index, the last step is to update the item in the `cards` array by removing the dragging card and insert it into the new position. Since the `cards` array is now a state variable, SwiftUI updates the card deck and renders the animation automatically.

Lastly, insert the following line of code under `// Rearrange the cards` to call the function:

```
self.rearrangeCards(with: card, dragOffset: drag.translation)
```

After that, you are ready to run the app to test it out. Congratulations, You've built the Wallet-like animation.

Summary

After going through this chapter, I hope you have a deeper understanding of SwiftUI animation and view transitions. If you compare SwiftUI with the original UIKit framework, SwiftUI has made it pretty easy to work with animation. Do you remember how you rendered the card animation when the user releases the dragging card? All you need to do is to update the state variable and SwiftUI handles the heavy lifting. That is the power of SwiftUI!

For reference, you can download the complete wallet project here:

- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUIWallet.zip)

Chapter 21
Working with JSON, Slider and Data Filtering

JSON, short for JavaScript Object Notation, is a common data format for data interchange in client-server applications. Even though we are mobile app developers, it's inevitable to work with JSON since nearly all web APIs or backend web services use JSON as the data exchange format.

In this chapter, we will discuss how you can work with JSON while building an app using the SwiftUI framework. If you have never worked with JSON, I would recommend you read this free chapter from our Intermediate programming book. It will explain to you, in detail, the two different approaches in handling JSON in Swift.

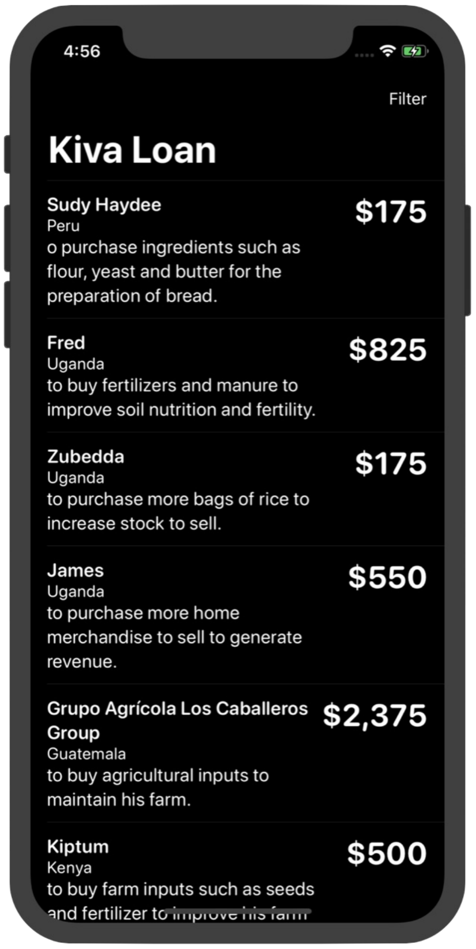

4:56
Filter
Kiva Loan
Sudy Haydee
Peru
o purchase ingredients such as flour, yeast and butter for the preparation of bread.
$175
Fred
Uganda
to buy fertilizers and manure to improve soil nutrition and fertility.
$825
Zubedda
Uganda
to purchase more bags of rice to increase stock to sell.
$175
James
Uganda
to purchase more home merchandise to sell to generate revenue.
$550
Grupo Agrícola Los Caballeros Group
Guatemala
to buy agricultural inputs to maintain his farm.
$2,375
Kiptum
Kenya
to buy farm inputs such as seeds and fertilizer to improve his farm
$500

Figure 1. The demo app

As usual, in order to learn about JSON and its related APIs, you will build a simple JSON app that utilizes a JSON-based API provided by Kiva.org. If you haven't heard of Kiva, it is a non-profit organization with a mission to connect people through lending to alleviate poverty. It lets individuals lend as little as $25 to help create opportunities around the world. Kiva provides free web-based APIs for developers to access their data. For our demo app, we'll call up a free Kiva API to retrieve the most recent fundraising loans and display them in a list view as shown in figure 1.

Additionally, we will demonstrate the usage of a Slider, which is one of the many built-in UI controls provided by SwiftUI. With the slider, you will implement a data filtering option in the app so that users can filter the loan data in the list.

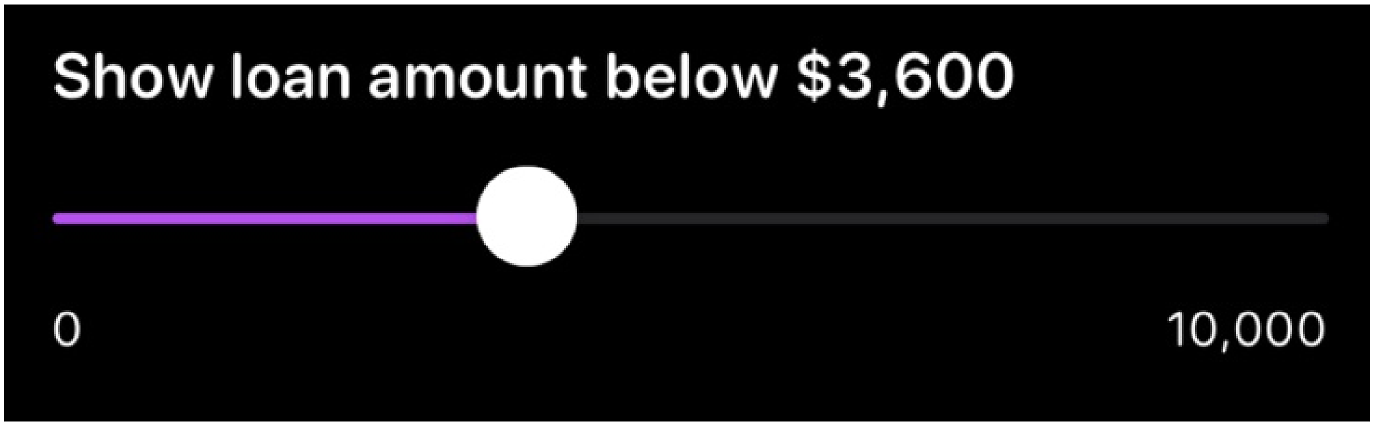


Figure 2. A slider control

Understanding JSON and Codable

First things first, What does the JSON format look like? If you have no idea what JSON looks like, open a browser and point it to the following web API, provided by Kiva:

```
https://api.kivaws.org/v1/loans/newest.json
```

You should see something like this:

```
{
    "loans": [
        {
            "activity": "Fruits & Vegetables",
            "basket_amount": 25,
            "bonus_credit_eligibility": false,
            "borrower_count": 1,
            "description": {
                "languages": [
                    "en"
                ]
            },
            "funded_amount": 0,
            "id": 1929744,
            "image": {
                "id": 3384817,
                "template_id": 1
            },
            "lender_count": 0,
            "loan_amount": 250,
            "location": {
                "country": "Papua New Guinea",
                "country_code": "PG",
                "geo": {
                    "level": "town",
                    "pairs": "-9.4438 147.180267",
                    "type": "point"
                },
                "town": "Port Moresby"
            },
            "name": "Mofa",
            "partner_id": 582,
            "planned_expiration_date": "2020-04-02T08:30:11Z",
            "posted_date": "2020-03-03T09:30:11Z",
            "sector": "Food",
            "status": "fundraising",
            "tags": [],
            "themes": [
                "Vulnerable Groups",
                "Rural Exclusion",
                "Underfunded Areas"
            ],
            "use": "to purchase additional vegetables to increase her currrent sal
es."
        },

        ...

        "paging": {
        "page": 1,
        "page_size": 20,
        "pages": 284,
        "total": 5667
    }
}
```

Your results may not be formatted the same but this is what a JSON response looks like. If you're using Chrome, you can download and install an extension called JSON Formatter (http://link.appcoda.com/json-formatter) to beautify the JSON response.

Alternatively, you can format the JSON data on Mac by using the following command:

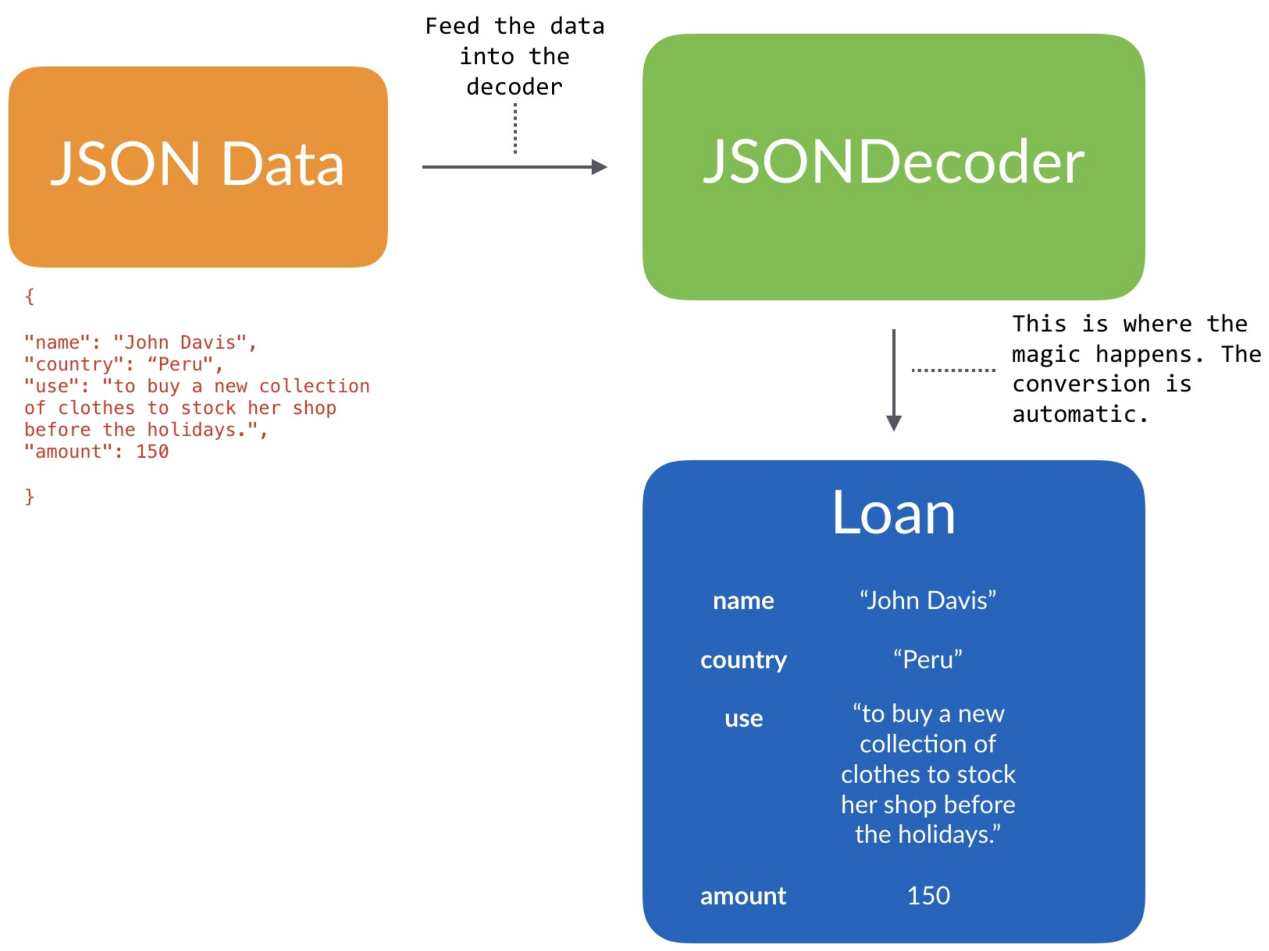
```
curl https://api.kivaws.org/v1/loans/newest.json | python -m json.tool > kiva-loan
s-data.txt
```

This will format the response and save it to a text file.

Now that you have seen JSON, Let's learn how to parse JSON data in Swift. Starting with Swift 4, Apple introduced a new way to encode and decode JSON data by adopting a protocol called `Codable`.

`Codable` simplifies the whole process by offering developers a different way to decode (or encode) JSON. As long as your type conforms to the `Codable` protocol, together with the new `JSONDecoder`, you will be able to decode the JSON data into your specified instances.

Figure 3 illustrates the decoding of sample loan data into an instance of `Loan` using `JSONDecoder`.


Feed the data into the decoder
JSON Data
JSONDecoder
This is where the magic happens. The conversion is automatic.
Loan
name
"John Davis"
country
"Peru"
use
"to buy a new collection of clothes to stock her shop before the holidays."
amount
150

```
{
"name": "John Davis",
"country": "Peru",
"use": "to buy a new collection
of clothes to stock her shop
before the holidays.",
"amount": 150
}
```

Figure 3. JSONDecoder decodes JSON data and convert it into an instance of Loan

Using JSONDecoder and Codable

Before building the demo app, let's try out JSON decoding on Playgrounds. Fire up Xcode and open a new Playground project. Once you have created your Playground project, declare the following `json` variable:

```
let json = """
{

"name": "John Davis",
"country": "Peru",
"use": "to buy a new collection of clothes to stock her shop before the holidays."
,
"amount": 150

}
"""
```

Assuming you're new to JSON parsing, let's make things simple. The above is a simplified JSON response, similar to that shown in the previous section.

To parse the data, declare the `Loan` structure like this:

```
struct Loan: Codable {
    var name: String
    var country: String
    var use: String
    var amount: Int
}
```

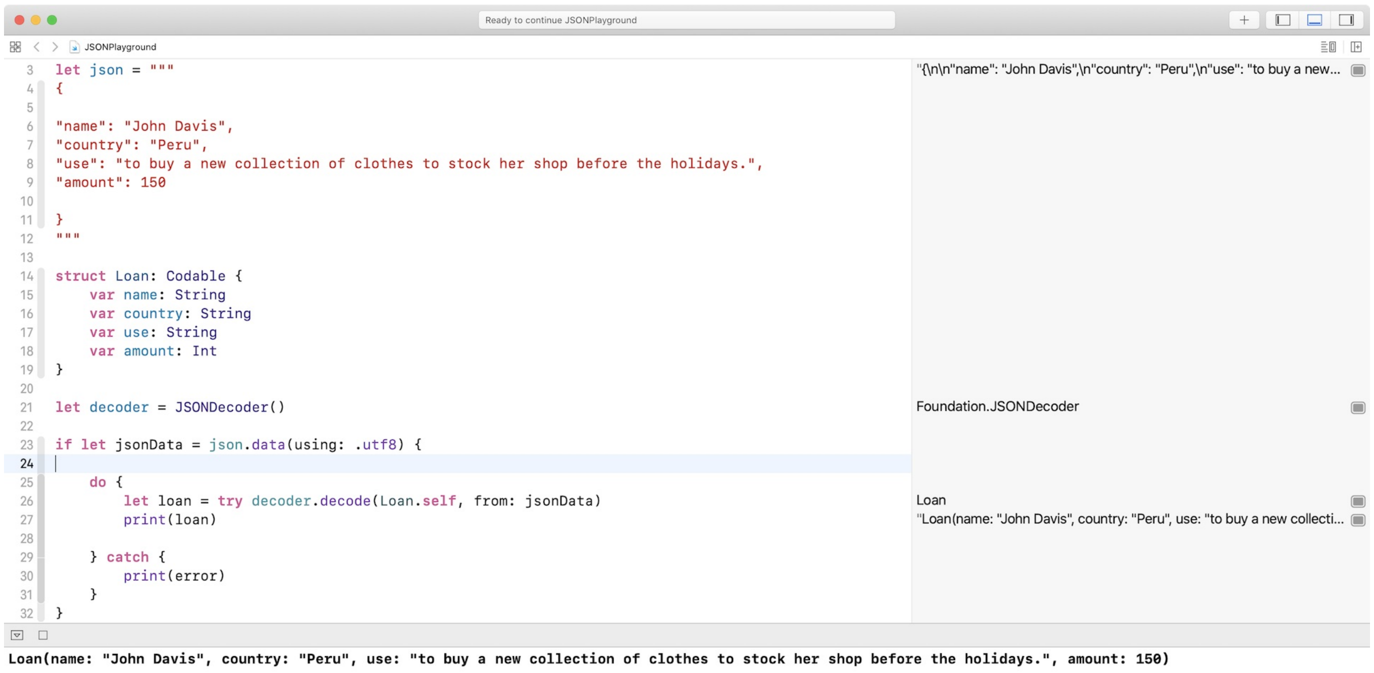
As you can see, the structure adopts the `Codable` protocol. The variables defined in the structure match the keys of the JSON response. This is how you let the decoder know how to decode the data.

Now let's see the magic!

Continue to insert the following code in your Playground file:

```
let decoder = JSONDecoder()

if let jsonData = json.data(using: .utf8) {

    do {
        let loan = try decoder.decode(Loan.self, from: jsonData)
        print(loan)

    } catch {
        print(error)
    }
}
```

If you run the project, you should see a message displayed in the console. That's a `Loan` instance, populated with the decoded values.



```
let json = """
{

"name": "John Davis",
"country": "Peru",
"use": "to buy a new collection of clothes to stock her shop before the holidays.",
"amount": 150

}
"""

struct Loan: Codable {
    var name: String
    var country: String
    var use: String
    var amount: Int
}

let decoder = JSONDecoder()

if let jsonData = json.data(using: .utf8) {

    do {
        let loan = try decoder.decode(Loan.self, from: jsonData)
        print(loan)

    } catch {
        print(error)
    }
}
```

```
"{\n\n"name": "John Davis",\n"country": "Peru",\n"use": "to buy a new...
Foundation.JSONDecoder
Loan
"Loan(name: "John Davis", country: "Peru", use: "to buy a new collecti...
```

```
Loan(name: "John Davis", country: "Peru", use: "to buy a new collection of clothes to stock her shop before the holidays.", amount: 150)
```

Figure 4. Display the decoded loan data in the console

Let's look into the code snippet again. We instantiate an instance of `JSONDecoder` and then convert the JSON string into `Loan`. The magic happened in this line of code:

```
let loan = try decoder.decode(Loan.self, from: jsonData)
```

You just need to call the `decode` method of the decoder with the JSON data and specify the type of value to decode (i.e. `Loan.self`). The decoder will automatically parse the JSON data and convert them into a `Loan` object.

Cool, right?

Working with Custom Property Names

Now, let's jump into something more complicated. What if the name of the property and the key of the JSON are different? How can you define the mapping?

For example, we modify the `json` variable like this:

```
let json = """
{

"name": "John Davis",
"country": "Peru",
"use": "to buy a new collection of clothes to stock her shop before the holidays."
,
"loan_amount": 150

}
"""
```

As you can see, the key *amount* is now *loan_amount*. In order to decode the JSON data, you can modify the property name from `amount` to `loan_amount`. However, we really want to keep the name `amount`. In this case, how can we define the mapping?

To define the mapping between the key and the property name, you are required to declare an enum called `CodingKeys` that has a rawValue of type `String` and conforms to the `CodingKey` protocol.

Now update the `Loan` structure like this:

```
struct Loan: Codable {
    var name: String
    var country: String
    var use: String
    var amount: Int

    enum CodingKeys: String, CodingKey {
        case name
        case country
        case use
        case amount = "loan_amount"
    }
}
```

In the enum, you define all the property names of your model and their corresponding keys in the JSON data. For example, the case `amount` is defined to map to the key `loan_amount`. If both the property name and the key of the JSON data are the same, you can omit the assignment.

Working with Nested JSON Objects

Now that you understand the basics, let's dive even deeper and decode a more realistic JSON response. First, update the `json` variable like this:

```
let json = """
{

"name": "John Davis",
"location": {
"country": "Peru",
},
"use": "to buy a new collection of clothes to stock her shop before the holidays."
,
"loan_amount": 150

}
"""
```

We've added the `location` key that has a nested JSON object with the nested key `country`. So, how do we decode the value of `country` from the nested object?

Let's modify the `Loan` structure like this:

```
struct Loan: Codable {
    var name: String
    var country: String
    var use: String
    var amount: Int

    enum CodingKeys: String, CodingKey {
        case name
        case country = "location"
        case use
        case amount = "loan_amount"
    }

    enum LocationKeys: String, CodingKey {
        case country
    }

    init(from decoder: Decoder) throws {
        let values = try decoder.container(keyedBy: CodingKeys.self)

        name = try values.decode(String.self, forKey: .name)

        let location = try values.nestedContainer(keyedBy: LocationKeys.self, forK
ey: .country)
        country = try location.decode(String.self, forKey: .country)

        use = try values.decode(String.self, forKey: .use)
        amount = try values.decode(Int.self, forKey: .amount)

    }
}
```

Similar to what we have done earlier, we have to define an enum `CodingKeys`. For the case `country`, we specify to map to the key `location`. To handle the nested JSON object, we need to define an additional enumeration. In the code above, we name it

`LocationKeys` and declare the case `country` that matches the key `country` of the nested object.

Since it is not a direct mapping, we need to implement the initializer of the `Decodable` protocol to handle the decoding of all properties. In the `init` method, we first invoke the `container` method of the decoder with `CodingKeys.self` to retrieve the data related to the specified coding keys, which are `name` , `location` , `use` and `amount` .

To decode a specific value, we call the `decode` method with the specific key (e.g. `.name`) and the associated type (e.g. `String.self`). The decoding of the `name` , `use` and `amount` is pretty straightforward. For the `country` property, the decoding is a little bit tricky. We have to call the `nestedContainer` method with `LocationKeys.self` to retrieve the nested JSON object. From the values returned, we further decode the value of `country` .

That is how you decode JSON data with nested objects.

Working with Arrays

The JSON data returned from Kiva API comes with more than one loan. Multiple loans are structured in the form of an array. Let's see how to decode an array of JSON objects using Codable.

First, modify the `json` variable like this:

```
let json = """
{
"loans":
[{
"name": "John Davis",
"location": {
"country": "Paraguay",
},
"use": "to buy a new collection of clothes to stock her shop before the holidays."
,
"loan_amount": 150
},
{
"name": "Las Margaritas Group",
"location": {
"country": "Colombia",
},
"use": "to purchase coal in large quantities for resale.",
"loan_amount": 200
}]
}
"""
```

In the example above, there are two loans in the `json` variable. How do you decode it into an array of `Loan` ?

To do that, declare another struct named `LoanStore` that also adopts `Codable` :

```
struct LoanStore: Codable {
    var loans: [Loan]
}
```

This `LoanStore` only has a `loans` property that matches the key `loans` of the JSON data. And, its type is defined as an array of `Loan` .

To decode the loans, modify this line of code from:

```
let loan = try decoder.decode(Loan.self, from: jsonData)
```

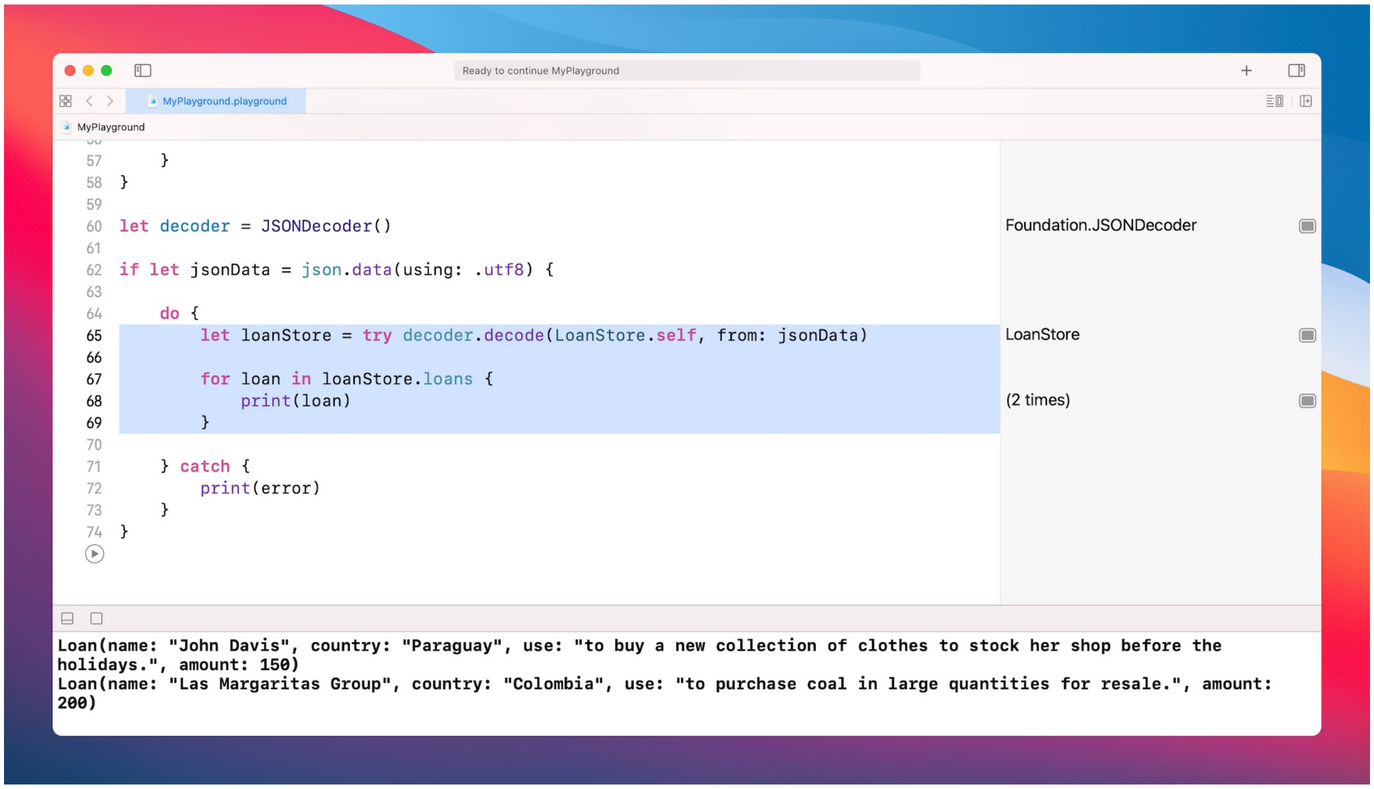
to:

```
let loanStore = try decoder.decode(LoanStore.self, from: jsonData)
```

The decoder will automatically decode the `loans` JSON objects and store them into the `loans` array of `LoanStore`. To print out the loans replace the line `print(loan)` with

```
for loan in loanStore.loans {
    print(loan)
}
```

You should see a similar message as shown in figure 5.



```
    }
}

let decoder = JSONDecoder()

if let jsonData = json.data(using: .utf8) {

    do {
        let loanStore = try decoder.decode(LoanStore.self, from: jsonData)

        for loan in loanStore.loans {
            print(loan)
        }

    } catch {
        print(error)
    }
}
```

```
Loan(name: "John Davis", country: "Paraguay", use: "to buy a new collection of clothes to stock her shop before the
holidays.", amount: 150)
Loan(name: "Las Margaritas Group", country: "Colombia", use: "to purchase coal in large quantities for resale.", amount:
200)
```

Figure 5. Print out the loans array

That's how you decode JSON using Swift. For reference, you can download the Playgrounds project from https://www.appcoda.com/resources/swiftui2/SwiftUIJSONPlayground.zip.

Building the Kiva Loan App

Okay, you should now understand how to handle JSON decoding. Let's begin to build the demo app and see how you apply the skills you just learned.

Assuming you have launched Xcode, go up to the menu and select *File* > *New* > *Projects* to create a new project. As usual, use the *App* template. Name the project *SwiftUIKivaLoan* or whatever name you prefer.

We will start by building the model class that stores all the latest loans retrieved from Kiva. We will handle the implementation of user interface later.

Retrieving the Latest Loans from Kiva

First, create a new file using the *Swift File* template and name it `Loan.swift`. This file stores the `Loan` structure that adopts the `Codable` protocol for JSON decoding.

Insert the following code in the file:

```
struct Loan: Identifiable {
    var id = UUID()
    var name: String
    var country: String
    var use: String
    var amount: Int

    init(name: String, country: String, use: String, amount: Int) {
        self.name = name
        self.country = country
        self.use = use
        self.amount = amount
    }

}
```

```
extension Loan: Codable {
    enum CodingKeys: String, CodingKey {
        case name
        case country = "location"
        case use
        case amount = "loan_amount"
    }

    enum LocationKeys: String, CodingKey {
        case country
    }

    init(from decoder: Decoder) throws {
        let values = try decoder.container(keyedBy: CodingKeys.self)

        name = try values.decode(String.self, forKey: .name)

        let location = try values.nestedContainer(keyedBy: LocationKeys.self, forK
ey: .country)
        country = try location.decode(String.self, forKey: .country)

        use = try values.decode(String.self, forKey: .use)
        amount = try values.decode(Int.self, forKey: .amount)

    }
}
```

The code is almost the same as we discussed in the previous section. We just use an extension to adopt the `Codable` protocol. Other than `Codable`, this structure also adopts the `Identifiable` protocol and has an `id` property default to `UUID()`. Later, we will use SwiftUI's `List` control to present the loans. This is why we make this structure adopt the `Identifiable` protocol.

Next, create another file using the *Swift File* template and name it `LoanStore.swift`. This class is to connect to the Kiva's web API, decode the JSON data, and store them locally.

Let's write the `LoanStore` class step by step, so you can better understand how I came up with the implementation. Insert the following code in `LoanStore.swift`:

```
class LoanStore: Decodable {
    var loans: [Loan] = []
}
```

Later the decoder will decode the `loans` JSON objects and store them into the `loans` array of `LoanStore`. This is why we create the `LoanStore` like above. The code looks very similar to the `LoanStore` structure we created before. However, it adopts the `Decodable` protocol instead of `Codable`.

If you look into the documentation of `Codable`, it is just a type alias of a protocol composition:

```
typealias Codable = Decodable & Encodable
```

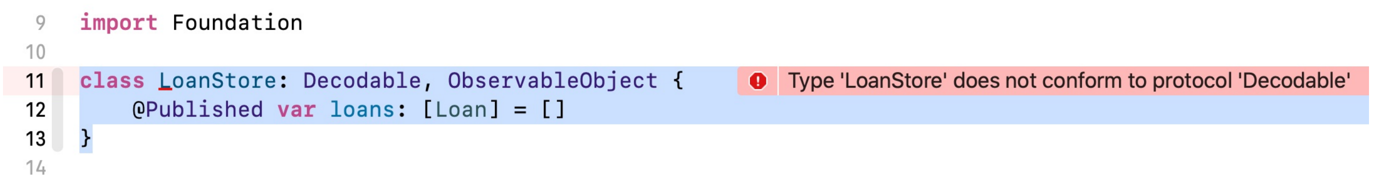
`Decodable` and `Encodable` are the two actual protocols you need to work with. Since `LoanStore` is only responsible for handling the JSON decoding, we adopt the `Decodable` protocol.

As mentioned earlier, we will display the `loans` using a List view. So, other than `Decodable`, we have to adopt the `ObservableObject` protocol and mark the `loans` variable with the `@Published` property wrapper like this:

```
class LoanStore: Decodable, ObservableObject {
    @Published var loans: [Loan] = []
}
```

By doing so, SwiftUI will manage the UI update automatically whenever there is any change to the `loans` variable. If you have forgotten what `ObservableObject` is, please read chapter 14 again.

Once you add the `@Published` property wrapper, Xcode shows you an error. The `Decodable` (or `Codable`) protocol doesn't play well with `@Published`.



```
import Foundation

class LoanStore: Decodable, ObservableObject {
    @Published var loans: [Loan] = []
}
```

Type 'LoanStore' does not conform to protocol 'Decodable'

Figure 6. Xcode error saying that LoanStore doesn't conform to Decodable

To fix the error, requires some extra work. When the `@Published` property wrapper is used, we need to implement the required method of `Decodable` manually. If you look into the documentation (https://developer.apple.com/documentation/swift/decodable), here is the method to adopt:

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

Actually, we've implemented the method before when decoding the nested JSON objects. Now, update the class like this:

```
class LoanStore: Decodable, ObservableObject {
    @Published var loans: [Loan] = []

    enum CodingKeys: CodingKey {
        case loans
    }

    required init(from decoder: Decoder) throws {
        let values = try decoder.container(keyedBy: CodingKeys.self)
        loans = try values.decode([Loan].self, forKey: .loans)
    }

    init() {

    }
}
```

We added the `CodingKeys` enum that explicitly specifies the key to decode. And then, we implemented the custom initializer to handle the decoding.

Okay, the error is now fixed. What's next?

Calling the Web API

So far, we just set up everything for JSON decoding but we haven't consumed the web API. Declare a new variable in the class to store the URL of the Kiva's API:

```
private static var kivaLoanURL = "https://api.kivaws.org/v1/loans/newest.json"
```

Next, insert the following methods in the class:

```
func fetchLatestLoans() {
    guard let loanUrl = URL(string: Self.kivaLoanURL) else {
        return
    }

    let request = URLRequest(url: loanUrl)
    let task = URLSession.shared.dataTask(with: request, completionHandler: { (dat
a, response, error) -> Void in

        if let error = error {
            print(error)
            return
        }

        // Parse JSON data
        if let data = data {
            DispatchQueue.main.async {
                self.loans = self.parseJsonData(data: data)
            }

        }
    })

    task.resume()
}

func parseJsonData(data: Data) -> [Loan] {
    let decoder = JSONDecoder()

    do {

        let loanStore = try decoder.decode(LoanStore.self, from: data)
        self.loans = loanStore.loans

    } catch {
        print(error)
    }

    return loans
}
```

The `fetchLatestLoans()` method connects to the web API by using `URLSession`. Once it receives the data returned by the API, it passes the data to the `parseJsonData` method to decode the JSON and convert the loan data into an array of `Loan`.

You may wonder why we need to wrap the following line of code with `DispatchQueue.main.async`:

```
DispatchQueue.main.async {
    self.loans = self.parseJsonData(data: data)
}
```

When calling the web API, the operation is performed in a background queue. Here, the `loans` variable is marked as `@Published`. That means, for any modification of the variable, SwiftUI will trigger an update of the user interface. UI updates are required to run in the main queue. This is the reason why we wrap it using `DispatchQueue.main.async`.

Implementing the User Interface

Now that we have created the classes ready for retrieving the loan data, let's move onto the implementation of the user interface. To help you remember what the UI looks like, look at the following figure. This is the UI we are going to build.

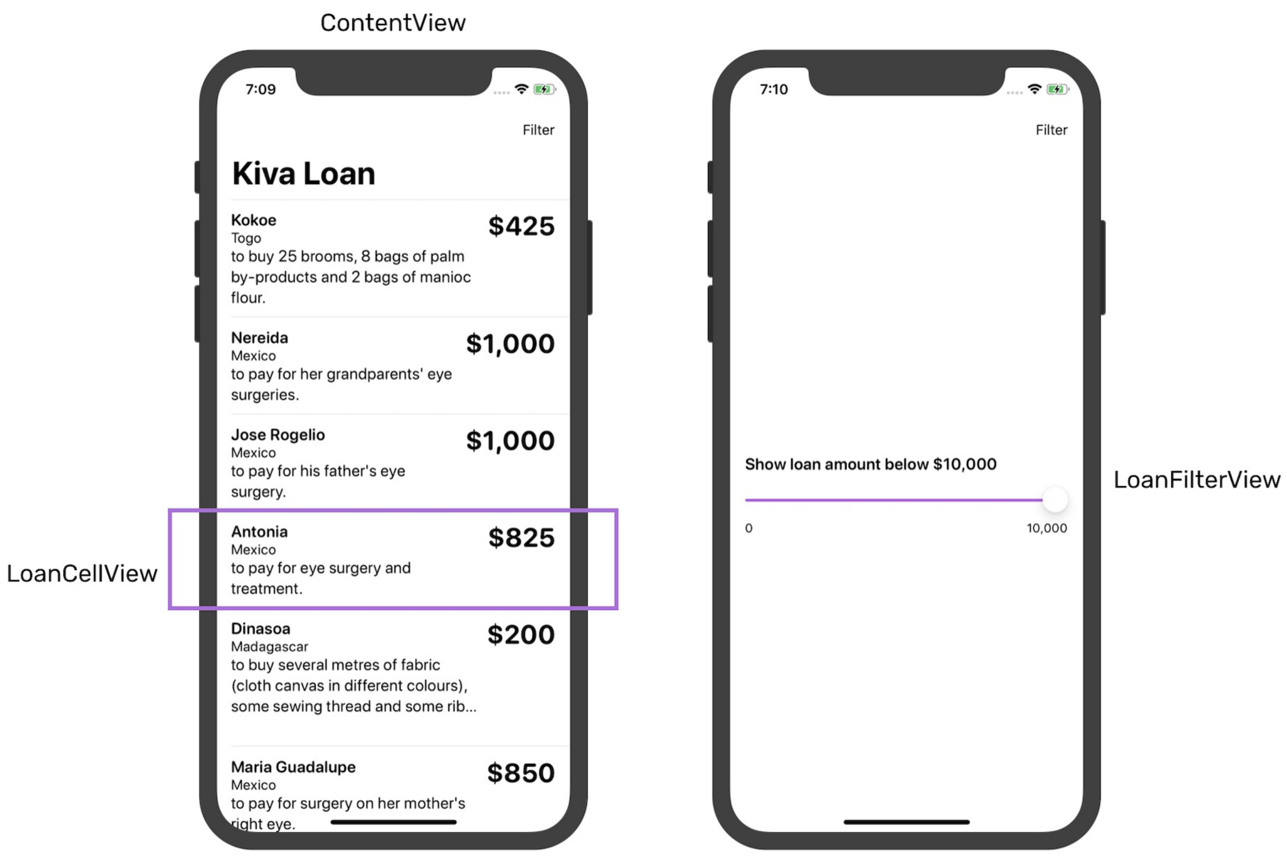

ContentView
7:09
Filter
Kiva Loan
Kokoe
Togo
to buy 25 brooms, 8 bags of palm by-products and 2 bags of manioc flour.
$425
Nereida
Mexico
to pay for her grandparents' eye surgeries.
$1,000
Jose Rogelio
Mexico
to pay for his father's eye surgery.
$1,000
LoanCellView
Antonia
Mexico
to pay for eye surgery and treatment.
$825
Dinasoa
Madagascar
to buy several metres of fabric (cloth canvas in different colours), some sewing thread and some rib...
$200
Maria Guadalupe
Mexico
to pay for surgery on her mother's right eye.
$850
7:10
Filter
Show loan amount below $10,000
0
10,000
LoanFilterView

Figure 7. The user interface of our demo app

And, instead of coding the UI in one file, we will break it down into three views:

- *ContentView.swift* - this is the main view presenting the list of loans
- *LoanCellView.swift* - this is the cell view
- *LoanFilterView.swift* - this is the view showing the filtering option

Let's begin with the cell view. In the project navigator, right click `SwiftUIKivaLoan` and choose *New file...*. Select the *SwiftUI View* template and name the file `LoanCellView.swift`.

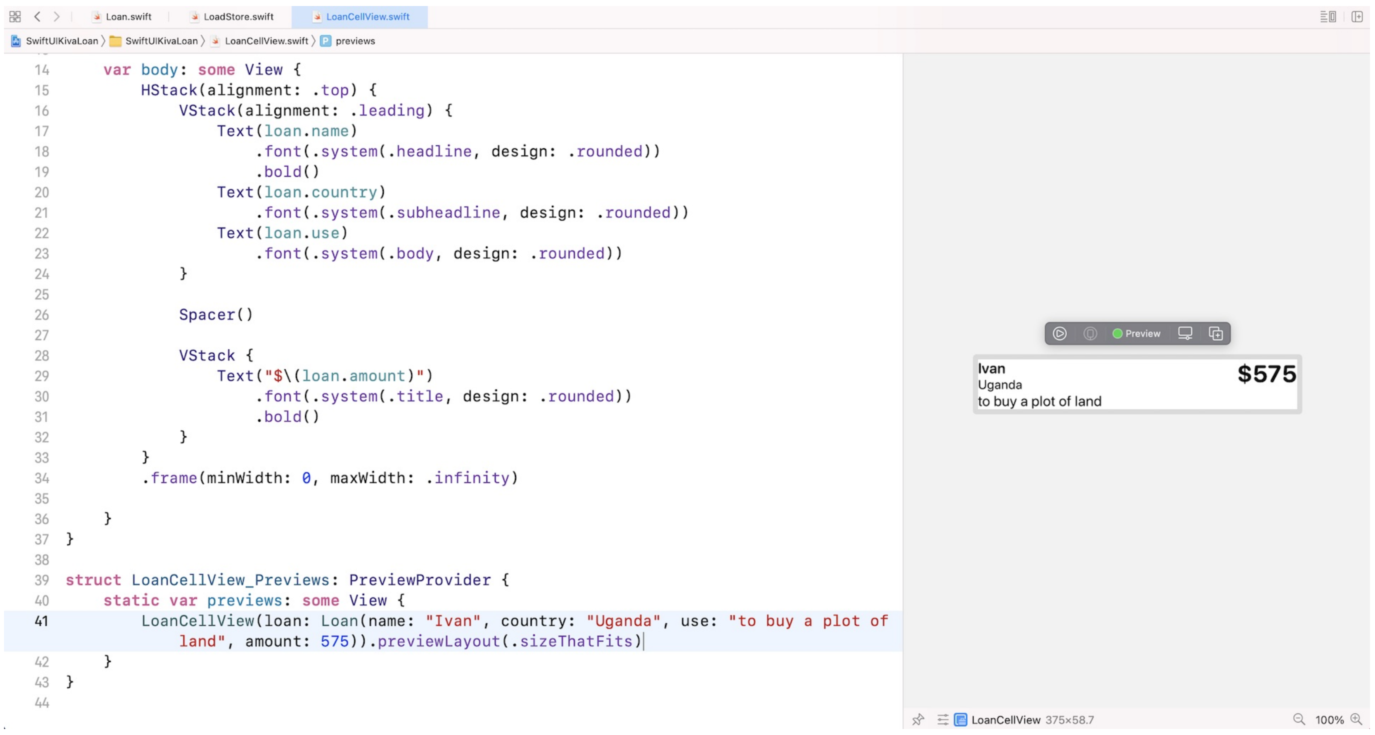
Update the `LoanCellView` like this:

```
struct LoanCellView: View {

    var loan: Loan

    var body: some View {
        HStack(alignment: .top) {
            VStack(alignment: .leading) {
                Text(loan.name)
                    .font(.system(.headline, design: .rounded))
                    .bold()
                Text(loan.country)
                    .font(.system(.subheadline, design: .rounded))
                Text(loan.use)
                    .font(.system(.body, design: .rounded))
            }

            Spacer()

            VStack {
                Text("$\(loan.amount)")
                    .font(.system(.title, design: .rounded))
                    .bold()
            }
        }
        .frame(minWidth: 0, maxWidth: .infinity)

    }
}
```

This view takes in a `Loan` and renders the cell view. The code is self explanatory but if you want to preview the cell view, you will need to modify `LoanCellView_Previews` like this:

```
struct LoanCellView_Previews: PreviewProvider {
    static var previews: some View {
        LoanCellView(loan: Loan(name: "Ivan", country: "Uganda", use: "to buy a pl
ot of land", amount: 575)).previewLayout(.sizeThatFits)
    }
}
```

We instantiate a dummy loan and pass it to the cell view for rendering. Your preview pane should be similar to that shown in figure 8.



```
    var body: some View {
        HStack(alignment: .top) {
            VStack(alignment: .leading) {
                Text(loan.name)
                    .font(.system(.headline, design: .rounded))
                    .bold()
                Text(loan.country)
                    .font(.system(.subheadline, design: .rounded))
                Text(loan.use)
                    .font(.system(.body, design: .rounded))
            }

            Spacer()

            VStack {
                Text("$\(loan.amount)")
                    .font(.system(.title, design: .rounded))
                    .bold()
            }
        }
        .frame(minWidth: 0, maxWidth: .infinity)

    }
}

struct LoanCellView_Previews: PreviewProvider {
    static var previews: some View {
        LoanCellView(loan: Loan(name: "Ivan", country: "Uganda", use: "to buy a plot of
            land", amount: 575)).previewLayout(.sizeThatFits)
    }
}
```

Figure 8. The loan cell view

Now go back to `ContentView.swift` to implement the list view. First, declare a variable named `loanStore`:

```
@ObservedObject var loanStore = LoanStore()
```

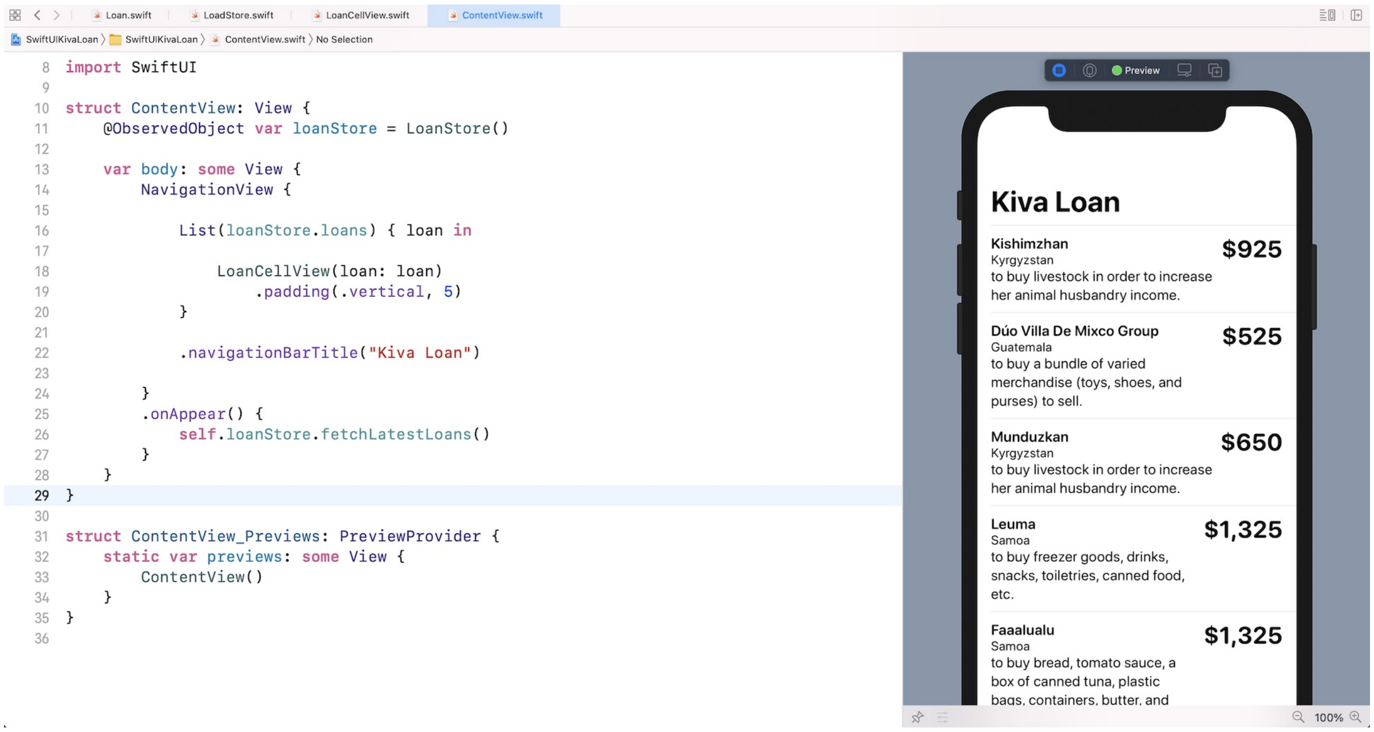
Since we want to observe the change of loan store and update the UI, the `loanStore` is marked with the `@ObservedObject` property wrapper.

Next, update the `body` variable like this:

```
var body: some View {
    NavigationView {

        List(loanStore.loans) { loan in

            LoanCellView(loan: loan)
                .padding(.vertical, 5)
        }

        .navigationBarTitle("Kiva Loan")

    }
    .onAppear() {
        self.loanStore.fetchLatestLoans()
    }
}
```

If you've read chapter 10 and 11, you should understand how to present a list view and embed it in a navigation view. That's what the code above does. The `onAppear()` function will be invoked when the view appears. And, we call up the `fetchLatestLoans()` method to retrieve the latest loans from Kiva.

Now run the app in the preview (press the play button) or on a simulator. You should be able to see the loan records.



```
import SwiftUI

struct ContentView: View {
    @ObservedObject var loanStore = LoanStore()

    var body: some View {
        NavigationView {

            List(loanStore.loans) { loan in

                LoanCellView(loan: loan)
                    .padding(.vertical, 5)
            }

            .navigationBarTitle("Kiva Loan")

        }
        .onAppear() {
            self.loanStore.fetchLatestLoans()
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Kiva Loan

Kishimzhan
Kyrgyzstan
to buy livestock in order to increase her animal husbandry income.
$925

Dúo Villa De Mixco Group
Guatemala
to buy a bundle of varied merchandise (toys, shoes, and purses) to sell.
$525

Munduzkan
Kyrgyzstan
to buy livestock in order to increase her animal husbandry income.
$650

Leuma
Samoa
to buy freezer goods, drinks, snacks, toiletries, canned food, etc.
$1,325

Faaalualu
Samoa
to buy bread, tomato sauce, a box of canned tuna, plastic bags, containers, butter, and
$1,325

Figure 9. Presenting the loans in a list view

Creating the Filter View with a Slider

Before we finish this chapter, I want to show you how to implement a filter feature. This filter function allows users to define a maximum loan amount and only display the records below that value. Figure 7 shows a sample filter view. Users can use a slider to configure the maximum amount.

Again, we want our code to be better organized. So, create a new file for the filter view and name it `LoanFilterView.swift` .

Next update the `LoanFilterView` struct like this:

```
struct LoanFilterView: View {

    @Binding var amount: Double

    var minAmount = 0.0
    var maxAmount = 10000.0

    var body: some View {
        VStack(alignment: .leading) {

            Text("Show loan amount below $\(Int(amount))")
                .font(.system(.headline, design: .rounded))

            HStack {

                Slider(value: $amount, in: minAmount...maxAmount, step: 100)
                    .accentColor(.purple)

            }

            HStack {
                Text("\(Int(minAmount))")
                    .font(.system(.footnote, design: .rounded))

                Spacer()

                Text("\(Int(maxAmount))")
                .font(.system(.footnote, design: .rounded))
            }

        }
        .padding(.horizontal)
        .padding(.bottom, 10)
    }
}
```

I assume you fully understand stack views. Therefore, I'm not going to discuss how they are used to create the layout. But let's talk a bit more about the Slider control. It's a standard component provided by SwiftUI. You can instantiate the slider by passing it the binding, range, and step of the slider. The binding holds the current value of the slider. Here is sample code for creating a slider:

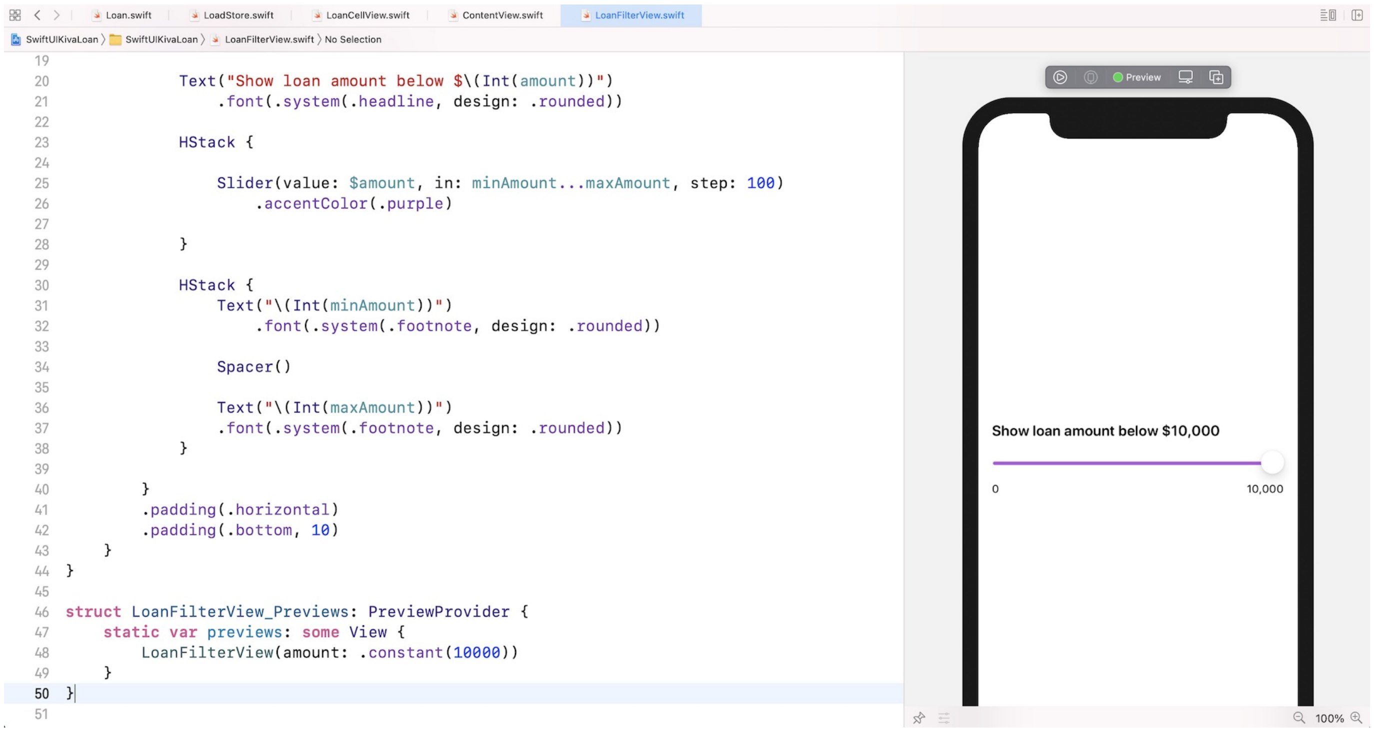
```
Slider(value: $amount, in: minAmount...maxAmount, step: 100)
```

The step controls the amount of change when the user drags the slider. If you let the user have finer control, set the step to a smaller number. For the code above, we set it to 100.

In order to preview the filter view, update the `FilterView_Previews` like this:

```
struct LoanFilterView_Previews: PreviewProvider {
    static var previews: some View {
        LoanFilterView(amount: .constant(10000))
    }
}
```

Now your preview should look like figure 10.


Loan.swift
LoadStore.swift
LoanCellView.swift
ContentView.swift
LoanFilterView.swift
SwiftUIKivaLoan 〉 SwiftUIKivaLoan 〉 LoanFilterView.swift 〉 No Selection
Preview
Show loan amount below $10,000
0
10,000
100%

```

            Text("Show loan amount below $\(Int(amount))")
                .font(.system(.headline, design: .rounded))

            HStack {

                Slider(value: $amount, in: minAmount...maxAmount, step: 100)
                    .accentColor(.purple)

            }

            HStack {
                Text("\(Int(minAmount))")
                    .font(.system(.footnote, design: .rounded))

                Spacer()

                Text("\(Int(maxAmount))")
                .font(.system(.footnote, design: .rounded))
            }

        }
        .padding(.horizontal)
        .padding(.bottom, 10)
    }
}

struct LoanFilterView_Previews: PreviewProvider {
    static var previews: some View {
        LoanFilterView(amount: .constant(10000))
    }
}

```

Figure 10. The filter view for setting the display criteria

Okay, we've implemented the filter view. However, we haven't implemented the actual logic for filtering the records. Let's enhance the `LoanStore.swift` to power it with the filter function.

First, declare the following variable which is used to store a copy of the loan records for the filter operation:

```
private var cachedLoans: [Loan] = []
```

To save the copy, insert the following line of code after `self.loans = self.parseJsonData(data: data)`:

```
self.cachedLoans = self.loans
```

Lastly, create a new function for the filtering:

```
func filterLoans(maxAmount: Int) {
    self.loans = self.cachedLoans.filter { $0.amount < maxAmount }
}
```

This function takes in the value of maximum amount and filter those loan items that are below this limit.

Cool! We are almost done.

Let's go back to `ContentView.swift` to present the filter view. What we are going to do is add a navigation bar button at the top-right corner. When a user taps this button, the app presents the filter view.

Let's first declare two state variables:

```
@State private var filterEnabled = false
@State private var maximumLoanAmount = 10000.0
```

The `filterEnabled` variable stores the current state of the filter view. It's set to `false` by default indicating that the filter view is not shown. The `maximumLoanAmount` stores the maximum loan amount for display. Any loan records with an amount larger than this limit will be hidden.

Next, insert the following code right below `NavigationView {`

```
if self.filterEnabled {
    LoanFilterView(amount: self.$maximumLoanAmount)
        .transition(.opacity)
}
```

When `filterEnabled` is set to `true`, the app will overlay the loan filter view on the list view. What's left is the navigation bar button. Insert the following code and place it after `.navigationBarTitle("Kiva Loan")`:

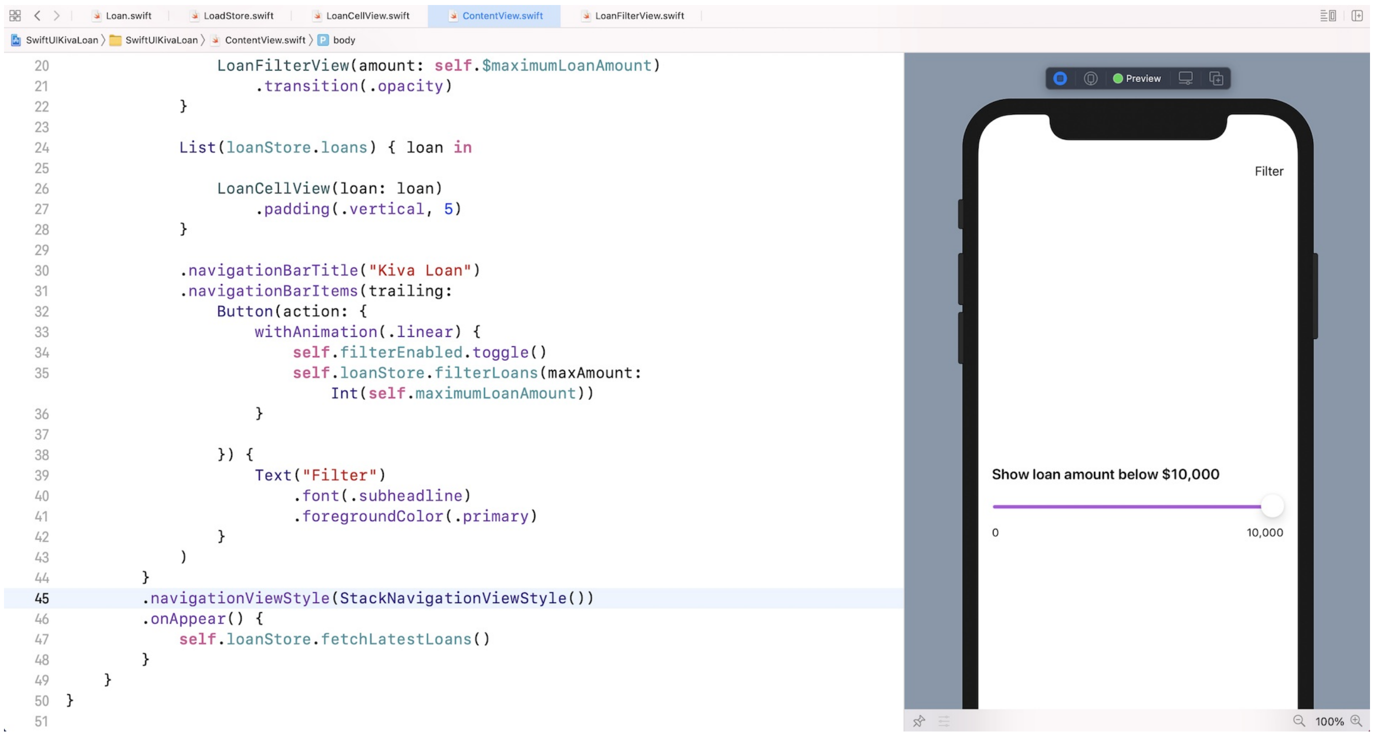
```
.navigationBarItems(trailing:
    Button(action: {
        withAnimation(.linear) {
            self.filterEnabled.toggle()
            self.loanStore.filterLoans(maxAmount: Int(self.maximumLoanAmount))
        }

    }) {
        Text("Filter")
            .font(.subheadline)
            .foregroundColor(.primary)
    }
)
```

This adds a navigation bar button at the top-right corner. When the button is tapped, we toggle the value of `filterEnabled` to show/hide the filter view. Additionally, we call the `filterLoans` function to filter the loan item.

Lastly, attach the `.navigationViewStyle` modifier to change the view's style back to `StackNavigationViewStyle`.

```
.navigationViewStyle(StackNavigationViewStyle())
```

Now run the app to test it. You should see a filter button on the navigation bar. Tap it once to bring up the filter view. You can then set a new limit (e.g. $500). Tap the button again and the app will only show you the loan records that are below $500.



```
                LoanFilterView(amount: self.$maximumLoanAmount)
                    .transition(.opacity)
            }

            List(loanStore.loans) { loan in

                LoanCellView(loan: loan)
                    .padding(.vertical, 5)
            }

            .navigationBarTitle("Kiva Loan")
            .navigationBarItems(trailing:
                Button(action: {
                    withAnimation(.linear) {
                        self.filterEnabled.toggle()
                        self.loanStore.filterLoans(maxAmount:
                            Int(self.maximumLoanAmount))
                    }

                }) {
                    Text("Filter")
                        .font(.subheadline)
                        .foregroundColor(.primary)
                }
            )
        }
        .navigationViewStyle(StackNavigationViewStyle())
        .onAppear() {
            self.loanStore.fetchLatestLoans()
        }
    }
}
```

Figure 11. Presenting the filter view

Summary

We covered quite a lot in this chapter. You should know how to consume web APIs, parse the JSON content, and present the data in a list view. We also briefly covered the usage of the Slider control.

If you've developed an app using UIKit before, you will be amazed by the simplicity of SwiftUI. Take a look at the code of ContentView again. It only takes around 40 lines of code to create the list view. Most importantly, you don't need to handle the UI update manually and pass the data around. Everything just works behind the scenes.

For reference, you can download the complete loan project here:

- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUIKivaLoan.zip)

Chapter 22
Building a ToDo App with Core Data

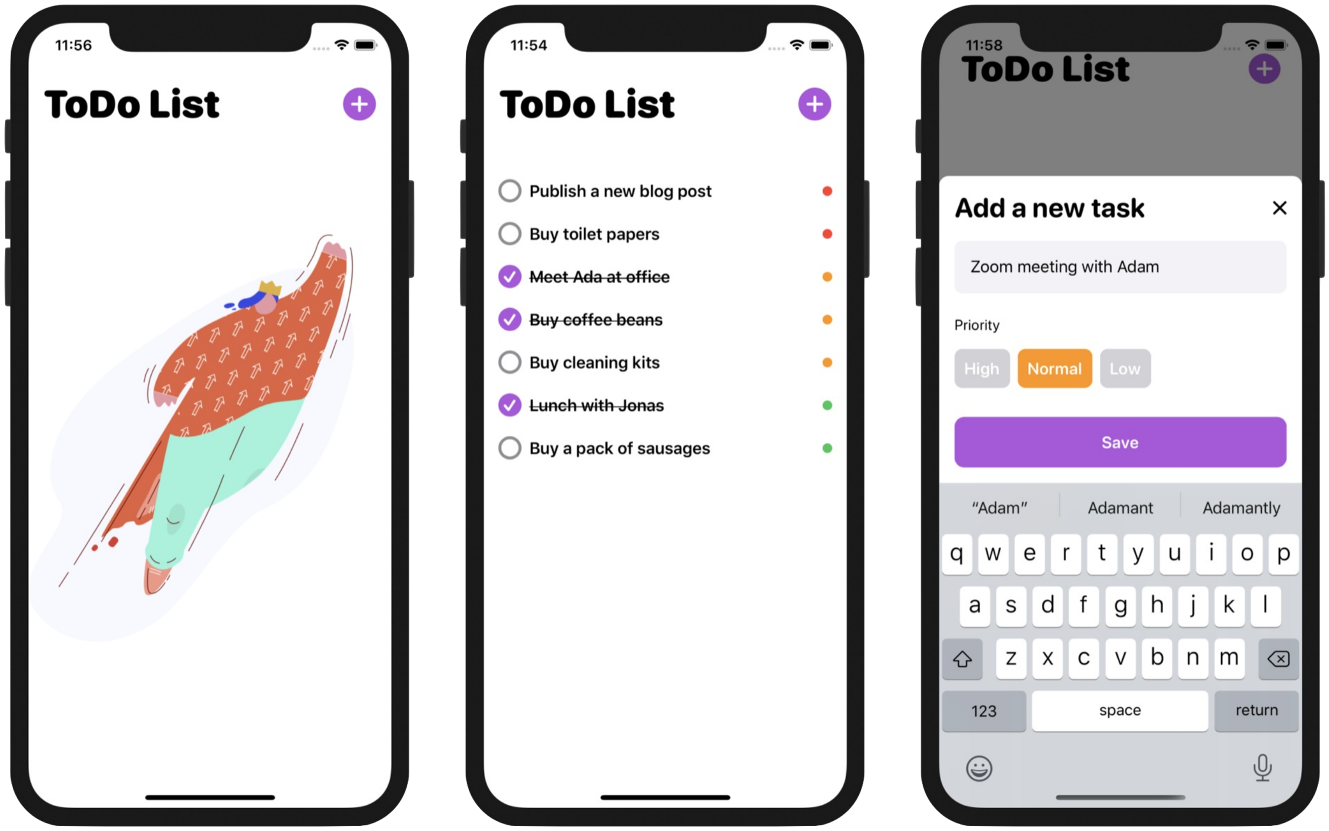
One common question of iOS app development is how do we work with Core Data and SwiftUI to save data permanently in the built-in database. In this chapter, we will answer this question by building a ToDo app.

Since the ToDo demo app makes use of List and Combine to handle the data presentation and sharing, I'll assume that you've read the following chapters:

- Chapter 7 - Understanding State and Binding
- Chapter 10 - Understanding Dynamic List, ForEach and Identifiable
- Chapter 14 - Data Sharing with Combine and Environment Objects

If you haven't done so or forgot what Combine and Environment Objects are, go back and read the chapters again.

What are we going to do in this chapter to understand Core Data? Instead of building the ToDo app from scratch, I've already built the core parts of the app. However, it can't save data permanently. To be more specific, it can only save the to-do items in an array. Whenever the user closes the app and starts it again, all the data is gone. We will modify the app and convert it to use Core Data for saving the data permanently to the local database. Figure 1 shows some sample screenshots of the ToDo app.



Publish a new blog post
Buy toilet papers
~~Meet Ada at office~~
~~Buy coffee beans~~
Buy cleaning kits
~~Lunch with Jonas~~
Buy a pack of sausages

Zoom meeting with Adam

Priority: High [✓ Normal] Low

Figure 1. The ToDo demo app

Before we perform the modification, I will walk you through the starter project so you fully understand how the code works. Other than Core Data, you will also learn how to customize the style of a toggle. Take a look at the screenshots above. The checkbox is actually a toggle view of SwiftUI. I will show you how to create these checkboxes by customizing the Toggle's style.

We've got a lot to cover in this chapter, so let's get started!

Understanding Core Data

Before we check out the starter project of the ToDo app, let me give you a quick introduction to Core Data and how you're going to work with it in SwiftUI projects.

What is Core Data?

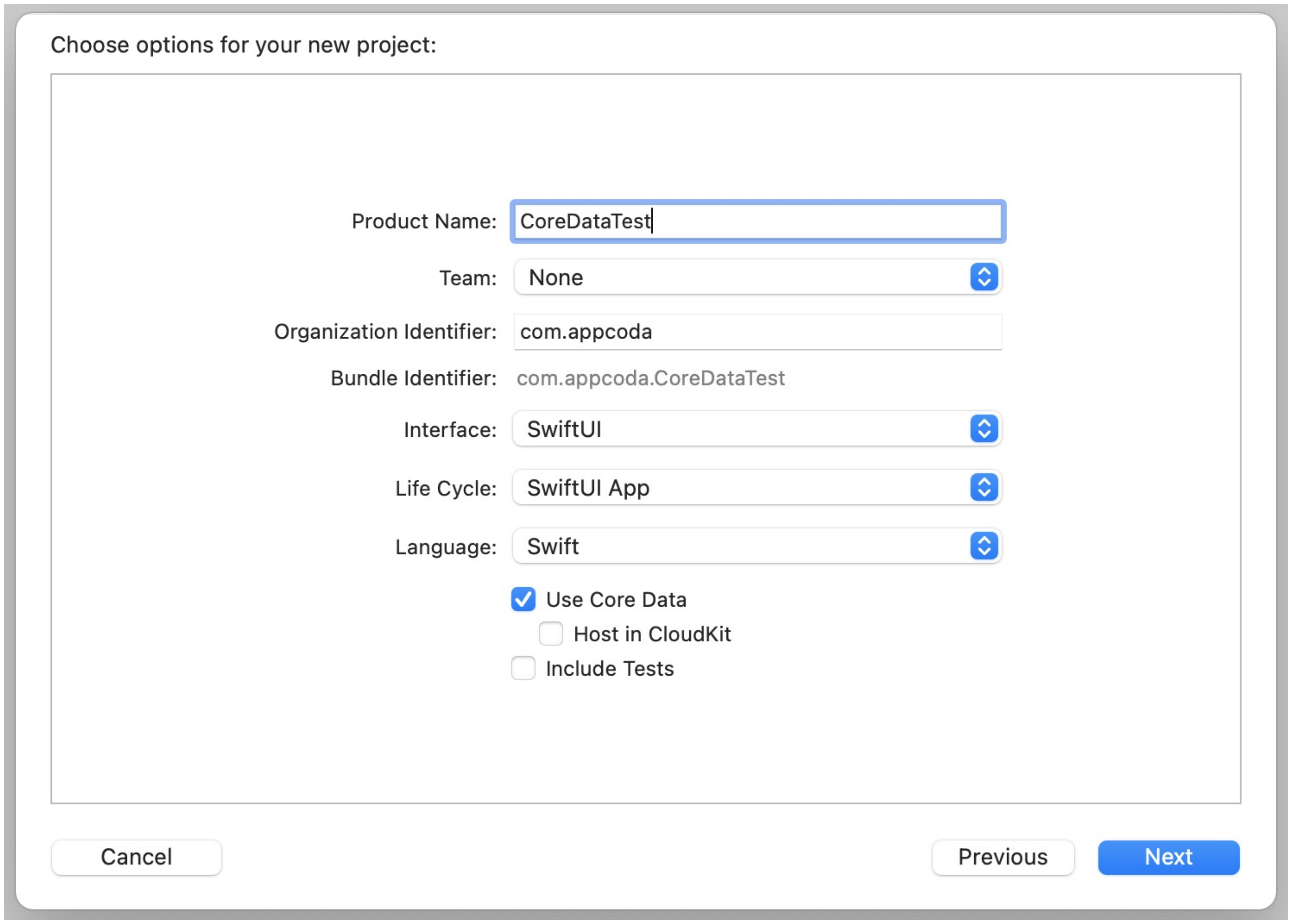
First things first, don't confuse the Core Data framework with a database. Core Data is not a database. It's just a framework for developers to manage and interact with data on a persistent store. Though the SQLite database is the default persistent store for Core Data on iOS, persistent stores are not limited to databases. For instance, you can also utilize Core Data to manage data in a local file (e.g. XML).

The Core Data framework simply shields developers from the inner details of the persistent store. Take the SQLite database as an example. You do not need to know how to connect to the database nor understand SQL to retrieve data records. All you need to figure out is how to work with the Core Data APIs such as `NSManagedObjectContext` and the Managed Object Model.

Feeling confused? No worries. You will understand what I mean after we convert the ToDo app from arrays to Core Data.

Using Core Data in SwiftUI projects

If you start from a brand new project, the easiest way to use the Core Data framework is by enabling the Core Data option. You can give it a try. Launch Xcode and create a new project using the *App* template. Name it to whatever name you like but please ensure you check the *Core Data* checkbox.


Choose options for your new project:

Product Name: CoreDataTest

Team: None

Organization Identifier: com.appcoda

Bundle Identifier: com.appcoda.CoreDataTest

Interface: SwiftUI

Life Cycle: SwiftUI App

Language: Swift

☑ Use Core Data

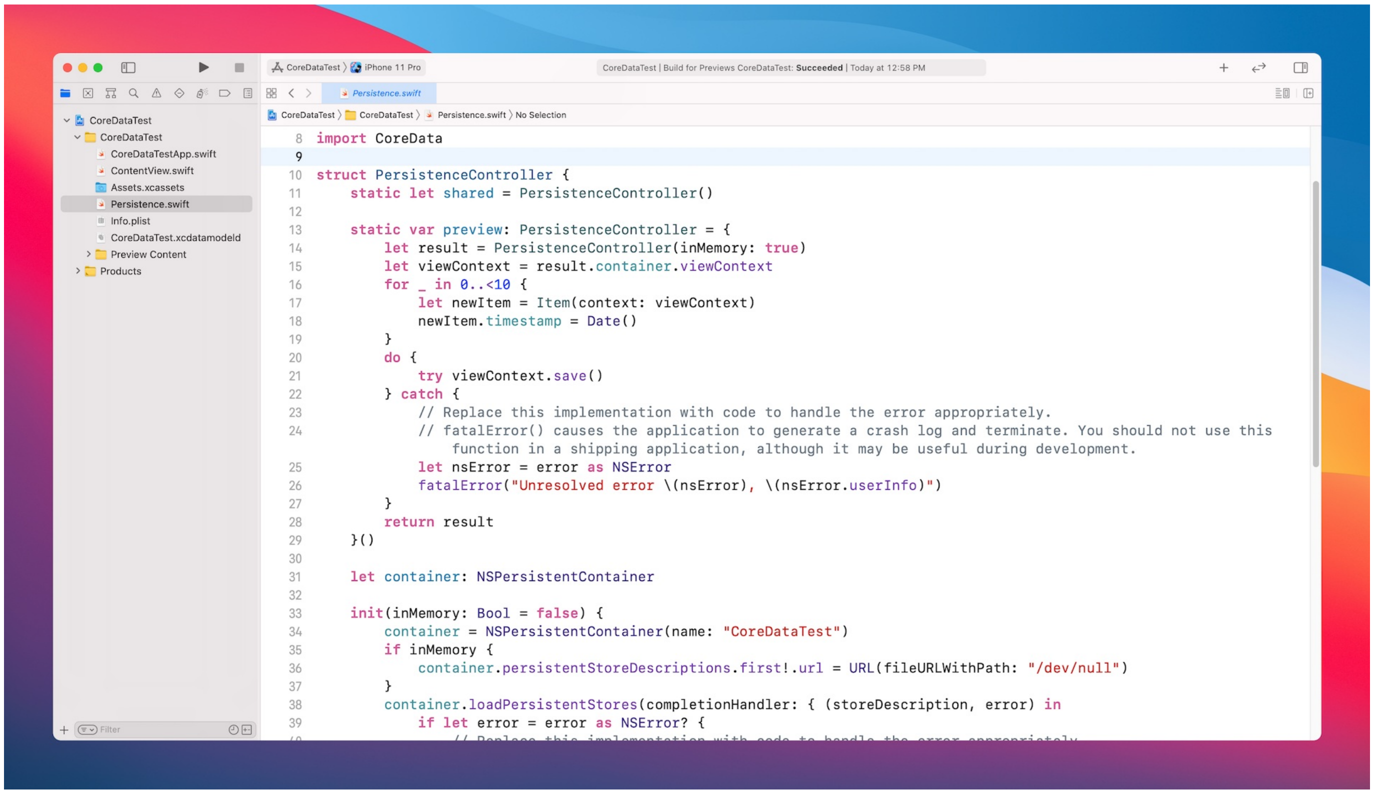
☐ Host in CloudKit

☐ Include Tests

Figure 2. Creating a new project with Core Data enabled

By enabling Core Data, Xcode will generate all the required code and the managed object model for you. Once the project created, you should see a new file named `CoreDataTest.xcdatamodeld`. In Xcode, the managed object model is defined in a file with the extension `.xcdatamodeld`. This is the managed object model generated for your project and this is where you define the entities for interacting with the persistent store.

Take a look at the `Persistence.swift` file, which is another file generated by Xcode. This file contains the code for loading the managed object model and saving the data to the persistent store.

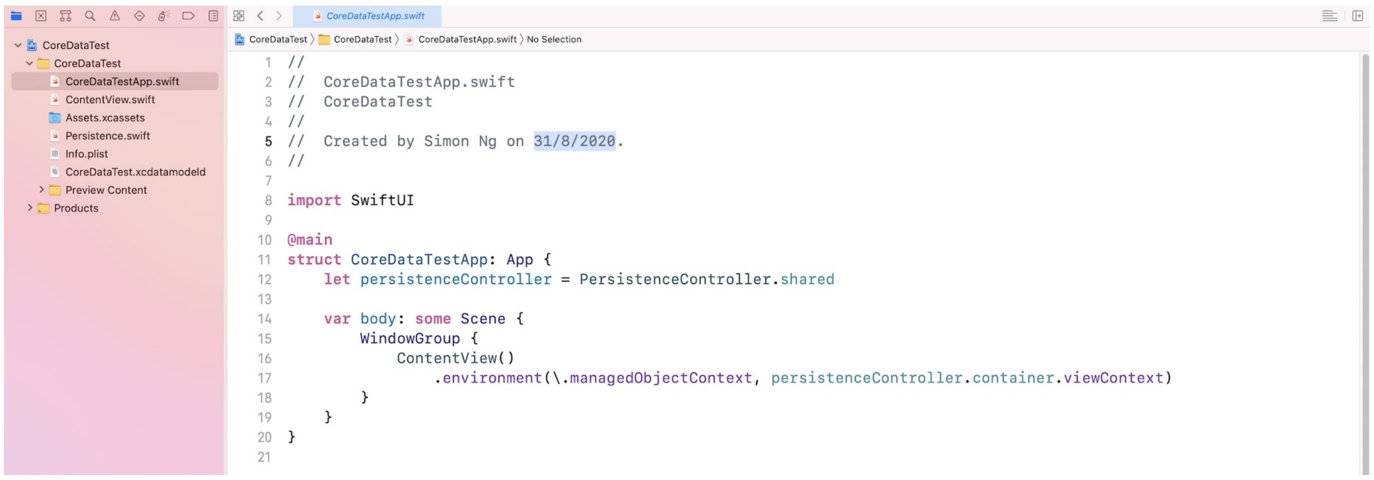

CoreDataTest 〉 iPhone 11 Pro
CoreDataTest | Build for Previews CoreDataTest: Succeeded | Today at 12:58 PM
Persistence.swift
CoreDataTest 〉 CoreDataTest 〉 Persistence.swift 〉 No Selection
CoreDataTest
CoreDataTest
CoreDataTestApp.swift
ContentView.swift
Assets.xcassets
Persistence.swift
Info.plist
CoreDataTest.xcdatamodeld
Preview Content
Products
Filter

```swift
import CoreData

struct PersistenceController {
    static let shared = PersistenceController()

    static var preview: PersistenceController = {
        let result = PersistenceController(inMemory: true)
        let viewContext = result.container.viewContext
        for _ in 0..<10 {
            let newItem = Item(context: viewContext)
            newItem.timestamp = Date()
        }
        do {
            try viewContext.save()
        } catch {
            // Replace this implementation with code to handle the error appropriately.
            // fatalError() causes the application to generate a crash log and terminate. You should not use this
                function in a shipping application, although it may be useful during development.
            let nsError = error as NSError
            fatalError("Unresolved error \(nsError), \(nsError.userInfo)")
        }
        return result
    }()

    let container: NSPersistentContainer

    init(inMemory: Bool = false) {
        container = NSPersistentContainer(name: "CoreDataTest")
        if inMemory {
            container.persistentStoreDescriptions.first!.url = URL(fileURLWithPath: "/dev/null")
        }
        container.loadPersistentStores(completionHandler: { (storeDescription, error) in
            if let error = error as NSError? {
```

Figure 3. The additional code for Core Data

If you've developed apps using UIKit before, you usually use the container to manage the data in the database or other persistent stores. In SwiftUI, it's a little bit different. We seldom use this container directly. Instead SwiftUI injects the managed object context into the environment, so that any view can retrieve the context and manage the data.

Take a look at the `CoreDataTestApp.swift` file. Xcode adds a constant that holds the instance of `PersistenceController` and a line of code to inject the managed object context is injected into the environment.



```swift
//
//  CoreDataTestApp.swift
//  CoreDataTest
//
//  Created by Simon Ng on 31/8/2020.
//

import SwiftUI

@main
struct CoreDataTestApp: App {
    let persistenceController = PersistenceController.shared

    var body: some Scene {
        WindowGroup {
            ContentView()
                .environment(\.managedObjectContext, persistenceController.container.viewContext)
        }
    }
}
```

Figure 4. Injecting the managed object context into the environment

This is all the code and files generated by Xcode when enabling the Core Data option. If you open `ContentView.swift`, Xcode also generates sample code for loading data from the local data store. Look at the code to get an idea of how this works. In general, to save and manage data on the local database, the procedures are:

1. Create an entity in the managed object model (i.e. .xcdatamodeld)
2. Define a managed object, which inherits from `NSManagedObject`, to associate with the entity
3. In the views that need to save and update the data, get the managed object context from the environment using `@Environment` like this:

   ```swift
   @Environment(\.managedObjectContext) var context
   ```

   And then create the managed object and use the `save` method of the context to add the object to the database. Here is a sample code snippet:

```
let task = ToDoItem(context: context)
task.id = UUID()
task.name = name
task.priority = priority
task.isComplete = isComplete
```

4. For data retrieval, Apple introduced a property wrapper called `@FetchRequest` for you to fetch data from the persistent store. Here is sample code:

```
@FetchRequest(
   entity: ToDoItem.entity(),
   sortDescriptors: [ NSSortDescriptor(keyPath: \ToDoItem.priorityNum, ascending:
false) ])
var todoItems: FetchedResults<ToDoItem>
```

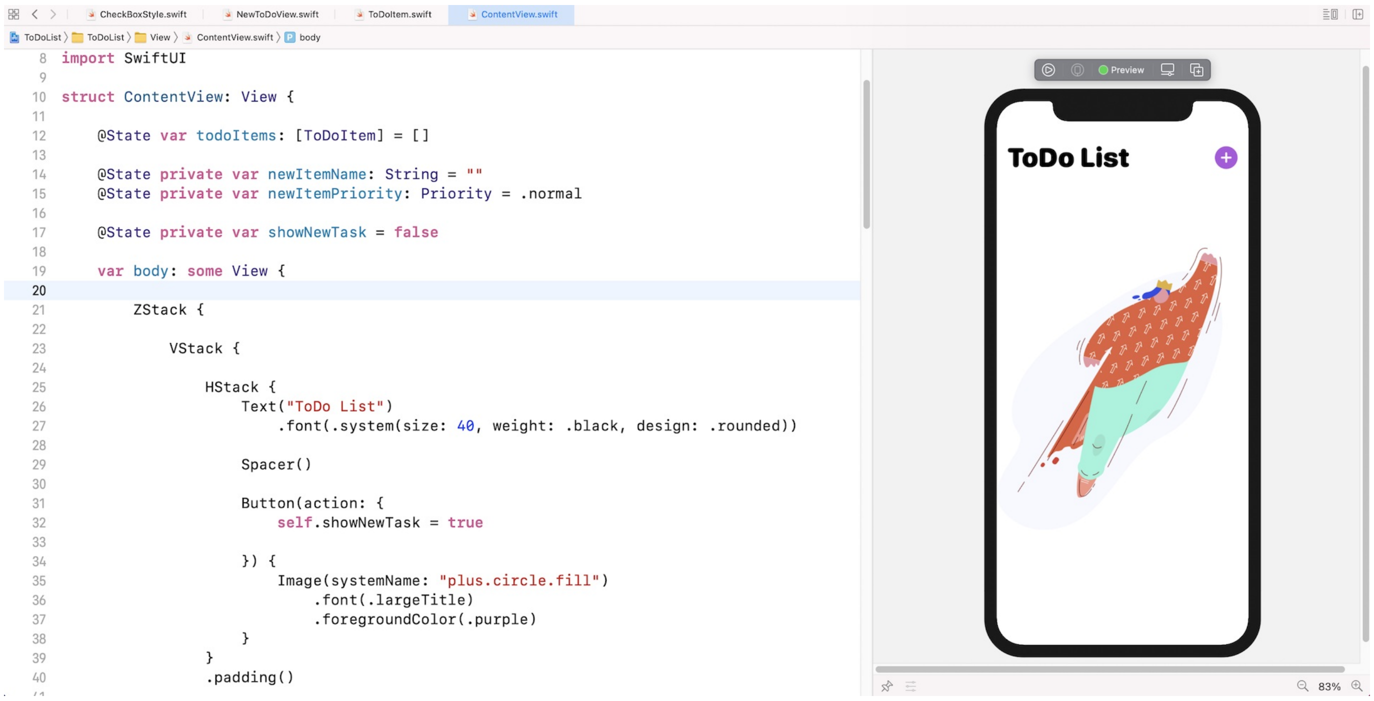
   This property wrapper makes it very easy to perform a fetch request. You just need to specify the entity object you want to retrieve and how the data is ordered. The framework will then use the environment's managed object context to fetch the data. Most importantly, SwiftUI will automatically update any views that are bound to the fetched results because the fetch result is a collection of `NSManagedObject`, which conforms to the `ObservableObject` protocol.

This is how you work with Core Data in SwiftUI projects. I know you may be confused by some of the terms and procedures. This section is just a quick introduction. Later, when you work on the demo app, we will go through these procedures in detail.

Understanding the ToDo App Demo

Now that you have a basic understanding of Core Data, let me go through the app demo with you. Later, we will convert this ToDo demo, allowing it to save the to-do items permanently. Right now, as mentioned before, all the data is stored in memory and will vanish when the app restarts.

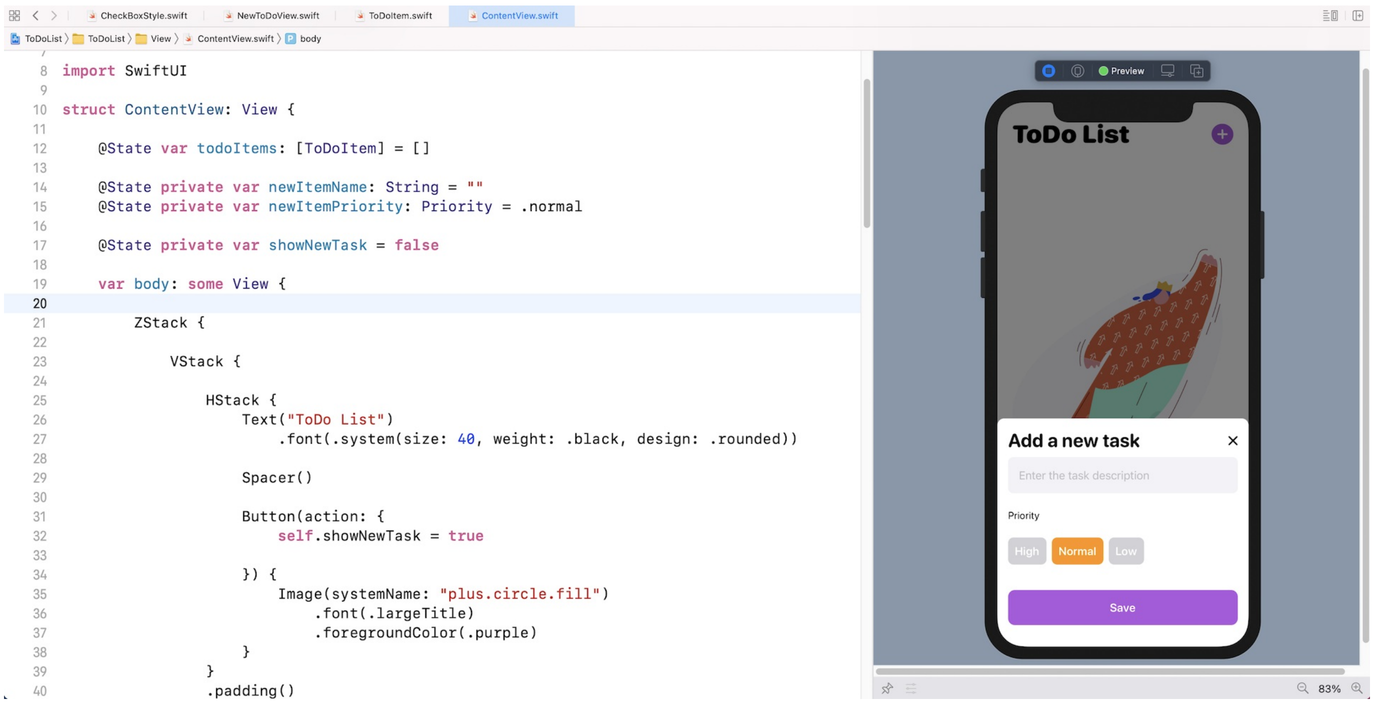
First, please download the starter project from https://www.appcoda.com/resources/swiftui2/SwiftUIToDoListStarter.zip. Unzip the file and open `ToDoList.xcodeproj` in Xcode. Select the `ContentView.swift` file and preview the UI. You should see a screen like that shown in figure 5.



```swift
import SwiftUI

struct ContentView: View {

    @State var todoItems: [ToDoItem] = []

    @State private var newItemName: String = ""
    @State private var newItemPriority: Priority = .normal

    @State private var showNewTask = false

    var body: some View {

        ZStack {

            VStack {

                HStack {
                    Text("ToDo List")
                        .font(.system(size: 40, weight: .black, design: .rounded))

                    Spacer()

                    Button(action: {
                        self.showNewTask = true

                    }) {
                        Image(systemName: "plus.circle.fill")
                            .font(.largeTitle)
                            .foregroundColor(.purple)
                    }
                }
                .padding()
```

Figure 5. Previewing the demo app

Run the app in the preview canvas or a simulator. Tap the + button to add a to-do item. Repeat the procedure to add a few more items. The app then lists the to-do items. Tapping the checkbox of a to-do item will cross out that item.



```swift
import SwiftUI

struct ContentView: View {

    @State var todoItems: [ToDoItem] = []

    @State private var newItemName: String = ""
    @State private var newItemPriority: Priority = .normal

    @State private var showNewTask = false

    var body: some View {

        ZStack {

            VStack {

                HStack {
                    Text("ToDo List")
                        .font(.system(size: 40, weight: .black, design: .rounded))

                    Spacer()

                    Button(action: {
                        self.showNewTask = true

                    }) {
                        Image(systemName: "plus.circle.fill")
                            .font(.largeTitle)
                            .foregroundColor(.purple)
                    }
                }
                .padding()
```

Figure 6. Adding a new task

How to present the list of Todo Items

Now let us walk through the code, so you understand how the code works. First, we start with the model class. Open `ToDoItem.swift` in the *Model* folder.

```swift
enum Priority: Int {
    case low = 0
    case normal = 1
    case high = 2
}

class ToDoItem: ObservableObject, Identifiable {
    var id = UUID()
    @Published var name: String = ""
    @Published var priority: Priority = .normal
    @Published var isComplete: Bool = false

    init(name: String, priority: Priority = .normal, isComplete: Bool = false) {
        self.name = name
        self.priority = priority
        self.isComplete = isComplete
    }
}
```

The ToDo app demo is a simplified version of an ordinary ToDo app. Each to-do item (or task), has three properties: *name*, *priority*, and *isComplete* (i.e. the status of the task). This class adopts the `ObservableObject` protocol. The three properties are marked with @Published so that the subscribers are informed whenever there are any changes of the values. Later, in the implementation of `ContentView`, SwiftUI listens for value changes and updates the views accordingly. For example, when the value of `isComplete` changes, it toggles the checkbox.

This class also conforms to the `Identifiable` protocol such that each instance of `ToDoItem` has an unique identifier. Later, we will use the `ForEach` and `List` to display the to-do items. This is why we need to adopt the protocol and create the `id` property.

Now let's move onto the views and begin with the `ContentView.swift` file. Assuming you've read chapter 10, you should understand most of the code. The content view has three main parts, which are embedded in a `ZStack`:

1. The list view that presents all the to-do items.
2. The empty view (NoDataView) that is displayed when there are no to-do items .
3. The "Add a new task" view that is shown when a user taps the + button.

Take a look at the first `VStack`:

```
VStack {

    HStack {
        Text("ToDo List")
            .font(.system(size: 40, weight: .black, design: .rounded))

        Spacer()

        Button(action: {
            self.showNewTask = true

        }) {
            Image(systemName: "plus.circle.fill")
                .font(.largeTitle)
                .foregroundColor(.purple)
        }
    }
    .padding()

    List {

        ForEach(todoItems) { todoItem in
            ToDoListRow(todoItem: todoItem)
        }

    }
}
.rotation3DEffect(Angle(degrees: showNewTask ? 5 : 0), axis: (x: 1, y: 0, z: 0))
.offset(y: showNewTask ? -50 : 0)
.animation(.easeOut)
```

I declared a state variable named `todoItems` to hold all the to-do items. It's marked with `@State` so that the list will be refreshed whenever there are any changes. In the `List` view, we use `ForEach` to loop through the items in the array.

We handle the rows of the list, by a separate view named `ToDoListRow`:

```
struct ToDoListRow: View {

    @ObservedObject var todoItem: ToDoItem

    var body: some View {
        Toggle(isOn: self.$todoItem.isComplete) {
            HStack {
                Text(self.todoItem.name)
                    .strikethrough(self.todoItem.isComplete, color: .black)
                    .bold()
                    .animation(.default)

                Spacer()

                Circle()
                    .frame(width: 10, height: 10)
                    .foregroundColor(self.color(for: self.todoItem.priority))
            }
        }.toggleStyle(CheckboxStyle())
    }

    private func color(for priority: Priority) -> Color {
        switch priority {
        case .high: return .red
        case .normal: return .orange
        case .low: return .green
        }
    }
}
```

This view takes in a to-do item, which is a `ObservableObject`. This means for any changes of that to-do item, the view that subscribes to the item will be invalidated automatically.

For each row of the to-do item, consists of three parts:

1. A toggle / checkbox - indicates whether the task is complete or not.
2. A text label - shows the name of the task
3. A dot / circle - shows the priority of the task

The second and third parts are pretty straightforward. For the checkbox, it's worth having a deeper discussion. SwiftUI comes with a standard control called `Toggle`. In an earlier chapter, we used it to create a Settings screen. The presentation of the toggle is more like a switch that lets you flip between on and off. In the ToDo app, we want to make the toggle look like a checkbox.

Customizing the look & feel of a Toggle

Similar to `Button` which we discussed in chapter 6, `Toggle` also lets developers customize its style. All you need to do is to implement the `ToggleStyle` protocol and provide the customizations. In the project navigator, open the `CheckBoxStyle.swift` file to take a look:

```
struct CheckboxStyle: ToggleStyle {

    func makeBody(configuration: Self.Configuration) -> some View {

        return HStack {

            Image(systemName: configuration.isOn ? "checkmark.circle.fill" : "circ
le")
                .resizable()
                .frame(width: 24, height: 24)
                .foregroundColor(configuration.isOn ? .purple : .gray)
                .font(.system(size: 20, weight: .bold, design: .default))
                .onTapGesture {
                    configuration.isOn.toggle()
                }

            configuration.label

        }

    }
}
```

In the code, we implement the `makeBody` method, which is the requirement of the protocol. We create an image view which displays a checkmark image or a circle image, depending on the status of the toggle (i.e. `configuration.isOn`). This is how you customize the style of a toggle.

To use the `CheckboxStyle`, attach the `toggleStyle` modifier to the `Toggle` and specify the checkbox style like this:

```
.toggleStyle(CheckboxStyle())
```

Handling the empty list view

When there are no items in the array, we present an image view instead of showing an empty list view. This is completely optional. However, I think it makes the app look better and let users know what to do when the app is first started.

```
// If there is no data, show an empty view
if todoItems.count == 0 {
    NoDataView()
}
```

Since we have a `ZStack` to embed the views, it's pretty easy to control the appearance of this empty view, which is only displayed when the array is empty.

Displaying the Add Task view

When a user taps the + button at the top-right corner, the app displays the `NewToDoView`, which I will go through with you shortly. This view overlays on top of the list view and appears like a bottom sheet. We also add a blank view for darkening the list view.

Here is the code for reference:

```
if showNewTask {
    BlankView(bgColor: .black)
        .opacity(0.5)
        .onTapGesture {
            self.showNewTask = false
        }

    NewToDoView(isShow: $showNewTask, todoItems: $todoItems, name: "", priority: .
normal)
        .transition(.move(edge: .bottom))
        .animation(.interpolatingSpring(stiffness: 200.0, damping: 25.0, initialVe
locity: 10.0))
}
```

Understanding the Add Task view

Now let me walk you through the code in `NewToDoView.swift`, which is for users to add a new task or to-do item. You can refer to figure 6 or simply open the file to preview it see what this view looks like.

The `NewToDoView` takes in two bindings: *isShow* and *todoItems*. The `isShow` parameter controls whether this *Add New Task* view should appear on screen. The `todoItems` variable holds a reference to the array of to-do items. We need the caller to pass us the binding to `todoItems` so that we can update the array with the new task.

```
@Binding var isShow: Bool
@Binding var todoItems: [ToDoItem]

@State var name: String
@State var priority: Priority
@State var isEditing = false
```

In the view, we let users input the name of the task and set its priority (low/normal/high). The state variable `isEditing` indicates whether the user is editing the task name. To avoid the software keyboard from obscuring the editing view, the app will shift the view upward while the user is editing the text field.

```
TextField("Enter the task description", text: $name, onEditingChanged: { (editingC
hanged) in

    self.isEditing = editingChanged

})


...

.offset(y: isEditing ? -320 : 0)
```

After the *Save* button is tapped, we verify if the text field is empty. If not, we create a new `ToDoItem` and call the `addTask` function to append it to the `todoItems` array, otherwise we do nothing.

```
// Save button for adding the todo item
Button(action: {

    if self.name.trimmingCharacters(in: .whitespaces) == "" {
        return
    }

    self.isShow = false
    self.addTask(name: self.name, priority: self.priority)

}) {
    Text("Save")
        .font(.system(.headline, design: .rounded))
        .frame(minWidth: 0, maxWidth: .infinity)
        .padding()
        .foregroundColor(.white)
        .background(Color.purple)
        .cornerRadius(10)
}
.padding(.bottom)
```

Since the `todoItems` array is a state variable, the list view will be automatically refreshed and display the new task. This is how the code works. If you don't understand how the *Add task* view is displayed at the bottom of the screen, please refer to chapter 18 on building an Expandable Bottom Sheet.

Working with Core Data

Now that I've walked you through the starter project, it's time to convert the app to use Core Data for storing the to-do items in the database. In the very beginning, we created a blank project with *Core Data* enabled. By checking the Core Data checkbox, Xcode automatically generated the basic skeleton of a Core Data project. This time, I will show you how to transform the project to use Core Data manually.

Creating the Persistent Controller

Let's first create a new file called the `Persistence.swift` file. In the project navigator, right click `Model` and use the *Swift file* template. Name the file `Persistence.swift` and insert the following code in the file:

```
import CoreData

struct PersistenceController {
    static let shared = PersistenceController()

    let container: NSPersistentContainer

    init(inMemory: Bool = false) {
        container = NSPersistentContainer(name: "ToDoList")
        if inMemory {
            container.persistentStoreDescriptions.first!.url = URL(fileURLWithPath
: "/dev/null")
        }
        container.loadPersistentStores(completionHandler: { (storeDescription, err
or) in
            if let error = error as NSError? {
                // Replace this implementation with code to handle the error appro
priately.
                // fatalError() causes the application to generate a crash log and
 terminate. You should not use this function in a shipping application, although i
t may be useful during development.

                /*
                Typical reasons for an error here include:
                * The parent directory does not exist, cannot be created, or disal
lows writing.
                * The persistent store is not accessible, due to permissions or da
ta protection when the device is locked.
                * The device is out of space.
                * The store could not be migrated to the current model version.
                Check the error message to determine what the actual problem was.
                */
                fatalError("Unresolved error \(error), \(error.userInfo)")
            }
        })
    }
}
```

You should be familar with the code because it is the same as the code generated by Xcode, except that the name of the container is changed to *ToDoList*.

Injecting the managed object context

Now open `ToDoListApp.swift` and inject the managed object context into the environment. In the `ToDoListApp` struct, declare the following variable to hold the `PersistenceController`:

```
let persistenceController = PersistenceController.shared
```

Next, in the same file, attach the `environment` modifier to `ContentView()` like this:

```
ContentView()
    .environment(\.managedObjectContext, persistenceController.container.viewConte
xt)
```

In the code above, we inject the managed object context into the environment of `ContentView`. This allows us to easily access the context in the content view for managing the data in the database.

Creating the managed object model

Next, we need to manually create the managed object model. In the project navigator, right click the *ToDoList* folder and select *New file...*. Choose *Data Model* and name the file `ToDoList.xcdatamodeld`. Please make sure you name the file correctly because it should match the name for initializing the `NSPersistentContainer`.

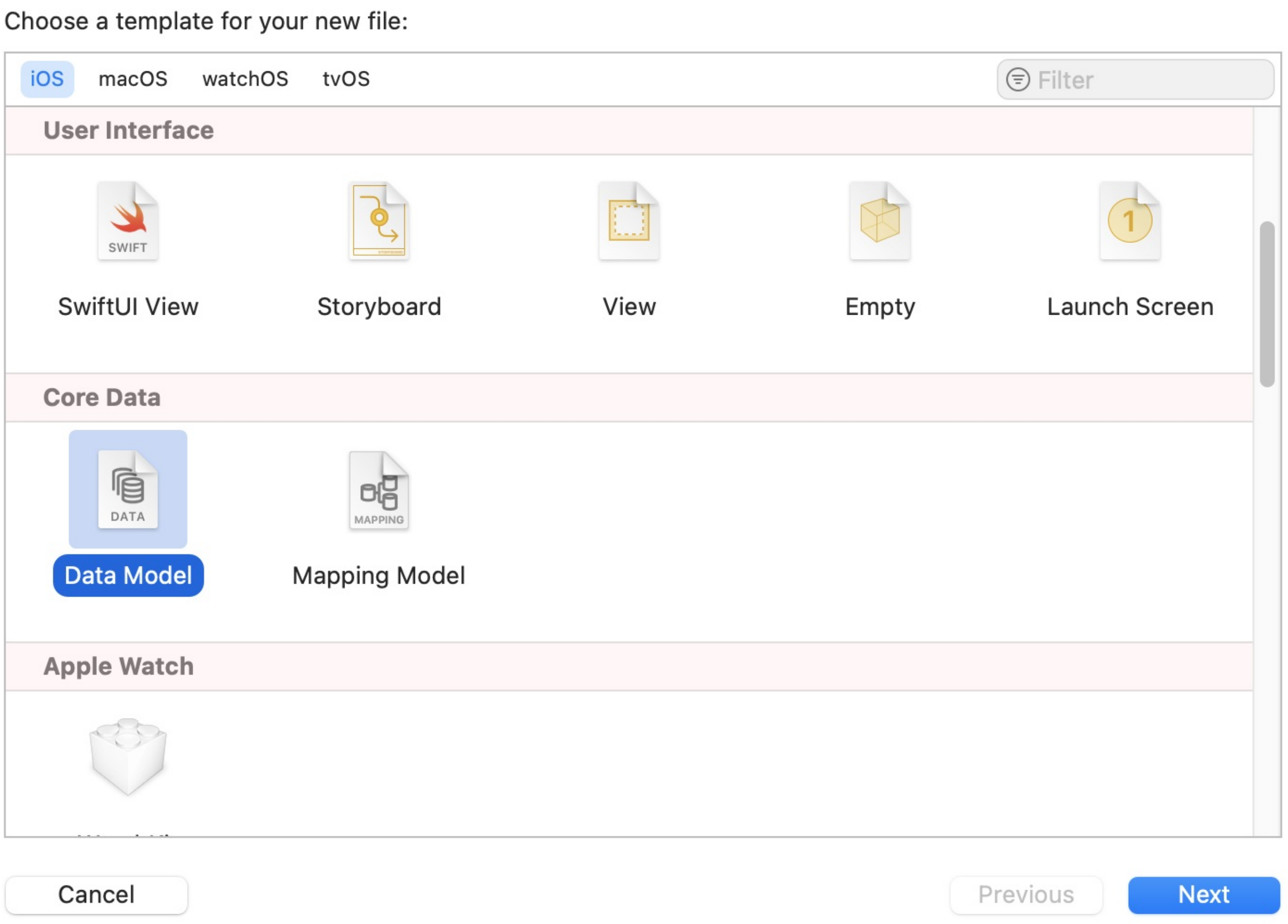


Figure 7. Choosing the Data Model template

Once created, select the model file and click the *Add Entity* button to create a new entity. Change the name of the entity from *Entity* to *ToDoItem*. You can think of this entity as a record in the database table. Therefore, this entity should store the properties of a `ToDoItem`. We need to add 4 attributes for the entity including (see figure 8):

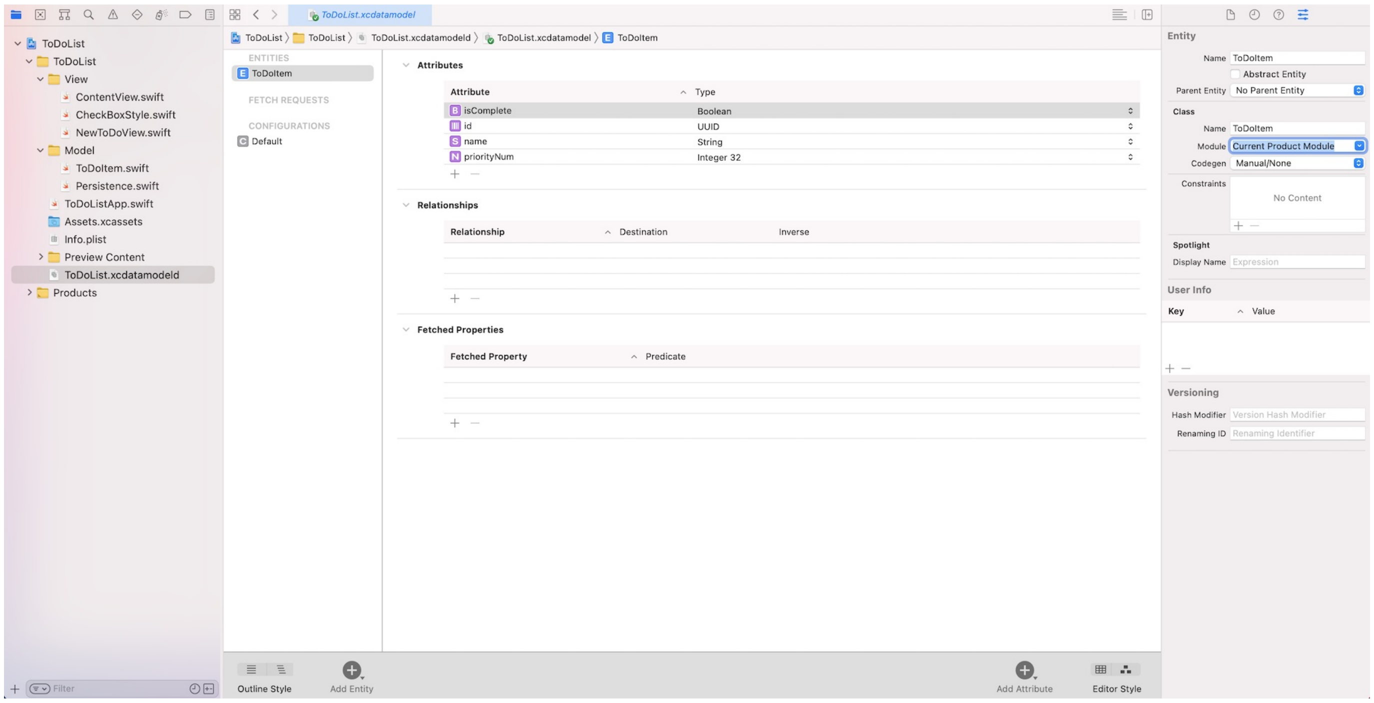
- *id* with the type *UUID*
- *name* with the type *String*
- *priorityNum* with the type *Integer 32*
- *isComplete* with the type *Boolean*

The types of `id`, `name`, and `isComplete` are exactly the same as that of the `ToDoItem` class. But why does the priority is set to the type *Integer 32*? If you take a look at the code in `ToDoItem.swift`, you see that the *priority* property is an Enum:

```
enum Priority: Int {
    case low = 0
    case normal = 1
    case high = 2
}
```

To save this enum into the database, we have to store its raw value which is an integer. This is why we use the type *Integer 32* and name the attribute *priorityNum* to avoid naming conflicts.

By default, Xcode automatically generates the model class of this `ToDoItem` entity. However, I prefer to create this class manually in order to have better control. So, select the `ToDoItem` entity and open the *Data Model Inspector*. If you can't see the inspector, go up to the menu and select *View* > *Inspectors* > *Show Data Model Inspector*. In the *Class* section, set the *Module* to *Current Product Module* and *Codegen* to *Manual/None*. This disables the code generation.



Attributes

Attribute	Type
isComplete	Boolean
id	UUID
name	String
priorityNum	Integer 32

Relationships

Relationship	Destination	Inverse

Fetched Properties

Fetched Property	Predicate

Entity
Name: ToDoItem
Abstract Entity
Parent Entity: No Parent Entity
Class
Name: ToDoItem
Module: Current Product Module
Codegen: Manual/None
Constraints: No Content
Spotlight
Display Name: Expression
User Info
Key | Value
Versioning
Hash Modifier: Version Hash Modifier
Renaming ID: Renaming Identifier

Figure 8. Disable code generation

As you can see, everything we've developed so far does not require you have the knowledge of database programming. No SQL, no database tables. All the things you deal with are object based. This is the beauty of Core Data.

Defining the model class

In Core Data, every entity should be paired with a model class. By default, this model class is generated by Xcode. Previously, we changed the setting from *code gen* to *manual*. So, we need to implement the model class `ToDoItem` manually. Switch over to `ToDoItem.swift` and import the *CoreData* package:

```
import CoreData
```

Replace the `ToDoItem` class like this:

```
public class ToDoItem: NSManagedObject {
    @NSManaged public var id: UUID
    @NSManaged public var name: String
    @NSManaged public var priorityNum: Int32
    @NSManaged public var isComplete: Bool
}

extension ToDoItem: Identifiable {

    var priority: Priority {
        get {
            return Priority(rawValue: Int(priorityNum)) ?? .normal
        }

        set {
            self.priorityNum = Int32(newValue.rawValue)
        }
    }
}
```

The model class of Core Data should be inherited from `NSManagedObject`. Each property is annotated with `@NSManaged` and corresponds to the attribute of the Core Data model we created earlier. By using `@NSManaged`, this tells the compiler that the property is taken care by Core Data.

In the original version of `ToDoItem`, we have the `priority` property which has a type of Enum. For the Core Data version, we have to create a computed property for `priority`. This computed property transforms the priority number into an Enum and vice versa.

Using @FetchRequest to fetch records

Now that we've prepared the model class, let's see how easy it is to fetch records from database. Switch over to `ContentView.swift`. Originally, we have an array variable holding all to-do items, which is also marked with `@State`:

```
@State var todoItems: [ToDoItem] = []
```

Since we are moving to store the items in database, we need to modify this line of code and fetch the data from it. Apple introduced a new property wrapper called `@FetchRequest`. This makes it very easy to load data from the database.

Replace the line of code above with `@FetchRequest` like this:

```
@FetchRequest(
    entity: ToDoItem.entity(),
    sortDescriptors: [ NSSortDescriptor(keyPath: \ToDoItem.priorityNum, ascending:
false) ])
var todoItems: FetchedResults<ToDoItem>
```

Recall that we've injected the managed object context in the environment, this fetch request automatically utilizes the context and fetches the required data for you. In the code above, we specify to fetch the `ToDoItem` entity and how the results should be ordered. Here, we would like to sort the items based on priority.

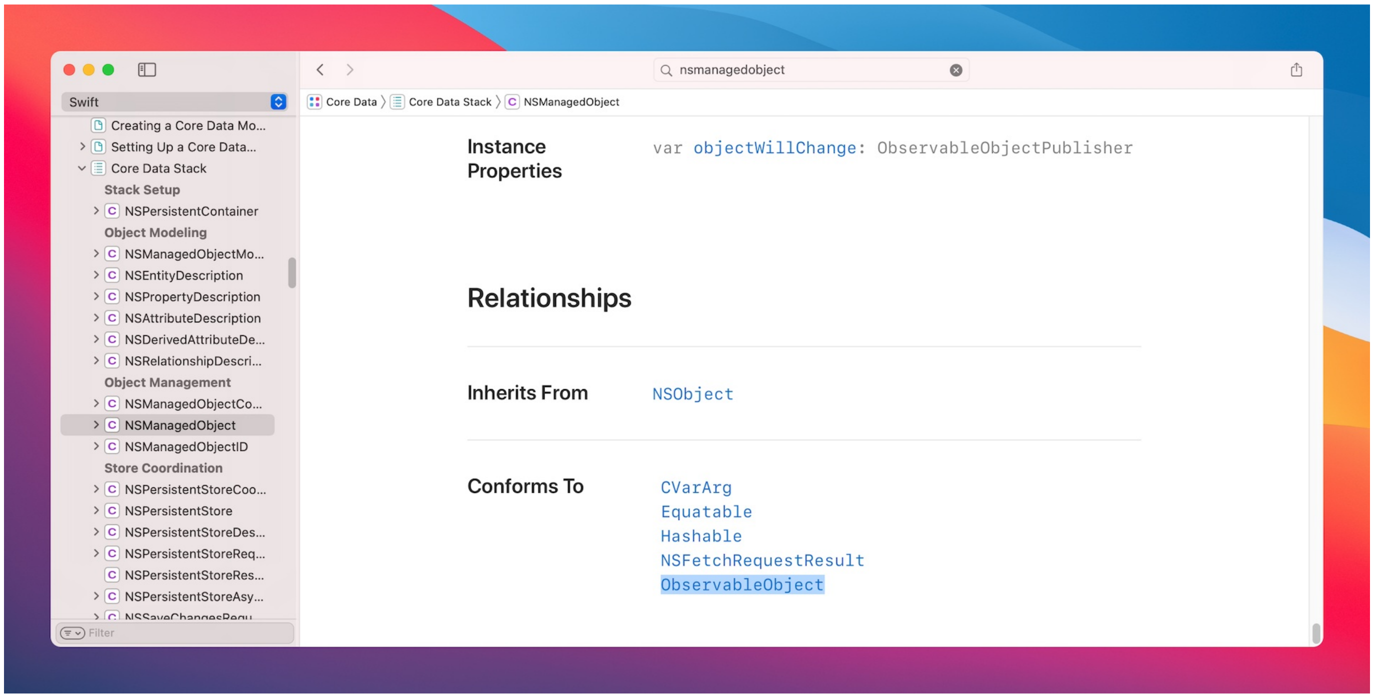
Once the fetch completes, you will have a collection of `ToDoItem` managed objects, these are based on the `ToDoItem` class we defined earlier in the model layer.

This is how you perform a fetch request and retrieve data from database. And, since the properties of `ToDoItem` are kept intact, we DO NOT need to make any code change for the list view. We can use the fetch result directly in `ForEach`:

```
List {

    ForEach(todoItems) { todoItem in
        ToDoListRow(todoItem: todoItem)
    }

}
```

On top of that, you can directly pass the `todoItem`, which is a `NSManageObject` to create a `ToDoListRow`. Do you know why we do not need to make any changes?

Take a look at the documation of `NSManagedObject`. It conforms to `ObservableObject`. This is why we can directly pass a `todoItem` to `ToDoListRow`.



Instance Properties `var objectWillChange: ObservableObjectPublisher`

Relationships

Inherits From `NSObject`

Conforms To `CVarArg` `Equatable` `Hashable` `NSFetchRequestResult` `ObservableObject`

Figure 9. NSManagedObject documentation

One more thing. You may also wonder if we need to manually perform a fetch request when there are changes to `todoItems` (say, we add a new item). This is another advantage of using `@FetchRequest`. SwiftUI automatically manages the changes and refreshes the UI accordingly.

Adding data to the persistent store

Now, let's continue to do the Core Data migration and update the code for `NewToDoView.swift`. To save a new task in the database, you need to first obtain the managed object context from the environment:

```
@Environment(\.managedObjectContext) var context
```

Since we no longer use an array to hold the to-do items, you can remove this line of code:

```
@Binding var todoItems: [ToDoItem]
```

Next, let's update the `addTask` function like this:

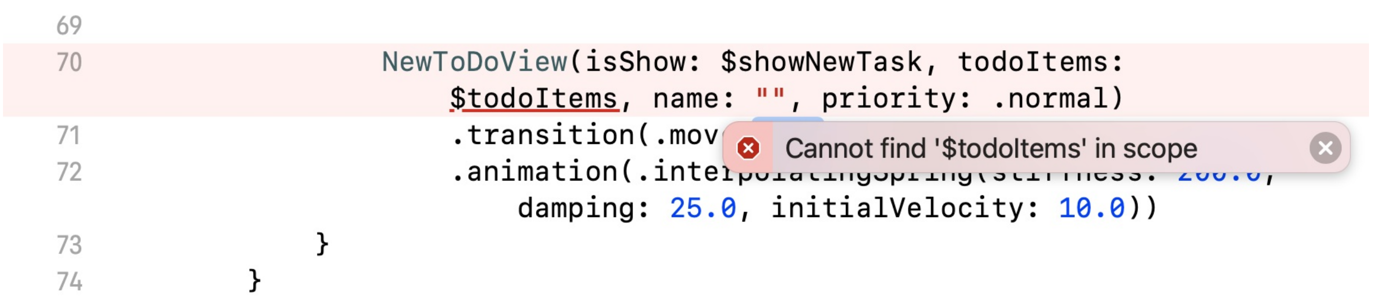
```
private func addTask(name: String, priority: Priority, isComplete: Bool = false) {

    let task = ToDoItem(context: context)
    task.id = UUID()
    task.name = name
    task.priority = priority
    task.isComplete = isComplete

    do {
        try context.save()
    } catch {
        print(error)
    }
}
```

To insert a new record into the database, you create a `ToDoItem` with the managed context and then call the `save()` function of the context to commit the changes.

Since we removed the `todoItems` binding, we need to update the preview code:

```
struct NewToDoView_Previews: PreviewProvider {
    static var previews: some View {
        NewToDoView(isShow: .constant(true), name: "", priority: .normal)
    }
}
```

Now let's move back to `ContentView.swift`. Similarly, you should see an error in the `ContentView` (see figure 10).



```
                    NewToDoView(isShow: $showNewTask, todoItems:
                        $todoItems, name: "", priority: .normal)
                        .transition(.mov
                        .animation(.interpolatingSpring(stiffness: 200.0,
                            damping: 25.0, initialVelocity: 10.0))
                }
            }
```

Cannot find '$todoItems' in scope

Figure 10. Xcode shows you an error in ContentView

Change the line of code like this to fix the error:

```
NewToDoView(isShow: $showNewTask, name: "", priority: .normal)
```

We simply remove the `todoItems` parameter. This is how we convert the demo app from using an in-memory array as storage to a persistent store.

Updating an existing item

When you mark an item as complete, the app should store the change in the database. In `ContentView.swift`, locate the `ToDoListRow` struct and declare the following variable:

```
@Environment(\.managedObjectContext) var context
```

Similar to adding a new record, we also need to obtain the managed object context for record update. For the `Toggle` view, attach the `onReceive` modifier and place it right after `.toggleStyle(CheckboxStyle())` like this:

```
var body: some View {
    Toggle(isOn: self.$todoItem.isComplete) {
        .
        .
        .
    }
    .toggleStyle(CheckboxStyle())
    // Add the following code
    .onReceive(todoItem.objectWillChange, perform: { _ in
        if self.context.hasChanges {
            try? self.context.save()
        }
    })
}
```

Whenever there is a change to the toggle, the `isComplete` property of a `todoItem` will be updated. But, how we can save it to the persistent store? Recall that the `todoItem` conforms to `ObservableObject`, this implies that it has a publisher that transmits changes in values.

Here, the `onReceive` modifier listens for these changes (say, the change of `isComplete`) and saves them to the persistent store by calling the `save()` function of the context.

Now you can run the app in a simulator to try it out. You should be able to add new tasks to the app. Once the new tasks are added, they should appear in the list view immediately. The checkbox should work too. Most importantly, all the changes are now saved permanently in the device's database. After you restart the app, all the items are still there.

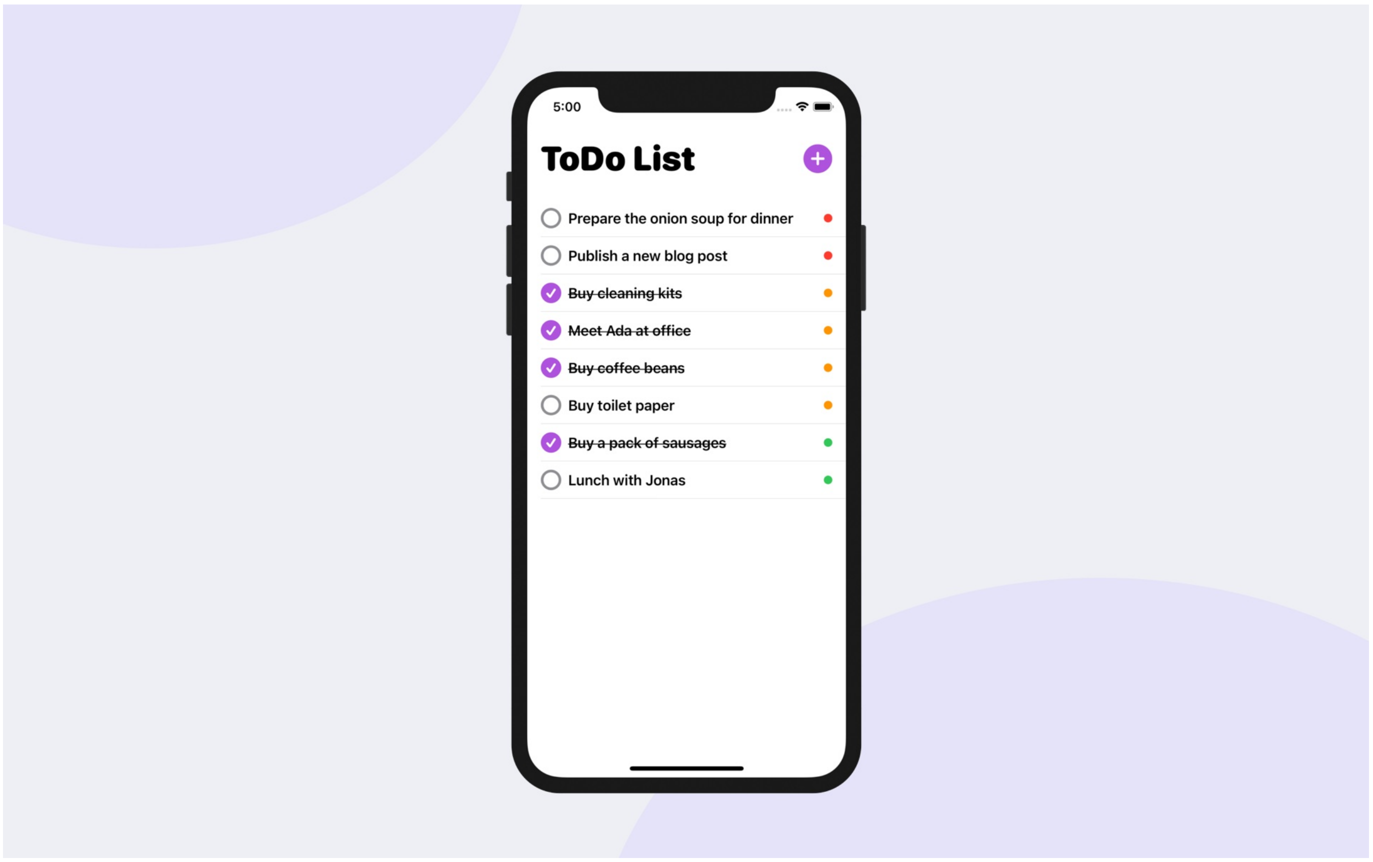


Figure 11. Your ToDo app now supports Core Data

Deleting an item from database

Now that I have shown you how to perform fetch, update, and insert, how about data deletion? We will add a feature to the app for removing a to-do item.

In the `ContentView` struct, declare a `context` variable:

```
@Environment(\.managedObjectContext) var context
```

Then add a new function called `deleteTask` like this:

```
private func deleteTask(indexSet: IndexSet) {
    for index in indexSet {
        let itemToDelete = todoItems[index]
        context.delete(itemToDelete)
    }

    DispatchQueue.main.async {
        do {
            try context.save()

        } catch {
            print(error)
        }
    }
}
```

This function takes in an index set which stores the index of the items for deletion. To delete an item from the persistent store, you can call the `delete` function of the context and specify the item to delete. Lastly, call `save()` to commit the change.

Now that we have prepared the delete function, where should we invoke it? Attach the `onDelete` modifier to `ForEach` of the list view like this:

```
List {

    ForEach(todoItems) { todoItem in
        ToDoListRow(todoItem: todoItem)
    }
    .onDelete(perform: deleteTask)

}
```

The `onDelete` modifier automatically enables the swipe-to-delete feature in the list view. When the user deletes an item, we call the `deleteTask` function to remove the item from the database.

Run the app and swipe to delete an item. This will completely remove it from the database.

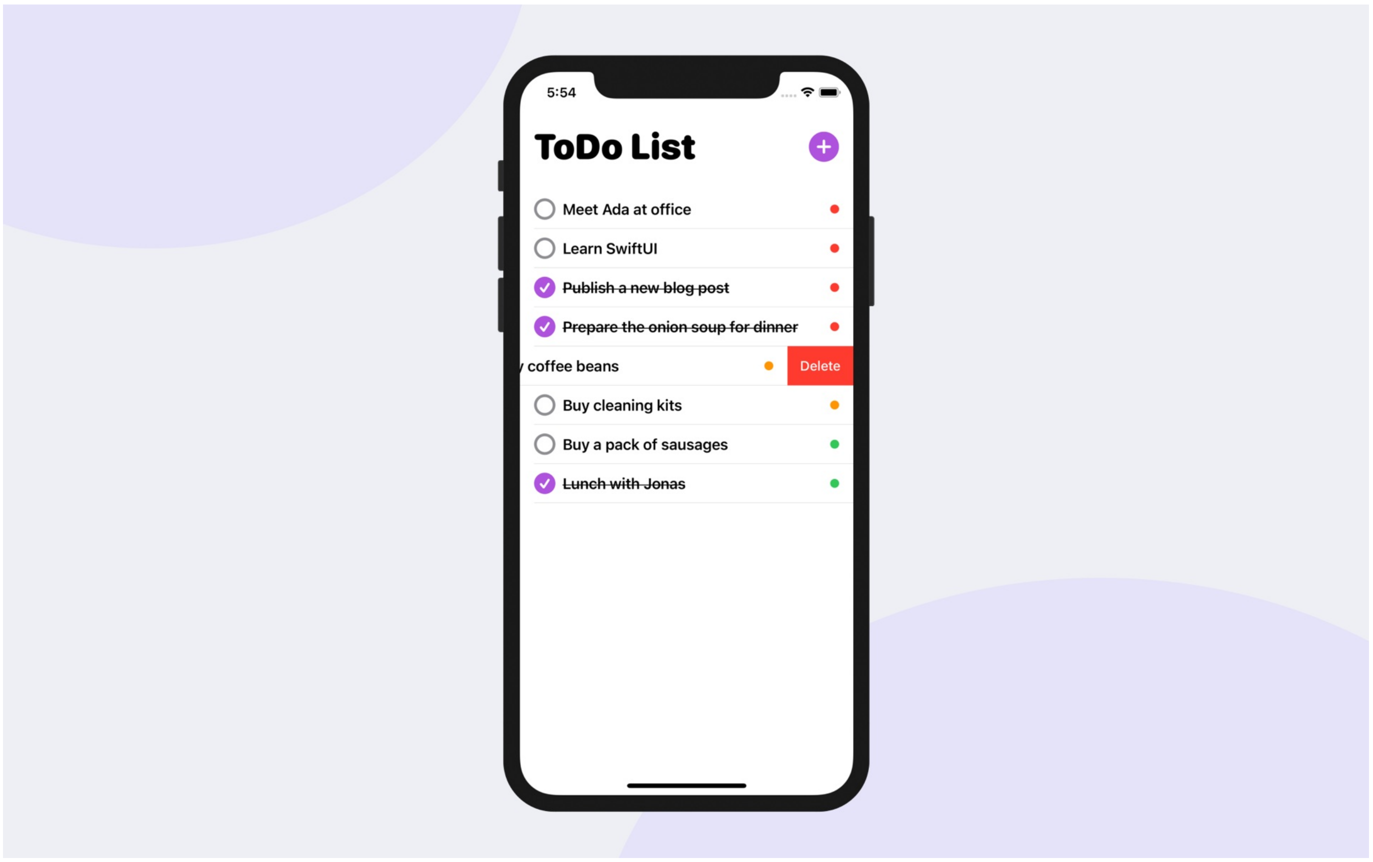


Figure 12. Deleting an item

Working with SwiftUI Preview

You should aware that the preview of your app doesn't work since we changed the app to use Core Data. This is understandable because we haven't injected the managed object context in the `ContentView_Previews` struct. So, how do we fix the issue and make the preview work.

First, we need to create an in-memory data store and populate it with some test data. Open `Persistence.swift` and declare a static variable like this:

```
static var preview: PersistenceController = {
    let result = PersistenceController(inMemory: true)
    let viewContext = result.container.viewContext
    for index in 0..<10 {
        let newItem = ToDoItem(context: viewContext)
        newItem.id = UUID()
        newItem.name = "To do item #\(index)"
        newItem.priority = .normal
        newItem.isComplete = false
    }
    do {
        try viewContext.save()
    } catch {
        // Replace this implementation with code to handle the error appropriately.

        // fatalError() causes the application to generate a crash log and termina
te. You should not use this function in a shipping application, although it may be
 useful during development.
        let nsError = error as NSError
        fatalError("Unresolved error \(nsError), \(nsError.userInfo)")
    }
    return result
}()
```

In the code above, we create an instance of `PersistenceController` with the `inMemory` parameter set to `true`. Then we add 10 sample to-do items and save them to the data store.

Now let's switch over to the `ContentView.swift` and update the preview code like this:

```
struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView().environment(\.managedObjectContext, PersistenceController.pr
eview.container.viewContext)
    }
}
```

We inject the context of the in-memory container to the environment of the content view. By doing so, the content view can now load the sample to-do items and display them in the preview canvas.

Summary

In this chapter, we converted a Todo list app from storing data in memory to a persistent store. I hope you now understand how to integrate Core Data in a SwiftUI project and know how to perform all basic CRUD (create, read, update & delete) operations. The introduction of the `@FetchRequest` property wrapper and the injection of the managed object context have made it very easy to manage data in a persistent store.

For reference, you can download the complete ToDoList project here:

- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUIToDoList.zip)

Chapter 23
Integrating UIKit with SwiftUI Using UIViewRepresentable

There are two common questions developers ask about SwiftUI. First is the question of how to implement Core Data and SwiftUI. The other common question is how to work with UIKit views in SwiftUI projects. In this chapter, you will learn this technique by integrating a UISearchBar in the Todo app.

If you are new to UIKit, UISearchBar is a built-in component of the framework that allows developers to present a search bar for data search. Figure 1 shows you the standard search bar in iOS. SwiftUI, however, doesn't come with this standard UI component. To implement a search bar in a SwiftUI project (say, our ToDo app), one approach is to make use of the `UISearchBar` component in UIKit.

So, how do we interface with UIKit views or controllers in SwiftUI?

For the purpose of backward compatibility, Apple introduced a couple of new protocols, namely `UIViewRepresentable` and `UIViewControllerRepresentable` in the iOS SDK. With these protocols, you can wrap a UIKit view (or view controller) and make it available to your SwiftUI project.

To see how it works, we will enhance our Todo app with a search function. We will add a search bar right below the app title and let users filter the to-do items by entering a search term.

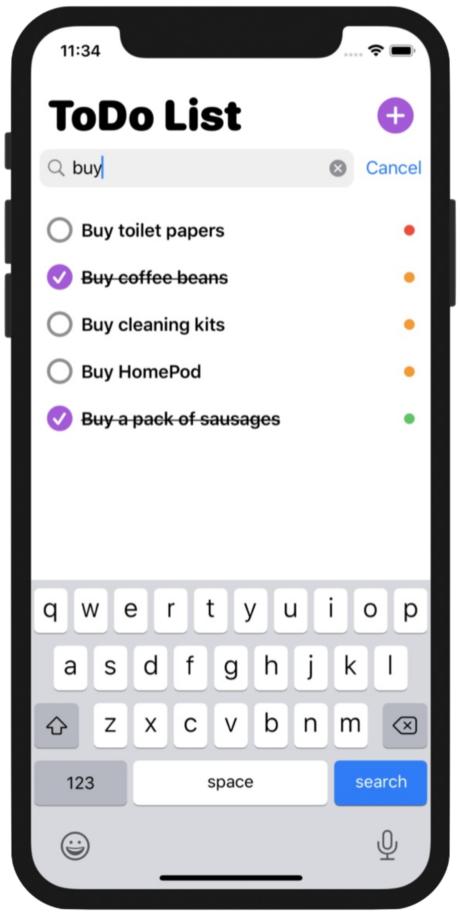


Figure 1. Adding a search bar in the ToDo app

To get started, download the ToDo project at https://www.appcoda.com/resources/swiftui2/SwiftUIToDoList.zip. We will build on top of the ToDoList project. In case you haven't read chapter 22, I recommend you read it first. This will help you better understand the topics we are going to discuss below, especially if you have no experience with Core Data.

Understanding UIViewRepresentable

To use a UIKit view in SwiftUI, you wrap the view with the `UIViewRepresentable` protocol. Basically, you just need to create a `struct` in SwiftUI that adopts the protocol to create and manage a `UIView` object. Here is the skeleton of the custom wrapper for a UIKit view:

```
struct CustomView: UIViewRepresentable {

    func makeUIView(context: Context) -> some UIView {
        // Return the UIView object
    }

    func updateUIView(_ uiView: some UIView, context: Context) {
        // Update the view
    }
}
```

In the actual implementation, you replace `some UIView` with the UIKit view you want to wrap. Let's say, we want to use `UISearchBar` in UIKit. The code can be written like this:

```
struct SearchBar: UIViewRepresentable {

    func makeUIView(context: Context) -> UISearchBar {

        return UISearchBar()
    }

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

        // Update the view
    }
}
```

In the `makeUIView` method, we return an instance of `UISearchBar`. This is how you wrap a UIKit view and make it available to SwiftUI. To use the `SearchBar`, you can treat it like any SwiftUI view and create it like this:

```
struct ContentView: View {
    var body: some View {
        SearchBar()
    }
}
```

Adding a Search Bar

Now back to the *ToDoList* project to add the search bar to the app. First, we will create a new file for the search bar. In the project navigator, right click the *View* folder and choose *New File...*. Select the *SwiftUI View* template and name the file `SearchBar.swift`.

Replace the content with the following code:

```
import SwiftUI

struct SearchBar: UIViewRepresentable {

    @Binding var text: String

    func makeUIView(context: Context) -> UISearchBar {

        let searchBar = UISearchBar()

        searchBar.searchBarStyle = .minimal
        searchBar.autocapitalizationType = .none
        searchBar.placeholder = "Search..."

        return searchBar
    }

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

        uiView.text = text
    }
}
```

The code is similar to the code shown in the previous section but with the following differences:

1. Instead of creating a `UISearchBar` with the default appearance, we initialize it with a minimal style, disable auto capitalization, and update its placeholder value.
2. We have added a binding to hold the search term. While the `makeUIView` method is responsible for creating and initializing the view object, the `updateUIView` method is responsible for updating the state of the UIKit view. Whenever there is a state

change in SwiftUI, the framework automatically calls the `updateUIView` method to update the configuration of the view. In this case, whenever you update the search term in SwiftUI, the method will be called and we will update the `text` of `UISearchBar`.

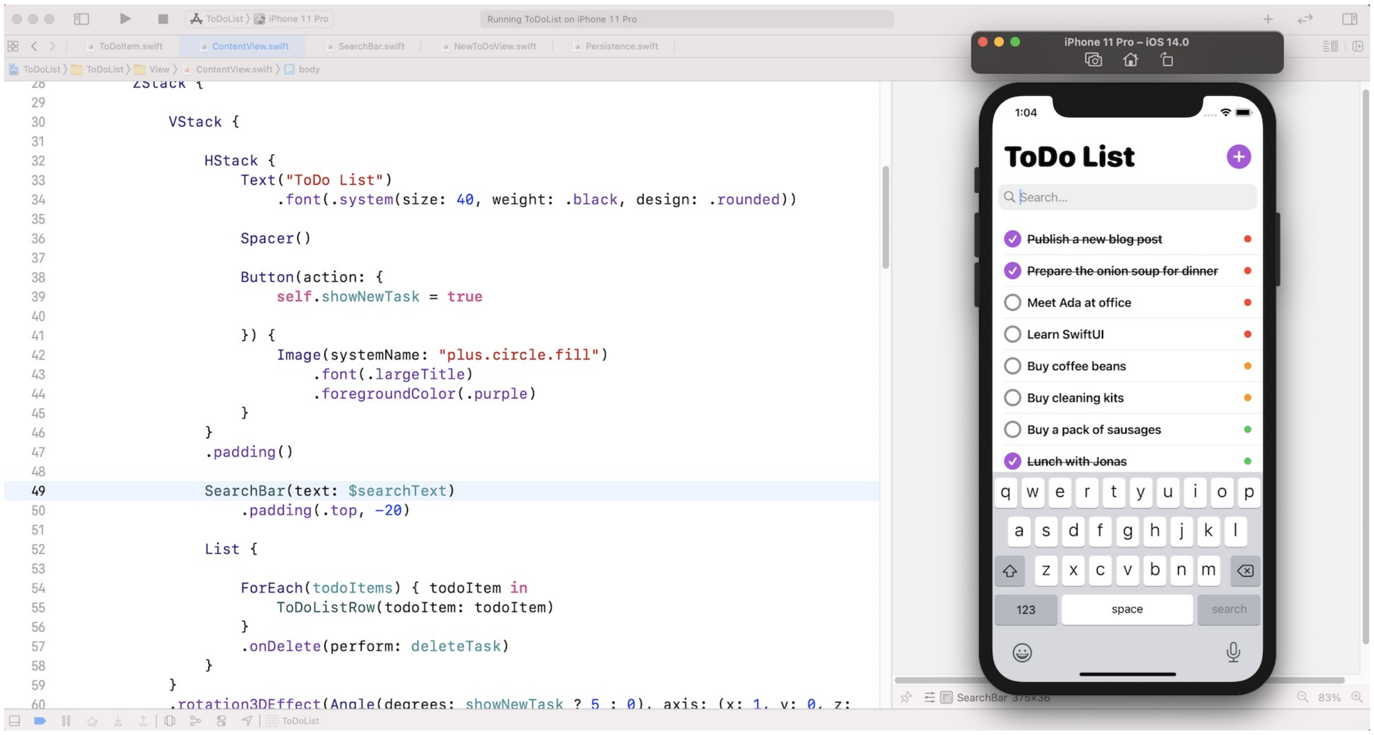
Now switch over to `ContentView.swift`. Declare a state variable to hold the search text:

```
@State private var searchText = ""
```

To present the search bar, insert the following code before the `List`:

```
SearchBar(text: $searchText)
    .padding(.top, -20)
```

The `SearchBar` is just like any other SwiftUI views. You can apply modifiers like padding to adjust the layout. If you run the app in a simulator, you should see a search bar, though it doesn't function yet.



```
ZStack {

    VStack {

        HStack {
            Text("ToDo List")
                .font(.system(size: 40, weight: .black, design: .rounded))

            Spacer()

            Button(action: {
                self.showNewTask = true

            }) {
                Image(systemName: "plus.circle.fill")
                    .font(.largeTitle)
                    .foregroundColor(.purple)
            }
        }
        .padding()

        SearchBar(text: $searchText)
            .padding(.top, -20)

        List {

            ForEach(todoItems) { todoItem in
                ToDoListRow(todoItem: todoItem)
            }
            .onDelete(perform: deleteTask)
        }
    }
    .rotation3DEffect(Angle(degrees: showNewTask ? 5 : 0), axis: (x: 1, y: 0, z:
```

Figure 2. The ToDo app now has a search bar

Capturing the Search Text

It's pretty easy to present a UIKit view in a SwiftUI app. That said, making the search bar work is another story. For now, you can type in the search field but the app doesn't perform the query yet. What we expect is that the app should search the to-do items on the fly as the user keys in the search term.

So, how do we detect the user is entering a search term?

The search bar has a companion protocol named `UISearchBarDelegate`. This protocol provides several methods for managing the search text. In particular, the following method is called whenever the user changes the search text:

```
optional func searchBar(_ searchBar: UISearchBar, textDidChange searchText: String)
```

To make the search bar functional, we have to adopt the `UISearchBarDelegate` protocol. This is where things become more complex.

So far, we have only discussed a couple of the methods in the `UIViewRepresentable` protocol. If you need to work with a delegate in UIKit and communicate back to SwiftUI, you have to implement the `makeCoordinator` method and provide a `Coordinator` instance. This `Coordinator` acts as a bridge between UIView's delegate and SwiftUI. Let's have a look at the code, so you will understand what it means.

In the `SearchBar` struct (SearchBar.swift file), create a `Coordinator` class and implement the `makeCoordinator` method like this:

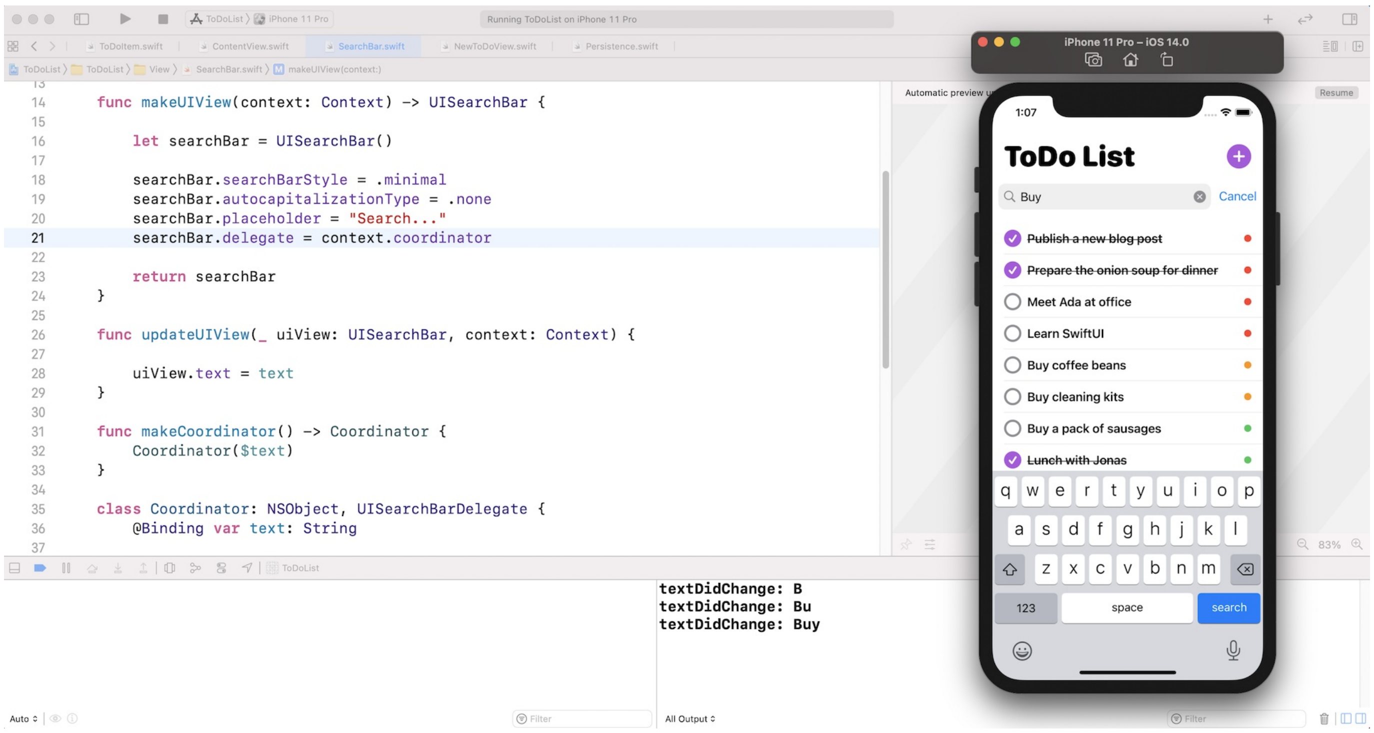
```
func makeCoordinator() -> Coordinator {
    Coordinator($text)
}

class Coordinator: NSObject, UISearchBarDelegate {
    @Binding var text: String

    init(_ text: Binding<String>) {
        self._text = text
    }

    func searchBar(_ searchBar: UISearchBar, textDidChange searchText: String) {

        searchBar.showsCancelButton = true
        text = searchText

        print("textDidChange: \(text)")
    }
}
```

The `makeCoordinator` method simply returns an instance of `Coordinator`. The `Coordinator`, adopts the `UISearchBarDelegate` protocol and implements the `searchBar(_:textDidChange:)` method. As mentioned, this method is called every time a user changes the search text. Therefore, we capture the updated search text and pass it back to SwiftUI by updating the `text` binding. I intentionally added a print statement in the method, so that you can see the changes when we test the app later.

Now that we have a `Coordinator` that adopts the `UISearchBarDelegate` protocol, we need to make one more change. In the `makeUIView` method, insert the following line of code to assign the coordinator to the search bar:

```
searchBar.delegate = context.coordinator
```

That's it! Run the app again and type in the search field. You should see the "textDidChange:" message in the console.



```
    func makeUIView(context: Context) -> UISearchBar {

        let searchBar = UISearchBar()

        searchBar.searchBarStyle = .minimal
        searchBar.autocapitalizationType = .none
        searchBar.placeholder = "Search..."
        searchBar.delegate = context.coordinator

        return searchBar
    }

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

        uiView.text = text
    }

    func makeCoordinator() -> Coordinator {
        Coordinator($text)
    }

    class Coordinator: NSObject, UISearchBarDelegate {
        @Binding var text: String
```

```
textDidChange: B
textDidChange: Bu
textDidChange: Buy
```

Figure 3. The console displays the message as you type

Handling the Cancel Button

Did you tap the *Cancel* button? If you've tried that, you know it is not functional. To make it work, we have to implement the following methods in the `Coordinator`:

```
func searchBarCancelButtonClicked(_ searchBar: UISearchBar) {
    text = ""
    searchBar.resignFirstResponder()
    searchBar.showsCancelButton = false
    searchBar.endEditing(true)
}

func searchBarShouldBeginEditing(_ searchBar: UISearchBar) -> Bool {
    searchBar.showsCancelButton = true

    return true
}
```

The first method is triggered when the cancel button is clicked. In the code, we call `resignFirstResponder()` to dismiss the keyboard and tell the search bar to end the editing. The second method ensures that the *Cancel* button appears when the user taps the search field.

You can perform a quick test by running the app in a simulator. Tapping the *Cancel* button while editing should dismiss the software keyboard.

Performing the Search

We can now retrieve the search text and handle the cancel button. Unfortunately, the search bar is still not working yet. This is what we are going to implement in this section. For this app, there are a couple of ways to perform the search:

1. Perform the search on the `todoItems` using the `filter` function
2. Perform the search on the `FetchRequest` by providing a predicate

Basically the first approach is good enough for this app because the `todoItems` is in sync with the to-do item stored in the database. I also want to show you how to perform the search using `FetchRequest`. So, we will look into both approaches.

Using the filter function

In Swift, you can use the `filter` function to loop over a collection and get an array of items that matches the filter criteria. Here is an example:

```
todoItems.filter({ $0.name.contains("Buy") })
```

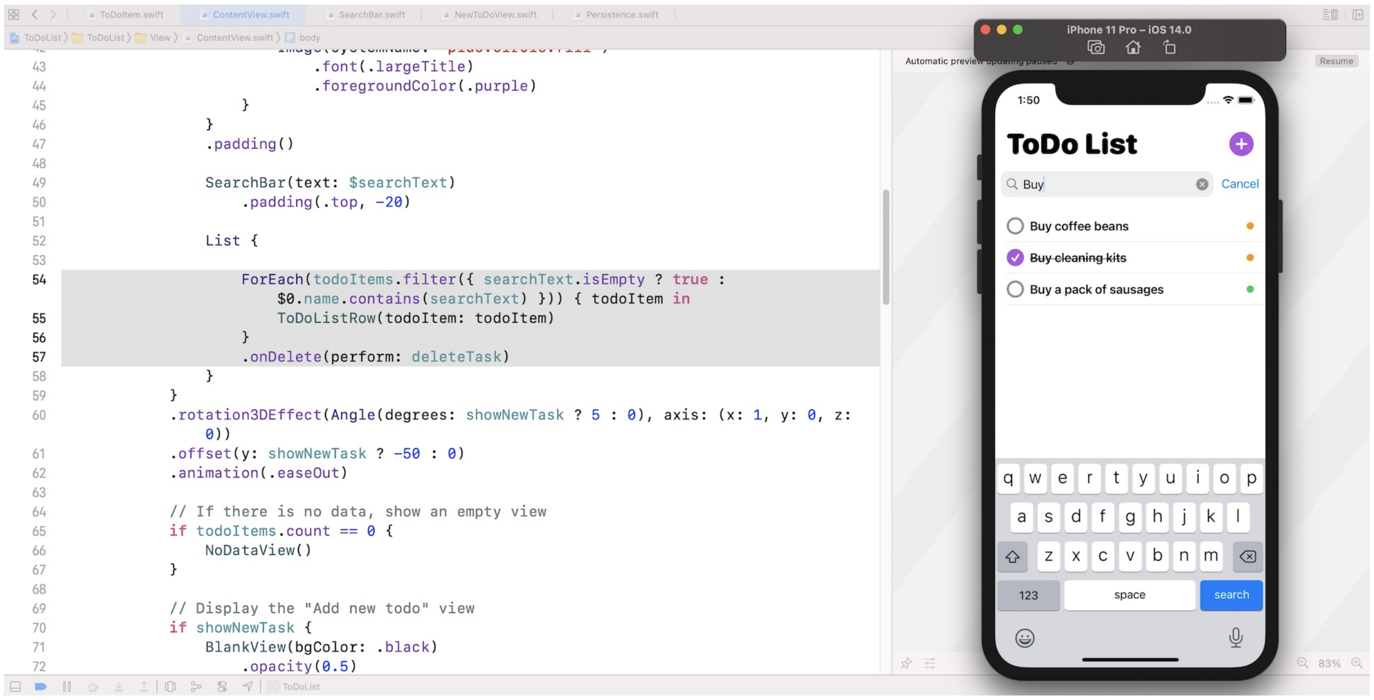
The `filter` function takes a closure as an argument that specifies the filter criteria. For example, the code above will return those items that contain the keyword "Buy" in its name field.

To implement the search, we can replace the `ForEach` loop of the `List` like this:

```
ForEach(todoItems.filter({ searchText.isEmpty ? true : $0.name.contains(searchText
) })) { todoItem in
    ToDoListRow(todoItem: todoItem)
}
.onDelete(perform: deleteTask)
```

In the closure of the `filter` function, we first check if the search text has a value. If not, we simply return `true` , which means that it returns all the items. Otherwise, we check if the name field contains the search term.

That's it. You can now run the app to test it out. Type in the search field and the app will filter those records that match the search term.



```
                .font(.largeTitle)
                .foregroundColor(.purple)
        }
    }
    .padding()

    SearchBar(text: $searchText)
        .padding(.top, -20)

    List {

        ForEach(todoItems.filter({ searchText.isEmpty ? true :
            $0.name.contains(searchText) })) { todoItem in
            ToDoListRow(todoItem: todoItem)
        }
        .onDelete(perform: deleteTask)
    }
}
.rotation3DEffect(Angle(degrees: showNewTask ? 5 : 0), axis: (x: 1, y: 0, z:
    0))
.offset(y: showNewTask ? -50 : 0)
.animation(.easeOut)

// If there is no data, show an empty view
if todoItems.count == 0 {
    NoDataView()
}

// Display the "Add new todo" view
if showNewTask {
    BlankView(bgColor: .black)
        .opacity(0.5)
```

Figure 4. Filtering the todo items

Using FetchRequest

The filter approach performs the search on the existing fetch results. The other approach is to perform the search directly using Core Data. When we fetch the data from database, we specify clearly the todo items to retrieve.

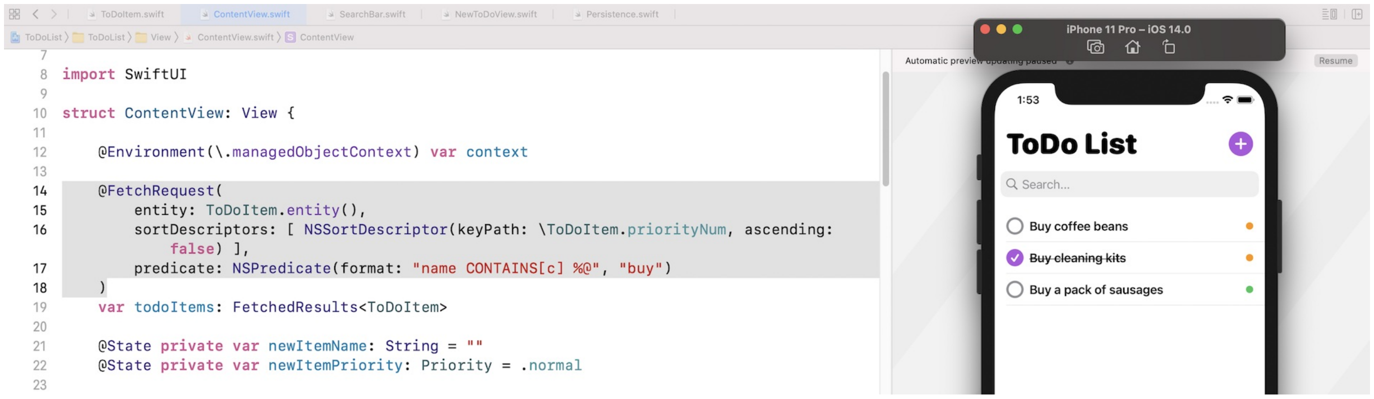
The `@FetchRequest` property wrapper allows you to pass a predicate, which we haven't discussed before, to specify the filter criteria.

Here is an example:

```
@FetchRequest(
    entity: ToDoItem.entity(),
    sortDescriptors: [ NSSortDescriptor(keyPath: \ToDoItem.priorityNum, ascending:
false) ],
    predicate: NSPredicate(format: "name CONTAINS[c] %@", "Buy")
)
```

By providing the predicate property, the fetch request will only fetch the to-do items who's name field contains the search term "buy". The `[c]` following `CONTAINS` means that the search is case insensitive. If you want to test it, please make sure you revert the `ForEach` to the original code (without the filter function). And then replace the `@FetchRequest` with the code above.

Assuming you've added some todo items with "Buy" in the item name, you should only see the to-do items with the search term "buy" after the code change.



```
import SwiftUI

struct ContentView: View {

    @Environment(\.managedObjectContext) var context

    @FetchRequest(
        entity: ToDoItem.entity(),
        sortDescriptors: [ NSSortDescriptor(keyPath: \ToDoItem.priorityNum, ascending:
            false) ],
        predicate: NSPredicate(format: "name CONTAINS[c] %@", "buy")
    )
    var todoItems: FetchedResults<ToDoItem>

    @State private var newItemName: String = ""
    @State private var newItemPriority: Priority = .normal
```

Figure 5. The app only displays the to-do items containing the keyword "buy"

It looks simple, right? But when you need to create a fetch request with a dynamic predicate, then it is not that simple. Once the fetch request is initialized with a specific predicate, you can't change it. The same goes for the sort descriptor.

So, how do we build a fetch request that supports different predicates?

The trick is not to use the `@FetchRequest` property wrapper. Instead, we create the fetch request manually. In order to do that, we will create a separate view called `FilteredList` which accepts the search text as an argument. This `FilteredList` is responsible to create the fetch request, search for the related to-do items, and present them in a list view.

In `ContentView.swift`, insert the following code to create the `FilteredList`:

```
struct FilteredList: View {

    @Environment(\.managedObjectContext) var context
    @Binding var searchText: String

    var fetchRequest: FetchRequest<ToDoItem>
    var todoItems: FetchedResults<ToDoItem> {
        fetchRequest.wrappedValue
    }

    init(_ searchText: Binding<String>) {
        self._searchText = searchText

        let predicate = searchText.wrappedValue.isEmpty ? NSPredicate(value: true)
   : NSPredicate(format: "name CONTAINS[c] %@", searchText.wrappedValue)

        self.fetchRequest = FetchRequest(entity: ToDoItem.entity(),
                                         sortDescriptors: [ NSSortDescriptor(keyPa
th: \ToDoItem.priorityNum, ascending: false) ],
                                         predicate: predicate,
                                         animation: .default)
    }

    var body: some View {

        ZStack {
            List {

                ForEach(todoItems) { todoItem in
                    ToDoListRow(todoItem: todoItem)
                }
                .onDelete(perform: deleteTask)

            }

            if todoItems.count == 0 {
                NoDataView()
            }
        }


    }

    private func deleteTask(indexSet: IndexSet) {
        for index in indexSet {
            let itemToDelete = todoItems[index]
            context.delete(itemToDelete)
        }

        do {
            try context.save()
        } catch {
            print(error)
        }
    }
}
```

Take a look at the `body` and `deleteTask`. Both are exactly the same as before. We just extract the code and put them in the `FilteredList`. The core changes are in the `init` method and the fetch request.

We declare a variable named `fetchRequest` to hold the fetch request and another variable named `todoItems` to store the fetched results. The fetched results can actually be retrieved from the `wrappedValue` property of the fetch request.

Now let's dive into the `init` method. This custom `init` method accepts the search text as an argument. To be clear, it's the binding for the search text. The reason why we need to create a custom `init` is that we are creating a dynamic fetch request based on the given search text.

The first line of the `init` method is to store the binding of the search text. To assign a binding, you use the underscore like this:

```
self._searchText = searchText
```

Next, we check if the search text is empty (or not) and build the predicate accordingly:

```
let predicate = searchText.wrappedValue.isEmpty ? NSPredicate(value: true) : NSPre
dicate(format: "name CONTAINS[c] %@", searchText.wrappedValue)
```

Once the predicate is ready, we create the fetch request like this:

```
self.fetchRequest = FetchRequest(entity: ToDoItem.entity(),
                                 sortDescriptors: [ NSSortDescriptor(keyPath: \ToD
oItem.priorityNum, ascending: false) ],
                                 predicate: predicate,
                                 animation: .default)
```

As you can see, the usage is very similar to that of the `@FetchRequest` property wrapper.

This is it! We now have a `FilteredList` that can handle a fetch request with different predicates. Now let's modify the `ContentView` struct to make use of this new `FilteredList`.

Since we've moved the fetch request to `FilteredList`, we can delete the following variables:

```
@Environment(\.managedObjectContext) var context

@FetchRequest(
    entity: ToDoItem.entity(),
    sortDescriptors: [ NSSortDescriptor(keyPath: \ToDoItem.priorityNum, ascending:
false) ],
    predicate: NSPredicate(format: "name CONTAINS[c] %@", "buy")
)
var todoItems: FetchedResults<ToDoItem>
```

Next, replace the following code:

```
List {

    ForEach(todoItems.filter({ searchText.isEmpty ? true : $0.name.contains(search
Text) })) { todoItem in
        ToDoListRow(todoItem: todoItem)
    }
    .onDelete(perform: deleteTask)

}
```

With:

```
FilteredList($searchText)
```

Here we use the `FilteredList` to render the list view. We pass the binding of `searchText` for performing the search. Since `searchText` is a state variable, any change on the search text will trigger the update of the `FilteredList`. In reality, the app creates a different predicate and fetches a new set of to-do items as the user types in the search field.

Next, remove the following code because it's in the `FilteredList` also:

```
// If there is no data, show an empty view
if todoItems.count == 0 {
    NoDataView()
}
```

Finally, delete the following code from the `deleteTask` method:

```
private func deleteTask(indexSet: IndexSet) {
    for index in indexSet {
        let itemToDelete = todoItems[index]
        context.delete(itemToDelete)
    }

    do {
        try context.save()
    } catch {
        print(error)
    }
}
```

Now you're ready to test! If you've made all the code changes correctly, the app should filter the to-do items as you type in the search term.

Summary

In this chapter, you've learned how to use the `UIViewRepresentable` protocol to integrate UIKit views with SwiftUI. While SwiftUI is still very new and doesn't come with all the standard UI components, this backward compatibility allows you to tap into the old framework and utilize any views you need.

We also explored a couple of approaches for performing data search. You should now know how to use the `filter` function and understand how to create a dynamic fetch request.

For reference, you can download the complete project here:

- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUIToDoListUISearchBar.zip)

Chapter 24
Creating a Search Bar View and Working with Custom Binding

Previously, we showed you how to implement a search bar by reusing the `UISearchBar` component of the old UIKit framework. Have you ever thought of building one from scratch? If you look at the search bar carefully, it's not too difficult to implement. So, let's try to build a SwiftUI version of a search bar in this chapter.

Not only will you learn how to create the search bar view, we will show you how to work with custom bindings. We've discussed bindings before, but haven't showed you how to create a custom binding. Custom binding is particularly useful when you need to insert additional program logic while it's being read and written. In addition to all that, you will learn how to dismiss the software keyboard in SwiftUI.

Figure 1 shows you the search bar we're going to build. The look & feel is the same as that of `UISearchBar` in UIKit. We will also implement the *Cancel* button which only appears when the user starts typing in the search field.

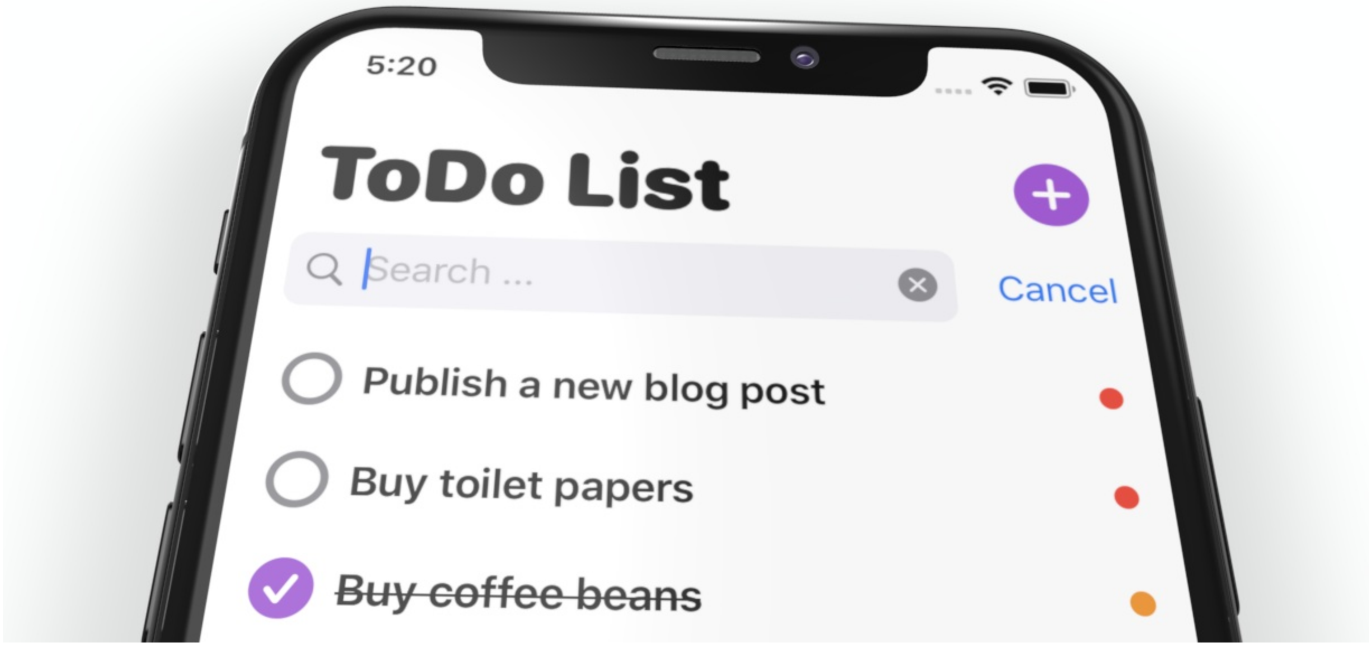


Figure 1. Building a search bar view entirely using SwiftUI

Implementing the Search Bar UI

We will convert the previous project from `UISearchBar` to our own implementation of search bar. To get started, please download the starter project from https://www.appcoda.com/resources/swiftui2/SwiftUIToDoListUISearchBar.zip. Once you download it, compile it to make sure it works. The app should show you a search bar, however, this bar is from UIKit. We are going to convert it to a search bar view built entirely using SwiftUI.

Open `SearchBar.swift`, which is the file we will focus on. We will rewrite the whole code but keep its struct name intact. We still call it `SearchBar`, which still accepts a binding of search text as an argument. To the caller (i.e. ContentView), there is nothing to change. The usage is still like this:

```
SearchBar(text: $searchText)
```

Now, let's begin with the UI implementation. If you want to challenge yourself, stop reading here and try to implement the search bar UI on your own. This UI is quite simple. It's composed of a text field, a couple of icons, and the cancel button.

If you have no idea how the UI is built, let's create it together. Replace the `SearchBar` struct in `SearchBar.swift` like this:

```
struct SearchBar: View {
    @Binding var text: String

    @State private var isEditing = false

    var body: some View {
        HStack {

            TextField("Search ...", text: $text)
                .padding(7)
                .padding(.horizontal, 25)
                .background(Color(.systemGray6))
                .cornerRadius(8)
                .overlay(
                    HStack {
                        Image(systemName: "magnifyingglass")
                            .foregroundColor(.gray)
                            .frame(minWidth: 0, maxWidth: .infinity, alignment: .l
eading)
                            .padding(.leading, 8)

                        if isEditing {
                            Button(action: {
                                self.text = ""
                            }) {
                                Image(systemName: "multiply.circle.fill")
                                    .foregroundColor(.gray)
                                    .padding(.trailing, 8)
                            }
                        }
                    }
                )
                .padding(.horizontal, 10)
                .onTapGesture {
                    self.isEditing = true
                }


            if isEditing {
                Button(action: {
                    self.isEditing = false
                    self.text = ""

                }) {
                    Text("Cancel")
                }
                .padding(.trailing, 10)
                .transition(.move(edge: .trailing))
                .animation(.default)
            }
        }
    }
}
```

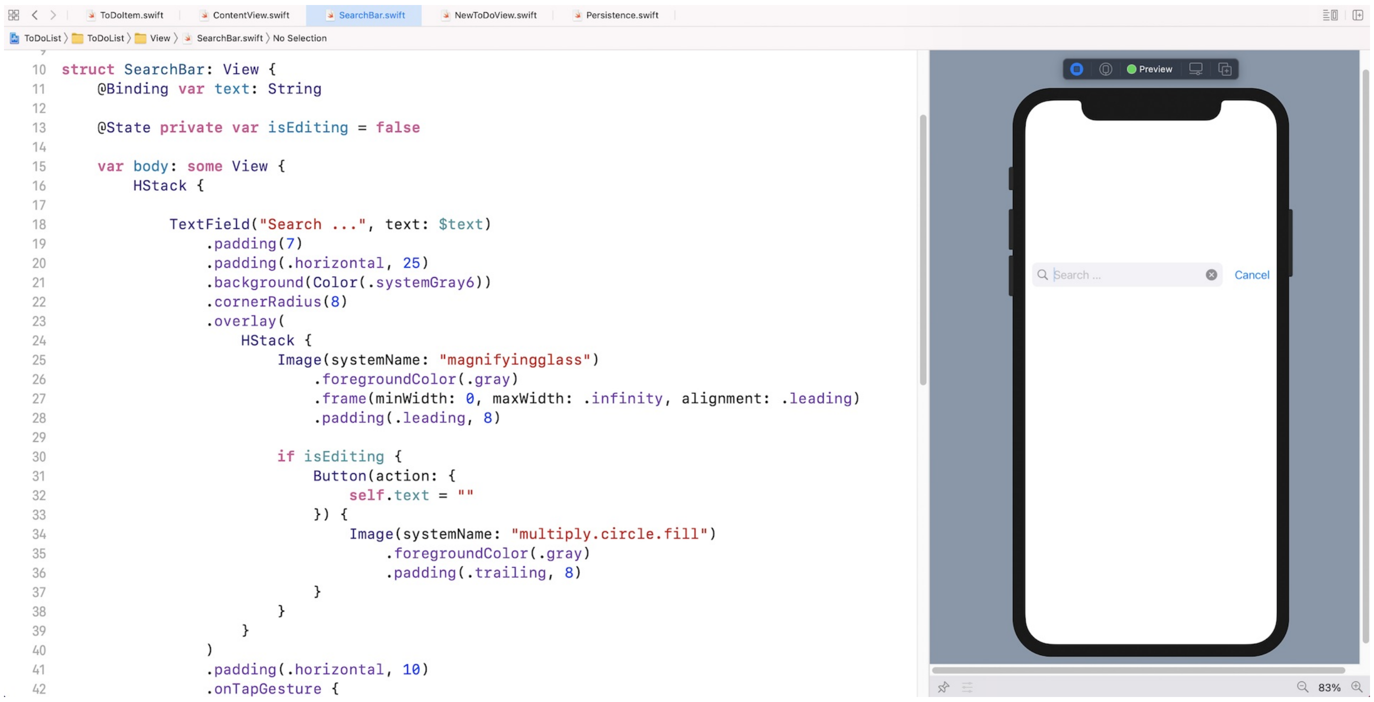
First, we declare two variables: one is the binding of the search text and the other one is a variable for storing the state of the search field (editing or not).

We used a `HStack` to layout the text field and the *Cancel* button. For the text field, we overlay a magnifying glass icon and the cross icon (i.e. multiply.circle.fill), which is only displayed when the search field is in editing mode. The same goes for the *Cancel* button, which appears when the user taps the search field.

In order to preview the search bar, please also insert the following code:

```
struct SearchBar_Previews: PreviewProvider {
    static var previews: some View {
        SearchBar(text: .constant(""))
    }
}
```

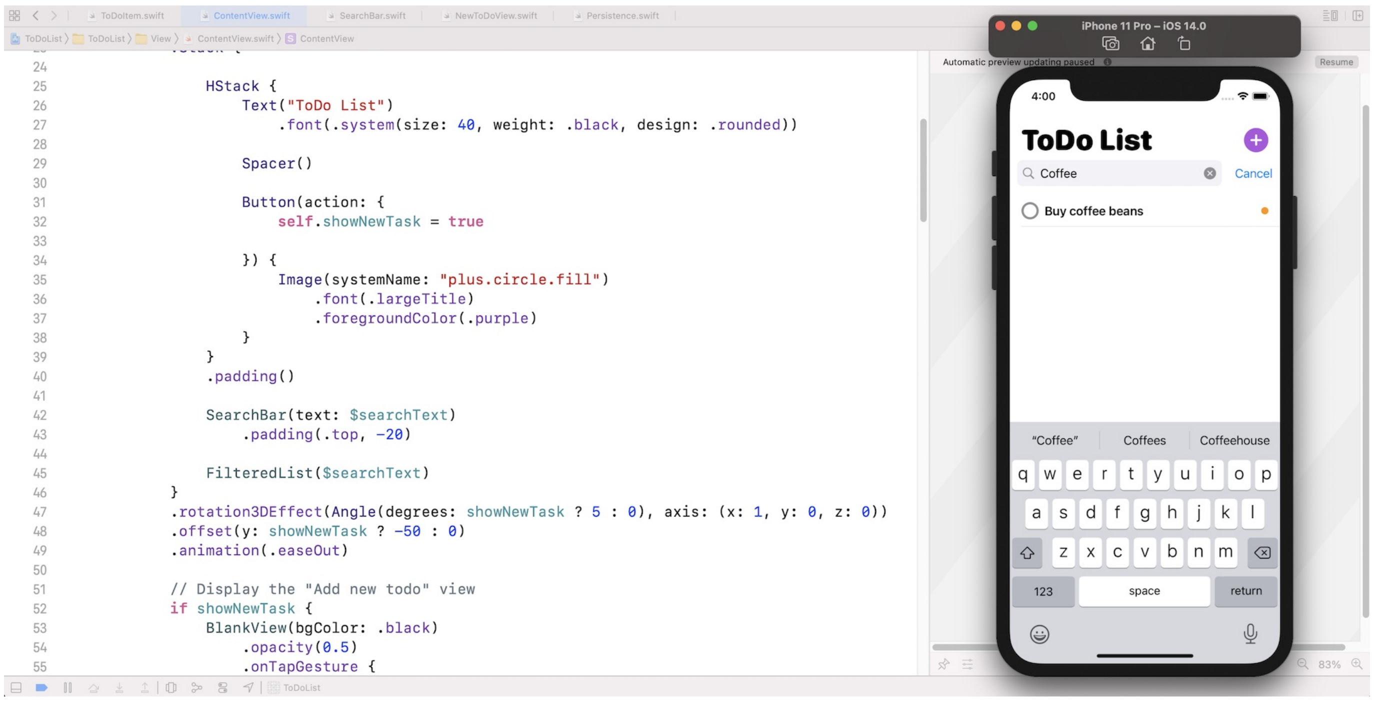
Once you have added the code, you should be able to preview the search field. Click the *play* button to test the search field. When you select the text field, the *Cancel* button should appear.



```
struct SearchBar: View {
    @Binding var text: String

    @State private var isEditing = false

    var body: some View {
        HStack {

            TextField("Search ...", text: $text)
                .padding(7)
                .padding(.horizontal, 25)
                .background(Color(.systemGray6))
                .cornerRadius(8)
                .overlay(
                    HStack {
                        Image(systemName: "magnifyingglass")
                            .foregroundColor(.gray)
                            .frame(minWidth: 0, maxWidth: .infinity, alignment: .leading)
                            .padding(.leading, 8)

                        if isEditing {
                            Button(action: {
                                self.text = ""
                            }) {
                                Image(systemName: "multiply.circle.fill")
                                    .foregroundColor(.gray)
                                    .padding(.trailing, 8)
                            }
                        }
                    }
                )
                .padding(.horizontal, 10)
                .onTapGesture {
```

Figure 2. Previewing the search bar

What's more is that the search bar already works! Run the app on a simulator and perform a search. It should filter the result based on the search term.



```
                HStack {
                    Text("ToDo List")
                        .font(.system(size: 40, weight: .black, design: .rounded))

                    Spacer()

                    Button(action: {
                        self.showNewTask = true

                    }) {
                        Image(systemName: "plus.circle.fill")
                            .font(.largeTitle)
                            .foregroundColor(.purple)
                    }
                }
                .padding()

                SearchBar(text: $searchText)
                    .padding(.top, -20)

                FilteredList($searchText)
            }
            .rotation3DEffect(Angle(degrees: showNewTask ? 5 : 0), axis: (x: 1, y: 0, z: 0))
            .offset(y: showNewTask ? -50 : 0)
            .animation(.easeOut)

            // Display the "Add new todo" view
            if showNewTask {
                BlankView(bgColor: .black)
                    .opacity(0.5)
                    .onTapGesture {
```

Figure 3. The search bar is already functional

Dismissing the Keyboard

As you can see, it's not hard to create our own search bar entirely using SwiftUI. While the search bar is working, there is a minor issue we have to fix. Have you tried to tap the cancel button? It does clear the search field. However, the software keyboard is not dismissed.

To fix that, we need to add a line of code in the `action` block of the *Cancel* button in `SearchBar.swift`:

```
// Dismiss the keyboard
UIApplication.shared.sendAction(#selector(UIResponder.resignFirstResponder), to: n
il, from: nil, for: nil)
```

In the code, we call the `sendAction` method to resign the first responder and dismiss the keyboard. You can now run the app using a simulator. When you tap the cancel button, it should clear the search field and dismiss the software keyboard.

Working with Custom Binding

The SwiftUI version of search bar already functions properly, but I want to take this opportunity to discuss custom binding with you. In `SearchBar.swift`, we declare a binding of the search text like this:

```
@Binding var text: String
```

It works great for our current implementation. But let me ask you. What if we needed to add extra logic when reading or writing this binding? For example, how can you capitalize each word in the search field?

Swift has a built-in feature to capitalize a string. You can use the `capitalized` property of the text and retrieve the capitalized text. The question is how do we update the binding of `text`?

In this case, you will need to create a custom binding in `SearchBar.swift` like this:

```
private var searchText: Binding<String> {

    return Binding<String>(
        get: {
            self.text.capitalized

        }, set: {
            self.text = $0
        }
    )
}
```

In the code above, we create a custom binding named `searchText` with closures that read (get) and write (set) the binding value. For the `get` part, we customize the binding value of `text` by accessing the `capitalized` property. This is how we capitalize each word the user types in the search field. For the `set` part, we do not make any changes and set it to the original value. However, if you need to add extra logic when setting the binding, you can modify the code in `set`.

As a side note, you can omit the `return` keyword and write the binding like this:

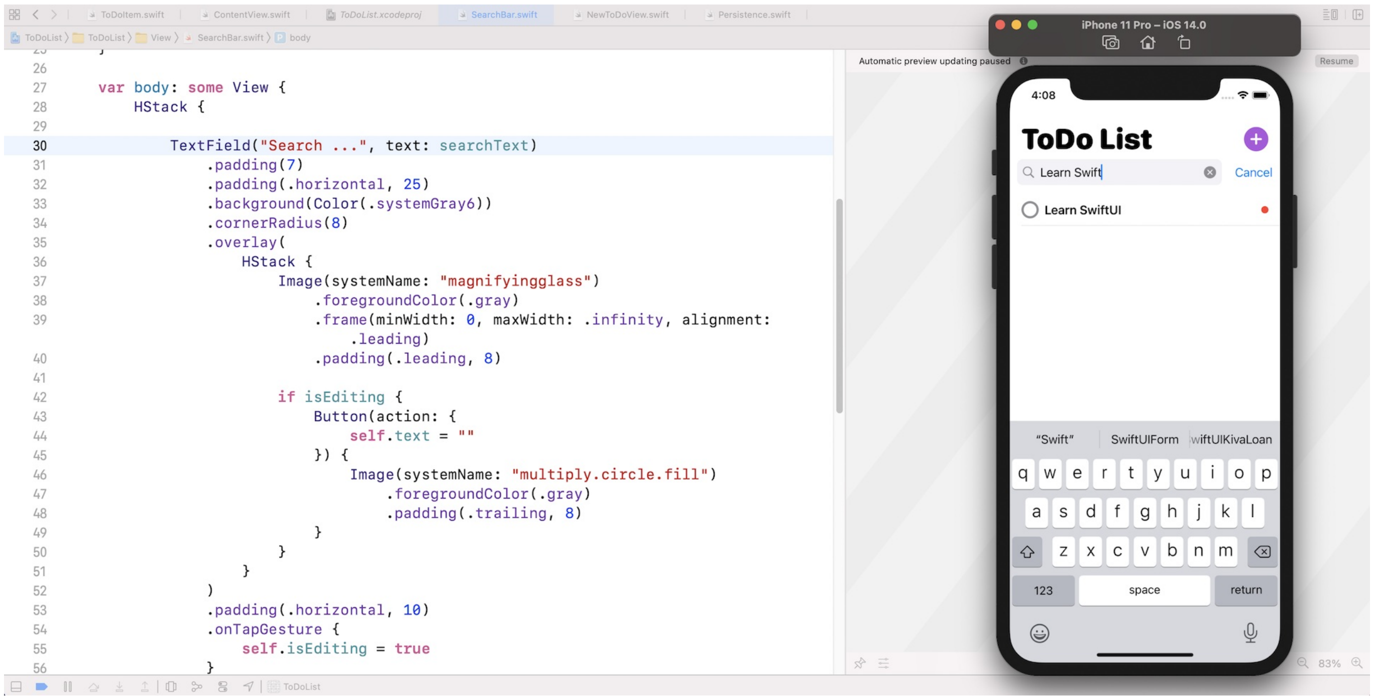
```
private var searchText: Binding<String> {

    Binding<String>(
        get: {
            self.text.capitalized

        }, set: {
            self.text = $0
        }
    )
}
```

This is a new feature in Swift 5.1, in case you are not aware.

We are still passing the `text` binding to `TextField`. Before the custom binding change takes effect, we will need to make one more change. Modify the parameter in `TextField` and make sure you pass the `searchText` as the binding:

```
TextField("Search ...", text: searchText)
```

Now run the app on a simulator. Type a few words into the search field. The app should automatically capitalize each word as you type.



```
    var body: some View {
        HStack {

            TextField("Search ...", text: searchText)
                .padding(7)
                .padding(.horizontal, 25)
                .background(Color(.systemGray6))
                .cornerRadius(8)
                .overlay(
                    HStack {
                        Image(systemName: "magnifyingglass")
                            .foregroundColor(.gray)
                            .frame(minWidth: 0, maxWidth: .infinity, alignment:
                                .leading)
                            .padding(.leading, 8)

                        if isEditing {
                            Button(action: {
                                self.text = ""
                            }) {
                                Image(systemName: "multiply.circle.fill")
                                    .foregroundColor(.gray)
                                    .padding(.trailing, 8)
                            }
                        }
                    }
                )
                .padding(.horizontal, 10)
                .onTapGesture {
                    self.isEditing = true
                }
```

Figure 4. The search field automatically capitalizes each word as you type

Summary

In this chapter, we showed you another approach to implementing a search bar. As you can see, it's not difficult to build one entirely using SwiftUI. You've also learned how to create a custom binding. This is very useful when you need to add extra program logic when setting or retrieving the binding value.

For reference, you can download the complete search bar project here:

- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUIToDoListSearchBarView.zip)

Chapter 25
Putting Everything Together to Build a Personal Finance App

By now, you should have a good understanding of SwiftUI and have built some simple apps using this new framework. In this chapter, you are going to use what you've learned so far to develop a personal finance app, allowing users to keep track of their income and expenses.

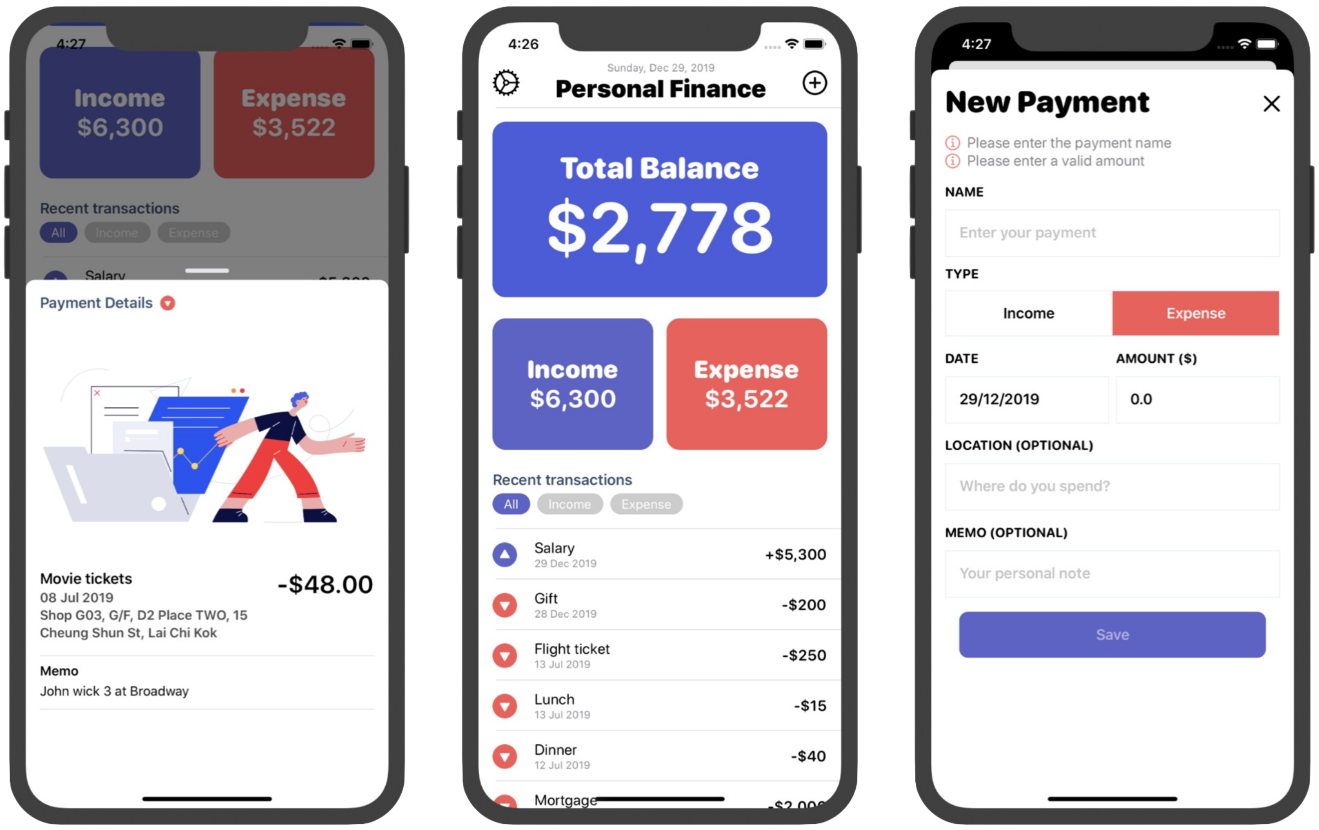


Figure 1. The Personal Finance App

This app is not too complicated to build but you will learn quite a lot about SwiftUI and understand how to apply the techniques you learned in developing this real world app. In brief, here is some of the stuff we will go through with you:

1. How to build a form and perform validation
2. How to filter records and refresh the list view
3. How to use bottom sheet to display record details
4. How to use MVVM (Model-View-ViewModel) in SwiftUI
5. How to save and manage data in a database using Core Data
6. How to use DatePicker for date selection
7. How to handle keyboard notification and adjust the form position

Let me stress this once again. This app is the result of what you learned so far. Therefore, I assume you have already read the book from chapter 1 to chapter 24. You should understand how a bottom sheet is built (chapter 18), how form validation with Combine works (chapter 14 & 15), and how to persist data using Core Data (chapter 22). If you haven't read these chapters, I suggest you go read them first. In this chapter, we will mostly focus on techniques that haven't been discussed before.

Downloading the Complete Project

Normally, we build a demo app from scratch. This time is a bit different. I've already built the Personal Finance app for you. You can download the full source code of the project from https://www.appcoda.com/resources/swiftui2/SwiftUIPFinance.zip to take a look. Unzip the project and run the app on a simulator to try it out. When the app is first launched, it looks different from the one shown in figure 1 because there are no records. You can tap the + button to add a new record. After you go back to the main view, you will see the new record in the *Recent Transactions* section. And, the total balance is automatically calculated.

The app uses Core Data for data management. The records are persisted locally in the built-in database, so you should see the records even after restarting the app.

For the rest of the chapter, I will explain how the code works in detail. But I encourage you to take a look at the code first to see how much you understand.

Understanding the Model

As you can see in the project navigator, the app is broken into three main parts: model, view model and view. Let's begin with the model layer and Core Data model. Open the `PaymentActivity.swift` file to take a look:

```
enum PaymentCategory: Int {
    case income = 0
    case expense = 1
}

public class PaymentActivity: NSManagedObject {

    @NSManaged public var paymentId: UUID
    @NSManaged public var date: Date
    @NSManaged public var name: String
    @NSManaged public var address: String?
    @NSManaged public var amount: Double
    @NSManaged public var memo: String?
    @NSManaged public var typeNum: Int32
}

extension PaymentActivity: Identifiable {
    var type: PaymentCategory {
        get {
            return PaymentCategory(rawValue: Int(typeNum)) ?? .expense
        }

        set {
            self.typeNum = Int32(newValue.rawValue)
        }
    }
}
```

The `PaymentActivity` class represents a payment record which can either be an expense or income. In the code above, we use an `enum` to differentiate the payment types. Each payment has the following properties:

- **paymentId** - an unique ID for the payment record

- **date** - the date of the transaction
- **name** - the name of the transaction
- **address** - where you spend / where the income comes from
- **amount** - the amount of the transaction
- **memo** - additional notes for the payment
- **typeNum** - the payment type (income / expense)

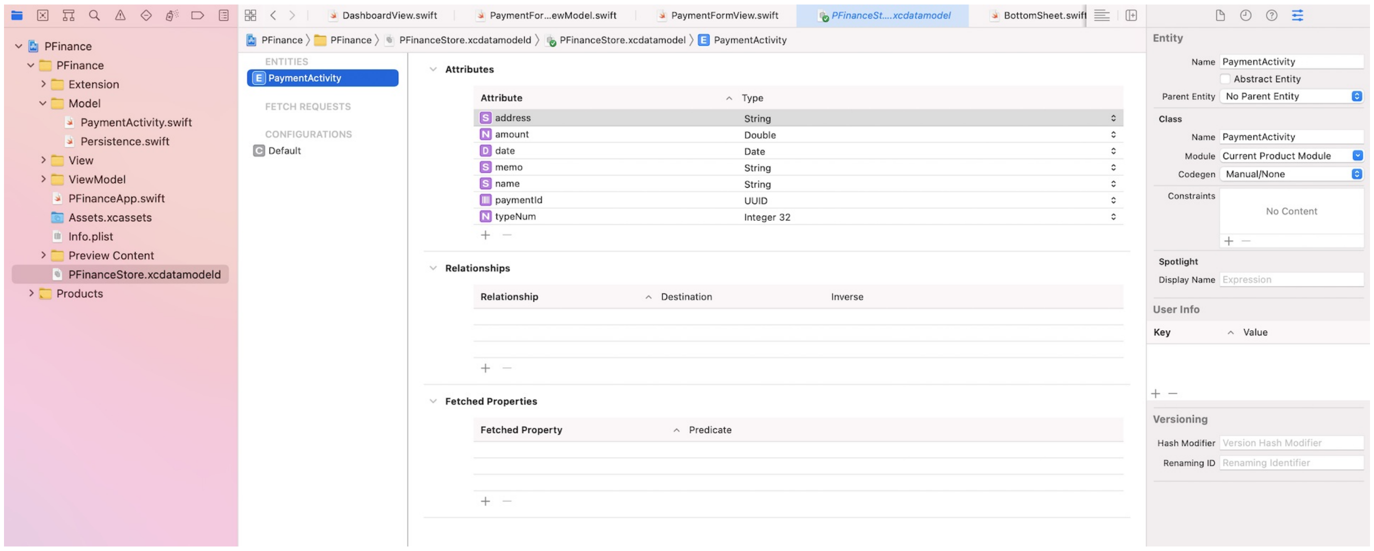
Since we use Core Data to persist the payment activity, this `PaymentActivity` class inherits from `NSManagedObject`. Later, you will see in the Core Data model that this class is set as a custom class of the managed object. Again, if you don't understand Core Data, please refer to chapter 22.

The payment type (i.e. `typeNum`), is saved as an integer in the database. Therefore, we need a conversion between the integer and the actual enumeration. This is one approach to save an enum in a persistent storage.

Lastly, we adopt the `Identifiable` protocol. Why do we need to adopt it? We will use the `List` view to present all the payment activities. This is why the `PaymentActivity` class adopts the protocol. If you forget what the `Identifiable` protocol is, you can read about it in chapter 10.

Working with Core Data

Now, open `PFinanceStore.xcdatamodeld` to have a look at the managed data model. In the model, we only have one entity, *PaymentActivity*.



Attribute	Type
address	String
amount	Double
date	Date
memo	String
name	String
paymentId	UUID
typeNum	Integer 32

Relationship	Destination	Inverse

Fetched Property	Predicate

Entity

Name	PaymentActivity
	Abstract Entity
Parent Entity	No Parent Entity

Class

Name	PaymentActivity
Module	Current Product Module
Codegen	Manual/None
Constraints	No Content

Spotlight

Display Name	Expression

User Info

Key	Value

Versioning

Hash Modifier	Version Hash Modifier
Renaming ID	Renaming Identifier

Figure 2. The PaymentActivity entity

This class, as we discussed earlier, is the custom class of this entity. You can click the Data Model inspector to reveal the settings. As mentioned before, I prefer to create the custom class manually (instead of codegen). This gives me more flexiblity to customize the class.

Next, let's head over to `Persistence.swift` (inside the *Model* group) to see how this data model is loaded. In the `PersistenceController` struct, you should see the following code:

```
struct PersistenceController {
    static let shared = PersistenceController()

    let container: NSPersistentContainer

    .
    .
    .

    init(inMemory: Bool = false) {
        container = NSPersistentContainer(name: "PFinanceStore")
        if inMemory {
            container.persistentStoreDescriptions.first!.url = URL(fileURLWithPath
: "/dev/null")
        }
        container.loadPersistentStores(completionHandler: { (storeDescription, err
or) in
            if let error = error as NSError? {
                fatalError("Unresolved error \(error), \(error.userInfo)")
            }
        })
    }
}
```

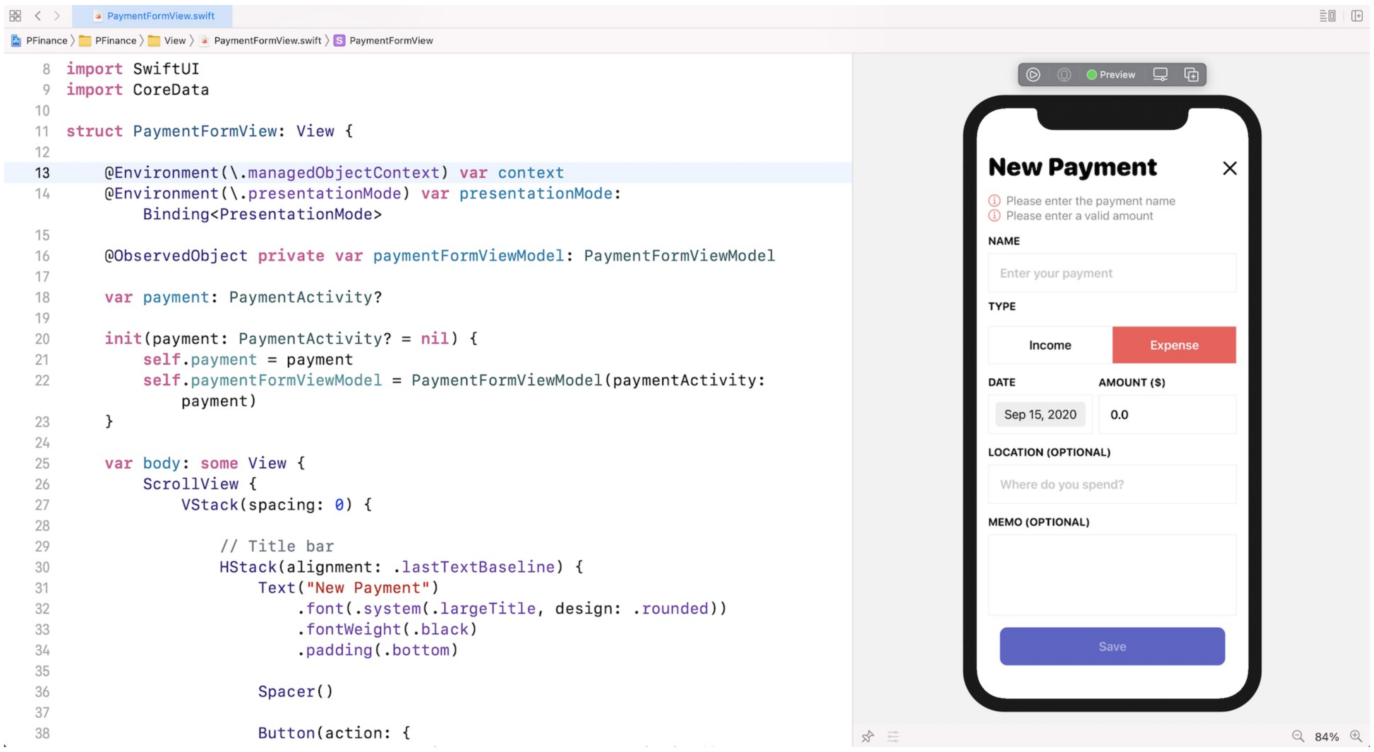
When the app starts, we load the `PFinanceStore.xcdatamodeld` using `NSPersistentContainer`. Now, switch over to `PFinanceApp.swift` and check out the code:

```
struct PFinanceApp: App {

    let persistenceController = PersistenceController.shared

    var body: some Scene {
        WindowGroup {
            DashboardView().environment(\.managedObjectContext, persistenceControl
ler.container.viewContext)
        }
    }
}
```

To trick to using Core Data in SwiftUI, is to inject the managed object context into the environment. Later, in the SwiftUI views, we can easily grab the context from the environment for further operations.

Implementing the New Payment View

Now that we have completed the walkthrough of the model layer, let's see how we implement each of the views. The *New Payment* view is designed for users to create a new payment activity. Open the `PaymentFormView.swift` file to take a look. You should be able to preview the input form.



```swift
import SwiftUI
import CoreData

struct PaymentFormView: View {

    @Environment(\.managedObjectContext) var context
    @Environment(\.presentationMode) var presentationMode:
        Binding<PresentationMode>

    @ObservedObject private var paymentFormViewModel: PaymentFormViewModel

    var payment: PaymentActivity?

    init(payment: PaymentActivity? = nil) {
        self.payment = payment
        self.paymentFormViewModel = PaymentFormViewModel(paymentActivity:
            payment)
    }

    var body: some View {
        ScrollView {
            VStack(spacing: 0) {

                // Title bar
                HStack(alignment: .lastTextBaseline) {
                    Text("New Payment")
                        .font(.system(.largeTitle, design: .rounded))
                        .fontWeight(.black)
                        .padding(.bottom)

                    Spacer()

                    Button(action: {
```

Figure 3. The Payment Form View

The Form Layout

Let me first walk you through how the form is laid out. It's always good practice to extract common views to create a more generic version. Since most of the form fields are very similar, we created a generic text field (i.e. `FormTextField`) to render the field name and the placeholder using a `VStack`:

```
struct FormTextField: View {
    let name: String
    var placeHolder: String

    @Binding var value: String

    var body: some View {
        VStack(alignment: .leading) {
            Text(name.uppercased())
                .font(.system(.subheadline, design: .rounded))
                .fontWeight(.bold)
                .foregroundColor(.primary)

            TextField(placeHolder, text: $value)
                .font(.headline)
                .foregroundColor(.primary)
                .padding()
                .border(Color("Border"), width: 1.0)

        }
    }
}
```

Do you notice the two validation errors under the form title? Since these validation messages have a similar format, we also create a generic view for this kind of message:

```
struct ValidationErrorText: View {

    var iconName = "info.circle"
    var iconColor = Color(red: 251/255, green: 128/255, blue: 128/255)

    var text = ""

    var body: some View {
        HStack {
            Image(systemName: iconName)
                .foregroundColor(iconColor)
            Text(text)
                .font(.system(.body, design: .rounded))
                .foregroundColor(.secondary)
            Spacer()
        }
    }
}
```

With these two common views created, it's very straightforward to layout the form. We use a `ScrollView`, together with a `VStack` to arrange the form fields. The validation error messages are only displayed when an error is detected:

```
Group {
    if !paymentFormViewModel.isNameValid {
        ValidationErrorText(text: "Please enter the payment name")
    }

    if !paymentFormViewModel.isAmountValid {
        ValidationErrorText(text: "Please enter a valid amount")
    }

    if !paymentFormViewModel.isMemoValid {
        ValidationErrorText(text: "Your memo should not exceed 300 characters")
    }
}
```

The *type* field is a bit different because it's not a text field. The user can either select *income* or *expense*. In this case, we created two buttons

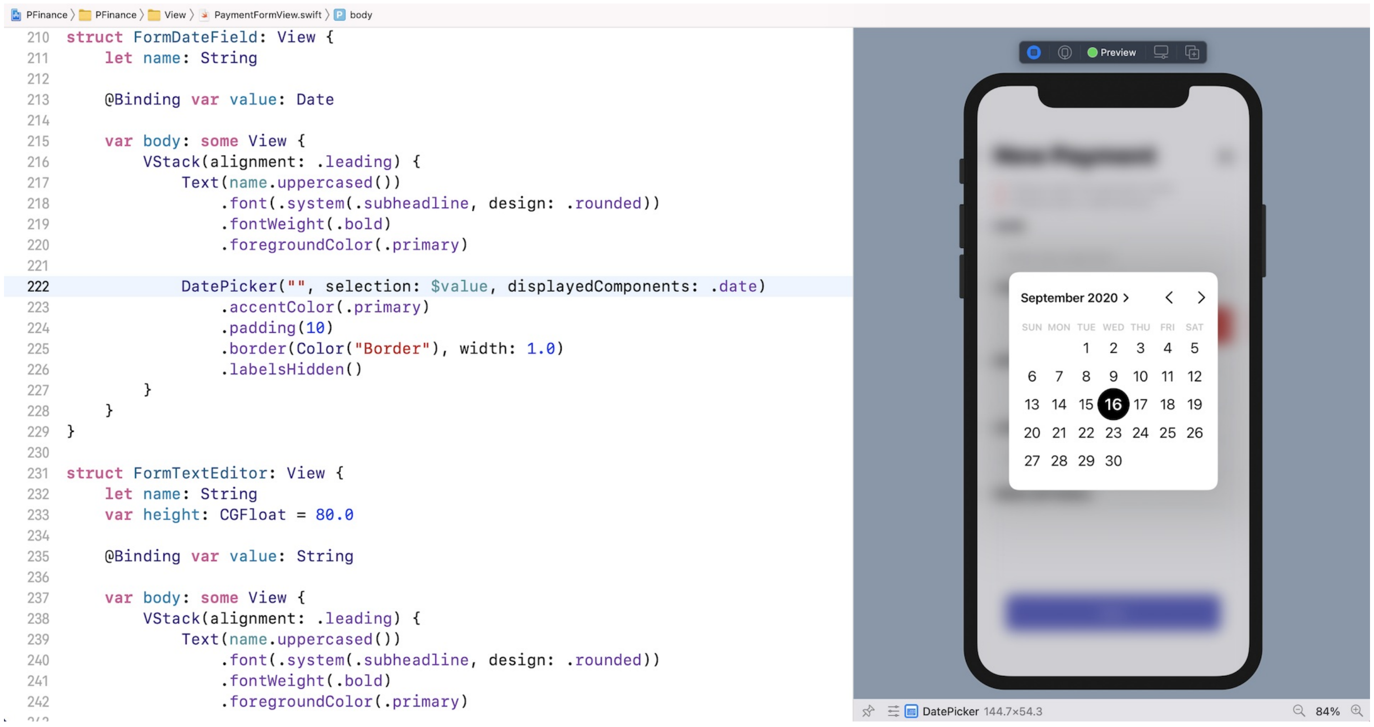
```
VStack(alignment: .leading) {
    Text("TYPE")
        .font(.system(.subheadline, design: .rounded))
        .fontWeight(.bold)
        .foregroundColor(.primary)
        .padding(.vertical, 10)

    HStack(spacing: 0) {
        Button(action: {
            self.paymentFormViewModel.type = .income
        }) {
            Text("Income")
                .font(.headline)
                .foregroundColor(self.paymentFormViewModel.type == .income ? Color
.white : Color.primary)
        }
        .frame(minWidth: 0.0, maxWidth: .infinity)
        .padding()
        .background(self.paymentFormViewModel.type == .income ? Color("IncomeCard"
) : Color.white)

        Button(action: {
            self.paymentFormViewModel.type = .expense
        }) {
            Text("Expense")
                .font(.headline)
                .foregroundColor(self.paymentFormViewModel.type == .expense ? Color
.white : Color.primary)
        }
        .frame(minWidth: 0.0, maxWidth: .infinity)
        .padding()
        .background(self.paymentFormViewModel.type == .expense ? Color("ExpenseCar
d") : Color.white)
    }
    .border(Color("Border"), width: 1.0)
}
```

The background color of the button varies depending on the type of the payment activity.

The date field is implemented using the `DatePicker` component. It's very easy to use the `DatePicker`. All you need is to provide the label, the binding to the date value, and the display components of the date.

```
struct FormDateField: View {
    let name: String

    @Binding var value: Date

    var body: some View {
        VStack(alignment: .leading) {
            Text(name.uppercased())
                .font(.system(.subheadline, design: .rounded))
                .fontWeight(.bold)
                .foregroundColor(.primary)

            DatePicker("", selection: $value, displayedComponents: .date)
                .accentColor(.primary)
                .padding(10)
                .border(Color("Border"), width: 1.0)
                .labelsHidden()
        }
    }
}
```

In iOS 14, the built-in `DatePicker` has been improved with better UI and more styles. If you run the view and tap the date field, the app displays a full calendar view for users to pick the date. The user interface is much much better than the old version of date picker.


PFinance 〉 PFinance 〉 View 〉 PaymentFormView.swift 〉 body

```swift
struct FormDateField: View {
    let name: String

    @Binding var value: Date

    var body: some View {
        VStack(alignment: .leading) {
            Text(name.uppercased())
                .font(.system(.subheadline, design: .rounded))
                .fontWeight(.bold)
                .foregroundColor(.primary)

            DatePicker("", selection: $value, displayedComponents: .date)
                .accentColor(.primary)
                .padding(10)
                .border(Color("Border"), width: 1.0)
                .labelsHidden()
        }
    }
}

struct FormTextEditor: View {
    let name: String
    var height: CGFloat = 80.0

    @Binding var value: String

    var body: some View {
        VStack(alignment: .leading) {
            Text(name.uppercased())
                .font(.system(.subheadline, design: .rounded))
                .fontWeight(.bold)
                .foregroundColor(.primary)
```

Figure 4. Tapping the date field shows you a full calendar

The *memo* field is not a text field but a text editor. In iOS 13, SwiftUI doesn't come with a multiline text field. To support multiline text editing, you will need to tap into the UIKit framework and wrap `UITextView` into a SwiftUI component. Starting with iOS 14, Swift introduced a new component called `TextEditor` for displaying and editing long-form text. In `PaymentFormView.swift`, you should find the following struct:

```
struct FormTextEditor: View {
    let name: String
    var height: CGFloat = 80.0

    @Binding var value: String

    var body: some View {
        VStack(alignment: .leading) {
            Text(name.uppercased())
                .font(.system(.subheadline, design: .rounded))
                .fontWeight(.bold)
                .foregroundColor(.primary)

            TextEditor(text: $value)
                .frame(minHeight: height)
                .font(.headline)
                .foregroundColor(.primary)
                .padding()
                .border(Color("Border"), width: 1.0)
        }
    }
}
```

The usage of `TextEditor` is very similar to `TextField`. All you need is to pass it the binding to a String variable. Just like any other SwiftUI view, you apply view modifiers to style its appearance. This is how we created the *Memo* field for users to type long form text.

At the end of the form, is the *Save* button. This button is disabled by default. It's only enabled when all the required fields are filled. The `disabled` modifier is used to control the button's state.

```
Button(action: {
    self.save()
    self.presentationMode.wrappedValue.dismiss()
}) {
    Text("Save")
        .opacity(paymentFormViewModel.isFormInputValid ? 1.0 : 0.5)
        .font(.headline)
        .foregroundColor(.white)
        .padding()
        .frame(minWidth: 0, maxWidth: .infinity)
        .background(Color("IncomeCard"))
        .cornerRadius(10)

}
.padding()
.disabled(!paymentFormViewModel.isFormInputValid)
```

When the button is tapped, it calls the `save()` method to save the payment activity permanently into the database. And then, it invokes the `dismiss()` method to dismiss the view. If you are not familiar with the environment value `presentationMode`, please read chapter 12.

Form Validation

That's pretty much how we layout the form UI. Let's talk about how the form validation is implemented. Basically, we followed what's discussed in chapter 15 to perform the form validation using Combine. Here is what we have done:

1. Create a view model to represent the payment activity form.
2. Implement form validation in the view model and publish the validation results using Combine.

We created a view model class to hold the values of the form fields. You can switch over to `PaymentFormViewModel.swift` to view the code:

```
class PaymentFormViewModel: ObservableObject {
    // Input
    @Published var name = ""
    @Published var location = ""
    @Published var amount = ""
    @Published var type = PaymentCategory.expense
    @Published var date = Date.today
    @Published var memo = ""

    // Output
    @Published var isNameValid = false
    @Published var isAmountValid = true
    @Published var isMemoValid = true
    @Published var isFormInputValid = false

    private var cancellableSet: Set<AnyCancellable> = []

    init(paymentActivity: PaymentActivity?) {

        self.name = paymentActivity?.name ?? ""
        self.location = paymentActivity?.address ?? ""
        self.amount = "\(paymentActivity?.amount ?? 0.0)"
        self.memo = paymentActivity?.memo ?? ""
        self.type = paymentActivity?.type ?? .expense
        self.date = paymentActivity?.date ?? Date.today

        $name
            .receive(on: RunLoop.main)
            .map { name in
                return name.count > 0
            }
            .assign(to: \.isNameValid, on: self)
            .store(in: &cancellableSet)

        $amount
            .receive(on: RunLoop.main)
            .map { amount in
                guard let validAmount = Double(amount) else {
                    return false
                }
                return validAmount > 0
            }
            .assign(to: \.isAmountValid, on: self)
```

```
                .store(in: &cancellableSet)

            $memo
                .receive(on: RunLoop.main)
                .map { memo in
                    return memo.count < 300
                }
                .assign(to: \.isMemoValid, on: self)
                .store(in: &cancellableSet)

            Publishers.CombineLatest3($isNameValid, $isAmountValid, $isMemoValid)
                .receive(on: RunLoop.main)
                .map { (isNameValid, isAmountValid, isMemoValid) in
                    return isNameValid && isAmountValid && isMemoValid
                }
                .assign(to: \.isFormInputValid, on: self)
                .store(in: &cancellableSet)
        }

    }
```

This class conforms to `ObservableObject`. All the properties are annotated with `@Published` because we want to notify the subscribers whenever there is a value change and perform the validation accordingly.

Whenever there are any changes to the form's input values, this view model will execute the validation code, update the results, and notify the subscribers.

So, who is the subscriber?

If you go back to `PaymentFormView.swift`, you should notice that we have declared a variable named `paymentFormViewModel` with the `@ObservedObject` wrapper:

```
@ObservedObject private var paymentFormViewModel: PaymentFormViewModel
```

The `PaymentFormView` subscribes to the changes of the view model. When any of the validation variables (e.g. `isNameValid`) are updated, `PaymentFormView` will be notified and the view itself will refresh to display the validation error on screen.

```
if !paymentFormViewModel.isNameValid {
    ValidationErrorText(text: "Please enter the payment name")
}
```

Form Initialization

Do you notice the initialization method? It accepts a `PaymentActivity` object and initializes the view model.

```
var payment: PaymentActivity?

init(payment: PaymentActivity? = nil) {
    self.payment = payment
    self.paymentFormViewModel = PaymentFormViewModel(paymentActivity: payment)
}
```

The `PaymentFormView` allows the user to create a new payment activity and edit an existing activity. This is why the init method takes in an optional payment activity object. If the object is `nil`, we display an empty form. Otherwise, we fill the form fields with the given values of the `PaymentActivity` object.

Previewing the form

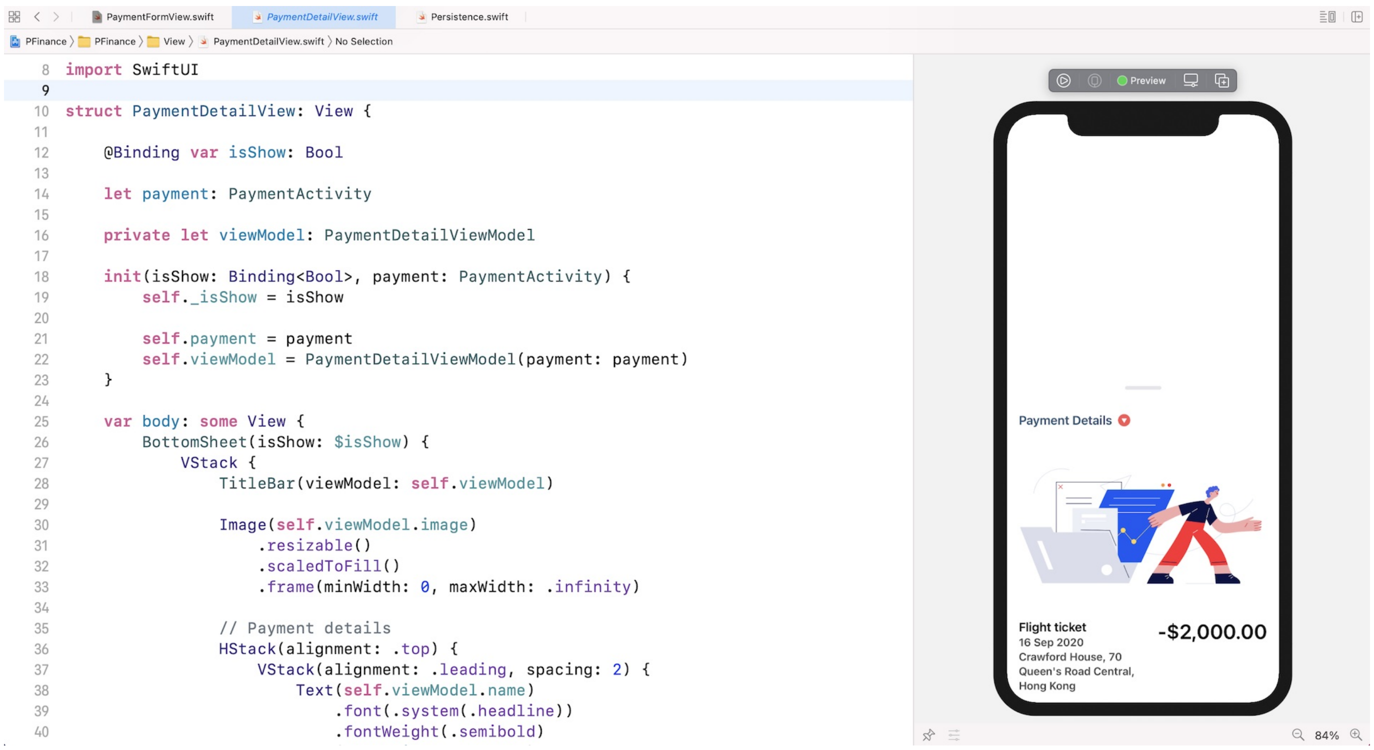
The instant preview feature is one of the many things I really enjoy when programming with SwiftUI. However, if your SwiftUI project integrates with Core Data, it will require some extra code to make the preview work. Here is the code for the preview:

```
struct PaymentFormView_Previews: PreviewProvider {
    static var previews: some View {

        let context = PersistenceController.shared.container.viewContext

        let testTrans = PaymentActivity(context: context)
        testTrans.paymentId = UUID()
        testTrans.name = ""
        testTrans.amount = 0.0
        testTrans.date = .today
        testTrans.type = .expense

        return Group {
            PaymentFormView(payment: testTrans)
            PaymentFormView(payment: testTrans)
                .preferredColorScheme(.dark)

            FormTextField(name: "NAME", placeHolder: "Enter your payment", value:
.constant("")).previewLayout(.sizeThatFits)

            ValidationErrorText(text: "Please enter the payment name").previewLayo
ut(.sizeThatFits)

        }
    }
}
```

To instantiate the `PaymentFormView`, you have to provide a `PaymentActivity` object. Since `PaymentActivity` is a managed object, we need to retrieve the context from `PersistenceController` to create one. Once we create the `PaymentActivity` object, we can initialize it with a test item and use it to preview the `PaymentFormView`. This is how you can preview a view that integrates with Core Data.

Implementing the Payment Activity Detail View

Now let's move onto the next view and discuss how the payment activity detail view is implemented. This view is activated when a user selects one of the payment activities in the *Recent Transactions* section. It displays the details of the activity such as amount and location. You can open `PaymentDetailView.swift` to see what the UI looks like. This will give you a better idea of the detail view.



```swift
import SwiftUI

struct PaymentDetailView: View {

    @Binding var isShow: Bool

    let payment: PaymentActivity

    private let viewModel: PaymentDetailViewModel

    init(isShow: Binding<Bool>, payment: PaymentActivity) {
        self._isShow = isShow

        self.payment = payment
        self.viewModel = PaymentDetailViewModel(payment: payment)
    }

    var body: some View {
        BottomSheet(isShow: $isShow) {
            VStack {
                TitleBar(viewModel: self.viewModel)

                Image(self.viewModel.image)
                    .resizable()
                    .scaledToFill()
                    .frame(minWidth: 0, maxWidth: .infinity)

                // Payment details
                HStack(alignment: .top) {
                    VStack(alignment: .leading, spacing: 2) {
                        Text(self.viewModel.name)
                            .font(.system(.headline))
                            .fontWeight(.semibold)
```

Figure 5. The payment activity detail view

The User Interface

The detail view is quite simple. I believe you know how to layout the components, so I will not explain the code line by line. One thing I want to highlight is the following lines of code:

```
@Binding var isShow: Bool

...

init(isShow: Binding<Bool>, payment: PaymentActivity) {
    self._isShow = isShow

    self.payment = payment
    self.viewModel = PaymentDetailViewModel(payment: payment)
}
```

This detail view is triggered when `isShow` is set to `true`. Since we need to perform some initialization to create the view model, we implement a custom `init` method.

The View Model

> Instead of putting everything in a single view, we can separate a view into two components: the view and its view model. The view itself is responsible for the UI layout, while the view model holds the state and data to be displayed in the view. The view model also handles the data validation and conversion. For experienced developers, we are applying a well known design pattern called MVVM (short for Model-View-ViewModel).
>
> - See Building a Registration Form with Combine and View Model (Chapter 15)

To separate the actual view data from the view UI, we have created a view model named `PaymentDetailViewModel`:

```
private let viewModel: PaymentDetailViewModel
```

Why do we need to create an extra view model to hold the view's data? Take a look at the icon right next to the title *Payment Details*. This is a dynamic icon that changes in reference to the payment type. Additionally, do you notice the format of the amount? A requirement for our app is to format the amount with only two decimal places. We can implement all this logic in the view, but if you keep adding all the logic in the view, the view will become too complex to maintain.

One well known computer programming principle is the single responsibility principle (SRP). It states that every class or module in a program should have responsibility for just a single piece of that program's functionality. SRP is one of the keys to writing good code, making your code easier to maintain and read.

This is why we separate the view into two components:

1. `PaymentDetailView` is only responsible for the UI layout.
2. `PaymentDetailViewModel` is responsible for converting the view's data into the expected presentation format.

Open `PaymentDetailViewModel` and take a look:

```
struct PaymentDetailViewModel {

    var payment: PaymentActivity

    var name: String {
        return payment.name
    }

    var date: String {
        return payment.date.string()
    }

    var address: String {
        return payment.address ?? ""
    }

    var typeIcon: String {

        let icon: String

        switch payment.type {
        case .income: icon = "arrowtriangle.up.circle.fill"
        case .expense: icon = "arrowtriangle.down.circle.fill"
        }

        return icon
    }

    var image: String = "payment-detail"

    var amount: String {
        let formatter = NumberFormatter()
        formatter.numberStyle = .decimal
        formatter.minimumFractionDigits = 2

        let formattedValue = formatter.string(from: NSNumber(value: payment.amount
)) ?? ""

        let formattedAmount = ((payment.type == .income) ? "+" : "-") + "$" + form
attedValue

        return formattedAmount
    }

    var memo: String {
        return payment.memo ?? ""
    }

    init(payment: PaymentActivity) {
        self.payment = payment
    }

}
```

As you can see, we implement all the conversion logic in this view model. Can we put this logic back into the view? Yes, of course. However, I believe the code is much cleaner when breaking the view into two parts.

The Bottom Sheet

The payment activity detail view is displayed as an overlay using a `BottomSheet`. When a user taps a payment record, the app brings up the bottom sheet and displays the payment details. This bottom sheet is expandable, so the user can drag the detail view up to expand it. Alternatively, the user can drag the view down to dismiss it.

```
var body: some View {
    BottomSheet(isShow: $isShow) {
        VStack {
            TitleBar(viewModel: self.viewModel)


    ...
}
```

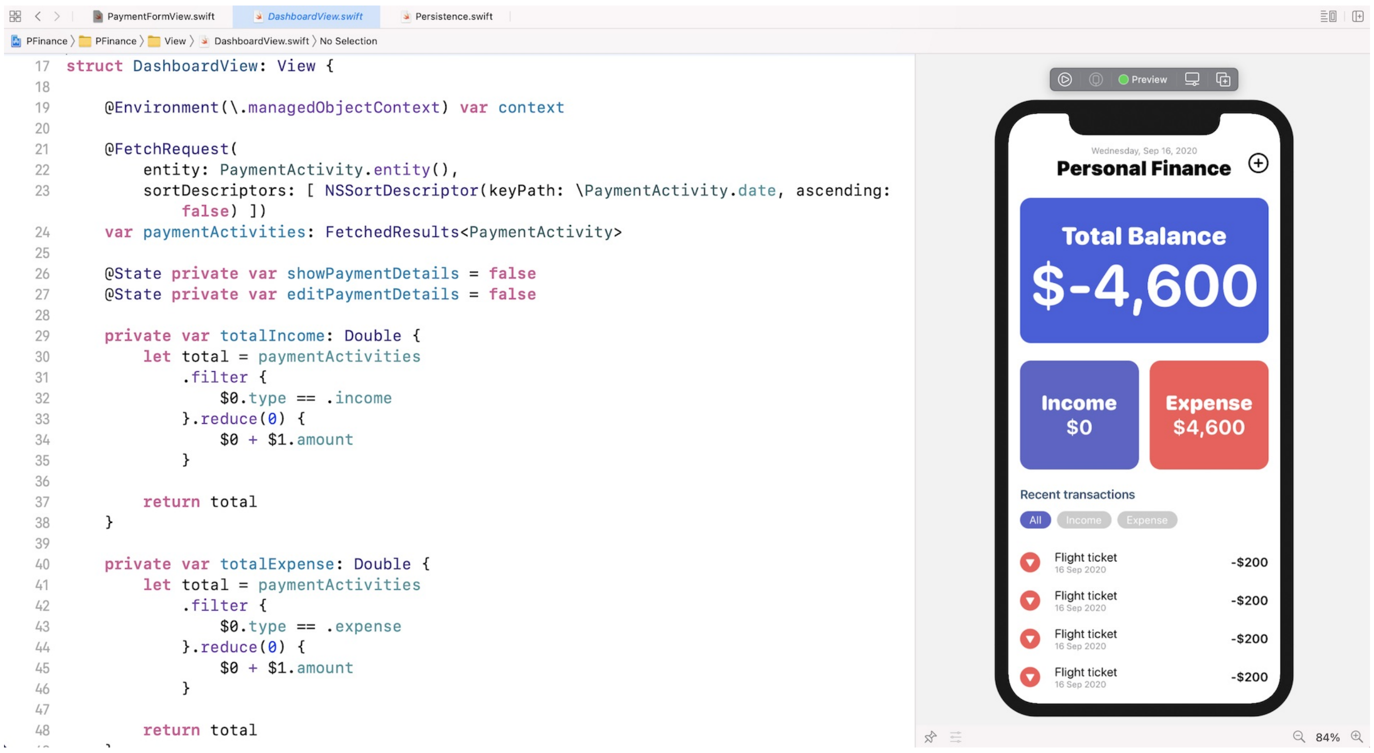
We have implemented a similar bottom sheet in chapter 18. Therefore, we reuse most of the code as discussed in that chapter. If you want to learn more about how the `BottomSheet` is built, you can re-read the chapter. However, there is one thing I want to highlight. In chapter 18, the bottom sheet is specifically designed for displaying the restaurant details. For this app, we converted it to a generic bottom sheet which can display any content.

The trick is to take in a `Content` view as a parameter. If you compare the code with the code used in chapter 18, it is exactly the same, except that the content inside the scroll view can be varied.

```
struct BottomSheet<Content>: View where Content: View  {

    .
    .
    .

    let content: () -> Content

    var body: some View {
        GeometryReader { geometry in
            VStack {
                Spacer()

                HandleBar()

                ScrollView(.vertical) {

                    .
                    .
                    .

                    self.content()
                }

               .
               .
               .
            }

     .
     .
     .
}
```

Walking Through the Dashboard View

Now it's time to walk you through the dashboard view. Among all the views in the personal finance app, this view is the most complicated one.



```
struct DashboardView: View {

    @Environment(\.managedObjectContext) var context

    @FetchRequest(
        entity: PaymentActivity.entity(),
        sortDescriptors: [ NSSortDescriptor(keyPath: \PaymentActivity.date, ascending:
            false) ])
    var paymentActivities: FetchedResults<PaymentActivity>

    @State private var showPaymentDetails = false
    @State private var editPaymentDetails = false

    private var totalIncome: Double {
        let total = paymentActivities
            .filter {
                $0.type == .income
            }.reduce(0) {
                $0 + $1.amount
            }

        return total
    }

    private var totalExpense: Double {
        let total = paymentActivities
            .filter {
                $0.type == .expense
            }.reduce(0) {
                $0 + $1.amount
            }

        return total
```

Figure 6. The dashboard view

The Menu Bar

Open `Dashboard.swift` and let's start with the menu bar:

```
struct MenuBar<Content>: View where Content: View {
    @State private var showPaymentForm = false

    let modalContent: () -> Content

    var body: some View {
        ZStack(alignment: .trailing) {
            HStack(alignment: .center) {
                Spacer()

                VStack(alignment: .center) {
                    Text(Date.today.string(with: "EEEE, MMM d, yyyy"))
                        .font(.caption)
                        .foregroundColor(.gray)
                    Text("Personal Finance")
                        .font(.title)
                        .fontWeight(.black)
                }

                Spacer()
            }

            Button(action: {
                self.showPaymentForm = true
            }) {
                Image(systemName: "plus.circle")
                    .font(.title)
                    .foregroundColor(.primary)
            }

            .sheet(isPresented: self.$showPaymentForm, onDismiss: {
                self.showPaymentForm = false
            }) {
                self.modalContent()
            }
        }

    }
}
```

The layout of the menu bar is simple. It shows the app's title, today's date, and the plus button. This menu bar view is designed to take in any modal view (i.e. `modalContent`). When the plus button is tapped, the modal view will be displayed. If you don't know how to create a generic view in SwiftUI, you can refer to chapter 17 on building a generic draggable view.

Income, Expense and Total Balance

Next, we have three card views to show the total balance, income, and expenses. Here is the code for the income card view:

```swift
struct IncomeCard: View {
    var income = 0.0

    var body: some View {

        ZStack {
            Rectangle()
                .foregroundColor(Color("IncomeCard"))
                .cornerRadius(15.0)

            VStack {
                Text("Income")
                    .font(.system(.title, design: .rounded))
                    .fontWeight(.black)
                    .foregroundColor(.white)
                Text(NumberFormatter.currency(from: income))
                    .font(.system(.title, design: .rounded))
                    .fontWeight(.bold)
                    .foregroundColor(.white)
                    .minimumScaleFactor(0.1)
            }
        }
        .frame(height: 150)

    }
}
```

We simply use a `ZStack` to overlay the text on a colored rectangle. We use a similar technique to layout both `TotalBalanceCard` and `ExpenseCard`. So, how do we compute the income, expense, and total balance? We have three computed properties declared at the beginning of `DashboardView`:

```
private var totalIncome: Double {
    let total = paymentActivities
        .filter {
            $0.type == .income
        }.reduce(0) {
            $0 + $1.amount
        }

    return total
}

private var totalExpense: Double {
    let total = paymentActivities
        .filter {
            $0.type == .expense
        }.reduce(0) {
            $0 + $1.amount
        }

    return total
}

private var totalBalance: Double {
    return totalIncome - totalExpense
}
```

The `paymentActivities` variable stores the collection of payment activities. So, to calculate the total income, we first use the `filter` function to filter those activities with type `.income` and then use the `reduce` function to compute the total amount. The same technique was applied to calculate the total expense. Higher order functions in Swift are very useful. If you don't know how to use filter and reduce, you can further check out this tutorial (https://www.appcoda.com/higher-order-functions-swift/).

Recent Transactions

The last part of the UI is the list of recent transactions. As all the rows share the same layout (except the icon of the payment type), we create a generic view for the transaction row like this:

```
struct TransactionCellView: View {

    @ObservedObject var transaction: PaymentActivity

    var body: some View {

        HStack(spacing: 20) {

            if transaction.isFault {
                EmptyView()

            }  else {

                Image(systemName: transaction.type == .income ? "arrowtriangle.up.
circle.fill" : "arrowtriangle.down.circle.fill")
                    .font(.title)
                    .foregroundColor(Color(transaction.type == .income ? "IncomeCa
rd" : "ExpenseCard"))

                VStack(alignment: .leading) {
                    Text(transaction.name)
                        .font(.system(.body, design: .rounded))
                    Text(transaction.date.string())
                        .font(.system(.caption, design: .rounded))
                        .foregroundColor(.gray)
                }

                Spacer()

                Text((transaction.type == .income ? "+" : "-") + NumberFormatter.c
urrency(from: transaction.amount))
                    .font(.system(.headline, design: .rounded))
            }
        }
        .padding(.vertical, 5)

    }
}
```

This cell view takes in a `PaymentActivity` object which is a managed object and then presents its content. To ensure the given managed object (i.e. `transaction`) is valid, we place a check inside the `HStack` by accessing the `isFault` property.

To list the transaction, we use `ForEach` to loop through the payment activities and create a `TransactionCellView` for each activity:

```
ForEach(paymentDataForView) { transaction in
    TransactionCellView(transaction: transaction)
        .onTapGesture {
            self.showPaymentDetails = true
            self.selectedPaymentActivity = transaction
        }
        .contextMenu {
            Button(action: {
                // Edit payment details
                self.editPaymentDetails = true
                self.selectedPaymentActivity = transaction

            }) {
                HStack {
                    Text("Edit")
                    Image(systemName: "pencil")
                }
            }

            Button(action: {
                // Delete the selected payment
                self.delete(payment: transaction)
            }) {
                HStack {
                    Text("Delete")
                    Image(systemName: "trash")
                }
            }
        }
}
.sheet(isPresented: self.$editPaymentDetails) {
    PaymentFormView(payment: self.selectedPaymentActivity).environment(\.managedOb
jectContext, self.context)
}
```

When a user taps and holds a row, it displays a context menu with both the delete and edit options. When selecting the edit option, the app will create the `PaymentFormView` with the selected payment activity. For the delete operation, the app will completely remove the activity from the database using Core Data.

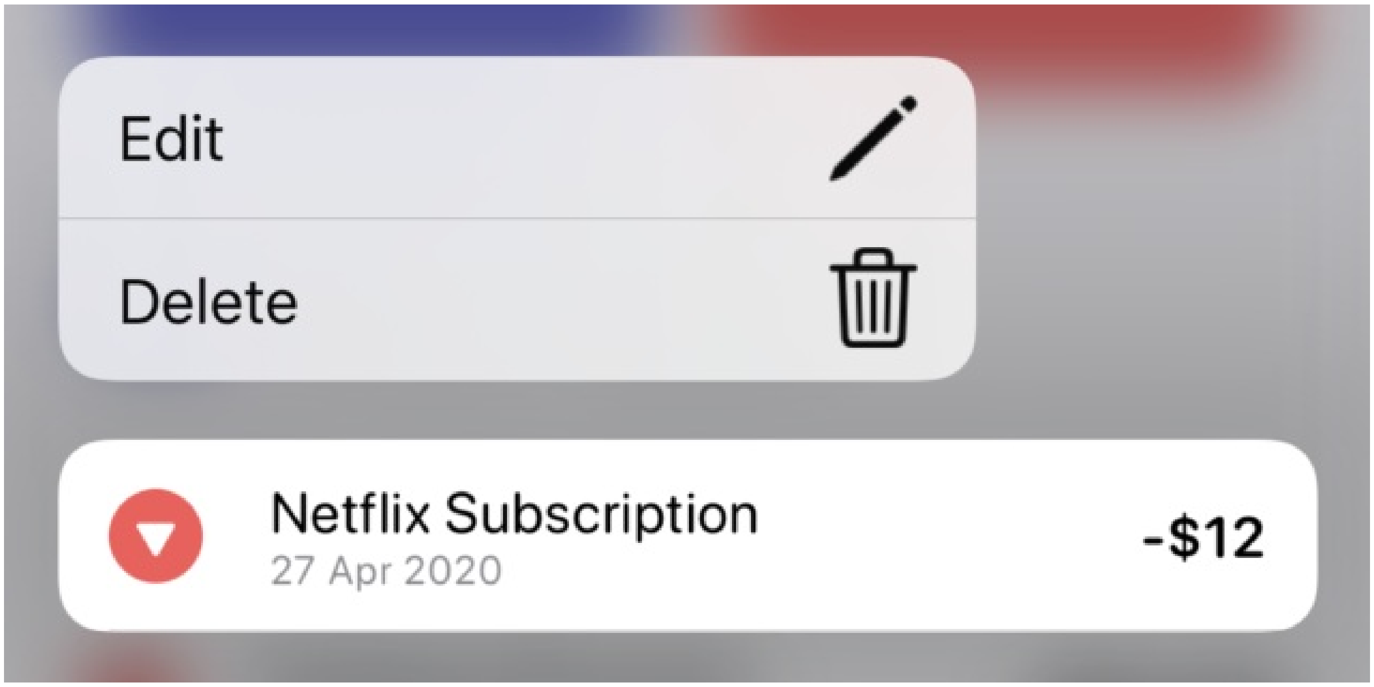


Figure 7. The context menu for the payment activity row

Do you notice the `paymentDataForView` variable? Instead of using `paymentActivities`, the list view presents items stored in `paymentDataForView`. Why is that?

In the *Recent Transactions* section, the app provides three options for the user to filter the payment activities including all, income, and expense. For example, if the *expense* option is selected, the app only shows those activities related to expenses.

```
private var paymentDataForView: [PaymentActivity] {

    switch listType {
    case .all:
        return paymentActivities
            .sorted(by: { $0.date.compare($1.date) == .orderedDescending })
    case .income:
        return paymentActivities
            .filter { $0.type == .income }
            .sorted(by: { $0.date.compare($1.date) == .orderedDescending })
    case .expense:
        return paymentActivities
            .filter { $0.type == .expense }
            .sorted(by: { $0.date.compare($1.date) == .orderedDescending })
    }
}
```

The `paymentDataForView` is another computed property which returns a collection of payment activities that match the list type. In the code, we use the `filter` function to filter the payment activities and call the `sort` function to sort the activities in reverse chronological order.

Managing Payment Activities with Core Data

As mentioned before, all the payment activities are saved in the local database and managed using Core Data. In the code, we use the `@FetchRequest` property wrapper to fetch the payment activities like this:

```
@FetchRequest(
    entity: PaymentActivity.entity(),
    sortDescriptors: [ NSSortDescriptor(keyPath: \PaymentActivity.date, ascending:
false) ])
var paymentActivities: FetchedResults<PaymentActivity>
```

This property wrapper makes it very easy to perform a fetch request. We simply specify the entity, which is the `PaymentActivity`, and the sort descriptor describing how the data should be ordered. The Core Data framework will then use the environment's managed object context to fetch the data. Most importantly, SwiftUI will automatically update the list views or any other views that are bound to the fetched results.

Deleting an activity from the database is also very straightforward. We call the `delete` function of the context and pass it with the activity object to remove:

```
private func delete(payment: PaymentActivity) {
    self.context.delete(payment)

    do {
        try self.context.save()
    } catch {
        print("Failed to save the context: \(error.localizedDescription)")
    }
}
```

FAdding a new activity or updating an existing activity happens in the `PaymentFormView`. If you look at the `PaymentFormView.swift` file again, you will find the `save()` function:

```swift
private func save() {
    let newPayment = payment ?? PaymentActivity(context: context)

    newPayment.paymentId = UUID()
    newPayment.name = paymentFormViewModel.name
    newPayment.type = paymentFormViewModel.type
    newPayment.date = paymentFormViewModel.date
    newPayment.amount = Double(paymentFormViewModel.amount)!
    newPayment.address = paymentFormViewModel.location
    newPayment.memo = paymentFormViewModel.memo

    do {
        try context.save()
    } catch {
        print("Failed to save the record...")
        print(error.localizedDescription)
    }
}
```

The first line of the code checks if we have any existing activity. If not, we will instantiate a new one. We then assign the form values to the payment object and call the `save` function of the managed object context to add/update the record in the database.

Exploring the Extensions

For convenience purposes, we have built two extensions for formatting the date and number. In the project navigator, you should find two files under the *Extension* folder. Let's take a look at the `Date+Ext.swift` file first:

```
extension Date {
    static var today: Date {
        return Date()
    }

    static var yesterday: Date {
        return Calendar.current.date(byAdding: .day, value: -1, to: Date())!
    }

    static var tomorrow: Date {
        return Calendar.current.date(byAdding: .day, value: 1, to: Date())!
    }

    var month: Int {
        return Calendar.current.component(.month, from: self)
    }

    static func fromString(string: String, with format: String = "yyyy-MM-dd") ->
Date? {
        let dateFormatter = DateFormatter()
        dateFormatter.dateFormat = format
        return dateFormatter.date(from: string)
    }

    func string(with format: String = "dd MMM yyyy") -> String {
        let dateFormatter = DateFormatter()
        dateFormatter.dateFormat = format
        return dateFormatter.string(from: self)
    }
}
```

In the code above, we extend `Date` to provide additional functionality including:

- Get today's date
- Get tomorrow's date
- Get yesterday's date
- Get the month of the date
- Convert the current date to a string or vice versa

For formatting the amount, we extend `NumberFormatter` to provide an additional function:

```
extension NumberFormatter {
    static func currency(from value: Double) -> String {
        let formatter = NumberFormatter()
        formatter.numberStyle = .decimal

        let formattedValue = formatter.string(from: NSNumber(value: value)) ?? ""

        return "$" + formattedValue
    }
}
```

This function takes in a value, converts it to a string and prepends it with the dollar sign ($).

Handling the Software Keyboard

In the `PaymentFormView.swift` file, we added the following modifier:

```
.keyboardAdaptive()
```

This is a custom view modifier, developed for handling the software keyboard. For iOS 14, this modifier is no longer required but I intentionally added it because you may need it if your app supports iOS 13.

On iOS 13, the software keyboard blocks parts of the form when it's brought up without applying the modifier. For example, if you try to tap the memo field, it's completely hidden behind the keyboard. Conversely, if you attach the modifier to the scroll view, the form will move up automatically when the keyboard appears. On iOS 14, the mobile operating system itself automatically handles the appearance of the software keyboard, preventing it from blocking the input field.

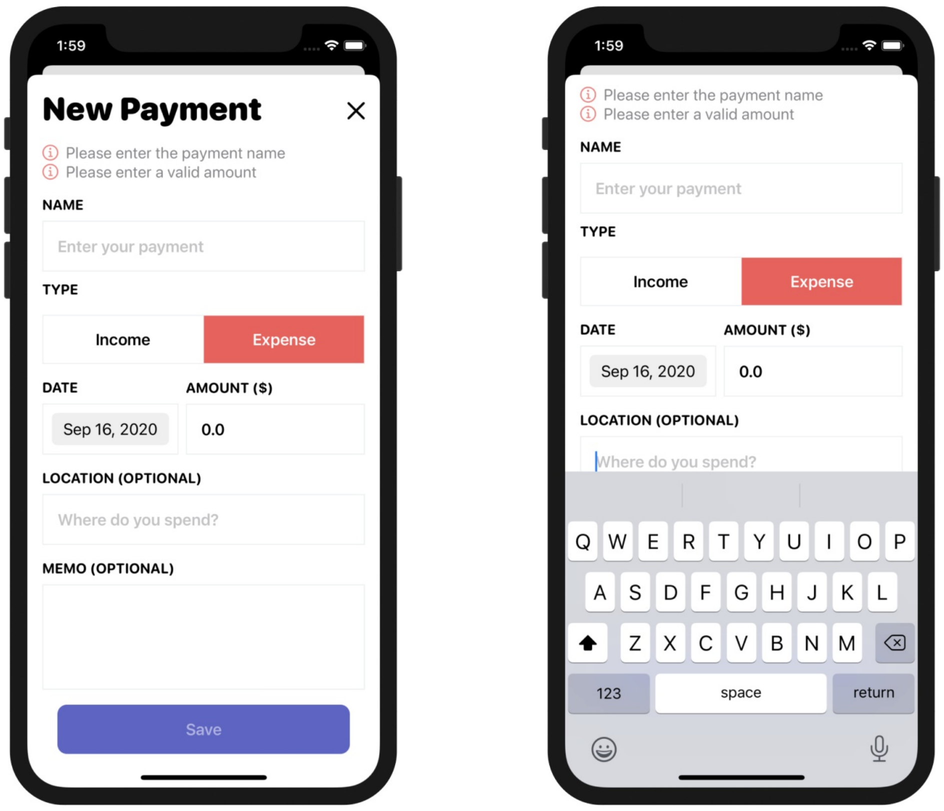


Figure 8. Without using keyboardAdaptive (left), Using keyboardAdaptive

Now let's check out the code (`KeyboardAdaptive.swift`) and see how we handle keyboard events:

```
struct KeyboardAdaptive: ViewModifier {

    @State var currentHeight: CGFloat = 0

    func body(content: Content) -> some View {
        content
            .padding(.bottom, currentHeight)
            .onAppear(perform: handleKeyboardEvents)
    }

    private func handleKeyboardEvents() {

        NotificationCenter.default.publisher(for: UIResponder.keyboardWillShowNoti
fication
        ).compactMap { (notification) in
            notification.userInfo?["UIKeyboardFrameEndUserInfoKey"] as? CGRect
        }.map { rect in
            rect.height
        }.subscribe(Subscribers.Assign(object: self, keyPath: \.currentHeight))

        NotificationCenter.default.publisher(for: UIResponder.keyboardWillHideNoti
fication
        ).compactMap { _ in
            CGFloat.zero
        }.subscribe(Subscribers.Assign(object: self, keyPath: \.currentHeight))

    }
}

extension View {
    func keyboardAdaptive() -> some View {
        ModifiedContent(content: self, modifier: KeyboardAdaptive())
    }
}
```

Whenever the keyboard appears (or disappears), iOS sends a notification to the app:

- *keyboardWillShowNotification* - this notification is sent when the keyboard is about to appear
- *keyboardWillHideNotification* - this notification is sent when the keyboard is going

to disappear

So, how do we make use of these notifications to scroll up the form? When the app receives the *keyboardWillShowNotification*, it adds padding to the form to move it up. Conversely, we set the padding to zero when the *keyboardWillHideNotification* is received.

In the code above, we have a state variable to store the height of the keyboard. By using the Combine framework, we have a publisher that captures the *keyboardWillShowNotification* and emits the current height of the keyboard. Additionally, we have another publisher which listens to the *keyboardWillHideNotification* and emits a value of zero. For both publishers, we use the built-in `assign` subscriber to assign the value emitted by these publishers to the `currentHeight` variable.

This is how we detect the keyboard appearance, capture its height, and add the bottom padding. But why do we need to have the `View` extension?

The code works without the extension. You write the code like this to detect the keyboard events:

```
.modifier(KeyboardAdaptive())
```

To make the code cleaner, we create the extension and add the `keyboardAdaptive()` function. After that, we can attach the modifier to any view like this:

```
.keyboardAdaptive()
```

Since this view modifier is only applicable to iOS 13, we use the `#available` check to verify the OS version in the `keyboardAdaptive()` function:

```
extension View {
    func keyboardAdaptive() -> some View {
        if #available(iOS 14.0, *) {
            return AnyView(self)
        } else {
            return AnyView(ModifiedContent(content: self, modifier: KeyboardAdapti
ve()))
        }
    }
}
```

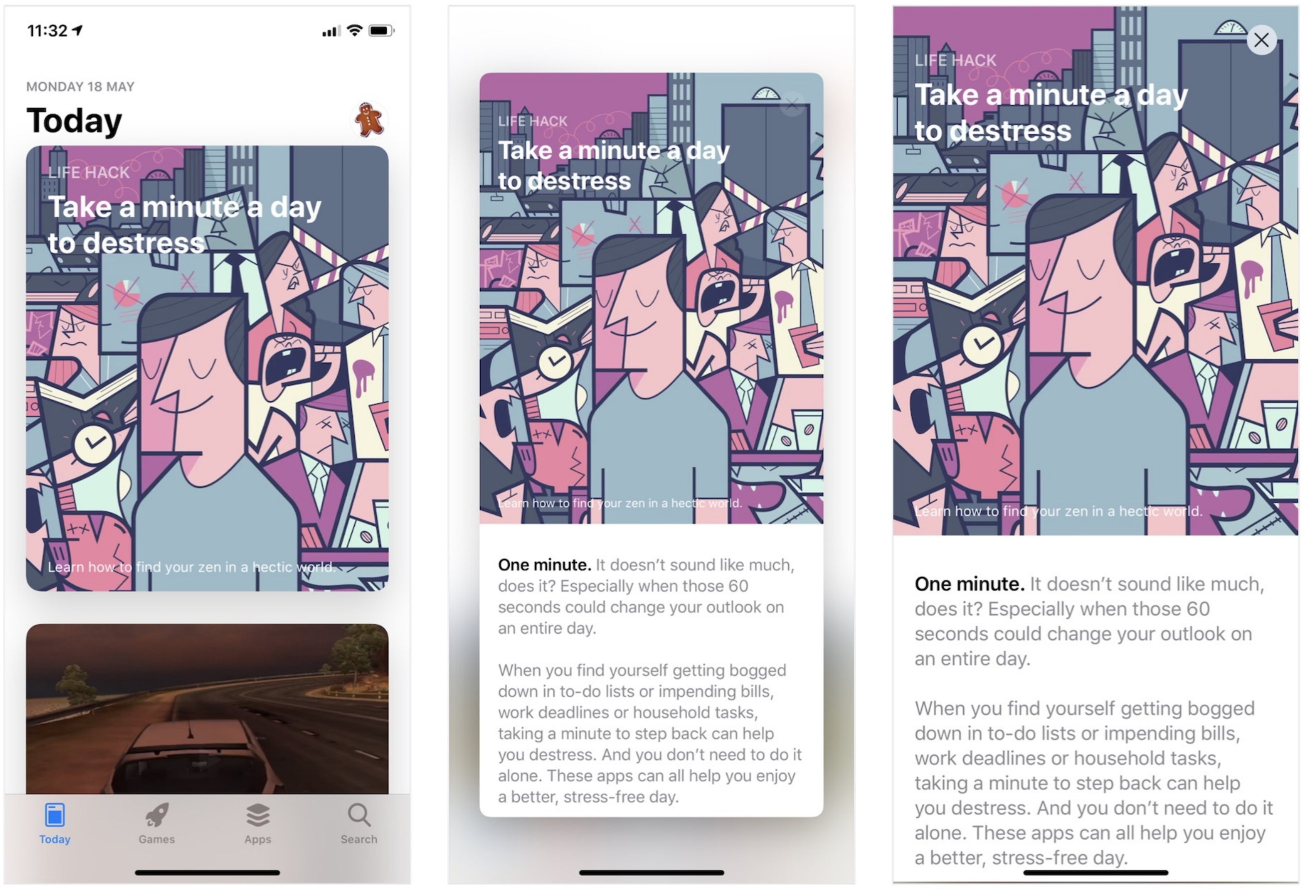
Summary

This is how we built the personal finance app from the ground up. Most of the techniques we used shouldn't be new to you. You combine what you learned in the earlier chapters to build the app.

SwiftUI is a very powerful and promising framework, allowing you to build the same app with less code than UIKit. If you have some programming experience with UIKit, you know it would take you more time and lines of code to create the personal finance app. I really hope you enjoy learning SwiftUI and building UIs with this new framework. However, one thing I have to point out again is that SwiftUI only works on iOS 13 (or up). If you need to support older versions of iOS, you will need to use UIKit.

Chapter 26
Creating an App Store like Animated View Transition

You probably have used Apple's the built-in *App Store* app. In the *Today* section, it presents users with a headline, various articles and app recommendations. What interests me and many of you is the animated view transition. As you can see in figure 1, the articles are listed in a card like format. When you tap it, the card pops out to reveal the full content. To dismiss the article view and return to the list view, you simply tap the close button . If you don't understand what I mean, the best way to understand these views is to open the *App Store* app on your iPhone to try it out.


11:32
MONDAY 18 MAY
Today
LIFE HACK
Take a minute a day to destress
Learn how to find your zen in a hectic world.
Today
Games
Apps
Search

One minute. It doesn't sound like much, does it? Especially when those 60 seconds could change your outlook on an entire day.

When you find yourself getting bogged down in to-do lists or impending bills, work deadlines or household tasks, taking a minute to step back can help you destress. And you don't need to do it alone. These apps can all help you enjoy a better, stress-free day.

Figure 1. Apple's App Store app

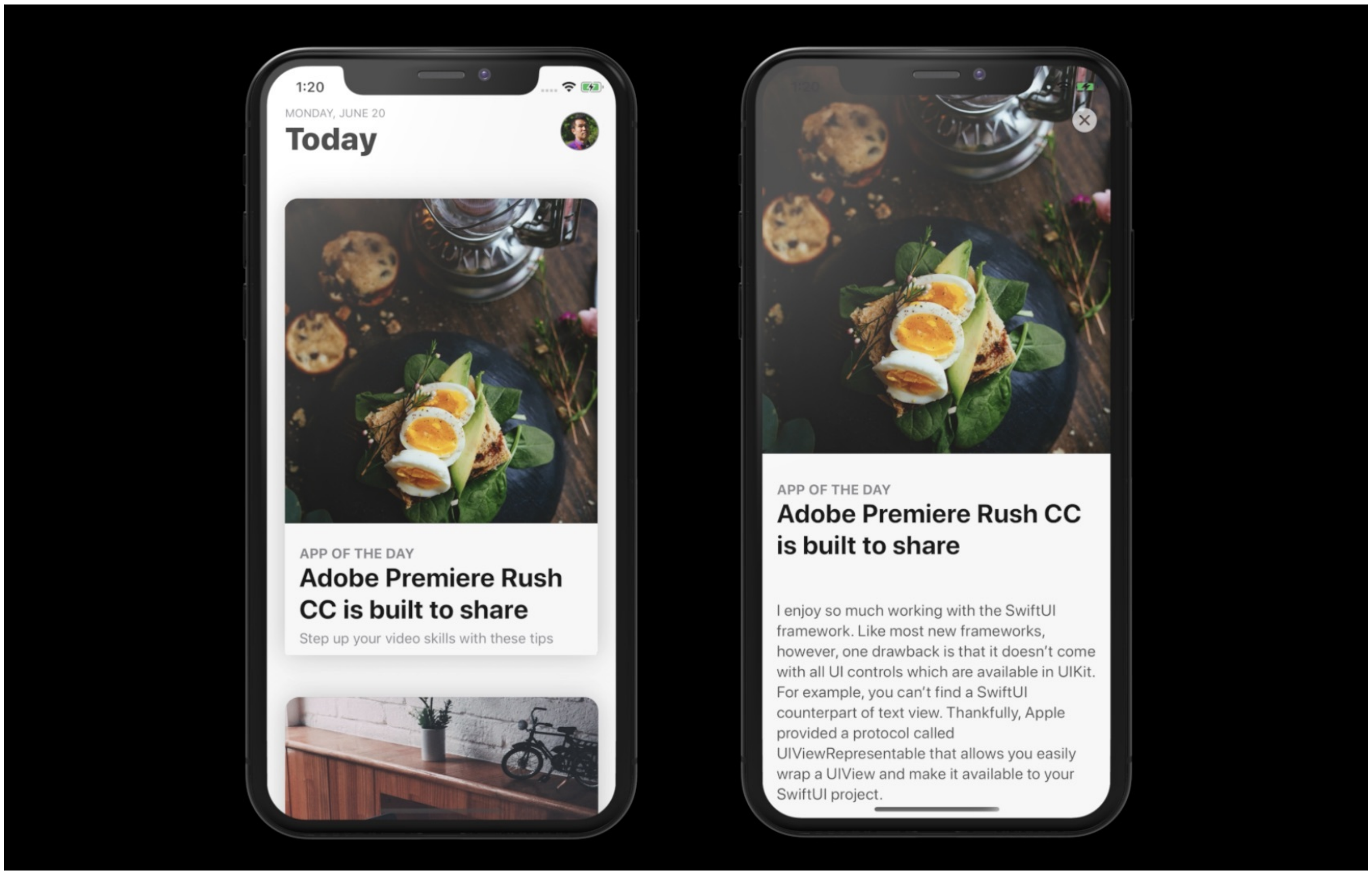
In this chapter, we will build a similar list view and implement the animated transition using SwiftUI. In particular, you will learn the following techniques:

- How to use GeometryReader to detect screen sizes
- How to create a variable-sized card view
- How to implement an App Store like animated view transition

Let's get started.

Introducing the Demo App

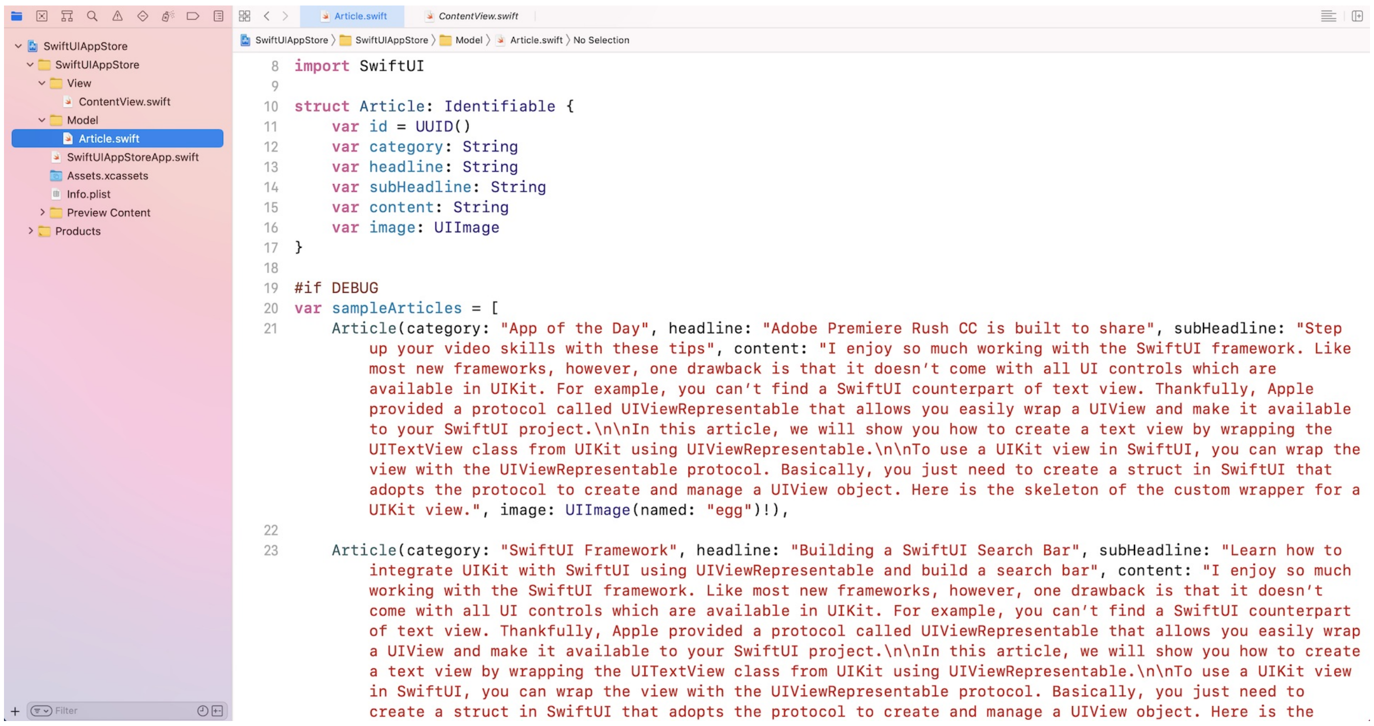
As usual, we will build a demo app together. The app looks very similar to the *App Store* app but without the tab bar. It only has a list view showing all the articles in card format. When a user taps any of the articles, the card expands to full screen and displays the article details. To return to the list view, the user can either tap the close button or drag down the article view to collapse it.


1:20
MONDAY, JUNE 20
Today
APP OF THE DAY
Adobe Premiere Rush CC is built to share
Step up your video skills with these tips
APP OF THE DAY
Adobe Premiere Rush CC is built to share

I enjoy so much working with the SwiftUI framework. Like most new frameworks, however, one drawback is that it doesn't come with all UI controls which are available in UIKit. For example, you can't find a SwiftUI counterpart of text view. Thankfully, Apple provided a protocol called UIViewRepresentable that allows you easily wrap a UIView and make it available to your SwiftUI project.

Figure 2. The demo app

We will build the app from scratch. But to save you time from typing some of the code, I have prepared a starter project for you. You can download it from https://www.appcoda.com/resources/swiftui2/SwiftUIAppStoreStarter.zip. After downloading the project, unzip it and open `SwiftUIAppStore.xcodeproj` to take a look.



```swift
import SwiftUI

struct Article: Identifiable {
    var id = UUID()
    var category: String
    var headline: String
    var subHeadline: String
    var content: String
    var image: UIImage
}

#if DEBUG
var sampleArticles = [
    Article(category: "App of the Day", headline: "Adobe Premiere Rush CC is built to share", subHeadline: "Step
        up your video skills with these tips", content: "I enjoy so much working with the SwiftUI framework. Like
        most new frameworks, however, one drawback is that it doesn't come with all UI controls which are
        available in UIKit. For example, you can't find a SwiftUI counterpart of text view. Thankfully, Apple
        provided a protocol called UIViewRepresentable that allows you easily wrap a UIView and make it available
        to your SwiftUI project.\n\nIn this article, we will show you how to create a text view by wrapping the
        UITextView class from UIKit using UIViewRepresentable.\n\nTo use a UIKit view in SwiftUI, you can wrap the
        view with the UIViewRepresentable protocol. Basically, you just need to create a struct in SwiftUI that
        adopts the protocol to create and manage a UIView object. Here is the skeleton of the custom wrapper for a
        UIKit view.", image: UIImage(named: "egg")!),

    Article(category: "SwiftUI Framework", headline: "Building a SwiftUI Search Bar", subHeadline: "Learn how to
        integrate UIKit with SwiftUI using UIViewRepresentable and build a search bar", content: "I enjoy so much
        working with the SwiftUI framework. Like most new frameworks, however, one drawback is that it doesn't
        come with all UI controls which are available in UIKit. For example, you can't find a SwiftUI counterpart
        of text view. Thankfully, Apple provided a protocol called UIViewRepresentable that allows you easily wrap
        a UIView and make it available to your SwiftUI project.\n\nIn this article, we will show you how to create
        a text view by wrapping the UITextView class from UIKit using UIViewRepresentable.\n\nTo use a UIKit view
        in SwiftUI, you can wrap the view with the UIViewRepresentable protocol. Basically, you just need to
        create a struct in SwiftUI that adopts the protocol to create and manage a UIView object. Here is the
```

Figure 3. The starter project

The starter projects comes with the following implementation:

1. It already bundles the required images in the asset catalog.
2. The `ContentView.swift` file is the default SwiftUI view generated by Xcode.
3. The `Article.swift` file contains the `Article` struct, which represents an article in the app. For testing purposes, this file also creates the `sampleArticles` array which includes some test data. You may modify its content if you want to change the article data.

Understanding the Card View

You've learned how to create a card-like UI before. This card view is very similar to that implemented in chapter 5, but it will be more flexible to support scrollable content. In other words, it has two modes: *excerpt* and *full content*. In the excerpt mode, it only displays the image, category, headline and sub-headline of the article. As its name suggests, the full content will display the article details as shown in figure 2.

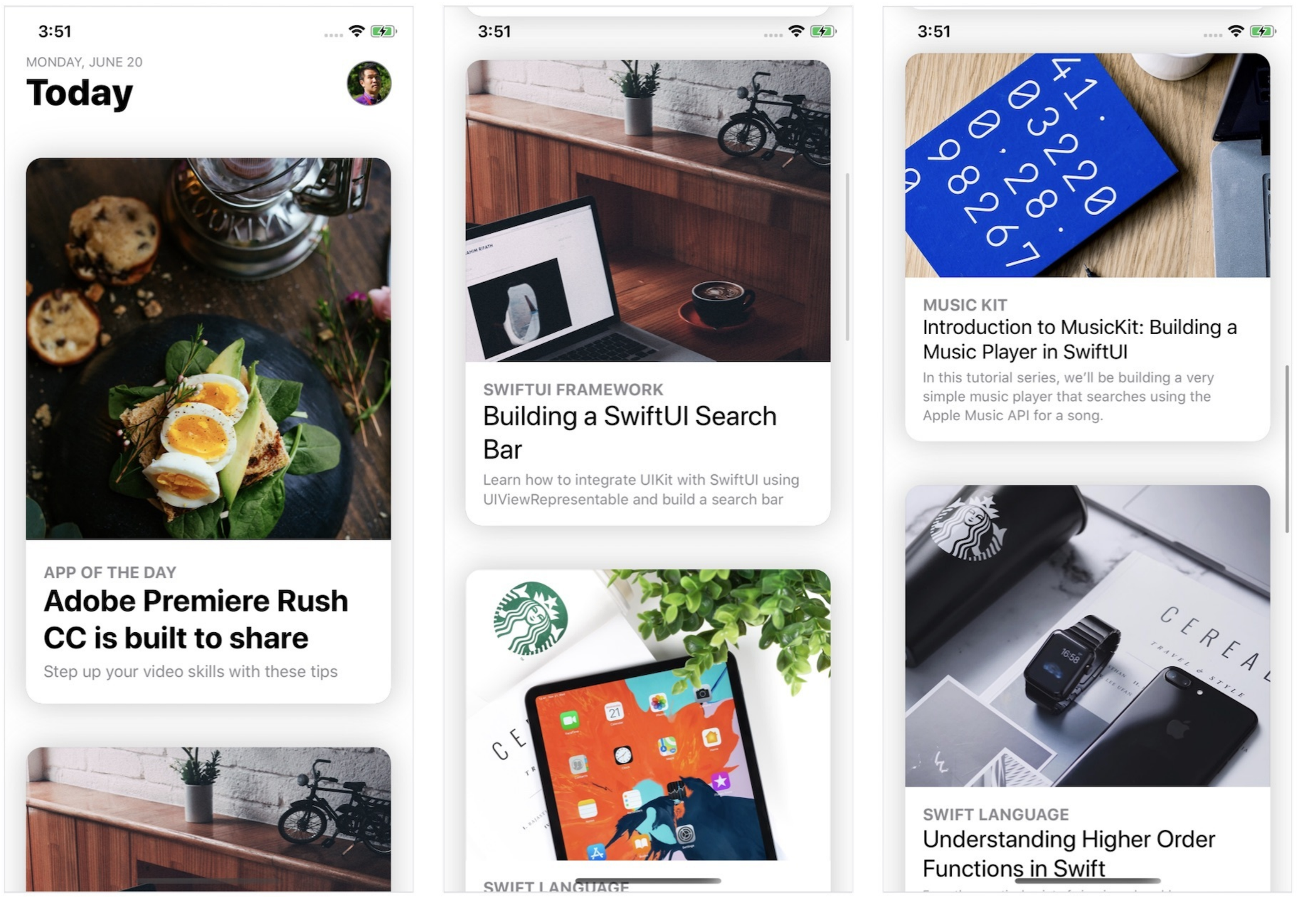

3:51
MONDAY, JUNE 20
Today
APP OF THE DAY
Adobe Premiere Rush CC is built to share
Step up your video skills with these tips
3:51
SWIFTUI FRAMEWORK
Building a SwiftUI Search Bar
Learn how to integrate UIKit with SwiftUI using UIViewRepresentable and build a search bar
SWIFT LANGUAGE
3:51
MUSIC KIT
Introduction to MusicKit: Building a Music Player in SwiftUI
In this tutorial series, we'll be building a very simple music player that searches using the Apple Music API for a song.
SWIFT LANGUAGE
Understanding Higher Order Functions in Swift

Figure 4. The sample card views

If you look a bit closer into the card views shown in figure 4, you will find that the size of card views varies according to the height of the image. However, the height of the card will not exceed 500 points.

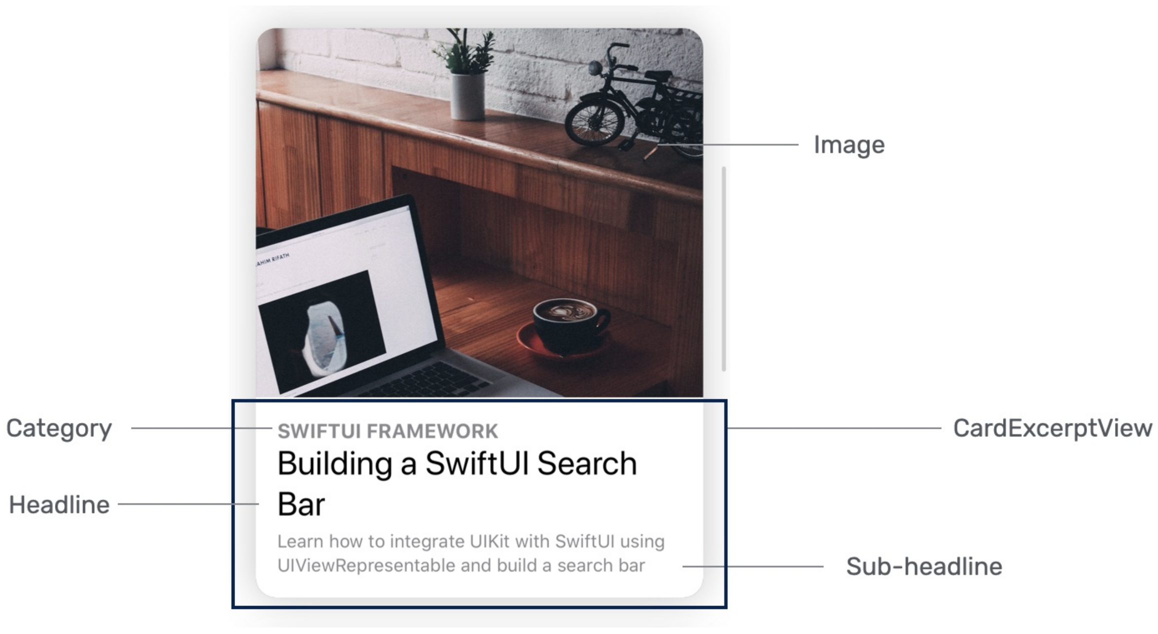
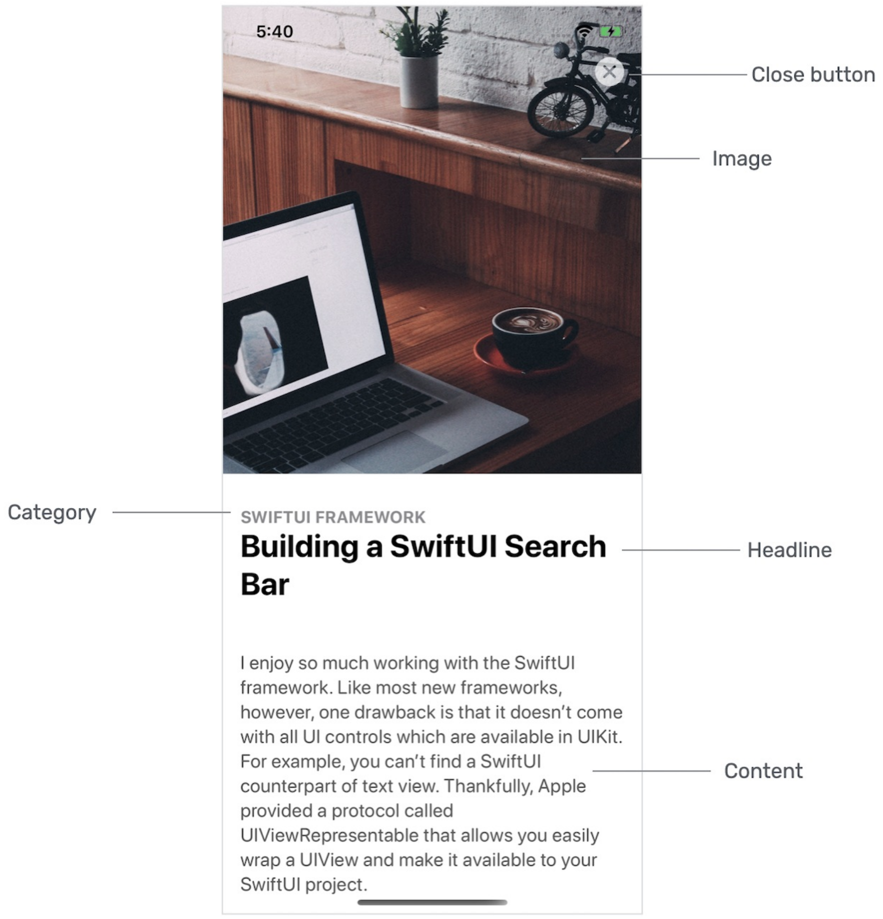


Image
Category
SWIFTUI FRAMEWORK
Building a SwiftUI Search Bar
Headline
CardExcerptView
Learn how to integrate UIKit with SwiftUI using UIViewRepresentable and build a search bar
Sub-headline

Figure 5. The components of a card view in excerpt mode

Let's also look at how the card view looks in full content mode. As you can see in the figure below, the card view expands to a full screen that displays the content. Other than that, the image is a little bit bigger and the sub-headline is hidden. Furthermore, the close button appears on screen for users to dismiss the view. Please also take note that this is a scrollable view.



I enjoy so much working with the SwiftUI framework. Like most new frameworks, however, one drawback is that it doesn't come with all UI controls which are available in UIKit. For example, you can't find a SwiftUI counterpart of text view. Thankfully, Apple provided a protocol called UIViewRepresentable that allows you easily wrap a UIView and make it available to your SwiftUI project.

Figure 6. The components of a card view in full content mode

Implementing the Card View

Now that you understand the requirements of this card view, let's see how to create it. We will use a separate file for implementing the card view. In the project navigator, right click the *View* folder and choose *New file...*. Select the *SwiftUI View* template and name the file `ArticleCardView.swift` .

First, let's begin with the excerpt view, which is the view overlayed on top of the image (see figure 5). Insert the following code in the file:

```
struct ArticleExcerptView: View {

    let category: String
    let headline: String
    let subHeadline: String

    @Binding var isShowContent: Bool

    var body: some View {
        VStack(alignment: .leading) {
            Spacer()

            Rectangle()
                .frame(minHeight: 100, maxHeight: 150)
                .overlay(
                    HStack {
                        VStack(alignment: .leading) {
                            Text(self.category.uppercased())
                                .font(.subheadline)
                                .fontWeight(.bold)
                                .foregroundColor(.secondary)

                            Text(self.headline)
                                .font(.title)
                                .fontWeight(.bold)
                                .foregroundColor(.primary)
                                .minimumScaleFactor(0.1)
                                .lineLimit(2)
                                .padding(.bottom, 5)

                            if !self.isShowContent {
                                Text(self.subHeadline)
                                    .font(.subheadline)
                                    .foregroundColor(.secondary)
                                    .minimumScaleFactor(0.1)
                                    .lineLimit(3)

                            }
                        }
                        .padding()

                        Spacer()
                    }
                )
        }
```

```
            .foregroundColor(.white)

        }
    }
```

The `ArticleExcerptView` should be flexible to support different content. Therefore, we define the variables above. As previously mentioned, the card view should be able to switch between *excerpt* and *full content* mode. This binding variable is declared for controlling the display of the content. When its value is set to false, it's in excerpt mode. Conversely, it's in full content mode when true. The sub-headline is displayed only when the value of `isShowContent` is set to `true`.

There are various ways to layout the excerpt view. In the code above, we create a `Rectangle` view and overlay it with the headline and sub-headline. You should be familiar with most of the modifiers attached to the `Text` view. But the `minimumScaleFactor` modifier is worth a mention. By applying the modifier, the system automatically shrinks the font size of the text to fit the available space. For example, if the headline contains too much text, iOS will scale it down to 10% of its original size before it truncates.

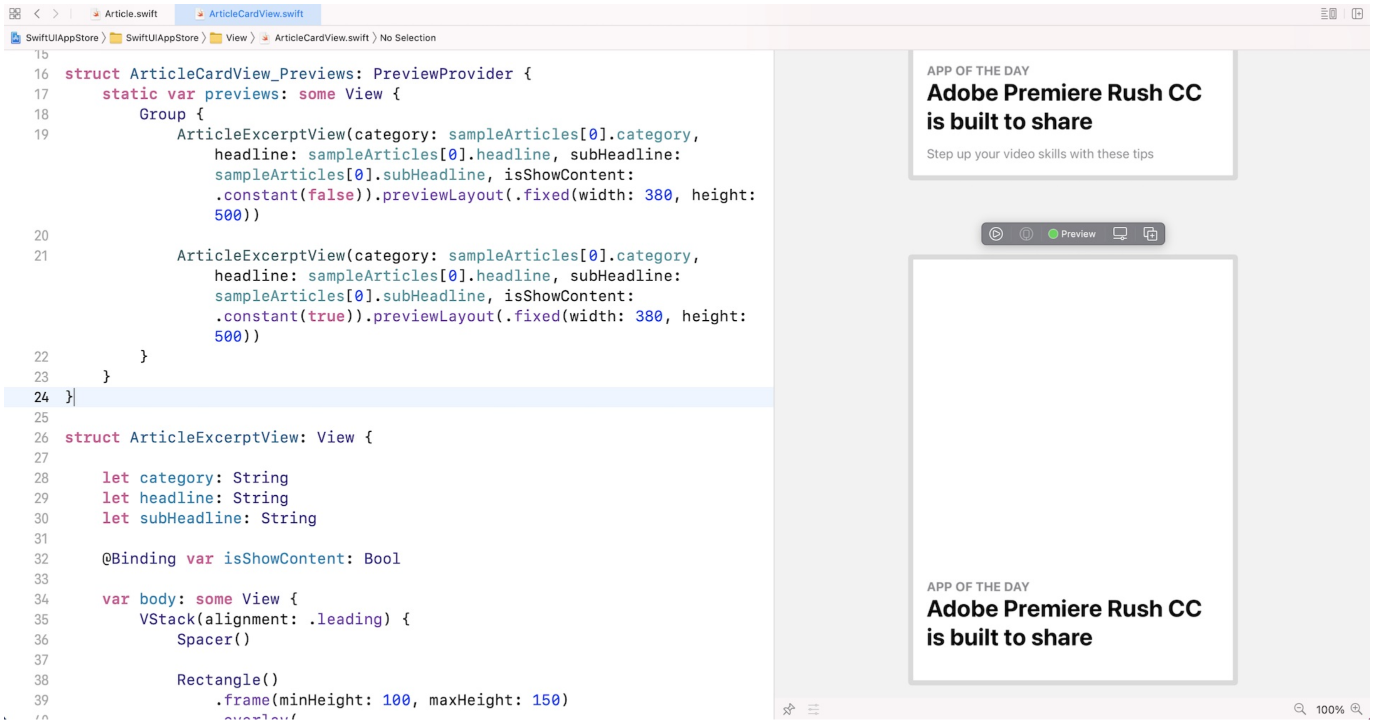
Previewing the UI

To preview the excerpt view, you can modify the preview code like this:

```
struct ArticleCardView_Previews: PreviewProvider {
    static var previews: some View {
        Group {
            ArticleExcerptView(category: sampleArticles[0].category, headline: sam
pleArticles[0].headline, subHeadline: sampleArticles[0].subHeadline, isShowContent
: .constant(false)).previewLayout(.fixed(width: 380, height: 500))

            ArticleExcerptView(category: sampleArticles[0].category, headline: sam
pleArticles[0].headline, subHeadline: sampleArticles[0].subHeadline, isShowContent
: .constant(true)).previewLayout(.fixed(width: 380, height: 500))
        }
    }
}
```

Here, we instantiate two excerpt views such that one has the `isShowContent` binding set to `false` and the other one set to `true`. The `sampleArticles` array is the test data which comes with the starter project.

Instead of previewing using a device, we preview the UI in a fixed size rectangle. If everything works perfectly, you should see the excerpt view in the preview canvas.


Article.swift
ArticleCardView.swift
SwiftUIAppStore 〉 SwiftUIAppStore 〉 View 〉 ArticleCardView.swift 〉 No Selection
APP OF THE DAY
Adobe Premiere Rush CC is built to share
Step up your video skills with these tips
Preview
APP OF THE DAY
Adobe Premiere Rush CC is built to share
100%

```
struct ArticleCardView_Previews: PreviewProvider {
    static var previews: some View {
        Group {
            ArticleExcerptView(category: sampleArticles[0].category,
                headline: sampleArticles[0].headline, subHeadline:
                sampleArticles[0].subHeadline, isShowContent:
                .constant(false)).previewLayout(.fixed(width: 380, height:
                500))

            ArticleExcerptView(category: sampleArticles[0].category,
                headline: sampleArticles[0].headline, subHeadline:
                sampleArticles[0].subHeadline, isShowContent:
                .constant(true)).previewLayout(.fixed(width: 380, height:
                500))
        }
    }
}

struct ArticleExcerptView: View {

    let category: String
    let headline: String
    let subHeadline: String

    @Binding var isShowContent: Bool

    var body: some View {
        VStack(alignment: .leading) {
            Spacer()

            Rectangle()
                .frame(minHeight: 100, maxHeight: 150)
```

Figure 7. Previewing the excerpt view

With the excerpt view ready, let's implement the article card view. Update the `ArticleCardView` struct like this:

```
struct ArticleCardView: View {

    let category: String
    let headline: String
    let subHeadline: String
    let image: UIImage
    var content: String = ""
    @Binding var isShowContent: Bool

    var body: some View {
        ScrollView {
            VStack(alignment: .leading) {
                Image(uiImage: self.image)
                    .resizable()
                    .scaledToFill()
                    .frame(height: min(self.image.size.height/3, 500))
                    .border(Color(.sRGB, red: 150/255, green: 150/255, blue: 150/2
55, opacity: 0.1), width: self.isShowContent ? 0 : 1)
                    .cornerRadius(15)
                    .overlay(
                        ArticleExcerptView(category: self.category, headline: self
.headline, subHeadline: self.subHeadline, isShowContent: self.$isShowContent)
                            .cornerRadius(self.isShowContent ? 0 : 15)
                    )

                // Content
                if self.isShowContent {
                    Text(self.content)
                        .foregroundColor(Color(.darkGray))
                        .font(.system(.body, design: .rounded))
                        .padding(.horizontal)
                        .padding(.bottom, 50)
                        .transition(.move(edge: .top))
                        .animation(.linear)
                }
            }
        }
        .shadow(color: Color(.sRGB, red: 64/255, green: 64/255, blue: 64/255, opac
ity: 0.3), radius: self.isShowContent ? 0 : 15)
    }
}
```

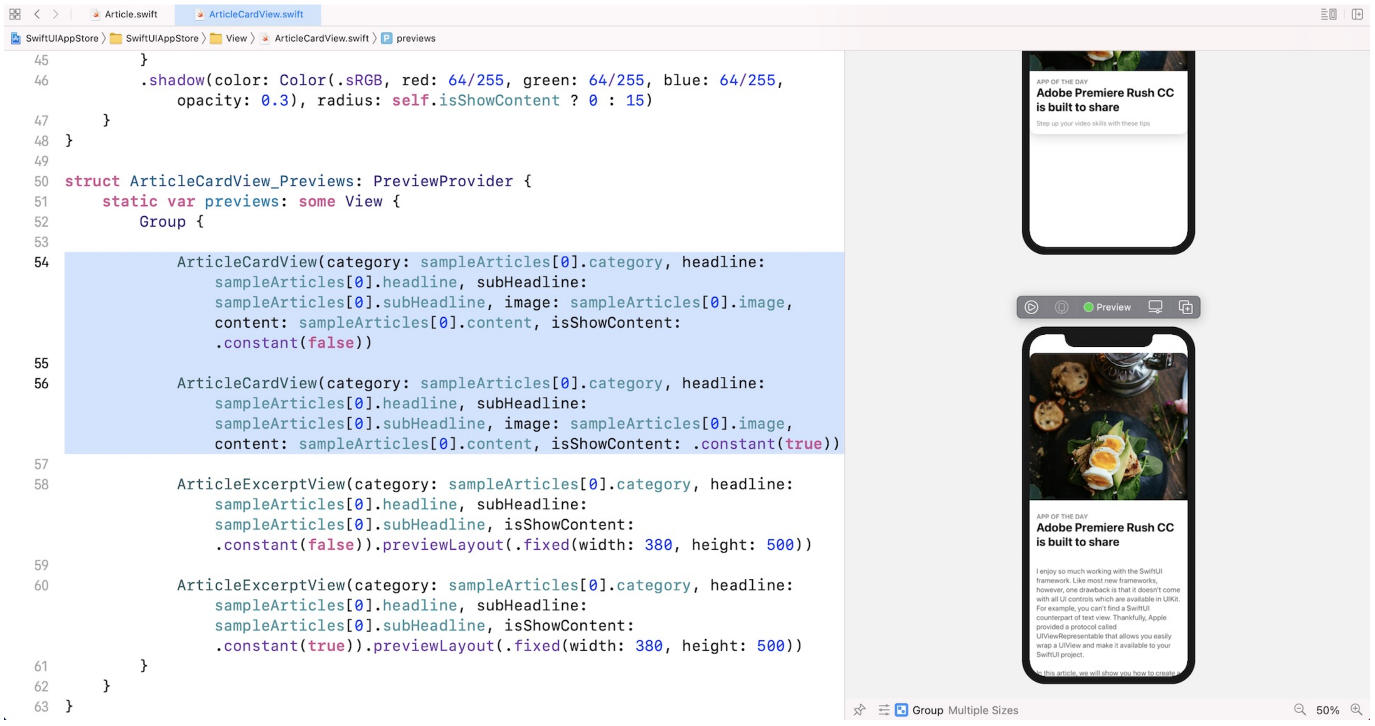
To arrange the layout of the card view, we overlay the `ArticleExcerptView` on top of an `Image` view. The image view is set to `.scaledToFill` with the height not exceeding 500 points. The `content` is only displayed when the `isShowContent` binding is set to `true`.

In order to make the view scrollable, we embed the `VStack` in a vertical scroll view. The `shadow` modifier is used to add a shadow to the card view.

To preview the article card view, you can insert the following code within the `Group` section of `ArticleCardView_Previews` :

```
ArticleCardView(category: sampleArticles[0].category, headline: sampleArticles[0].
headline, subHeadline: sampleArticles[0].subHeadline, image: sampleArticles[0].ima
ge, content: sampleArticles[0].content, isShowContent: .constant(false))

ArticleCardView(category: sampleArticles[0].category, headline: sampleArticles[0].
headline, subHeadline: sampleArticles[0].subHeadline, image: sampleArticles[0].ima
ge, content: sampleArticles[0].content, isShowContent: .constant(true))
```

Once you have made the changes, you should be able to see the card UI in the preview canvas. Additionally, you should see two simulators such that one displays the excerpt view and the other displays the full content.



```
        }
        .shadow(color: Color(.sRGB, red: 64/255, green: 64/255, blue: 64/255,
            opacity: 0.3), radius: self.isShowContent ? 0 : 15)
    }
}

struct ArticleCardView_Previews: PreviewProvider {
    static var previews: some View {
        Group {

            ArticleCardView(category: sampleArticles[0].category, headline:
                sampleArticles[0].headline, subHeadline:
                sampleArticles[0].subHeadline, image: sampleArticles[0].image,
                content: sampleArticles[0].content, isShowContent:
                .constant(false))

            ArticleCardView(category: sampleArticles[0].category, headline:
                sampleArticles[0].headline, subHeadline:
                sampleArticles[0].subHeadline, image: sampleArticles[0].image,
                content: sampleArticles[0].content, isShowContent: .constant(true))

            ArticleExcerptView(category: sampleArticles[0].category, headline:
                sampleArticles[0].headline, subHeadline:
                sampleArticles[0].subHeadline, isShowContent:
                .constant(false)).previewLayout(.fixed(width: 380, height: 500))

            ArticleExcerptView(category: sampleArticles[0].category, headline:
                sampleArticles[0].headline, subHeadline:
                sampleArticles[0].subHeadline, isShowContent:
                .constant(true)).previewLayout(.fixed(width: 380, height: 500))
        }
    }
}
```

APP OF THE DAY
Adobe Premiere Rush CC is built to share
Step up your video skills with these tips

APP OF THE DAY
Adobe Premiere Rush CC is built to share

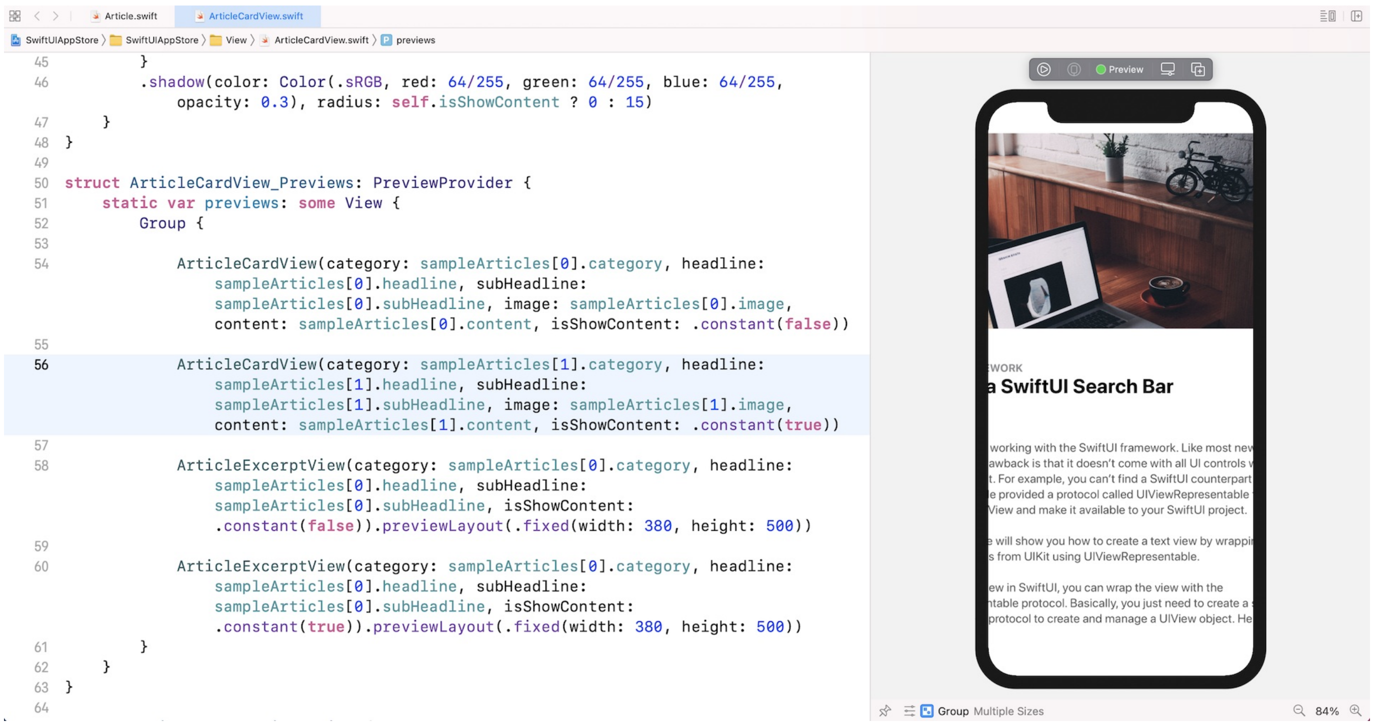
I enjoy so much working with the SwiftUI framework. Like most new frameworks, however, one drawback is that it doesn't come with all UI controls which are available in UIKit. For example, you can't find a SwiftUI counterpart of text view. Thankfully, Apple provided a protocol called UIViewRepresentable that allows you easily wrap a UIView and make it available to your SwiftUI project.

In this article, we will show you how to create a

Figure 8. Previewing the article card view

Using GeometryReader

It seems everything works great. But if you try to preview the card view with another sample article (say, `sampleArticles[1]`), the UI doesn't look good. Both the featured image and the content go beyond the screen edge.



Article.swift
ArticleCardView.swift
SwiftUIAppStore 〉 SwiftUIAppStore 〉 View 〉 ArticleCardView.swift 〉 previews
Preview
Group Multiple Sizes
84%

```
        }
        .shadow(color: Color(.sRGB, red: 64/255, green: 64/255, blue: 64/255,
            opacity: 0.3), radius: self.isShowContent ? 0 : 15)
    }
}

struct ArticleCardView_Previews: PreviewProvider {
    static var previews: some View {
        Group {

            ArticleCardView(category: sampleArticles[0].category, headline:
                sampleArticles[0].headline, subHeadline:
                sampleArticles[0].subHeadline, image: sampleArticles[0].image,
                content: sampleArticles[0].content, isShowContent: .constant(false))

            ArticleCardView(category: sampleArticles[1].category, headline:
                sampleArticles[1].headline, subHeadline:
                sampleArticles[1].subHeadline, image: sampleArticles[1].image,
                content: sampleArticles[1].content, isShowContent: .constant(true))

            ArticleExcerptView(category: sampleArticles[0].category, headline:
                sampleArticles[0].headline, subHeadline:
                sampleArticles[0].subHeadline, isShowContent:
                .constant(false)).previewLayout(.fixed(width: 380, height: 500))

            ArticleExcerptView(category: sampleArticles[0].category, headline:
                sampleArticles[0].headline, subHeadline:
                sampleArticles[0].subHeadline, isShowContent:
                .constant(true)).previewLayout(.fixed(width: 380, height: 500))
        }
    }
}
```

EWORK
a SwiftUI Search Bar

working with the SwiftUI framework. Like most new
awback is that it doesn't come with all UI controls w
t. For example, you can't find a SwiftUI counterpart
le provided a protocol called UIViewRepresentable
View and make it available to your SwiftUI project.

e will show you how to create a text view by wrappi
s from UIKit using UIViewRepresentable.

ew in SwiftUI, you can wrap the view with the
ntable protocol. Basically, you just need to create a
protocol to create and manage a UIView object. He

Figure 9. The card view doesn't fit the content

Let's look at our code again. For the `Image` view, we only limited the height of the image, we don't have any limits on its width:

```
.frame(height: min(self.image.size.height/3, 500))
```

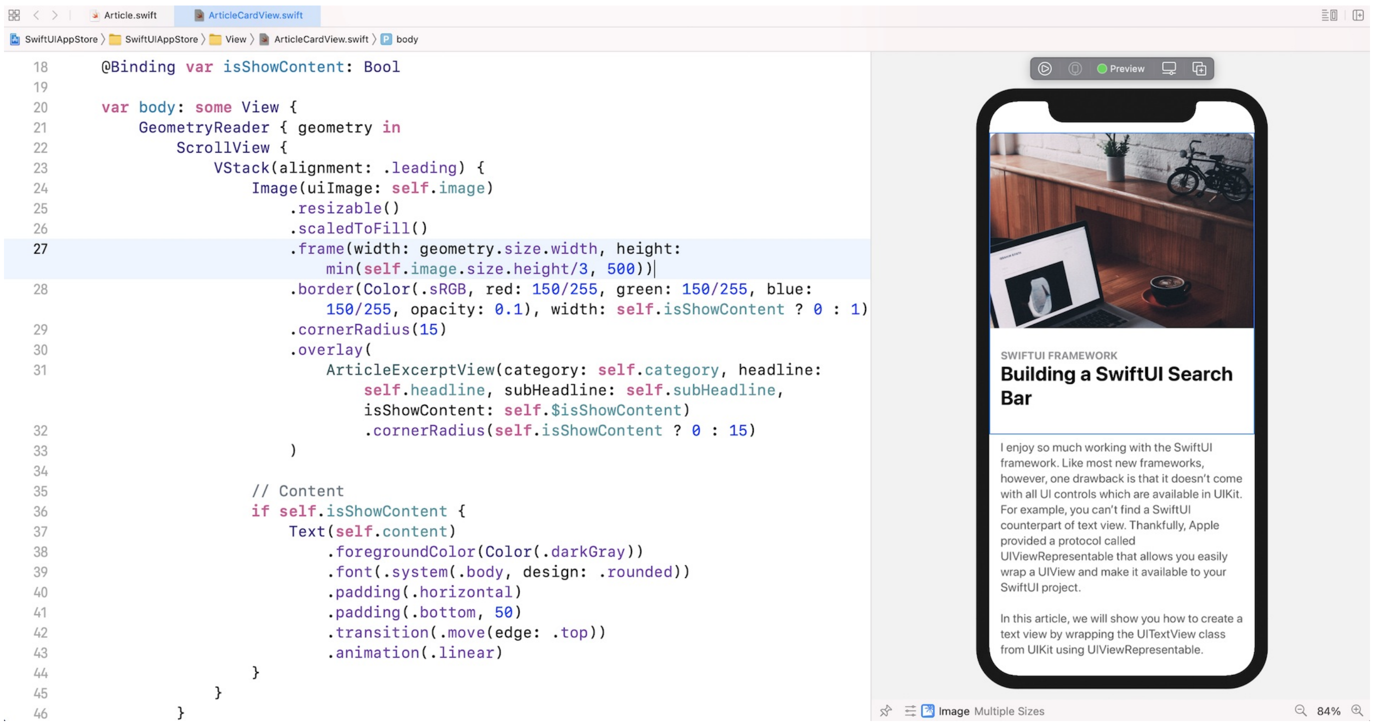
To fix the issue, we have to set the frame's width and ensure it doesn't exceed the width of the screen. The question is how do you find out the screen width? SwiftUI provides a container view called `GeometryReader` which lets you access the size of its parent view. Therefore, we need to embed the `ScrollView` within a `GeometryReader` like this:

```
var body: some View {
    GeometryReader { geometry in
        ScrollView {
            VStack(alignment: .leading) {
                .
                .
                .
            }
        }
        .shadow(color: Color(.sRGB, red: 64/255, green: 64/255, blue: 64/255, opac
ity: 0.3), radius: self.isShowContent ? 0 : 15)
    }
}
```

Within the closure of `GeometryReader`, it has a parameter that provides you with extra information about the view such as size and position. So, to limit the width of the frame to the size of the screen, you can modify the `.frame` modifier like this:

```
.frame(width: geometry.size.width, height: min(self.image.size.height/3, 500))
```

In the code, we set the width equal to that of the screen. Once you complete the change, the card view should look great.


Article.swift
ArticleCardView.swift
SwiftUIAppStore 〉 SwiftUIAppStore 〉 View 〉 ArticleCardView.swift 〉 body
Preview
Image Multiple Sizes
84%

```
    @Binding var isShowContent: Bool

    var body: some View {
        GeometryReader { geometry in
            ScrollView {
                VStack(alignment: .leading) {
                    Image(uiImage: self.image)
                        .resizable()
                        .scaledToFill()
                        .frame(width: geometry.size.width, height:
                            min(self.image.size.height/3, 500))
                        .border(Color(.sRGB, red: 150/255, green: 150/255, blue:
                            150/255, opacity: 0.1), width: self.isShowContent ? 0 : 1)
                        .cornerRadius(15)
                        .overlay(
                            ArticleExcerptView(category: self.category, headline:
                                self.headline, subHeadline: self.subHeadline,
                                isShowContent: self.$isShowContent)
                                .cornerRadius(self.isShowContent ? 0 : 15)
                        )

                    // Content
                    if self.isShowContent {
                        Text(self.content)
                            .foregroundColor(Color(.darkGray))
                            .font(.system(.body, design: .rounded))
                            .padding(.horizontal)
                            .padding(.bottom, 50)
                            .transition(.move(edge: .top))
                            .animation(.linear)
                    }
                }
            }
```

SWIFTUI FRAMEWORK
Building a SwiftUI Search Bar

I enjoy so much working with the SwiftUI framework. Like most new frameworks, however, one drawback is that it doesn't come with all UI controls which are available in UIKit. For example, you can't find a SwiftUI counterpart of text view. Thankfully, Apple provided a protocol called UIViewRepresentable that allows you easily wrap a UIView and make it available to your SwiftUI project.

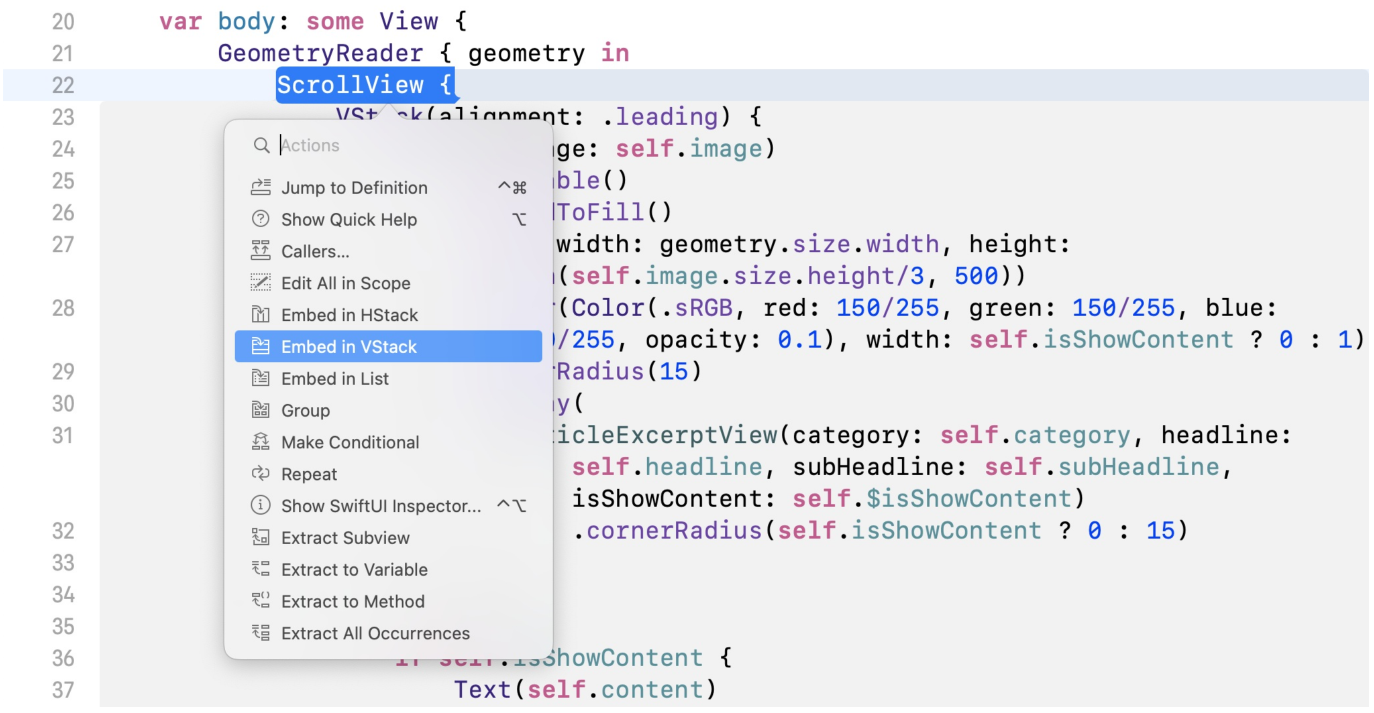
In this article, we will show you how to create a text view by wrapping the UITextView class from UIKit using UIViewRepresentable.

Figure 10. The width of the image is now equal to that of the screen

Adding the close button

The card view is almost done, but there is still one thing left. We haven't implemented the close button yet. To overlay the button on top of the image, we will embed the scroll view in a `ZStack`. You can modify the code directly to add the `ZStack` or let's show you another way to do that.

Hold the command key and click on `ScrollView`, you should then see a context menu. Since there is no option for `ZStack`, choose *Embed in VStack* to embed the scroll view in a `VStack`.



```
    var body: some View {
        GeometryReader { geometry in
            ScrollView {
                VSt  k(alignment: .leading) {
                              ge: self.image)
                              ble()
                              ToFill()
                              width: geometry.size.width, height:
                              (self.image.size.height/3, 500))
                              (Color(.sRGB, red: 150/255, green: 150/255, blue:
                              /255, opacity: 0.1), width: self.isShowContent ? 0 : 1)
                              Radius(15)
                              y(
                              icleExcerptView(category: self.category, headline:
                                 self.headline, subHeadline: self.subHeadline,
                                 isShowContent: self.$isShowContent)
                                 .cornerRadius(self.isShowContent ? 0 : 15)

                          showContent {
                        Text(self.content)
```

Figure 11. Embed the scroll view in a VStack

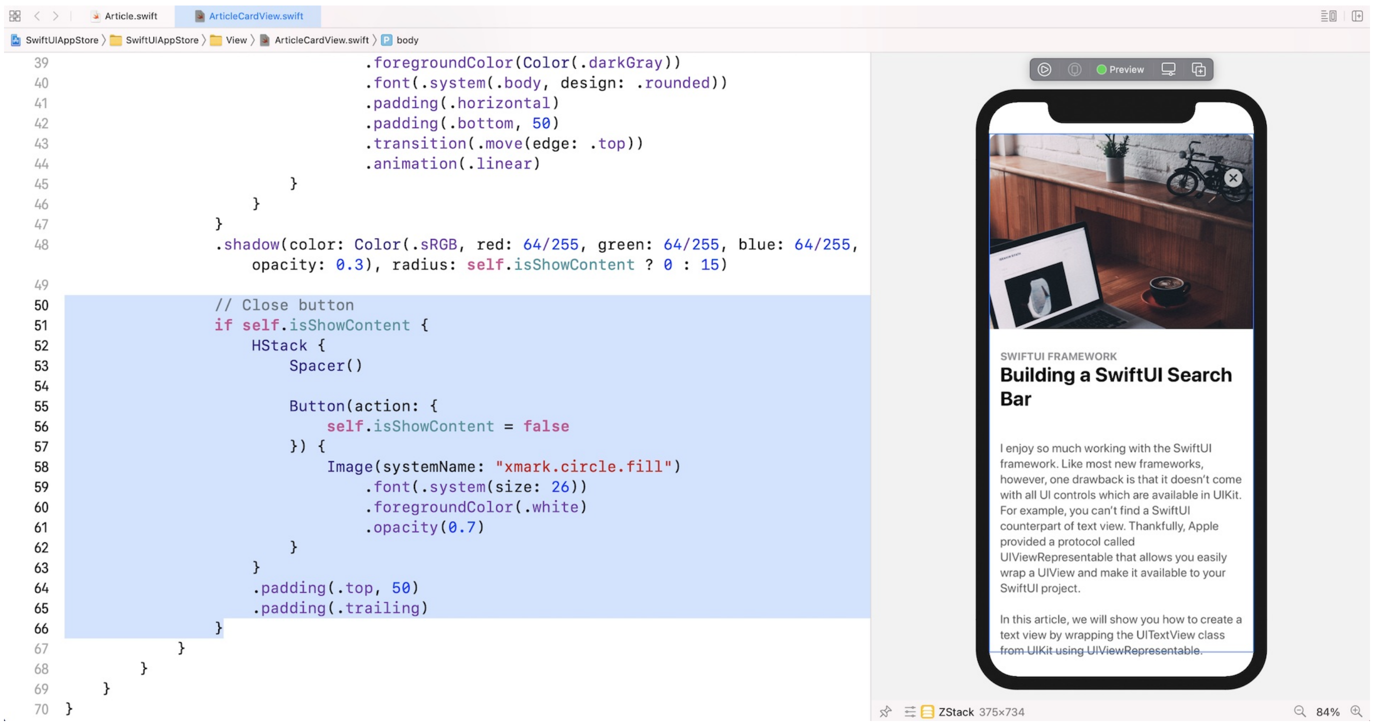
Xcode will automatically indent the code and embed the scroll view in the `VStack`. Now change `VStack` to `ZStack` and set its `alignment` to `.topTrailing` because we want to place the close button near the top-right corner. Your code should look like this:

```
var body: some View {
    GeometryReader { geometry in
        ZStack(alignment: .topTrailing) {
            ScrollView {
                VStack(alignment: .leading) {
                    .
                    .
                    .
                }
            }
            .shadow(color: Color(.sRGB, red: 64/255, green: 64/255, blue: 64/255,
opacity: 0.3), radius: self.isShowContent ? 0 : 15)
        }
    }
}
```

Next, insert the following code right below the `.shadow` modifier to add the close button:

```
if self.isShowContent {
    HStack {
        Spacer()

        Button(action: {
            self.isShowContent = false
        }) {
            Image(systemName: "xmark.circle.fill")
                .font(.system(size: 26))
                .foregroundColor(.white)
                .opacity(0.7)
        }
    }
    .padding(.top, 50)
    .padding(.trailing)
}
```

After the modification, the preview should display the close button when the value of `isShowContent` is set to `true`.


Article.swift
ArticleCardView.swift
SwiftUIAppStore 〉 SwiftUIAppStore 〉 View 〉 ArticleCardView.swift 〉 body
Preview
ZStack 375×734
84%

```
                                .foregroundColor(Color(.darkGray))
                                .font(.system(.body, design: .rounded))
                                .padding(.horizontal)
                                .padding(.bottom, 50)
                                .transition(.move(edge: .top))
                                .animation(.linear)
                        }
                    }
                }
                .shadow(color: Color(.sRGB, red: 64/255, green: 64/255, blue: 64/255,
                    opacity: 0.3), radius: self.isShowContent ? 0 : 15)

                // Close button
                if self.isShowContent {
                    HStack {
                        Spacer()

                        Button(action: {
                            self.isShowContent = false
                        }) {
                            Image(systemName: "xmark.circle.fill")
                                .font(.system(size: 26))
                                .foregroundColor(.white)
                                .opacity(0.7)
                        }
                    }
                    .padding(.top, 50)
                    .padding(.trailing)
                }
            }
        }
    }
}
```

SWIFTUI FRAMEWORK

Building a SwiftUI Search Bar

I enjoy so much working with the SwiftUI framework. Like most new frameworks, however, one drawback is that it doesn't come with all UI controls which are available in UIKit. For example, you can't find a SwiftUI counterpart of text view. Thankfully, Apple provided a protocol called UIViewRepresentable that allows you easily wrap a UIView and make it available to your SwiftUI project.

In this article, we will show you how to create a text view by wrapping the UITextView class from UIKit using UIViewRepresentable.

Figure 12. Adding the close button

Building the List View

Now that we've implemented the layout of the card view, let's switch over to `ContentView.swift` and create the list view. At the very top of the list view, is the top bar with a heading and a profile photo.

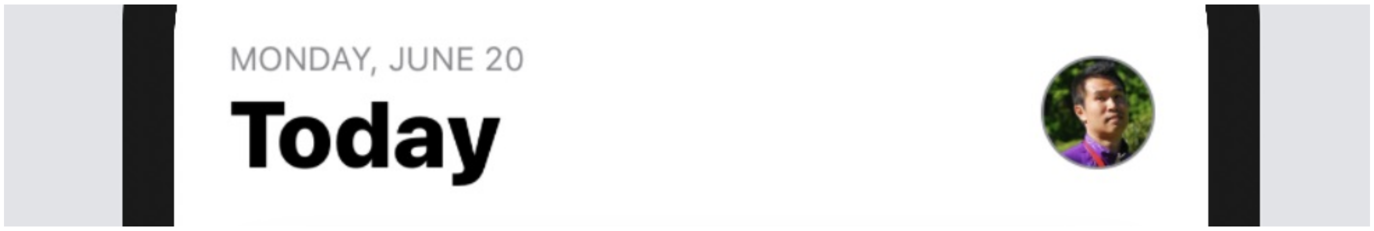


Figure 13. The top bar

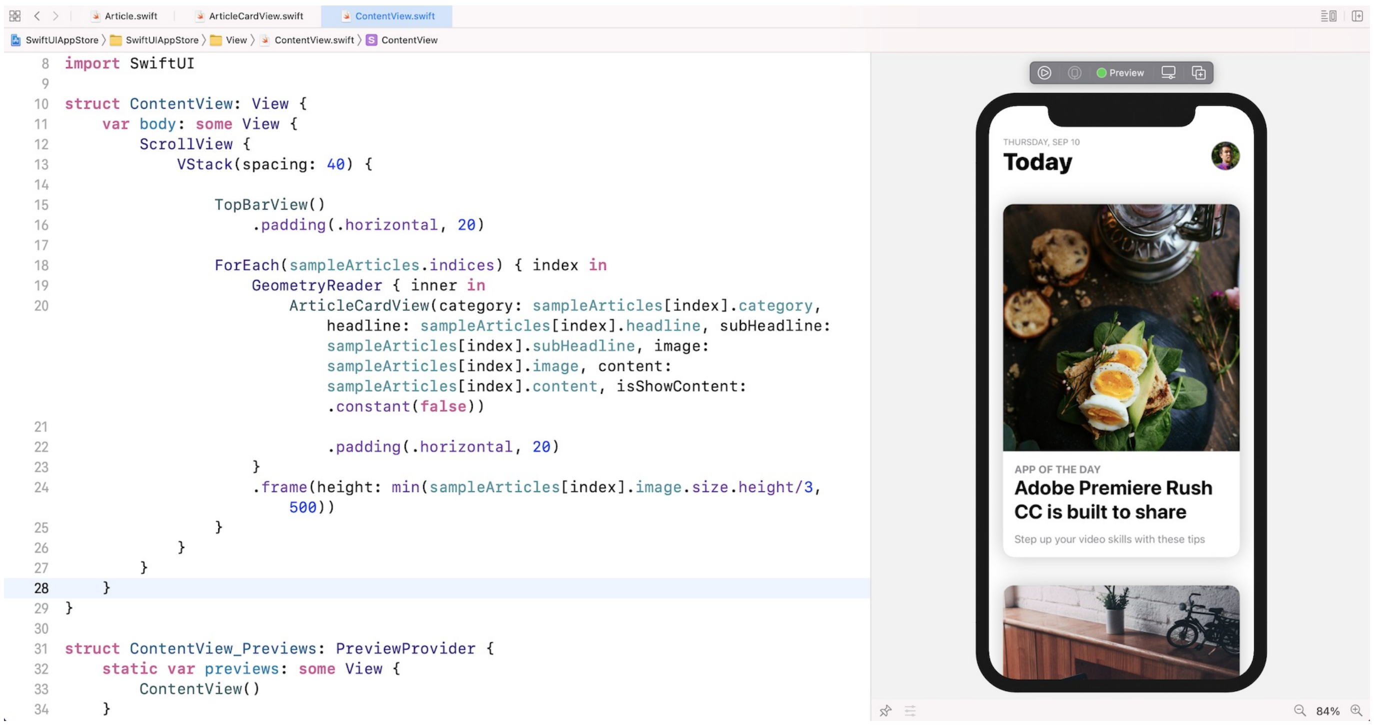
I believe you should know how to create the layout by using `VStack` and `HStack`. To better organize the code, I will create the top bar and the avatar in two separate structs. Insert the following code in `ContentView.swift`:

```
struct TopBarView : View {

    var body: some View {
        HStack(alignment: .lastTextBaseline) {
            VStack(alignment: .leading) {
                Text(getCurrentDate().uppercased())
                    .font(.caption)
                    .foregroundColor(.secondary)
                Text("Today")
                    .font(.largeTitle)
                    .fontWeight(.heavy)
            }

            Spacer()

            AvatarView(image: "profile", width: 40, height: 40)

        }
    }

    func getCurrentDate(with format: String = "EEEE, MMM d") -> String {
        let dateFormatter = DateFormatter()
        dateFormatter.dateFormat = format
        return dateFormatter.string(from: Date())
    }
}

struct AvatarView: View {
    let image: String
    let width: CGFloat
    let height: CGFloat

    var body: some View {
        Image(image)
            .resizable()
            .frame(width: width, height: height)
            .clipShape(Circle())
            .overlay(Circle().stroke(Color.gray, lineWidth: 1))
    }
}
```

Next, update the code of `ContentView` like this:

```
struct ContentView: View {

    var body: some View {
        ScrollView {
            VStack(spacing: 40) {

                TopBarView()
                    .padding(.horizontal, 20)

                ForEach(sampleArticles.indices) { index in
                    GeometryReader { inner in
                        ArticleCardView(category: sampleArticles[index].category,
headline: sampleArticles[index].headline, subHeadline: sampleArticles[index].subHe
adline, image: sampleArticles[index].image, content: sampleArticles[index].content
, isShowContent: .constant(false))

                            .padding(.horizontal, 20)
                    }
                    .frame(height: min(sampleArticles[index].image.size.height/3,
500))
                }
            }
        }
    }
}
```

We embed a `VStack` in a `ScrollView` to create the vertical scroll view. In the code block, we loop through all the `sampleArticles` using `ForEach` and create an `ArticleCardView` for each article. If your code works properly, the preview canvas should show you a list of articles.


Article.swift
ArticleCardView.swift
ContentView.swift
SwiftUIAppStore 〉 SwiftUIAppStore 〉 View 〉 ContentView.swift 〉 ContentView
Preview
THURSDAY, SEP 10
Today
APP OF THE DAY
Adobe Premiere Rush CC is built to share
Step up your video skills with these tips
84%

```swift
import SwiftUI

struct ContentView: View {
    var body: some View {
        ScrollView {
            VStack(spacing: 40) {

                TopBarView()
                    .padding(.horizontal, 20)

                ForEach(sampleArticles.indices) { index in
                    GeometryReader { inner in
                        ArticleCardView(category: sampleArticles[index].category,
                            headline: sampleArticles[index].headline, subHeadline:
                            sampleArticles[index].subHeadline, image:
                            sampleArticles[index].image, content:
                            sampleArticles[index].content, isShowContent:
                            .constant(false))

                            .padding(.horizontal, 20)
                    }
                    .frame(height: min(sampleArticles[index].image.size.height/3,
                        500))
                }
            }
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
```

Figure 14. The list view showing a list of card views

You may wonder why we need to wrap each of the `ArticleCardView` with a `GeometryReader`. I will explain its purpose in a later section.

Expanding the Card View to Full Screen

Now it comes to the hard part. How do you switch the card view from excerpt mode to full content mode? Right now, we set the `isShowContent` parameter to `.constant(false)`. To switch between these two modes, each of the card views should have a variable to keep track of its state.

Therefore, declare the following state variable in `ContentView`:

```swift
@State private var showContents: [Bool] = Array(repeating: false, count: sampleArt
icles.count)
```

We need a variable to store the state of each card view. This is why we declare the `showContents` array and mark it as a state variable. By default, all card views are in the excerpt state. Thus, the values of the `showContents` array are all set to `false`. Later, when a card is tapped, we will change the state of that particular card view from `false` to `true`.

Furthermore, we also need to have a handy way to find out the current content mode. Continue to insert the following code:

```
enum ContentMode {
    case list
    case content
}

private var contentMode: ContentMode {
    self.showContents.contains(true) ? .content : .list
}
```

The `ContentView` can either display a list of articles or an article in full content mode. So, in the code above, we declare a `ContentMode` enum to represent these states. The `contentMode` variable is a computed property that computes the current mode of the view. When one of the items in the `showContents` array is set to `true`, we know it's in the *content* mode. Otherwise, it's in the *list* mode.

Now, modify the initialization of `ArticleCardView`. Instead of using `.constant(false)`, pass it the binding of the state variable (i.e. `self.$showContents[index]`):

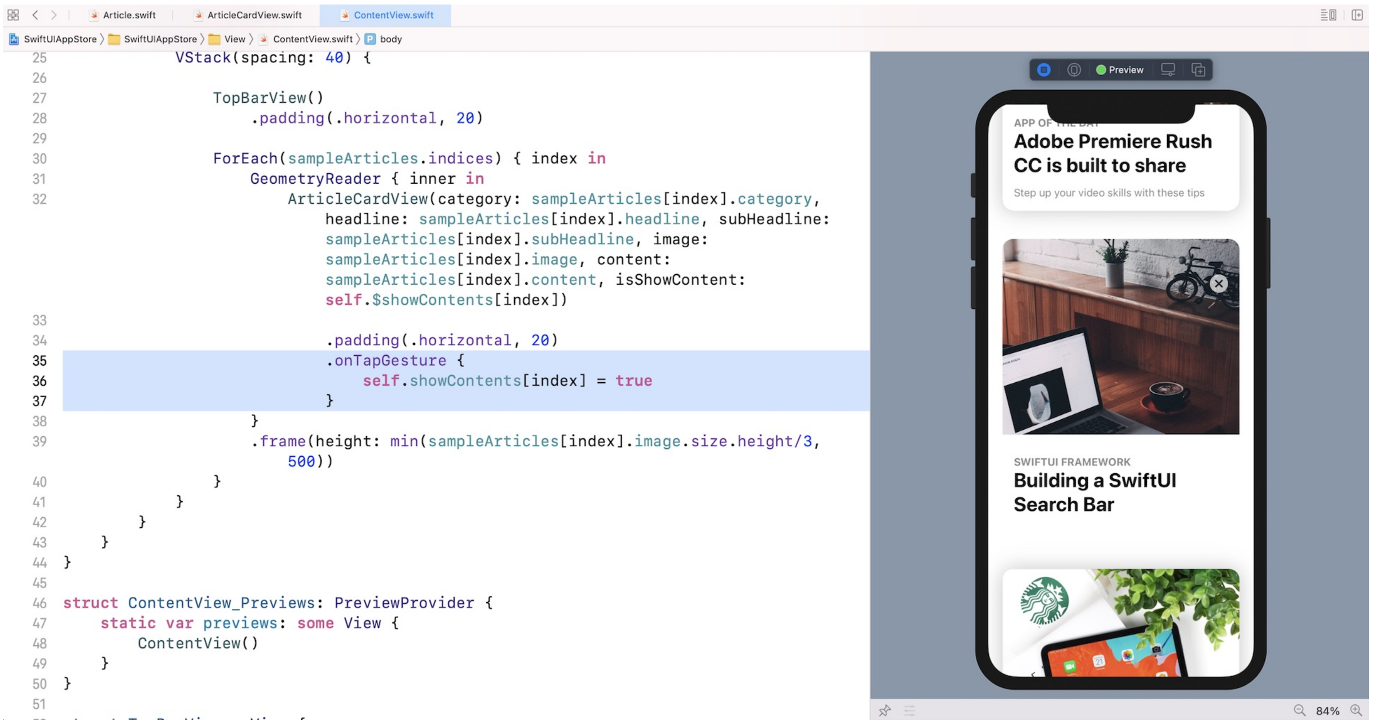
```
ArticleCardView(category: sampleArticles[index].category, headline: sampleArticles
[index].headline, subHeadline: sampleArticles[index].subHeadline, image: sampleArt
icles[index].image, content: sampleArticles[index].content, isShowContent: self.$s
howContents[index])
```

Handling the Tap Gesture

When the user taps one of the card views, the selected card will be changed to full screen mode. To capture the tap gesture, attach the `.onTapGesture` modifier below the code you just added:

```
.onTapGesture {
    self.showContents[index] = true
}
```

Let's have a quick test to see how the app functions after making the changes. When you run the app in the preview canvas, tap any of the card views to see the result. Though it doesn't work as expected, the card view should show the content of the article and hide the sub-headline. Additionally, you should be able to tap the close button to return to the excerpt mode. If you can't see the content, drag up the card view to reveal it.



```
            VStack(spacing: 40) {

                TopBarView()
                    .padding(.horizontal, 20)

                ForEach(sampleArticles.indices) { index in
                    GeometryReader { inner in
                        ArticleCardView(category: sampleArticles[index].category,
                            headline: sampleArticles[index].headline, subHeadline:
                            sampleArticles[index].subHeadline, image:
                            sampleArticles[index].image, content:
                            sampleArticles[index].content, isShowContent:
                            self.$showContents[index])

                            .padding(.horizontal, 20)
                            .onTapGesture {
                                self.showContents[index] = true
                            }
                    }
                    .frame(height: min(sampleArticles[index].image.size.height/3,
                        500))
                }
            }
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 15. Testing the tap gesture

Controlling the Padding and Opacity

There is still a lot of work to do. Let's fix the issues one by one. First, the padding. When the card view is in full content mode, there should be no padding. So, in `ContentView.swift`, change the `.padding` modifier of `ArticleCardView` like this:

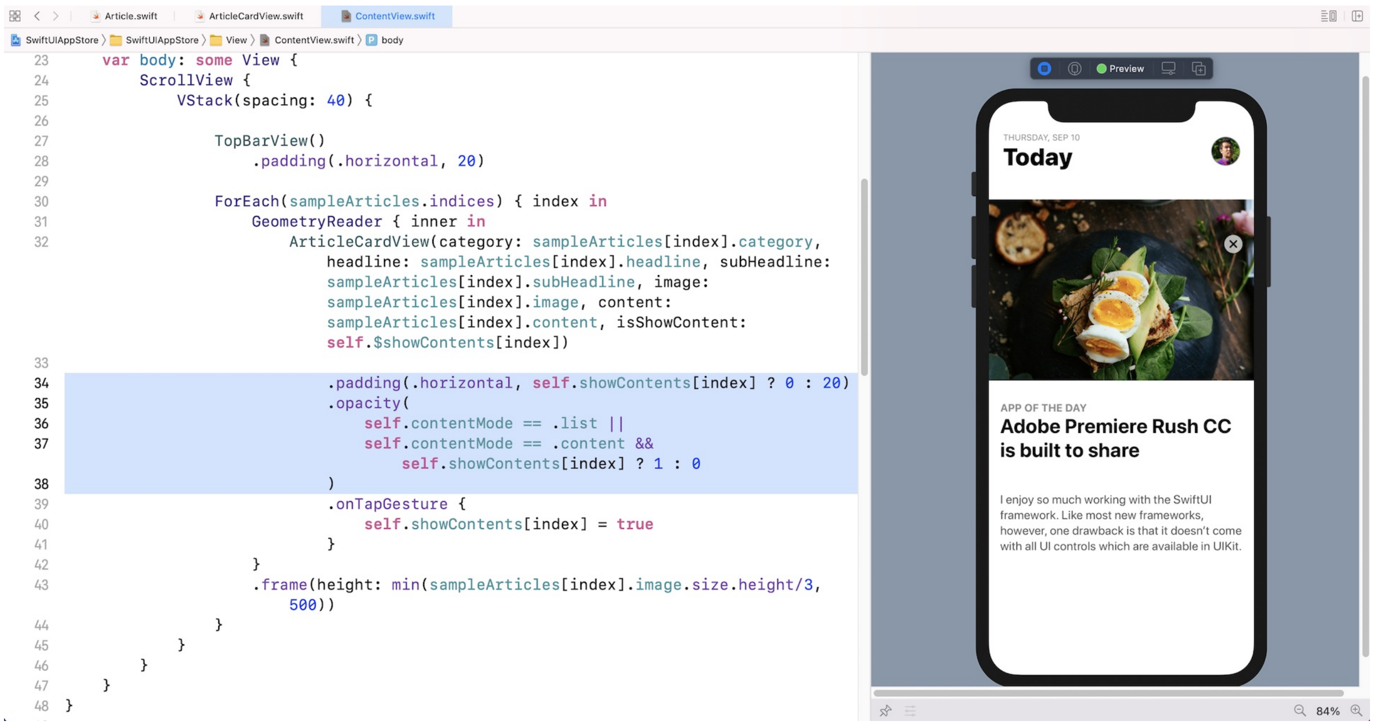
```
.padding(.horizontal, self.showContents[index] ? 0 : 20)
```

When one of the card views is in full content mode, the rest of the card views should be hidden. To do that, we can vary the opacity of the card views to control their appearance. Insert the following code after the `.padding` modifier:

```
.opacity(
    self.contentMode == .list ||
    self.contentMode == .content && self.showContents[index] ? 1 : 0
)
```

When the content view is in *list* mode, all card views should be visible. But when the content view is switched to *content* mode, only the selected card should be visible. The rest of the card views are set to be invisible.

Now let's test the app again. This time, when you tap any of the card views, only the selected view appears on the screen. The rest are temporarily hidden.


Article.swift
ArticleCardView.swift
ContentView.swift
SwiftUIAppStore 〉 SwiftUIAppStore 〉 View 〉 ContentView.swift 〉 body
Preview
84%

```
    var body: some View {
        ScrollView {
            VStack(spacing: 40) {

                TopBarView()
                    .padding(.horizontal, 20)

                ForEach(sampleArticles.indices) { index in
                    GeometryReader { inner in
                        ArticleCardView(category: sampleArticles[index].category,
                            headline: sampleArticles[index].headline, subHeadline:
                            sampleArticles[index].subHeadline, image:
                            sampleArticles[index].image, content:
                            sampleArticles[index].content, isShowContent:
                            self.$showContents[index])

                            .padding(.horizontal, self.showContents[index] ? 0 : 20)
                            .opacity(
                                self.contentMode == .list ||
                                self.contentMode == .content &&
                                    self.showContents[index] ? 1 : 0
                            )
                            .onTapGesture {
                                self.showContents[index] = true
                            }
                    }
                    .frame(height: min(sampleArticles[index].image.size.height/3,
                        500))
                }
            }
        }
    }
}
```

THURSDAY, SEP 10

Today

APP OF THE DAY

Adobe Premiere Rush CC is built to share

I enjoy so much working with the SwiftUI framework. Like most new frameworks, however, one drawback is that it doesn't come with all UI controls which are available in UIKit.

Figure 16. The card view in full content mode

You may want to hide the top bar when the content view is in content mode. Similarly, you can attach an `.opacity` modifier to the `TopBarView` to control its appearance:

```
.opacity(self.contentMode == .content ? 0 : 1)
```

Adjusting the Height of the Card View

Right now, one of the major issues is that the card view doesn't expand to fit the whole screen. Its height doesn't change when the card view is switched to full content mode. The following line of code (attached to `GeometryReader`) is the reason why the card view doesn't expand:

```
.frame(height: min(sampleArticles[index].image.size.height/3, 500))
```

The height of the card view is defined not to exceed 500 points. To adjust the view to take up the whole screen, we need to alter the frame's height and make it equal to the height of the screen.

To detect the screen height, we need to add another `GeometryReader` and wrap it around the `ScrollView` in `ContentView` like this:

```
var body: some View {
    GeometryReader { fullView in
        ScrollView {
            VStack(spacing: 40) {
                .
                .
                .
            }
        }
    }
}
```

The parameter `fullView` allows you to access the full size of the screen because the parent of `GeometryReader` is the whole screen.

Next, modify the `frame` modifier like this:

```
.frame(height: self.showContents[index] ? fullView.size.height + fullView.safeArea
Insets.top + fullView.safeAreaInsets.bottom : min(sampleArticles[index].image.size
.height/3, 500))
```

For the selected card view, we adjust its height to take up the whole screen. By default, the `height` property only gives us the height of the safe area. To calculate the height of the actual screen, we need to include the top and bottom parts of the safe area insets.

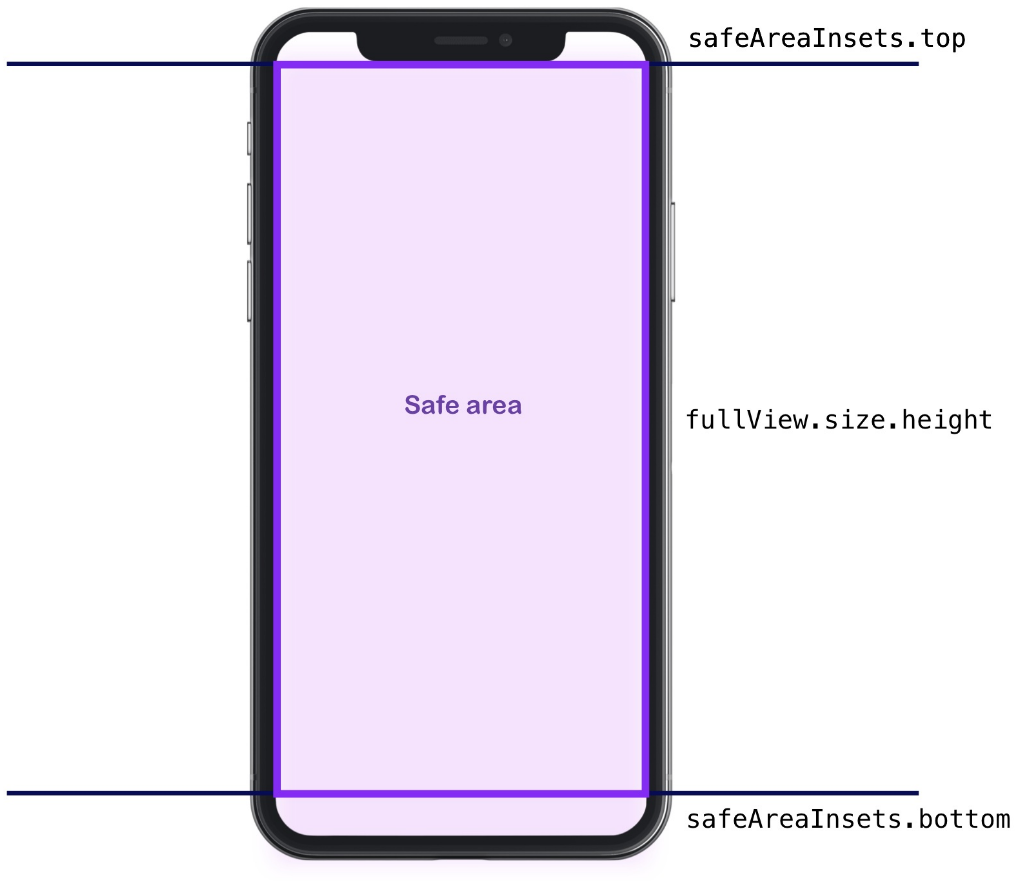
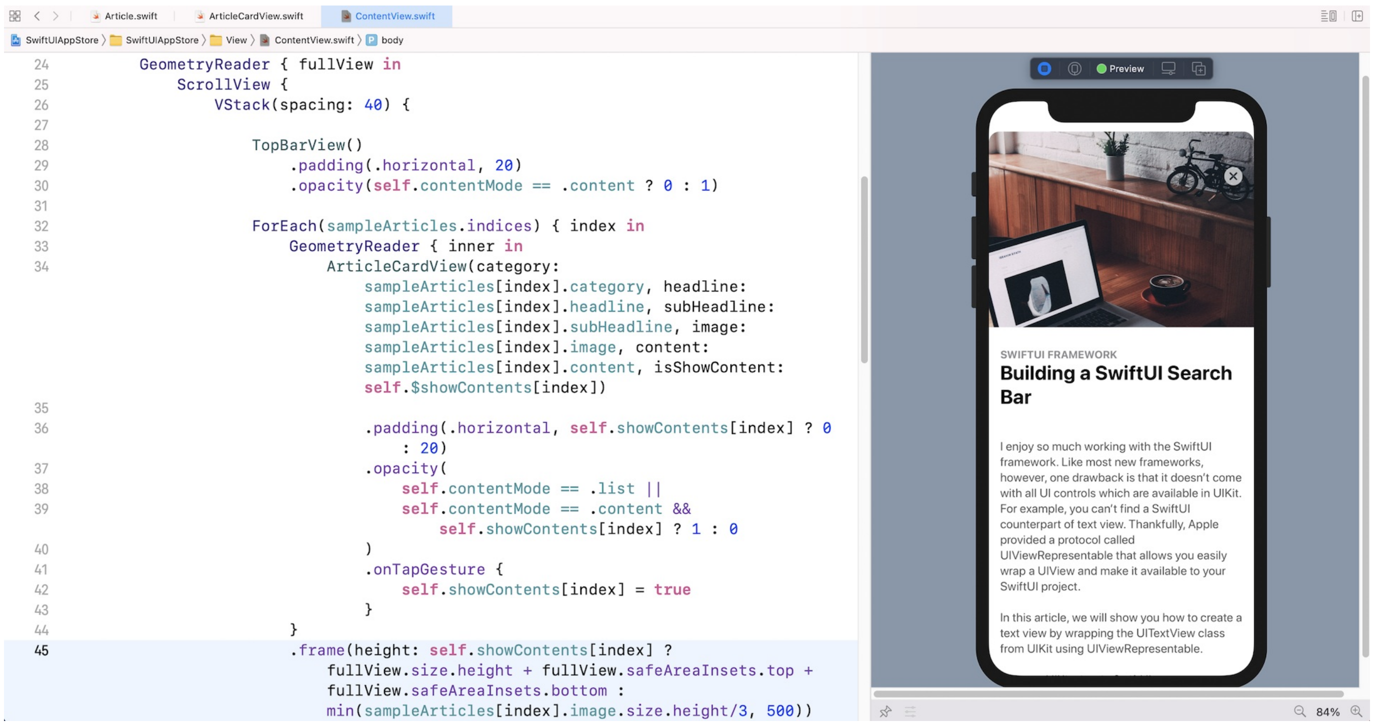


Figure 17. Understanding the safe area insets

Run the app in the preview canvas again and see the changes. Now the height of the selected card view will be adjusted automatically.


Article.swift
ArticleCardView.swift
ContentView.swift
SwiftUIAppStore 〉 SwiftUIAppStore 〉 View 〉 ContentView.swift 〉 body
Preview
84%

```
        GeometryReader { fullView in
            ScrollView {
                VStack(spacing: 40) {

                    TopBarView()
                        .padding(.horizontal, 20)
                        .opacity(self.contentMode == .content ? 0 : 1)

                    ForEach(sampleArticles.indices) { index in
                        GeometryReader { inner in
                            ArticleCardView(category:
                                sampleArticles[index].category, headline:
                                sampleArticles[index].headline, subHeadline:
                                sampleArticles[index].subHeadline, image:
                                sampleArticles[index].image, content:
                                sampleArticles[index].content, isShowContent:
                                self.$showContents[index])

                                .padding(.horizontal, self.showContents[index] ? 0
                                    : 20)
                                .opacity(
                                    self.contentMode == .list ||
                                    self.contentMode == .content &&
                                        self.showContents[index] ? 1 : 0
                                )
                                .onTapGesture {
                                    self.showContents[index] = true
                                }
                        }
                        .frame(height: self.showContents[index] ?
                            fullView.size.height + fullView.safeAreaInsets.top +
                            fullView.safeAreaInsets.bottom :
                            min(sampleArticles[index].image.size.height/3, 500))
```

SWIFTUI FRAMEWORK

Building a SwiftUI Search Bar

I enjoy so much working with the SwiftUI framework. Like most new frameworks, however, one drawback is that it doesn't come with all UI controls which are available in UIKit. For example, you can't find a SwiftUI counterpart of text view. Thankfully, Apple provided a protocol called UIViewRepresentable that allows you easily wrap a UIView and make it available to your SwiftUI project.

In this article, we will show you how to create a text view by wrapping the UITextView class from UIKit using UIViewRepresentable.

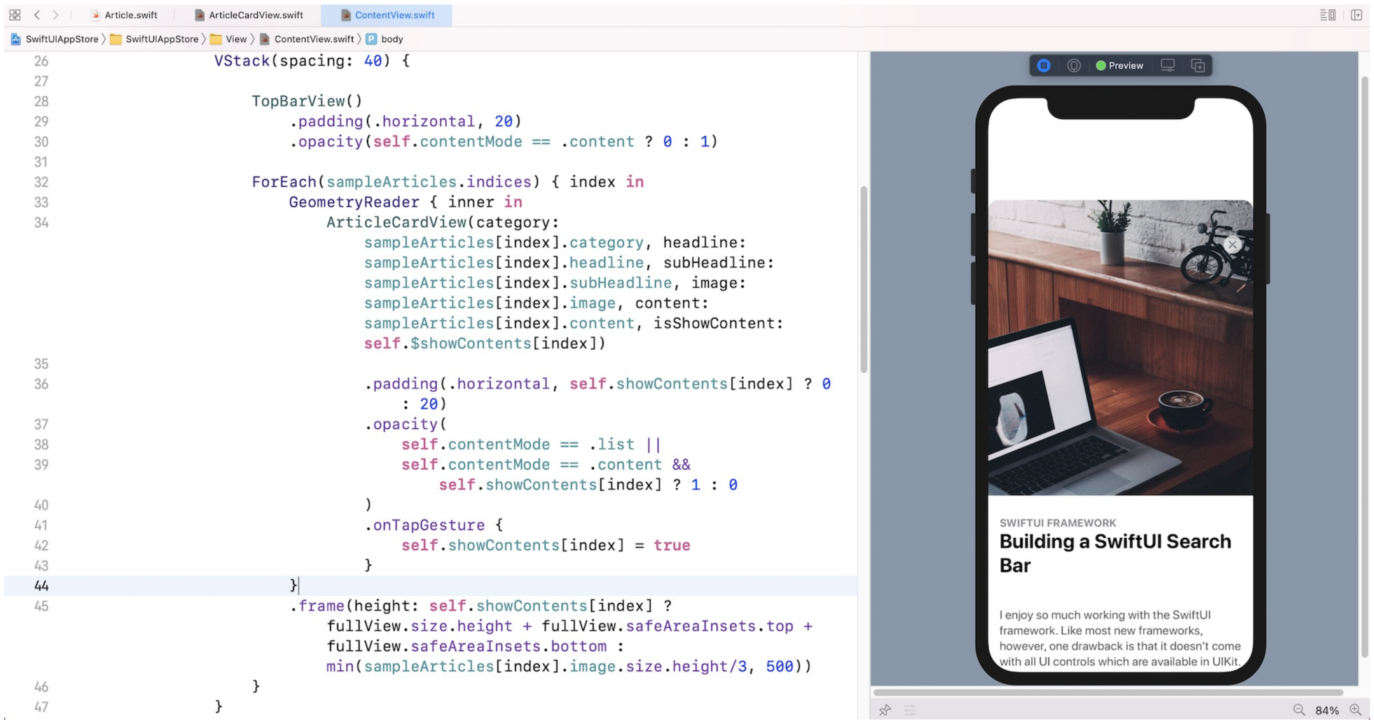
Figure 18. Adjusting the height of the card in full content mode

Enlarging the Image

We haven't finished the implementation yet. Even though we fixed one of the major issues, there are still a few issues waiting for us. Next up is the featured image. In full content mode, I want to make the image a bit larger. This is an easy fix. Just switch over to `ArticleCardView.swift` and change the `.frame` modifier of the `Image` view like this:

```
.frame(width: geometry.size.width, height: self.isShowContent ? geometry.size.heig
ht * 0.7 : min(self.image.size.height/3, 500))
```

When the card view is displaying the article content, the height of the image is now adjusted to 70% of the screen height. You may alter the value to suit your preference. Now go back to `ContentView.swift` and test the change. The featured image becomes larger in full content mode.


Article.swift
ArticleCardView.swift
ContentView.swift
SwiftUIAppStore 〉 SwiftUIAppStore 〉 View 〉 ContentView.swift 〉 body
Preview
84%

```swift
VStack(spacing: 40) {

    TopBarView()
        .padding(.horizontal, 20)
        .opacity(self.contentMode == .content ? 0 : 1)

    ForEach(sampleArticles.indices) { index in
        GeometryReader { inner in
            ArticleCardView(category:
                sampleArticles[index].category, headline:
                sampleArticles[index].headline, subHeadline:
                sampleArticles[index].subHeadline, image:
                sampleArticles[index].image, content:
                sampleArticles[index].content, isShowContent:
                self.$showContents[index])

                .padding(.horizontal, self.showContents[index] ? 0
                    : 20)
                .opacity(
                    self.contentMode == .list ||
                    self.contentMode == .content &&
                        self.showContents[index] ? 1 : 0
                )
                .onTapGesture {
                    self.showContents[index] = true
                }
        }
        .frame(height: self.showContents[index] ?
            fullView.size.height + fullView.safeAreaInsets.top +
            fullView.safeAreaInsets.bottom :
            min(sampleArticles[index].image.size.height/3, 500))
    }
}
```

SWIFTUI FRAMEWORK

Building a SwiftUI Search Bar

I enjoy so much working with the SwiftUI framework. Like most new frameworks, however, one drawback is that it doesn't come with all UI controls which are available in UIKit.

Figure 19. The featured image becomes larger in full content mode

Adjusting the Offset

Next, the view's offset. When the card view is in full content mode, we expect it to take up the whole screen. The card view does extend its height but it doesn't shift its position to cover the screen, as you can see in figure 18 and 19.

You probably know that we can adjust the view's offset by attaching an `.offset` modifier. However, the question is what value of offset should we pass to the modifier? In other words, how can we find out the offset value between the top edge of the card view and the top edge of the screen?

Notice that we have wrapped each of the `ArticleCardView` with a `GeometryReader`. I haven't explained why we implemented this. We also haven't made use of the `inner` parameter.

Now it's time for the `inner` parameter to come into play. In addition to giving us the size of a view, the `inner` parameter, which is a `GeometryProxy`, also allows us to access the frame of the view and its values (like position). By using the following line of code, we can find out the distance between the top edge of the card view and the screen:

```
inner.frame(in: .global).minY)
```

The `.global` value indicates that we want to find out the frame's values using the *global* coordinate space. In other words, the values are relative to the whole screen. The `minY` property gives you the distance between the top edge of the card view and the screen. Figure 20 illustrates the values of `minY` for a couple of the card views. Please note that the value of `minY` changes accordingly as you scroll through the list.

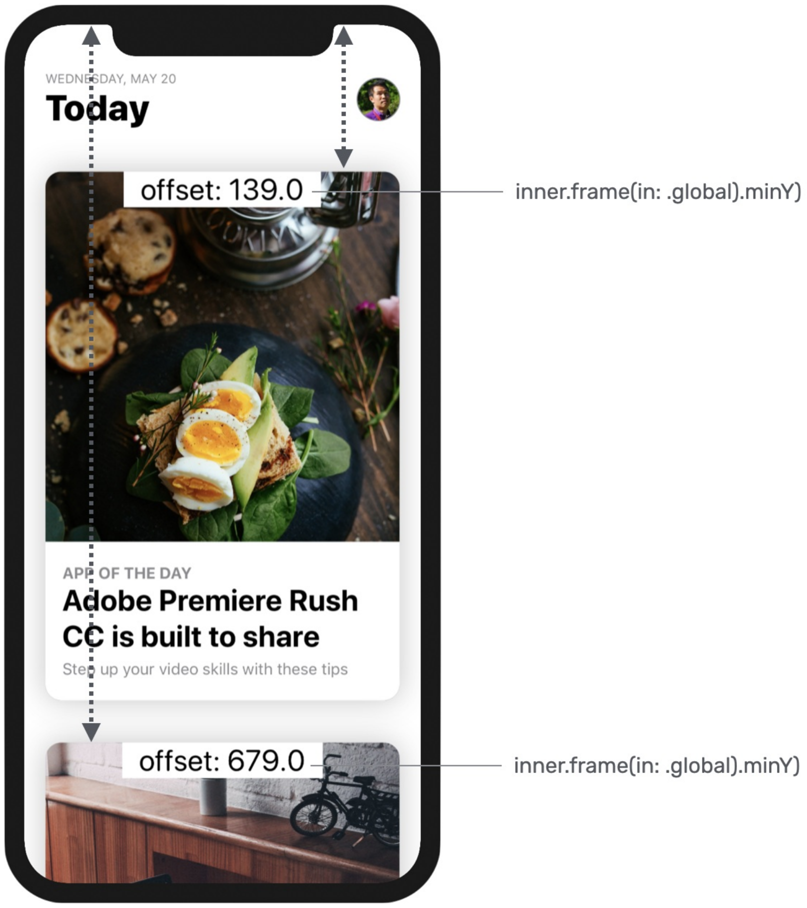
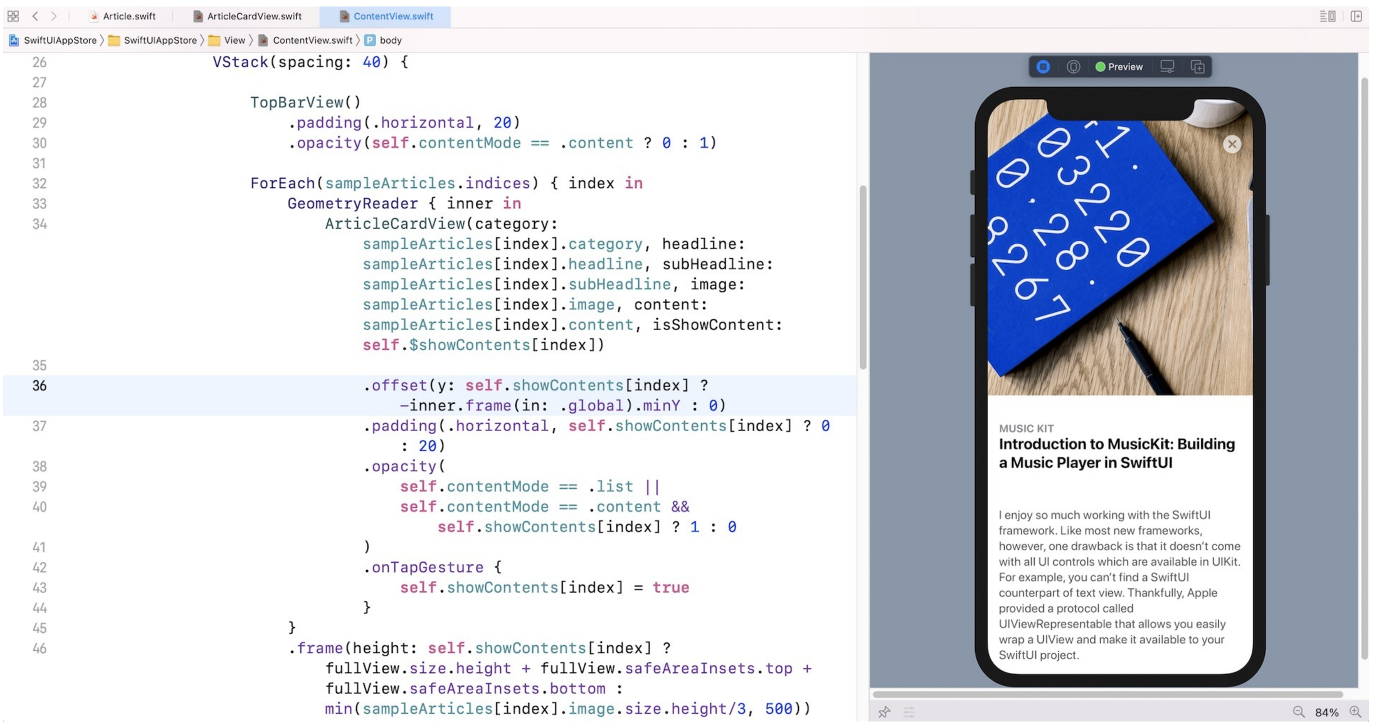

WEDNESDAY, MAY 20
Today
offset: 139.0
inner.frame(in: .global).minY)
APP OF THE DAY
Adobe Premiere Rush CC is built to share
Step up your video skills with these tips
offset: 679.0
inner.frame(in: .global).minY)

Figure 20. Understanding the minY value of the frame

Now that we have figured out the offset value, all we need to do is attach the `.offset` modifier to `ArticleCardView` in `ContentView.swift` like this:

```
.offset(y: self.showContents[index] ? -inner.frame(in: .global).minY : 0)
```

The line of code above adjusts the offset of the card view only when it's in full content mode. By setting a negative value of `inner.frame(in: .global).minY`, this will shift the card view up. If you try to test the app again and tap any of the card views, the selected card should take up the whole screen.


Article.swift
ArticleCardView.swift
ContentView.swift
SwiftUIAppStore 〉 SwiftUIAppStore 〉 View 〉 ContentView.swift 〉 body
Preview
84%

```
            VStack(spacing: 40) {

                TopBarView()
                    .padding(.horizontal, 20)
                    .opacity(self.contentMode == .content ? 0 : 1)

                ForEach(sampleArticles.indices) { index in
                    GeometryReader { inner in
                        ArticleCardView(category:
                            sampleArticles[index].category, headline:
                            sampleArticles[index].headline, subHeadline:
                            sampleArticles[index].subHeadline, image:
                            sampleArticles[index].image, content:
                            sampleArticles[index].content, isShowContent:
                            self.$showContents[index])

                            .offset(y: self.showContents[index] ?
                                -inner.frame(in: .global).minY : 0)
                            .padding(.horizontal, self.showContents[index] ? 0
                                : 20)
                            .opacity(
                                self.contentMode == .list ||
                                self.contentMode == .content &&
                                    self.showContents[index] ? 1 : 0
                            )
                            .onTapGesture {
                                self.showContents[index] = true
                            }
                    }
                    .frame(height: self.showContents[index] ?
                        fullView.size.height + fullView.safeAreaInsets.top +
                        fullView.safeAreaInsets.bottom :
                        min(sampleArticles[index].image.size.height/3, 500))
```

MUSIC KIT
Introduction to MusicKit: Building a Music Player in SwiftUI

I enjoy so much working with the SwiftUI framework. Like most new frameworks, however, one drawback is that it doesn't come with all UI controls which are available in UIKit. For example, you can't find a SwiftUI counterpart of text view. Thankfully, Apple provided a protocol called UIViewRepresentable that allows you easily wrap a UIView and make it available to your SwiftUI project.

Figure 21. The card view now takes up the whole screen

Animating the View Changes

The transition of a card view from excerpt mode to full content mode is not animated. To animate the transition, you can attach an `.animation` modifier to the `GeometryReader` of `ArticleCardView`. Place the following line of code after the `.frame` modifier:

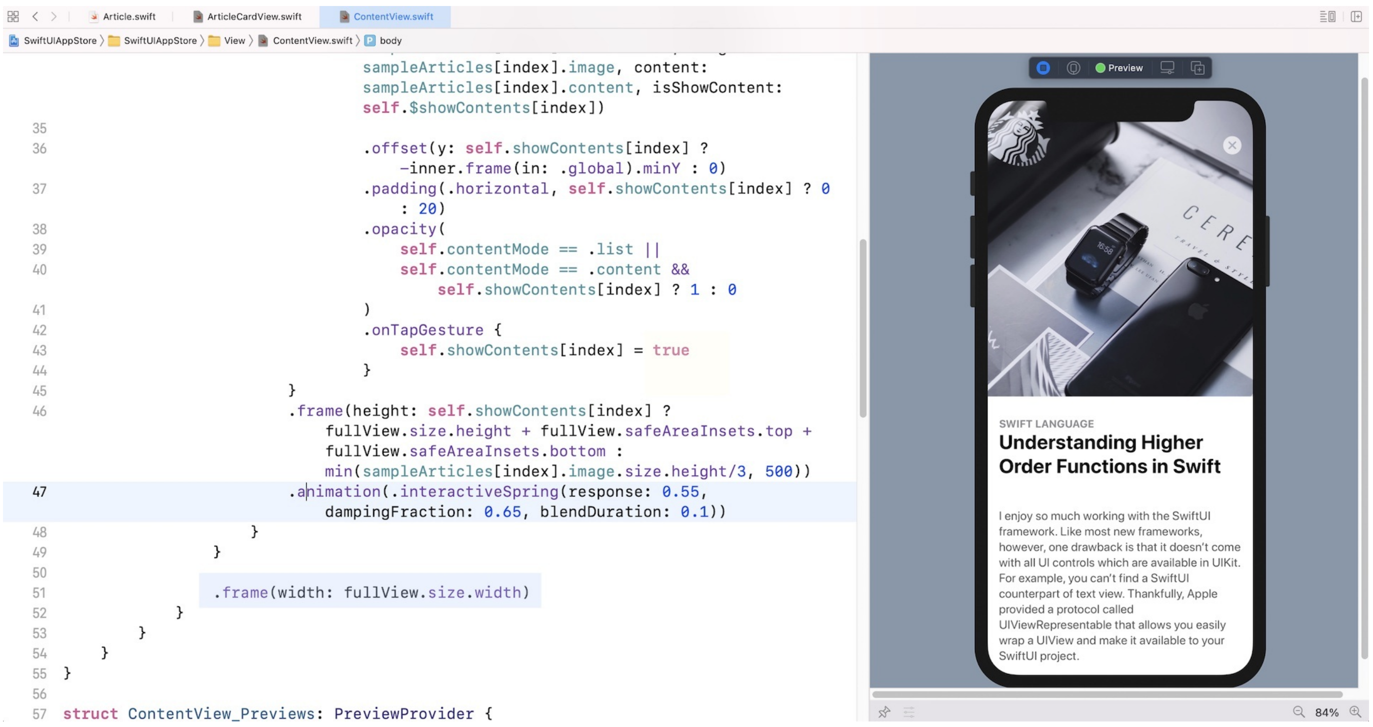
```
.animation(.interactiveSpring(response: 0.55, dampingFraction: 0.65, blendDuration
: 0.1))
```

With just a line of code, SwiftUI automatically animates the view transition. Run the app in the simulator or in the preview canvas. You will see a slick animation when the card view expands to full screen.

There is a minor issue after adding the animation modifier. When the app is first started, it also animates the population of the card views. To resolve the issue, you can fix the width of the `VStack`. Attach the following line of code to the `VStack`:

```
.frame(width: fullView.size.width)
```

Figure 22 shows you where the code above is added.



Article.swift
ArticleCardView.swift
ContentView.swift
SwiftUIAppStore 〉 SwiftUIAppStore 〉 View 〉 ContentView.swift 〉 body
Preview
84%

```
                                sampleArticles[index].image, content:
                                sampleArticles[index].content, isShowContent:
                                self.$showContents[index])

                                .offset(y: self.showContents[index] ?
                                    -inner.frame(in: .global).minY : 0)
                                .padding(.horizontal, self.showContents[index] ? 0
                                    : 20)
                                .opacity(
                                    self.contentMode == .list ||
                                    self.contentMode == .content &&
                                        self.showContents[index] ? 1 : 0
                                )
                                .onTapGesture {
                                    self.showContents[index] = true
                                }
                        }
                        .frame(height: self.showContents[index] ?
                            fullView.size.height + fullView.safeAreaInsets.top +
                            fullView.safeAreaInsets.bottom :
                            min(sampleArticles[index].image.size.height/3, 500))
                        .animation(.interactiveSpring(response: 0.55,
                            dampingFraction: 0.65, blendDuration: 0.1))
                    }
                }

                .frame(width: fullView.size.width)
            }
        }
    }
}

struct ContentView_Previews: PreviewProvider {
```

SWIFT LANGUAGE

Understanding Higher Order Functions in Swift

I enjoy so much working with the SwiftUI framework. Like most new frameworks, however, one drawback is that it doesn't come with all UI controls which are available in UIKit. For example, you can't find a SwiftUI counterpart of text view. Thankfully, Apple provided a protocol called UIViewRepresentable that allows you easily wrap a UIView and make it available to your SwiftUI project.

Figure 22. Attaching the animation modifier to animate the view transition

Summary

Congratulations! You've built an App Store like animation using SwiftUI. After implementing this demo project, I hope you understand how to create complex view animations and understand how to use GeometryReader to perfect your UI.

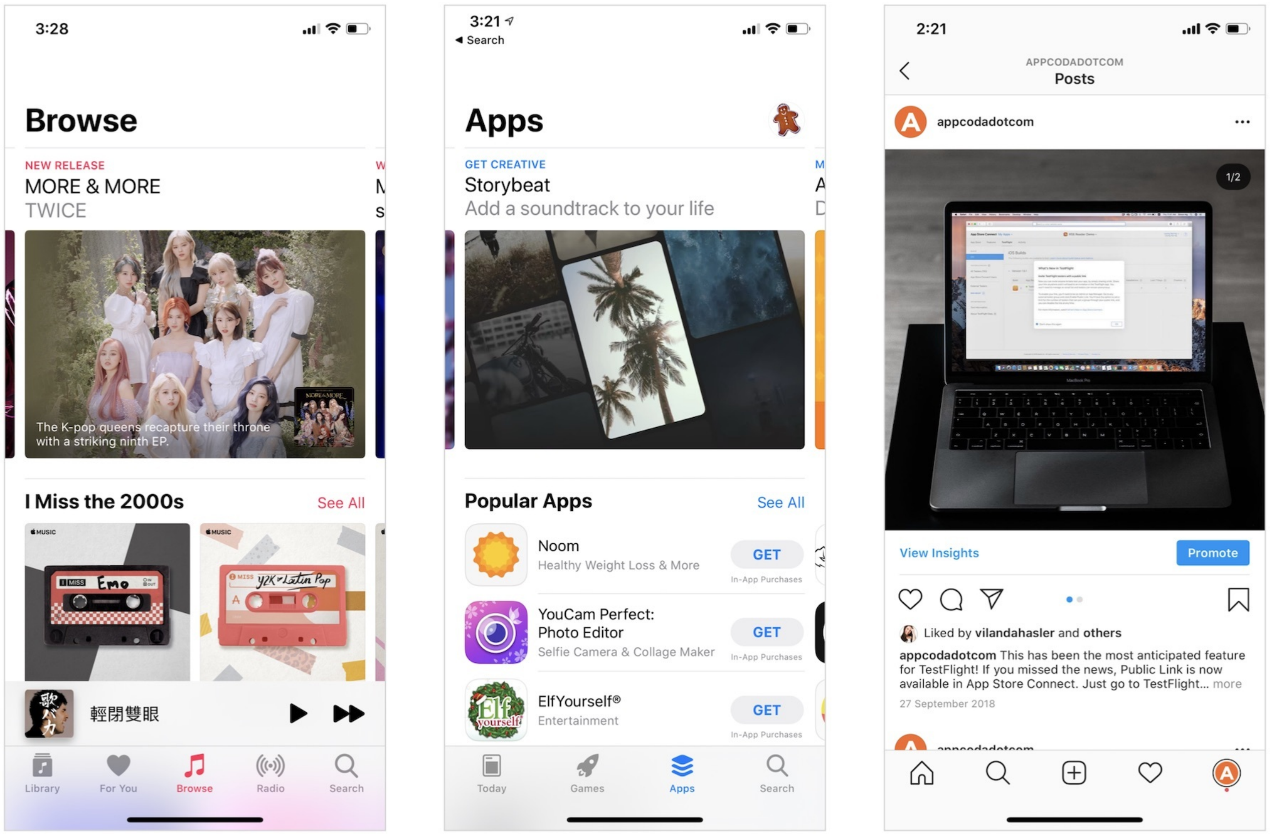
Animation is an essential part of the UI these days. As you can see, SwiftUI has made it very easy for developers to build some beautiful animations and screen transitions. In your next app project, don't forget to apply the techniques you learned in this chapter to improve the user experience of your app.

For reference, you can download the complete project here:

- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUIAppStore.zip)

Chapter 27
Building an Image Carousel

Carousel is one of the common UI patterns that you see in most mobile and web apps. Some people refer it as an image slider or rotator. However, whatever name you call it, a carousel is designed to display a set of data in finite screen space. For example, an image carousel may show a single image from its collection with a navigation control suggesting additional content. Users can swipe the screen to navigate through the image set. This is how Instagram presents multiple images to users. You can also find similar carousels in many other iOS apps such as Apple's Music and App Store.



3:28
Browse
NEW RELEASE
MORE & MORE
TWICE
The K-pop queens recapture their throne with a striking ninth EP.
I Miss the 2000s
See All
輕閉雙眼
Library
For You
Browse
Radio
Search
3:21
Search
Apps
GET CREATIVE
Storybeat
Add a soundtrack to your life
Popular Apps
See All
Noom
Healthy Weight Loss & More
GET
In-App Purchases
YouCam Perfect: Photo Editor
Selfie Camera & Collage Maker
ElfYourself®
Entertainment
Today
Games
Apps
Search
2:21
APPCODADOTCOM
Posts
appcodadotcom
1/2
View Insights
Promote
Liked by vilandahasler and others
appcodadotcom This has been the most anticipated feature for TestFlight! If you missed the news, Public Link is now available in App Store Connect. Just go to TestFlight... more
27 September 2018

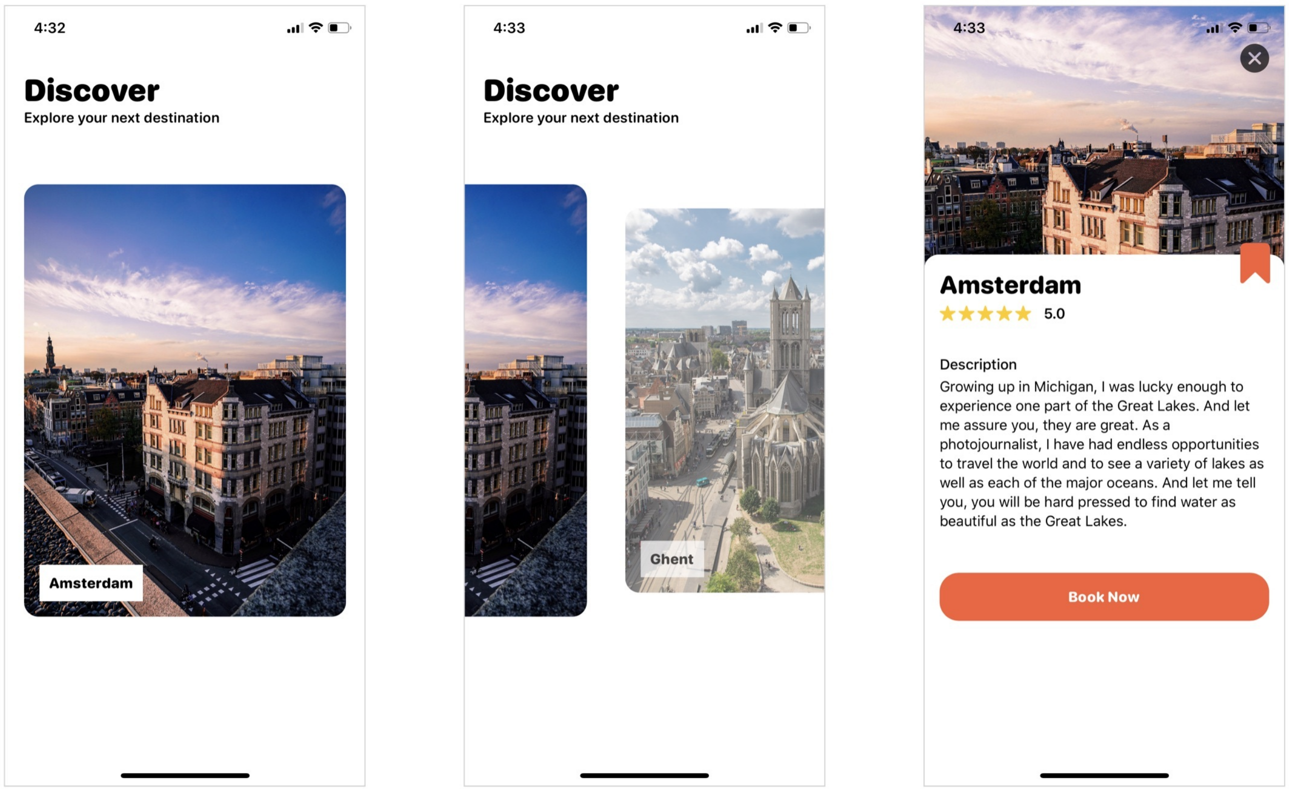
Figure 1. Sample carousel in the Music, App Store, and Instagram app

In this chapter, you will learn how to build an image carousel entirely in SwiftUI. There are various ways to implement a carousel. One approach is to integrate with the UIKit component `UIPageViewController` and use it to create the carousel. However, we will explore an alternative approach and create the carousel completely in SwiftUI.

Let's get started.

Introducing the Travel Demo App

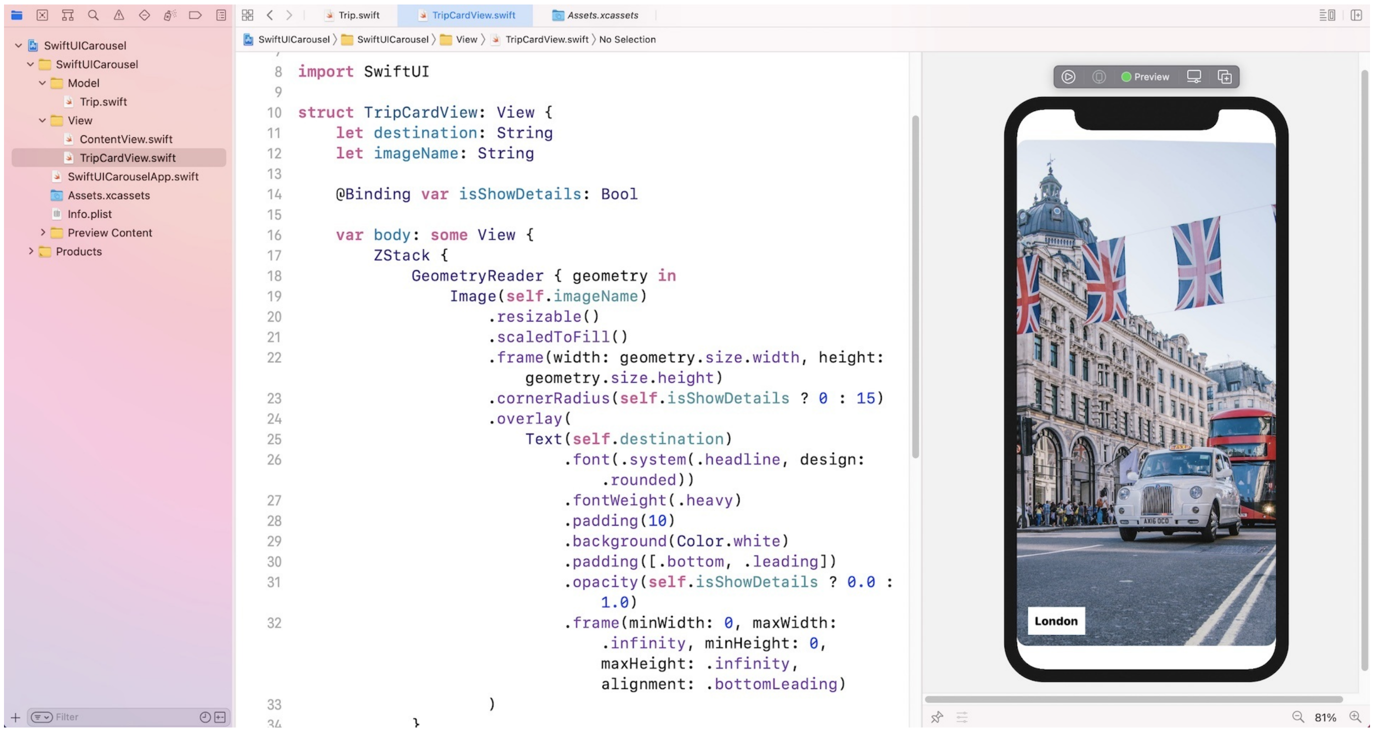
Likes other chapters, I walk you through the implementation by building a demo app. The app displays a collection of travel destinations in the form of a carousel. To browse through the trips, the user can swipe right to view the subsequent destination or swipe left to check out the previous trip. To make this demo app more engaging, the user can tap a destination to see its detail. So, in addition to the implementation of a carousel, you will also learn some animation techniques that can be applied in your own apps. Figure 2 shows you some sample screenshots of the demo app. To see it in action, you can check out the video at https://link.appcoda.com/carousel-demo.



Description
Growing up in Michigan, I was lucky enough to experience one part of the Great Lakes. And let me assure you, they are great. As a photojournalist, I have had endless opportunities to travel the world and to see a variety of lakes as well as each of the major oceans. And let me tell you, you will be hard pressed to find water as beautiful as the Great Lakes.

Figure 2. The demo app

To save you time from building the app from scratch and to focus on developing the carousel, I've created a starter project for you. Please download it from https://www.appcoda.com/resources/swiftui2/SwiftUICarouselStarter.zip and unzip the package.



```swift
import SwiftUI

struct TripCardView: View {
    let destination: String
    let imageName: String

    @Binding var isShowDetails: Bool

    var body: some View {
        ZStack {
            GeometryReader { geometry in
                Image(self.imageName)
                    .resizable()
                    .scaledToFill()
                    .frame(width: geometry.size.width, height:
                        geometry.size.height)
                    .cornerRadius(self.isShowDetails ? 0 : 15)
                    .overlay(
                        Text(self.destination)
                            .font(.system(.headline, design:
                                .rounded))
                            .fontWeight(.heavy)
                            .padding(10)
                            .background(Color.white)
                            .padding([.bottom, .leading])
                            .opacity(self.isShowDetails ? 0.0 :
                                1.0)
                            .frame(minWidth: 0, maxWidth:
                                .infinity, minHeight: 0,
                                maxHeight: .infinity,
                                alignment: .bottomLeading)

                    )
```

Figure 3. The starter project

The starter project comes with the following features:

1. It bundles the required images in the asset catalog.
2. The `ContentView.swift` file is the default SwiftUI view generated by Xcode.
3. The `Trip.swift` file contains the `Trip` struct, which represents a travel destination in the app. For testing purposes, this file also creates the `sampleTrips` array which includes some test data. You may modify its content.
4. The `TripCardView.swift` file implements the UI of a card view. Each card view is designed to display the destination's image. The `isShowDetails` binding controls the appearance of the text label. When `isShowDetails` is set to true, the label will be hidden.

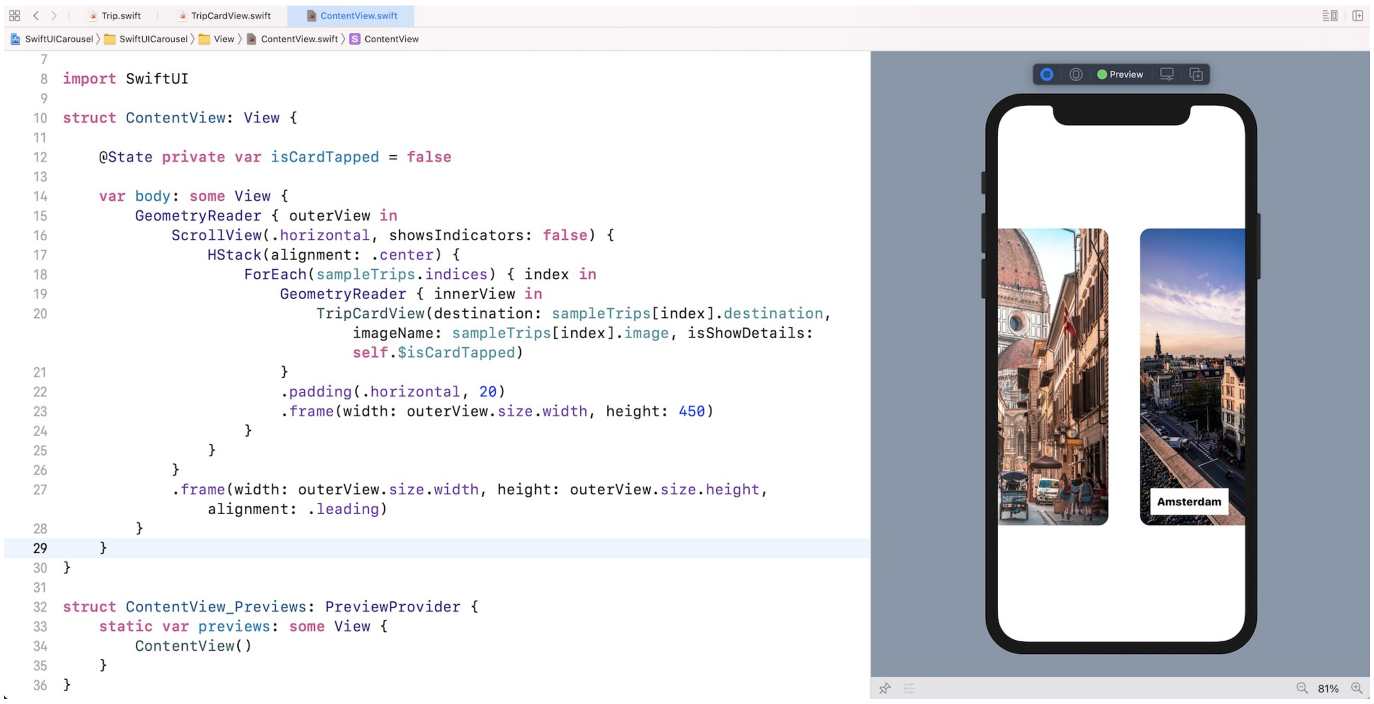
The ScrollView Problem

So, how would you implement the carousel in SwiftUI? At first thought, you may want to create the carousel by using a scroll view. Probably you will write the code in `ContentView.swift` like this:

```
struct ContentView: View {

    @State private var isCardTapped = false

    var body: some View {
        GeometryReader { outerView in
            ScrollView(.horizontal, showsIndicators: false) {
                HStack(alignment: .center) {
                    ForEach(sampleTrips.indices) { index in
                        GeometryReader { innerView in
                            TripCardView(destination: sampleTrips[index].destinati
on, imageName: sampleTrips[index].image, isShowDetails: self.$isCardTapped)
                        }
                        .padding(.horizontal, 20)
                        .frame(width: outerView.size.width, height: 450)
                    }
                }
            }
            .frame(width: outerView.size.width, height: outerView.size.height, ali
gnment: .leading)
        }
    }
}
```

In the code above, we embed an `HStack` with a horizontal `ScrollView` to create the image slider. In the `HStack`, we loop through the `sampleTrips` array and create a `TripCardView` for each trip. To have better control of the card size, we have two GeometryReaders: *outerView* and *innerView*, where the outer view represents the size of the device's screen and the inner view wraps around the card view to control its size. If you haven't read the previous chapter and don't understand what `GeometryReader` is , please refer to chapter 26.

This looks simple, right? If you run the code in the preview canvas, it should result in a horizontal scroll view. You can swipe the screen to scroll through all the card views.


Trip.swift
TripCardView.swift
ContentView.swift
SwiftUICarousel 〉 SwiftUICarousel 〉 View 〉 ContentView.swift 〉 ContentView
Preview
Amsterdam
81%

```swift

import SwiftUI

struct ContentView: View {

    @State private var isCardTapped = false

    var body: some View {
        GeometryReader { outerView in
            ScrollView(.horizontal, showsIndicators: false) {
                HStack(alignment: .center) {
                    ForEach(sampleTrips.indices) { index in
                        GeometryReader { innerView in
                            TripCardView(destination: sampleTrips[index].destination,
                                imageName: sampleTrips[index].image, isShowDetails:
                                self.$isCardTapped)
                        }
                        .padding(.horizontal, 20)
                        .frame(width: outerView.size.width, height: 450)
                    }
                }
            }
            .frame(width: outerView.size.width, height: outerView.size.height,
                alignment: .leading)
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 4. Testing the horizontal scroll view

Does this mean we have completed the carousel? Not yet. There are a couple of major issues:

1. The current implementation doesn't support paging. In other words, you can swipe the screen to continuously scroll through the content. The scroll view can stop at *any* location. For instance, take a look at figure 4. The scroll stops in between two card views. This is not our desired behavior. We expect the scrolling will stop on *paging* boundaries of the content view.
2. When a card view is tapped, we need to find out its index and display its details in a separate view. The problem is that it is not easy to figure out which card view the user has tapped with the current implementation.

Both issues are related to the built-in `ScrollView`. The UIKit version of the scroll view supports paging. However, Apple didn't bring that feature to the SwiftUI framework. To resolve the issue, We need to build our own horizontal scroll view with paging support.

At first, you may think it's hard to develop our own scroll view. But in reality, it is not that hard. If you understand the usage of `HStack` and `DragGesture`, you can build a horizontal scroll view with paging support.

Building a Carousel with HStack and DragGesture

The idea is to layout all of the card views (i.e. trips) in a horizontal stack (HStack). The `HStack` should be long enough to accomodate all the card views but only display a single card view at any time. By default, the horizontal stack is non-scrollable. Therefore, we need to attach a drag gesture recognizer to the stack view and handle the drag on our own. Figure 5 illustrates our implementation of the horizontal scroll view.

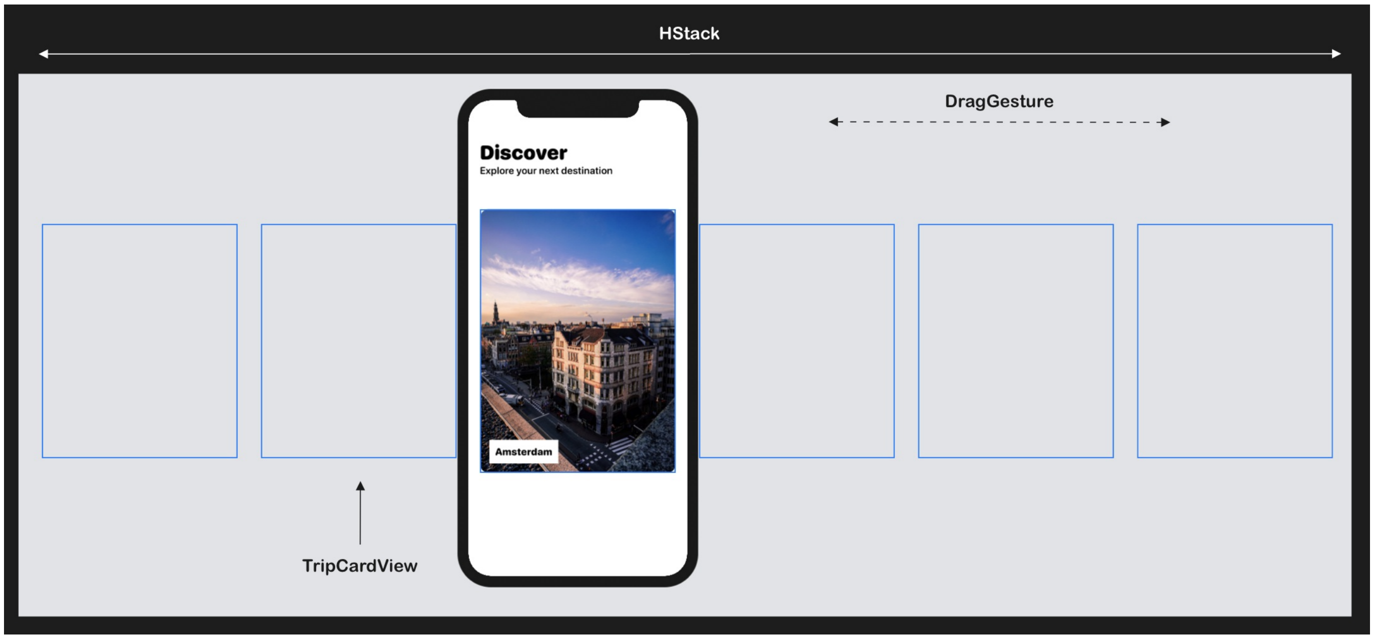


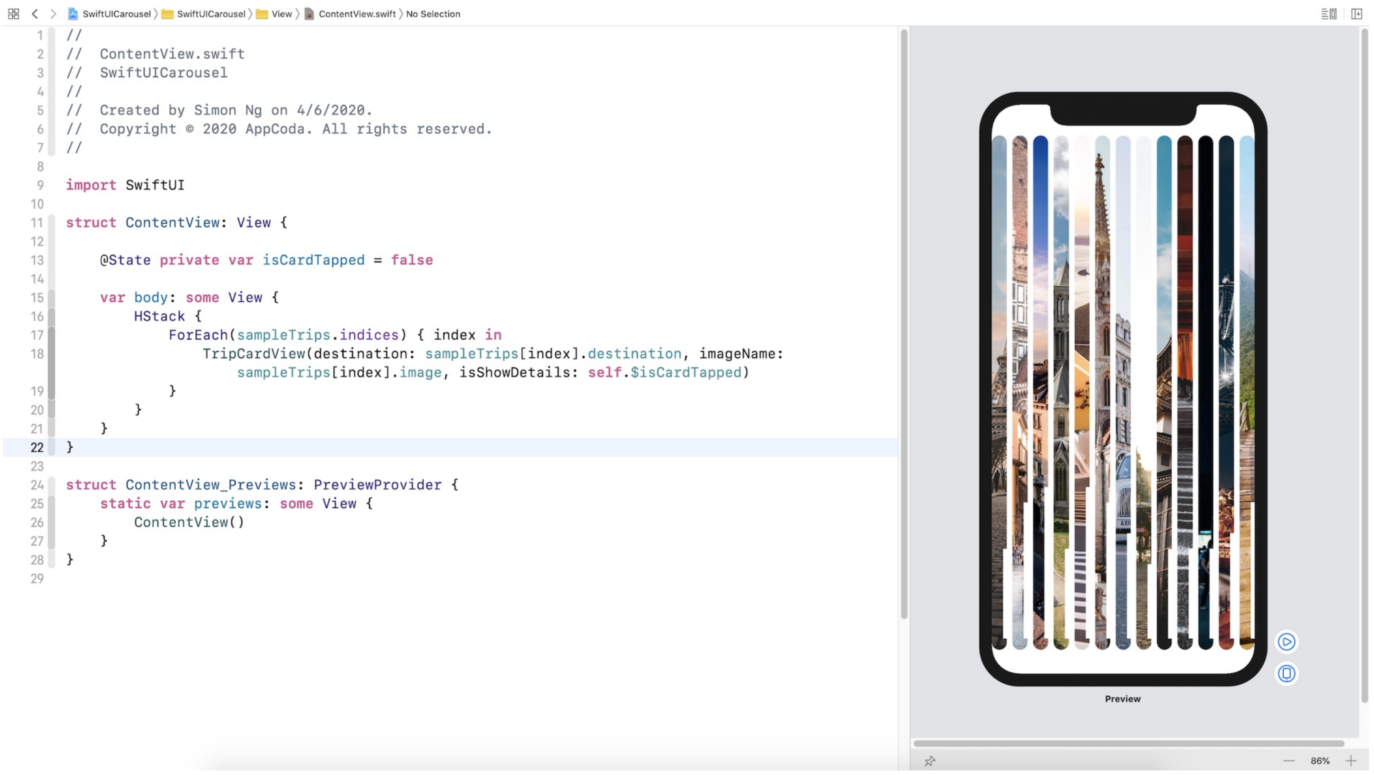
Figure 5. Understanding how to create a horizontal scroll view using HStack and DragGesture

Implementing the horizontal stack

Now let's see how we turn this idea into code. Please bear with me in that you will need to update the code several times. I want to show you the implementation step by step. Open `Content.swift` and update the `body` like this:

```
var body: some View {
    HStack {
        ForEach(sampleTrips.indices) { index in
            TripCardView(destination: sampleTrips[index].destination, imageName: s
ampleTrips[index].image, isShowDetails: self.$isCardTapped)
        }
    }
}
```

In the code above, we start by laying out all card views within an `HStack`. By default, the horizontal stack tries its best to fit all the card views in the available screen space. You should see something like figure 6 in the preview canvas.


SwiftUICarousel 〉 SwiftUICarousel 〉 View 〉 ContentView.swift 〉 No Selection
Preview
86%

```
//
//  ContentView.swift
//  SwiftUICarousel
//
//  Created by Simon Ng on 4/6/2020.
//  Copyright © 2020 AppCoda. All rights reserved.
//

import SwiftUI

struct ContentView: View {

    @State private var isCardTapped = false

    var body: some View {
        HStack {
            ForEach(sampleTrips.indices) { index in
                TripCardView(destination: sampleTrips[index].destination, imageName:
                    sampleTrips[index].image, isShowDetails: self.$isCardTapped)
            }
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

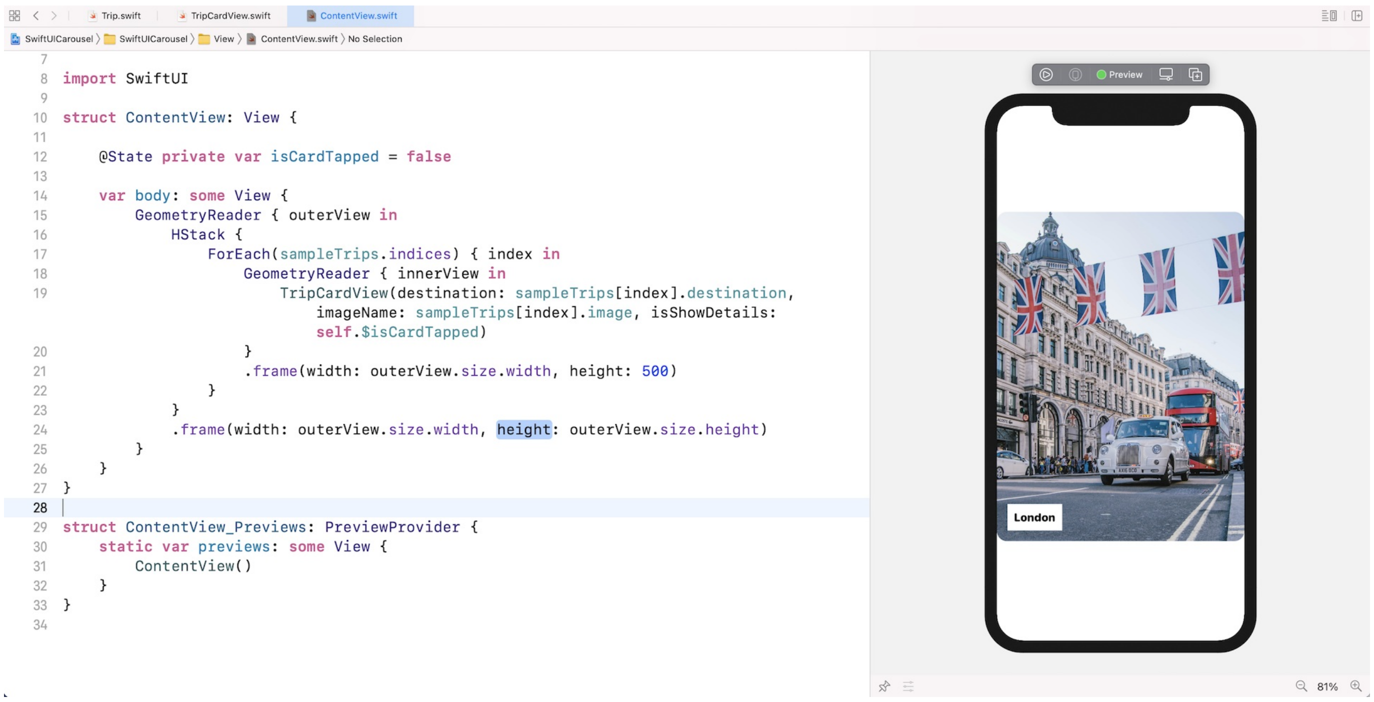
Figure 6. Squeezing all card views to fit the screen

Obviously, this isn't the horizontal stack we want to build. We expect each card view to takes up the width of the screen. To do so, we have to wrap the HStack in a GeometryReader to retrieve the screen size. Update the code in the `body` like this:

```
var body: some View {
    GeometryReader { outerView in
        HStack {
            ForEach(sampleTrips.indices) { index in
                GeometryReader { innerView in
                    TripCardView(destination: sampleTrips[index].destination, imag
eName: sampleTrips[index].image, isShowDetails: self.$isCardTapped)
                }
                .frame(width: outerView.size.width, height: 500)
            }
        }
        .frame(width: outerView.size.width, height: outerView.size.height)
    }
}
```

The `outerView` parameter provides us the screen width and height, while the `innerView` parameter allows us to have better control of the size and position of the card view.

In the code above, we attach the `.frame` modifier to the card view and set its width to the screen width (i.e. `outerView.size.width`). This ensures that each card view takes up the whole screen width. For the height of the card view, we set it to 500 points to make it a bit smaller. After making the changes, you should see the card view showing the "London" image.


Trip.swift
TripCardView.swift
ContentView.swift
SwiftUICarousel 〉 SwiftUICarousel 〉 View 〉 ContentView.swift 〉 No Selection
Preview
London
81%

```swift
import SwiftUI

struct ContentView: View {

    @State private var isCardTapped = false

    var body: some View {
        GeometryReader { outerView in
            HStack {
                ForEach(sampleTrips.indices) { index in
                    GeometryReader { innerView in
                        TripCardView(destination: sampleTrips[index].destination,
                            imageName: sampleTrips[index].image, isShowDetails:
                            self.$isCardTapped)
                    }
                    .frame(width: outerView.size.width, height: 500)
                }
            }
            .frame(width: outerView.size.width, height: outerView.size.height)
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 7. The horizontal stack now shows only a single card view

Why the "London" card view? If you place the cursor on the line of the `.frame` modifier, the preview canvas should display something like that shown in figure 8. We have 13 items in the `sampleTrips` array. Since each of the card views has a width equal to the screen width, the horizontal stack view has to expand beyond the screen. It happens that the "London" card view is the center (7th) item of the array. This is why you see the "London" card view.

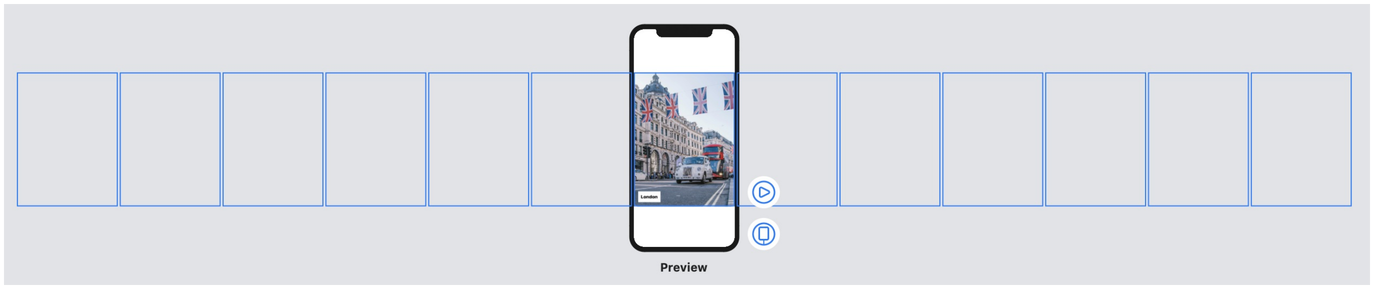


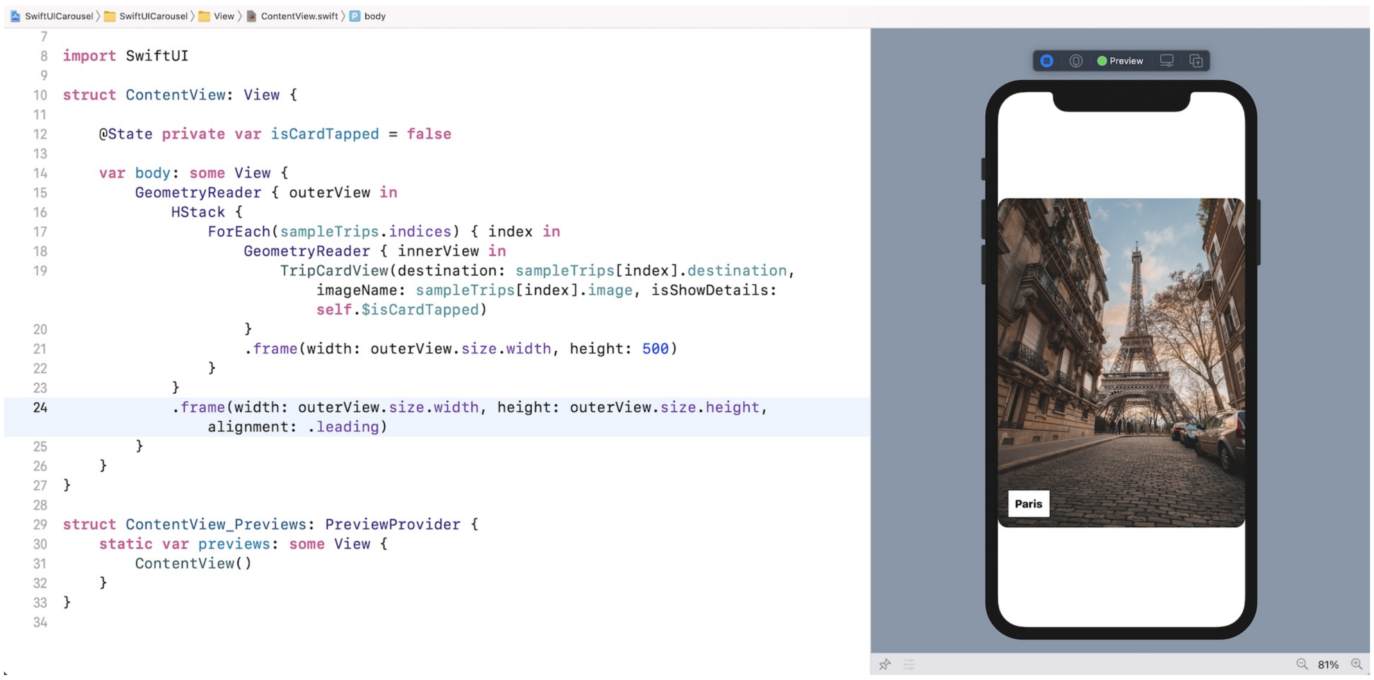
Figure 8. The horizontal stack view has 13 card views

Changing the alignment

So, how can we display the first item of the array instead of the center (7th) item? The trick is to attach a `.frame` modifier to the `HStack` with the alignment set to `.leading` like this:

```
.frame(width: outerView.size.width, height: outerView.size.height, alignment: .lea
ding)
```

The default alignment is set to `.center`. This is why the 7th element of the horizontal view is shown on screen. Once you change the alignment to `.leading`, you should see the first element.


SwiftUICarousel 〉 SwiftUICarousel 〉 View 〉 ContentView.swift 〉 body
Preview
Paris
81%

```
import SwiftUI

struct ContentView: View {

    @State private var isCardTapped = false

    var body: some View {
        GeometryReader { outerView in
            HStack {
                ForEach(sampleTrips.indices) { index in
                    GeometryReader { innerView in
                        TripCardView(destination: sampleTrips[index].destination,
                            imageName: sampleTrips[index].image, isShowDetails:
                            self.$isCardTapped)
                    }
                    .frame(width: outerView.size.width, height: 500)
                }
            }
            .frame(width: outerView.size.width, height: outerView.size.height,
                alignment: .leading)
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 9. The horizontal stack view shows the first card view

If you want to understand how the alignment affects the horizontal stack view, you can change its value to `.center` or `.trailing` to see its effect. Figure 10 shows what the stack view looks likes with different alignment settings.

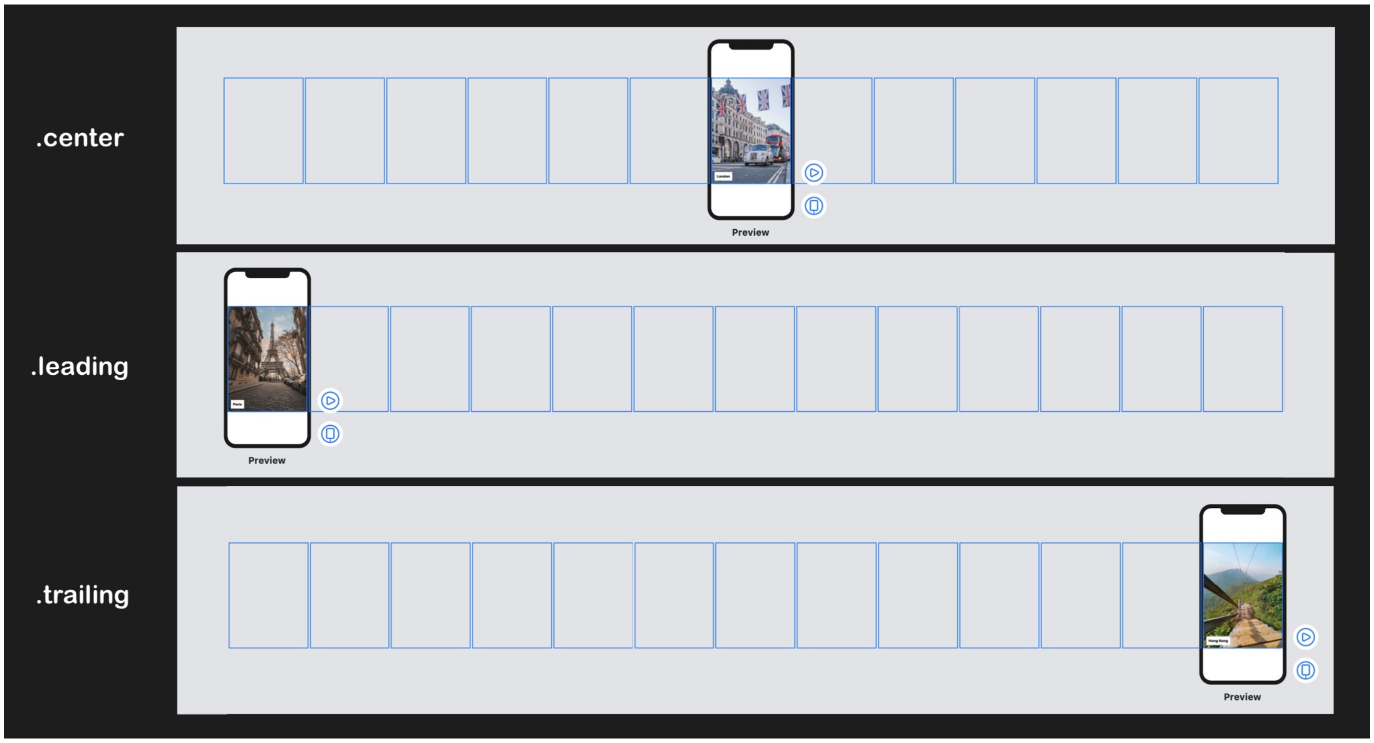


Figure 10. The horizontal stack view with different alignment settings

Did you notice the gap between each of the card views? This is also related to the default setting of `HStack`. To eliminate the spacing, you can update the `HStack` and set its spacing to zero like this:

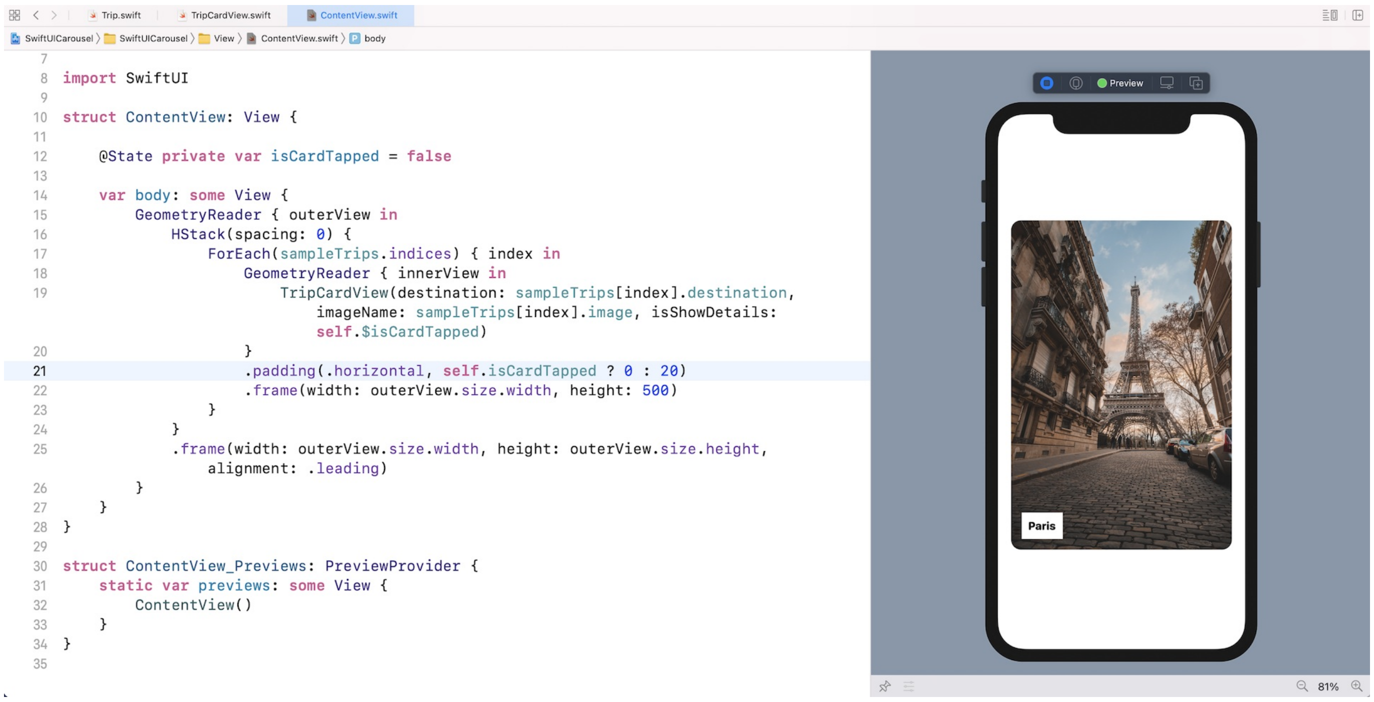
```
HStack(spacing: 0)
```

Adding padding

Optionally, you can add horizontal padding to the image. I think this will make the card view look better. Insert the following line of code and attach it to the GeometryReader that wraps the card view (before `.frame(width: outerView.size.width, height: 500)`):

```
.padding(.horizontal, self.isCardTapped ? 0 : 20)
```

While it's too early for us to talk about the implementation of the detailed view, we added a condition for the padding. The horizontal padding will be removed when the user taps the card view.



```
import SwiftUI

struct ContentView: View {

    @State private var isCardTapped = false

    var body: some View {
        GeometryReader { outerView in
            HStack(spacing: 0) {
                ForEach(sampleTrips.indices) { index in
                    GeometryReader { innerView in
                        TripCardView(destination: sampleTrips[index].destination,
                            imageName: sampleTrips[index].image, isShowDetails:
                            self.$isCardTapped)
                    }
                    .padding(.horizontal, self.isCardTapped ? 0 : 20)
                    .frame(width: outerView.size.width, height: 500)
                }
            }
            .frame(width: outerView.size.width, height: outerView.size.height,
                alignment: .leading)
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 11. Adding the horizontal padding

Moving the HStack Card by Card

Now that we have created a horizontal stack that defaults to show the first card view, the next question is how do we move the stack to display a particular card?

It's just simple math! The card view's width equals the width of the screen. Suppose the screen width is 300 points and we want to display the third card view, we can shift the horizontal stack to the left by 600 points (300 x 2). Figure 12 shows the result.

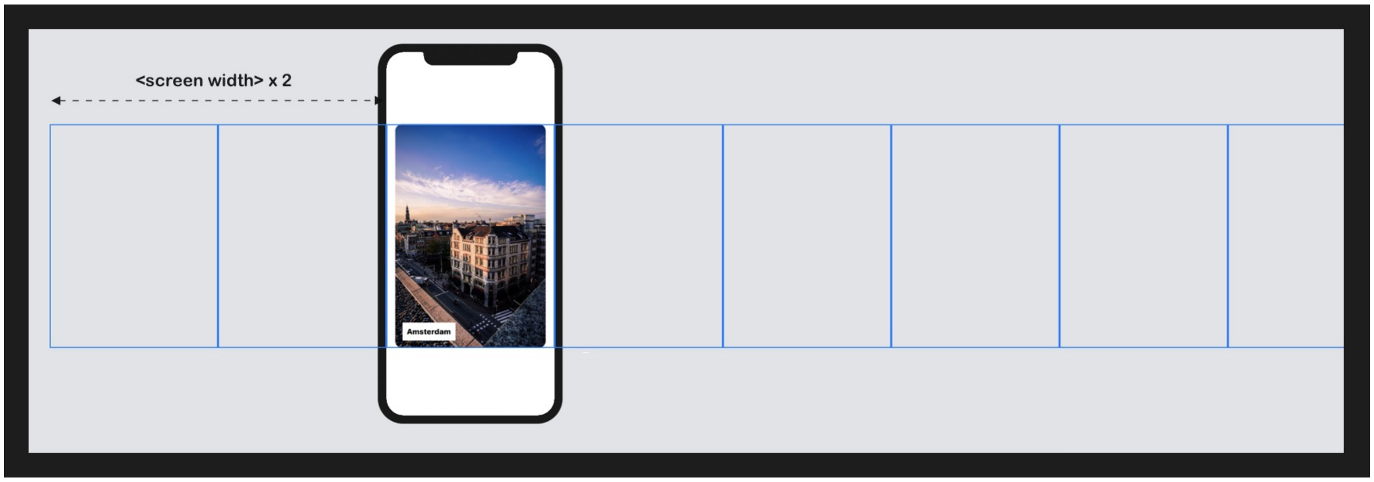


Figure 12. Moving the stack view to the left

To translate the description above into code, we first declare a state variable to keep track of the index of the visible card view:

```
@State private var currentTripIndex = 2
```

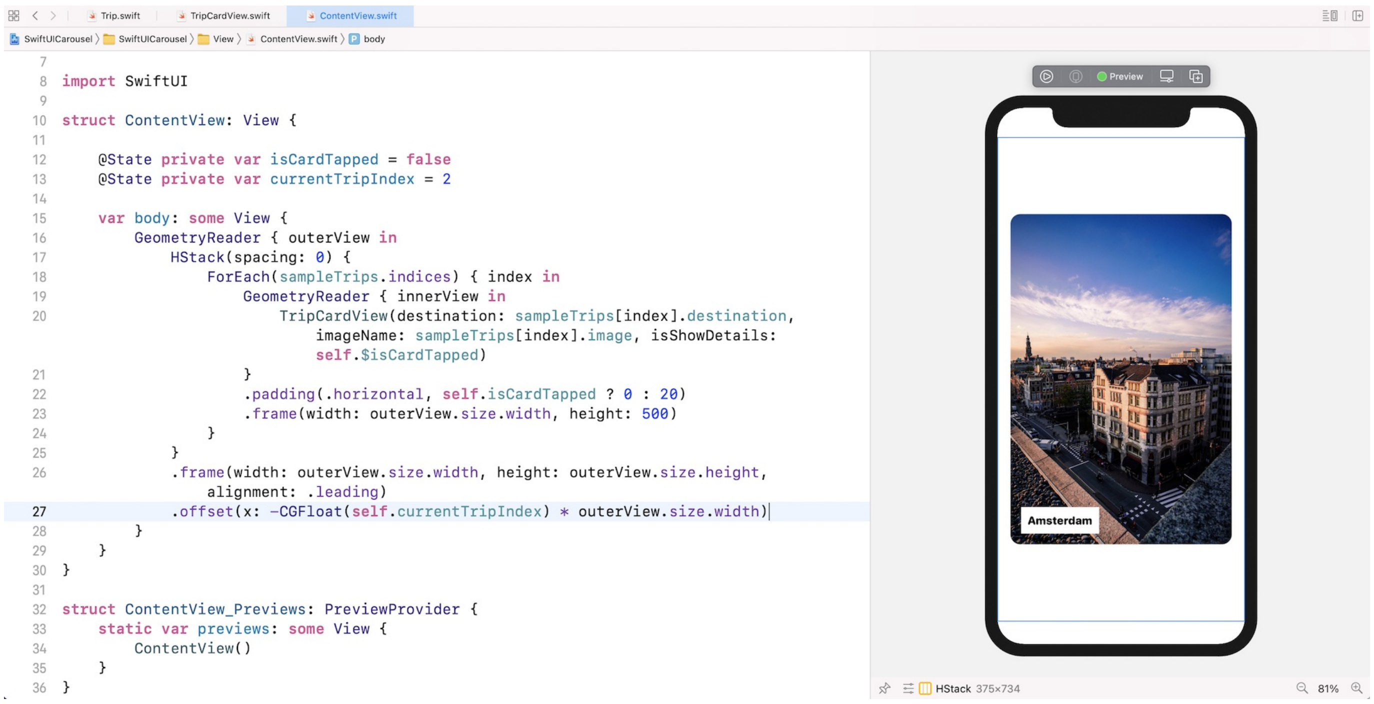
By default, I want to display the third card view. This is why I set the `currentTripIndex` variable to 2. You can change it to other values.

To move the horizontal stack to the left, we can attach the `.offset` modifier to the `HStack` like this:

```
.offset(x: -CGFloat(self.currentTripIndex) * outerView.size.width)
```

The `outerView`'s width is actually the width of the screen. In order to display the third card view, as explained before, we need to move the stack by 2 x screen width. This is why we multiply the `currentTripIndex` with the `outerView`'s width. A negative value for the horizontal offset will shift the stack view to the left.

Once you have made the change, you should see the "Amsterdam" card view in your preview canvas.



```
import SwiftUI

struct ContentView: View {

    @State private var isCardTapped = false
    @State private var currentTripIndex = 2

    var body: some View {
        GeometryReader { outerView in
            HStack(spacing: 0) {
                ForEach(sampleTrips.indices) { index in
                    GeometryReader { innerView in
                        TripCardView(destination: sampleTrips[index].destination,
                            imageName: sampleTrips[index].image, isShowDetails:
                            self.$isCardTapped)
                    }
                    .padding(.horizontal, self.isCardTapped ? 0 : 20)
                    .frame(width: outerView.size.width, height: 500)
                }
            }
            .frame(width: outerView.size.width, height: outerView.size.height,
                alignment: .leading)
            .offset(x: -CGFloat(self.currentTripIndex) * outerView.size.width)
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 13. The stack now shows the Amsterdam card view

Adding the Drag Gesture

With the current implementation, we can change the visible card view by altering the value of `currentTripIndex`. Remember, the horizontal stack doesn't allow users to drag the view. This is what we are going to implement in this section. I assume you already understand how gestures work in SwiftUI. If you don't understand gestures or `@GestureState`, please read chapter 17 first.

The drag gesture of the horizontal stack is expected work like this:

1. The user can tap the image and start dragging.
2. The drag can be in both directions.
3. When the drag's distance exceeds a certain threshold, the horizontal stack will move to the next or previous card view depending on the drag direction.
4. Otherwise, the stack view returns to the original position and displays the current card view.

To translate the description above into code, we first declare a variable to hold the drag offset:

```
@GestureState private var dragOffset: CGFloat = 0
```

Next, we attach the `.gesture` modifier to the `HStack` and initialize a `DragGesture` like this:

```
.gesture(
    !self.isCardTapped ?

    DragGesture()
        .updating(self.$dragOffset, body: { (value, state, transaction) in
            state = value.translation.width
        })
        .onEnded({ (value) in
            let threshold = outerView.size.width * 0.65
            var newIndex = Int(-value.translation.width / threshold) + self.curren
tTripIndex
            newIndex = min(max(newIndex, 0), sampleTrips.count - 1)

            self.currentTripIndex = newIndex
        })

    : nil
)
```

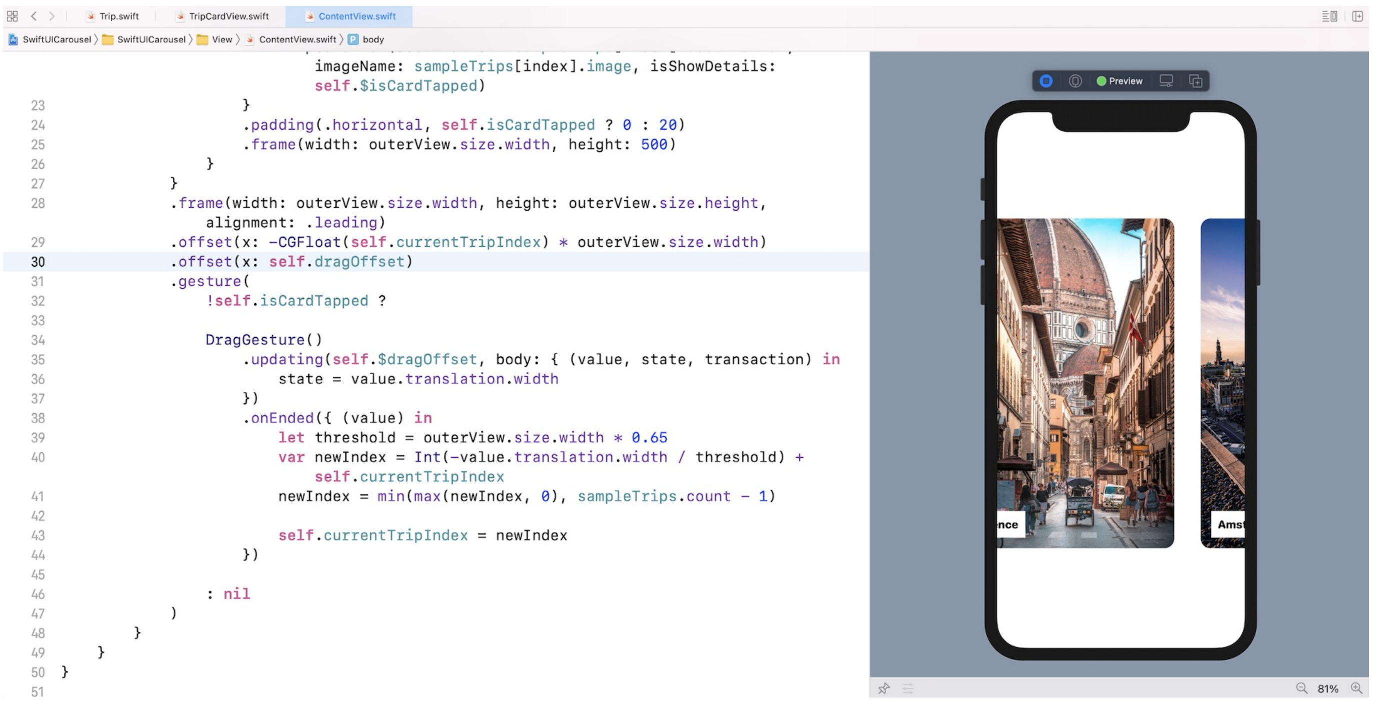
As you drag the horizontal stack, the `updating` function is called. We save the horizontal drag distance to the `dragOffset` variable. When the drag ends, we check if the drag distance exceeds the threshold, which is set to 65% of the screen width, and computes the new index. Once we have the `newIndex` computed, we verify if it is within the range of the `sampleTrips` array. Lastly, we assign the value of `newIndex` to `currentTripIndex`. SwiftUI will then update the UI and display the corresponding card view automatically.

Please take note that we have a condition for enabling the drag gesture. When the card view is tapped, there is no gesture recognizer.

To move the stack view during the drag, we have to make one more change. Attach an additional `.offset` modifier to the `HStack` (right after the previous .offset) like this:

```
.offset(x: self.dragOffset)
```

Here, we update the horizontal offset of the stack view to the drag offset. Now you are ready to test the changes. Run the app in a simulator or in the preview canvas. You should be able to drag the stack view. When your drag exceeds the threshold, the stack view shows you the next trip.


Trip.swift
TripCardView.swift
ContentView.swift
SwiftUICarousel 〉 SwiftUICarousel 〉 View 〉 ContentView.swift 〉 body
Preview
nce
Amst
81%

```
                            imageName: sampleTrips[index].image, isShowDetails:
                            self.$isCardTapped)
                    }
                    .padding(.horizontal, self.isCardTapped ? 0 : 20)
                    .frame(width: outerView.size.width, height: 500)
                }
            }
            .frame(width: outerView.size.width, height: outerView.size.height,
                alignment: .leading)
            .offset(x: -CGFloat(self.currentTripIndex) * outerView.size.width)
            .offset(x: self.dragOffset)
            .gesture(
                !self.isCardTapped ?

                DragGesture()
                    .updating(self.$dragOffset, body: { (value, state, transaction) in
                        state = value.translation.width
                    })
                    .onEnded({ (value) in
                        let threshold = outerView.size.width * 0.65
                        var newIndex = Int(-value.translation.width / threshold) +
                            self.currentTripIndex
                        newIndex = min(max(newIndex, 0), sampleTrips.count - 1)

                        self.currentTripIndex = newIndex
                    })

                : nil
            )
        }
    }
}
```

Figure 14. Dragging the horizontal stack view

Animating the Card Transition

To improve the user experience, I want to add a nice animation when the app moves from one card view to another. First, modify the following line of code from:

```
.frame(width: outerView.size.width, height: 500)
```

To:

```
.frame(width: outerView.size.width, height: self.currentTripIndex == index ? (self
.isCardTapped ? outerView.size.height : 450) : 400)
```

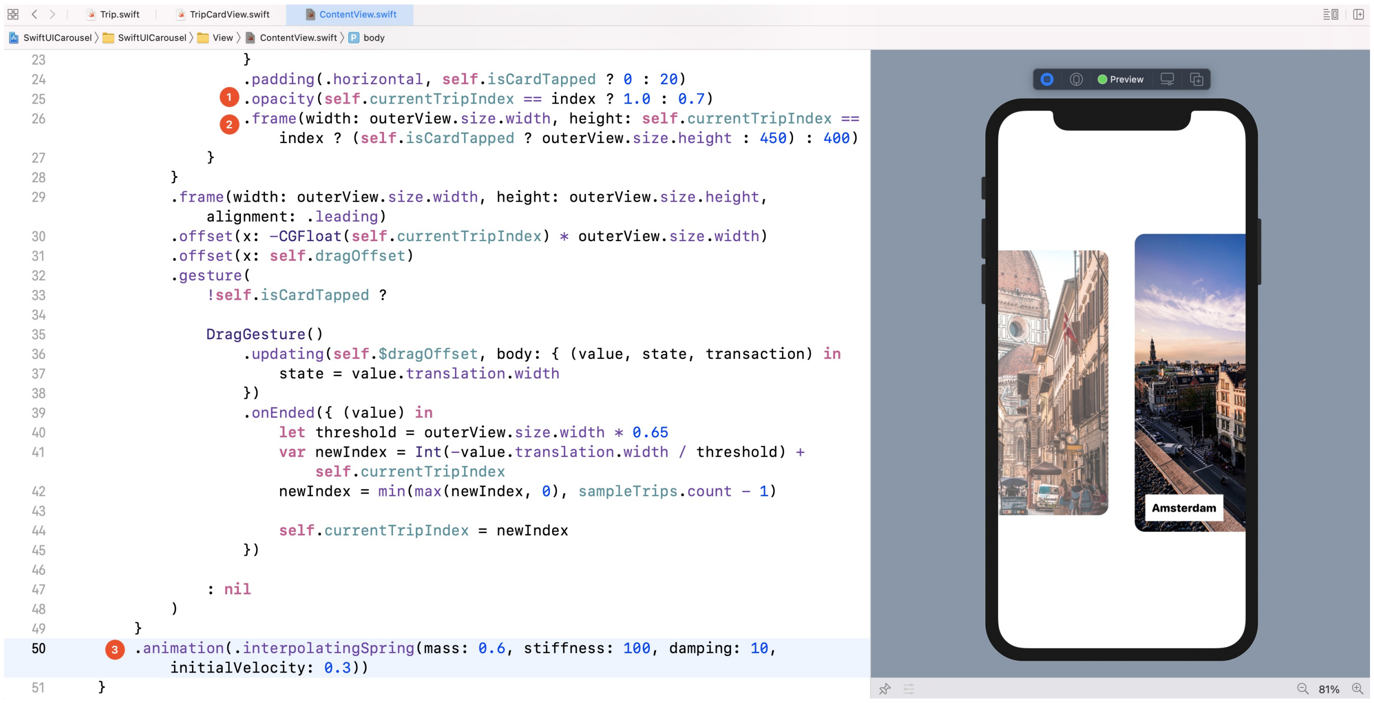
By updating the code, we make the visible card view a little bit larger than the rest. On top of that, attach the `.opacity` modifier to the card view like this:

```
.opacity(self.currentTripIndex == index ? 1.0 : 0.7)
```

Other than the card view's height, we also want to set a different opacity value for the visible and invisible card views. All these changes are not animated yet. Now insert the following line of code to the outer view's GeometryReader:

```
.animation(.interpolatingSpring(mass: 0.6, stiffness: 100, damping: 10, initialVel
ocity: 0.3))
```

SwiftUI will then animate the size and opacity changes of the card views automatically. Run the app in the preview canvas to test out the changes. This is how we implement a scroll view with HStack and add paging support.


Trip.swift
TripCardView.swift
ContentView.swift
SwiftUICarousel 〉 SwiftUICarousel 〉 View 〉 ContentView.swift 〉 body
Preview
Amsterdam
81%

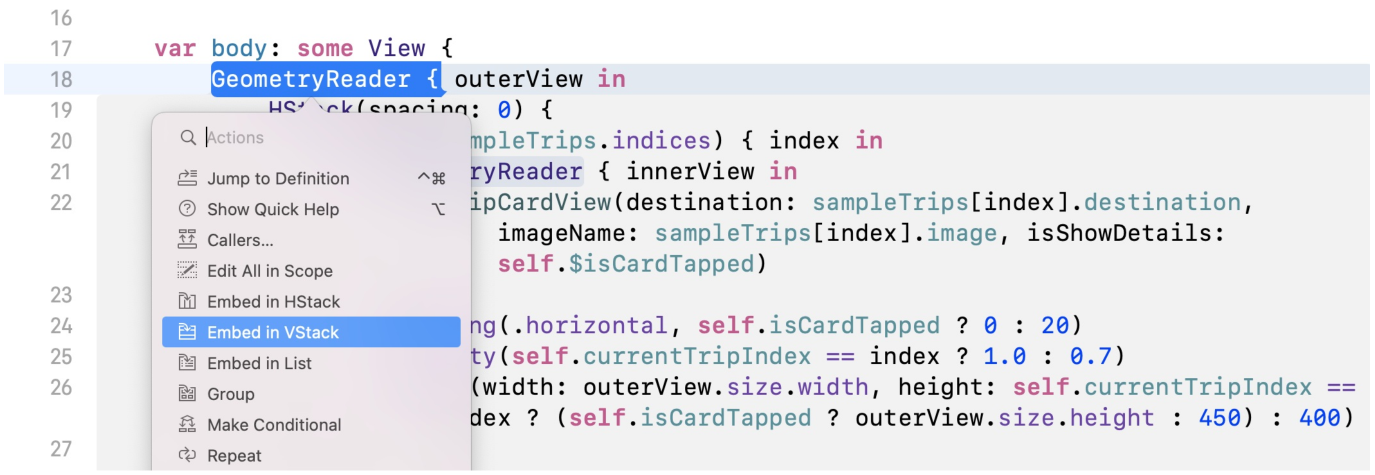
```
                    }
                    .padding(.horizontal, self.isCardTapped ? 0 : 20)
                    .opacity(self.currentTripIndex == index ? 1.0 : 0.7)
                    .frame(width: outerView.size.width, height: self.currentTripIndex ==
                        index ? (self.isCardTapped ? outerView.size.height : 450) : 400)
                }
            }
            .frame(width: outerView.size.width, height: outerView.size.height,
                alignment: .leading)
            .offset(x: -CGFloat(self.currentTripIndex) * outerView.size.width)
            .offset(x: self.dragOffset)
            .gesture(
                !self.isCardTapped ?

                DragGesture()
                    .updating(self.$dragOffset, body: { (value, state, transaction) in
                        state = value.translation.width
                    })
                    .onEnded({ (value) in
                        let threshold = outerView.size.width * 0.65
                        var newIndex = Int(-value.translation.width / threshold) +
                            self.currentTripIndex
                        newIndex = min(max(newIndex, 0), sampleTrips.count - 1)

                        self.currentTripIndex = newIndex
                    })

                : nil
            )
        }
        .animation(.interpolatingSpring(mass: 0.6, stiffness: 100, damping: 10,
            initialVelocity: 0.3))
    }
```

Figure 15. Adding the card transition animation

Adding the Title

Now that we have built the image carousel, wouldn't it be great if we implement the detail view to make the demo app more complete? Let's start by adding a title for the app.

Command-click the `GeometryReader` of the outer view and choose *embed in VStack*. Actually, I'm not going to use a vertical stack to layout the title. Instead, we will use a ZStack. Therefore, change `VStack` to `ZStack`.



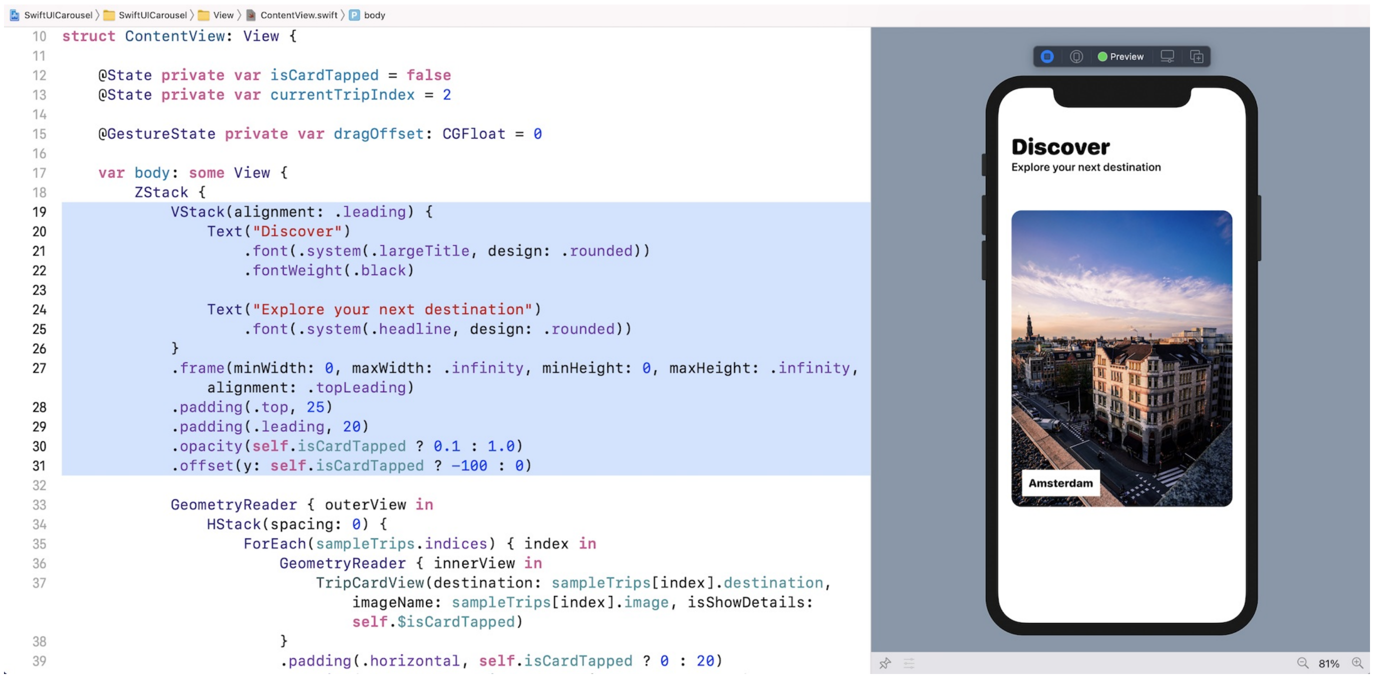
```
var body: some View {
    GeometryReader { outerView in
        HS  ck(spacing: 0) {
                mpleTrips.indices) { index in
                ryReader { innerView in
                ipCardView(destination: sampleTrips[index].destination,
                  imageName: sampleTrips[index].image, isShowDetails:
                  self.$isCardTapped)

                ng(.horizontal, self.isCardTapped ? 0 : 20)
                ty(self.currentTripIndex == index ? 1.0 : 0.7)
                (width: outerView.size.width, height: self.currentTripIndex ==
                dex ? (self.isCardTapped ? outerView.size.height : 450) : 400)
```

Figure 16. Embed the outer view in a VStack

Next, insert the following code at the beginning of `ZStack`:

```
VStack(alignment: .leading) {
    Text("Discover")
        .font(.system(.largeTitle, design: .rounded))
        .fontWeight(.black)

    Text("Explore your next destination")
        .font(.system(.headline, design: .rounded))
}
.frame(minWidth: 0, maxWidth: .infinity, minHeight: 0, maxHeight: .infinity, align
ment: .topLeading)
.padding(.top, 25)
.padding(.leading, 20)
.opacity(self.isCardTapped ? 0.1 : 1.0)
.offset(y: self.isCardTapped ? -100 : 0)
```

The code above is self explanatory but I'd like to highlight two lines of code. Both `.opacity` and `.offset` are optional. The purpose of the `.opacity` modifier is to hide the title when the card is tapped. The change to the vertical offset will add a nice touch to the user experience.

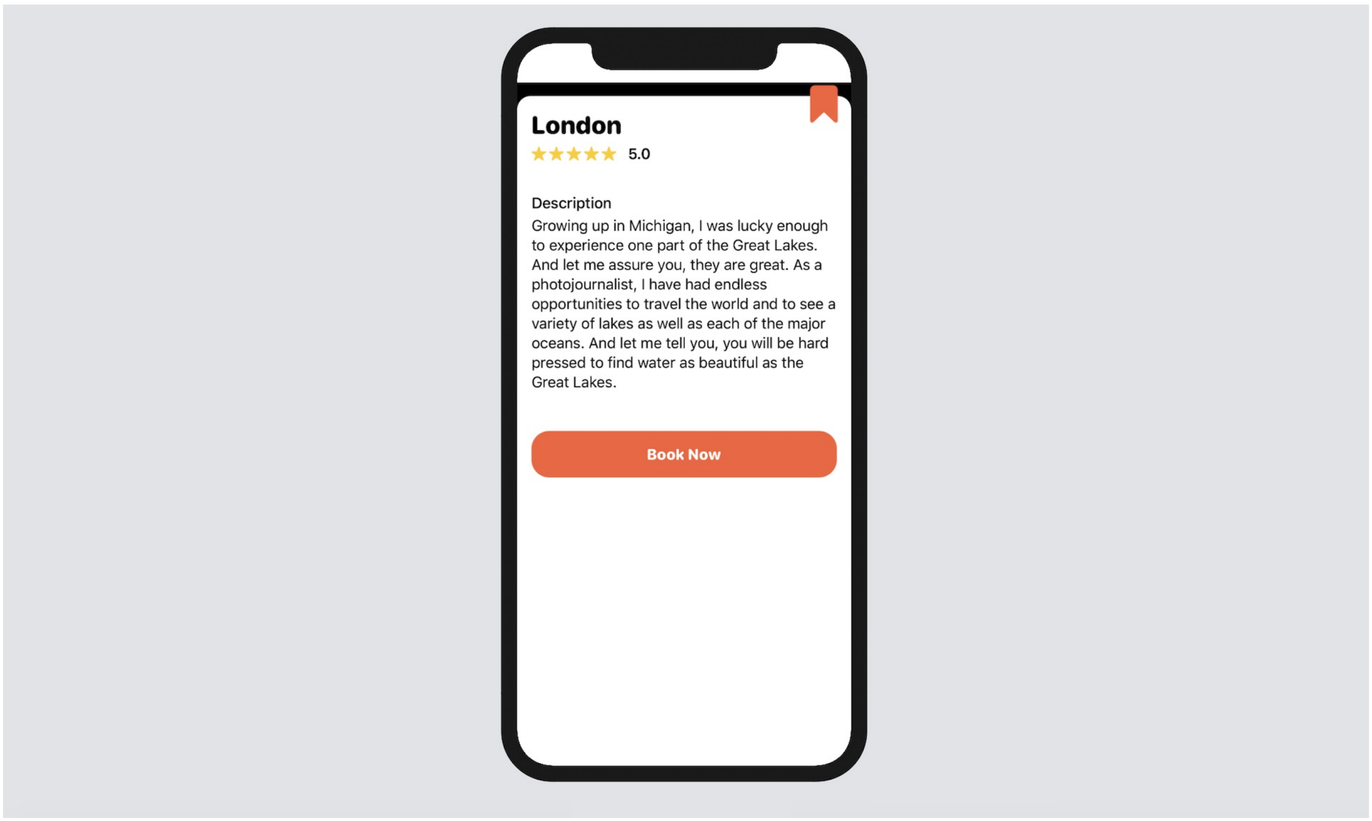

SwiftUICarousel 〉 SwiftUICarousel 〉 View 〉 ContentView.swift 〉 body
Preview
Discover
Explore your next destination
Amsterdam
81%

```
struct ContentView: View {

    @State private var isCardTapped = false
    @State private var currentTripIndex = 2

    @GestureState private var dragOffset: CGFloat = 0

    var body: some View {
        ZStack {
            VStack(alignment: .leading) {
                Text("Discover")
                    .font(.system(.largeTitle, design: .rounded))
                    .fontWeight(.black)

                Text("Explore your next destination")
                    .font(.system(.headline, design: .rounded))
            }
            .frame(minWidth: 0, maxWidth: .infinity, minHeight: 0, maxHeight: .infinity,
                alignment: .topLeading)
            .padding(.top, 25)
            .padding(.leading, 20)
            .opacity(self.isCardTapped ? 0.1 : 1.0)
            .offset(y: self.isCardTapped ? -100 : 0)

            GeometryReader { outerView in
                HStack(spacing: 0) {
                    ForEach(sampleTrips.indices) { index in
                        GeometryReader { innerView in
                            TripCardView(destination: sampleTrips[index].destination,
                                imageName: sampleTrips[index].image, isShowDetails:
                                self.$isCardTapped)
                        }
                        .padding(.horizontal, self.isCardTapped ? 0 : 20)
```

Figure 17. Adding the title bar

Exercise: Working on the Detail View

Let's begin the implementation of the detail view with an exercise. I assume you have some experience with SwiftUI and should be able to create the detail view shown in figure 18. You can create a separate file named `TripDetailView.swift` and write the code there.


London

★★★★★ 5.0

Description

Growing up in Michigan, I was lucky enough to experience one part of the Great Lakes. And let me assure you, they are great. As a photojournalist, I have had endless opportunities to travel the world and to see a variety of lakes as well as each of the major oceans. And let me tell you, you will be hard pressed to find water as beautiful as the Great Lakes.

Figure 18. The detail view of a trip

To keep things simple, the rating and description are just dummy data. The same goes for the *Book Now* button, which is not functional. This detail view only takes in a destination like this:

```swift
struct TripDetailView: View {
    let destination: String

    var body: some View {
        .
        .
        .
    }
}
```

Please take some time to create the detail view. I will walk you through my solution in a later section.

Implementing the Trip Detail View

Were you able to develop the detail view? I hope you tried to complete the exercise. Let me show you my solution. First, create a new file named `TripDetailView.swift` using the *SwiftUI View* template.

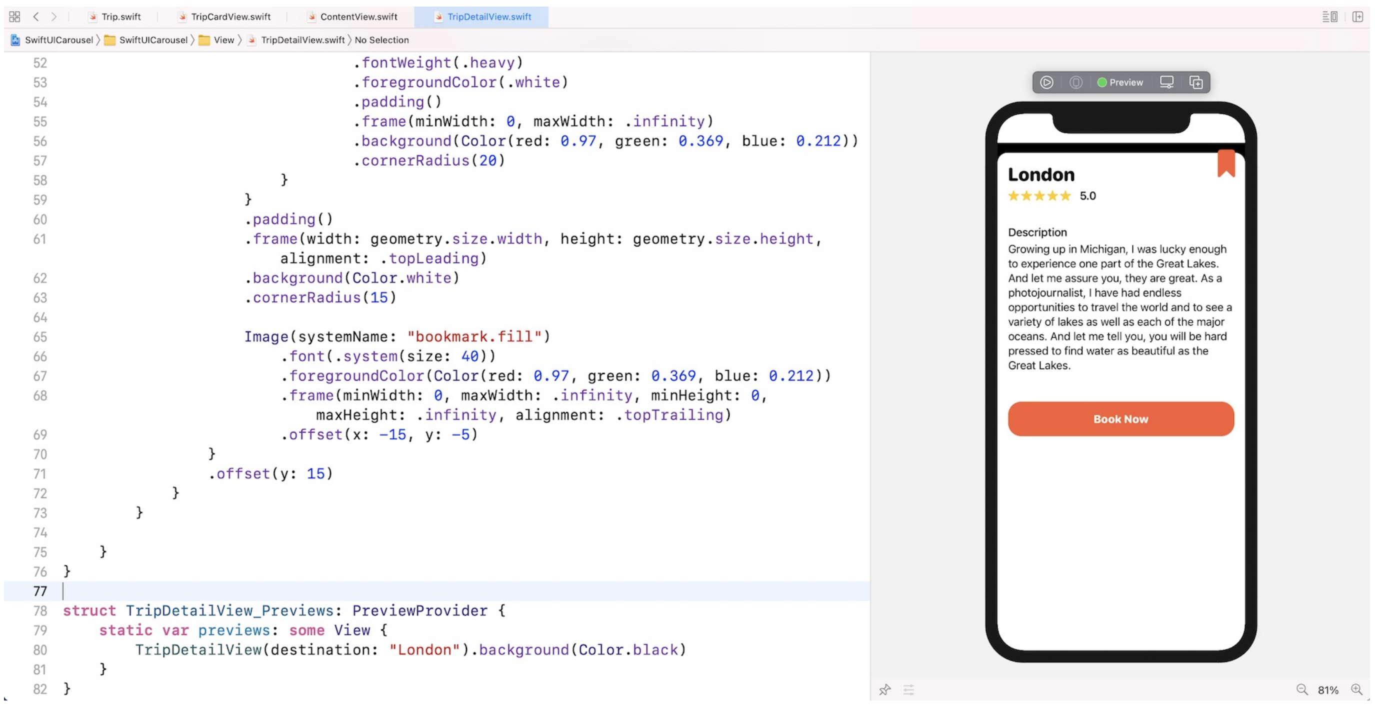
Next, replace the `TripDetailView` struct like this:

```
struct TripDetailView: View {
    let destination: String

    var body: some View {
        GeometryReader { geometry in
            ScrollView {
                ZStack {
                    VStack(alignment: .leading, spacing: 5) {

                        VStack(alignment: .leading, spacing: 5) {
                            Text(self.destination)
                                .font(.system(.title, design: .rounded))
                                .fontWeight(.heavy)

                            HStack(spacing: 3) {
                                ForEach(1...5, id: \.self) { _ in
                                    Image(systemName: "star.fill")
                                        .foregroundColor(.yellow)
                                        .font(.system(size: 15))
                                }

                                Text("5.0")
                                    .font(.system(.headline))
                                    .padding(.leading, 10)
                            }

                        }
                        .padding(.bottom, 30)


                        Text("Description")
                            .font(.system(.headline))
                            .fontWeight(.medium)
                        Text("Growing up in Michigan, I was lucky enough to experi
ence one part of the Great Lakes. And let me assure you, they are great. As a phot
ojournalist, I have had endless opportunities to travel the world and to see a var
iety of lakes as well as each of the major oceans. And let me tell you, you will b
e hard pressed to find water as beautiful as the Great Lakes.")
                            .padding(.bottom, 40)

                        Button(action: {
                            // tap me
                        }) {
                            Text("Book Now")
                                .font(.system(.headline, design: .rounded))
                                .fontWeight(.heavy)
                                .foregroundColor(.white)
                                .padding()
                                .frame(minWidth: 0, maxWidth: .infinity)
                                .background(Color(red: 0.97, green: 0.369, blue: 0
.212))
                                .cornerRadius(20)
                        }
                    }
                    .padding()
                    .frame(width: geometry.size.width, height: geometry.size.heigh
t, alignment: .topLeading)
                    .background(Color.white)
                    .cornerRadius(15)

                    Image(systemName: "bookmark.fill")
                        .font(.system(size: 40))
                        .foregroundColor(Color(red: 0.97, green: 0.369, blue: 0.212
))
                        .frame(minWidth: 0, maxWidth: .infinity, minHeight: 0, max
Height: .infinity, alignment: .topTrailing)
                        .offset(x: -15, y: -5)
                }
                .offset(y: 15)
            }
        }

    }
}
```

Basically, we embed the whole content in a scroll view. Inside the scroll view, we use a ZStack to layout the content and the bookmark image. Since the `TripDetailView` requires a valid destination in order to work properly, you need to update the preview code like this:

```
struct TripDetailView_Previews: PreviewProvider {
    static var previews: some View {
        TripDetailView(destination: "London").background(Color.black)
    }
}
```

I also changed the background color to *black*, so that we can see the rounded corners of the detail view.


Trip.swift
TripCardView.swift
ContentView.swift
TripDetailView.swift
SwiftUICarousel 〉 SwiftUICarousel 〉 View 〉 TripDetailView.swift 〉 No Selection
Preview
81%

```
                                .fontWeight(.heavy)
                                .foregroundColor(.white)
                                .padding()
                                .frame(minWidth: 0, maxWidth: .infinity)
                                .background(Color(red: 0.97, green: 0.369, blue: 0.212))
                                .cornerRadius(20)
                        }
                    }
                    .padding()
                    .frame(width: geometry.size.width, height: geometry.size.height,
                        alignment: .topLeading)
                    .background(Color.white)
                    .cornerRadius(15)

                    Image(systemName: "bookmark.fill")
                        .font(.system(size: 40))
                        .foregroundColor(Color(red: 0.97, green: 0.369, blue: 0.212))
                        .frame(minWidth: 0, maxWidth: .infinity, minHeight: 0,
                            maxHeight: .infinity, alignment: .topTrailing)
                        .offset(x: -15, y: -5)
                }
                .offset(y: 15)
            }
        }

    }
}

struct TripDetailView_Previews: PreviewProvider {
    static var previews: some View {
        TripDetailView(destination: "London").background(Color.black)
    }
}
```

London
★★★★★ 5.0

Description
Growing up in Michigan, I was lucky enough to experience one part of the Great Lakes. And let me assure you, they are great. As a photojournalist, I have had endless opportunities to travel the world and to see a variety of lakes as well as each of the major oceans. And let me tell you, you will be hard pressed to find water as beautiful as the Great Lakes.

Book Now

Figure 19. Previewing the detail view

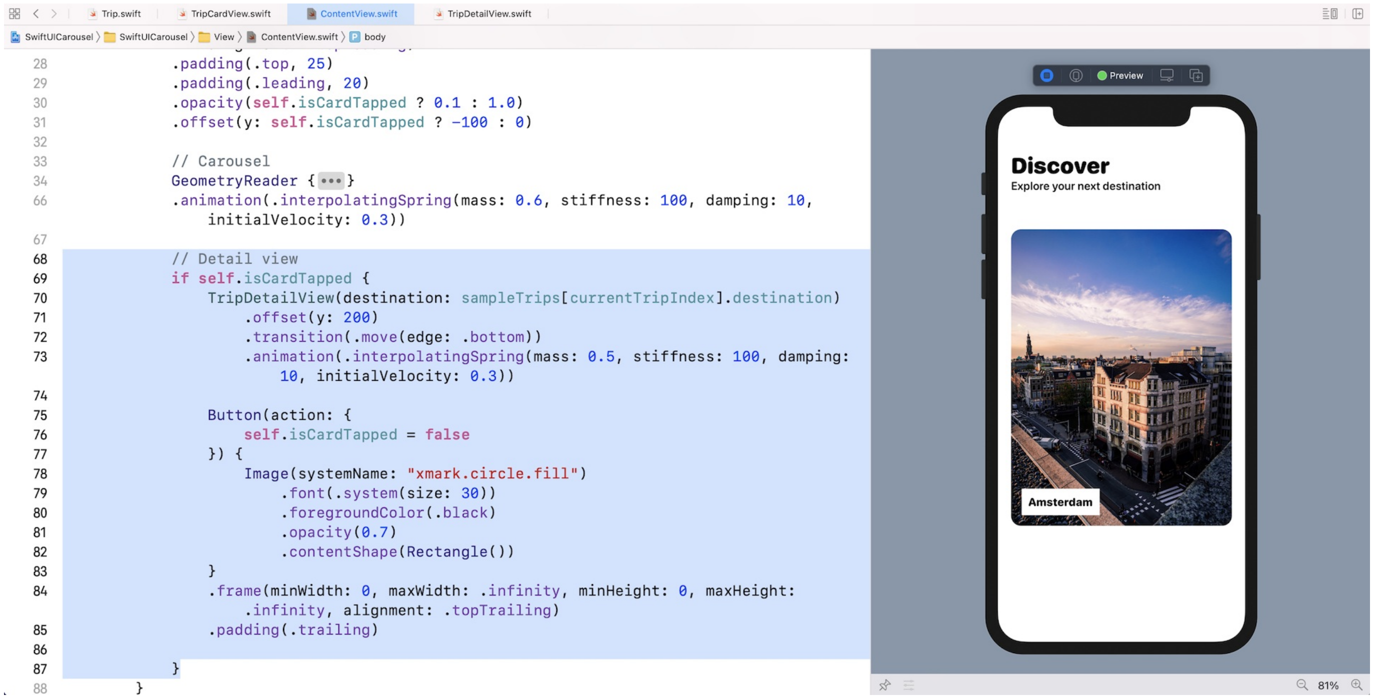
Bringing up the Detail View

Now let's go back to `ContentView.swift` to bring up the detail view. When a user taps a card view, we will bring up the detail view with an animated transition. Since the content view has a ZStack which wraps its core content, it's very easy for us to integrate with the detail view.

Insert the following code snippet in the `ZStack` :

```
if self.isCardTapped {
    TripDetailView(destination: sampleTrips[currentTripIndex].destination)
        .offset(y: 200)
        .transition(.move(edge: .bottom))
        .animation(.interpolatingSpring(mass: 0.5, stiffness: 100, damping: 10, in
itialVelocity: 0.3))

    Button(action: {
        self.isCardTapped = false
    }) {
        Image(systemName: "xmark.circle.fill")
            .font(.system(size: 30))
            .foregroundColor(.black)
            .opacity(0.7)
            .contentShape(Rectangle())
    }
    .frame(minWidth: 0, maxWidth: .infinity, minHeight: 0, maxHeight: .infinity, a
lignment: .topTrailing)
    .padding(.trailing)

}
```

The `TripDetailView` is only brought up when the card view is tapped. It's expected that the detail view will appear from the bottom of the screen and move upward with an animation. This is why we attach both the `.transition` and `.animation` modifiers to the detail view. To let users dismiss the detail view, we also add a close button, which appears at the top-right corner of the screen. In case you are not sure where to insert the code above, please refer to figure 20.


Trip.swift
TripCardView.swift
ContentView.swift
TripDetailView.swift
SwiftUICarousel 〉 SwiftUICarousel 〉 View 〉 ContentView.swift 〉 body
Preview
Discover
Explore your next destination
Amsterdam
81%

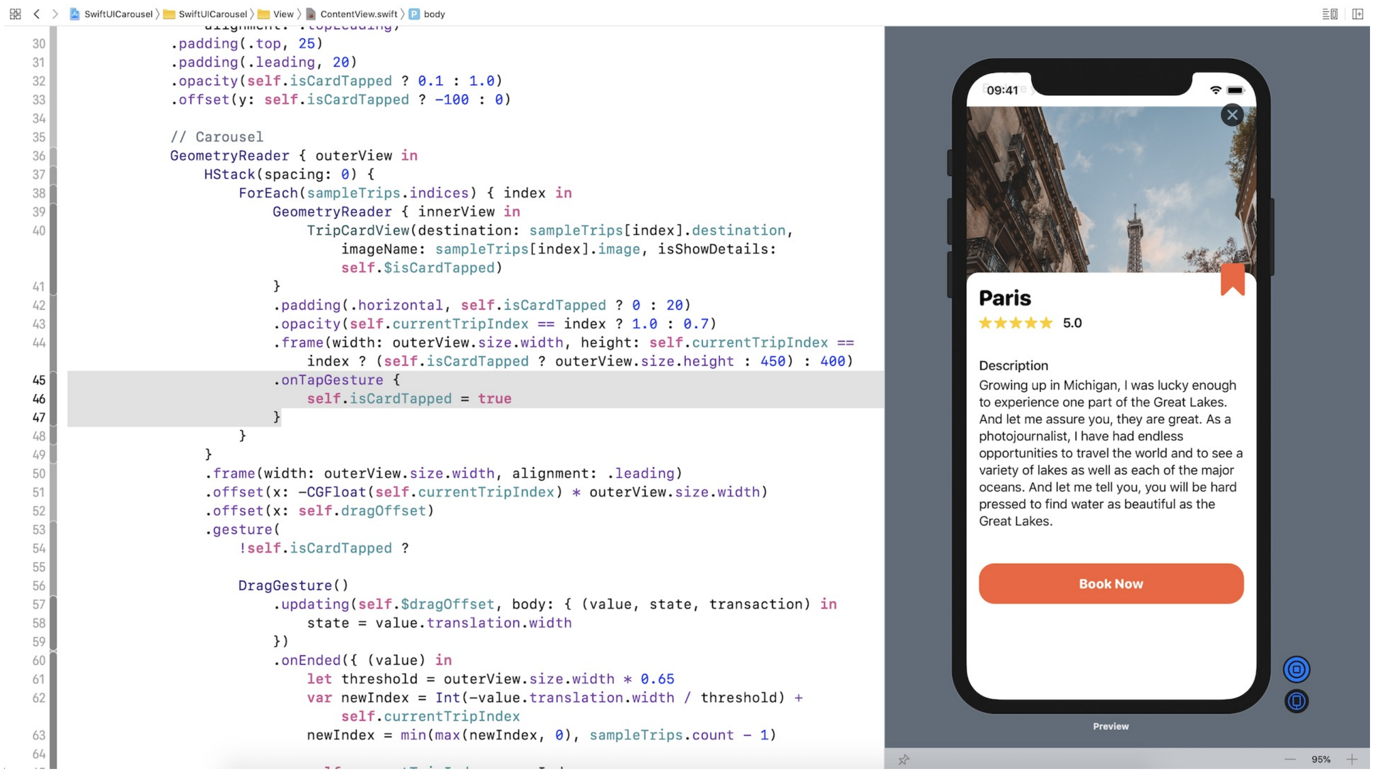
```
        .padding(.top, 25)
        .padding(.leading, 20)
        .opacity(self.isCardTapped ? 0.1 : 1.0)
        .offset(y: self.isCardTapped ? -100 : 0)

        // Carousel
        GeometryReader {•••}
        .animation(.interpolatingSpring(mass: 0.6, stiffness: 100, damping: 10,
            initialVelocity: 0.3))

        // Detail view
        if self.isCardTapped {
            TripDetailView(destination: sampleTrips[currentTripIndex].destination)
                .offset(y: 200)
                .transition(.move(edge: .bottom))
                .animation(.interpolatingSpring(mass: 0.5, stiffness: 100, damping:
                    10, initialVelocity: 0.3))

            Button(action: {
                self.isCardTapped = false
            }) {
                Image(systemName: "xmark.circle.fill")
                    .font(.system(size: 30))
                    .foregroundColor(.black)
                    .opacity(0.7)
                    .contentShape(Rectangle())
            }
            .frame(minWidth: 0, maxWidth: .infinity, minHeight: 0, maxHeight:
                .infinity, alignment: .topTrailing)
            .padding(.trailing)

        }
    }
```

Figure 20. The code snippet for bringing up the detail view

The code won't work yet because we haven't captured the tap gesture. Thus, attach the `.onTapGesture` function to the card view like this:

```
.onTapGesture {
    self.isCardTapped = true
}
```

When someone taps the card view, we simply change the `isCardTapped` state variable to `true`. Run the app and tap any of the card views. The app should bring up the detail view.


SwiftUICarousel 〉 SwiftUICarousel 〉 View 〉 ContentView.swift 〉 body

```swift
            .padding(.top, 25)
            .padding(.leading, 20)
            .opacity(self.isCardTapped ? 0.1 : 1.0)
            .offset(y: self.isCardTapped ? -100 : 0)

            // Carousel
            GeometryReader { outerView in
                HStack(spacing: 0) {
                    ForEach(sampleTrips.indices) { index in
                        GeometryReader { innerView in
                            TripCardView(destination: sampleTrips[index].destination,
                                imageName: sampleTrips[index].image, isShowDetails:
                                self.$isCardTapped)
                        }
                        .padding(.horizontal, self.isCardTapped ? 0 : 20)
                        .opacity(self.currentTripIndex == index ? 1.0 : 0.7)
                        .frame(width: outerView.size.width, height: self.currentTripIndex ==
                            index ? (self.isCardTapped ? outerView.size.height : 450) : 400)
                        .onTapGesture {
                            self.isCardTapped = true
                        }
                    }
                }
                .frame(width: outerView.size.width, alignment: .leading)
                .offset(x: -CGFloat(self.currentTripIndex) * outerView.size.width)
                .offset(x: self.dragOffset)
                .gesture(
                    !self.isCardTapped ?

                    DragGesture()
                        .updating(self.$dragOffset, body: { (value, state, transaction) in
                            state = value.translation.width
                        })
                        .onEnded({ (value) in
                            let threshold = outerView.size.width * 0.65
                            var newIndex = Int(-value.translation.width / threshold) +
                                self.currentTripIndex
                            newIndex = min(max(newIndex, 0), sampleTrips.count - 1)
```

Description
Growing up in Michigan, I was lucky enough to experience one part of the Great Lakes. And let me assure you, they are great. As a photojournalist, I have had endless opportunities to travel the world and to see a variety of lakes as well as each of the major oceans. And let me tell you, you will be hard pressed to find water as beautiful as the Great Lakes.

Figure 21. The code snippet for bringing up the detail view

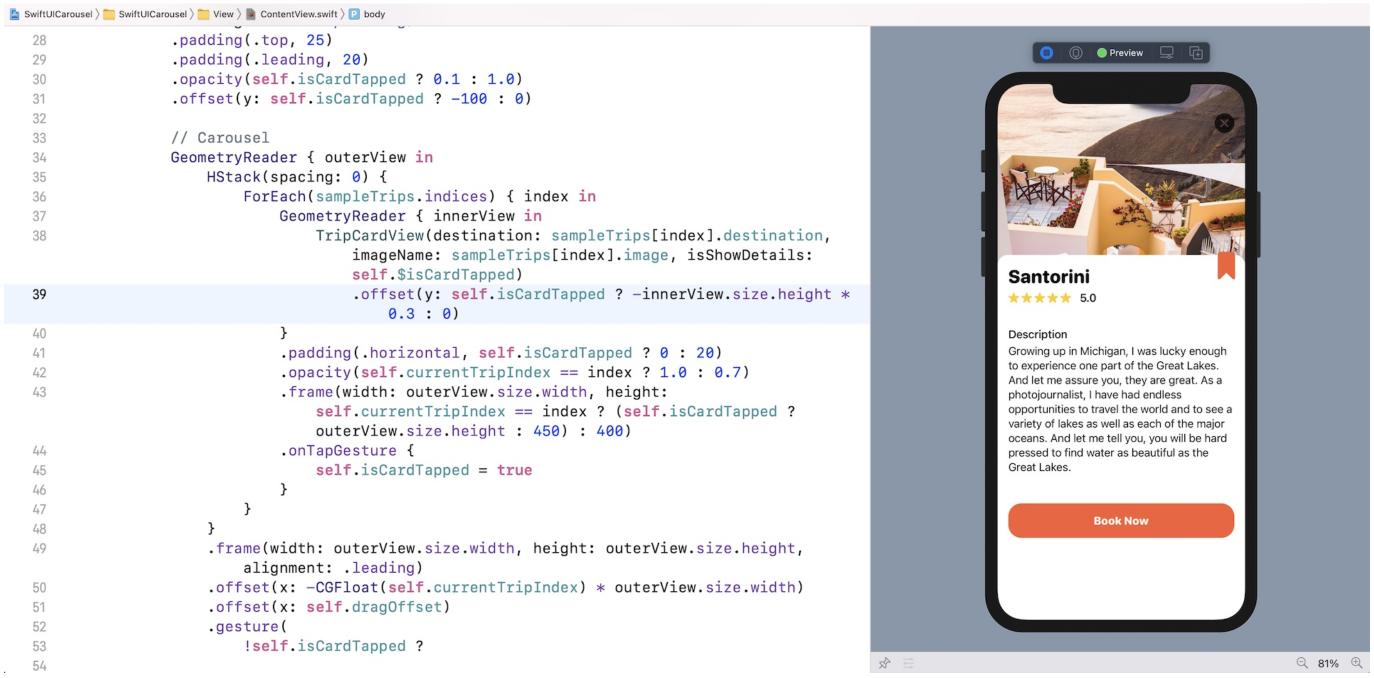
The detail view works! However, the animation doesn't work well. When the detail view is brought up, the card view grows a little bit bigger, which is achieved by the following line of code:

```swift
.frame(width: outerView.size.width, height: self.currentTripIndex == index ? (self
.isCardTapped ? outerView.size.height : 450) : 400)
```

To make the animation look better, let's move the image upward when the detail view appears. Attach the `.offset` modifier to `TripCardView`:

```swift
.offset(y: self.isCardTapped ? -innerView.size.height * 0.3 : 0)
```

I set the vertical offset to 30% of the card view's height. You are free to change the value. Now run the app again and you should see a more slick animation.


SwiftUICarousel 〉 SwiftUICarousel 〉 View 〉 ContentView.swift 〉 body
Preview
81%

```
.padding(.top, 25)
.padding(.leading, 20)
.opacity(self.isCardTapped ? 0.1 : 1.0)
.offset(y: self.isCardTapped ? -100 : 0)

// Carousel
GeometryReader { outerView in
    HStack(spacing: 0) {
        ForEach(sampleTrips.indices) { index in
            GeometryReader { innerView in
                TripCardView(destination: sampleTrips[index].destination,
                    imageName: sampleTrips[index].image, isShowDetails:
                    self.$isCardTapped)
                    .offset(y: self.isCardTapped ? -innerView.size.height *
                        0.3 : 0)
            }
            .padding(.horizontal, self.isCardTapped ? 0 : 20)
            .opacity(self.currentTripIndex == index ? 1.0 : 0.7)
            .frame(width: outerView.size.width, height:
                self.currentTripIndex == index ? (self.isCardTapped ?
                outerView.size.height : 450) : 400)
            .onTapGesture {
                self.isCardTapped = true
            }
        }
    }
    .frame(width: outerView.size.width, height: outerView.size.height,
        alignment: .leading)
    .offset(x: -CGFloat(self.currentTripIndex) * outerView.size.width)
    .offset(x: self.dragOffset)
    .gesture(
        !self.isCardTapped ?
```

Santorini

★★★★★ 5.0

Description

Growing up in Michigan, I was lucky enough to experience one part of the Great Lakes. And let me assure you, they are great. As a photojournalist, I have had endless opportunities to travel the world and to see a variety of lakes as well as each of the major oceans. And let me tell you, you will be hard pressed to find water as beautiful as the Great Lakes.

Book Now

Figure 22. Adding an offset modifier to TripCardView

Summary

Great! You've built a custom scroll view with paging support and learned how to bring up a detail view with animated transition. This technique is not limited to to an image carousel. In fact, you can modify the code to create a set of onboarding screens. I hope you love what you learned in this chapter and will apply it to your next app project.

For reference, you can download the complete carousel project here:

- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUICarousel.zip)

Chapter 28
Building an Expandable List View Using OutlineGroup

SwiftUI list is very similar to UITableView in UIKit. In the first release of SwiftUI, Apple's engineers made list view construction a breeze. You do not need to create a prototype cell and there is no delegate/data source protocol. With just a few lines of code, you can build a list view with custom cells. In iOS 14, Apple continued to improve the `List` view and introduced several new features. In this chapter, we will show you how to build an expandable list / outline view and explore the inset grouped list style.

The Demo App

First, let's take a look at the final deliverable. I'm a big fan of La Marzocco, so I used the navigation menu on its website as an example. The list view below shows an outline of the menu. Users can tap the disclosure button to expand the list.

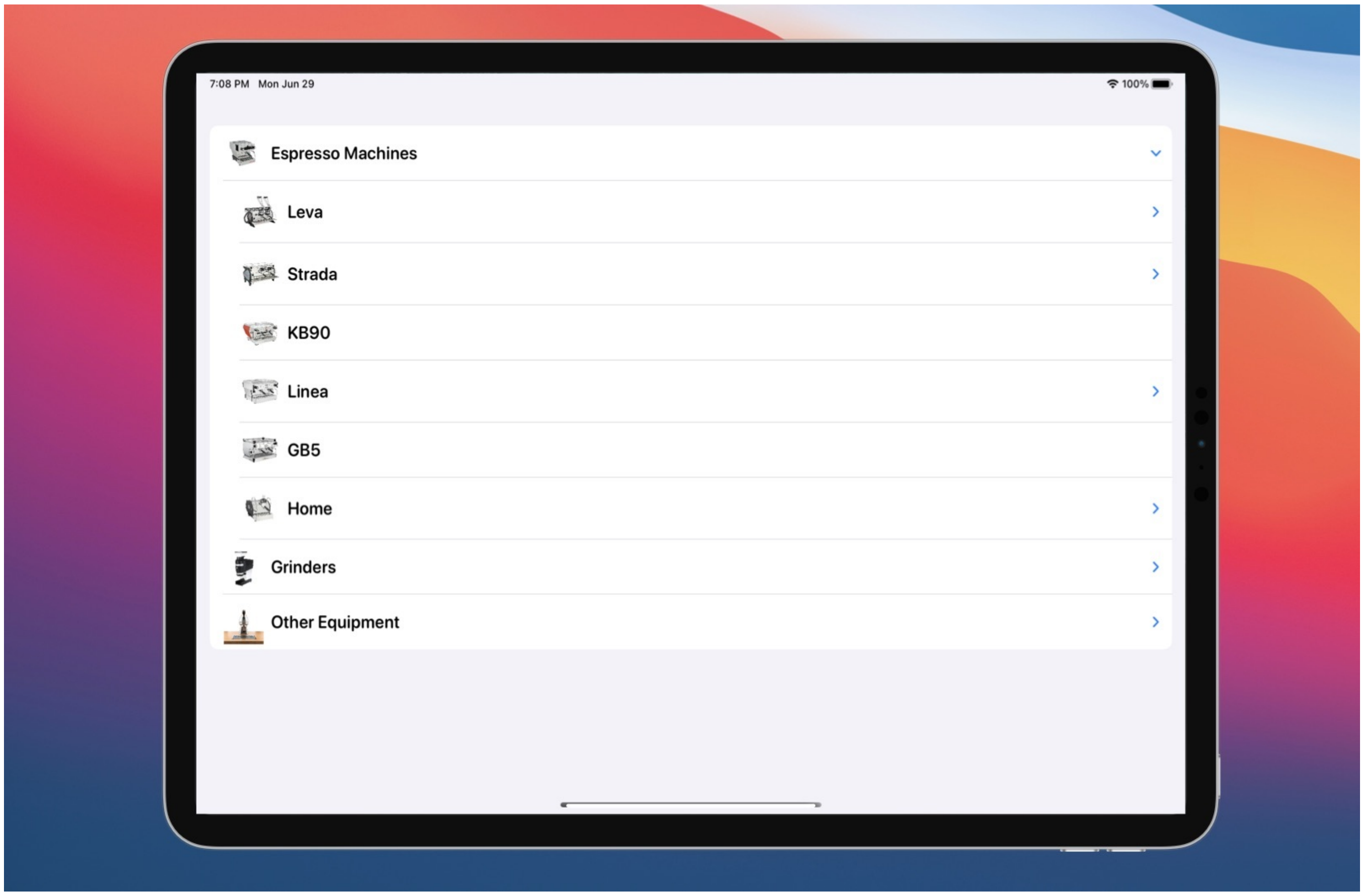


Figure 1. The expandable list view

Of course, you can build this outline view using your own implementation. Starting from iOS 14, Apple made it simpler for developers to build this kind of outline view, which automatially works on iOS, iPadOS, and macOS.

Creating the Expandable List

In order to follow this chapter, please download these image assets from https://www.appcoda.com/resources/swiftui/expandablelist-images.zip. Then create a new SwiftUI project using the *App* template. I named the project *SwiftUIExpandableList* but you are free to set the name to whatever you want.

Once the project is created, unzip the image archive and add the images to the asset catalog.

In the project navigator, right click *SwiftUIExpandableList* and choose to create a new file. Select the *Swift File* template and name it *MenuItem.swift*.

Setting up the data model

To make the list view expandable, all you need to do is create a data model like this. Insert the following code in the file:

```
struct MenuItem: Identifiable {
    var id = UUID()
    var name: String
    var image: String
    var subMenuItems: [MenuItem]?
}
```

In the code above, we have a struct that models a menu item. The key to making a nested list is to include a property that contains an optional array of child menu items (i.e. `subMenuItems`). Note that the children are of the same type (MenuItem) as their parent.

For the top level menu items, we create an array of `MenuItem` like this:

```
// Main menu items
let sampleMenuItems = [ MenuItem(name: "Espresso Machines", image: "linea-mini", s
ubMenuItems: espressoMachineMenuItems),
                        MenuItem(name: "Grinders", image: "swift-mini", subMenuIte
ms: grinderMenuItems),
                        MenuItem(name: "Other Equipment", image: "espresso-ep", su
bMenuItems: otherMenuItems)
                    ]
```

For each of the menu item, we specify the array of the sub-menu items. If there are no sub-menu items, you can omit the `subMenuItems` parameter or pass it a `nil` value. We define the sub-menu items like this:

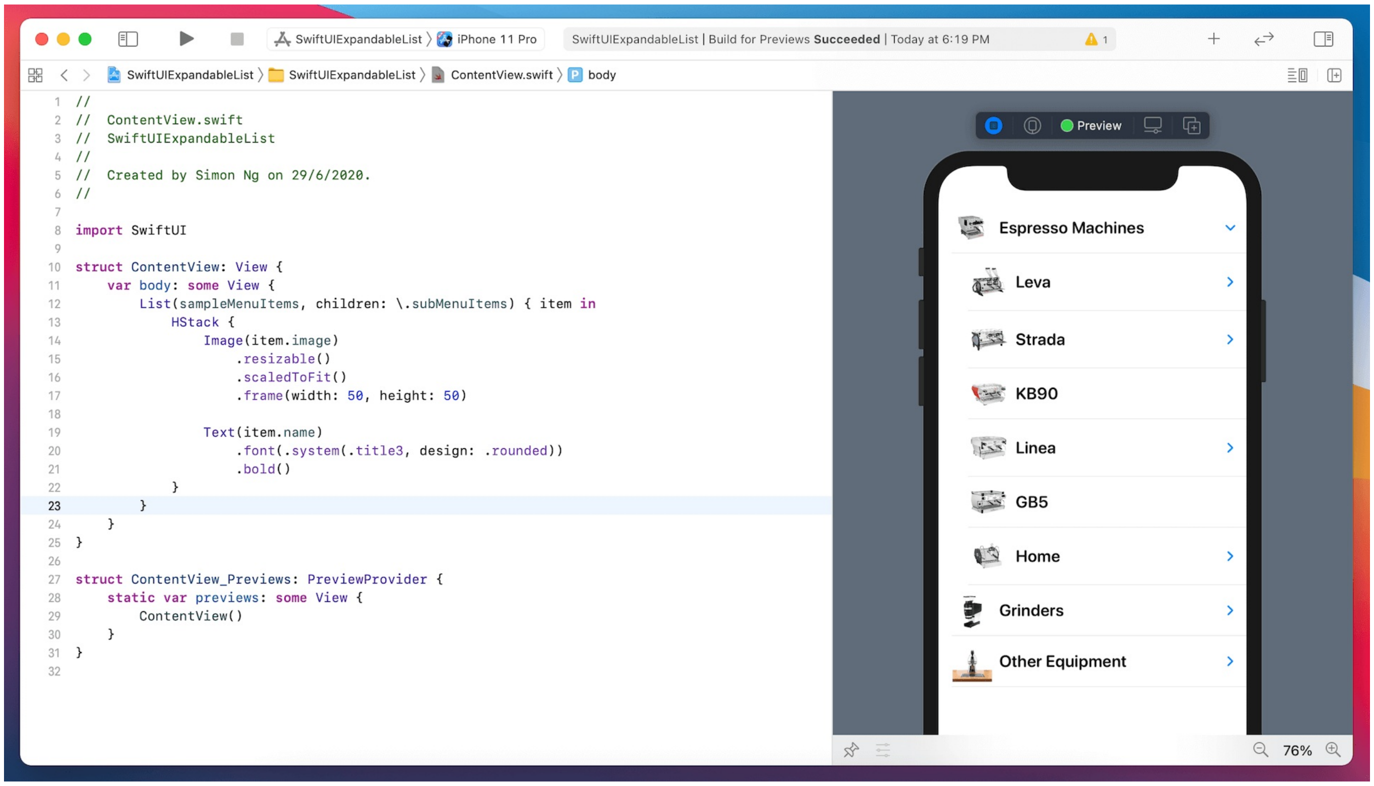
```
// Sub-menu items for Espressco Machines
let espressoMachineMenuItems = [ MenuItem(name: "Leva", image: "leva-x", subMenuIt
ems: [ MenuItem(name: "Leva X", image: "leva-x"), MenuItem(name: "Leva S", image:
"leva-s") ]),
                                 MenuItem(name: "Strada", image: "strada-ep", subM
enuItems: [ MenuItem(name: "Strada EP", image: "strada-ep"), MenuItem(name: "Strad
a AV", image: "strada-av"), MenuItem(name: "Strada MP", image: "strada-mp"), MenuI
tem(name: "Strada EE", image: "strada-ee") ]),
                                 MenuItem(name: "KB90", image: "kb90"),
                                 MenuItem(name: "Linea", image: "linea-pb-x", subM
enuItems: [ MenuItem(name: "Linea PB X", image: "linea-pb-x"), MenuItem(name: "Lin
ea PB", image: "linea-pb"), MenuItem(name: "Linea Classic", image: "linea-classic"
) ]),
                                 MenuItem(name: "GB5", image: "gb5"),
                                 MenuItem(name: "Home", image: "gs3", subMenuItems
: [ MenuItem(name: "GS3", image: "gs3"), MenuItem(name: "Linea Mini", image: "line
a-mini") ])
                                ]

// Sub-menu items for Grinder
let grinderMenuItems = [ MenuItem(name: "Swift", image: "swift"),
                         MenuItem(name: "Vulcano", image: "vulcano"),
                         MenuItem(name: "Swift Mini", image: "swift-mini"),
                         MenuItem(name: "Lux D", image: "lux-d")
                        ]

// Sub-menu items for other equipment
let otherMenuItems = [ MenuItem(name: "Espresso AV", image: "espresso-av"),
                         MenuItem(name: "Espresso EP", image: "espresso-ep"),
                         MenuItem(name: "Pour Over", image: "pourover"),
                         MenuItem(name: "Steam", image: "steam")
                        ]
```

Presenting the List

With the data model prepared, we can now present the list view. The `List` view has an optional `children` parameter. If you have any sub items, you can provide their key path. SwiftUI will then look up the sub menu items recursively and present them in outline form. Open `ContentView.swift` and insert the following code in `body`:

```
List(sampleMenuItems, children: \.subMenuItems) { item in
    HStack {
        Image(item.image)
            .resizable()
            .scaledToFit()
            .frame(width: 50, height: 50)

        Text(item.name)
            .font(.system(.title3, design: .rounded))
            .bold()
    }
}
```

In the closure of the `List` view, you describe how each row looks. In the code above, we layout an image and a text description using `HStack`. If you've added the code in `ContentView` correctly, SwiftUI should render the outline view as shown in figure 2.


SwiftUIExpandableList 〉 iPhone 11 Pro
SwiftUIExpandableList | Build for Previews Succeeded | Today at 6:19 PM
SwiftUIExpandableList 〉 SwiftUIExpandableList 〉 ContentView.swift 〉 body
Preview
Espresso Machines
Leva
Strada
KB90
Linea
GB5
Home
Grinders
Other Equipment
76%

```
//
//  ContentView.swift
//  SwiftUIExpandableList
//
//  Created by Simon Ng on 29/6/2020.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        List(sampleMenuItems, children: \.subMenuItems) { item in
            HStack {
                Image(item.image)
                    .resizable()
                    .scaledToFit()
                    .frame(width: 50, height: 50)

                Text(item.name)
                    .font(.system(.title3, design: .rounded))
                    .bold()
            }
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 2. The expandable list view

To test the app, run it in a simulator or the preview canvas. You can tap the disclosure indicator to access the submenu.

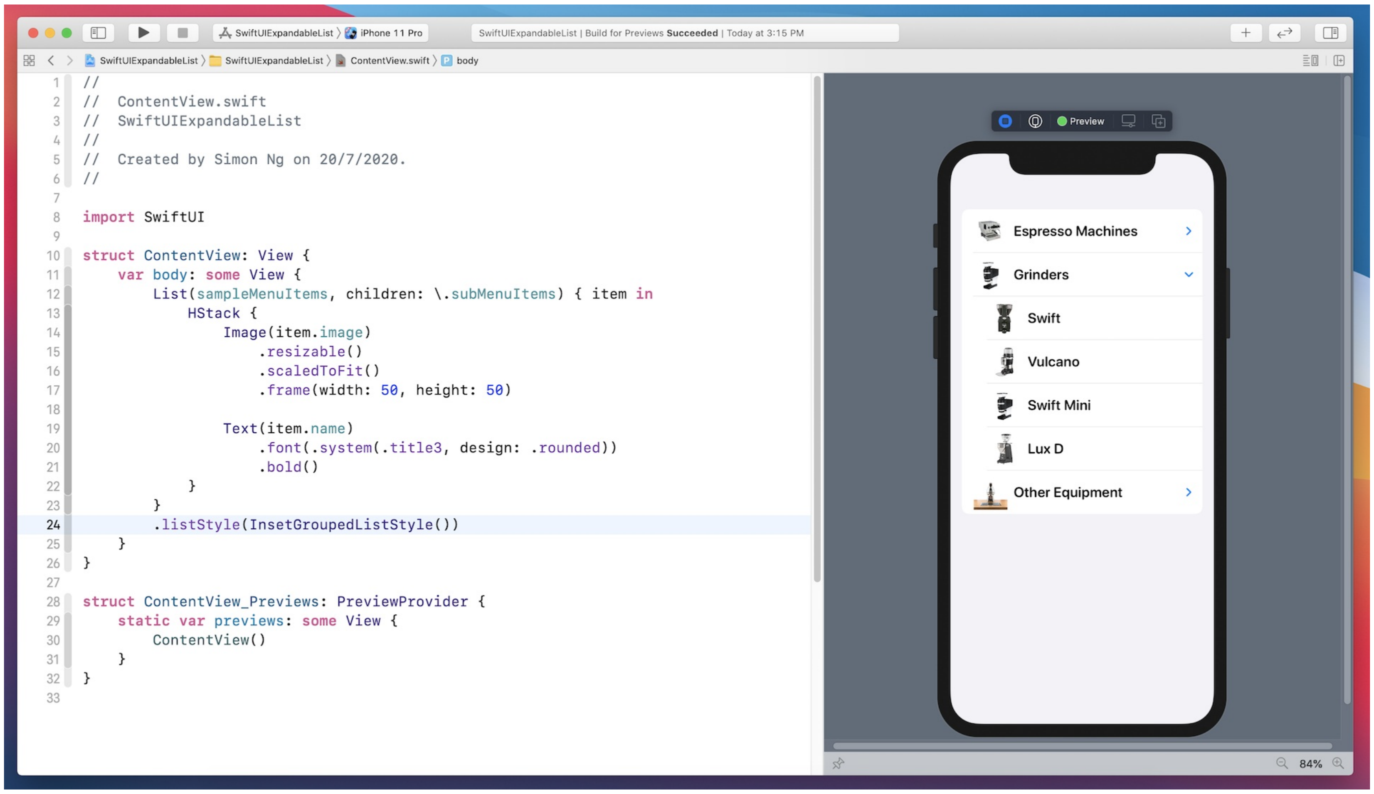
Using Inset Grouped List Style

In iOS 13, Apple brought a new style to the UITableView called Inset Grouped, where the grouped sections are inset with rounded corners. However, this style was not available to the `List` view in SwiftUI. With the release of iOS 14, Apple added this new style to SwiftUI list.

To use this new list style, you can attach the `.listStyle` modifier to the `List` view and pass it the instance of `InsetGroupedListStyle` like this:

```
List {
  ...
}
.listStyle(InsetGroupedListStyle())
```

If you've followed me, the list view should now change to the inset grouped style.



```swift
//
//  ContentView.swift
//  SwiftUIExpandableList
//
//  Created by Simon Ng on 20/7/2020.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        List(sampleMenuItems, children: \.subMenuItems) { item in
            HStack {
                Image(item.image)
                    .resizable()
                    .scaledToFit()
                    .frame(width: 50, height: 50)

                Text(item.name)
                    .font(.system(.title3, design: .rounded))
                    .bold()
            }
        }
        .listStyle(InsetGroupedListStyle())
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 3. Using inset grouped list style

Using OutlineGroup to Customize the Expandable List

As you can see in the earlier example, it is pretty easy to create an outline view using the `List` view. However, if you want to have a better control of the appearance of the outline view (e.g. adding a section header), you will need to use `OutlineGroup`. This new view is introduced in iOS 14 for you to present a hierarchy of data.

If you understand how to build an expandable list view, the usage of `OutlineGroup` is very similar. For example, the following code allows you to build the same expandable list view like the one shown in figure 1:

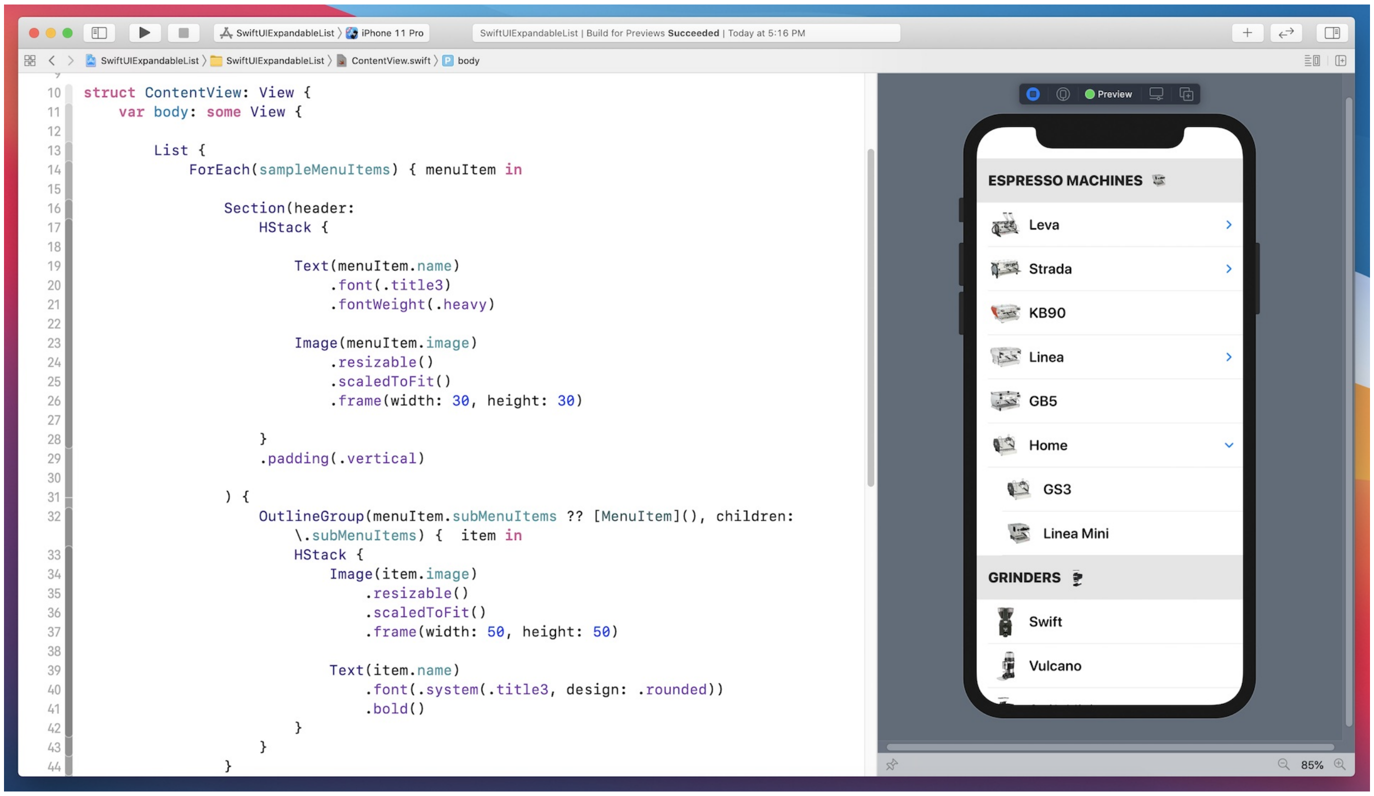
```
List {
    OutlineGroup(sampleMenuItems, children: \.subMenuItems) {  item in
        HStack {
            Image(item.image)
                .resizable()
                .scaledToFit()
                .frame(width: 50, height: 50)

            Text(item.name)
                .font(.system(.title3, design: .rounded))
                .bold()
        }
    }
}
```

Similar to the `List` view, you just need to pass `OutlineGroup` the array of items and specify the key path for the sub menu items (or children).

With `OutlineGroup`, you have better control on the appearance of the outline view. For example, we want to display the top-level menu items as the section header. You can write the code like this:

```
List {
    ForEach(sampleMenuItems) { menuItem in

        Section(header:
            HStack {

                Text(menuItem.name)
                    .font(.title3)
                    .fontWeight(.heavy)

                Image(menuItem.image)
                    .resizable()
                    .scaledToFit()
                    .frame(width: 30, height: 30)

            }
            .padding(.vertical)

        ) {
            OutlineGroup(menuItem.subMenuItems ?? [MenuItem](), children: \.subMen
uItems) {  item in
                HStack {
                    Image(item.image)
                        .resizable()
                        .scaledToFit()
                        .frame(width: 50, height: 50)

                    Text(item.name)
                        .font(.system(.title3, design: .rounded))
                        .bold()
                }
            }
        }
    }
}
```

In the code above, we use `ForEach` to loop through the menu items. We present the top-level items as section headers. For the rest of the sub menu items, we rely on `OutlineGroup` to create the hierachy of data. If you've made the change in `ContentView.swift`, you should see an outline view like that shown in figure 4.


SwiftUIExpandableList 〉 iPhone 11 Pro
SwiftUIExpandableList | Build for Previews Succeeded | Today at 5:16 PM
SwiftUIExpandableList 〉 SwiftUIExpandableList 〉 ContentView.swift 〉 body
Preview
ESPRESSO MACHINES
Leva
Strada
KB90
Linea
GB5
Home
GS3
Linea Mini
GRINDERS
Swift
Vulcano
85%

```
struct ContentView: View {
    var body: some View {

        List {
            ForEach(sampleMenuItems) { menuItem in

                Section(header:
                    HStack {

                        Text(menuItem.name)
                            .font(.title3)
                            .fontWeight(.heavy)

                        Image(menuItem.image)
                            .resizable()
                            .scaledToFit()
                            .frame(width: 30, height: 30)

                    }
                    .padding(.vertical)

                ) {
                    OutlineGroup(menuItem.subMenuItems ?? [MenuItem](), children:
                        \.subMenuItems) {  item in
                        HStack {
                            Image(item.image)
                                .resizable()
                                .scaledToFit()
                                .frame(width: 50, height: 50)

                            Text(item.name)
                                .font(.system(.title3, design: .rounded))
                                .bold()
                        }
                    }
                }
```

Figure 4. Building the outline view using OutlineGroup

Similarly, if you prefer to use the inset group list style, you can attach the `listStyle` modifier to the `List` view:

```
.listStyle(InsetGroupedListStyle())
```

You then achieve an outline view like figure 5.

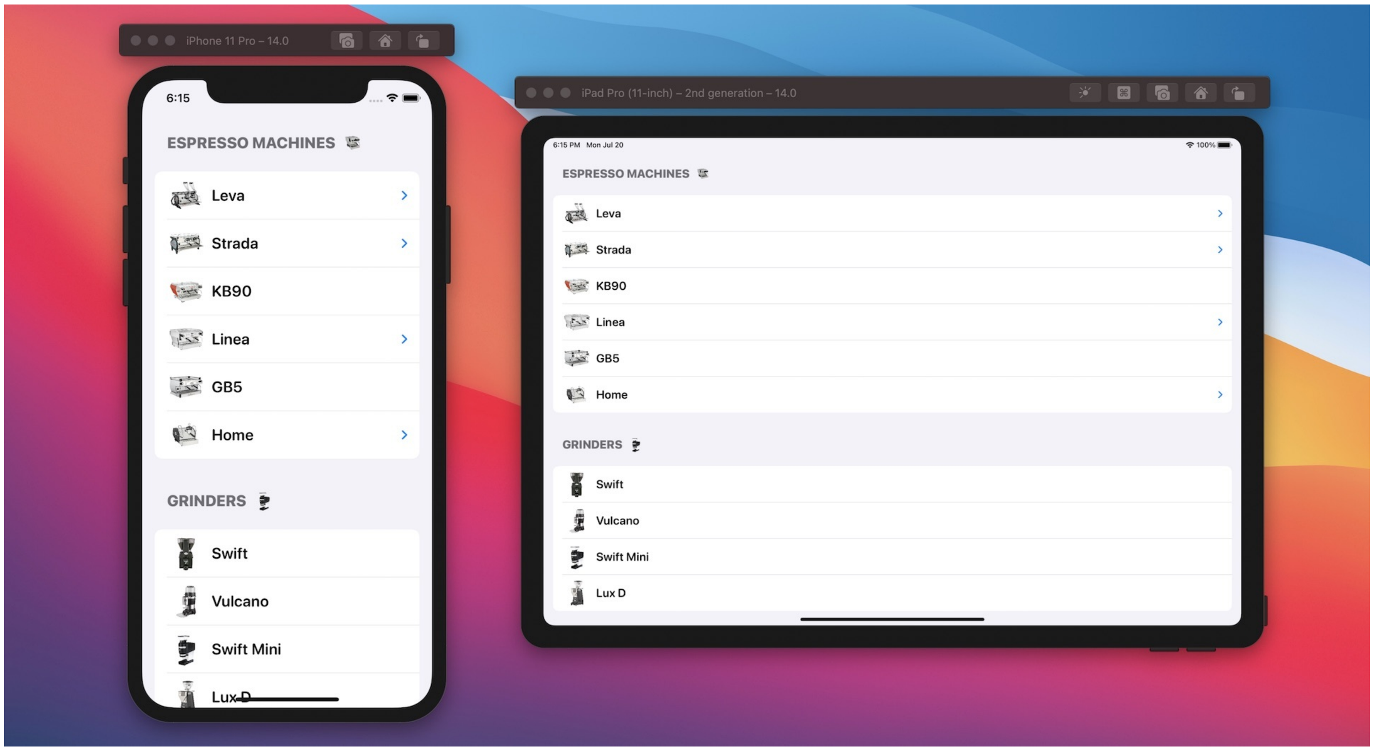


Figure 5. Applying the inset grouped list style

Understanding DisclosureGroup

In the outline view, you can show/hide the sub menu items by tapping the disclosure indicator. Whether you use `List` or `OutlineGroup` to implement the expandable list, this "expand & collapse" feature is supported by a new view called `DisclosureGroup`, introduced in iOS 14.

The disclosure group view is designed to show or hide another content view. While `DisclosureGroup` is automatically embedded in `OutlineGroup`, you can use this view independently. For example, you can use the following code to show & hide a question and an answer:

```
DisclosureGroup(
    content: {
        Text("Absolutely! You are allowed to reuse the source code in your own pro
jects (personal/commercial). However, you're not allowed to distribute or sell the
 source code without prior authorization.")
            .font(.body)
            .fontWeight(.light)
    },
    label: {
        Text("1. Can I reuse the source code?")
            .font(.body)
            .bold()
    }
)
```

The disclosure group view takes in two parameters: *label* and *content*. In the code above, we specify the question in the `label` parameter and the answer in the `content` parameter. Figure 6 shows you the result.

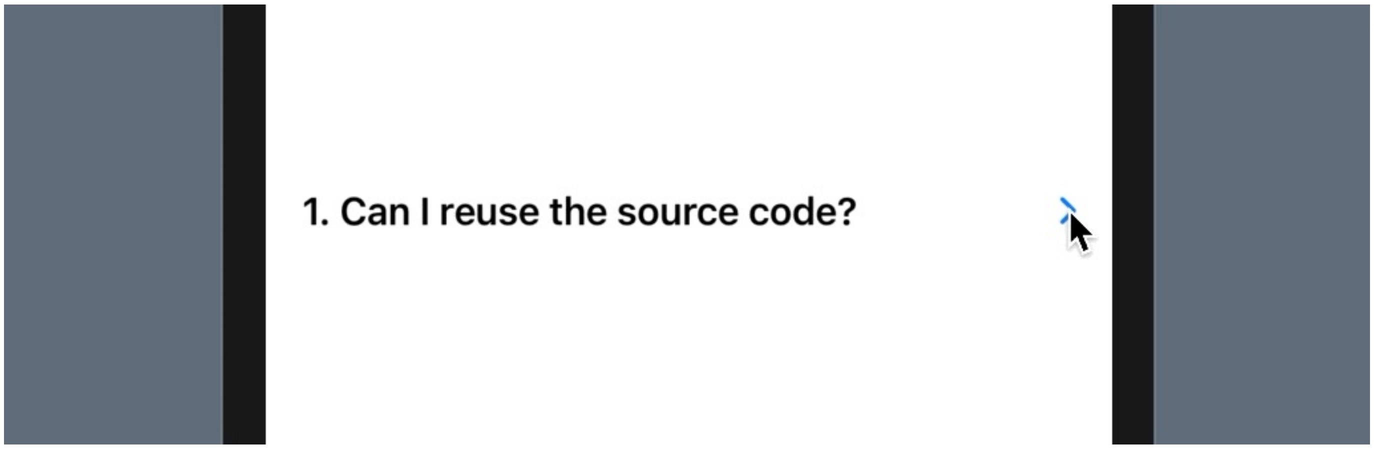


Figure 6. Using DisclosureGroup for showing and hiding content

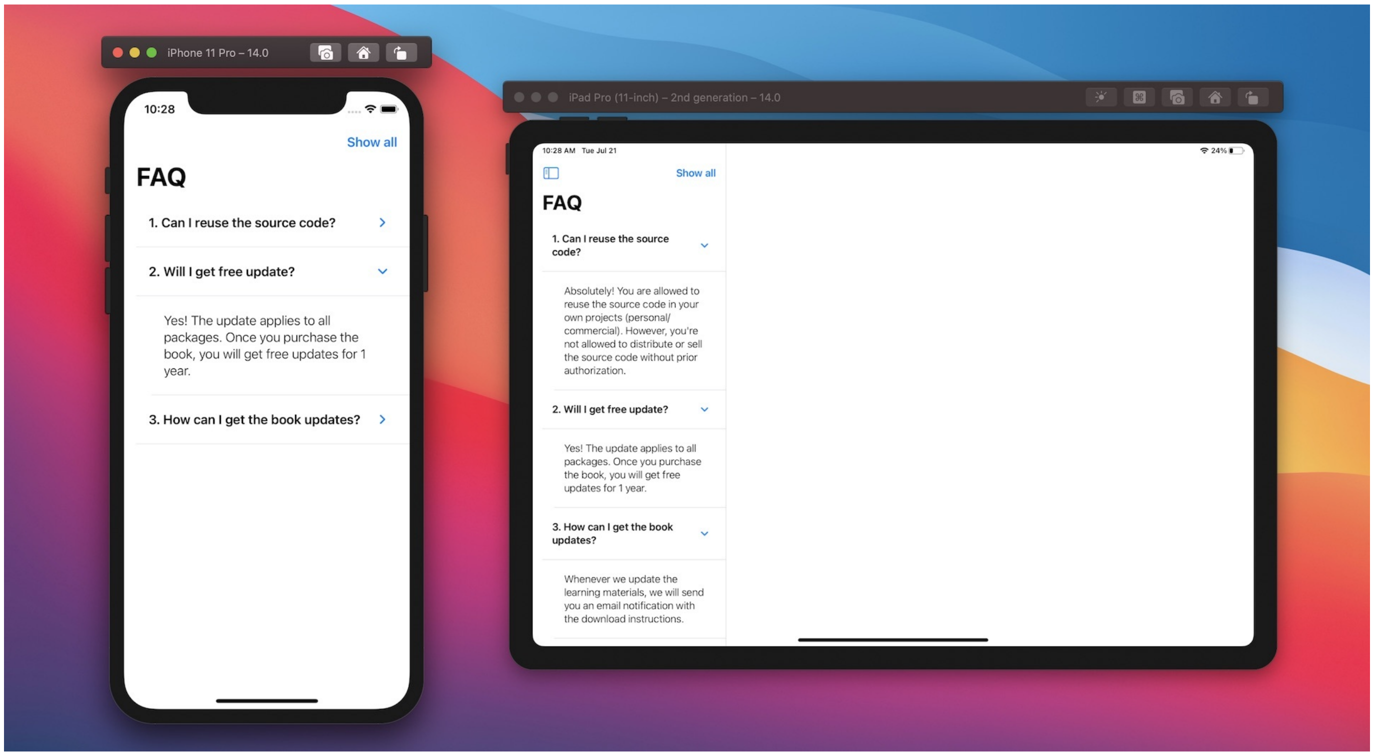
By default, the disclosure group view is in hidden mode. To reveal the content view, you tap the disclosure indicator to switch the disclosure group view to the "expand" state.

Optionally, you can control the state of `DisclosureGroup` by passing it a binding which specifies the state of the disclosure indicator (expanded or collapsed) like this:

```swift
struct FaqView: View {
    @State var showContent = true

    var body: some View {
        DisclosureGroup(
            isExpanded: $showContent,
            content: {
                ...
            },
            label: {
                ...
            }
        )
        .padding()
    }
}
```

Exercise

The `DisclosureGroup` view allows you to have finer control over the state of the disclosure indicator. Your exercise is to create a FAQ screen similar to the one shown in figure 7.


iPhone 11 Pro – 14.0
10:28
Show all
FAQ
1. Can I reuse the source code?
2. Will I get free update?
3. How can I get the book updates?
iPad Pro (11-inch) – 2nd generation – 14.0
10:28 AM Tue Jul 21
24%
Show all
FAQ
1. Can I reuse the source code?
2. Will I get free update?
3. How can I get the book updates?

Yes! The update applies to all packages. Once you purchase the book, you will get free updates for 1 year.

Absolutely! You are allowed to reuse the source code in your own projects (personal/ commercial). However, you're not allowed to distribute or sell the source code without prior authorization.

Yes! The update applies to all packages. Once you purchase the book, you will get free updates for 1 year.

Whenever we update the learning materials, we will send you an email notification with the download instructions.

Figure 7. Your exercise

Users can tap the disclosure indicator to show or hide an individual question. Additionally, the app provides a "Show All" button to expand all questions and reveal the answers at once.

Summary

In this chapter, I've introduced a couple of new features of SwiftUI. As you can see in the demo, it is very easy to build an outline view or expandable list view. All you need to do is define a correct data model. The List view handles the rest, traverses the data structure, and renders the outline view. On top of that, the new update provides `OutlineGroup` and `DisclosureGroup` for you to further customize the outline view.

For reference, you can download the complete expandable list project here:

- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUIExpandableList.zip)

Please note that you can refer to `FaqView.swift` for the solution to the exercise.

Chapter 29
Building Grid Layouts Using LazyVGrid and LazyHGrid

The initial release of SwiftUI didn't come with a native collection view. You can either build your own solution or use third party libraries. In WWDC 2020, Apple introduced tons of new features for the SwiftUI framework. One of them is to address the need for grid views. SwiftUI now provides developers two new UI components called *LazyVGrid* and *LazyHGrid*. One is for creating vertical grids and the other is for horizontal grids. The word *Lazy*, as mentioned by Apple, refers to the grid view not creating items until they are needed. What this means to you is that the performance of these grid views are already optimized by default.

In this chapter, I will walk you through how to create both horizontal and vertical views. Both *LazyVGrid* and *LazyHGrid* are designed to be flexible, so that developers can easily create various types of grid layouts. We will also look into how to vary the size of grid items to achieve different layouts. After you manage the basics, we will dive a little bit deeper and create complex layouts like that shown in figure 1.

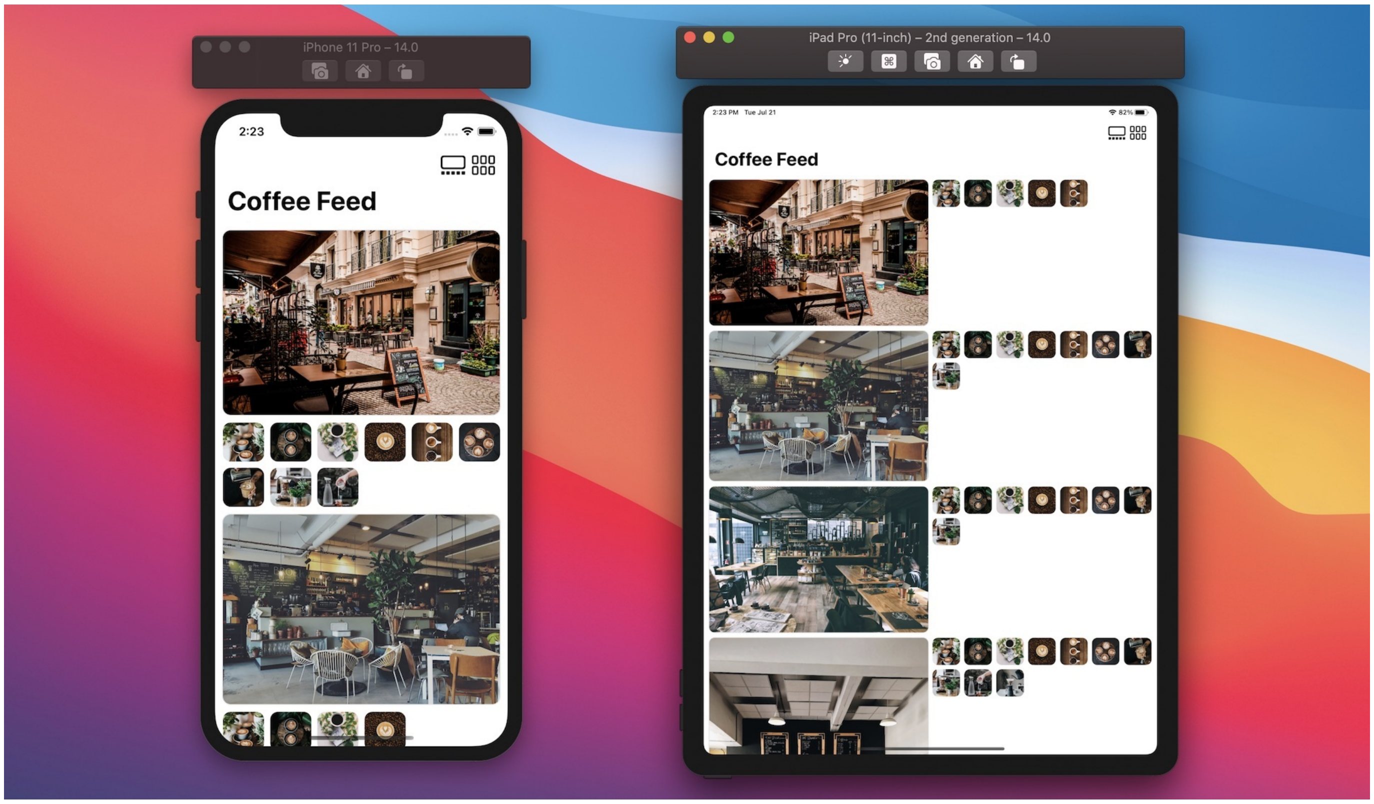


Figure 1. Sample grid layouts

The Essential of Grid Layout in SwiftUI

To create a grid layout, whether it's horizontal or vertical, here are the steps you follow:

1. First, you need to prepare the raw data for presentation in the grid. For example, here is an array of SF symbols that we are going to present in the demo app:

```
private var symbols = ["keyboard", "hifispeaker.fill", "printer.fill", "tv.fill",
"desktopcomputer", "headphones", "tv.music.note", "mic", "plus.bubble", "video"]
```

2. Create an array of type `GridItem` that describes what the grid will look like. Including, how many columns the grid should have. Here is a code snippet for describing a 3-column grid:

```
private var threeColumnGrid = [GridItem(.flexible()), GridItem(.flexible()), GridI
tem(.flexible())]
```

3. Next, you layout the grid by using `LazyVGrid` and `ScrollView`. Here is an example:

```
ScrollView {
    LazyVGrid(columns: threeColumnGrid) {
        // Display the item
    }
}
```

4. Alternatively, if you want to build a horizontal grid, you use `LazyHGrid` like this:

```
ScrollView(.horizontal) {
    LazyHGrid(rows: threeColumnGrid) {
        // Display the item
    }
}
```

Using LazyVGrid to Create Vertical Grids

With a basic understanding of the grid layout, let's put the code to work. We will start with something simple by building a 3-column grid. Open Xcode 12 (or greater) and create a new project with the *App* template. Please make sure you select *SwiftUI* for the Interface option. Name the project *SwiftUIGridLayout* or whatever name you prefer.

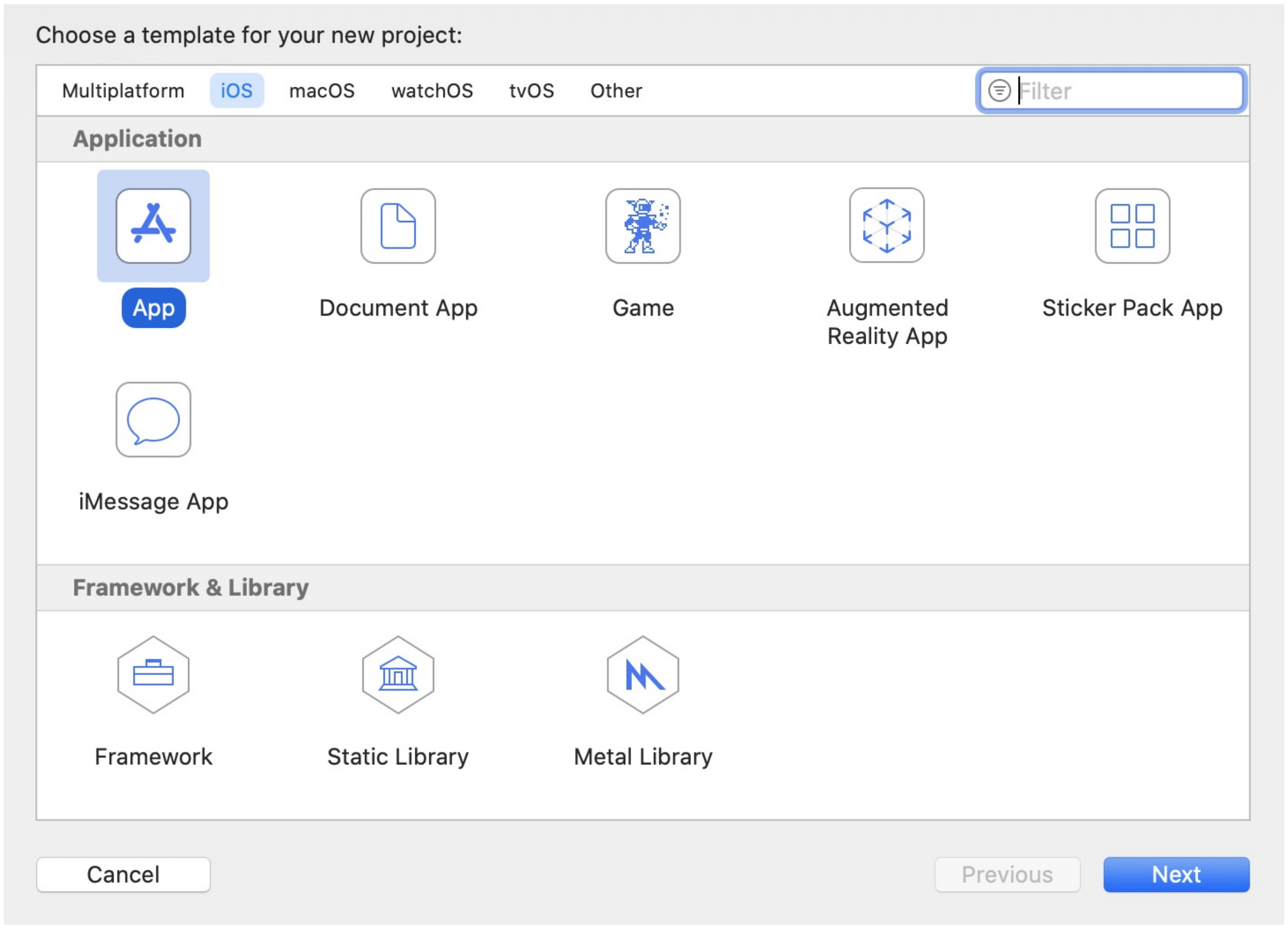


Figure 2. Creating a new project using the App template

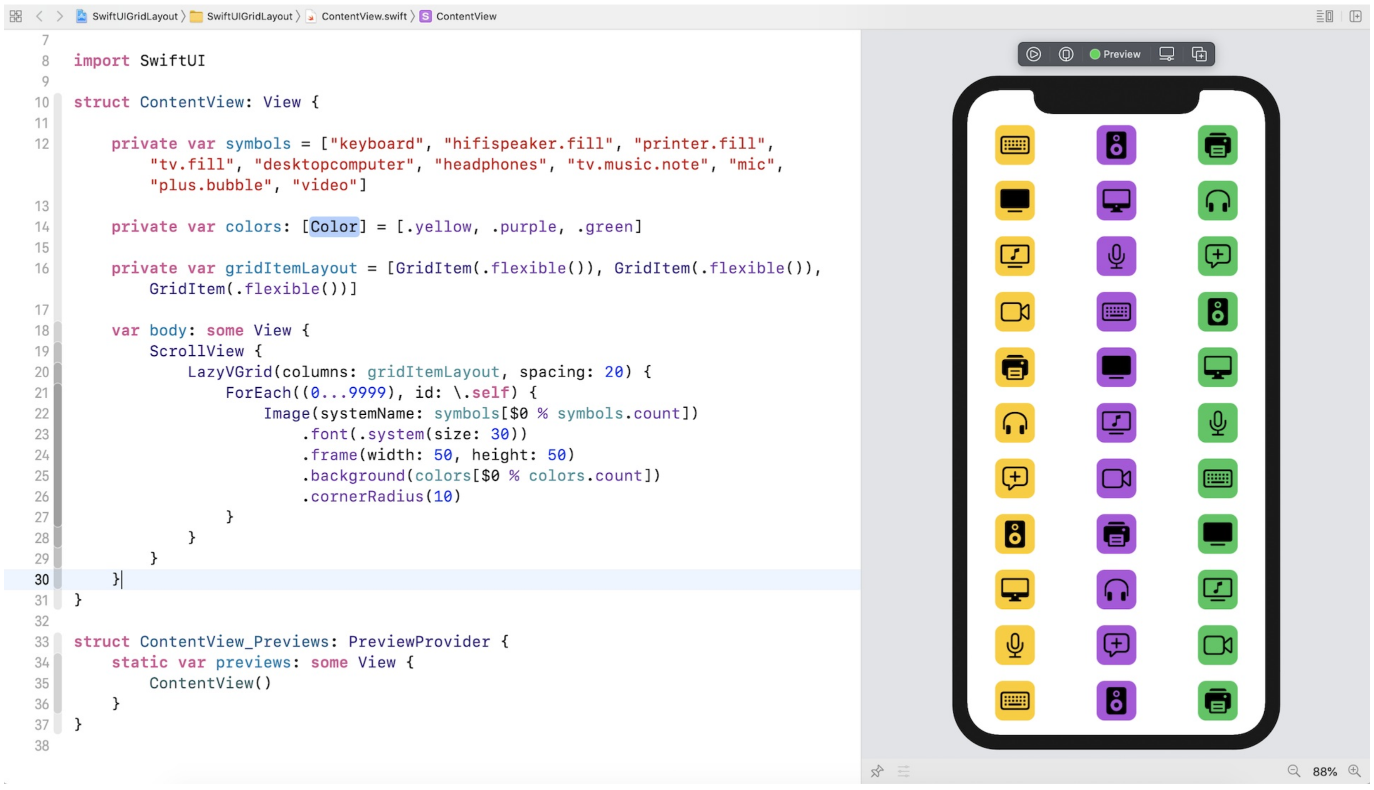
Once the project is created, choose `ContentView.swift`. In `ContentView`, declare the following variables:

```
private var symbols = ["keyboard", "hifispeaker.fill", "printer.fill", "tv.fill",
"desktopcomputer", "headphones", "tv.music.note", "mic", "plus.bubble", "video"]

private var colors: [Color] = [.yellow, .purple, .green]

private var gridItemLayout = [GridItem(.flexible()), GridItem(.flexible()), GridIt
em(.flexible())]
```

We are going to display a set of SF symbols in a 3-column grid. To present the grid, update the `body` variable like this:

```
var body: some View {
    ScrollView {
        LazyVGrid(columns: gridItemLayout, spacing: 20) {
            ForEach((0...9999), id: \.self) {
                Image(systemName: symbols[$0 % symbols.count])
                    .font(.system(size: 30))
                    .frame(width: 50, height: 50)
                    .background(colors[$0 % colors.count])
                    .cornerRadius(10)
            }
        }
    }
}
```

We use `LazyVGrid` and tell the vertical grid to use a 3-column layout. We also specify that there is a 20 point space between rows. In the code block, we have a `ForEach` loop to present a total of 10,000 image views. If you've made the change correctly, you should see a three column grid in the preview.


SwiftUIGridLayout 〉 SwiftUIGridLayout 〉 ContentView.swift 〉 ContentView
Preview
88%

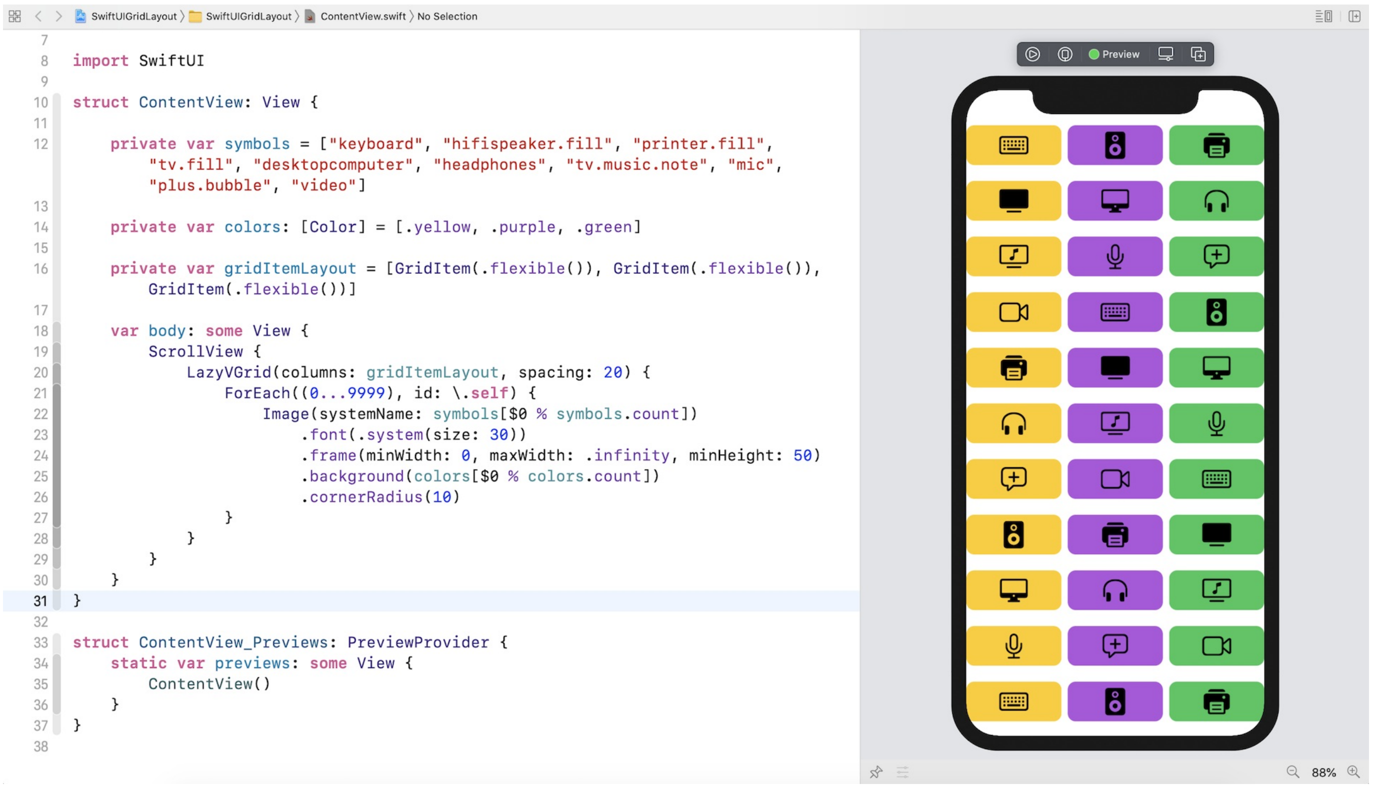
```swift
import SwiftUI

struct ContentView: View {

    private var symbols = ["keyboard", "hifispeaker.fill", "printer.fill",
        "tv.fill", "desktopcomputer", "headphones", "tv.music.note", "mic",
        "plus.bubble", "video"]

    private var colors: [Color] = [.yellow, .purple, .green]

    private var gridItemLayout = [GridItem(.flexible()), GridItem(.flexible()),
        GridItem(.flexible())]

    var body: some View {
        ScrollView {
            LazyVGrid(columns: gridItemLayout, spacing: 20) {
                ForEach((0...9999), id: \.self) {
                    Image(systemName: symbols[$0 % symbols.count])
                        .font(.system(size: 30))
                        .frame(width: 50, height: 50)
                        .background(colors[$0 % colors.count])
                        .cornerRadius(10)
                }
            }
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 3. Displaying a 3-column grid

This is how we create a vertical grid with three columns. The frame size of the image is fixed to 50 by 50 points, which is controlled by the `.frame` modifier. If you want to make a grid item wider, you can alter the frame modifier like this:

```swift
.frame(minWidth: 0, maxWidth: .infinity, minHeight: 50)
```

The image's width will expand to take up the column's width like that shown in figure 4.


SwiftUIGridLayout 〉 SwiftUIGridLayout 〉 ContentView.swift 〉 No Selection
Preview
88%

```swift
import SwiftUI

struct ContentView: View {

    private var symbols = ["keyboard", "hifispeaker.fill", "printer.fill",
        "tv.fill", "desktopcomputer", "headphones", "tv.music.note", "mic",
        "plus.bubble", "video"]

    private var colors: [Color] = [.yellow, .purple, .green]

    private var gridItemLayout = [GridItem(.flexible()), GridItem(.flexible()),
        GridItem(.flexible())]

    var body: some View {
        ScrollView {
            LazyVGrid(columns: gridItemLayout, spacing: 20) {
                ForEach((0...9999), id: \.self) {
                    Image(systemName: symbols[$0 % symbols.count])
                        .font(.system(size: 30))
                        .frame(minWidth: 0, maxWidth: .infinity, minHeight: 50)
                        .background(colors[$0 % colors.count])
                        .cornerRadius(10)
                }
            }
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 4. Changing the frame size of the grid items

Note that there is a space between the columns and rows. Sometimes, you may want to create a grid without any spaces. How can you achieve that? The space between rows is controlled by the `spacing` parameter of `LazyVGrid`. We have set its value to `20` points. You can simply change it to `0` such that there is no space between rows.

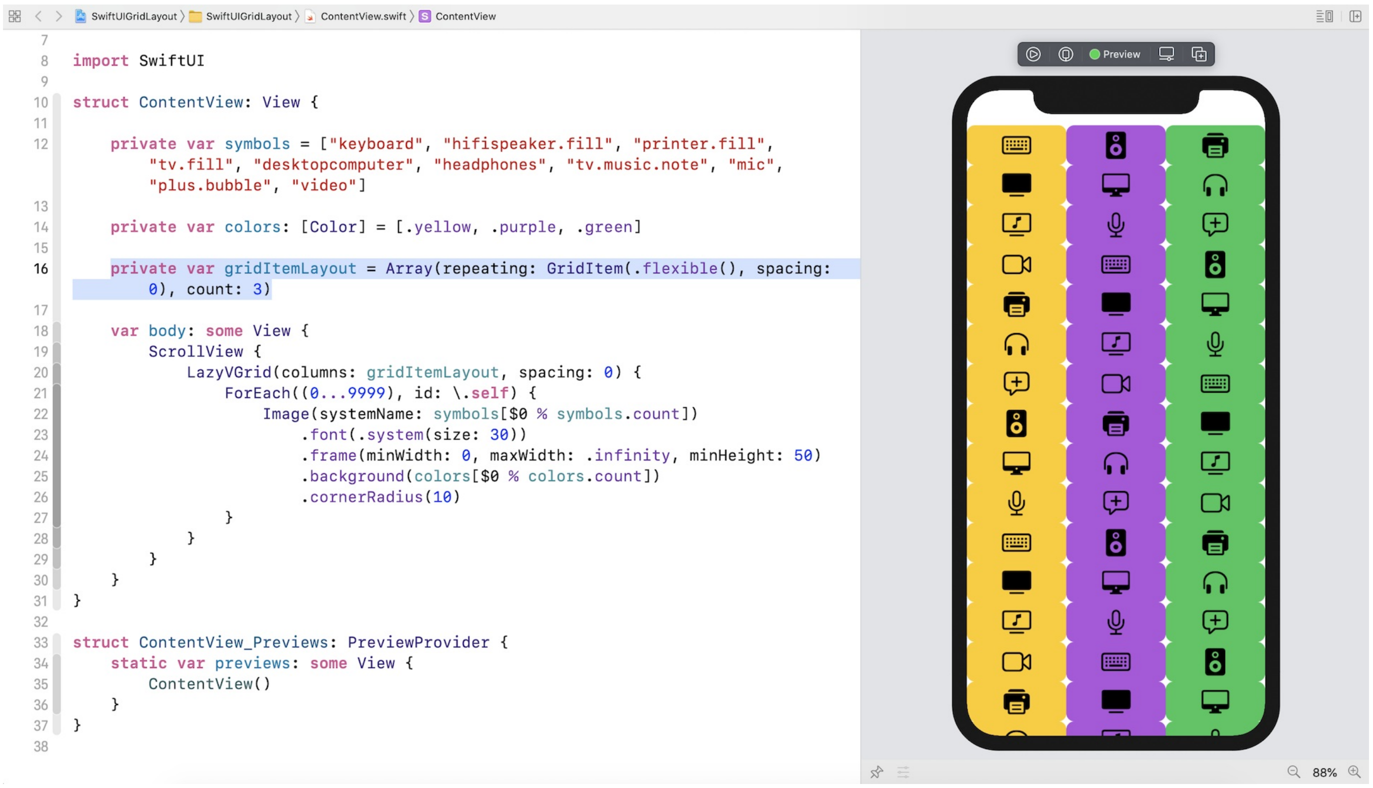
The spacing between grid items is controlled by the instances of `GridItem` initialized in `gridItemLayout`. You can set the spacing between items by passing a value to the `spacing` parameter. Therefore, to remove the spacing between rows, you can initialize the `gridLayout` variable like this:

```swift
private var gridItemLayout = [GridItem(.flexible(), spacing: 0), GridItem(.flexibl
e(), spacing: 0), GridItem(.flexible(), spacing: 0)]
```

For each `GridItem`, we specify to use a spacing of zero. For simplicity, the code above can be rewritten like this:

```
private var gridItemLayout = Array(repeating: GridItem(.flexible(), spacing: 0), c
ount: 3)
```

If you've made both changes, your preview canvas should show you a grid view without any spacing.


SwiftUIGridLayout 〉 SwiftUIGridLayout 〉 ContentView.swift 〉 ContentView
Preview
88%

```
import SwiftUI

struct ContentView: View {

    private var symbols = ["keyboard", "hifispeaker.fill", "printer.fill",
        "tv.fill", "desktopcomputer", "headphones", "tv.music.note", "mic",
        "plus.bubble", "video"]

    private var colors: [Color] = [.yellow, .purple, .green]

    private var gridItemLayout = Array(repeating: GridItem(.flexible(), spacing:
        0), count: 3)

    var body: some View {
        ScrollView {
            LazyVGrid(columns: gridItemLayout, spacing: 0) {
                ForEach((0...9999), id: \.self) {
                    Image(systemName: symbols[$0 % symbols.count])
                        .font(.system(size: 30))
                        .frame(minWidth: 0, maxWidth: .infinity, minHeight: 50)
                        .background(colors[$0 % colors.count])
                        .cornerRadius(10)
                }
            }
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 5. Removing the spacing between columns and rows

Using GridItem to Vary the Grid Layout (Flexible/Fixed/Adaptive)

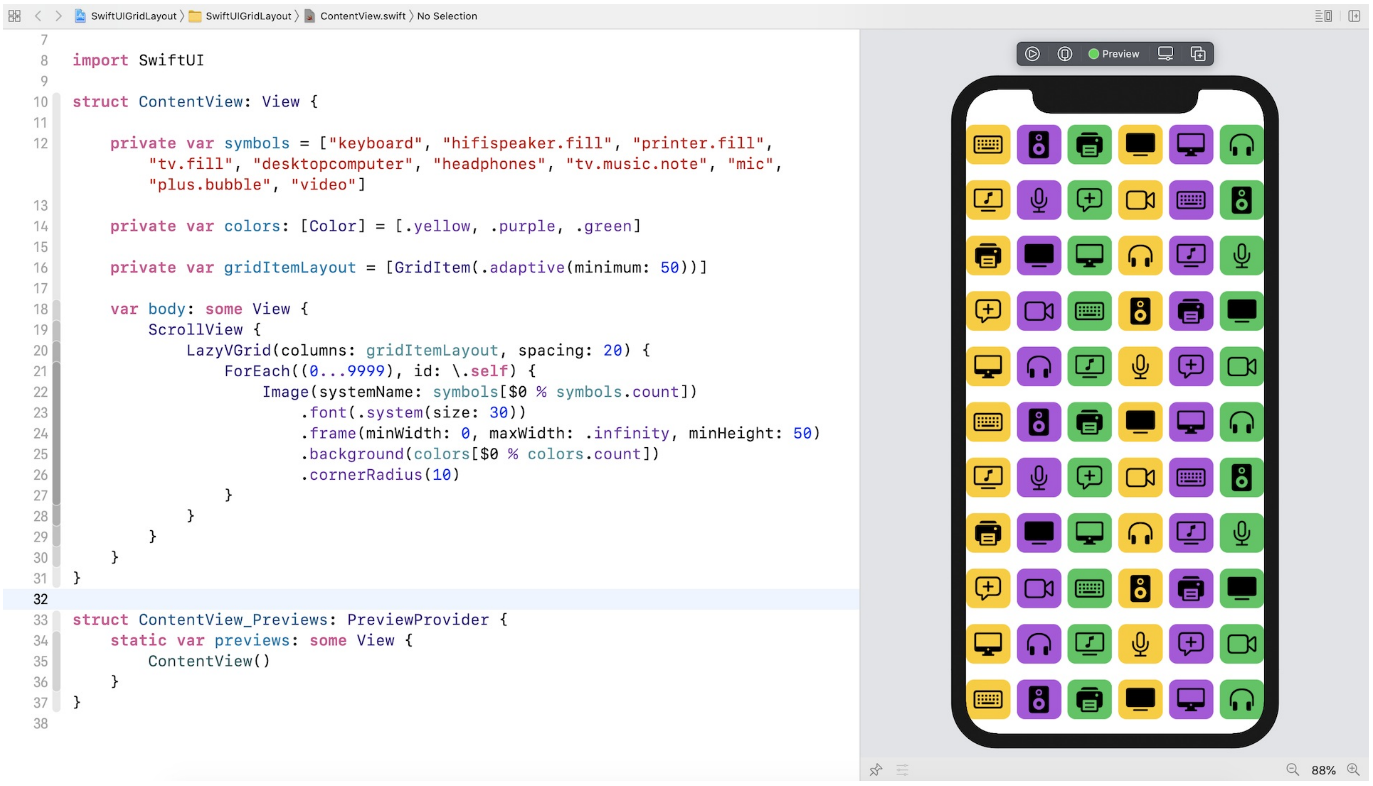
Let's take a further look at `GridItem`. You use `GridItem` instances to configure the layout of items in `LazyHGrid` and `LazyVGrid` views. Earlier, we defined an array of three `GridItem` instances, each of which uses the size type `.flexible()`. The flexible size type enables you to create three columns with equal size. If you want to describe a 6-column grid, you can create the array of `GridItem` like this:

```
private var sixColumnGrid: [GridItem] = Array(repeating: .init(.flexible()), count
: 6)
```

`.flexible()` is just one of the size types for controlling the grid layout. If you want to place as many items as possible in a row, you can use the *adaptive* size type:

```
private var gridItemLayout = [GridItem(.adaptive(minimum: 50))]
```

The *adaptive* size type requires you to specify the minimize size for a grid item. In the code above, each grid item has a minimum size of 50. If you modify the `gridItemLayout` variable as above and set the spacing of `LazyVGrid` back to `20`, you should achieve a grid layout similar to the one shown in figure 6.


SwiftUIGridLayout 〉 SwiftUIGridLayout 〉 ContentView.swift 〉 No Selection
Preview
88%

```

import SwiftUI

struct ContentView: View {

    private var symbols = ["keyboard", "hifispeaker.fill", "printer.fill",
        "tv.fill", "desktopcomputer", "headphones", "tv.music.note", "mic",
        "plus.bubble", "video"]

    private var colors: [Color] = [.yellow, .purple, .green]

    private var gridItemLayout = [GridItem(.adaptive(minimum: 50))]

    var body: some View {
        ScrollView {
            LazyVGrid(columns: gridItemLayout, spacing: 20) {
                ForEach((0...9999), id: \.self) {
                    Image(systemName: symbols[$0 % symbols.count])
                        .font(.system(size: 30))
                        .frame(minWidth: 0, maxWidth: .infinity, minHeight: 50)
                        .background(colors[$0 % colors.count])
                        .cornerRadius(10)
                }
            }
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}

```

Figure 6. Using adaptive size to create the grid

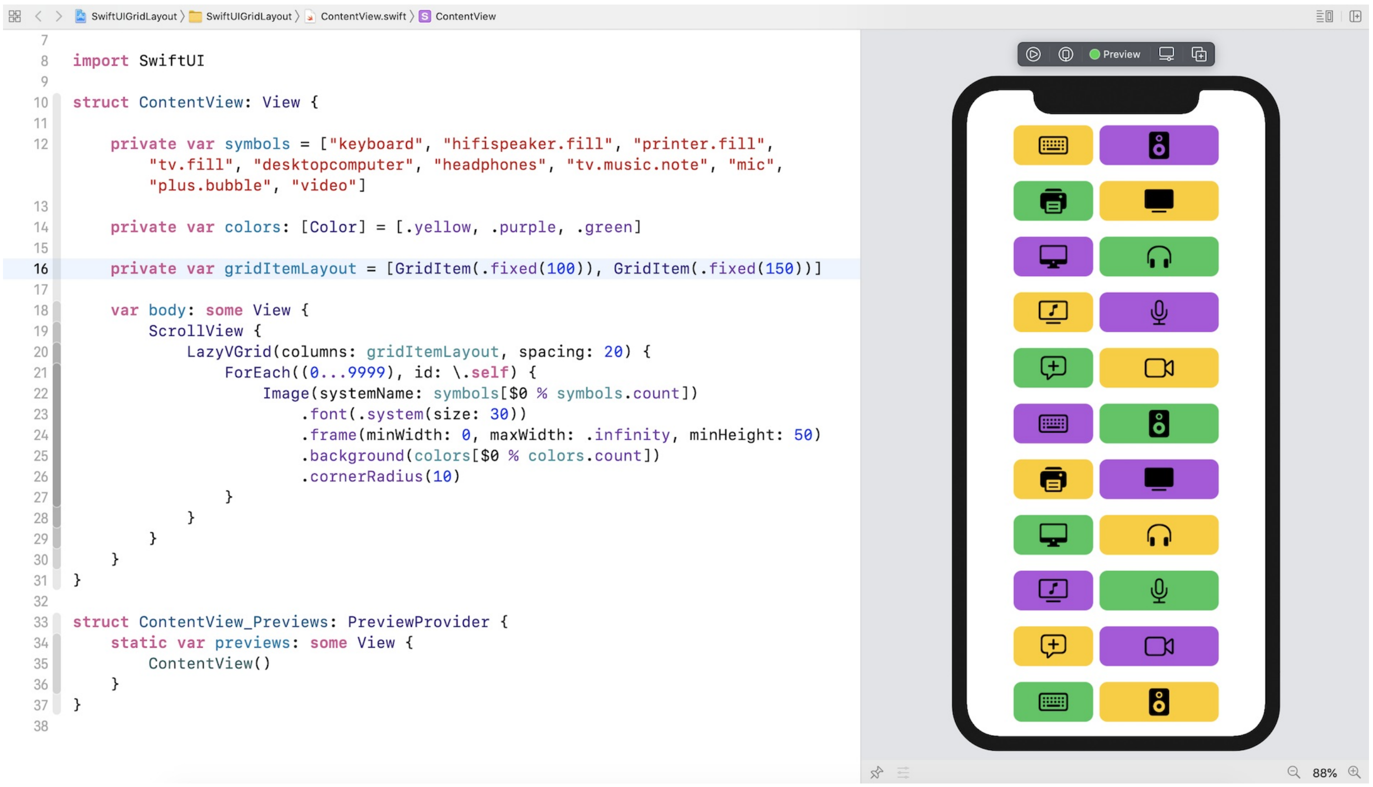
By using `.adaptive(minimum: 50)`, you instruct `LazyVGrid` to fill as many images as possible in a row such that each item has a minimum size of 50 points.

Note: I used iPhone 11 Pro as the simulator. If you use other iOS simulators with different screen sizes, you may achieve a different result.

In addition to `.flexible` and `.adaptive`, you can also use `.fixed` if you want to create fixed width columns. For example, you want to layout the image in two columns such that the first column has a width of 100 points and the second one has a width of 150 points. You write the code like this:

```
private var gridItemLayout = [GridItem(.fixed(100)), GridItem(.fixed(150))]
```

Update the `gridItemLayout` variable as shown above, this will result in a two-column grid with different sizes.


SwiftUIGridLayout 〉 SwiftUIGridLayout 〉 ContentView.swift 〉 ContentView
Preview
88%

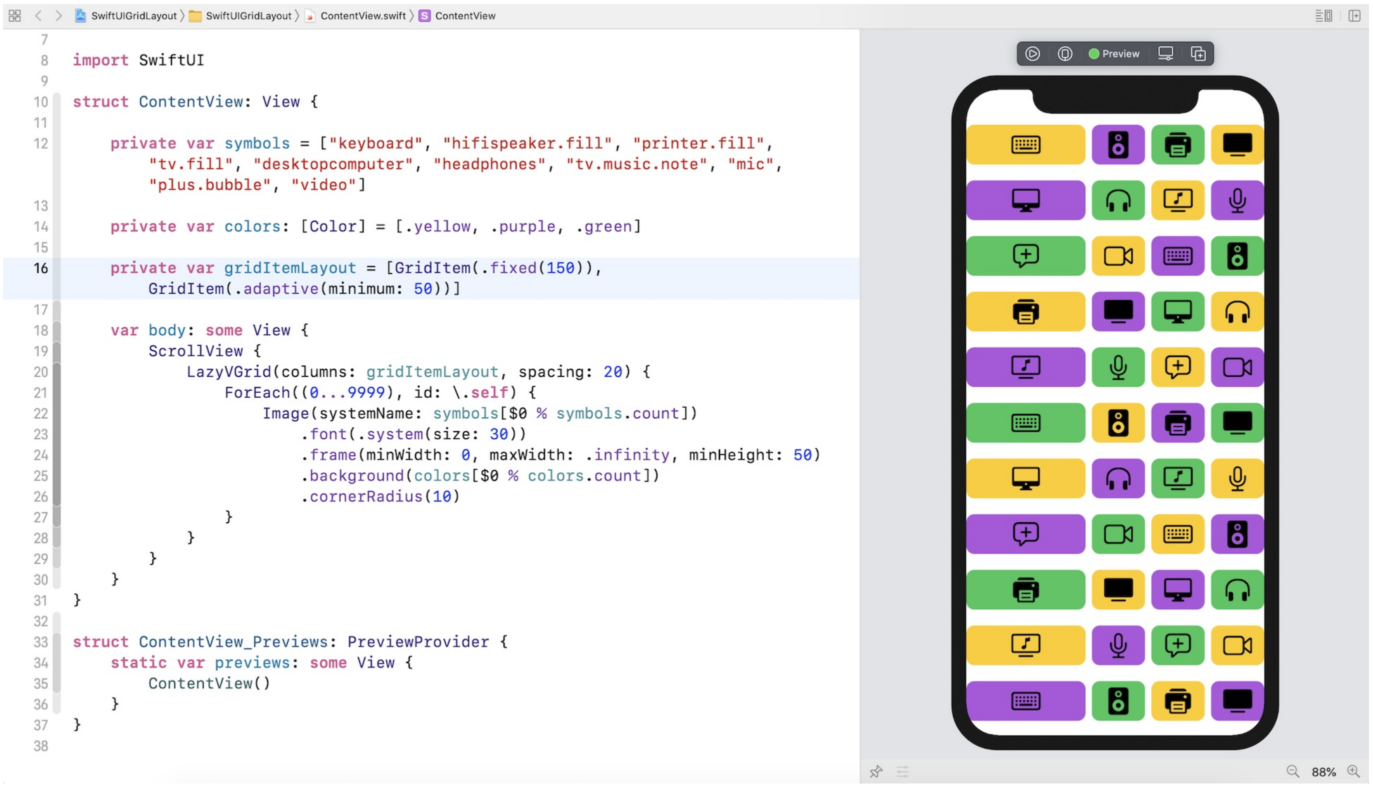
```

import SwiftUI

struct ContentView: View {

    private var symbols = ["keyboard", "hifispeaker.fill", "printer.fill",
        "tv.fill", "desktopcomputer", "headphones", "tv.music.note", "mic",
        "plus.bubble", "video"]

    private var colors: [Color] = [.yellow, .purple, .green]

    private var gridItemLayout = [GridItem(.fixed(100)), GridItem(.fixed(150))]

    var body: some View {
        ScrollView {
            LazyVGrid(columns: gridItemLayout, spacing: 20) {
                ForEach((0...9999), id: \.self) {
                    Image(systemName: symbols[$0 % symbols.count])
                        .font(.system(size: 30))
                        .frame(minWidth: 0, maxWidth: .infinity, minHeight: 50)
                        .background(colors[$0 % colors.count])
                        .cornerRadius(10)
                }
            }
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}

```

Figure 7. A grid with fixed-size items

You are allowed to mix different size types to create more complex grid layouts. For example, you can define a fixed size `GridItem`, followed by a `GridItem` with an adaptive size like this:

```
private var gridItemLayout = [GridItem(.fixed(150)), GridItem(.adaptive(minimum: 50
))]
```

In this case, `LazyVGrid` creates a fixed size column of 100 point width. And then, it tries to fill as many items as possible within the remaining space.


SwiftUIGridLayout 〉 SwiftUIGridLayout 〉 ContentView.swift 〉 ContentView
Preview
88%

```
import SwiftUI

struct ContentView: View {

    private var symbols = ["keyboard", "hifispeaker.fill", "printer.fill",
        "tv.fill", "desktopcomputer", "headphones", "tv.music.note", "mic",
        "plus.bubble", "video"]

    private var colors: [Color] = [.yellow, .purple, .green]

    private var gridItemLayout = [GridItem(.fixed(150)),
        GridItem(.adaptive(minimum: 50))]

    var body: some View {
        ScrollView {
            LazyVGrid(columns: gridItemLayout, spacing: 20) {
                ForEach((0...9999), id: \.self) {
                    Image(systemName: symbols[$0 % symbols.count])
                        .font(.system(size: 30))
                        .frame(minWidth: 0, maxWidth: .infinity, minHeight: 50)
                        .background(colors[$0 % colors.count])
                        .cornerRadius(10)
                }
            }
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 8. Mixing a fixed-size item with adaptive size items

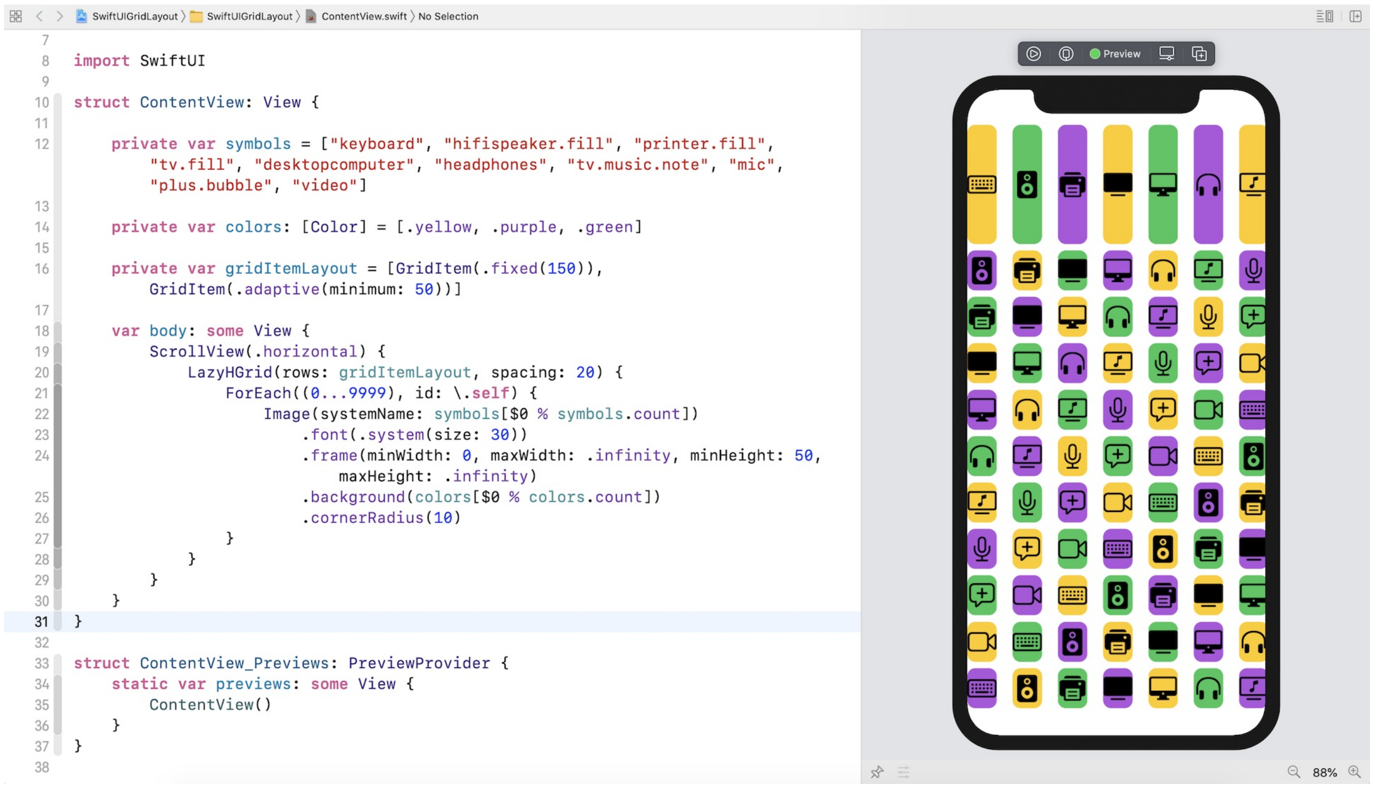
Using LazyHGrid to Create Horizontal Grids

Now that you've created a vertical grid, `LazyHGrid` has made it so easy to convert a vertical grid to a horizontal one. The usage of horizontal grid is nearly the same as `LazyVGrid` except that you embed it in a horizontal scroll view. Furthermore, `LazyHGrid` takes in a parameter named `rows` instead of `columns`.

Therefore, you can rewrite a couple lines of code to transform a grid view from vertical orientation to horizontal:

```
ScrollView(.horizontal) {
    LazyHGrid(rows: gridItemLayout, spacing: 20) {
        ForEach((0...9999), id: \.self) {
            Image(systemName: symbols[$0 % symbols.count])
                .font(.system(size: 30))
                .frame(minWidth: 0, maxWidth: .infinity, minHeight: 50, maxHeight:
 .infinity)
                .background(colors[$0 % colors.count])
                .cornerRadius(10)
        }
    }
}
```

Run the demo in the preview or test it on a simulator. You should see a horizontal grid.


SwiftUIGridLayout 〉 SwiftUIGridLayout 〉 ContentView.swift 〉 No Selection
Preview
88%

```swift
import SwiftUI

struct ContentView: View {

    private var symbols = ["keyboard", "hifispeaker.fill", "printer.fill",
        "tv.fill", "desktopcomputer", "headphones", "tv.music.note", "mic",
        "plus.bubble", "video"]

    private var colors: [Color] = [.yellow, .purple, .green]

    private var gridItemLayout = [GridItem(.fixed(150)),
        GridItem(.adaptive(minimum: 50))]

    var body: some View {
        ScrollView(.horizontal) {
            LazyHGrid(rows: gridItemLayout, spacing: 20) {
                ForEach((0...9999), id: \.self) {
                    Image(systemName: symbols[$0 % symbols.count])
                        .font(.system(size: 30))
                        .frame(minWidth: 0, maxWidth: .infinity, minHeight: 50,
                            maxHeight: .infinity)
                        .background(colors[$0 % colors.count])
                        .cornerRadius(10)
                }
            }
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 9. Creating a horizontal grid with LazyHGrid

Switching Between Different Grid Layouts

Now that you have some experience with LazyVGrid and LazyHGrid, let's create something more complicated. Imagine you are going to build a photo app that displays a collection of coffee photos. In the app, it provides a feature for users to change the layout. By default, it shows the list of photos in a single column. The user can tap a *Grid* button to change the list view to a grid view with 2 columns. Tap the same button again for a 3-column layout, followed by a 4-column layout.

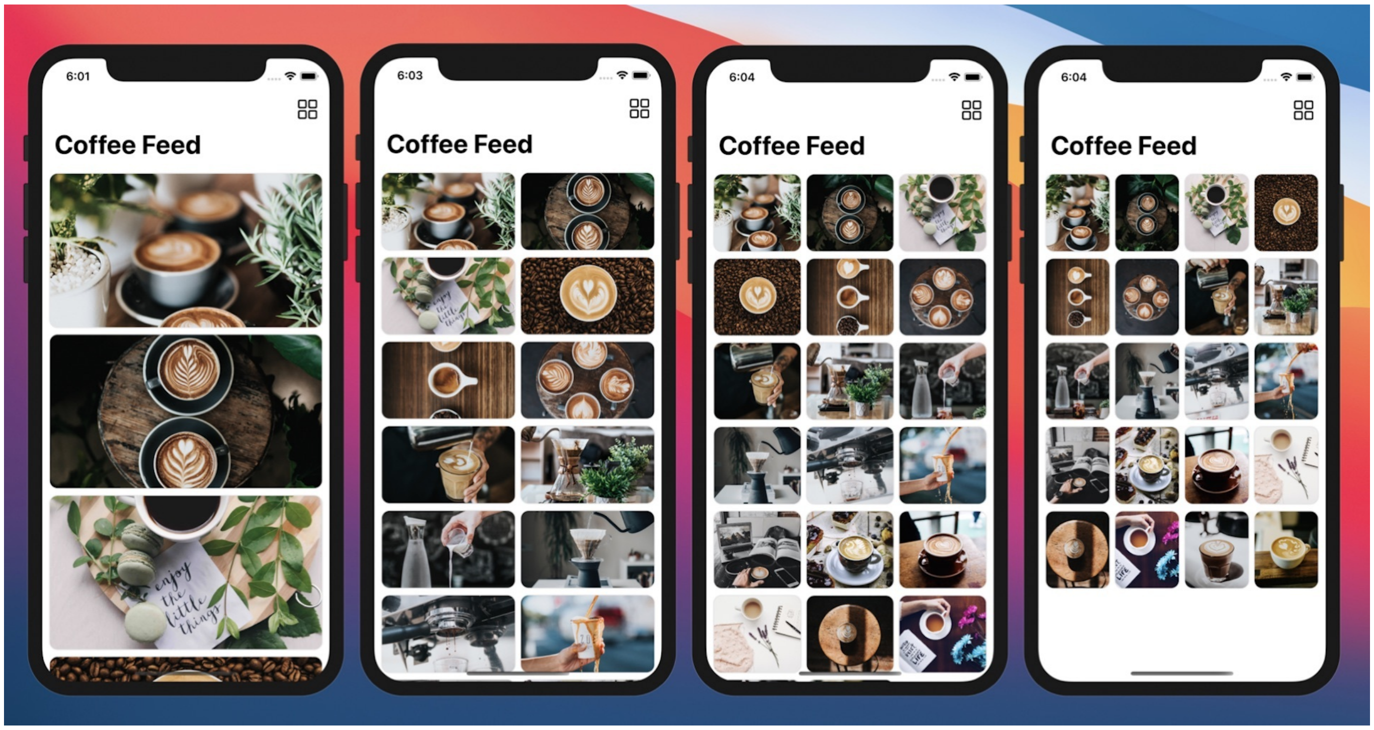


Figure 10. Creating a horizontal grid with LazyHGrid

Create a new project for this demo app. Again, choose the *App* template and name the project *SwiftUIPhotoGrid*. Next, download the image pack at https://www.appcoda.com/resources/swiftui/coffeeimages.zip. Unzip the images and add them to the asset catalog.

Before creating the grid view, we will create the data model for the collection of photos. In the project navigator, right click *SwiftUIGridView* and choose *New file...* to create a new file. Select the *Swift File* template and name the file *Photo.swift*.

Insert the following code in the `Photo.swift` file to create the `Photo` struct:

```
struct Photo: Identifiable {
    var id = UUID()
    var name: String
}

let samplePhotos = (1...20).map { Photo(name: "coffee-\($0)") }
```

We have 20 coffee photos in the image pack, so we initialize an array of 20 `Photo` instances. With the data model ready, let's switch over to `ContentView.swift` to build the grid.

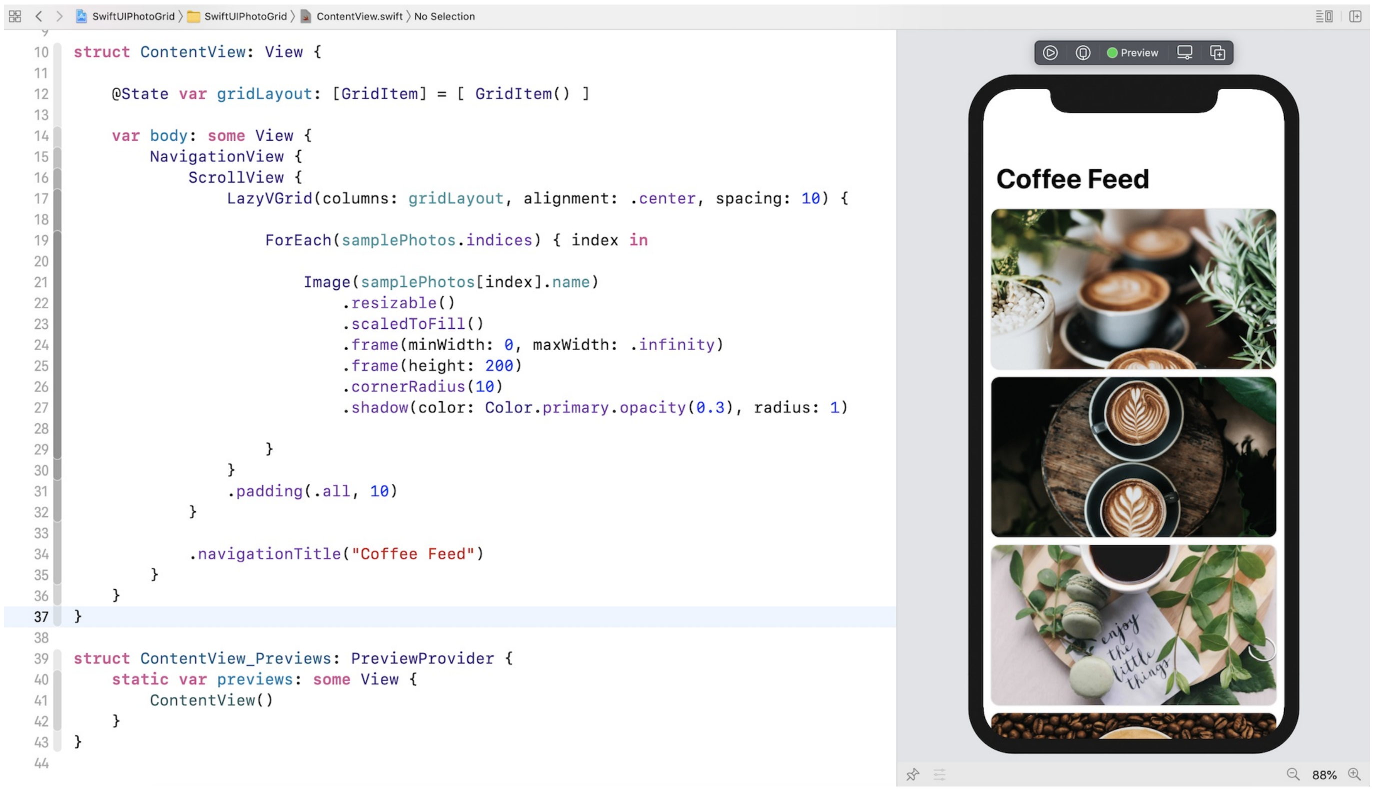
First, declare a `gridLayout` variable to define our preferred grid layout:

```
@State var gridLayout: [GridItem] = [ GridItem() ]
```

By default, we want to display a list view. Other than using `List`, you can actually use `LazyVGrid` to build a list view. We do this by defining the `gridLayout` with one grid item. By telling `LazyVGrid` to use a single column grid layout, it will arrange the items like a list view. Insert the following code in `body` to create the grid view:

```
NavigationView {
    ScrollView {
        LazyVGrid(columns: gridLayout, alignment: .center, spacing: 10) {

            ForEach(samplePhotos.indices) { index in

                Image(samplePhotos[index].name)
                    .resizable()
                    .scaledToFill()
                    .frame(minWidth: 0, maxWidth: .infinity)
                    .frame(height: 200)
                    .cornerRadius(10)
                    .shadow(color: Color.primary.opacity(0.3), radius: 1)

            }
        }
        .padding(.all, 10)
    }

    .navigationTitle("Coffee Feed")
}
```

We use `LazyVGrid` to create a vertical grid with a spacing of 10 points between rows. The grid is used to display coffee photos, so we use `ForEach` to loop through the `samplePhotos` array. We embed the grid in a scroll view to make it scrollable and wrap it with a navigation view. Once you have made the change, you should see a list of photos in the preview canvas.


SwiftUIPhotoGrid 〉 SwiftUIPhotoGrid 〉 ContentView.swift 〉 No Selection
Preview
Coffee Feed
88%

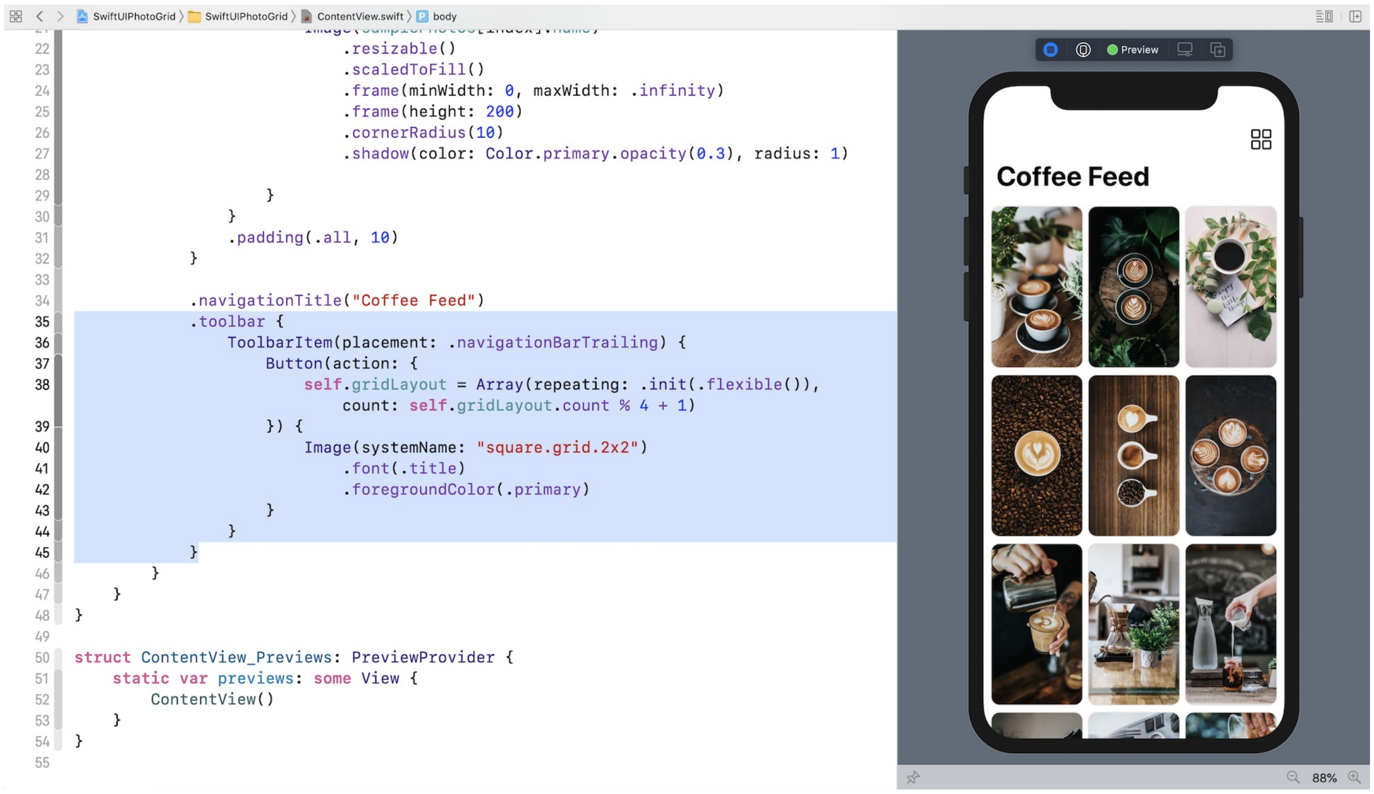
```
struct ContentView: View {

    @State var gridLayout: [GridItem] = [ GridItem() ]

    var body: some View {
        NavigationView {
            ScrollView {
                LazyVGrid(columns: gridLayout, alignment: .center, spacing: 10) {

                    ForEach(samplePhotos.indices) { index in

                        Image(samplePhotos[index].name)
                            .resizable()
                            .scaledToFill()
                            .frame(minWidth: 0, maxWidth: .infinity)
                            .frame(height: 200)
                            .cornerRadius(10)
                            .shadow(color: Color.primary.opacity(0.3), radius: 1)

                    }
                }
                .padding(.all, 10)
            }

            .navigationTitle("Coffee Feed")
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 11. Creating a list view with LazyVGrid

Now we need to a button for users to switch between different layouts. We will add the button to the navigation bar. In iOS 14, Apple introduced a new modifier called `.toolbar` for you to populate items within the navigation bar. Right after `.navigationTitle`, insert the following code to create the bar button:

```
.toolbar {
    ToolbarItem(placement: .navigationBarTrailing) {
        Button(action: {
            self.gridLayout = Array(repeating: .init(.flexible()), count: self.gri
dLayout.count % 4 + 1)
        }) {
            Image(systemName: "square.grid.2x2")
                .font(.title)
                .foregroundColor(.primary)
        }
    }
}
```

In the code above, we update the `gridLayout` variable and initialize the array of `GridItem`. Let's say the current item count is one, we will create an array of two `GridItem`s to change to a 2-column grid. Since we've marked `gridLayout` as a state variable, SwiftUI will render the grid view every time we update the variable.



```
                            .resizable()
                            .scaledToFill()
                            .frame(minWidth: 0, maxWidth: .infinity)
                            .frame(height: 200)
                            .cornerRadius(10)
                            .shadow(color: Color.primary.opacity(0.3), radius: 1)

                    }
                }
                .padding(.all, 10)
            }

            .navigationTitle("Coffee Feed")
            .toolbar {
                ToolbarItem(placement: .navigationBarTrailing) {
                    Button(action: {
                        self.gridLayout = Array(repeating: .init(.flexible()),
                            count: self.gridLayout.count % 4 + 1)
                    }) {
                        Image(systemName: "square.grid.2x2")
                            .font(.title)
                            .foregroundColor(.primary)
                    }
                }
            }
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 12. Adding a bar button for switching the grid layout

You can run the app to have a quick test. Tapping the grid button will switch to another grid layout.

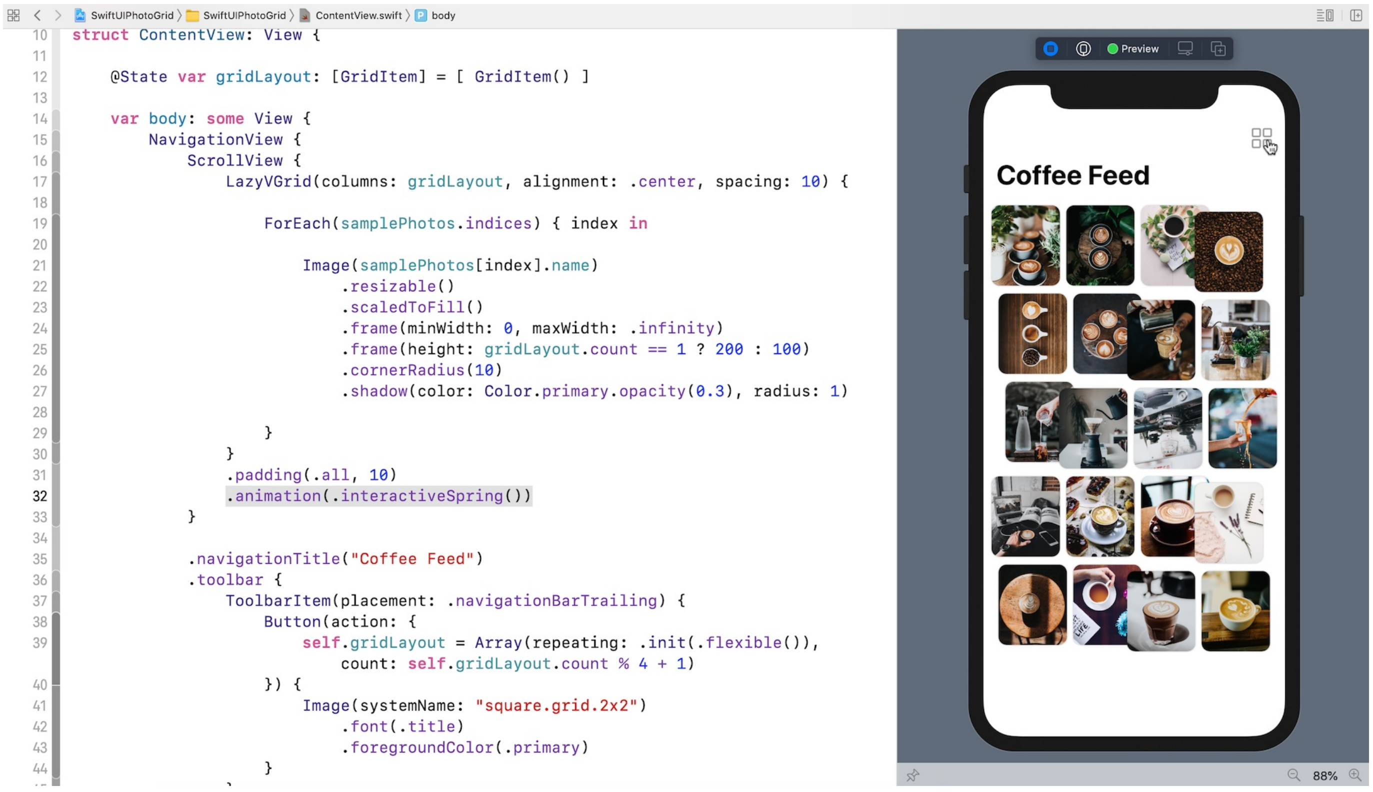
There are couple of things we want to improve. First, the height of the grid item should be adjusted to 100 points for grids with two or more columns. Update the `.frame` modifier with the `height` parameter like this:

```
.frame(height: gridLayout.count == 1 ? 200 : 100)
```

Second, when you switch from one grid layout to another, SwiftUI simply redraws the grid view without any animation. Wouldn't it be great if we added a nice transition between layout changes? To do that, you just add a single line of code. Insert the following code after `.padding(.all, 10)`:

```
.animation(.interactiveSpring())
```

This is the power of SwiftUI. By telling SwiftUI that you want to animate changes, the framework handles the rest and you will see a nice transition between the layout changes.


SwiftUIPhotoGrid 〉 SwiftUIPhotoGrid 〉 ContentView.swift 〉 body
Preview
Coffee Feed
88%

```
struct ContentView: View {

    @State var gridLayout: [GridItem] = [ GridItem() ]

    var body: some View {
        NavigationView {
            ScrollView {
                LazyVGrid(columns: gridLayout, alignment: .center, spacing: 10) {

                    ForEach(samplePhotos.indices) { index in

                        Image(samplePhotos[index].name)
                            .resizable()
                            .scaledToFill()
                            .frame(minWidth: 0, maxWidth: .infinity)
                            .frame(height: gridLayout.count == 1 ? 200 : 100)
                            .cornerRadius(10)
                            .shadow(color: Color.primary.opacity(0.3), radius: 1)

                    }
                }
                .padding(.all, 10)
                .animation(.interactiveSpring())
            }

            .navigationTitle("Coffee Feed")
            .toolbar {
                ToolbarItem(placement: .navigationBarTrailing) {
                    Button(action: {
                        self.gridLayout = Array(repeating: .init(.flexible()),
                            count: self.gridLayout.count % 4 + 1)
                    }) {
                        Image(systemName: "square.grid.2x2")
                            .font(.title)
                            .foregroundColor(.primary)
                    }
```

Figure 13. SwiftUI automatically animates the transition

Building Grid Layout with Multiple Grids

You are not limited to using a single `LazyVGrid` or `LazyHGrid` in your app. By combining more than one `LazyVGrid`, you are able to build some interesting layouts. Take a look at figure 14. We are going to create this kind of grid layout. The grid displays a list of cafe photos. Under each cafe photo, it shows a list of coffee photos. When the device is in landscape orientation, the cafe photo and the list of coffee photos will be arranged side by side.

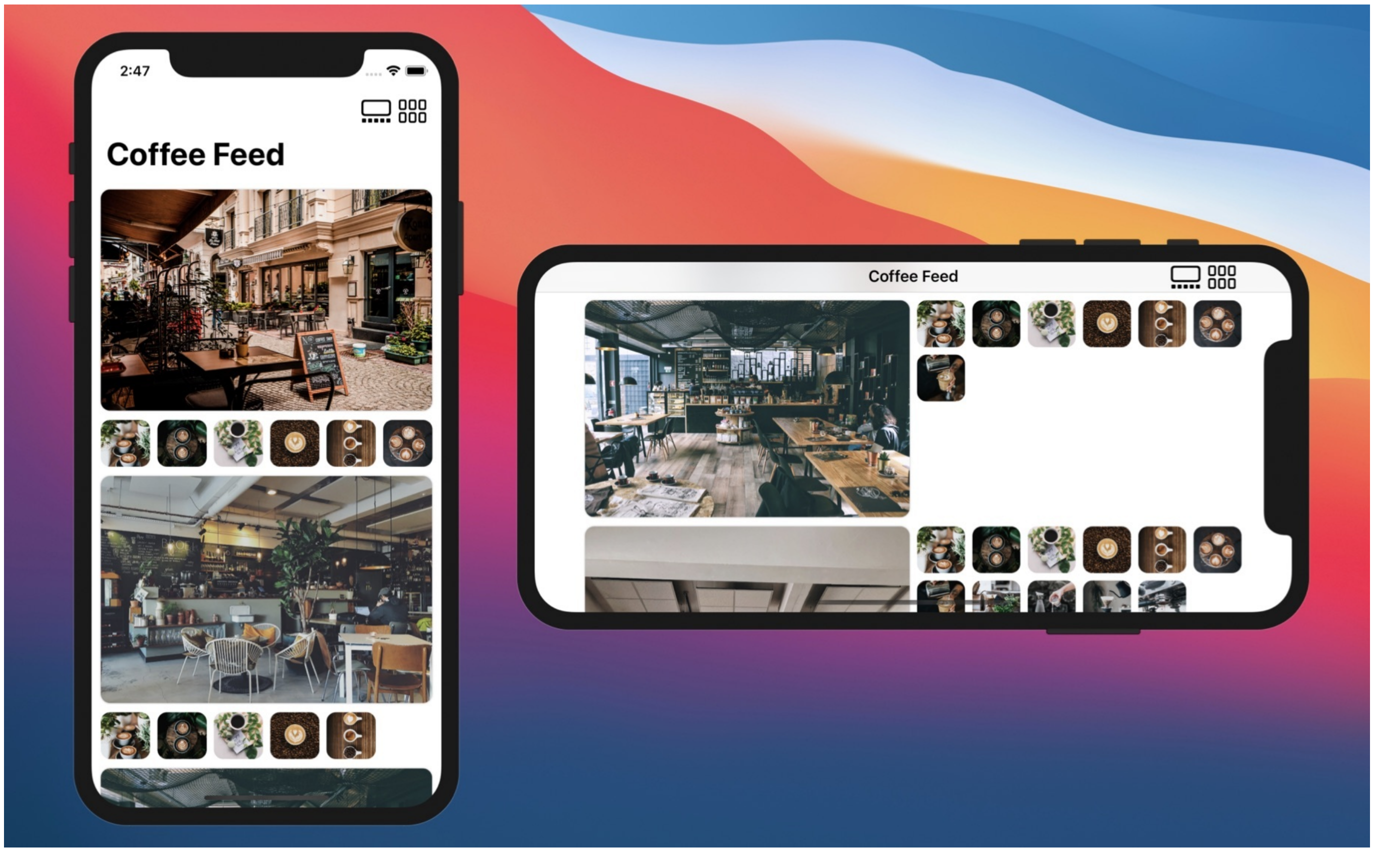


Figure 14. Building complex grid layout with two grids

Let's go back to our Xcode project and create the data model first. The image pack you downloaded earlier comes a set of cafe photos. So, create a new Swift file and name it *Cafe.swift*. In the file, insert the following code:

```
struct Cafe: Identifiable {
    var id = UUID()
    var image: String
    var coffeePhotos: [Photo] = []
}

let sampleCafes: [Cafe] = {

    var cafes = (1...18).map { Cafe(image: "cafe-\($0)") }

    for index in cafes.indices {
        let randomNumber = Int.random(in: (2...12))
        cafes[index].coffeePhotos = (1...randomNumber).map { Photo(name: "coffee-\
($0)") }
    }

    return cafes
}()
```

The `Cafe` struct is self explanatory. It has an `image` property for storing the cafe photo and the `coffeePhotos` property for storing a list of coffee photos. In the code above, we also create an array of `Cafe` for demo purposes. For each cafe, we randomly pick some coffee photos. Please feel free to modify the code if you have other images you prefer.

Instead of modifying the `ContentView.swift` file, let's create a new file for implementing this grid view. Right click *SwiftUIPhotoGrid* and choose *New File...*. Select the *SwiftUI View* template and name the file *MultiGridView*.

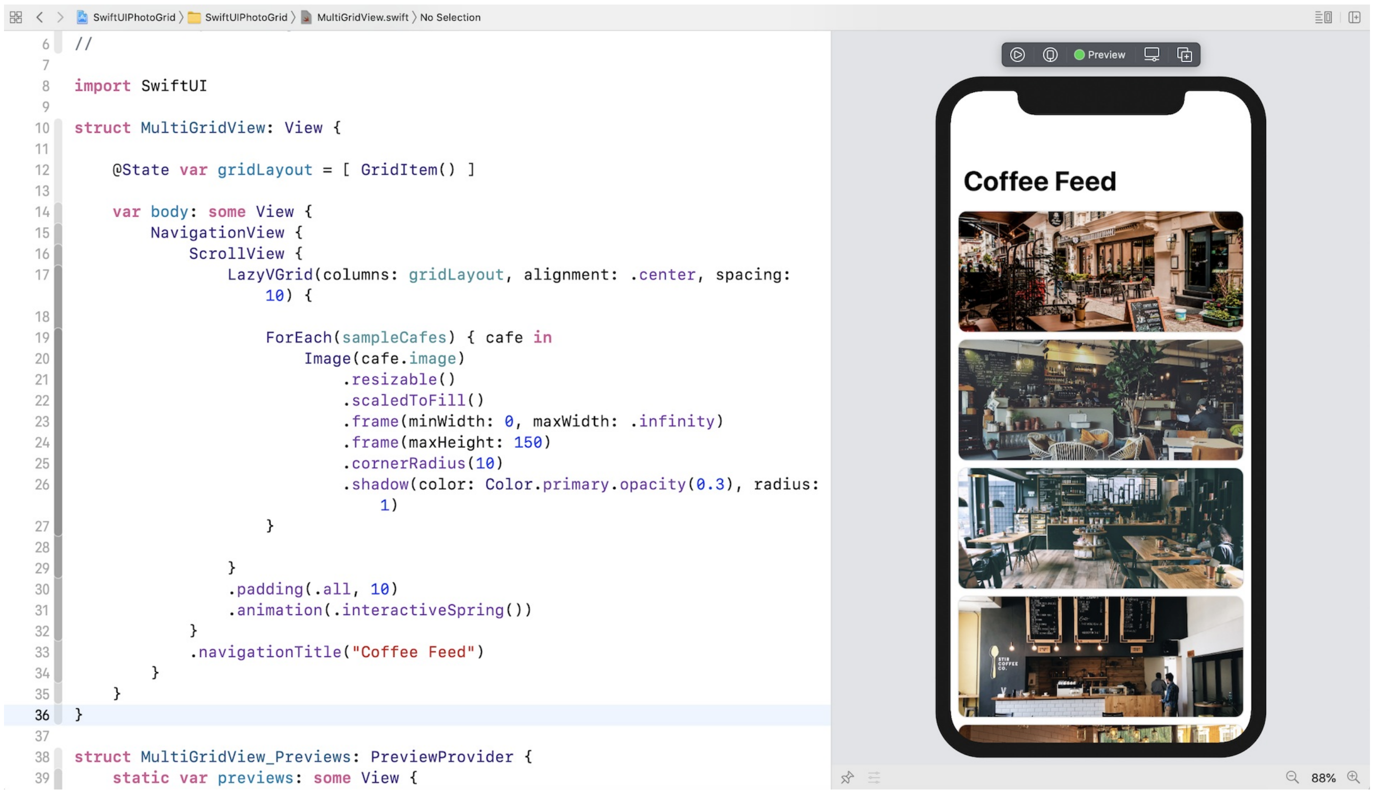
Similar to the earlier implementation, let's declare a `gridLayout` variable to store the current grid layout:

```
@State var gridLayout = [ GridItem() ]
```

By default, our grid is initialized with one `GridItem`. Next, insert the following code in `body` to create a vertical grid with a single column:

```
NavigationView {
    ScrollView {
        LazyVGrid(columns: gridLayout, alignment: .center, spacing: 10) {

            ForEach(sampleCafes) { cafe in
                Image(cafe.image)
                    .resizable()
                    .scaledToFill()
                    .frame(minWidth: 0, maxWidth: .infinity)
                    .frame(maxHeight: 150)
                    .cornerRadius(10)
                    .shadow(color: Color.primary.opacity(0.3), radius: 1)
            }

        }
        .padding(.all, 10)
        .animation(.interactiveSpring())
    }
    .navigationTitle("Coffee Feed")
}
```

I don't think we need to go through the code again because it's almost the same as the code we wrote earlier. If your code works properly, you should see a list view that shows the collection of cafe photos.


SwiftUIPhotoGrid 〉 SwiftUIPhotoGrid 〉 MultiGridView.swift 〉 No Selection
Preview
Coffee Feed
88%

```
//

import SwiftUI

struct MultiGridView: View {

    @State var gridLayout = [ GridItem() ]

    var body: some View {
        NavigationView {
            ScrollView {
                LazyVGrid(columns: gridLayout, alignment: .center, spacing:
                    10) {

                    ForEach(sampleCafes) { cafe in
                        Image(cafe.image)
                            .resizable()
                            .scaledToFill()
                            .frame(minWidth: 0, maxWidth: .infinity)
                            .frame(maxHeight: 150)
                            .cornerRadius(10)
                            .shadow(color: Color.primary.opacity(0.3), radius:
                                1)
                    }

                }
                .padding(.all, 10)
                .animation(.interactiveSpring())
            }
            .navigationTitle("Coffee Feed")
        }
    }
}

struct MultiGridView_Previews: PreviewProvider {
    static var previews: some View {
```

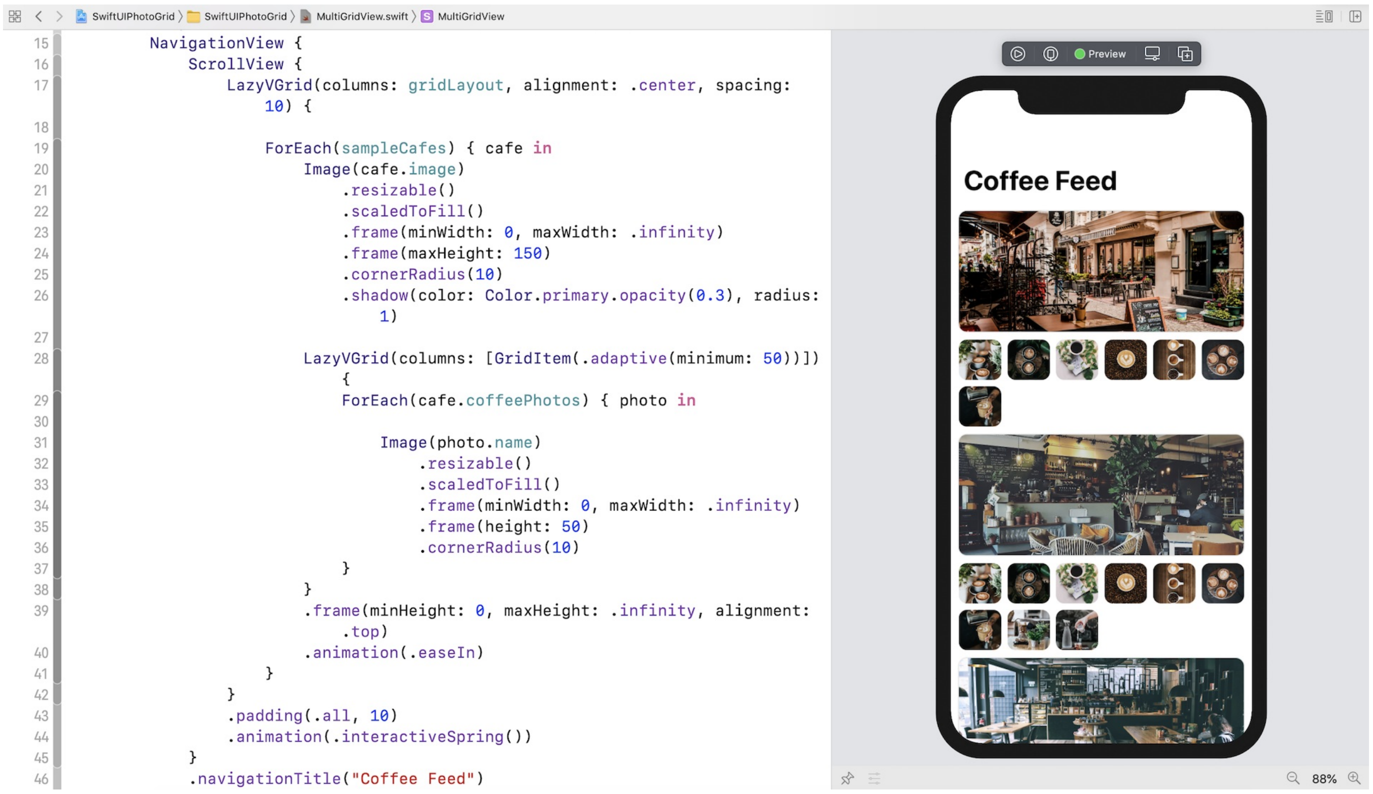
Figure 15. A list of cafe photos

Adding an Additional Grid

How do we display another grid under each of the cafe photos? All you need to do is add another `LazyVGrid` inside the `ForEach` loop. Insert the following code after the `Image` view of the loop:

```
LazyVGrid(columns: [GridItem(.adaptive(minimum: 50))]) {
    ForEach(cafe.coffeePhotos) { photo in
        Image(photo.name)
            .resizable()
            .scaledToFill()
            .frame(minWidth: 0, maxWidth: .infinity)
            .frame(height: 50)
            .cornerRadius(10)
    }
}
.frame(minHeight: 0, maxHeight: .infinity, alignment: .top)
.animation(.easeIn)
```

Here we create another vertical grid for the coffee photos. By using the *adaptive* size type, this grid will fill as many photos as possible in a row. Once you make the code change, the app UI will look like that shown in figure 16.


SwiftUIPhotoGrid 〉 SwiftUIPhotoGrid 〉 MultiGridView.swift 〉 MultiGridView
Preview
Coffee Feed
88%

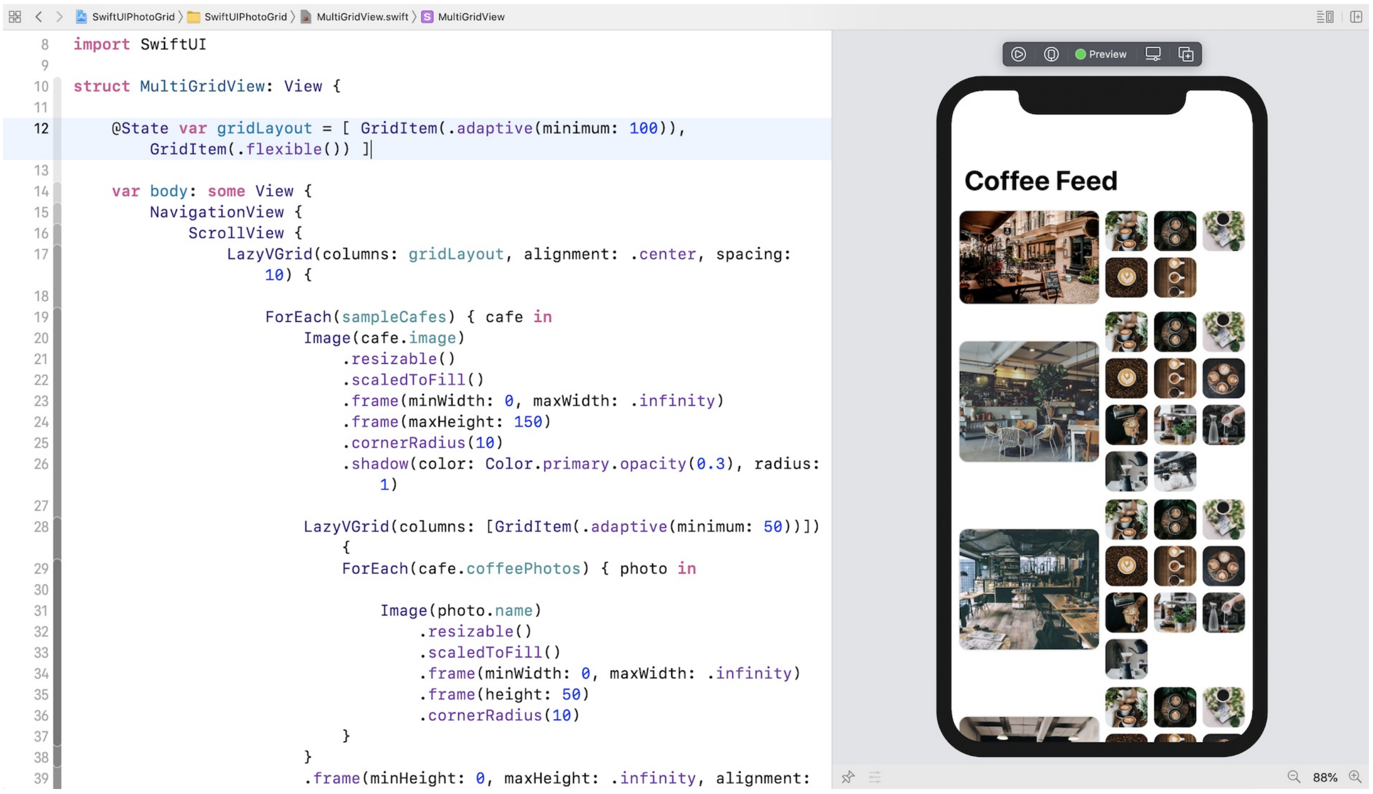
```
        NavigationView {
            ScrollView {
                LazyVGrid(columns: gridLayout, alignment: .center, spacing:
                    10) {

                    ForEach(sampleCafes) { cafe in
                        Image(cafe.image)
                            .resizable()
                            .scaledToFill()
                            .frame(minWidth: 0, maxWidth: .infinity)
                            .frame(maxHeight: 150)
                            .cornerRadius(10)
                            .shadow(color: Color.primary.opacity(0.3), radius:
                                1)

                        LazyVGrid(columns: [GridItem(.adaptive(minimum: 50))])
                            {
                            ForEach(cafe.coffeePhotos) { photo in

                                Image(photo.name)
                                    .resizable()
                                    .scaledToFill()
                                    .frame(minWidth: 0, maxWidth: .infinity)
                                    .frame(height: 50)
                                    .cornerRadius(10)
                            }
                        }
                        .frame(minHeight: 0, maxHeight: .infinity, alignment:
                            .top)
                        .animation(.easeIn)
                    }
                }
                .padding(.all, 10)
                .animation(.interactiveSpring())
            }
            .navigationTitle("Coffee Feed")
```

Figure 16. Adding another grid for the coffee photos

If you prefer to arrange the cafe and coffee photos side by side, you can modify the `gridLayout` variable like this:

```
@State var gridLayout = [ GridItem(.adaptive(minimum: 100)), GridItem(.flexible())
 ]
```

As soon as you change the `gridLayout` variable, your preview will be updated to display the cafe and coffee photos side by side.



```
import SwiftUI

struct MultiGridView: View {

    @State var gridLayout = [ GridItem(.adaptive(minimum: 100)),
        GridItem(.flexible()) ]

    var body: some View {
        NavigationView {
            ScrollView {
                LazyVGrid(columns: gridLayout, alignment: .center, spacing:
                    10) {

                    ForEach(sampleCafes) { cafe in
                        Image(cafe.image)
                            .resizable()
                            .scaledToFill()
                            .frame(minWidth: 0, maxWidth: .infinity)
                            .frame(maxHeight: 150)
                            .cornerRadius(10)
                            .shadow(color: Color.primary.opacity(0.3), radius:
                                1)

                        LazyVGrid(columns: [GridItem(.adaptive(minimum: 50))])
                            {
                            ForEach(cafe.coffeePhotos) { photo in

                                Image(photo.name)
                                    .resizable()
                                    .scaledToFill()
                                    .frame(minWidth: 0, maxWidth: .infinity)
                                    .frame(height: 50)
                                    .cornerRadius(10)
                            }
                        }
                        .frame(minHeight: 0, maxHeight: .infinity, alignment:
```

Figure 17. Arrange the cafe and coffee photos side by side

Handling Landscape Orientation

To test the app in landscape orientation, you need to run it on a simulator. The preview canvas doesn't allow you to rotate the device yet.

Before you run the app, you will need to perform a simple modification in `SwiftUIPhotoGridApp.swift`. Since we have created a new file for implementing this multi-grid, modify the view in `WindowGroup` from `ContentView()` to `MultiGridView()` like below:

```
struct SwiftUIPhotoGridApp: App {
    var body: some Scene {
        WindowGroup {
            MultiGridView()
        }
    }
}
```

Now you're ready to run the app in an iPhone simulator. It works great in the portrait orientation just like the preview canvas. However, if you rotate the simulator sideways by pressing command-left (or right), the grid layout doesn't look as expected. What we expect is that it should look pretty much the same as that in portrait mode.

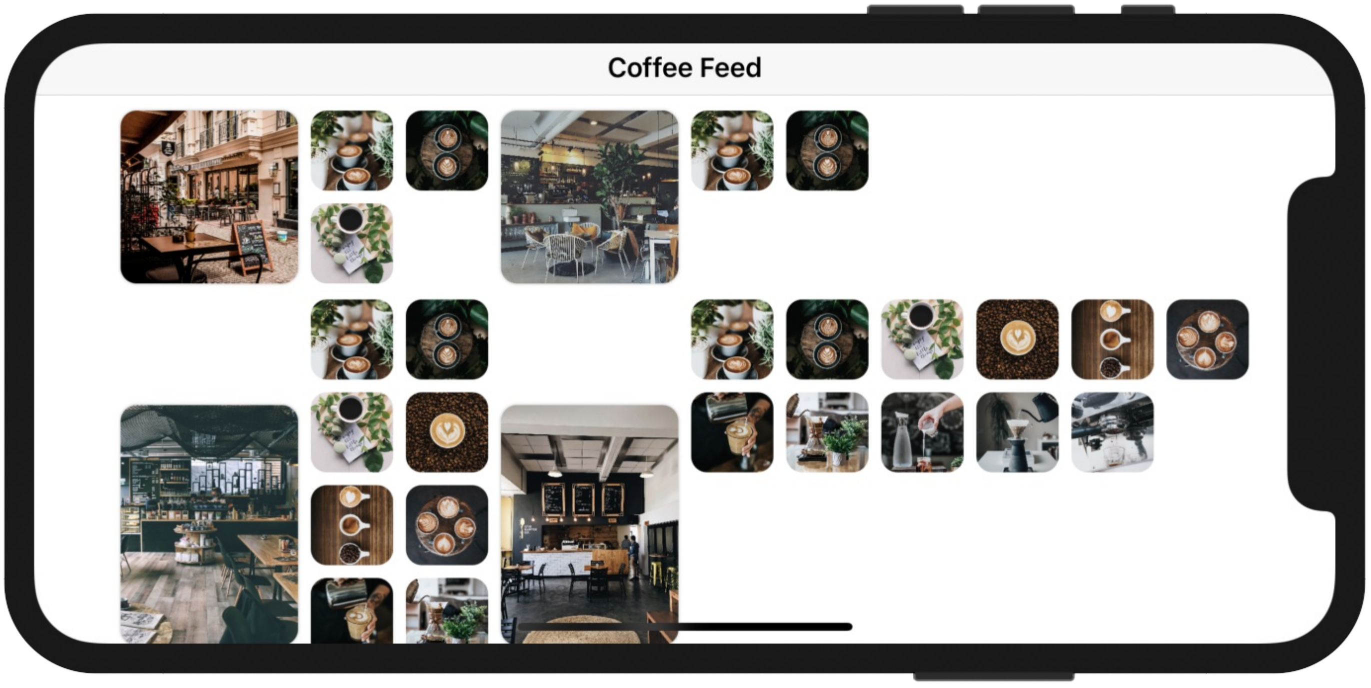


Figure 18. The app UI in landscape mode

To fix the issue, we can adjust the minimum width of the adaptive grid item and make it a bit wider when the device is in landscape orientation. The question is how can you detect the orientation changes?

In SwiftUI, every view comes with a set of environment variables. You can find out the current device orientation by accessing both the horizontal and vertical size class variables like this:

```
@Environment(\.horizontalSizeClass) var horizontalSizeClass: UserInterfaceSizeClass
?
@Environment(\.verticalSizeClass) var verticalSizeClass: UserInterfaceSizeClass?
```

The `@Environment` property wrapper allows you to access the environment values. In the code above, we tell SwiftUI that we want to read both the horizontal and vertical size classes, and subscribe to their changes. In other words, we will be notified whenever the device's orientation changes.

If you haven't done so, please make sure you insert the code above in `MultiGridView`.

The next question is how do we capture the notification and respond to the changes? In iOS 14, Apple introduced a new modifier called `.onChange()`. You can attach this modifier to any view to monitor any state changes. In this case, we can attach the modifier to `NavigationView` like this:

```
.onChange(of: verticalSizeClass) { value in
    self.gridLayout = [ GridItem(.adaptive(minimum:  verticalSizeClass == .compact
  ? 250 : 100)), GridItem(.flexible()) ]
}
```

We monitor the change of both `horizontalSizeClass` and `verticalSizeClass` variables. Whenever there is a change, we will update the `gridLayout` variable with a new grid configuration. The iPhone has a *compact* height in landscape orientation. Therefore, if the value of `verticalSizeClass` equals `.compact`, we alter the minimum size of the grid item to 250 points.

Now run the app on an iPhone simulator again. When you turn the device sideways, it now shows the cafe photo and coffee photos side by side.

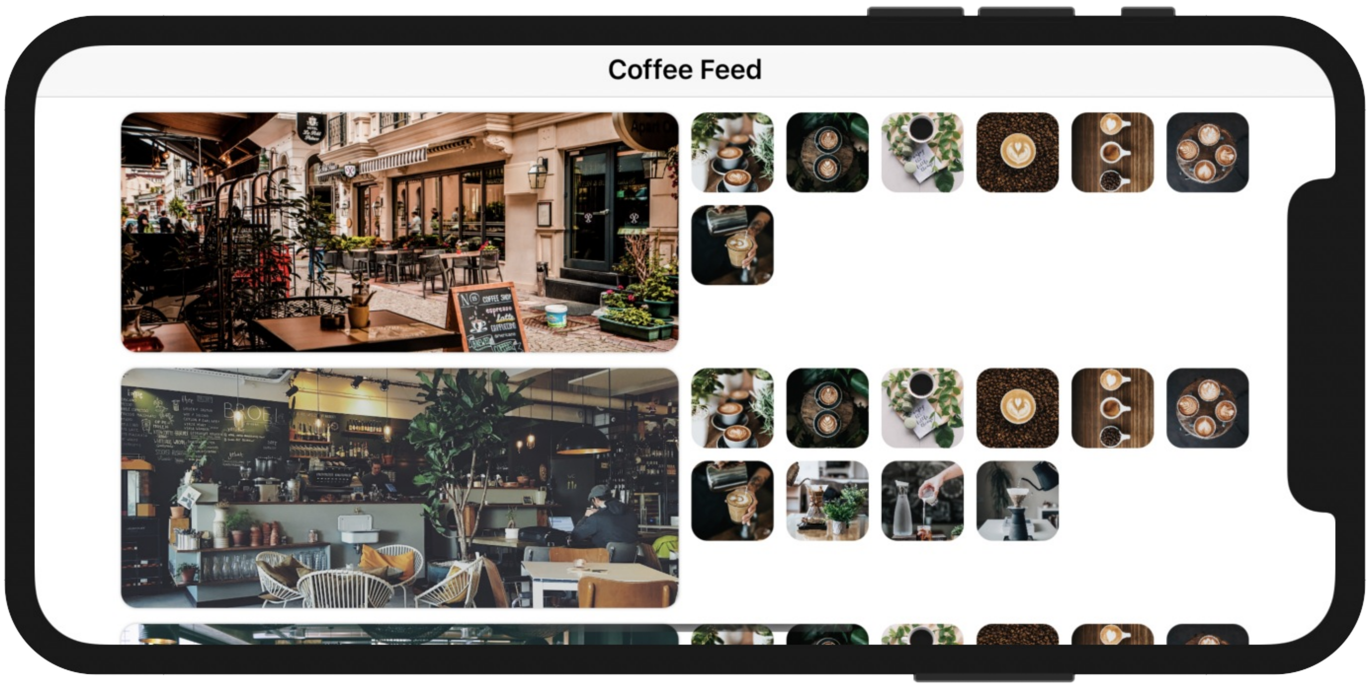


Figure 19. The app UI in landscape mode now looks better

Understanding Navigation View Style

Earlier, I used the iPhone 11 Pro as the simulator. The app works perfectly in both portrait and landscape mode. But when you run the app on iPhone Max models, it looks a bit weird in landscape orientation. iOS automatically turns the view into a primary detail split view. You can only access the grid view by tapping a navigation button at the top-left corner of the screen.

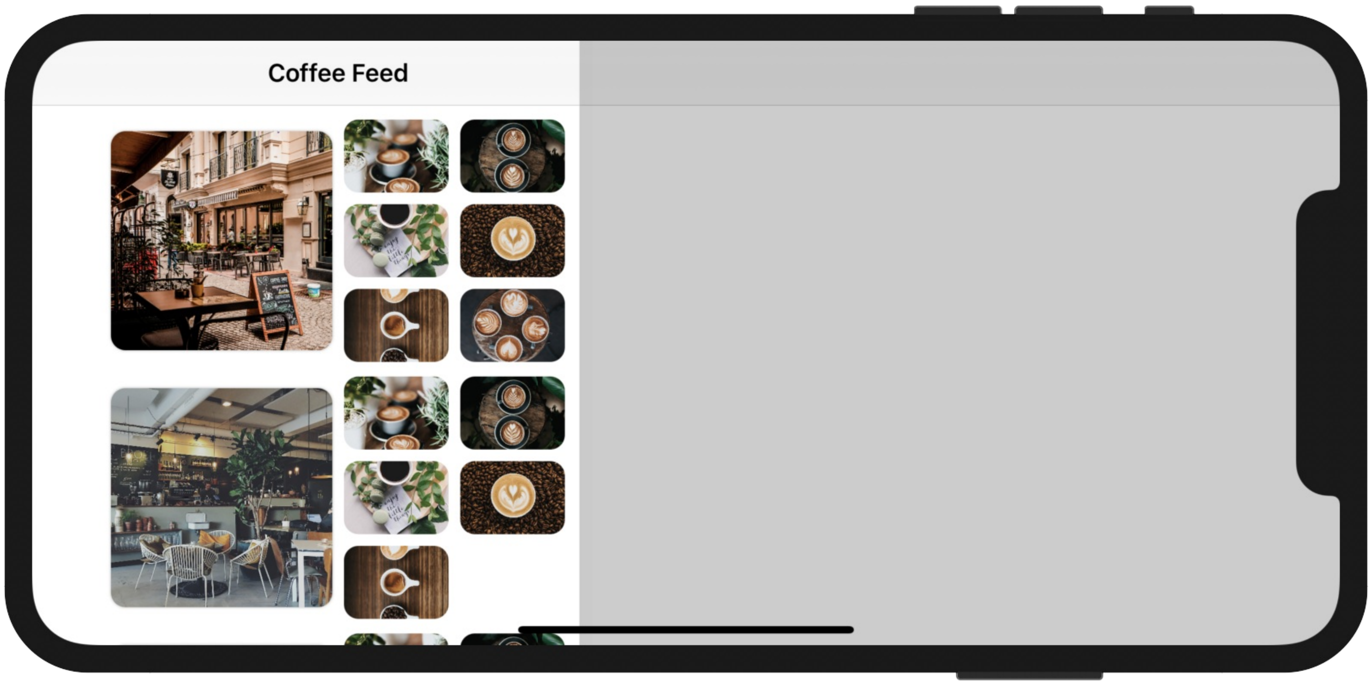


Figure 20. Split view on iPhone 11 Max

This is a default behavior of `NavigationView` on large devices like the iPhone 11 Pro Max. To solve the issue and disable the split-view behavior on iPhone Max models, you can attach the following modifier to the navigation view:

```
.navigationViewStyle(StackNavigationViewStyle())
```

By using the `.navigationViewStyle` modifier, we explicitly instruct `NavigationView` to use the stack navigation view style, regardless of the screen size. Now test the app again on iPhone 11 Max Pro. The app UI should look well even in landscape orientation just like that shown in figure 19.

Exercise

I have a couple of exercises for you. First, the app UI doesn't look good on iPad. Modify the code and fix the issue such that it only shows two columns: one for the cafe photo and the other for the coffee photos.

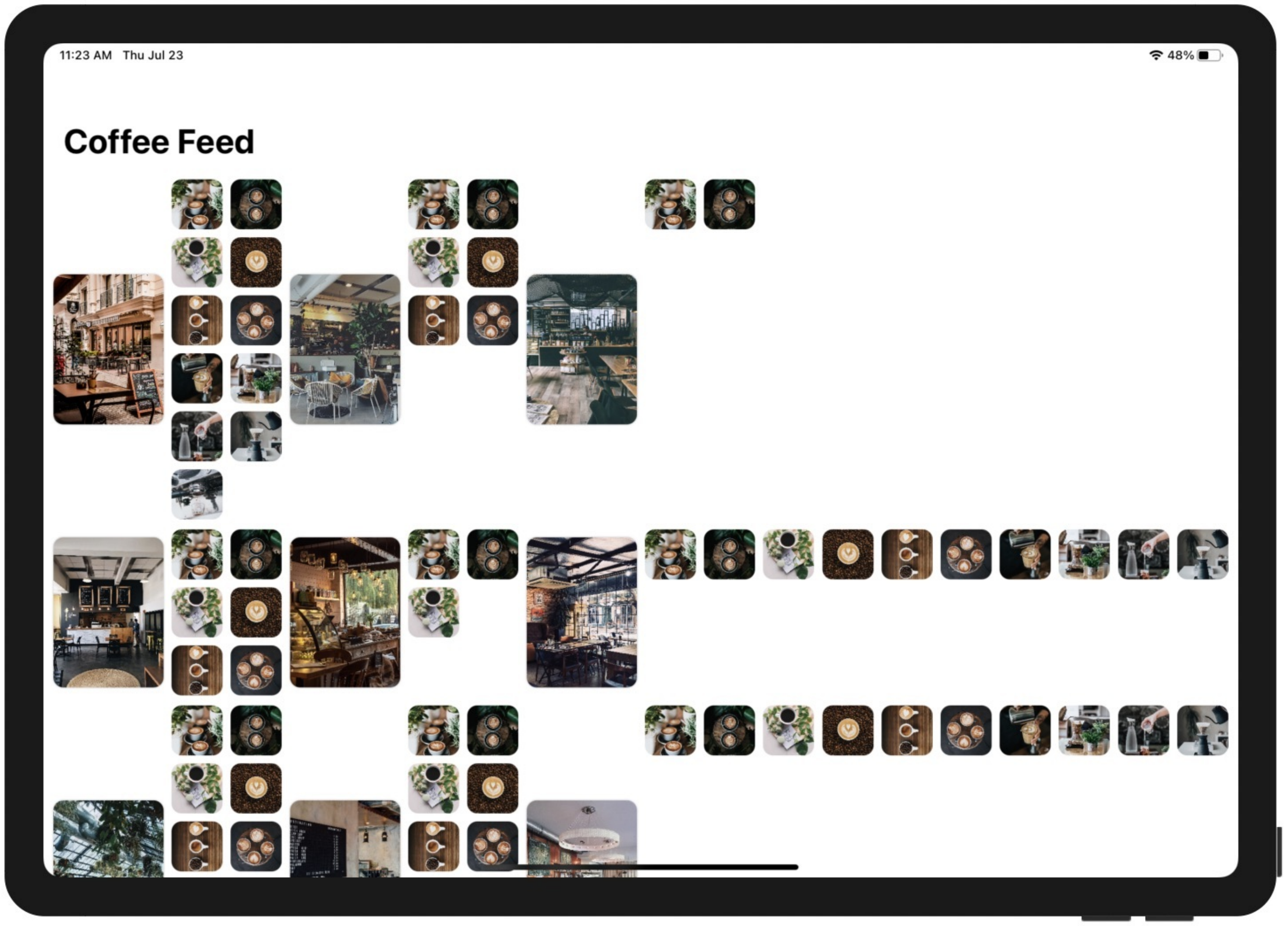


Figure 21. App UI on iPad

The next exercise is more complicated with a number of requirements:

1. **Different default grid layout for iPhone and iPad** - When the app is first loaded up, it displays a single column grid for iPhone in portrait mode. For iPad and iPhone landscape, the app shows the cafe photos in a 2-column grid.
2. **Show/hide button for the coffee photos** - Add a new button in the navigation bar for toggling the display of coffee photos. By default, the app only shows the list of cafe photos. When this button is tapped, it shows the coffee photo grid.
3. **Another button for switching grid layout** - Add another bar button for toggling the grid layout between one and two columns.

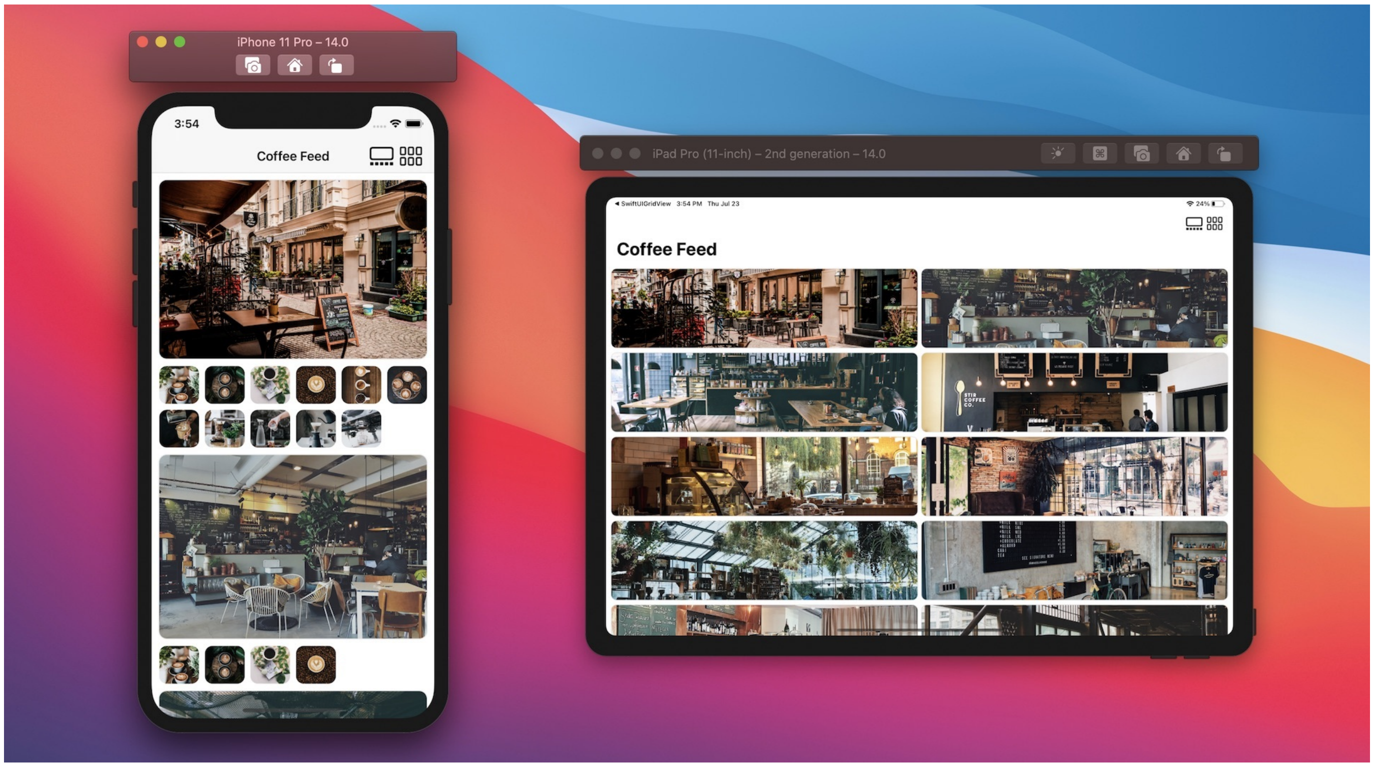


Figure 22. Enhancing the app to support both iPhone and iPad

To help you better understand what the final deliverable looks like, please check out this video demo at https://link.appcoda.com/multigrid-demo.

Summary

The missing collection view in the first release of SwiftUI is now here. The introduction of `LazyVGrid` and `LazyHGrid` in SwiftUI lets developers create different types of grid layouts with a few lines of code. This tutorial is just a quick overview of these two new UI components. I encourage you to try out different configurations of `GridItem` to see what grid layouts you can achieve.

For reference, you can download the complete grid project and the solution to the exercise here:

- Grid Layout Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUIGridLayout.zip)

- PhotoGrid Demo project
(https://www.appcoda.com/resources/swiftui2/SwiftUIPhotoGrid.zip)

Chapter 30
Creating an Animated Activity Ring with Shape and Animatable

The built-in Activity app uses three circular progress bars to show your progress of *Move*, *Exercise*, and *Stand*. This kind of progress bar is known as an activity ring. Take a look at figure 1 if you haven't used the Activity app or you don't know what an activity ring is. Apple Watch has played a big part in making this round progress bar a popular UI pattern.

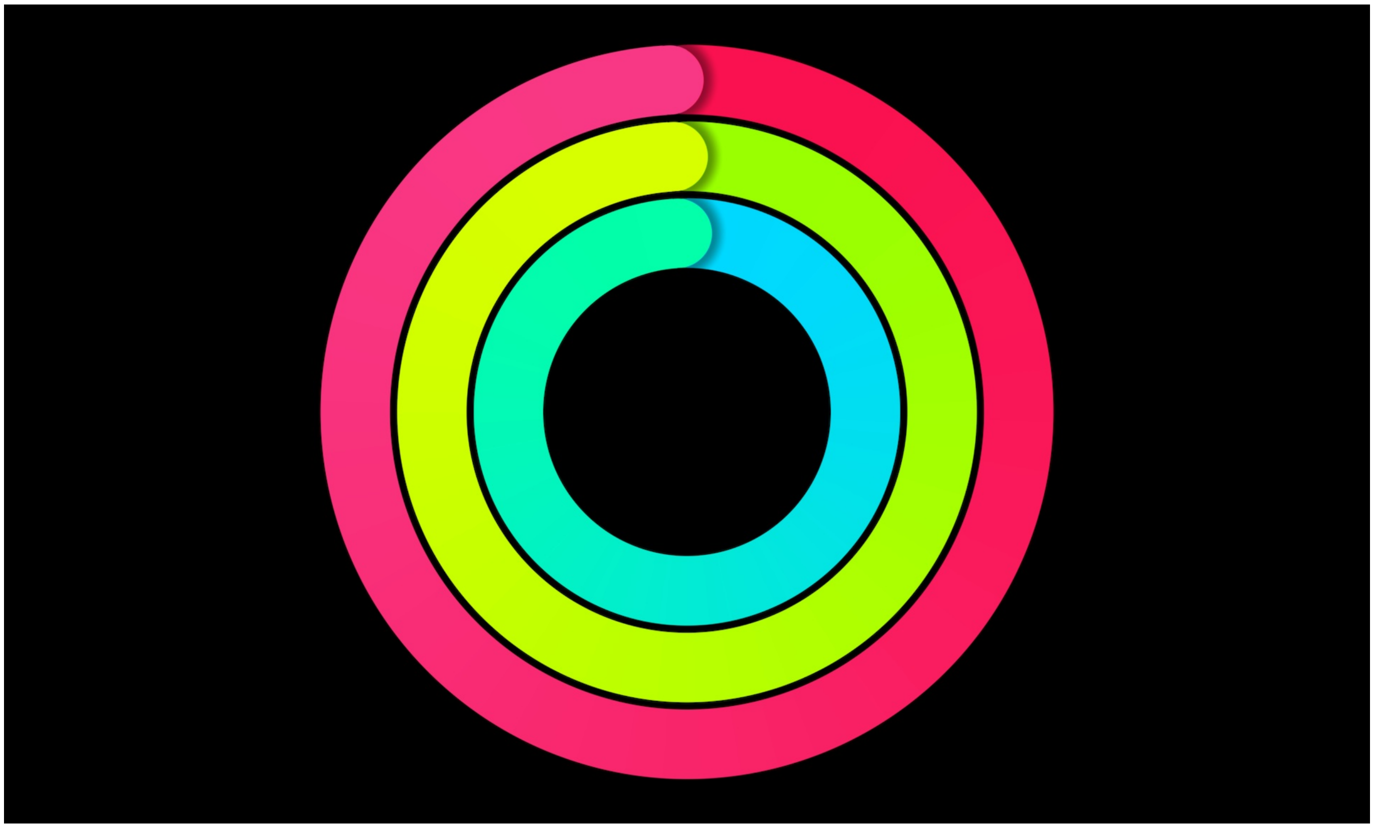


Figure 1. A sample activity ring

In this chapter, we will look into its implementation and build a similar activity ring in SwiftUI. Our goal is not just to create a static activity ring. It will be an animated one that shows progress changes like that shown in figure 2. Or you can check out the demo video at https://link.appcoda.com/progressring.

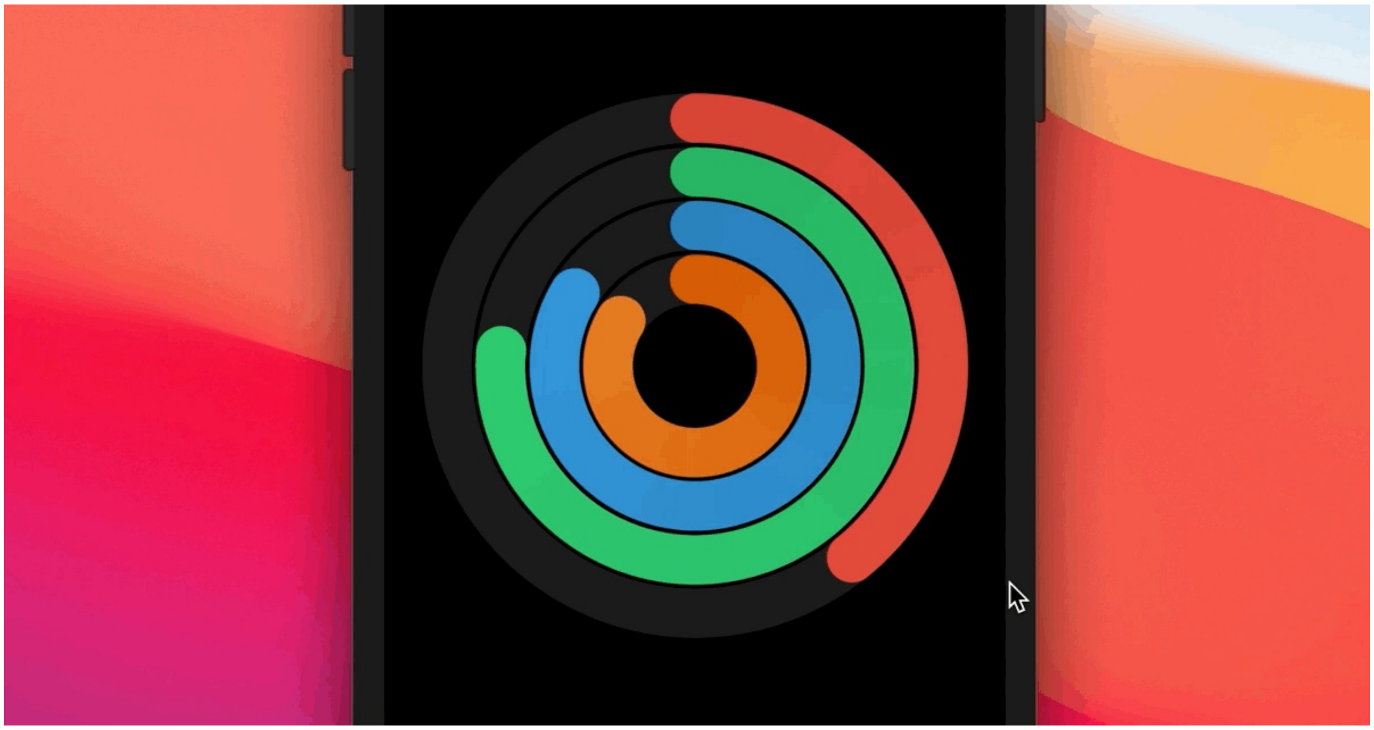


Figure 2. Animated progress ring

Creating a New Project

Let's create a new project to build the progress indicator. As usual, you use the *App* template for the project. Name it *SwiftUIProgressRing* or whatever name you like.

Make sure you use *SwiftUI* for the *Interface* option and *SwiftUI App* for the *Life Cycle* option.

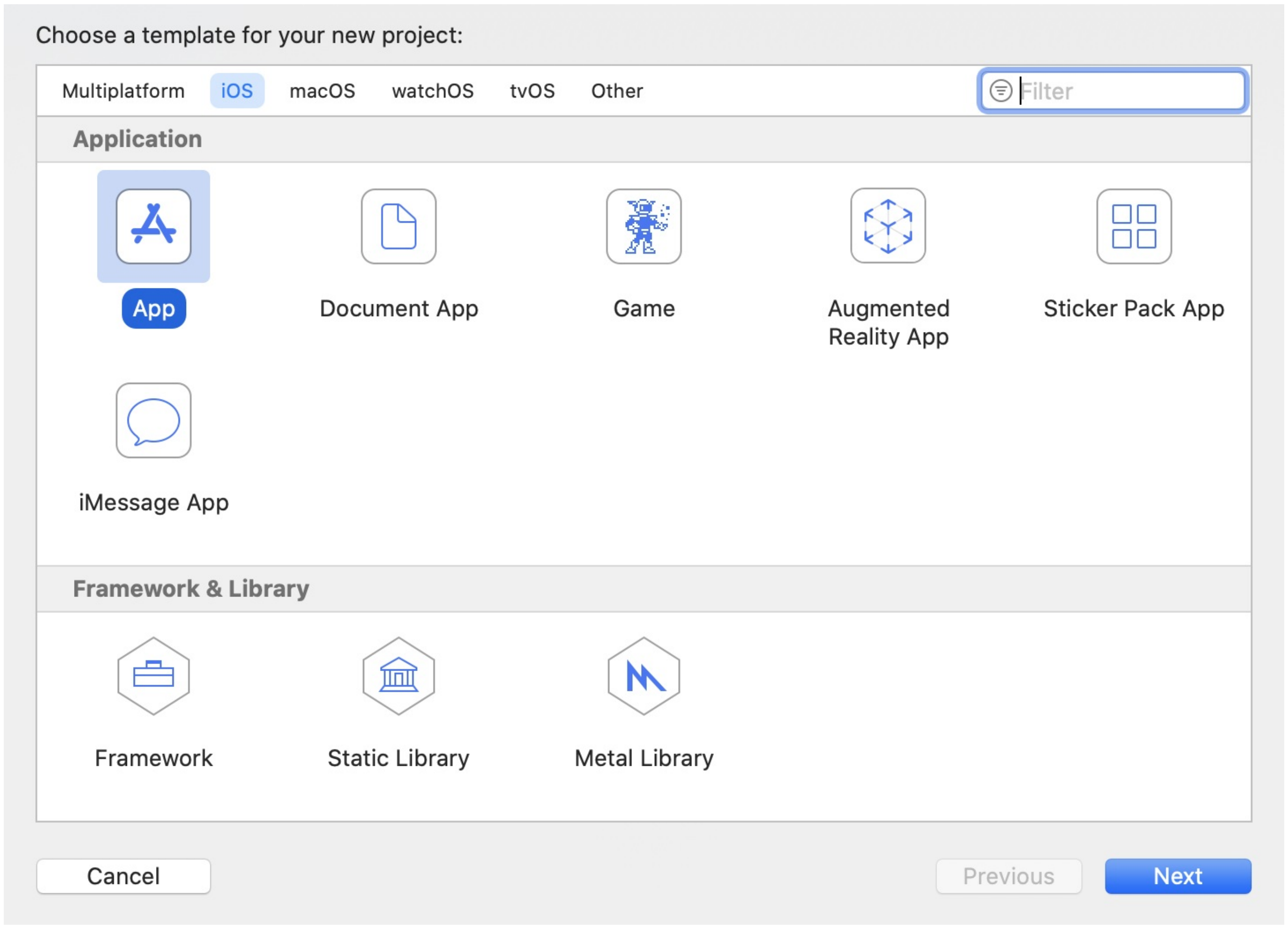


Figure 3. Creating a new project with the App template

To organize our code better, create a new file named `ProgressRingView.swift` by using the *SwiftUI View* template. Xcode should generate the following code after the file creation:

```
import SwiftUI

struct ProgressRingView: View {
    var body: some View {
        Text("Hello, World!")
    }
}

struct ProgressRingView_Previews: PreviewProvider {
    static var previews: some View {
        ProgressRingView()
    }
}
```

Dissecting the Activity Ring

Before we dive into the implementation, take a look at figures 1 and 2 again. You should find that an activity ring is actually composed of two or more circular progress bars. So, what we need to build is a circular progress bar view and that should be flexible enough to display a certain percentage value and allow the user to adjust the bar width and color.

For example, if you tell the bar view to display 60% progress in red and set its width to 250 points. The circular progress view should show something like this:

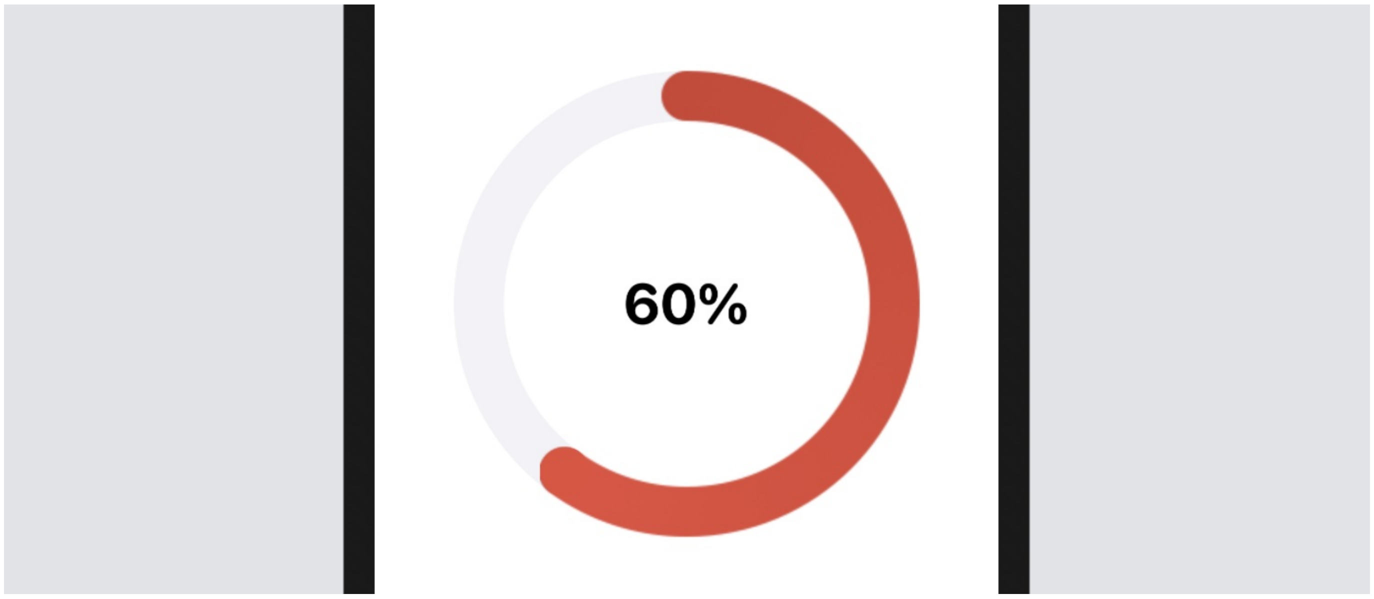


Figure 4. A sample circular progress bar

By building a flexible circular progress bar view, it is very easy to create an activity ring. For example, we can overlay another circular progress bar with bigger size & different color on top of the one shown in figure 4 to become an activity ring.

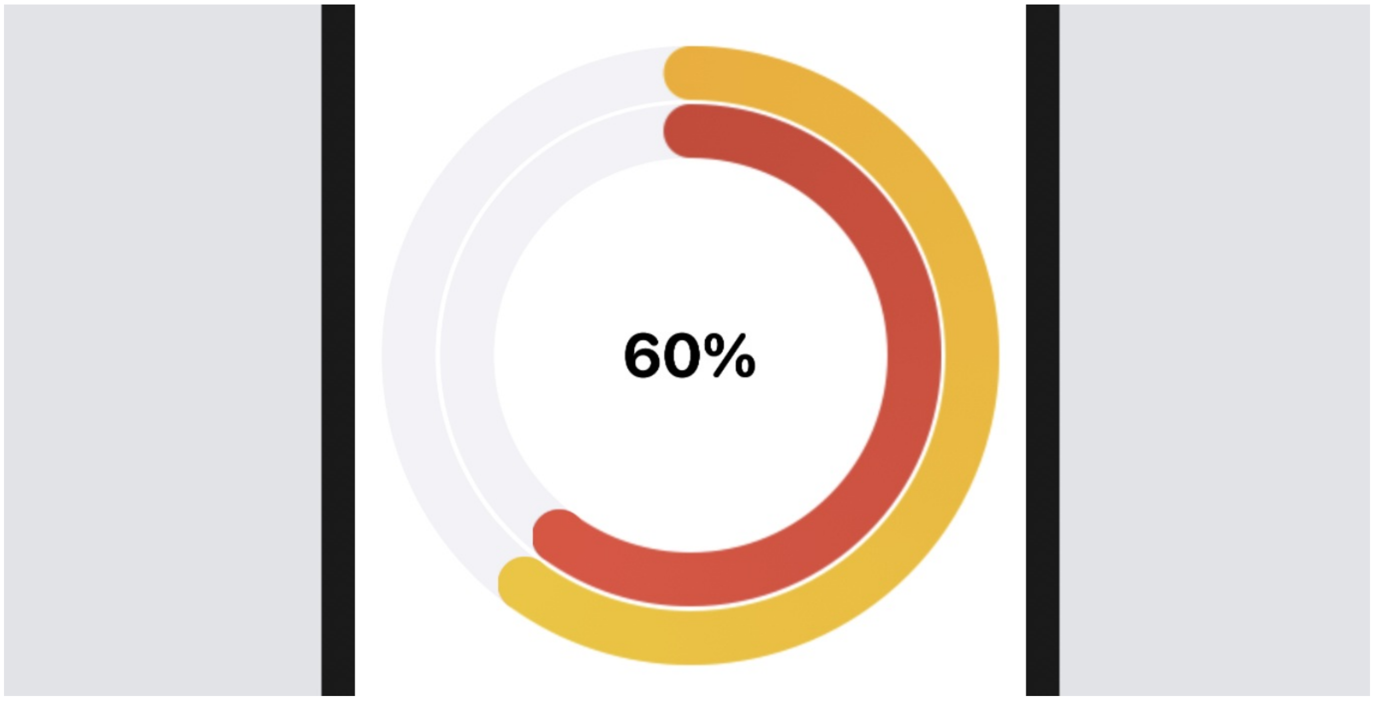


Figure 5. A sample circular progress bar

That's how we are going to build the activity ring. Now let's begin to implement the circular progress bar.

Preparing the Color Extension

As mentioned, the circular progress bar that we are going to implement can support multiple colors and gradients. For demo and convenience purposes, we will prepare a set of default colors by using a `Color` extension. In the project navigator, right click *SwiftUIProgressRing* and choose *New file...*. Select the *Swift file* template and name the file `Color+Ext.swift`. Replace the file content with the following code:

```
import SwiftUI

extension Color {

    public init(red: Int, green: Int, blue: Int, opacity: Double = 1.0) {
        let redValue = Double(red) / 255.0
        let greenValue = Double(green) / 255.0
        let blueValue = Double(blue) / 255.0

        self.init(red: redValue, green: greenValue, blue: blueValue, opacity: opac
ity)
    }

    public static let lightRed = Color(red: 231, green: 76, blue: 60)
    public static let darkRed = Color(red: 192, green: 57, blue: 43)
    public static let lightGreen = Color(red: 46, green: 204, blue: 113)
    public static let darkGreen = Color(red: 39, green: 174, blue: 96)
    public static let lightPurple = Color(red: 155, green: 89, blue: 182)
    public static let darkPurple = Color(red: 142, green: 68, blue: 173)
    public static let lightBlue = Color(red: 52, green: 152, blue: 219)
    public static let darkBlue = Color(red: 41, green: 128, blue: 185)
    public static let lightYellow = Color(red: 241, green: 196, blue: 15)
    public static let darkYellow = Color(red: 243, green: 156, blue: 18)
    public static let lightOrange = Color(red: 230, green: 126, blue: 34)
    public static let darkOrange = Color(red: 211, green: 84, blue: 0)
    public static let purpleBg = Color(red: 69, green: 51, blue: 201)
}
```

In the code above, we create an `init` method which takes in the values of `red` , `green` , and `blue` . This makes it easier to initialize an instance of Color with an RGB color code. All the colors are derived from the flat color palette (https://flatuicolors.com/palette/defo). If you prefer to use other colors, you can simply modify the color values or create your own color constants.

Implementing the Circular Progress Bar

Referring to figure 4, a circular progress bar literally consists of two circles: a full circle in gray underneath and another partial (or full) circle in gradient color on top. Thus, to implement the progress bar, we need a `ZStack` to overlay two views:

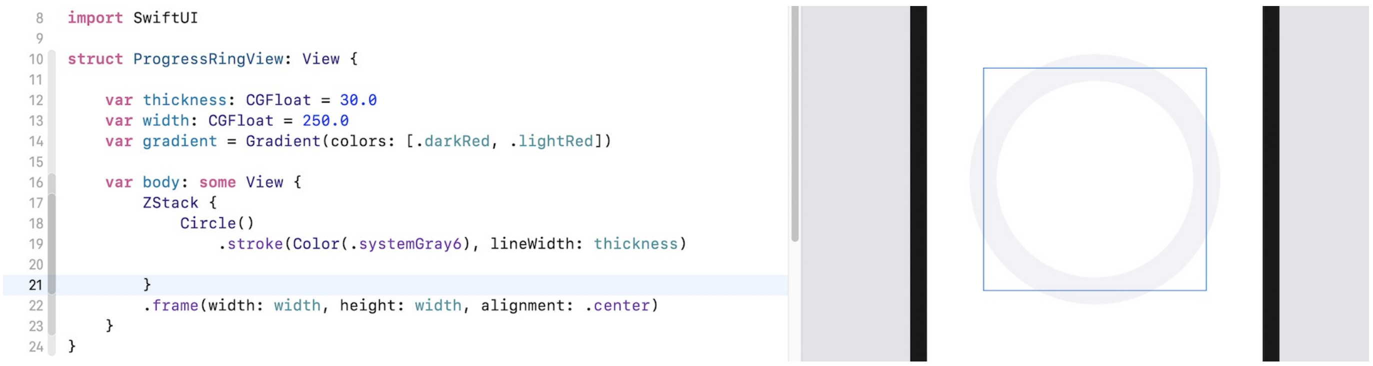
1. A circle view in gray
2. A ring shape in gradient color sitting on top of #1

Now open `ProgressRingView.swift` and declare the following variables:

```
var thickness: CGFloat = 30.0
var width: CGFloat = 250.0
```

Since this circular progress bar should support various sizes, we declare the variables above with default values. As the name suggests, the `thickness` variable controls the thickness of the progress bar. The `width` variable stores the diameter of the circle.

You can create the circle view using the built-in `Circle` shape like this:



```
import SwiftUI

struct ProgressRingView: View {

    var thickness: CGFloat = 30.0
    var width: CGFloat = 250.0
    var gradient = Gradient(colors: [.darkRed, .lightRed])

    var body: some View {
        ZStack {
            Circle()
                .stroke(Color(.systemGray6), lineWidth: thickness)

        }
        .frame(width: width, height: width, alignment: .center)
    }
}
```

Figure 6. Drawing the Circle view

We use the `stroke` modifier to draw the outline of the circle in gray. As you can see in the figure, the `thickness` property is used to control the width of the outline. The `width` property is the diameter of the circle. I intentionally highlight the frame, so that you can see the thickness and width.

Next, we are going to implement the ring shape. One way to create this ring shape is by using `Circle`. We have discussed drawing circles in chapter 8. This time, let me show you an alternate implementation. We will use the `Shape` protocol to create a custom Ring shape.

In the same file, insert the following code:

```
struct RingShape: Shape {
    var progress: Double = 0.0
    var thickness: CGFloat = 30.0

    func path(in rect: CGRect) -> Path {

        var path = Path()

        path.addArc(center: CGPoint(x: rect.width / 2.0, y: rect.height / 2.0),
                    radius: min(rect.width, rect.height) / 2.0,
                    startAngle: .degrees(0),
                    endAngle: .degrees(360 * progress), clockwise: false)

        return path.strokedPath(.init(lineWidth: thickness, lineCap: .round))
    }
}
```

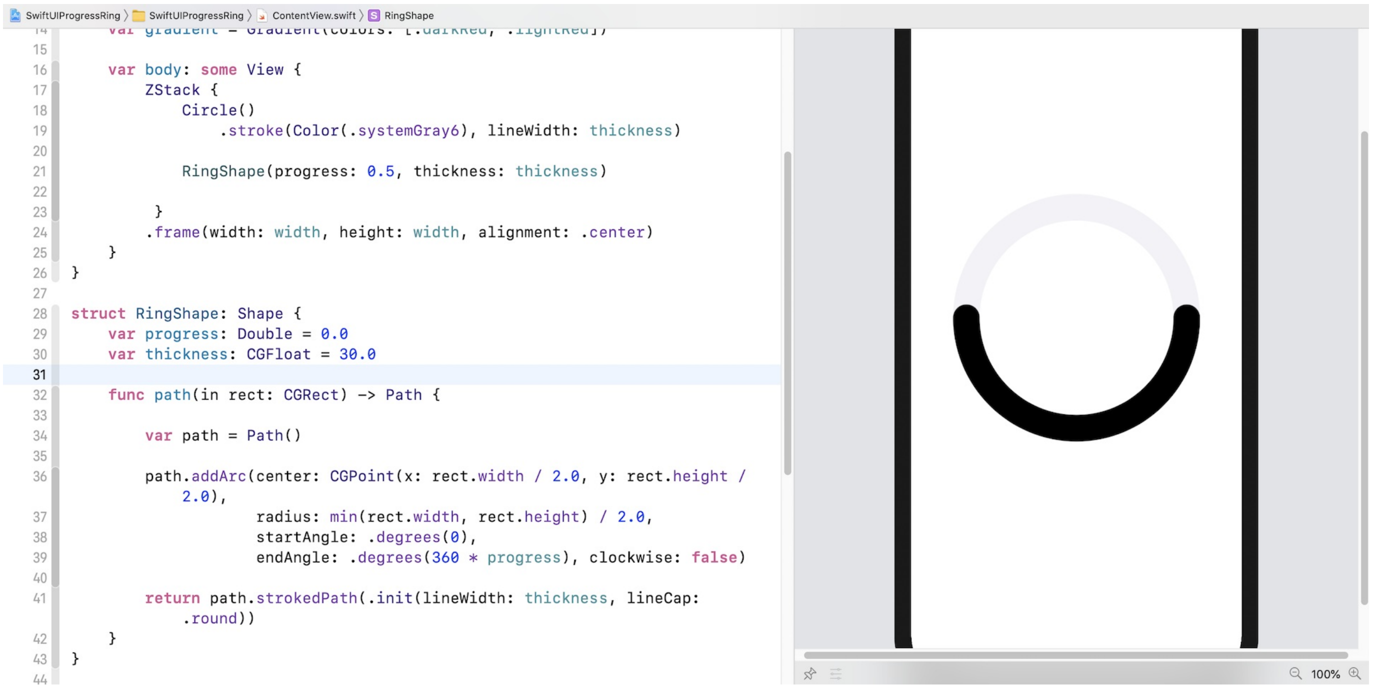
We create a `RingShape` struct by adopting the `Shape` protocol. We declare two properties in the struct. The `progress` property allows the user to specify the percentage of progress. The `thickness` property, similar to that in `ProgressRingView`, lets you control the width of the ring.

To draw the ring, we use the `addArc` method, followed by `strokedPath`. The radius of the arc can be calculated by dividing the frame's width (or height) by 2. The starting angle is currently set to zero degrees. We calculate the ending angle by multiplying 360 with the progress value. For example, if we set the `progress` to 0.5, we draw a half ring (from 0 to 180 degrees).

To use the `RingShape`, you can update the `body` variable like this:

```
ZStack {
    Circle()
        .stroke(Color(.systemGray6), lineWidth: thickness)

    RingShape(progress: 0.5, thickness: thickness)
 }
.frame(width: width, height: width, alignment: .center)
```

Once you make the changes, you should see a partial ring overlay on top of the gray circle. Note that it has round cap at both ends since we set the `lineCap` parameter of `strokedPath` to `.round`.



```
    var gradient = Gradient(colors: [.darkRed, .lightRed])

    var body: some View {
        ZStack {
            Circle()
                .stroke(Color(.systemGray6), lineWidth: thickness)

            RingShape(progress: 0.5, thickness: thickness)

         }
        .frame(width: width, height: width, alignment: .center)
    }
}

struct RingShape: Shape {
    var progress: Double = 0.0
    var thickness: CGFloat = 30.0

    func path(in rect: CGRect) -> Path {

        var path = Path()

        path.addArc(center: CGPoint(x: rect.width / 2.0, y: rect.height /
            2.0),
                    radius: min(rect.width, rect.height) / 2.0,
                    startAngle: .degrees(0),
                    endAngle: .degrees(360 * progress), clockwise: false)

        return path.strokedPath(.init(lineWidth: thickness, lineCap:
            .round))
    }
}
```

Figure 7. Displaying the RingShape

Other than the ring's color, you may also notice something that we need to tweak. The start point of the arc is not the same as that in figure 4. To fix the issue, you need change the `startAngle` from zero to -90.

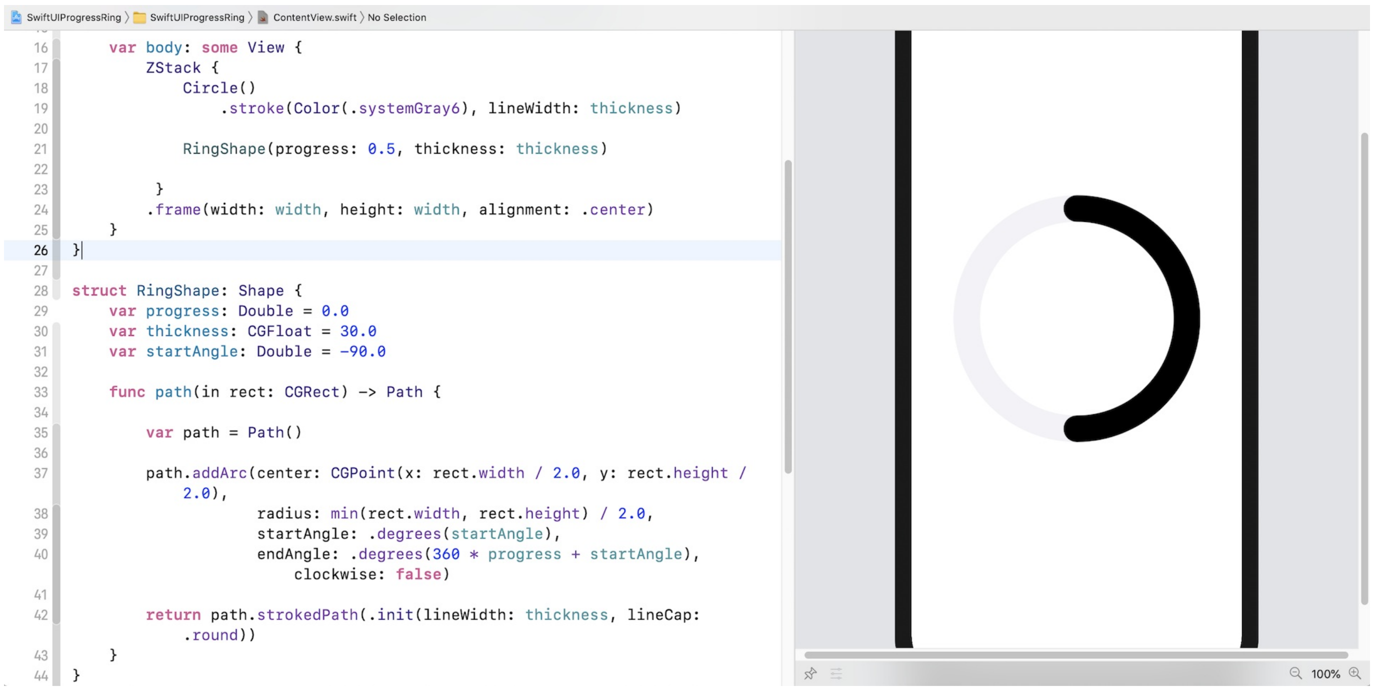
Declare the following property in `RingShape`:

```
var startAngle: Double = -90.0
```

Then update the `addArc` method like this:

```
path.addArc(center: CGPoint(x: rect.width / 2.0, y: rect.height / 2.0),
            radius: min(rect.width, rect.height) / 2.0,
            startAngle: .degrees(startAngle),
            endAngle: .degrees(360 * progress + startAngle), clockwise: false)
```

We change the `startAngle` parameter to `-90` degree. we also need to alter the `endAngle` parameter, because the starting angle is changed. With the modification, the arc now rotates by 90 degrees anticlockwise.


SwiftUIProgressRing 〉 SwiftUIProgressRing 〉 ContentView.swift 〉 No Selection
100%

```
    var body: some View {
        ZStack {
            Circle()
                .stroke(Color(.systemGray6), lineWidth: thickness)

            RingShape(progress: 0.5, thickness: thickness)

        }
        .frame(width: width, height: width, alignment: .center)
    }
}

struct RingShape: Shape {
    var progress: Double = 0.0
    var thickness: CGFloat = 30.0
    var startAngle: Double = -90.0

    func path(in rect: CGRect) -> Path {

        var path = Path()

        path.addArc(center: CGPoint(x: rect.width / 2.0, y: rect.height /
            2.0),
                    radius: min(rect.width, rect.height) / 2.0,
                    startAngle: .degrees(startAngle),
                    endAngle: .degrees(360 * progress + startAngle),
                        clockwise: false)

        return path.strokedPath(.init(lineWidth: thickness, lineCap:
            .round))
    }
}
```

Figure 8. The partial ring after changing the start angle

Adding a Gradient

Now that you have a ring shape that is adjustable by passing different `progress` values to it, wouldn't it be great if we add a gradient color to the bar? SwiftUI provides three types of gradients including linear gradient, angular gradient, and radial gradient. Apple uses the angular gradient to fill the progress bar.

Here is an example using `AngularGradient`:

```
AngularGradient(gradient: Gradient(colors: [.darkPurple, .lightYellow]), center: .
center, startAngle: .degrees(0), endAngle: .degrees(180))
```

The angular gradient applies the gradient color as the angle changes. In the code above, we render the gradient from 0 degrees to 180 degrees. Figure 9 shows you the result of two different angular gradients.

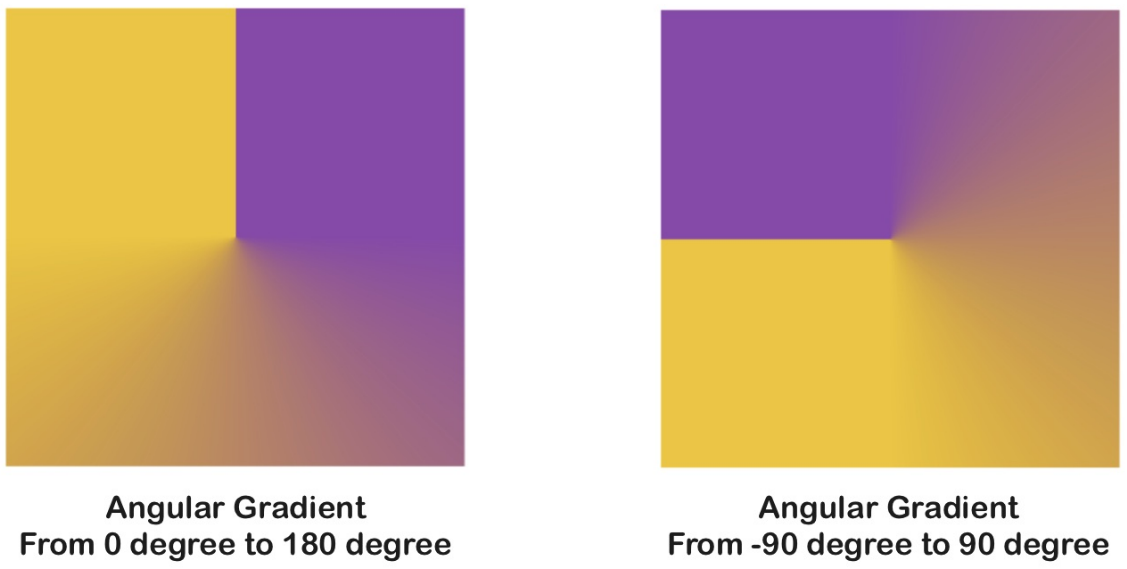


Figure 9. Angular gradient with different start and end angles

Since the starting angle of the ring shape is set to -90 degrees, we will apply the angular gradient like this (assuming the progress is set to 0.5):

```
AngularGradient(gradient: Gradient(colors: [.darkPurple, .lightYellow]), center: .
center, startAngle: .degrees(startAngle), endAngle: .degrees(360 * 0.5 + startAngl
e))
```

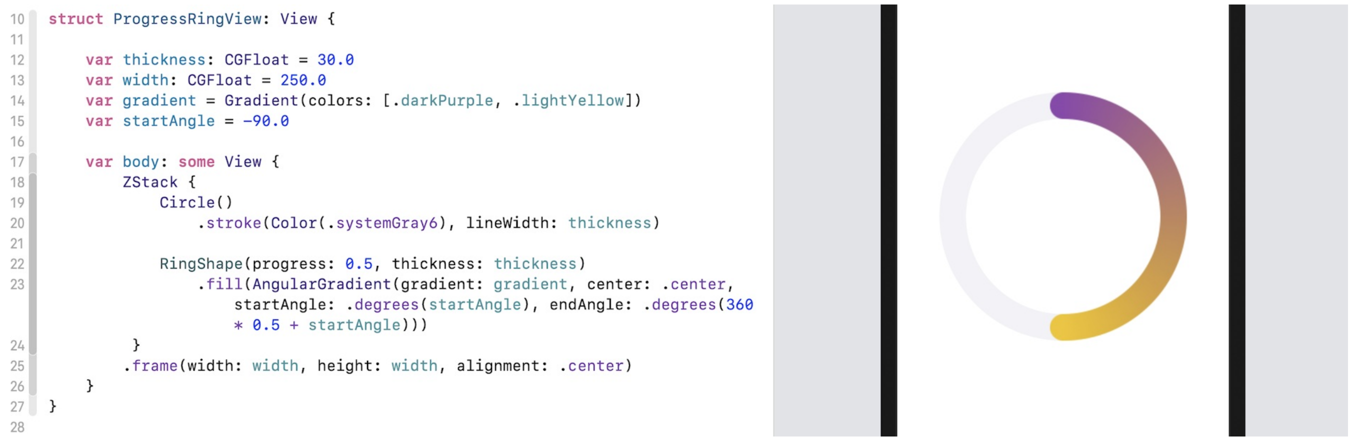
Now let's modify the code to apply the gradient to the `RingShape`. First, declare the following properties in `ProgressRingView`:

```
var gradient = Gradient(colors: [.darkPurple, .lightYellow])
var startAngle = -90.0
```

Then fill the `RingShape` with the angular gradient by attaching the `.fill` modifier like below:

```
RingShape(progress: 0.5, thickness: thickness)
    .fill(AngularGradient(gradient: gradient, center: .center, startAngle: .degree
s(startAngle), endAngle: .degrees(360 * 0.5 + startAngle)))
```

As soon as you complete the modification, the circular progress bar should be filled with the specified gradient.



```
struct ProgressRingView: View {

    var thickness: CGFloat = 30.0
    var width: CGFloat = 250.0
    var gradient = Gradient(colors: [.darkPurple, .lightYellow])
    var startAngle = -90.0

    var body: some View {
        ZStack {
            Circle()
                .stroke(Color(.systemGray6), lineWidth: thickness)

            RingShape(progress: 0.5, thickness: thickness)
                .fill(AngularGradient(gradient: gradient, center: .center,
                    startAngle: .degrees(startAngle), endAngle: .degrees(360
                    * 0.5 + startAngle)))
        }
        .frame(width: width, height: width, alignment: .center)
    }
}
```

Figure 10. A circular progress bar with gradient

Varying Progress

The percentage of progress is now fixed at 0.5. Obviously, we need to create a variable for that to make it adjustable. In `ProgressRingView`, declare a variable named `progress` like this:

```
@Binding var progress: Double
```

We are developing a flexible `ProgressRingView` and want to let the caller control the percentage of progress. Therefore, the source of truth (i.e. progress) should be provided by the caller. This is the reason why `progress` is marked as a binding variable.

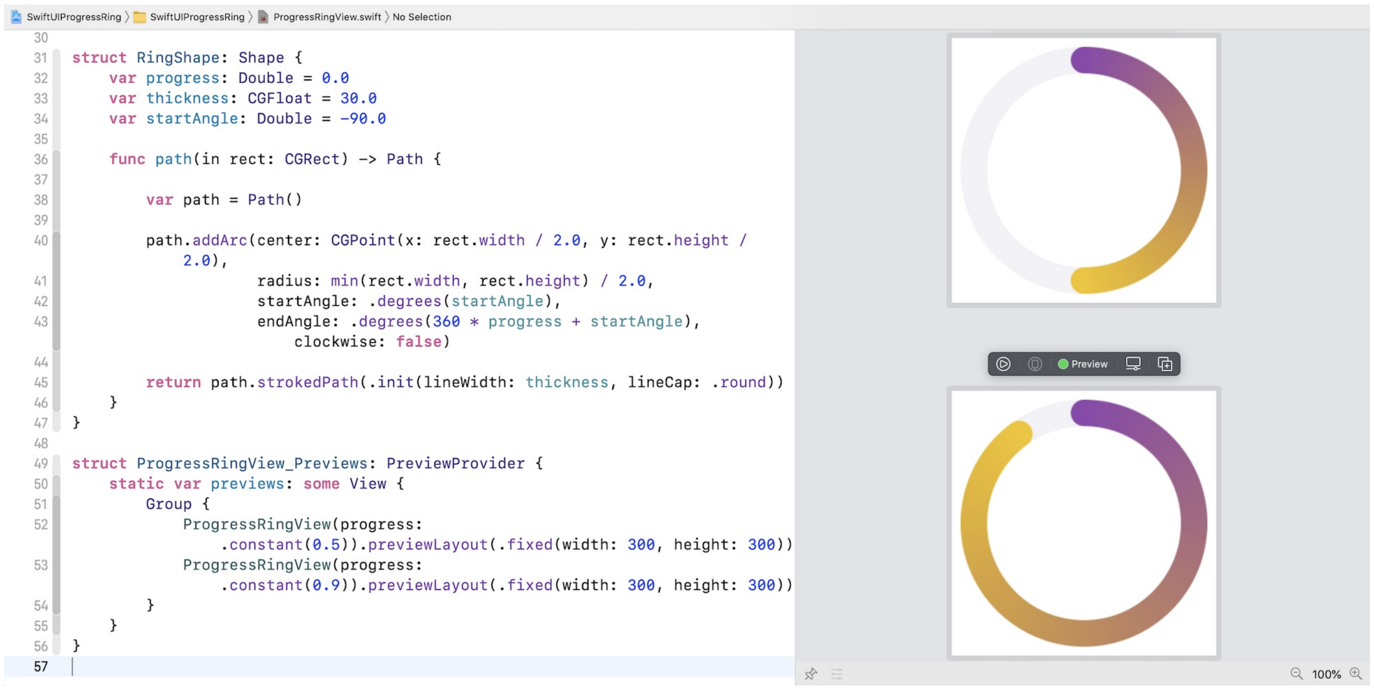
With the variable, we can update the following line of code accordingly:

```
RingShape(progress: progress, thickness: thickness)
    .fill(AngularGradient(gradient: gradient, center: .center, startAngle: .degree
s(startAngle), endAngle: .degrees(360 * progress + startAngle)))
```

Xcode should now indicate an error in `ProgressRingView_Previews` because we have to pass `ProgressRingView` the `progress` parameter. Therefore, update the `ProgressRingView_Previews` like this:

```
struct ProgressRingView_Previews: PreviewProvider {
    static var previews: some View {
        Group {
            ProgressRingView(progress: .constant(0.5)).previewLayout(.fixed(width:
300, height: 300))
            ProgressRingView(progress: .constant(0.9)).previewLayout(.fixed(width:
300, height: 300))
        }
    }
}
```

I want to see the end result of two different values of progress, so we create two instances of `ProgressRingView` in the preview. Instead of previewing the progress ring on a simulator, we use `previewLayout` to preview it in a fixed size canvas. This allows us to easily see both results all at once.


SwiftUIProgressRing 〉 SwiftUIProgressRing 〉 ProgressRingView.swift 〉 No Selection

```
struct RingShape: Shape {
    var progress: Double = 0.0
    var thickness: CGFloat = 30.0
    var startAngle: Double = -90.0

    func path(in rect: CGRect) -> Path {

        var path = Path()

        path.addArc(center: CGPoint(x: rect.width / 2.0, y: rect.height /
            2.0),
                    radius: min(rect.width, rect.height) / 2.0,
                    startAngle: .degrees(startAngle),
                    endAngle: .degrees(360 * progress + startAngle),
                        clockwise: false)

        return path.strokedPath(.init(lineWidth: thickness, lineCap: .round))
    }
}

struct ProgressRingView_Previews: PreviewProvider {
    static var previews: some View {
        Group {
            ProgressRingView(progress:
                .constant(0.5)).previewLayout(.fixed(width: 300, height: 300))
            ProgressRingView(progress:
                .constant(0.9)).previewLayout(.fixed(width: 300, height: 300))
        }
    }
}
```

Figure 11. A circular progress bar with gradient

Animating the Ring Shape with Animatable

The circular progress bar looks pretty good. Let's put it into practice and create a simple demo like figure 12. This demo has three buttons for adjusting the progress. We expect that the progress bar will gradually increase (or decrease) to the chosen percentage when any of the buttons is tapped. For example, the current progress is set to 0. When the "50%" button is tapped, the progress bar will gradually goes up from 0% to 50%.

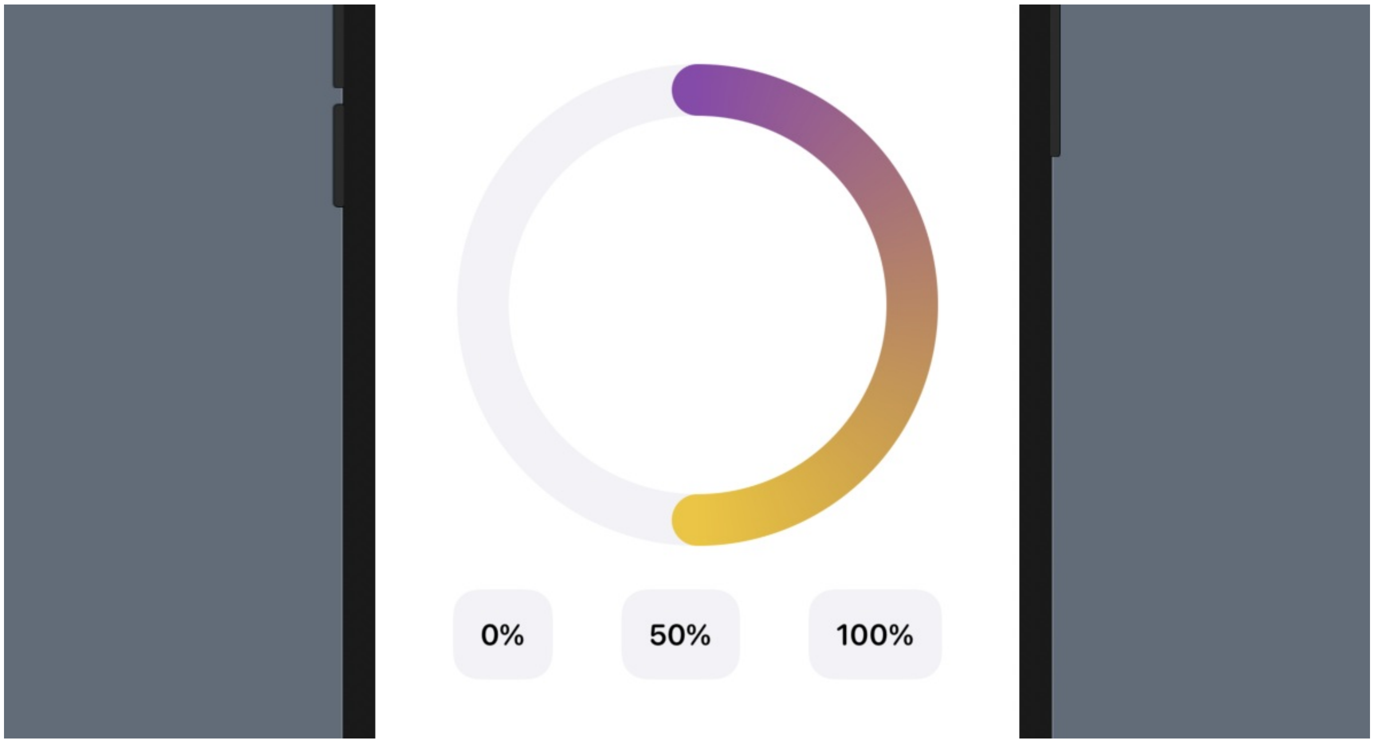


Figure 12. A quick demo

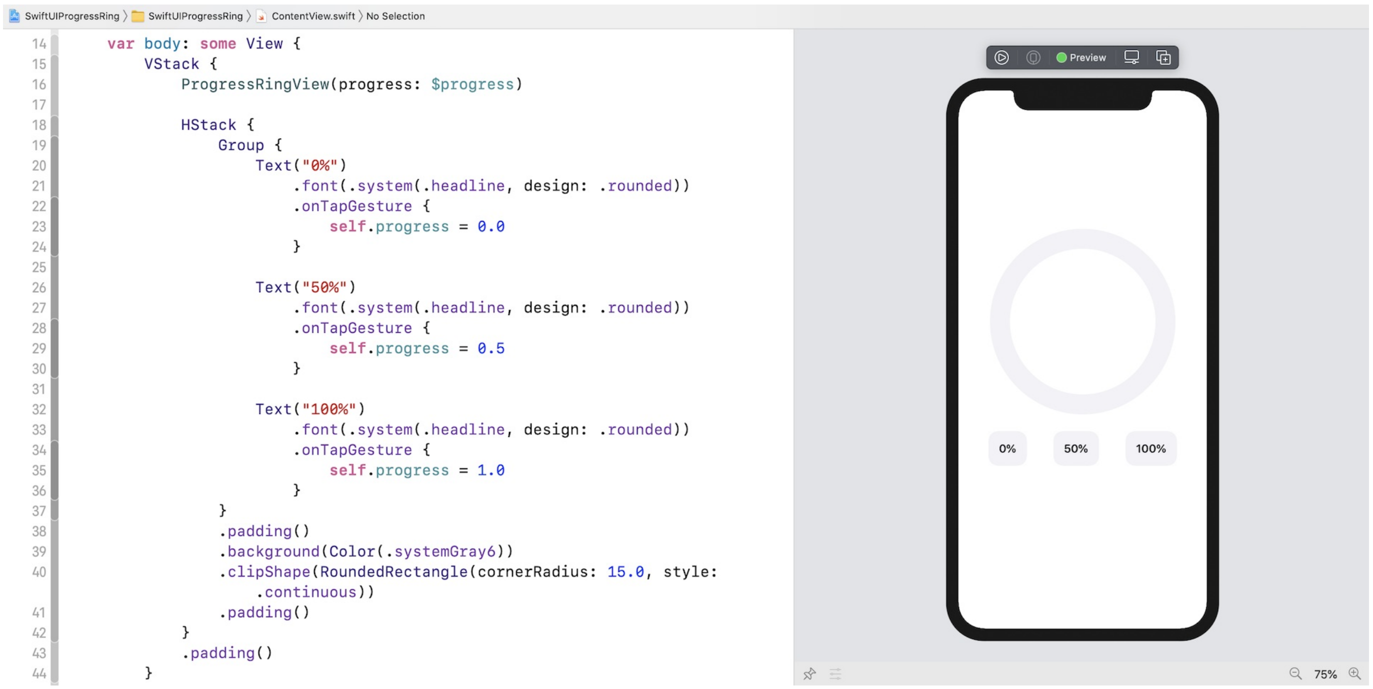
Now let's switch over to `ContentView.swift` to create this demo. First, declare a state variable to keep track of the progress like this:

```
@State var progress = 0.0
```

Then insert the following code in the `body` variable to create the UI:

```
VStack {
    ProgressRingView(progress: $progress)

    HStack {
        Group {
            Text("0%")
                .font(.system(.headline, design: .rounded))
                .onTapGesture {
                    self.progress = 0.0
                }

            Text("50%")
                .font(.system(.headline, design: .rounded))
                .onTapGesture {
                    self.progress = 0.5
                }

            Text("100%")
                .font(.system(.headline, design: .rounded))
                .onTapGesture {
                    self.progress = 1.0
                }
        }
        .padding()
        .background(Color(.systemGray6))
        .clipShape(RoundedRectangle(cornerRadius: 15.0, style: .continuous))
        .padding()
    }
    .padding()
}
```

In your preview canvas, you should have something like below. The progress bar only shows the gray circle underneath because the progress is defaulted to zero. Click the *Play* button to run the demo. Try tapping different buttons to see how the progress bar responds.


SwiftUIProgressRing 〉 SwiftUIProgressRing 〉 ContentView.swift 〉 No Selection

```
    var body: some View {
        VStack {
            ProgressRingView(progress: $progress)

            HStack {
                Group {
                    Text("0%")
                        .font(.system(.headline, design: .rounded))
                        .onTapGesture {
                            self.progress = 0.0
                        }

                    Text("50%")
                        .font(.system(.headline, design: .rounded))
                        .onTapGesture {
                            self.progress = 0.5
                        }

                    Text("100%")
                        .font(.system(.headline, design: .rounded))
                        .onTapGesture {
                            self.progress = 1.0
                        }
                }
                .padding()
                .background(Color(.systemGray6))
                .clipShape(RoundedRectangle(cornerRadius: 15.0, style:
                    .continuous))
                .padding()
            }
            .padding()
        }
```

Figure 13. The demo UI

Does it work up to your expections? I think not. When you tap the 50% button, the progress bar instantly fills half of the ring without any animation. This isn't what we expect.

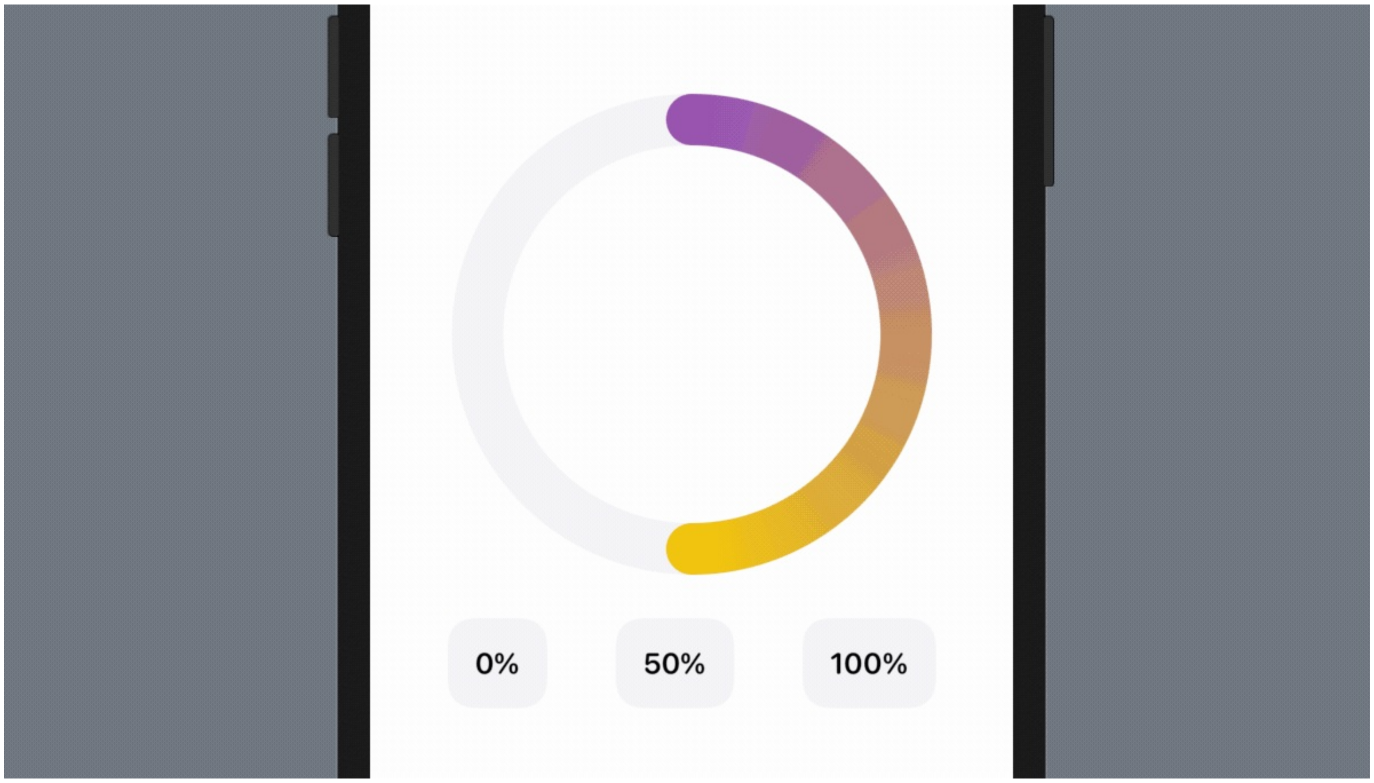


Figure 14. The progress bar doesn't animate its change

I guess you may know why the view is not animated. We haven't attached an `.animation` modifier to the ring shape. Switch back to `ProgressRingView.swift` and attach the `.animation` modifier to the `ZStack` of `ProgressRingView`. You can insert the code after the `.frame` modifier:

```
.animation(Animation.easeInOut(duration: 1.0))
```

Okay, it seems like we've figured out the solution. Let's go back to `ContentView.swift` and test the demo again. Run the demo and tap any of the buttons to try it out.

What's your result? Does the fix work?

Unfortunately, the ring still doesn't animate the progress change, but it does animate the gradient change.

What's the root cause?

Before solving the issue, let me further explain how the `.animation` modifier works. In the official documentation) for the `.animation` modifier, it mentions that *the modifier applies the given animation to all animatable values within the view*. The keyword here is *animatable*. When you use the `.animation` modifier on a view, SwiftUI automatically animates any changes to animatable properties of the view.

SwiftUI comes with a protocol called `Animatable`. For a view that supports animation, you can adopt the protocol and provide the `animatableData` property. This property tells SwiftUI what data the view can animate.

In chapter 9, I introduced you the basics of SwiftUI animation. You can easily animate the size change of a view using `.scaleEffect` or the position change by using `.offset`. It may seem to you that all these animations work automatically. Behind the scenes, Apple's engineers actually adopted the protocol and provided the animatable data for `CGSize` and `CGPoint`.

So, why can't `RingShape` animate its progress change?

The `RingShape` struct conforms to the `Shape` protocol. If you look at its documentation, `Shape` adopts the `Animatable` protocol and provides the default implementation. However, the default implementation of the `animatableData` property is to return an instance of `EmptyAnimatableData`, which means no animatable data. This is why `ProgressRingView` cannot animate the progress change.

To fix the issue and make the progress animatable, all you need to do is to override the default implementation and provide the animatable values. In the `RingShape` struct, insert the following code before the `path` function:

```
var animatableData: Double {
    get { progress }
    set { progress = newValue }
}
```

The implementation is very simple. We just tell SwiftUI to animate the `progress` value. That's it!

Now go back to `ContentView.swift` and play the demo app to have another test. This time the progress bar should animate the progress change.

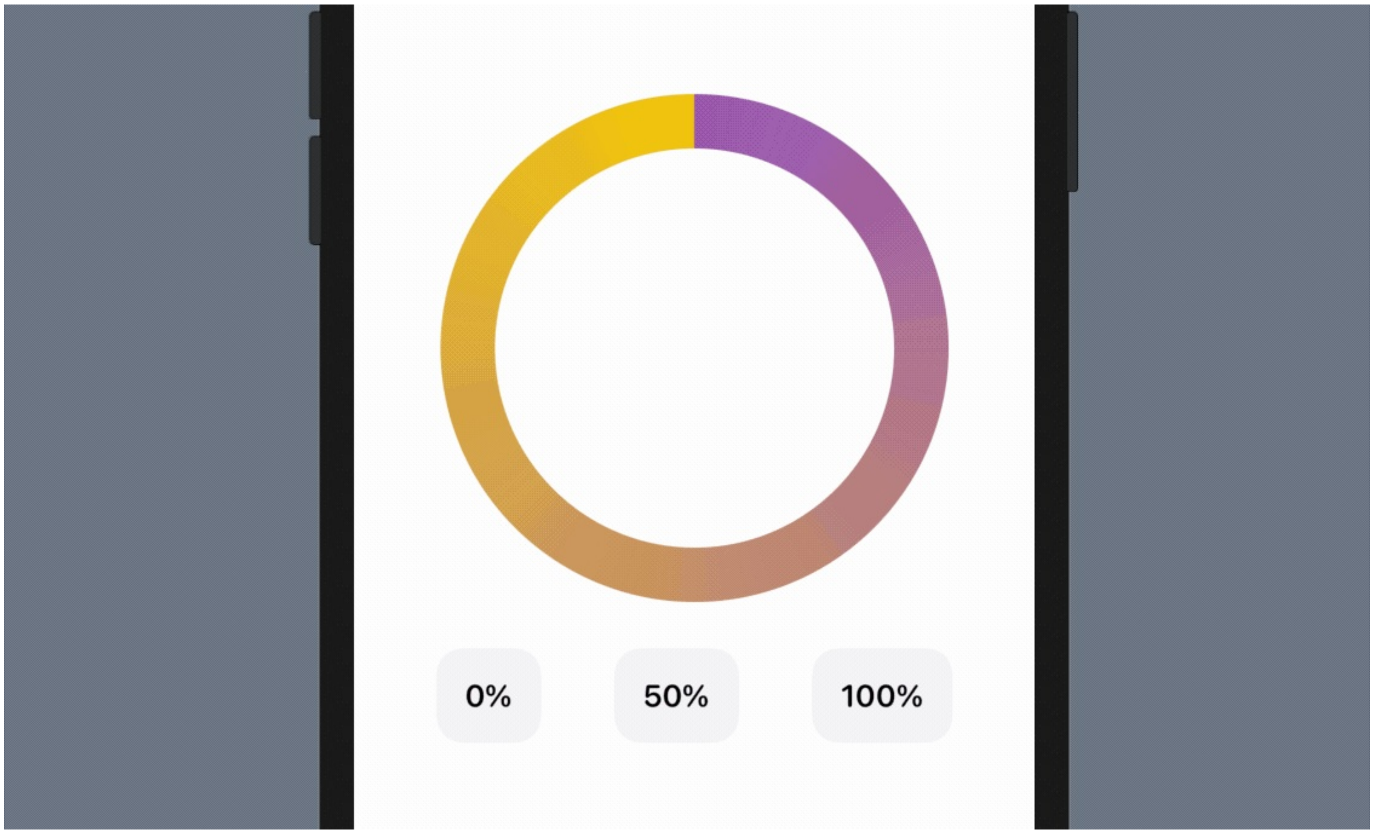


Figure 15. The progress bar doesn't animate its change

The 100% Problem

With the animation, this circular progress bar is now even better. However, there is a little issue that you may notice. When the percentage is set to 100%, the arc becomes a full circle, hiding the round caps. To highlight where the arc ends, it's better to add the round cap with a drop shadow like the activity ring in figure 1.

To resolve the issue, my idea is to overlay a little circle, who's size is based on the thickness of the ring, at the end of the arc. Additionally, we will add a drop shadow for that little circle. Figure 16 illustrates this solution. *Please note that for the final solution, the circle should have the same color as the arc's end. I just highlighted it using red color for purpose of illustration.*

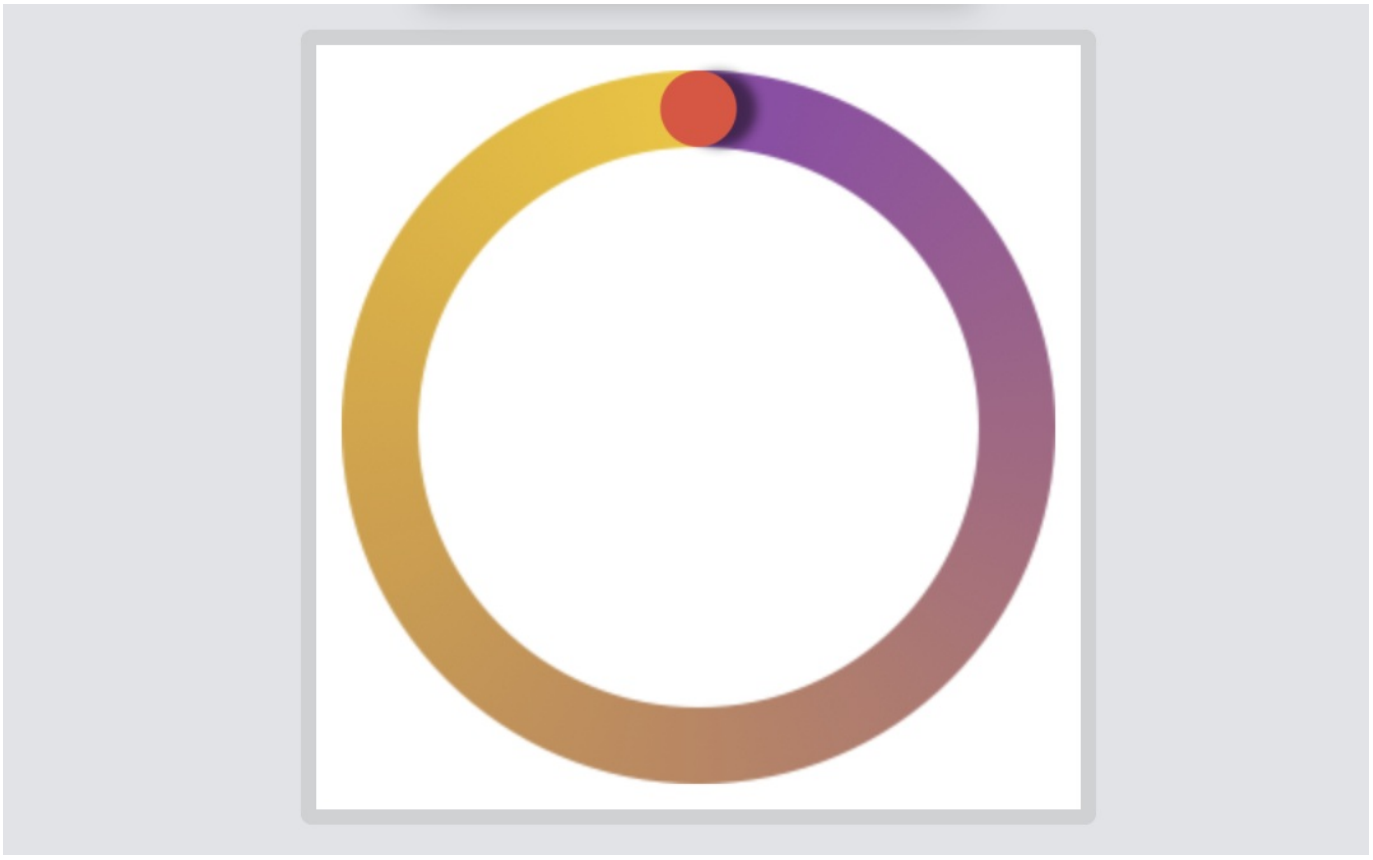


Figure 16. Overlaying a little circle

The question is how do you calculate the position of this little circle or the end position of the arc? This requires some mathematical knowledge. Figure 17 shows you how we calculate the position of the little circle.

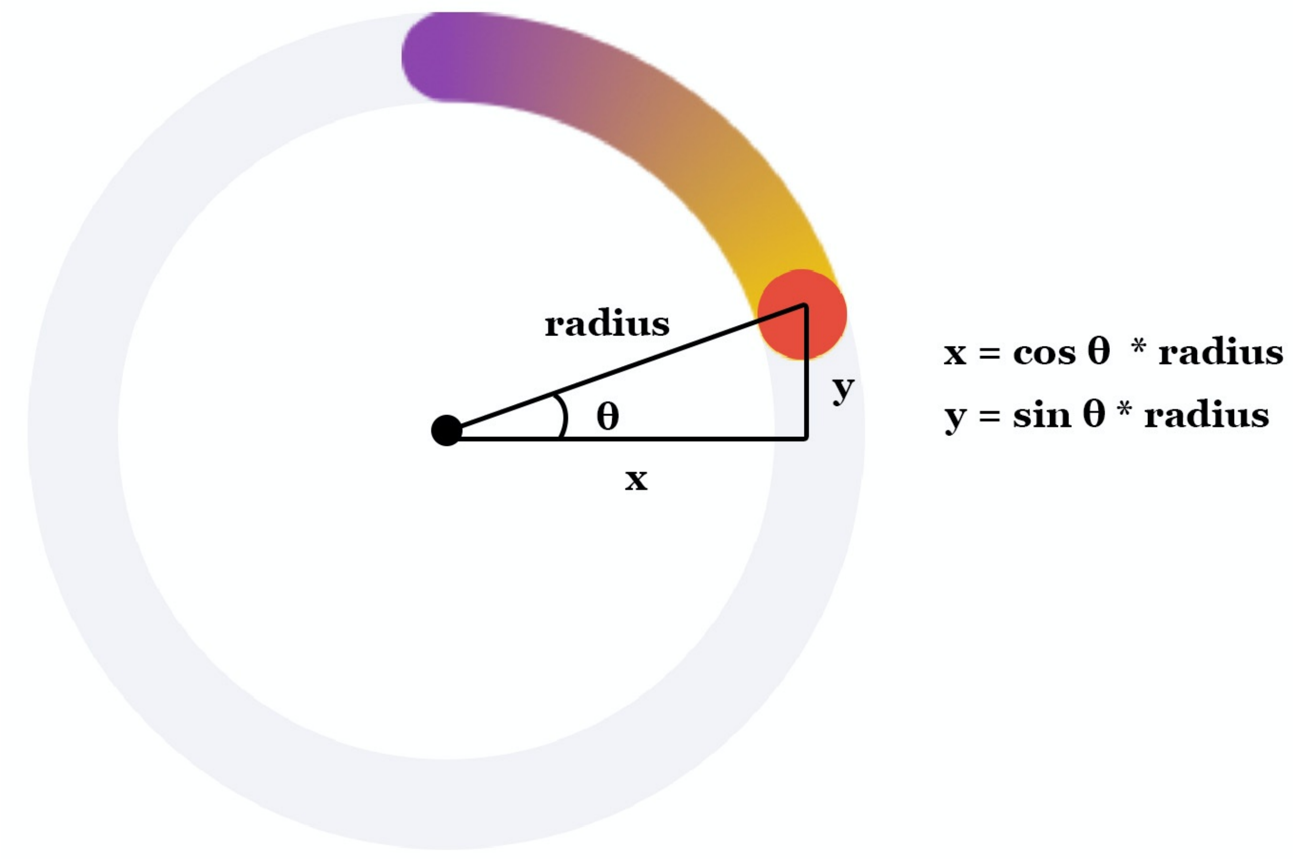


Figure 17. Overlaying a little circle

Now, let's dive into the implementation and create the little circle. Let's call it `RingTip` and implement it in the `ProgressRingView.swift` file like this:

```
struct RingTip: Shape {
    var progress: Double = 0.0
    var startAngle: Double = -90.0
    var ringRadius: Double

    private var position: CGPoint {
        let angle = 360 * progress + startAngle
        let angleInRadian = angle * .pi / 180

        return CGPoint(x: ringRadius * cos(angleInRadian), y: ringRadius * sin(ang
leInRadian))
    }

    var animatableData: Double {
        get { progress }
        set { progress = newValue }
    }

    func path(in rect: CGRect) -> Path {
        var path = Path()

        guard progress > 0.0 else {
            return path
        }

        let frame = CGRect(x: position.x, y: position.y, width: rect.size.width, h
eight: rect.size.height)

        path.addRoundedRect(in: frame, cornerSize: frame.size)

        return path
    }

}
```

The `RingTip` struct takes in three parameters: `progress`, `startAngle`, and `ringRadius` for the calculation of the circle's position. Once we figure out the position, we can draw the path of the circle by using `addRoundedRect`.

Now go back to `ProgressRingView` and declare the following computed property to calculate the ring's radius:

```
private var radius: Double {
    Double(width / 2)
}
```

Next, create `RingTip` by inserting the following code after `RingShape` in the `ZStack`:

```
RingTip(progress: progress, startAngle: startAngle, ringRadius: radius)
    .frame(width: thickness, height: thickness)
    .foregroundColor(progress > 0.96 ? gradient.stops[1].color : Color.clear)
```

We instantiate `RingTip` by passing the current progress, start angle, and the radius of the ring. The foreground color is set to the ending gradient color. You may wonder why we only display the gradient color when the progress is greater than 0.96. Take a look at figure 18 and you will understand why I come up with this decision.

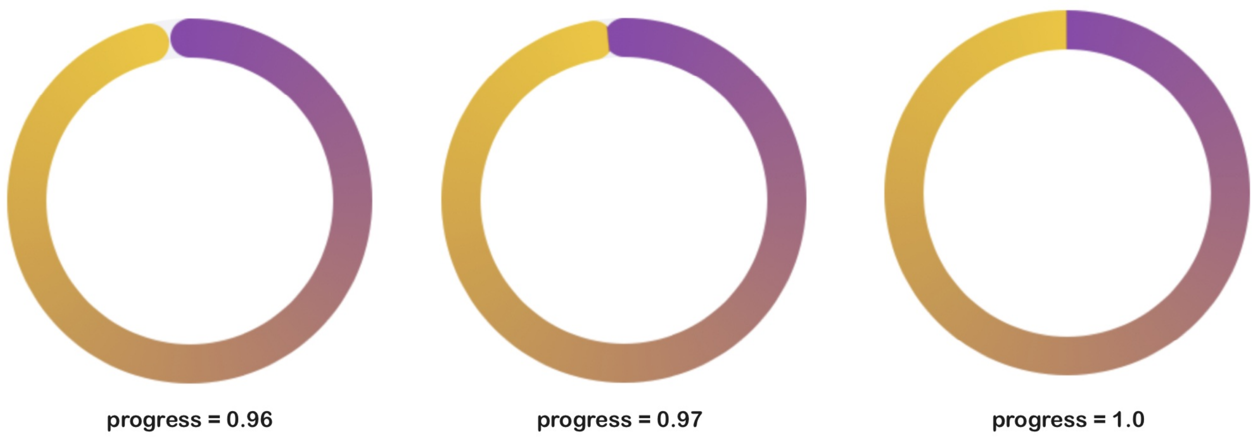
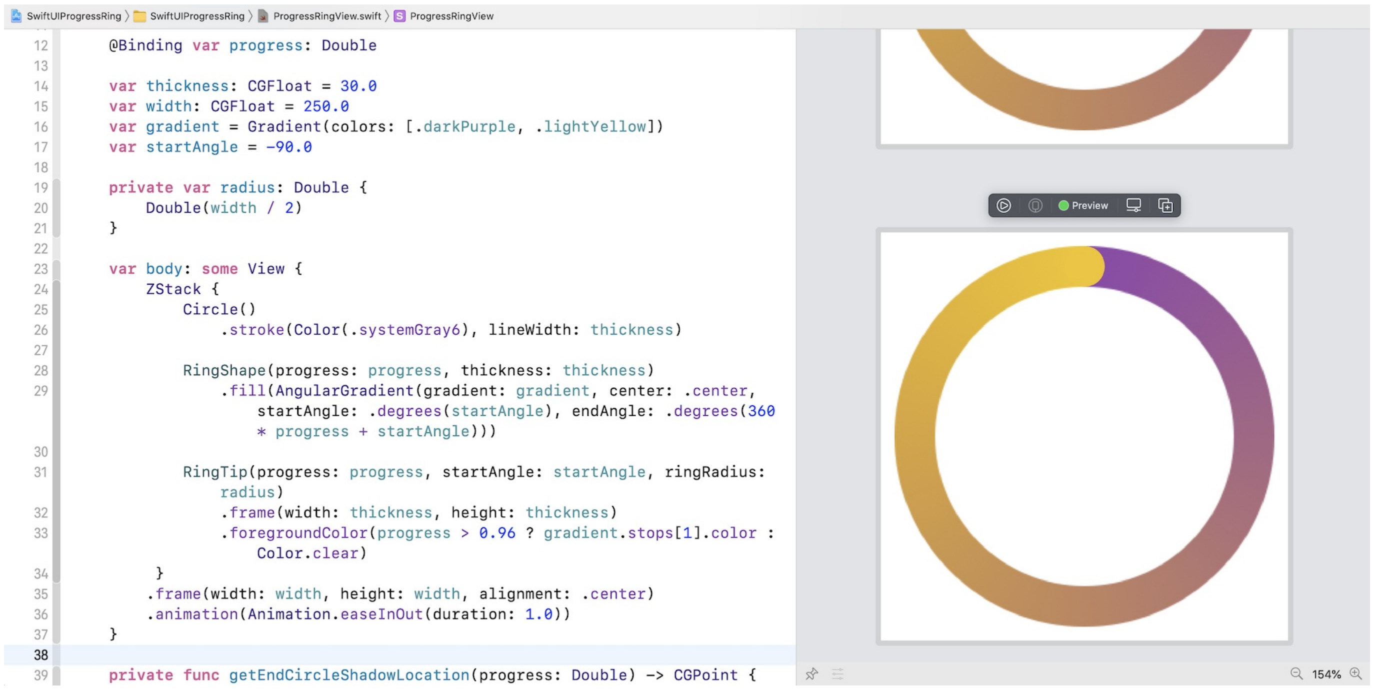


Figure 18. Need to overlay the circle only when the progress is greater than 0.96

After adding the instance of `RingTip` in the `ZStack`, run the program in the preview. Click the 100% button. The progress bar should now have a round cap.



```
    @Binding var progress: Double

    var thickness: CGFloat = 30.0
    var width: CGFloat = 250.0
    var gradient = Gradient(colors: [.darkPurple, .lightYellow])
    var startAngle = -90.0

    private var radius: Double {
        Double(width / 2)
    }

    var body: some View {
        ZStack {
            Circle()
                .stroke(Color(.systemGray6), lineWidth: thickness)

            RingShape(progress: progress, thickness: thickness)
                .fill(AngularGradient(gradient: gradient, center: .center,
                    startAngle: .degrees(startAngle), endAngle: .degrees(360
                    * progress + startAngle)))

            RingTip(progress: progress, startAngle: startAngle, ringRadius:
                radius)
                .frame(width: thickness, height: thickness)
                .foregroundColor(progress > 0.96 ? gradient.stops[1].color :
                    Color.clear)
         }
        .frame(width: width, height: width, alignment: .center)
        .animation(Animation.easeInOut(duration: 1.0))
    }

    private func getEndCircleShadowLocation(progress: Double) -> CGPoint {
```

Figure 19. Overlaying a little circle at the ring end

You've already built a pretty nice circular progress bar. But we can make it even better by adding a drop shadow at the arc end. In SwiftUI, you can simply attach the `.shadow` modifier to add a drop shadow. In this case, we can attach the modifier to `RingTip`. The hard part is that we need to figure out where we add the drop shadow.

The calculation of the shadow position is very similar to that of the ring tip. So, in `ProgressRingView.swift`, insert a function for computing the position of the ring tip:

```
private func ringTipPosition(progress: Double) -> CGPoint {
    let angle = 360 * progress + startAngle
    let angleInRadian = angle * .pi / 180

    return CGPoint(x: radius * cos(angleInRadian), y: radius * sin(angleInRadian))
}
```

Then add a new computed property for calculating the shadow offset of the ring tip like this:

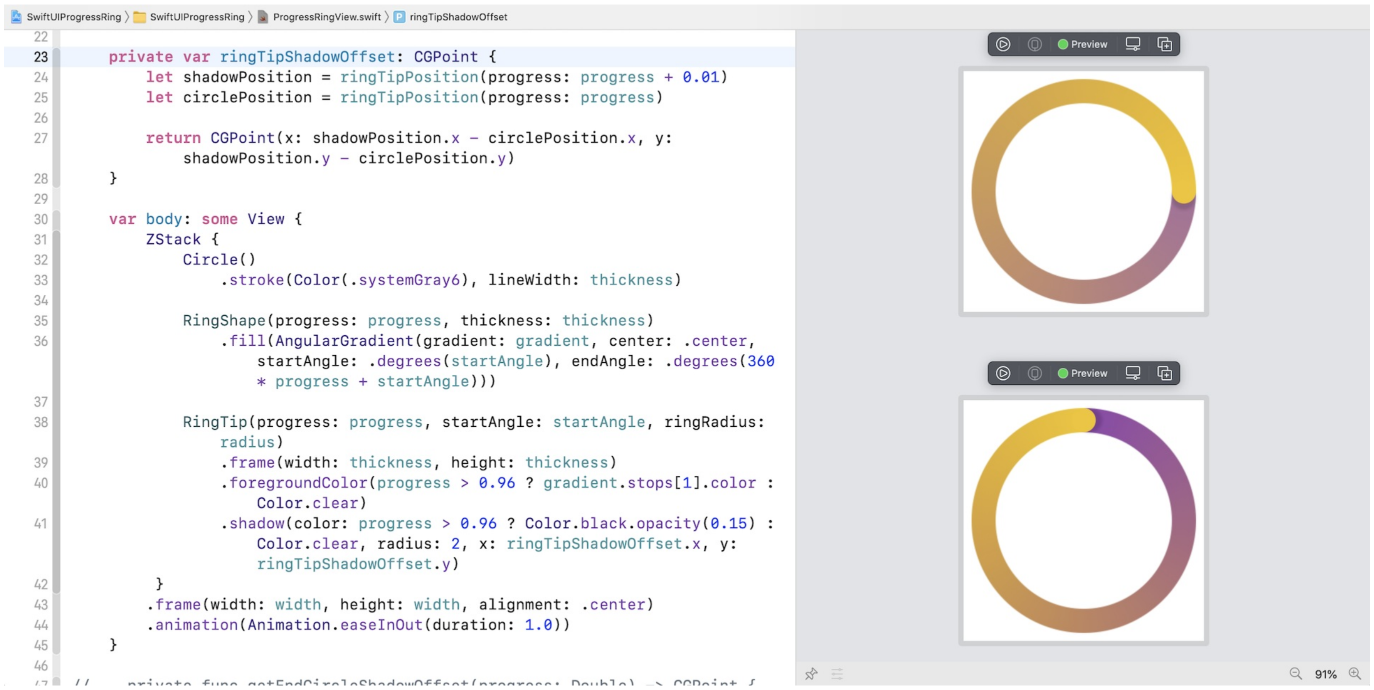
```
private var ringTipShadowOffset: CGPoint {
    let shadowPosition = ringTipPosition(progress: progress + 0.01)
    let circlePosition = ringTipPosition(progress: progress)

    return CGPoint(x: shadowPosition.x - circlePosition.x, y: shadowPosition.y - c
irclePosition.y)
}
```

By adding 0.01 to the current progress, we can compute the shadow position. This is my solution for finding the shadow position. You may come up with an alternative solution.

With the shadow offset, we can attach the `.shadow` modifier to `RingTip`:

```
.shadow(color: progress > 0.96 ? Color.black.opacity(0.15) : Color.clear, radius: 2
, x: ringTipShadowOffset.x, y: ringTipShadowOffset.y)
```

I just want to add a light shadow, so the opacity is set to 0.15. If you prefer to have a darker shadow, increase the opacity value (say, 1.0). After the code change, you should see a drop shadow at the end of the ring, provided that the progress is greater than 0.96. You can also try to set the progress value to a value larger than 1.0 and see how the progress bar looks.



SwiftUIProgressRing 〉 SwiftUIProgressRing 〉 ProgressRingView.swift 〉 ringTipShadowOffset

```
    private var ringTipShadowOffset: CGPoint {
        let shadowPosition = ringTipPosition(progress: progress + 0.01)
        let circlePosition = ringTipPosition(progress: progress)

        return CGPoint(x: shadowPosition.x - circlePosition.x, y:
            shadowPosition.y - circlePosition.y)
    }

    var body: some View {
        ZStack {
            Circle()
                .stroke(Color(.systemGray6), lineWidth: thickness)

            RingShape(progress: progress, thickness: thickness)
                .fill(AngularGradient(gradient: gradient, center: .center,
                    startAngle: .degrees(startAngle), endAngle: .degrees(360
                    * progress + startAngle)))

            RingTip(progress: progress, startAngle: startAngle, ringRadius:
                radius)
                .frame(width: thickness, height: thickness)
                .foregroundColor(progress > 0.96 ? gradient.stops[1].color :
                    Color.clear)
                .shadow(color: progress > 0.96 ? Color.black.opacity(0.15) :
                    Color.clear, radius: 2, x: ringTipShadowOffset.x, y:
                    ringTipShadowOffset.y)
         }
        .frame(width: width, height: width, alignment: .center)
        .animation(Animation.easeInOut(duration: 1.0))
    }

//    private func getEndCircleShadowOffset(progress: Double) -> CGPoint {
```

Figure 20. The ring end now has a drop shadow

Exercise

Now that you've created a flexible circular progress bar, it's time to have an exercise. Your task is to make use of what you've built and create an activity ring. The app also needs to provide four buttons for adjusting the activity ring like you see in figure 21.

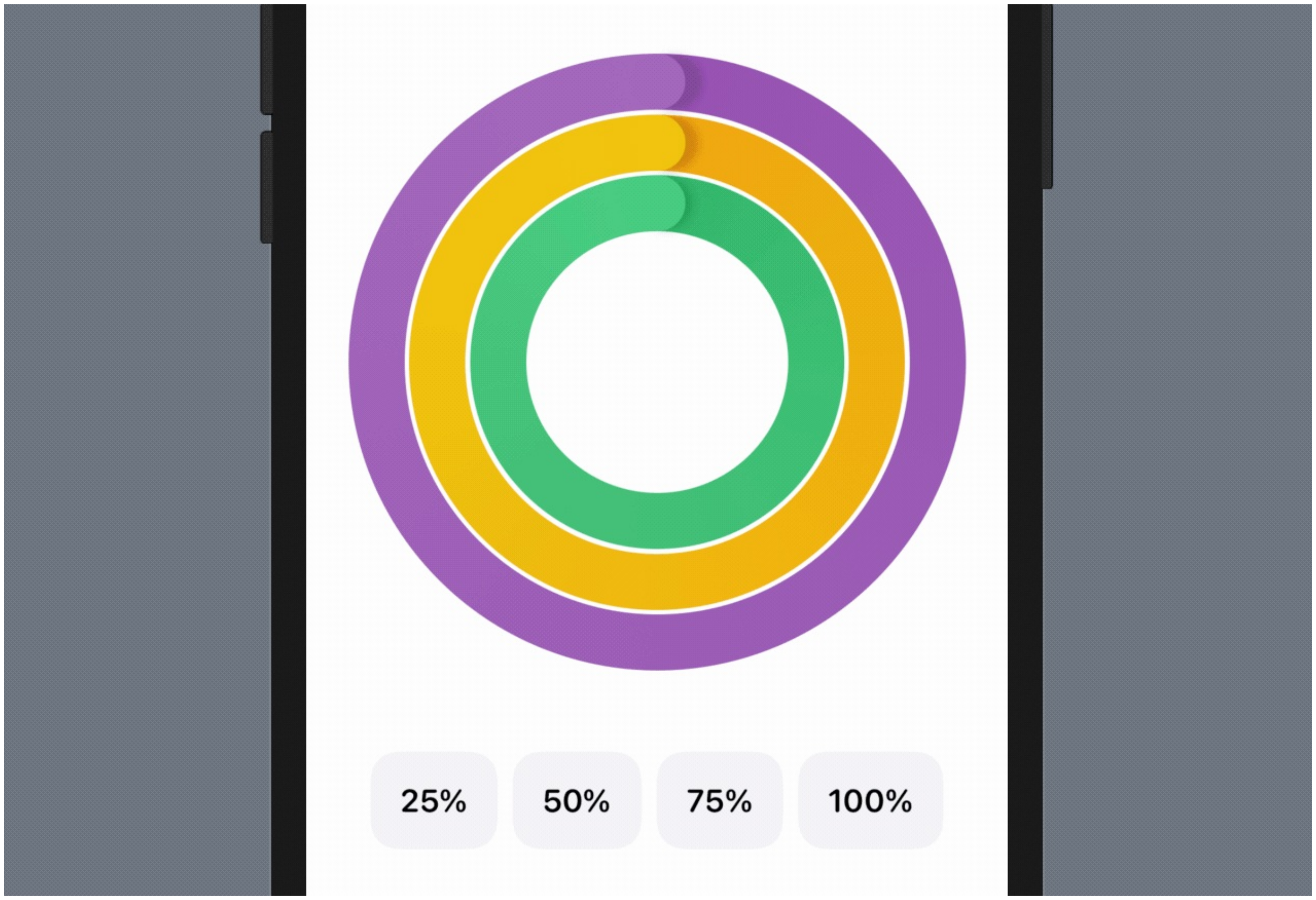


Figure 21. A sample activity ring

Summary

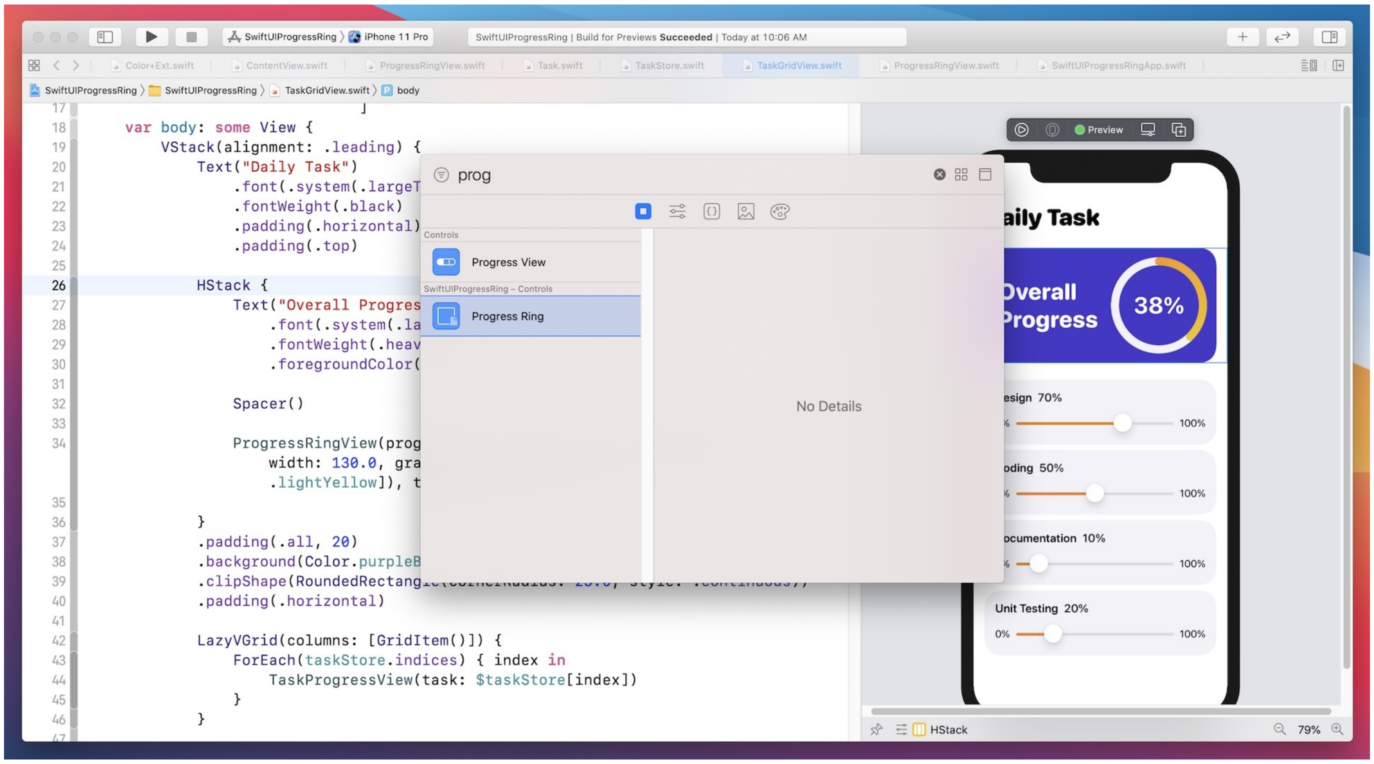
By building an activity ring, we covered a number of SwiftUI features in this chapter. You should now have a better idea of implementing your custom shape and how to animate a shape using the Animatable protocol.

For reference, you can download the complete project here:

- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUIProgressRing.zip)

Chapter 31
Working with AnimatableModifier and LibraryContentProvider

Earlier, you learned how to animate a shape by using `Animatable` and `AnimatableData`. In this chapter, we will take this further and show you how to animate a view using another protocol called `AnimatableModifier`. Additionally, I will walk you through a new feature of SwiftUI introduced in Xcode 12 that will allow developers to easily share a custom view to the View library and make it easier for reuse. Later, I will show you how to take the progress ring view to the View library for reuse. As a sneak peek, you can take a look at figure 1 or check out this demo video (https://link.appcoda.com/librarycontentprovider) to see how it works.



```
    var body: some View {
        VStack(alignment: .leading) {
            Text("Daily Task")
                .font(.system(.largeT
                .fontWeight(.black)
                .padding(.horizontal)
                .padding(.top)

            HStack {
                Text("Overall Progres
                    .font(.system(.la
                    .fontWeight(.heav
                    .foregroundColor(

                Spacer()

                ProgressRingView(prog
                    width: 130.0, gra
                    .lightYellow]), t

            }
            .padding(.all, 20)
            .background(Color.purpleB
            .clipShape(RoundedRectang
            .padding(.horizontal)

            LazyVGrid(columns: [GridItem()]) {
                ForEach(taskStore.indices) { index in
                    TaskProgressView(task: $taskStore[index])
                }
            }
```

Figure 1. Using a custom view in the View library

Understanding AnimatableModifier

Let's first look at the `AnimatableModifier` protocol. As its name suggests, `AnimatableModifier` is a view modifier and it conforms to the `Animatable` protocol. This makes it very powerful to animate value changes for different types of views.

```
protocol AnimatableModifier : Animatable, ViewModifier
```

So, what are we going to animate? We will build on top of what we've implemented in the previous chapter and add a text label at the center of the progress ring. The label will show the current percentage of progress. As the progress bar moves, the label will be updated accordingly. Figure 2 shows you what the label looks like.

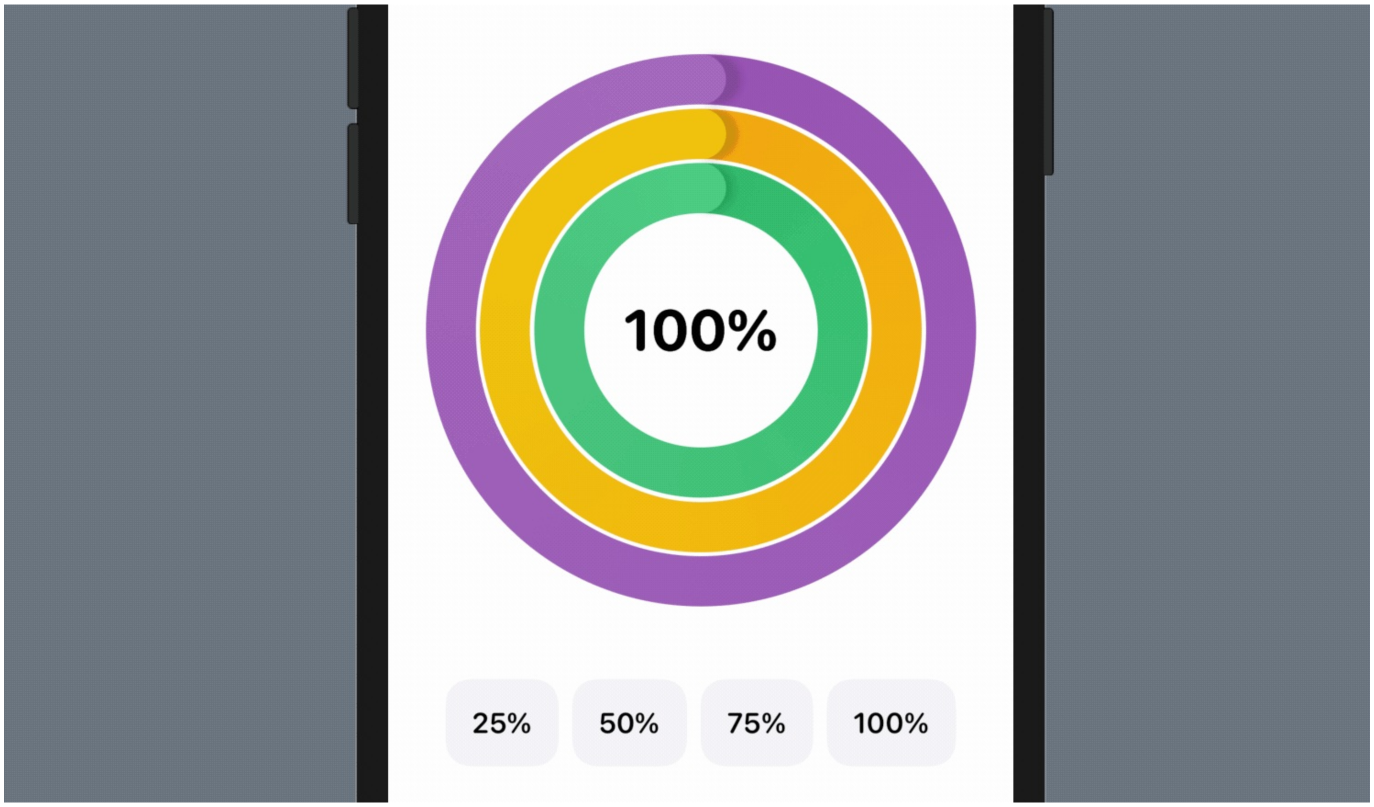


Figure 2. Animating the progress label

Animating Text using AnimatableModifer

I highly recommend you read chapter 30 first as this demo project is built on top of the previous one. In case you haven't worked on the project, you can download it at https://www.appcoda.com/resources/swiftui2/SwiftUIProgressRing.zip to get started.

Before we dive into the `AnimatableModifier` protocol, let me ask you. How are you going to layout the progress label and animate its change? Actually, we've built a similar progress indicator in chapter 9. So, base on what you learned, you may layout the progress label (in the `ProgressRingView.swift` file) like this:

```
ZStack {
    Circle()
        .stroke(Color(.systemGray6), lineWidth: thickness)

    Text(progressText)
        .font(.system(.largeTitle, design: .rounded))
        .fontWeight(.bold)
        .foregroundColor(.black)

    ...
}
```

You add a `Text` view in the `ZStack` and display the current progress in a formatted text using the below conversion:

```
private var progressText: String {
    let formatter = NumberFormatter()
    formatter.numberStyle = .percent
    formatter.percentSymbol = "%"

    return formatter.string(from: NSNumber(value: progress)) ?? ""
}
```

Since the `progress` variable is a state variable, the `progressText` will be automatically updated whenever the value of `progress` changes. This is a very straightforward implementation. However, there is an issue with the solution. The text animation doesn't work so well.

If you've made the above changes in `ProgressRingView.swift`, you can go back to `ContentView.swift` to see the result. The app does display the progress label, but when you change the progress from one value to another, the progress label immediately shows the new value using the fade animation.

This is not what we expect. The progress label shouldn't jump from one value (e.g. 100%) to another value (e.g. 50%) directly. We expect the progress label follows the animation of the progress bar and updates its value step by step like this:

100 -> 99 -> 98 -> 97 -> 96 ... ... 53 -> 52 -> 51 -> 50

The current implementation doesn't allow you to animate the text change. This is why I have to introduce you the `AnimatableModifier` protocol.

To animate the progress text, we will create a new struct called `ProgressTextModifier`, which adopts `AnimatableModifier`:

```
struct ProgressTextModifier: AnimatableModifier {

    var progress: Double = 0.0
    var textColor: Color = .primary

    private var progressText: String {
        let formatter = NumberFormatter()
        formatter.numberStyle = .percent
        formatter.percentSymbol = "%"

        return formatter.string(from: NSNumber(value: progress)) ?? ""
    }

    var animatableData: Double {
        get { progress }
        set { progress = newValue }
    }

    func body(content: Content) -> some View {
        content
            .overlay(
                Text(progressText)
                    .font(.system(.largeTitle, design: .rounded))
                    .fontWeight(.bold)
                    .foregroundColor(textColor)
                    .animation(nil)
            )
    }
}
```

Does the code look familiar to you? As mentioned earlier, the `AnimatableModifier` protocol conforms to both `Animatable` and `ViewModifier`. Therefore, we specify in the `animatableData` property what values to animate. Here, it's `progress`. To conform with the requirements of `ViewModifier`, we implement the `body` function and add the `Text` view.

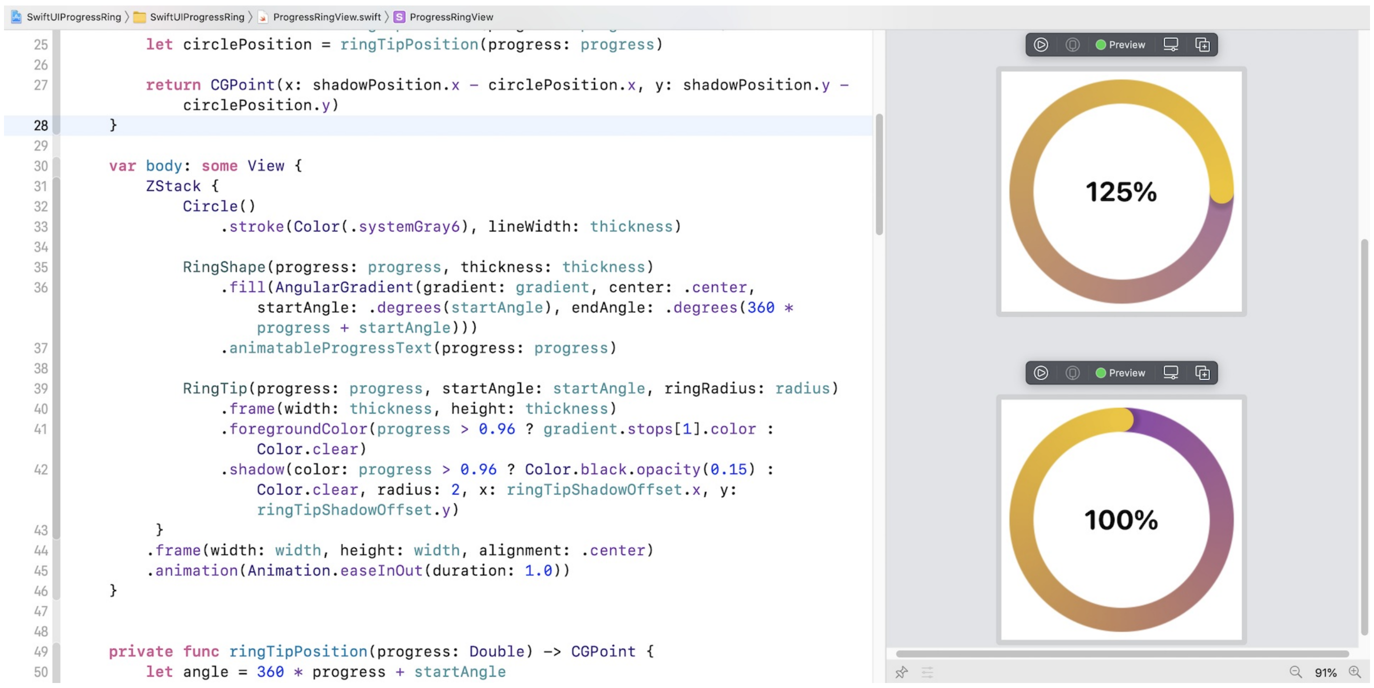
This is how we animate the text using `AnimatableModifier`. For convenience purposes, insert the following code, at the end of `ProgressRingView`, to create an extension for applying the `ProgressTextModifier`:

```
extension View {
    func animatableProgressText(progress: Double, textColor: Color = Color.primary)
 -> some View {
        self.modifier(ProgressTextModifier(progress: progress, textColor: textColo
r))
    }
}
```

Now you can attach the `animatableProgressText` modifier to `RingShape` like this:

```
RingShape(progress: progress, thickness: thickness)
    .fill(AngularGradient(gradient: gradient, center: .center, startAngle: .degree
s(startAngle), endAngle: .degrees(360 * progress + startAngle)))
    .animatableProgressText(progress: progress)
```

Once you have made the change, you should see the progress label in the preview canvas. To test the animation, run the app on an iPhone simulator or play the app in `ContentView.swift`. When you change the progress, the progress text now animates.

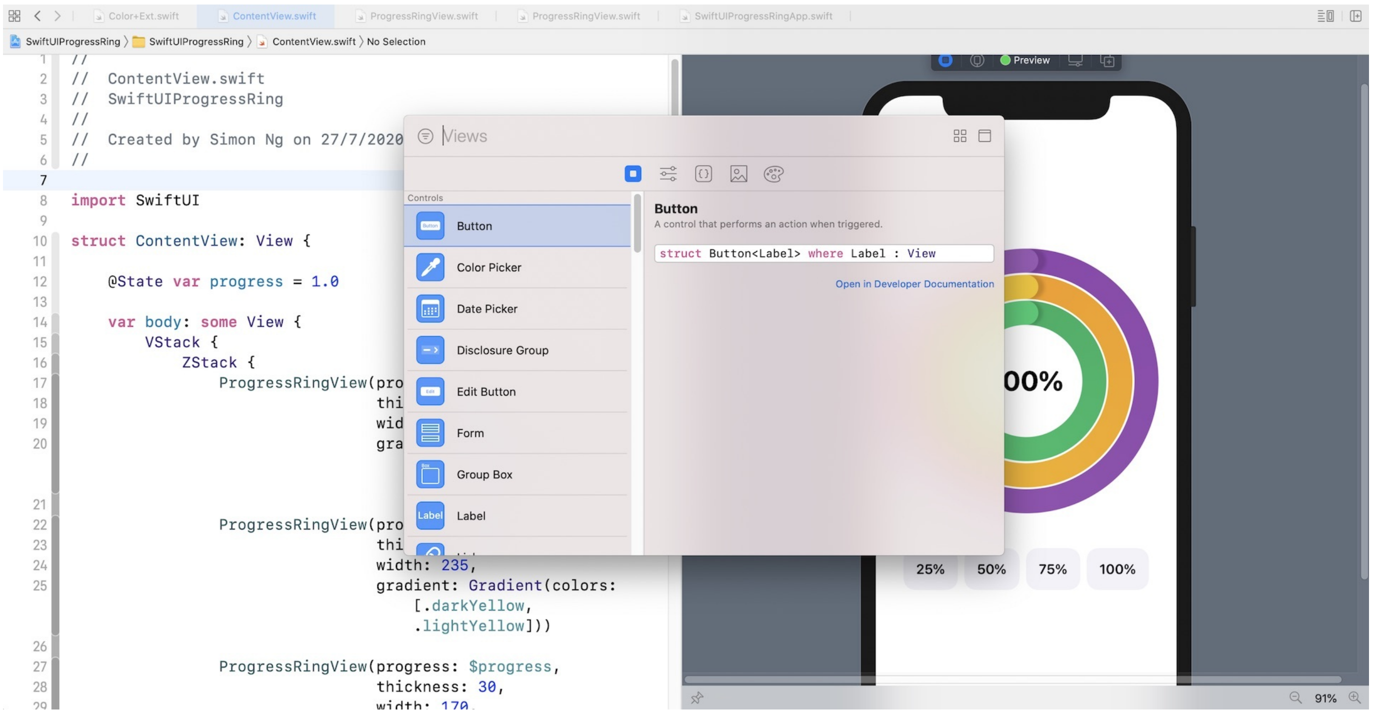

SwiftUIProgressRing 〉 SwiftUIProgressRing 〉 ProgressRingView.swift 〉 ProgressRingView
Preview
125%
Preview
100%
91%

```
        let circlePosition = ringTipPosition(progress: progress)

        return CGPoint(x: shadowPosition.x - circlePosition.x, y: shadowPosition.y -
            circlePosition.y)
    }

    var body: some View {
        ZStack {
            Circle()
                .stroke(Color(.systemGray6), lineWidth: thickness)

            RingShape(progress: progress, thickness: thickness)
                .fill(AngularGradient(gradient: gradient, center: .center,
                    startAngle: .degrees(startAngle), endAngle: .degrees(360 *
                    progress + startAngle)))
                .animatableProgressText(progress: progress)

            RingTip(progress: progress, startAngle: startAngle, ringRadius: radius)
                .frame(width: thickness, height: thickness)
                .foregroundColor(progress > 0.96 ? gradient.stops[1].color :
                    Color.clear)
                .shadow(color: progress > 0.96 ? Color.black.opacity(0.15) :
                    Color.clear, radius: 2, x: ringTipShadowOffset.x, y:
                    ringTipShadowOffset.y)
         }
        .frame(width: width, height: width, alignment: .center)
        .animation(Animation.easeInOut(duration: 1.0))
    }


    private func ringTipPosition(progress: Double) -> CGPoint {
        let angle = 360 * progress + startAngle
```

Figure 3. Displaying the progress label by applying the custom modifier

Using LibraryContentProvider

In Xcode 12, Apple introduced a new feature in the SwiftUI framework, allowing developers to take any custom views to the View library. If you forget what the View library is, just press *command-shift-L* to bring it up. The library lets you easily access all the available UI controls in SwiftUI. You can drag a control from the library and add it to the user interface directly.



```
//
//  ContentView.swift
//  SwiftUIProgressRing
//
//  Created by Simon Ng on 27/7/2020
//

import SwiftUI

struct ContentView: View {

    @State var progress = 1.0

    var body: some View {
        VStack {
            ZStack {
                ProgressRingView(pro
                                 thi
                                 wid
                                 gra


                ProgressRingView(pro
                                 thi
                                 width: 235,
                                 gradient: Gradient(colors:
                                    [.darkYellow,
                                    .lightYellow]))

                ProgressRingView(progress: $progress,
                                 thickness: 30,
                                 width: 170,
```

Button
A control that performs an action when triggered.

```
struct Button<Label> where Label : View
```

Figure 4. Displaying the progress label by applying the custom modifier

The new version of Xcode 12 now makes it possible to add your custom views the library by using a new protocol called `LibraryContentProvider`. To add a custom view to the View library, all you need to do is to create a new struct that conforms to the `LibraryContentProvider` protocol.

For example, to share the progress ring view to the View library, we can create a struct called `ProgressBar_Library` like this:

```
struct ProgressBar_Library: LibraryContentProvider {
    @LibraryContentBuilder var views: [LibraryItem] {
        LibraryItem(ProgressRingView(progress: .constant(1.0), thickness: 12.0, wi
dth: 130.0, gradient: Gradient(colors: [.darkYellow, .lightYellow])), title: "Prog
ress Ring", category: .control)
    }
}
```

The way to add a view to the View library is very simple. You create a struct that conforms to `LibraryContentProvider` and override the `views` property to return an array of custom views. In the code above, we return the progress ring view with some default values, name it "Progress Ring", and put it into the control category.

Optionally, if you want to add more than one library item, you can write the code like this:

```
struct ProgressBar_Library: LibraryContentProvider {
    @LibraryContentBuilder var views: [LibraryItem] {
        LibraryItem(ProgressRingView(progress: .constant(1.0), thickness: 12.0, wi
dth: 130.0, gradient: Gradient(colors: [.darkYellow, .lightYellow])), title: "Prog
ress Ring", category: .control)

        LibraryItem(ProgressRingView(progress: .constant(1.0), thickness: 30.0, wi
dth: 250.0, gradient: Gradient(colors: [.darkPurple, .lightYellow])), title: "Prog
ress Ring - Bigger", category: .control)
    }
}
```

As a side note, there are four possible values that can be given to item's category, depending on what the library item is supposed to represent:

- `control`
- `effect`
- `layout`
- `other`

You may also wonder what the `@LibraryContentBuilder` property wrapper is. It just saves you from writing the code for creating the array of `LibraryItem` instances. The code above can be rewritten like this:

```
struct ProgressBar_Library: LibraryContentProvider {
    var views: [LibraryItem] {
        return [LibraryItem(ProgressRingView(progress: .constant(1.0), thickness:
12.0, width: 130.0, gradient: Gradient(colors: [.darkYellow, .lightYellow])), titl
e: "Progress Ring", category: .control),

                LibraryItem(ProgressRingView(progress: .constant(1.0), thickness:
30.0, width: 250.0, gradient: Gradient(colors: [.darkPurple, .lightYellow])), titl
e: "Progress Ring - Bigger", category: .control)
    }
}
```

Once you create the struct, Xcode automatically discovers the implementation of the `LibraryContentProvider` protocol in your project and adds the progress ring view to the View library. You can now add the progress ring view to your UI by using drag and drop. *Note that at the time of this writing, you can't add documentation for your custom control.*

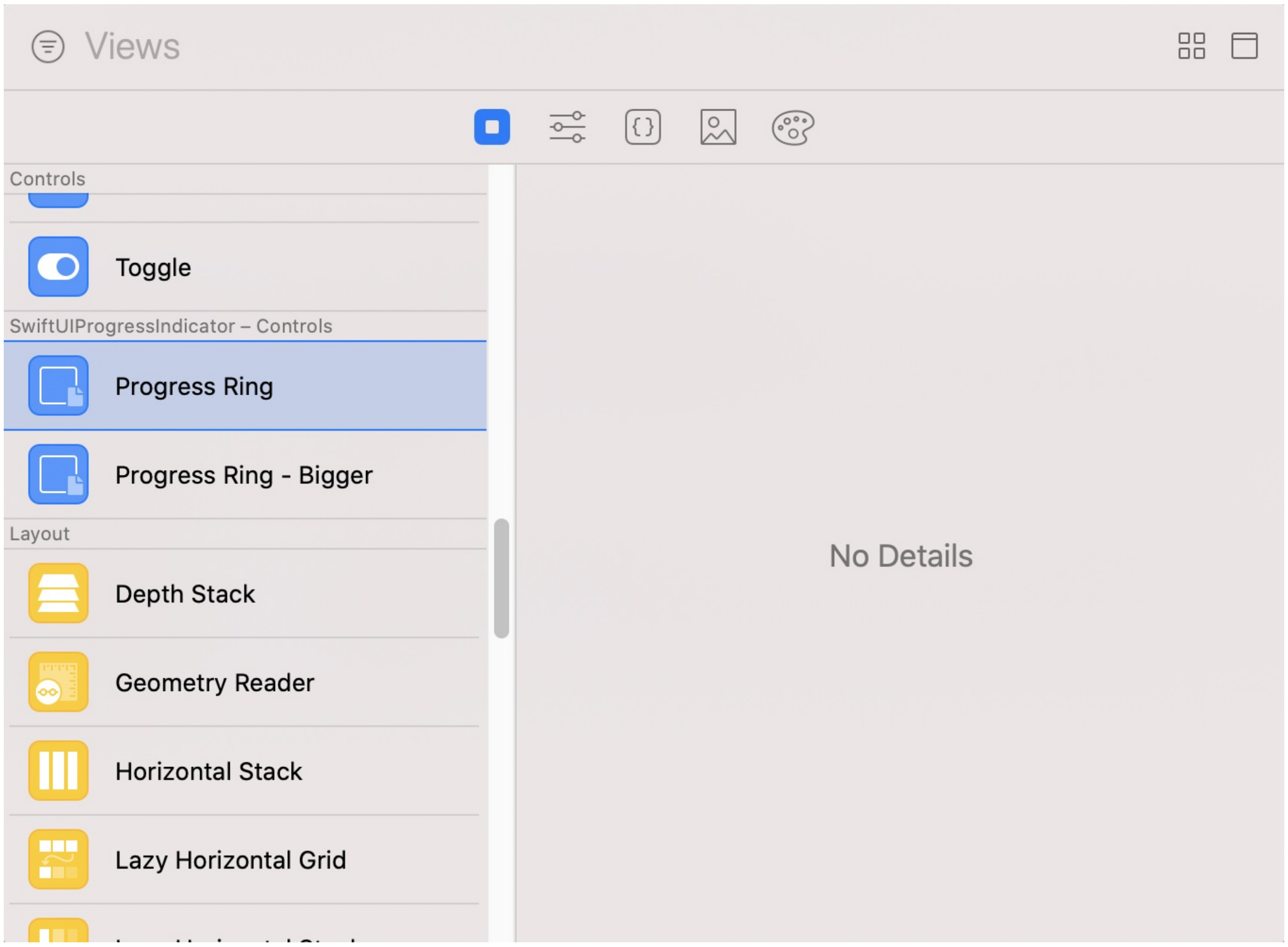


Figure 5. The progress ring view is added to the View library

Not only can you add a custom view to the Xcode library, you can also add your own modifiers by implementing the `modifiers` method and return the array of library items. You can add the `animatableProgressText` modifier to the View library by implementing the method like this:

```
struct ProgressBar_Library: LibraryContentProvider {
    .
    .
    .

    @LibraryContentBuilder
    func modifiers(base: Circle) -> [LibraryItem] {
        LibraryItem(base.animatableProgressText(progress: 1.0), title: "Progress I
ndicator", category: .control)
    }
}
```

The `base` parameter lets you specify the type of control that can be modified by the modifier. In the code above, it's the `Circle` view. Again, once you insert the code in `ProgressBar_Library`, Xcode will scan the library item and add it to the Modifier library.

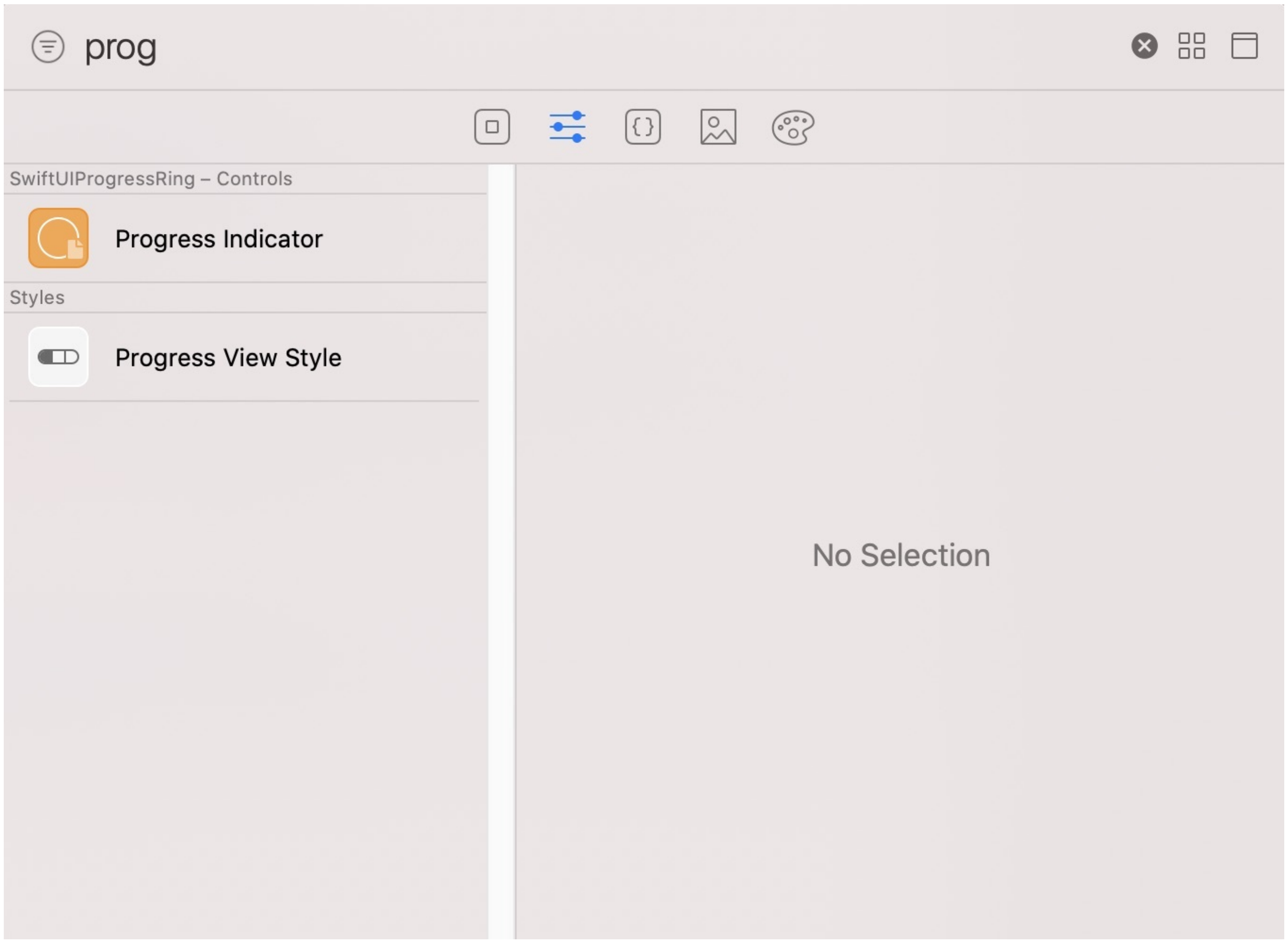


Figure 6. The progress indicator is added to the Modifier library

Exercise

The progress ring is now incorporated in the View library. Try to use it and build an app like below. The app has 4 sliders for adjusting the progress of different tasks. The overall progress is calculated by averaging the progress values of all tasks.

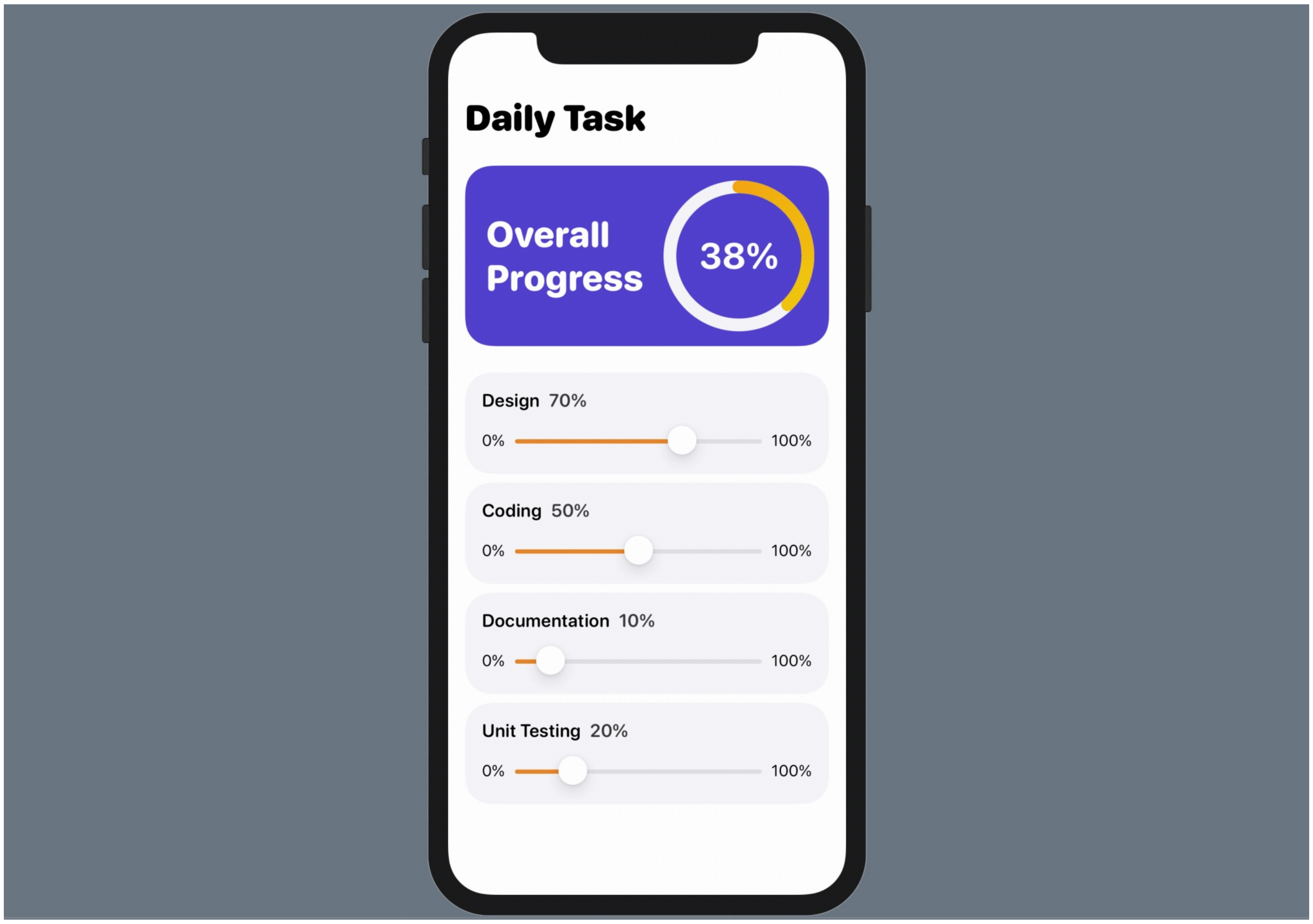


Figure 7. The progress ring view is added to the View library

Summary

The `AnimatableModifier` protocol is a very powerful protocol for animating changes of any views. In this chapter, we showed you how to animate the text change of a label. You can apply this technique to animate other values such as color and size.

The introduction of `LibraryContentProvider` makes it very easy for developers to share custom views and encourages code reuse. Imagine that you can build a library of custom components and put them into the View/Modifier library, every member in your team can easily access the controls and use them by drag & drop. Right now, you can only use the controls within the same Xcode project. We will discuss how you can make this possible by using Swift *Package* in the next chapter.

For reference, you can download the complete project here:

- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUITextAnimation.zip)

The solution of the exercise is also included in the demo project. Please refer to the `TaskGridView.swift` file.

Chapter 32
Working with TextEditor to Create Multiline Text Fields

The first version of SwiftUI, released along with iOS 13, didn't come with a native UI component for a multiline text field. To support multiline input, you still need to wrap a `UITextView` from the UIKit framework and make it available to your SwiftUI project by adopting the `UIViewRepresentable` protocol. In iOS 14, Apple introduced a new component called `TextEditor` for the SwiftUI framework. This `TextEditor` enables developers to display and edit multiline text in your apps. In this chapter, we will show you how to use `TextEditor` for multiline input.

Using TextEditor

It is very easy to use `TextEditor`. You just need to have a state variable to hold the input text. Then create a `TextEditor` instance in the body of your view like this:

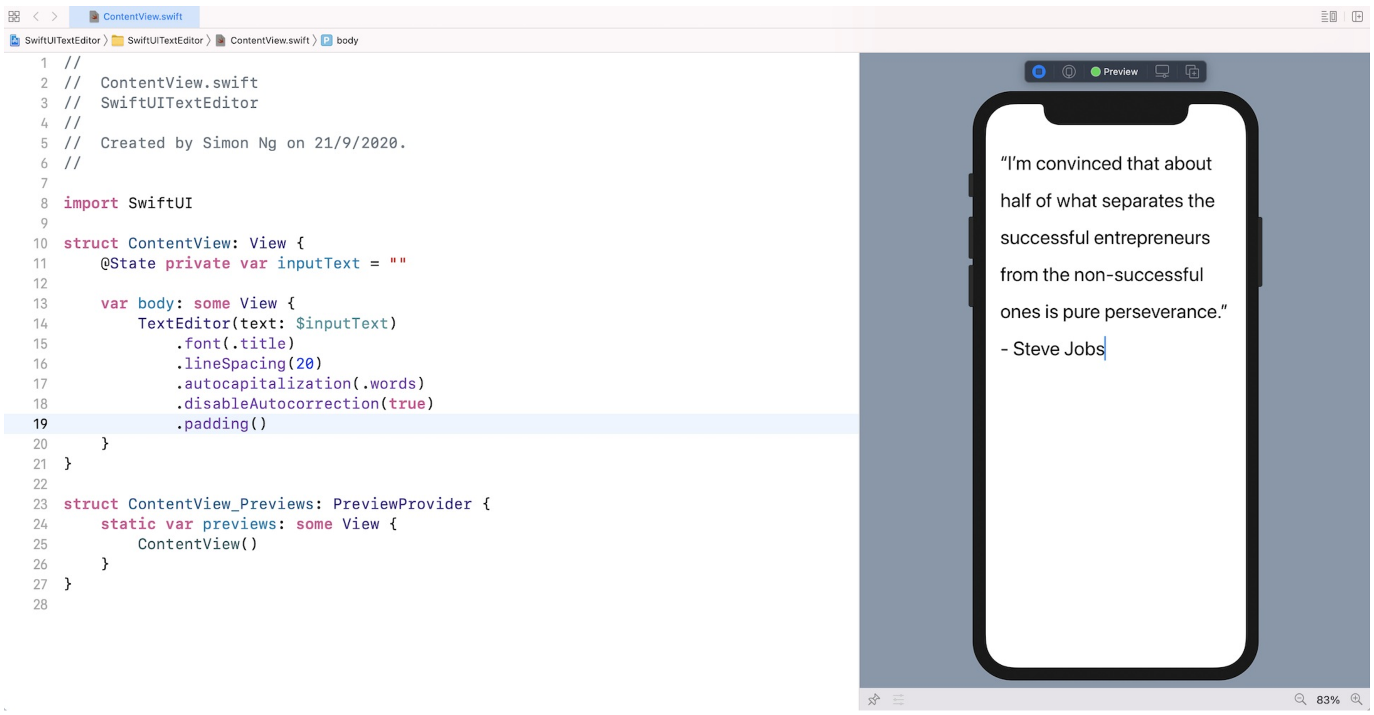
```
struct ContentView: View {
    @State private var inputText = ""

    var body: some View {
        TextEditor(text: $inputText)
    }
}
```

To instantiate the text editor, you pass the binding of `inputText` so that the state variable can store the user input.

You can customize the editor like any SwiftUI view. For example, the below code changes the font type and adjust the line spacing of the text editor:

```
TextEditor(text: $inputText)
    .font(.title)
    .lineSpacing(20)
    .autocapitalization(.words)
    .disableAutocorrection(true)
    .padding()
```

Optionally, you can enable/disable the auto-capitalization and auto-correction features.


ContentView.swift
SwiftUITextEditor 〉 SwiftUITextEditor 〉 ContentView.swift 〉 body
Preview
83%

```
//
//  ContentView.swift
//  SwiftUITextEditor
//
//  Created by Simon Ng on 21/9/2020.
//

import SwiftUI

struct ContentView: View {
    @State private var inputText = ""

    var body: some View {
        TextEditor(text: $inputText)
            .font(.title)
            .lineSpacing(20)
            .autocapitalization(.words)
            .disableAutocorrection(true)
            .padding()
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

"I'm convinced that about half of what separates the successful entrepreneurs from the non-successful ones is pure perseverance." - Steve Jobs

Figure 1. Using TextEditor

Using the onChange() Modifier to Detect Text Input Change

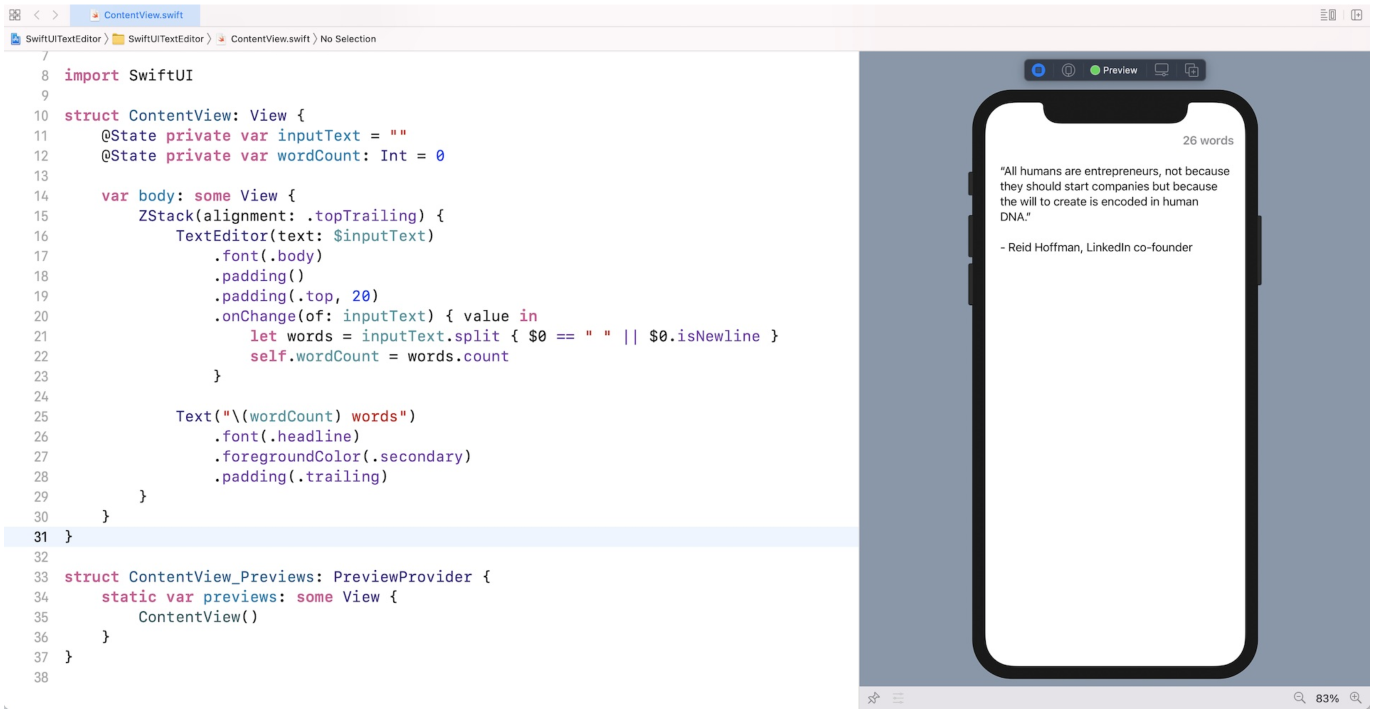
`UITextView` of the UIKit framework, works with the `UITextViewDelegate` protocol to handle editing changes. So, how about `TextEditor`? How do we detect the change of user input and perform further processing?

The new version of SwiftUI introduces an `onChange()` modifier which can be attached to `TextEditor` or any other view. Let's say you are building a note application using `TextEditor` and need to display a word count in real time, you can attach the `onChange()` modifier to `TextEditor` like this:

```
struct ContentView: View {
    @State private var inputText = ""
    @State private var wordCount: Int = 0

    var body: some View {
        ZStack(alignment: .topTrailing) {
            TextEditor(text: $inputText)
                .font(.body)
                .padding()
                .padding(.top, 20)
                .onChange(of: inputText) { value in
                    let words = inputText.split { $0 == " " || $0.isNewline }
                    self.wordCount = words.count
                }

            Text("\(wordCount) words")
                .font(.headline)
                .foregroundColor(.secondary)
                .padding(.trailing)
        }
    }
}
```

In the code above, we declare a state property to store the word count. And, we specify in the `onChange()` modifier to monitor the change of `inputText`. Whenever a user types a character, the code inside the `onChange()` modifier will be invoked. In the closure, we compute the total number of words in `inputText` and update the `wordCount` variable accordingly.

If you run the code in a simulator, you should see a plain text editor that also displays the word count in real time.


ContentView.swift
SwiftUITextEditor 〉 SwiftUITextEditor 〉 ContentView.swift 〉 No Selection
Preview
83%

```swift
import SwiftUI

struct ContentView: View {
    @State private var inputText = ""
    @State private var wordCount: Int = 0

    var body: some View {
        ZStack(alignment: .topTrailing) {
            TextEditor(text: $inputText)
                .font(.body)
                .padding()
                .padding(.top, 20)
                .onChange(of: inputText) { value in
                    let words = inputText.split { $0 == " " || $0.isNewline }
                    self.wordCount = words.count
                }

            Text("\(wordCount) words")
                .font(.headline)
                .foregroundColor(.secondary)
                .padding(.trailing)
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

26 words

"All humans are entrepreneurs, not because they should start companies but because the will to create is encoded in human DNA."

- Reid Hoffman, LinkedIn co-founder

Figure 2. Using onChange() to detect text input and display the word count

Summary

Since the initial release of SwiftUI, `TextEditor` has been one of the most anticipated UI components. You can now use this native component to handle multiline input on iOS 14. However, if you still need to support older versions of iOS, you will need to use UIKit and implement `UITextView` in your SwiftUI project using the `UIViewRepresentable` protocol.

For reference, you can download the complete text editor project here:

- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUITextEditor.zip)

Chapter 33
Using matchedGeometryEffect to Create View Animations

In iOS 14, Apple introduced a lot of new additions to the SwiftUI framework like LazyVGrid and LazyHGrid. But `matchedGeometryEffect` is a new one that really caught my attention because it allows developers to create some amazing view animations with just a few lines of code. In earlier chapters, you learned how to create view animations. `matchedGeometryEffect` takes the implementation of view animations to the next level.

For any mobile apps, it is very common that you need to move from one view to another. Creating a delightful transition between views will definitely improve the user experience. With the `matchedGeometryEffect` modifier, you describe the appearance of two views. The modifier will then compute the difference between those two views and automatically animate the size/position change.

Feeling confused? No worries. You will understand what I mean after going through the demo apps.

Revisiting SwiftUI Animation

Before I walk you through the usage of `matchedGeometryEffect`, let's take a look at how we implement animation using SwiftUI. Figure 1 shows the beginning and final states of a view. When you tap the circle view on your left, it should grow bigger and move upward. Conversely, if you tap the one on the right, it returns to the original size and position.

```
}
```

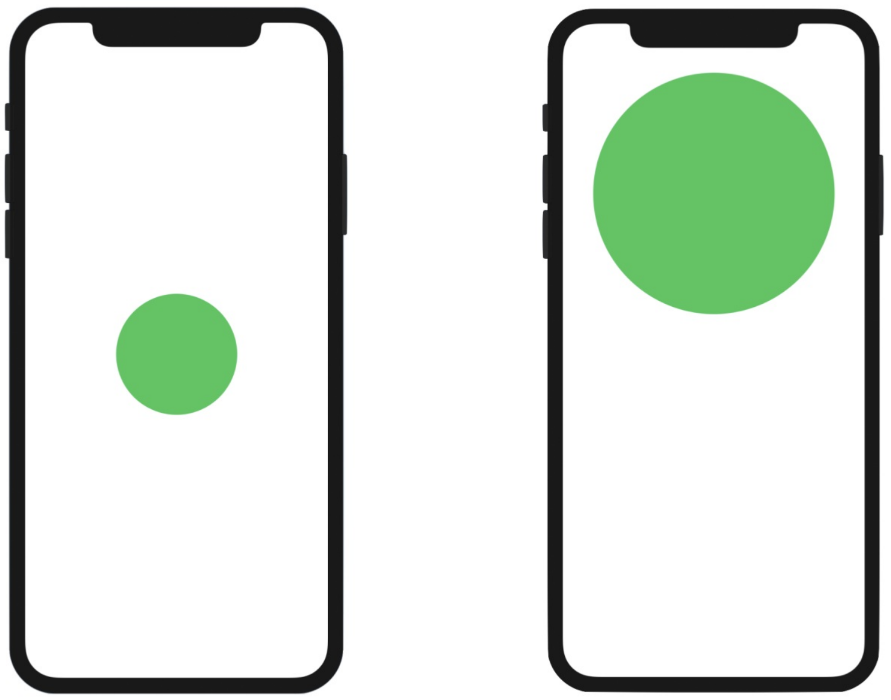
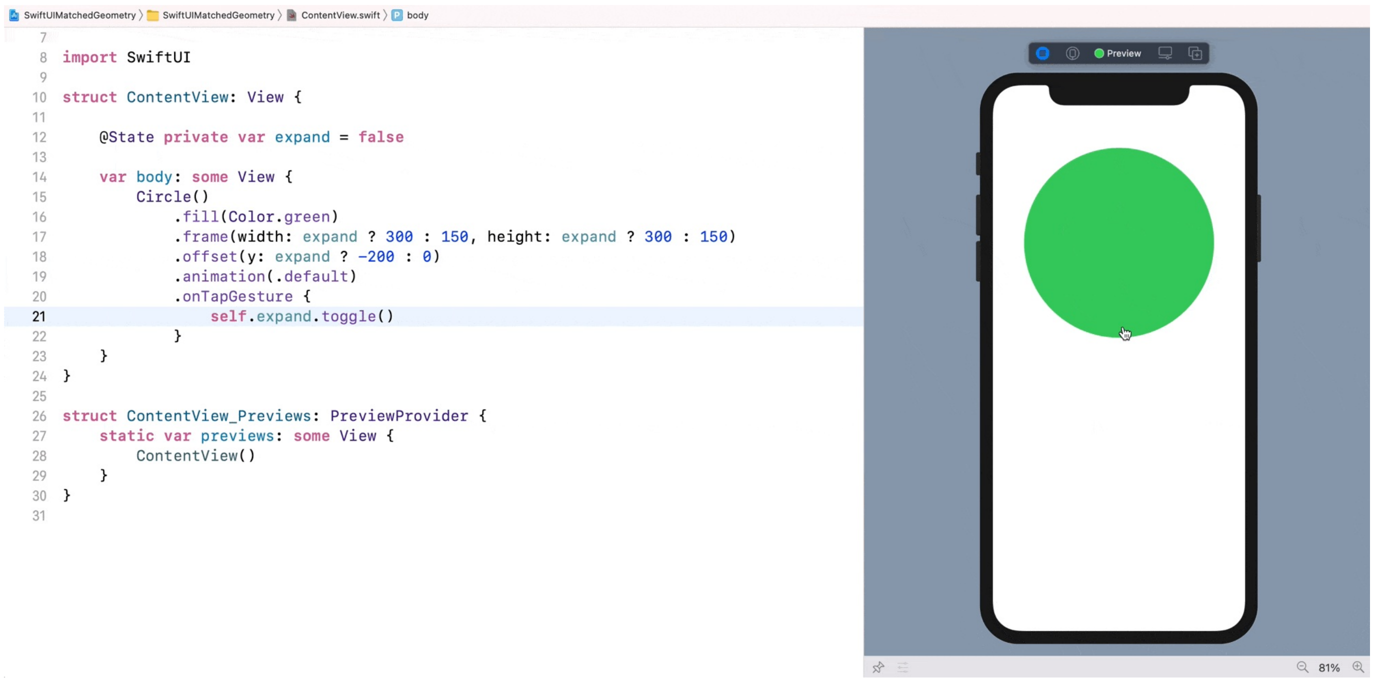


Figure 1. The Circle view at the start state (left), The Circle view at the end state (right)

The implementation of this tappable circle is very straightforward. Assuming you've created a new SwiftUI project, you can update the `ContentView` struct like this:

```
struct ContentView: View {

    @State private var expand = false

    var body: some View {
        Circle()
            .fill(Color.green)
            .frame(width: expand ? 300 : 150, height: expand ? 300 : 150)
            .offset(y: expand ? -200 : 0)
            .animation(.default)
            .onTapGesture {
                self.expand.toggle()
            }
    }
}
```

We have a state variable `expand` to keep track of the current state of the `Circle` view. In both the `.frame` and `.offset` modifiers, we vary the frame size and offset when the state changes. If you run the app in the preview canvas, you should see the animation when you tap the circle.


SwiftUIMatchedGeometry 〉 SwiftUIMatchedGeometry 〉 ContentView.swift 〉 body
Preview
81%

```swift
import SwiftUI

struct ContentView: View {

    @State private var expand = false

    var body: some View {
        Circle()
            .fill(Color.green)
            .frame(width: expand ? 300 : 150, height: expand ? 300 : 150)
            .offset(y: expand ? -200 : 0)
            .animation(.default)
            .onTapGesture {
                self.expand.toggle()
            }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 2. The Circle view animation

Understanding the matchedGeometryEffect Modifier

So, what is `matchedGeometryEffect` ? How does it simplify the implementation of the view animation? Take a look at figure 1 and the code of the circle animation again. We have to figure out the exact value change between the start and the final state. In the example, they are the frame size and the offset.

With the `matchedGeometryEffect` modifier, you no longer need to figure out these differences. All you need to do is describe two views: one represents the start state and the other is for the final state. `matchedGeometryEffect` will automatically interpolate the size and position between the views.

To create the same animation as shown in figure 2 with `matchedGeometryEffect`, you first declare a namespace variable:

```
@Namespace private var shapeTransition
```

And then, rewrite the `body` part like this:

```
var body: some View {
    if expand {

        // Final State
        Circle()
            .fill(Color.green)
            .matchedGeometryEffect(id: "circle", in: shapeTransition)
            .frame(width: 300, height: 300)
            .offset(y: -200)
            .animation(.default)
            .onTapGesture {
                self.expand.toggle()
            }

    } else {

        // Start State
        Circle()
            .fill(Color.green)
            .matchedGeometryEffect(id: "circle", in: shapeTransition)
            .frame(width: 150, height: 150)
            .offset(y: 0)
            .animation(.default)
            .onTapGesture {
                self.expand.toggle()
            }
    }
}
```

In the code, we created two circle views: one is for the start state and the other is for the final state. When our app first initialized, we present a `Circle` view which is centered and has a width of 150 points. When the `expand` state variable is changed from `false` to `true`, the app displays another `Circle` view which is positioned 200 points from the center of the screen and has a width of 300 points.

For both `Circle` views, we attach the `matchedGeometryEffect` modifier and specify the same ID & namespace. By doing so, SwiftUI computes the size & position difference between these two views and interpolates the transition. Along with the `animation` modifier, the framework will automatically animate the transition.

The ID and namespace are used for identifying which views are part of the same transition. This is why both `Circle` views use the same ID and namespace.

This is how you use `matchedGeometryEffect` to animate transition between two views. If you've used Magic Move in Keynote before, this new modifier is very much like *Magic Move*. To test the animation, I suggest you run the app in an iPhone simulator. At the time of this writing, there is a bug in Xcode 12 that you can't test the animation in the preview canvas.

Morphing From a Circle to a Rounded Rectangle

Let's try to implement another animated view transition. This time, we will morph a circle into a rounded rectangle. The circle is positioned at the top of the screen, while the rounded rectangle is close to the bottom part of the screen.

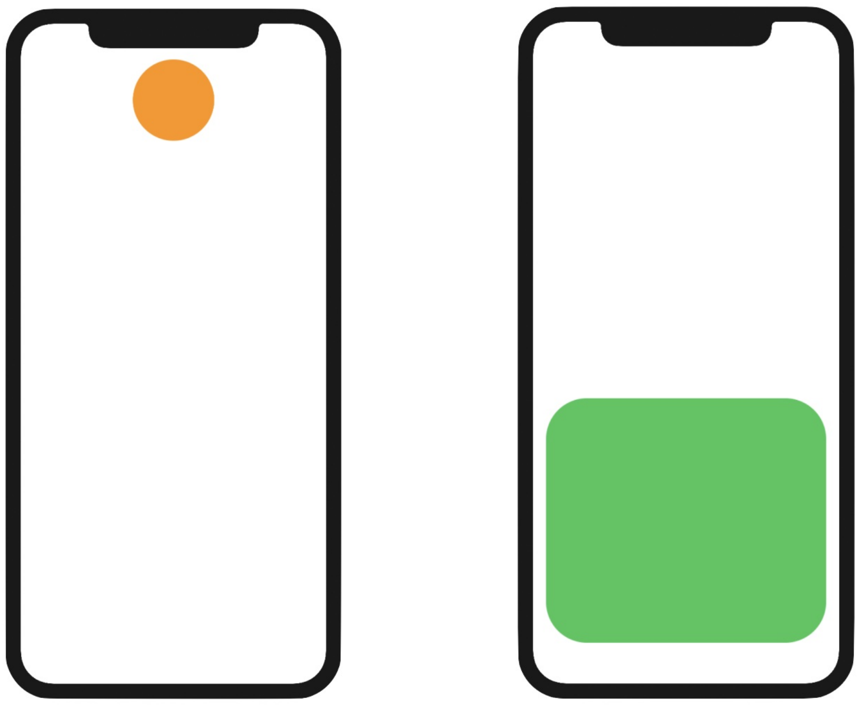


Figure 3. The Circle view at the start state (left), The Rounded Rectangle view at the end state (right)

Using the same technique you just learned, you need to prepare two views: the circle view and the rounded rectangle view. The `matchedGeometryEffect` modifier will then handle the transformation. Replace the `body` variable of the `ContentView` struct like this:

```
VStack {
    if expand {

        // Rounded Rectangle
        Spacer()

        RoundedRectangle(cornerRadius: 50.0)
            .matchedGeometryEffect(id: "circle", in: shapeTransition)
            .frame(minWidth: 0, maxWidth: .infinity, maxHeight: 300)
            .padding()
            .foregroundColor(Color(.systemGreen))
            .animation(.easeIn)
            .onTapGesture {
                expand.toggle()
            }

    } else {

        // Circle
        RoundedRectangle(cornerRadius: 50.0)
            .matchedGeometryEffect(id: "circle", in: shapeTransition)
            .frame(width: 100, height: 100)
            .foregroundColor(Color(.systemOrange))
            .animation(.easeIn)
            .onTapGesture {
                expand.toggle()
            }

        Spacer()
    }
}
```

We still make use of the `expand` state variable to toggle between the circle view and the rounded rectangle view. The code is very similar to the previous example, except that we use a `VStack` and a `Spacer` to position the view. You may wonder why we used `RoundedRectangle` to create the circle. The main reason is that it gives you a more smooth transition.

For both views, we attach the `matchedGeometryEffect` modifier and specify the same ID & namespace. That's all we need to do. The modifier will compare the difference between these two views and animate the changes. If you run the app in the preview canvas or on an iPhone simulator, you will see a nice transition between the circle and the rounded rectangle views. This is the magic of `matchedGeometryEffect`.

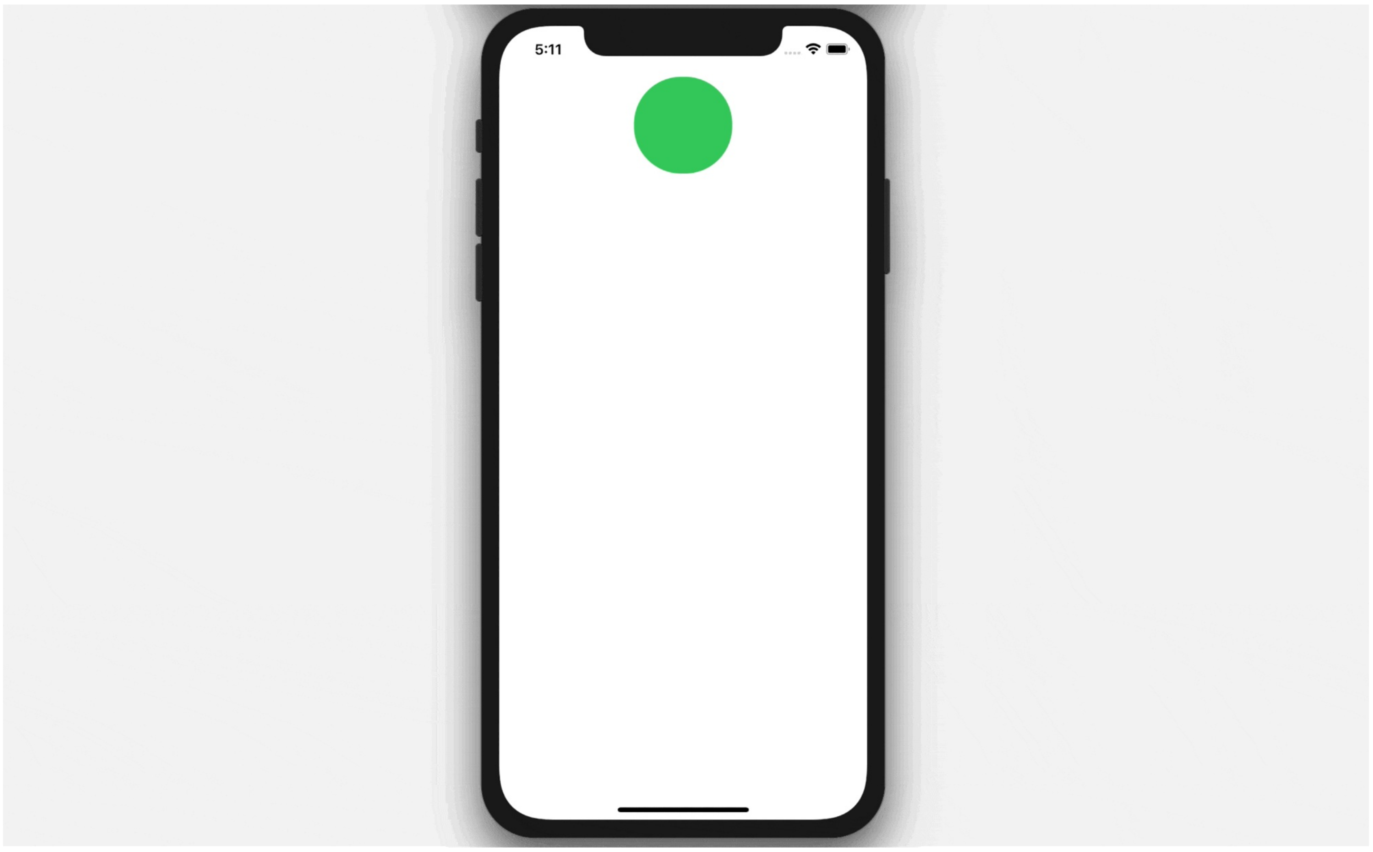


Figure 4. The Circle view animation

However, you may notice that the modifier doesn't animate the color change. This is right. `matchedGeometryEffect` only handles position and size changes.

Exercise #1

Let's have a simple exercise to test your understanding of `matchedGeometryEffect`. Your task is to create the animated transition as shown in figure 5. It starts with an orange circle view. When the circle is tapped, it will transform into a full screen background. You can find the solution in the final project.

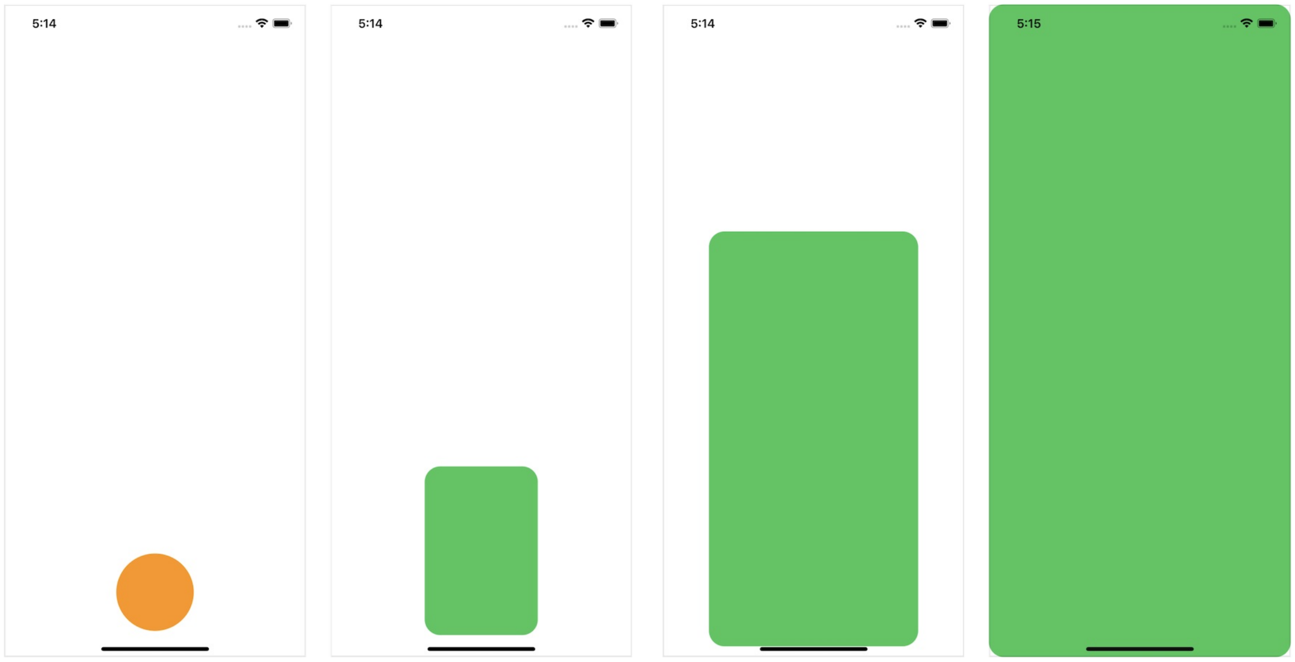


Figure 5. Transforming a circle button to a full screen background

Swapping Two Views with Animated Transition

Now that you have some basic knowledge of `matchedGeometryEffect`, let's continue to see how it can help us create some nice animations. In this example, we will swap the position of two circle views and apply a modifier to create a smooth transition.

Figure 6. Swapping the position of two circles

We will use a state variable to store the state of the swap and create a namespace variable for `matchedGeometryEffect`. Declare the following variable in `ContentView`:

```
@State private var swap = false

@Namespace private var dotTransition
```

By default, the orange circle is on the left side of the screen, while the green circle is positioned on the right. When the user taps any of the circles, it will trigger the swap. You don't need to figure out how the swap is done when using `matchedGeometryEffect`. To create the transition, all you need to do is:

1. Create the layout of the orange and green circles before the swap
2. Create the layout of the two circles after the swap

To translate the layout into code, you write the `body` variable like this:

```
if swap {
        // After swap
        // Green dot on the left, Orange dot on the right

        HStack {
            Circle()
                .fill(Color.green)
                .frame(width: 30, height: 30)
                .matchedGeometryEffect(id: "greenCircle", in: dotTransition)

            Spacer()

            Circle()
                .fill(Color.orange)
                .frame(width: 30, height: 30)
                .matchedGeometryEffect(id: "orangeCircle", in: dotTransition)
        }
        .frame(width: 100)
        .animation(.linear)
        .onTapGesture {
            swap.toggle()
        }

    } else {

        // Start state
        // Orange dot on the left, Green dot on the right

        HStack {
            Circle()
                .fill(Color.orange)
                .frame(width: 30, height: 30)
                .matchedGeometryEffect(id: "orangeCircle", in: dotTransition)

            Spacer()

            Circle()
                .fill(Color.green)
                .frame(width: 30, height: 30)
                .matchedGeometryEffect(id: "greenCircle", in: dotTransition)
        }
        .frame(width: 100)
        .animation(.linear)
        .onTapGesture {
            swap.toggle()
        }
    }
```

We use a `HStack` to layout the two circles horizontally and have a `Spacer` in between them to create some separation. When the `swap` variable is set to `true`, the green circle is placed to the left of the orange circle. When `false`, the green circle is positioned to the right of the orange circle.

As you can see, we just describe the layout of the circle views in difference states and let `matchedGeometryEffect` handle the rest. We attach the modifier to each of the `Circle` views. However, this time is a bit different. Since we have two different `Circle` views to match, we use two distinct IDs for the `matchedGeometryEffect` modifier. For the orange circles, we set the identifier to `orangeCircle`, while the green circles uses the identifier `greenCircle`.

Run the app on a simulator, you should see the swap animation when you tap any of the circles.

Exercise #2

Earlier, we used the `matchedGeometryEffect` on two circles and swap their position. Your exercise is to apply the same technique but on two images. Figure 6 shows you the sample UI. When the swap button is tapped, the app swaps the two photos with a nice animation.

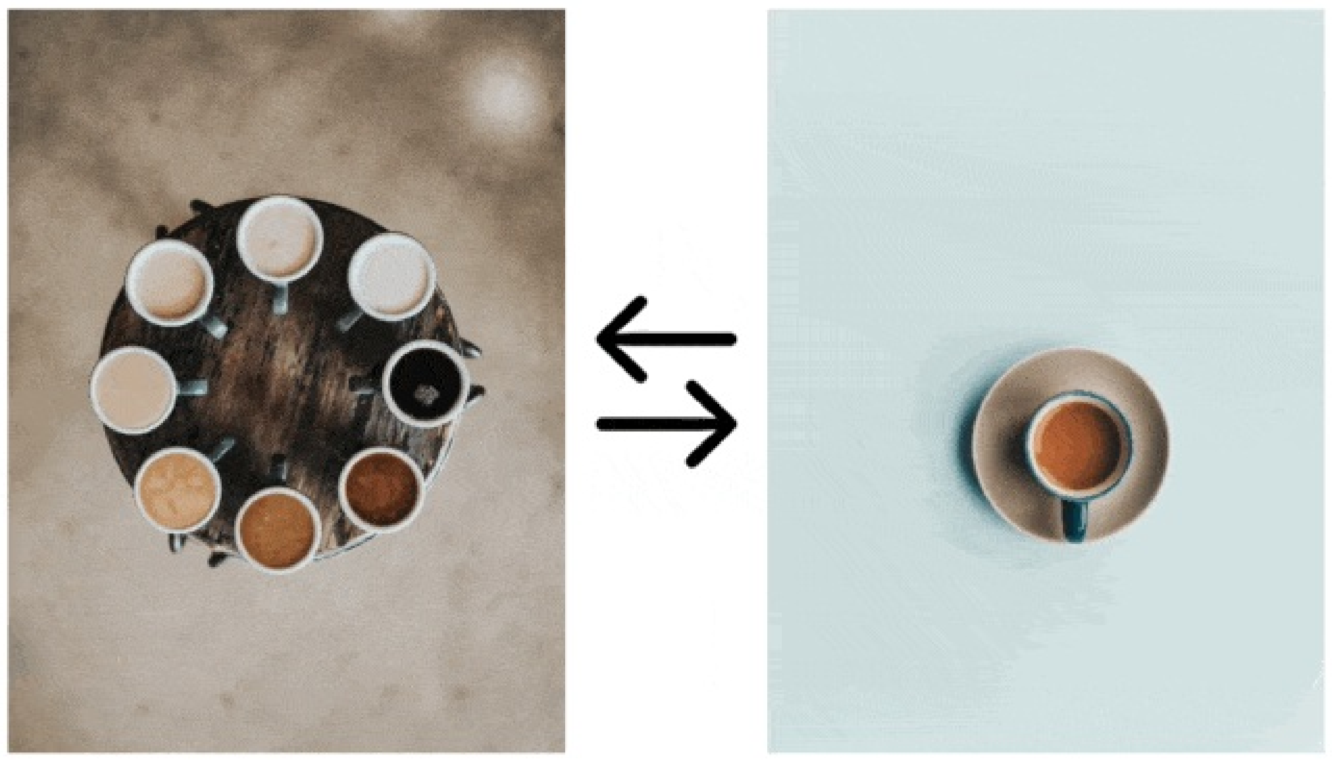


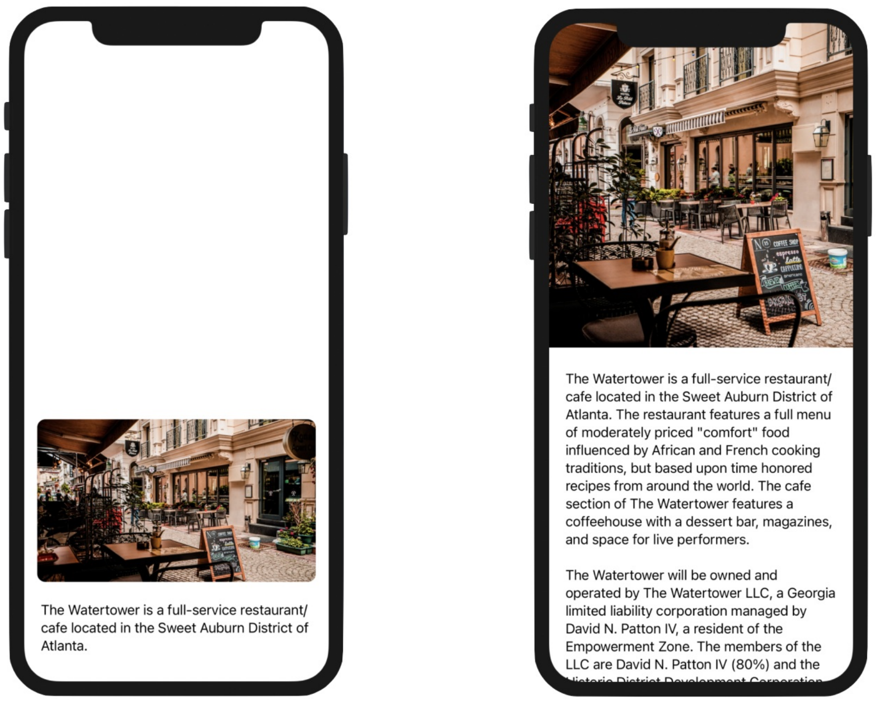
Figure 7. Swapping the position of two photos

You are free to use your own photos. For my demo, I used these free photos from Unsplash.com:

- https://unsplash.com/photos/pMW4jzELQCw
- https://unsplash.com/photos/PM4Vu1B0gxk

Creating a Basic Hero Animation

Other than transforming from one shape to another, you can use the `matchedGeometryEffect` modifier to create a basic hero animation. Figure 8 shows you a sample stack view of an image and text. When the view is tapped, both the image and text will be expanded to take up the full screen. This type of animation is usually known as a Hero Animation.


The Watertower is a full-service restaurant/ cafe located in the Sweet Auburn District of Atlanta.

The Watertower is a full-service restaurant/ cafe located in the Sweet Auburn District of Atlanta. The restaurant features a full menu of moderately priced "comfort" food influenced by African and French cooking traditions, but based upon time honored recipes from around the world. The cafe section of The Watertower features a coffeehouse with a dessert bar, magazines, and space for live performers.

The Watertower will be owned and operated by The Watertower LLC, a Georgia limited liability corporation managed by David N. Patton IV, a resident of the Empowerment Zone. The members of the LLC are David N. Patton IV (80%) and the

Figure 8. Expanding a stack view into a full screen

Again, we apply the `matchedGeometryEffect` technique to create this type of animated transition. If you refer to figure 8, there are two views in the view transition:

1. One is the view showing a smaller image and an excerpt for the article.
2. The other one is the view expanded into full screen showing a featured photo and the full article.

To begin, first declare a state variable to control the status of the view mode:

```
@State private var showDetail = false
```

When `showDetail` is set to false, the article view with a smaller image is displayed. when true, a full screen article view will be shown. Again, to use the `matchedGeometryEffect` modifier, we have to declare a namespace variable:

```
@Namespace private var articleTransition
```

Next, update the `body` variable like this:

```
// Display an article view with smaller image
if !showDetail {
    VStack {
        Spacer()

        VStack {
            Image("latte")
                .resizable()
                .scaledToFill()
                .frame(minWidth: 0, maxWidth: .infinity)
                .frame(height: 200)
                .matchedGeometryEffect(id: "image", in: articleTransition)
                .cornerRadius(10)
                .animation(.interactiveSpring())
                .padding()
                .onTapGesture {
                    showDetail.toggle()
                }

            Text("The Watertower is a full-service restaurant/cafe located in the
Sweet Auburn District of Atlanta.")
                .matchedGeometryEffect(id: "text", in: articleTransition)
                .animation(nil)
                .padding(.horizontal)
        }
    }
}

// Display the article view in a full screen
if showDetail {

    ScrollView {
        VStack {
            Image("latte")
                .resizable()
                .scaledToFill()
                .frame(minWidth: 0, maxWidth: .infinity)
                .frame(height: 400)
                .clipped()
```

```
                    .matchedGeometryEffect(id: "image", in: articleTransition)
                    .animation(.interactiveSpring())
                    .onTapGesture {
                        showDetail.toggle()
                    }

                Text("The Watertower is a full-service restaurant/cafe located in the
 Sweet Auburn District of Atlanta. The restaurant features a full menu of moderatel
y priced \"comfort\" food influenced by African and French cooking traditions, but
 based upon time honored recipes from around the world. The cafe section of The Wa
tertower features a coffeehouse with a dessert bar, magazines, and space for live
performers.\n\nThe Watertower will be owned and operated by The Watertower LLC, a
Georgia limited liability corporation managed by David N. Patton IV, a resident of
 the Empowerment Zone. The members of the LLC are David N. Patton IV (80%) and the
 Historic District Development Corporation (20%).\n\nThis business plan offers fin
ancial institutions an opportunity to review our vision and strategic focus. It al
so provides a step-by-step plan for the business start-up, establishing favorable
sales numbers, gross margin, and profitability.\n\nThis plan includes chapters on
the company, products and services, market focus, action plans and forecasts, mana
gement team, and financial plan.")
                    .matchedGeometryEffect(id: "text", in: articleTransition)
                    .animation(.easeOut)
                    .padding(.all, 20)

                Spacer()
            }

        }
        .edgesIgnoringSafeArea(.all)
}
```

In the code above, we layout the views in different states. When `showDetail` is set to `false`, we use a `VStack` to layout the article image and the excerpt. The height of the image is set to 200 points to make it smaller.

The layout of the article view is very similar in full screen mode. The main difference is that the `VStack` view is embedded in a `ScrollView` to make the content scrollable. The image's height is set to 400 points, so that the image is a little bit bigger. In order to extend the image and text views outside of the screen's safe area, we attach the `.edgesIgnoringSafeArea` modifier to the scroll view and set its value to `.all`.

Since we have two different views in the transition, we use two different IDs for the `matchedGeometryEffect` modifier. For the image, we set the ID to `image` :

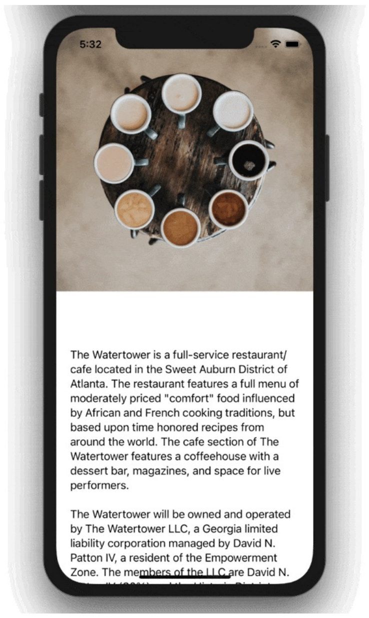
```
.matchedGeometryEffect(id: "image", in: articleTransition)
```

On the other hand, we set the ID of the text view to `text` :

```
.matchedGeometryEffect(id: "text", in: articleTransition)
```

Furthermore, we use two different animations for the text and image views. We apply the `.interactiveSpring` animation for the image view, while for the text view, we use the `.easeOut` animation.

The implementation is very straightforward, similar to what we have done in the earlier examples. Run the app in a simulator to try it out. When you tap the image view, the app renders a nice animation and shows the article in full screen.



The Watertower is a full-service restaurant/cafe located in the Sweet Auburn District of Atlanta. The restaurant features a full menu of moderately priced "comfort" food influenced by African and French cooking traditions, but based upon time honored recipes from around the world. The cafe section of The Watertower features a coffeehouse with a dessert bar, magazines, and space for live performers.

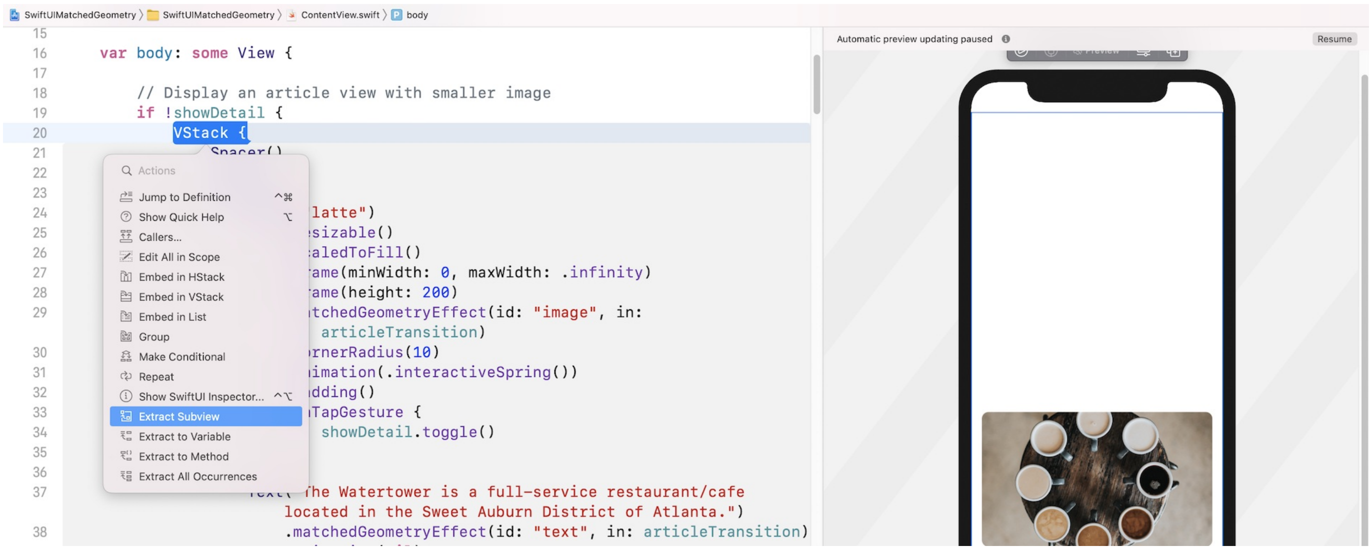
The Watertower will be owned and operated by The Watertower LLC, a Georgia limited liability corporation managed by David N. Patton IV, a resident of the Empowerment Zone. The members of the LLC are David N.

Figure 9. A basic hero animation

Passing @Namespace between Views

Referring to the previous example, we can better organize the code by breaking the two different stack views into subviews. But the problem is how we can pass the `@Namespace` variable between views. Let's see how it can be done.

First, hold the command key and click on the `VStack` keyword of the first stack view. Choose *Extract Subview* from the context menu and name the subview `ArticleExcerptView`.



```
    var body: some View {

        // Display an article view with smaller image
        if !showDetail {
            VStack {
                Spacer()



                    latte")
                    sizable()
                    aledToFill()
                    ame(minWidth: 0, maxWidth: .infinity)
                    ame(height: 200)
                    tchedGeometryEffect(id: "image", in:
                     articleTransition)
                    rnerRadius(10)
                    imation(.interactiveSpring())
                    dding()
                    TapGesture {
                     showDetail.toggle()



                    he Watertower is a full-service restaurant/cafe
                 located in the Sweet Auburn District of Atlanta.")
                 .matchedGeometryEffect(id: "text", in: articleTransition)
```

Figure 10. Extracting the stack view into a subview

You should see quite a number of errors in the `ArticleExcerptView` struct, complaining about the missing of the namespace and the `showDetail` variable. To fix the error of the `showDetail` variable, you can declare a binding in `ArticleExcerptView` like this:

```
@Binding var showDetail: Bool
```

To accept a namespace from another view, the trick is to declare a variable with the type `Namespace.ID` like this:

```
var articleTransition: Namespace.ID
```

This should now fix all the errors in `ArticleExcerptView`. Now go back to `ContentView` and replace `ArticleExcerptView()` with:

```
ArticleExcerptView(showDetail: $showDetail, articleTransition: articleTransition)
```

We pass the binding to `showDetail` and the namespace variable to the subview. This is how you share a namespace across different views. Repeat the same procedure to extract the `ScrollView` into another subview. Name the subview `ArticleDetailView`.

Again, you need to declare the following variable and binding in `ArticleDetailView` to resolve all the errors:

```
@Binding var showDetail: Bool

var articleTransition: Namespace.ID
```

You should also update the instantiation of `ArticleDetailView()` like this:

```
ArticleDetailView(showDetail: $showDetail, articleTransition: articleTransition)
```

After all these changes, the `ContentView` struct is now simplified like this:

```
struct ContentView: View {

    @State private var showDetail = false

    @Namespace private var articleTransition

    var body: some View {

        // Display an article view with smaller image
        if !showDetail {
            ArticleExcerptView(showDetail: $showDetail, articleTransition: article
Transition)
        }

        // Display the article view in a full screen
        if showDetail {
            ArticleDetailView(showDetail: $showDetail, articleTransition: articleT
ransition)
        }

    }
}
```

Everything works the same but the code is now more readable and organized.

Summary

The introduction of the `matchedGeometryEffect` modifier takes the implementation of view animation to the next level. You can create some nice view transitions with much less code. Even if you are a beginner to SwiftUI, you can take advantage of this new modifier to make your app more awesome.

For reference, you can download the complete matched geometry project, with the solutions to the exercises, here:

- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUIMatchedGeometry.zip)

Chapter 34 ScrollViewReader and Grid Animation

Earlier, I introduced you to the new `matchedGeometryEffect` modifier and showed you how to create some basic view animations. In this chapter, let's see how to use the modifier and animate item selection in a grid view. Additionally, you will learn another brand new UI component called `ScrollViewReader`.

The Demo App

Before we step into the implementation, let me show you the final deliverable. This should give you an idea of what you are going to build. When developing real world apps, you may need to display a grid of photo items and let users select some of the items.

One way of presenting the item selection is to have a dock at the bottom of the screen. When an item is selected, it is removed from the grid and inserted into the dock. As you select more items, the dock will hold more items. You can swipe horizontally to navigate through the items in the dock. If you tap an item in the dock, that item will be removed and inserted back into the grid. Figure 1 illustrates how the insertion and removal of an item works.

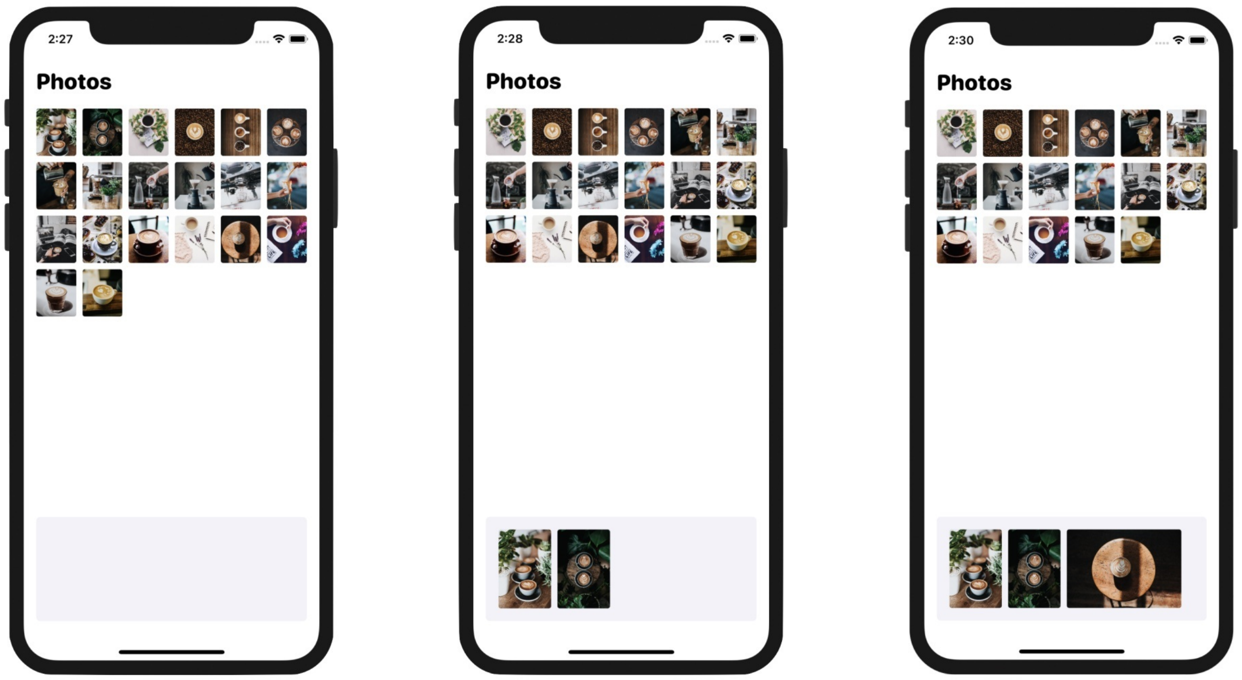


Figure 1. The demo app

We will implement the grid view and the item selection. We will use the `matchedGeometryEffect` modifier to animate the selection. To get started, please first download the starter project at https://www.appcoda.com/resources/swiftui2/SwiftUIGridViewAnimationStarter.zip. This project includes sample data and images.

Building the Photo Grid

First, let's create the photo grid. In the `ContentView` struct, declare a state variable like this:

```
@State private var photoSet = samplePhotos
```

The `samplePhotos` constant is predefined in the starter project and stores the array of photos. The reason why `photoSet` is declared as a state variable is that we will change its content for photo selection.

To present the photos in a grid, we use the built-in `LazyVGrid` component. Insert the following code in `body`:

```
VStack {
    ScrollView {

        HStack {
            Text("Photos")
                .font(.system(.title, design: .rounded))
                .fontWeight(.heavy)

            Spacer()
        }

        LazyVGrid(columns: [ GridItem(.adaptive(minimum: 50)) ]) {

            ForEach(photoSet) { photo in

                Image(photo.name)
                    .resizable()
                    .scaledToFill()
                    .frame(minWidth: 0, maxWidth: .infinity)
                    .frame(height: 60)
                    .cornerRadius(3.0)
            }
        }
    }
}
.padding()
```

Assuming you have read the earlier chapter about grid view, the code is self explanatory. We simply use the adaptive layout to arrange the set of photos in a grid.

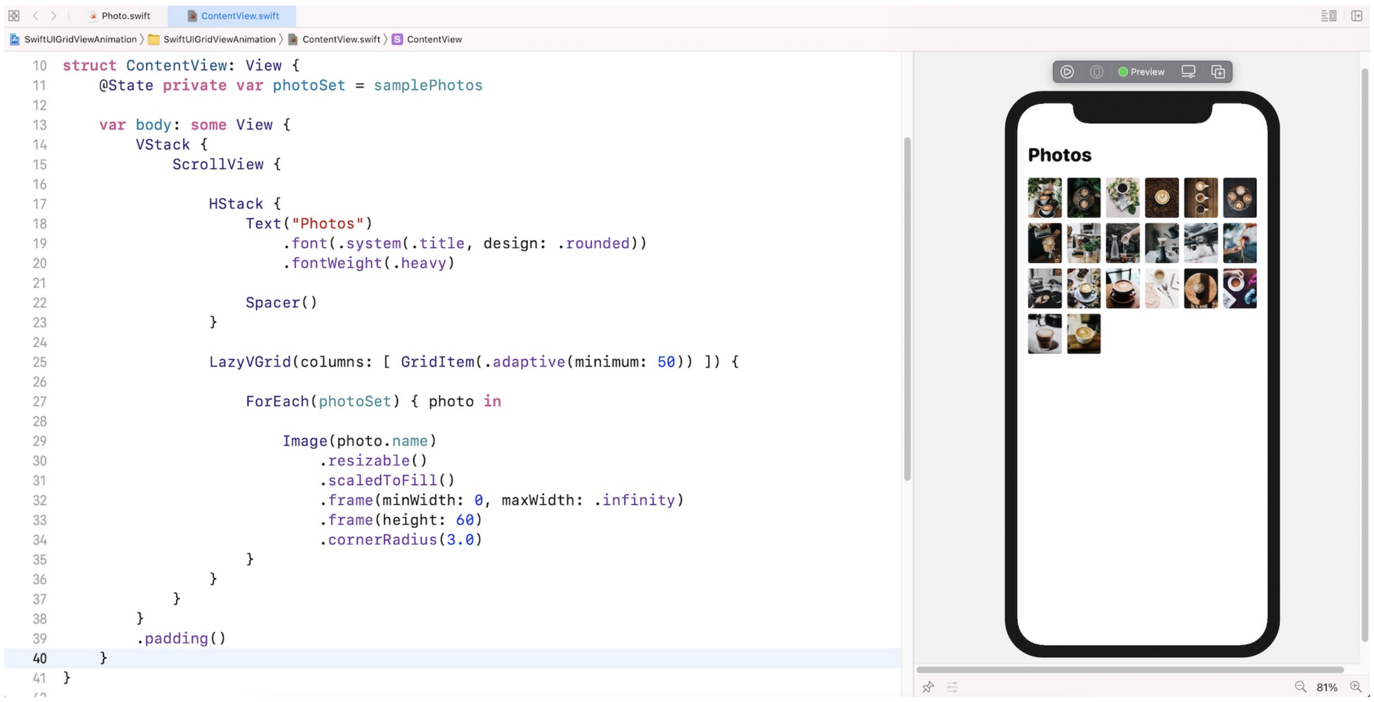


Photo.swift
ContentView.swift
SwiftUIGridViewAnimation 〉 SwiftUIGridViewAnimation 〉 ContentView.swift 〉 ContentView
Preview
Photos
81%

```
struct ContentView: View {
    @State private var photoSet = samplePhotos

    var body: some View {
        VStack {
            ScrollView {

                HStack {
                    Text("Photos")
                        .font(.system(.title, design: .rounded))
                        .fontWeight(.heavy)

                    Spacer()
                }

                LazyVGrid(columns: [ GridItem(.adaptive(minimum: 50)) ]) {

                    ForEach(photoSet) { photo in

                        Image(photo.name)
                            .resizable()
                            .scaledToFill()
                            .frame(minWidth: 0, maxWidth: .infinity)
                            .frame(height: 60)
                            .cornerRadius(3.0)
                    }
                }
            }
        }
        .padding()
    }
}
```

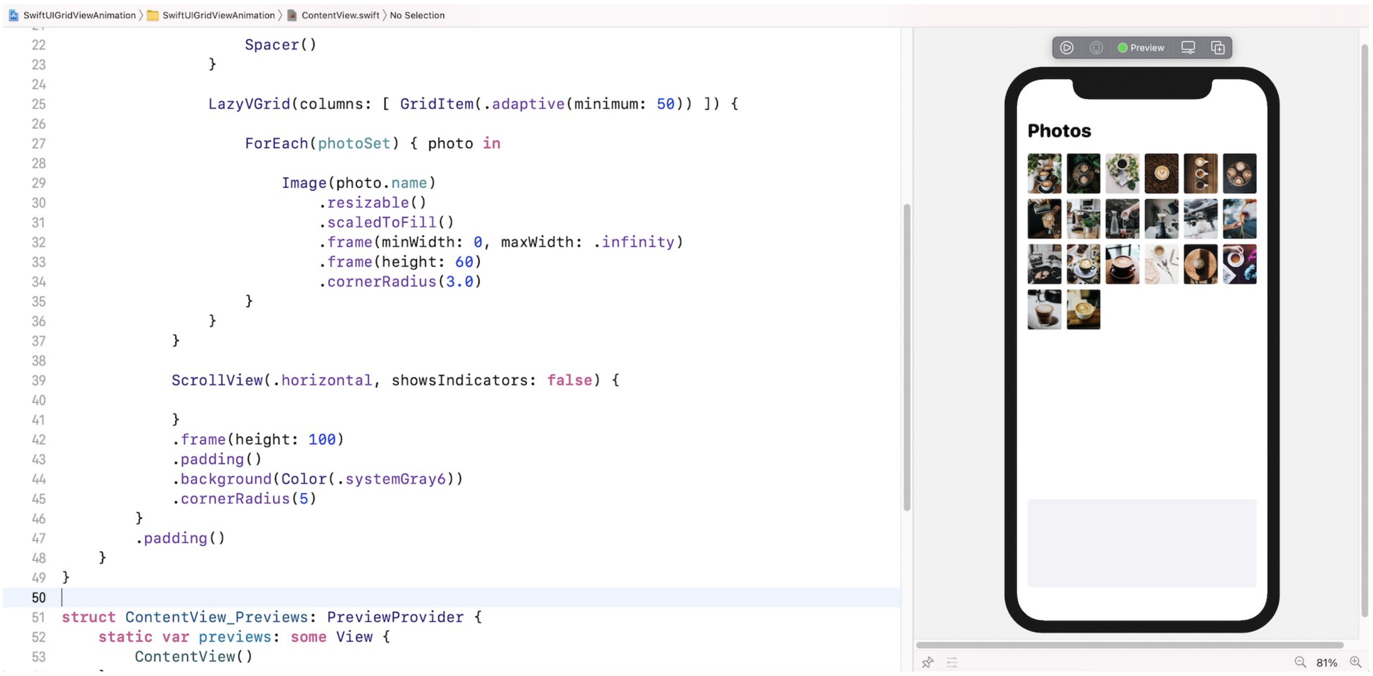
Figure 2. Presenting the photo set in a grid

Adding the Dock

For photo selection, we will create a dock to hold the selected photos. Insert the following code inside the `VStack`:

```
ScrollView(.horizontal, showsIndicators: false) {

}
.frame(height: 100)
.padding()
.background(Color(.systemGray6))
.cornerRadius(5)
```

This creates a scrollable rectangle area for holding the selected photos. Right now, it's just blank.


SwiftUIGridViewAnimation 〉 SwiftUIGridViewAnimation 〉 ContentView.swift 〉 No Selection
Preview
Photos
81%

```
                    Spacer()
                }

                LazyVGrid(columns: [ GridItem(.adaptive(minimum: 50)) ]) {

                    ForEach(photoSet) { photo in

                        Image(photo.name)
                            .resizable()
                            .scaledToFill()
                            .frame(minWidth: 0, maxWidth: .infinity)
                            .frame(height: 60)
                            .cornerRadius(3.0)
                    }
                }
            }

            ScrollView(.horizontal, showsIndicators: false) {

            }
            .frame(height: 100)
            .padding()
            .background(Color(.systemGray6))
            .cornerRadius(5)
        }
        .padding()
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
```

Figure 3. Adding a gray area

Handling Photo Selection

When a photo is selected, we will remove it from the photo grid and insert it into the dock. To handle photo selection, we will create a state variable to hold the selected photos. Insert the following code in `ContentView` to declare the variable:

```
@State private var selectedPhotos: [Photo] = []
```

Each photo in the `photoSet` has its own ID of the type `UUID`. To store the current selected photo, declare another state variable of the type `UUID`:

```
@State private var selectedPhotoId: UUID?
```

To handle the photo selection, attach a `onTapGesture` function to the `Image` component of `LazyVGrid` like this:

```
Image(photo.name)
    .resizable()
    .scaledToFill()
    .frame(minWidth: 0, maxWidth: .infinity)
    .frame(height: 60)
    .cornerRadius(3.0)
    .onTapGesture {
        selectedPhotos.append(photo)
        selectedPhotoId = photo.id
        if let index = photoSet.firstIndex(where: { $0.id == photo.id }) {
            photoSet.remove(at: index)
        }
    }
```

In the block `onTapGesture`, we add the selected photo to the `selectedPhotos` array and update the `selectedPhotoId`. Additionally, we remove the selected photo from `photoSet`. Since `photoSet` is a state variable, the selected photo will be removed from the grid once it's removed from the array.

The selected photo should be added to the dock. So, update the empty `ScrollView` of the dock like this:

```
ScrollView(.horizontal, showsIndicators: false) {
    LazyHGrid(rows: [ GridItem() ]) {
        ForEach(selectedPhotos) { photo in
            Image(photo.name)
                .resizable()
                .scaledToFill()
                .frame(minWidth: 0, maxWidth: .infinity)
                .frame(height: 100)
                .cornerRadius(3.0)
                .onTapGesture {
                    photoSet.append(photo)
                    if let index = selectedPhotos.firstIndex(where: { $0.id == pho
to.id }) {
                        selectedPhotos.remove(at: index)
                    }
                }
        }
    }
}
```

We create a horizontal grid to present the selected photos. For each photo, we attach the `onTapGesture` function to it. When someone taps a photo in the dock, it will be added back to the photo grid and removed from `selectedPhotos`. In other words, the photo will be deleted from the dock.

If you run the app in the preview canvas, you should be able to select any of the photos in the grid. When you tap a photo, it will be automatically added to the dock and that photo will be removed from the grid. Conversely, you can tap a photo in the dock to move it back to the photo grid.

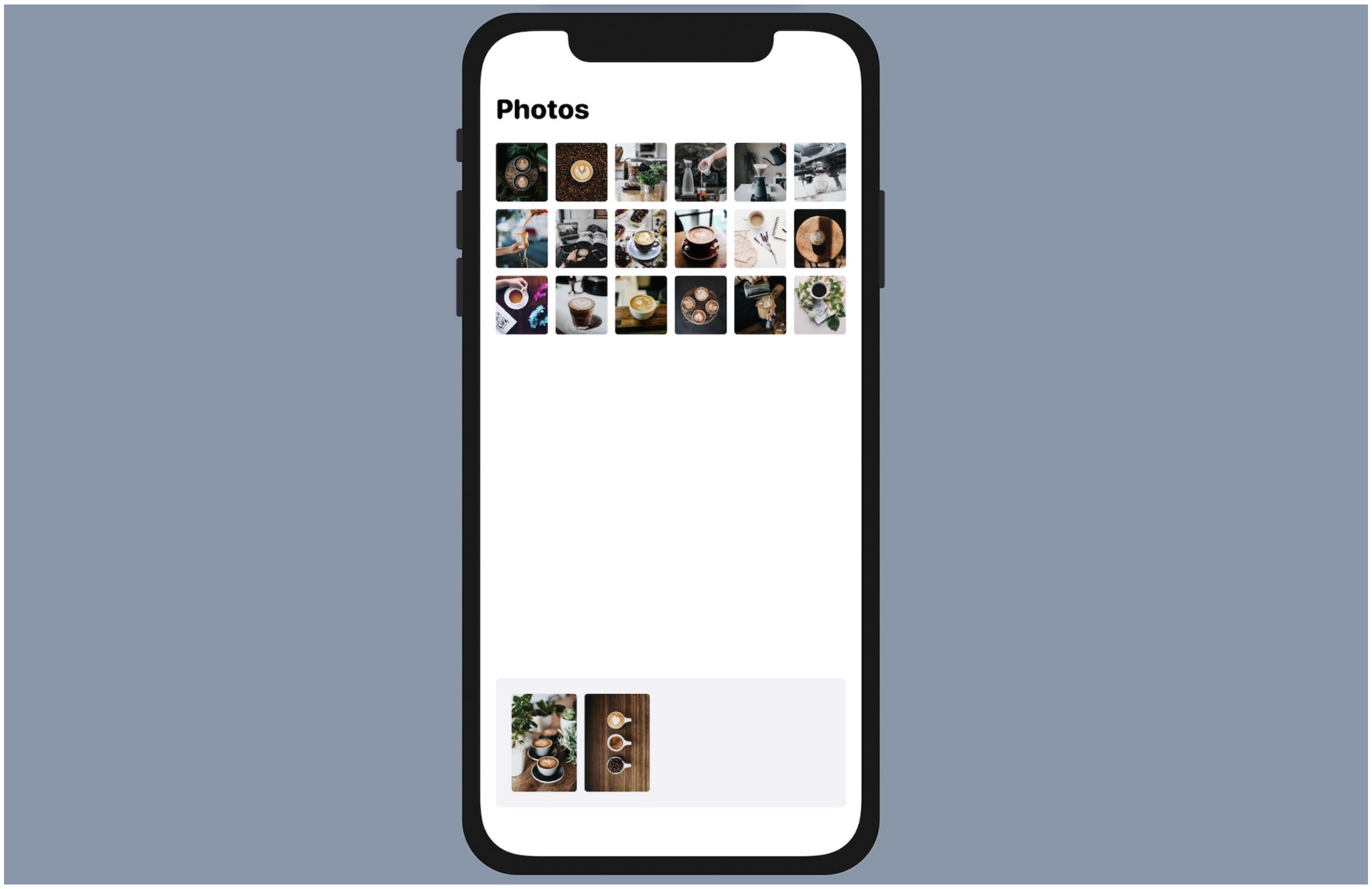


Figure 4. The selected photos are added to the dock

Using MatchedGeometryEffect to Animate the Transition

The photo selection works pretty well but we can make it even better by animating the transition of the photo selection. Currently, the selected photo immediately appears in the dock. What I want to do is to animate the transition of the photo selection. Once selected, the photo should look like it flies from the photo grid to the dock.

With the `matchedGeometryEffect` modifier, it is very easy to implement this type of animation. First, declare the namespace variable for this transition in `ContentView`:

```
@Namespace private var photoTransition
```

Next, attach the `.matchedGeometryEffect` modifer to both `Image` objects:

```
.matchedGeometryEffect(id: photo.id, in: photoTransition)
```

The trick here is to assign each image a distinct ID, so that the app will only animate the change of the selected photo.

To enable the animation, attach the `.animation` modifier to the `VStack` and insert the following line of code under `.padding()` :

```
.animation(.interactiveSpring())
```

This is the code you need to create the animated transtion. Run the app on a simulator or in the preview canvas. When you tap a photo in the grid, you can see a beautiful transition before it's added to the dock.

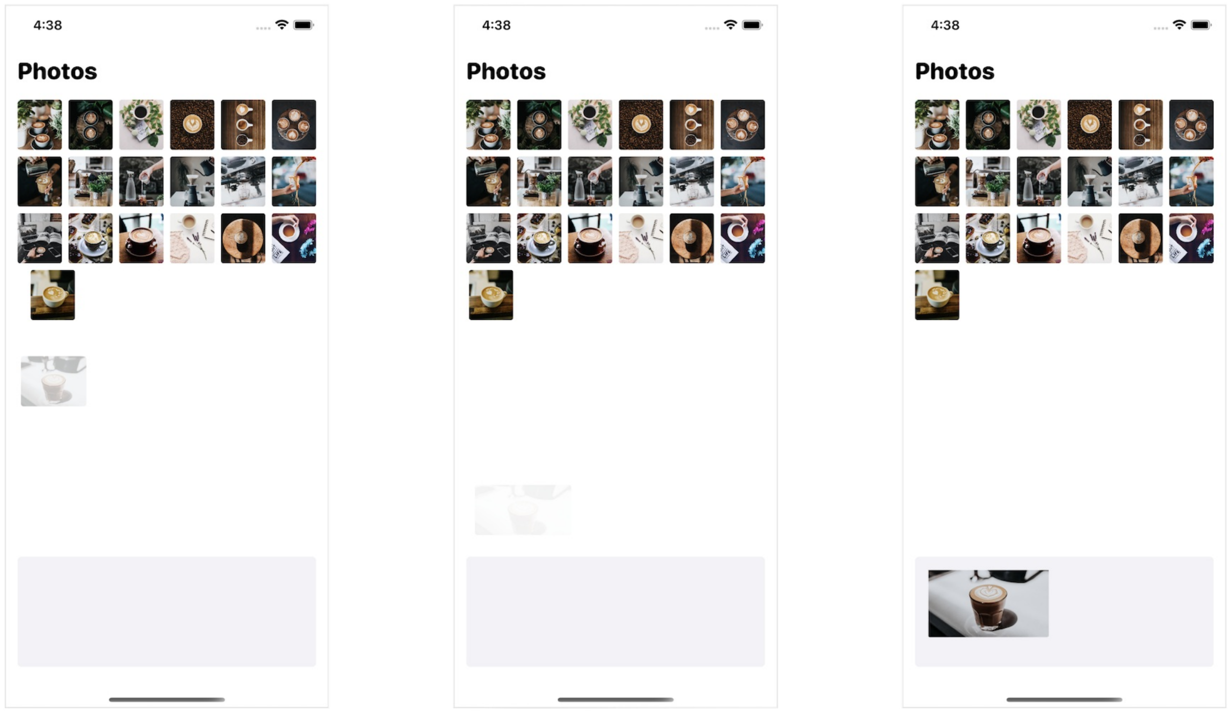


Figure 5. The selected photos are added to the dock with animation

Using ScrollViewReader to Move a Scroll View

The animated transition works great. But did you notice a bug in the app? The dock doesn't scroll automatically to display the most recent selected photo. If you select more than 4 photos, you will need to manually scroll the dock to reveal other selected photos.

How can we fix this bug? In iOS 14, Apple introduced a new component called `ScrollViewReader`. As its name suggests, this reader is designed to work with `ScrollView`. It allows developers to programmatically move a scroll view to a specific location. To use `ScrollViewReader`, you wrap it around a `ScrollView`. Each of the child views should be given their own identifier. You can then call the `scrollTo` function of the `ScrollViewProxy` with the specific ID to move the scroll view to that particular location.

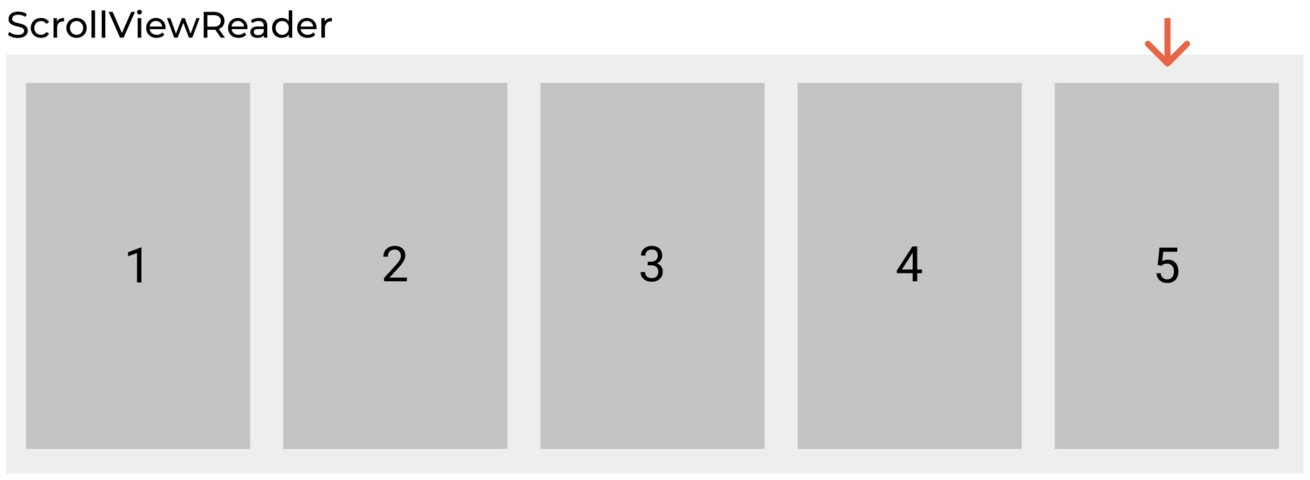


Figure 6. Understanding ScrollViewReader

Now let's get back to our demo app. To programmatically scroll the `ScrollView` of the dock, we need to first give each photo an identifier. The `scrollTo` function requires us to provide an identifier of the view to scroll to. Since each photo already has its unique identifier, we can use the photo ID as the view's identifier.

To set the identifier of the `Image` views in the dock, attach the `.id` modifier to it:

```
.id(photo.id)
```

Once we assign each `Image` view an identifier, attach the `.onChange` function to the `ScrollView` of the dock like this:

```
.onChange(of: selectedPhotoId, perform: { id in
    guard id != nil else { return }

    scrollProxy.scrollTo(id)
})
```

We use `.onChange` to listen for the update of the `selectedPhotoId`. Whenever the selected photo ID is changed, we call `scrollTo` with that photo ID to scroll the scroll view to that particular location. This ensures the dock always shows the most recent selected photo. You can run the app again to try it out.

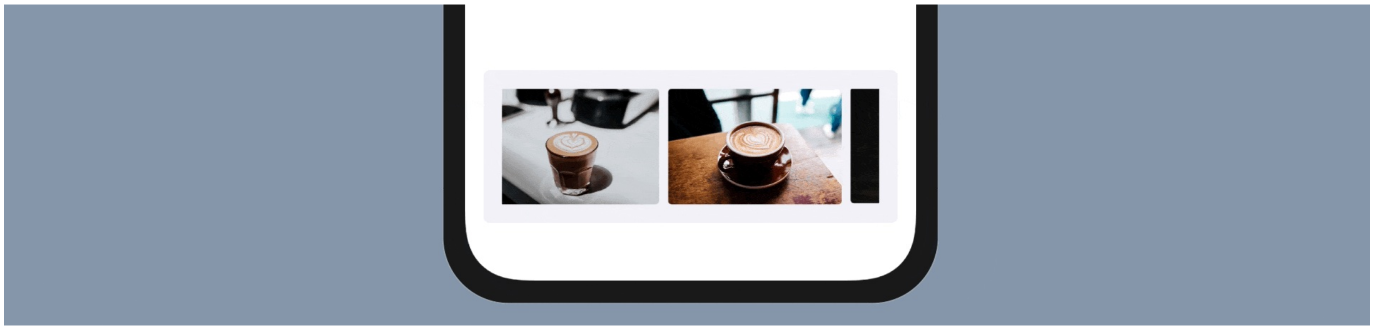


Figure 7. Scrolling the dock automatically

Summary

In this chapter, we continue to explore the usage of `matchedGeometryEffect` and use this modifier to create an amazing view transition. The modifier opens up a lot of opportunities for developers to improve the user experience of their iOS apps. We also experimented with the new `ScrollViewReader` to see how to use it to scroll a scroll view programatically.

For reference, you can download the complete project here:

- Demo project

(https://www.appcoda.com/resources/swiftui2/SwiftUIGridViewAnimation.zip)

Chapter 35
Working with Tab View and Tab Bar Customization

The tab bar interface appears in some of the most popular mobile apps such as Facebook, Instagram, and Twitter. A tab bar appears at the bottom of an app screen and let users quickly switch between different functions of an app. In UIKit, you use the `UITabBarController` to create the tab bar interface. The SwiftUI framework provides a UI component called `TabView` for developers to display tabs in the app.

In this chapter, we will show you how to create a tab bar interface using `TabView`, handle the tab selection, and customize the appearance of the tab bar.

Using TabView to Create the Tab Bar Interface

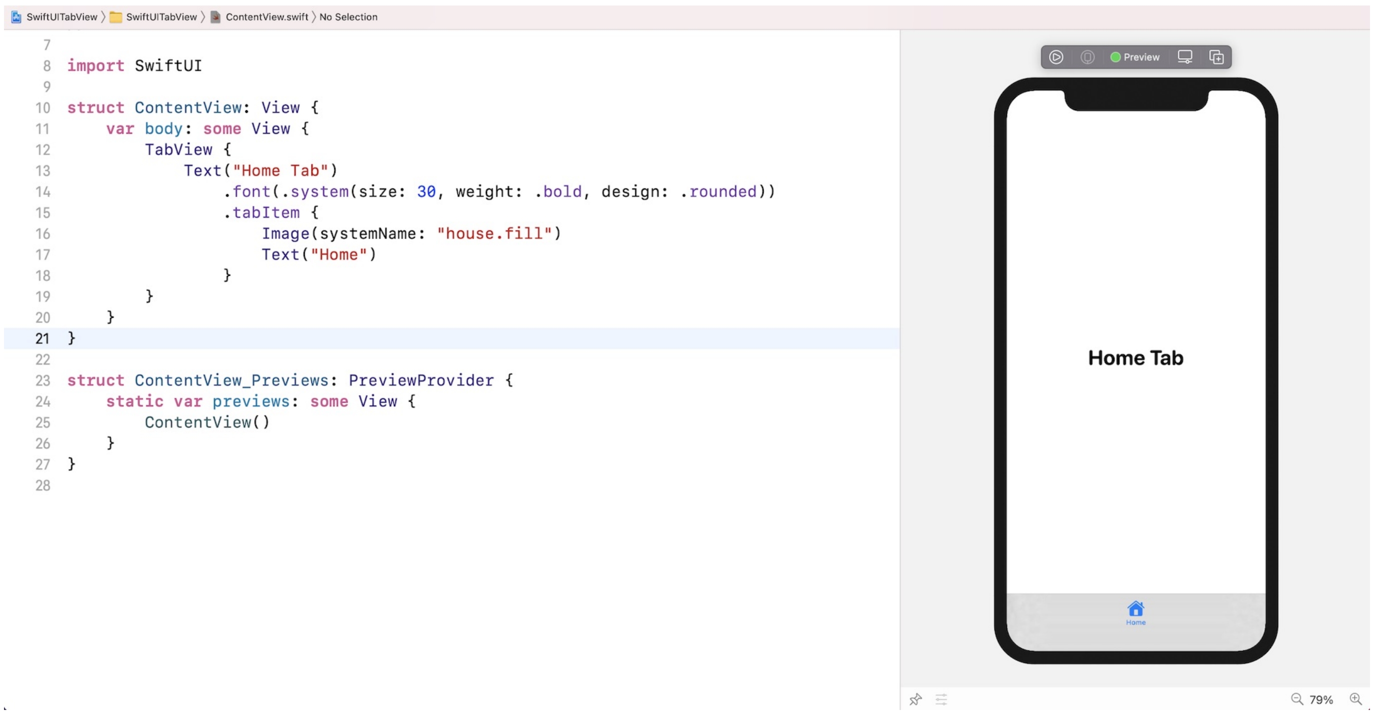
Assuming you've created a SwiftUI project using Xcode 12, let's start with a simple text view like this:

```
struct ContentView: View {
    var body: some View {
        Text("Home Tab")
            .font(.system(size: 30, weight: .bold, design: .rounded))
    }
}
```

To embed this text view in a tab bar, all you need to do is wrap it with the `TabView` component and set the tab item description by attaching the `.tabItem` modifier like this:

```
struct ContentView: View {
    var body: some View {
        TabView {
            Text("Home Tab")
                .font(.system(size: 30, weight: .bold, design: .rounded))
                .tabItem {
                    Image(systemName: "house.fill")
                    Text("Home")
                }
        }
    }
}
```

This will create a tab bar with a single tab item. In the sample code, the tab item has both image and text, but you are free to remove either one of the those.


SwiftUITabView 〉 SwiftUITabView 〉 ContentView.swift 〉 No Selection
Preview
Home Tab
Home
79%

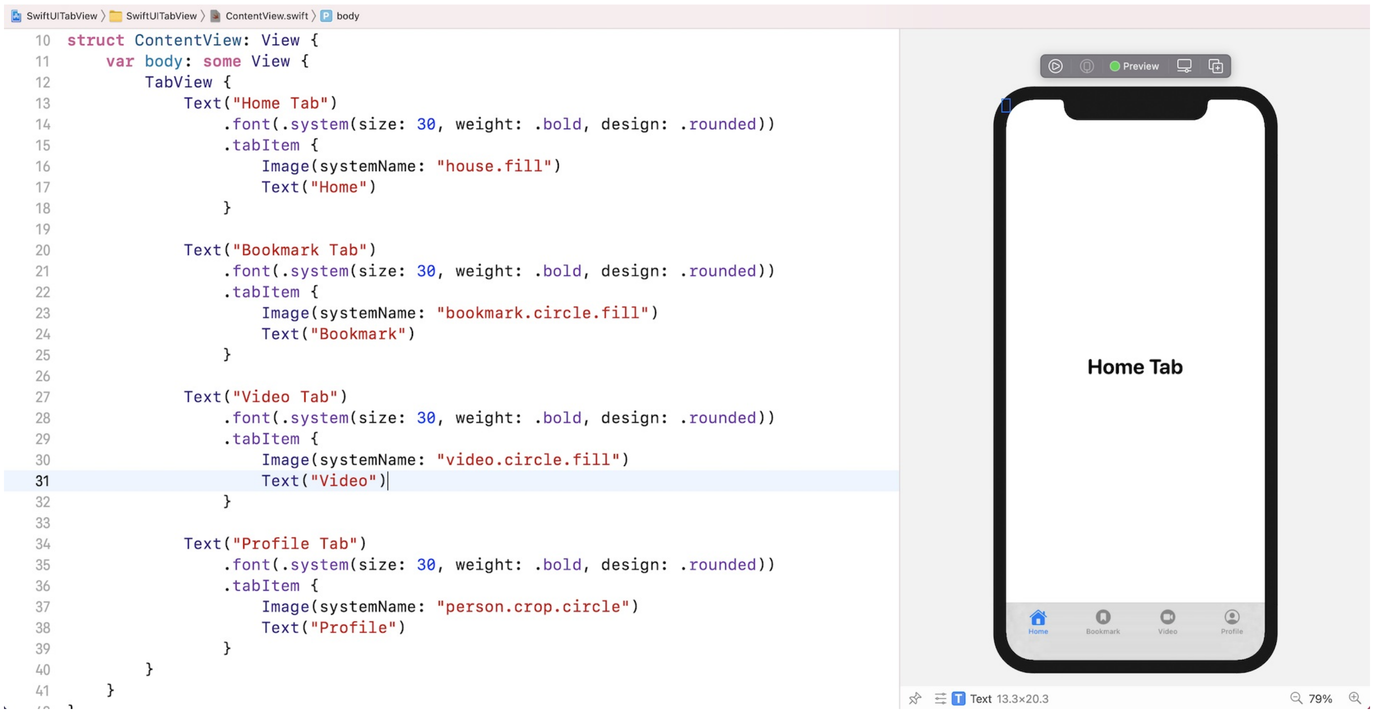
```

import SwiftUI

struct ContentView: View {
    var body: some View {
        TabView {
            Text("Home Tab")
                .font(.system(size: 30, weight: .bold, design: .rounded))
                .tabItem {
                    Image(systemName: "house.fill")
                    Text("Home")
                }
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}

```

Figure 1. Tab bar with a single tab item

To display more tabs, you just need to add child views inside the `TabView` like this:

```
TabView {
    Text("Home Tab")
        .font(.system(size: 30, weight: .bold, design: .rounded))
        .tabItem {
            Image(systemName: "house.fill")
            Text("Home")
        }

    Text("Bookmark Tab")
        .font(.system(size: 30, weight: .bold, design: .rounded))
        .tabItem {
            Image(systemName: "bookmark.circle.fill")
            Text("Bookmark")
        }

    Text("Video Tab")
        .font(.system(size: 30, weight: .bold, design: .rounded))
        .tabItem {
            Image(systemName: "video.circle.fill")
            Text("Video")
        }

    Text("Profile Tab")
        .font(.system(size: 30, weight: .bold, design: .rounded))
        .tabItem {
            Image(systemName: "person.crop.circle")
            Text("Profile")
        }
}
```

This gives you a tab bar interface with 4 tab items.



```
struct ContentView: View {
    var body: some View {
        TabView {
            Text("Home Tab")
                .font(.system(size: 30, weight: .bold, design: .rounded))
                .tabItem {
                    Image(systemName: "house.fill")
                    Text("Home")
                }

            Text("Bookmark Tab")
                .font(.system(size: 30, weight: .bold, design: .rounded))
                .tabItem {
                    Image(systemName: "bookmark.circle.fill")
                    Text("Bookmark")
                }

            Text("Video Tab")
                .font(.system(size: 30, weight: .bold, design: .rounded))
                .tabItem {
                    Image(systemName: "video.circle.fill")
                    Text("Video")
                }

            Text("Profile Tab")
                .font(.system(size: 30, weight: .bold, design: .rounded))
                .tabItem {
                    Image(systemName: "person.crop.circle")
                    Text("Profile")
                }
        }
    }
```

Figure 2. 4 Adding tab items in the tab bar

Customizing the Tab Bar Color

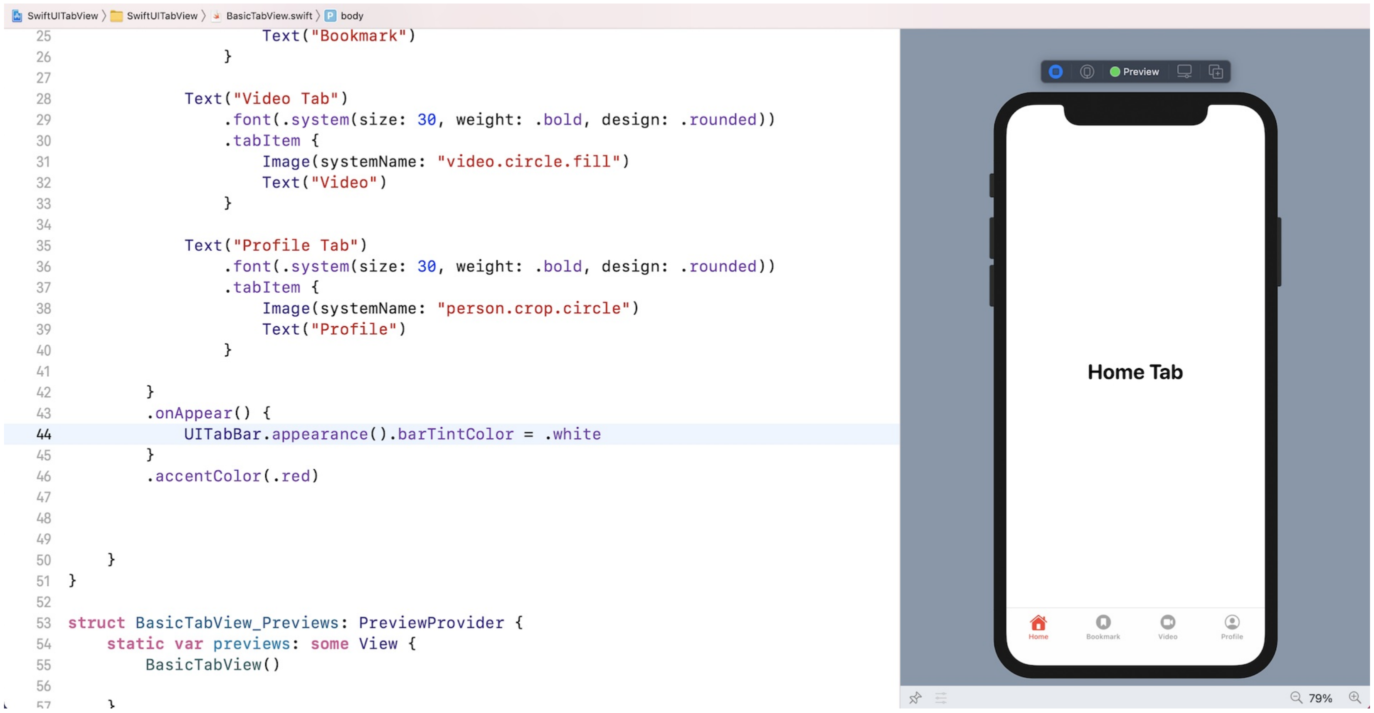
By default, the color of the tab bar item is set to blue. You can change its color by attaching the `.accentColor` modifier to `TabView` like this:

```
TabView {

}
.accentColor(.red)
```

Yet the SwiftUI framework doesn't have a built-in modifier for changing the tab bar's color. If you want to change that, you may use the appearance API of UIKit like below:

```
.onAppear() {
    UITabBar.appearance().barTintColor = .white
}
```

If you attach the call to `TabView`, the color of the tab bar should be changed to white.



```
                    Text("Bookmark")
                }

            Text("Video Tab")
                .font(.system(size: 30, weight: .bold, design: .rounded))
                .tabItem {
                    Image(systemName: "video.circle.fill")
                    Text("Video")
                }

            Text("Profile Tab")
                .font(.system(size: 30, weight: .bold, design: .rounded))
                .tabItem {
                    Image(systemName: "person.crop.circle")
                    Text("Profile")
                }

        }
        .onAppear() {
            UITabBar.appearance().barTintColor = .white
        }
        .accentColor(.red)



    }
}

struct BasicTabView_Previews: PreviewProvider {
    static var previews: some View {
        BasicTabView()

    }
```

Figure 3. Changing the color of the tab bar

Switching Between Tabs Programmatically

Users tap the tab bar items to switch between tabs, which is automatically handled the `TabView`. In some use cases, you may want to switch to a specific tab programmatically. The `TabView` has another `init` method for this purpose. The method requires a state variable which contains the tag value of the tab.

```
TabView(selection: $selection)
```

As an example, declare the following state variable in `ContentView`:

```
@State private var selection = 0
```

Here we initialize the `selection` variable with a value of `0`, which is the tag value of the first tab item. We haven't defined the tag value for the tab items yet. Therefore, update the code like this and attach the `tag` modifier for each of the tab items:

```
TabView(selection: $selection) {
    Text("Home Tab")
        .font(.system(size: 30, weight: .bold, design: .rounded))
        .tabItem {
            Image(systemName: "house.fill")
            Text("Home")
        }
        .tag(0)

    Text("Bookmark Tab")
        .font(.system(size: 30, weight: .bold, design: .rounded))
        .tabItem {
            Image(systemName: "bookmark.circle.fill")
            Text("Bookmark")
        }
        .tag(1)

    Text("Video Tab")
        .font(.system(size: 30, weight: .bold, design: .rounded))
        .tabItem {
            Image(systemName: "video.circle.fill")
            Text("Video")
        }
        .tag(2)

    Text("Profile Tab")
        .font(.system(size: 30, weight: .bold, design: .rounded))
        .tabItem {
            Image(systemName: "person.crop.circle")
            Text("Profile")
        }
        .tag(3)
}
```

We give each tab item a unique index by attaching the `tag` modifier. The `TabView` is also bound to the `selection` value. To switch to your preferred tab programmatically, update the value of the `selection` variable.

You can create a *Next* button that switches to the next tab like this:

```
ZStack(alignment: .topTrailing) {
    TabView(selection: $selection) {
        .
        .
        .
    }
    .accentColor(.red)
    .onAppear() {
        UITabBar.appearance().barTintColor = .white
    }

    Button(action: {
        selection = (selection + 1) % 4
    }) {
        Text("Next")
            .font(.system(.headline, design: .rounded))
            .padding()
            .foregroundColor(.white)
            .background(Color.red)
            .cornerRadius(10.0)
            .padding()

    }
}
```

After making the changes, run the app in the preview canvas, you step through the tabs by tapping the *Next* button.

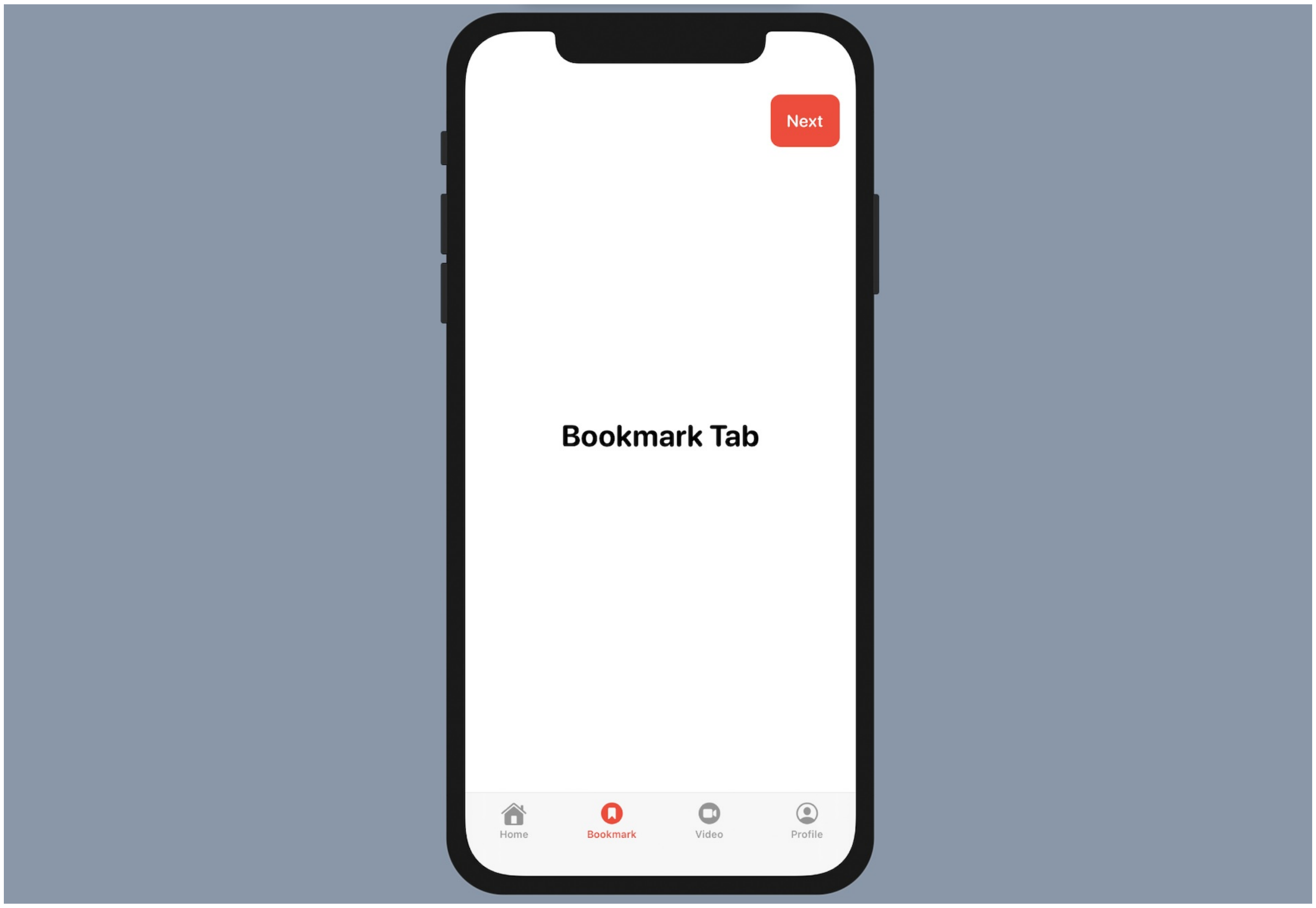


Figure 4. Using the Next button to switch the tab

Hiding the Tab Bar in a Navigation View

You can embed a tab view in a navigation view by wrapping the `TabView` component with `NavigationView` like this:

```
NavigationView {
    TabView(selection: $selection) {
        .
        .
        .
    }

    .navigationTitle("TabView Demo")
}
```

In UIKit, there is another option called `hidesBottomBarWhenPushed`, which allows you to hide the tab bar when the UI is changed to the detail view in a navigation interface. SwiftUI also has this feature built-in. You can modify the code like this to see it in action:

```
NavigationView {
    TabView(selection: $selection) {
        List(1...10, id: \.self) { index in
            NavigationLink(
                destination: Text("Item #\(index) Details"),
                label: {
                    Text("Item #\(index)")
                        .font(.system(size: 20, weight: .bold, design: .rounded))
                })

        }
        .tabItem {
            Image(systemName: "house.fill")
            Text("Home")
        }
        .tag(0)

        Text("Bookmark Tab")
            .font(.system(size: 30, weight: .bold, design: .rounded))
            .tabItem {
                Image(systemName: "bookmark.circle.fill")
                Text("Bookmark")
            }
            .tag(1)

        Text("Video Tab")
            .font(.system(size: 30, weight: .bold, design: .rounded))
            .tabItem {
                Image(systemName: "video.circle.fill")
                Text("Video")
            }
            .tag(2)

        Text("Profile Tab")
            .font(.system(size: 30, weight: .bold, design: .rounded))
            .tabItem {
                Image(systemName: "person.crop.circle")
                Text("Profile")
            }
            .tag(3)
    }
    .accentColor(.red)
    .onAppear() {
        UITabBar.appearance().barTintColor = .white
    }

    .navigationTitle("TabView Demo")
}
```

We just altered the code of the *Home* tab to display a list of items. We wrap each list item with a `NavigationLink`, so that it will navigate to the detail view when the item is tapped. If you run the app using a simulator or in the preview canvas, you should see that the tab bar is hidden when we navigate to the detail view.

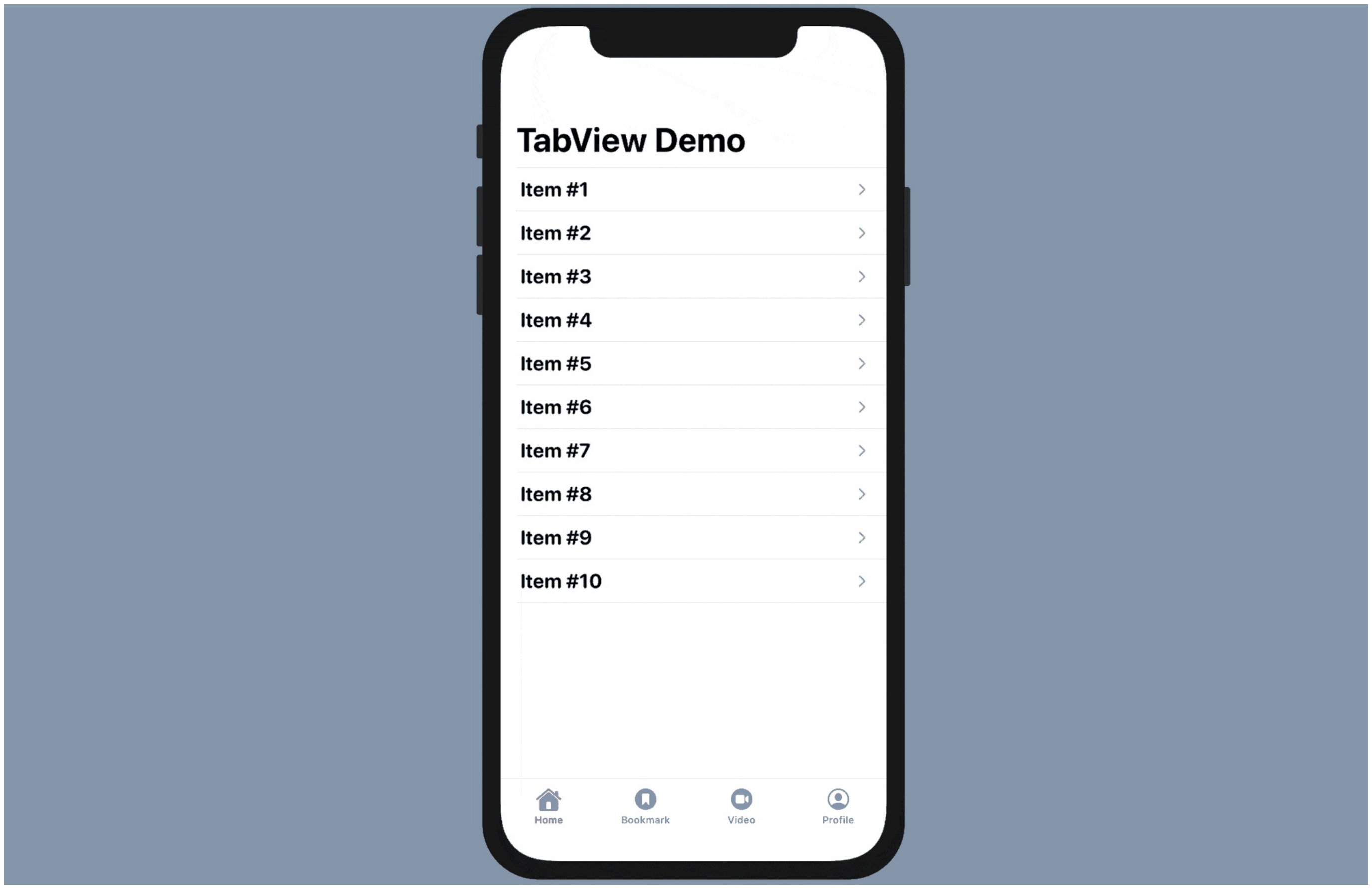


Figure 5. Hiding the tab bar in the detailed view

For some scenarios, you probably don't want the tab bar to be hidden. In these cases, you can create the navigation interface the other way round. Instead of wrapping the tab view in a navigation view, you embed the navigation view in a tab view like this:

```
TabView(selection: $selection) {
    NavigationView {
        List(1...10, id: \.self) { index in
            NavigationLink(
                destination: Text("Item #\(index) Details"),
                label: {
                    Text("Item #\(index)")
                        .font(.system(size: 20, weight: .bold, design: .rounded))
                })

        }

        .navigationTitle("TabView Demo")
    }
    .tabItem {
        Image(systemName: "house.fill")
        Text("Home")
    }
    .tag(0)

    .
    .
    .
}
```

Now when you navigate to the detail view of the item, the tab bar is still there.

Summary

In this chapter, we walked you through the basics of `TabView`, which is the UI component in SwiftUI for building a tab view interface. The framework doesn't provide you with many options for customizing the tab bar. However, you may still rely on the APIs of UIKit to customize its appearance. This chapter only shows you how to work with the built-in tab bar. You can actually create your own tab bar if you need full customization. We will discuss it in the future chapters.

For reference, you can download the complete project here:

- Demo project (https://www.appcoda.com/resources/swiftui2/SwiftUITabView.zip)