



Xcode 11, Swift 5 and iOS 13 Ready

Mastering SwiftUI



SIMON NG

APPCODA

Table of Contents

Preface

Chapter 1 - Introduction to SwiftUI

Chapter 2 - Getting Started with SwiftUI and Working with Text

Chapter 3 - Working with Images

Chapter 4 - Layout User Interfaces with Stacks

Chapter 5 - Understanding ScrollView and Building a Carousel UI

Chapter 6 - Working with SwiftUI Buttons and Gradient

Chapter 7 - Understanding State and Binding

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

Chapter 9 - Basic Animations and Transitions

Chapter 10 - Understanding Dynamic List, ForEach and Identifiable

Chapter 11 - Working with Navigation UI and Navigation Bar Customization

Chapter 12 - Playing with Modal Views, Floating Buttons and Alerts

Chapter 13 - Building a Form with Picker, Toggle and Stepper

Chapter 14 - Data Sharing with Combine and Environment Objects

Chapter 15 - Building a Registration Form with Combine and View Model

Chapter 16 - Working with Swipe-to-Delete, Context Menu and Action Sheets

Chapter 17 - Using Gestures

Chapter 18 - Building an Expandable Bottom Sheet with SwiftUI Gestures and GeometryReader

Chapter 19 - Creating a Tinder-like UI with Gestures and Animations

Chapter 20 - Advanced Animations and Transitions

Chapter 21 - Putting Everything Together to Build a Real World App



Copyright ©2019 by AppCoda Limited All right reserved. No part of this book may be used or reproduced, stored or transmitted in any manner whatsoever without written permission from the publisher. 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. Today, you know Apple has released a brand new framework called *SwiftUI* alongside with Xcode 11. The release of SwiftUI is huge, really huge for existing iOS developers or someone who is going to learn iOS app building. It's 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.

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.

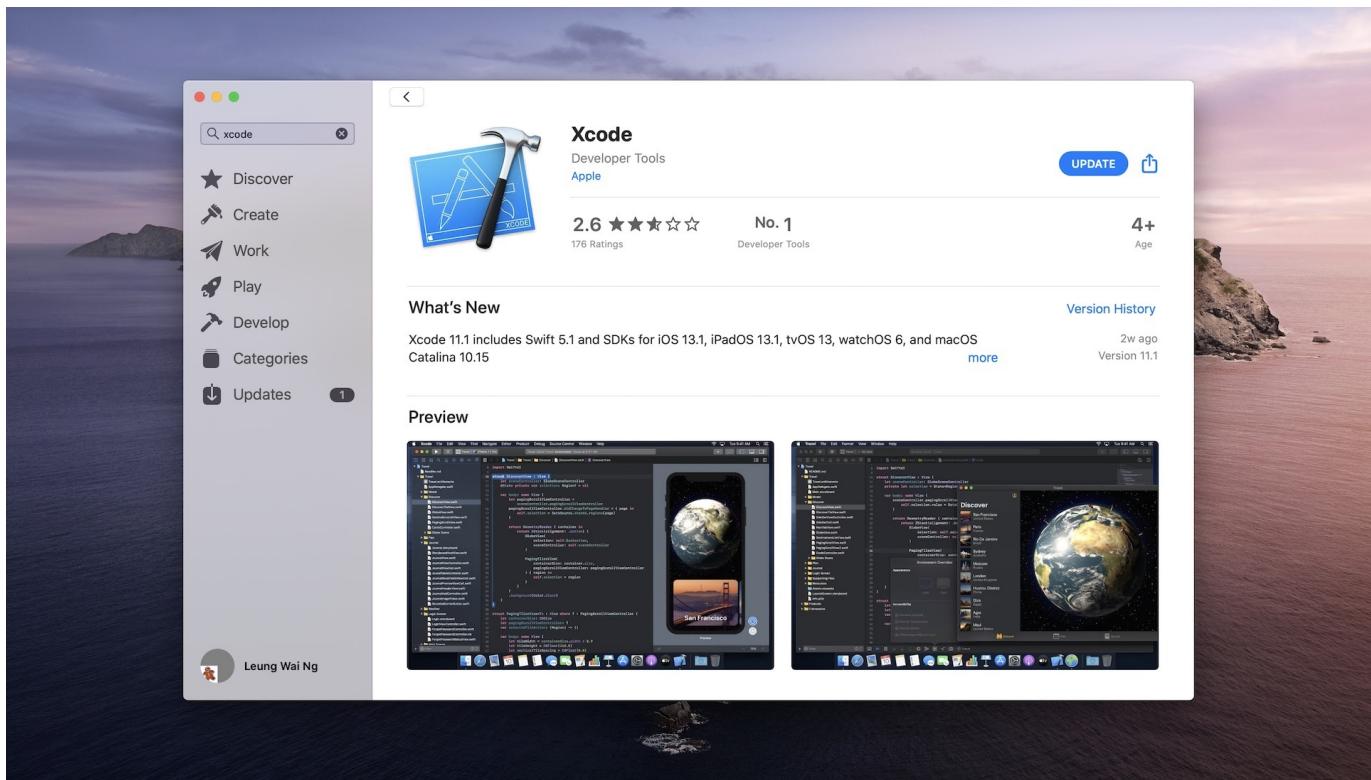
What You Need to Develop Apps with SwiftUI

Having a Mac is the basic requirement for iOS development. To use SwiftUI, you need to have a Mac which is installed with macOS Catalina (v10.15) and Xcode 11.

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 11 and macOS Catalina, you can instantly preview the result of your SwiftUI code.

To install Xcode, go up to the Mac App Store and download it. Simply search "Xcode" and click the "Get" button to download it.

Once you complete the installation process, you will find Xcode in the Launchpad. At the time of this writing, the latest version of Xcode is 11.1. Throughout this book, we will use this version of Xcode to create the demo apps. Even if you have installed Xcode before, I suggest you upgrade to the latest version. This should make it easier for you to follow the materials.



Frequently 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.

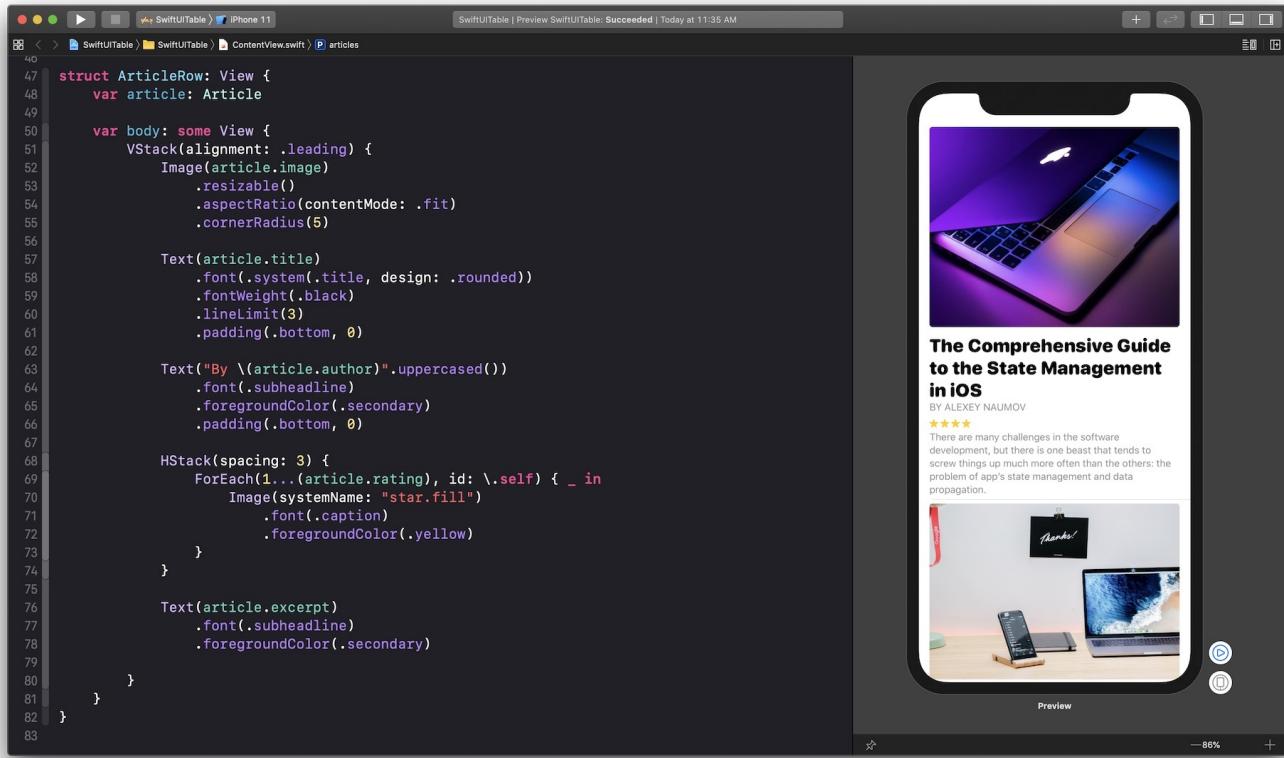


Figure 1. Programming in SwiftUI

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 and Xcode 11, you get immediate feedback of the UI you are coding. For example, you add a new record to a table, Xcode 11 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 away from learning SwiftUI, which is an awesome framework for beginners.

Declarative vs Imperative Programming

Like Java, C++, PHP, and C#, Swift is an imperative programming language since its release. 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 the difference because everything is new to you. However, if you have some experience in Object-oriented programming or developed with UIKit before, this paradigm shift affects how you think about building user interface. You may need to unlearn some old concepts and relearn the new ones.

So, what's the difference between imperative and declarative programming? If you go up 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). Assuming you ask your helper to prepare a 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 the toppings (including onions, sliced mushrooms, pepperoni, cooked sausage, cooked bacon, diced peppers) and cheese
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 how the UI looks like and what you want to do when a state changes.

The declarative way of coding style would make the code more 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.

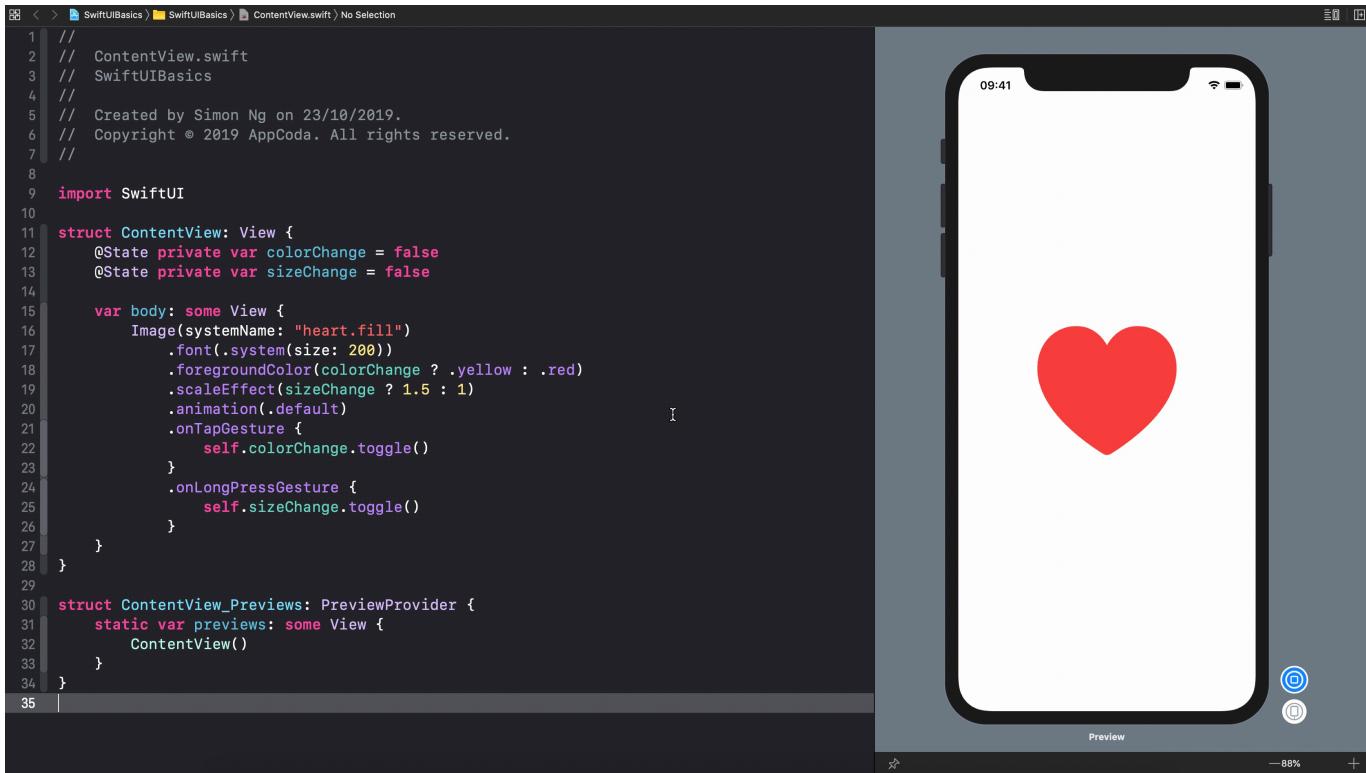


Figure 2. The implementation of an interactive heart button

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 power of this declarative UI framework.

No more Interface Builder and Auto Layout

In Xcode 11, you can choose between SwiftUI and Storyboard to build user interface. If you have built an app before, you may use Interface Builder to layout UI on 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 display it in the canvas.

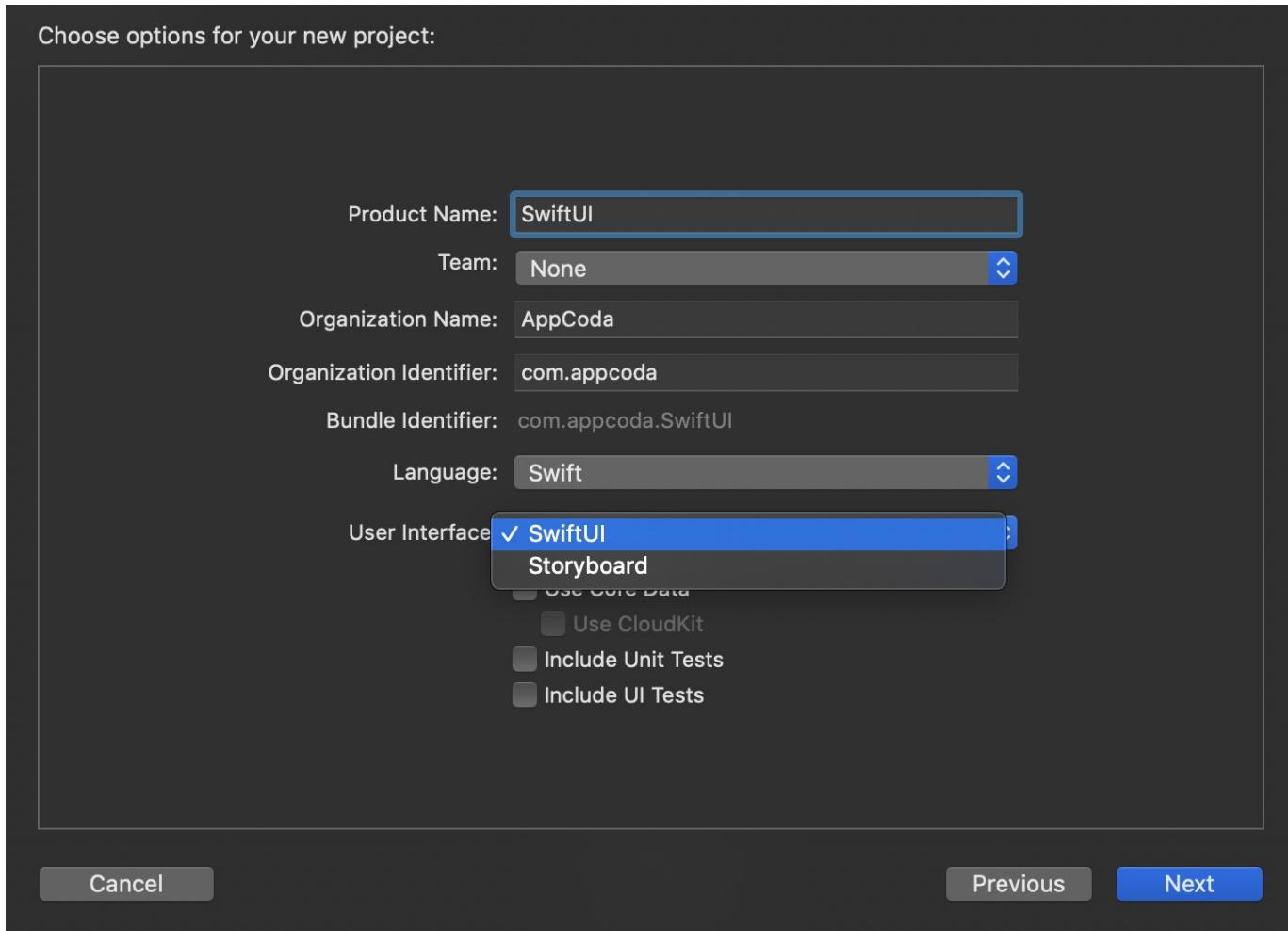


Figure 3. User interface option in Xcode 11

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 a layout constraints and resolve the conflicts. Now you compose the desired UI by using stacks, spacers, and padding. We will discuss this concept in details 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 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 UI for iOS, everything you learn here is applicable to other Apple platforms such as watchOS. Prior to the launch of SwiftUI, you'll have to use platform-specific UI frameworks to develop the user interface. You use AppKit to write UI for macOS apps. To develop tvOS apps, you rely on TVUIKit. And, for watchOS apps, you use WatchKit.

With SwiftUI, Apple offers developers a unified UI framework for building user interface 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 like. 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.

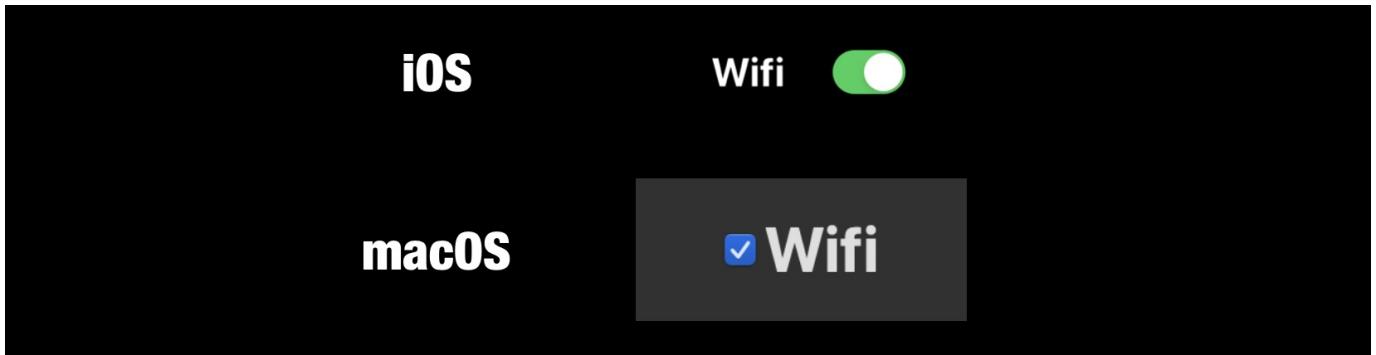


Figure 4. Toggle on macOS and iOS

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, if 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 `MapView` in SwiftUI.

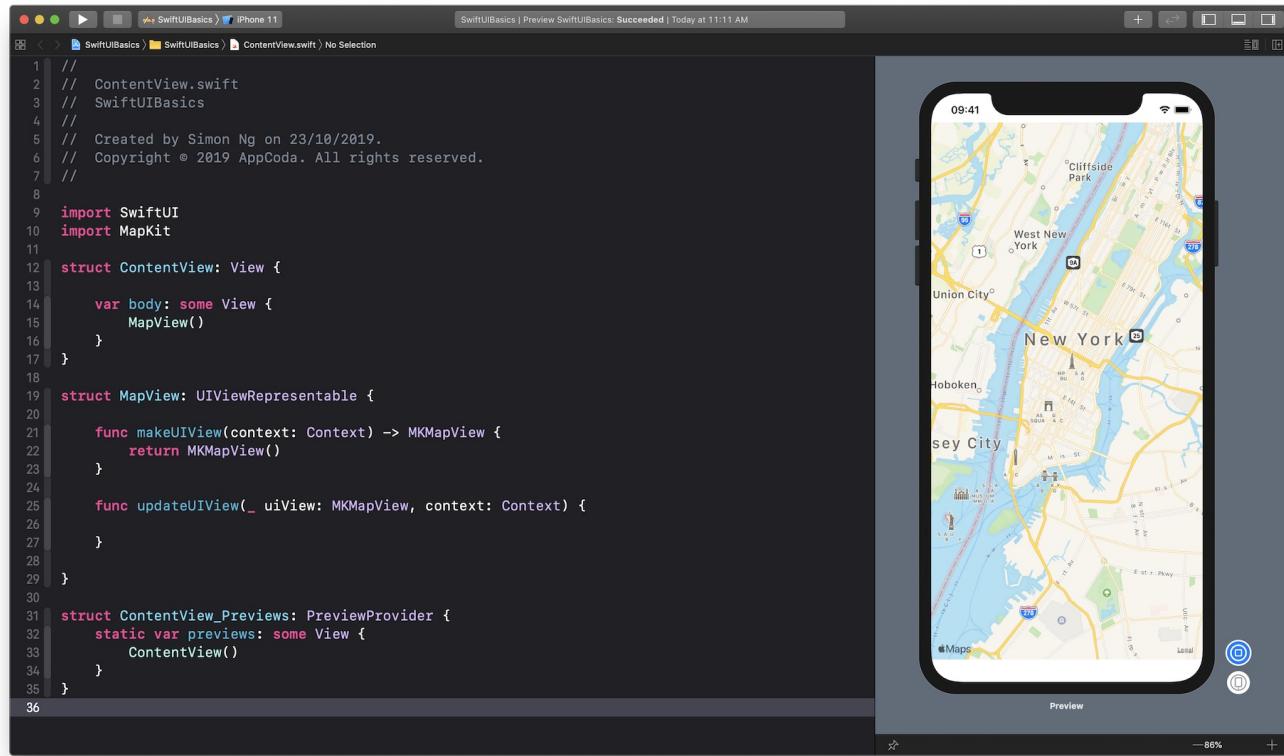


Figure 6. Porting MKMapView to SwiftUI

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 new to most developers, it's a right time to learn and incorporate the framework in your new project. If you have some personal projects or side projects in your company, there is no reason why you shouldn't try out SwiftUI.

Having that said, you need to consider carefully whether you should apply SwiftUI to your commercial projects. One major drawback of SwiftUI is that it only runs on iOS 13, macOS 10.15, tvOS 13, and watchOS 6. It's very likely your app requires to support lower versions of the platform. In this case, you will have to wait at least a year before adopting SwiftUI.

At the time of this writing, SwiftUI has only been officially released for around a month or two. 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 very obvious 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 unarguably very new. It takes 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 be applicable to your production projects, it's recommended to start a side project and explore the framework. Once you try out SwiftUI and master the skills, you will enjoy developing UI 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 and create a new project using the *Single View App* template. 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 to your own value. If you have a website, set it to your domain in reverse domain name notation. To use SwiftUI, you have to explicitly check the *Use SwiftUI* checkbox. Click *Next* and choose a folder to create the project.

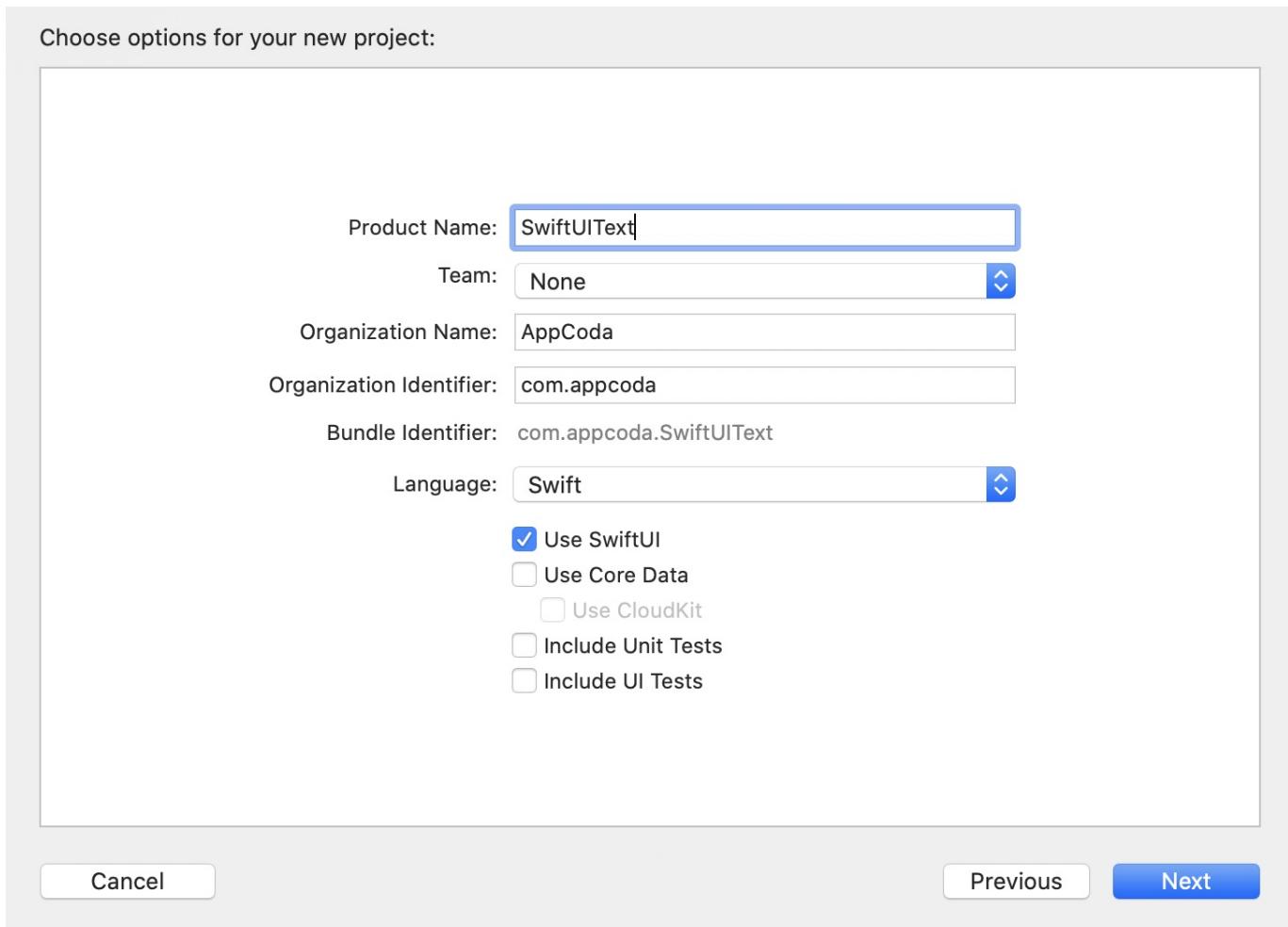


Figure 1. Creating a new project

Once you save the project, Xcode should load the `ContentView.swift` file and display a design/preview canvas. If you can't see the design canvas, you can go up to the Xcode menu and choose *Editor > Canvas* to enable it.

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 hit the button, Xcode renders the preview in a simulator that you choose in the simulator selection (e.g. iPhone XR). Optionally, you can hide the project navigator and the utility panes to free up more space for both the code editor and canvas.

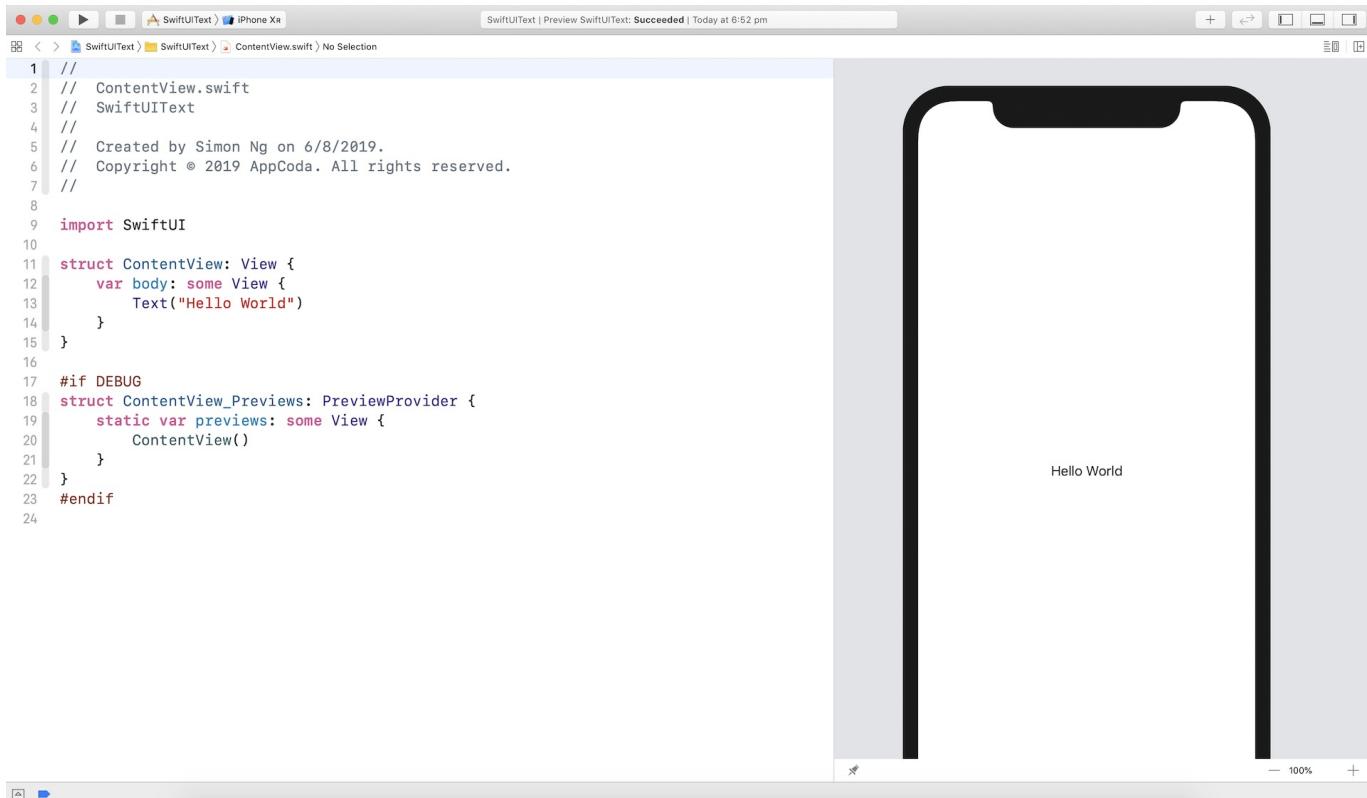


Figure 2. The code editor and the canvas

Displaying a Simple Text

The sample code generated in `ContentView` already shows you how to display a single line of text. You initialize a `Text` and pass it with 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.

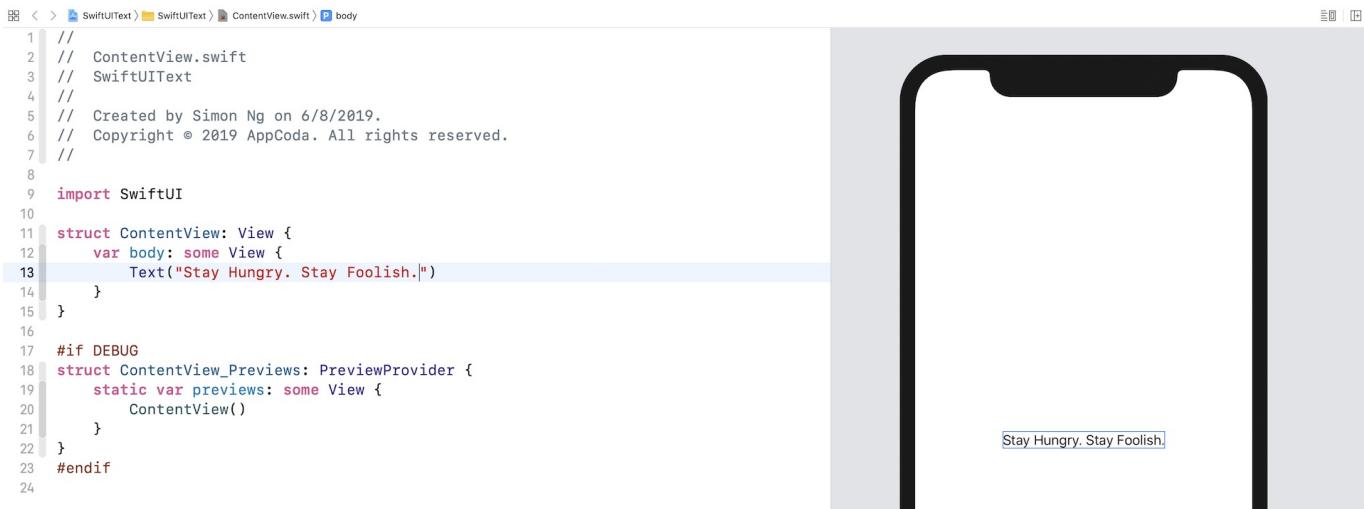


Figure 3. Changing the text

Changing the Font Type and Color

In SwiftUI, you can change the properties (e.g. color) 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 named `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`.



```
9 import SwiftUI
10
11 struct ContentView: View {
12     var body: some View {
13         Text("Stay Hungry. Stay Foolish.").fontWeight(. ) 2 ⓘ Expected identifier after '! expr...
14     }
15 }
16
17 #if DEBUG
18 struct ContentView_Previews: PreviewProvider {
19     static var previews: some View {
20         ContentView()
21     }
22 }
23 #endif
24
```

A screenshot of an Xcode code editor showing a completion dropdown for the `.fontWeight()` modifier. The dropdown lists several font weight options: `Font.Weight? nil`, `Optional<Font.Weight> none`, `Optional<Font.Weight> some(Font.Weight)`, `Font.Weight black`, `Font.Weight bold`, `Font.Weight heavy` (which is highlighted in blue), `Font.Weight light`, and `Font.Weight medium`.

Figure 4. Choosing your preferred font weight

By calling `fontWeight` with the value `.bold`, it actually returns you with a new view that has the bolded text. What's 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 can write the code like this:

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

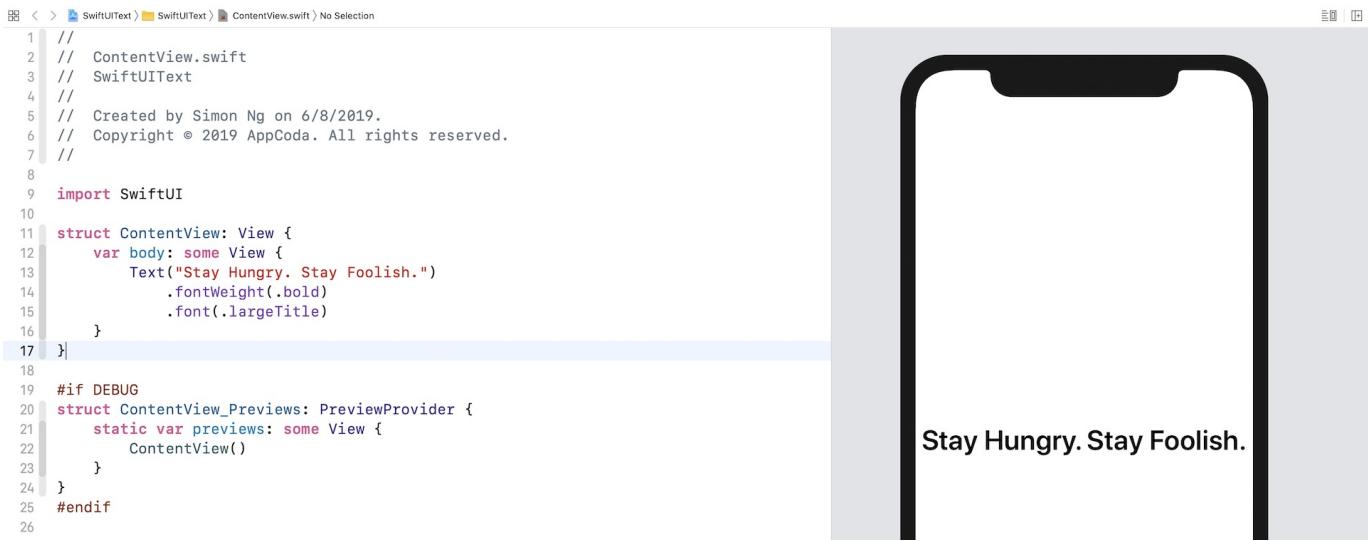
Since we may chain multiple modifiers together, we usually write the code above in the following format:

```
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 to use 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 refer to the documentation (<https://developer.apple.com/documentation/swiftui/font>) to find out all the supported values of the `font` modifier.



The screenshot shows the Xcode interface with the code editor on the left and a preview canvas on the right. The code editor contains the following Swift code:

```
1 // ContentView.swift
2 // ContentView.swift
3 // SwiftUI
4 //
5 // Created by Simon Ng on 6/8/2019.
6 // Copyright © 2019 AppCoda. All rights reserved.
7 //
8
9 import SwiftUI
10
11 struct ContentView: View {
12     var body: some View {
13         Text("Stay Hungry. Stay Foolish.")
14             .fontWeight(.bold)
15             .font(.largeTitle)
16     }
17 }
18
19 #if DEBUG
20 struct ContentView_Previews: PreviewProvider {
21     static var previews: some View {
22         ContentView()
23     }
24 }
25 #endif
```

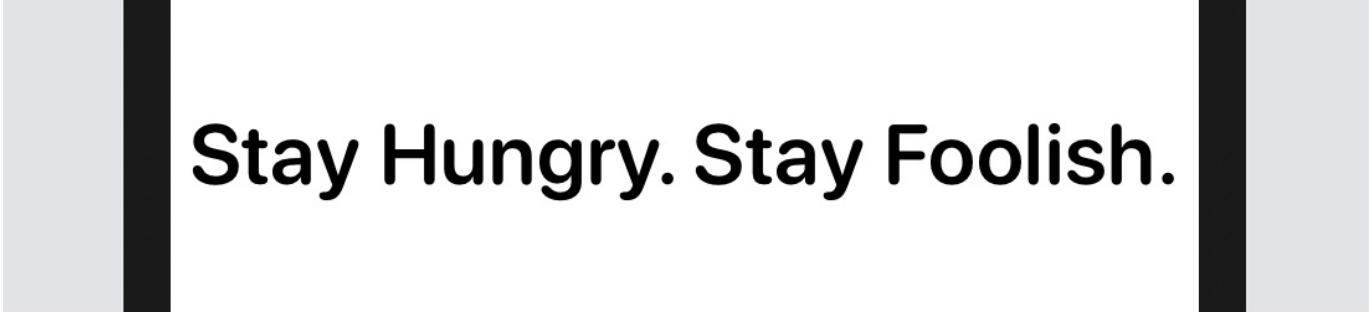
The preview canvas shows a black iPhone-like device with rounded corners. Inside the screen, the text "Stay Hungry. Stay Foolish." is displayed in a large, bold, black font.

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(.largeTitle, design: .rounded))
```

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



Stay Hungry. Stay Foolish.

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.

In case you want to use a fixed-size font, you can write the line of code like this:

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

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

As said, you can continue to chain other modifiers to customize the text. Now let's change the font color. To do that, you can use the `foregroundColor` modifier like this:

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

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

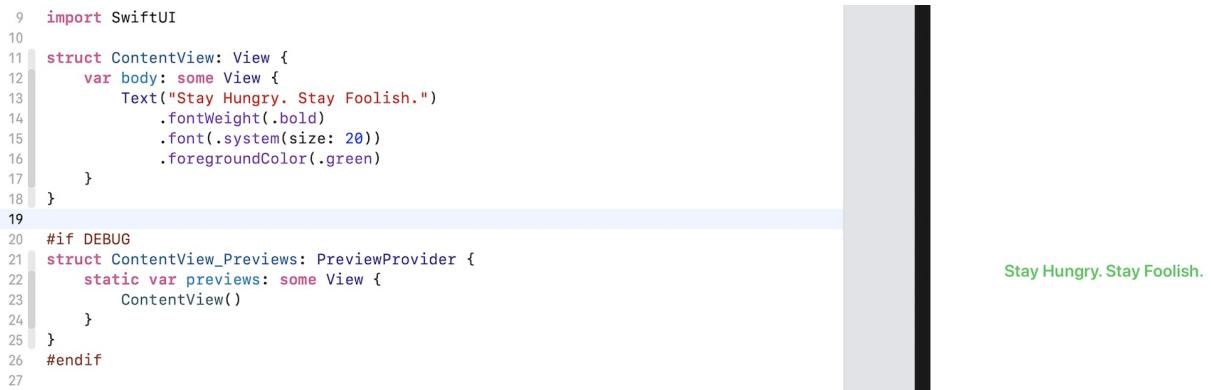


Figure 7. Changing the font color

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 popover menu. Choose *Inspect...* and then you can edit the text/font properties.

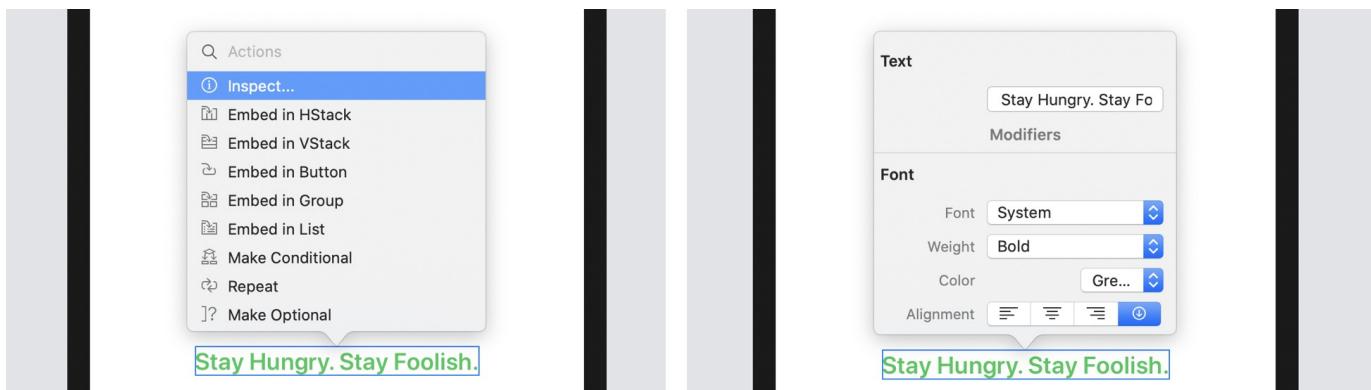


Figure 8. Using the Inspect feature to edit the properties of the text

Using Custom Fonts

By default, all text are 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 use `.custom` and specify the preferred font name. The name of the font can be found in Font Book. You can open Finder > Application and click *Font Book* to launch the app.

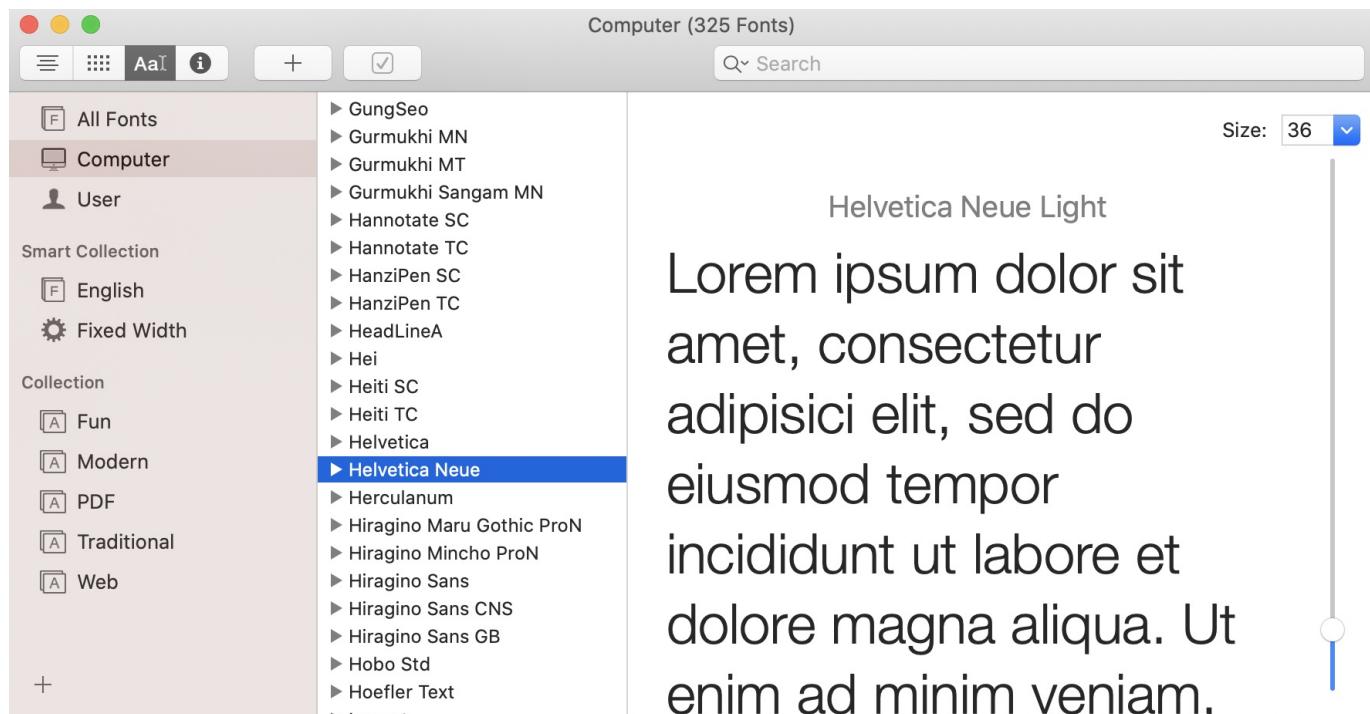


Figure 9. Font Book

Working with Multiline Text

`Text` supports multiple lines by default, so it can display a paragraph of text without using any additional modifiers. Replace the 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 important, 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 value. Just make sure it's long enough. Once you made the change, the design canvas should render a multiline text label.

```
9 import SwiftUI  
10  
11 struct ContentView: View {  
12     var body: some View {  
13         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)  
    }  
}  
19  
20 #if DEBUG  
21 struct ContentView_Previews: PreviewProvider {  
22     static var previews: some View {  
23         ContentView()  
24     }  
25 }  
26 #endif  
27
```

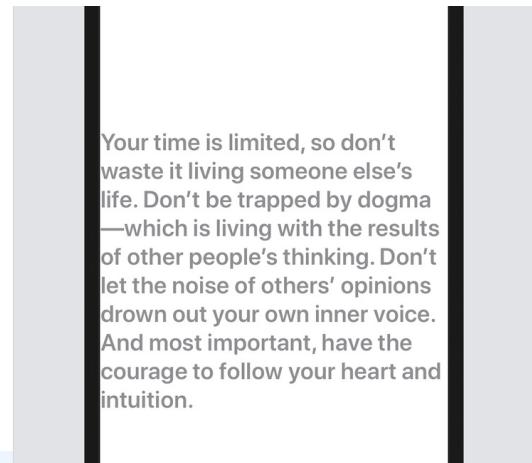


Figure 10. Display multiline text

To align the text in centre, insert the `multilineTextAlignment` modifier and set the value to `.center` like this:

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

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

```
.lineLimit(3)
```

By default, the system is set to use tail truncation. To modify the truncation mode of the text, you can 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.

```
9 import SwiftUI
10
11 struct ContentView: View {
12     var body: some View {
13         Text("Your time is limited, so don't waste it living someone else's life.
14             Don't be trapped by dogma—which is living with the results of other
15             people's thinking. Don't let the noise of others' opinions drown out
16             your own inner voice. And most important, have the courage to follow
17             your heart and intuition.")
18             .fontWeight(.bold)
19             .font(.title)
20             .foregroundColor(.gray)
21             .multilineTextAlignment(.center)
22             .lineLimit(3)
23             .truncationMode(.head)
24     }
25 }
26
27 #if DEBUG
28 struct ContentView_Previews: PreviewProvider {
29     static var previews: some View {
30         ContentView()
31     }
32 }
33#endif
```

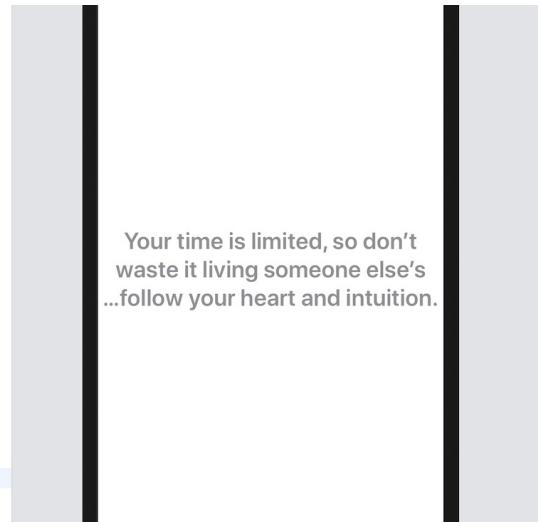


Figure 11. Using the `.head` truncation mode

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. In case you want to alter the default setting, you can 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 for each side of the text. Insert the following line of code after the `lineSpacing` modifier:

```
.padding()
```

Your design canvas should now show the result like this:

```
9 import SwiftUI
10
11 struct ContentView: View {
12     var body: some View {
13         Text("Your time is limited, so don't waste it living someone else's life.
14             Don't be trapped by dogma—which is living with the results of other
15             people's thinking. Don't let the noise of others' opinions drown out
16             your own inner voice. And most important, have the courage to follow
17             your heart and intuition.")
18             .fontWeight(.bold)
19             .font(.title)
20             .foregroundColor(.gray)
21             .multilineTextAlignment(.center)
22             .lineSpacing(10)
23             .padding()
24     }
25 }
26
27 #if DEBUG
28 struct ContentView_Previews: PreviewProvider {
29     static var previews: some View {
30         ContentView()
31     }
32 }
33#endif
```

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 API to let you easily rotate the text. You can use the `rotateEffect` modifier and pass the rotation degree 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.

```

8 import SwiftUI
9
10 struct ContentView: View {
11     var body: some View {
12         Text("Your time is limited, so don't waste it living someone else's life.
13             Don't be trapped by dogma—which is living with the results of other
14             people's thinking. Don't let the noise of others' opinions drown out
15             your own inner voice. And most important, have the courage to follow
16             your heart and intuition.")
17             .fontWeight(.bold)
18             .font(.title)
19             .foregroundColor(.gray)
20             .multilineTextAlignment(.center)
21             .lineSpacing(10)
22             .padding()
23             .rotationEffect(.degrees(45))
24 }
25
26 #if DEBUG
27 struct ContentView_Previews: PreviewProvider {
28     static var previews: some View {
29         ContentView()
30     }
31 #endif

```

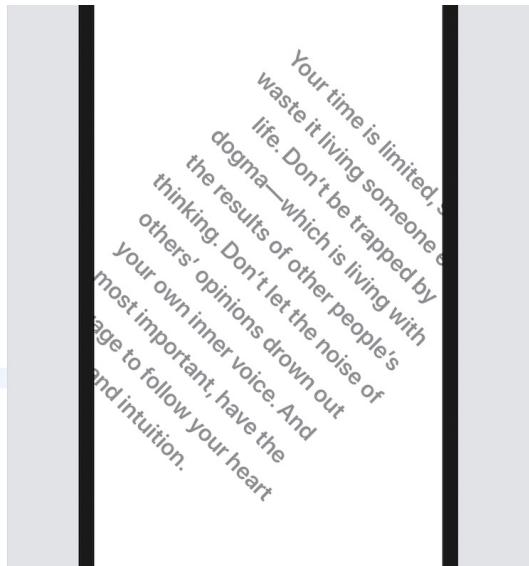


Figure 13. Rotate the text

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 can write the code like this:

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

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

```

9 import SwiftUI
10
11 struct ContentView: View {
12     var body: some View {
13         Text("Your time is limited, so don't waste it living someone else's life.
14             Don't be trapped by dogma—which is living with the results of other
15             people's thinking. Don't let the noise of others' opinions drown out
16             your own inner voice. And most important, have the courage to follow
17             your heart and intuition.")
18             .fontWeight(.bold)
19             .font(.title)
20             .foregroundColor(.gray)
21             .multilineTextAlignment(.center)
22             .lineSpacing(10)
23             .padding()
24             .rotationEffect(.degrees(20), anchor: UnitPoint(x: 0, y: 0))
25 }
26
27 #if DEBUG
28 struct ContentView_Previews: PreviewProvider {
29     static var previews: some View {
30         ContentView()
31     }
32 }

```

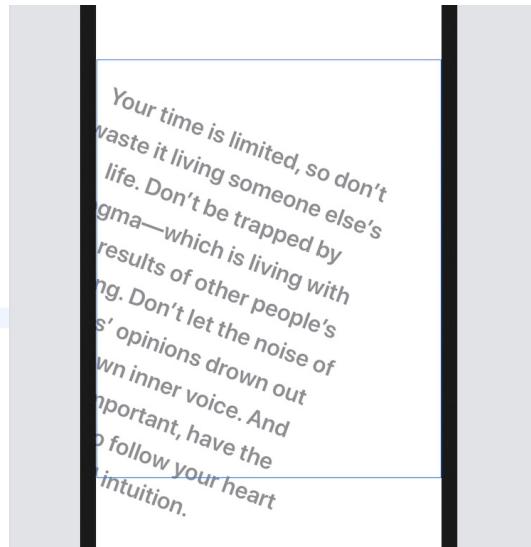


Figure 14. Rotate the text around the top-left of the text view

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

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

With just a line of code, you already re-create the Star Wars perspective text.

```

9 import SwiftUI
10
11 struct ContentView: View {
12     var body: some View {
13         Text("Your time is limited, so don't waste it living someone else's life.
14             Don't be trapped by dogma—which is living with the results of other
15             people's thinking. Don't let the noise of others' opinions drown out
16             your own inner voice. And most important, have the courage to follow
17             your heart and intuition.")
18             .fontWeight(.bold)
19             .font(.title)
20             .foregroundColor(.gray)
21             .multilineTextAlignment(.center)
22             .lineSpacing(10)
23             .padding()
24             .rotation3DEffect(.degrees(60), axis: (x: 1, y: 0, z: 0))
25 }
26
27 #if DEBUG
28 struct ContentView_Previews: PreviewProvider {
29     static var previews: some View {
30         ContentView()
31     }
32 }
33 #endif

```



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:

```
.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.

```
9 import SwiftUI
10
11 struct ContentView: View {
12     var body: some View {
13         Text("Your time is limited, so don't waste it living someone else's life.
14             Don't be trapped by dogma—which is living with the results of other
15             people's thinking. Don't let the noise of others' opinions drown out
16             your own inner voice. And most important, have the courage to follow
17             your heart and intuition.")
18             .fontWeight(.bold)
19             .font(.title)
20             .foregroundColor(.gray)
21             .multilineTextAlignment(.center)
22             .lineSpacing(10)
23             .padding()
24             .rotation3DEffect(.degrees(60), axis: (x: 1, y: 0, z: 0))
25             .shadow(color: .gray, radius: 2, x: 0, y: 15)
26     }
27 }
28
29 #if DEBUG
30 struct ContentView_Previews: PreviewProvider {
31     static var previews: some View {
32         ContentView()
33     }
34 }
35 #endif
```

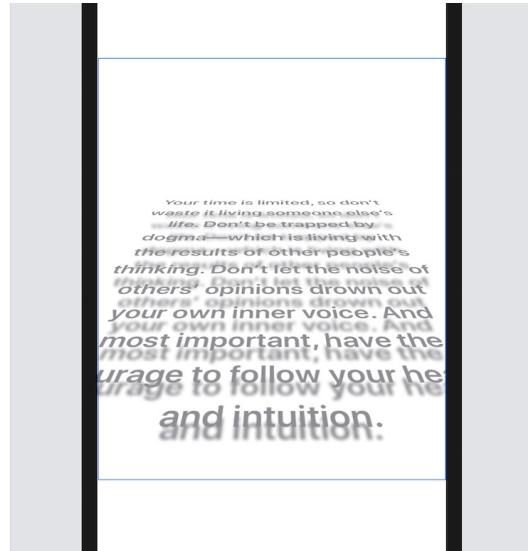


Figure 16. Applying the drop shadow effect

Summary

Do you enjoy creating user interface with SwiftUI? I hope so. The declarative syntax of SwiftUI made 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 project here:

- Demo project (<https://www.appcoda.com/resources/swiftui/SwiftUIText.zip>)

Chapter 3

Working with Images

Now that you should have some basic ideas about SwiftUI and understand how to display textual content, let's see how to display images in this chapter.

Other than text, image is 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 SwiftUI

First, fire up Xcode and create a new project using the *Single View App* template. Type the name of the project and set it to *SwiftUIImage*. 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, you need to check the *Use SwiftUI* checkbox. Click *Next* and choose a folder to create the project.

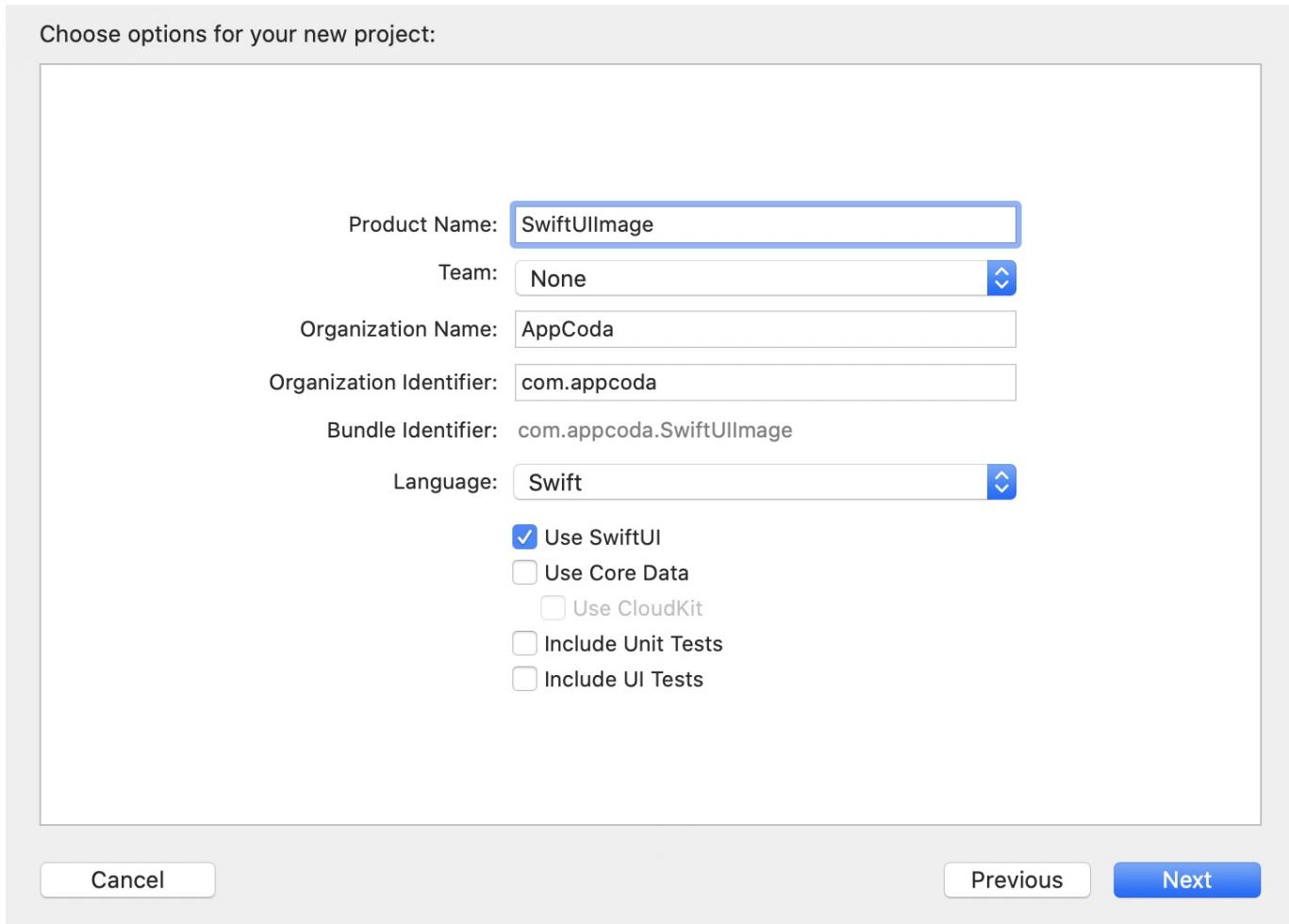


Figure 1. Creating a new project

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.

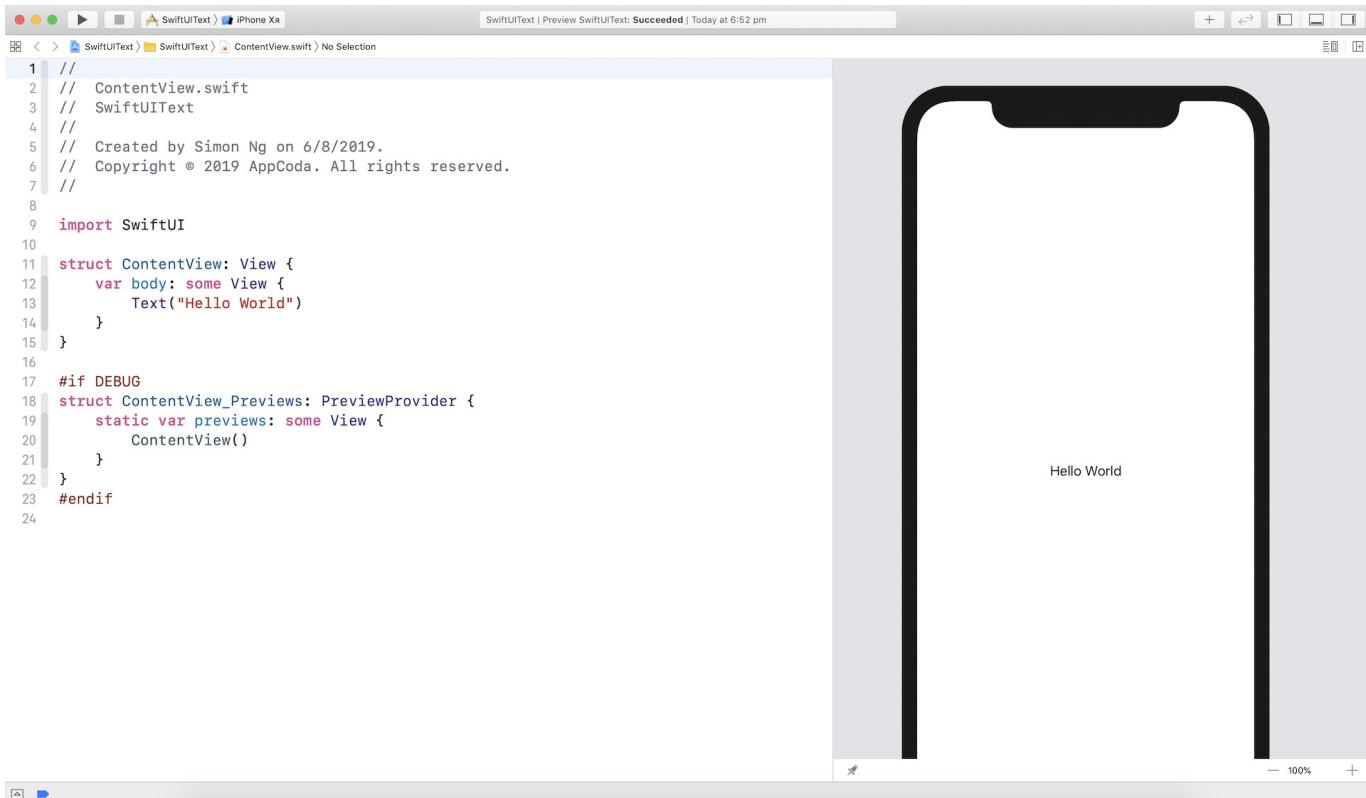


Figure 2. Previewing the generated code

Understanding SF Symbols

SF Symbols provides a set of over 1,500 consistent, highly configurable symbols you can use in your app. Apple designed SF Symbols to integrate seamlessly with the San Francisco system font, so the symbols automatically ensure optical vertical alignment with text for all weights and sizes. SF Symbols are available in a wide range of weights and scales to help you create adaptable designs.

Before I showed you how to display an image on screen, let's first talk about where the images come from. Needless to say, you can provide your own images to use in the app. In iOS 13, Apple introduced a large set of system images called SF Symbols that allows developers to use them in any apps.

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 to prepare is find out the name of the symbol. With over 1,500 symbols available for your use, Apple has released an app called **SF Symbols** (<https://developer.apple.com/design/downloads/SF-Symbols.dmg>), so that you can easily explore the symbols and locate the one that fits your need. I highly recommend you to install the app before proceeding to the next section.

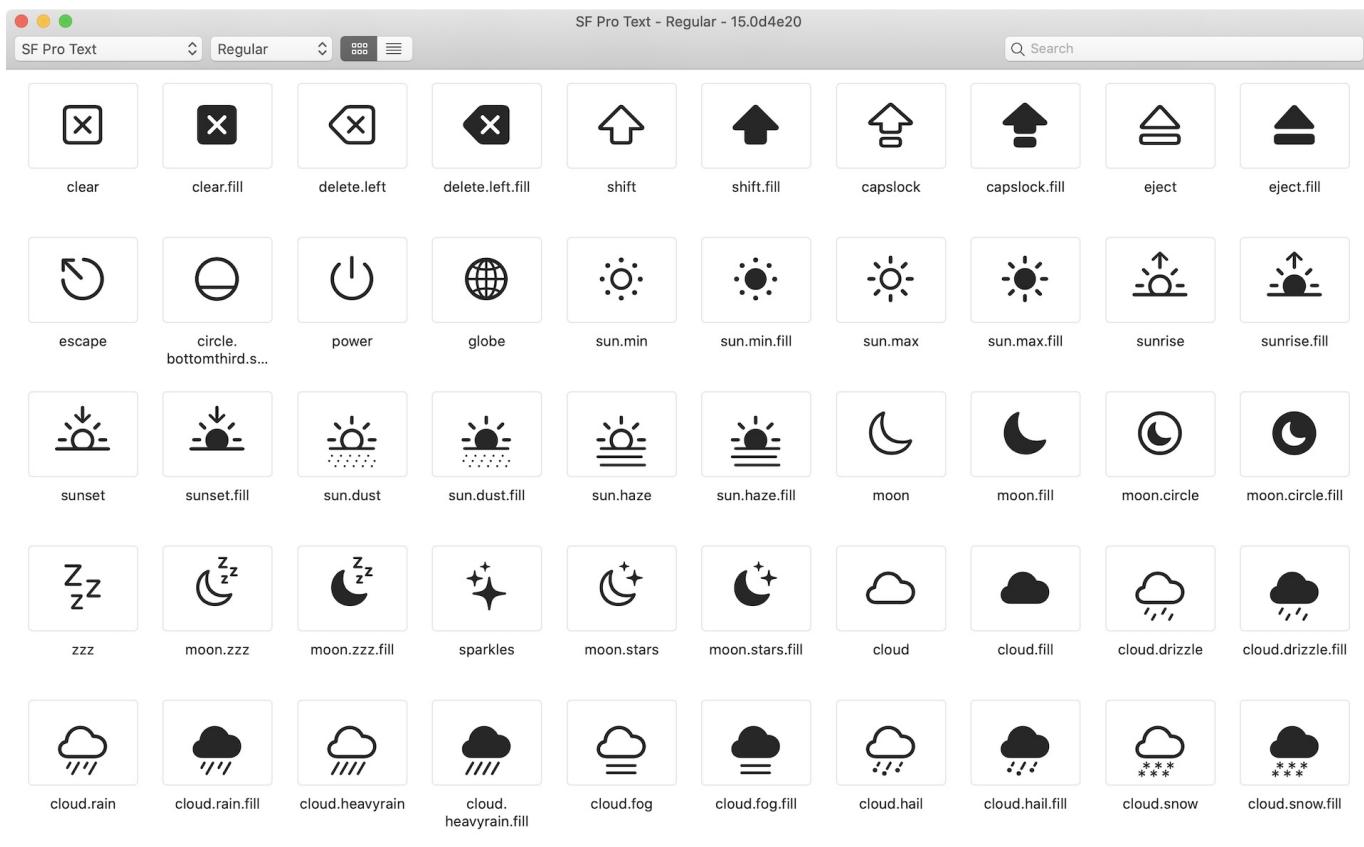


Figure 3. SF Symbols App

Displaying a System Image

To display a system image 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))
```

You can vary the font size to see how the image responds.

```
8 import SwiftUI
9
10 struct ContentView: View {
11     var body: some View {
12         Image(systemName: "cloud.heavyrain")
13             .font(.system(size: 100))
14     }
15 }
16
17 #if DEBUG
18 struct ContentView_Previews: PreviewProvider {
19     static var previews: some View {
20         ContentView()
21     }
22 }
23
24 #endif
25
```

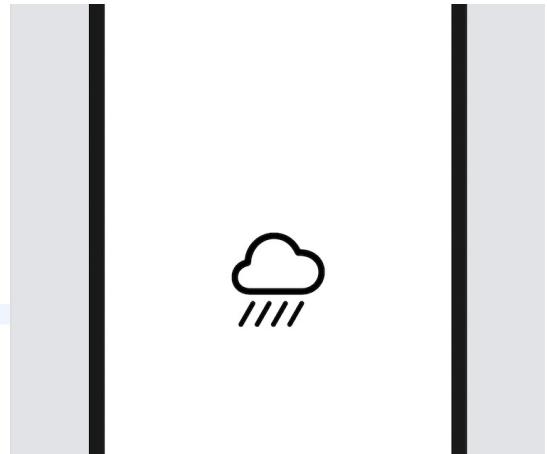


Figure 4. Display a system image

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

Say, to change its 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 just need to 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

Earlier, we use the built-in images provided by Apple. Obviously, you will have your own images to use in the app. So, how can you load the 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/Qo-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 the image in your project, the very first thing you have to do is import them into the asset catalog (i.e. `Assets.xcassets`). Assuming you already prepared the image (`paris.jpg`), press command+o to reveal the project navigator and then choose `Assets.xcassets` . Now open Finder and drag the image to the outline view.

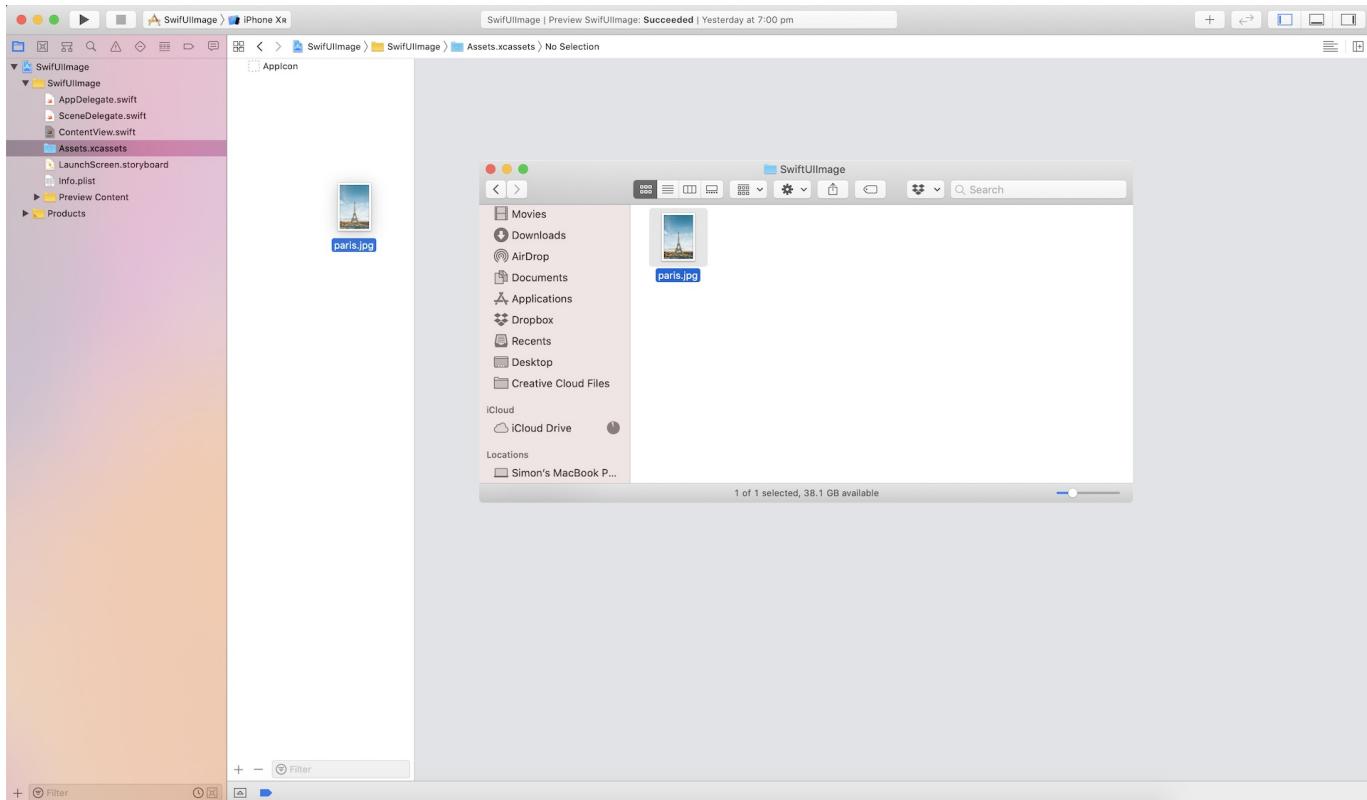


Figure 5. Drag the image to the asset catalog

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. Furthermore, 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:

```
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 can only see a part of the image.

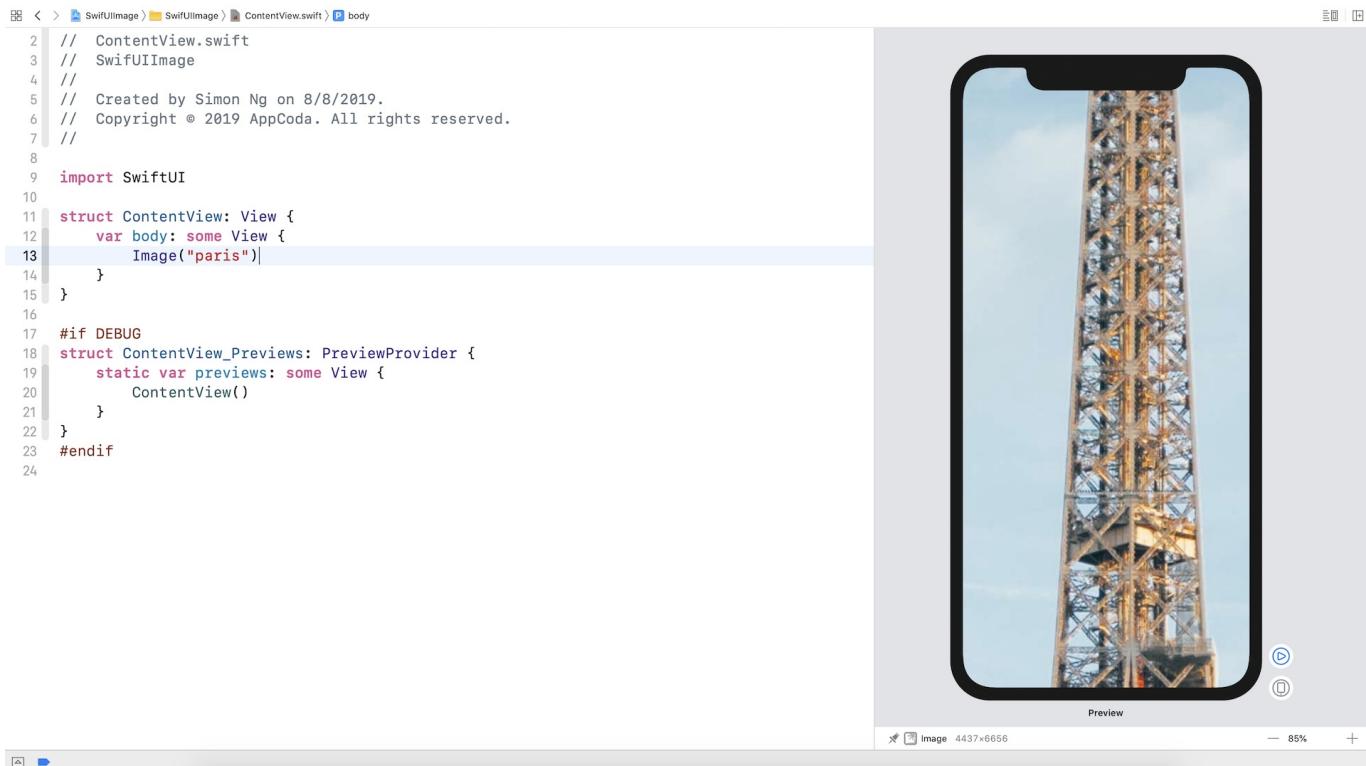


Figure 6. Loading a custom image

Resizing an Image

So, how can you resize the image? The `resizable()` modifier can be used for this:

```
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).

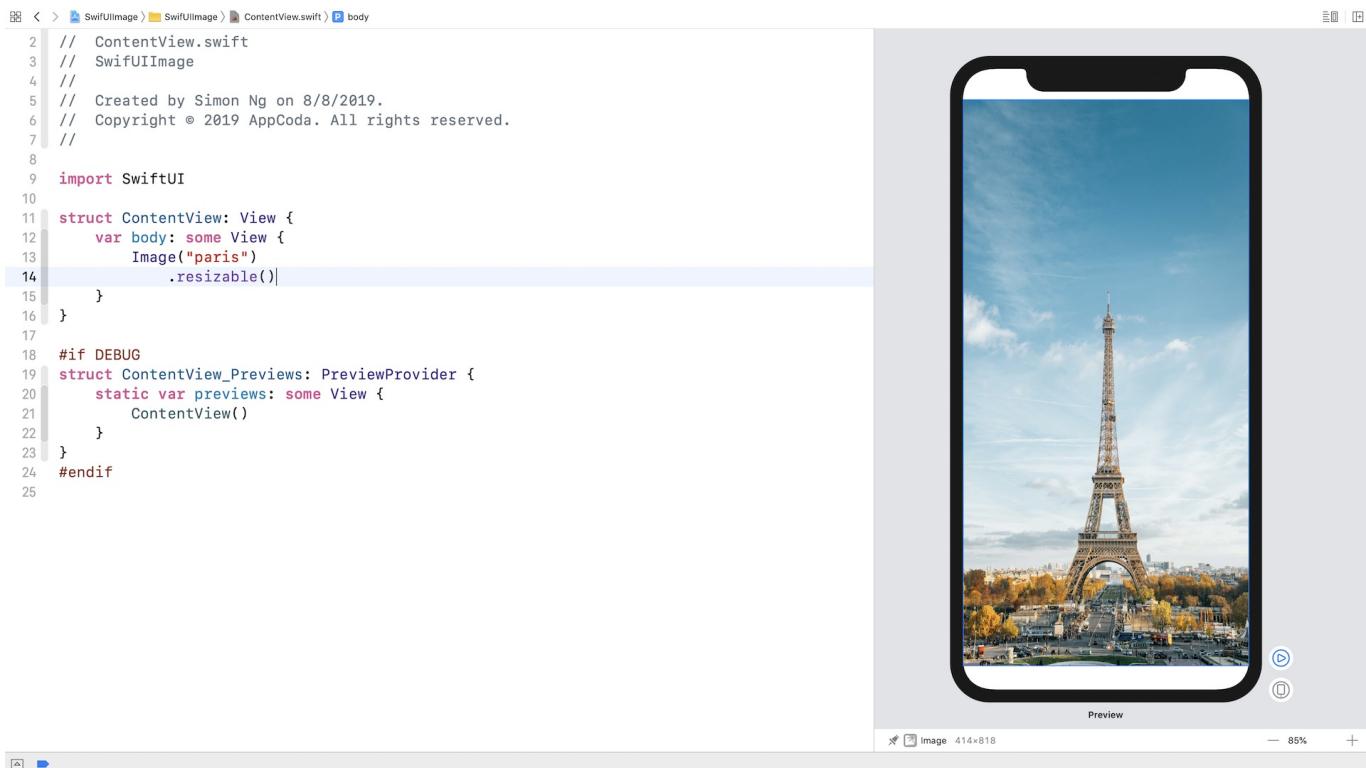


Figure 7. Resizing the image with the `resizable` modifier

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 that defines 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. With safe area, it can prevent you from accidentally hiding the system UI component like status bar, navigation bar, and tab bar.

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

The screenshot shows the Xcode interface with the code editor on the left containing the following Swift code:

```
2 // ContentView.swift
3 // SwiftUIImage
4 //
5 // Created by Simon Ng on 8/8/2019.
6 // Copyright © 2019 AppCoda. All rights reserved.
7 //
8
9 import SwiftUI
10
11 struct ContentView: View {
12     var body: some View {
13         Image("paris")
14             .resizable()
15
16             .edgesIgnoringSafeArea(.all)
17     }
18 }
19
20 #if DEBUG
21 struct ContentView_Previews: PreviewProvider {
22     static var previews: some View {
23         ContentView()
24     }
25 }
26 #endif
```

To the right is a simulator window titled "Preview" showing an iPhone X displaying the Eiffel Tower image. The image is centered and covers the entire screen, ignoring the safe area.

Figure 8. Ignoring the safe area

You can choose to ignore the safe area for a specific edge. Say, to ignore the safe area for the top edge, you can pass `.top` as the parameter. In the sample code, we pass `.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 one, you should find that the aspect ratio is a bit distorted. The *stretch* mode doesn't care the aspect ratio of the image. It just keeps stretching 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()
```

The screenshot shows the Xcode interface with the ContentView.swift file open. The code defines a struct ContentView with a body containing an Image("paris") view. The image is made resizable and has its aspect ratio set to fit. A preview of the app is shown in the simulator, displaying the Eiffel Tower in Paris with its original aspect ratio preserved.

```
2 // ContentView.swift
3 // SwiftUIImage
4 //
5 // Created by Simon Ng on 8/8/2019.
6 // Copyright © 2019 AppCoda. All rights reserved.
7 //
8
9 import SwiftUI
10
11 struct ContentView: View {
12     var body: some View {
13         Image("paris")
14             .resizable()
15             .aspectRatio(contentMode: .fit)
16     }
17 }
18
19 #if DEBUG
20 struct ContentView_Previews: PreviewProvider {
21     static var previews: some View {
22         ContentView()
23     }
24 }
25 #endif
26
```

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 would 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 it to as large as possible, you can apply the `.fill` content mode:

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

Let's try to limit the size of the image. This would give you a better idea about the difference between these two modes. The `frame` modifier allows you to control the size of a view. For example, by setting the frame's width to 300 points, the image's width will be limited to 300 points.

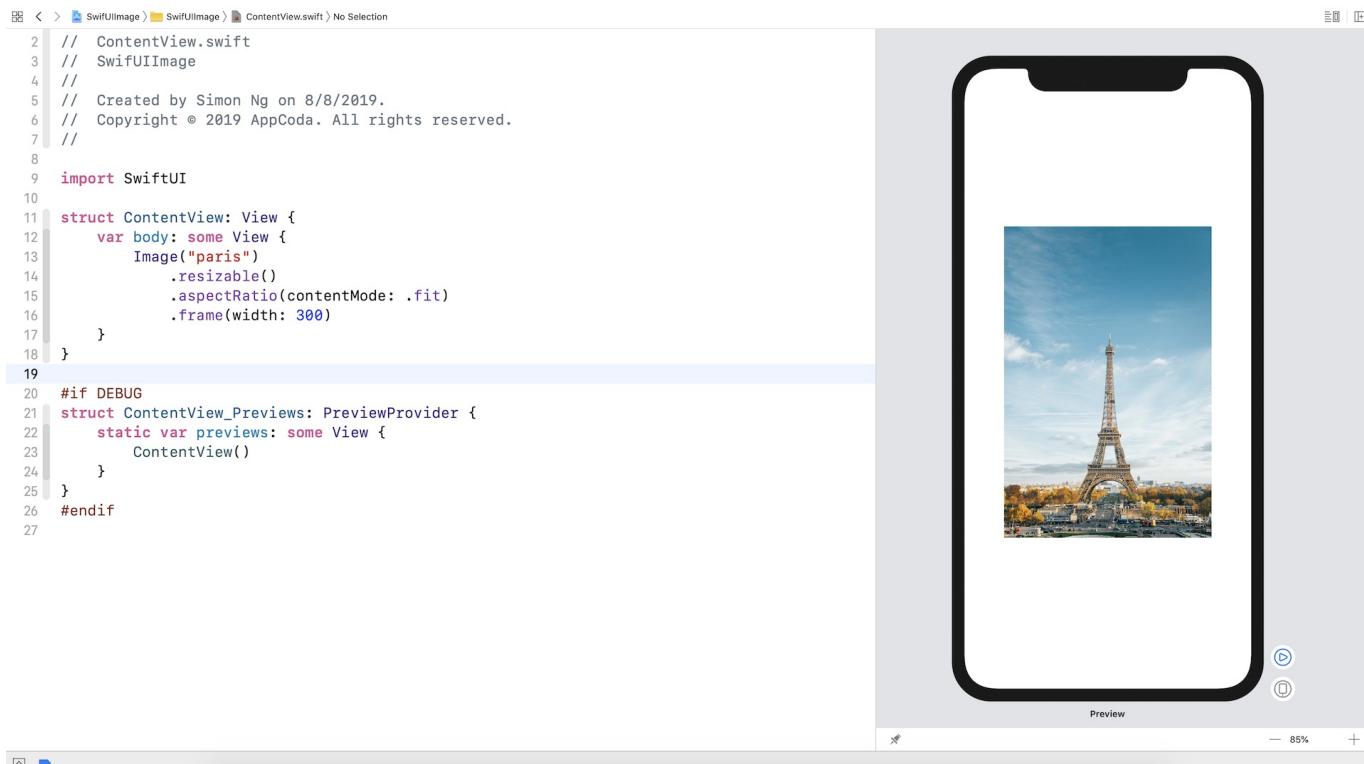


Figure 10. Scaling down the image and keep the original aspect ratio

Now replace the `Image` code with the following:

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

The image will be scaled down 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 is kept.

The screenshot shows the Xcode interface with the code editor on the left and a preview window on the right. The code editor displays the following Swift code:

```
2 // ContentView.swift
3 // SwiftUIImage
4 //
5 // Created by Simon Ng on 8/8/2019.
6 // Copyright © 2019 AppCoda. All rights reserved.
7 //
8
9 import SwiftUI
10
11 struct ContentView: View {
12     var body: some View {
13         Image("paris")
14             .resizable()
15             .aspectRatio(contentMode: .fill)
16             .frame(width: 300)
17     }
18 }
19
20 #if DEBUG
21 struct ContentView_Previews: PreviewProvider {
22     static var previews: some View {
23         ContentView()
24     }
25 }
26 #endif
```

The preview window shows a smartphone displaying an image of the Eiffel Tower against a blue sky. The image is centered and fills the screen, demonstrating the use of the `.fill` content mode.

Figure 11. Using `.fill` content mode

One thing you may notice is that the image's width was not what we expected. It still takes up the whole screen width. To make it work correctly, you have to use the `clipped` modifier to eliminate extra parts of the view.

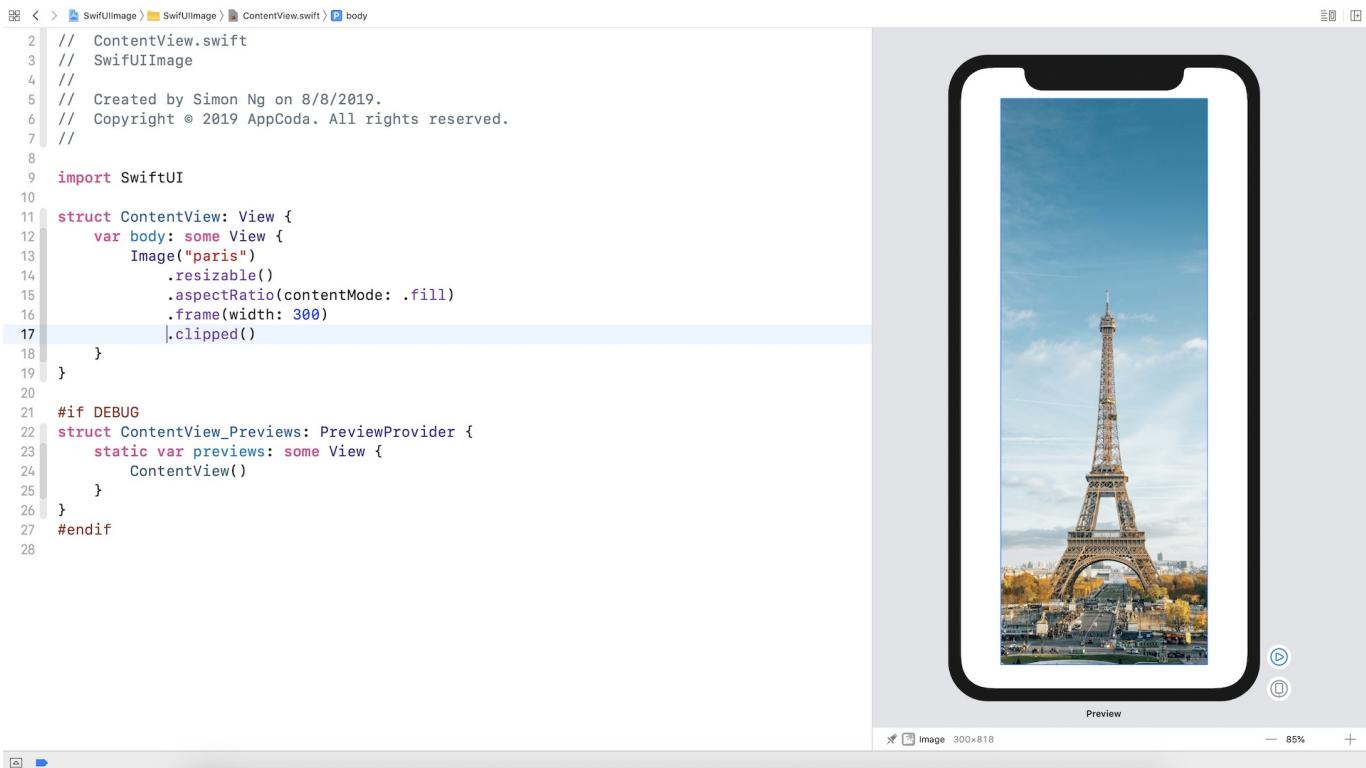


Figure 12. Use .clipped to clip the view

Creating a Circular Image

Other than clipping the image in rectangle shape, SwiftUI provides other modifiers for you to clip the image into various shapes such as circle. 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 (e.g. `Ellipse()`) to create an image with different shape. Figure 13 shows you some of the examples.

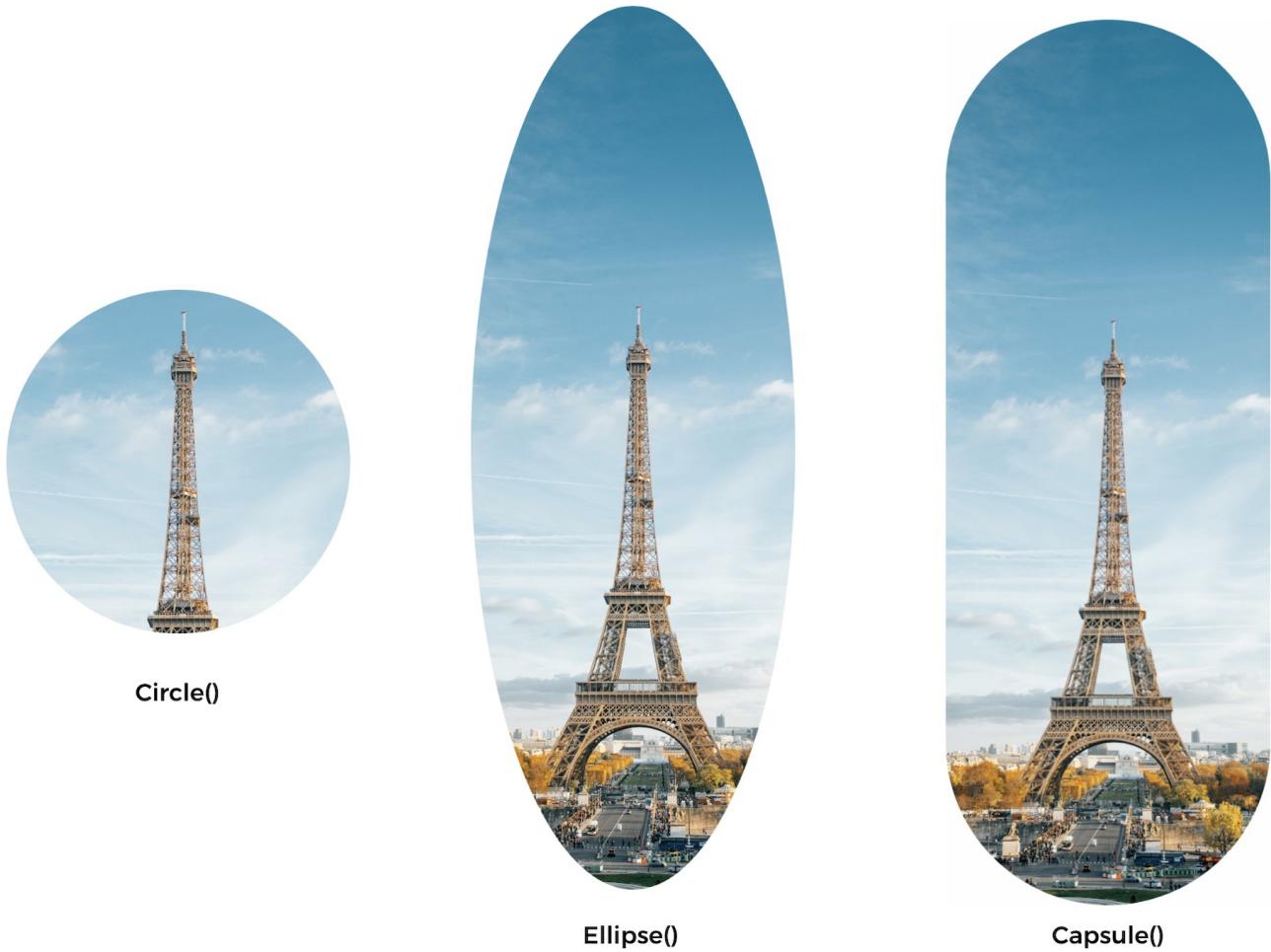


Figure 13. Use the `.clipShape` modifier to create image with different shape

Adjusting the Opacity

SwiftUI comes with a modifier named `opacity` that you can use to control the opacity of an image (or any views). You pass a value between 0 and 1 to indicate the opacity of the image. Here, 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.

```

9 import SwiftUI
10
11 struct ContentView: View {
12     var body: some View {
13         Image("paris")
14             .resizable()
15             .aspectRatio(contentMode: .fill)
16             .frame(width: 300)
17             .clipShape(Circle())
18             .opacity(0.5)
19     }
20 }
21
22 #if DEBUG
23 struct ContentView_Previews: PreviewProvider {
24     static var previews: some View {
25         ContentView()
26     }
27 }
28#endif
29

```

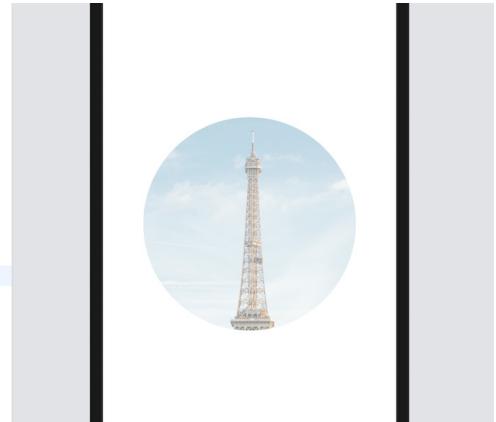


Figure 14. Adjusting the opacity to 50%

Applying an Overlay to an Image

When designing your app, sometimes 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`) onto the existing image. You can 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).

```
10 struct ContentView: View {  
11     var body: some View {  
12         Image("paris")  
13             .resizable()  
14             .aspectRatio(contentMode: .fill)  
15             .frame(width: 300)  
16             .clipShape(Circle())  
17             .overlay(  
18                 Image(systemName: "heart.fill")  
19                     .font(.system(size: 50))  
20                     .foregroundColor(.black)  
21                     .opacity(0.5)  
22             )  
23     }  
24 }  
25 }  
26 }  
27 }
```



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. You can write the code 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 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  
)
```

In the `overlay` modifier, you just need to create a `Text` view and this text view will be applied as an overlay to the image. I believe you should be familiar with the modifiers of the `Text` view as we have discussed in the earlier chapter. We simply change the font and its color. In addition to that, we add some paddings 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. You may change the value to `.center` or `.bottom` to see how it works.

```
2 // ContentView.swift
3 // SwiftUIImage
4 //
5 // Created by Simon Ng on 8/8/2019.
6 // Copyright © 2019 AppCoda. All rights reserved.
7 //
8
9 import SwiftUI
10
11 struct ContentView: View {
12     var body: some View {
13         Image("paris")
14             .resizable()
15             .aspectRatio(contentMode: .fit)
16             .overlay(
17
18                 Text("If you are lucky enough to
19                     then wherever you go to
20                     for Paris is a moveable
21                     .fontWeight(.heavy)
22                     .font(.system(.headline))
23                     .foregroundColor(.white)
24                     .padding()
25                     .background(Color.black)
26                     .cornerRadius(10)
27                     .opacity(0.8)
28                     .padding(),
29
30                     alignment: .top
31
32             )
33
34 #if DEBUG
35 struct ContentView_Previews: PreviewProvider {
36     static var previews: some View {
```

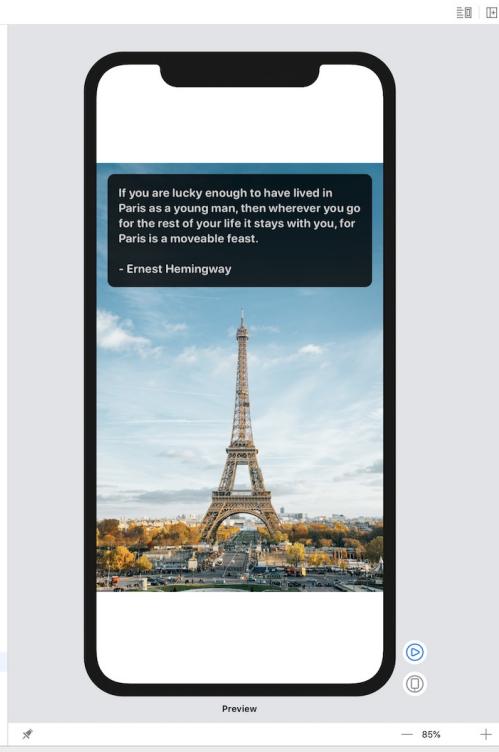


Figure 16. Applying an overlay to the existing image

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 and 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 darken effect, we set the opacity to `0.4`, giving it a 40% opacity. The image should now be darken.

This technique is particularly useful if you want to over some text with light color on an image that is too bright. Let's replace the `Image` code like this:

```
Image("paris")
    .resizable()
    .aspectRatio(contentMode: .fit)
    .frame(width: 300)
    .overlay(
        Rectangle()
            .foregroundColor(.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 views. In the code above, we use a transparent rectangle to darken the image. On top of the rectangle, we apply an overlay and place a `Text` over it. If you've made the change correctly, you should see the word "Paris" placing over the darkened image.

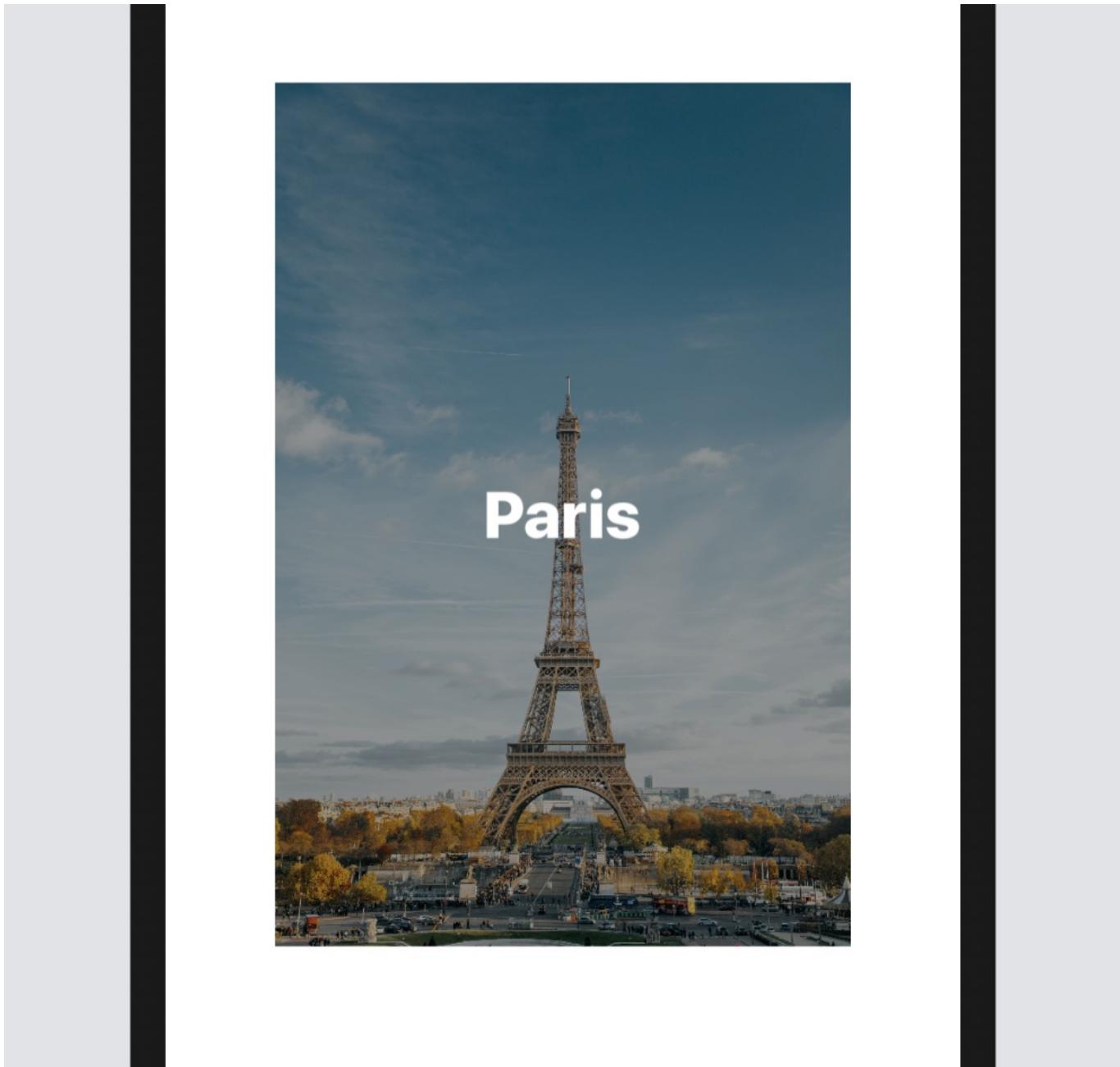


Figure 17. Darken an image and apply a text overlay

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 let us use different modifiers to apply various image effects. If you're an indie developer, the newly introduced SF Symbols would save you a

lot of time from searching the third-party icons.

For reference, you can download the complete project here:

- Demo project (<https://www.appcoda.com/resources/swiftui/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.

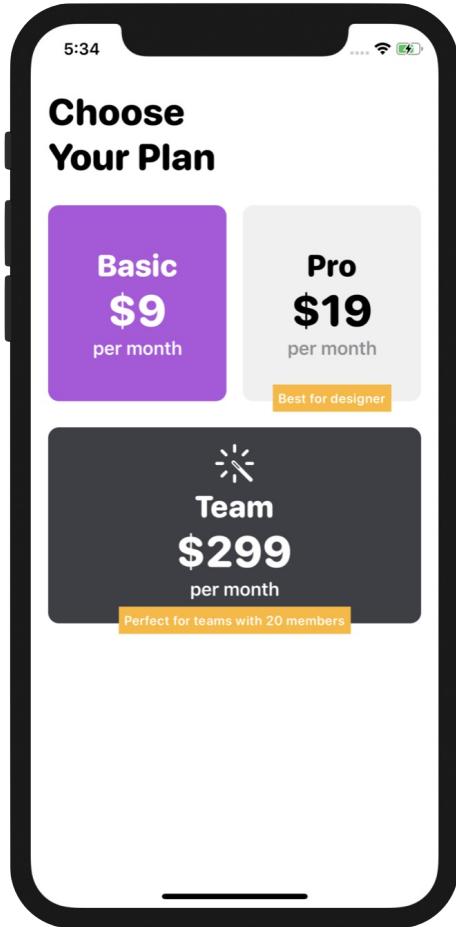


Figure 1. The demo app

Understanding VStack, HStack, and ZStack

SwiftUI provides three different types of stacks for developers to combine view in various orientation. 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 the others

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

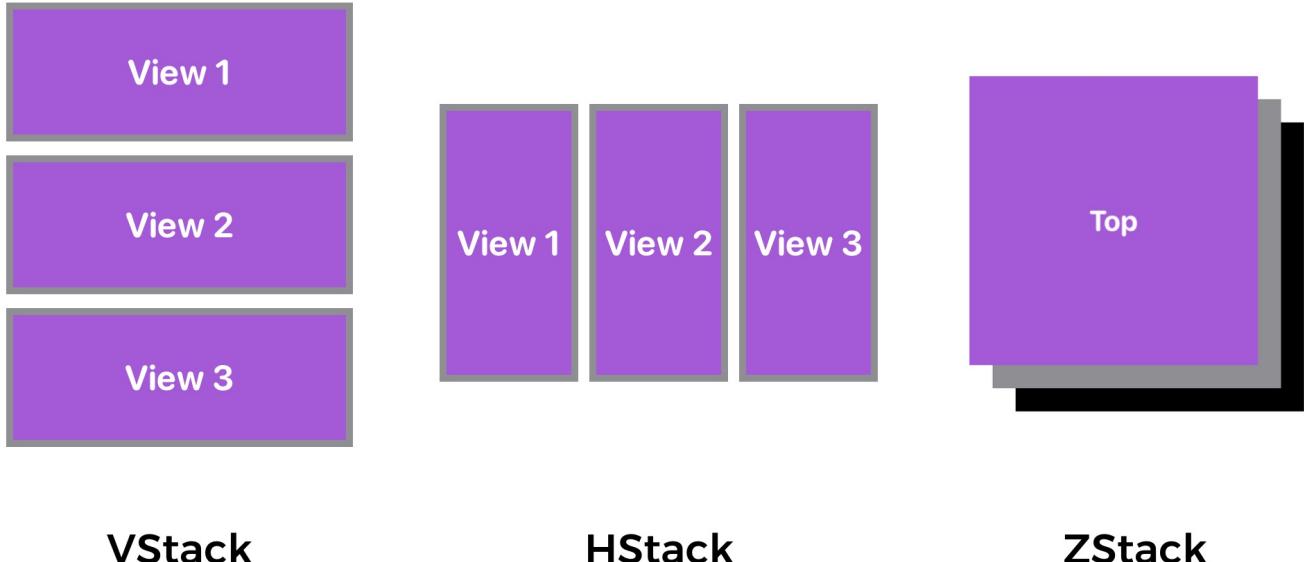


Figure 2. Creating a new project

Creating a New Project with SwiftUI enabled

First, fire up Xcode and create a new project using the *Single View Application* template. Type the name of the project. I set it to *SwiftUIStacks* but you're free to use any other name. All you need to ensure is select the *SwiftUI* option for User Interface.

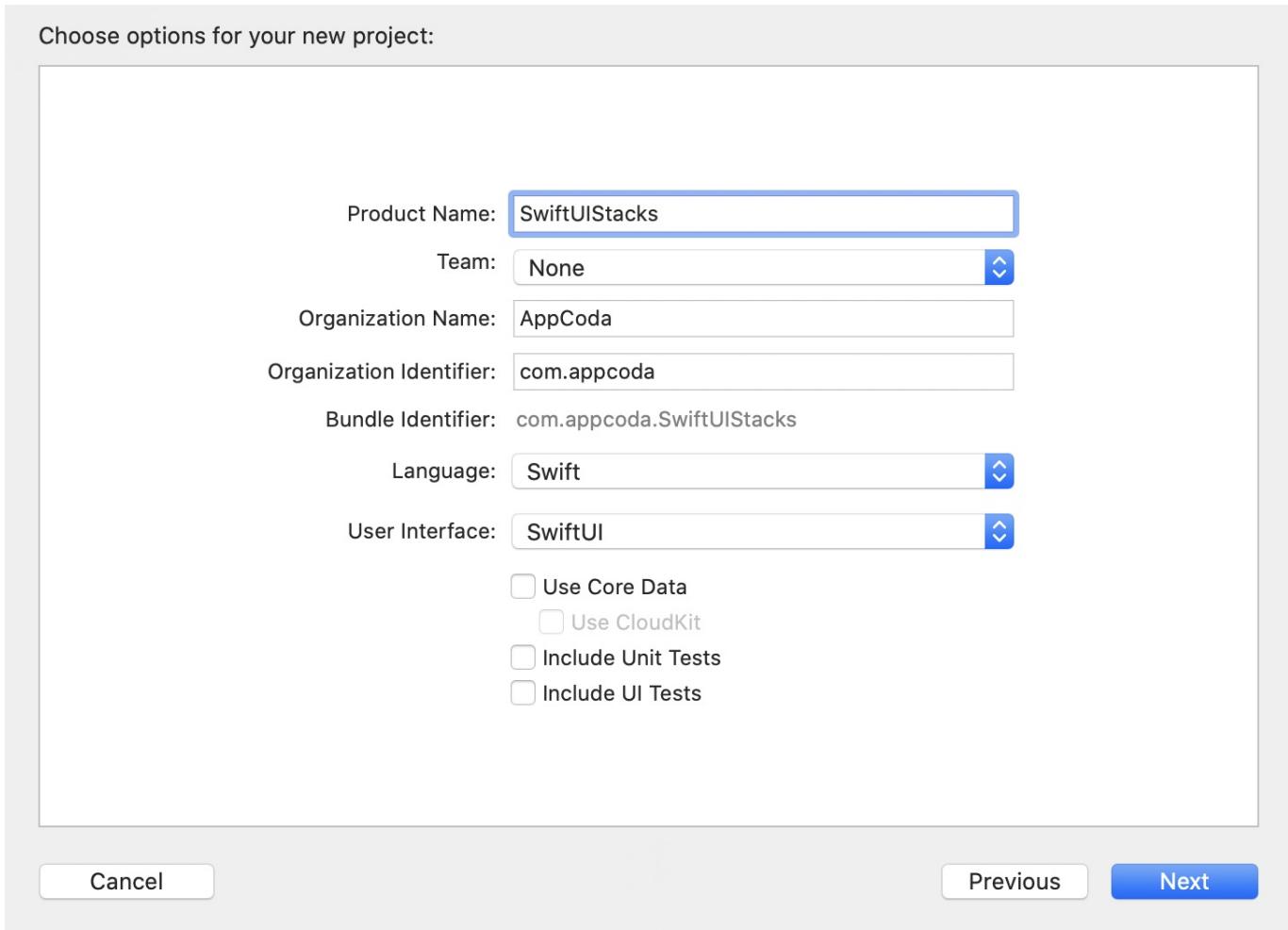


Figure 3. Creating a new project

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, you can click the *Resume* button in the canvas.

Using VStack

We're going to build the UI as displayed in figure 1, but 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 already generate 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 with 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)  
}
```

Optionally, you can adjust the space of the embedded views by using the `space` parameter. The figure below shows the resulting view.

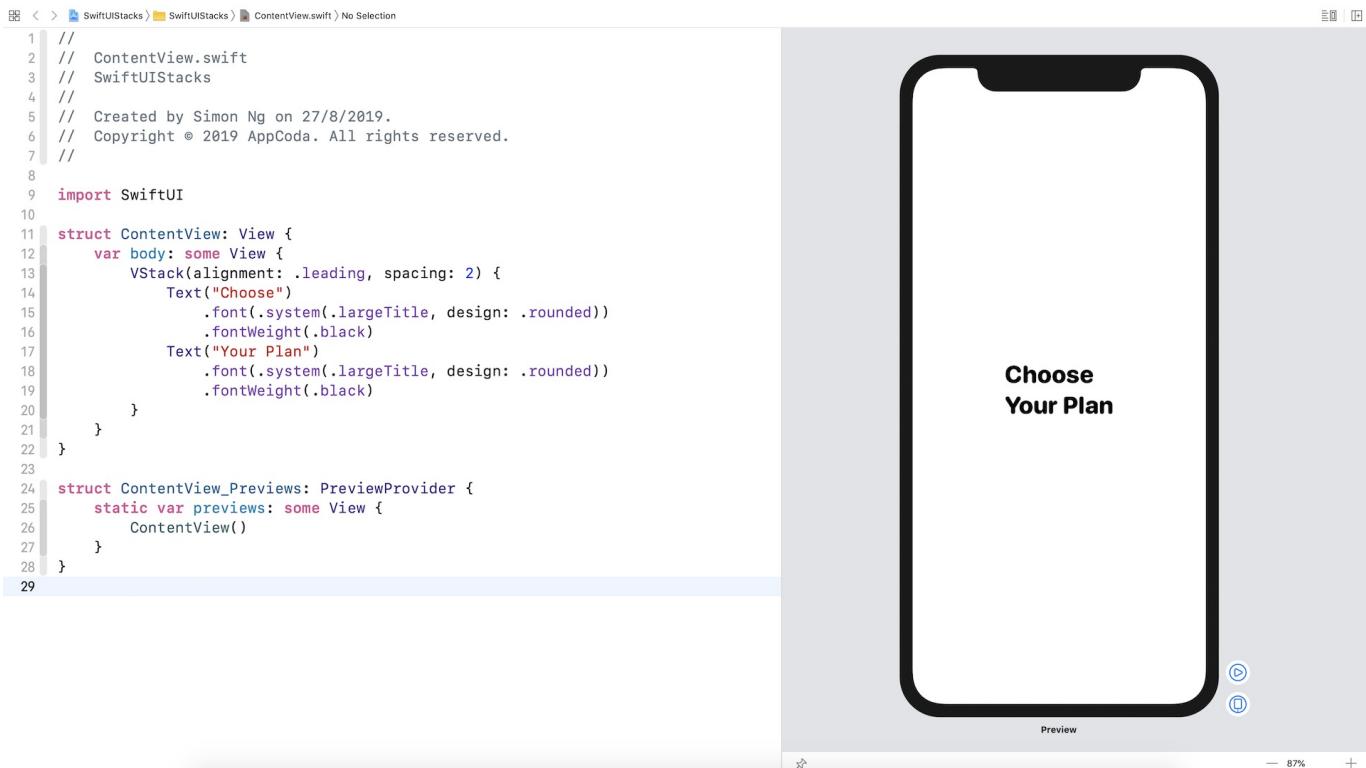


Figure 6. Changing the alignment of VStack

Using HStack

Next, let's lay out the first two pricing plans. If you look into the *Basic* and *Pro* plans, the look & feel of these two components are very similar. Take the *Basic* plan as an example, to realize the layout, you can use `vstack` to combine three text views.

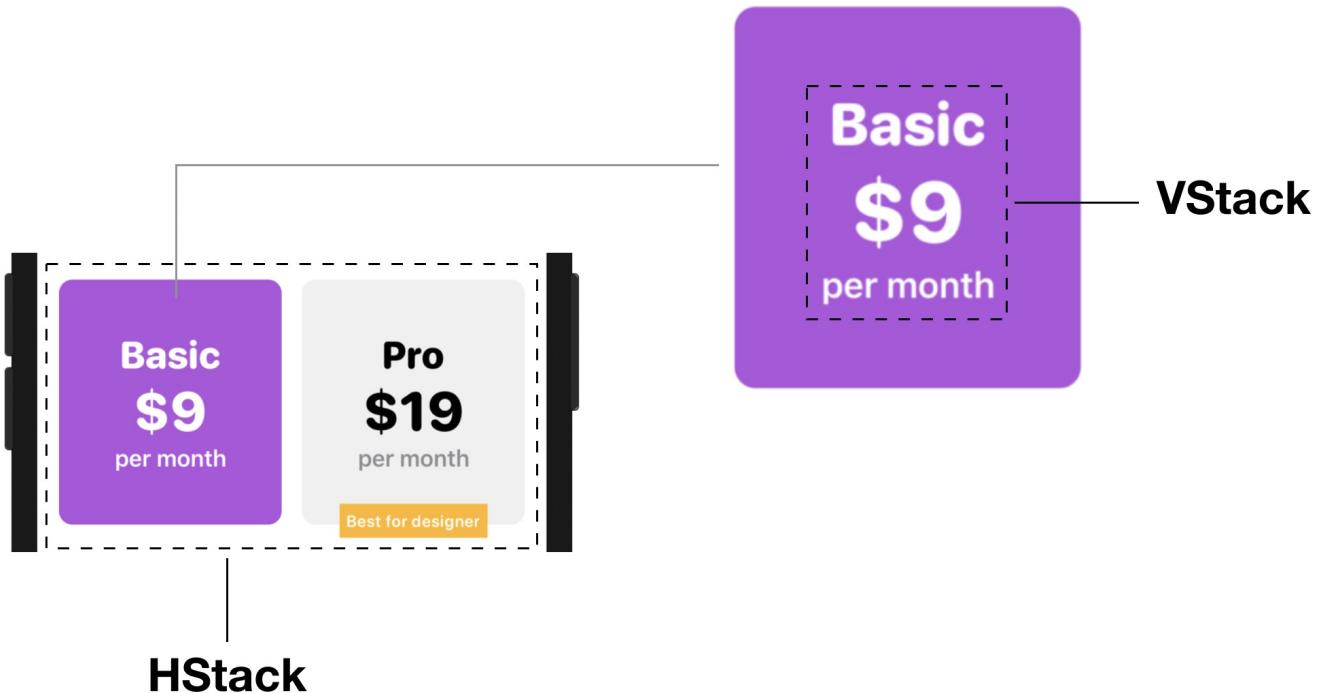


Figure 7. Layout the pricing plans

Both the *Basic* and *Pro* components are arranged side by side. By using `HStack`, you can lay out views horizontally. Stacks can be nested 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).

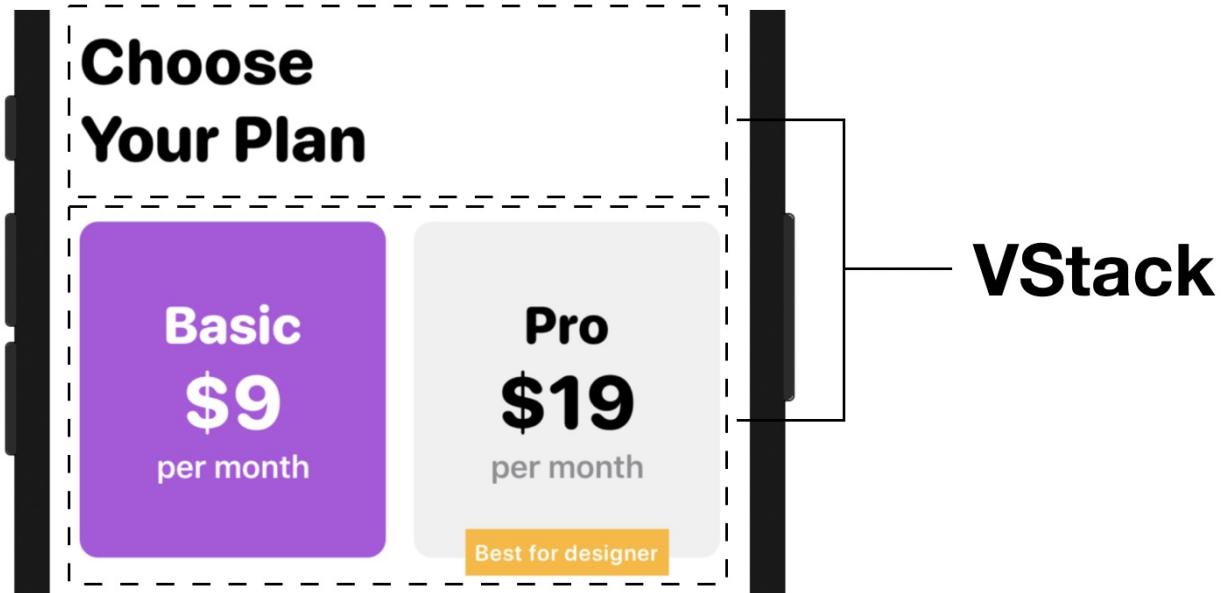


Figure 8. Using a VStack to embed other stack views

Now that you should have some basic ideas about 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 can 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`.

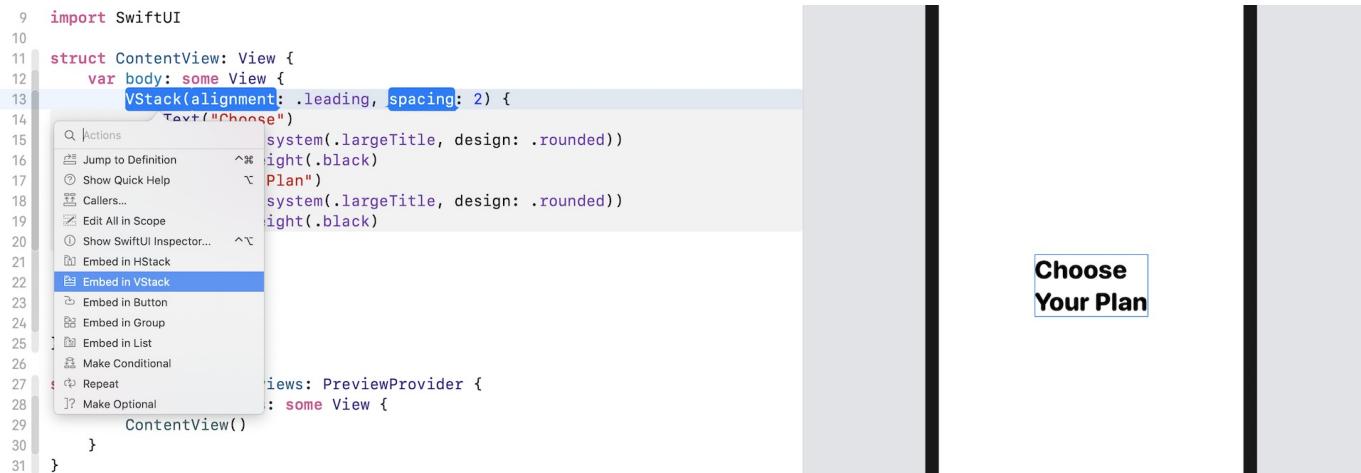


Figure 9. Embed in VStack

Xcode will then generate the required code to be embed the stack. Your code should become the following:

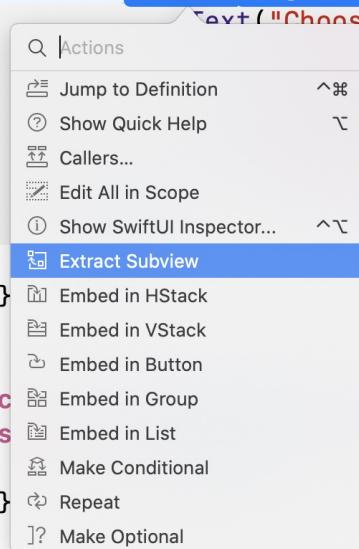
```
struct ContentView: View {
    var body: some View {
        VStack {
            VStack(alignment: .leading, spacing: 2) {
                Text("Choose")
                    .font(.system(.largeTitle, design: .rounded))
                    .fontWeight(.black)
                Text("Your Plan")
                    .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 would be wise to break this block of code into smaller parts so that the code would be easier to read and maintain.

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

```
11 struct ContentView: View {
12     var body: some View {
13         VStack {
14             VStack(alignment: .leading, spacing: 2) {
15                 Text("Choose")
16             }
17             system(.largeTitle, design: .rounded))
18             .font(.largeTitle)
19             .fontWeight(.black)
20             .padding(.top, 10)
21             .padding(.bottom, 10)
22         }
23     }
24 }
25 struct ContentView_Previews: PreviewProvider {
26     static var previews: some View {
27         ContentView()
28     }
29 }
30 }
```



The screenshot shows a portion of a Swift code editor. A context menu is open over a line of code. The menu has a search bar at the top labeled 'Actions'. Below the search bar are several options: 'Jump to Definition', 'Show Quick Help', 'Callers...', 'Edit All in Scope', 'Show SwiftUI Inspector...', and 'Extract Subview'. The 'Extract Subview' option is highlighted with a blue background and white text. The rest of the menu items have a standard grey background.

Figure 10. Extract subview

Xcode extracts the code block and create a default struct named `ExtractedView`. Type `HeaderView` to give it a better name (see the figure below for details).

```
11 struct ContentView: View {
12     var body: some View {
13         VStack {
14             HeaderView()
15         }
16     }
17 }
18
19 struct ContentView_Previews: PreviewProvider {
20     static var previews: some View {
21         ContentView()
22     }
23 }
24
25 struct HeaderView: View {
26     var body: some View {
27         VStack(alignment: .leading, spacing: 2) {
28             Text("Choose")
29                 .font(.system(.largeTitle, design: .rounded))
30                 .fontWeight(.black)
31             Text("Your Plan")
32                 .font(.system(.largeTitle, design: .rounded))
33                 .fontWeight(.black)
34         }
35     }
36 }
37
```

Figure 11. Extract subview

The UI is now 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 create the UI for the *Basic* plan first. Now 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 just 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.

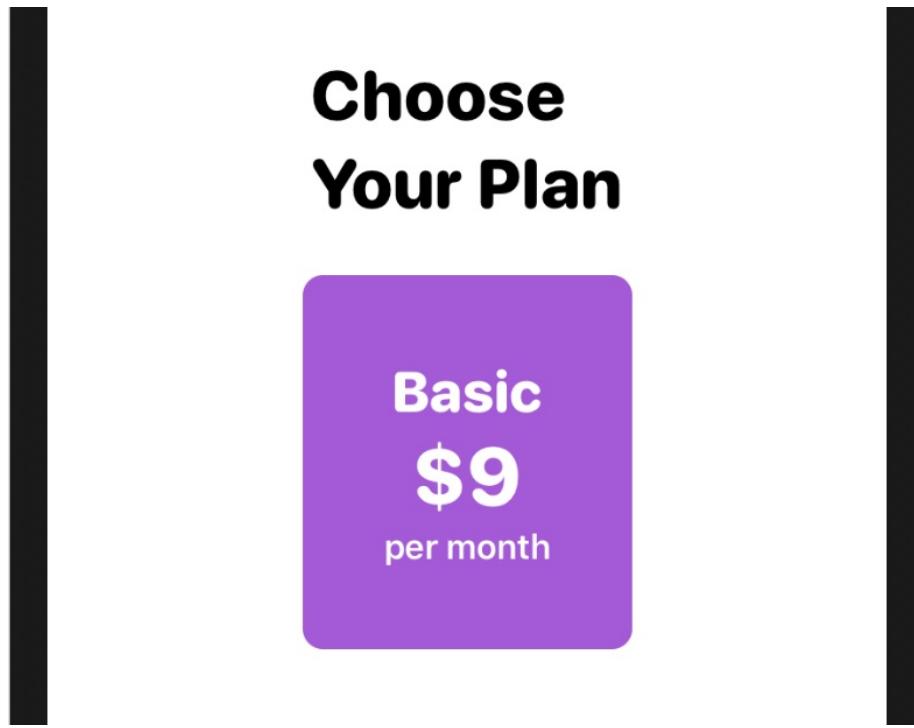


Figure 12. The Basic Plan

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`. Now hold the command key and click the `vstack` keyword. Choose *Embed in HStack*.

```
11 struct ContentView: View {  
12     var body: some View {  
13         VStack {  
14             HeaderView()  
15  
16             VStack {  
17                 Text("Basic")  
18             }  
19             .system(.title, design: .rounded))  
20             eight(.black)  
21             roundColor(.white)  
22             .system(size: 40, weight: .heavy, design: .rounded))  
23             roundColor(.white)  
24             month")  
25             headline)  
26             roundColor(.white)  
27             lor.purple)  
28             10)  
29             ]? Make Optional  
30             Make Conditional  
31             Repeat  
32             Embed in List  
33             Embed in Group  
34             Embed in Button  
35             Embed in HStack  
36             Embed in VStack  
37             Jump to Definition  
38             Show Quick Help  
39             Callers...  
40             Edit All in Scope  
41             Show SwiftUI Inspector...  
42             Extract Subview  
43             Actions
```

Figure 13. The Basic Plan

Xcode should create the code of `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 the background color and text color. 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.

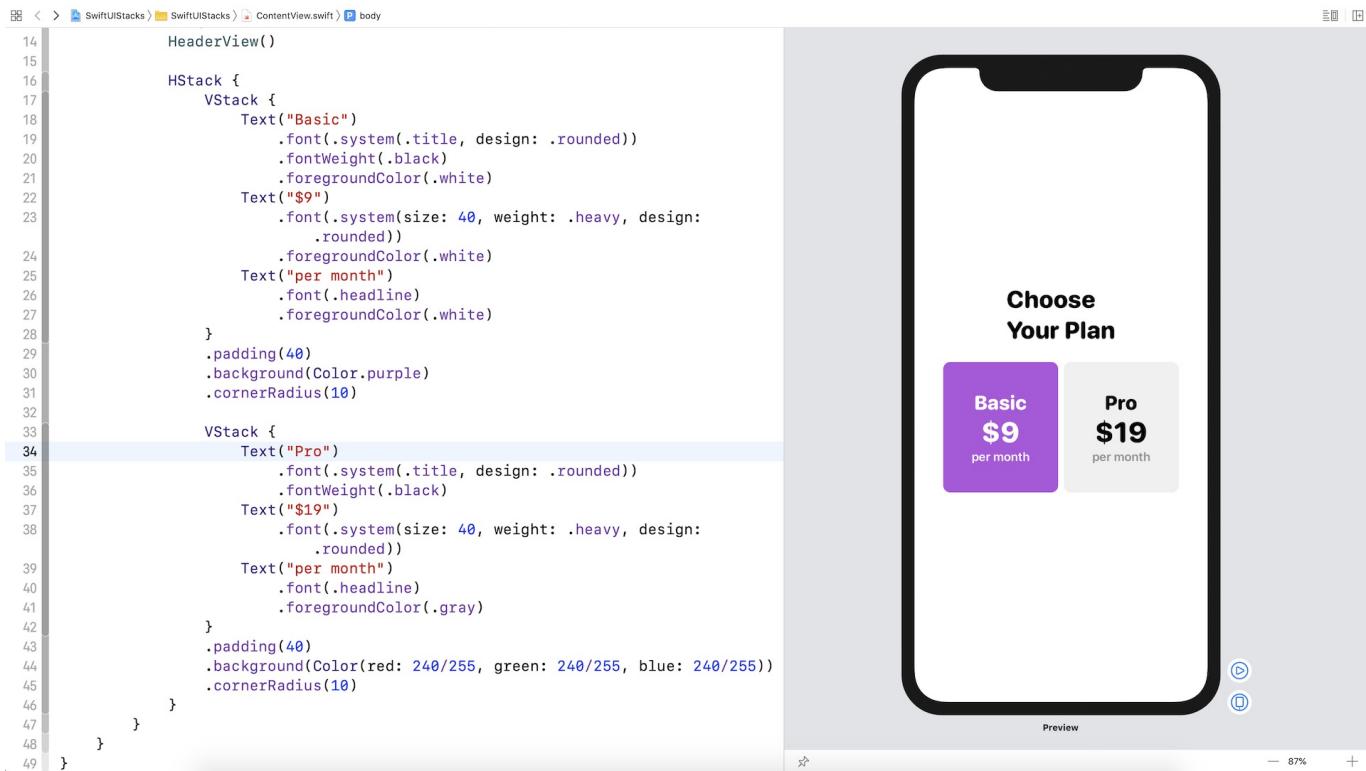


Figure 14. Using HStack to layout two views horizontally

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 "Premium". 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.

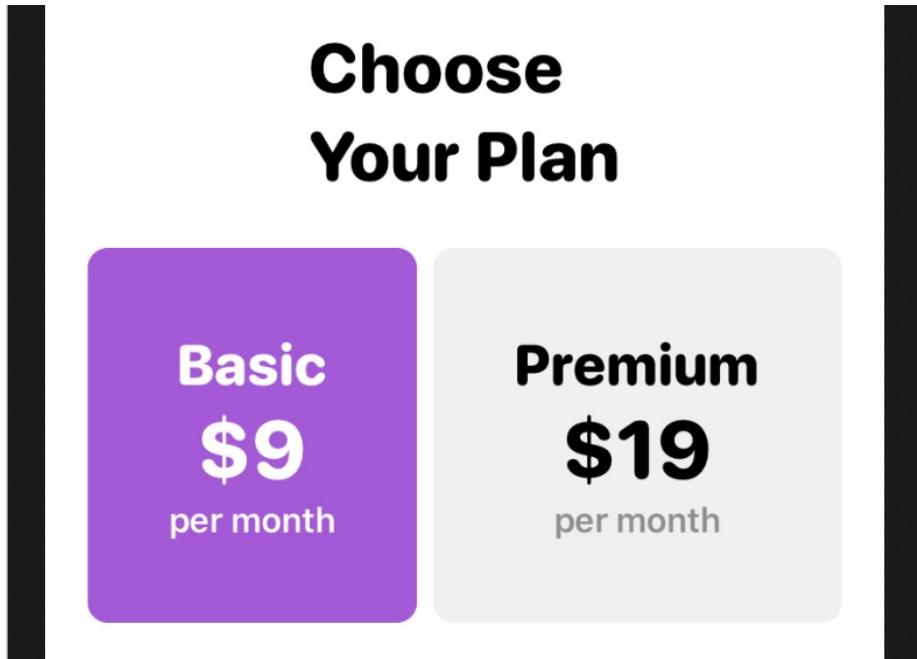


Figure 15. The size of the Pro block becomes wider

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 to 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. Say, for example, if there is only one pricing block, it will take up the whole screen width.

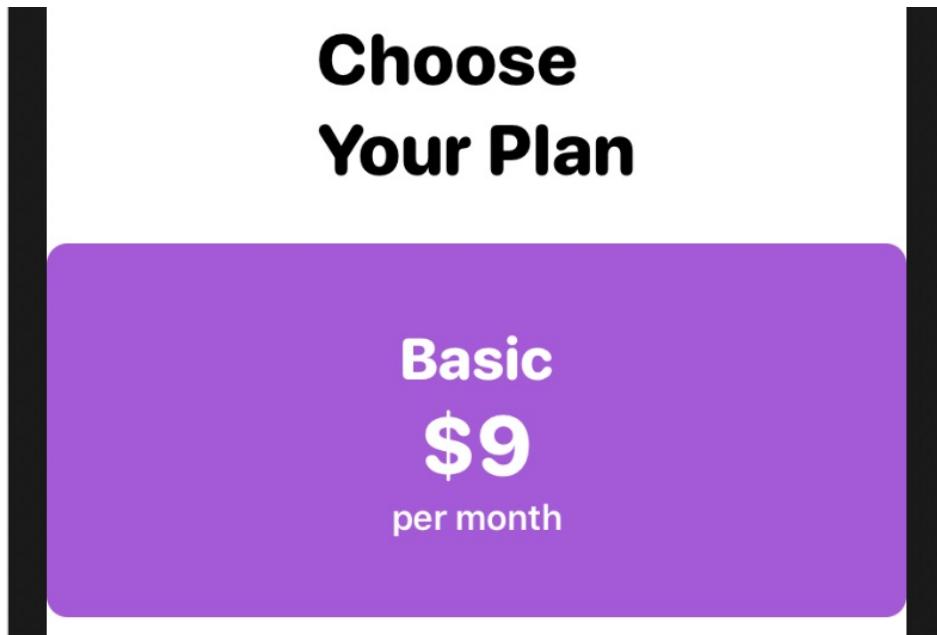


Figure 16. Setting the `maxWidth` to `.infinity`

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

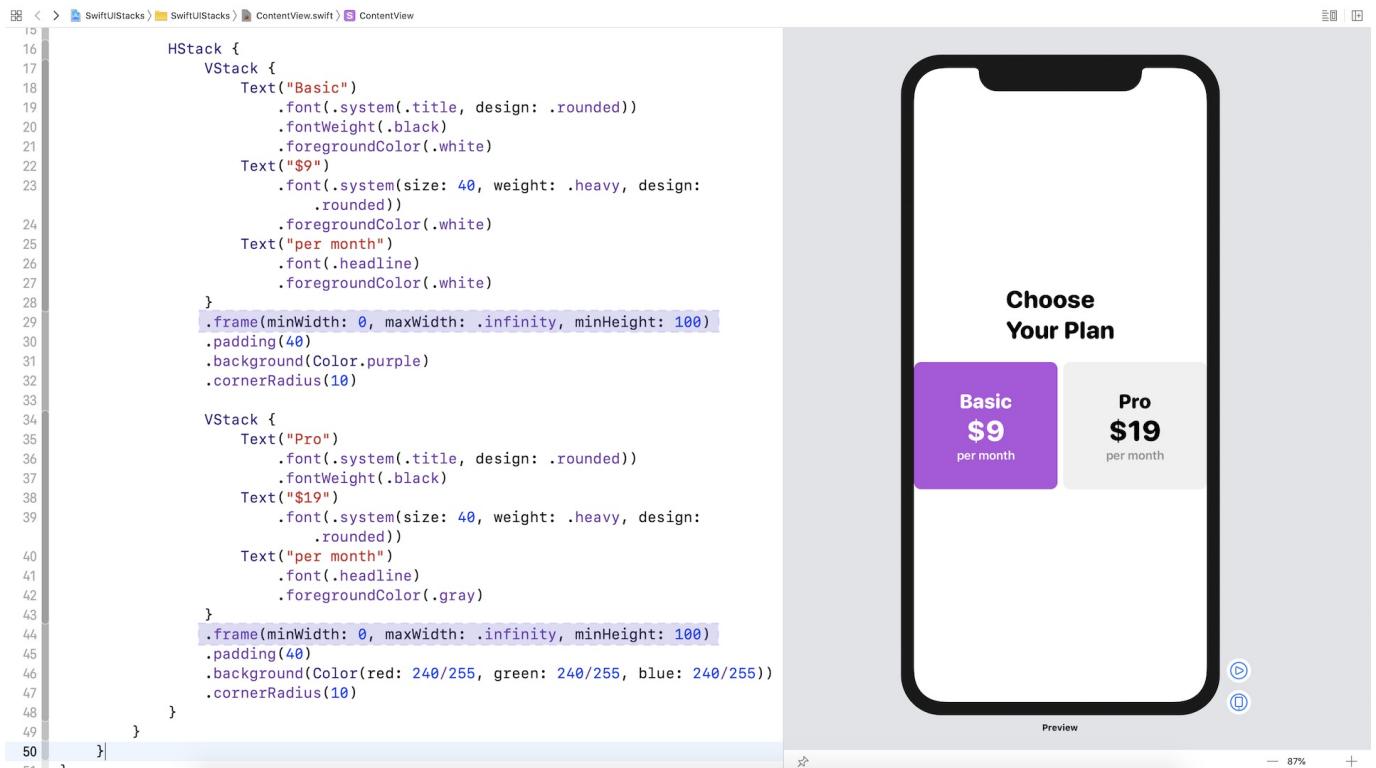


Figure 17. Arranging both pricing blocks with equal width

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

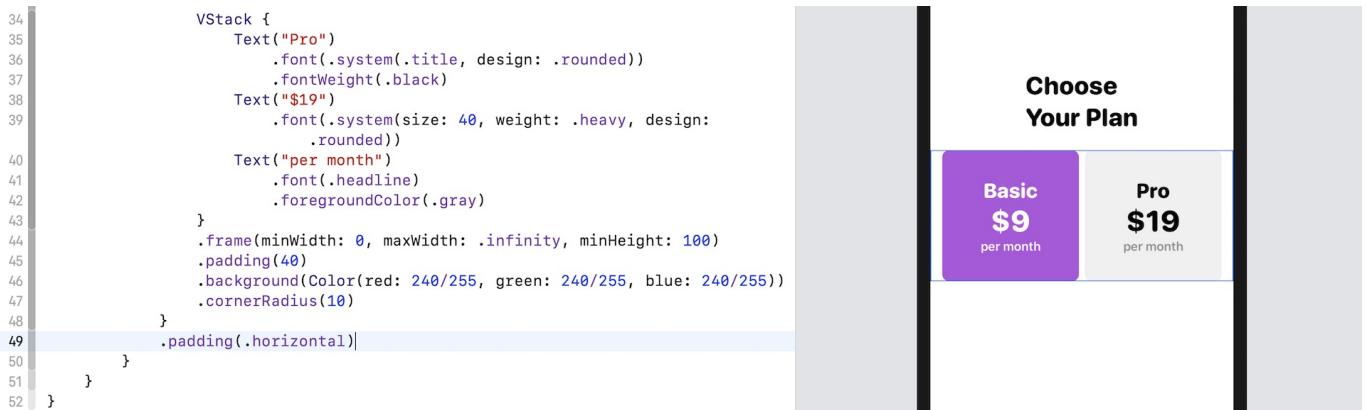


Figure 18. Arranging both pricing blocks with equal width

The `.horizontal` parameter means we only want to add some paddings 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.

Let's see how we can do it.

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`.

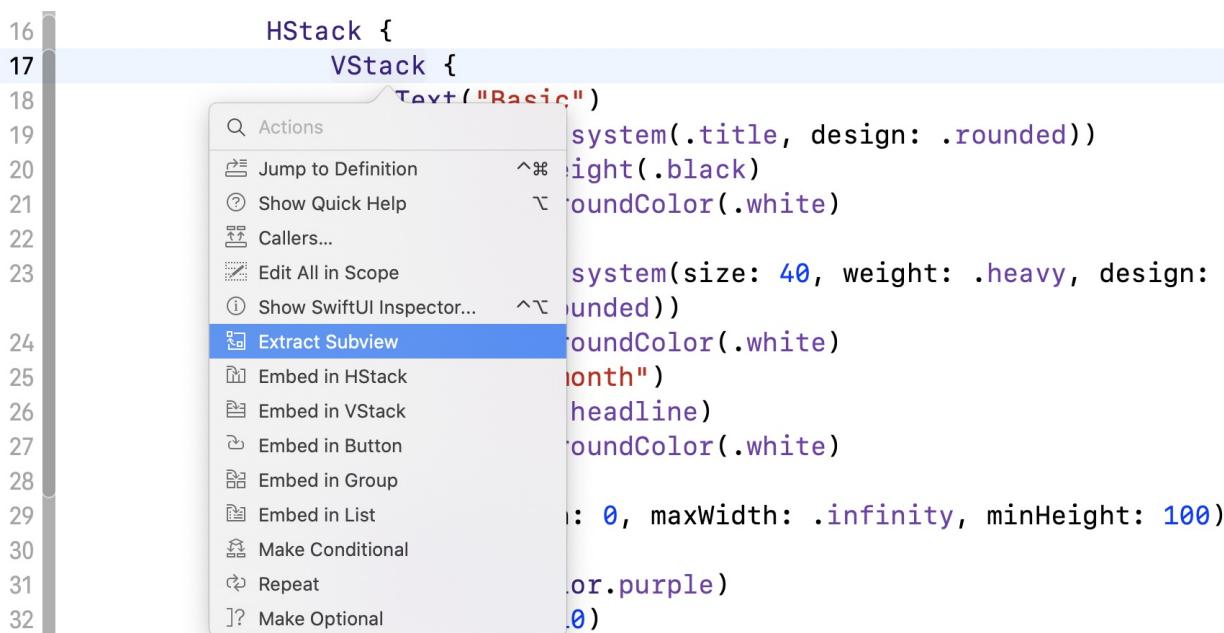


Figure 19. Extracting the subview

As we mentioned earlier, the `PricingView` should be flexible to display different pricing plans. We will add four variables in the `PricingView` struct. Now 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)
    }
}
```

Here, we added the 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 made the changes, you'll see an error. While Xcode indicates an error on the `Text` view, the root cause is due to the change of `PricingView`.

```
15
16     HStack {
17         PricingView()
18
19         VStack {
20             Text("Pro")  '(LocalizedStringKey) -> Text' is not convertible to '(Localized...
21                 .font(.system(.title, design: .rounded))
22                 .fontWeight(.black)
23             Text("$19")
24                 .font(.system(size: 40, weight: .heavy, design:
25                     .rounded))
26             Text("per month")
27                 .font(.headline)
28                 .foregroundColor(.gray)
}
```

Figure 20. Extracting the subview

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

```
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 much cleaner and easier to read.

The screenshot shows the Xcode interface with the ContentView.swift file open. The code defines a ContentView struct with a body property containing a VStack. This stack contains a HeaderView and an HStack with two PricingView components. The first PricingView has a title of "Basic", a price of "\$9", and a purple background. The second PricingView has a title of "Pro", a price of "\$19", and a light gray background. Both views have white text and are separated by a horizontal padding. Below the preview window, there are zoom controls (minus, plus, 87%) and a preview status bar.

```
1 //  
2 // ContentView.swift  
3 // SwiftUIStacks  
4 //  
5 // Created by Simon Ng on 27/8/2019.  
6 // Copyright © 2019 AppCoda. All rights reserved.  
7 //  
8  
9 import SwiftUI  
10  
11 struct ContentView: View {  
12     var body: some View {  
13         VStack {  
14             HeaderView()  
15  
16             HStack {  
17                 PricingView(title: "Basic", price: "$9", textColor: .white,  
18                               bgColor: .purple)  
19  
20                 PricingView(title: "Pro", price: "$19", textColor: .black,  
21                               bgColor: Color(red: 240/255, green: 240/255, blue:  
22                               240/255))  
23             }.padding(.horizontal)  
24         }  
25     }  
26  
27     struct ContentView_Previews: PreviewProvider {  
28         static var previews: some View {  
29             ContentView()  
30         }  
31     }  
32     struct HeaderView: View {  
33         var body: some View {  
34             VStack(alignment: .leading, spacing: 2) {  
35                 Text("Choose")  
36             }  
37         }  
38     }  
39  
40     struct PricingView: View {  
41         var title: String  
42         var price: String  
43         var textColor: Color  
44         var bgColor: Color  
45  
46         var body: some View {  
47             Text(title).font(.bold).color(textColor)  
48             Text(price).font(.bold).color(textColor)  
49             Text("per month").font(.normal).color(.gray)  
50         }  
51     }  
52 }
```

Figure 21. *ContentView* after refactoring the code

Using ZStack

Now that you've laid out the pricing blocks and refactor the code, but there is still one thing missing for the *Pro* pricing. In the design, 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.

Now embed the `PricingView` of the *Pro* plan with `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` determines how the views are overplayed 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.

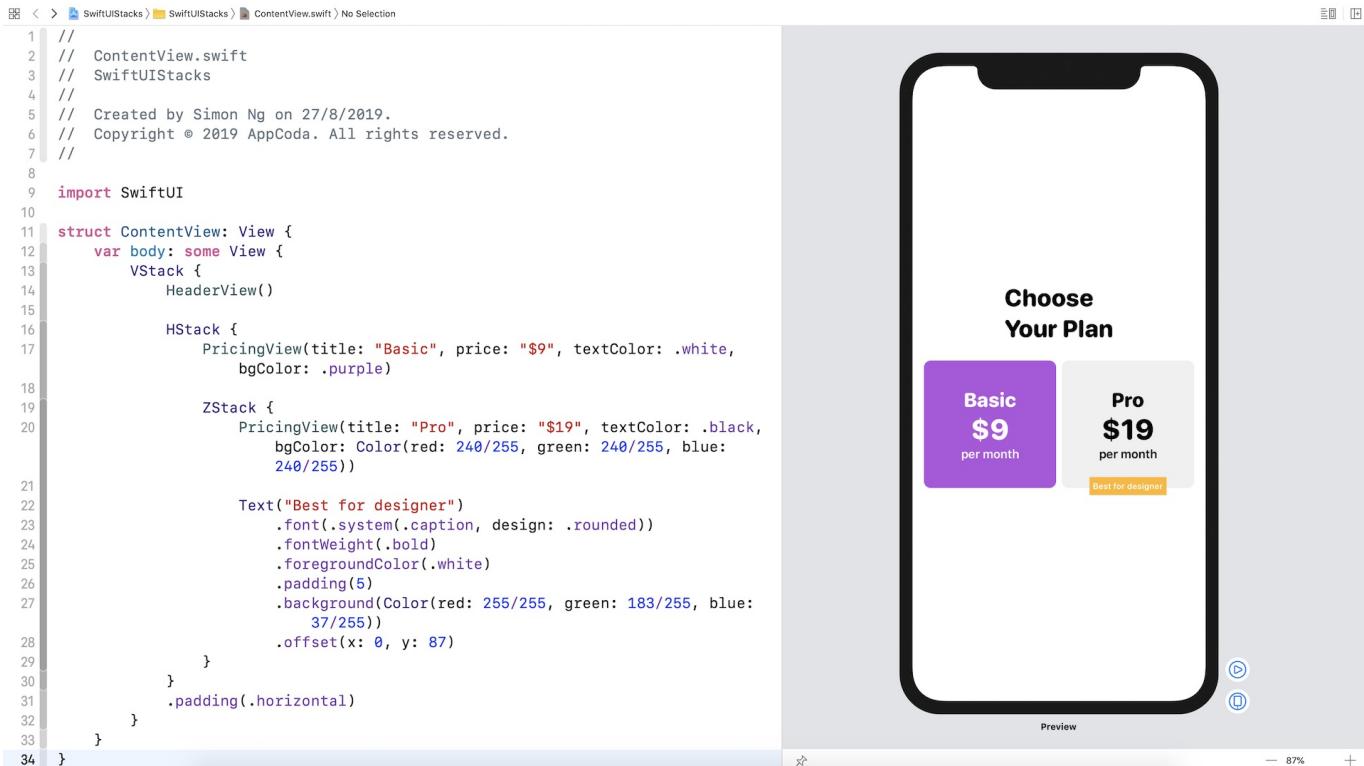


Figure 23. Position the text view using `.offset`

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

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

Exercise #1

We haven't finished yet. I still want to discuss with you about 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.

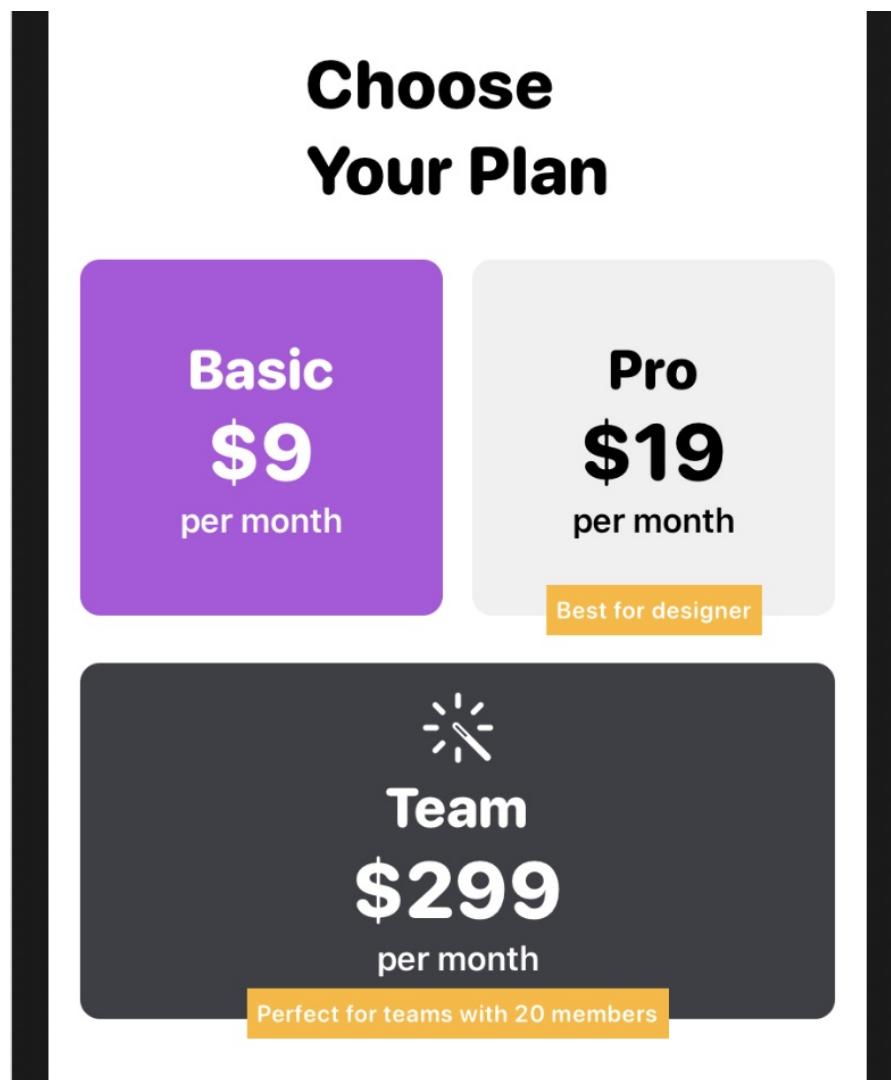


Figure 24. Adding the Team plan

Please don't look at the solution yet and develop your own solution.

Handling Optionals in SwiftUI

Have you tried out the exercise and come up with your own solution? The layout of the *Team* plan is very similar to the *Basic & Pro* plans. You may replicate the `vStack` of these two plans and create the *Team* plan. But 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 this need. Since the icon is not mandatory for a pricing plan, declare an optional in `PricingView`:

```
var icon: String?
```

If you're new to Swift, an optional means that the variable may have a value or no value. Here we define a variable named `icon` of the type `String`. It's expected the caller 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 optional in SwiftUI? In Swift, you usually use `if let` to check if an optional has a value and unwrap it. However, you can't use `if let` in SwiftUI. You may give it a try but it'll end up with the following error:

```
76     var body: some View {
77         VStack {
78             if let icon = icon { ❶ Closure containing control flow statement cannot be used...
79
80                 Image(systemName: icon)
81                     .font(.largeTitle)
82                     .foregroundColor(textColor)
83
84 }
```

Figure 25. Using `if let` in SwiftUI

One way to work with optionals in SwiftUI is to check if the optional has a non-nil value. Say, 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)  
  
}
```

This works but a more elegant way to unwrap the optional is by using `map()`. The code can be rewritten like this:

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

The final code of `PricingView` should be updated like below to support the rendering of an icon:

```

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 made this change, you can then create the *Team* plan by using `zStack` and `PricingView` like this:

```

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)
}

```

Using Spacer

When comparing your current UI with that in 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 layout.

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 first item, 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` with a `HStack`. By using a `Spacer`, this will push the `vStack` to the left. Figure 26 illustrates how the spacer works.

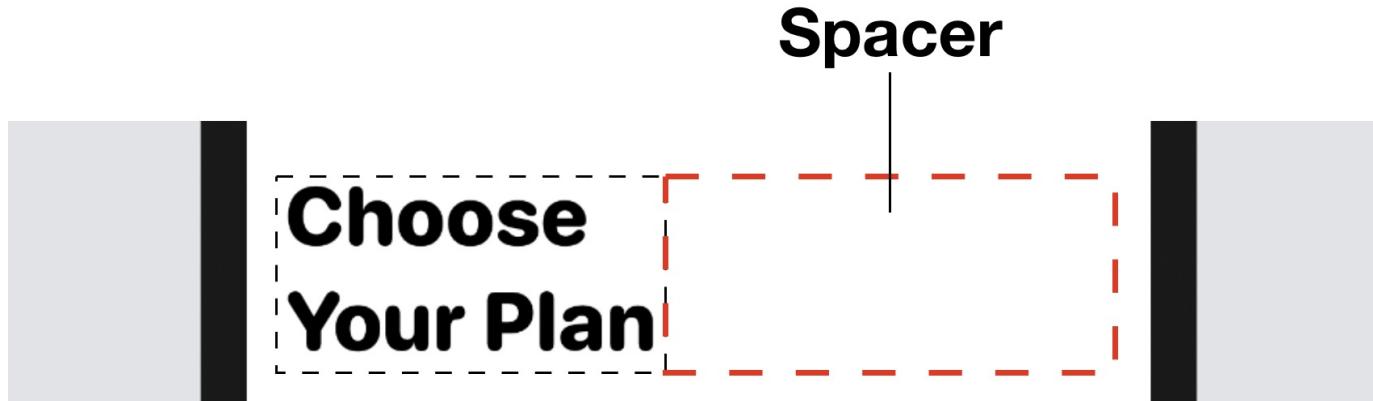


Figure 26. Using Spacer in HStack

Now you may know how to fix the second issue. 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()
        }
    }
}
```

Again, here is the figure that shows you how the spacer works visually.

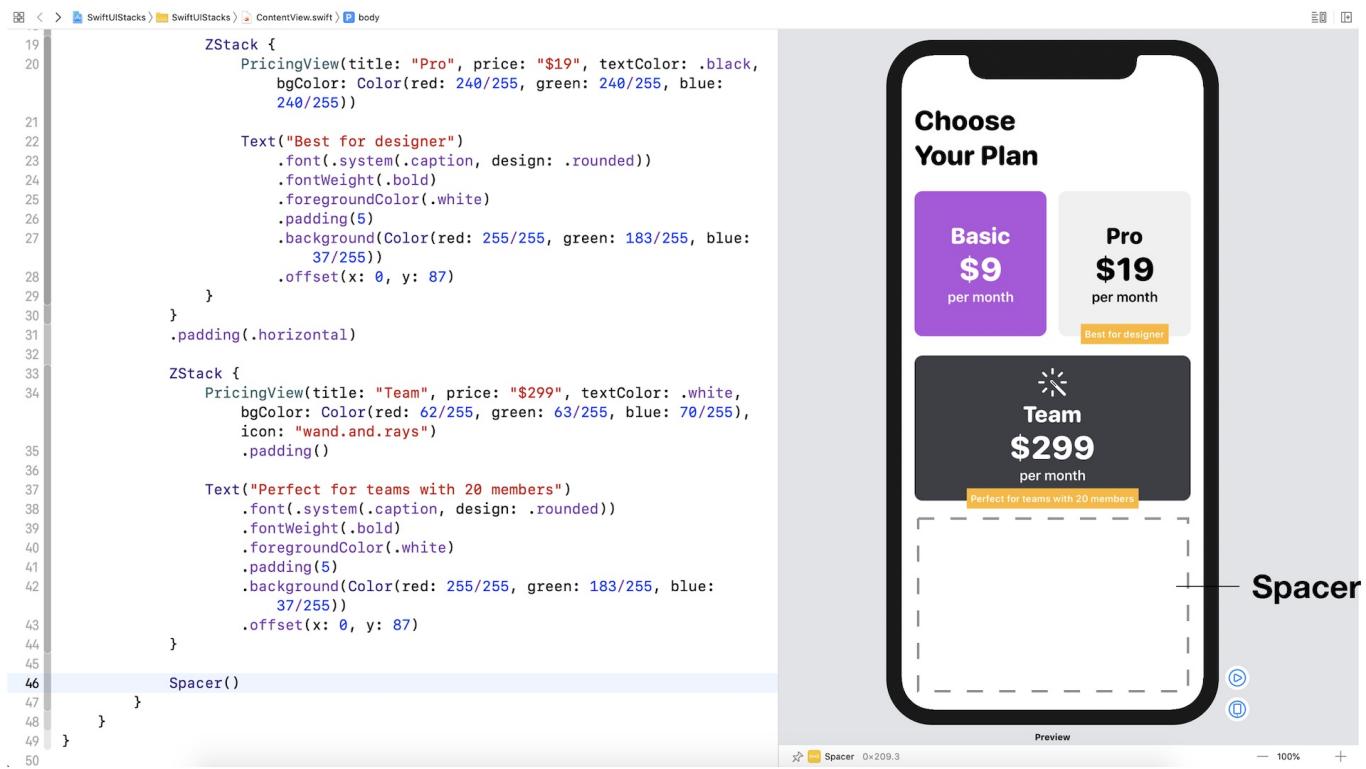


Figure 27. Using spacer in VStack

Exercise #2

Now that you should 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 the system image from SF Symbols. You're free to pick any of the images instead of following mine.

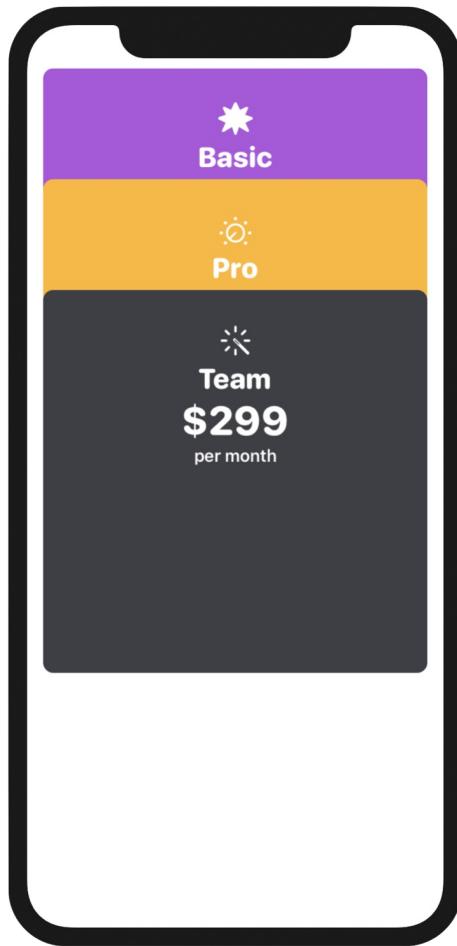


Figure 28. Your exercise

For reference, you can download the complete projects here:

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

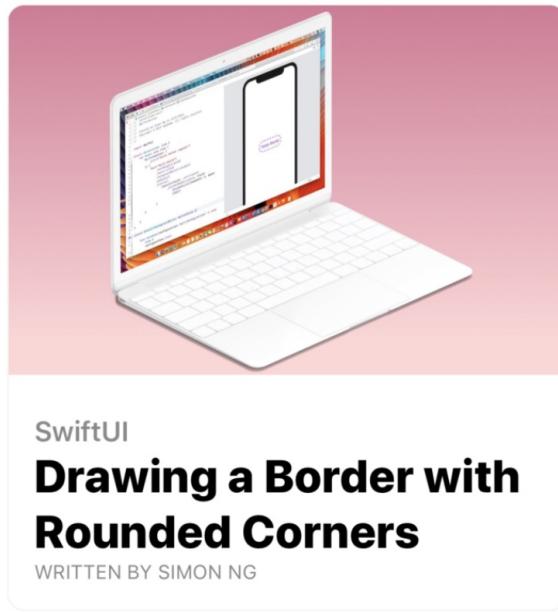


Figure 1. The card view

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 do take some time to work on the exercise and figure out your own solution.

Once you manage to 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.

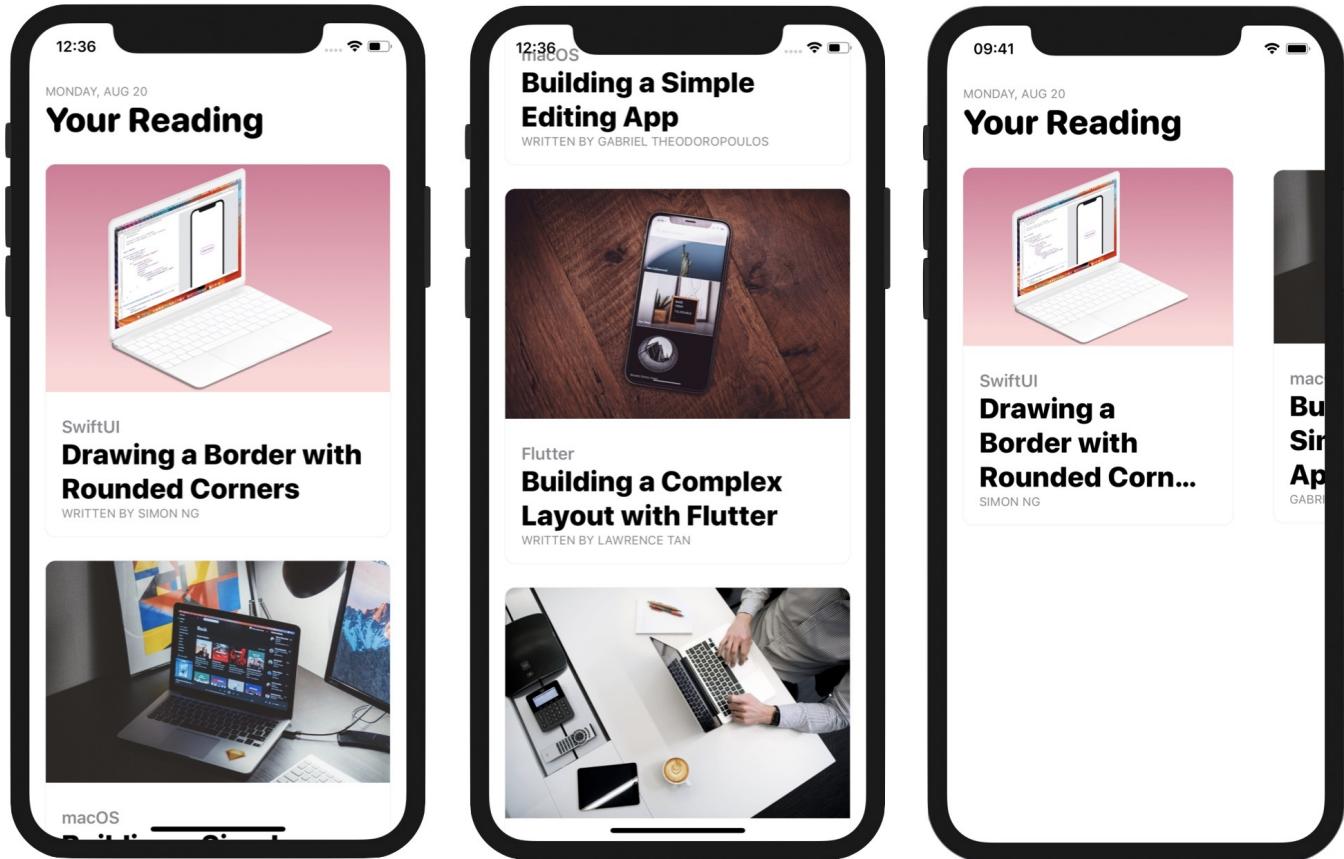


Figure 2. Building a scrollable UI with `ScrollView`

Creating a Card-like UI

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

So far, we code the user interface in the `ContentView.swift` file. It's very likely you write the 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 for it. In the project navigator, right click `SwiftUIScrollView` and choose *New File....*

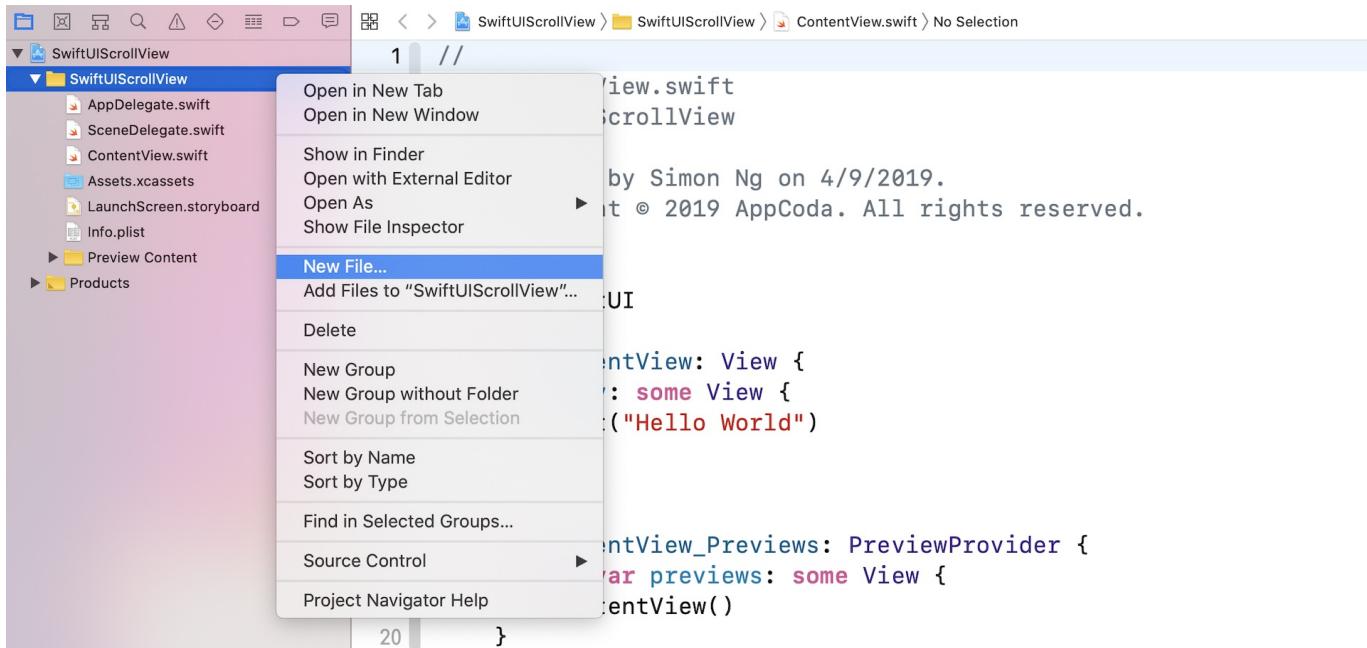


Figure 3. Creating a new file

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.

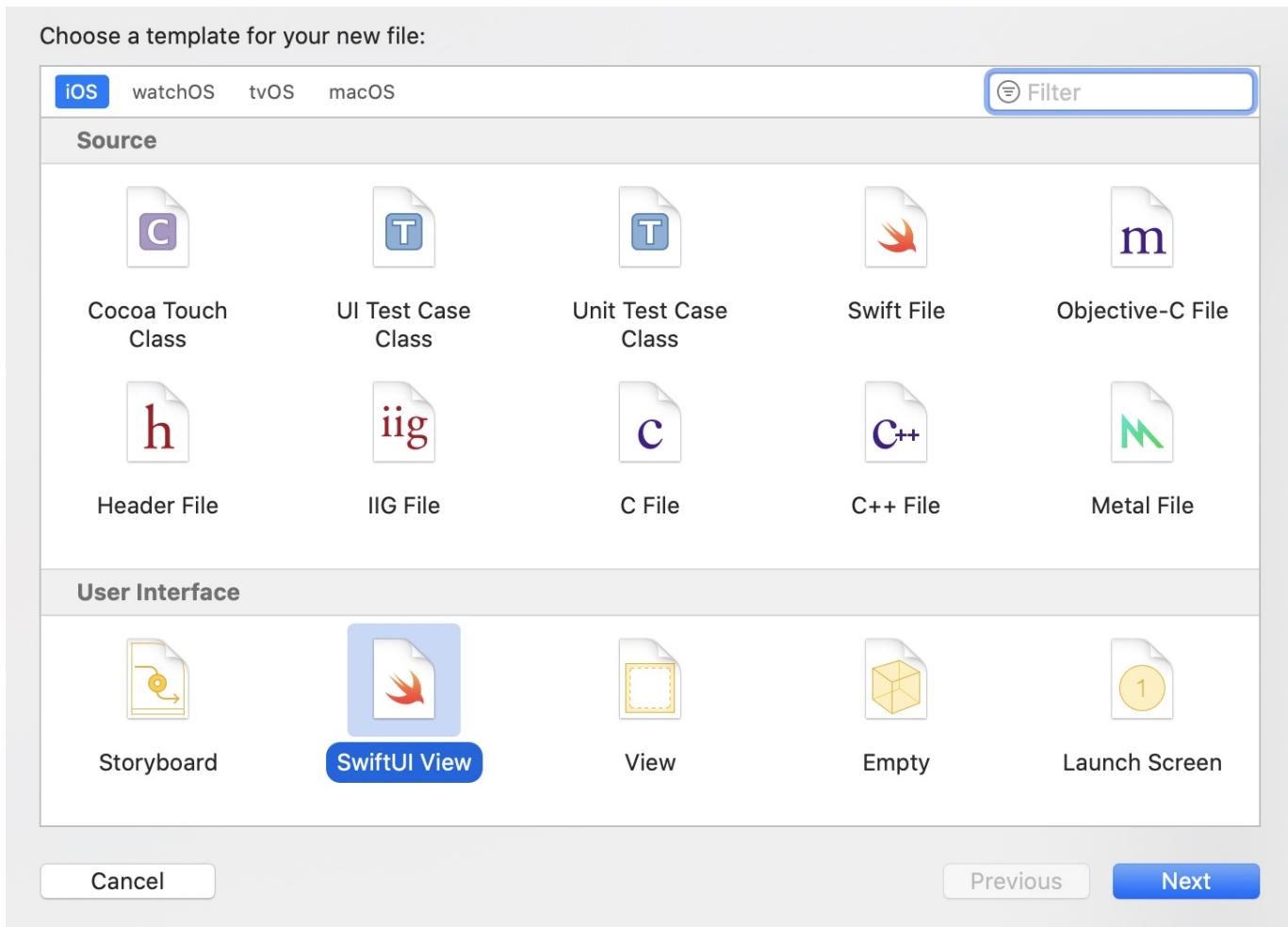


Figure 4. Choose the SwiftUI View template

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

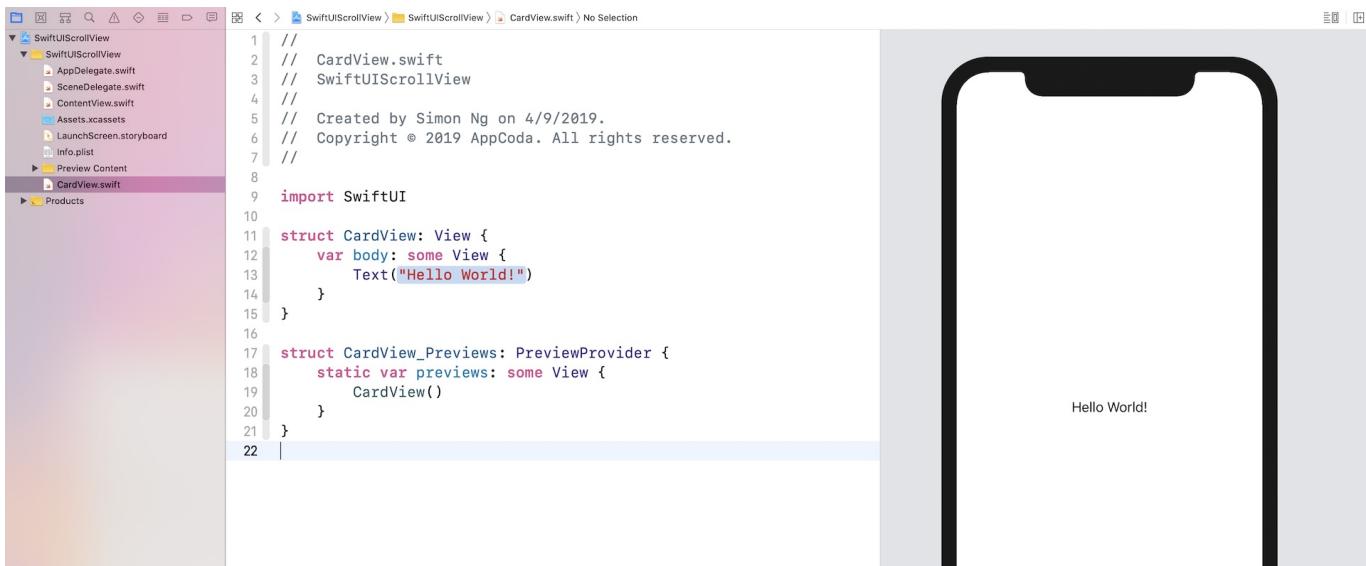


Figure 5. Just like ContentView.swift, you can preview CardView.swift in the canvas

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.

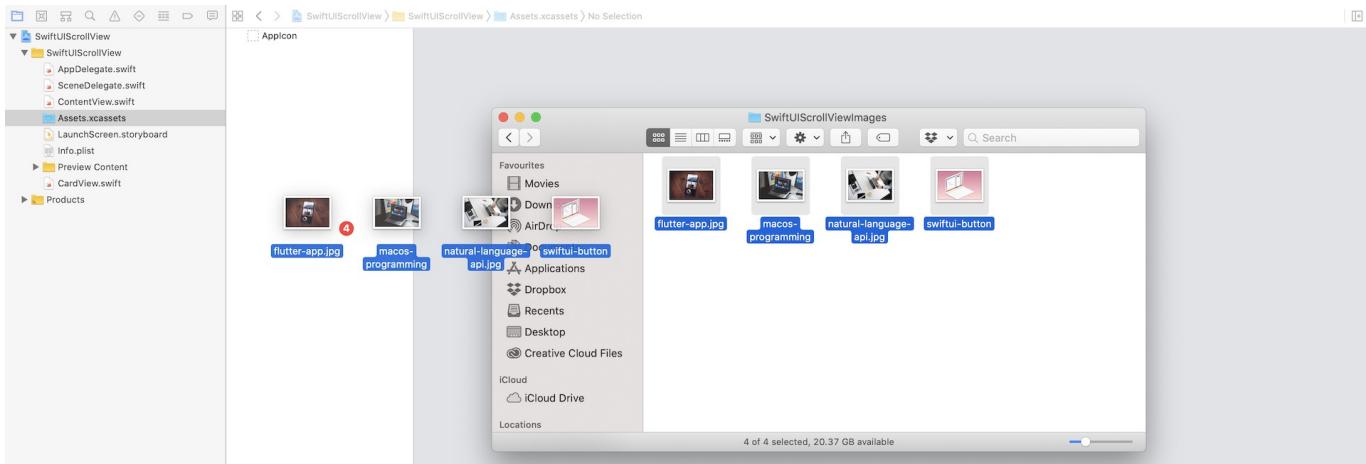


Figure 6. Adding the image files to the asset catalog

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 but retain the aspect ratio. You can 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 are about, go back and read the chapter about the `Image` object. Next, let's implement the text description. You may write the code like this:

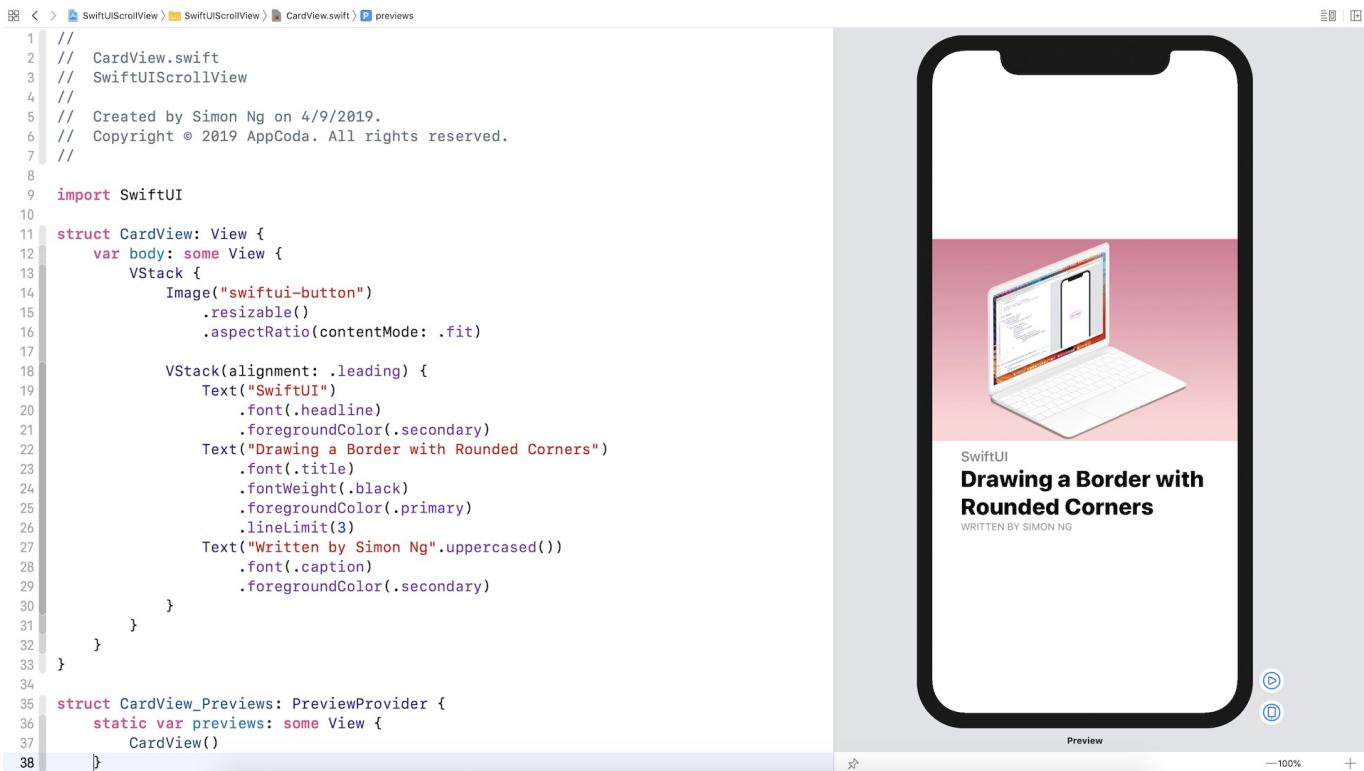
```
 VStack(alignment: .leading) {
    Text("SwiftUI")
        .font(.headline)
        .foregroundColor(.secondary)
    Text("Drawing a Border with Rounded Corners")
        .font(.title)
        .fontWeight(.black)
        .foregroundColor(.primary)
        .lineLimit(3)
    Text("Written by Simon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

Obviously, 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 thing 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 color that contains primary, secondary, and tertiary variants. By using this 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 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.



The screenshot shows the Xcode interface with the CardView.swift file open. The code defines a `CardView` struct that contains an image and a `VStack` of text views. The `VStack` has a leading alignment and contains several `Text` components with different styles. To the right of the code editor is a preview window showing a white iPhone displaying the card view. The card features a white background with rounded corners, a white border around the text area, and a white keyboard icon at the bottom. The text includes the word "SwiftUI" and the title "Drawing a Border with Rounded Corners".

```
1 //  
2 //  CardView.swift  
3 //  SwiftUI ScrollView  
4 //  
5 //  Created by Simon Ng on 4/9/2019.  
6 //  Copyright © 2019 AppCoda. All rights reserved.  
7 //  
8  
9 import SwiftUI  
10  
11 struct CardView: View {  
12     var body: some View {  
13         VStack {  
14             Image("swiftui-button")  
15                 .resizable()  
16                 .aspectRatio(contentMode: .fit)  
17  
18             VStack(alignment: .leading) {  
19                 Text("SwiftUI")  
20                     .font(.headline)  
21                     .foregroundColor(.secondary)  
22                 Text("Drawing a Border with Rounded Corners")  
23                     .font(.title)  
24                     .fontWeight(.black)  
25                     .foregroundColor(.primary)  
26                     .lineLimit(3)  
27                 Text("Written by Simon Ng".uppercased())  
28                     .font(.caption)  
29                     .foregroundColor(.secondary)  
30             }  
31         }  
32     }  
33 }  
34  
35 struct CardView_Previews: PreviewProvider {  
36     static var previews: some View {  
37         CardView()  
38     }  
}
```

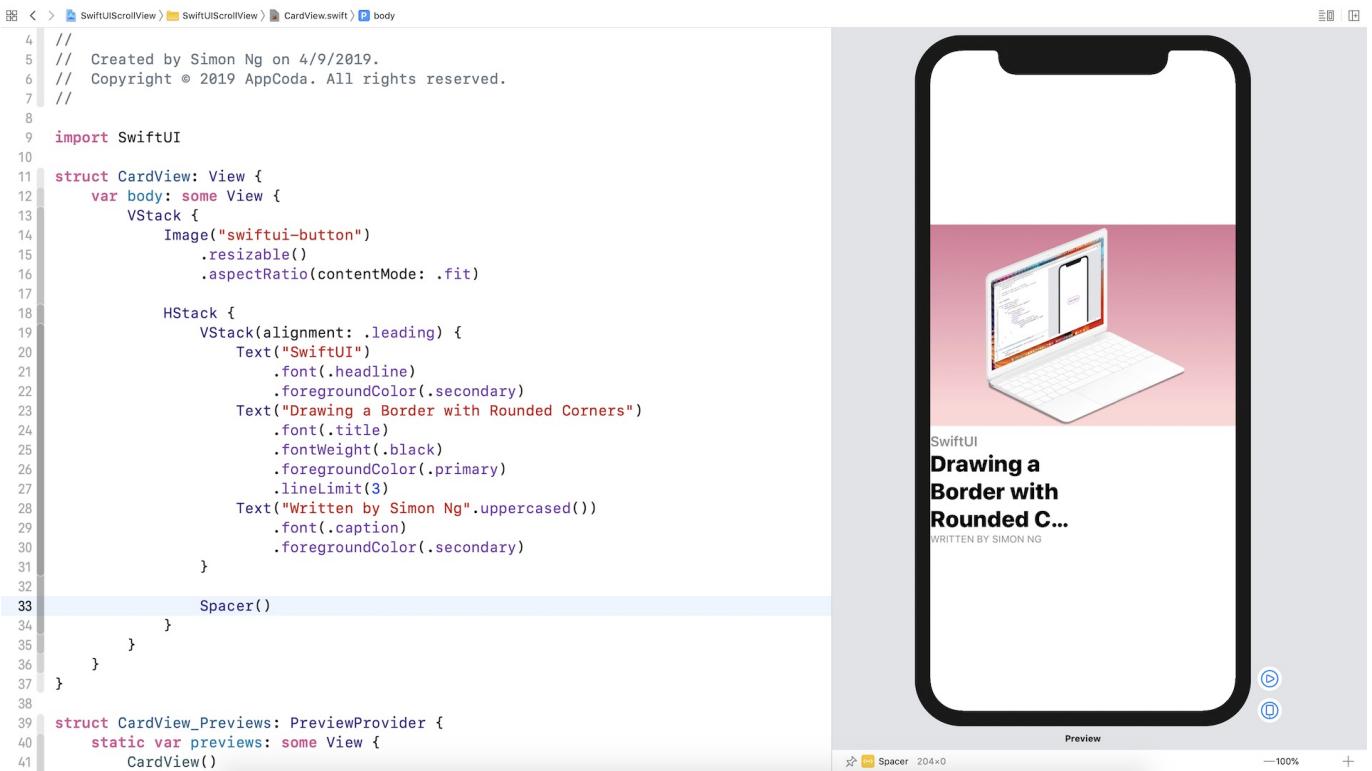
Figure 7. Embed the image and text views in a `VStack`

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

How do you do that?

Base 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 the one shown in figure 8, the `vstack` of the text views are aligned to the left of the screen. However, the heading is truncated.



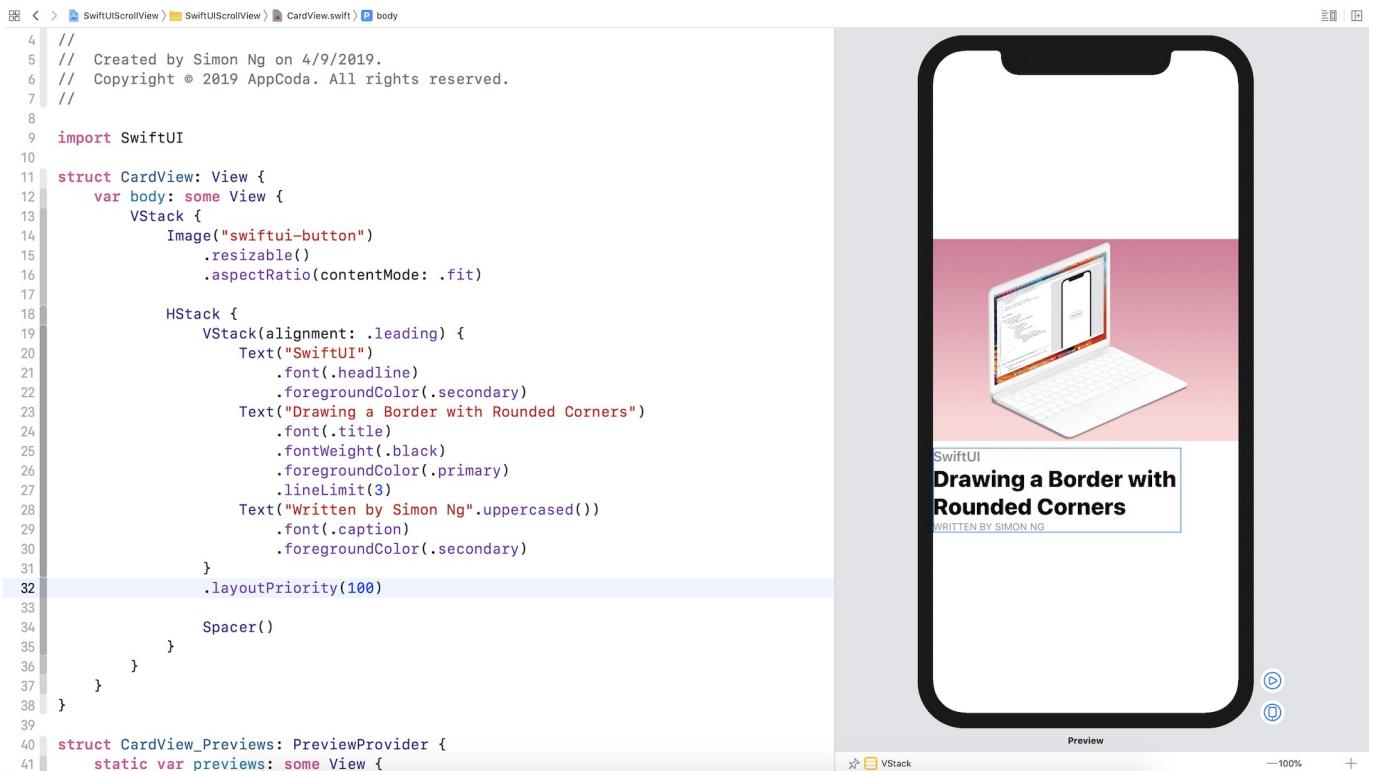
The screenshot shows the Xcode interface with the Swift code for a CardView. The code defines a struct CardView with a body property. Inside the body, there's a VStack containing an image and an HStack. The HStack contains two Text views: one for the title "SwiftUI" and one for the description "Drawing a Border with Rounded Corners". Both text views have specific font styles and colors. Below the HStack is a Spacer. The preview window on the right shows a white card with rounded corners. On the card, there's a small image of a laptop displaying code, followed by the title "SwiftUI" and the description "Drawing a Border with Rounded C...". At the bottom of the card, it says "WRITTEN BY SIMON NG".

```
4 //  
5 // Created by Simon Ng on 4/9/2019.  
6 // Copyright © 2019 AppCoda. All rights reserved.  
7 //  
8  
9 import SwiftUI  
10  
11 struct CardView: View {  
12     var body: some View {  
13         VStack {  
14             Image("swiftui-button")  
15                 .resizable()  
16                 .aspectRatio(contentMode: .fit)  
17  
18             HStack {  
19                 VStack(alignment: .leading) {  
20                     Text("SwiftUI")  
21                         .font(.headline)  
22                         .foregroundColor(.secondary)  
23                     Text("Drawing a Border with Rounded Corners")  
24                         .font(.title)  
25                         .fontWeight(.black)  
26                         .foregroundColor(.primary)  
27                         .lineLimit(3)  
28                     Text("Written by Simon Ng".uppercased())  
29                         .font(.caption)  
30                         .foregroundColor(.secondary)  
31                 }  
32  
33             Spacer()  
34         }  
35     }  
36 }  
37  
38 struct CardView_Previews: PreviewProvider {  
39     static var previews: some View {  
40         CardView()  
41     }  
42 }
```

Figure 8. Aligning the text description

Adjusting the Layout Priority

By default, both the text stack and the spacer occupy half of the parent view. This is why the heading couldn't be fully displayed. To fix the issue, you will need to adjust the layout priority of the text stack using the `layoutPriority` modifier. The larger the value the higher is the priority. This means if we set the layout priority of the `vstack` of the text views to a larger value, iOS will offer more space to fully render the text views before allocating the space to the `spacer`. Figure 9 shows you the result.



The screenshot shows the Xcode interface with the code editor on the left and a preview window on the right. The code editor displays Swift code for a `CardView` struct. The preview window shows a smartphone displaying a card with rounded corners containing text and an image.

```
4 //  
5 // Created by Simon Ng on 4/9/2019.  
6 // Copyright © 2019 AppCoda. All rights reserved.  
7 //  
8  
9 import SwiftUI  
10  
11 struct CardView: View {  
12     var body: some View {  
13         VStack {  
14             Image("swiftui-button")  
15                 .resizable()  
16                 .aspectRatio(contentMode: .fit)  
17  
18             HStack {  
19                 VStack(alignment: .leading) {  
20                     Text("SwiftUI")  
21                         .font(.headline)  
22                         .foregroundColor(.secondary)  
23                     Text("Drawing a Border with Rounded Corners")  
24                         .font(.title)  
25                         .fontWeight(.black)  
26                         .foregroundColor(.primary)  
27                         .lineLimit(3)  
28                     Text("Written by Simon Ng".uppercased())  
29                         .font(.caption)  
30                         .foregroundColor(.secondary)  
31                 }  
32                 .layoutPriority(100)  
33  
34             Spacer()  
35         }  
36     }  
37 }  
38  
39 struct CardView_Previews: PreviewProvider {  
40     static var previews: some View {  
41         CardView()
```

Figure 9. Adjusting the layout priority

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

```
HStack {  
    VStack(alignment: .leading) {  
        .  
        .  
        .  
    }  
    .layoutPriority(100)  
  
    Spacer()  
}  
.padding()
```

Lastly, it's the border. We have discussed how to draw a border with rounded corners in the earlier chapter. We can use the `overlay` modifier and draw the border using the `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)
                }
                .layoutPriority(100)

                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 paddings for the top, left, and right sides. Now you should have built the card view layout.

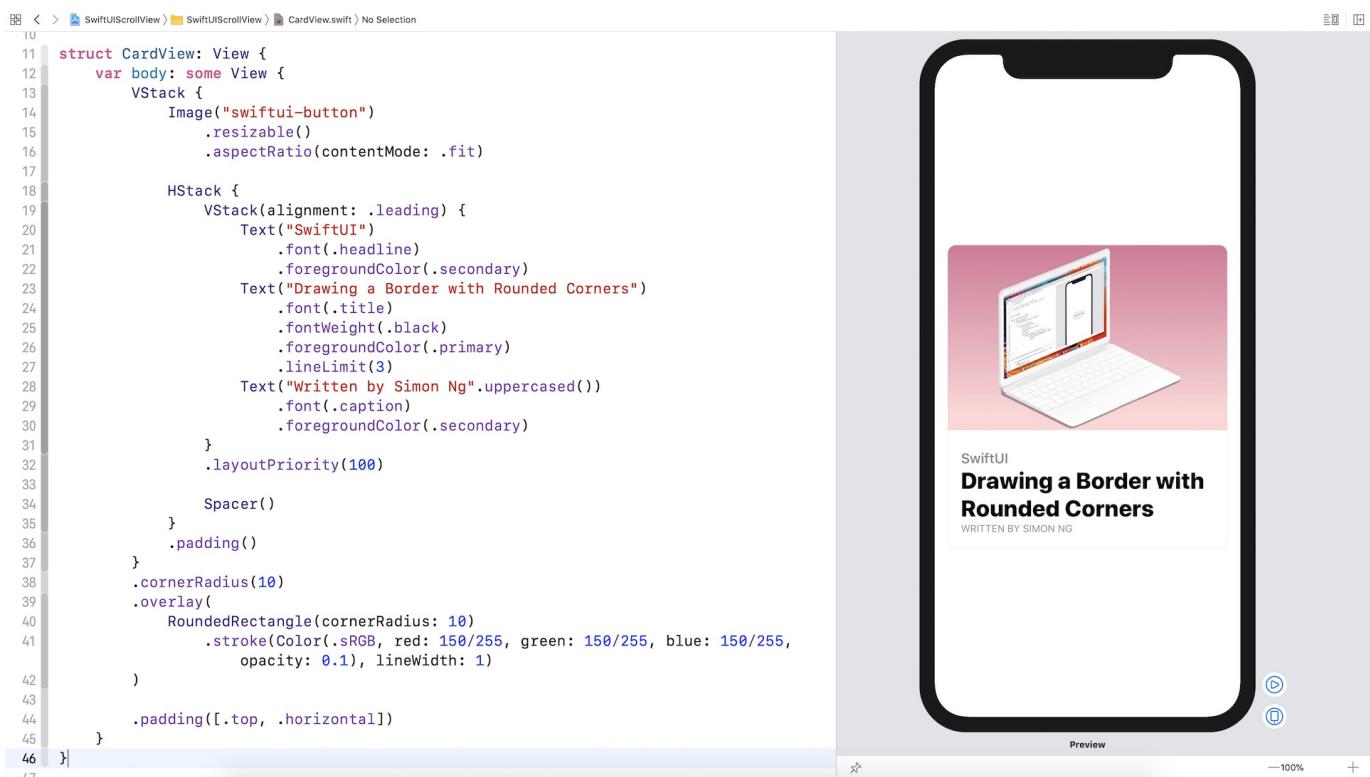


Figure 10. Adjusting the layout priority

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 these 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(author.uppercased())
                .font(.caption)
                .foregroundColor(.secondary)
        }
        .layoutPriority(100)

        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.

```

53
54 struct CardView_Previews: PreviewProvider {
55     static var previews: some View {
56         CardView() // Missing argument for parameter 'image' in call
57     }
58 }
59

```

Figure 11. Missing parameters when calling the CardView

So replace 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`. If you did that, these card views will be squeezed to fit the screen because `vstack` is non-scrollable.

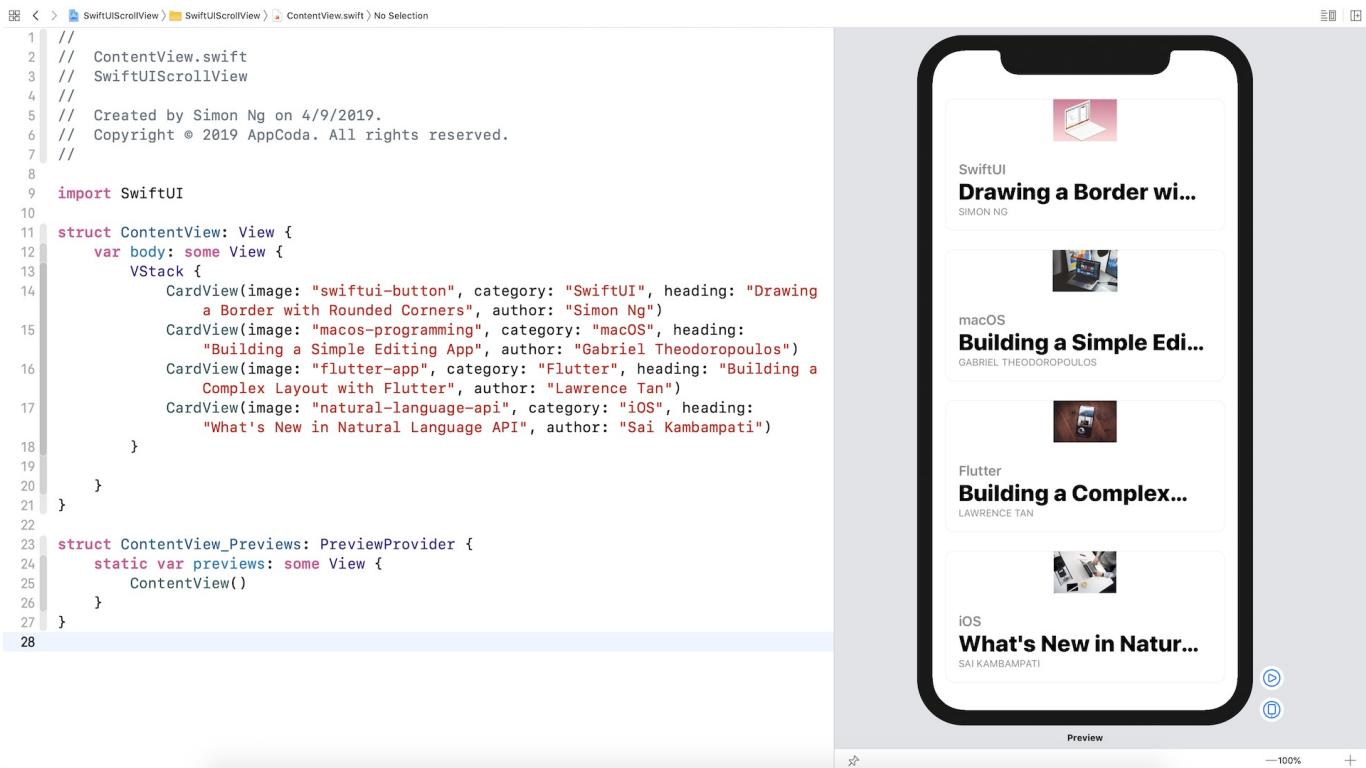
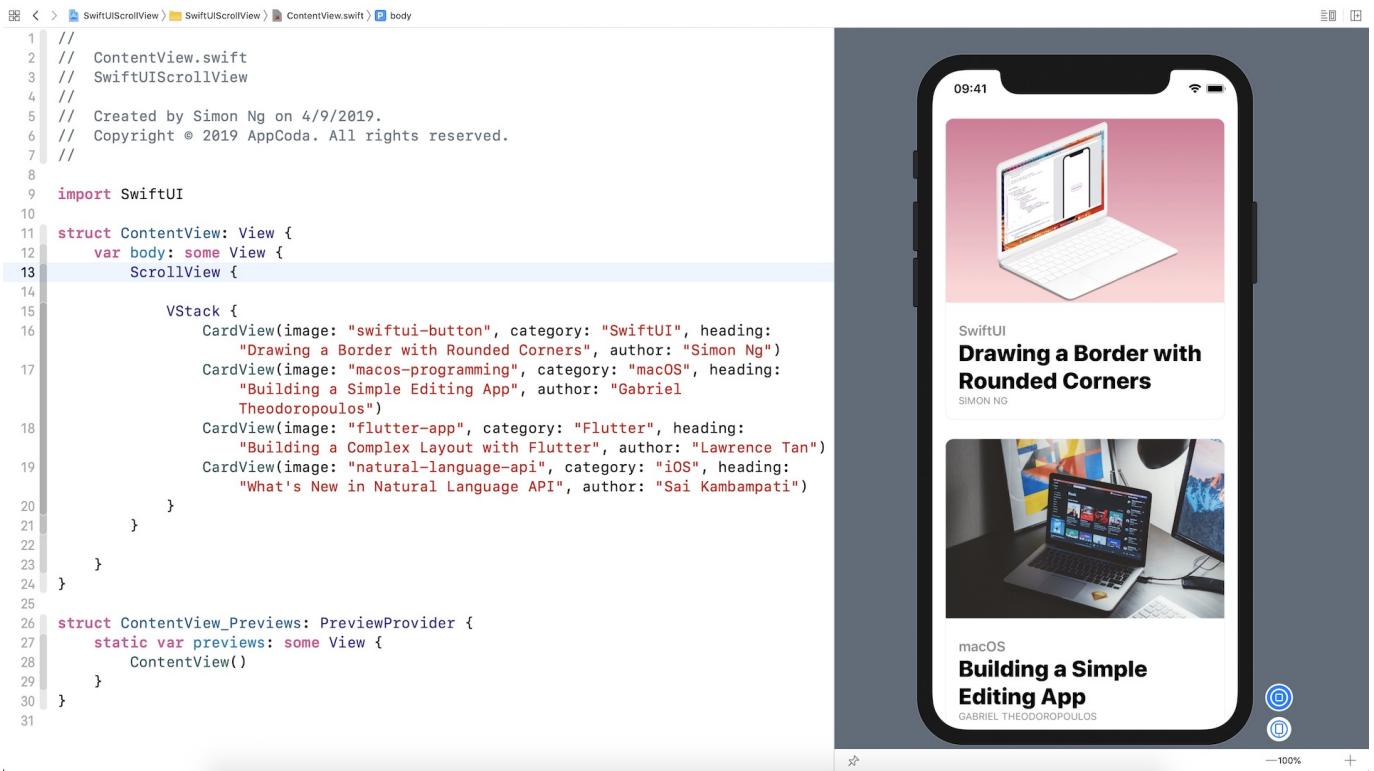


Figure 12. Embedding the card views in a VStack

To support scrollable content, SwiftUI provides a view called `ScrollView`. When the content is embedded in a `ScrollView`, it becomes scrollable. So, 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.



The screenshot shows the Xcode interface with the code editor on the left and a preview window on the right. The code editor displays `ContentView.swift` with the following content:

```
1 // ContentView.swift
2 // SwiftUIScrollView
3 // SwiftUIScrollView
4 //
5 // Created by Simon Ng on 4/9/2019.
6 // Copyright © 2019 AppCoda. All rights reserved.
7 //
8
9 import SwiftUI
10
11 struct ContentView: View {
12     var body: some View {
13         ScrollView {
14
15             VStack {
16                 CardView(image: "swiftui-button", category: "SwiftUI", heading:
17                     "Drawing a Border with Rounded Corners", author: "Simon Ng")
18                 CardView(image: "macos-programming", category: "macOS", heading:
19                     "Building a Simple Editing App", author: "Gabriel
20                     Theodoropoulos")
21                 CardView(image: "flutter-app", category: "Flutter", heading:
22                     "Building a Complex Layout with Flutter", author: "Lawrence Tan")
23                 CardView(image: "natural-language-api", category: "iOS", heading:
24                     "What's New in Natural Language API", author: "Sai Kambampati")
25             }
26         }
27     }
28 }
29
30 struct ContentView_Previews: PreviewProvider {
31     static var previews: some View {
32         ContentView()
33     }
34 }
```

The preview window shows a mobile application interface. At the top, there is a large image of a laptop displaying code. Below it, the title "SwiftUI" is followed by "Drawing a Border with Rounded Corners" by "SIMON NG". Further down, there is another section titled "macOS" with the heading "Building a Simple Editing App" by "GABRIEL THEODOROPoulos". The bottom of the screen has two circular icons.

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. As a hint, you will need to adjust the layout priority in order to render the views correctly.

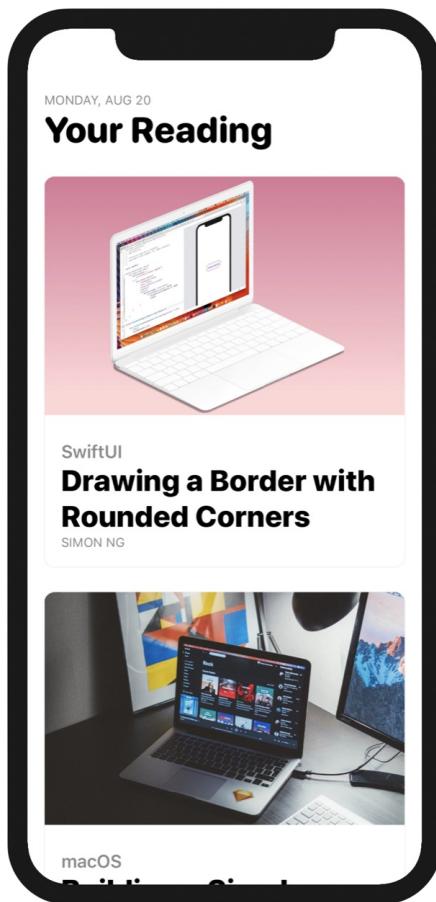


Figure 14. Exercise #1

Creating a Carousel UI with 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: "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)
            }
        }
    }
}

```

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.

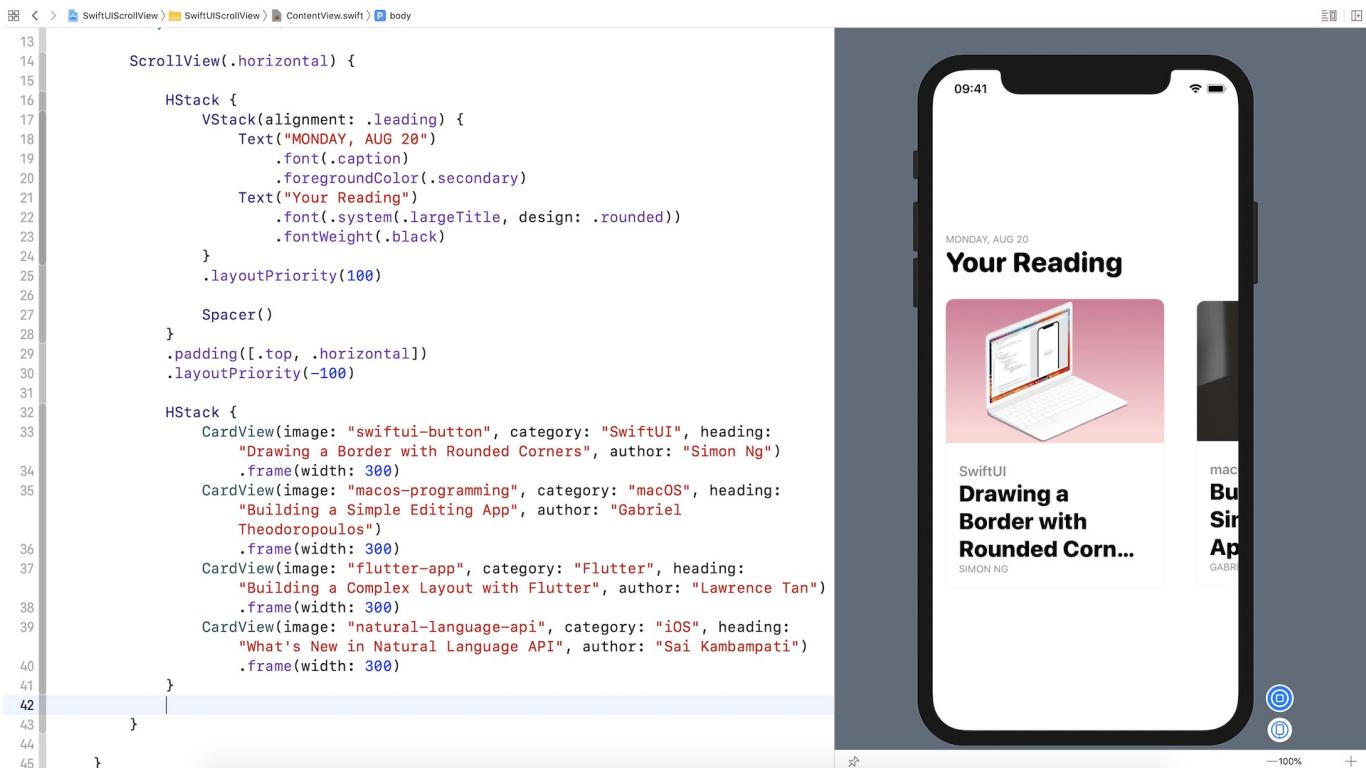


Figure 15. Carousel UI

Hiding the Scroll Indicator

While you're scrolling the views, 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` to the following code:

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

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

Exercise #2

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

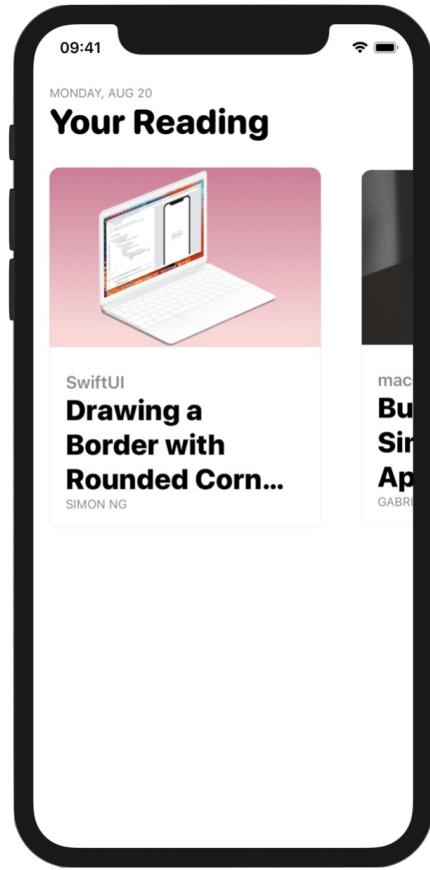


Figure 16. Aligning the views to the top

For reference, you can download the complete project here:

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

Chapter 6

Working with SwiftUI Buttons 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 with you this SwiftUI control and you're to 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 *Single View Application* template. Type the name of the project. I set it to *SwiftUIButton* but you're free to use any other name. All

you need to ensure is check the *Use SwiftUI* option.

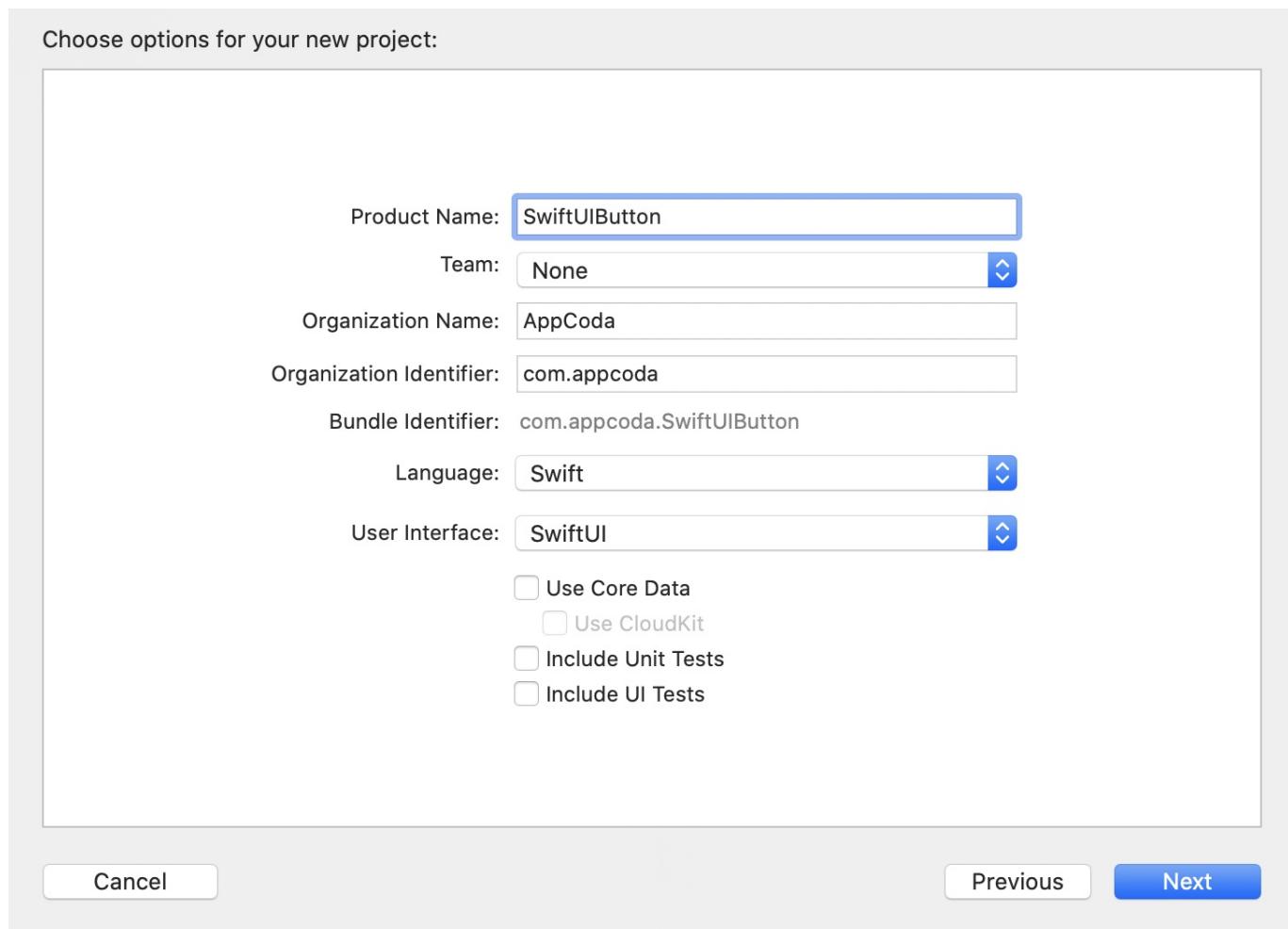


Figure 1. Creating a new project

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, you can click the *Resume* button in the canvas.

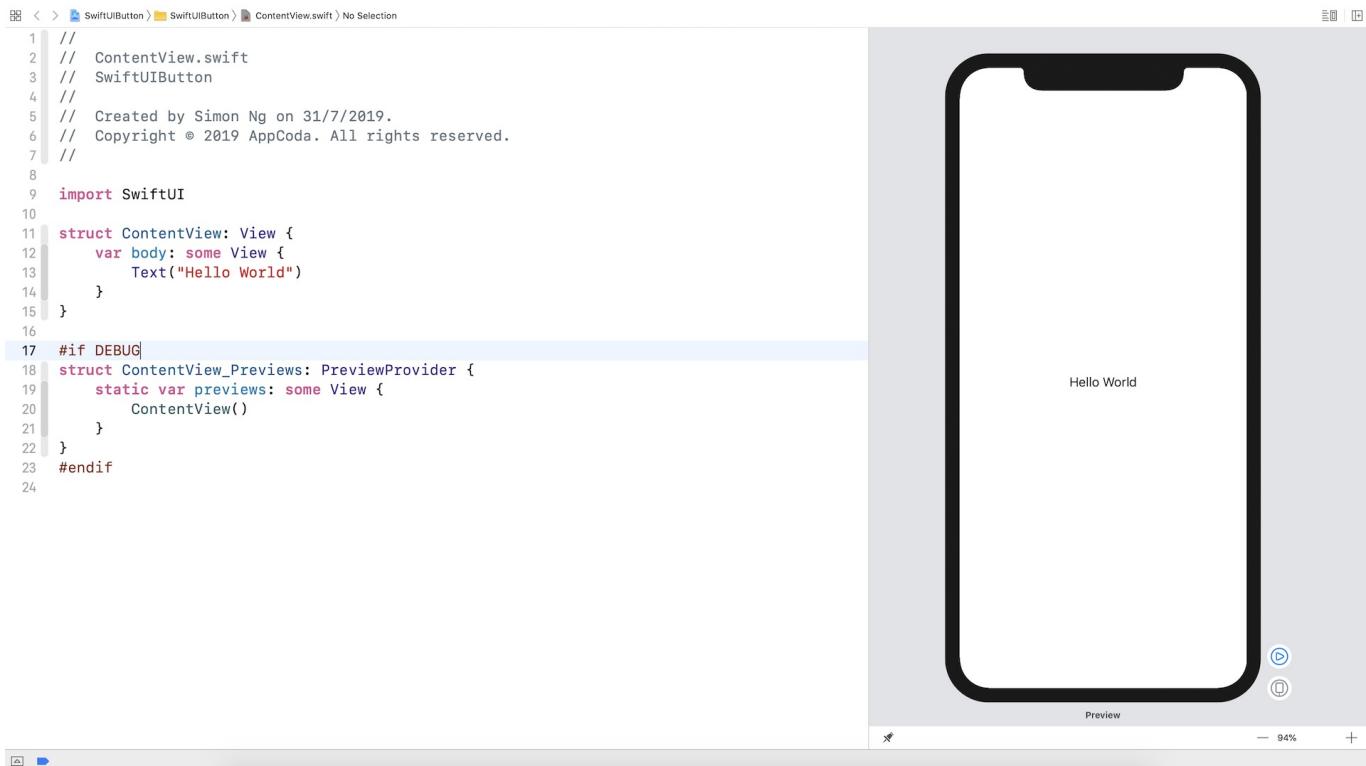


Figure 2. Previewing the default content view

It's very easy to create a button using SwiftUI. Basically, you can 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 to perform** - the code to perform after the button is tapped or selected by the user.
- 2. How the button looks like** - 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 can see it in the canvas.

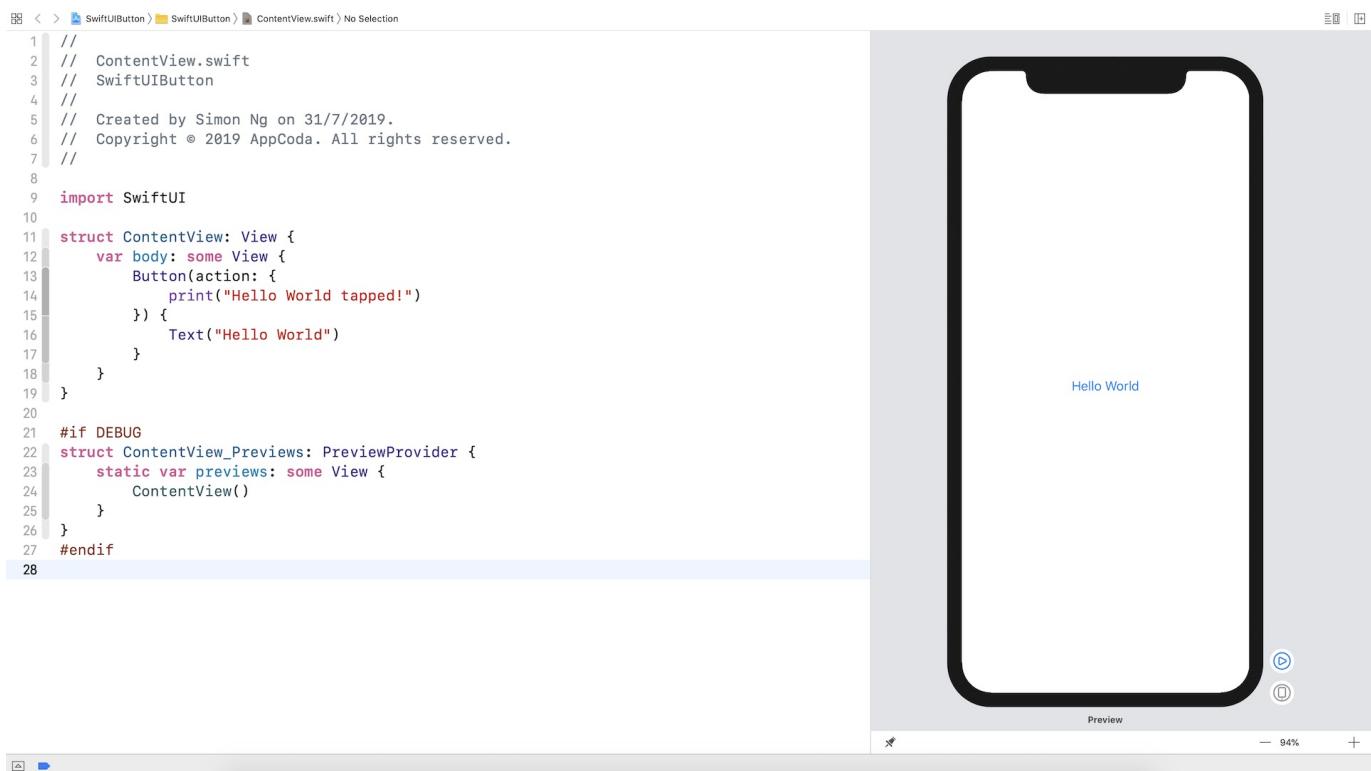


Figure 3. Creating a simple button

The button is now 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 message on the console when you tap the button.

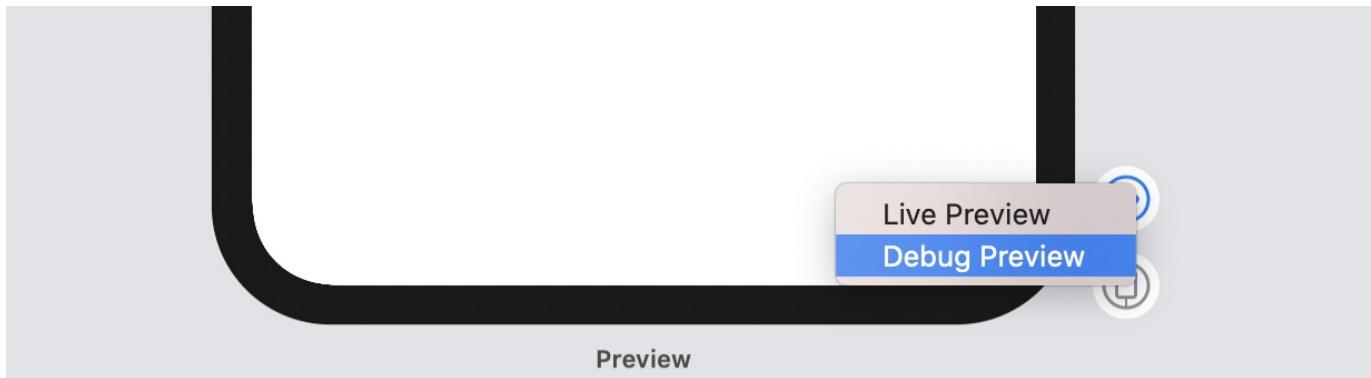


Figure 4. The console message can only be displayed in debug mode

Customizing the Button's Font and Background

Now that you should know how to create a simple button, let's see how to customize its look & feel with the built-in modifiers. Say, 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 further 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 made the change, the canvas should 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 like below, the end result will be different.

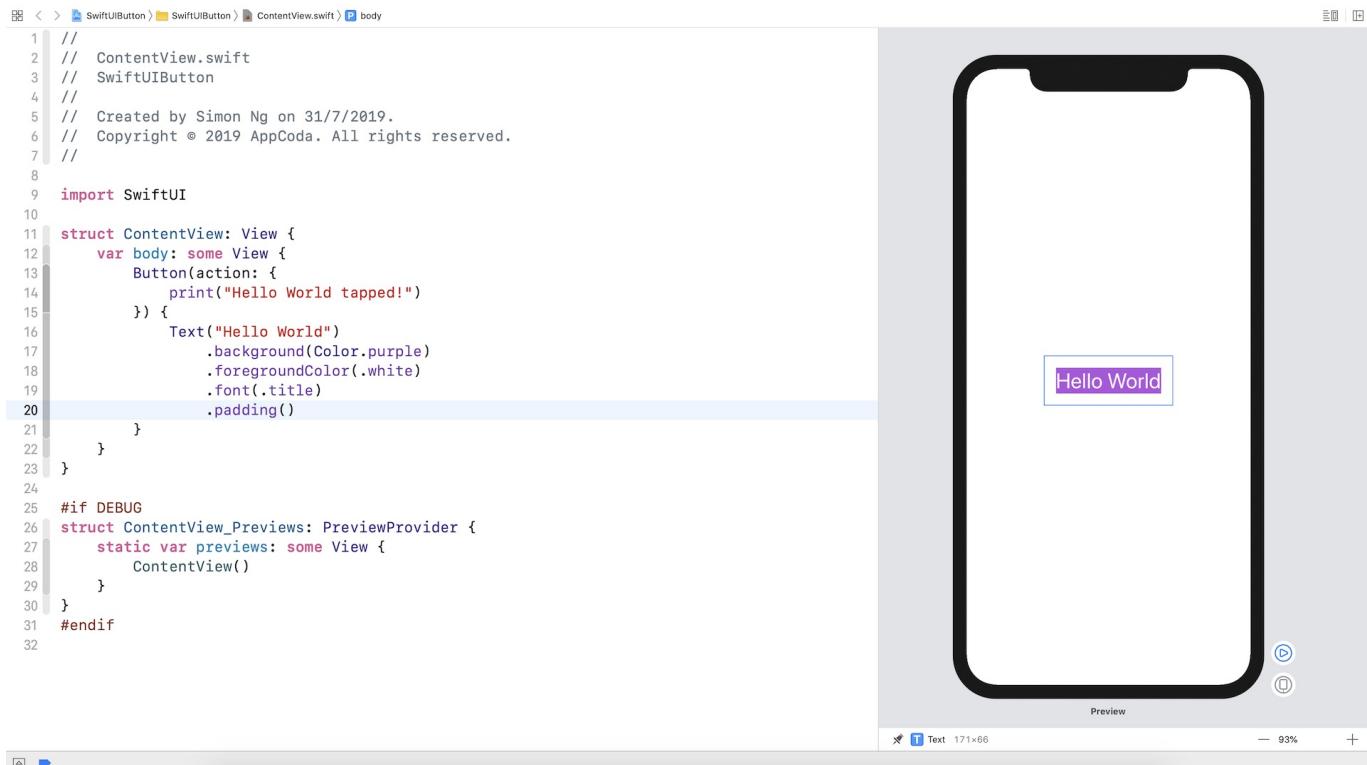


Figure 7. Placing the `padding` modifier after the `background` modifier

If you place the `padding` modifier after the `background` modifier, you can still add some paddings to the button but without the preferred background color. If you wonder why, you can read 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

It doesn't mean the `padding` modifier should always place 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 paddings around the text. The `border` modifier is used to define the border's width and color. Please feel free to 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, please give yourself a few minutes to figure out the solution.



Figure 9. A button with both background and border

Okay, here is the code you need:

```
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 purple background. The `padding(10)` modifier adds extra paddings around the button and the

`border` modifier specifies to paint a rounded border in purple.

Let's see a more complex example. What if you want to design the 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 can simply use the modifier and specify the corner radius like this:

```
.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. So, 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, what we did was to 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 only work 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 to use `.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 11 shows you an example.

The screenshot shows the Xcode interface with the code for `ContentView.swift`. The code defines a `ContentView` struct containing a `Button` with a `trash` icon, red background, circular shape, large title font, and white foreground color. The preview window shows a red trash can icon on an iPhone X screen.

```
1 //  
2 // Content View.swift  
3 // SwiftUI Button  
4 //  
5 // Created by Simon Ng on 31/7/2019.  
6 // Copyright © 2019 AppCoda. All rights reserved.  
7 //  
8  
9 import SwiftUI  
10  
11 struct ContentView: View {  
12     var body: some View {  
13         Button(action: {  
14             print("Delete tapped!")  
15         }) {  
16             Image(systemName: "trash")  
17                 .padding()  
18                 .background(Color.red)  
19                 .clipShape(Circle())  
20                 .font(.largeTitle)  
21                 .foregroundColor(.white)  
22         }  
23     }  
24 }  
25  
26 #if DEBUG  
27 struct ContentView_Previews: PreviewProvider {  
28     static var previews: some View {  
29         ContentView()  
30     }  
31 }  
32 #endif  
33
```

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, paddings, and round the button's corners. Figure 12 shows the resulting button.

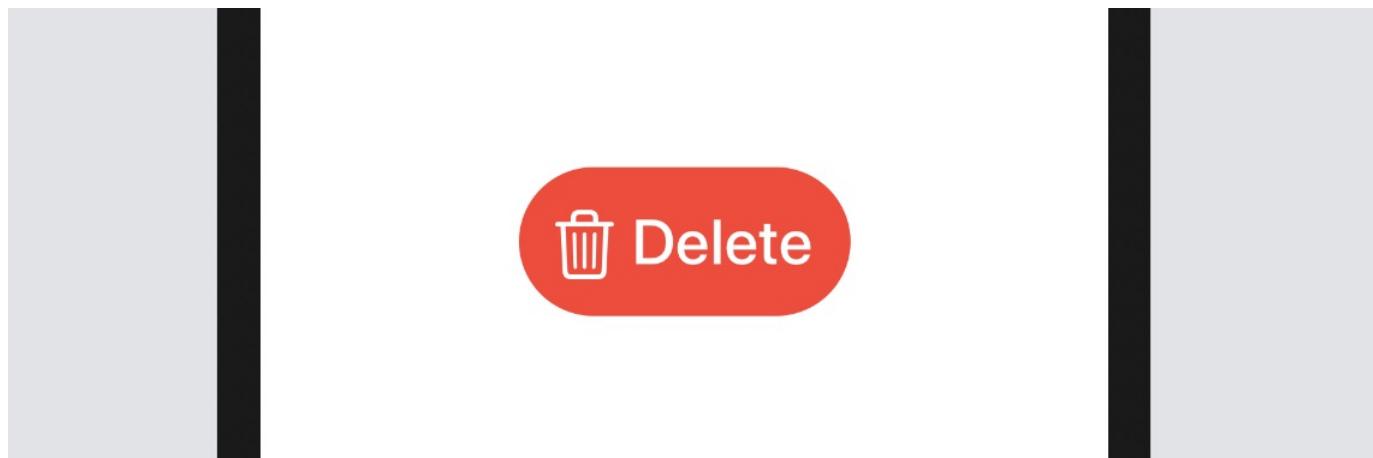


Figure 13. A button with both image and text

Creating a Button with Gradient Background and Shadow

With SwiftUI, you can easily style the button with gradient background. Not only can you apply a specific color to the `background` modifier, you can easily apply a gradient effect for any buttons. 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]), startPoint: .leading, endPoint: .trailing))
```

The SwiftUI framework comes with several built-in gradient effect. 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 can change the line of code like this:

```
.background(LinearGradient(gradient: Gradient(colors: [Color.red, Color.blue]), startPoint: .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 I'm going to show one of the approaches. In the project navigator, choose the asset catalog (i.e. `Assets.xcassets`). Right click the blank area (under AppIcon) and select *New Color Set*.

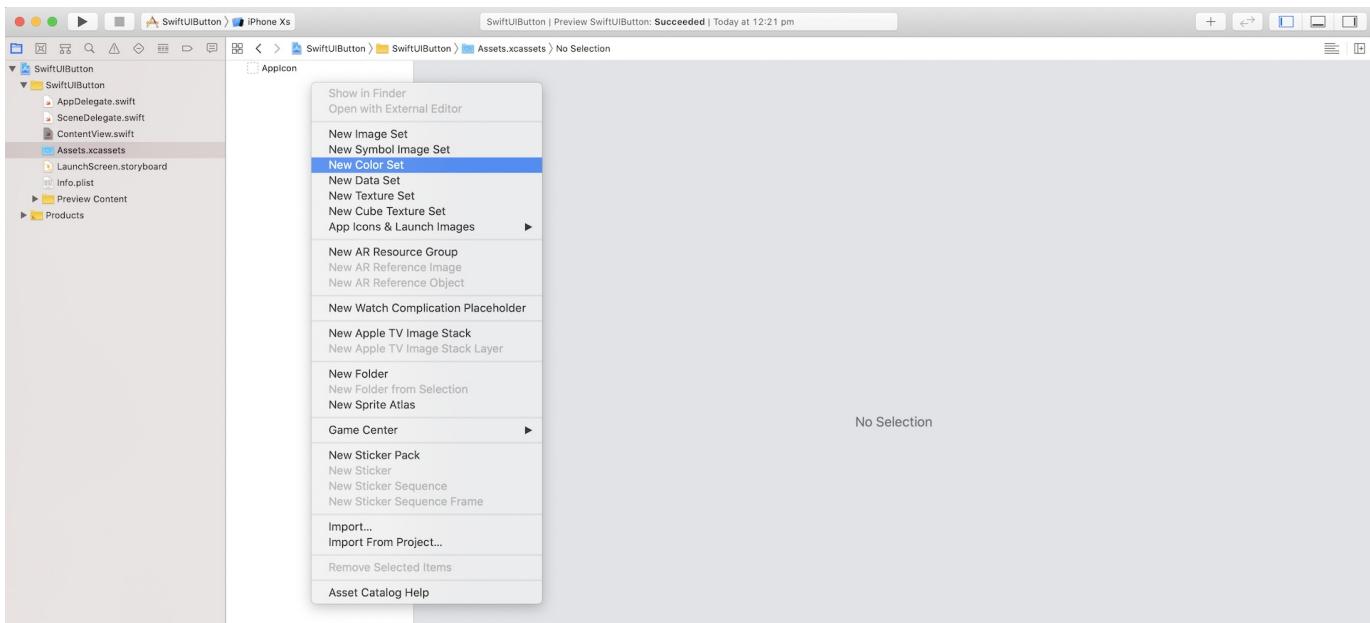


Figure 16. Define a new color set in the asset catalog

Next, choose the color well 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*.

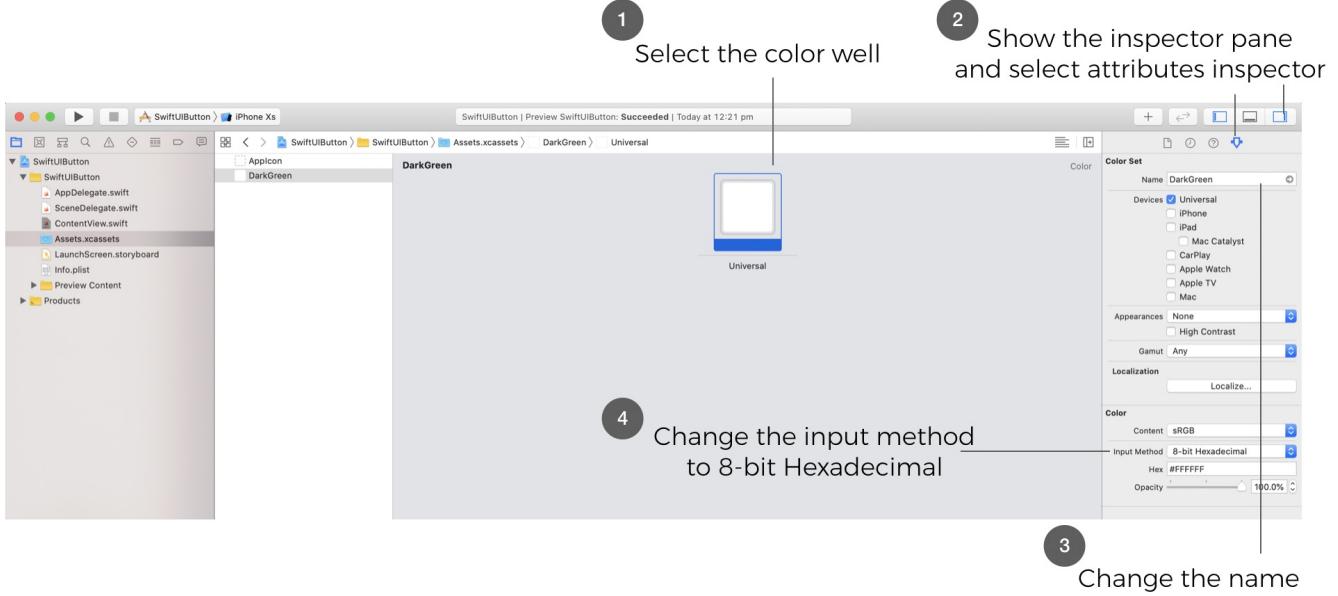


Figure 17. Editing the attributes of a color set

Now you can set the color code in the `Hex` field. For this demo, enter `#11998e` to define the color. Repeat the procedures to define another color set for `#38ef7d`. You can name the color set *LightGreen*.



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")
```

Therefore, to render the gradient with the *DarkGreen* and *LightGreen* color sets, all you need is 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.

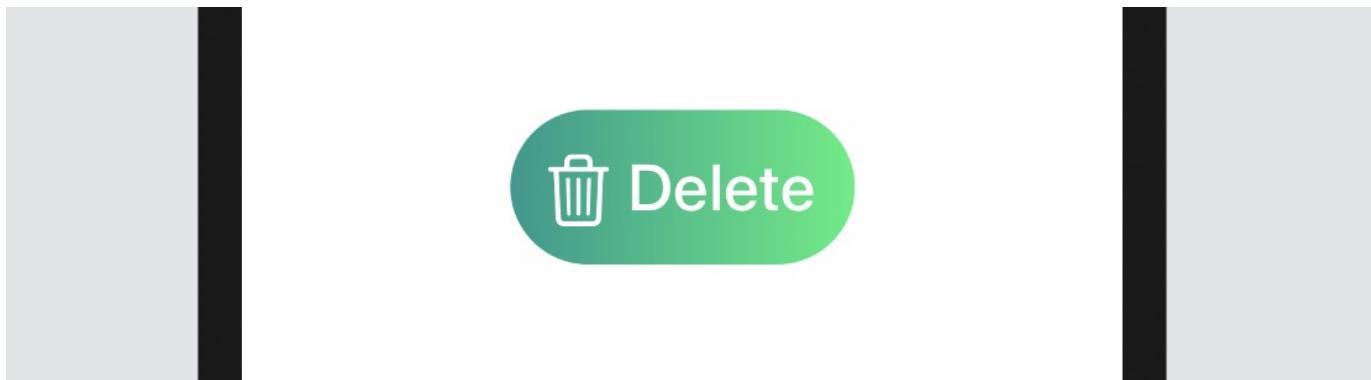


Figure 19. Generating a gradient with our own colors

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

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

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

```
.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 let you control the size of a view. Whether you want to create a fixed size button or a button with variable width, you can make use of 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"), Color("LightGreen")]), startPoint: .leading, endPoint: .trailing))
    .cornerRadius(40)
}
```

The code is very similar to the one 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`. That means the button will fill the width of the container view. In the canvas, it should now show you a full-width button.

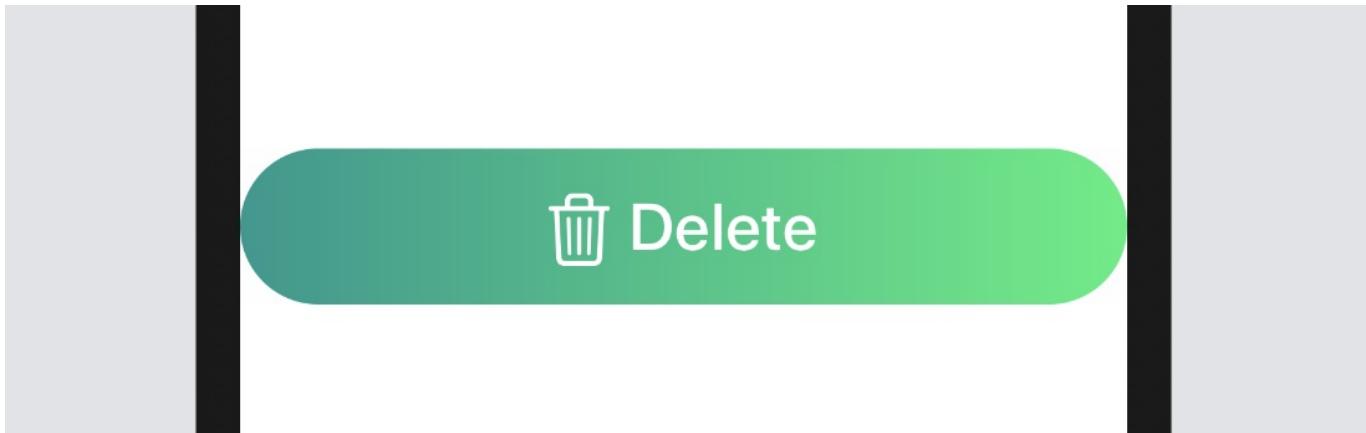


Figure 20. A full-width button

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:

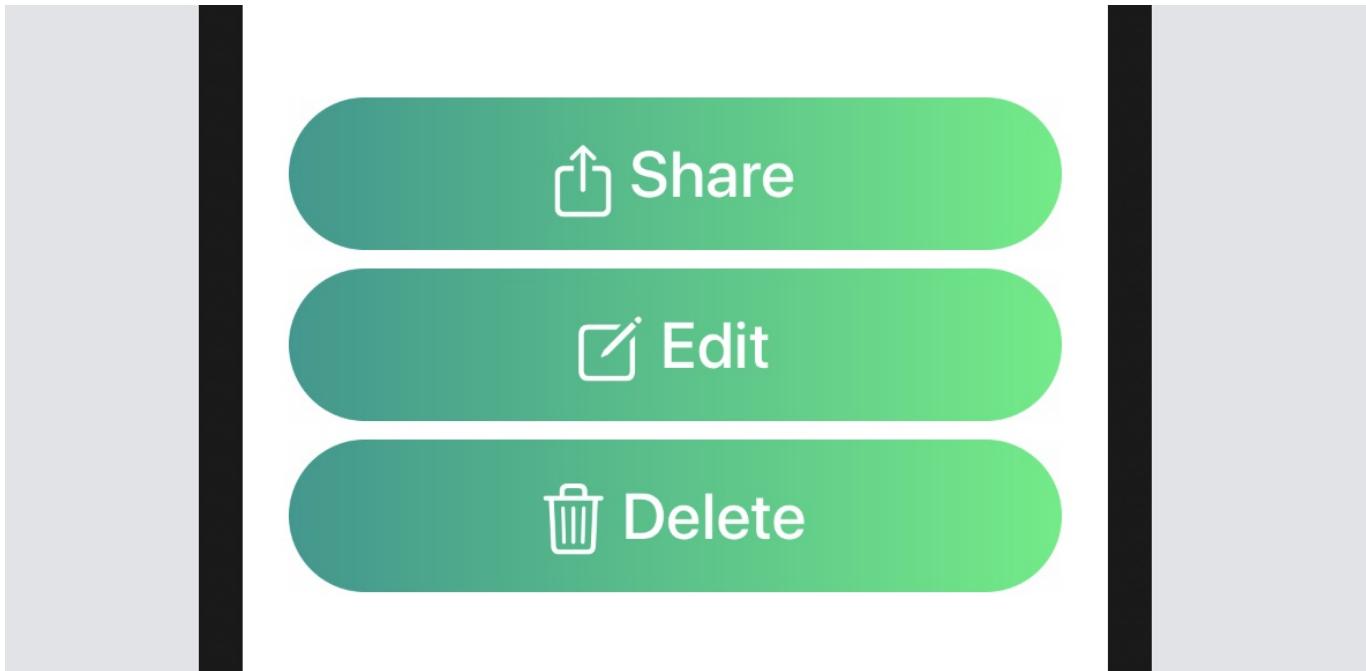


Figure 21. A full-width button

You'll probably write the code like below:

```
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. And, 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 a 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, we create a new struct called `GradientBackgroundStyle` that conforms to the `ButtonStyle` protocol. Insert the following code snippet and put it right above `#if DEBUG`:

```
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)
    }
}
```

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

So, how can 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 buttons with just one line of code.

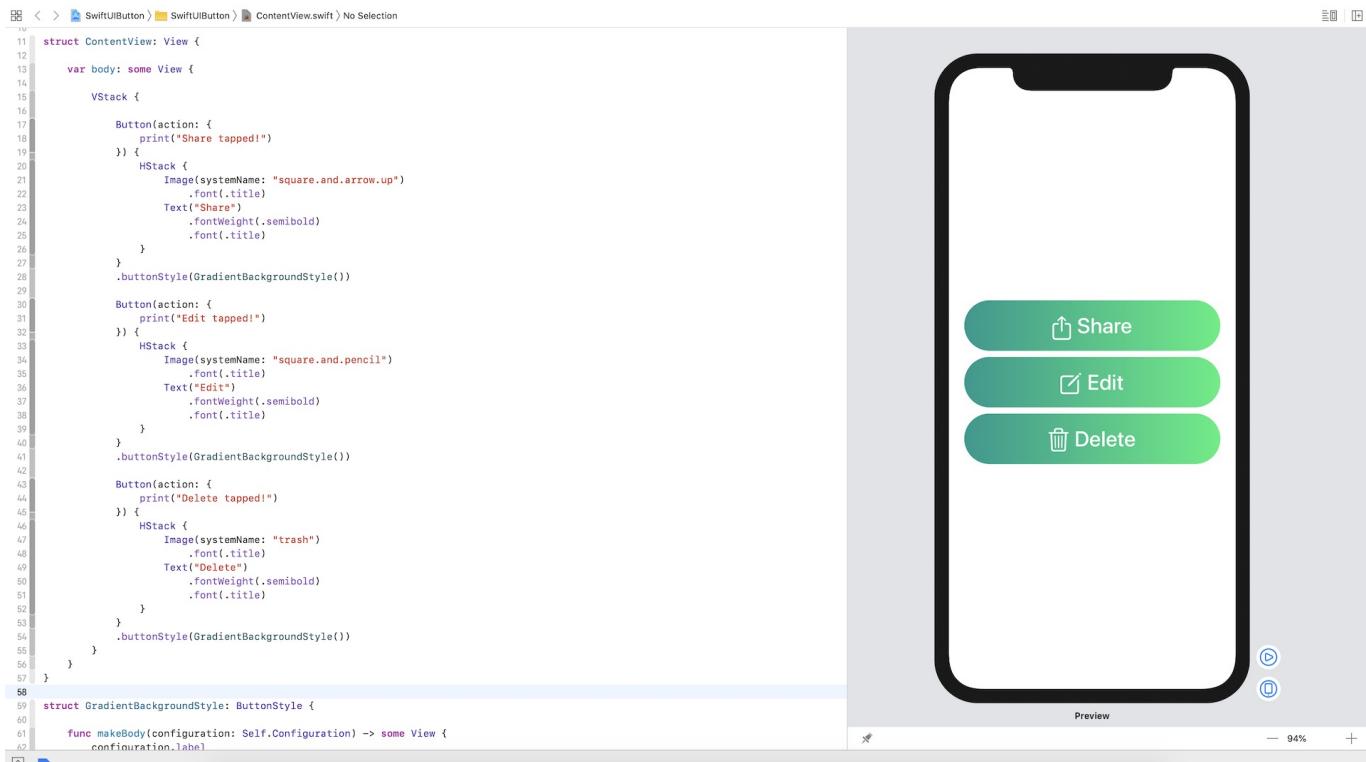


Figure 22. Applying the custom style using .buttonStyle modifier

You can also determine if the button is pressed by accessing the `isPressed` properties of the configuration. 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 down the button. You can add a line of code like this:

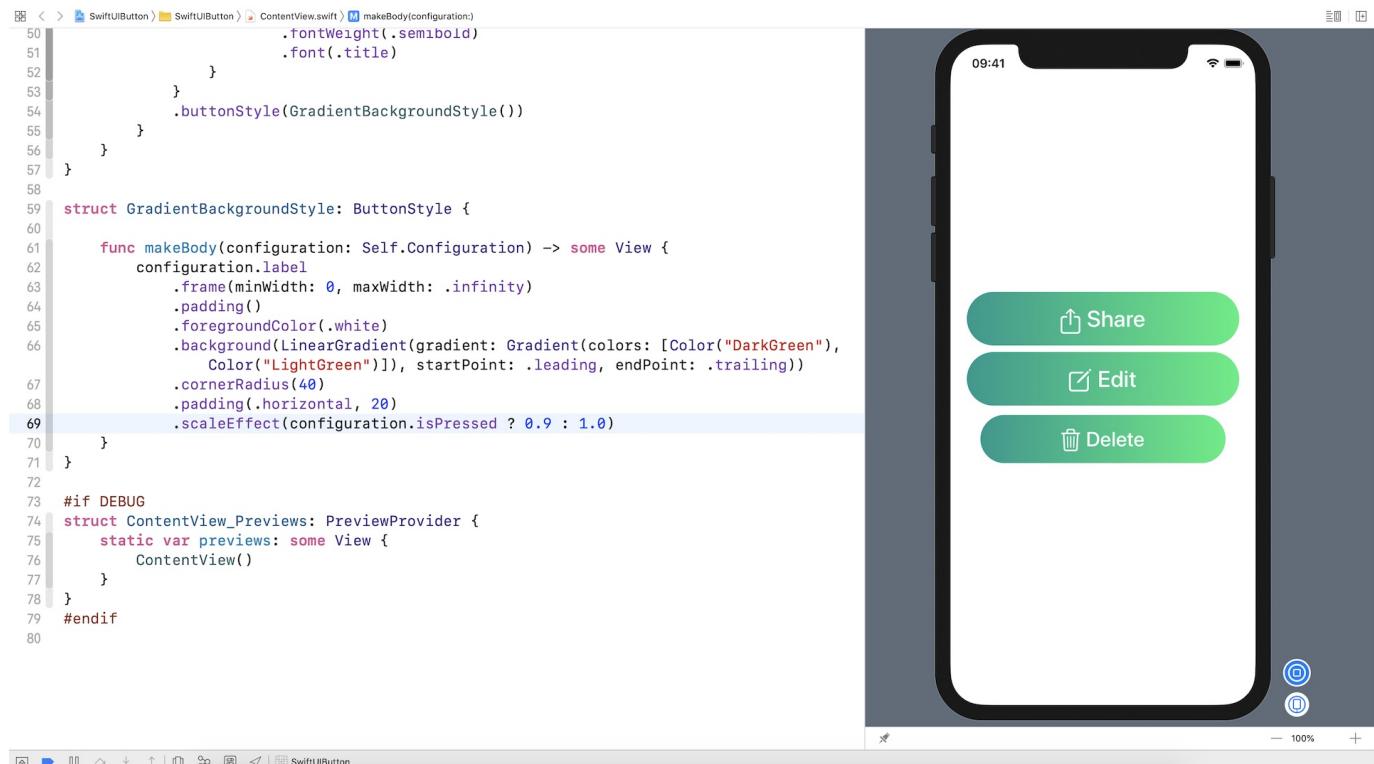


Figure 23. Applying the custom style using `.buttonStyle` modifier

The `scaleEffect` modifier lets you scale up or down a button (and any views). To scale up the button, you pass a value greater than 1.0. A value less than 1.0 can make the button smaller.

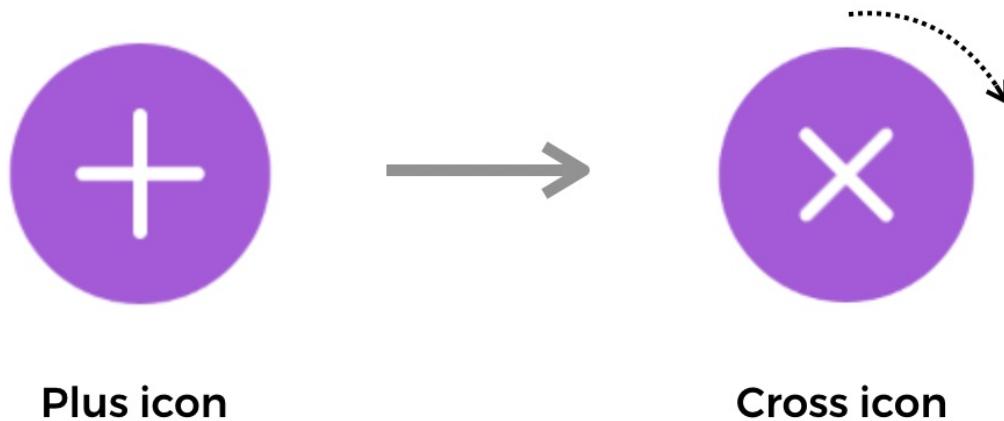
```
.scaleEffect(configuration.isPressed ? 0.9 : 1.0)
```

So, what the line of code does is that it scales 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 the finger. If you've 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 line of code and it comes with a 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/anticlockwise) to become a cross icon.



Plus icon

Cross icon

Figure 24. Rotate the icon when a user presses it

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

Summary

In this chapter, we cover the basics of creating buttons in SwiftUI. Buttons play a key role in mobile interface or to be exact, any application UI. Well designed buttons, not only make your UI more appealing, would 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 an attractive and useful button.

For reference, you can download the complete project here:

- Demo project (<https://www.appcoda.com/resources/swiftui/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 ways 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

In SwiftUI, it 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?

Anyway, you will get a better understanding of state and binding after going through the coding examples in this chapter. And, I've prepared a couple of exercises for you. Please do spare some time to work on it. This would 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 *Single View Application* template. Set the name of the project to *SwiftUIState* but you're free to use any other name. All you need to ensure is check the *Use SwiftUI* option.

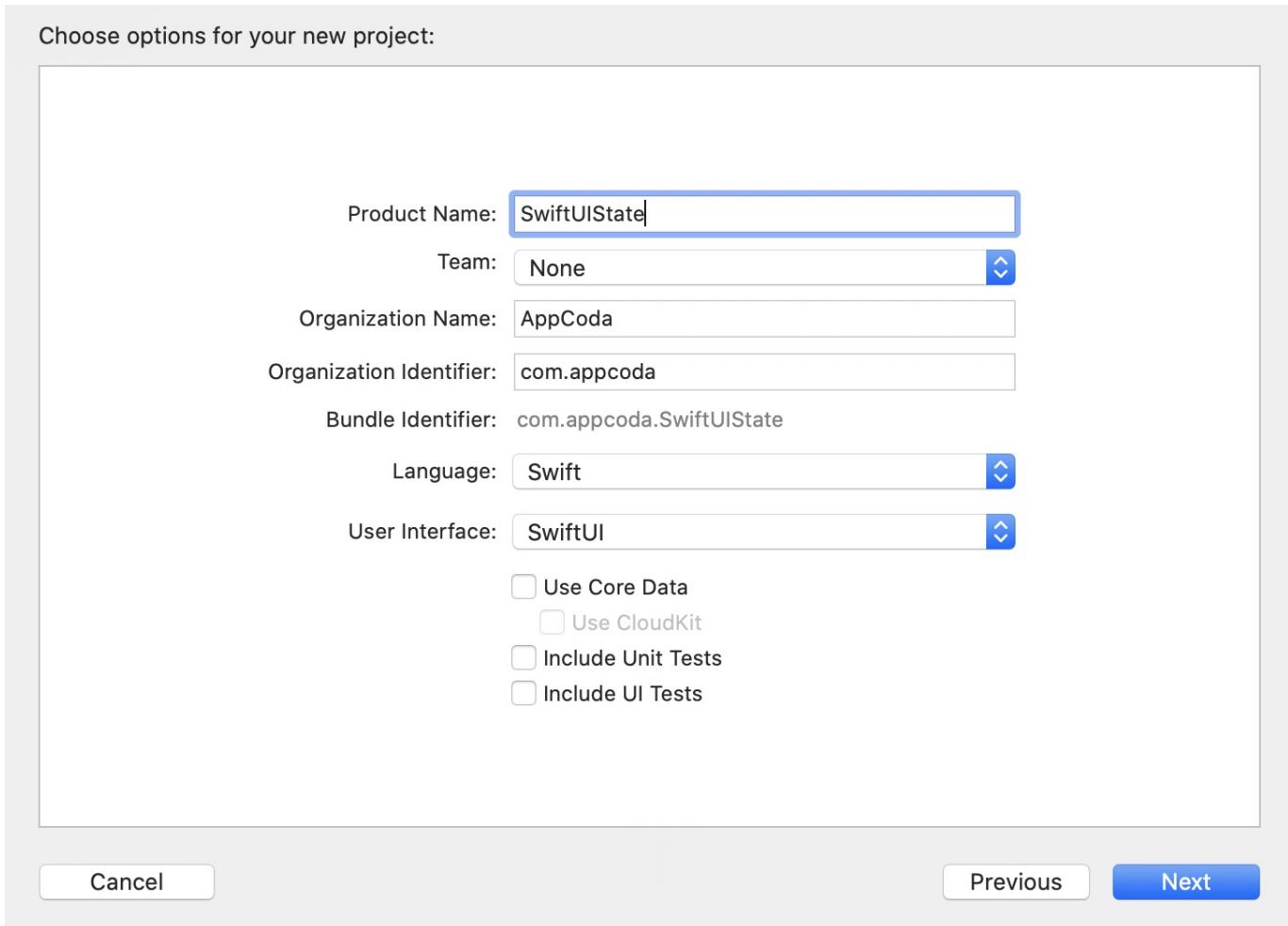


Figure 2. Creating a new project

Once you save the project, Xcode should load the `ContentView.swift` file and display a preview in the design canvas. Now we 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 and paint the button in green.

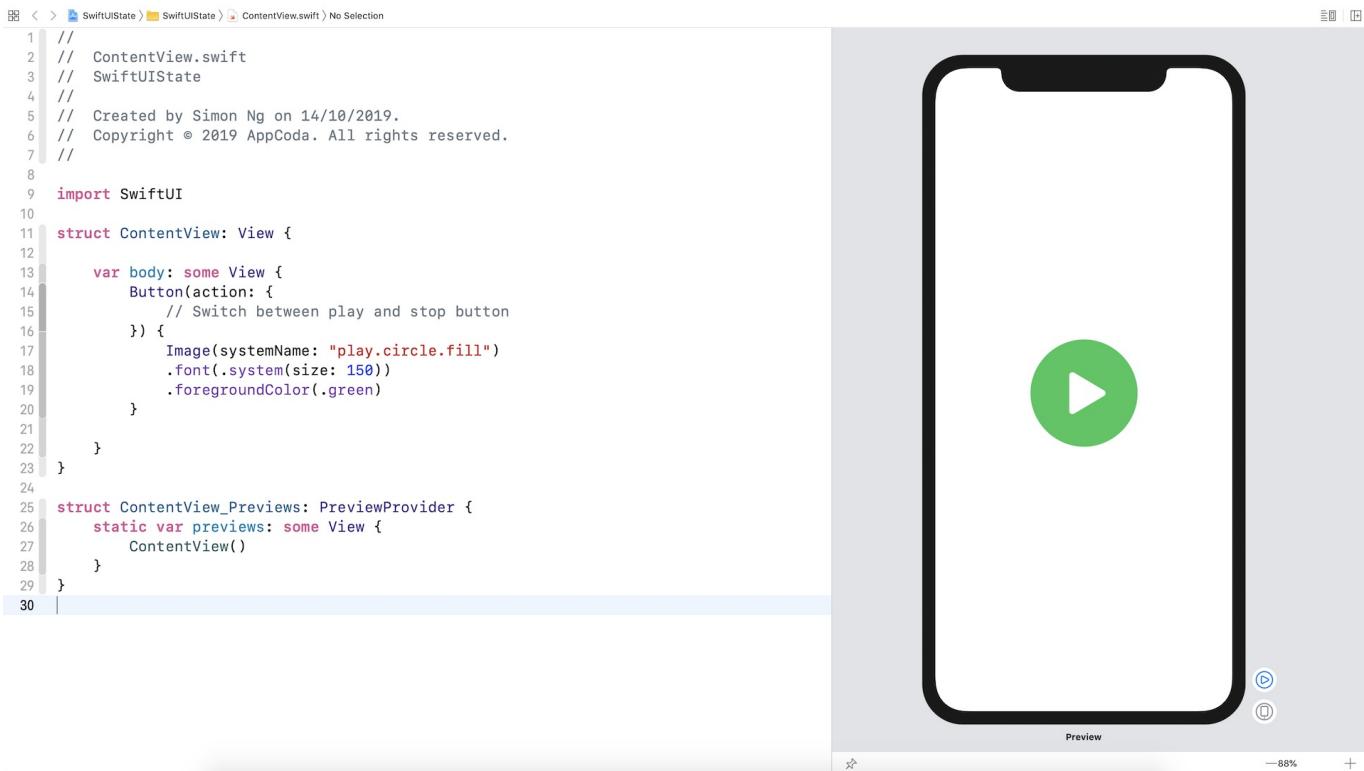


Figure 3. Previewing the play button

Controlling the Button's State

The button's action is now empty. What we want to do is change the button's 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 it's set to `true`, the app should show a *Stop* button. Conversely, the app shows a *Play* button. The code can be 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 to 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, SwiftUI manages the storage of `isPlaying` and monitor its value change. When the value of `isPlaying` changes, SwiftUI automatically recomputes the views that are referencing the `isPlaying` state. Here, it's the `Button`.

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, modify the code like this:

```
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`. Now run the app by clicking the play icon in the preview canvas and try to toggle between the *Play* and *Stop* button.

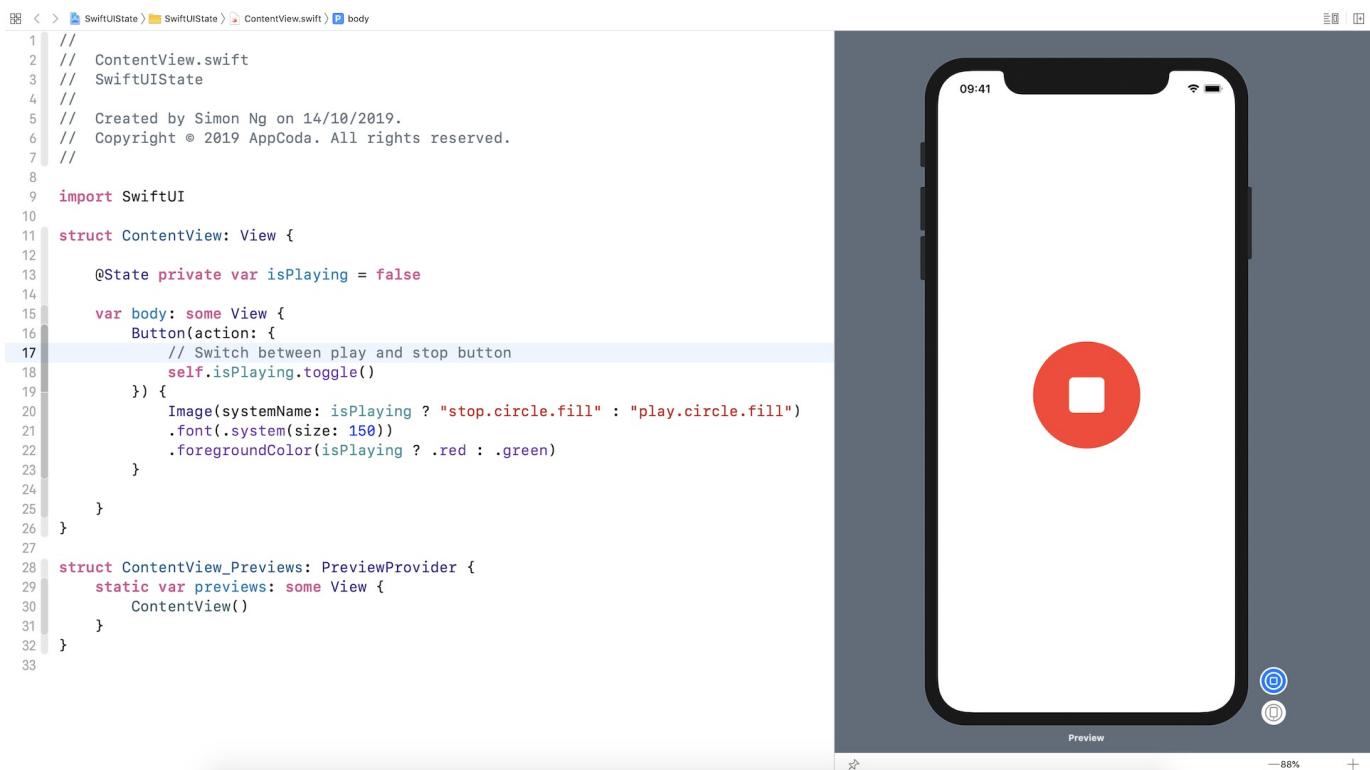


Figure 4. Toggle between the Play and Stop button

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 to 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

Are you able to create 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 for your reference.

The screenshot shows the Swift code for `ContentView` and its preview. The code uses `@State` to track a counter value and `Button` to increment it. The preview shows a red circle containing the number 21.

```
7 //  
8  
9 import SwiftUI  
10  
11 struct ContentView: View {  
12     @State private var counter = 1  
13  
14     var body: some View {  
15         Button(action: {  
16             self.counter += 1  
17         }) {  
18             Circle()  
19                 .frame(width: 200, height: 200)  
20                 .foregroundColor(.red)  
21                 .overlay(  
22                     Text("\(counter)")  
23                         .font(.system(size: 100, weight: .bold, design: .rounded))  
24                         .foregroundColor(.white)  
25                 )  
26         }  
27     }  
28 }  
29  
30  
31 struct ContentView_Previews: PreviewProvider {  
32     static var previews: some View {  
33         ContentView()  
34     }  
35 }  
36  
37
```

Figure 6. Three counter buttons

Okay, now we will further modify the code to display three counter buttons (see figure 7). All the 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.

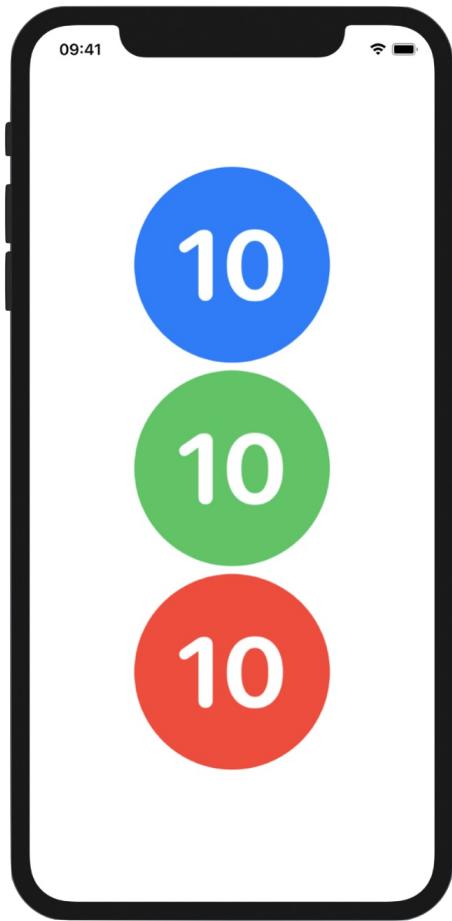


Figure 7. Three counter buttons

As you can see, all the buttons share the same look & feel. Like what 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. So, we can extract the Button to create an independent view 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 then create a button in red like this:

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

You should notice that the `counter` variable is annotated with `@Binding`. And, 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. By doing so, it's just like creating a 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.

If 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, you can run the app to test it. Tapping any of the buttons will increase the count by one.

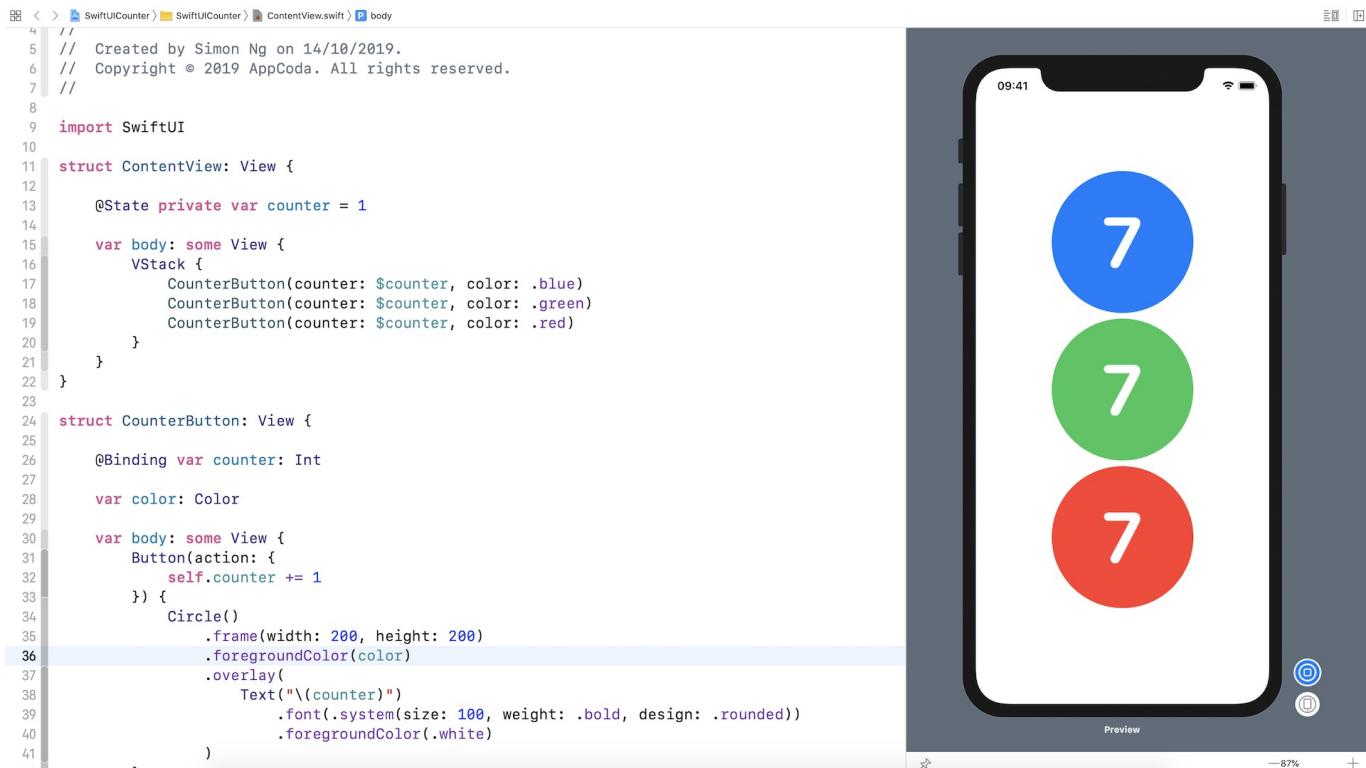


Figure 9. Testing the three counter buttons

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 counter. Say, when the user taps the blue button, the app only increases the counter of the blue button by 1. On top of that, you will need to provide a master counter that sums up the counter of all buttons. Figure 10 shows the sample layout of 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 project below:

- Demo project (<https://www.appcoda.com/resources/swiftui/SwiftUICounter.zip>)
- Exercise (<https://www.appcoda.com/resources/swiftui/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. In SwiftUI, it 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 I will go through with you:

- Understanding Path and how to draw a line with it
- What is the `Shape` protocol and how to drawing a custom shape by conforming to the protocol
- How to draw a pie chart
- How to create a progress indication 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 the later sections.

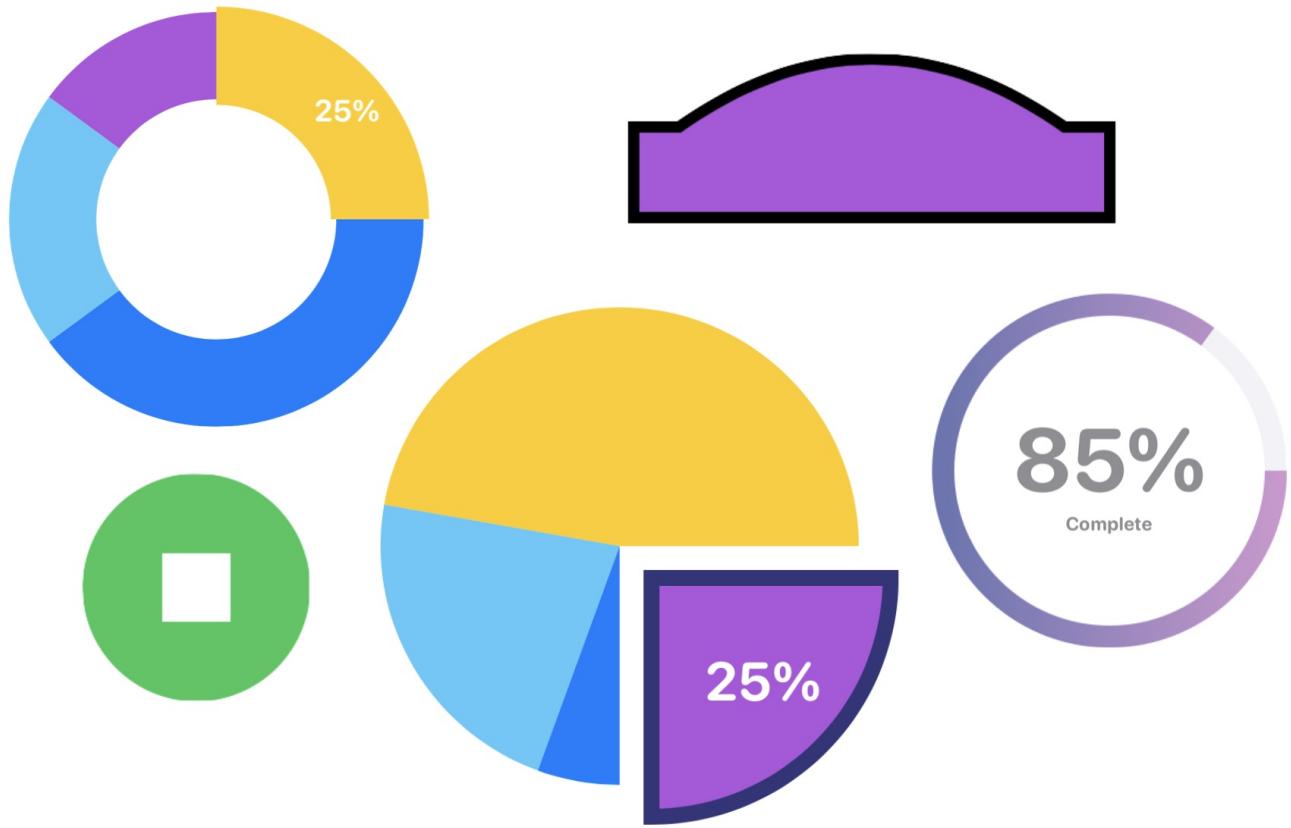


Figure 1. Sample shapes and charts

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 a step-by-step description of drawing a line or shapes. Let me give you an example. Take a look at figure 2. It's a rectangle that we want to draw on screen.

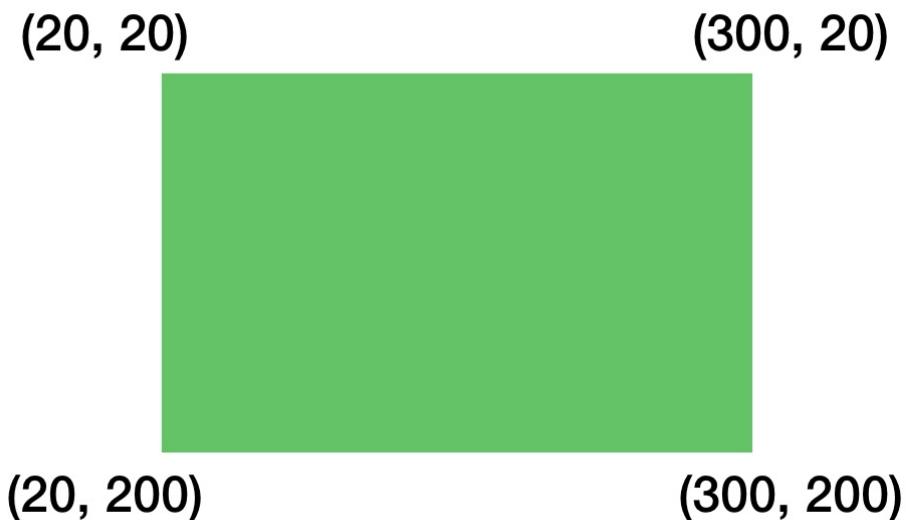


Figure 2. A rectangle with coordinates

Tell me verbally how you would draw the square step by step. You would probably provide the following description:

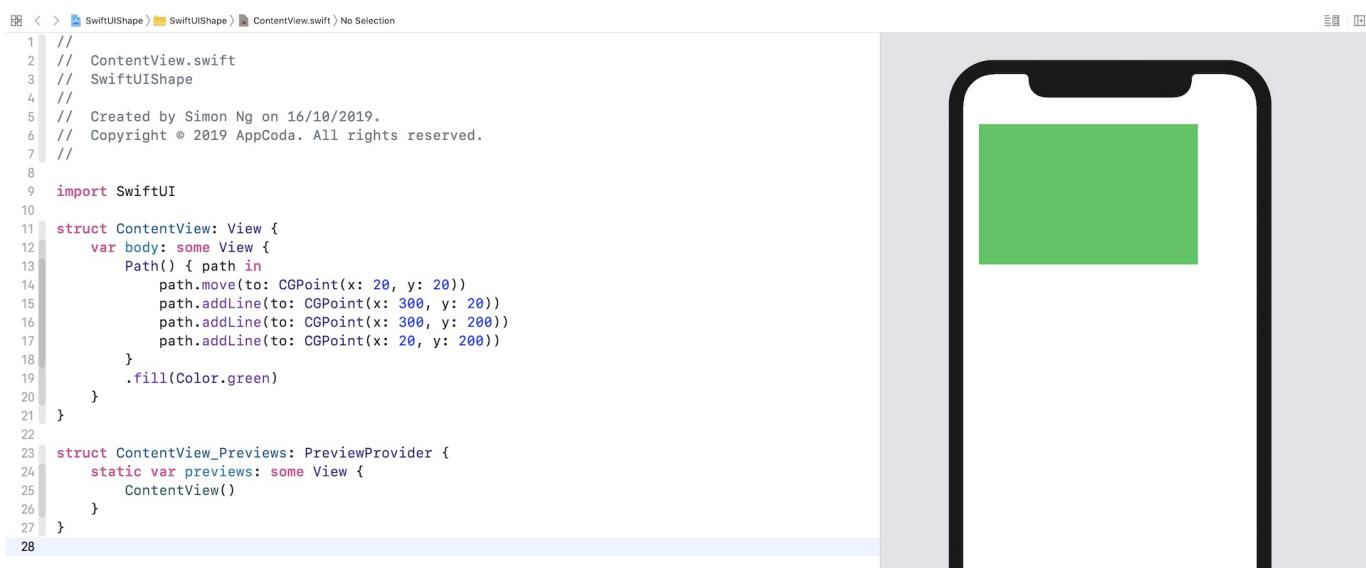
1. Move the point (20, 20)
2. Draw a line from (20, 20) to (300, 20)
3. Draw a line from (300, 20) to (300, 200)
4. Draw a line from (300, 200) to (20, 200)
5. Fill the whole area in green

That's what a `Path` is about. If you write the description above in code, this is what the code looks like:

```
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)
```

Here you initialize a `Path` and provides the detailed instruction in the closure. You can 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.

You can test the code by creating a new project using the Single View application 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.



The screenshot shows the Xcode interface with the code editor on the left and a preview canvas on the right. The code editor displays the following Swift code:

```
1 //  
2 // ContentView.swift  
3 // SwiftUIShape  
4 //  
5 // Created by Simon Ng on 16/10/2019.  
6 // Copyright © 2019 AppCoda. All rights reserved.  
7 //  
8  
9 import SwiftUI  
10  
11 struct ContentView: View {  
12     var body: some View {  
13         Path() { path in  
14             path.move(to: CGPoint(x: 20, y: 20))  
15             path.addLine(to: CGPoint(x: 300, y: 20))  
16             path.addLine(to: CGPoint(x: 300, y: 200))  
17             path.addLine(to: CGPoint(x: 20, y: 200))  
18         }  
19         .fill(Color.green)  
20     }  
21 }  
22  
23 struct ContentView_Previews: PreviewProvider {  
24     static var previews: some View {  
25         ContentView()  
26     }  
27 }
```

The preview canvas shows a smartphone-like device with a black frame and a solid green rectangular area inside the screen.

Figure 3. Drawing a rectangle using `Path`

Using Stroke to Draw Borders

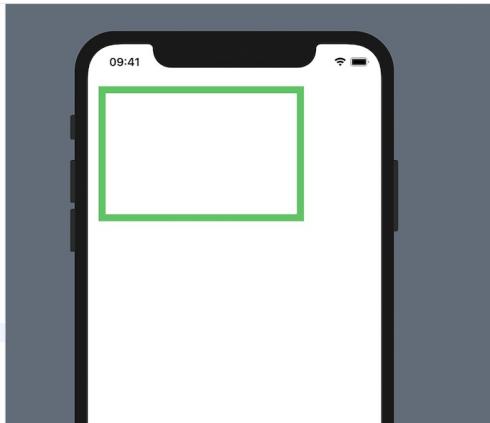
You're not required to fill the whole area with color. If you just want to draw the lines, you can use the `.stroke` modifier and specify the line width and color. Figure 4 shows the result.



```
1 //  
2 // ContentView.swift  
3 // SwiftUI  
4 //  
5 // Created by Simon Ng on 16/10/2019.  
6 // Copyright © 2019 AppCoda. All rights reserved.  
7 //  
8  
9 import SwiftUI  
10  
11 struct ContentView: View {  
12     var body: some View {  
13         Path() { path in  
14             path.move(to: CGPoint(x: 20, y: 20))  
15             path.addLine(to: CGPoint(x: 300, y: 20))  
16             path.addLine(to: CGPoint(x: 300, y: 200))  
17             path.addLine(to: CGPoint(x: 20, y: 200))  
18         }  
19         .stroke(Color.green, lineWidth: 10)  
20     }  
21 }
```

Figure 4. Drawing the lines with stroke

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



```
1 //  
2 // ContentView.swift  
3 // SwiftUI  
4 //  
5 // Created by Simon Ng on 16/10/2019.  
6 // Copyright © 2019 AppCoda. All rights reserved.  
7 //  
8  
9 import SwiftUI  
10  
11 struct ContentView: View {  
12     var body: some View {  
13         Path() { path in  
14             path.move(to: CGPoint(x: 20, y: 20))  
15             path.addLine(to: CGPoint(x: 300, y: 20))  
16             path.addLine(to: CGPoint(x: 300, y: 200))  
17             path.addLine(to: CGPoint(x: 20, y: 200))  
18             path.closeSubpath()  
19         }  
20         .stroke(Color.green, lineWidth: 10)  
21     }  
22 }
```

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 draw straight lines. The `addQuadCurve`, `addCurve`, and `addArc` allow you to create curves and arc. Let's say, you want to draw a dome on top of a rectangle like that

shown in figure 6.

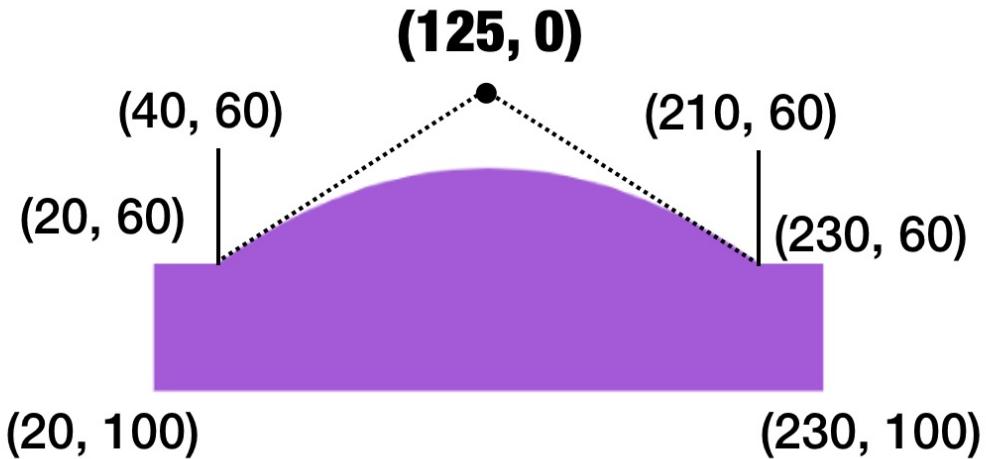


Figure 6. A dome with a rectangle bottom

You can 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, you can try to change the value of the control point to see the effect. In brief, this control point controls how the curve is drawn. If it's placing 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. That said, 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.

The screenshot shows the Xcode interface with the Swift file 'ContentView.swift' open. The code defines a 'ContentView' struct that contains a ZStack with two Path components. The first Path creates a purple-filled dome shape with a black stroke. The second Path creates a black-outlined dome shape with a purple fill. A preview of the view is shown in an iPhone X simulator, displaying the dome shapes.

```
1 // ContentView.swift
2 // SwiftUISHape
3 // SwiftUISHape
4 //
5 // Created by Simon Ng on 16/10/2019.
6 // Copyright © 2019 AppCoda. All rights reserved.
7 //
8 import SwiftUI
9
10 struct ContentView: View {
11     var body: some View {
12
13         ZStack {
14             Path() { path in
15                 path.move(to: CGPoint(x: 20, y: 60))
16                 path.addLine(to: CGPoint(x: 40, y: 60))
17                 path.addQuadCurve(to: CGPoint(x: 210, y: 60), control: CGPoint(x: 125, y: 0))
18                 path.addLine(to: CGPoint(x: 230, y: 60))
19                 path.addLine(to: CGPoint(x: 230, y: 100))
20                 path.addLine(to: CGPoint(x: 20, y: 100))
21                 path.addLine(to: CGPoint(x: 20, y: 60))
22             }
23             .fill(Color.purple)
24
25             Path() { path in
26                 path.move(to: CGPoint(x: 20, y: 60))
27                 path.addLine(to: CGPoint(x: 40, y: 60))
28                 path.addQuadCurve(to: CGPoint(x: 210, y: 60), control: CGPoint(x: 125, y: 0))
29                 path.addLine(to: CGPoint(x: 230, y: 60))
30                 path.addLine(to: CGPoint(x: 230, y: 100))
31                 path.addLine(to: CGPoint(x: 20, y: 100))
32                 path.closeSubpath()
33             }
34             .stroke(Color.black, lineWidth: 5)
35         }
36     }
37 }
38
39 struct ContentView_Previews: PreviewProvider {
40     static var previews: some View {
41 }
```

Figure 7. A dome rectangle with borders

Drawing Arcs and Pie Charts

SwiftUI provides a convenient API for developers to draw arcs. This API is incrediblly useful to compose various shapes and objects including pie charts. To draw an arc, you can 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(degrees: 0.0), endAngle: Angle(degrees: 90), clockwise: true)
}
.fill(Color.green)
```

If you've put the code in the body, you will see an arc that fills with green color in the preview canvas.

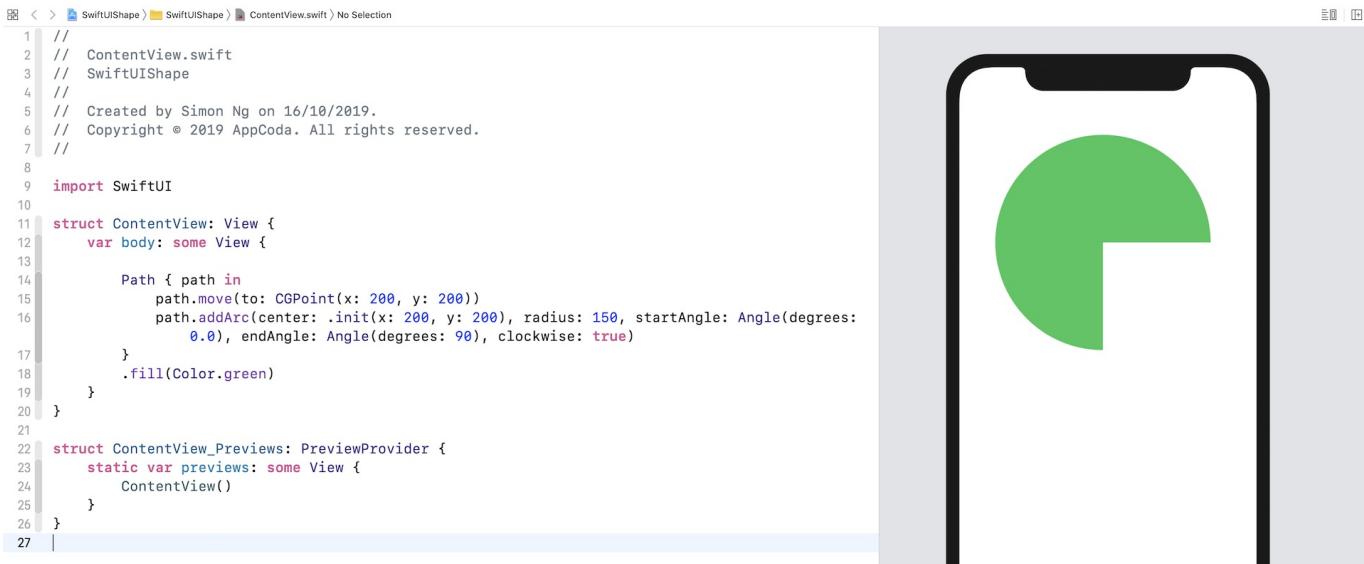


Figure 8. A sample arc

In the code, we first move to the starting point to (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 end 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 should be a bit confused with its meaning. Figure 9 will give you a better idea about what these parameters are about.

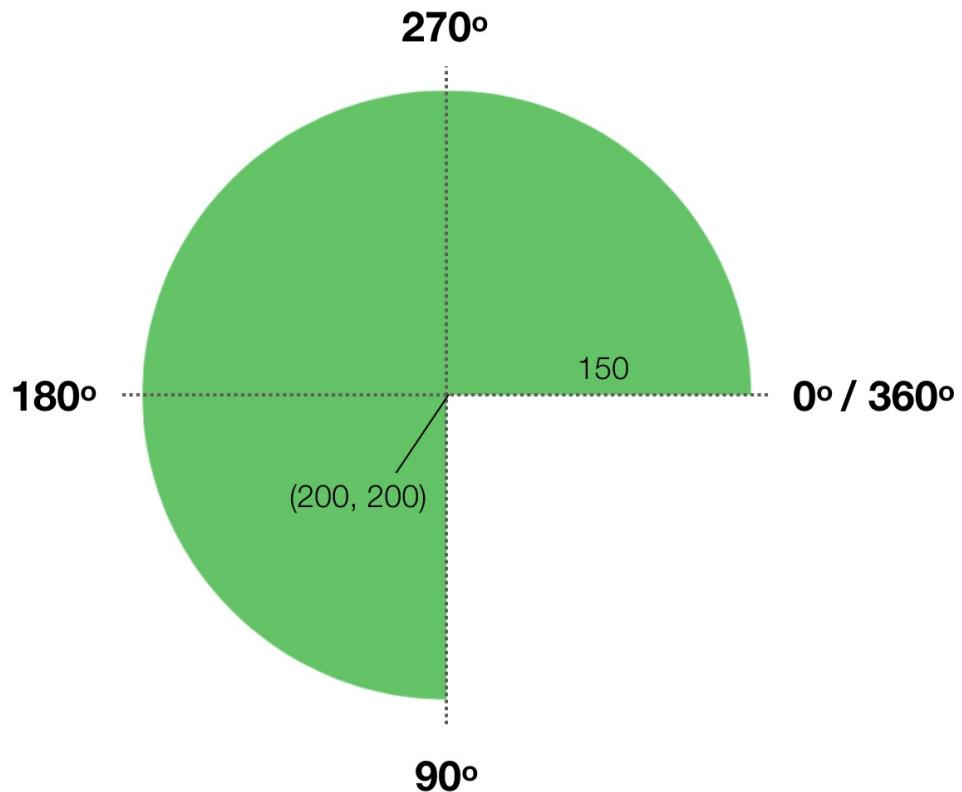


Figure 9. Understanding starting and end angle

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 a different values of `startAngle` and `endAngle` to compose the chart. Here is the code snippet:

```

ZStack {
    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: 190), clockwise: true)
    }
    .fill(Color(.systemYellow))

    Path { path in
        path.move(to: CGPoint(x: 200, y: 200))
        path.addArc(center: .init(x: 200, y: 200), radius: 150, startAngle: Angle(degrees: 190), endAngle: Angle(degrees: 110), clockwise: true)
    }
    .fill(Color(.systemTeal))

    Path { path in
        path.move(to: CGPoint(x: 200, y: 200))
        path.addArc(center: .init(x: 200, y: 200), radius: 150, startAngle: Angle(degrees: 110), endAngle: Angle(degrees: 90), clockwise: true)
    }
    .fill(Color(.systemBlue))

    Path { path in
        path.move(to: CGPoint(x: 200, y: 200))
        path.addArc(center: .init(x: 200, y: 200), radius: 150, startAngle: Angle(degrees: 90.0), endAngle: Angle(degrees: 360), clockwise: true)
    }
    .fill(Color(.systemPurple))
}

```

This would render a pie chart with 4 segments. If you need 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.

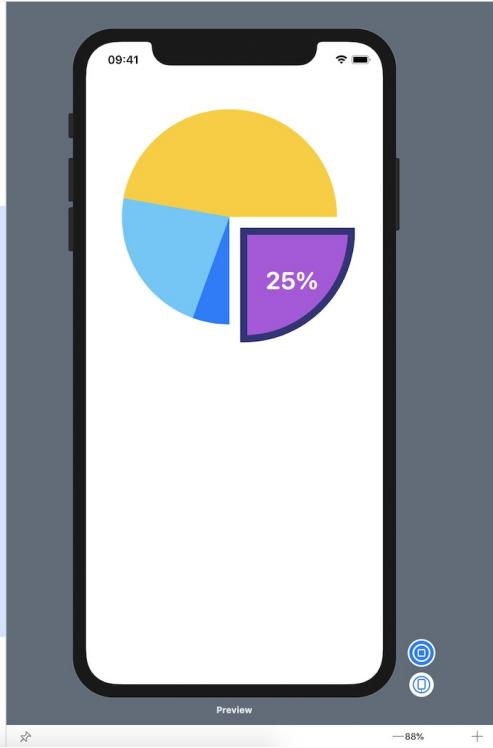
Sometimes, you may want to highlight a particular segment by splitting it from the pie chart. Say, to highlight the segment in purple, you can apply the `offset` modifier to re-position the segment:

```
Path { path in
    path.move(to: CGPoint(x: 200, y: 200))
    path.addArc(center: .init(x: 200, y: 200), radius: 150, startAngle: Angle(degrees: 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: 200, y: 200))
    path.addArc(center: .init(x: 200, y: 200), 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: -120)
)
```

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



```

25 } .fill(Color(.systemTeal))
26
27 Path { path in
28     path.move(to: CGPoint(x: 200, y: 200))
29     path.addArc(center: .init(x: 200, y: 200), radius: 150, startAngle:
30         Angle(degrees: 110), endAngle: Angle(degrees: 90), clockwise: true)
31 }
32 .fill(Color(.systemBlue))
33
34 Path { path in
35     path.move(to: CGPoint(x: 200, y: 200))
36     path.addArc(center: .init(x: 200, y: 200), radius: 150, startAngle:
37         Angle(degrees: 90.0), endAngle: Angle(degrees: 360), clockwise: true)
38 }
39 .fill(Color(.systemPurple))
40 .offset(x: 20, y: 20)
41
42 Path { path in
43     path.move(to: CGPoint(x: 200, y: 200))
44     path.addArc(center: .init(x: 200, y: 200), radius: 150, startAngle:
45         Angle(degrees: 90.0), endAngle: Angle(degrees: 360), clockwise: true)
46     path.closeSubpath()
47 }
48 .stroke(Color(red: 52/255, green: 52/255, blue: 122/255), lineWidth: 10)
49 .offset(x: 20, y: 20)
50 .overlay(
51     Text("25%")
52         .font(.system(.largeTitle, design: .rounded))
53         .bold()
54         .foregroundColor(.white)
55         .offset(x: 80, y: -120)
56 )
57 }
58 }
59 struct ContentView_Previews: PreviewProvider {
60     static var previews: some View {
61         ContentView()
62     }
63 }

```

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. Base 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 a shape like this, you can 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 of `Path`, you may find another function called `addRect`, which lets you draw a rectangle with a specific width and height. So, here is an alternate solution:

```
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)
```

Now let's talk about the `Shape` protocol. The protocol is very simple with only one requirement. When adopting it, here is the function you have to implement:

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

So, when do we need to adopt the `shape` protocol? Let me ask you: how can you reuse the `Path` that you have just created? Say, for example, you want to create a button with the dome shape but flexible size. How can you implement it?

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 with the `rect` size. You can then draw the path within that `rect`.

Let's see how we create the `Dome` shape so you will have a better understanding of the `Shape` protocol.

```
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))

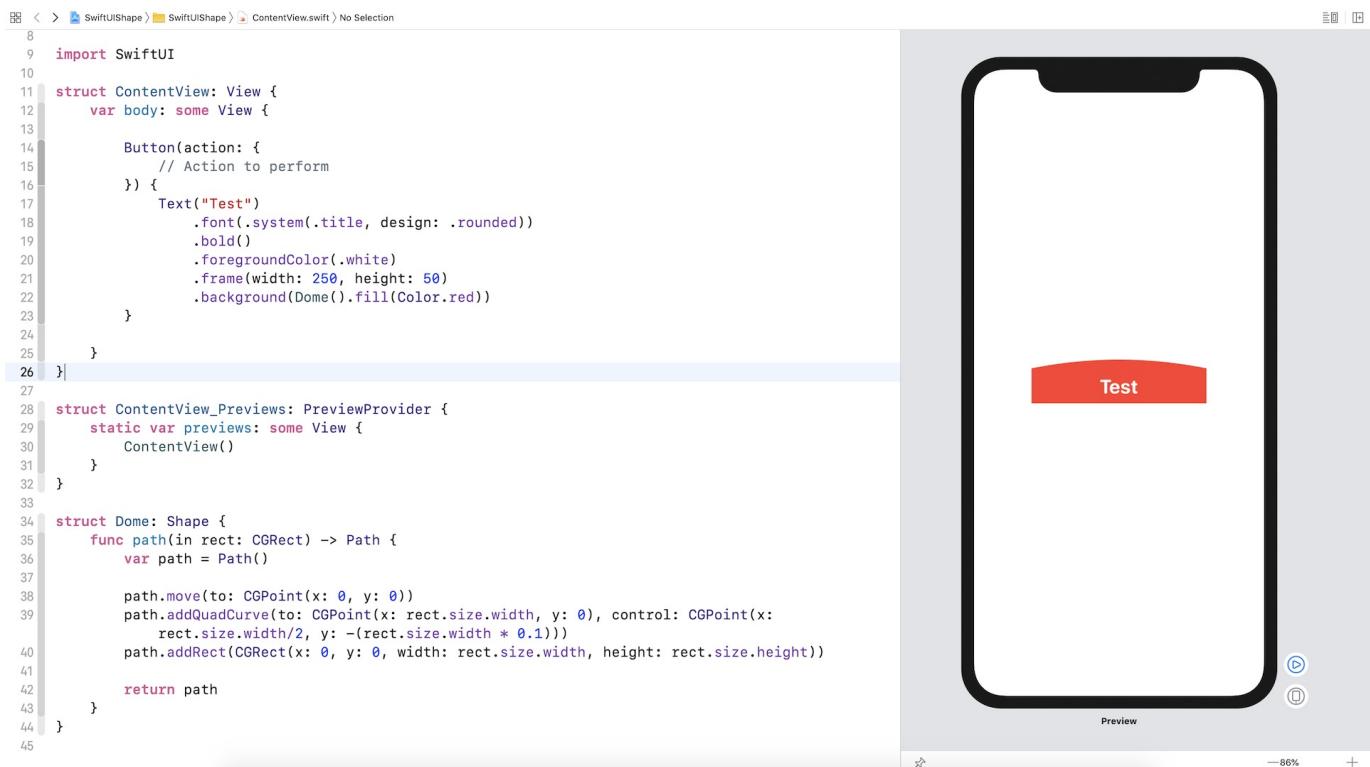
        return path
    }
}
```

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

With the shape, you can then use it to 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.



The screenshot shows the Xcode interface with the code editor open. The file is `ContentView.swift`. The code defines a `ContentView` struct containing a `Button` with a `Dome` background shape. To the right is a preview window showing a red button with rounded corners and a slight upward curve at the top, labeled "Test".

```
8 import SwiftUI
9
10 struct ContentView: View {
11     var body: some View {
12
13         Button(action: {
14             // Action to perform
15         }) {
16             Text("Test")
17                 .font(.system(.title, design: .rounded))
18                 .bold()
19                 .foregroundColor(.white)
20                 .frame(width: 250, height: 50)
21                 .background(Dome().fill(Color.red))
22         }
23     }
24 }
25 }
26 }
27
28 struct ContentView_Previews: PreviewProvider {
29     static var previews: some View {
30         ContentView()
31     }
32 }
33
34 struct Dome: Shape {
35     func path(in rect: CGRect) -> Path {
36         var path = Path()
37
38         path.move(to: CGPoint(x: 0, y: 0))
39         path.addQuadCurve(to: CGPoint(x: rect.size.width, y: 0), control: CGPoint(x:
40             rect.size.width/2, y: -(rect.size.width * 0.1)))
41         path.addRect(CGRect(x: 0, y: 0, width: rect.size.width, height: rect.size.height))
42
43         return path
44     }
45 }
```

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 some 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`.

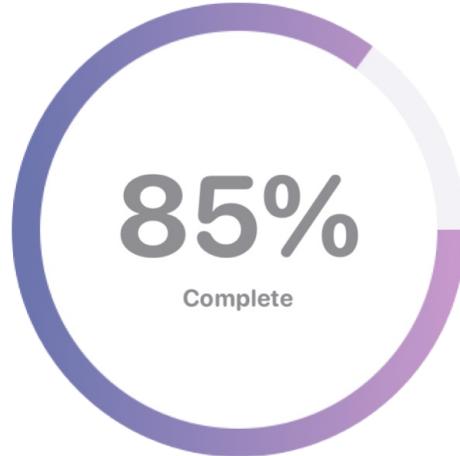


Figure 14. A progress indicator

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

```
struct ContentView: View {

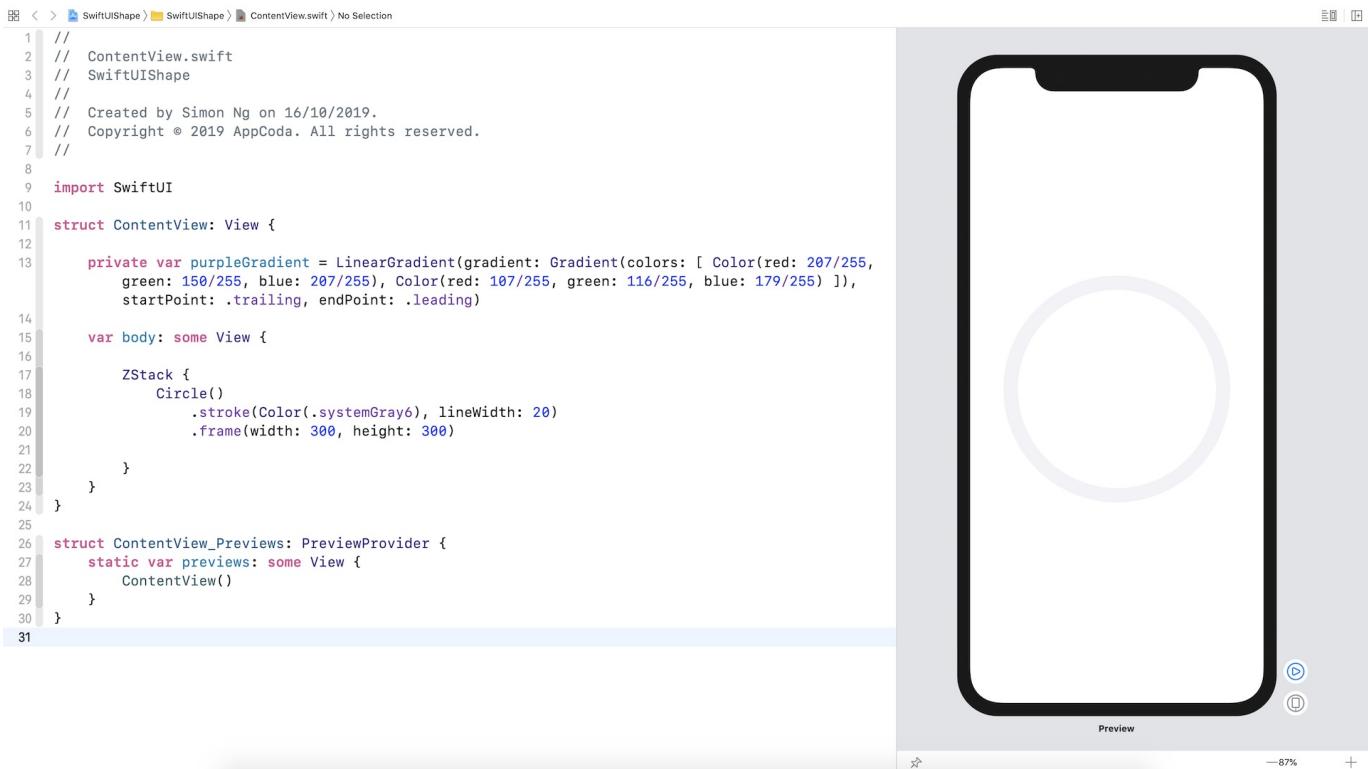
    private var purpleGradient = LinearGradient(gradient: Gradient(colors: [ Color(red: 207/255, green: 150/255, blue: 207/255), Color(red: 107/255, green: 116/255, blue: 179/255) ]), startPoint: .trailing, endPoint: .leading)

    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 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.



The screenshot shows the Xcode interface with the Swift code for a gray circle. On the left is the code editor with the file path "SwiftUIShape > ContentView.swift" and the content:

```
1 // ContentView.swift
2 // SwiftUIShape
3 //
4 //
5 // Created by Simon Ng on 16/10/2019.
6 // Copyright © 2019 AppCoda. All rights reserved.
7 //
8
9 import SwiftUI
10
11 struct ContentView: View {
12
13     private var purpleGradient = LinearGradient(gradient: Gradient(colors: [ Color(red: 207/255,
14         green: 150/255, blue: 207/255), Color(red: 107/255, green: 116/255, blue: 179/255) ]),
15         startPoint: .trailing, endPoint: .leading)
16
17     var body: some View {
18
19         ZStack {
20             Circle()
21                 .stroke(Color(.systemGray6), lineWidth: 20)
22                 .frame(width: 300, height: 300)
23         }
24     }
25
26     struct ContentView_Previews: PreviewProvider {
27         static var previews: some View {
28             ContentView()
29         }
30     }
31 }
```

On the right is the preview area showing an iPhone X simulator displaying a gray circle with a thick black outline. The preview window has standard Xcode controls at the bottom.

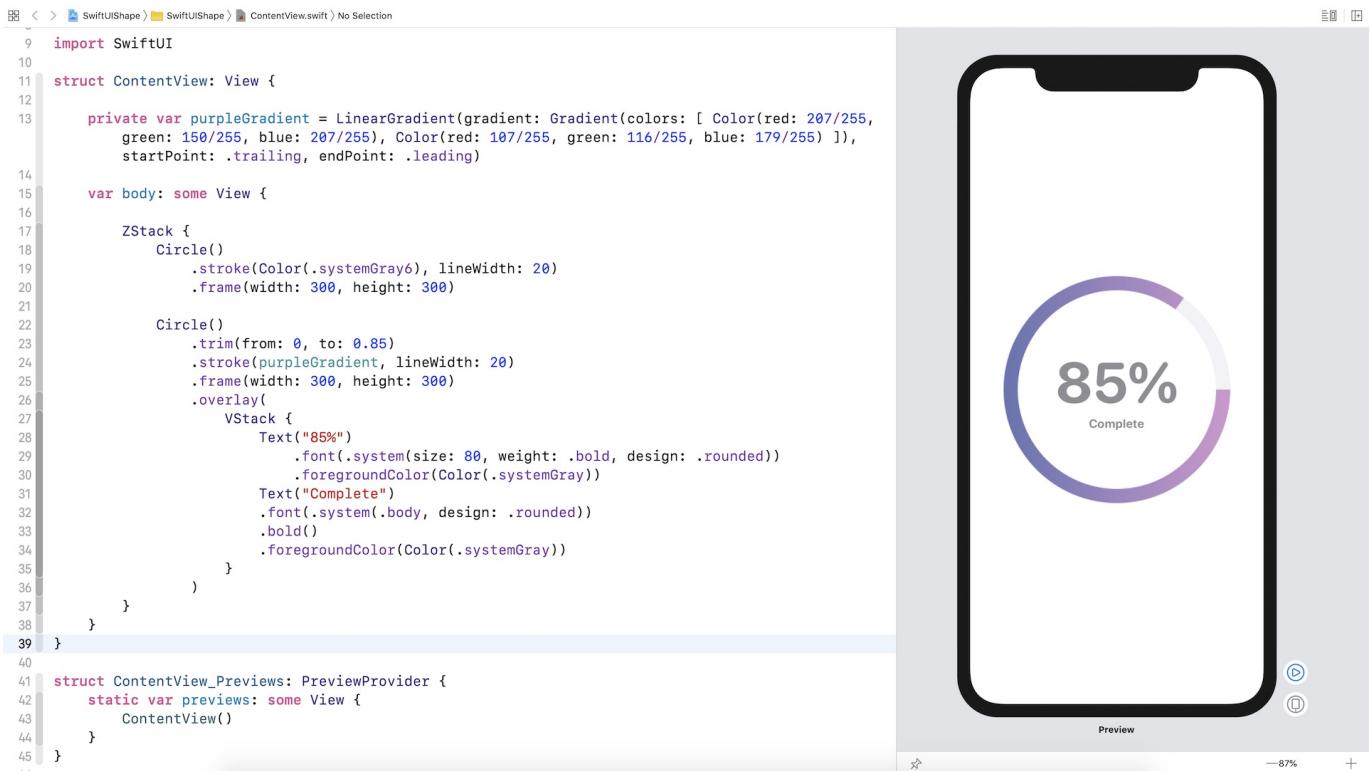
Figure 15. Drawing a gray circle

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))
        }
    )
)
```

The trick to create an open circle is by attaching 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 a 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.



The screenshot shows the Xcode interface with the Swift code for a progress view on the left and its preview on the right. The code uses ZStack, Circle, and trim to create a donut chart with a purple gradient. The preview shows a black iPhone displaying the chart with the text "85%" and "Complete".

```
9 import SwiftUI
10
11 struct ContentView: View {
12
13     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)
14
15     var body: some View {
16
17         ZStack {
18             Circle()
19                 .stroke(Color(.systemGray6), lineWidth: 20)
20                 .frame(width: 300, height: 300)
21
22             Circle()
23                 .trim(from: 0, to: 0.85)
24                 .stroke(purpleGradient, lineWidth: 20)
25                 .frame(width: 300, height: 300)
26             .overlay(
27                 VStack {
28                     Text("85%")
29                         .font(.system(size: 80, weight: .bold, design: .rounded))
30                         .foregroundColor(Color(.systemGray))
31                     Text("Complete")
32                         .font(.system(.body, design: .rounded))
33                         .bold()
34                         .foregroundColor(Color(.systemGray))
35                 }
36             )
37         }
38     }
39 }
40
41 struct ContentView_Previews: PreviewProvider {
42     static var previews: some View {
43         ContentView()
44     }
45 }
```

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)

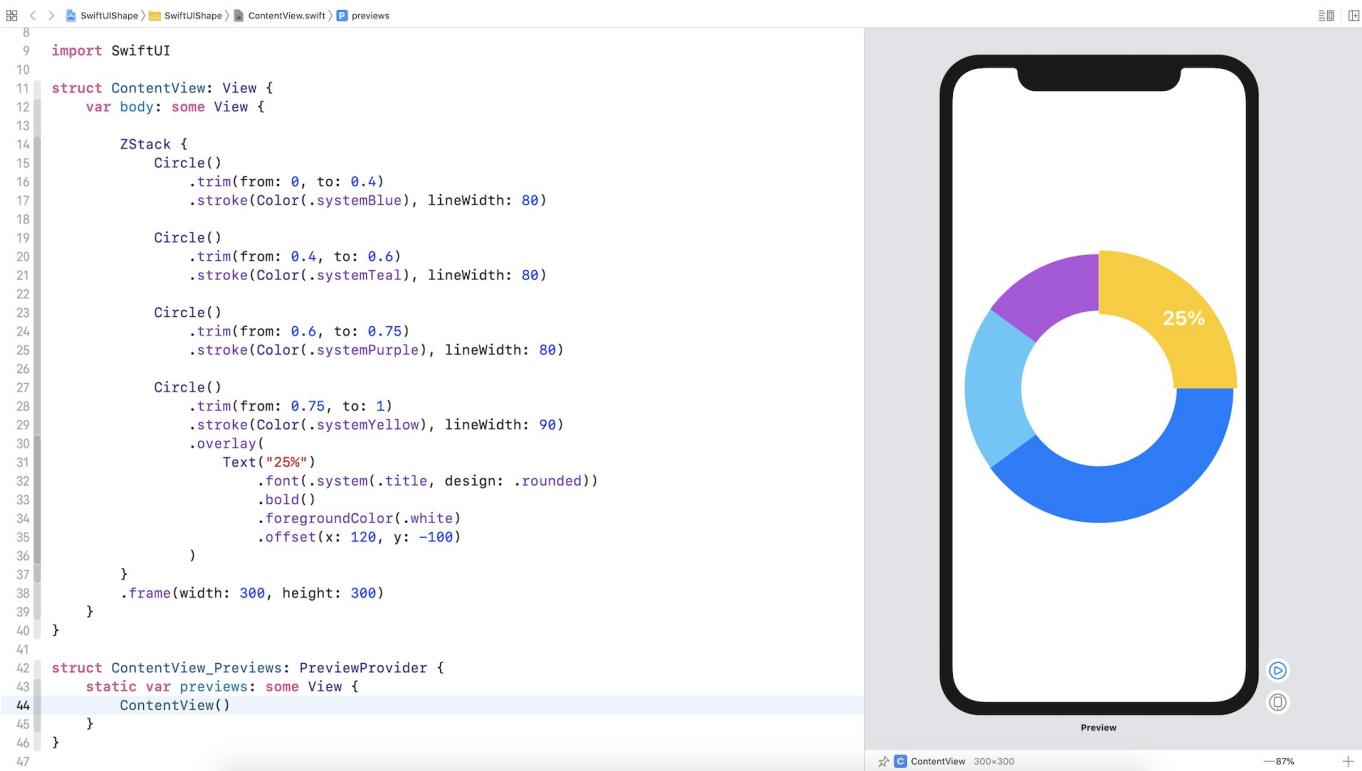
    Circle()
        .trim(from: 0.6, to: 0.75)
        .stroke(Color(.systemPurple), lineWidth: 80)

    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: 120, y: -100)
        )
}
.frame(width: 300, height: 300)

```

The first segment only show 40% of the circle. The second segment shows 20% of the circle, but note that the `from` value is 0.4 instead of 0. This allows the second segment connects with 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`.



The screenshot shows the Xcode interface with a Swift file named ContentView.swift open. The code defines a ContentView struct that returns a ZStack containing four concentric circles with different colors and stroke widths. An overlay on the innermost circle displays the text "25%" in bold white font. The preview pane on the right shows a smartphone displaying the donut chart with three visible segments: yellow (top), blue (bottom), and purple (left). The Xcode status bar at the bottom indicates the file is ContentView, the size is 300x300, and the zoom level is 87%.

```
8 import SwiftUI
9
10 struct ContentView: View {
11     var body: some View {
12
13         ZStack {
14             Circle()
15                 .trim(from: 0, to: 0.4)
16                 .stroke(Color(.systemBlue), lineWidth: 80)
17
18             Circle()
19                 .trim(from: 0.4, to: 0.6)
20                 .stroke(Color(.systemTeal), lineWidth: 80)
21
22             Circle()
23                 .trim(from: 0.6, to: 0.75)
24                 .stroke(Color(.systemPurple), lineWidth: 80)
25
26             Circle()
27                 .trim(from: 0.75, to: 1)
28                 .stroke(Color(.systemYellow), lineWidth: 90)
29             .overlay(
30                 Text("25%")
31                     .font(.system(.title, design: .rounded))
32                     .bold()
33                     .foregroundColor(.white)
34                     .offset(x: 120, y: -100)
35             )
36         }
37         .frame(width: 300, height: 300)
38     }
39 }
40
41 struct ContentView_Previews: PreviewProvider {
42     static var previews: some View {
43         ContentView()
44     }
45 }
46 }
```

Figure 17. Drawing the donut chart

Summary

I hope you enjoy reading this chapter and love the demo projects. With these drawing APIs provided by the framework, you can easily create custom shapes for your application. There are a lot you can do with Path and Shape. I just cover some of the techniques in this chapter, but try to apply what you've learned and make some magic!

For reference, you can download the sample project below:

- Demo project (<https://www.appcoda.com/resources/swiftui/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 these two states.

SwiftUI empowers you to animate changes for individual views and transitions between views. The framework already 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, you need to 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 a 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 a couple of the examples.

Implicit Animations

Let's begin with implicit animations. I suggest you to create a new project to see the animations in action. You can name the project to whatever name you like. For me, I name it `SwiftUIAnimation`.

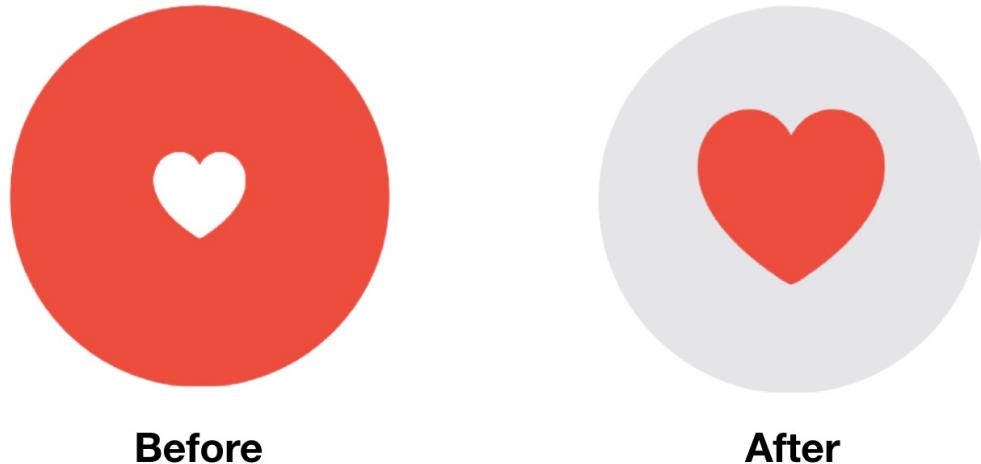


Figure 1. Animate a button's state change

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. So, 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.

If you implement the tappable circle using SwiftUI, this is what the code looks like:

```

struct ContentView: View {
    @State private var circleColorChanged = false
    @State private var heartColorChanged = false
    @State private var heartSizeChanged = false

    var body: some View {

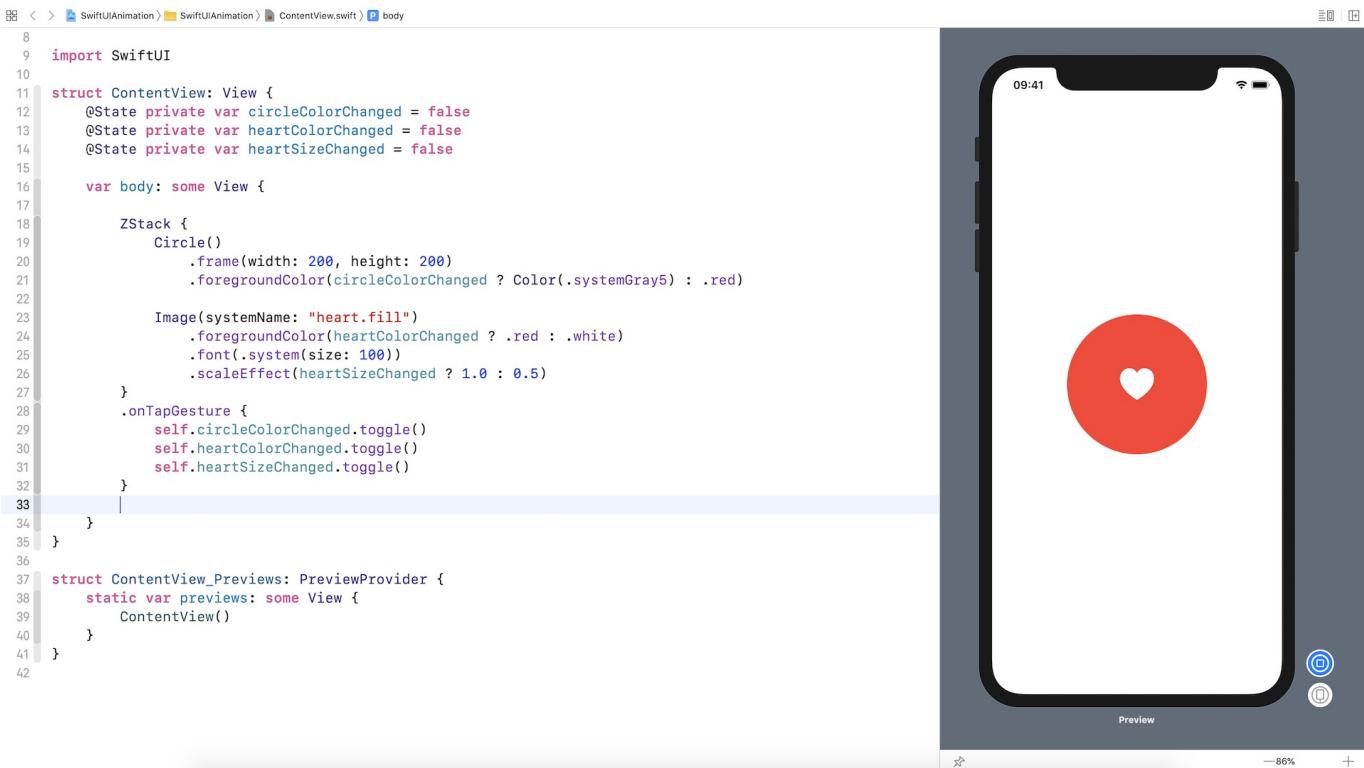
        ZStack {
            Circle()
                .frame(width: 200, height: 200)
                .foregroundColor(circleColorChanged ? Color(.systemGray5) : .red)

            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 with the initial 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 views to make it tappable. In the `onTapGesture` closure, we toggle the states to change the view's appearance.



```
8 import SwiftUI
9
10 struct ContentView: View {
11     @State private var circleColorChanged = false
12     @State private var heartColorChanged = false
13     @State private var heartSizeChanged = false
14
15     var body: some View {
16
17         ZStack {
18             Circle()
19                 .frame(width: 200, height: 200)
20                 .foregroundColor(circleColorChanged ? Color(.systemGray5) : .red)
21
22             Image(systemName: "heart.fill")
23                 .foregroundColor(heartColorChanged ? .red : .white)
24                 .font(.system(size: 100))
25                 .scaleEffect(heartSizeChanged ? 1.0 : 0.5)
26         }
27         .onTapGesture {
28             self.circleColorChanged.toggle()
29             self.heartColorChanged.toggle()
30             self.heartSizeChanged.toggle()
31         }
32     }
33 }
34
35 }
36
37 struct ContentView_Previews: PreviewProvider {
38     static var previews: some View {
39         ContentView()
40     }
41 }
```

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, all you need to do is attach the `animation` modifier to the `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 the `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 animation. 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. Well, you can still do that. You can attach the `animation(nil)` modifier to the view 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(nil) // Cancel the animation from here
        .scaleEffect(heartSizeChanged ? 1.0 : 0.5)
        .animation(.spring(response: 0.3, dampingFraction: 0.3, blendDuration: 0.3
    ))

    .onTapGesture {
        self.circleColorChanged.toggle()
        self.heartColorChanged.toggle()
        self.heartSizeChanged.toggle()
    }
}

```

We insert the `animation(nil)` 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 take care how the views are animated. All you need is to provide the start and end state. SwiftUI will then figure out the rest. If you understand this concept, you can create various types of animation.

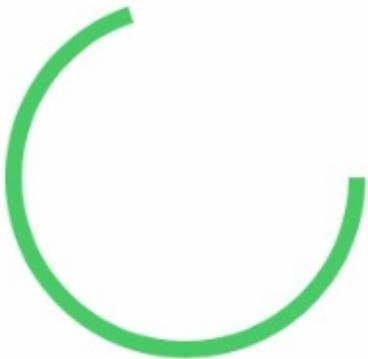


Figure 3. A sample loading indicator

For example, let's create a simple loading indicator that you can commonly find in some real-world application such as Medium. To create a loading indicator like that shown in figure 3, we can 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)
```

So, how can we keep rotating the circle? We can 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. 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 something difference. We tell SwiftUI to repeat the same animation again and again. This is the trick to create 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.linear(duration: 1).repeatForever(autoreverses: false)
```

The greater the duration the slower is the animation.

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. Say, for example, you can overlay an arc on a circle to create a fancy loading indicator.



Figure 4. A sample loading indicator

And, here is the code snippet:

```
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. But instead of changing the rotation angle, you can modify the value of the offset to create an animation like this.

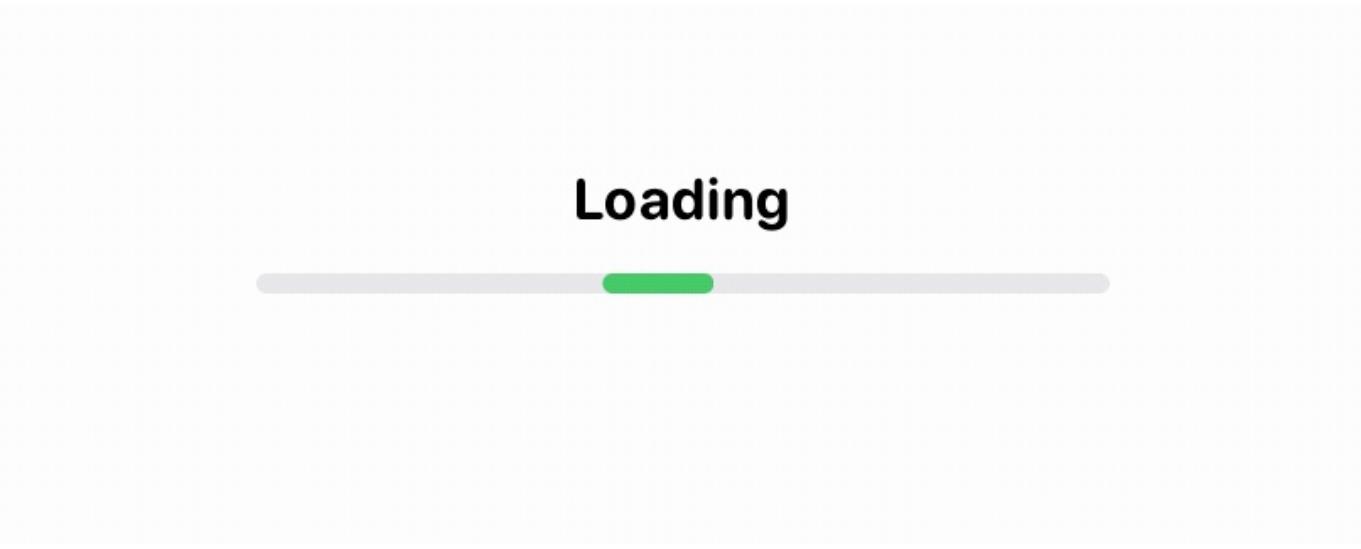


Figure 5. Another example of the loading indicator

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. And, when you repeat the same animation over and over again, it becomes a loading animation. Figure 6 illustrates the offset values.

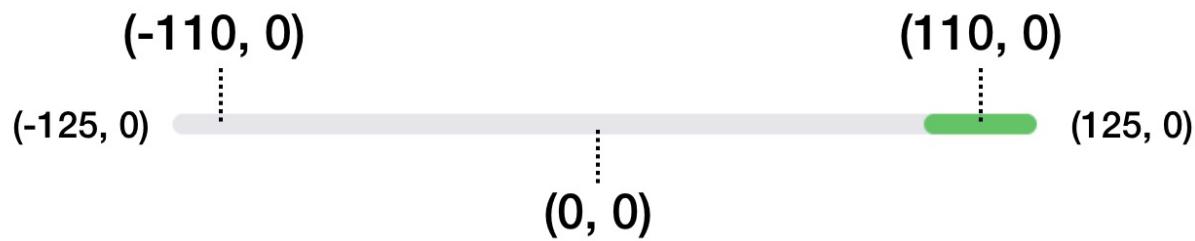


Figure 6. Another example of the loading indicator

Creating a Progress Indicator

The loading indicator provides some kinds of feedback to users indicating 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.

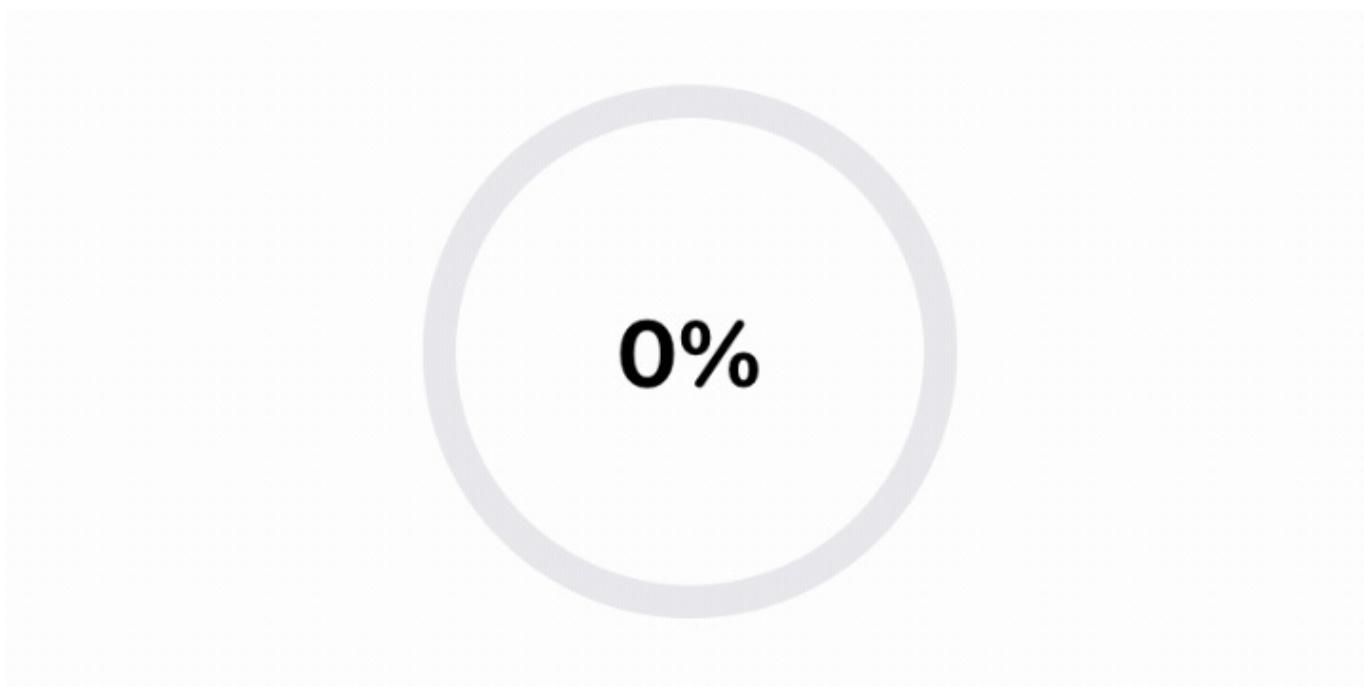


Figure 7. A progress indicator

The way of building a progress indicator is very similar to that of the loading indicator. But you need to state variable to keep track of the progress. Here is the 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 having a boolean state variable, we use a floating point number to store the status. To display the 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 demo, we just initiate a time 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 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 animation 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. Let's see 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().delay(0.2 * Double(index)))
            }
        }
        .onAppear() {
            self.isLoading = true
        }
    }
}

```

We first use a `HStack` to layout the circles horizontally. Since all the five circles (i.e. dots) are with 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 the ordinary 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 complication. Let's break it down and look into it step by step:

```
Animation.linear(duration: 0.6).repeatForever().delay(0.2 * Double(index))
```

The first part creates a linear animation with the 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.

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). How can it be implemented? With the built-in shape and animation, you can easily create such transformation like the one 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. So, here is the implementation of the morphing button:

```

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 : 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 to realize 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. And, SwiftUI automatically renders the animation of this transformation.

The `recording` state variable, on the other hand, 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 grow and shrink 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 has been existed in the view hierarchy. We create animation to scale it up and down. Or we animate the view's size.

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 transition together to create your desired transition.

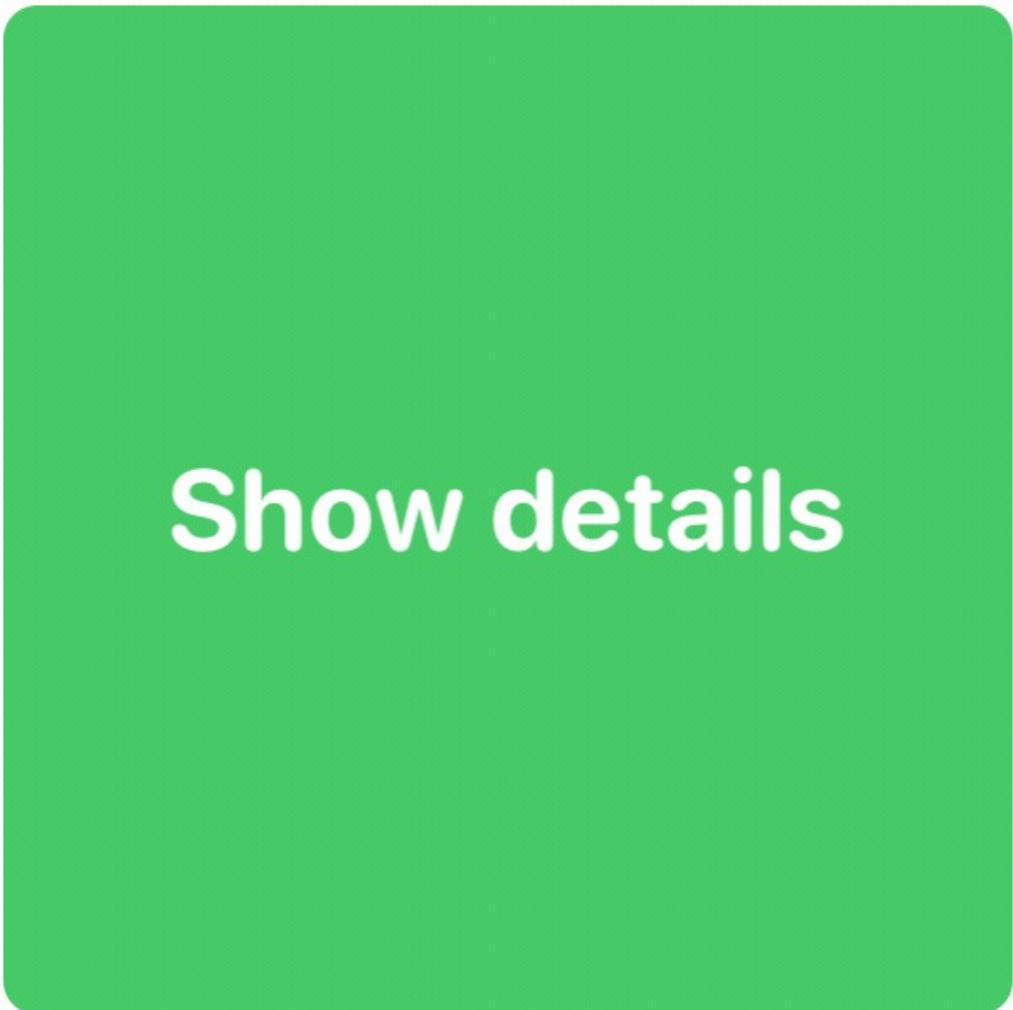


Figure 10. A sample transition created using SwiftUI

Building a Simple Transition

Let's take a look at a simple example to better understand what a transition means 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 rectangles vertically using `vStack`. What I want to do is to make the stack tappable. At first, the purple rectangle should be hidden. It's displayed only when a user taps the green rectangle (i.e. Show details).

The screenshot shows the Xcode interface with the Swift code for `ContentView`. The code defines a `ContentView` struct that contains a `body` property. This property is a `VStack { }` block that stacks two `RoundedRectangle` components vertically. The top rectangle is green and contains the text "Show details". The bottom rectangle is purple and contains the text "Well, here is the details". Both rectangles have a `frame(width: 300, height: 300)`, a `cornerRadius: 10`, and a white foreground color. The preview window on the right shows an iPhone displaying this UI.

```
8 import SwiftUI
9
10 struct ContentView: View {
11     var body: some View {
12         VStack {
13             RoundedRectangle(cornerRadius: 10)
14                 .frame(width: 300, height: 300)
15                 .foregroundColor(.green)
16                 .overlay(
17                     Text("Show details")
18                         .font(.system(.largeTitle, design: .rounded))
19                         .bold()
20                         .foregroundColor(.white)
21                 )
22
23         )
24
25         RoundedRectangle(cornerRadius: 10)
26             .frame(width: 300, height: 300)
27             .foregroundColor(.purple)
28             .overlay(
29                 Text("Well, here is the details")
30                     .font(.system(.largeTitle, design: .rounded))
31                     .bold()
32                     .foregroundColor(.white)
33             )
34     }
35 }
36
37 }
38 }
39
40 struct ContentView_Previews: PreviewProvider {
41     static var previews: some View {
42         ContentView()
43     }
44 }
```

Figure 11. Layout two rectangles vertically

To do that, we need to declare a state variable to determine whether the purple rectangle is shown or not. Insert this line of code in `ContentView` :

```
@State private var show = false
```

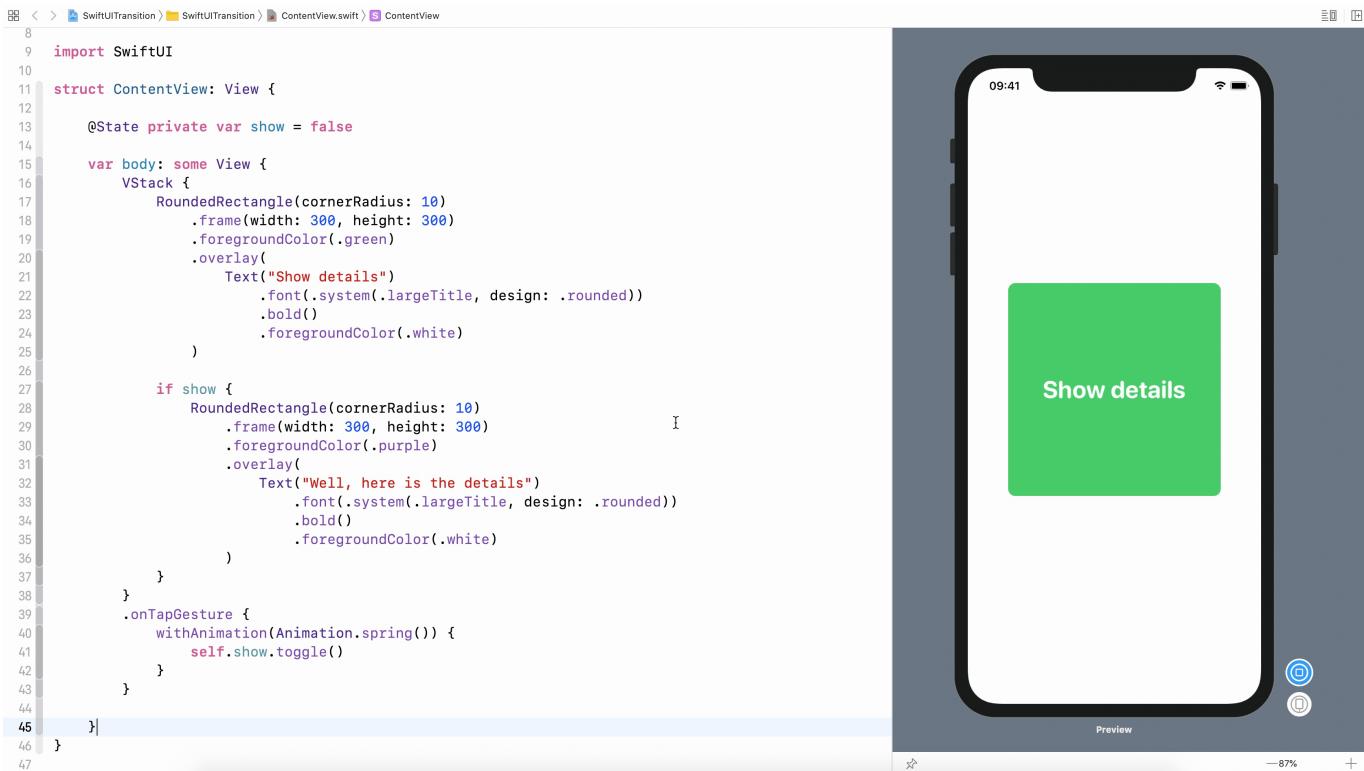
Next, we wrap the purple rectangle within a `if` clause like this:

```
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 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 rectangle. If you run the app in the simulator or the preview canvas, you should only see the green rectangle. Tapping it will display the purple rectangle and you should see a smooth fade in/out transition.



```
8 import SwiftUI
9
10 struct ContentView: View {
11     @State private var show = false
12
13     var body: some View {
14         VStack {
15             RoundedRectangle(cornerRadius: 10)
16                 .frame(width: 300, height: 300)
17                 .foregroundColor(.green)
18                 .overlay(
19                     Text("Show details")
20                         .font(.system(.largeTitle, design: .rounded))
21                         .bold()
22                         .foregroundColor(.white)
23                 )
24
25         }
26
27         if show {
28             RoundedRectangle(cornerRadius: 10)
29                 .frame(width: 300, height: 300)
30                 .foregroundColor(.purple)
31                 .overlay(
32                     Text("Well, here is the details")
33                         .font(.system(.largeTitle, design: .rounded))
34                         .bold()
35                         .foregroundColor(.white)
36                 )
37         }
38     }
39     .onTapGesture {
40         withAnimation(Animation.spring()) {
41             self.show.toggle()
42         }
43     }
44 }
45 }
```

Figure 12. Default 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 rectangle 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 set to 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 and see what you get. You should see a pop animation when the app reveals the purple rectangle.

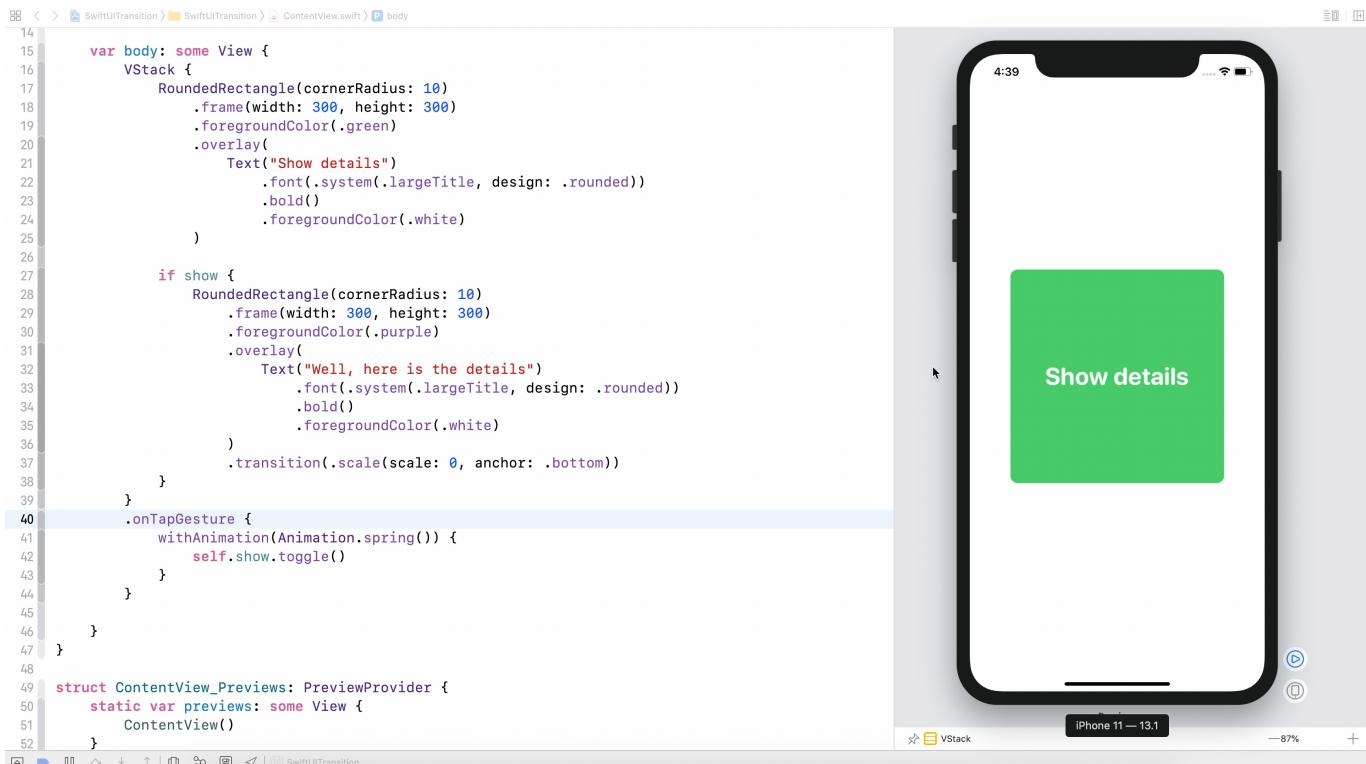


Figure 13. Scaling transition

Other than `.scale`, the SwiftUI framework comes with several built-in transitions including `.opaque`, `.offset`, `.move`, and `.slide`. Try to replace the scale transition with the offset transition like this:

```
.transition(.offset(x: -600, y: 0))
```

This time, the purple rectangle 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 can write the code like this:

```
.transition(AnyTransition.offset(x: -600, y: 0).combined(with: .scale))
```

Here is another sample line of code if you need to combine three transitions:

```
.transition(AnyTransition.offset(x: -600, y: 0).combined(with: .scale).combined(with: .opacity))
```

In some cases, if 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?

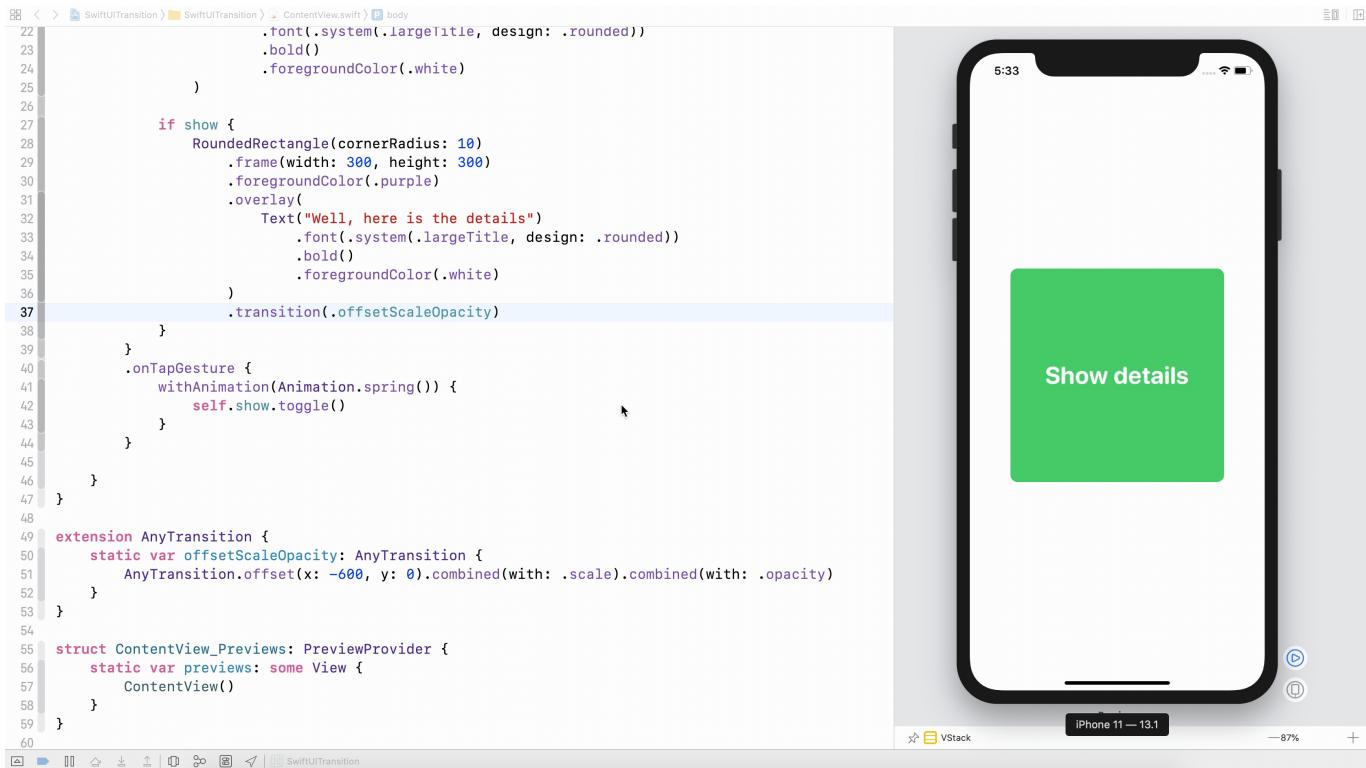


Figure 14. Combining the scale, offset, and opacity transition

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 *Assymmetric Transitions* in SwiftUI. It's very simple to use this type of transition. You just need to call the `.asymmetric` 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)
        )
    }
}
```

After you make the change in your code, run the app using the built-in simulator. You should see the `scale` transition when the purple rectangle appears on screen. When you tap the rectangles again, the purple rectangle will slide off the screen.

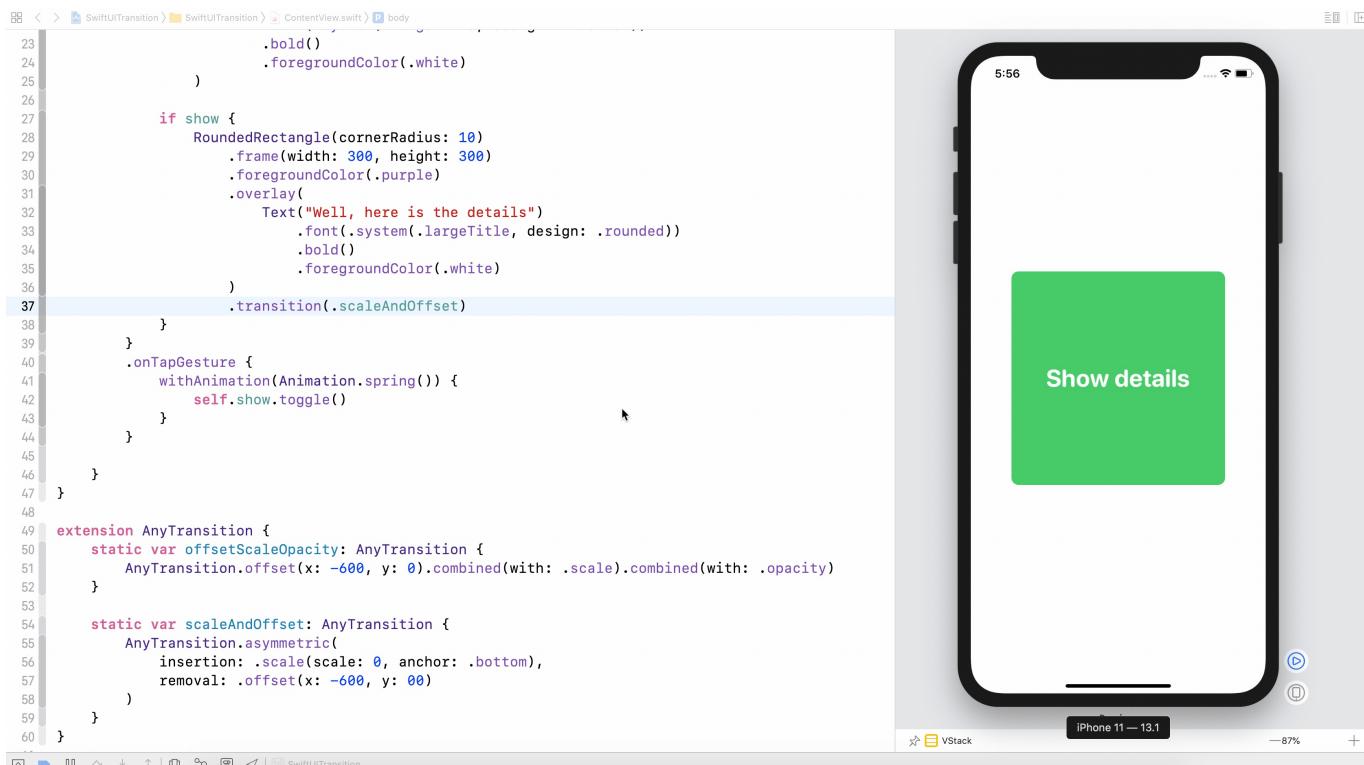


Figure 15. Assymmetric transition demo

Exercise #1: Using Animation and Transition to Build a Fancy Button

Now that you should have some ideas about 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 take a look.

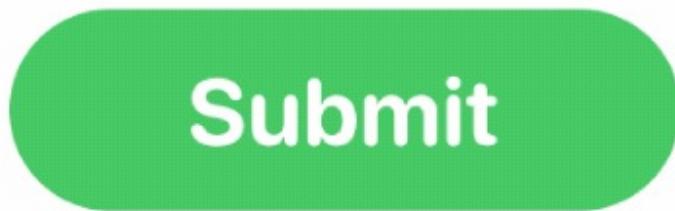


Figure 16. A fancy button

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 challenge project that tests your knowledge on 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. As a hint, I use the following code to simulate the 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 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.

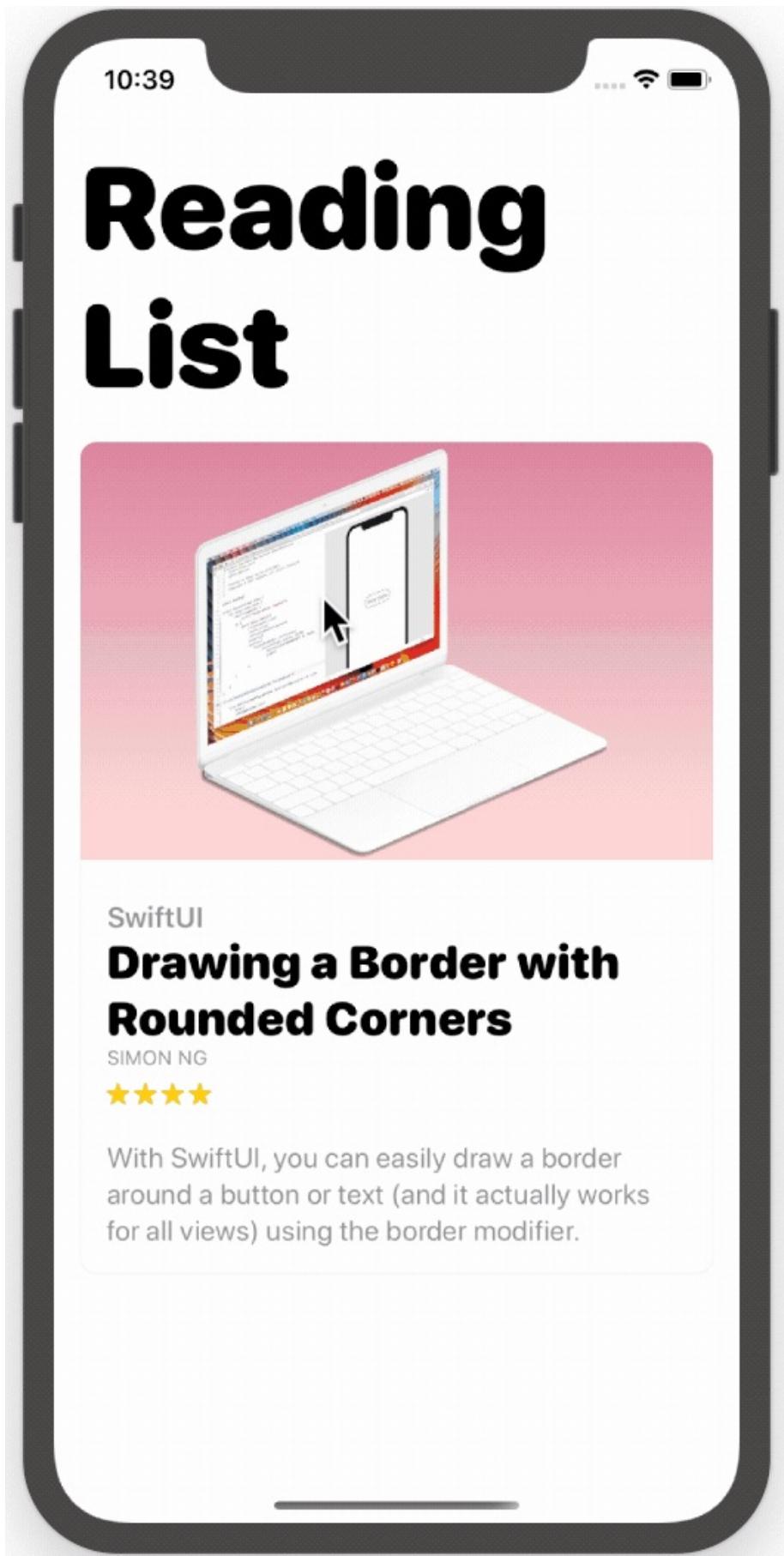


Figure 17. Animated view transition

If you can't view the animation above, you can click this link (<https://www.appcoda.com/wp-content/uploads/2019/10/swiftui-view-animation.gif>) to check out the animated effect.

Summary

Animation has a special role in mobile UI design. Well thought out animation improves user experience and brings meaning to the UI interaction. A smooth and effortless transition between two views would 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, well-designed UI and animation would definitely make a fundamental difference.

Having that said, even for experienced developers, it's not an easy task to code a slick animation. Fortunately, the SwiftUI framework has simplified the development of UI animation and transition. You tell the framework how the view should look like at the beginning and the end. SwiftUI figures out the rest, rendering a smooth and nice animation.

In this chapter, I've just walked you through the basics. But as you can see, you've already built some delightful animations and transitions. Most importantly, it just need 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 solution to exercise below:

- Demo projects & Exercise #1
(<https://www.appcoda.com/resources/swiftui/SwiftUIAnimation.zip>)
- Exercise #2
(<https://www.appcoda.com/resources/swiftui/SwiftUICardAnimation.zip>)

Chapter 10

Understanding Dynamic List, ForEach and Identifiable

In UIKit, table view is one of the most common UI controls in iOS. If you've developed apps with UIKit before, you should 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.

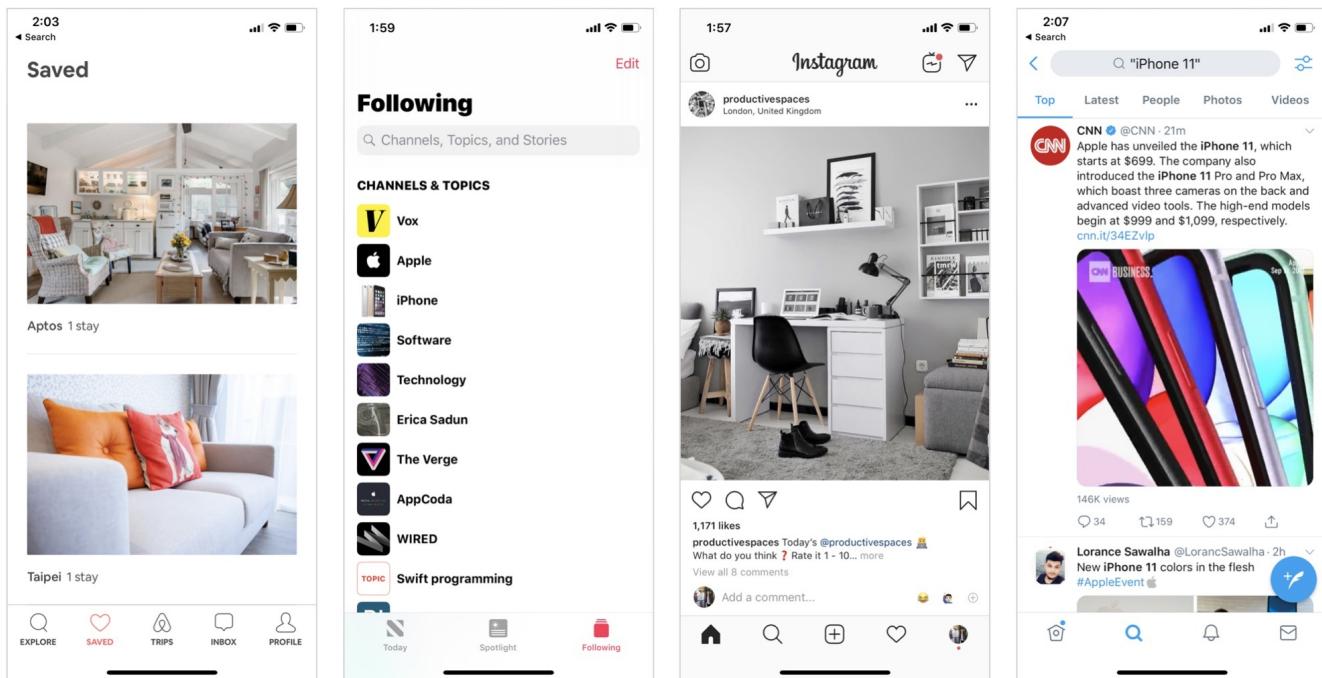


Figure 1. Sample list views

Instead of using a table view, we use `List` in SwiftUI 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 efforts for building 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 efforts.

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.

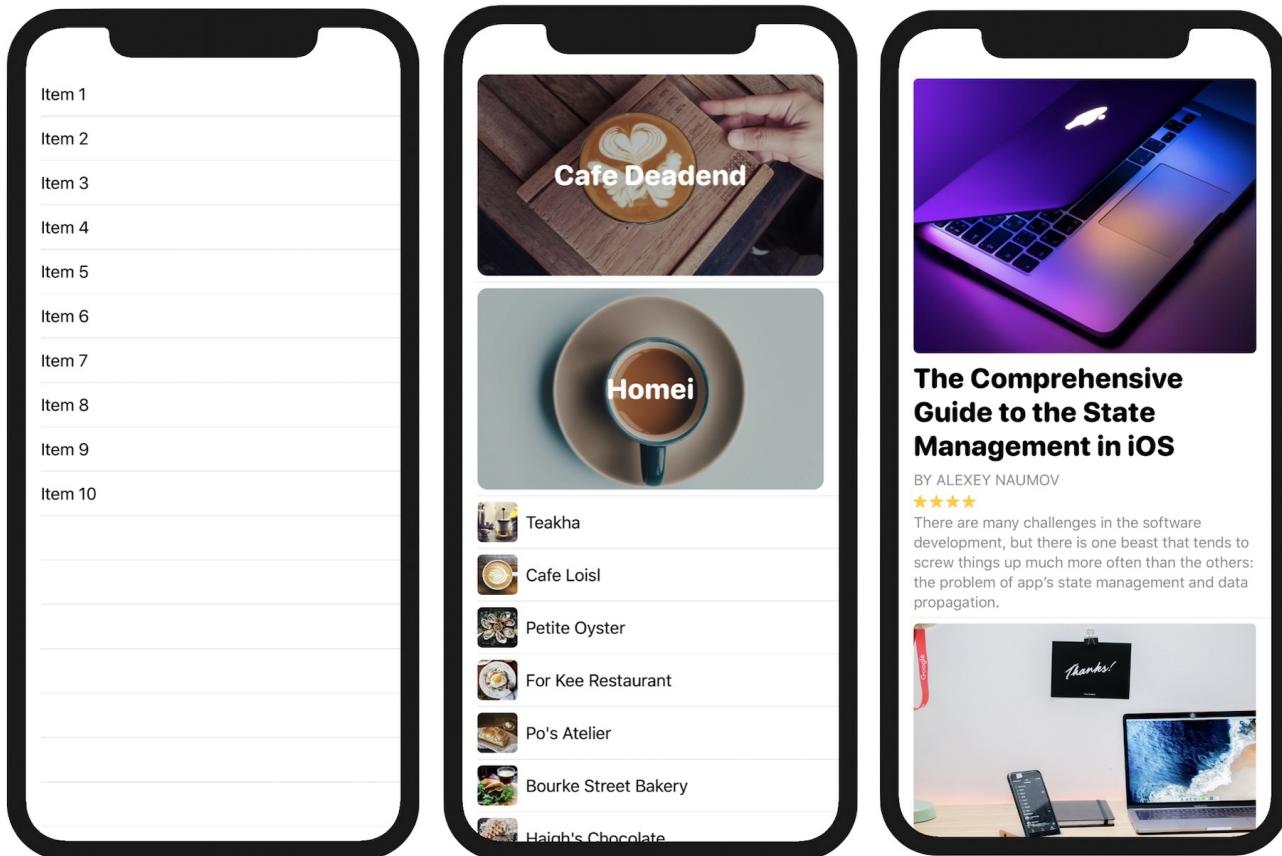


Figure 2. Building a simple and complex list

Creating a Simple List

Let's begin with a simple list. First, fire up Xcode and create a new project using the *Single View Application* template. In the next screen, set the product name to `SwiftUIList` (or whatever name you like) and fill in all the required values. Just make

sure you select `SwiftUI` for the *User Interface* option.

Xcode should generate some code in the `ContentView.swift` file. Now update the code like this:

```
struct ContentView: View {  
    var body: some View {  
        List {  
            Text("Item 1")  
            Text("Item 2")  
            Text("Item 3")  
            Text("Item 4")  
        }  
    }  
}
```

That's 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.

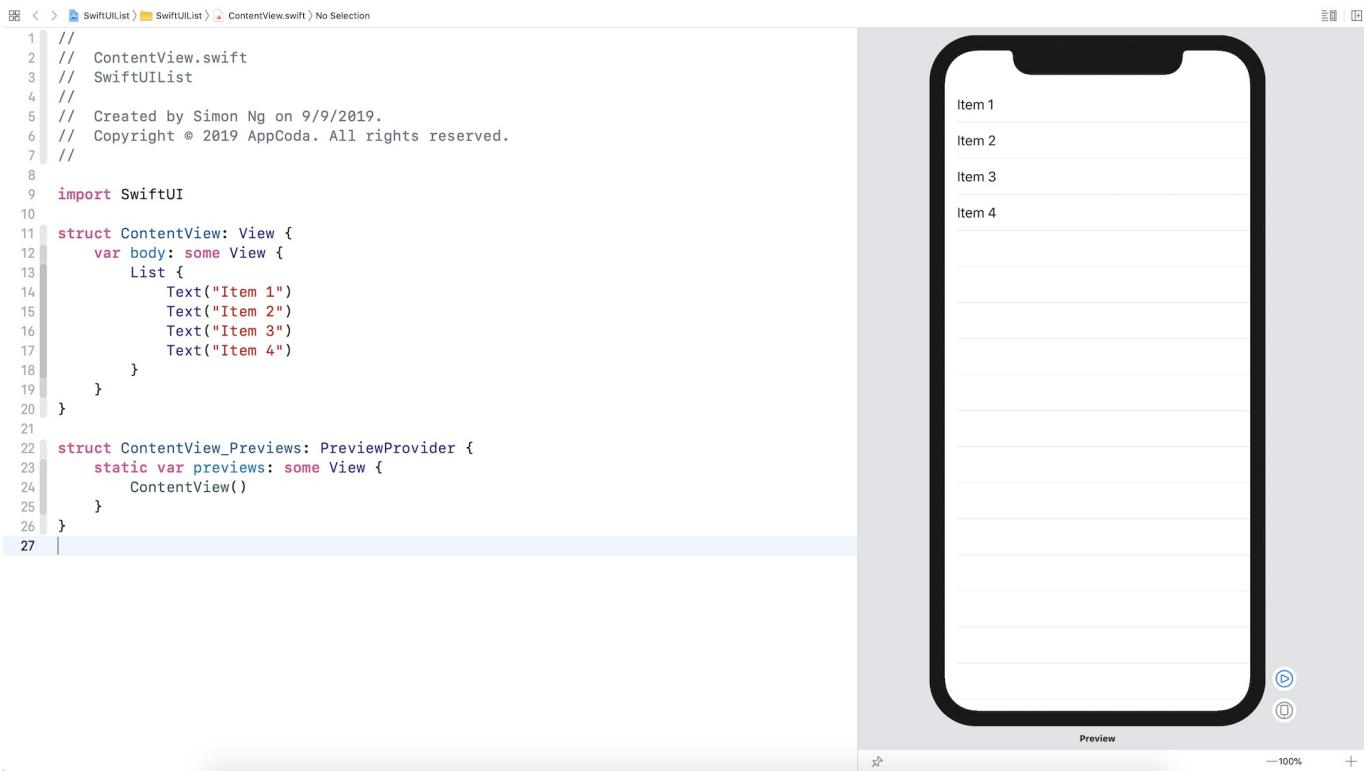


Figure 3. Creating a simple list

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 loop.

A structure that computes views on demand from an underlying collection 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 is that you need to tell `ForEach` how to identify each of the item 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, so it needs an identifier to 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.

In the closure, it's the code you need to render the views. Here, it's the text view we need to create. Its description will change depending on the value of `index` in the loop. That's how you can 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 simpler. 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 just 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 can you implement that? If you know how `Image` , `Text` , `VStack` , and `HStack` work, you should have some ideas about how to create one.

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.

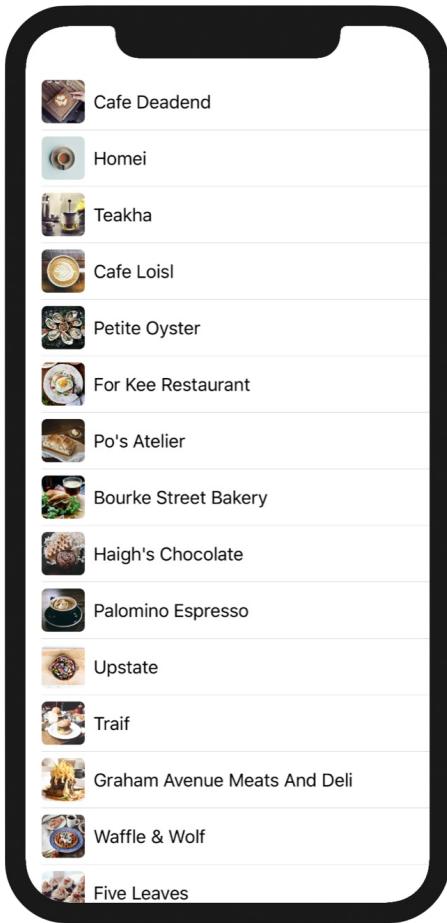


Figure 4. A simple table view showing rows of restaurants

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. Now, let's see how the same table view can be 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.

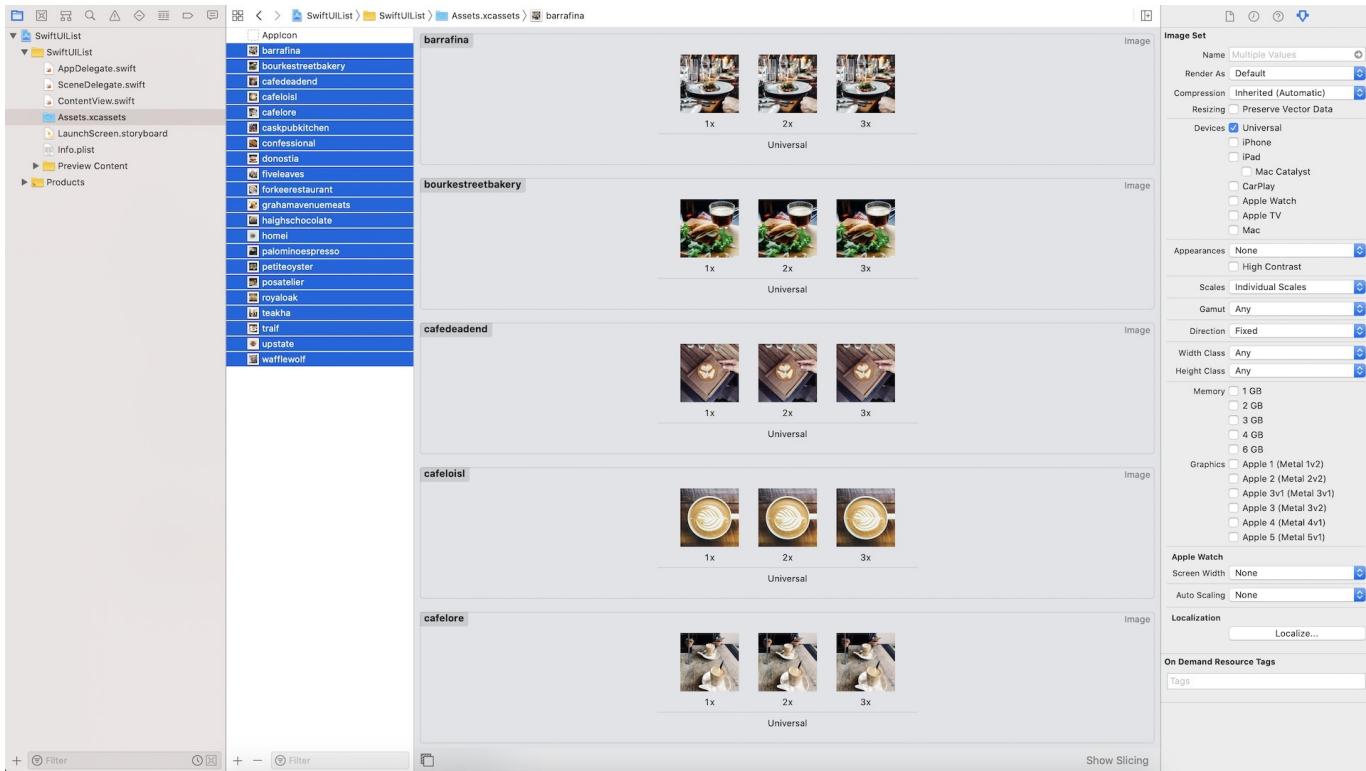


Figure 5. Import images to the asset catalog

Now switch over to `ContentView.swift` to code the UI. First, let's declare two arrays in `ContentView`. These arrays are for storing the 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", "Donostia", "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. While the `restaurantNames` array stores the name of the restaurants, the `restaurantImages` variable 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, it's the `List` view. Instead of a fixed range, we pass the range of the restaurant names (i.e. `restaurantNames.indices`). Say, the `restaurantNames` array has 21 items. We'll have a range from 0 to 20.

In the closure, the code was updated to create the row layout. I'll not go into the details as the code is self-explanatory if you fully understand stack views. With less than 10 lines of code, we can create a list (or table) view with custom layout.

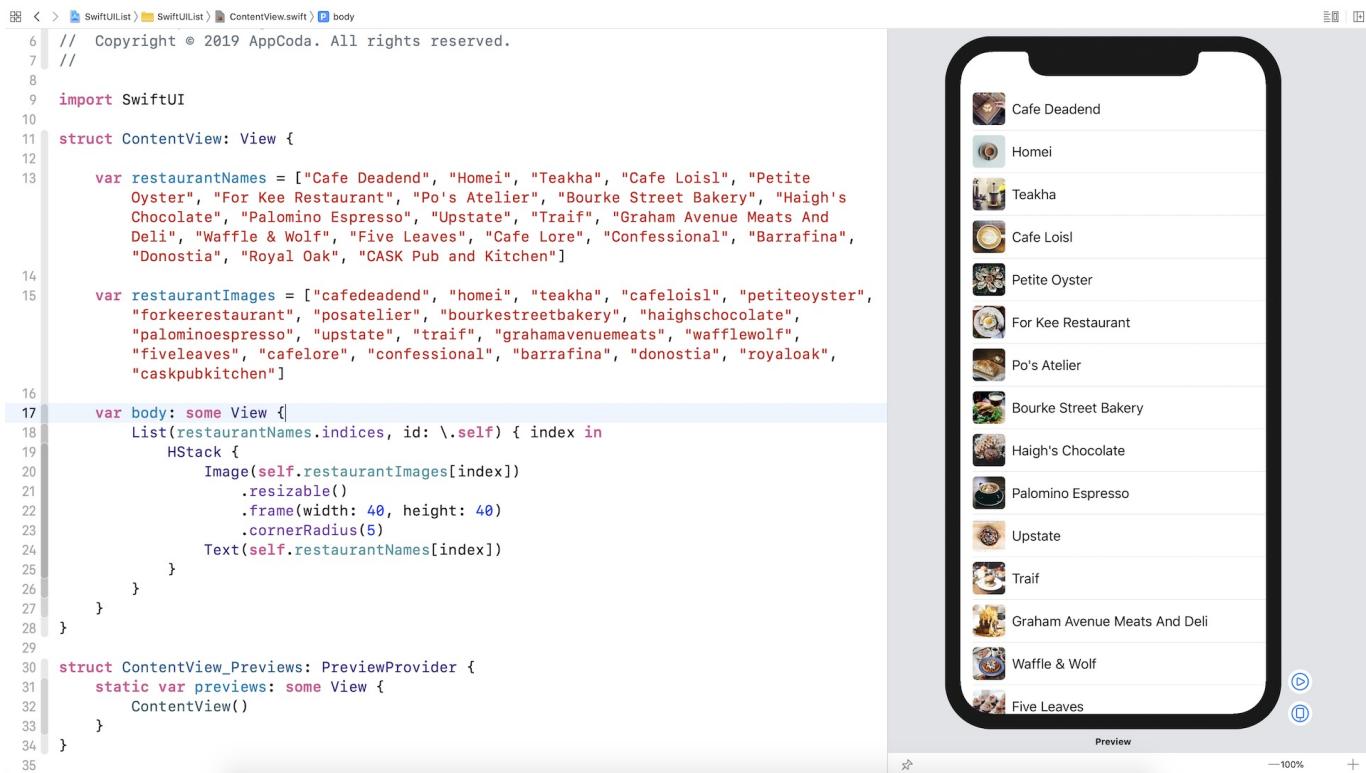


Figure 6. A list view with custom row layout

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 data.

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. Now delete the `restaurantNames` and `restaurantImages` variable 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  
ueemeats"),  
    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 a record of a specific restaurant. Once you made the change, you'll see an error in Xcode, complaining the missing of the `restaurantNames` variable. That's normal 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 instruct the list how we want to present the restaurant row. Here, we simply present both the restaurant image and name in a `HStack`.

Everything is unchanged. The UI is still the same but the underlying code was modified to utilize `List` with a collection of data.

The screenshot shows the Xcode interface with the ContentView.swift file open. The code defines a Content View that contains a List of restaurants. Each restaurant item is represented by a HStack containing an image and a text label. The preview window shows a vertical list of 15 restaurant items, each with a small image icon and its name: 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, and Five Leaves.

```

11 struct ContentView: View {
12
13     var restaurants = [ Restaurant(name: "Cafe Deadend", image: "cafedeadend"),
14         Restaurant(name: "Homei", image: "homei"),
15         Restaurant(name: "Teakha", image: "teakha"),
16         Restaurant(name: "Cafe Loisl", image: "cafeloisl"),
17         Restaurant(name: "Petite Oyster", image: "petiteoyster"),
18         Restaurant(name: "For Kee Restaurant", image: "forkeerrestaurant"),
19         Restaurant(name: "Po's Atelier", image: "posatelier"),
20         Restaurant(name: "Bourke Street Bakery", image: "bourkestreetbakery"),
21         Restaurant(name: "Haigh's Chocolate", image: "haighschocolate"),
22         Restaurant(name: "Palomino Espresso", image: "palominoespresso"),
23         Restaurant(name: "Upstate", image: "upstate"),
24         Restaurant(name: "Traif", image: "traif"),
25         Restaurant(name: "Graham Avenue Meats And Deli", image:
26             "grahamavenuemeats"),
27         Restaurant(name: "Waffle & Wolf", image: "wafflewolf"),
28         Restaurant(name: "Five Leaves", image: "fiveleaves"),
29         Restaurant(name: "Cafe Lore", image: "cafelore"),
30         Restaurant(name: "Confessional", image: "confessional"),
31         Restaurant(name: "Barrafina", image: "barrafina"),
32         Restaurant(name: "Donostia", image: "donostia"),
33         Restaurant(name: "Royal Oak", image: "royaloak"),
34         Restaurant(name: "CASK Pub and Kitchen", image: "caskpubkitchen")
35     ]
36
37     var body: some View {
38         List(restaurants, id: \.name) { restaurant in
39             HStack {
40                 Image(restaurant.image)
41                     .resizable()
42                     .frame(width: 40, height: 40)
43                     .cornerRadius(5)
44                 Text(restaurant.name)
45             }
46         }
47     }
48 }

```

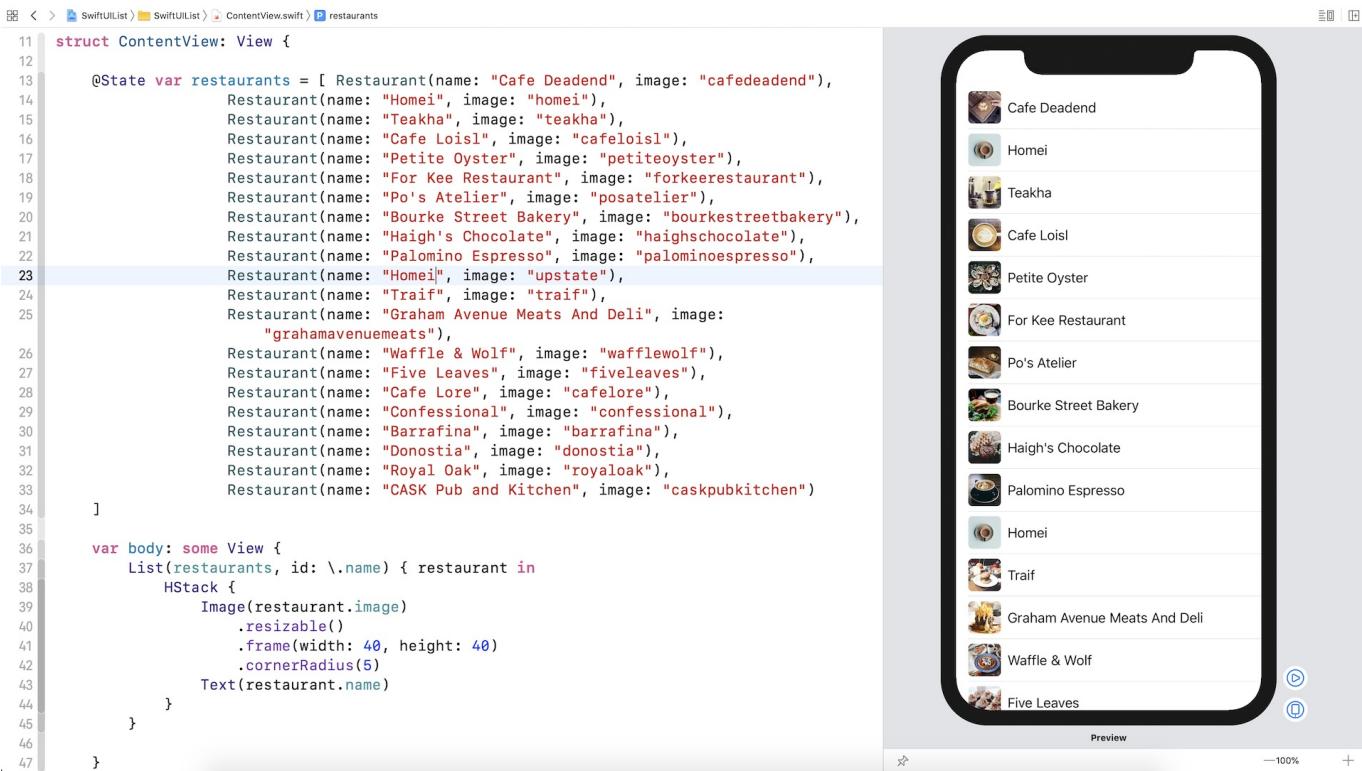
Figure 7. Same UI as figure 6

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. Since we use the name of the restaurant as the identifier, let's see what happens when we have two records with the same restaurant name. Now change `Upstate` to `Homei` in the `restaurants` array like this:

```
Restaurant(name: "Homei", image: "upstate")
```

Please take note that we only change the value of the `name` property and keep the image to `upstate`. The preview canvas should render the view automatically. But in case you see the message "Automatic preview updating paused", click the *Resume* button to reload.



```

11 struct ContentView: View {
12
13     @State var restaurants = [ Restaurant(name: "Cafe Deadend", image: "cafedeadend"),
14         Restaurant(name: "Homei", image: "homei"),
15         Restaurant(name: "Teakha", image: "teakha"),
16         Restaurant(name: "Cafe Loisl", image: "cafeois1"),
17         Restaurant(name: "Petite Oyster", image: "petiteoyster"),
18         Restaurant(name: "For Kee Restaurant", image: "forkeerestaurant"),
19         Restaurant(name: "Po's Atelier", image: "posatelier"),
20         Restaurant(name: "Bourke Street Bakery", image: "bourkestreetbakery"),
21         Restaurant(name: "Haigh's Chocolate", image: "haighschocolate"),
22         Restaurant(name: "Palomino Espresso", image: "palominoespresso"),
23         Restaurant(name: "Homei", image: "upstate"),
24         Restaurant(name: "Traif", image: "traif"),
25         Restaurant(name: "Graham Avenue Meats And Deli", image:
26             "grahamavenuemeats"),
27         Restaurant(name: "Waffle & Wolf", image: "wafflewolf"),
28         Restaurant(name: "Five Leaves", image: "fiveleaves"),
29         Restaurant(name: "Cafe Lore", image: "cafelore"),
30         Restaurant(name: "Confessional", image: "confessional"),
31         Restaurant(name: "Barrafina", image: "barrafina"),
32         Restaurant(name: "Donostia", image: "donostia"),
33         Restaurant(name: "Royal Oak", image: "royaloak"),
34         Restaurant(name: "CASK Pub and Kitchen", image: "caskpubkitchen")
35     ]
36
37     var body: some View {
38         List(restaurants, id: \.name) { restaurant in
39             HStack {
40                 Image(restaurant.image)
41                     .resizable()
42                     .frame(width: 40, height: 40)
43                     .cornerRadius(5)
44                 Text(restaurant.name)
45             }
46         }
47     }
}

```

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 may expect the second *Homei* record would show the *upstate* image. In reality, 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 the two restaurants have the same name, iOS would consider both restaurants as 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 ID, you should give each restaurant a unique identifier. Now 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 initialize 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 for having two same identifiers is almost zero.

Now each restaurant should have a unique ID, but we still have to make one more change before the bug is fixed. 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 again. The second *Homei* record should show its own image.

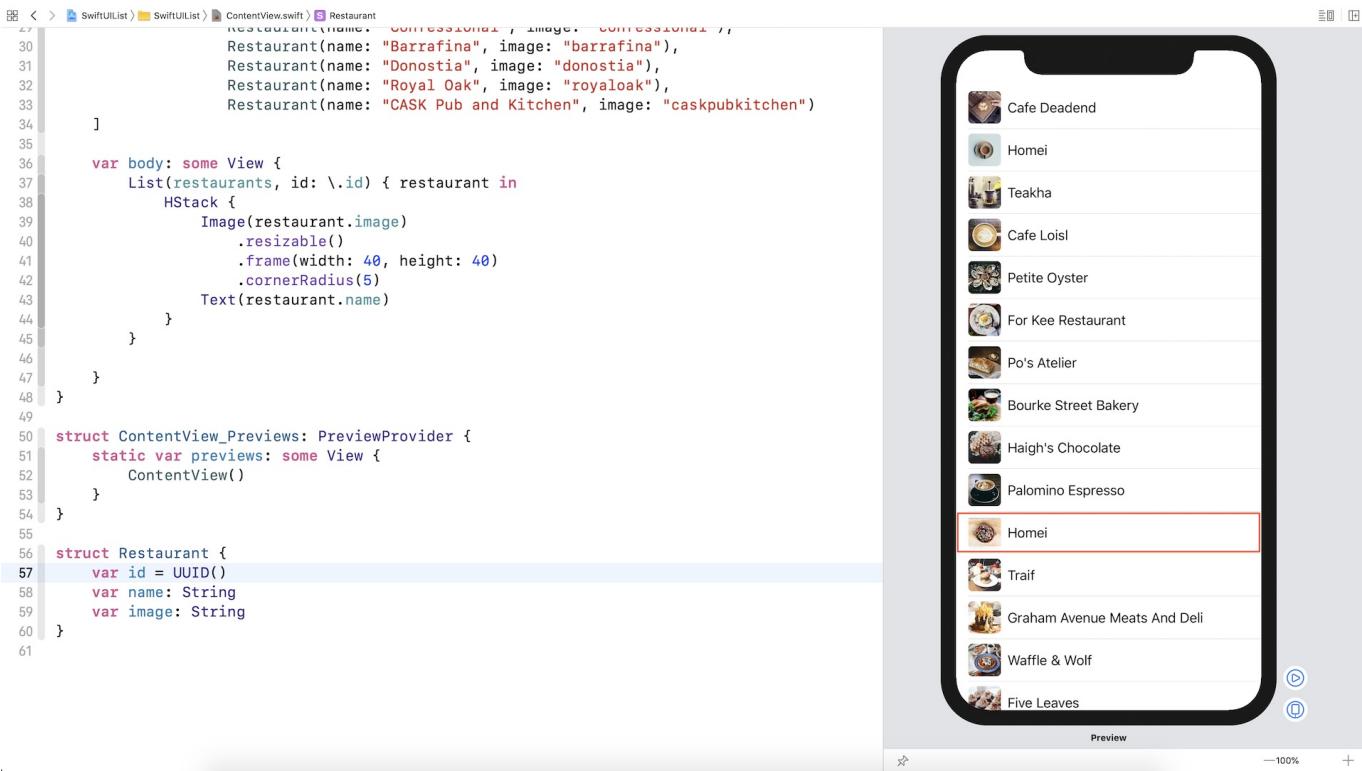


Figure 9. The bug is now fixed showing the correct image

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 sorts of `id` as the unique identifier. Now 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.

So, what's the purpose of implementing the `Identifiable` protocol here? It can further save you some code. When the `Restaurant` struct conforms to the `Identifiable` protocol, you can initialize the `List` without the `id` parameter. 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 to refactor it to make it even better. You've learned how to extract a view. Now we'll 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`.

```
22     Restaurant(name: "Palomino Espresso", image: "palominoespresso"),
23             t(name: "Homei", image: "upstate"),
24             t(name: "Traif", image: "traif"),
25             t(name: "Graham Avenue Meats And Deli", image:
26                 amavenuemeats"),
27                 t(name: "Waffle & Wolf", image: "wafflewolf"),
28                 t(name: "Five Leaves", image: "fiveleaves"),
29                 t(name: "Cafe Lore", image: "cafelore"),
30                 t(name: "Confessional", image: "confessional"),
31                 t(name: "Barrafina", image: "barrafina"),
32                 t(name: "Donostia", image: "donostia"),
33                 t(name: "Royal Oak", image: "royaloak"),
34                 t(name: "CASK Pub and Kitchen", image: "caskpubkitchen")
35 ]
36 va
37 List(restaurant) {
38     restaurant in
39         HStack {
40             Image(restaurant.image)
41                 .resizable()
42                 .frame(width: 40, height: 40)
43                 .cornerRadius(5)
44             Text(restaurant.name)
45         }
46     }
47 }
48 }
```

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 rendered 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 overlaid on top of it. 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.

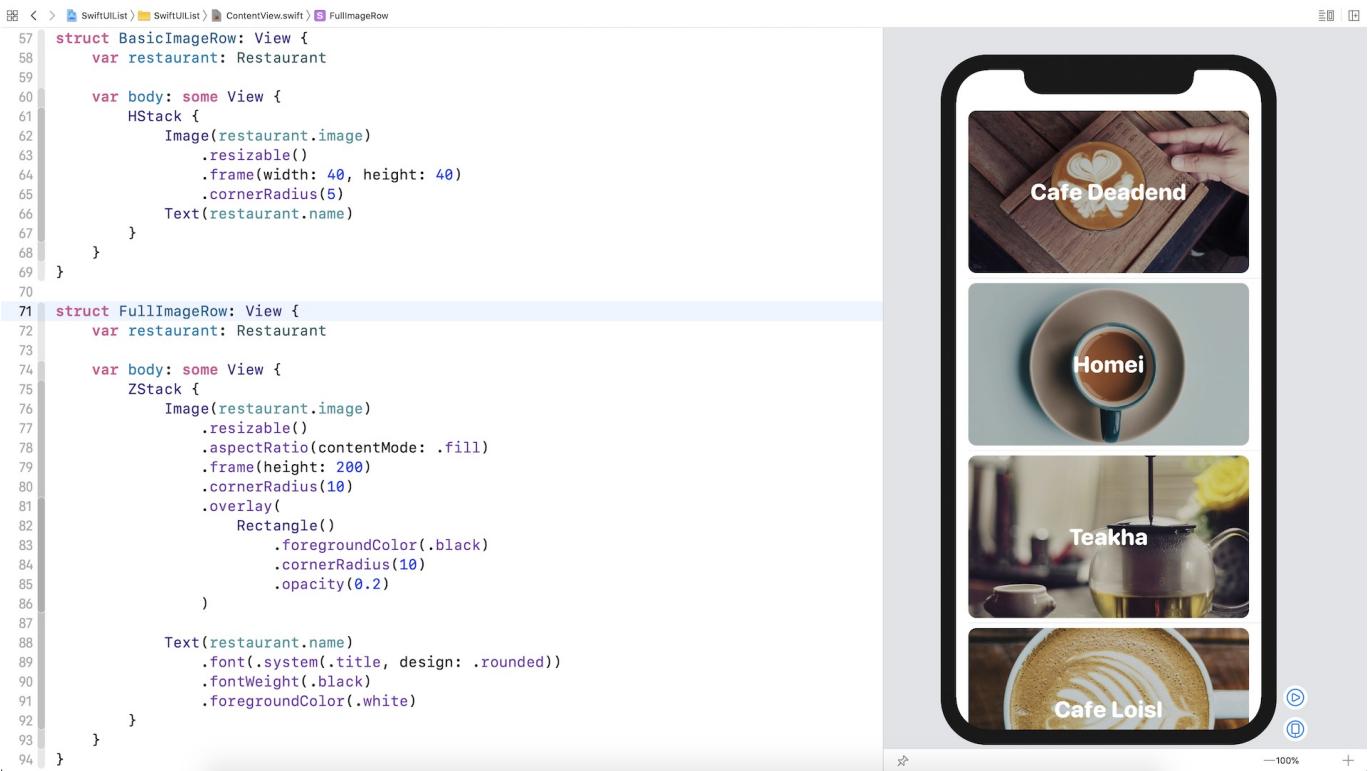


Figure 11. Changing the row layout

You can further mix the row layout to build a more interesting UI. For example, the new design is to use `FullImageRow` for the first two rows of data. The rest of the rows will utilize the `BasicImageRow`. You can update the `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 range of the restaurant data. In the closure, we check the value of `index` to determine which row layout to use.

```

27
28
29
30
31
32
33
34
35
36 var body: some View {
37     List(restaurants.indices) { index in
38
39         if (0...1).contains(index) {
40             FullImageRow(restaurant: self.restaurants[index])
41         } else {
42             BasicImageRow(restaurant: self.restaurants[index])
43         }
44
45     }
46 }
47
48 struct ContentView_Previews: PreviewProvider {
49     static var previews: some View {
50         ContentView()
51     }
52 }
53
54 struct Restaurant: Identifiable {
55     var id = UUID()
56     var name: String
57     var image: String
58 }
59
60 struct BasicImageRow: View {
61     var restaurant: Restaurant
62
63     var body: some View {

```

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 what I taught in this chapter, you should be able to build the UI. Spare some time to work on this exercise. I guarantee you'll learn a lot.

To save you time from finding your own images, you can download the image pack for this exercise from

<https://www.appcoda.com/resources/swiftui/SwiftUIArticleImages.zip>.

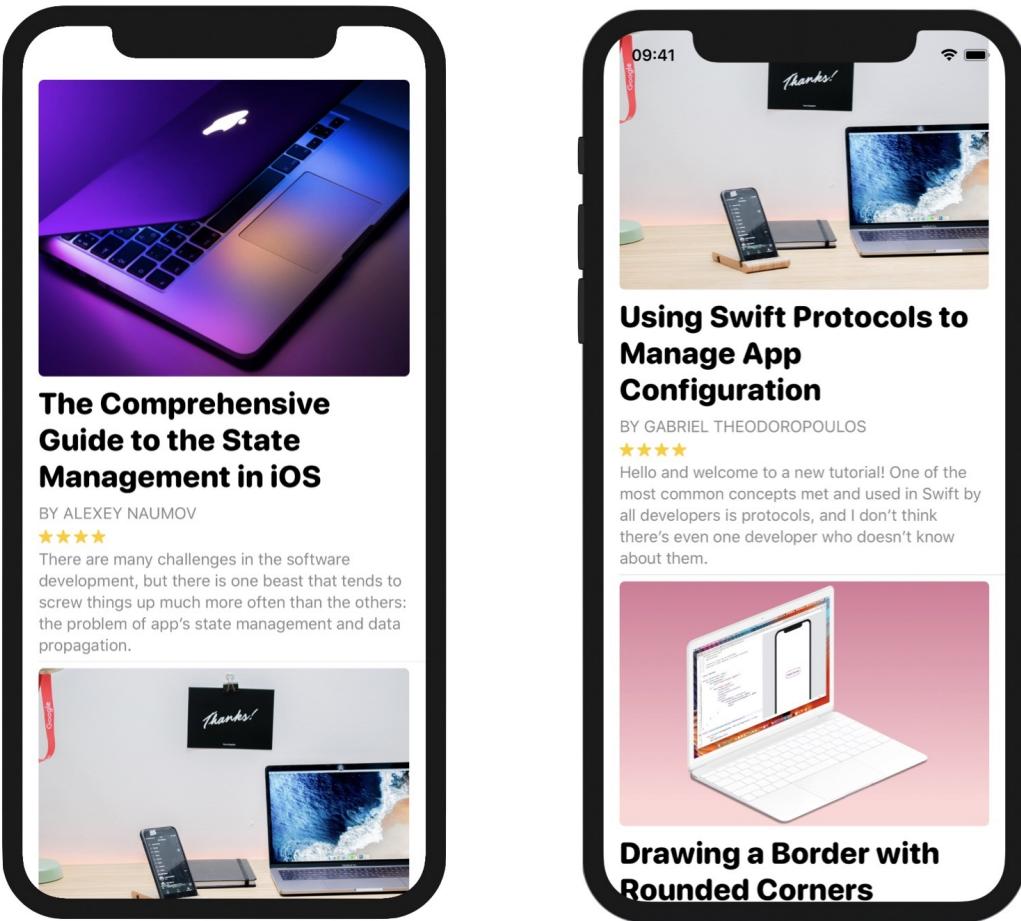


Figure 13. Building a list view with complex row layout

For reference, you can download the complete project here:

- Demo project (<https://www.appcoda.com/resources/swiftui/SwiftUIList.zip>)
- Solution to exercise
(<https://www.appcoda.com/resources/swiftui/SwiftUIListExercise.zip>)

Chapter 11

Working with Navigation UI and Navigation Bar Customization

In most apps, especially those content based app, you should have experienced a navigational interface. This kind of UI usually has a navigation bar with list of data and it allows users navigate to a detail view when tapping the content.

In UIKit, we can implement this type of interface using `UINavigationController`. For SwiftUI, Apple calls it `NavigationView`. In this chapter, I will walk you through the implementation of navigation UI 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`.

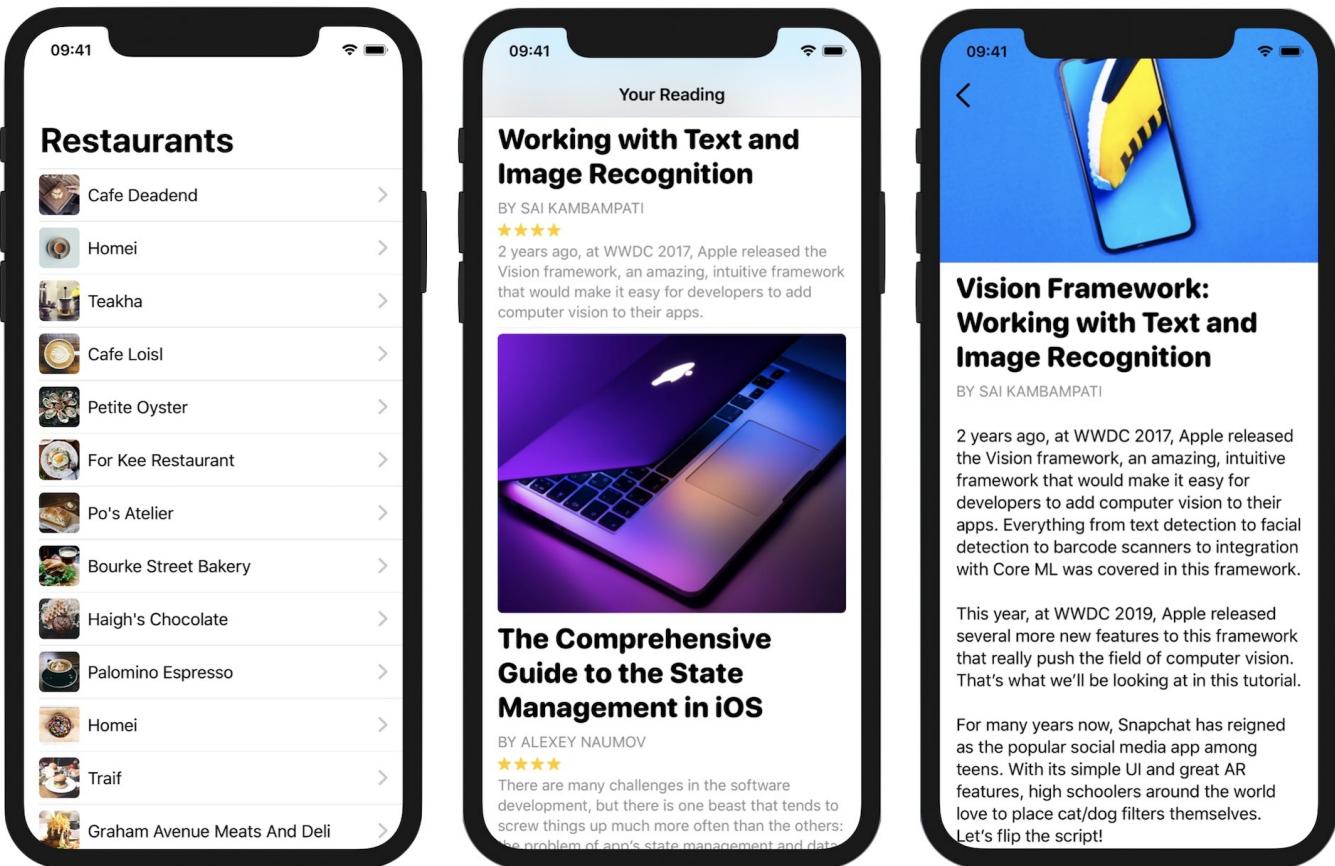


Figure 1. Sample navigation interface for our demo projects

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/swiftui/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.

The screenshot shows the Xcode interface with the ContentView.swift file open. The code defines a `List` view containing `BasicImageRow` items for various restaurants. A `ContentView_Previews` struct provides a preview of the list view, showing 15 restaurant entries with their names and small icons.

```

13 var restaurants = [ Restaurant(name: "Cafe Deadend", image: "cafedeadend"),
14     Restaurant(name: "Homei", image: "homei"),
15     Restaurant(name: "Teakha", image: "teakha"),
16     Restaurant(name: "Cafe Loisl", image: "cafeloisl"),
17     Restaurant(name: "Petite Oyster", image: "petiteoyster"),
18     Restaurant(name: "For Kee Restaurant", image: "forkeerrestaurant"),
19     Restaurant(name: "Po's Atelier", image: "posatelier"),
20     Restaurant(name: "Bourke Street Bakery", image: "bourkestreetbakery"),
21     Restaurant(name: "Haigh's Chocolate", image: "haighschocolate"),
22     Restaurant(name: "Palomino Espresso", image: "palominoespresso"),
23     Restaurant(name: "Homei", image: "upstate"),
24     Restaurant(name: "Traif", image: "traif"),
25     Restaurant(name: "Graham Avenue Meats And Deli", image:
26         "grahamavenuemeats"),
27     Restaurant(name: "Waffle & Wolf", image: "wafflewolf"),
28     Restaurant(name: "Five Leaves", image: "fiveleaves"),
29     Restaurant(name: "Cafe Lore", image: "cafelore"),
30     Restaurant(name: "Confessional", image: "confessional"),
31     Restaurant(name: "Barrafina", image: "barrafina"),
32     Restaurant(name: "Donostia", image: "donostia"),
33     Restaurant(name: "Royal Oak", image: "royaloak"),
34     Restaurant(name: "CASK Pub and Kitchen", image: "caskpubkitchen")
35 ]
36
37 var body: some View {
38     List {
39         ForEach(restaurants) { restaurant in
40             BasicImageRow(restaurant: restaurant)
41         }
42     }
43 }
44
45 struct ContentView_Previews: PreviewProvider {
46     static var previews: some View {
47         ContentView()
48     }
49 }

```

Figure 2. The starter project should display a simple list view

What we're going to do is embedding this list view in a navigation view.

Implementing a Navigation View

As mentioned before, 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:

```

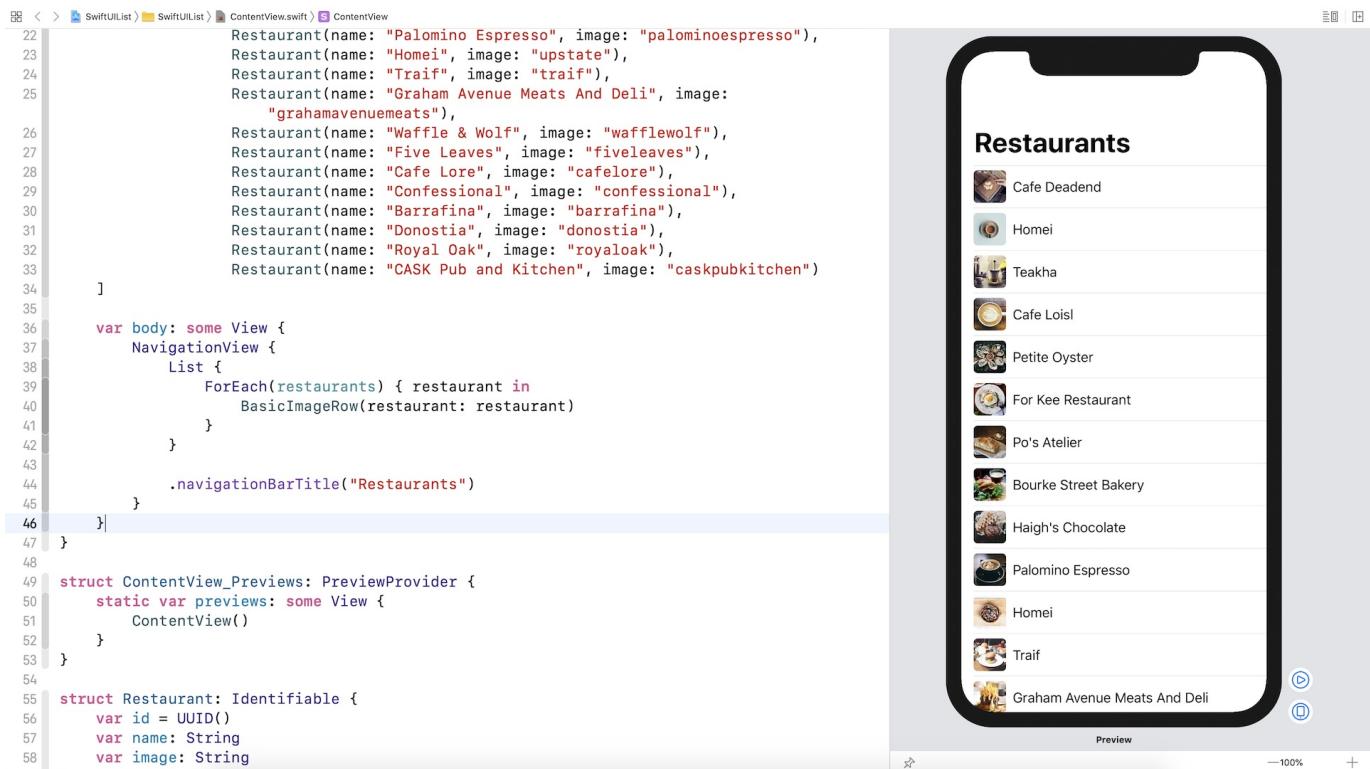
NavigationView {
    List {
        ForEach(restaurants) { restaurant in
            BasicImageRow(restaurant: restaurant)
        }
    }
}

```

Once you 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 should have a navigation bar with the large title.



The screenshot shows the Xcode interface with the ContentView.swift file open. The code defines a `ContentView` struct containing a `body` property which is a `NavigationView` with a `List` of `BasicImageRow`s for each restaurant. The `navigationBarTitle` modifier is applied to the `NavigationView` with the value "Restaurants". Below the code, there's a `struct ContentView_Previews` for a `PreviewProvider`. At the bottom, there's a `struct Restaurant` definition. To the right of the code editor is a simulator window showing a smartphone screen with the title "Restaurants" at the top. The main content area displays a list of ten restaurants, each with a small image icon and the restaurant's name: Cafe Deadend, Homei, Teakha, Cafe Loisl, Petite Oyster, For Kee Restaurant, Po's Atelier, Bourke Street Bakery, Haigh's Chocolate, Palomino Espresso, Homei, Traif, and Graham Avenue Meats And Deli.

```
22     Restaurant(name: "Palomino Espresso", image: "palominoespresso"),  
23     Restaurant(name: "Homei", image: "upstate"),  
24     Restaurant(name: "Traif", image: "traif"),  
25     Restaurant(name: "Graham Avenue Meats And Deli", image:  
26         "grahamavenuemeats"),  
27     Restaurant(name: "Waffle & Wolf", image: "wafflewolf"),  
28     Restaurant(name: "Five Leaves", image: "fiveleaves"),  
29     Restaurant(name: "Cafe Lore", image: "cafelore"),  
30     Restaurant(name: "Confessional", image: "confessional"),  
31     Restaurant(name: "Barrafina", image: "barrafina"),  
32     Restaurant(name: "Donostia", image: "donostia"),  
33     Restaurant(name: "Royal Oak", image: "royaloak"),  
34     Restaurant(name: "CASK Pub and Kitchen", image: "caskpubkitchen")  
35 ]  
36  
37 var body: some View {  
38     NavigationView {  
39         List {  
40             ForEach(restaurants) { restaurant in  
41                 BasicImageRow(restaurant: restaurant)  
42             }  
43         }  
44         .navigationBarTitle("Restaurants")  
45     }  
46 }  
47 }  
48  
49 struct ContentView_Previews: PreviewProvider {  
50     static var previews: some View {  
51         ContentView()  
52     }  
53 }  
54  
55 struct Restaurant: Identifiable {  
56     var id = UUID()  
57     var name: String  
58     var image: String  
--
```

Figure 3. A basic navigation UI

Passing Data to a Detail View Using `NavigationLink`

So far, we just added a navigation bar to the list view. We usually use a navigation interface for 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.

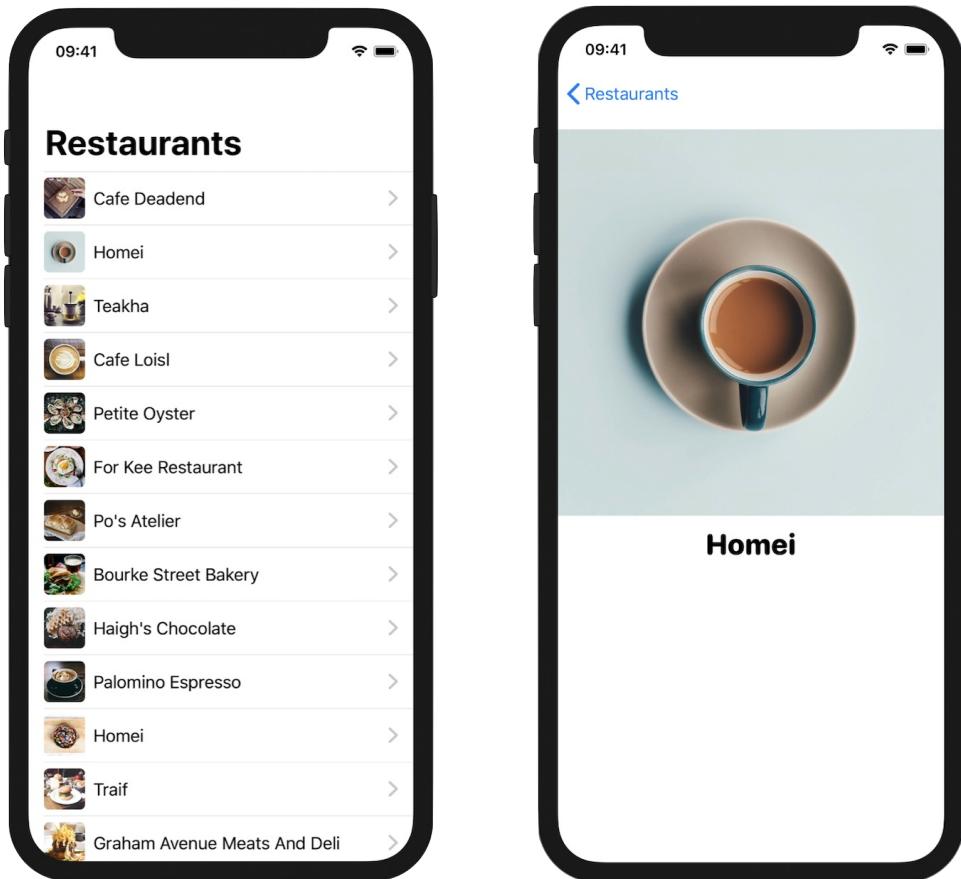


Figure 4. The content view and detail view

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: .fill)
                .clipped()

            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 that 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.

Okay, the detail view is 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 `NavLink`, which is able to detect users' touches and triggers the navigation presentation. The basic usage of `NavLink` is like this:

```
NavLink(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 detail view when any of the restaurants is tapped. In this case, we can apply `NavLink` to each of the rows.

Update the `List` view like this:

```
List {  
    ForEach(restaurants) { restaurant in  
        NavigationLink(destination: RestaurantDetailView(restaurant: restaurant))  
    }  
}  
}
```

In the code above, we tell `NavLink` 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.

```
18 Restaurant(name: "For Kee Restaurant", image: "forkeerestaurant"),
19 Restaurant(name: "Po's Atelier", image: "posatelier"),
20 Restaurant(name: "Bourke Street Bakery", image: "bourkestreetbakery"),
21 Restaurant(name: "Haigh's Chocolate", image: "haighschocolate"),
22 Restaurant(name: "Palomino Espresso", image: "palominoespresso"),
23 Restaurant(name: "HomeI", image: "upstate"),
24 Restaurant(name: "Traif", image: "traif"),
25 Restaurant(name: "Graham Avenue Meats And Deli", image:
26     "grahamavenuemeats"),
27 Restaurant(name: "Waffle & Wolf", image: "wafflewolf"),
28 Restaurant(name: "Five Leaves", image: "fiveleaves"),
29 Restaurant(name: "Cafe Lore", image: "cafelore"),
30 Restaurant(name: "Confessional", image: "confessional"),
31 Restaurant(name: "Barrafina", image: "barrafina"),
32 Restaurant(name: "Donostia", image: "donostia"),
33 Restaurant(name: "Royal Oak", image: "royaloak"),
34 Restaurant(name: "CASK Pub and Kitchen", image: "caskpubkitchen")
35 ]
36
37 var body: some View {
38     NavigationView {
39         List {
40             ForEach(restaurants) { restaurant in
41                 NavigationLink(destination: RestaurantDetailView(restaurant:
42                     restaurant)) {
43                     BasicImageRow(restaurant: restaurant)
44                 }
45             }
46         }
47         .navigationBarTitle("Restaurants")
48     }
49 }
50
51 struct ContentView_Previews: PreviewProvider {
52     static var previews: some View {
53         ContentView()
```

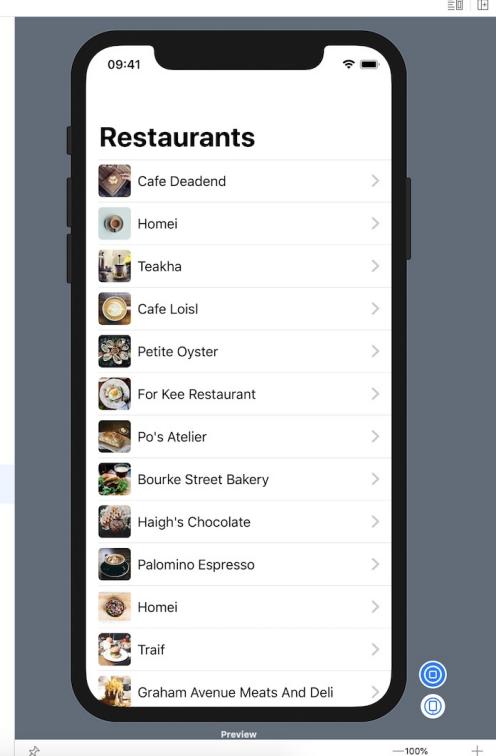


Figure 5. Run the app to test the navigation

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

iOS 13 introduces a new API called `UINavigationBarAppearance` for navigation bar customizations. Its usage is very similar to the old API but offers you more granularity. In iOS 13, you're allowed to configure the following appearance of a navigation bar:

1. **Standard Appearance** (`.standardAppearance`) - the appearance of a standard-height navigation bar (e.g. the navigation bar appeared in iPhone portrait)
2. **Compact Appearance** (`.compactAppearance`) - the appearance of a compact-height navigation bar (e.g. the navigation bar appeared in iPhone landscape)
3. **Scroll Edge Appearance** (`.scrollEdgeAppearance`) - this is the appearance when the edge of the scrolled content reaches the navigation bar

It's not a mandate to configure different appearance settings for these three appearances of the navigation bar. You can apply the same settings for all of them.

So, what customizations can you apply to the navigation bar? Actually, quite a lot. Basically, you can change the navigation bar's font, color, background, etc. I will discuss with you 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 is default behaviour since Apple introduces the "Large Title" navigation bar.

In case you want to keep the navigation bar compact and disable the use of 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 one. 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.

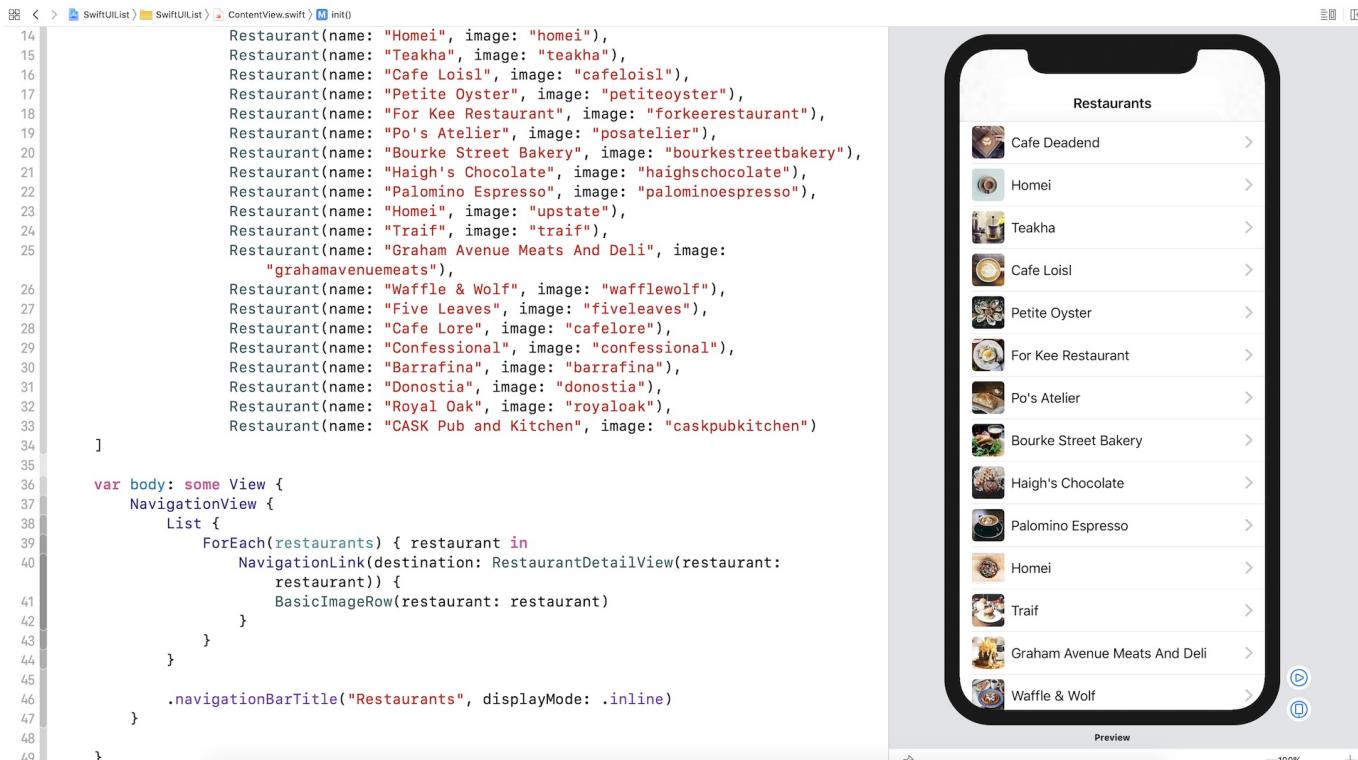


Figure 6. Setting the display mode to `.inline` to use the compact bar

Now let's change the display mode to `.automatic` and see what you get. The navigation bar should become a large title bar again.

```
.navigationBarTitle("Restaurants", displayMode: .automatic)
```

Configuring Font and Color

Next, let's see how to 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, for example, we want to change the bar title's color to red and the font to *Arial Rounded MT Bold*. We can 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.systemRed, .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 designed for 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`. As said before, if you need, you can create and assign a separate appearance object for `scrollEdgeAppearance`, and `compactAppearance`.

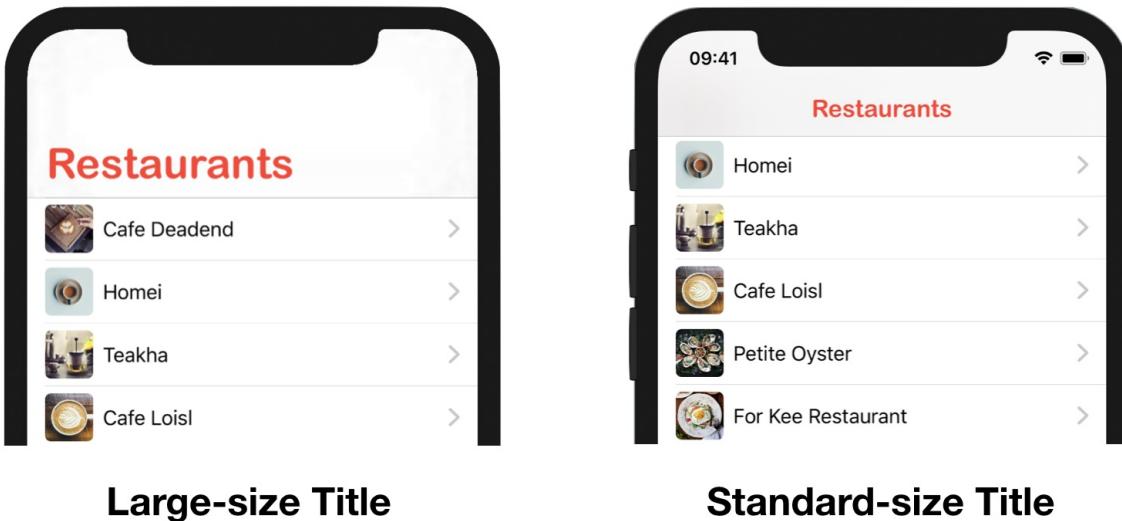


Figure 7. Changing the font type and color for both large-size and standard-size titles

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.

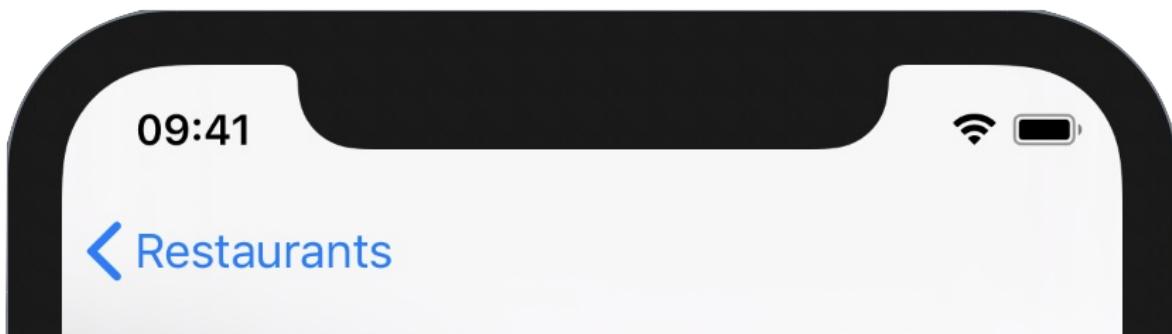


Figure 8. A standard back button

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
```

If you've made the changes, run the app to have a quick test. The back button should be like that shown in figure 9.



The image shows a split-screen view. On the left is a code editor window for `ContentView.swift` in Xcode. The code defines a custom navigation bar appearance and sets its `tintColor` to black. On the right is a screenshot of an iPhone displaying a navigation bar with a black back button labeled "Restaurants".

```
51 init() {
52     let navBarAppearance = UINavigationBarAppearance()
53     navBarAppearance.largeTitleTextAttributes = [.foregroundColor: UIColor.systemRed,
54         .font: UIFont(name: "ArialRoundedMTBold", size: 35)!]
55     navBarAppearance.titleTextAttributes = [.foregroundColor: UIColor.systemRed,
56         .font: UIFont(name: "ArialRoundedMTBold", size: 20)!]
57     navBarAppearance.setBackIndicatorImage(UIImage(systemName: "arrow.turn.up.left"),
58         transitionMaskImage: UIImage(systemName: "arrow.turn.up.left"))
59
60     UINavigationBar.appearance().standardAppearance = navBarAppearance
61     UINavigationBar.appearance().scrollEdgeAppearance = navBarAppearance
62     UINavigationBar.appearance().compactAppearance = navBarAppearance
63
64     UINavigationBar.appearance().tintColor = .black
65 }
```

Figure 9. Customizing the appearance of the back button

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/swiftui/SwiftUINavigationStarter.zip>. Open the project and you will see a demo app showing a list of article.

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 has come with some sample data. If you look into the `Article` struct closely, it now has the `content` property for storing the 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 discuss the solution with you in the next section, but please try your best to figure out your own solution.

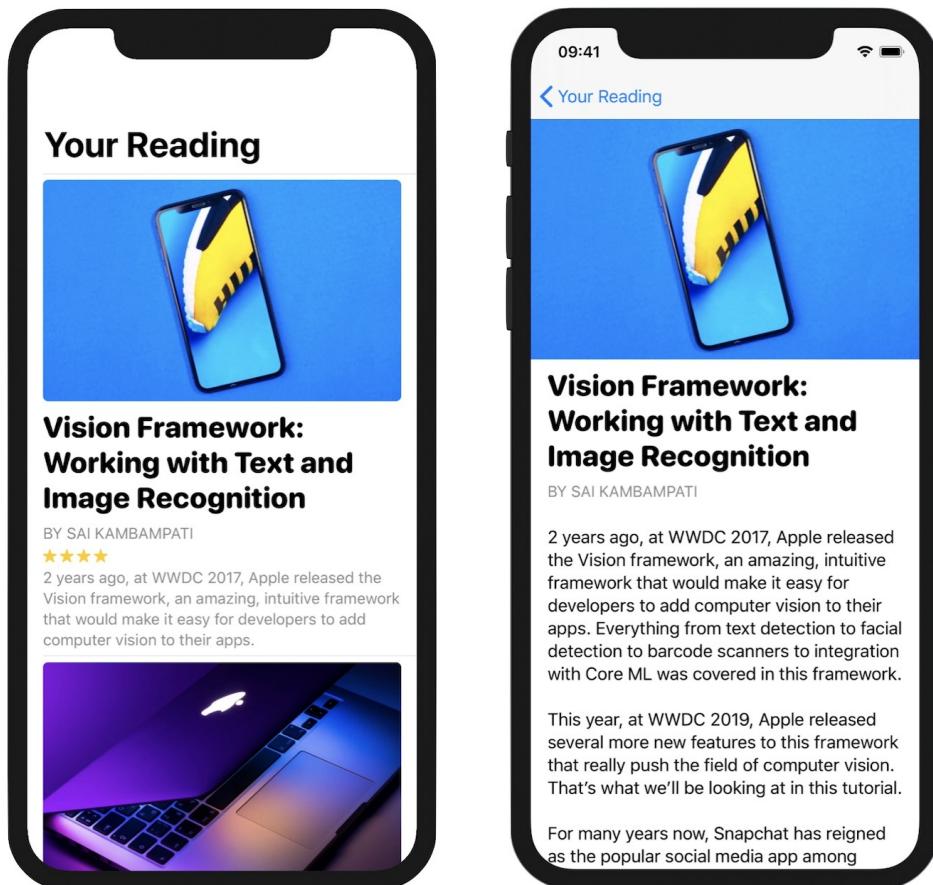


Figure 10. Building a navigation UI for a Reading app

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 `SwiftUITable` folder and select *New File...*. Choose the *SwiftUI View* template and name

the file `ArticleDetailView.swift`.

Since this detail view is going to display the article detail, 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 \u{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 the scrollable content. I'll not go over the code line by line as I believe you should 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 certain configuration. In the code above, we need to apply a certain padding configuration to both `Text` views. To avoid code duplication, we group both views together and apply the paddings.

Now that we have completed the layout of the detail view, however, you should 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 other values if you want to preview other articles. Once you made this change, the preview canvas should render the detail view properly.

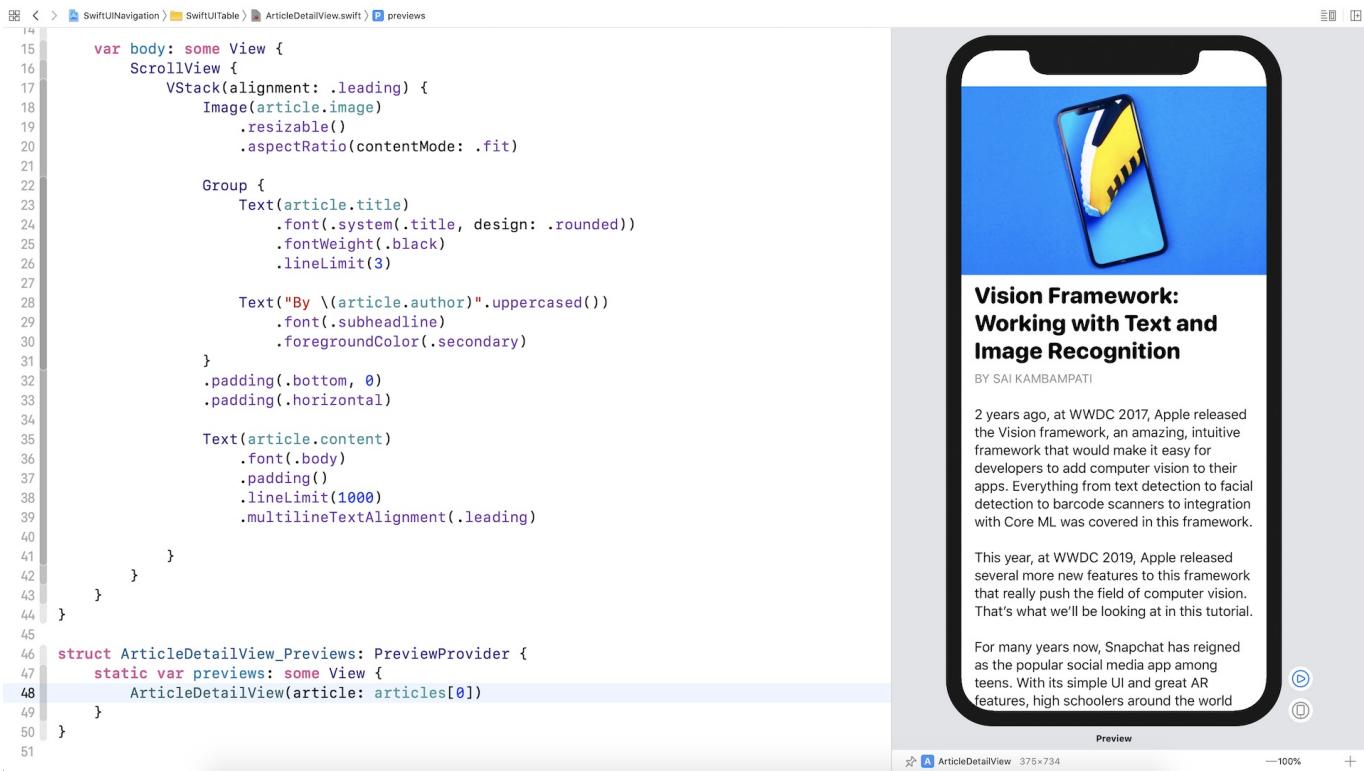


Figure 11. The detail view for showing the article

Let's try one more thing. Since this view is going to embed in a `NavigationView`, you can modify the preview code to preview how it looks in a navigation interface:

```

struct ArticleDetailView_Previews: PreviewProvider {
    static var previews: some View {
        NavigationView {
            ArticleDetailView(article: articles[0])
        }
    }
}

```

By updating the code, you should 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. Can we disable it? The second issue is the empty space appeared right above the featured image in the detail view. Let's discuss the issues one by one.

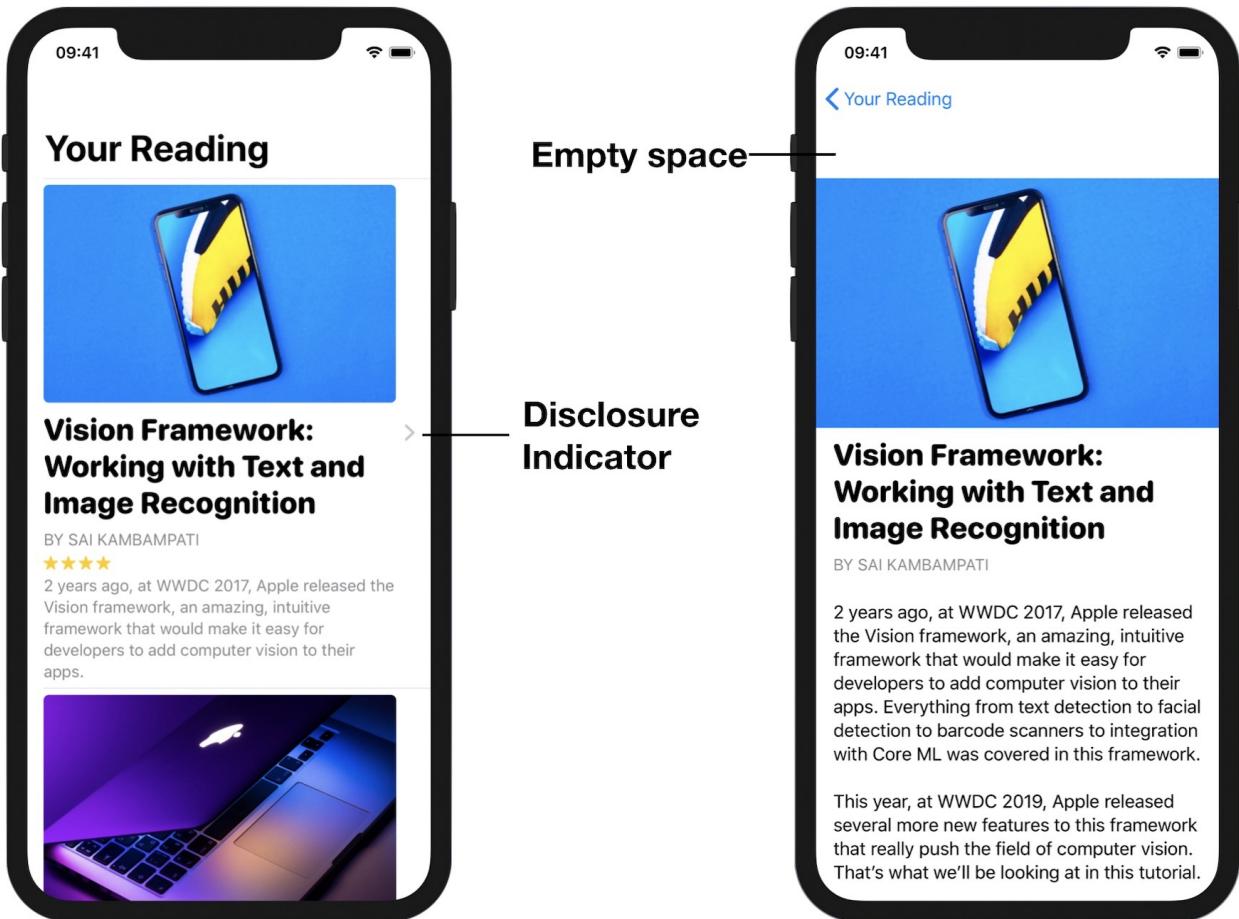


Figure 12. Two issues in the current design

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. Now 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 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).

```

15 var body: some View {
16     ScrollView {
17         VStack(alignment: .leading) {
18             Image(article.image)
19                 .resizable()
20                 .aspectRatio(contentMode: .fit)
21
22             Group {
23                 Text(article.title)
24                     .font(.system(.title, design: .rounded))
25                     .fontWeight(.black)
26                     .lineLimit(3)
27
28                 Text("By \(article.author)".uppercased())
29                     .font(.subheadline)
30                     .foregroundColor(.secondary)
31             }
32             .padding(.bottom, 0)
33             .padding(.horizontal)
34
35             Text(article.content)
36                 .font(.body)
37                 .padding()
38                 .lineLimit(1000)
39                 .multilineTextAlignment(.leading)
40
41         }
42     }
43 }
44
45 struct ArticleDetailView_Previews: PreviewProvider {
46     static var previews: some View {
47         NavigationView {
48             ArticleDetailView(article: articles[0])
49         }
50     }
51 }
52

```

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 navigate 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?

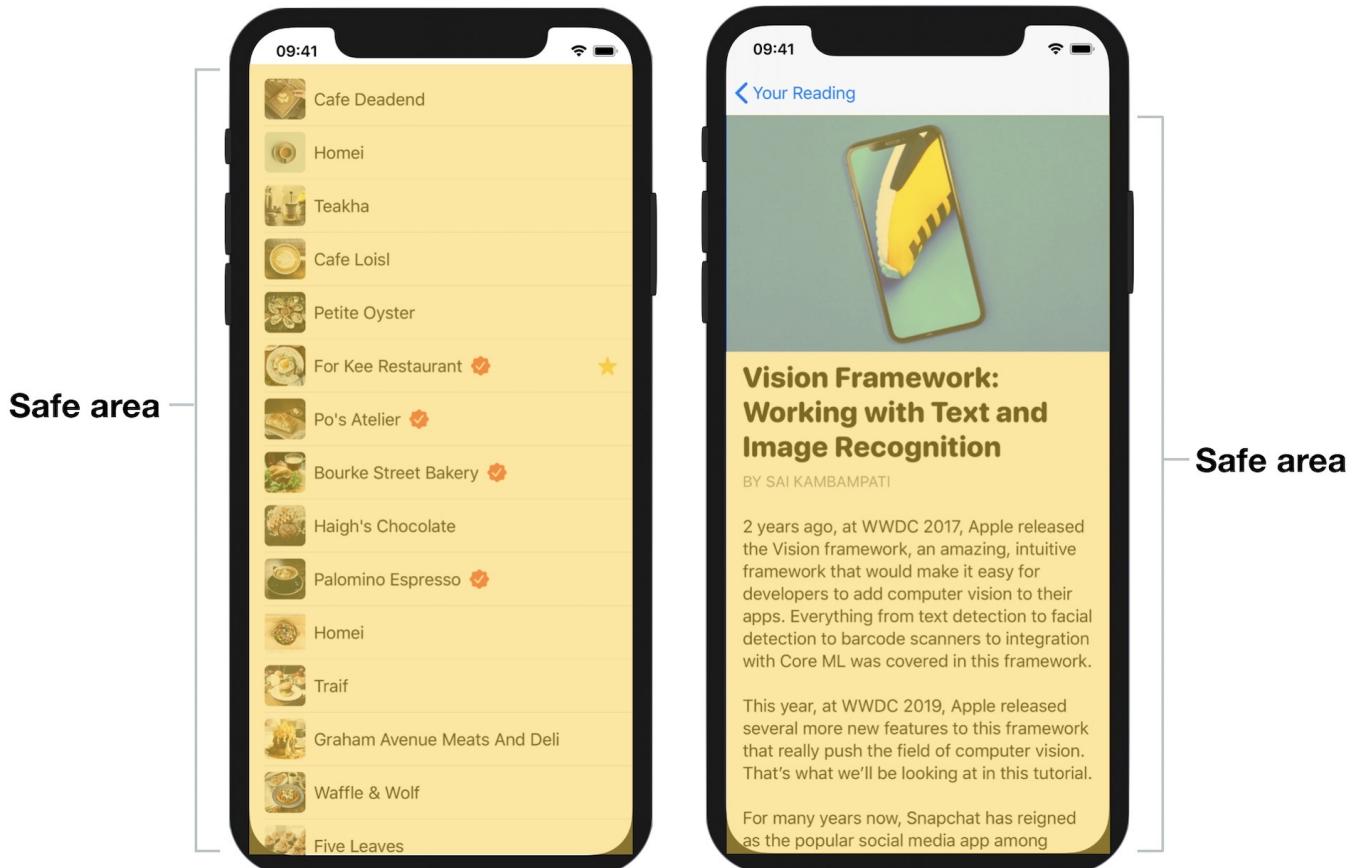


Figure 14. The revised design of the detail view

To lay out this screen, we have to tackle two issues:

1. How to extend the scroll view to the very top of the screen
2. How to create a custom back button and trigger the navigation programmatically

iOS has a concept known as *safe areas* for aiding the layout of the views. Safe areas help you place the views within the visible portion of the interface. Say, for example, safe areas prevent the views from hiding the status bar. And, if your UI introduces a navigation bar, the safe area will automatically be adjusted to prevent you from positioning the views that occludes the navigation bar.



![Figure 15. Safe areas](images/navigation/swiftui-navigation-15.jpg)

To place content that extends outside the safe areas, you can use a modifier named `edgesIgnoringSafeArea`. For our project, as we want the scroll view to go beyond the top edge of the safe area, we can write the modifier like this:

```
.edgesIgnoringSafeArea(.top)
```

This modifier 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 these two modifiers to the `ScrollView`, we can hide the navigation bar and achieve a visually pleasing detail view.

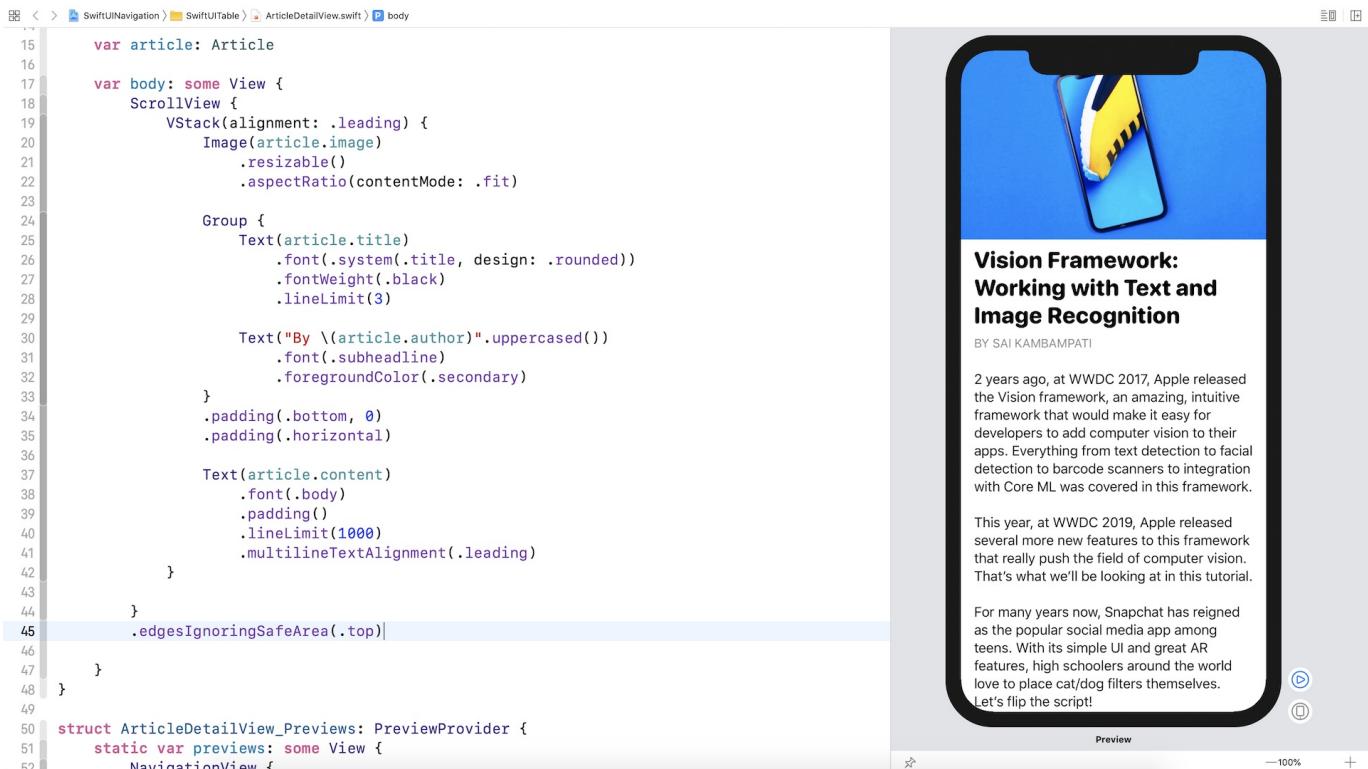


Figure 16. Applying the modifiers to the scroll view

Now it comes to the second issue about 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 `false` 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.

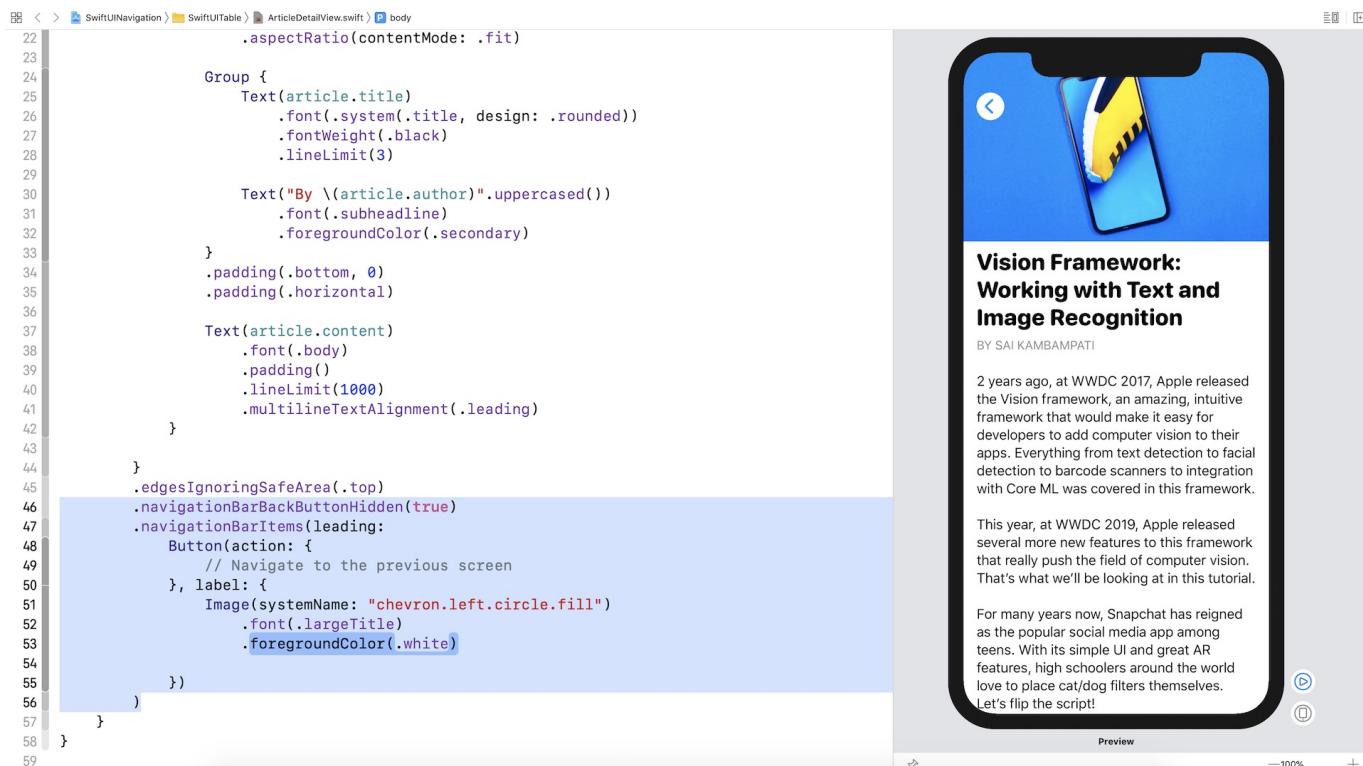


Figure 17. Creating our own back button

You may aware 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. Here is the problem. How can we programmatically trigger this operation? Thanks to the environment values built into the SwiftUI framework. You can refer to an environment binding named `presentationMode` to reveal 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. Now you can 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 what I covered in this chapter. If you fully understand the materials, you are already capable to build a simple content-based app, though the data is static.

For reference, you can download the complete project here:

- Demo project and solution to exercise
(<https://www.appcoda.com/resources/swiftui/SwiftUINavigation.zip>)

To further study navigation view, you can also refer to these 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. Say, 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.

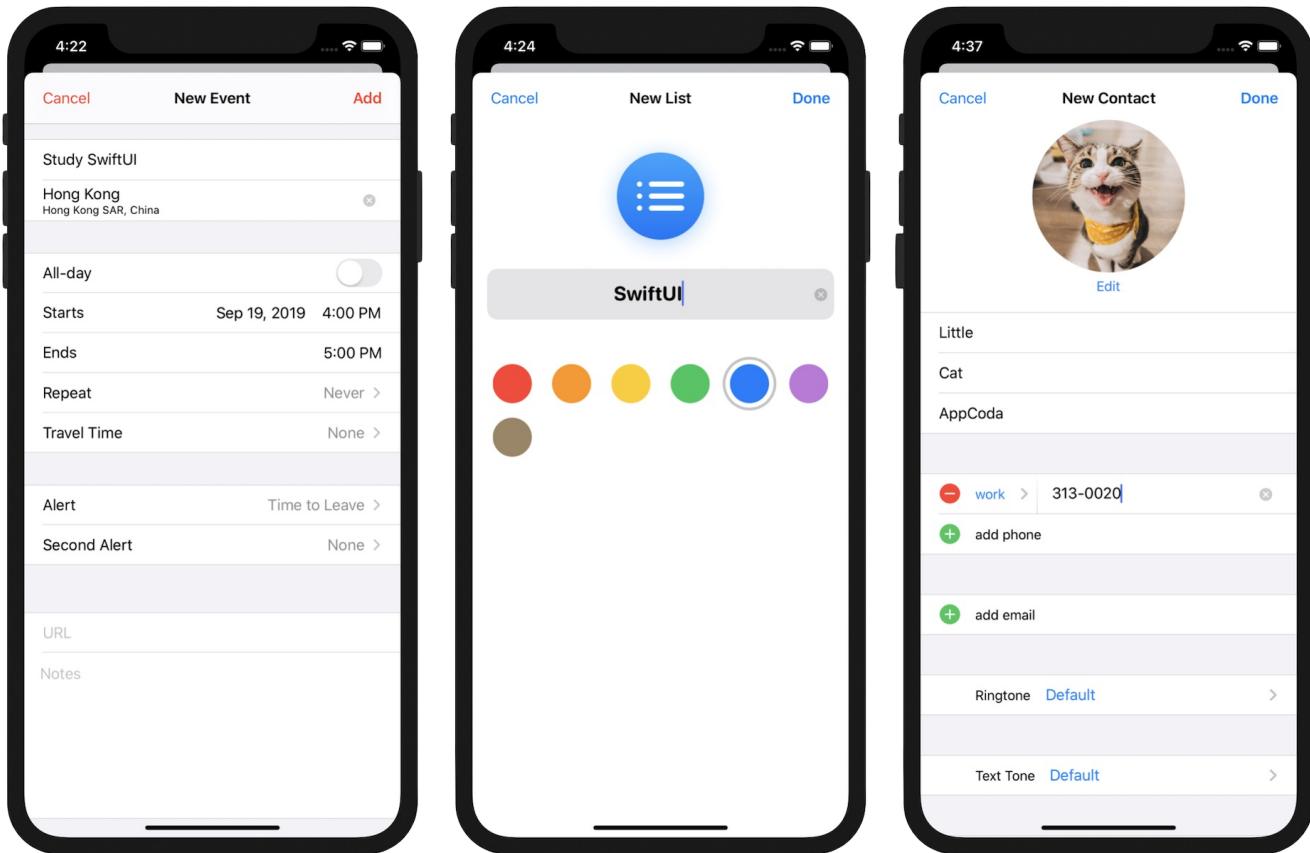


Figure 1. Sample modal views in Calendar, Reminders, and Contact apps

From the user experience point of view, this 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 users, you may find the look & feel of the modal views displayed in figure 1 is not exactly the same as the usual ones. Prior to iOS 13, the presentation of modal views covers the entire screen, while in 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. Other than 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 views.

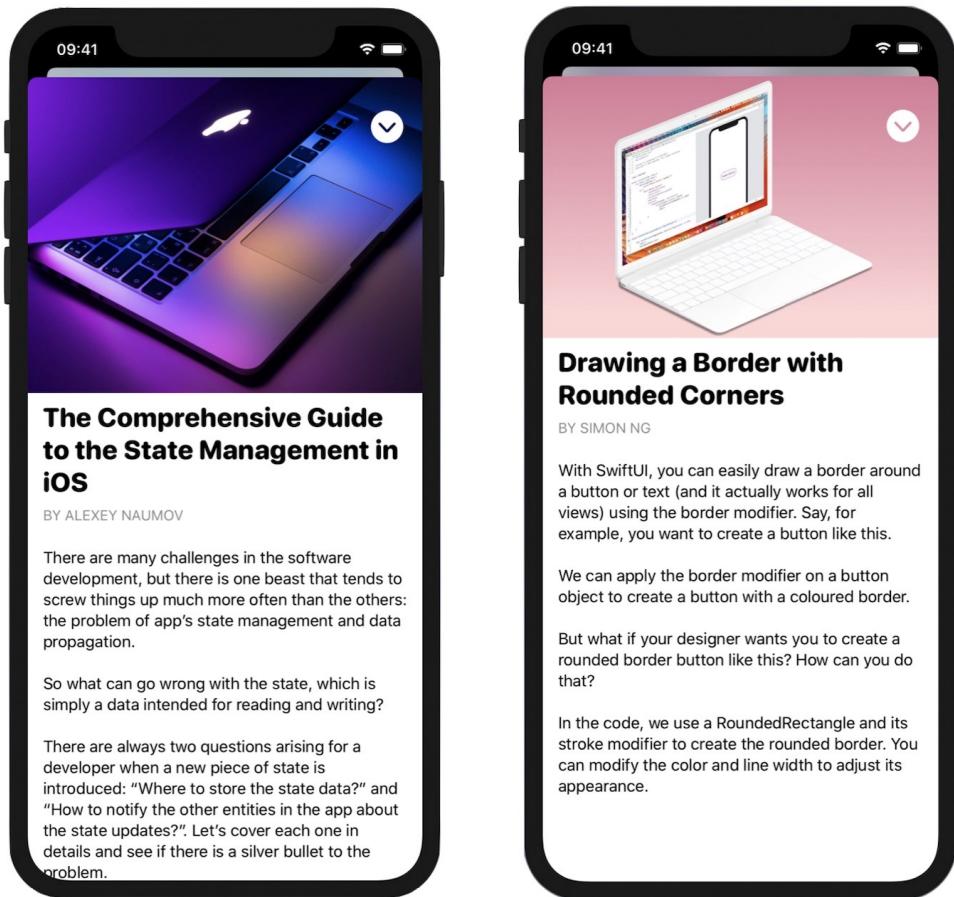


Figure 2. Presenting the detail screen using modal views

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 realized 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 can be 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 still remember our discussion on `actionSheet` in the earlier chapter, you should find that the usage of `sheet` is very similar to that of `actionSheet`.

Preparing the Starter Project

That's enough for the 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/swiftui/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.

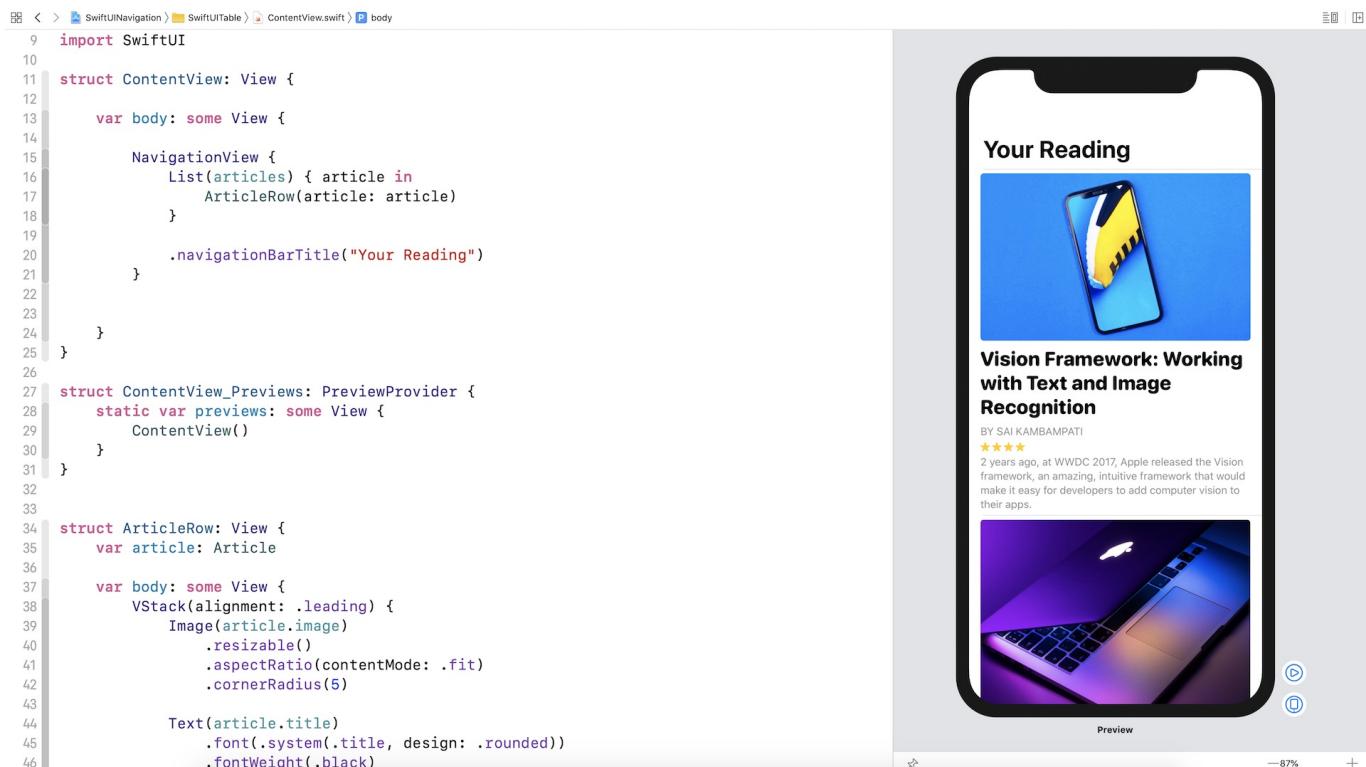


Figure 3. Starter project

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 self.selectedArticle != nil {
            ArticleDetailView(article: self.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 final question is how can we detect touch? When building the navigation UI, we utilize `NavigationLink` to handle the touch. However, this special button is designed for navigation interface. In SwiftUI, there is a handler called `onTapGesture` which can be used to recognize a tap gesture. So, you can attach this handler to each of the `ArticleRow` to detect users' touch. Now 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 self.selectedArticle != nil {
            ArticleDetailView(article: self.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.

Now run the app in the preview canvas. You should be able to bring up the detail view modally.

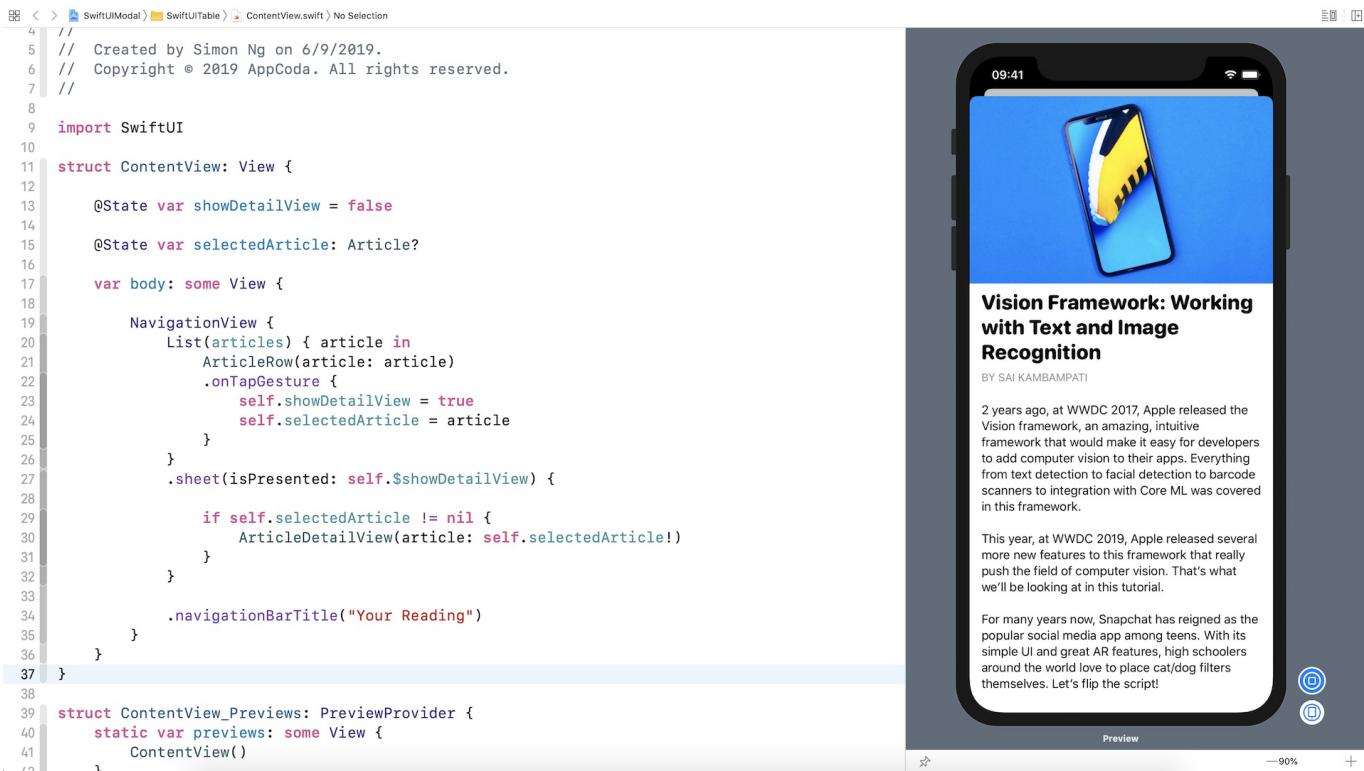


Figure 4. Presenting the detail view modally

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 us 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
```

After changing the code, you can test the app again. Everything should work like before but the underlying is cleaner than the original one.

Creating a Floating Button for Dismissing the Modal View

The modal view has a built-in support for the swipe-down gesture. Right now, you can swipe down the modal view to close it. I guess this works pretty natural 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 to dismissing the modal view.

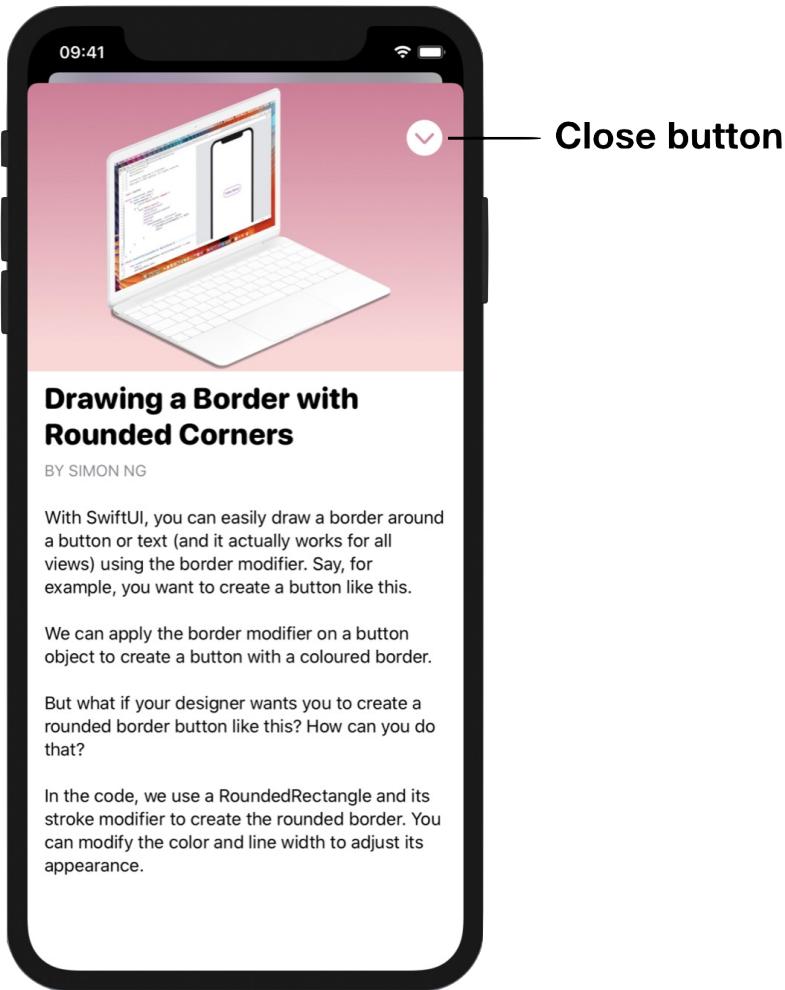


Figure 5. The close button for dismissing the modal view

Now switch over to `ArticleDetailView.swift`. We'll add the close button to the view as shown in figure 5.

Do you know how to position the button at the top-right corner? Try not to follow my code straight away but come up with your own implementation.

Okay, back to the implementation.

Similar to `NavigationView`, we can dismiss the modal 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()  
        }  
    }  
)
```

By doing so, the button will be overlayed on top of the scroll view that it appears as a floating button. Even 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 can call the `dismiss()` function of `presentationMode`.

The screenshot shows the Xcode interface with the code editor on the left and a preview window on the right. The code editor displays Swift code for ArticleDetailView.swift, specifically focusing on the implementation of a close button. The preview window shows a mobile application interface with a blue header, a yellow and blue image placeholder, and a card-like view containing text about the Vision Framework.

```

32         .foregroundColor(.secondary)
33     }
34     .padding(.bottom, 0)
35     .padding(.horizontal)
36
37     Text(article.content)
38     .font(.body)
39     .padding()
40     .lineLimit(1000)
41     .multilineTextAlignment(.leading)
42   }
43
44 }
45 .overlay(
46
47   HStack {
48     Spacer()
49
50     VStack {
51       Button(action: {
52         self.presentationMode.wrappedValue.dismiss()
53       }, label: {
54         Image(systemName: "chevron.down.circle.fill")
55         .font(.largeTitle)
56         .foregroundColor(.white)
57       })
58       .padding(.trailing, 20)
59       .padding(.top, 40)
60
61     }
62   }
63 )
64
65 .edgesIgnoringSafeArea(.top)
66
67 }
68 }
69 }

```

Figure 6. Implementing the close button

Now 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

Other than the card-like modal views, *Alerts* are another kind of modal views. When it's presented, the entire screen is blocked. You can't move away without choosing one of the options. Figure 7 shows a sample alert and this is something we're going to implement in our demo project. What we're going to do is display an alert after a user taps the close button.

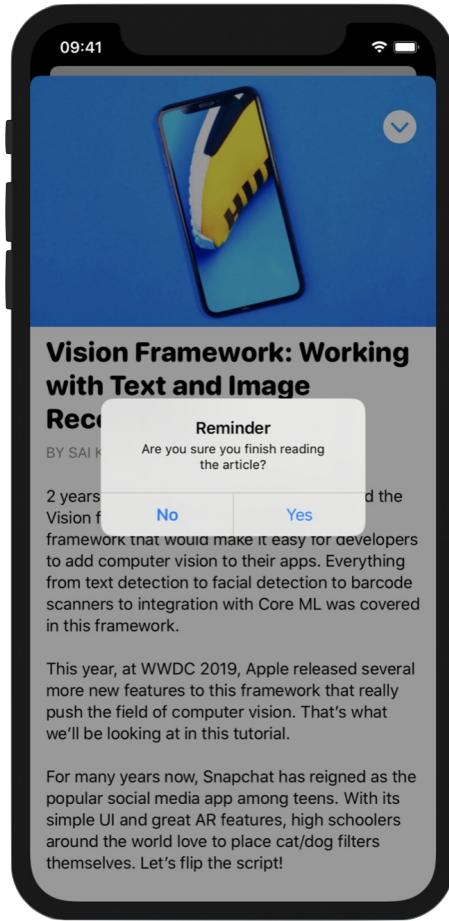


Figure 7. Displaying an alert

In SwiftUI, you can create an alert using the `Alert` struct. Here is a sample usage of `Alert`:

```
Alert(title: Text("Warning"), message: Text("Are you sure you want to leave?"), primaryButton: .default(Text("Confirm")), secondaryButton: .cancel())
```

The sample code initiates an alert view with title "Warning". The alert prompt also displays a message "Are you sure you want to leave" to the user. There are two buttons in the alert view: *Confirm* and *Cancel*.

To create the alert as shown in figure 7, the code will be like this:

```
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")))
```

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, it will continue to close the modal view. 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` like this:

```
.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")))
}
```

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."

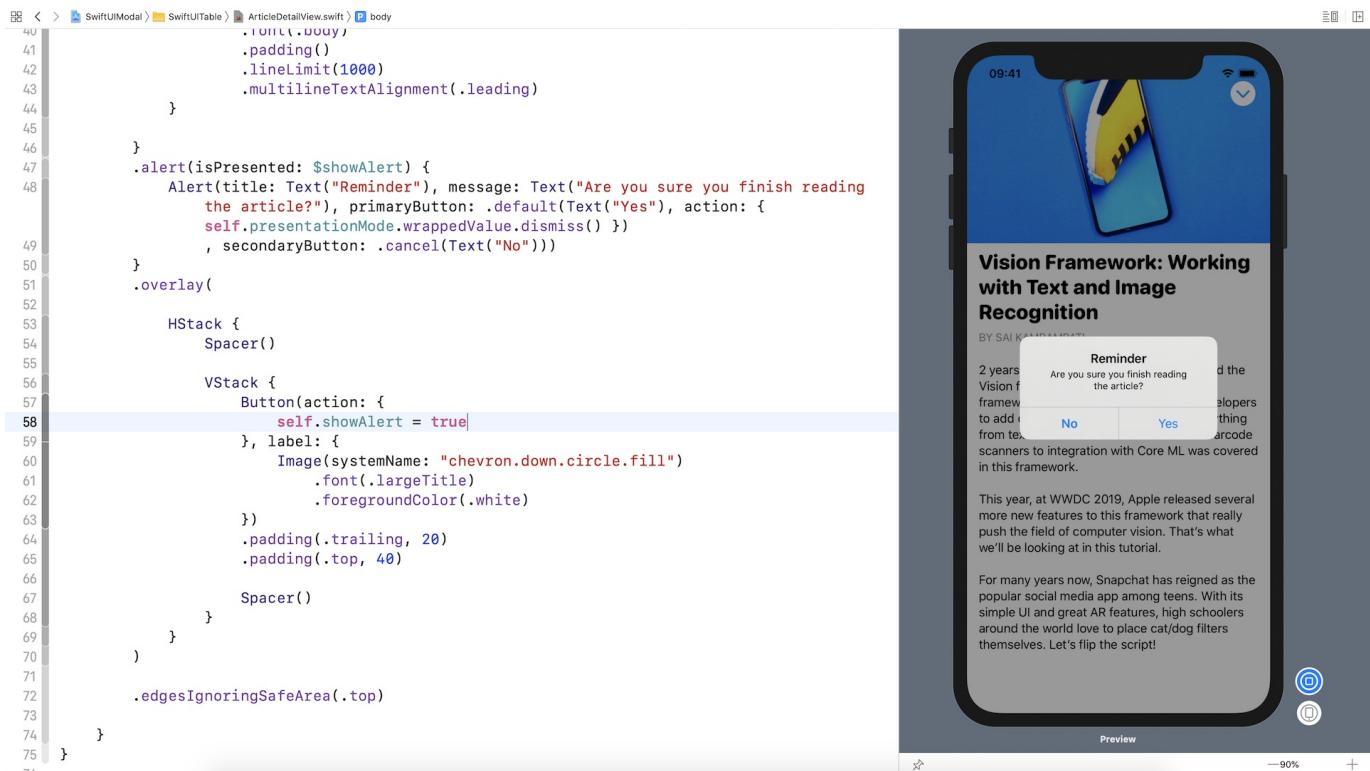


Figure 8. Tapping the close button will show you the alert

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 project here:

- Demo project (<https://www.appcoda.com/resources/swiftui/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 would come across a mobile form. For example, a calendar app may present you a form to fill in the information of a new event. Or 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, it comes with 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.

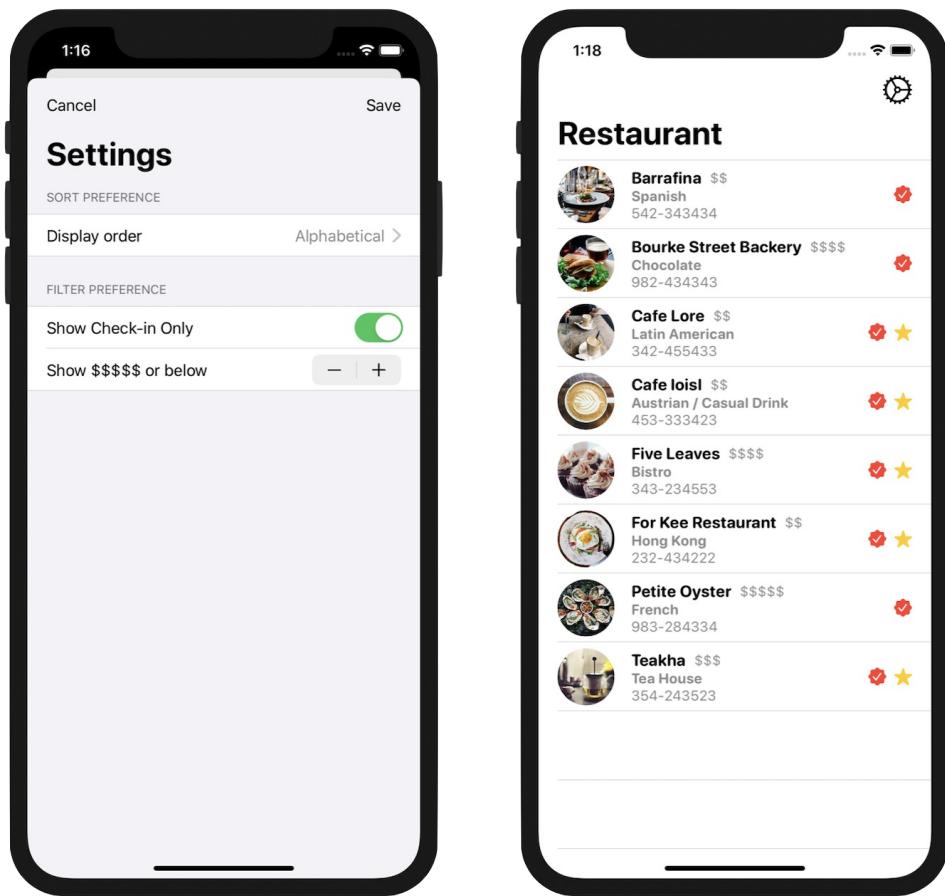


Figure 1. Building a Setting screen

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 screens 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 the sort preference, plus create a toggle and a stepper for indicating the 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 to 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 already created a starter project for you. First, download it from

<https://www.appcoda.com/resources/swiftui/SwiftUIFormStarter.zip>. Once downloaded, open the `SwiftUIList.xcodeproj` file with Xcode. Preview `ContentView.swift` in the canvas and you'll see a familiar UI except that it incorporates more detailed information of a restaurant.

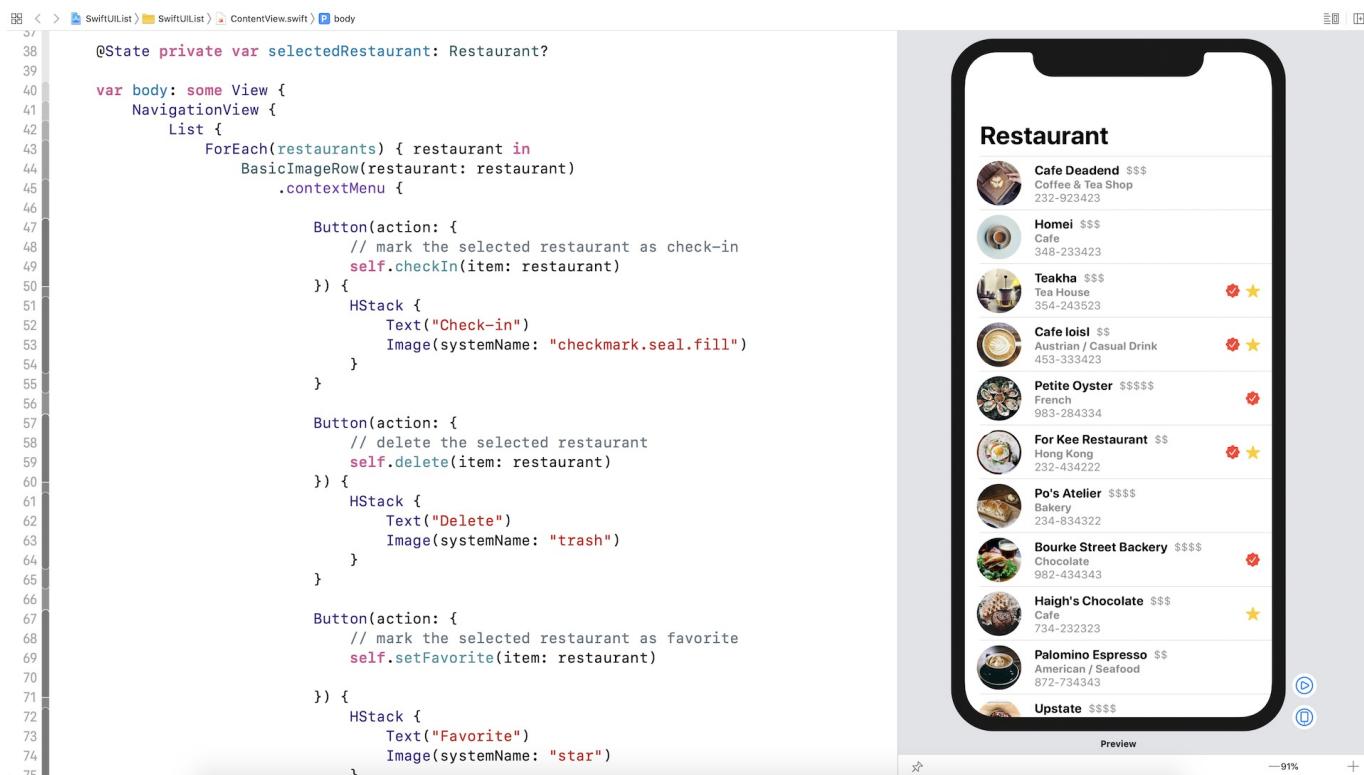


Figure 2. The restaurant list view

The `Restaurant` struct now has three more properties: `type`, `phone`, and `priceLevel`. I think both type and phone are self explanatory. For the price level, it 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 sample 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 to you its usage in words, 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 *SwiftUIList* folder and choose "New File...." Next, select to use *SwiftUI View* as the template and name the file *SettingView.swift*.



Figure 3. Creating a new SwiftUI file

Now, let's start to create the form. Replace `SettingView` like 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, all you need is to use the `Form` container. Inside it, you create the required UI components. 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.

The screenshot shows the Xcode interface with the SettingView.swift file open. The code defines a struct SettingView that returns a NavigationView containing a Form with two sections: SORT PREFERENCE and FILTER PREFERENCE, each with a single text entry. A preview of the app's interface is shown on the right, titled "Settings", displaying the two sections.

```

1 // 
2 //  SettingView.swift
3 //  SwiftUIList
4 //
5 //  Created by Simon Ng on 25/9/2019.
6 //  Copyright © 2019 AppCoda. All rights reserved.
7 //
8
9 import SwiftUI
10
11 struct SettingView: View {
12     var body: some View {
13         NavigationView {
14             Form {
15                 Section(header: Text("SORT PREFERENCE")) {
16                     Text("Display Order")
17                 }
18
19                 Section(header: Text("FILTER PREFERENCE")) {
20                     Text("Filters")
21                 }
22             }
23
24             .navigationBarTitle("Settings")
25         }
26     }
27 }
28
29 struct SettingView_Previews: PreviewProvider {
30     static var previews: some View {
31         SettingView()
32     }
33 }
34
35

```

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 users' input including a picker view, a toggle, and a stepper. Let's begin with the sort preference 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, how do you represent the above options in code? You may consider to use an array to hold the options. Okay, let's declare an array named `displayOrders` in `SettingView`:

```
private var displayOrders = [ "Alphabetical", "Show Favorite First", "Show Check-in 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 including 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 in form of `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.

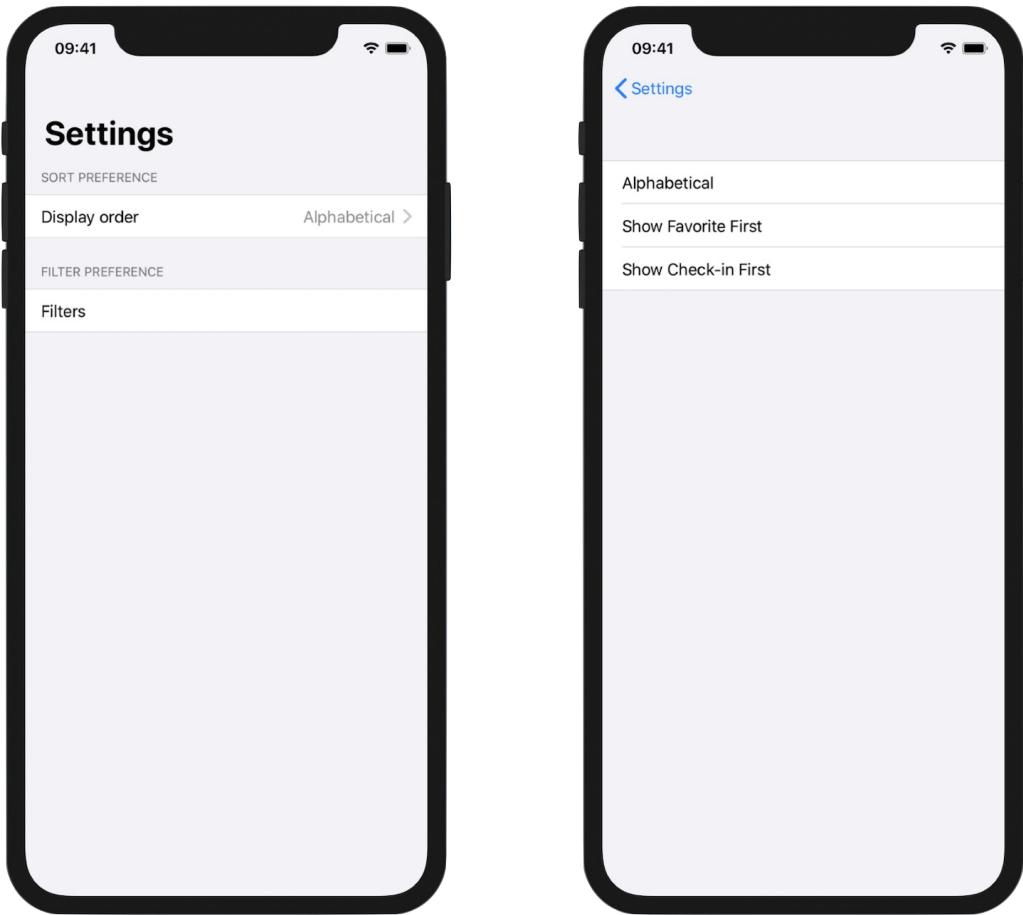


Figure 5. Using Picker view for display order selection

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*. It's particularly 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
```

Next, 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 with the current state of the toggle. In the closure, you present the description of the toggle. Here, we simply use a `Text` view.

That's the code you need to implement a toggle. 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.

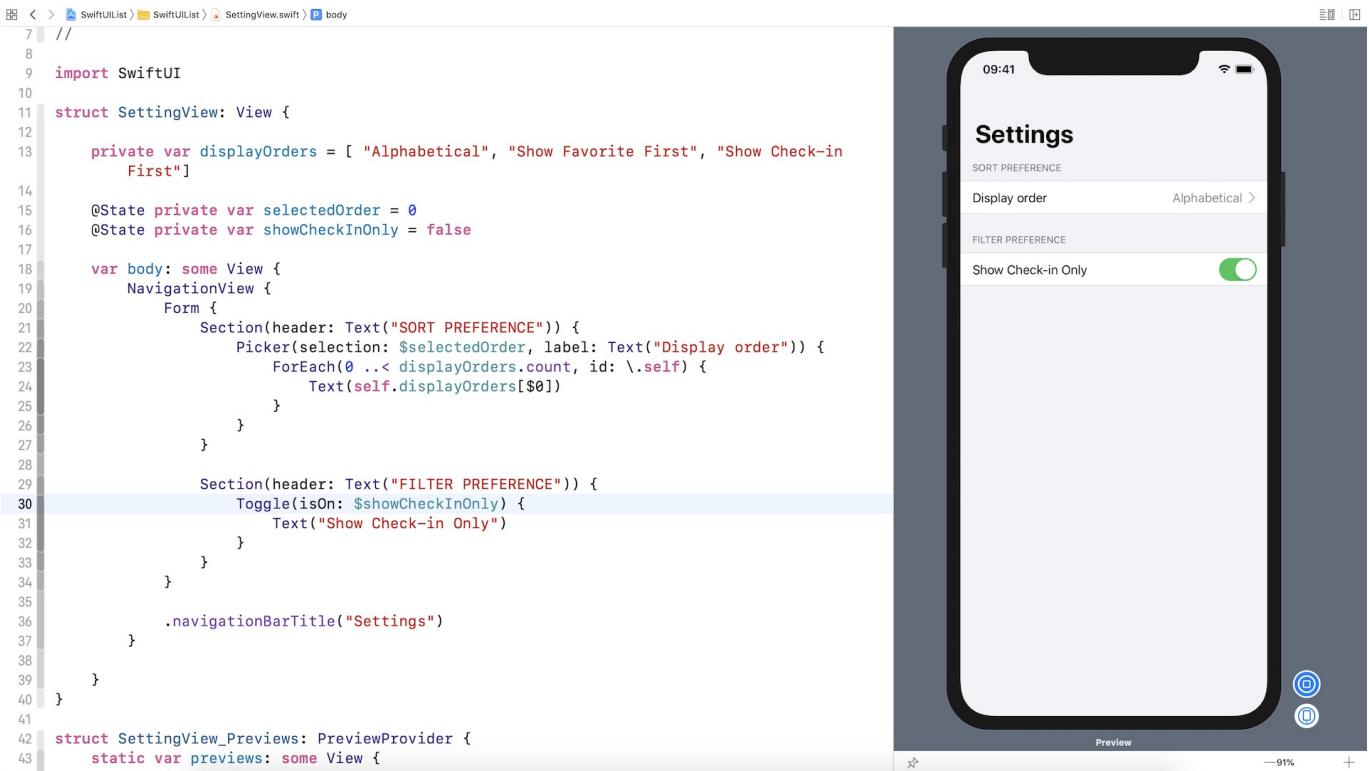


Figure 6. Showing a toggle switch

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 restaurant has a pricing indicator of the range 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 *plus* and *minus* buttons to perform increment and decrement actions.

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 price level of the user selection.

Declare the state variable in `SettingView` like this:

```
@State private var maxPriceLevel = 5 {  
    didSet {  
        if maxPriceLevel > 5 {  
            maxPriceLevel = 5  
        }  
  
        if maxPriceLevel < 1 {  
            maxPriceLevel = 1  
        }  
    }  
}
```

By default, we set the `maxPriceLevel` to `5`. Since the price level is in the range of `1` to `5`, we create a `didSet` to make sure the value of `maxPriceLevel` is between the value of `1` and `5`.

Now 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  
    }, onDecrement: {  
        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. In the closure, we display the text description of the filter preference. The maximum price level is indicated by dollar signs.

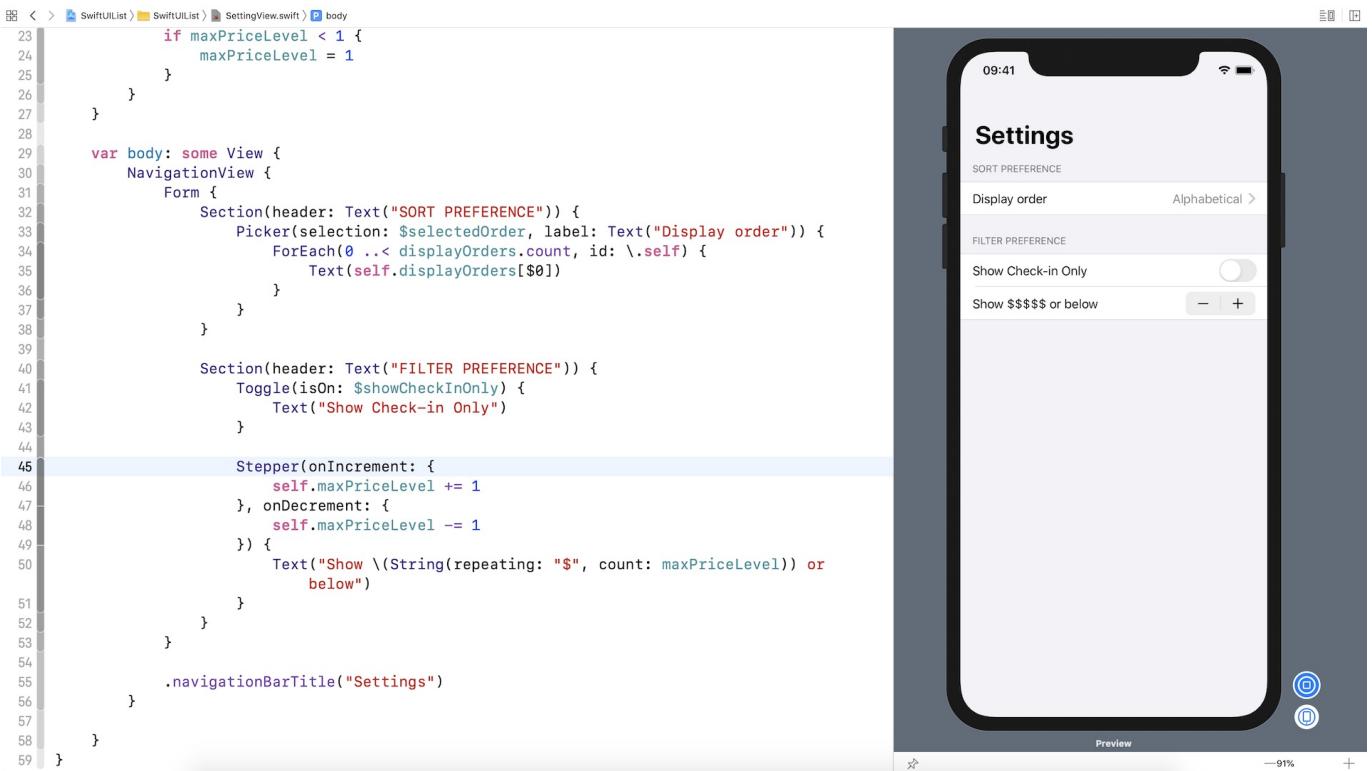


Figure 7. Implementing a stepper

To test the stepper, click the *Play* button to run the app. The number of \$ sign 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 further 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. If you run the app and click the *gear* icon, it should bring up the *Setting* view.

The screenshot shows the Xcode interface with the ContentView.swift file open. The code is a SwiftUI view definition for a restaurant list. A specific section of the code is highlighted in blue, which corresponds to the gear icon in the navigation bar of the simulator preview. The preview shows a list of restaurants with their names, cuisine type, price range, and phone numbers.

```

72     }
73     HStack {
74         Text("Favorite")
75         Image(systemName: "star")
76     }
77 }
78 .onTapGesture {
79     self.selectedRestaurant = restaurant
80 }
81 }
82 .onDelete { (indexSet) in
83     self.restaurants.remove(atOffsets: indexSet)
84 }
85 }
86 }
87 .navigationBarTitle("Restaurant")
88 .navigationBarItems(trailing:
89     Button(action: {
90         self.showSettings = true
91     }, label: {
92         Image(systemName: "gear").font(.title)
93         .foregroundColor(.black)
94     })
95 )
96 .sheet(isPresented: $showSettings) {
97     SettingView()
98 }
99 }
100 }
101 }
102 }
103
104 private func delete(item restaurant: Restaurant) {
105     if let index = self.restaurants.firstIndex(where: { $0.id == restaurant.id }) {
106         self.restaurants.remove(at: index)
107     }
108 }
109

```

Figure 8. Creating the navigation bar button

Exercise

Now the only way to dismiss the Setting view is by using the swipe-down gesture. In the modal view chapter, you've learned how to dismiss a modal view programmatically. As a revision 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, you just need to dismiss the setting view.

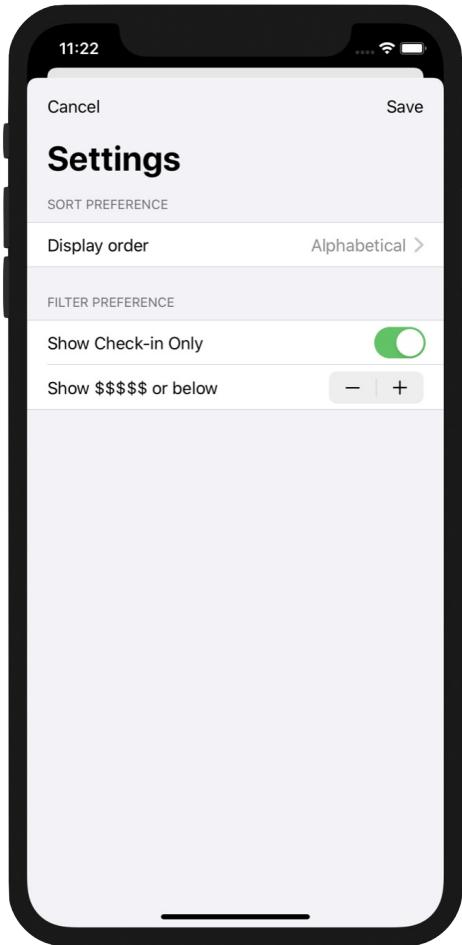


Figure 9. Adding two buttons (Save & Cancel) in the navigation bar

What's Coming Next

I hope you understand how the *Form* component works and you know how to build a form UI with components like Picker and Stepper. Up till now, the app can't store the user preference permanently. Every time you launch the app, the settings are reset to the original setting. In the upcoming chapter, I will show you how to save these settings in a local storage. More importantly, we will update the list view in accordance to the user's preferences.

For reference, you can download the complete project here:

- Demo project (<https://www.appcoda.com/resources/swiftui/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 setting 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/swiftui/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. On top of that, `Enum` in Swift is a first-class type in their own right. That means you can create instance methods to provide additional functionalities related to the values. Later, we will add a function for handling the filtering. Meanwhile, please create a new Swift file named `SettingStore.swift` to store the `Enum`.

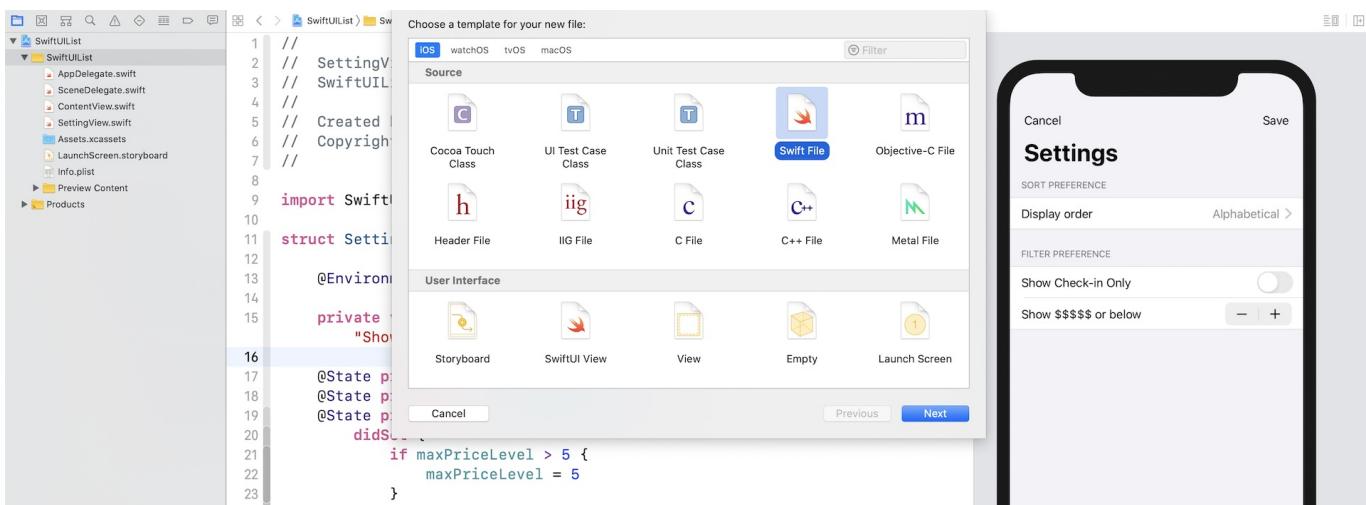


Figure 1. Creating a new Swift file

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-in 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*. If you compare it to the previous value, which is `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 find out all the display orders by accessing the `allCases` property, which contains an array of all the enum's cases.

Now you can test the *Setting* screen again. It should work and look the same. That said, 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 Setting screen reset to the 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 one of the options. This defaults system allows an app to store user's preference in key-value pairs. To interact with this defaults database, you can use a programmatic interface called `UserDefault`s .

In the `SettingStore.swift` file, we will create a `SettingStore` class to provide some convenience methods for saving and loading the user's preference. Insert the following code snippet in `SettingStore.swift` :

```

final class SettingStore {
    var defaults: UserDefaults

    init(defaults: UserDefaults = .standard) {
        self.defaults = defaults

        defaults.register(defaults: [

```

```
        "view.preferences.showCheckInOnly" : false,
        "view.preferences.displayOrder" : 0,
        "view.preferences.maxPriceLevel" : 5
    ])
}

var showCheckInOnly: Bool {
    get {
        defaults.bool(forKey: "view.preferences.showCheckInOnly")
    }

    set {
        defaults.set(newValue, forKey: "view.preferences.showCheckInOnly")
    }
}

var displayOrder: DisplayOrderType {
    get {
        DisplayOrderType(type: defaults.integer(forKey: "view.preferences.displayOrder"))
    }

    set {
        defaults.set(newValue.rawValue, forKey: "view.preferences.displayOrder")
    }
}

var maxPriceLevel: Int {
    get {
        defaults.integer(forKey: "view.preferences.maxPriceLevel")
    }

    set {
        defaults.set(newValue, 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 preference is not found in the database.

As mentioned, you can save the settings in key-value pairs with `UserDefaults`. In the code above, we declare three computed properties for this purpose. In the setter, we use the `set` method of `UserDefaults` to save the value in the default system. In the getter, we load the value from database with the specific key.

With the `SettingStore` ready, let's switch over to the `ContentView.swift` file to implement the *Save* operation. First, declare a property in `SettingView` for the `SettingStore`:

```
var settingStore: SettingStore
```

For the *Save* button, update the code like 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 insert three lines of code to save the user's preference. To load the preference when the *Setting* 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, so we load the user's setting 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 just 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.

```
17 func scene(_ scene: UIScene, willConnectTo session: UISceneSession, options connectionOptions: UIScene.ConnectionOptions) {
18     // Use this method to optionally configure and attach the UIWindow `window` to the provided UIWindowScene `scene`.
19     // If using a storyboard, the `window` property will automatically be initialized and attached to the scene.
20     // This delegate does not imply the connecting scene or session are new (see
21     // `application:configurationForConnectingSceneSession` instead).
22
23     // Create the SwiftUI view that provides the window contents.
24     let contentView = ContentView()
```

Figure 2. An error in SceneDelegate.swift

Go to `SceneDelegate.swift` and add this property to create a `SettingStore` instance:

```
var settingStore = SettingStore()
```

Next, change the line code to the following to fix the error:

```
let contentView = ContentView(settingStore: settingStore)
```

Now you should be able to execute app and play around with the settings. Once you save the settings, it's stored permanently in the local defaults system. You can try to stop the app and launch it again. The saved settings should be loaded in the Setting screen.

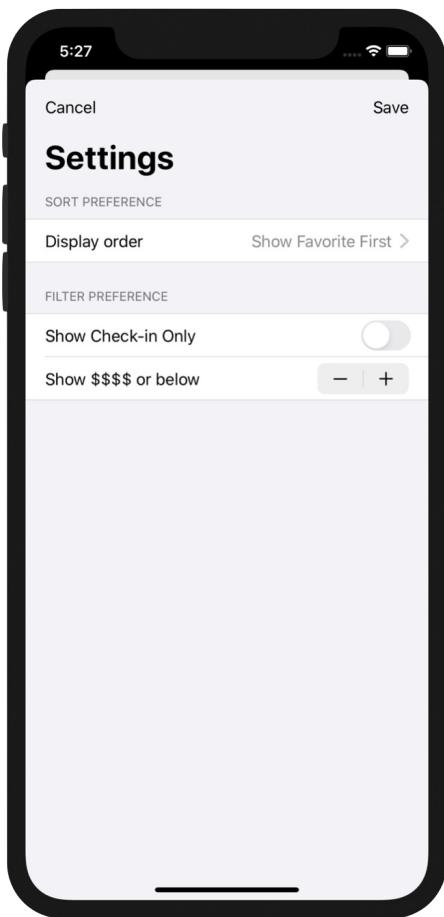


Figure 3. The Setting screen should load your user preference

Sharing Data Between Views Using @EnvironmentObject

Now that the user's preference is already saved in the local defaults system, what's missing is that the list view doesn't change in accordance to the user's setting. 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 *Setting* 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 is capable to monitor the changes of the defaults system and triggers the 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. When working along with SwiftUI, we can even trigger an update of a view without writing a line of code. Everything is handled behind the scene 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** - 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 is any 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 in this situation. 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 preference is changed? By implementing this protocol, the object can serve as a publisher that emits the changed value. Those subscribers that monitor the value change will get notified.
3. **@Published** - it's 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 setting view and the list view need to monitor the change of the user preference, `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 in front of the `defaults` property:

```
@Published var defaults: UserDefaults
```

By using the `@Published` property wrapper, the publisher, which is now the `SettingStore`, will let subscribers know whenever there is a value change of the `defaults` 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 to 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 purpose.

Okay, what we have done is on the *Publisher* side. So, how 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 purpose, 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 `SceneDelegate.swift` and update this line of code inside the `scene(_:willConnectTo:options:)` method:

```
let contentView = 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 views that monitor the updates. Though you can'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.

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.

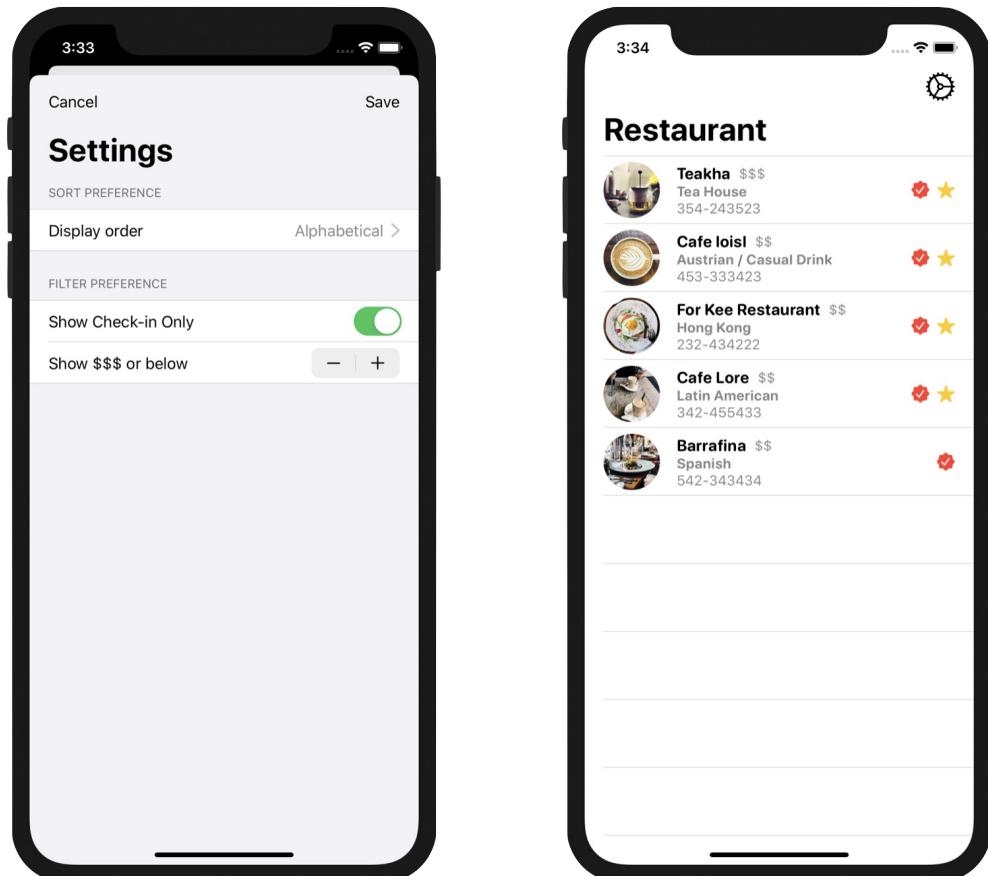


Figure 4. The list view now refreshes its items when you change the filter preference

Implementing the Sort Option

Now that we've completed the implementation of the filtering option, let's continue to 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 predicate 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 `DisplayOrder` enum. Create a new function in `DisplayOrder` 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! You can 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

Do 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 change of an object and trigger the UI updates. Everything is nearly automatic and taken care 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 modular code.

For reference, you can download the complete project here:

- Demo project (<https://www.appcoda.com/resources/swiftui/SwiftUIFormData.zip>)

Chapter 15

Building a Registration Form with Combine and View Model

Now that you should have some basic ideas about Combine, let's continue to explore how Combine can make SwiftUI really shine. When developing a real-world app, it's very common to have a user registration for people to sign up 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.

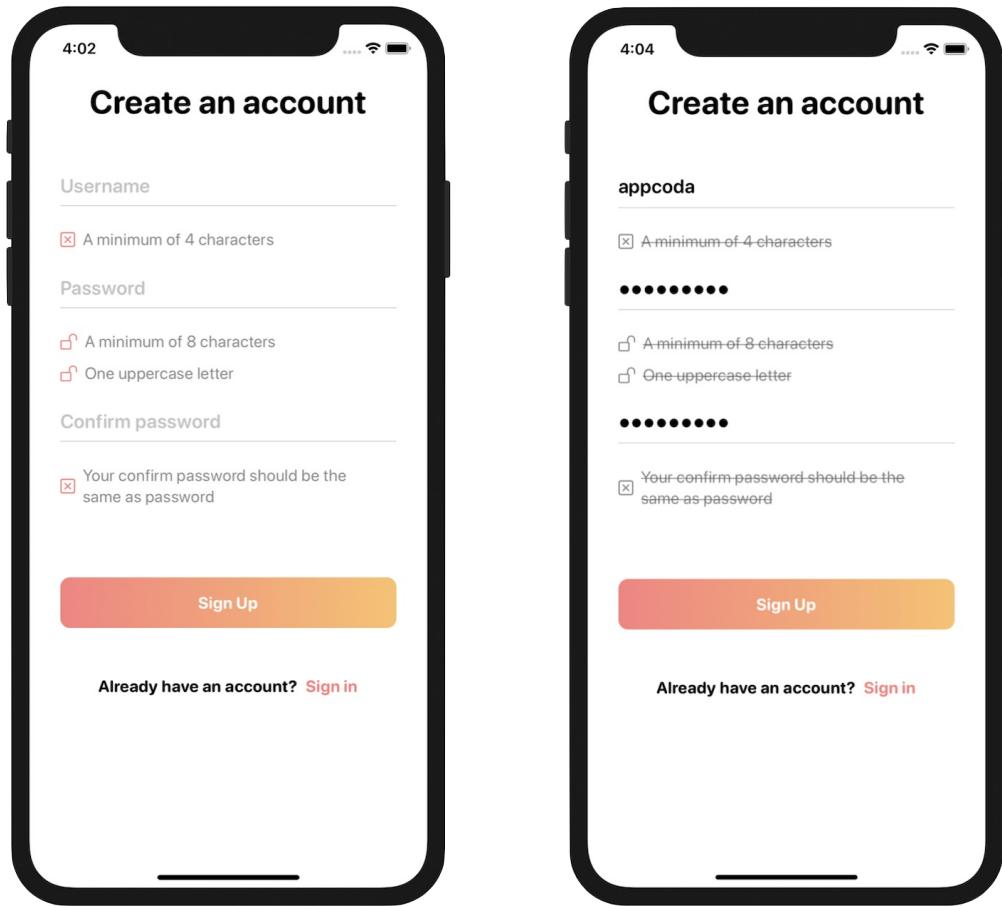


Figure 1. User registration demo

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 some 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 into the solution.

Okay, are you able to create the form? Even if you can't finish the exercise, that's completely fine. Now, download this project from <https://www.appcoda.com/resources/swiftui/SwiftUIFormRegistrationUI.zip>. I will go through my solution with you.

```
9 import SwiftUI
10
11 struct ContentView: View {
12
13     @State private var username = ""
14     @State private var password = ""
15     @State private var passwordConfirm = ""
16
17     var body: some View {
18         VStack {
19             Text("Create an account")
20                 .font(.system(.largeTitle, design: .rounded))
21                 .bold()
22                 .padding(.bottom, 30)
23
24             FormField(fieldName: "Username", fieldValue: $username)
25             RequirementText(text: "A minimum of 4 characters")
26                 .padding()
27
28             FormField(fieldName: "Password", fieldValue: $password, isSecure: true)
29             VStack {
30                 RequirementText(iconName: "lock.open", text: "A minimum of 8 characters",
31                                 isStrikeThrough: false)
32                 RequirementText(iconName: "lock.open", text: "One uppercase letter",
33                                 isStrikeThrough: false)
34             }
35             .padding()
36
37             FormField(fieldName: "Confirm Password", fieldValue: $passwordConfirm,
38                     isSecure: true)
39             RequirementText(text: "Your confirm password should be the same as password",
40                             isStrikeThrough: false)
41             .padding()
42             .padding(.bottom, 50)
43
44             Button(action: {
45                 // Proceed to the next screen
46             }) {
47
48             }
49
50         }
51     }
52 }
```

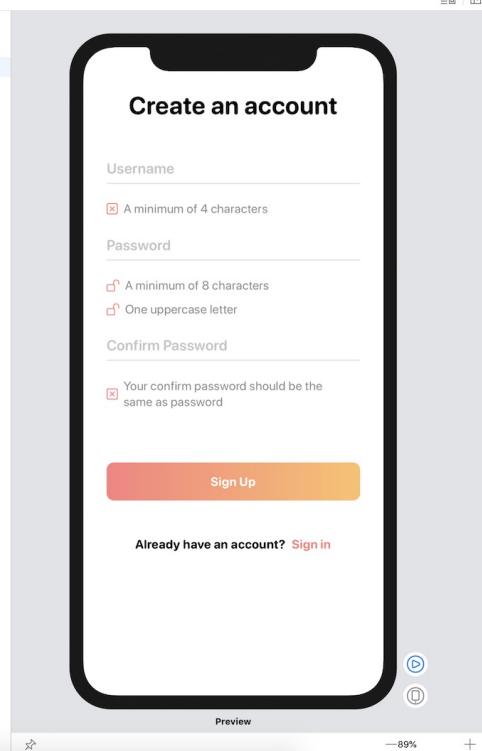


Figure 2. The starter project

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 and 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 requirement text (see figure 3)? If you look at any of the requirement text, it has an icon and a description. Instead of creating each of the requirement text from scratch, we can generalize the code and build a generic one for it.

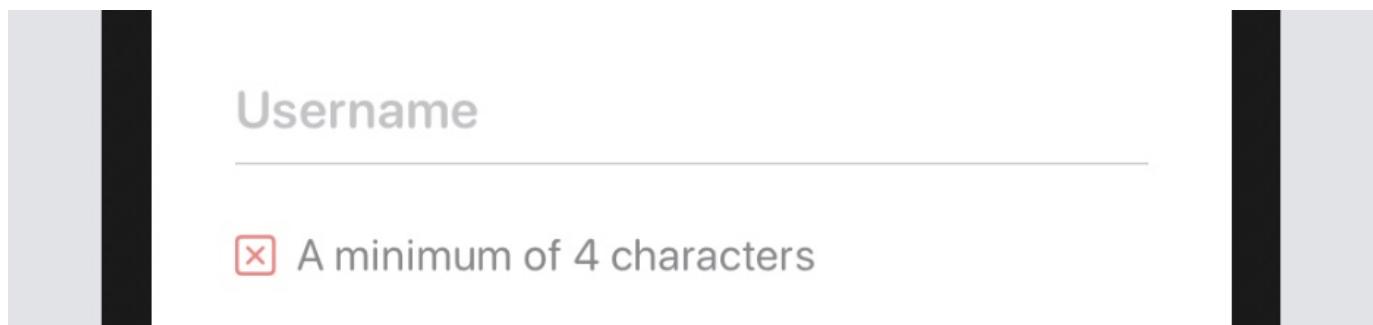


Figure 3. A sample text field and its requirement text

The `RequirementText` view has four properties including `iconName`, `iconColor`, `text`, and `isStrikeThrough`. It's flexible enough to support different styles of the requirement 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 requirement 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 minimum 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.

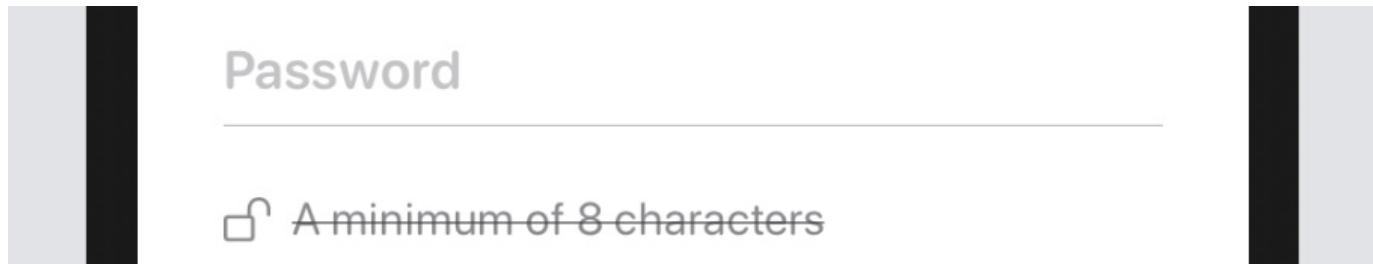


Figure 4. The requirement text is crossed out

Now that you should 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 some 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 field 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 can 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 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(red: 251/255, green: 128/255, blue: 128/255), Color(red: 253/255, green: 193/255, blue: 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 would 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 details. You can always refer to the Button chapter.

Lastly, it's the description text of *Already have an account*. This text and the *Sign in* button are completely optional. I just want to mimic the layout of a common sign up form.

That's how I lay out the user registration screen. If you have tried out the exercise, you may come up with a different solution. That's completely fine. Here I just want to show you one of the approaches to build 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 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. Say, each keystroke a user enters in the text field triggers an event and this can become 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 continue the processing. Let's say, the subscriber can use that value to determine if the submit button should be enabled or not.

You may wonder `@Published` works pretty much like `@State` in SwiftUI. While it works pretty much the same for this example, `@State` can only apply 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 should have some very 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 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, you know 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 these requirement text, indicating whether they should be crossed out:

- A minimum of 4 characters (username)
- A minimum of 8 characters (password)
- One uppercase letter (password)
- Your confirm password should the same as 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 can we validate the username, password, and passwordConfirm in accordance to the requirements?

With Combine, you have to develop a publisher/subscriber mindset to solve the problem. Consider the username, we actually have two publishers here: `username` and `isUsernameLengthValid`. The `username` publisher emits value changes whenever the user keys 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 requirement 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.

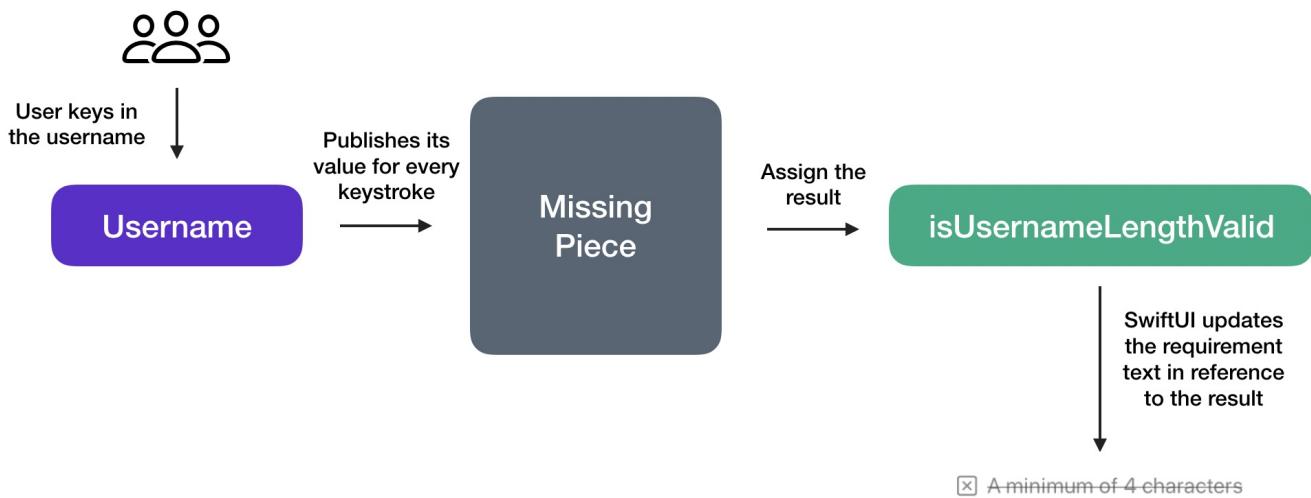


Figure 6. The `username` and `isUsernameLengthValid` publishers

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 and go 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 known as an operator in Combine that it takes an input, processes it, and transforms the input into something that the subscriber expects. So, what we did in the code above is that:

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.usernameLengthValid ? Color  
.secondary : Color(red: 251/255, green: 128/255, blue: 128/255), text: "A minimum  
of 4 characters", isStrikeThrough: userRegistrationViewModel.usernameLengthValid  
)
```

Both the icon color and the status of strike through depend on the validation result (i.e. `usernameLengthValid`).

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

If you understand how the validation of the username field is done, we will apply a similar implementation for the password and password confirm validation.

For the password field, it has two requirements:

1. The length of password should have at least 8 characters.
2. It should contain at least one uppercase letter.

To do so, we can 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 assign it to the `isPasswordLengthValid` property. The second one is for handling the validation of 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 to a 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 approach, 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 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 = []

    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 one we discussed earlier, but one thing you may notice is the `cancellableSet` variable. And, for each of the subscribers, we call the `store` function at the very end.

What are they?

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 update 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.

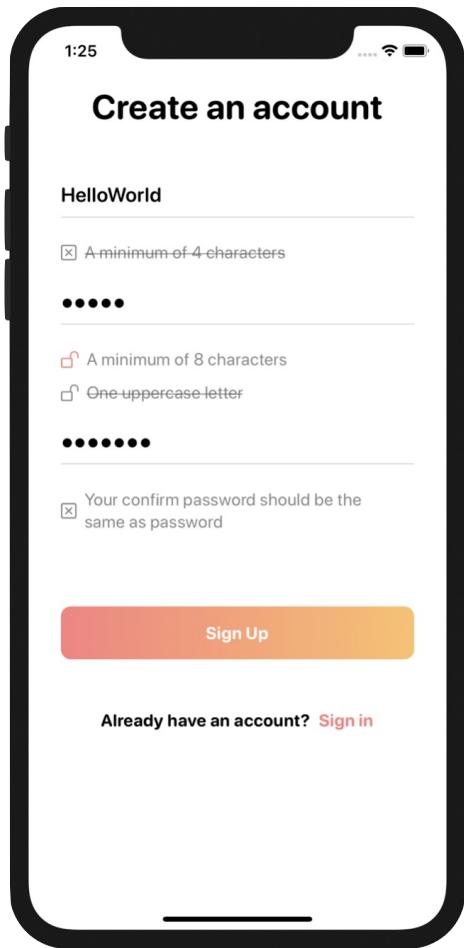


Figure 7. The registration form now validates the user input

Summary

I hope you now equip yourself with some basic knowledge with 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, but it's the first time Apple released their own functional reactive framework. To me, it's a major paradigm shift. The company finally took position on FRP and recommend Apple developers to embrace this new programming methodology.

Like the introduction of any new technologies, there must be a learning curve. Even if you've been programming in iOS before, it will take you some time to move from the programming methodology of delegates to publishers/subscribers.

However, once you manage the Combine framework, you will be very glad that it will help you achieve more maintainable and modular code. Together with SwiftUI, as you can see, communication between a view and a view model is a breeze.

For reference, you can download the complete project here:

- Demo project
(<https://www.appcoda.com/resources/swiftui/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

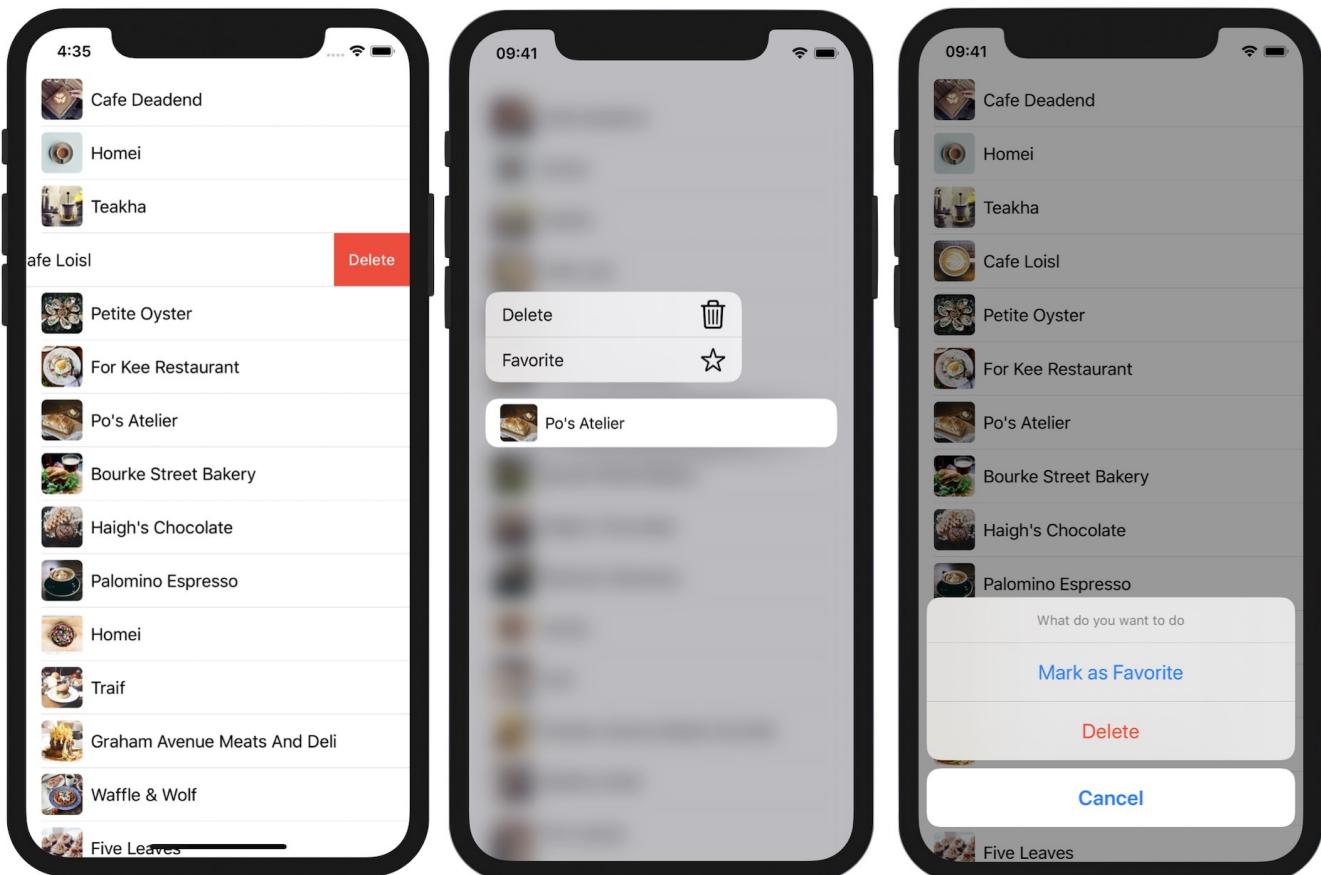


Figure 1. Swipe to delete (left), context menu, and action sheet (right)

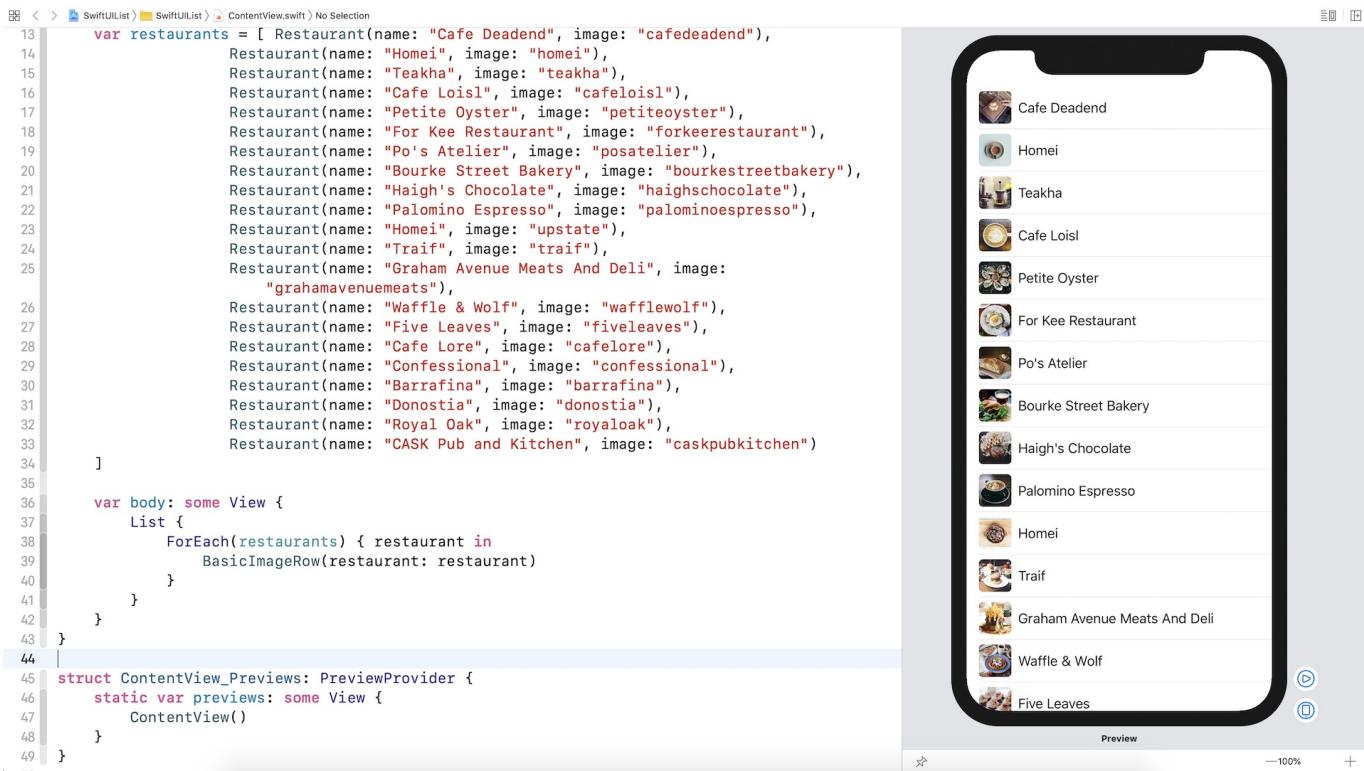
Referring to figure 1, I believe you should be very familiar with swipe-to-delete and action sheet. These two UI elements have been existed in iOS for several years. Context menus are newly introduced in iOS 13, though they look similar to peek and pop of 3D Touch. For any views (e.g. button) implemented 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 list app created before. You can download the starter project from

<https://www.appcoda.com/resources/swiftui/SwiftUINavigationListStarter.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.



```

13 var restaurants = [ Restaurant(name: "Cafe Deadend", image: "cafedeadend"),
14     Restaurant(name: "Homei", image: "homei"),
15     Restaurant(name: "Teakha", image: "teakha"),
16     Restaurant(name: "Cafe Loisl", image: "cafeloisl"),
17     Restaurant(name: "Petite Oyster", image: "petiteoyster"),
18     Restaurant(name: "For Kee Restaurant", image: "forkeerrestaurant"),
19     Restaurant(name: "Po's Atelier", image: "posatelier"),
20     Restaurant(name: "Bourke Street Bakery", image: "bourkestreetbakery"),
21     Restaurant(name: "Haigh's Chocolate", image: "haighschocolate"),
22     Restaurant(name: "Palomino Espresso", image: "palominoespresso"),
23     Restaurant(name: "Homei", image: "upstate"),
24     Restaurant(name: "Traif", image: "traif"),
25     Restaurant(name: "Graham Avenue Meats And Deli", image:
26         "grahamavenuemeats"),
27     Restaurant(name: "Waffle & Wolf", image: "wafflewolf"),
28     Restaurant(name: "Five Leaves", image: "fiveleaves"),
29     Restaurant(name: "Cafe Lore", image: "cafelore"),
30     Restaurant(name: "Confessional", image: "confessional"),
31     Restaurant(name: "Barrafina", image: "barrafina"),
32     Restaurant(name: "Donostia", image: "donostia"),
33     Restaurant(name: "Royal Oak", image: "royaloak"),
34     Restaurant(name: "CASK Pub and Kitchen", image: "caskpubkitchen")
35 ]
36
37 var body: some View {
38     List {
39         ForEach(restaurants) { restaurant in
40             BasicImageRow(restaurant: restaurant)
41         }
42     }
43 }
44
45 struct ContentView_Previews: PreviewProvider {
46     static var previews: some View {
47         ContentView()
48     }
49 }

```

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 to implement 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`, it'll pass you 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 property changes.

To do that, insert the `@State` keyword to the `restaurants` variable:

```
@State var restaurants = [ ... ]
```

Once you made the change, you should be able to execute 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?

The screenshot shows the Xcode interface with the ContentView.swift file open. The code defines a Content View with a state variable 'restaurants' containing a list of restaurants. It uses a List view with a swipe-to-delete feature. The simulator preview shows a list of restaurants with a red 'Delete' button appearing when swiping right on a row.

```

11 struct ContentView: View {
12
13     @State var restaurants = [ Restaurant(name: "Cafe Deadend", image: "cafedeadend"),
14         Restaurant(name: "Homei", image: "homei"),
15         Restaurant(name: "Teakha", image: "teakha"),
16         Restaurant(name: "Cafe Loisl", image: "cafeloisl"),
17         Restaurant(name: "Petite Oyster", image: "petiteoyster"),
18         Restaurant(name: "For Kee Restaurant", image: "forkeerestaurant"),
19         Restaurant(name: "Po's Atelier", image: "posatelier"),
20         Restaurant(name: "Bourke Street Bakery", image: "bourkestreetbakery"),
21         Restaurant(name: "Haigh's Chocolate", image: "haighschocolate"),
22         Restaurant(name: "Palomino Espresso", image: "palominoespresso"),
23         Restaurant(name: "Homei", image: "upstate"),
24         Restaurant(name: "Traif", image: "traif"),
25         Restaurant(name: "Graham Avenue Meats And Deli", image:
26             "grahamavenuemeats"),
27         Restaurant(name: "Waffle & Wolf", image: "wafflewolf"),
28         Restaurant(name: "Five Leaves", image: "fiveleaves"),
29         Restaurant(name: "Cafe Lore", image: "cafelore"),
30         Restaurant(name: "Confessional", image: "confessional"),
31         Restaurant(name: "Barrafina", image: "barrafina"),
32         Restaurant(name: "Donostia", image: "donostia"),
33         Restaurant(name: "Royal Oak", image: "royaloak"),
34         Restaurant(name: "CASK Pub and Kitchen", image: "caskpubkitchen")
35     ]
36
37     var body: some View {
38         List {
39             ForEach(restaurants) { restaurant in
40                 BasicImageRow(restaurant: restaurant)
41             }
42             .onDelete { (indexSet) in
43                 self.restaurants.remove(atOffsets: indexSet)
44             }
45         }
46     }
47 }

```

Figure 3. Deleting an item from the list

If you've written the same feature using UIKit before, I'm sure you should be amazed by SwiftUI. With just a few lines of code and a keyword, you already implemented the swipe-to-delete feature.

Creating a Context Menu

Next, let's talk about context menus, a new feature that is just 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 the device doesn't support 3D Touch. To bring up a context menu, people uses 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. You just need to 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. In the menu, it 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 can 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)  
    }  
}
```

Right now, we haven't implemented any of the button actions. However, if you execute the app, the app will bring up the context menu when you touch and hold one of the rows.

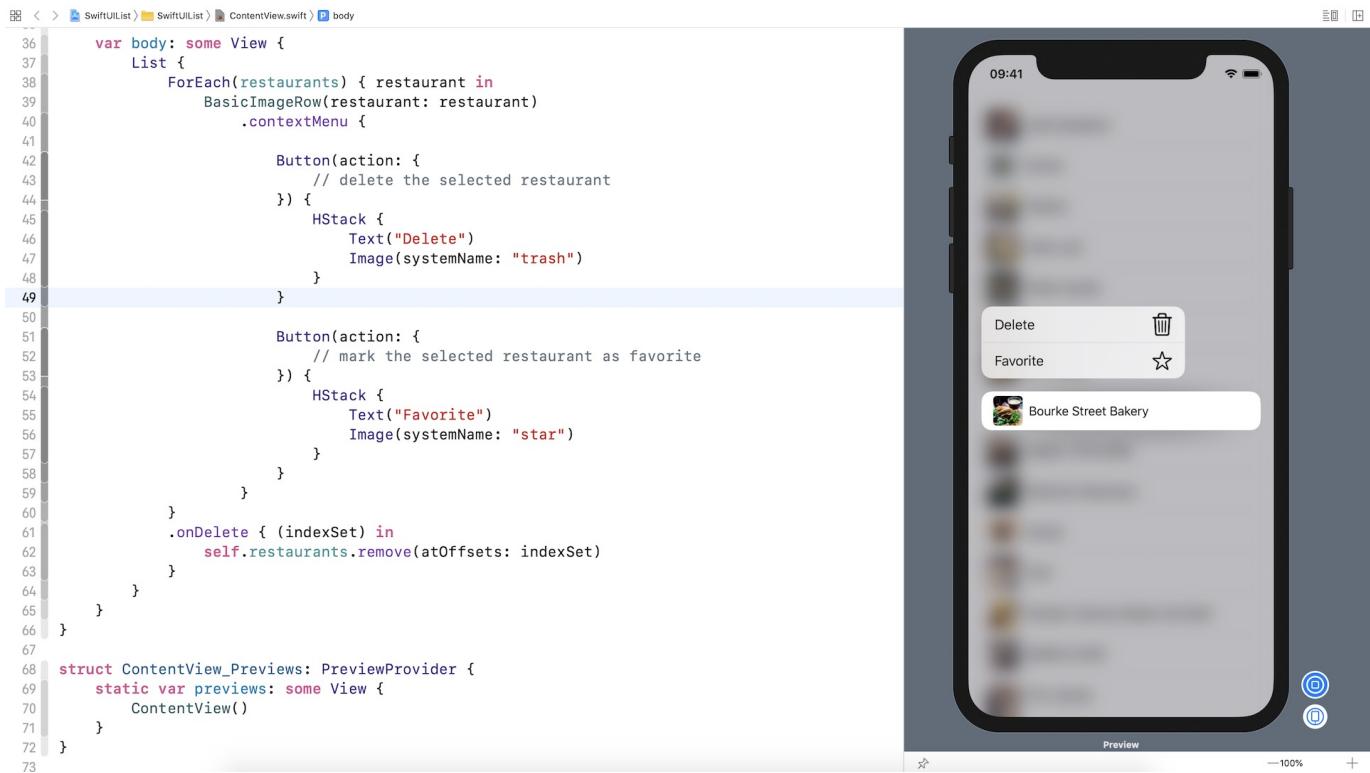


Figure 4. Deleting an item from the list

Now let's continue to implement 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. What the function does is loop through the array and compare 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 users select 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. 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 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. Say, 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 of 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 appear at the end of the row.

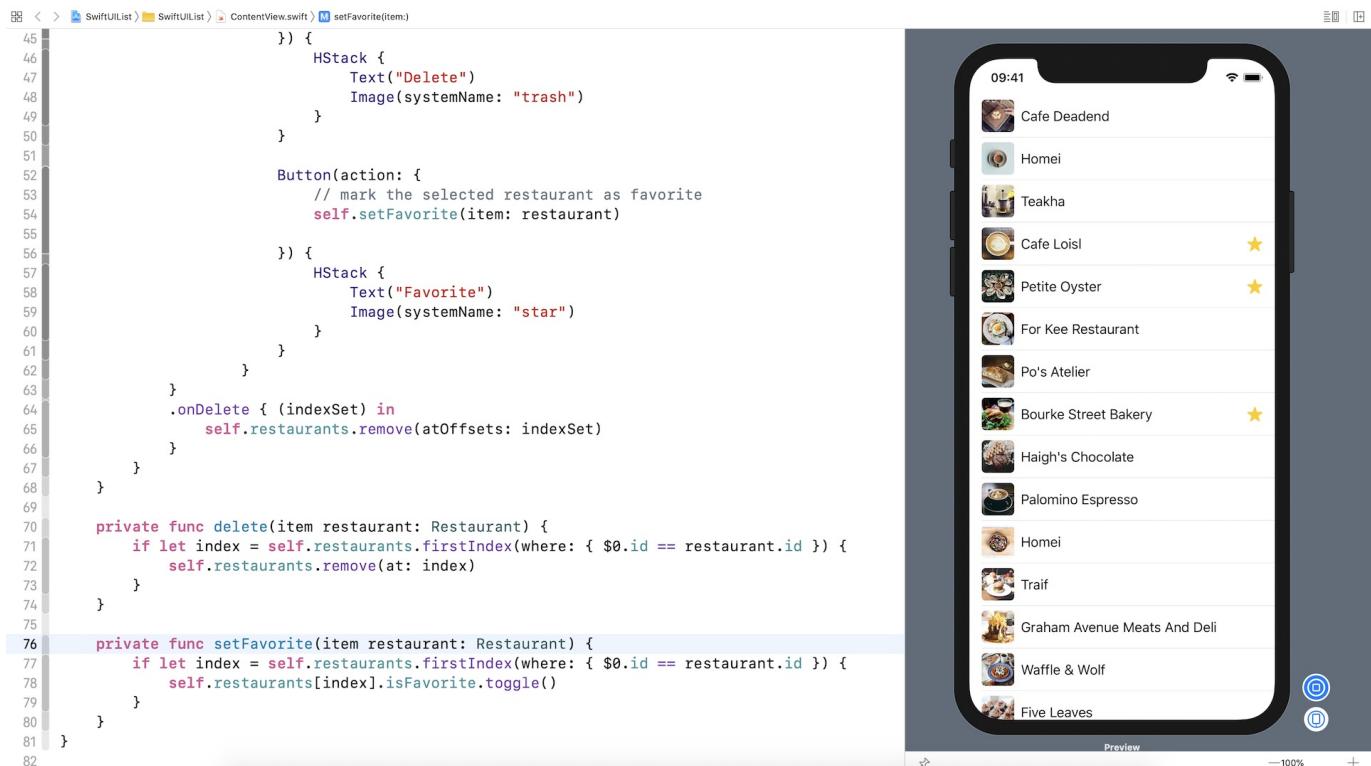


Figure 5. Using the Favorite function

Working with Action Sheets

That's how you implement context menus. Lastly, let's see how to create an action sheet in SwiftUI. The action sheet, that we're going to build, provides the same options as the context menu. If you forgot how 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, it provides a default button titled *Delete*.

To activate an action sheet, you attach the `actionSheet` modifier to a button or any views triggers the action sheet. 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 View
```

Or through an optional binding:

```
func actionSheet<T>(item: Binding<T?>, content: (T) -> ActionSheet) -> some View where 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 users select 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, buttons: [
                .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)
    }
}

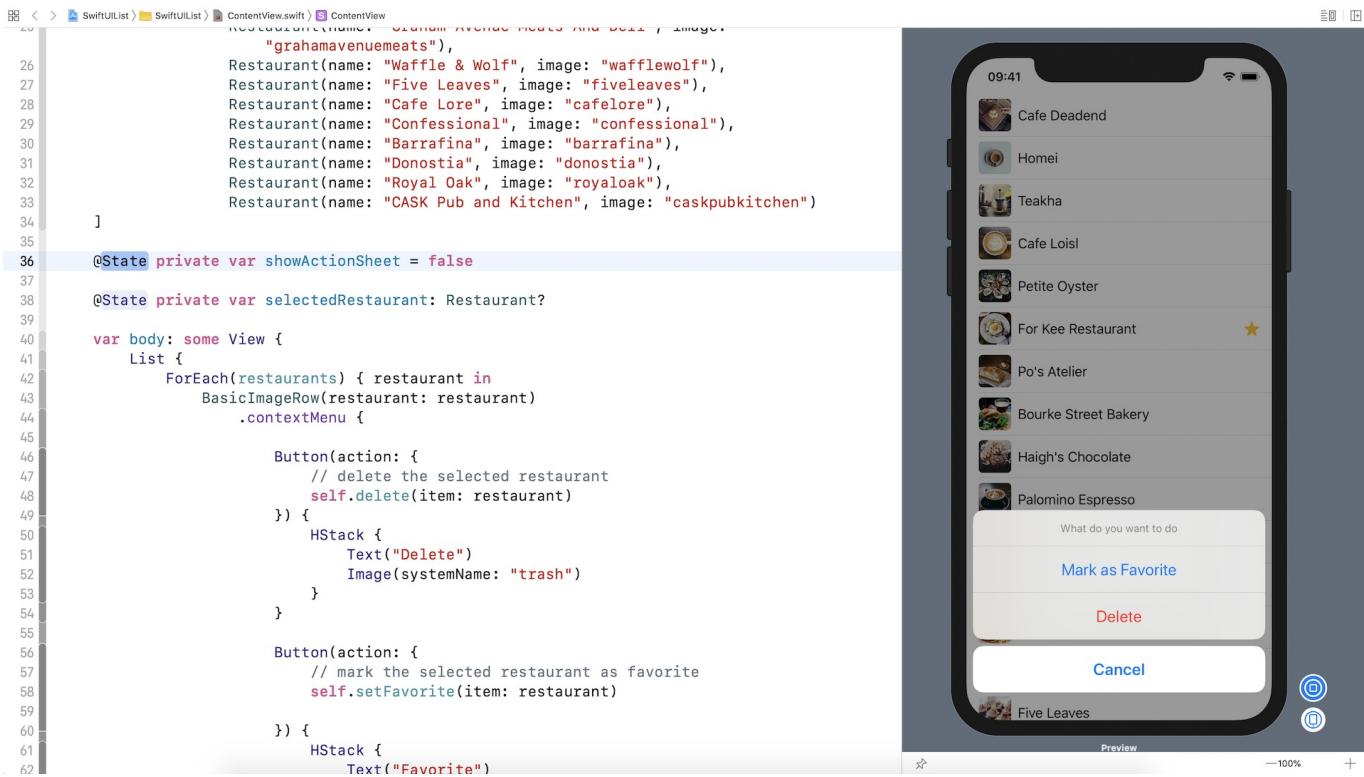
```

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 already 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 action. A destructive button is very similar to a default button but the font color is set to red to indicate some destructive actions such as delete. The *cancel* button is a special type for dismissing the action sheet.

For the *Mark as Favorite* button, we create it as a default button. In the `action` closure, we call the `setFavorite` function to add the *star*. For the *Delete* button, it's created as a destructive button. 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.



```

26
27
28
29
30
31
32
33
34
35
36 @State private var showActionSheet = false
37
38 @State private var selectedRestaurant: Restaurant?
39
40 var body: some View {
41     List {
42        ForEach(restaurants) { restaurant in
43             BasicImageRow(restaurant: restaurant)
44                 .contextMenu {
45
46                     Button(action: {
47                         // delete the selected restaurant
48                         self.delete(item: restaurant)
49                     }) {
50                         HStack {
51                             Text("Delete")
52                             Image(systemName: "trash")
53                         }
54                     }
55
56                     Button(action: {
57                         // mark the selected restaurant as favorite
58                         self.setFavorite(item: restaurant)
59                     }) {
60                         HStack {
61                             Text("Favorite")
62                         }
63                     }
64                 }
65             }
66         }
67     }

```

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 where 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) { _ in

    ActionSheet(title: Text("What do you want to do"), message: nil, buttons: [
        .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()
    ])
}

```

If the `selectedRestaurant` has a value, the app will bring up the action sheet. 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 should have some ideas about how to build a context menu, let's have an exercise to test your knowledge. Your task is 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 spare some time to work on the exercise before checking out the solution. Have fun!

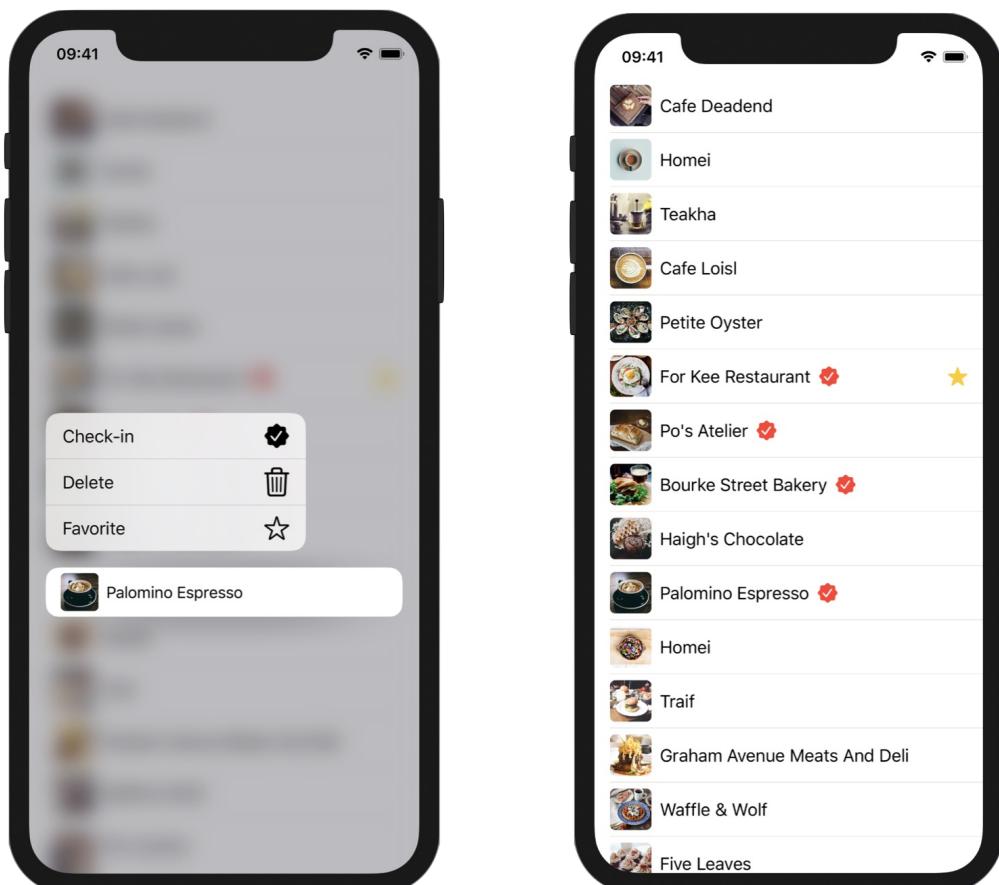


Figure 7. Adding a check-in feature

For reference, you can download the complete project here:

- Demo project and solution to exercise
(<https://www.appcoda.com/resources/swiftui/SwiftUIActionSheet.zip>)

Chapter 17

Understanding Gestures

In the earlier chapters, you already got a taste of building gestures with SwiftUI. We use 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. Other than that, *DragGesture*, *MagnificationGesture*, and *LongPressGesture* are some of the ready-to-use gestures. We will looking into a couple of them and see how we work with gestures in SwiftUI. On top of that, you will learn how to build a generic view that supports the drag gesture.

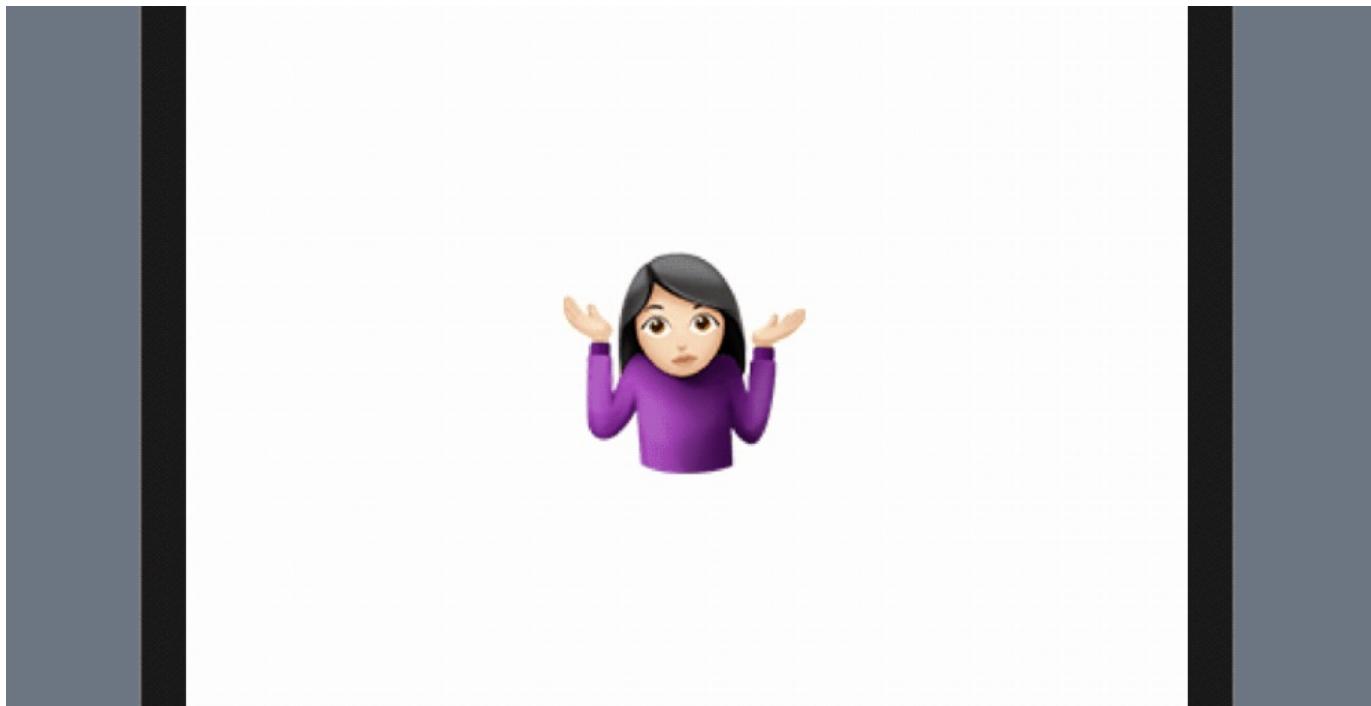


Figure 1. A demo showing the draggable view

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 *Single View Application* template and make sure you select *SwiftUI* as the UI 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 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.

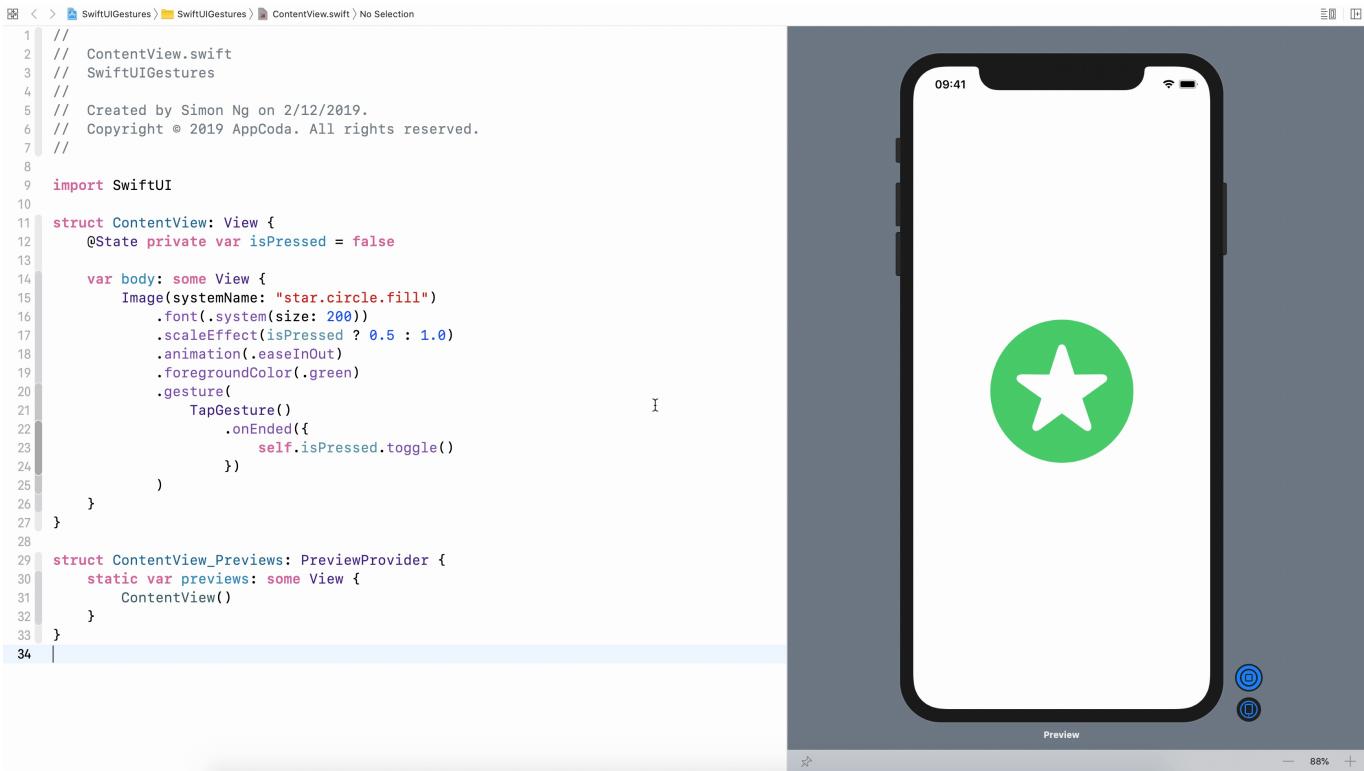


Figure 2. A simple scaling effect

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 have a quick test. 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 users 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 an immediate feedback when he/she taps the image. Any kind of feedback will help improving the situation. Say, we can 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 the work. Figure 3 illustrates how the animation works.

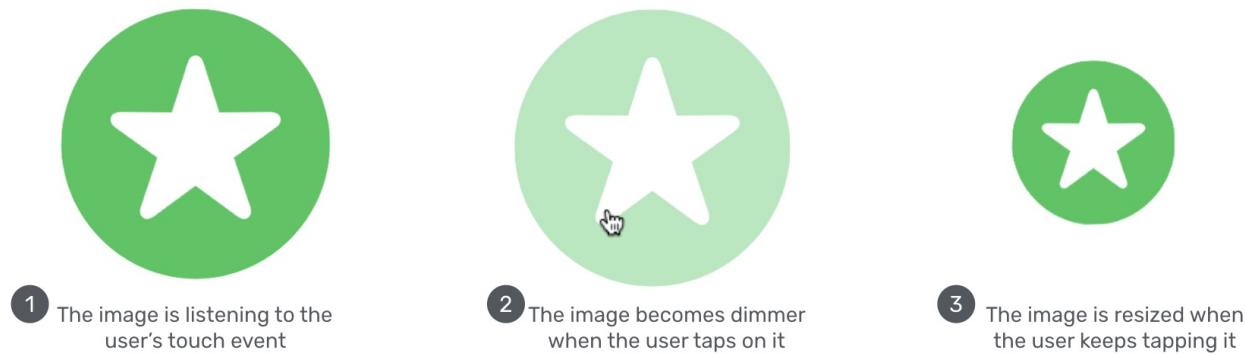


Figure 3. Applying a dimming effect when the image is tapped

To implement the animation, one of the tasks is to keep track of the state of gestures. During the performance of the long press gesture, we have to differentiate the tap and long press events. So, how can 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, it's the addition of the `.opacity` modifier. When the tap event is detected, we set the opacity value to `0.4` so that the image becomes dimmer.

Secondly, it's the `updating` method of the `LongPressGesture`. During the performance of the long press gesture, this method will be called and 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 made 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 finger. Have you wondered 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, that's `false` in our case.

Using Drag Gesture

Now that you should 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.

Now 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 to the 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. And, 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 the original value when the gesture ends. So, when you release the finger to end the drag, the `dragOffset` is reset to `.zero`, which is the original position.

But what if you want to keep the image to stay at the end point of the drag? How can you do that? Give yourself a few minutes to think about the implementation.

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 + dragOffset.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 in the code:

1. Other than the `update` function, we also 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 took 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.

```

1 // ContentView.swift
2 // SwiftUIGestures
3 // Created by Simon Ng on 2/12/2019.
4 // Copyright © 2019 AppCoda. All rights reserved.
5
6 import SwiftUI
7
8 struct ContentView: View {
9     @GestureState private var dragOffset = CGSize.zero
10    @State private var position = CGSize.zero
11
12    var body: some View {
13        Image(systemName: "star.circle.fill")
14            .font(.system(size: 100))
15            .offset(x: position.width + dragOffset.width, y: position.height +
16                dragOffset.height)
17            .animation(.easeInOut)
18            .foregroundColor(.green)
19            .gesture(
20                DragGesture()
21                    .updating($dragOffset, body: { (value, state, transaction) in
22
23                        state = value.translation
24
25                    })
26                    .onEnded({ (value) in
27                        self.position.height = self.position.height +
28                            value.translation.height
29                        self.position.width += value.translation.width
30
31                    })
32            )
33    }
34
35    struct ContentView_Previews: PreviewProvider {
36        static var previews: some View {
37            ContentView()
38        }
39    }
}

```

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, if 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. And, when you combine gestures exclusively, 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 exactly the type of the composition that we will use to sequence the long press and drag gestures.

To work with multiple gestures, you can 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 + dragOffset.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 can 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). This is why we have the `switch` statement to differentiate the gesture. 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, 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.

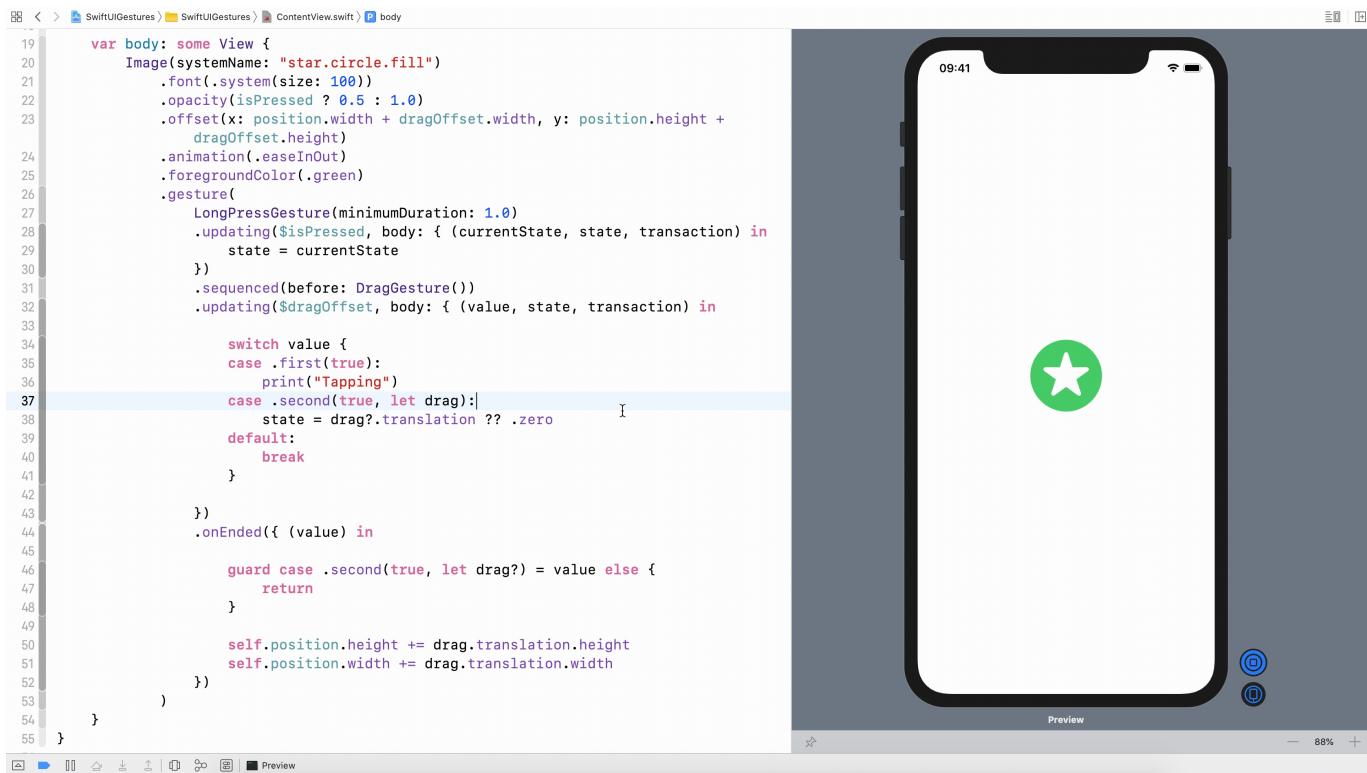


Figure 5. Dragging only happens when a user presses and holds the image for at least one second

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, it's associated 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.height + 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 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 a 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 always a better way to implement that. Let's see how we can build a generic draggable view.

In the project navigator, right click the `SwiftUIGestures` 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 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
        default:
            return false
        }
    }
}
```

```

        return true
    case .inactive:
        return false
    }
}

struct DraggableView<Content>: View where Content: View {
    @GestureState private var dragState = DragState.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.height + 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 the code are 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` and 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 initialize 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. Isn't it cool?

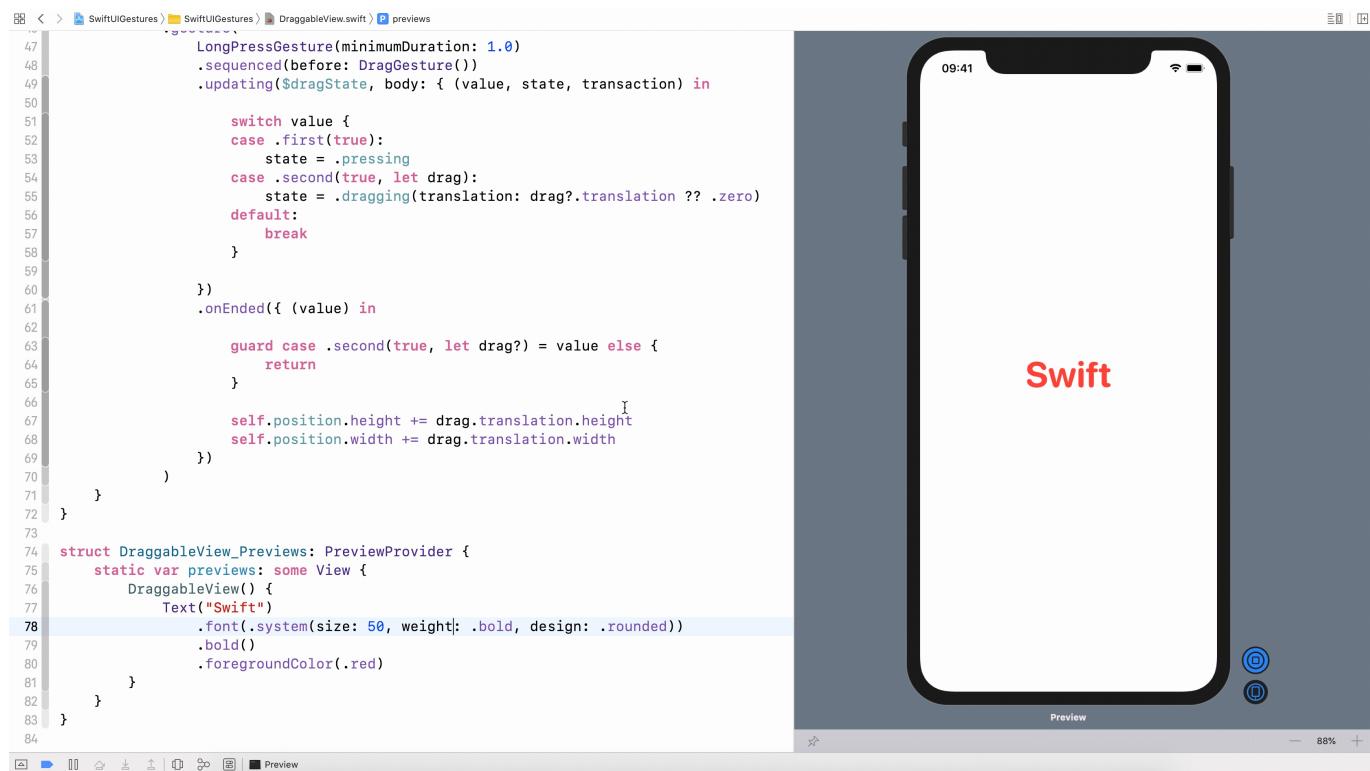


Figure 6. A draggable text view

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 can 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 it can recognize the `MagnificationGesture` and scale any given view accordingly. Figure 7 shows you a sample result.

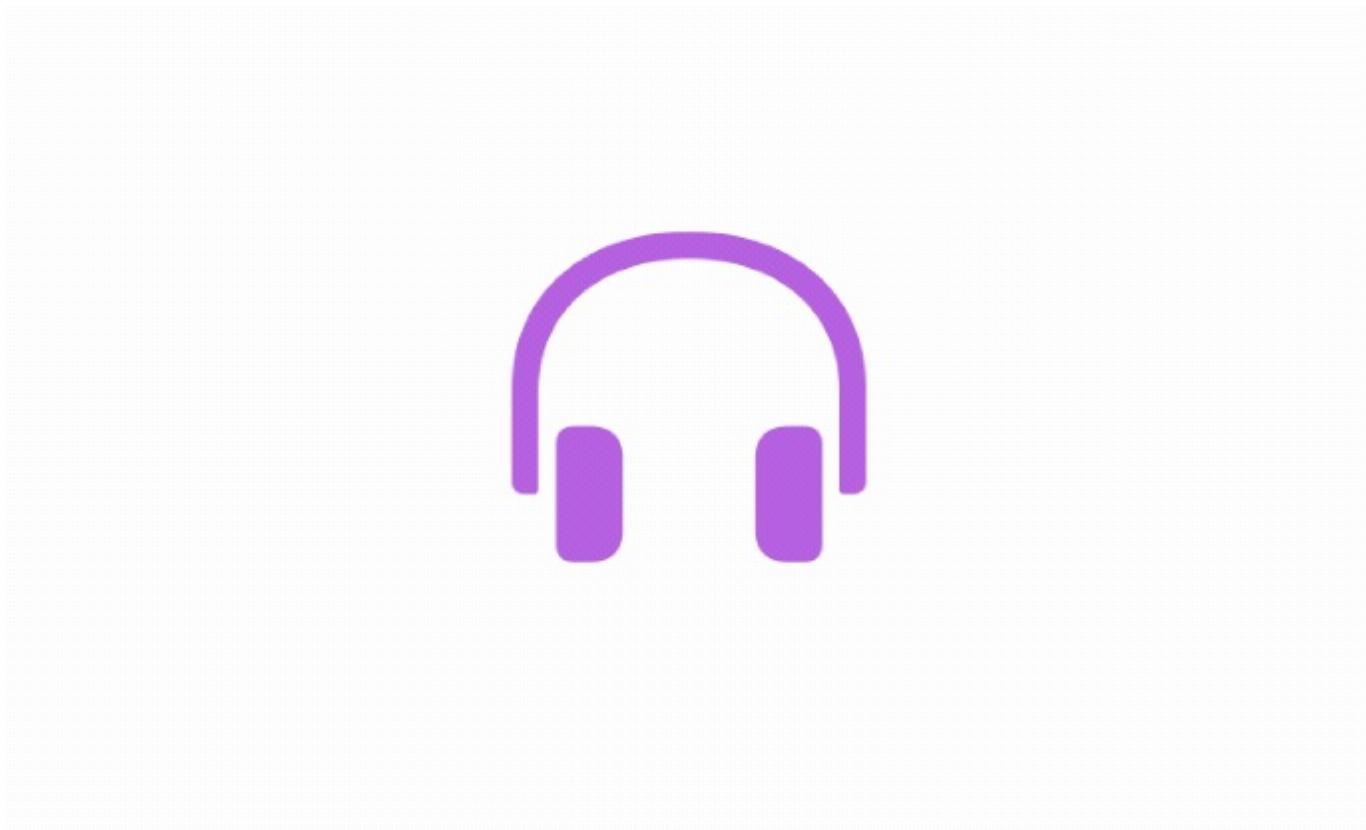


Figure 7. A scalable image view

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 gestures, all you need to do is attach it with the `.gesture` modifier. Composing multiple gestures has never been so simple.

It's a growing trend to build a gesture driven user interface 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 project here:

- Demo project (<https://www.appcoda.com/resources/swiftui/SwiftUIGestures.zip>)

Chapter 18

Building an Expandable Bottom Sheet with SwiftUI Gestures and GeometryReader

Now that you should 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 that you can easily find one in famous apps like Facebook and Uber. It's more like an enhanced version of action sheet that slides up from the bottom of screen and overlays on top of the original view to provide contextual information or additional options of the user's selection. For instance, when you save a photo to a collection in Instagram, the app shows you a bottom sheet to choose a collection. In the Facebook app, it displays the sheet with additonal action items when you click the ellipsis button of a post. 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) that 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.

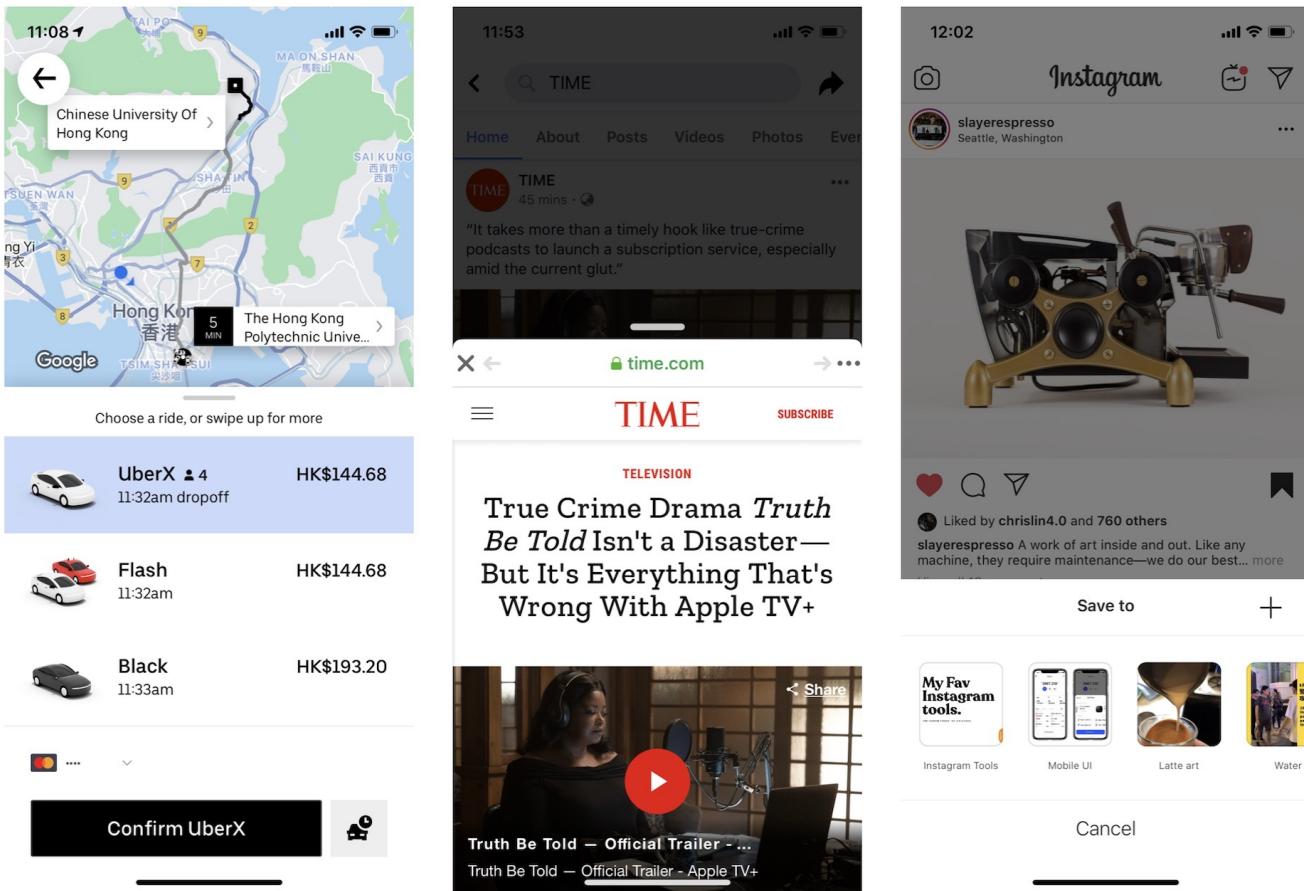


Figure 1. Uber, Facebook and Instagram all use bottom sheets in their apps

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.

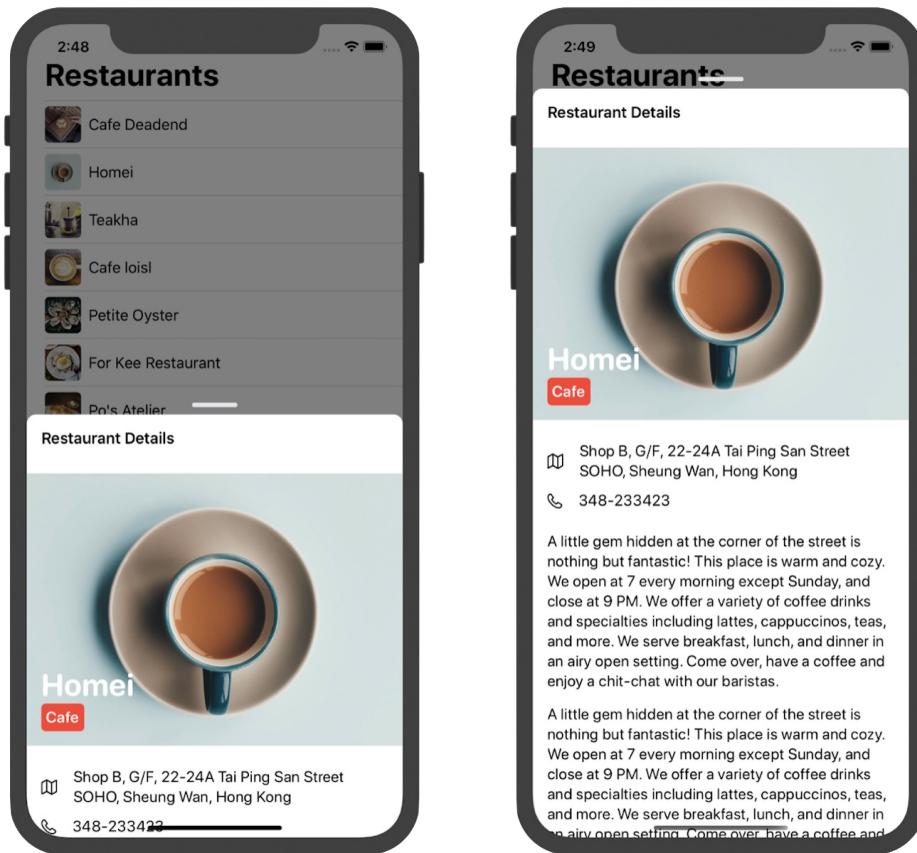


Figure 2. Building a expandable bottom sheet

Understanding the Starter Project

To save you some time from 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/swiftui/SwiftUIBottomSheetStarter.zip>. Unzip the file and open `SwiftUIBottomSheet.xcodeproj` to get started.

The starter project already comes with a set of restaurant images and the restaurant data. If you look into 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: "", image: "", isVisited: false)
    }
}
```

On top of that, I've created the main view for you that it 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.

The screenshot shows the Xcode interface with the ContentView.swift file open. The code defines a Content View that contains a navigation view with a list of restaurants. Each restaurant item has a thumbnail image and a name. The simulator preview shows a list of 15 restaurants, each with a small image and a name. The restaurants listed are: Cafe Deadend, Homei, Teakha, Cafe loisl, Petite Oyster, For Kee Restaurant, Po's Atelier, Bourke Street Backery, Haigh's Chocolate, Palomino Espresso, Upstate, Traif, Graham Avenue Meats, and Waffle & Wolf.

```
4 //  
5 // Created by Simon Ng on 4/12/2019.  
6 // Copyright © 2019 AppCoda. All rights reserved.  
7 //  
8  
9 import SwiftUI  
10  
11 struct ContentView: View {  
12     var body: some View {  
13         NavigationView {  
14             List {  
15                 ForEach(restaurants) { restaurant in  
16                     BasicImageRow(restaurant: restaurant)  
17                 }  
18             }  
19             .navigationBarTitle("Restaurants")  
20         }  
21     }  
22 }  
23  
24 struct ContentView_Previews: PreviewProvider {  
25     static var previews: some View {  
26         ContentView()  
27     }  
28 }  
29  
30 struct BasicImageRow: View {  
31     var restaurant: Restaurant  
32  
33     var body: some View {  
34         HStack {  
35             Image(restaurant.image)  
36                 .resizable()  
37                 .frame(width: 40, height: 40)  
38                 .cornerRadius(5)  
39             Text(restaurant.name)  
40         }  
41     }  
42 }  
43 }
```

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 actually contains 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.

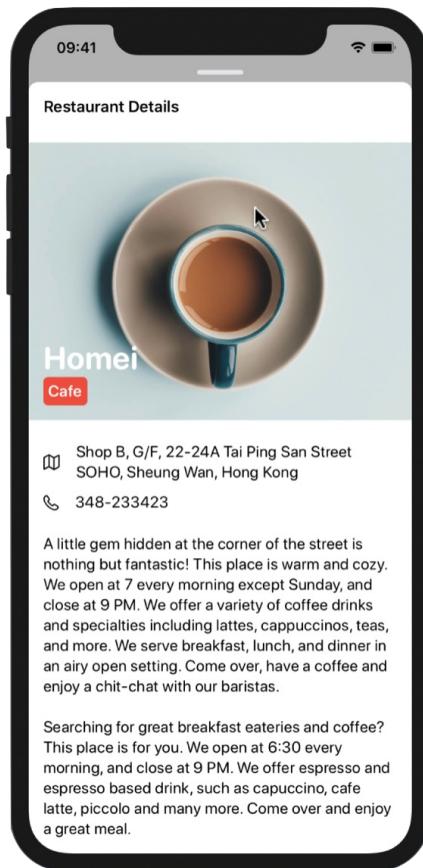


Figure 4. The restaurant detail view with a small handlebar

Before you follow me to implement the view, I suggest you to 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, it's the handlebar. It is actually a small rectangle with rounded corners. To create it, all we need to do is to create a `Rectangle` and make 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, it's 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 it the content mode to `scaledToFill`. The height of the image is fixed at 300 points. Since we use the `scaledToFill` 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, it comes to 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 can 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 can 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 just 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 the detail view to have a white background, we attach the `background` modifier and make the change.

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.

The screenshot shows the Swift code for `RestaurantDetailView` and its preview. The code defines a `RestaurantDetailView` struct that contains a `body` view which includes a `HandleBar`, `TitleBar`, and `HeaderView`. It also includes `DetailInfoView`s for location, phone, and description. The `body` has a white background and rounded corners. A `RestaurantDetailView_Previews` struct provides a preview provider for the first restaurant in the `restaurants` array. The preview shows a smartphone displaying the app's interface. The header says "Restaurant Details". Below it is a circular image of a latte art coffee cup. The main content area displays the restaurant's name "Cafe Deadend" and category "Coffee & Tea Shop". It shows the address "G/F, 72 Po Hing Fong, Sheung Wan, Hong Kong" and phone number "232-923423". A descriptive text block follows, mentioning the place is a little gem hidden at the corner of the street, open at 7 every morning except Sunday, and serves breakfast, lunch, and dinner in an airy open setting. At the bottom of the preview screen, there are three circular icons: a magnifying glass, a double arrow, and a square.

```
8 import SwiftUI
9
10 struct RestaurantDetailView: View {
11     let restaurant: Restaurant
12
13     var body: some View {
14         VStack {
15             Spacer()
16
17             HandleBar()
18
19             TitleBar()
20
21             HeaderView(restaurant: restaurant)
22
23             DetailInfoView(icon: "map", info: self.restaurant.location)
24                 .padding(.top)
25             DetailInfoView(icon: "phone", info: self.restaurant.phone)
26             DetailInfoView(icon: nil, info: self.restaurant.description)
27                 .padding(.top)
28         }
29         .background(Color.white)
30         .cornerRadius(10, antialiased: true)
31     }
32 }
33
34 struct RestaurantDetailView_Previews: PreviewProvider {
35     static var previews: some View {
36         RestaurantDetailView(restaurant: restaurants[0])
37     }
38 }
39
40 }
41
42 struct HandleBar: View {
43
44     var body: some View {
45         Rectangle()
46             .frame(width: 50, height: 5)
47     }
48 }
```

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 overlayed on top of the original content but usually 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.

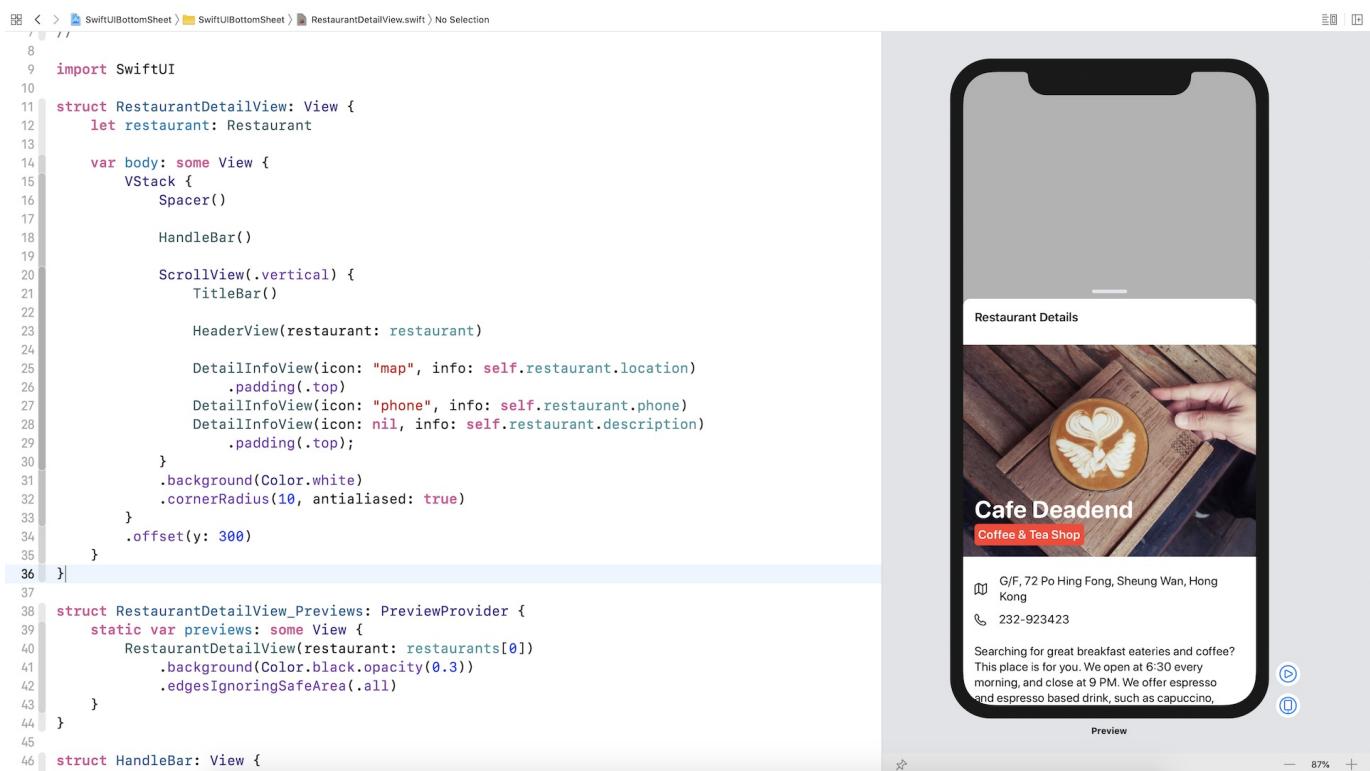


Figure 6. Adjusting the offset of the detail view

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.

What I want to do is that the offset value should be set to half of the screen height. So, how can you find out the screen size of a device? In SwiftUI, it 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.

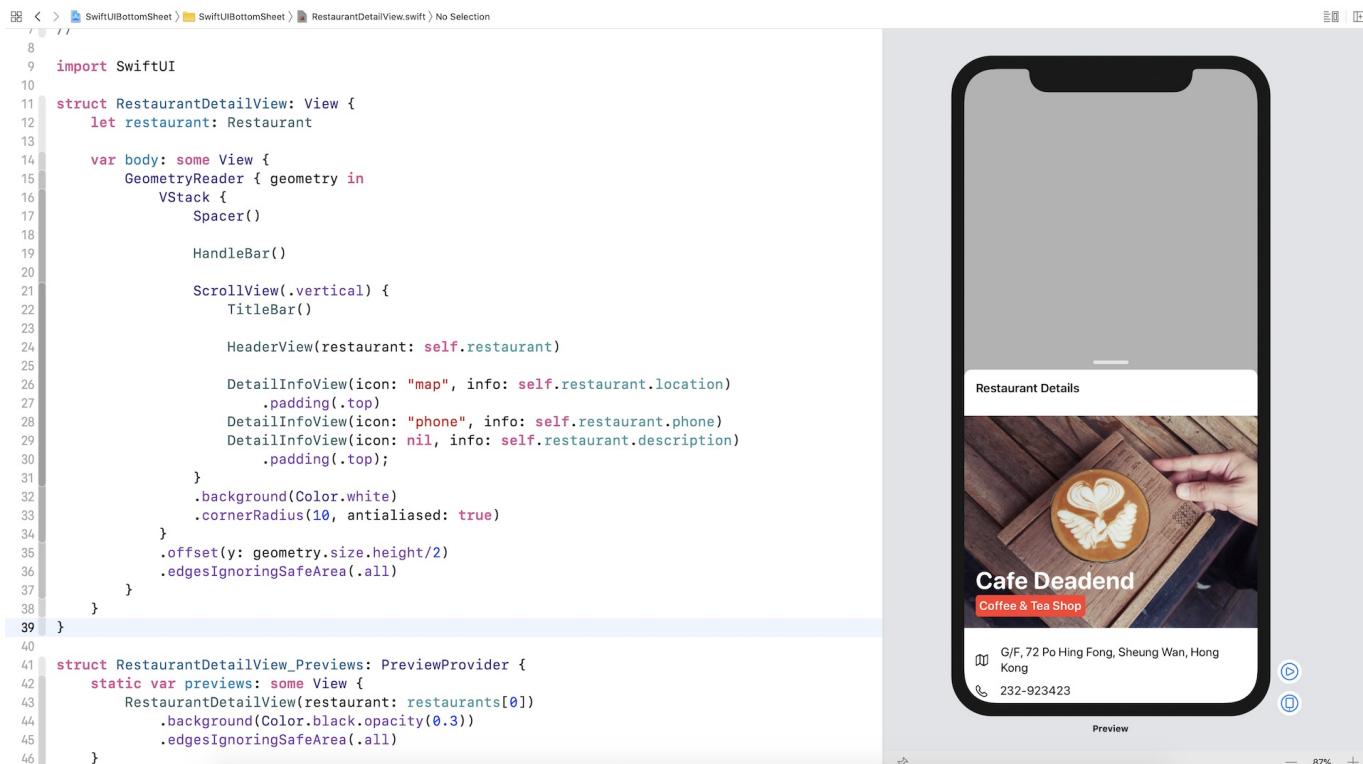


Figure 7. Adjusting the offset of the detail view

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:

```
@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 the earlier chapter, you can attach the `onTapGesture` modifier to detect the tap gesture. So, when a tap is recognized, we can 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 using a `zStack`. And 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 {
        selectedRestaurant.map {
            RestaurantDetailView(restaurant: $0)
                .transition(.move(edge: .bottom))
        }
    }
}

```

We also attach the `transition` modifier to the detail view such that it uses the `move` transition type. If you have some experience with Swift, you may know what the `map` function is for. But let me further explain how it works in case you are new to Swift.

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. The code can actually be written like this:

```
if selectedRestaurant != nil {
    RestaurantDetailView(restaurant: selectedRestaurant!)
        .transition(.move(edge: .bottom))
}
```

The map version, which is displayed below, works exactly the same as the code snippet above. However, it looks more elegant, right? This is why I use `map` to work with optionals in SwiftUI.

```
selectedRestaurant.map {
    RestaurantDetailView(restaurant: $0)
        .transition(.move(edge: .bottom))

}
```

Now that if you run the app in the preview canvas, it can already bring up the detail view when you select a restaurant. That said, 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 can 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: .infinity)
        .background(bgColor)
        .edgesIgnoringSafeArea(.all)
    }
}
```

Next, update the `if` clause like this:

```
if showDetail {

    BlankView(bgColor: .black)
        .opacity(0.5)
        .onTapGesture {
            self.showDetail = false
        }

    selectedRestaurant.map {
        RestaurantDetailView(restaurant: $0)
            .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.

```

17 ZStack {
18     NavigationView {
19         List {
20            ForEach(restaurants) { restaurant in
21                 BasicImageRow(restaurant: restaurant)
22                     .onTapGesture {
23                         self.showDetail = true
24                         self.selectedRestaurant = restaurant
25                     }
26             }
27         }
28     }
29     .navigationBarTitle("Restaurants")
30 }
31
32 if showDetail {
33     BlankView(bgColor: .black)
34         .opacity(0.5)
35         .onTapGesture {
36             self.showDetail = false
37         }
38
39 selectedRestaurant.map {
40     RestaurantDetailView(restaurant: $0)
41         .transition(.move(edge: .bottom))
42 }
43
44 }
45 }
46 }
47 }
48 }
49 struct ContentView_Previews: PreviewProvider {
50     static var previews: some View {
51         ContentView()
52     }
53 }
54 }
55 
```

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

It's a little bit closer to the final product, but there are a few things we still need to take care of. Do you aware that the transition of the detail view was not animated? While we have the `.transition` modifier, the transition will only be animated until 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.

Besides, 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.

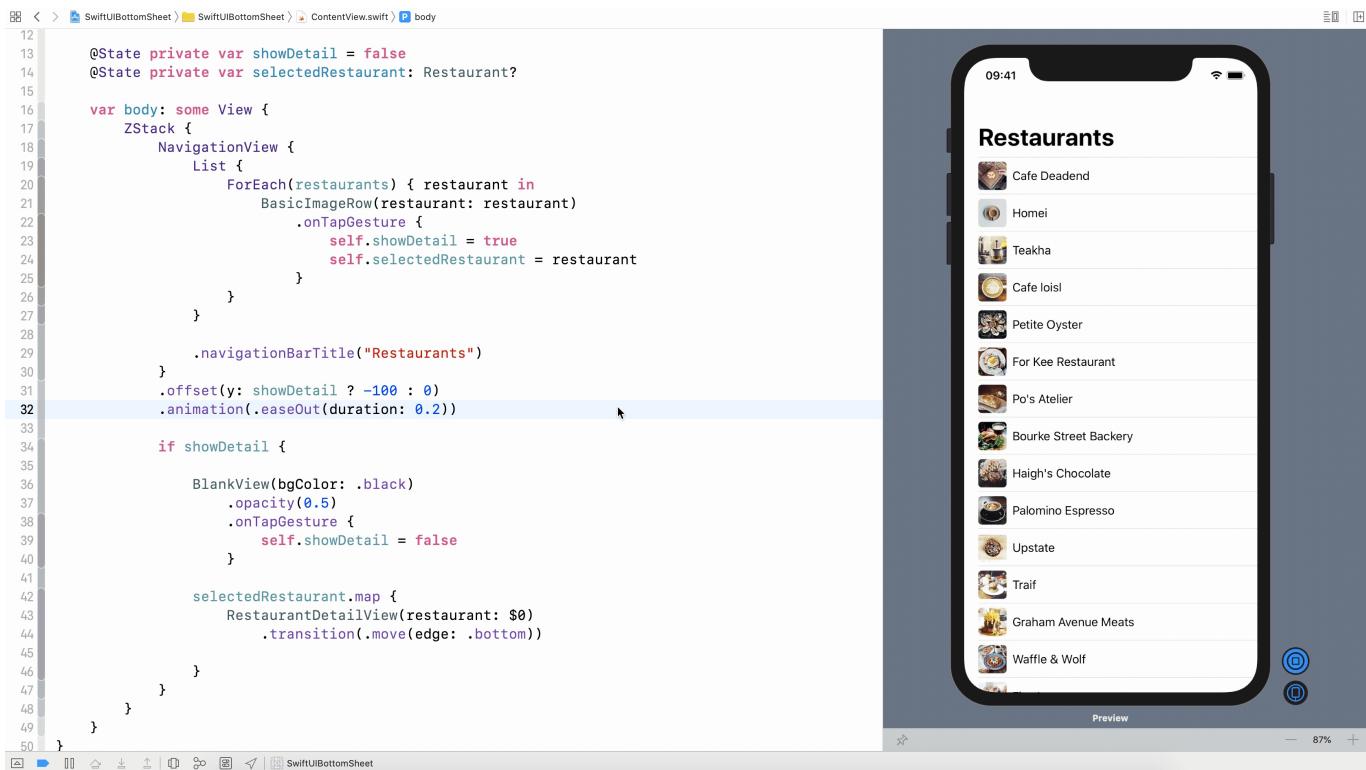


Figure 9. Adding animations to the bottom sheet

The animation looks pretty nice but I guess you may feel the animation is a little bit bouncy. This is a characteristic of spring-based animation, that the animated object often moves beyond the target position and then bounces back. And, if you look at the

.`animation` modifier in `RestaurantDetailView.swift`, it's attached to the `vStack`, which contains the scroll view. This is why you see the title bar goes a little bit off the screen and bounces back.

To avoid the scroll view from bouncing, we can remove the animated effect by attaching a nil `.animation` modifier to it. Open `RestaurantDetailView.swift` and add the `.animation` modifier to `scrollView` like this:

```
ScrollView(.vertical) {  
    .  
    .  
    .  
}  
.animation(nil)  
.background(Color.white)  
.cornerRadius(10, antialiased: true)
```

By setting the modifier's value to nil, SwiftUI will not render the animation for the scroll view. Now run the app again and see how the change performs.

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 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. That said, 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.positionOffset)
```

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. However, it's nearly impossible to slide up/down the view by dragging the content view.

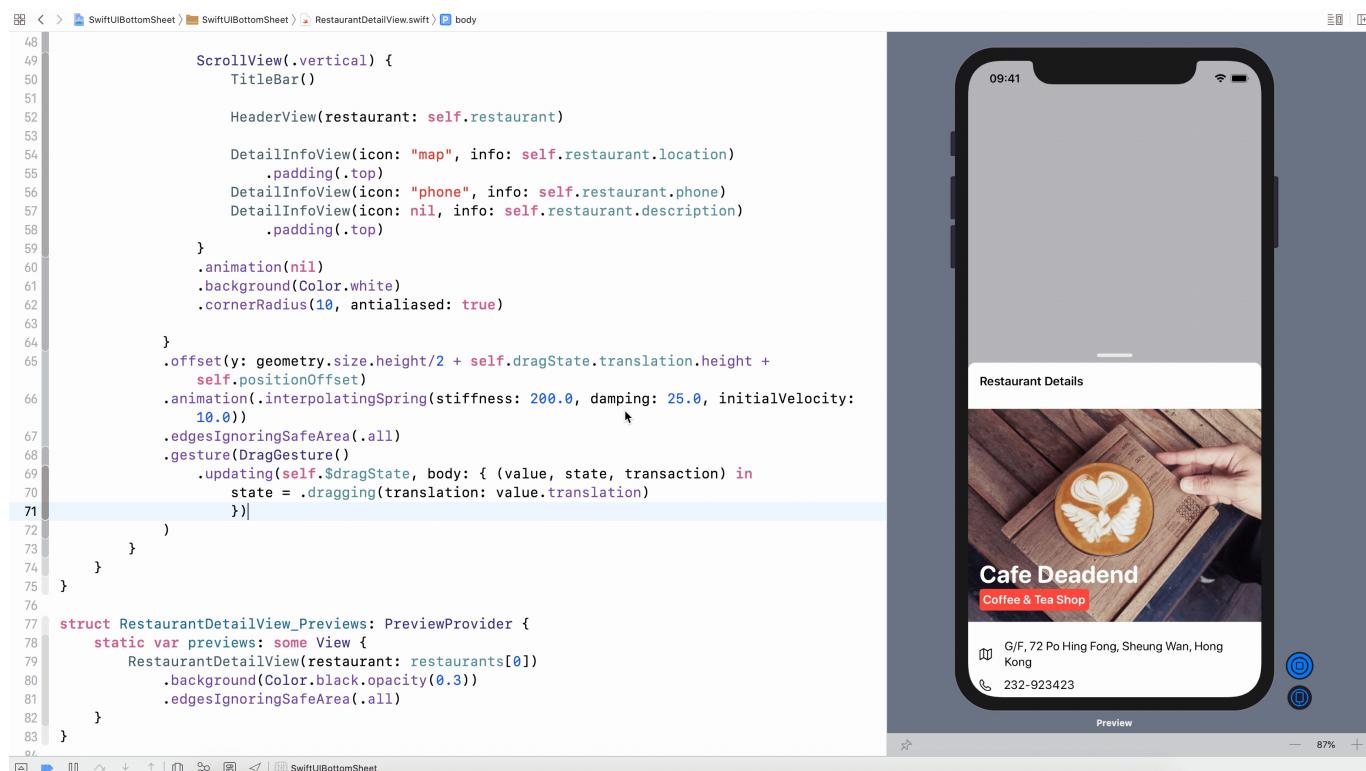


Figure 10. Dragging the content part of the detail view won't slide up the view

Why? Why couldn't we drag the content view to expand the detail view?

If you're not forgetful, 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 can 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 or while the user is dragging it. After the detail view is fully opened, we enable the scrolling again.

The other problem of the current implementation is that the detail view can't stay fully open even 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. Conversely, you can't dismiss the detail view even you slide it down to the end of the screen.

How do we tackle these problems?

Let's consider how the users are going to interact with the detail view:

When the detail view is brought up, it's half opened:

1. In this case, the user can choose to slide up the view to fully open it.
 - However, the user may move the view a bit upward and then drag it down to cancel the action.
2. Alternatively, the user can slide the view down to dismiss it.

- Again, the user may drag it back to up to cancel the dismiss.

When the detail view is fully opened:

1. In this case, the user can drag the view down a bit to revert it to the half open state.
2. Or the user can drag the view all the way down to dismiss it.
3. Again, for both cases, the user can cancel the drag action.

As you can reveal from these scenarios, other than keeping track of the drag offset, we need some kinds of threshold to control the opening and dismissal of the view.

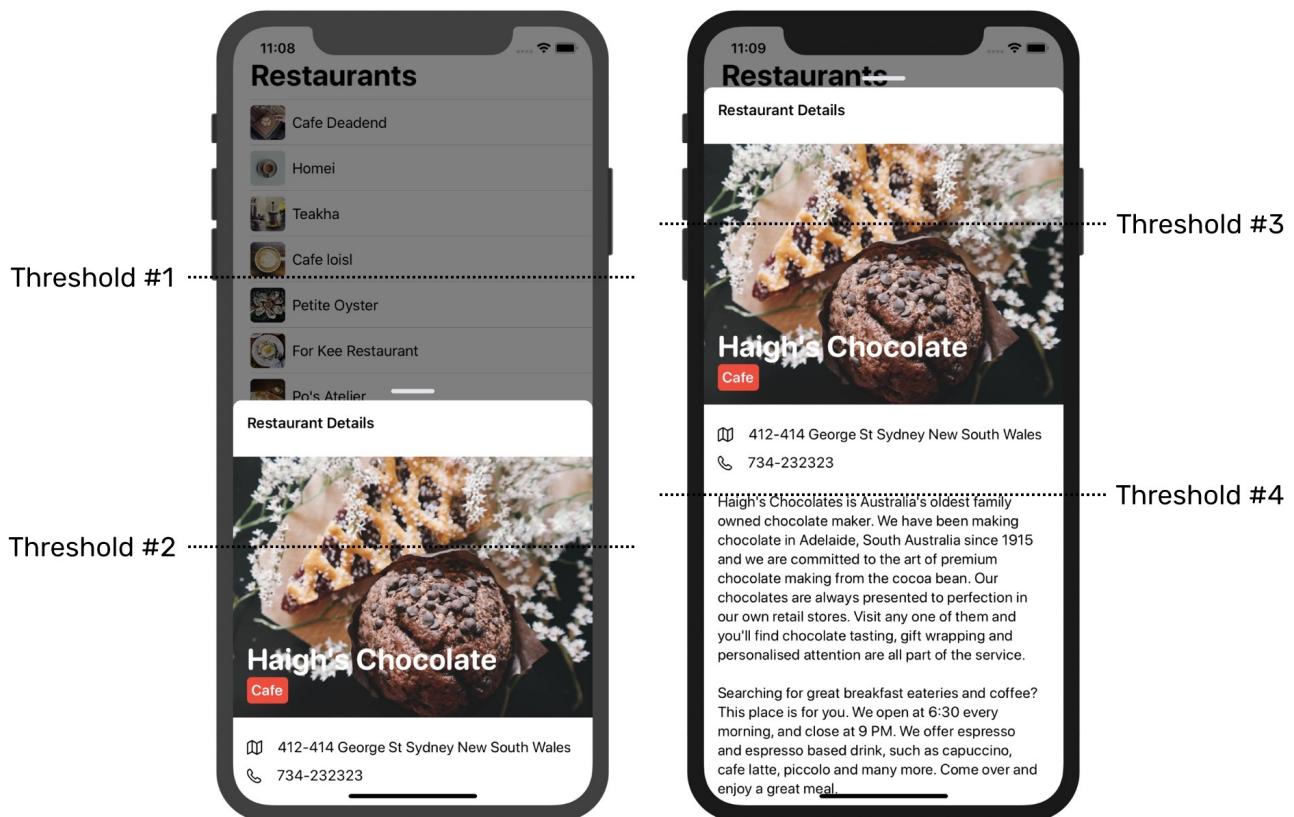


Figure 11. Adding thresholds to control the dragging and view state

The figure above shows you the thresholds we are going to define:

Half open 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.

Fully opened state

- **Threshold #3** - when the user slides down the view and the drag passes this threshold, the view will revert to half open state.
- **Threshold #4** - when the user slides the view all the way down and goes pass this threshold, the view will be dismissed.

Now that you should know how the half open and fully opened state 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 viewType = ViewState.half
```

Also, in order to dismiss the view itself, we need to 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 or while the user is dragging it. After the detail view is fully opened, we enable the scrolling again.

Next, let's fix the scrolling issue of the scroll view. Attach the `.disabled` modifier to the `ScrollView` like this:

```
.disabled(self.viewState == .half || self.dragState.isDragging)
```

Here, we disable the user interaction of the scroll view when the detail view is in half open state and while someone is dragging it.

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

        switch self.viewState {
        case .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
    }

    case .full:
        // Threshold #3
        // Slide down and when it goes pass the threshold
        // return to the half open state
        if value.translation.height > geometry.size.height * 0.25 {
            self.positionOffset = 0
            self.viewState = .half
        }

        // Threshold #4
        // Slide down further and when it goes pass the threshold
        // dismiss the view
        if value.translation.height > geometry.size.height * 0.75 {
            self.isShow = false
        }
    }

})
)

```

I compute the threshold by using the screen height. For instance, threshold #3 is set to one-fourth of the screen height. This is just a sample value. You may alter it to fit your need.

Take a look at the code comment 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` :

```
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:

```
selectedRestaurant.map {
    RestaurantDetailView(restaurant: $0, isShow: $showDetail)
        .transition(.move(edge: .bottom))

}
```

After all the hard work, it's time to test out the expandable detail view. Run the app in the simulator or preview canvas, the drag gesture should now work as expected.

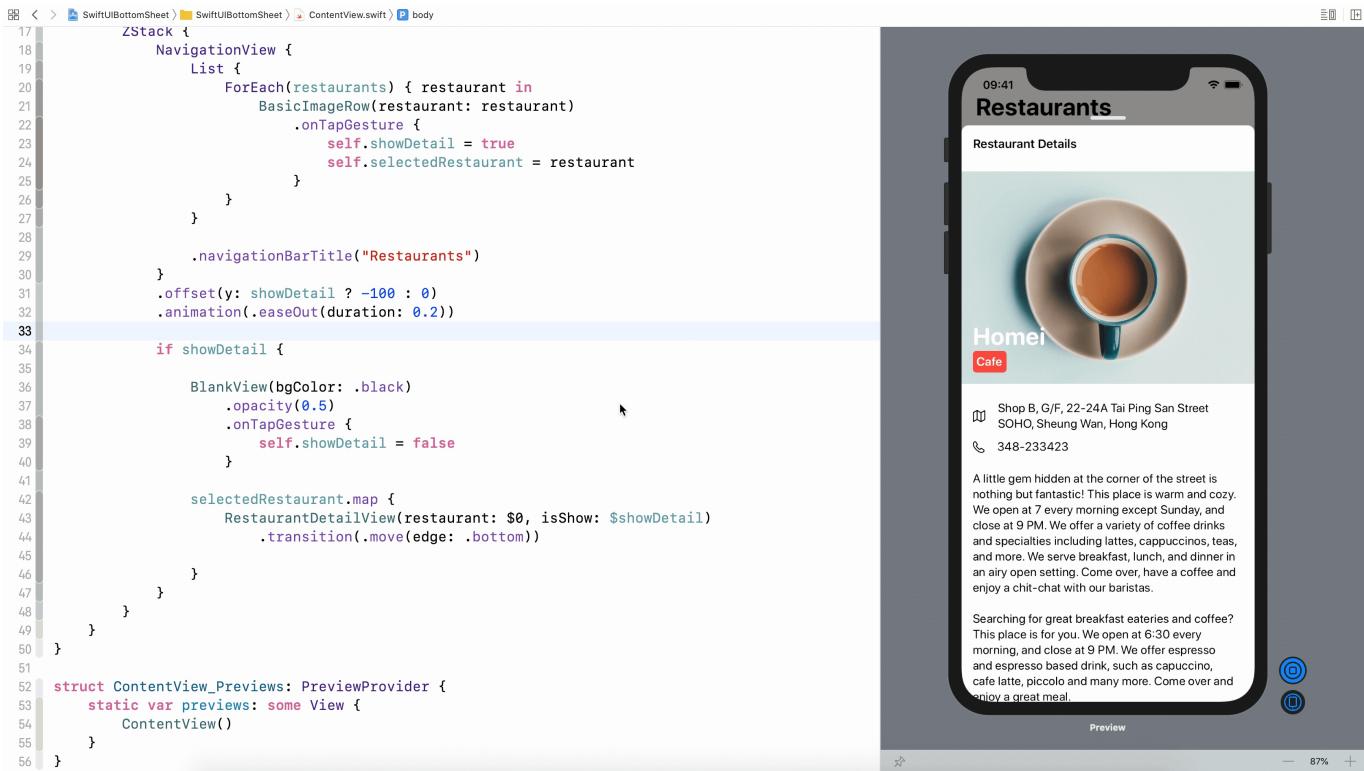
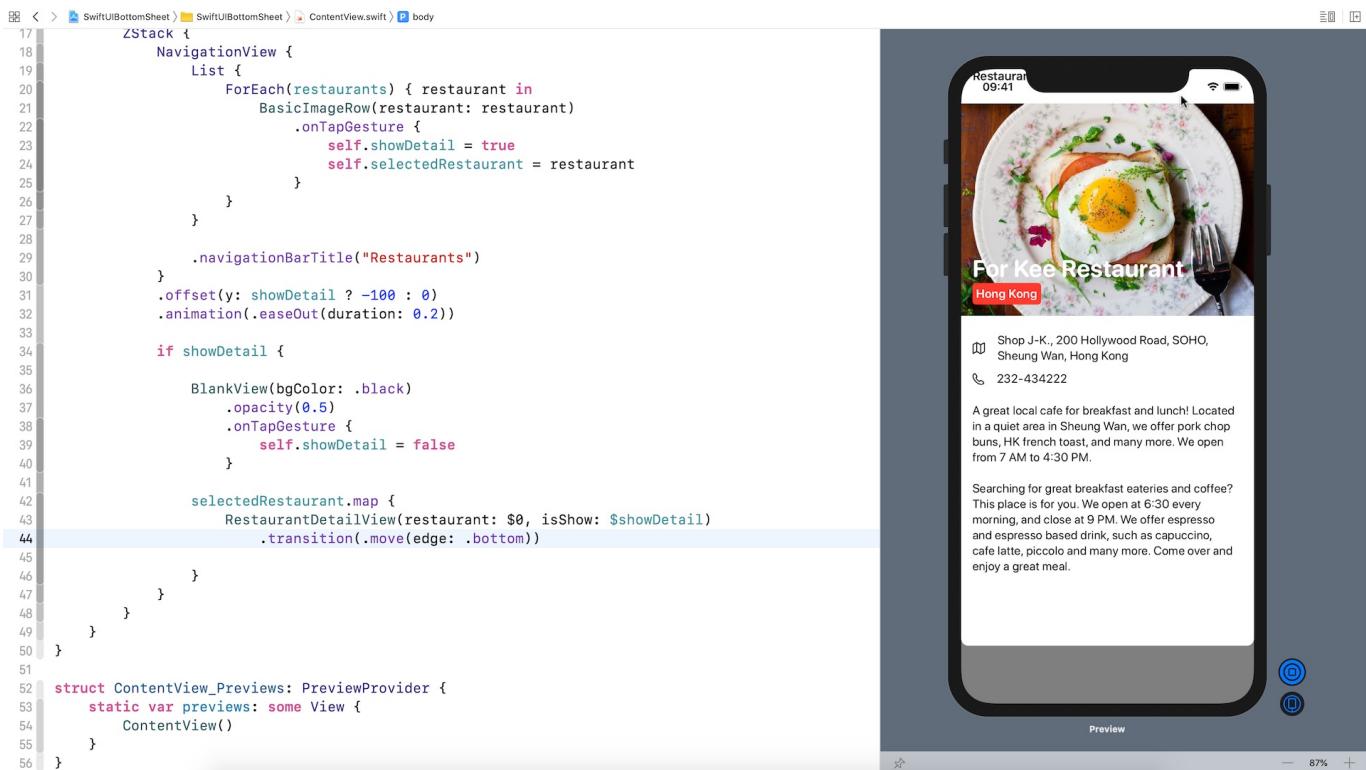


Figure 12. Reverting the detail view from the fully open state to the half open state

Exercise

When the detail view is in fully opened state, the app should not allow users to slide the view up. However, this doesn't work as expected. Your task is to fix this bug.



The screenshot shows the Xcode interface with the code editor on the left and a preview window on the right. The code is for a SwiftUI bottom sheet component. It uses a ZStack to layer a NavigationView over a list of restaurants. A tap gesture on the list item triggers a transition to a detailed view for the selected restaurant. The preview shows a smartphone displaying a restaurant's details, including a photo of a dish, the restaurant's name, address, phone number, and a description.

```

17 ZStack {
18     NavigationView {
19         List {
20             ForEach(restaurants) { restaurant in
21                 BasicImageRow(restaurant: restaurant)
22                     .onTapGesture {
23                         self.showDetail = true
24                         self.selectedRestaurant = restaurant
25                     }
26             }
27         }
28     }
29     .navigationBarTitle("Restaurants")
30 }
31 .offset(y: showDetail ? -100 : 0)
32 .animation(.easeOut(duration: 0.2))
33
34 if showDetail {
35
36     BlankView(bgColor: .black)
37         .opacity(0.5)
38         .onTapGesture {
39             self.showDetail = false
40         }
41
42     selectedRestaurant.map {
43         RestaurantDetailView(restaurant: $0, isShow: $showDetail)
44             .transition(.move(edge: .bottom))
45
46     }
47 }
48 }
49 }
50 }
51 struct ContentView_Previews: PreviewProvider {
52     static var previews: some View {
53         ContentView()
54     }
55 }
56

```

Figure 13. Sliding up the view when it's in fully opened state

Summary

This is a huge chapter but I hope you enjoy it. In this chapter, we utilize everything you learned so far to build a expandable bottom sheet. I tried my best to document my thought process on tackling the issues when I built the sheet. I really hope this would help 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 reveal, it is pretty easy to turn this restaurant detail view into a generic bottom sheet that supports varies types of content by using the technique that I covered in the previous chapter.

So, try to build the generic bottom sheet by yourself and make it as a sharable component for your projects.

For reference, you can download the complete project here:

- Demo project
(<https://www.appcoda.com/resources/swiftui/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 and apply it to a real-world project. I'm not sure if you've used the Tinder app before. But somehow you probably have come across a Tinder-like user interface in some 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.

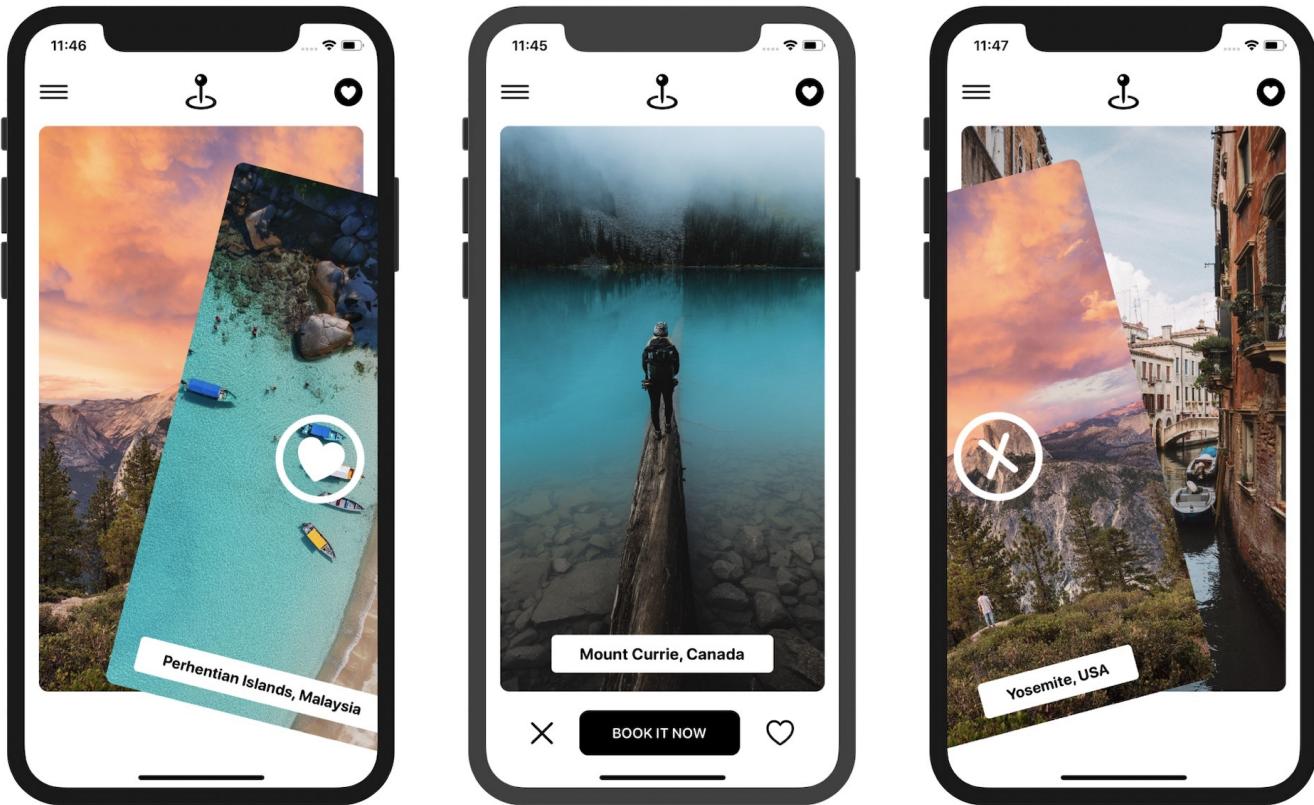


Figure 1. Building a tinder-like user interface

Note that we are not going to build the fully functional app but focus only on the Tinder-like UI.

Project Preparation

It would be great if you use your own image. But to save you time from preparing the trip image, I have created a starter project for you. You can download it from <https://www.appcoda.com/resources/swiftui/SwiftUITripStarter.zip>. This project already comes with a set of photos for the travel cards.

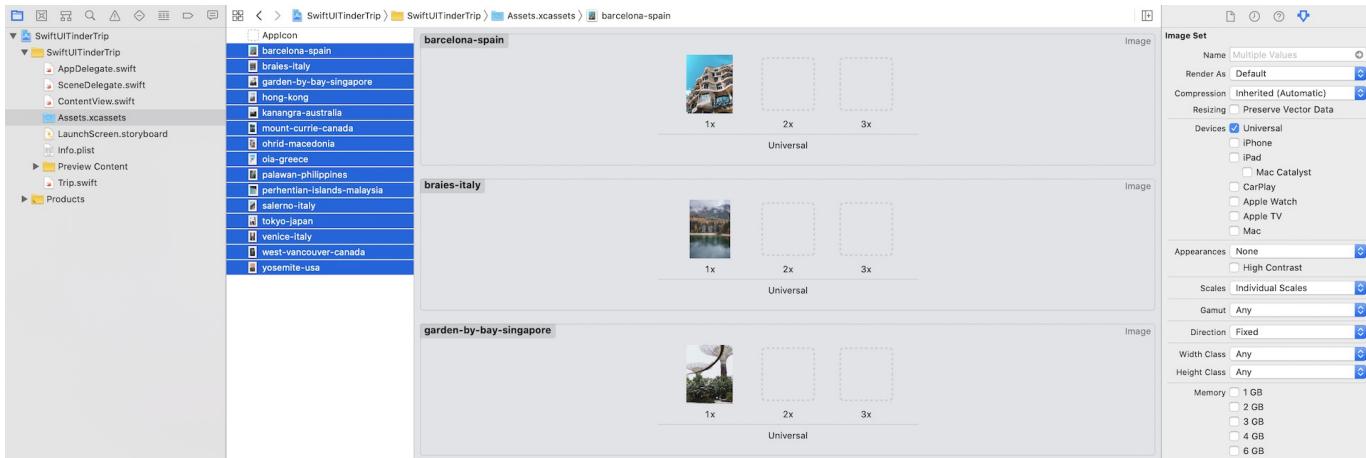


Figure 2. Preloaded with a set of travel photos

On top of that, I have 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

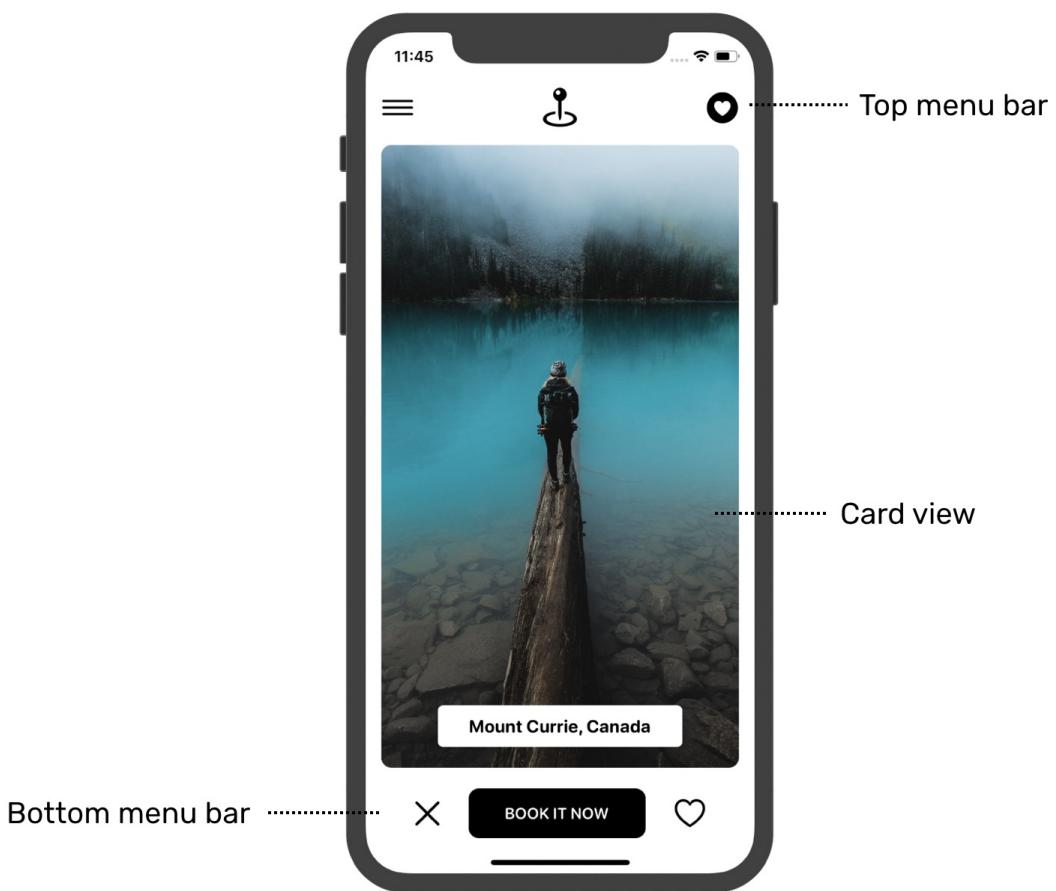


Figure 3. The main screen

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` conformable 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 forget 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 `scaledToFill` 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.

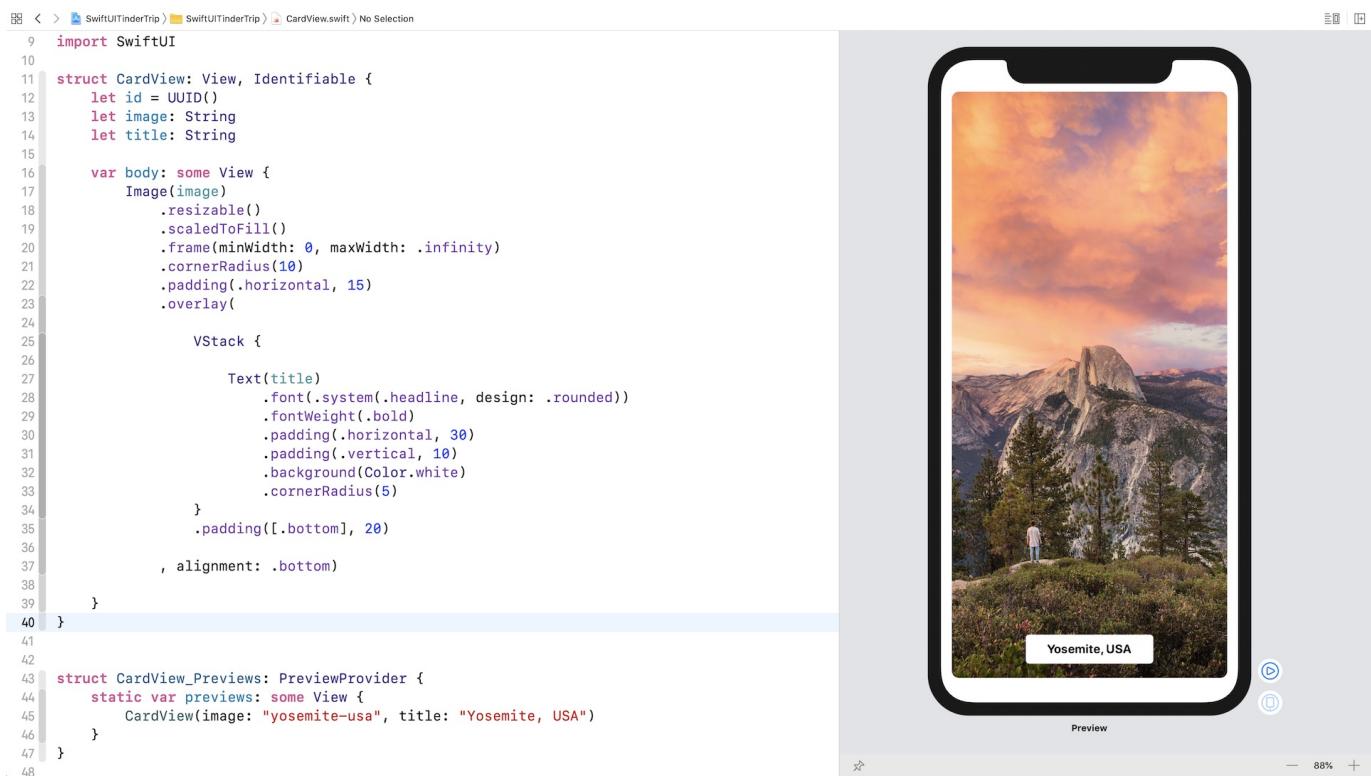


Figure 4. Previewing the card view

Menu Bars and Main UI

With the card view ready, we can move on to implementing the main UI. As said before, 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 out blank. The rest of the code should be self explanatory assuming you understand how stacks and image work.

Before building the main UI, let me show you a trick to preview these two menu bars. It's not a mandate 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 about what the preview looks like.



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 layout the main UI. Update the `ContentView` like this:

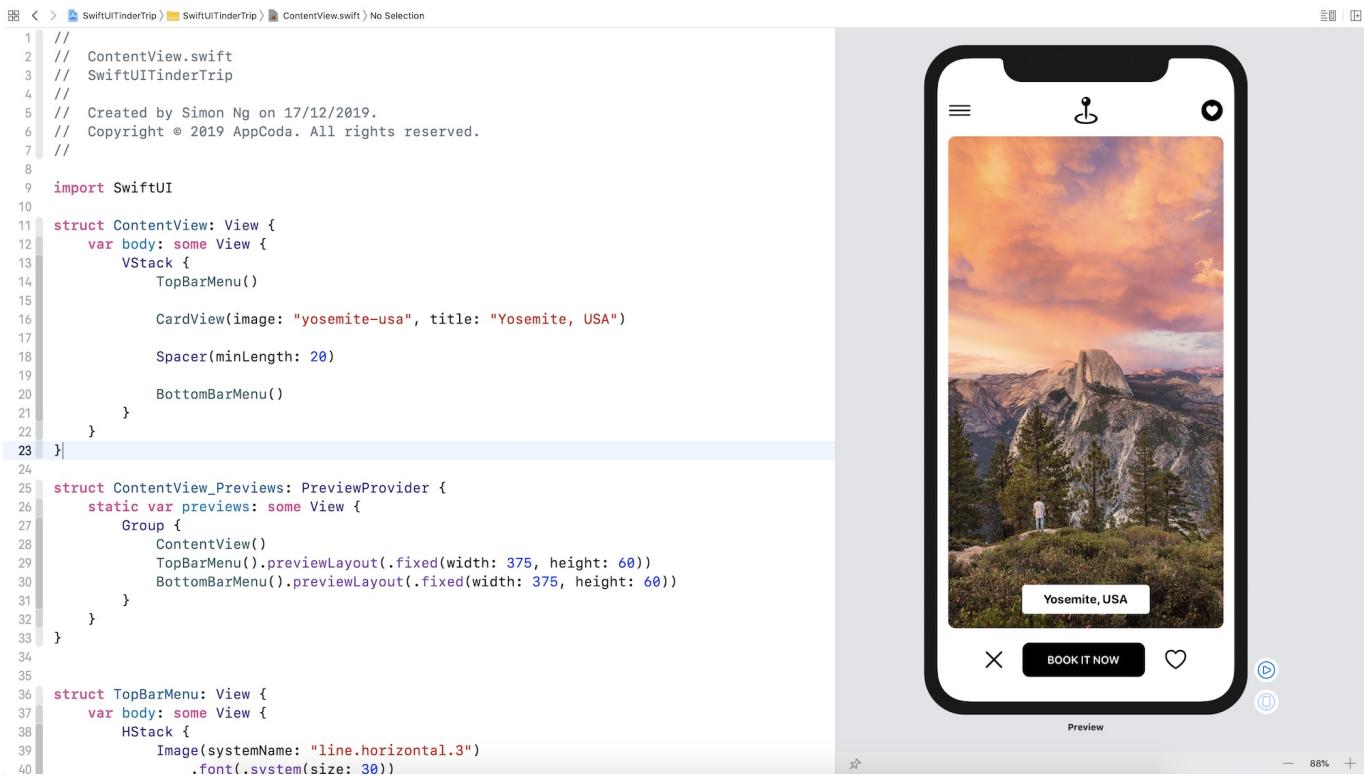
```
struct ContentView: View {
    var body: some View {
        VStack {
            TopBarMenu()

            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.



The screenshot shows the Xcode editor with the ContentView.swift file open. The code defines a ContentView struct with a body containing a VStack of components: TopBarMenu, CardView, a Spacer, and BottomBarMenu. It also includes a PreviewProvider section for generating previews. The preview itself shows a smartphone displaying a scenic view of Half Dome in Yosemite at sunset. The UI includes a top navigation bar with three icons, a large central image, a bottom bar with a 'BOOK IT NOW' button, and social sharing icons.

```

1 // ContentView.swift
2 // SwiftUITinderTrip
3 // Created by Simon Ng on 17/12/2019.
4 // Copyright © 2019 AppCoda. All rights reserved.
5
6 import SwiftUI
7
8 struct ContentView: View {
9     var body: some View {
10         VStack {
11             TopBarMenu()
12
13             CardView(image: "yosemite-usa", title: "Yosemite, USA")
14
15             Spacer(minLength: 20)
16
17             BottomBarMenu()
18         }
19     }
20 }
21
22 }
23
24 struct ContentView_Previews: PreviewProvider {
25     static var previews: some View {
26         Group {
27             ContentView()
28             TopBarMenu().previewLayout(.fixed(width: 375, height: 60))
29             BottomBarMenu().previewLayout(.fixed(width: 375, height: 60))
30         }
31     }
32 }
33
34
35 struct TopBarMenu: View {
36     var body: some View {
37         HStack {
38             Image(systemName: "line.horizontal.3")
39                 .font(.system(size: 30))
40

```

Figure 6. Previewing the main UI

Implementing the Card Deck

With all the preparation, it 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 cards that each card shows a photo. 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.

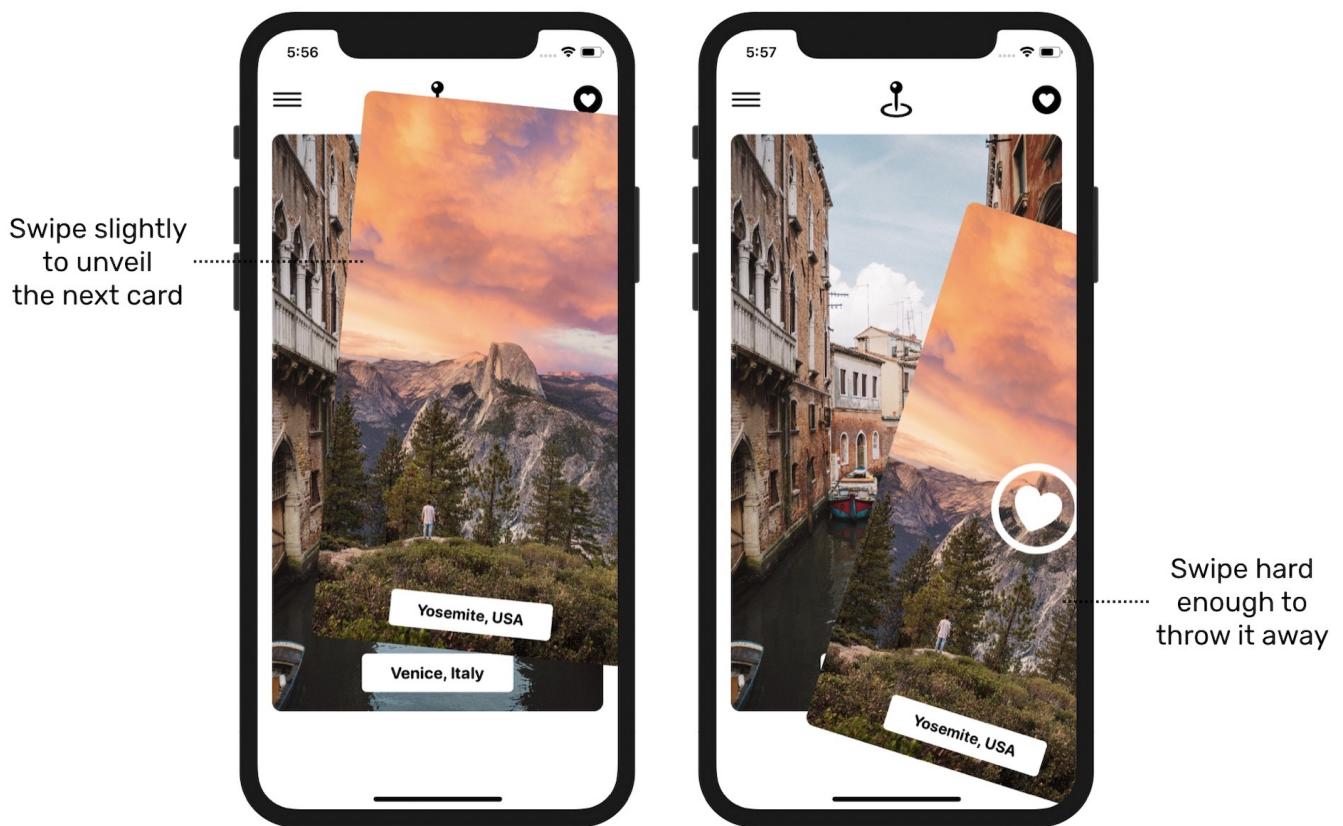


Figure 7. How the Tinder-like UI works

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()

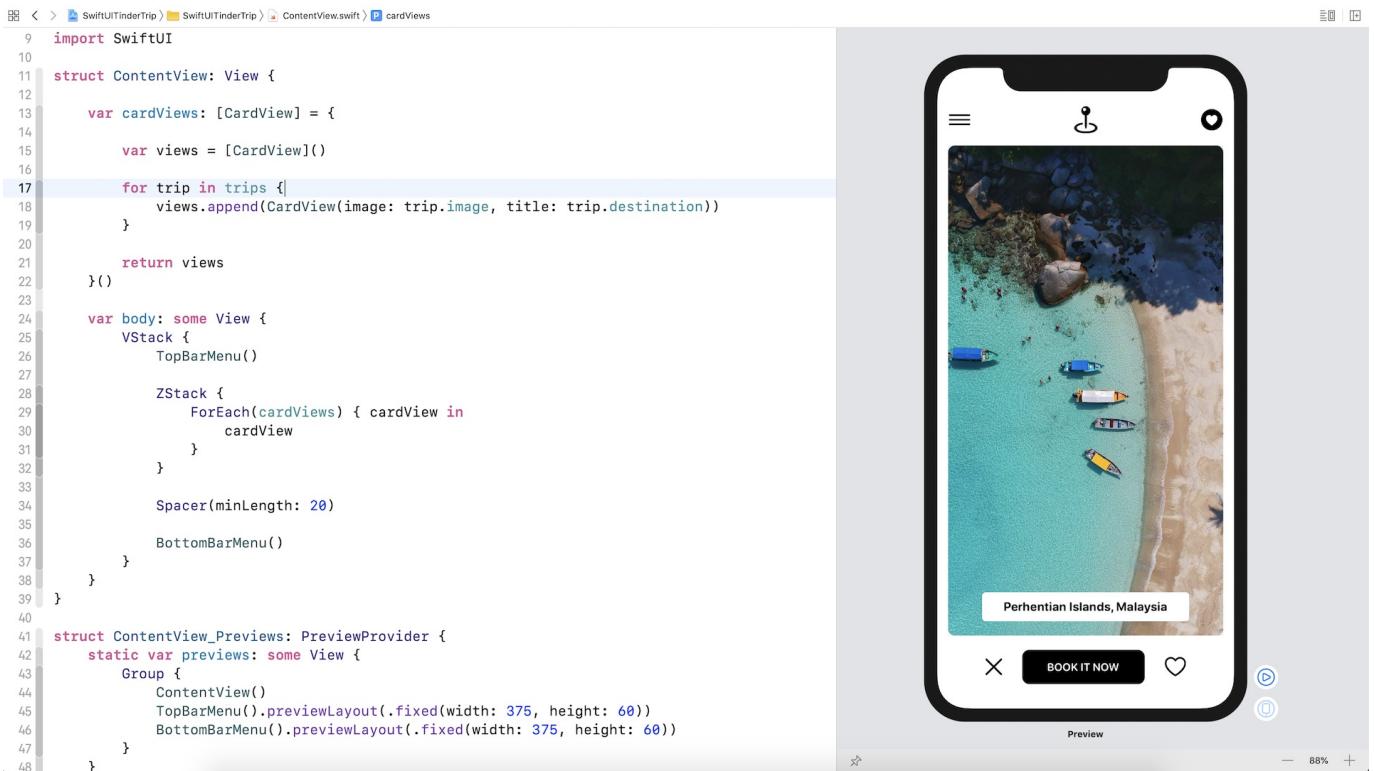
            ZStack {
                ForEach(cardViews) { cardView in
                    cardView
                }
            }

            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.



The screenshot shows the Xcode editor with the ContentView.swift file open. The code defines a Content View that contains a list of trips. Each trip is represented by a CardView, which is a ZStack with a background image and a title label. The preview window shows a smartphone displaying an aerial view of a beach with several small boats, labeled "Perhentian Islands, Malaysia". Below the image is a "BOOK IT NOW" button and other UI elements.

```
9 import SwiftUI
10
11 struct ContentView: View {
12
13     var cardViews: [CardView] = {
14
15         var views = [CardView]()
16
17         for trip in trips {
18             views.append(CardView(image: trip.image, title: trip.destination))
19         }
20
21         return views
22     }()
23
24     var body: some View {
25         VStack {
26             TopBarMenu()
27
28             ZStack {
29                 ForEach(cardViews) { cardView in
30                     cardView
31                 }
32             }
33
34             Spacer(minLength: 20)
35
36             BottomBarMenu()
37         }
38     }
39 }
40
41 struct ContentView_Previews: PreviewProvider {
42     static var previews: some View {
43         Group {
44             ContentView()
45             TopBarMenu().previewLayout(.fixed(width: 375, height: 60))
46             BottomBarMenu().previewLayout(.fixed(width: 375, height: 60))
47         }
48     }
}
```

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.



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.

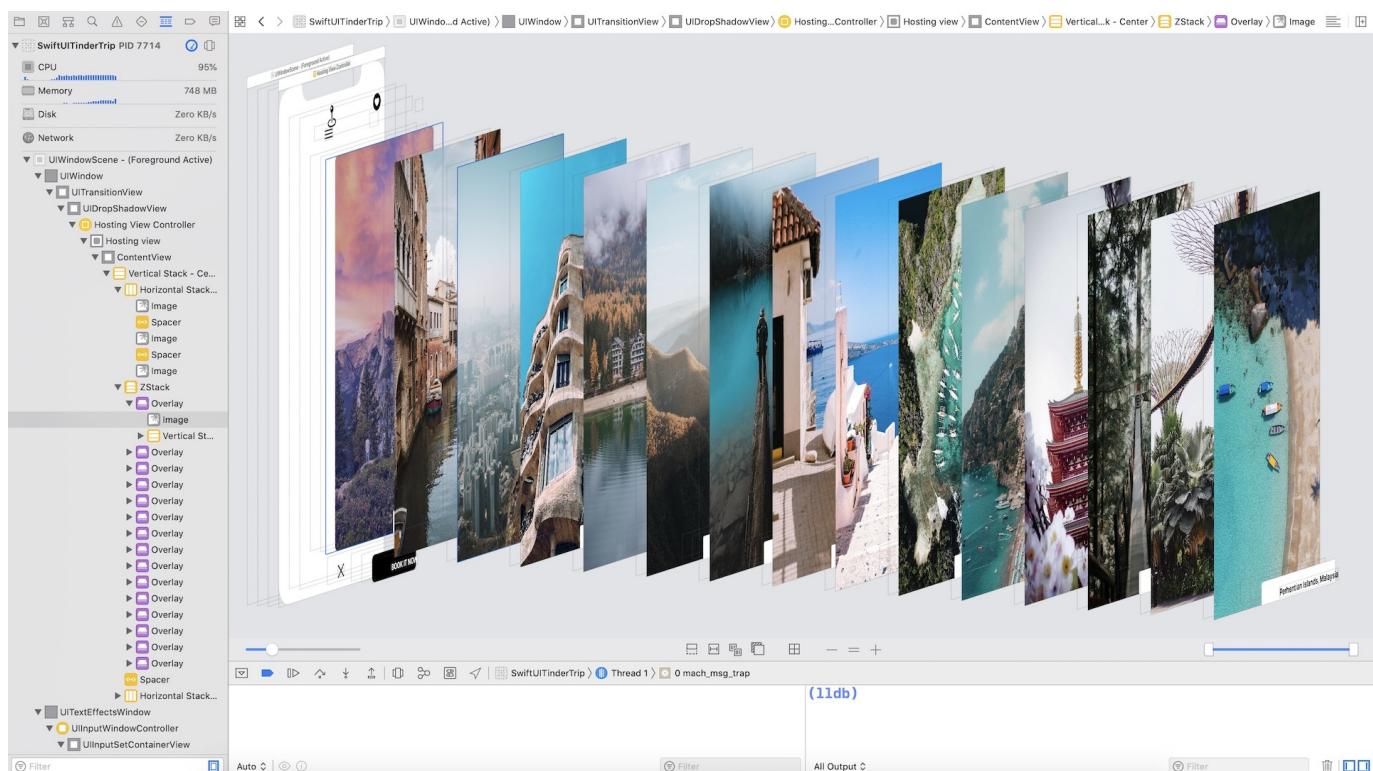


Figure 10. Click the Debug View Hierarchy button

While we implemented the card deck, we actually have two issues here:

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 first 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 tell 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. For the topmost card, we assign it with 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.

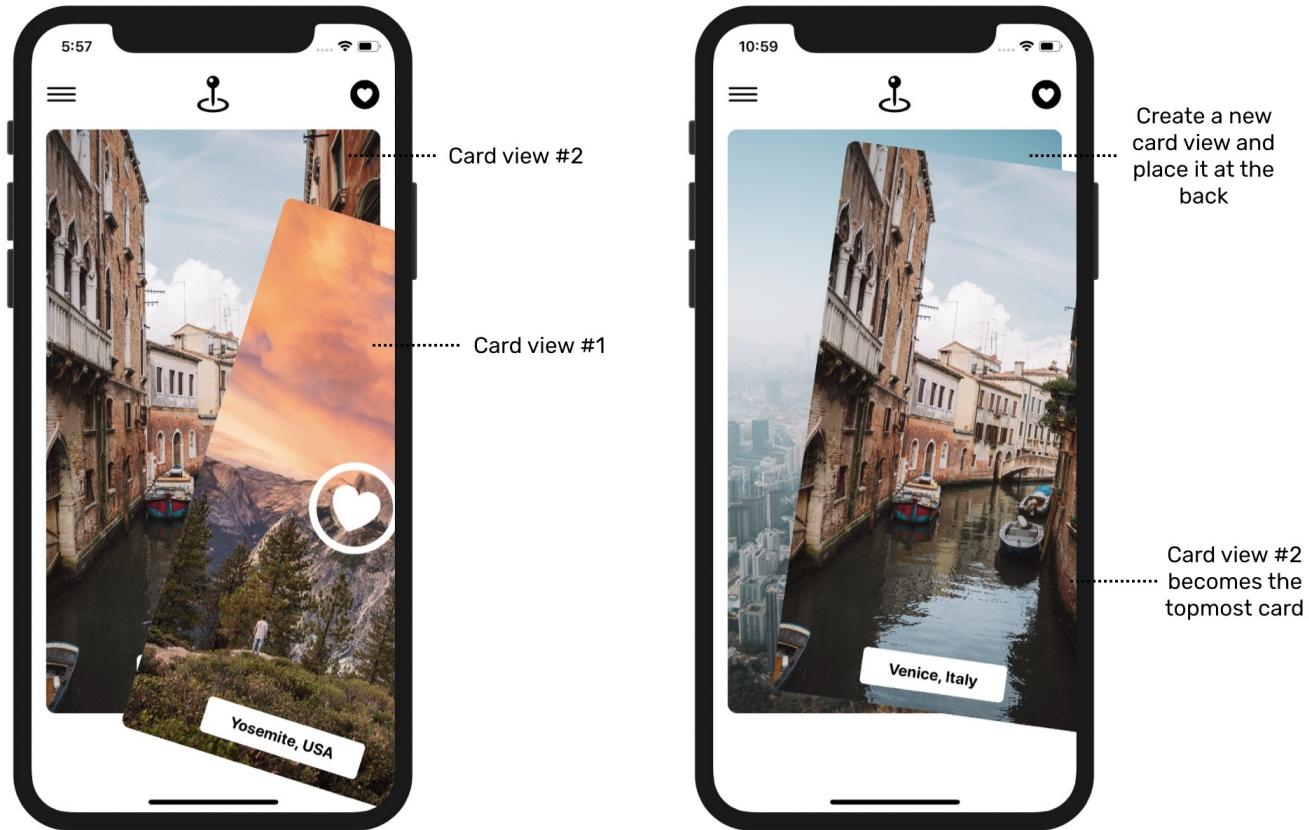


Figure 11. How we use two card views to create a deck

Now that you should 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].destination))
    }

    return views
}()


```

After the code change, the UI should look exactly the same. That said, you run it in a simulator and debug the view hierarchy. You should only see two card views in the deck.

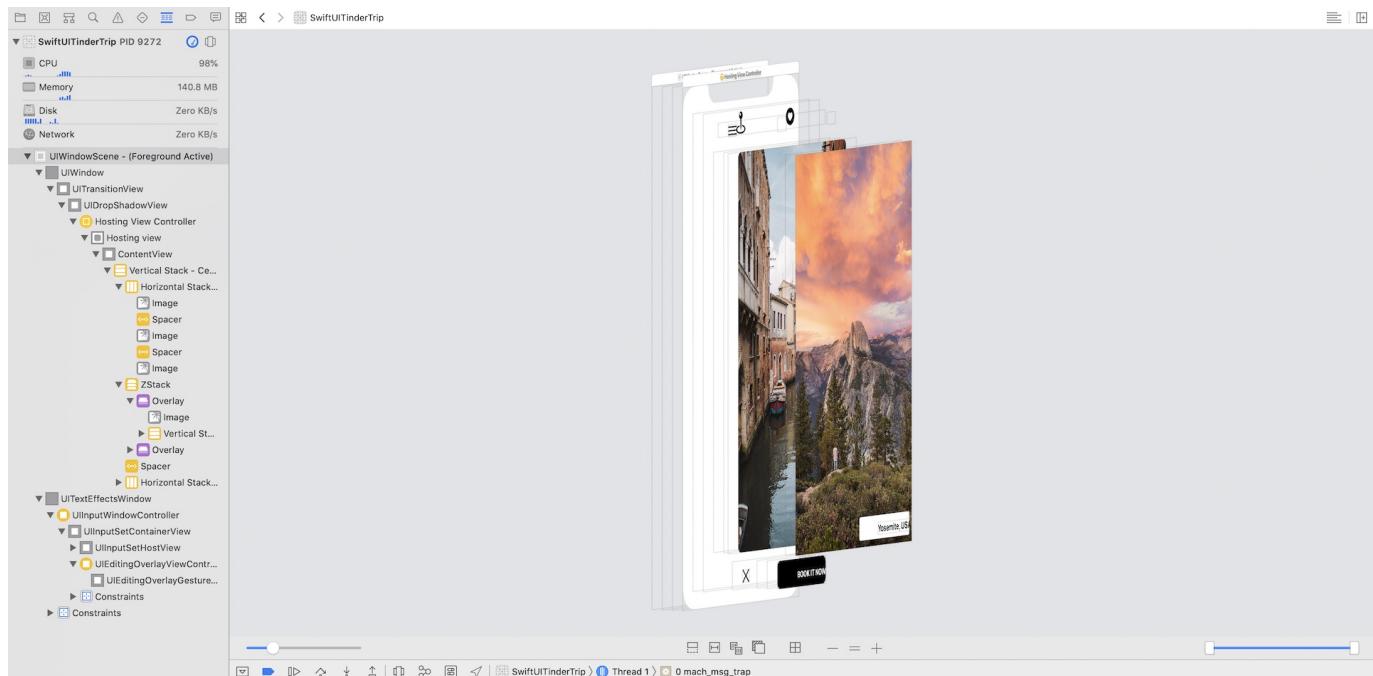


Figure 12. Using debug view hierarchy to view the card views

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 chapter 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 the enum is for, stop here and review the chapters about 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.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
                            }
                        })
                    )
            }
        }
    }
}
```

```
        }

        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. And 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 made 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 the original position.

The screenshot shows the Xcode interface with the code editor on the left and a simulator preview on the right. The code editor displays the `ContentView.swift` file with Swift code for a Tinder-like card deck. The simulator preview shows an iPhone displaying a landscape image of Half Dome in Yosemite at sunset. A white callout bubble in the bottom right corner of the image contains the text "Yosemite, USA". A cursor icon is visible on the screen, indicating interactivity. The Xcode toolbar at the top includes icons for back, forward, and search.

```
61 var body: some View {
62     VStack {
63         TopBarMenu()
64
65         ZStack {
66             ForEach(cardViews) { cardView in
67                 cardView
68                     .zIndex(self.isTopCard(cardView) ? 1 : 0)
69                     .offset(x: self.dragState.translation.width, y:
70                             self.dragState.translation.height)
71                     .scaleEffect(self.dragState.isDragging ? 0.95 : 1.0)
72                     .rotationEffect(Angle(degrees: Double(
73                         self.dragState.translation.width / 10)))
74                     .animation(.interpolatingSpring(stiffness: 180, damping: 100))
75                     .gesture(LongPressGesture(minimumDuration: 0.01)
76                         .sequenced(before: DragGesture())
77                         .updating(self.$dragState, body: { (value, state, transaction) in
78                             switch value {
79                                 case .first(true):
80                                     state = .pressing
81                                 case .second(true, let drag):
82                                     state = .dragging(translation: drag?.translation ?? .zero)
83                                 default:
84                                     break
85                             }
86                         })
87                     )
88                 }
89             }
90         }
91         Spacer(minLength: 20)
92
93         BottomBarMenu()
94             .opacity(self.dragState.isDragging ? 0.0 : 1.0)
95             .animation(.default)
96     }
97 }
```

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 that the user can 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.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
                    }
                })
            )
    }
}

```

Just focus the changes on 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.

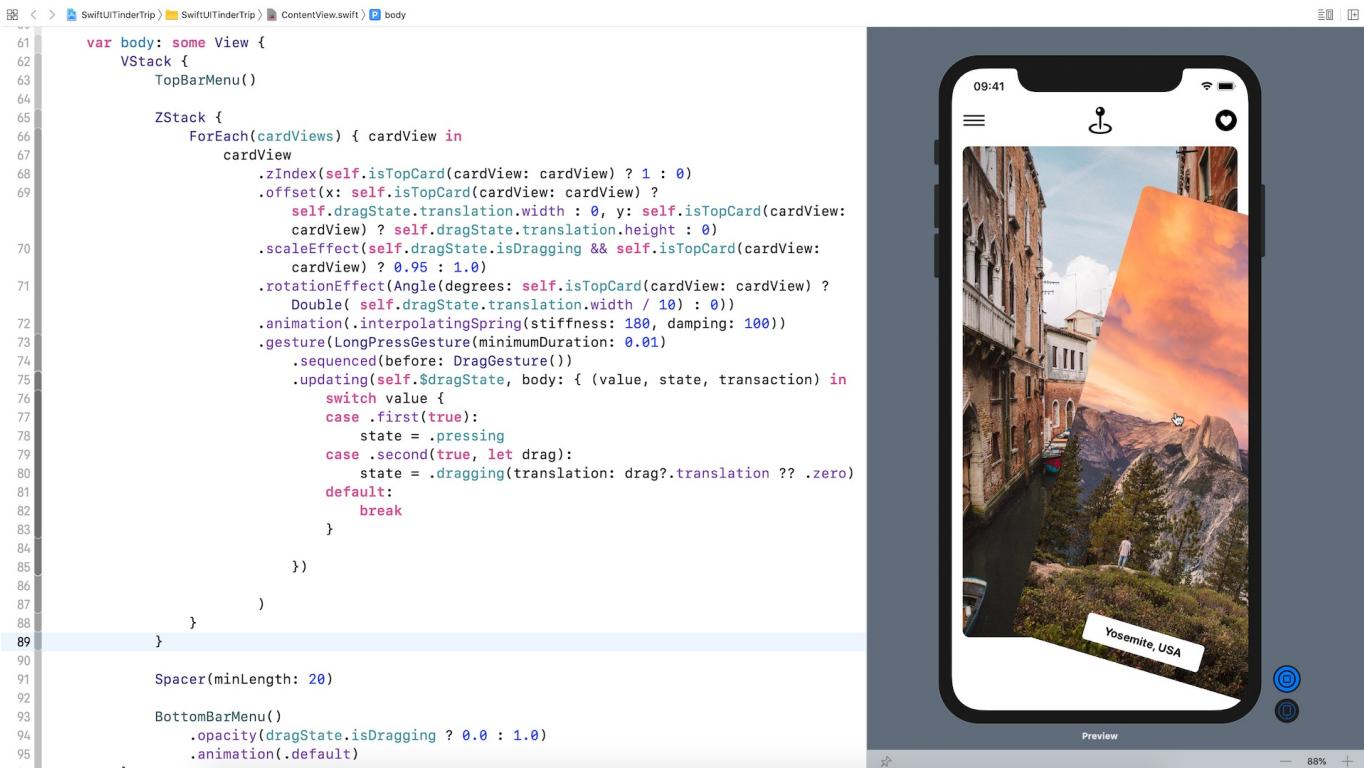


Figure 14. The dragging effect only applies to the topmost card

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.

```
57
58     private let dragThreshold: CGFloat = 80.0
59
60     @GestureState private var dragState = DragState.inactive
61
62
63     var body: some View {
64         VStack {
65             TopBarMenu()
66
67             ZStack {
68                 ForEach(cardViews) { cardView in
69                     cardView
70                         .zIndex(self.isTopCard(cardView: cardView) ? 1 : 0)
71                         .overlay(
72                             ZStack {
73                                 Image(systemName: "x.circle")
74                                     .foregroundColor(.white)
75                                     .font(.system(size: 100))
76                                     .opacity(self.dragState.translation.width <
77                                         -self.dragThreshold && self.isTopCard(cardView:
78                                         cardView) ? 1.0 : 0)
79
79                                 Image(systemName: "heart.circle")
80                                     .foregroundColor(.white)
81                                     .font(.system(size: 100))
82                                     .opacity(self.dragState.translation.width >
83                                         self.dragThreshold && self.isTopCard(cardView:
84                                         cardView) ? 1.0 : 0.0)
85
86                             }
87                         )
88                         .offset(x: self.isTopCard(cardView: cardView) ?
89                             self.dragState.translation.width : 0, y: self.isTopCard(cardView:
90                             cardView) ? self.dragState.translation.height : 0)
91                         .scaleEffect(self.dragState.isDragging && self.isTopCard(cardView:
92                             cardView) ? 0.95 : 1.0)
93                         .rotationEffect(Angle(degrees: self.isTopCard(cardView: cardView) ?
94                             Double( self.dragState.translation.width / 10) : 0))
95                         .animation(.interpolatingSpring(stiffness: 180, damping: 100))
96                         .gesture(LongPressGesture(minimumDuration: 0.01)
```

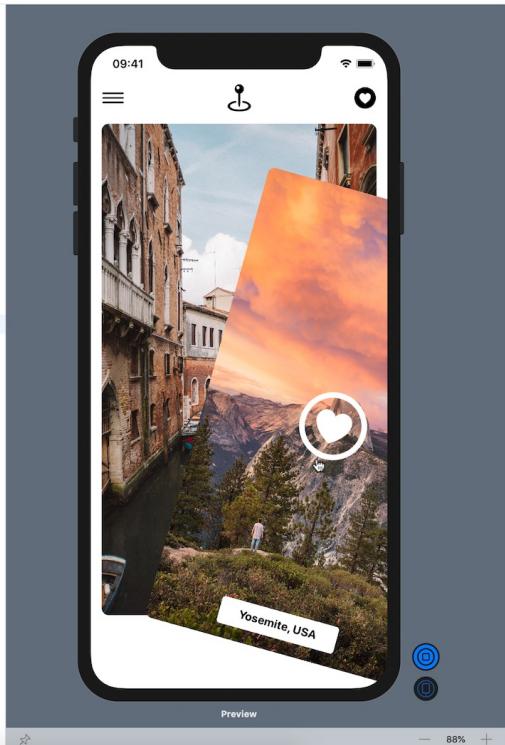


Figure 15. The heart icon appears

Removing/Inserting the Cards

Now when you release the card, it will still return to its original position. How can 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..  
        views.append(CardView(image: trips[index].image, title: trips[index].destination))  
    }  
  
    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 to 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. And 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 when you run the project in the preview canvas. Drag the image to the right/left until the icon appears. Releasing the drag and the topmost card should be replaced by the next card.

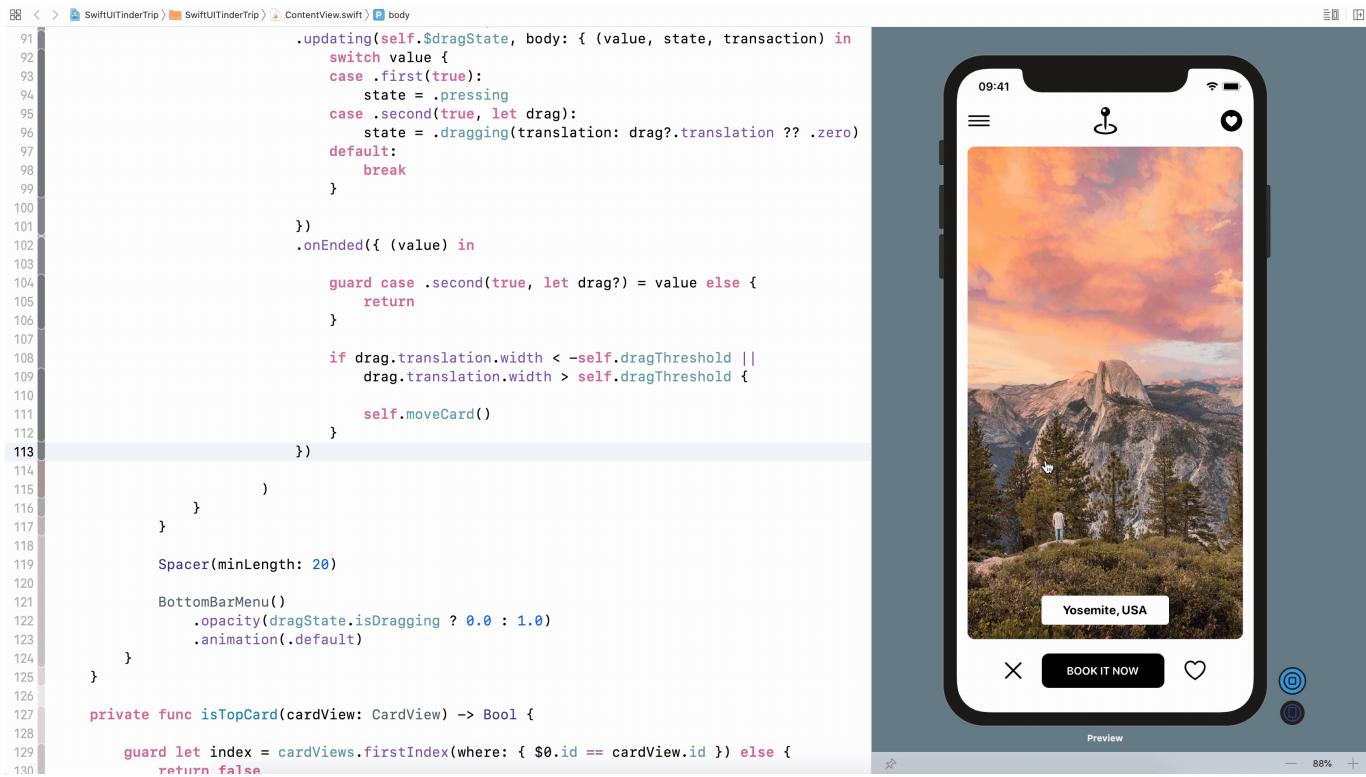


Figure 16. Removing the topmost card

Fine Tuning the Animations

The app almost works but the animation falls short of expectation. Instead of having the card view disappeared abruptly, the card should fall out of the screen gradually when it's thrown away.

To fine tune the animated effect, we will attach the `transition` modifier and apply an asymmetric transition to the card views.

Now create an extension of `AnyTransition` 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, updating 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()
        }
    })
})
```

```
})
```

```
)
```

What the code does is to set 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 one of the examples.

I hope you truly 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 the best practices. This is one of the many approaches to create this kind of UI. If you have 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 project here:

- Demo project (<https://www.appcoda.com/resources/swiftui/SwiftUITinderTrip.zip>)

Chapter 20

Advanced Animations and Transitions

This chapter will be released before end Jan 2019. You will receive the update for free via email.

Chapter 21

Putting Everything Together to Build a Personal Finance App

By now, you should have a good understanding of SwiftUI and build 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 his/her expense and income.

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 a real world app. In brief, here are some of the stuff we will go through with you:

1. How to build a form and perform validation
2. How to pass the form data between views using Combine
3. How to filter records and refresh the list view
4. How to use bottom sheet to display record details
5. How to use MVVM (Model-View-ViewModel) in SwiftUI

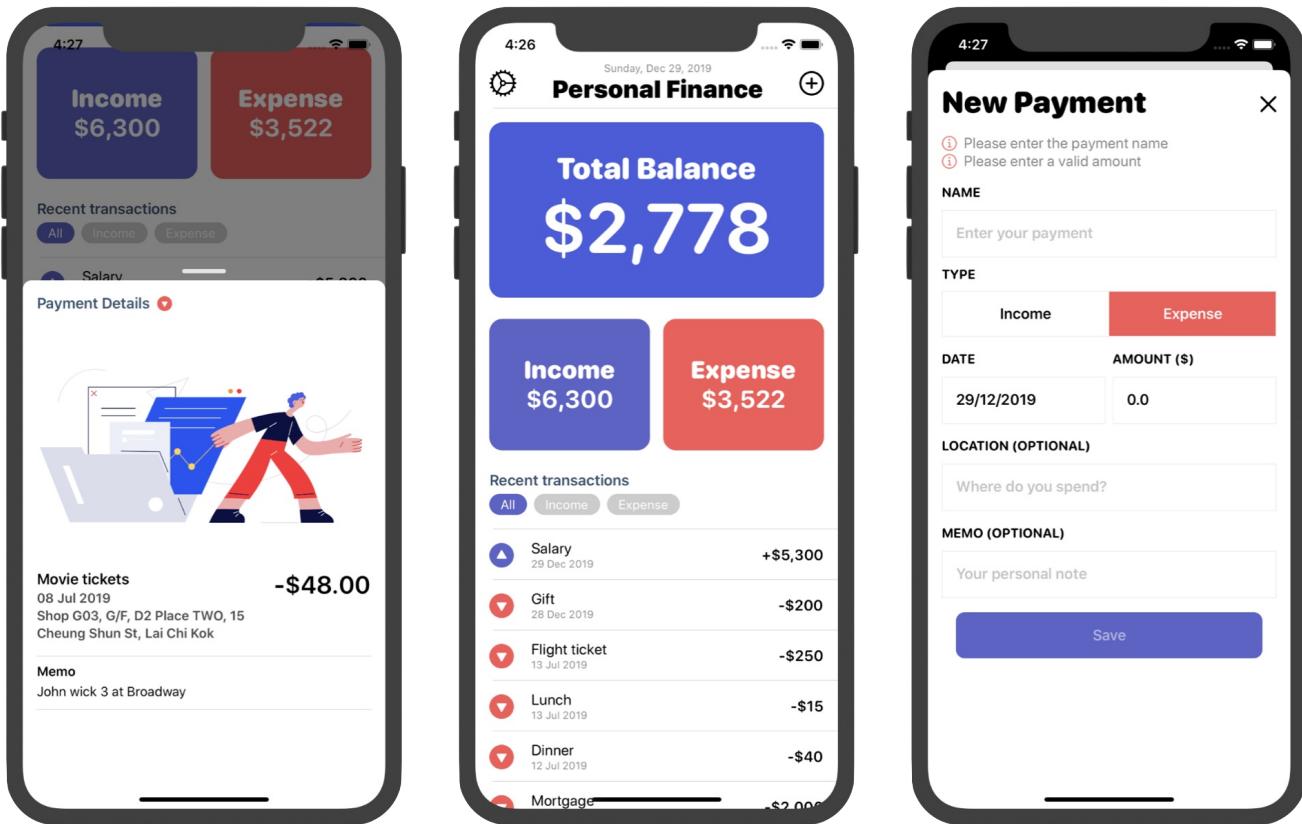


Figure 1. The Personal Finance App

First, download the full source code of the project (<https://www.appcoda.com/resources/swiftui/SwiftUIPFinance.zip>) to take a look. In the next update (by mid Jan), we will explain in details how each piece of the code works.

Note that this app will be further modified to use Core Data for data management. Right now, I want you to study the code of the project. If you want to challenge yourself, try to layout the main view and the form.