SAMPLE

Xcode 13, Swift 5.5 and iOS 15 Ready

Mastering SwiftUI

SIMON NG

APPCODA

Table of Contents


Xcode 13, Swift 5.5 and iOS 15 Ready
Mastering
SwiftUI
SIMON NG
APPCODA

Published by AppCoda Limited

Preface

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

I have been doing iOS programming for over 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.

SwiftUI has evolved so much in these two years. Apple has packed even more features and brought more UI components to the SwiftUI framework, which comes alongside with Xcode 13. It just takes UI development on iOS, iPadOS, and macOS to the next level. You can develop some fancy animations with way less code, as compared to UIKit. Most importantly, the latest version of the SwiftUI framework makes it easier for developers to develop apps for Apple platforms. You will understand what I mean after you go through the book.

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

Simon Ng
Founder of AppCoda

What You Will Learn in This Book

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

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

Audience

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

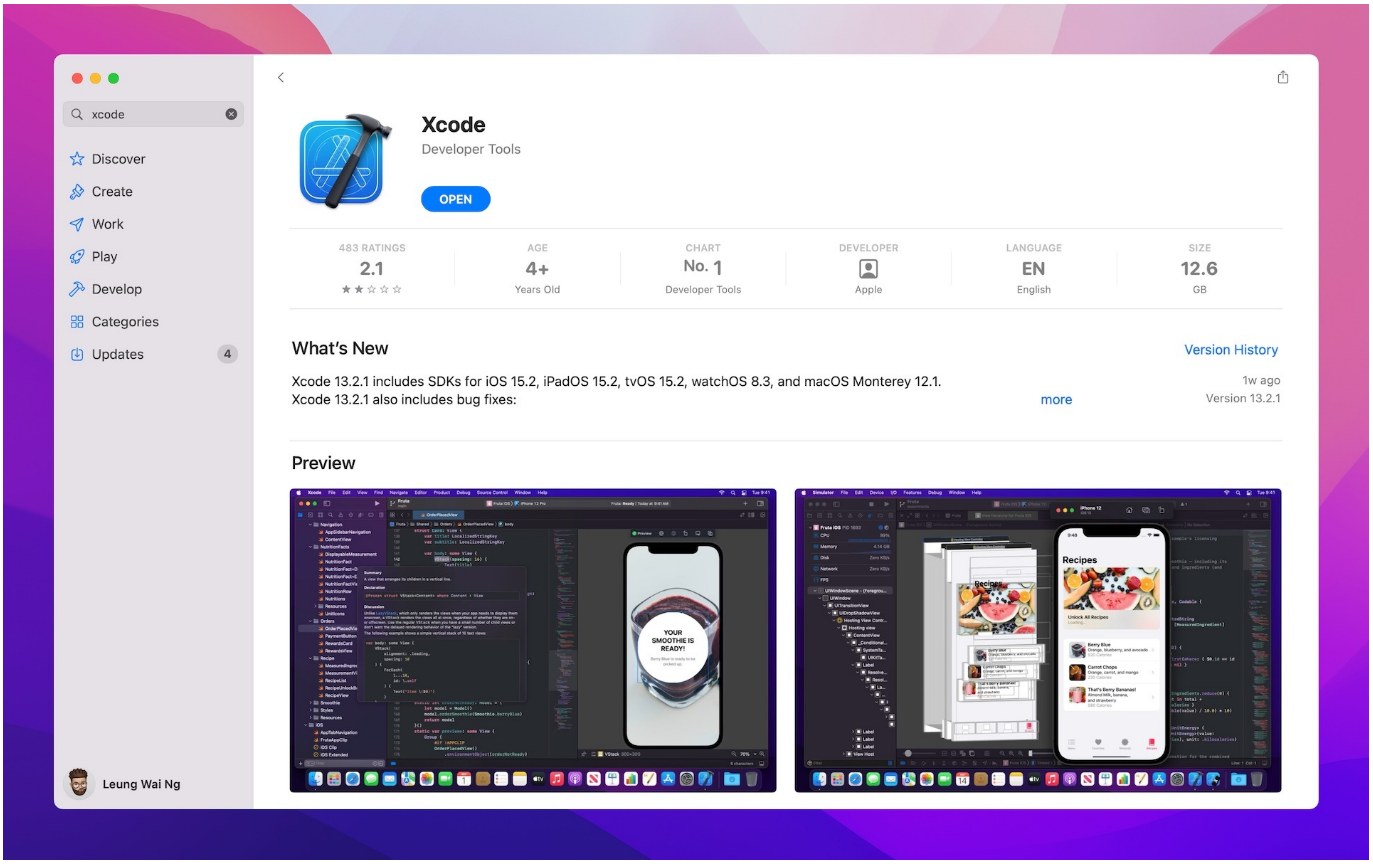
What You Need to Develop Apps with SwiftUI

Having a Mac is the basic requirement for iOS development. To use SwiftUI, you need to have a Mac installed with macOS Catalina and Xcode 11 (or up). That said, to properly follow the content of this book, you are required to have Xcode 13 installed.

If you are new to iOS app development, Xcode is an integrated development environment (IDE) provided by Apple. Xcode provides everything you need to kick start your app development. It already bundles the latest version of the iOS SDK (short for Software Development Kit), a built-in source code editor, graphic user interface (UI) editor, debugging tools and much more. Most importantly, Xcode comes with an iPhone (and iPad) simulator so you can test your app without the real devices. With Xcode 13, you can instantly preview the result of your SwiftUI code.

Installing Xcode

To install Xcode, go up to the Mac App Store and download it. Simply search "Xcode" and click the "Get" button to download it. At the time of this writing, the latest official version of Xcode is 13.2. Once you complete the installation process, you will find Xcode in the Launchpad.


xcode
Discover
Create
Work
Play
Develop
Categories
Updates
4
Leung Wai Ng
Xcode
Developer Tools
OPEN
483 RATINGS
2.1
AGE
4+
Years Old
CHART
No. 1
Developer Tools
DEVELOPER
Apple
LANGUAGE
EN
English
SIZE
12.6
GB
What's New
Version History
Xcode 13.2.1 includes SDKs for iOS 15.2, iPadOS 15.2, tvOS 15.2, watchOS 8.3, and macOS Monterey 12.1.
Xcode 13.2.1 also includes bug fixes:
more
1w ago
Version 13.2.1
Preview

Frequestly Asked Questions about SwiftUI

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

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

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

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

   The short answer is Both. That said, it all depends on your goals. If you target to become a professional iOS developer and apply for a job in iOS development, you better equip yourself with knowledge of SwiftUI and UIKit. Over 90% 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.

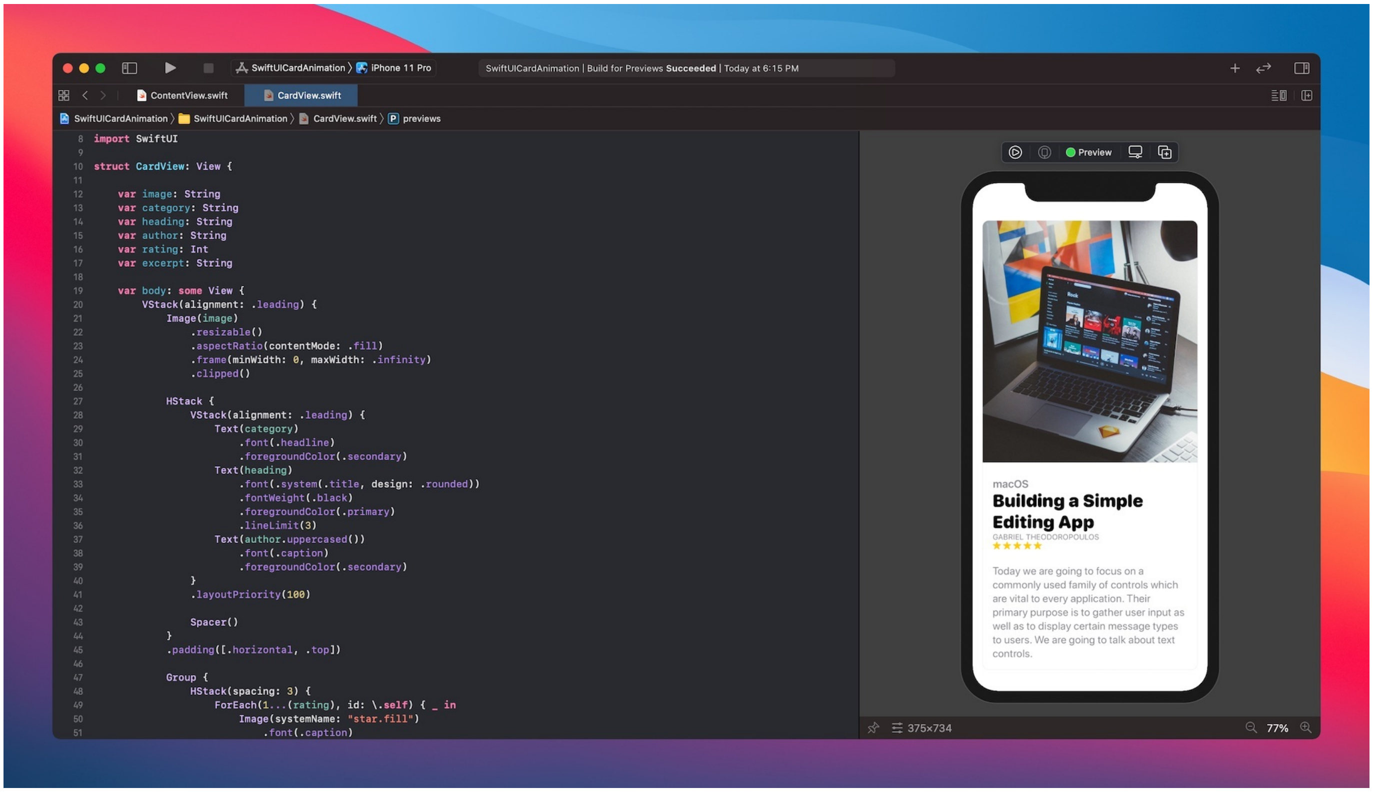


Figure 1. Programming in SwiftUI

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

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

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

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

Declarative vs Imperative Programming

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

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

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

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

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

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

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

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

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

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

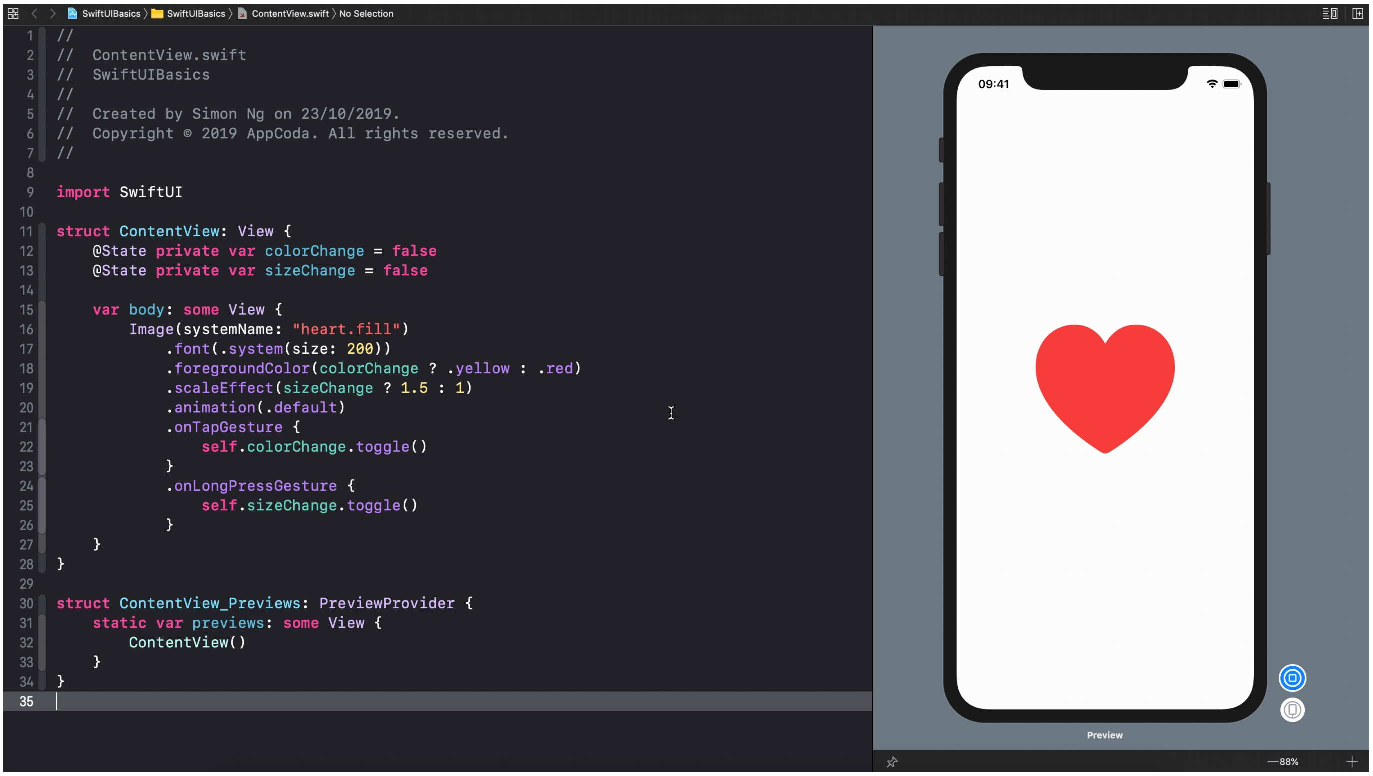


Figure 2. The implementation of an interactive heart button

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

No more Interface Builder and Auto Layout

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

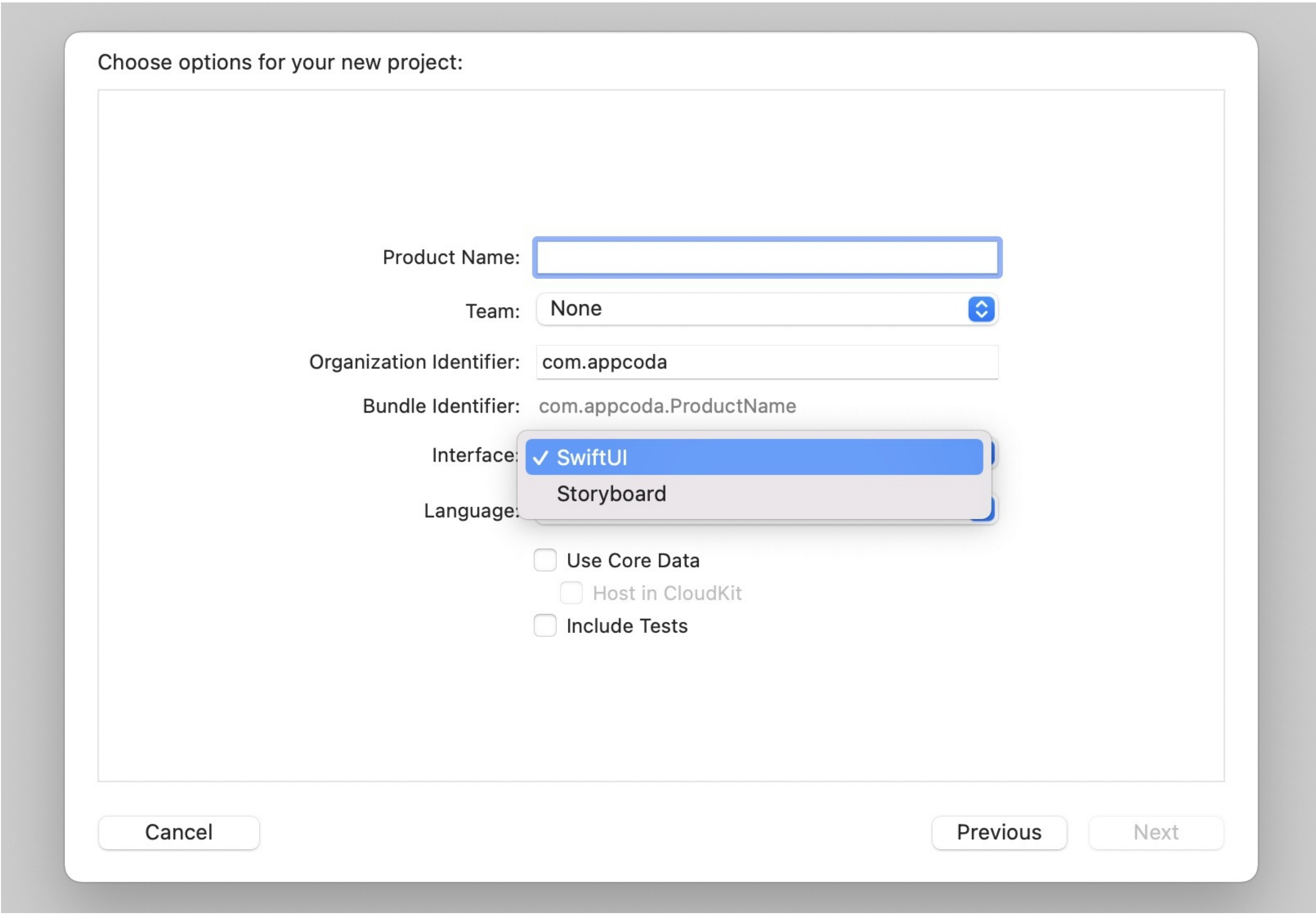


Figure 3. User interface option in Xcode

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

The Combine Approach

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

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

Learn Once, Apply Anywhere

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

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

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

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

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

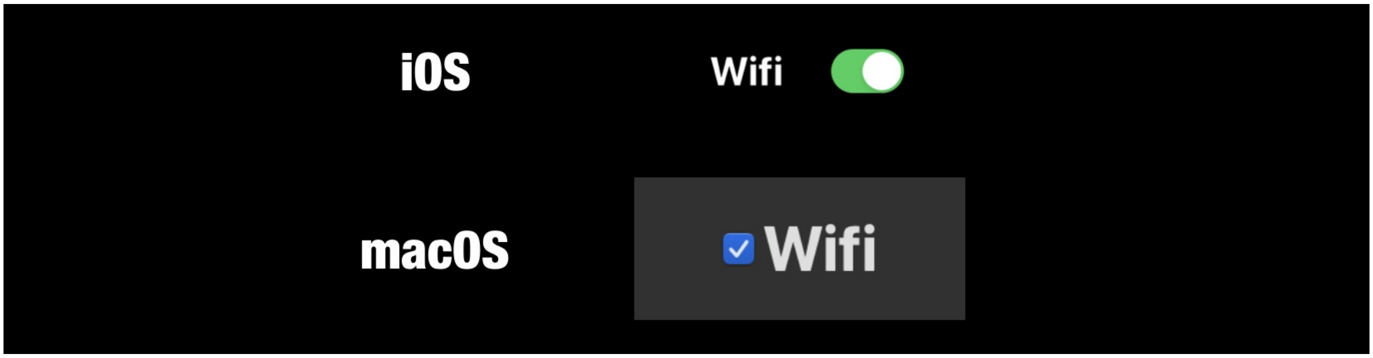


Figure 4. Toggle on macOS and iOS

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

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

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

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

Interfacing with UIKit/AppKit/WatchKit

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

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

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

Figure 5. The Representable protocols for existing UI frameworks

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

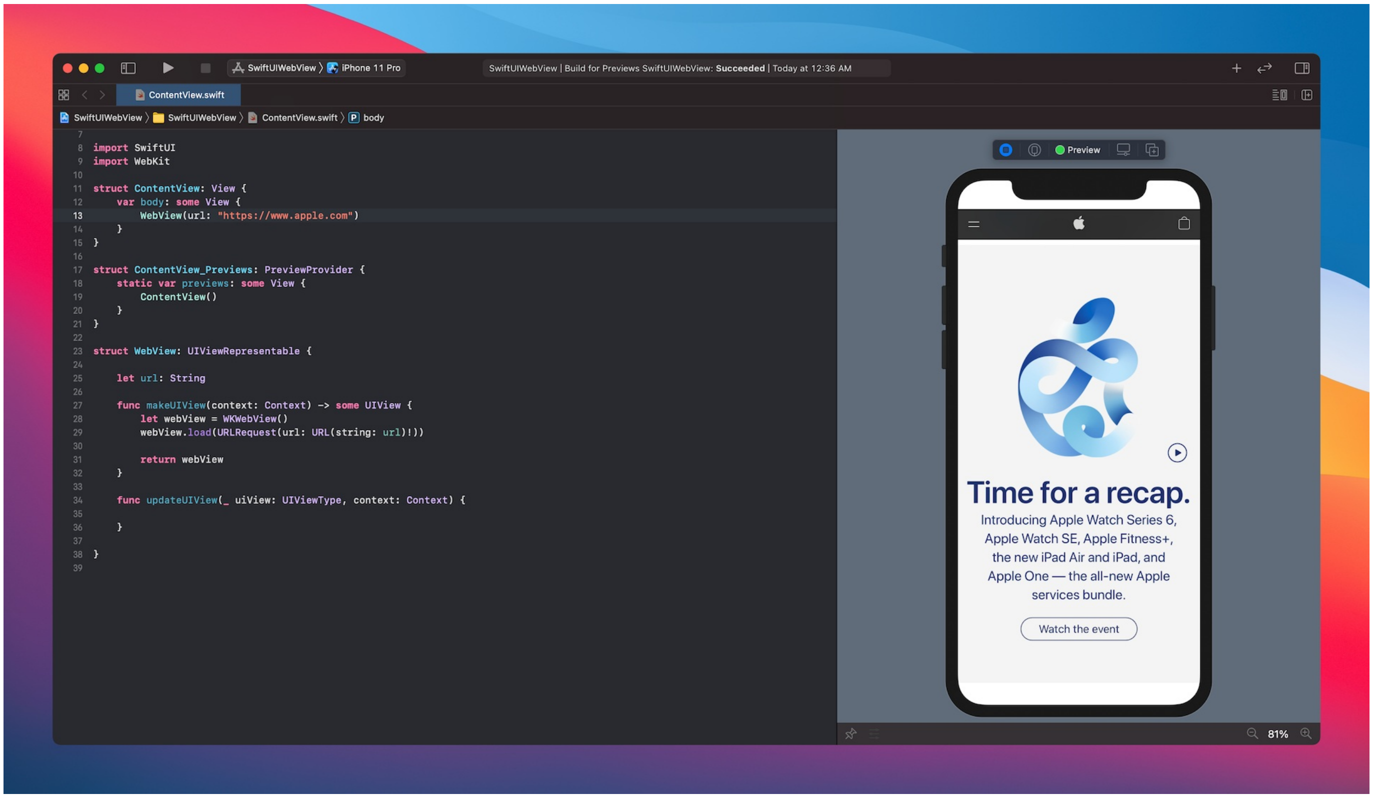


Figure 6. Porting WKWebView to SwiftUI

Use SwiftUI for Your Next Project

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

Though SwiftUI is still new to most developers, now is the right time to learn and incorporate the framework into your new project. Along with the release of Xcode 13, Apple has made the SwiftUI framework more stable and feature-rich. If you have some personal projects or side projects for personal use or at work, there is no reason why you shouldn't try out SwiftUI.

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

At the time of this writing, SwiftUI has been officially released for more than two years. The debut of Xcode 13 has brought us more UI controls and new APIs for SwiftUI. In terms of features, you can't compare it with the existing UI frameworks (e.g. UIKit), which has been available for years. Some features (e.g. changing the separator style in table views) which are present in the old framework may not be available in SwiftUI. You may need to develop some solutions to work around the issue. This is something you have to take into account when adopting SwiftUI in production projects.

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

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

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 *App* template under the iOS category. Apple has revamped some of the project templates. If you have used the older version of Xcode before, the *Single Application* template is now replaced with the *App* template.

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

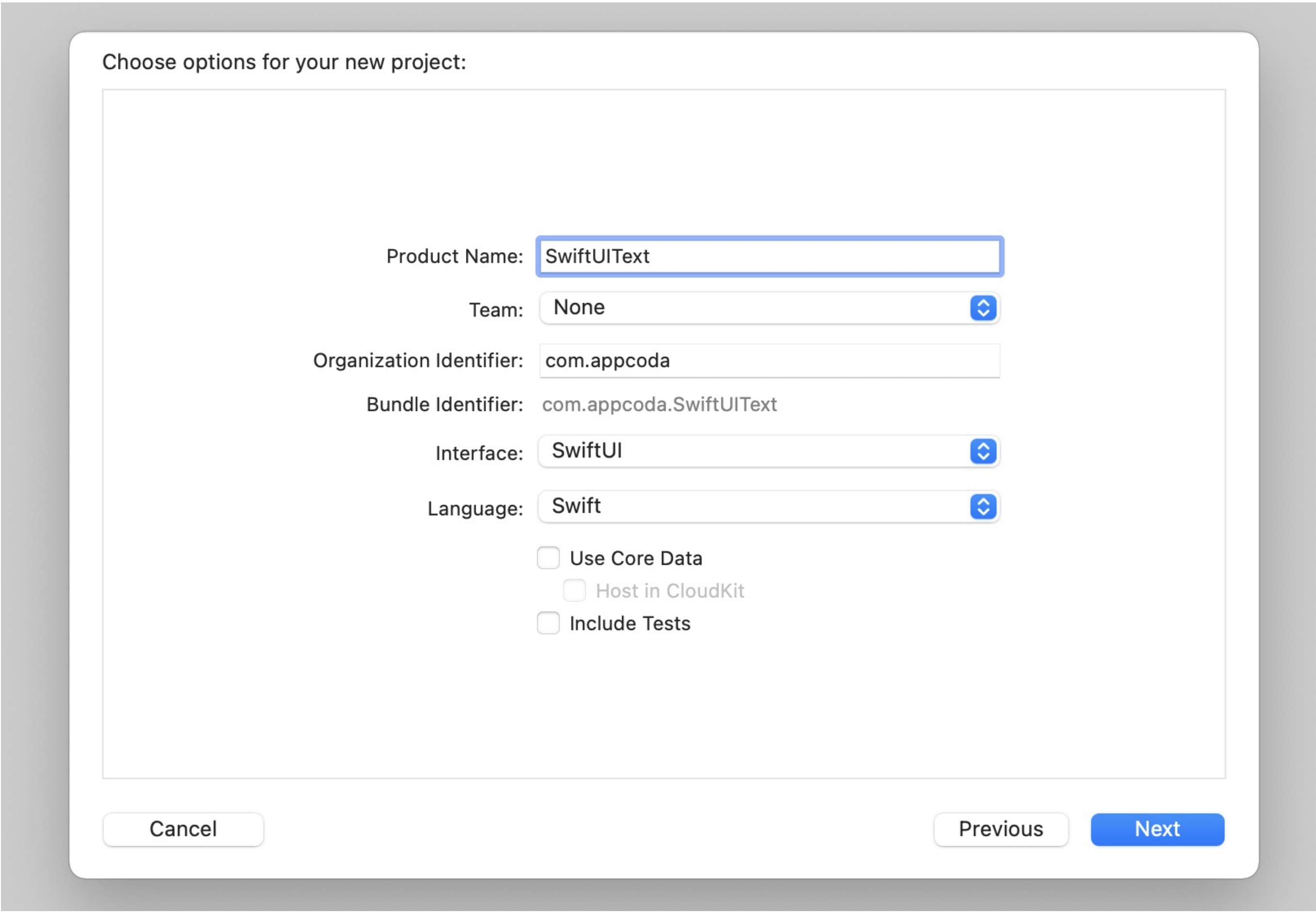


Figure 1. Creating a new project

To use SwiftUI, you have to choose *SwiftUI* in the *Interface* option. The language should be set to *Swift*. Click *Next* and choose a folder to create the project.

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

By default, Xcode generates some SwiftUI code for `ContentView.swift`. However, the preview canvas doesn't render the app preview. You have to click the *Resume* button in order to see the preview. After you click the button, Xcode renders the preview in a simulator that you choose in the simulator selection (e.g. iPhone 13 Pro).

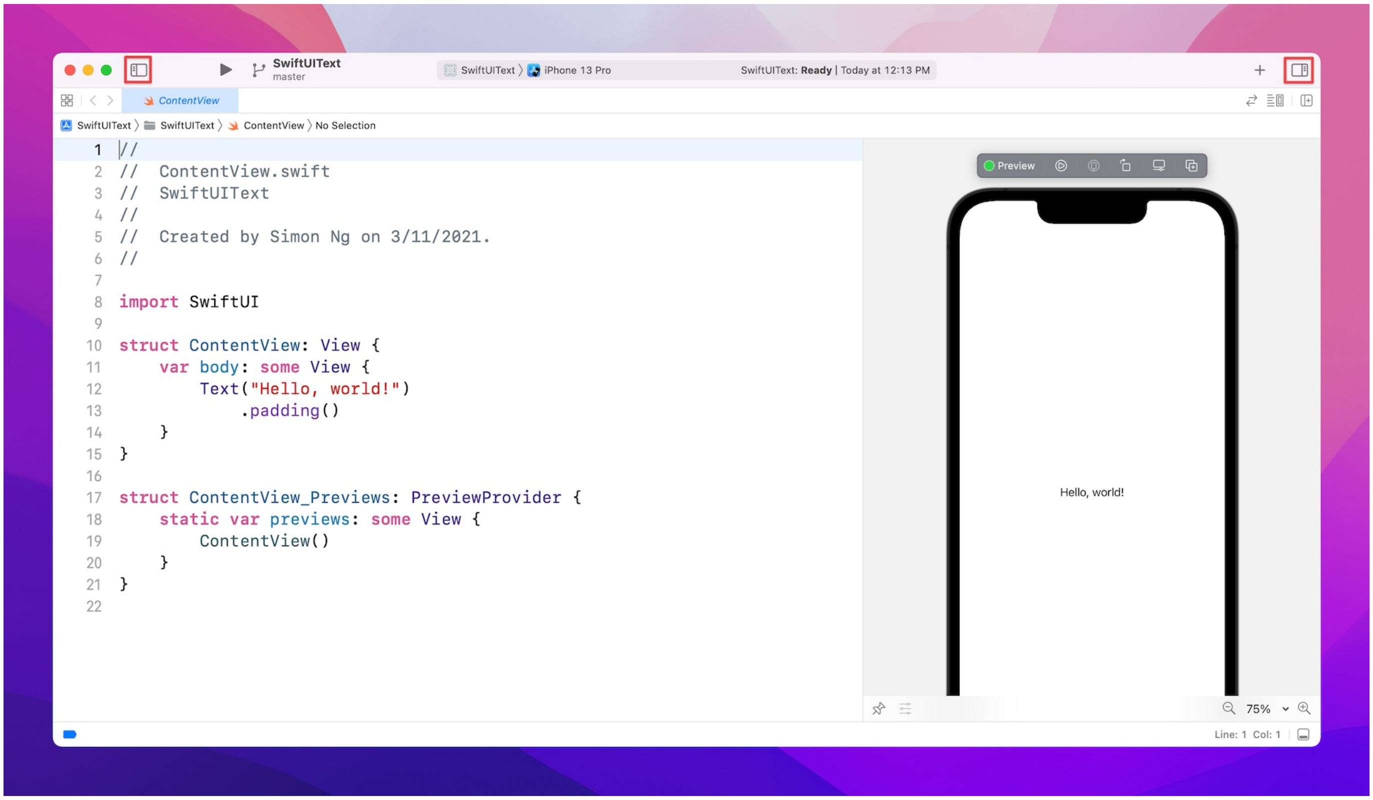


Figure 2. The code editor and the canvas

Displaying Simple Text

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

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

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

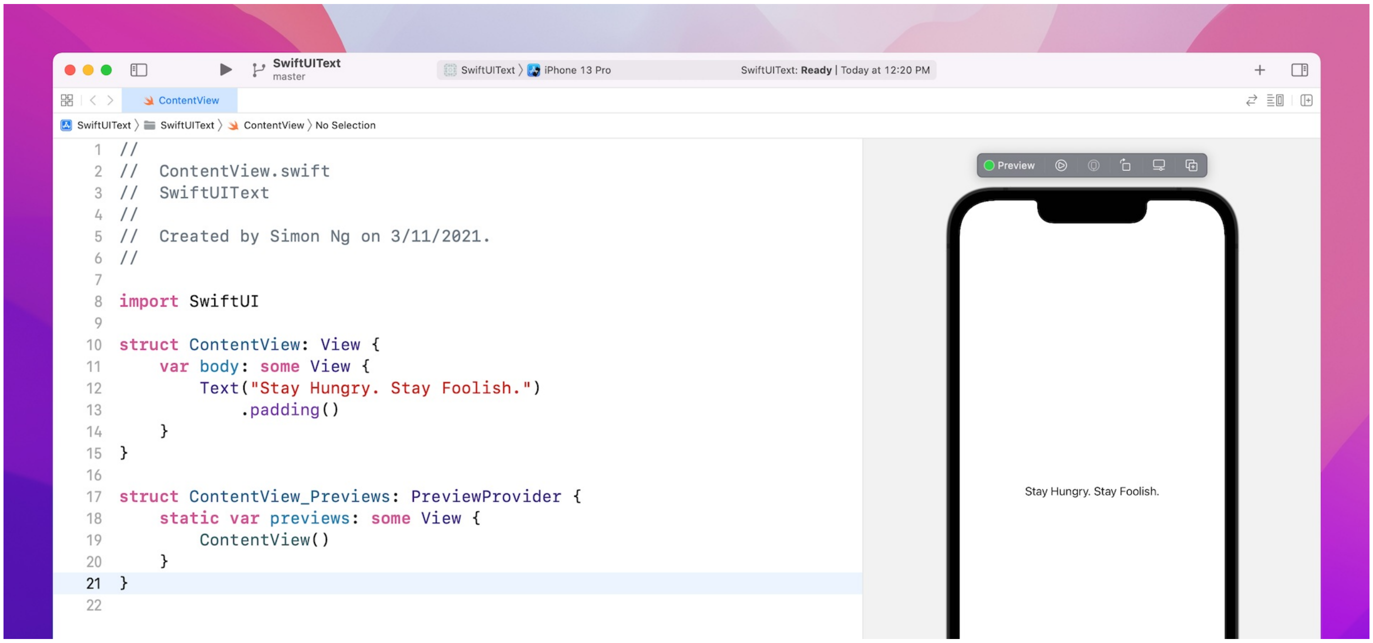


Figure 3. Changing the text

Changing the Font Type and Color

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

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

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

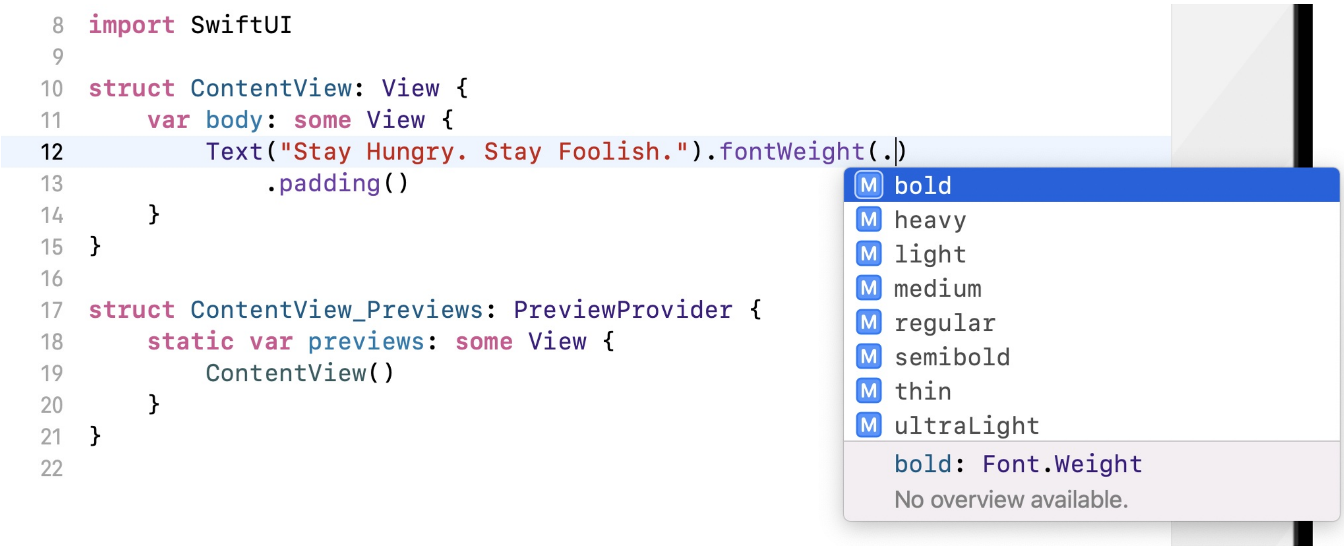


Figure 4. Choosing your preferred font weight

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

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

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

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

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

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

`.title` with `.largeTitle`.

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

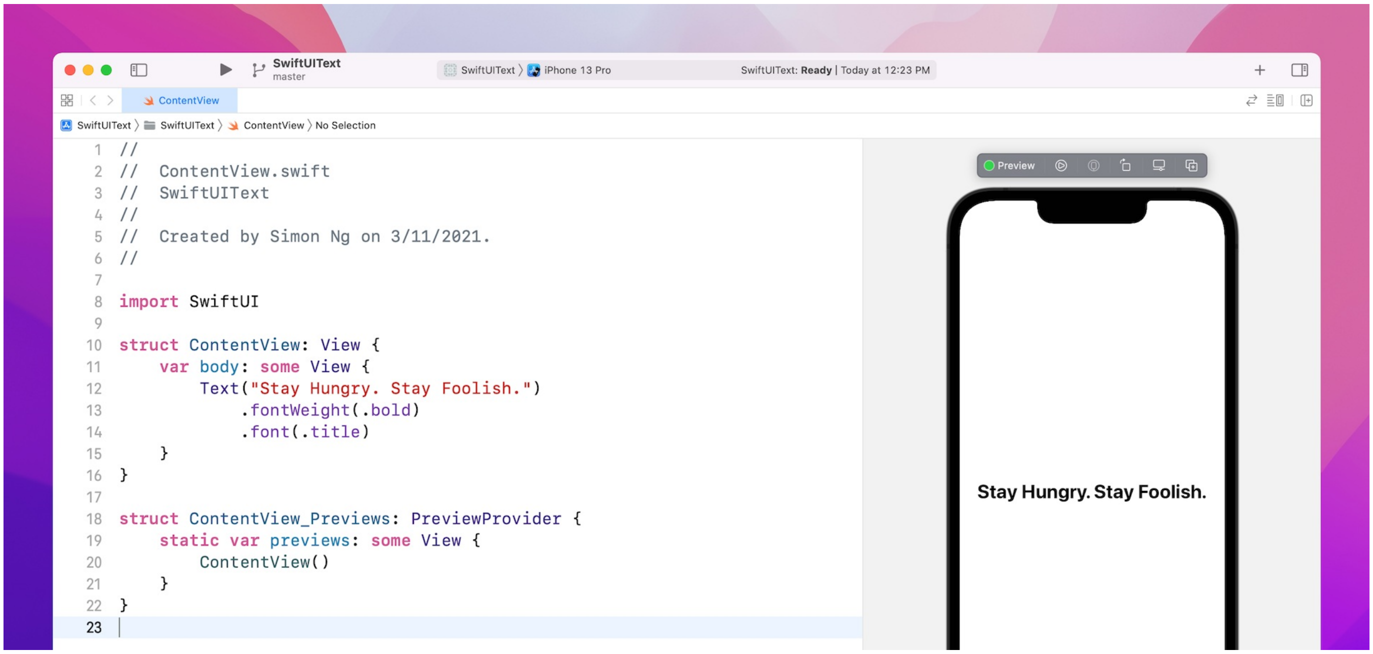


Figure 5. Changing the font type

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

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

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

Figure 6. Using the rounded font design

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

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

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

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

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

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

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

Figure 7. Changing the font color

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

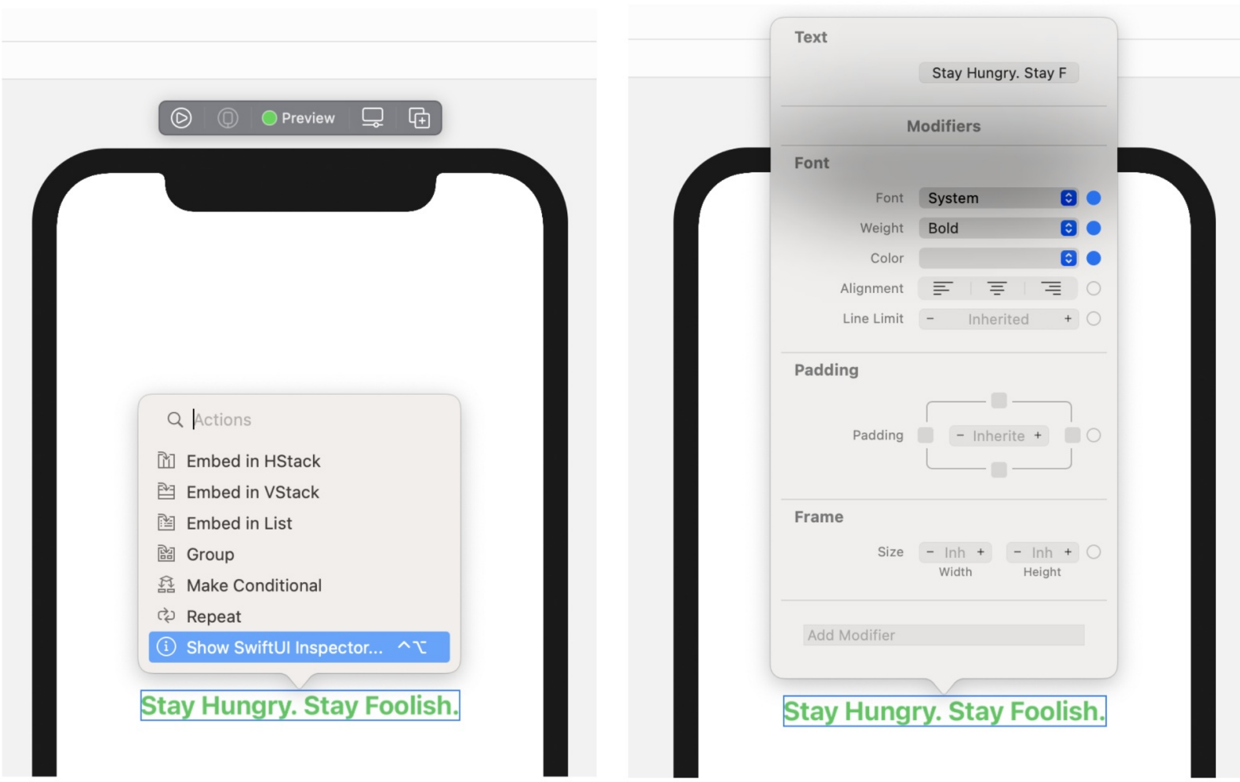


Figure 8. Using the Inspect feature to edit the properties of the text

Working with Multiline Text

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

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

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

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

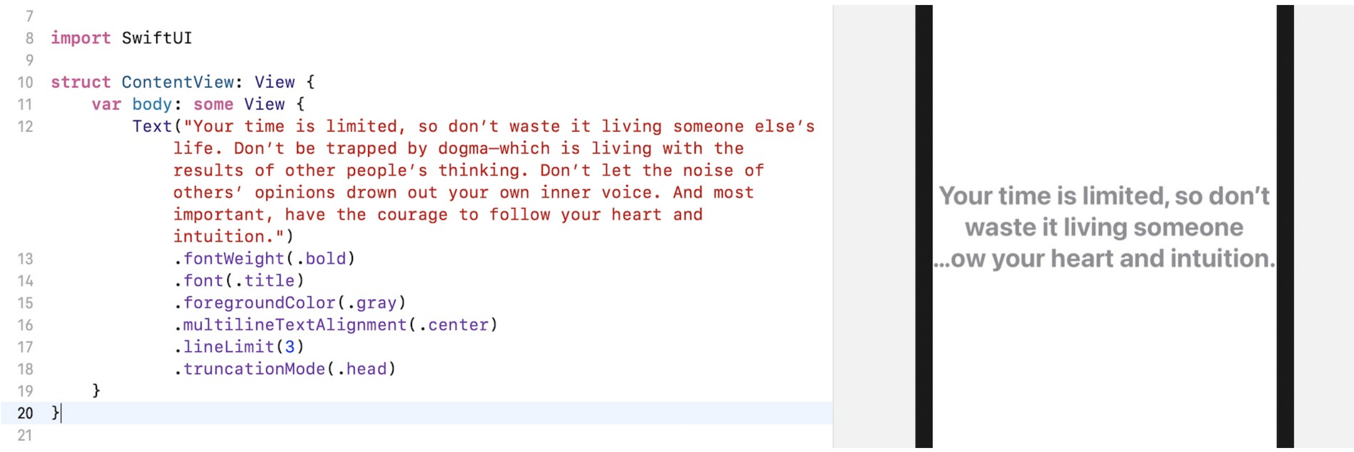


Figure 10. Using the .head truncation mode

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

```
.lineLimit(nil)
```

Setting the Padding and Line Spacing

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

```
.lineSpacing(10)
```

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

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

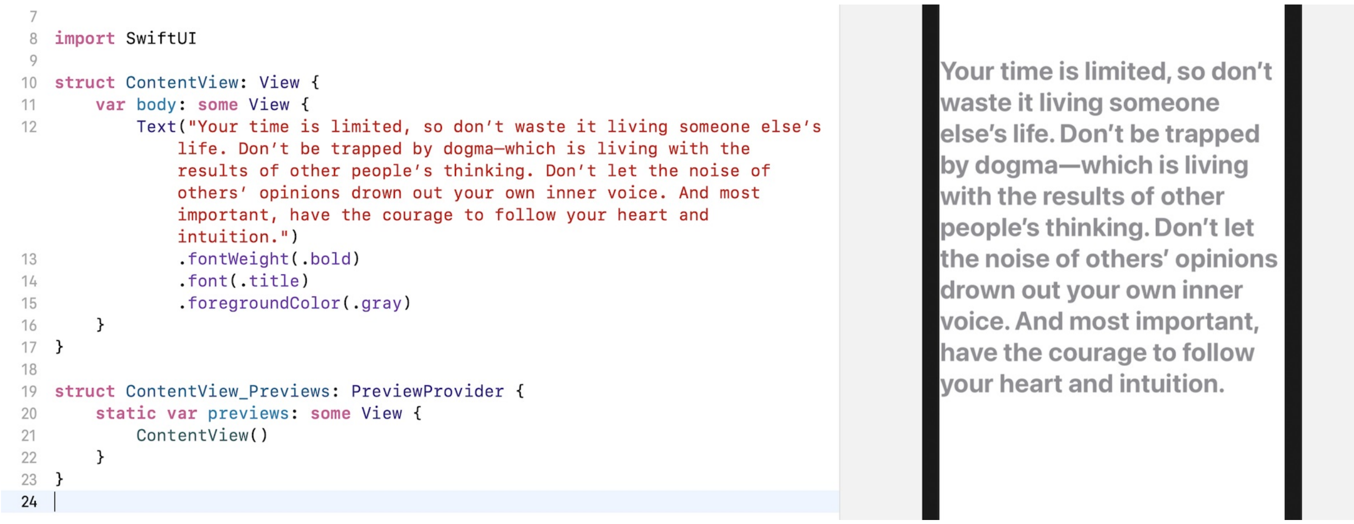


Figure 9. Display multiline text

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

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

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

```
.lineLimit(3)
```

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

```
.padding()
```

Your design canvas should now look like this:

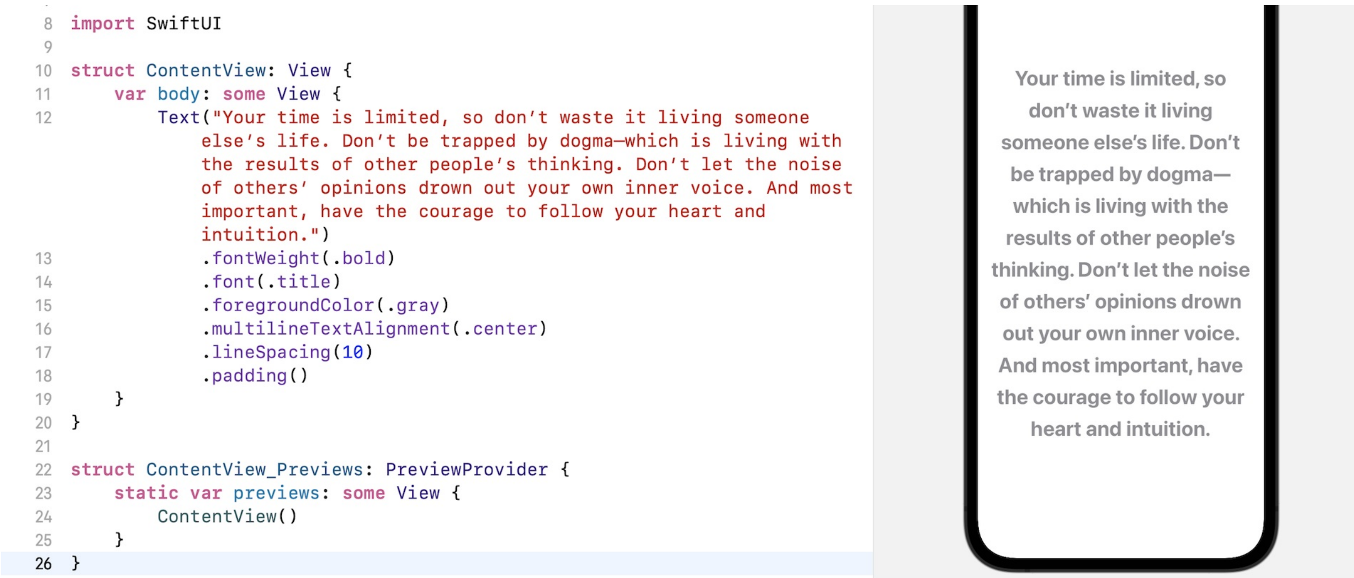


Figure 11. Setting the padding and line spacing of the text

Rotating the Text

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

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

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

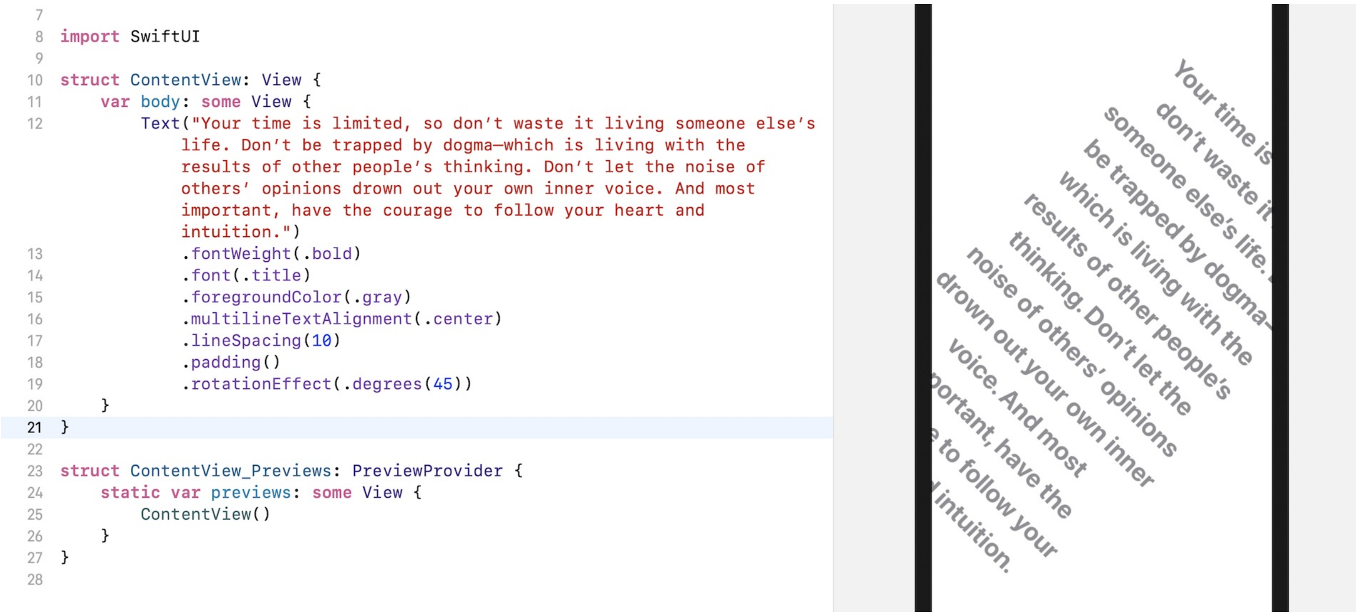


Figure 12. Rotate the text

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

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

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

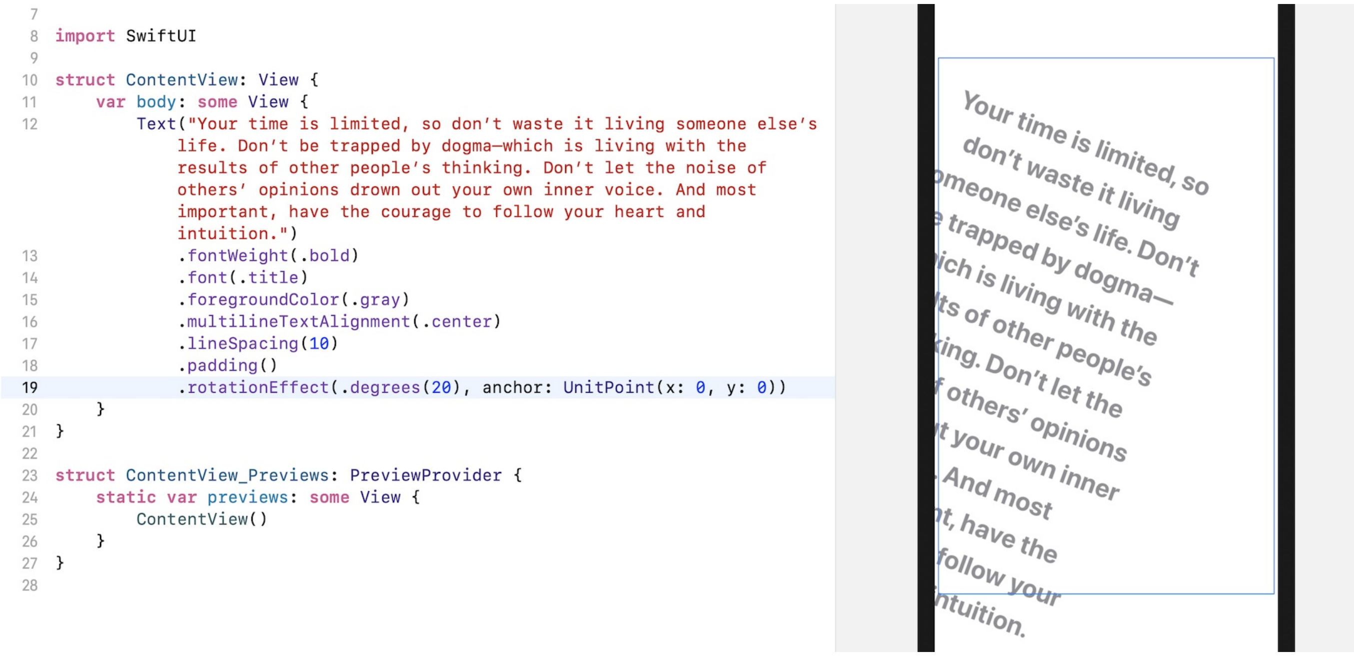


Figure 13. Rotate the text around the top-left of the text view

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

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

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

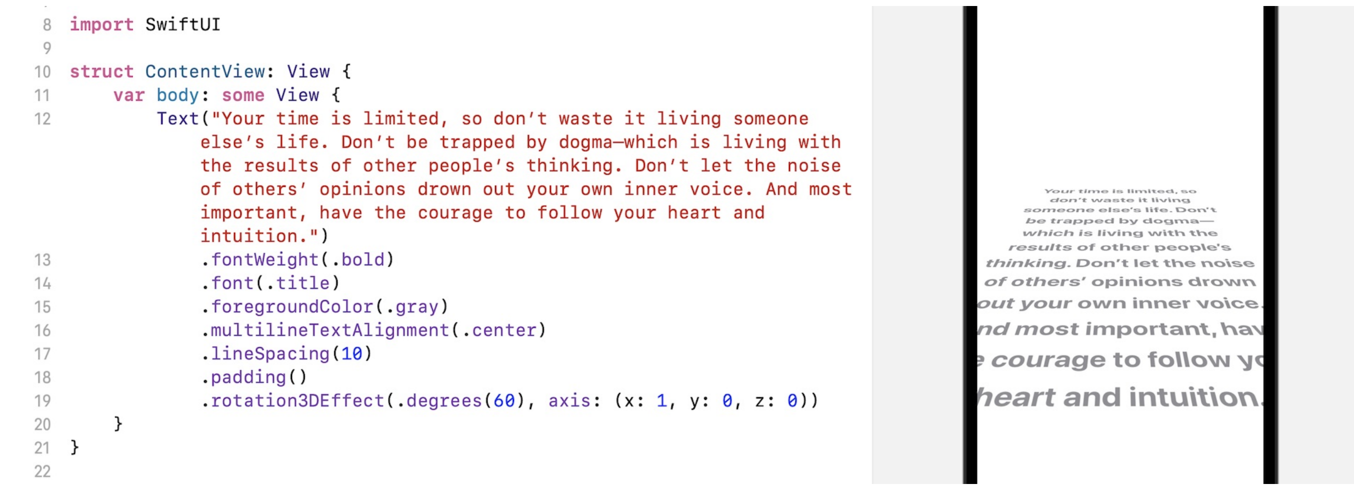


Figure 14. Create amazing text effect by using 3D rotation

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.

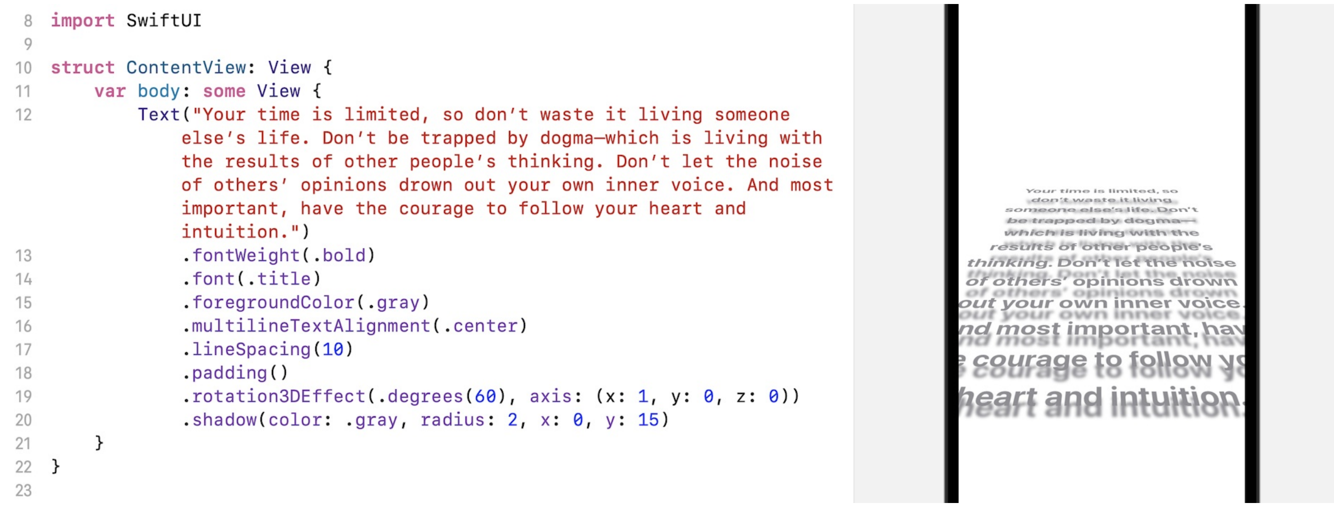


Figure 15. Applying the drop shadow effect

Using Custom Fonts

By default, all text is displayed using the system font. Say, you find a free font on Google Fonts (e.g. https://fonts.google.com/specimen/Nunito). How can you use a custom font in the app?

Assuming you've downloaded the font files, you should first add it to your Xcode project. You can simply drag the font files to the project navigator and insert them under the *SwiftUIText* folder. For this demo, I just add the regular font file (i.e. Nunito-Regular.ttf). If you need to use the bold or italic font, please add the corresponding font files.

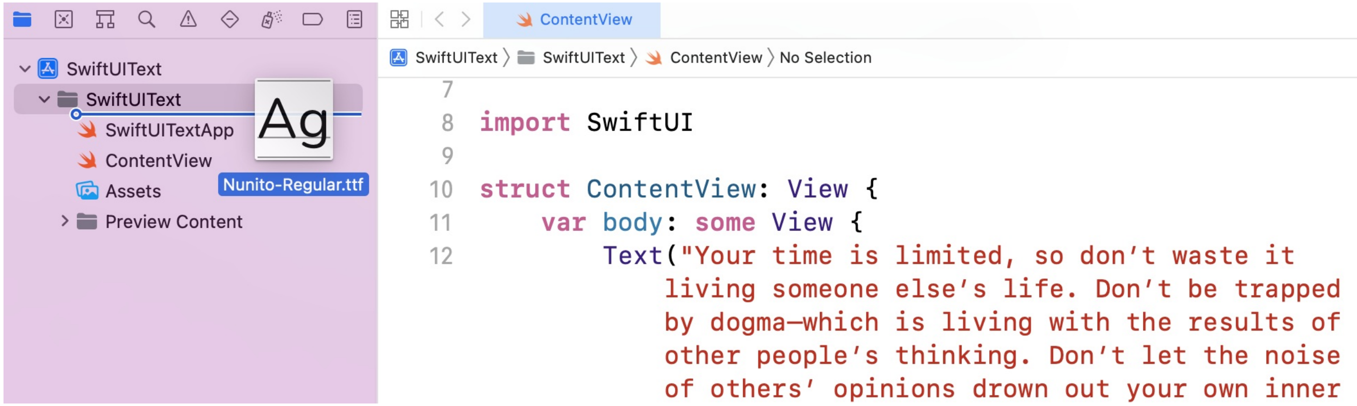


Figure 16. Adding the font file to the project

Once you added the font, Xcode will prompt you an option dialog. Please make sure you enable *Copy items if added* and check the *SwiftUIText* target.

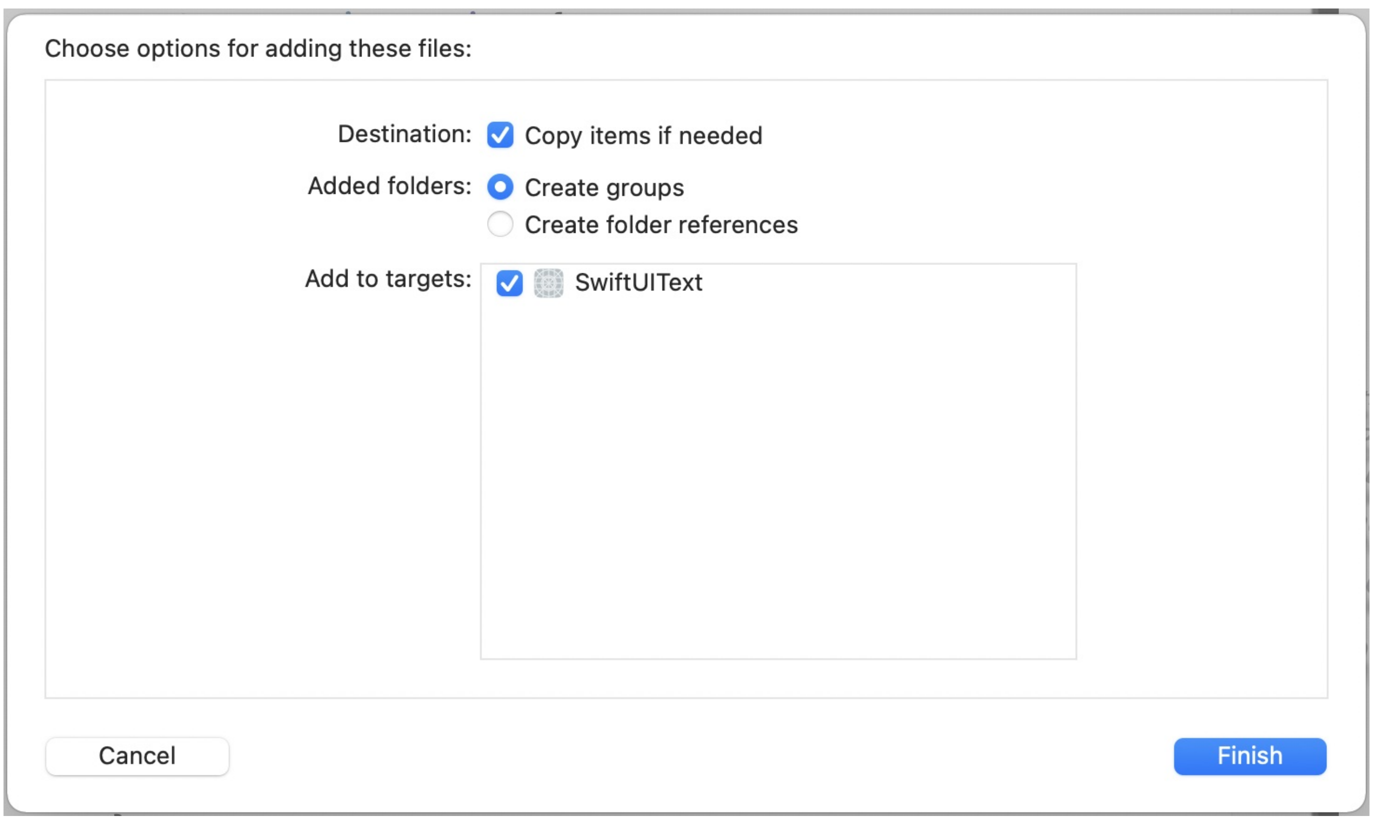


Figure 17. Choosing the options for adding files

After adding the font file, you still can't use the font directly. Xcode requires developers to register the font in the project configuration. In the project navigator, select *SwiftUIText* and then click *SwiftUIText* under Targets. Choose the *Info* tab, which displays the project configuration.

You can right *Bundle name* and choose *Add row*. Set the key name to *Fonts provided by application*. Next, click the disclosure indicator to expand the entry. For *item 0*, set the value to `Nunito-Regular.ttf`, which is the font file you've just added. If you have multiple font files, you can click the `+` button to add another item.

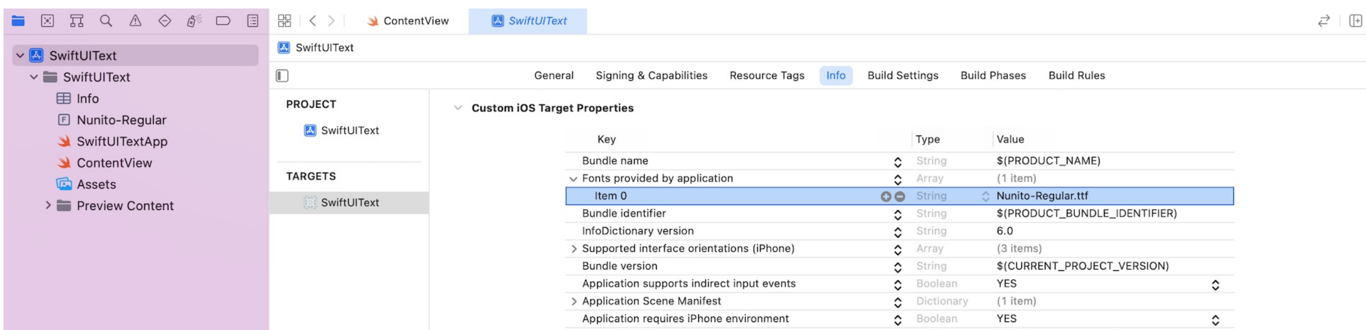


Figure 18. Set the font file in the project configuration

Now you can go back to `ContentView.swift`. To use the custom font, you can replace the following line of code:

```
.font(.title)
```

With:

```
.font(.custom("Nunito", size: 25))
```

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

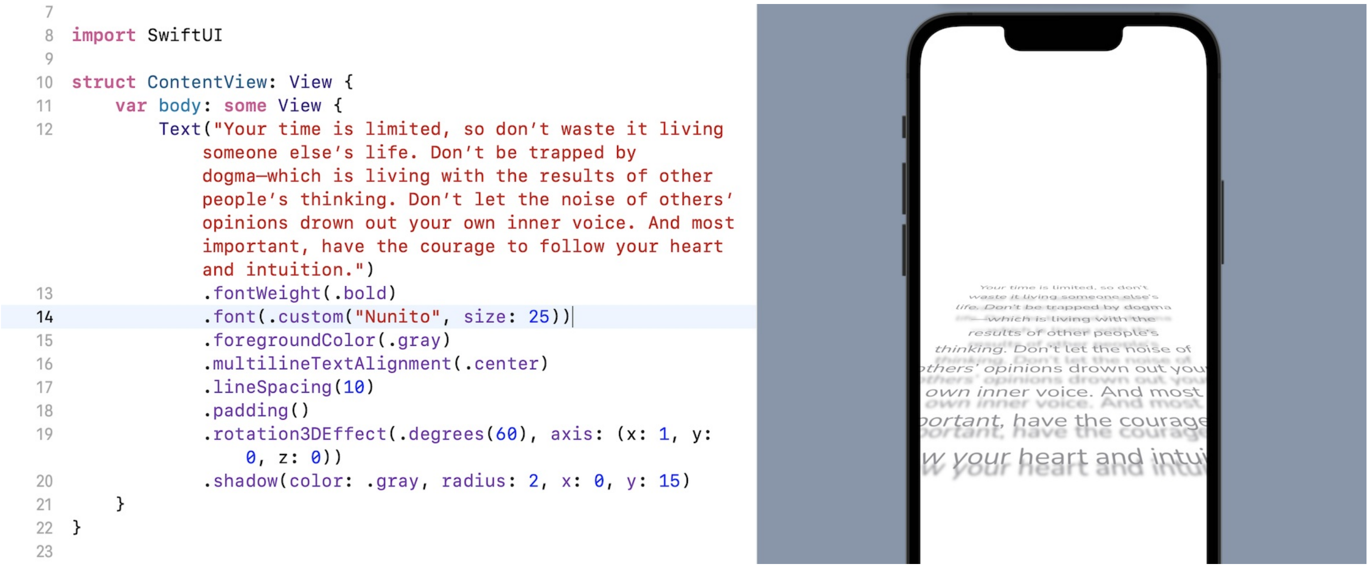


Figure 19. Using the custom font

Displaying Markdown Text

> Markdown is a lightweight markup language that you can use to add formatting elements to plaintext text documents. Created by John Gruber in 2004, Markdown is now one of the world's most popular markup languages.

In iOS 15, SwiftUI has built-in support for rendering Markdown. If you don't know what Markdown is, it allows you to style plain text using an easy to read format. To learn more about Markdown, you can check out this guide (https://www.markdownguide.org/getting-started/).

To use Markdown for rending text, all you need to do is specify the text in Markdown. The `Text` view automatically renders the text for you. Here is an example:

```
Text("**This is how you bold a text**. *This is how you make text italic.* You can
 [click this link](https://www.appcoda.com) to go to appcoda.com")
    .font(.title)
```

If you write the code in `ContentView`, you will see how the given text is rendered. To test the hyperlink, you have to run the app in simulators. When you tap the link, iOS will redirect to mobile Safari and open the URL.

Figure 20. Using Markdown

Summary

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

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

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

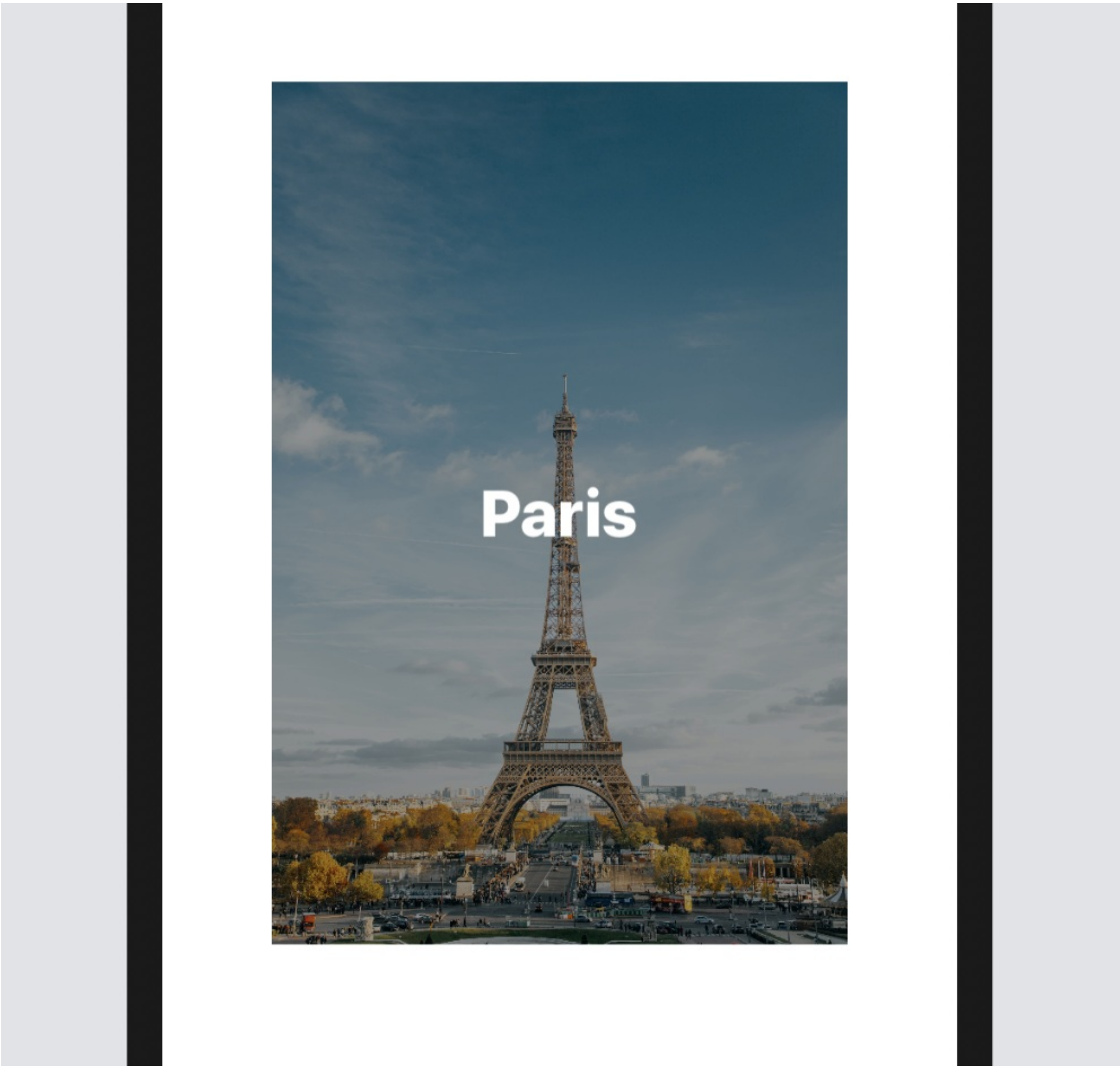


Figure 17. Darken an image and apply a text overlay

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

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 unevitable 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.

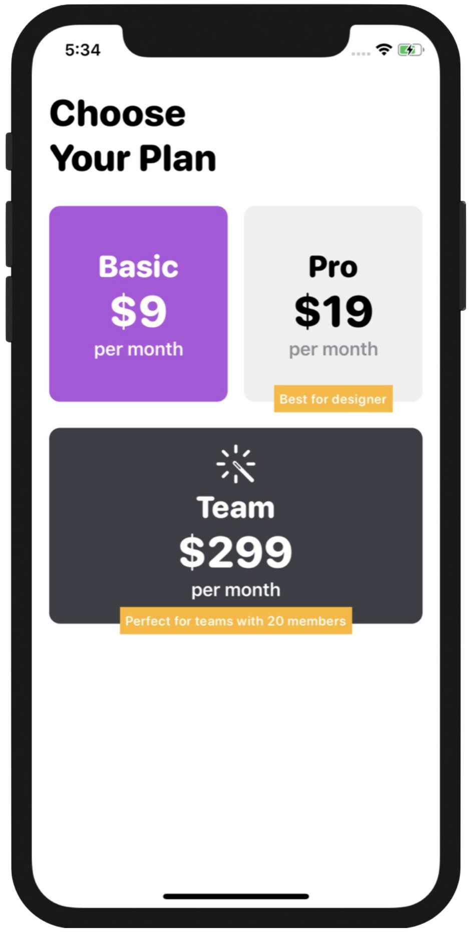


Figure 1. The demo app

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

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.

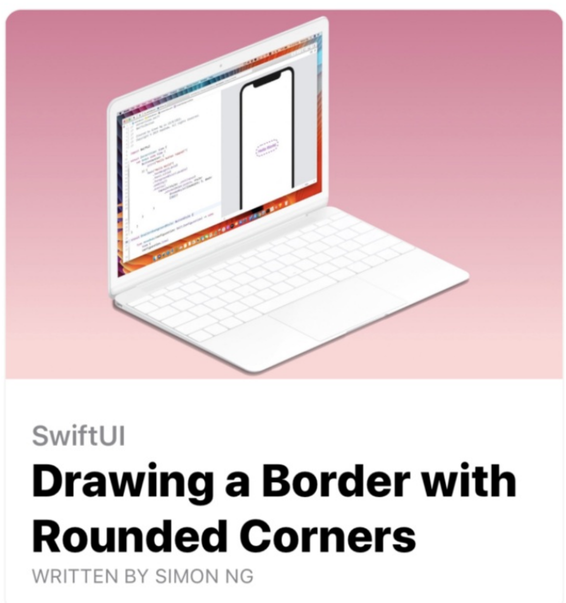


Figure 1. The card view

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.

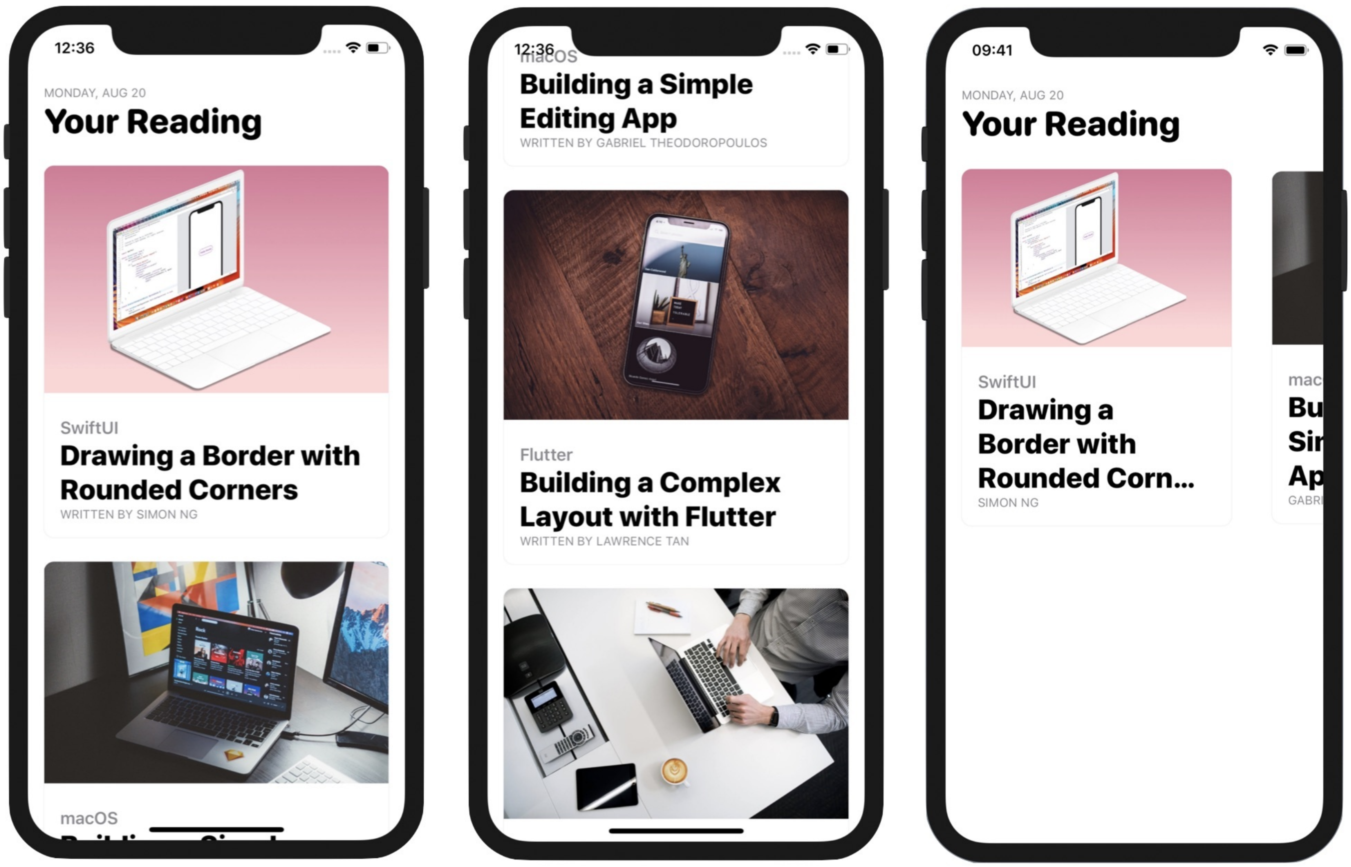


Figure 2. Building a scrollable UI with ScrollView

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

Chapter 6
Working with SwiftUI Buttons, Labels and Gradient

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

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

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

Creating a New Project with SwiftUI enabled

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

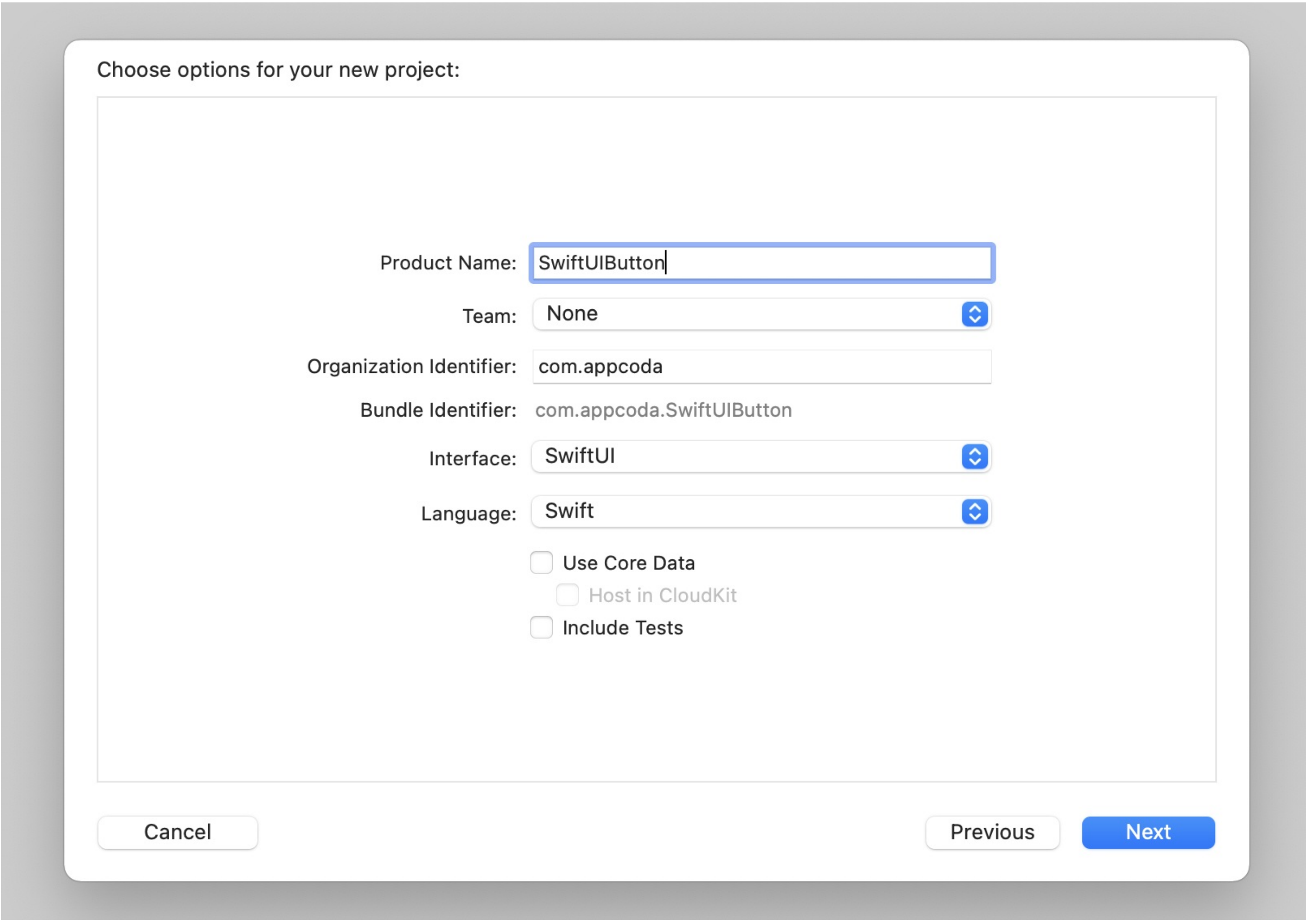


Figure 1. Creating a new project

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

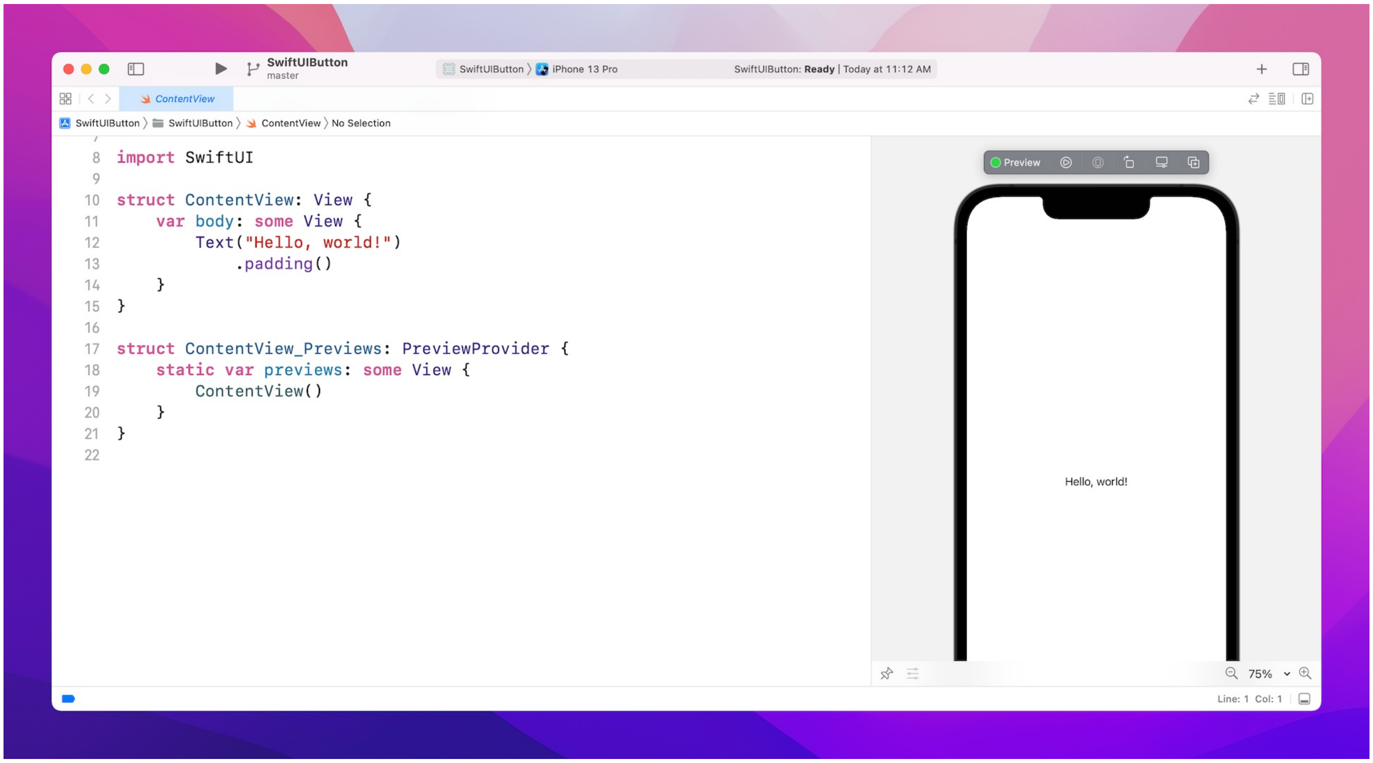


Figure 2. Previewing the default content view

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

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

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

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

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

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

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

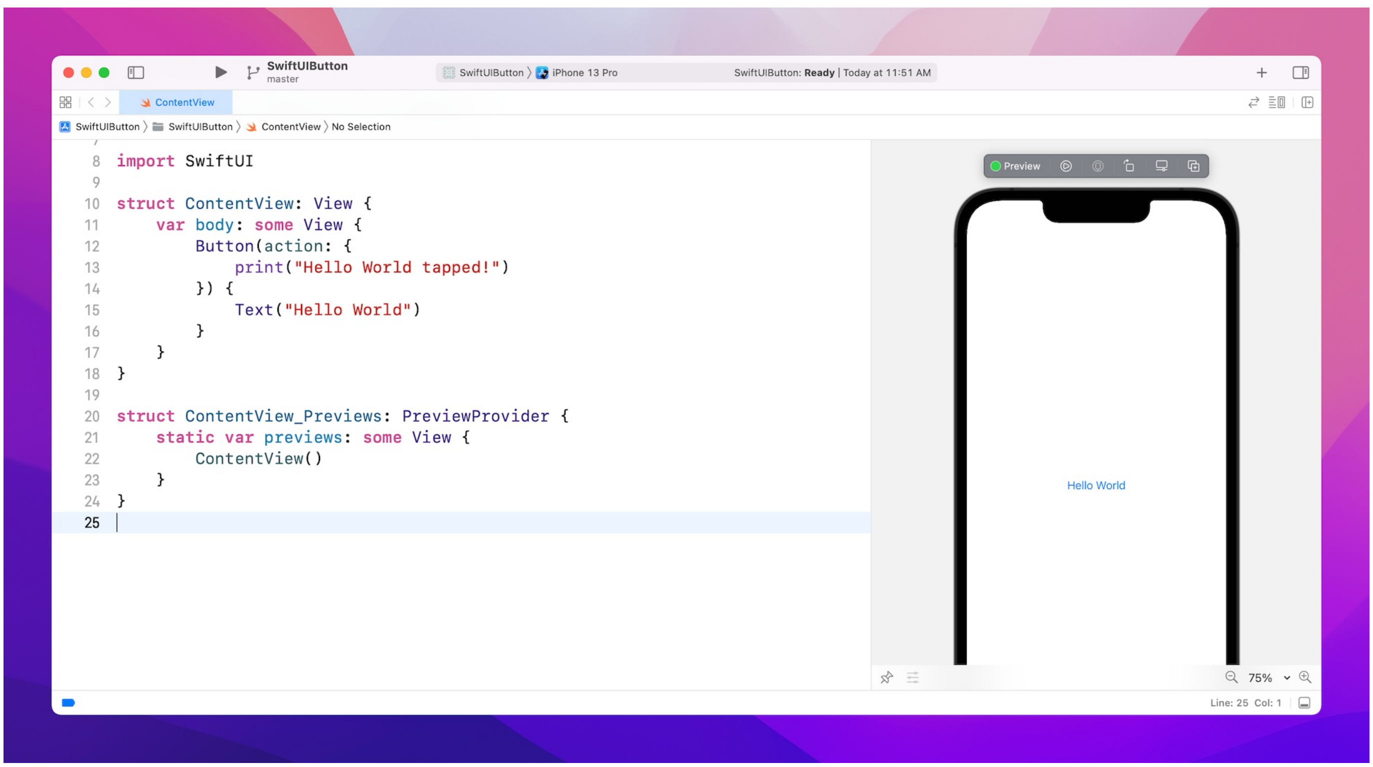


Figure 3. Creating a simple button

The button is non-tappable in the design canvas. To test it, click the *Play* button to run the app. For Xcode 12 (or lower), you can right-click the *Play* button and then choose *Debug Preview*. You will see the *Hello World tapped* message on the console when you tap the button. If you can't see the console, go up to the Xcode menu and choose *View > Debug Area > Activate Console*.

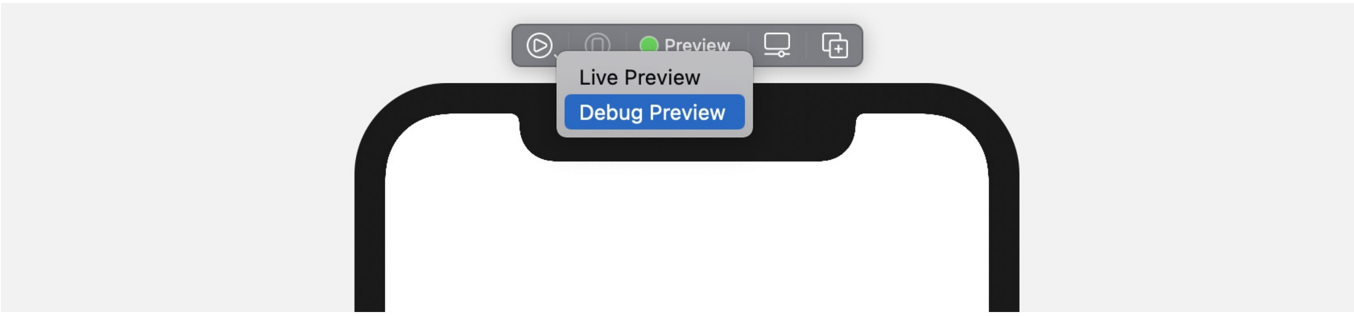


Figure 4. The console message can only be displayed in debug mode

For Xcode 13 (or up), it seems Apple has removed the *Debug Preview* feature. To see the console message, you have to run the app on a simulator.

Customizing the Button's Font and Background

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

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

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

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

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

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

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

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

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

Figure 6. Adding padding to the button

The Order of Modifiers is Important

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

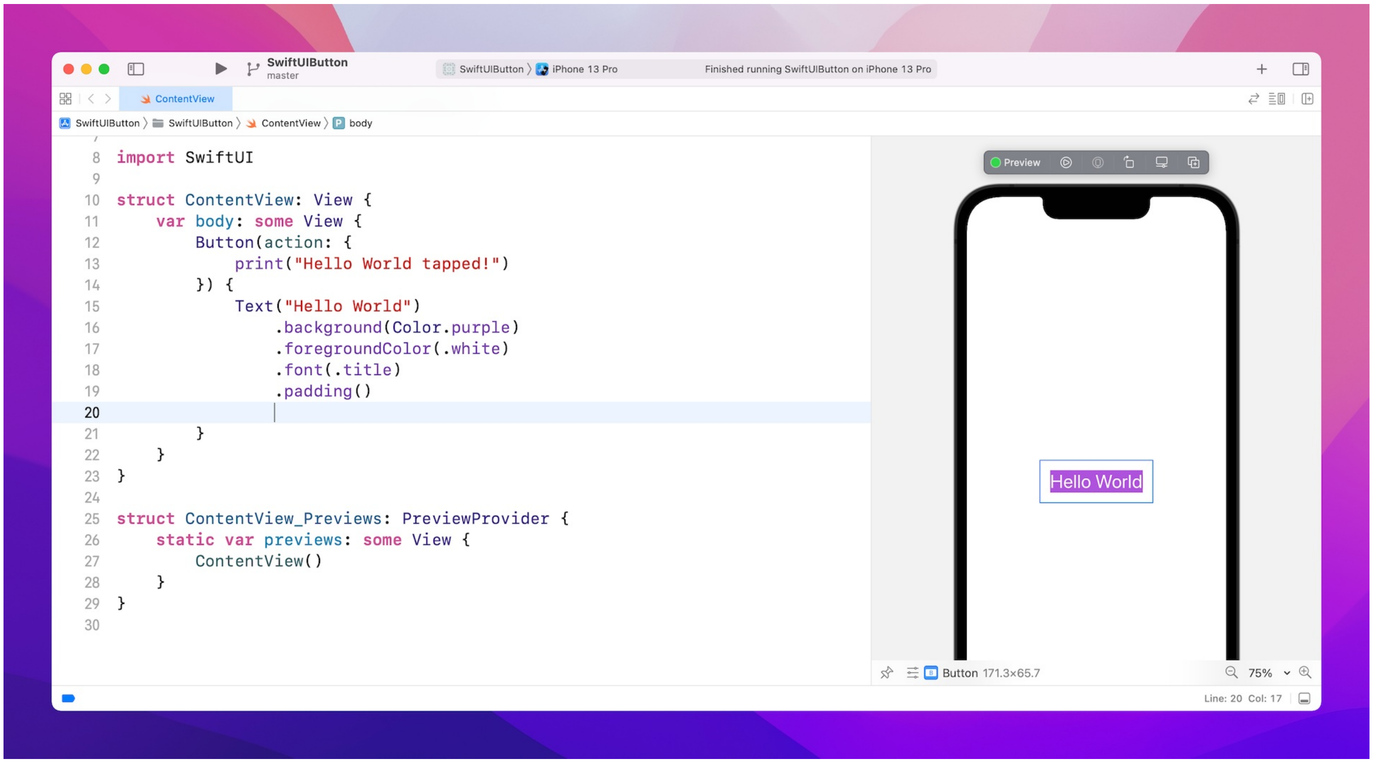


Figure 7. Placing the padding modifier after the background modifier

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

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

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

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

Adding Borders to the Button

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

Figure 8. A button with borders

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

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

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

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

Figure 9. A button with both background and border

Okay, here is the solution:

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

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

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

Figure 10. A button with a rounded border

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

```
.cornerRadius(40)
```

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

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.

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

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.

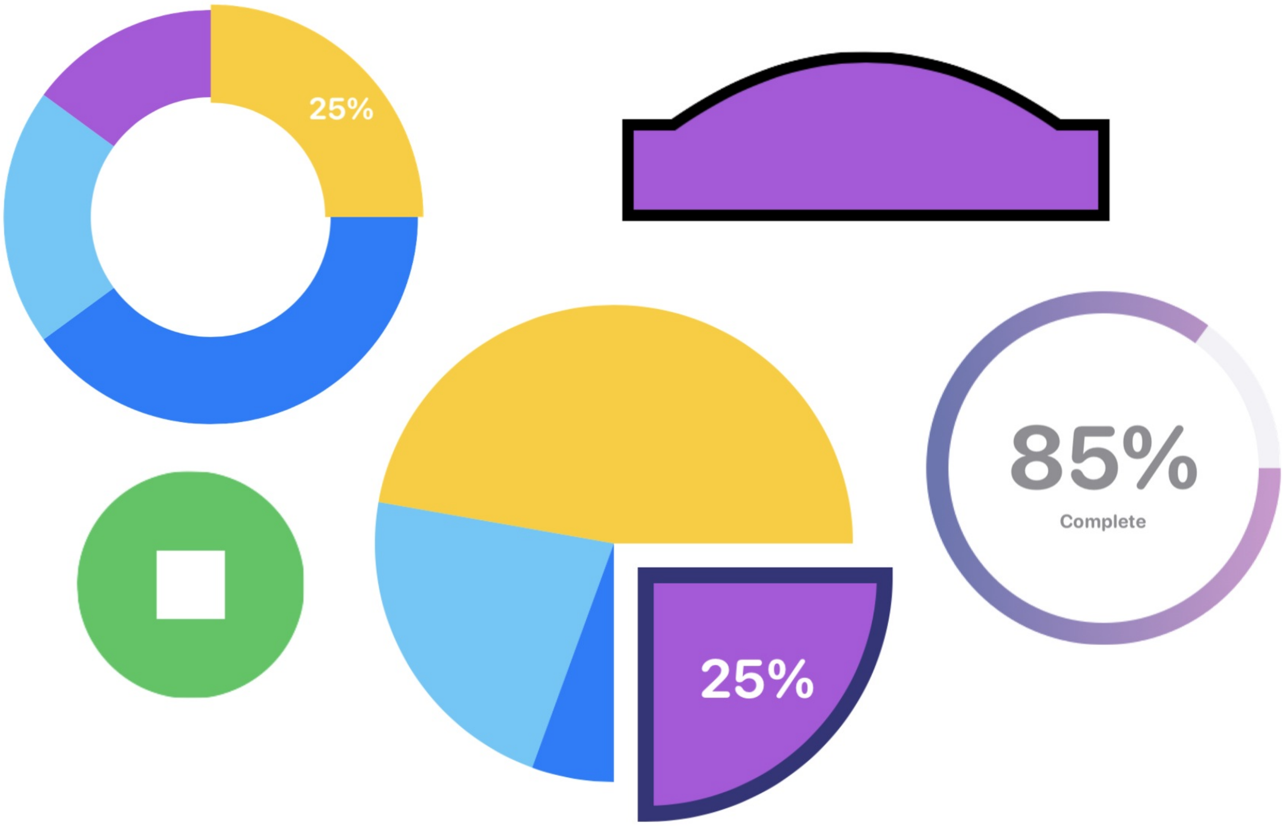


Figure 1. Sample shapes and charts

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

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.

Below are some of the sample animations you will learn to build. *To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.*

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

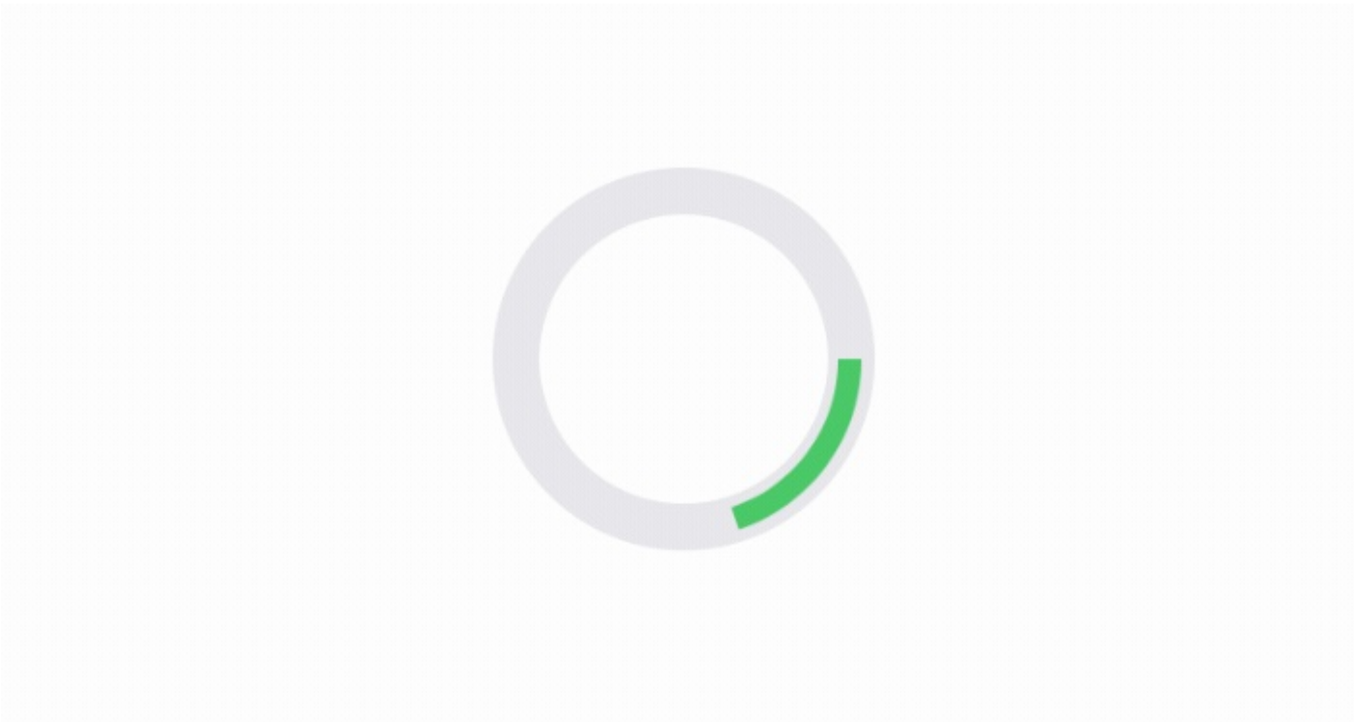


Figure 4. A sample loading indicator

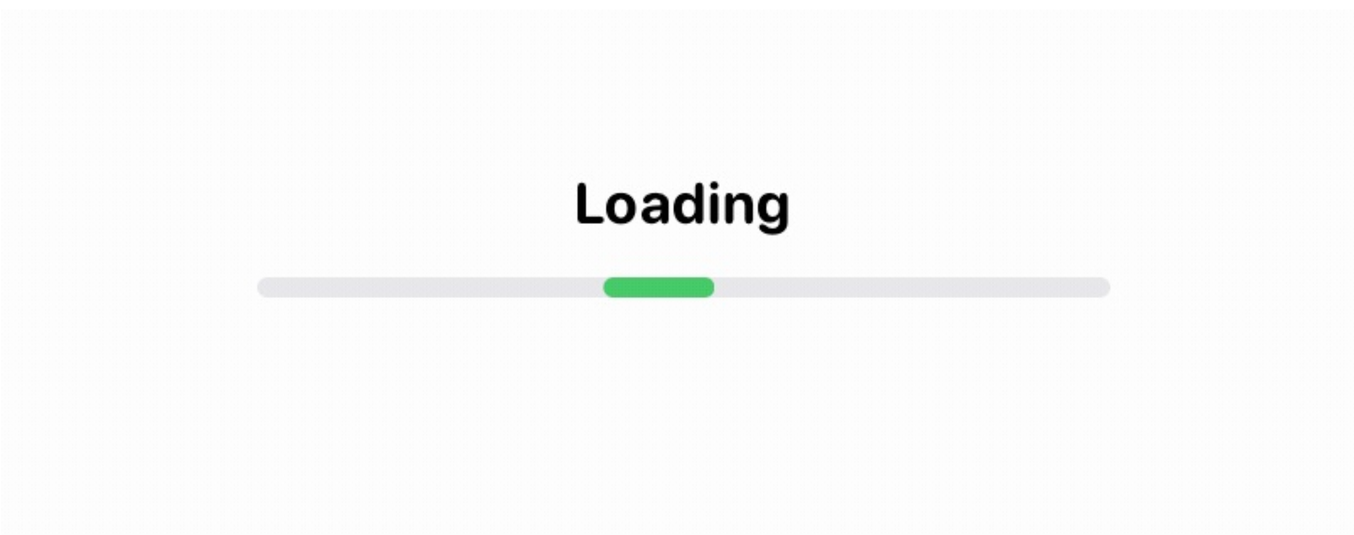


Figure 5. Another example of the loading indicator

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.

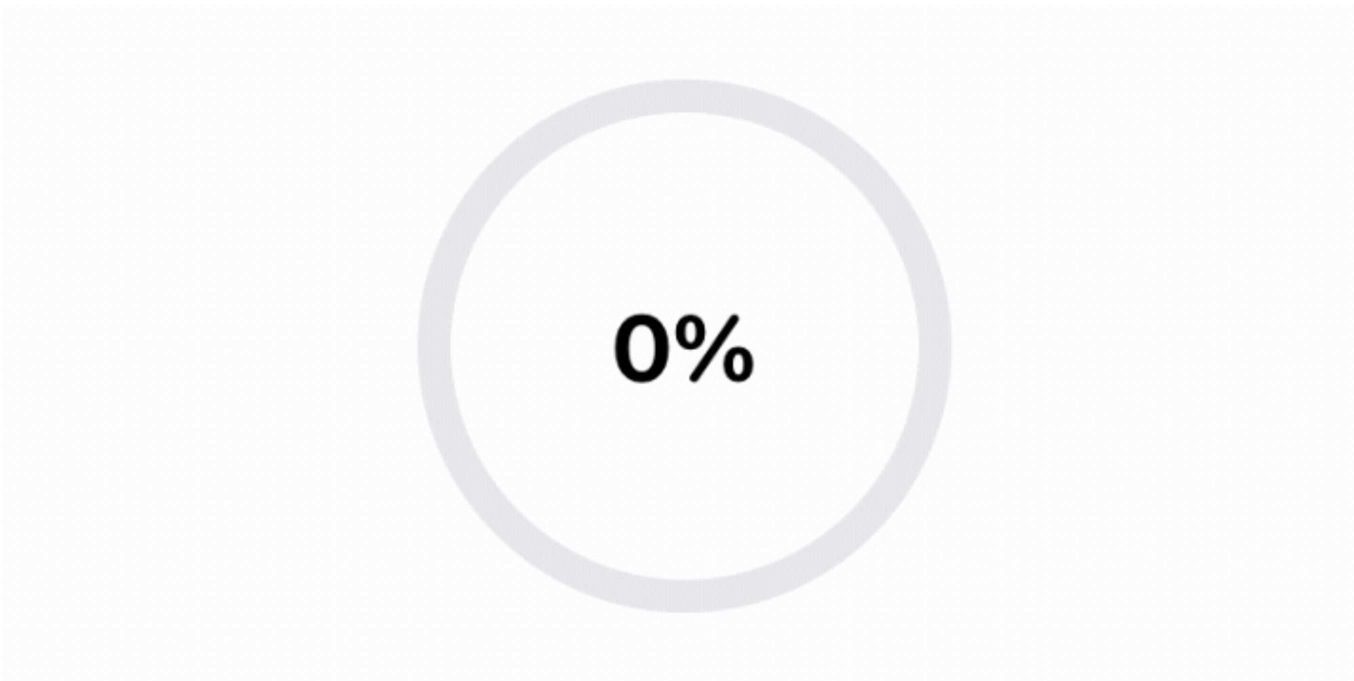


Figure 7. A progress indicator

Delaying an Animation

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

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.

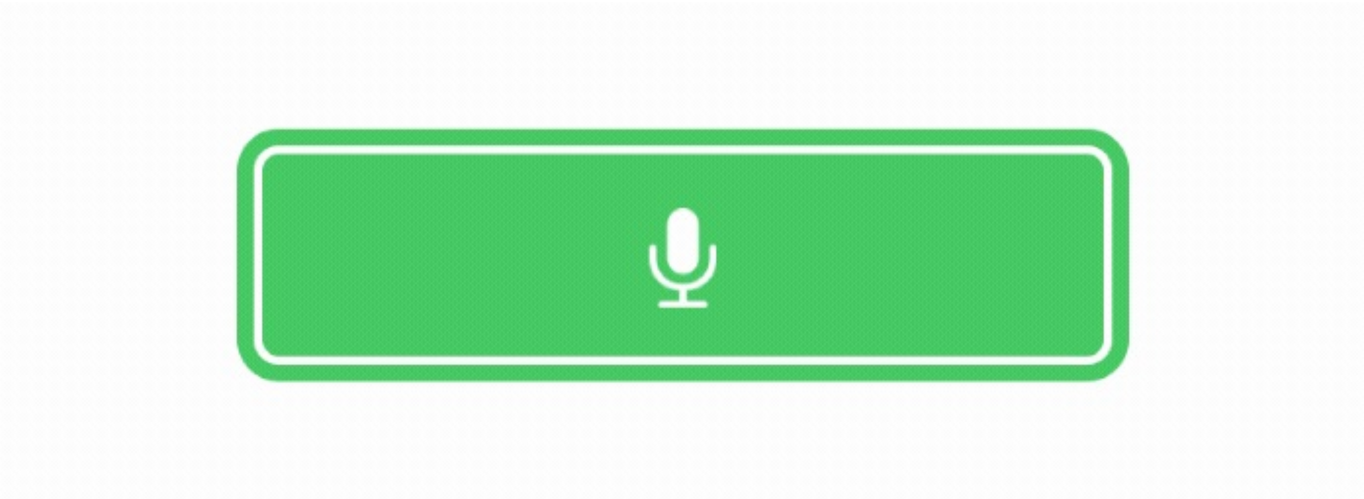


Figure 9. Morphing a rectangle into a circle

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.

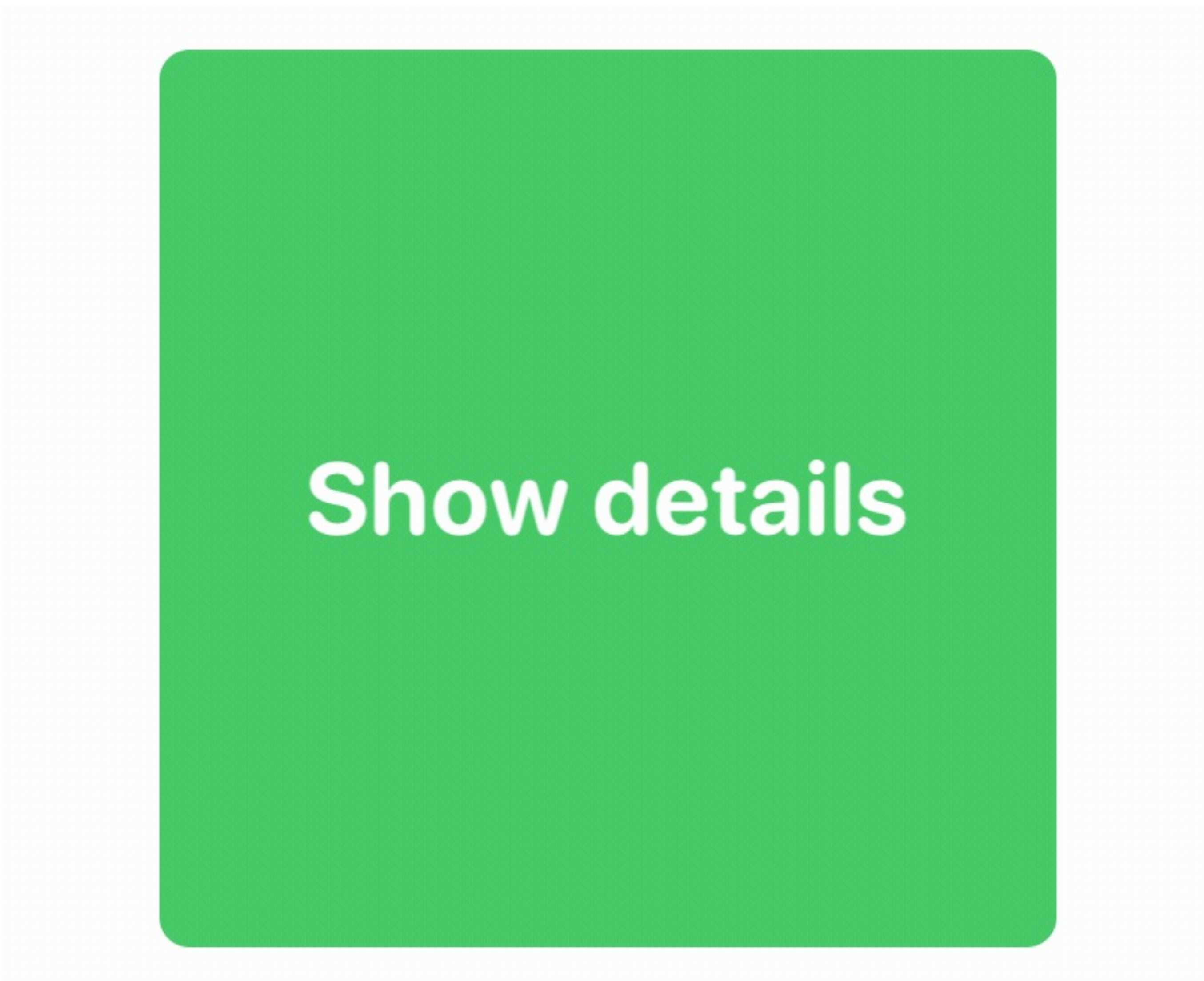


Figure 10. A sample transition created using SwiftUI

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.

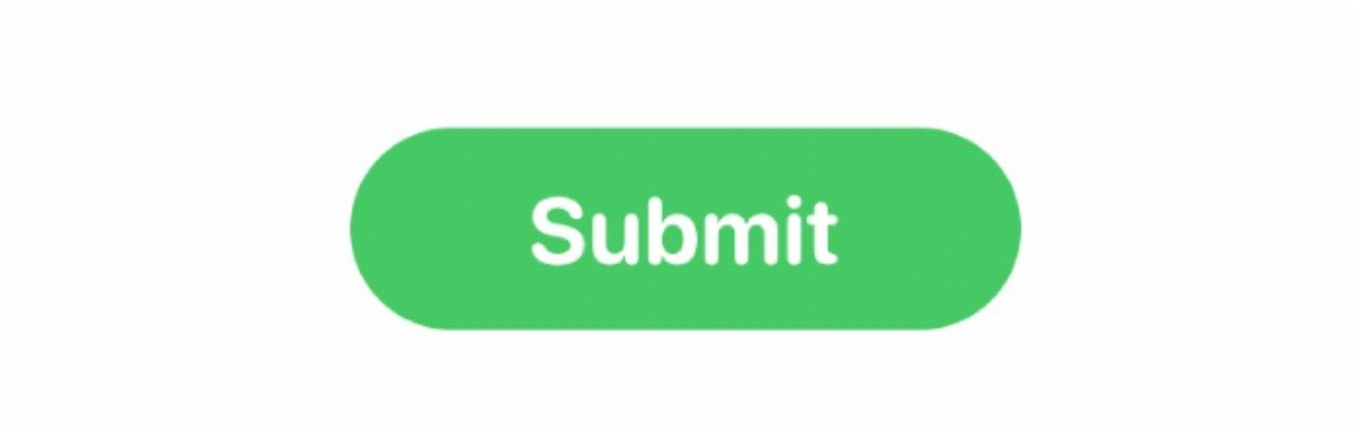


Figure 16. A fancy button

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.

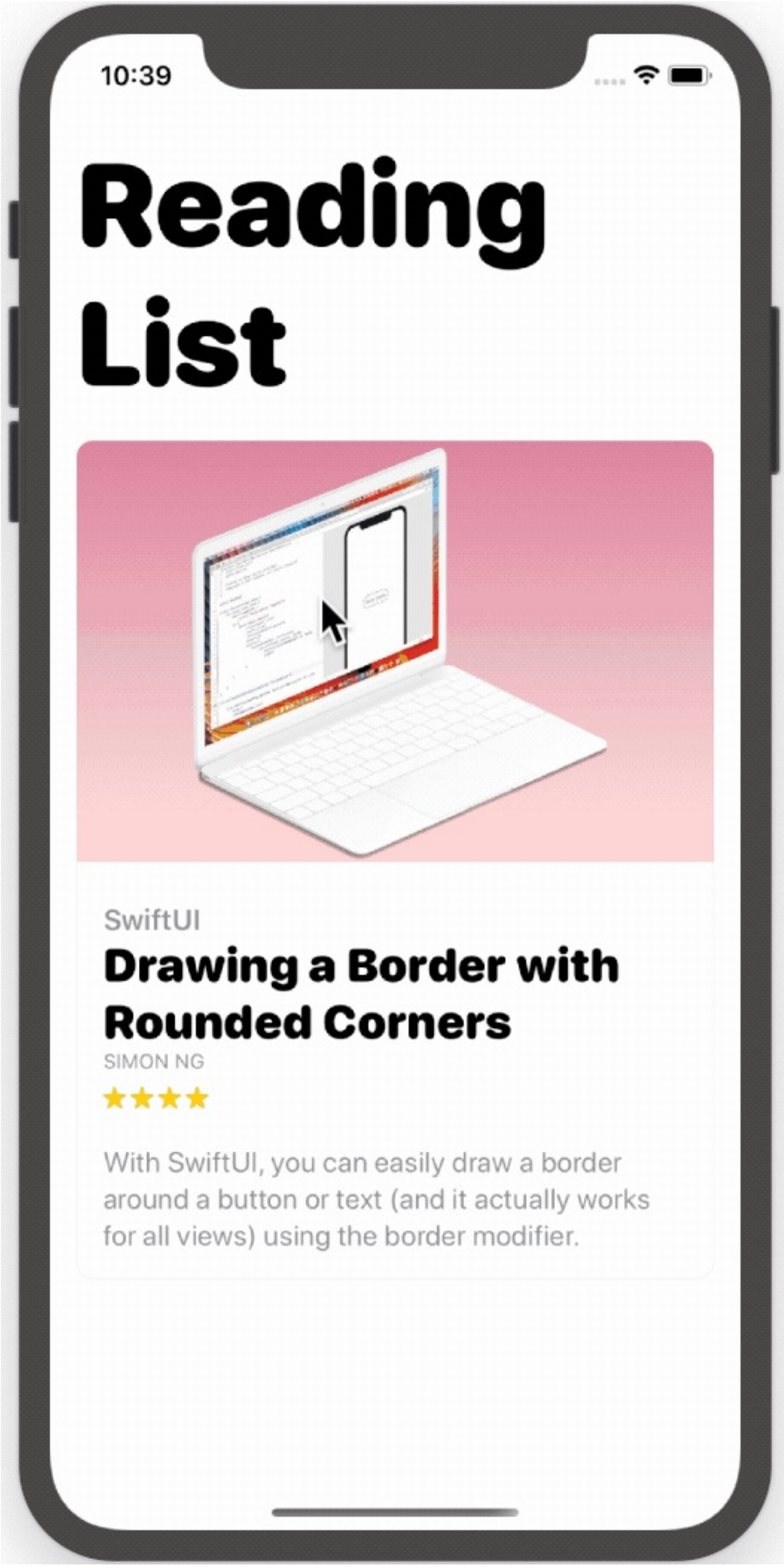

10:39
Reading List
SwiftUI
Drawing a Border with Rounded Corners
SIMON NG
With SwiftUI, you can easily draw a border around a button or text (and it actually works for all views) using the border modifier.

Figure 17. Animated view transition

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

Chapter 10
Understanding Dynamic List, ForEach and Identifiable

In UKit, 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.

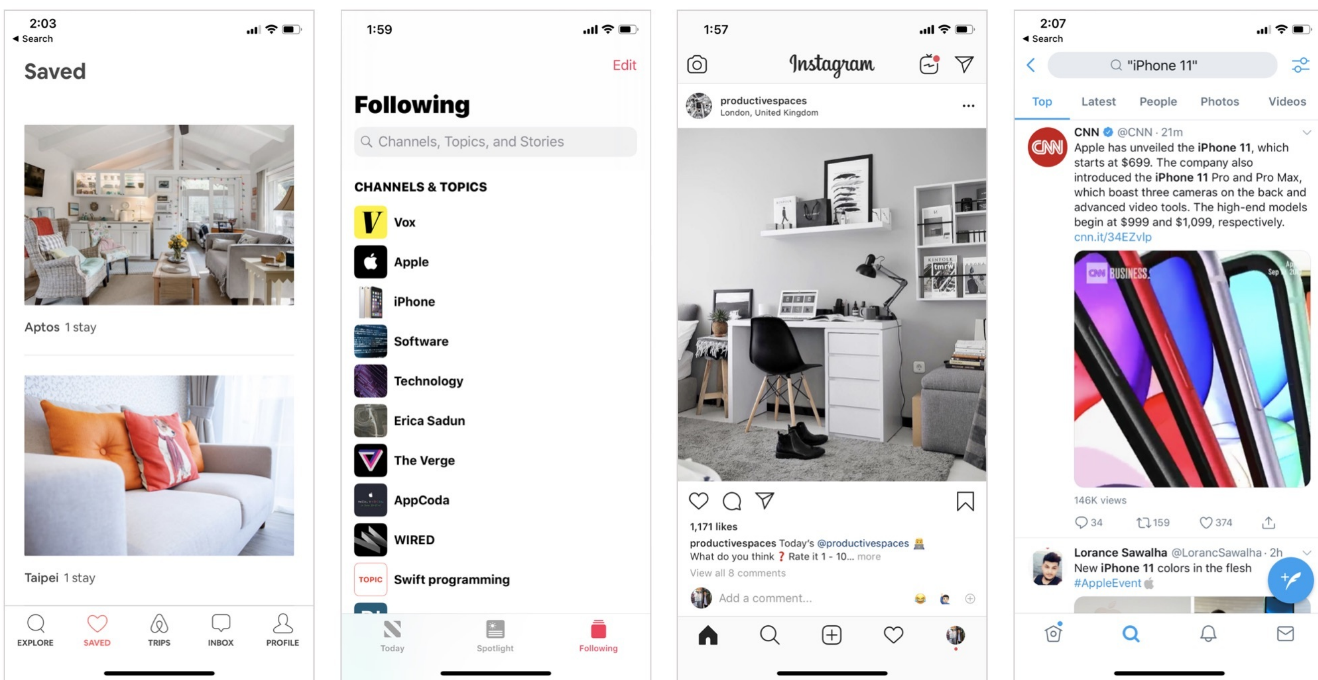


Figure 1. Sample list views

Instead of using a table view, we use `List` in Swift UI to present rows of data. If you've built a table view with UIKit before, you know it'll take you a bit of work to implement a simple table view. It'll take even more 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.

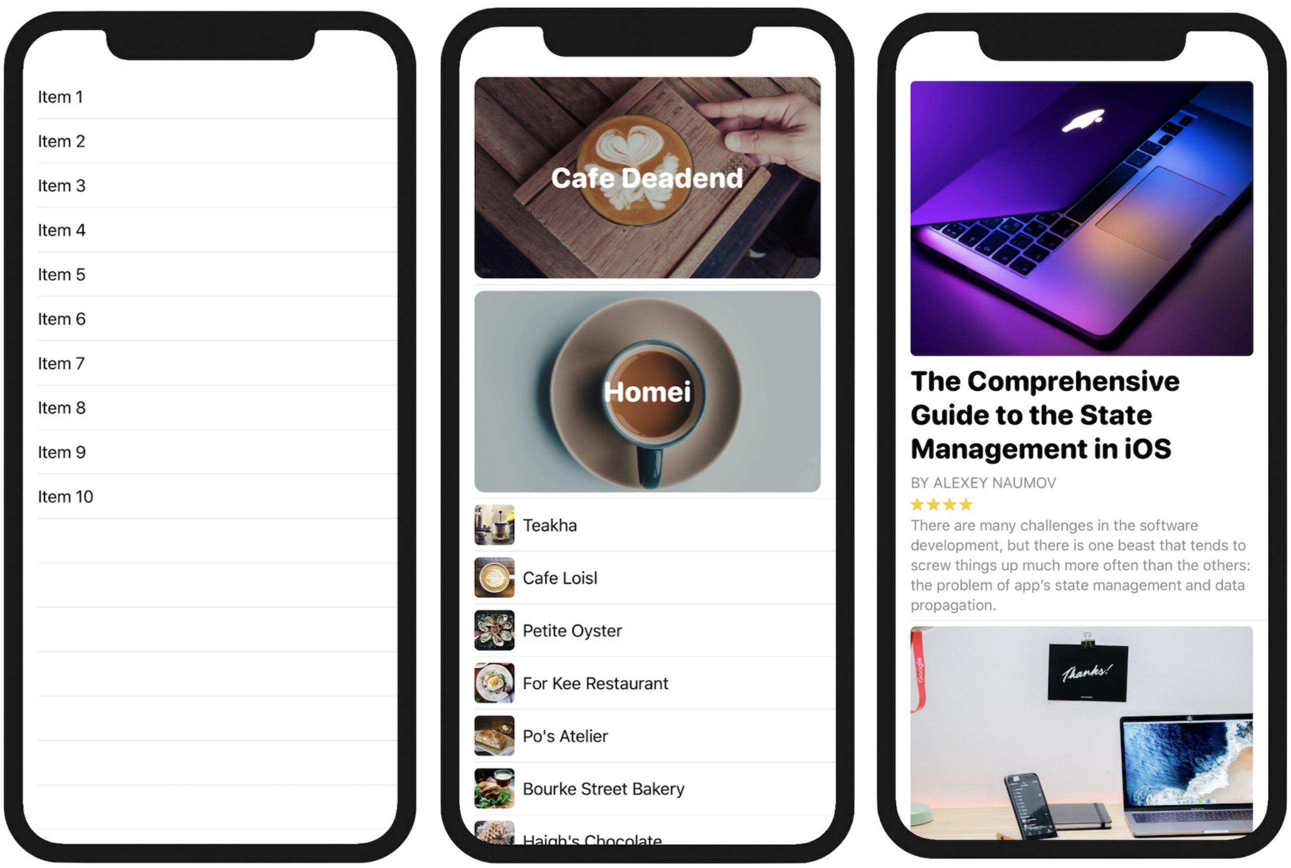


Figure 2. Building a simple and complex list

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

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.

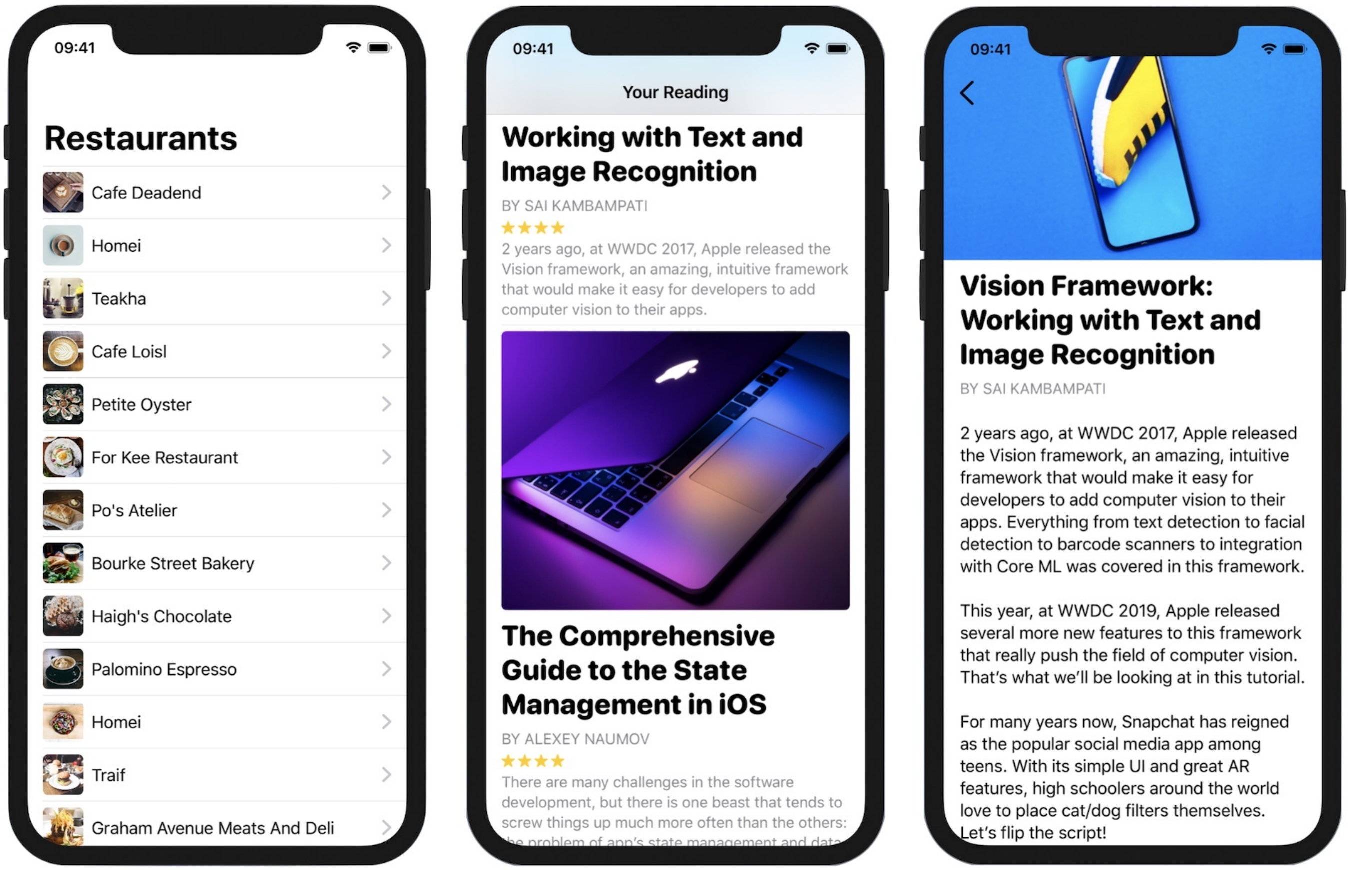


Figure 1. Sample navigation interface for our demo projects

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

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.

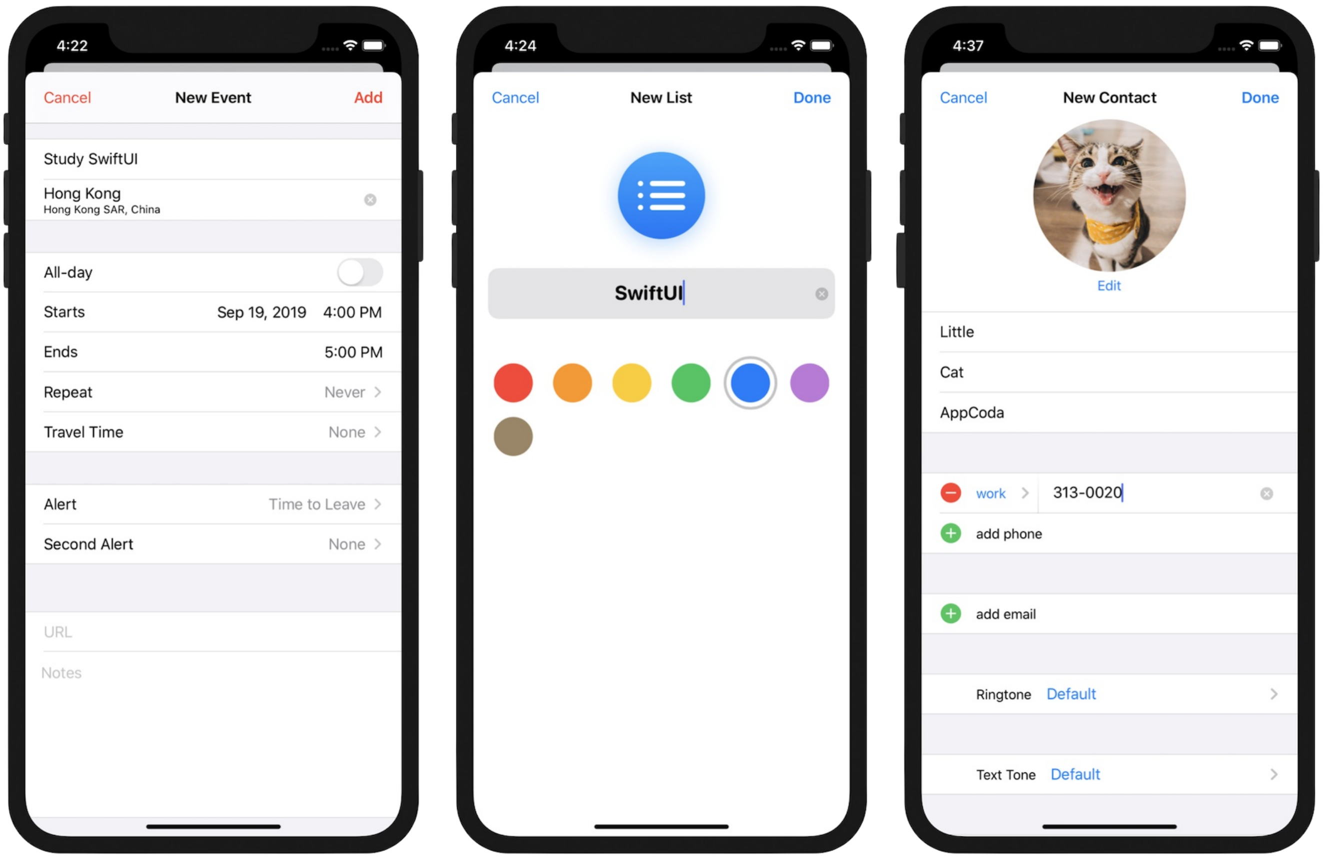


Figure 1. Sample modal views in Calendar, Reminders, and Contact apps

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.

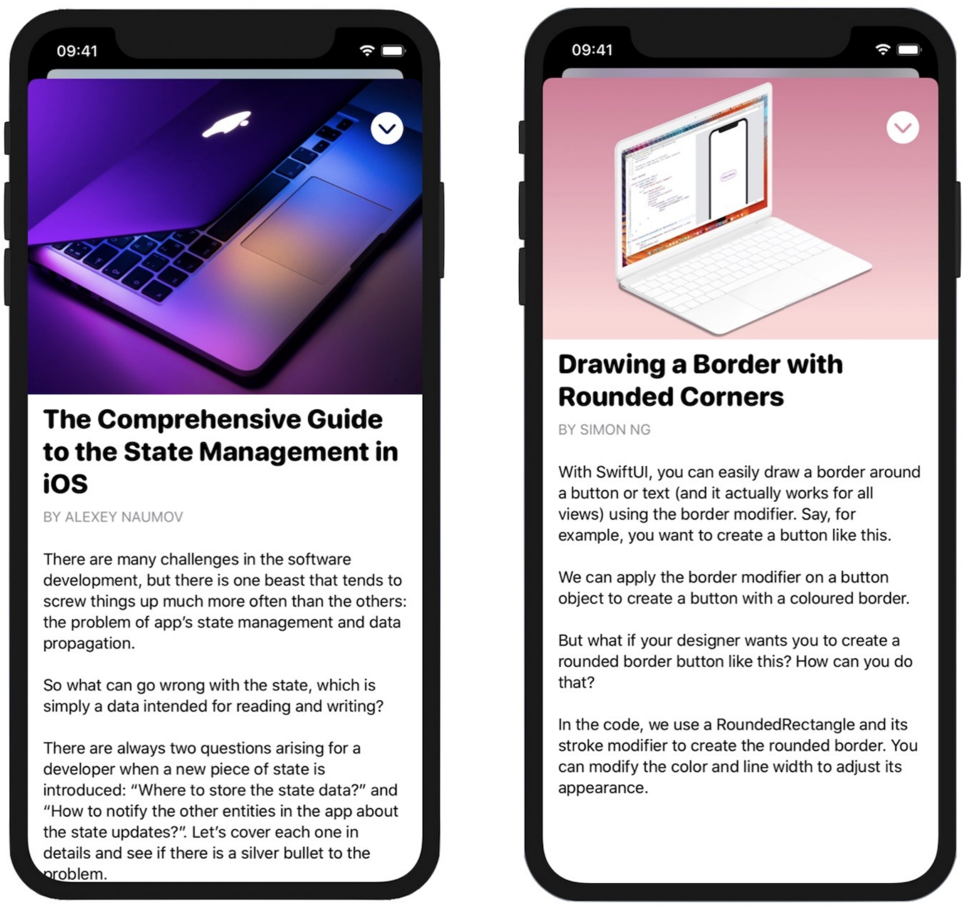


Figure 2. Presenting the detail screen using modal views

We got a lot to discuss in this chapter. Let's get started.

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.

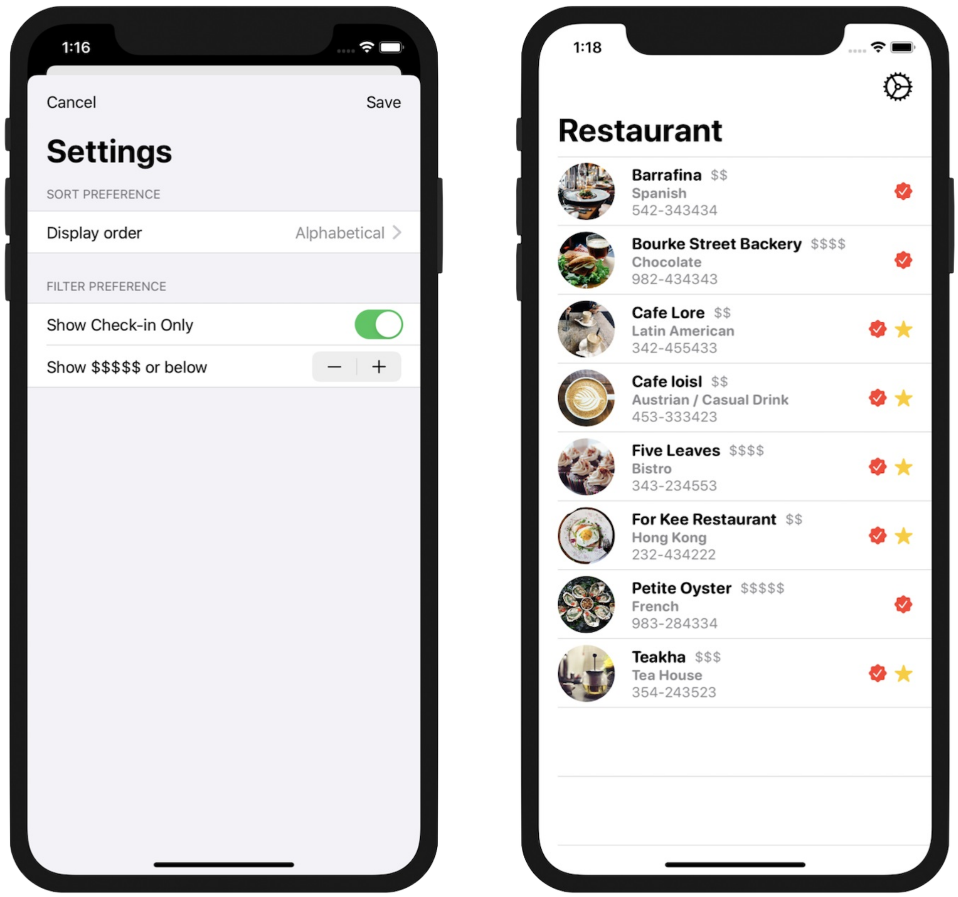


Figure 1. Building a Setting screen

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`.

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

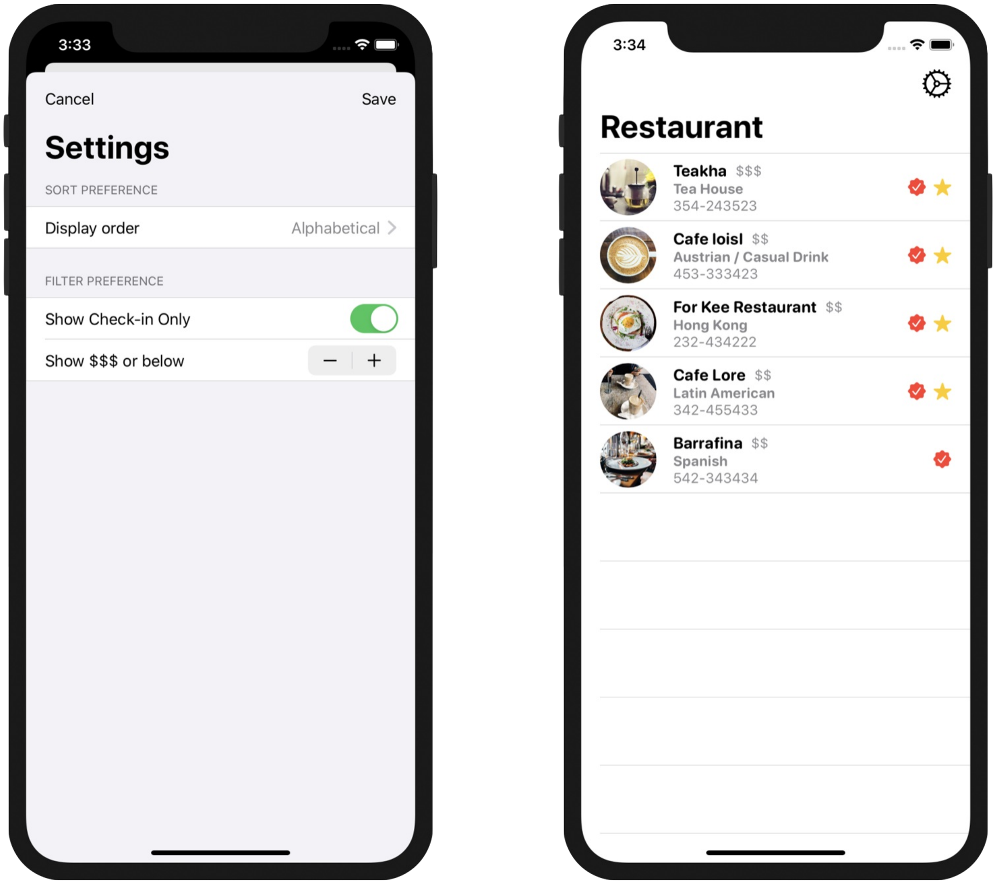


Figure 4. The list view now refreshes its items when you change the filter preference

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

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.

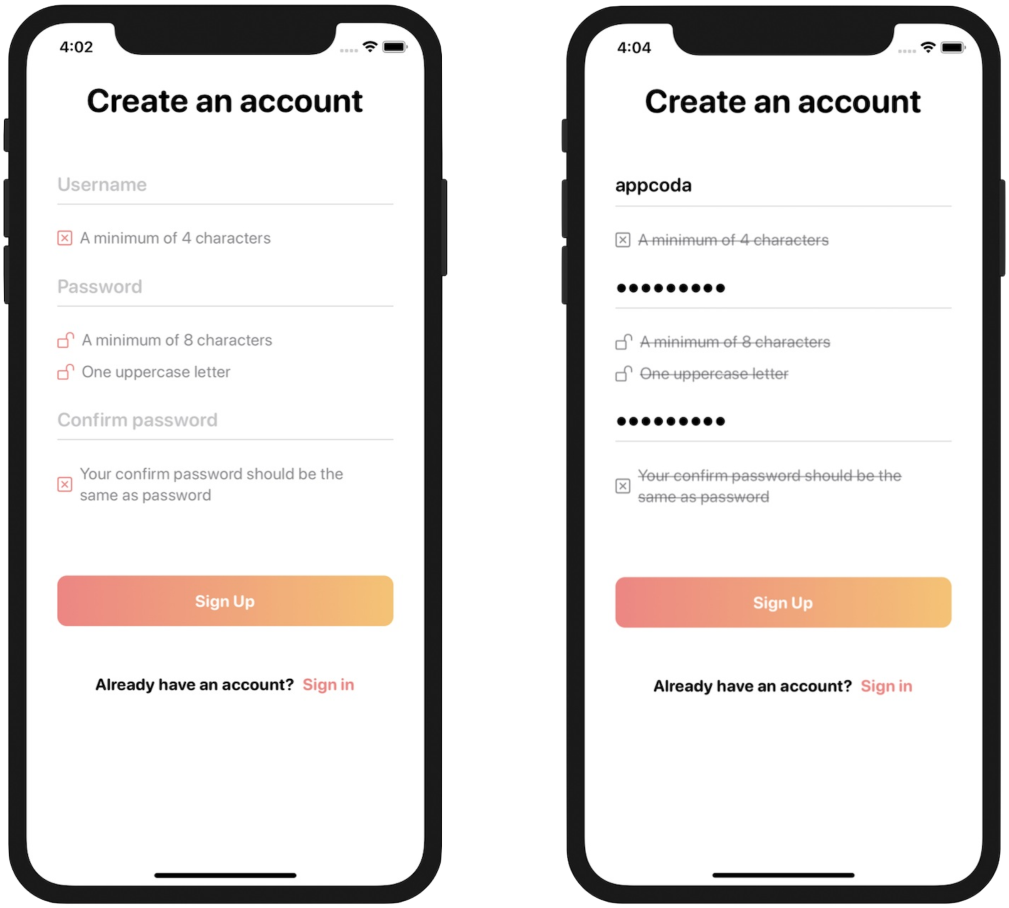


Figure 1. User registration demo

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.

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

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

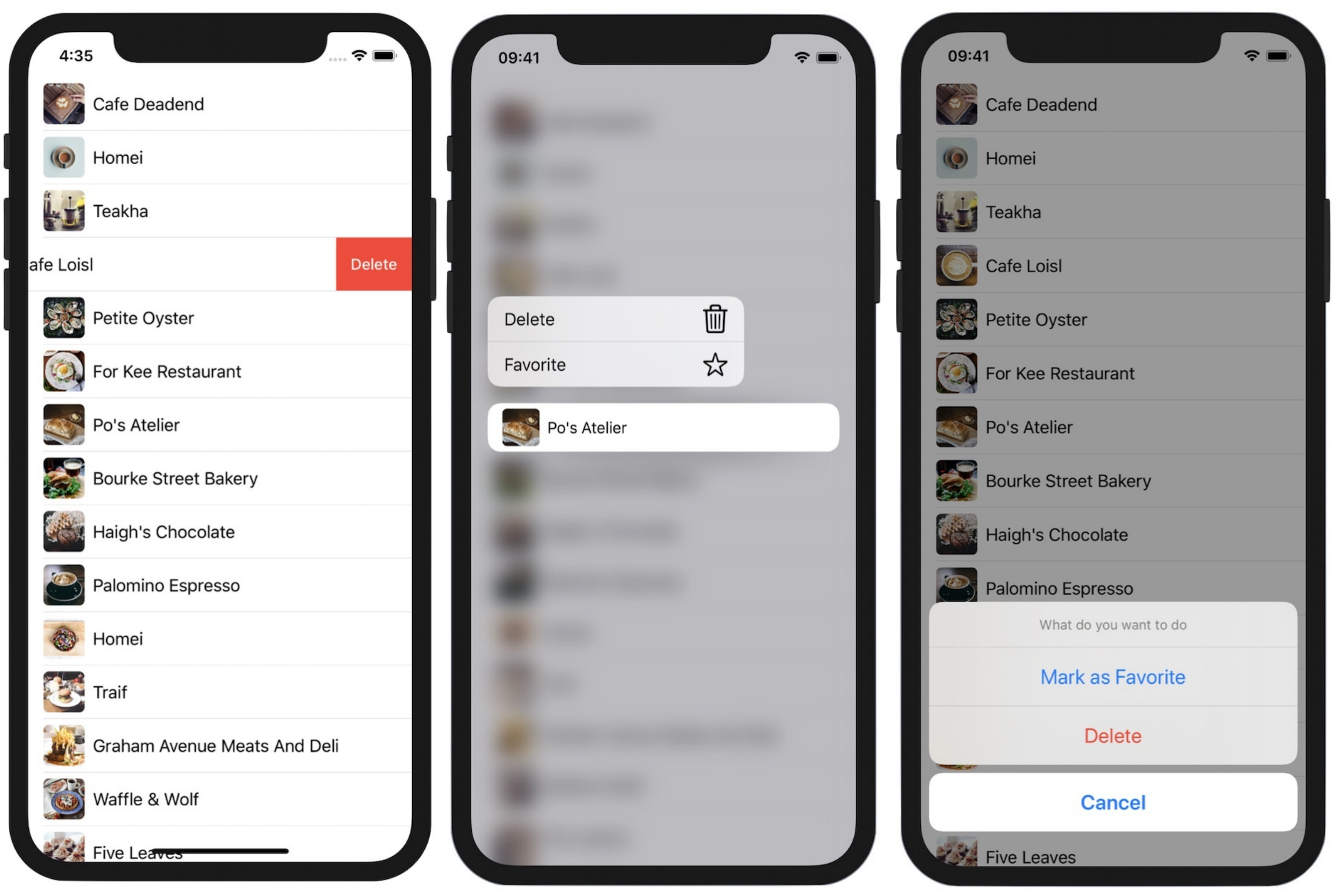


Figure 1. Swipe to delete (left), context menu, and action sheet (right)

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.

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

Chapter 17
Understanding Gestures

In earlier chapters, you got a taste of building gestures with SwiftUI. We used the `onTapGesture` modifier to handle a user's touch and provide a corresponding response. In this chapter, let's dive deeper to see how we work with various types of gestures in SwiftUI.

The framework provides several built-in gestures such as the tap gesture we have used before. additionally, *DragGesture*, *MagnificationGesture*, and *LongPressGesture* are some of the ready-to-use gestures. We will be looking at a couple of them and seeing how to work with gestures in SwiftUI. On top of that, you will learn how to build a generic view that supports the drag gesture.

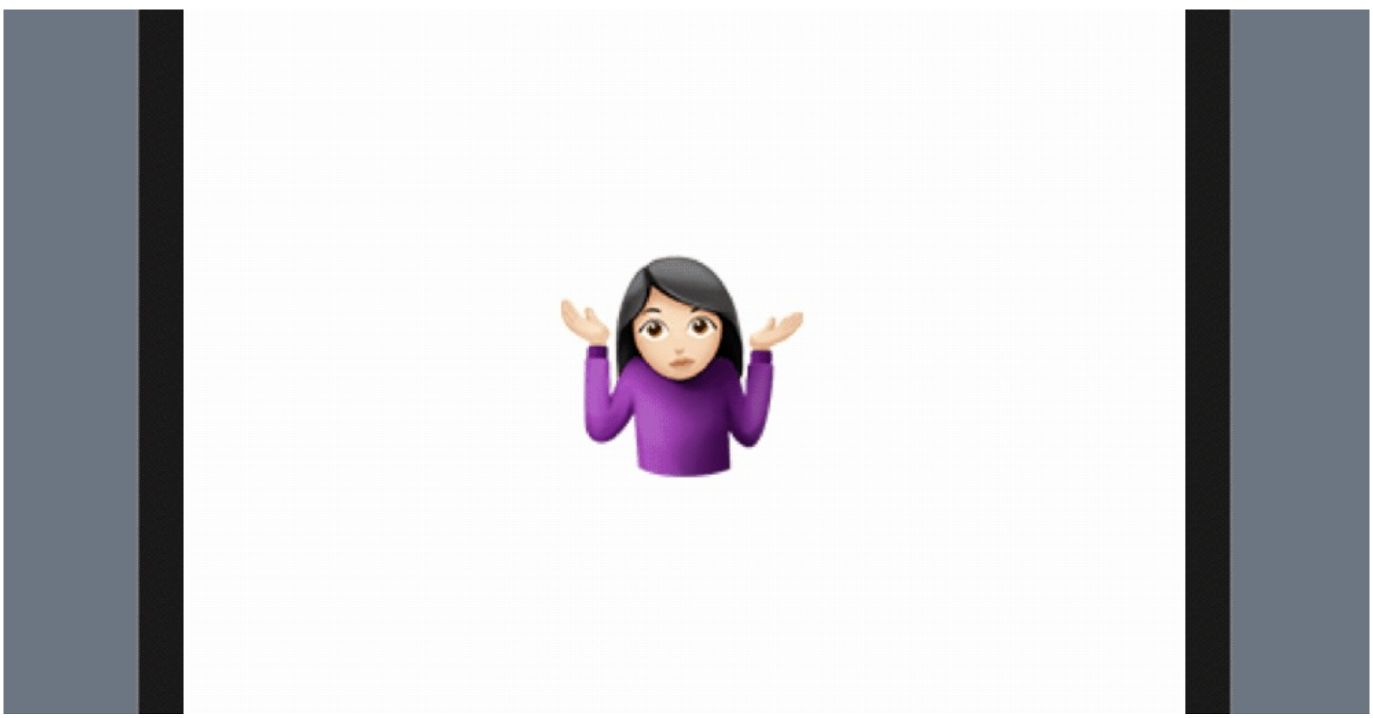


Figure 1. A demo showing the draggable view

Using the Gesture Modifier

To recognize a particular gesture using the SwiftUI framework, all you need to do is attach a gesture recognizer to a view using the `.gesture` modifier. Here is a sample code snippet which attaches the `TapGesture` using the `.gesture` modifier:

```
var body: some View {
    Image(systemName: "star.circle.fill")
        .font(.system(size: 200))
        .foregroundColor(.green)
        .gesture(
            TapGesture()
                .onEnded({
                    print("Tapped!")
                })
        )
}
```

If you want to try out the code, create a new project using the *App* template and make sure you select *SwiftUI* for the *Interface* option. Then paste the code in `ContentView.swift`.

By modifying the code above a bit and introducing a state variable, we can create a simple scale animation when the star image is tapped. Here is the updated code:

```
struct ContentView: View {
    @State private var isPressed = false

    var body: some View {
        Image(systemName: "star.circle.fill")
            .font(.system(size: 200))
            .scaleEffect(isPressed ? 0.5 : 1.0)
            .animation(.easeInOut, value: isPressed)
            .foregroundColor(.green)
            .gesture(
                TapGesture()
                    .onEnded({
                        self.isPressed.toggle()
                    })
            )
    }
}
```

When you run the code in the canvas or simulator, you should see a scaling effect. This is how you use the `.gesture` modifier to detect and respond to certain touch events. If you forget how animation works, please go back to read chapter 9.

Figure 2. A simple scaling effect

Using Long Press Gesture

One of the built-in gesture is `LongPressGesture`. This gesture recognizer allows you to detect a long-press event. For example, if you want to resize the star image only when the user presses and holds it for at least 1 second, you can use the `LongPressGesture` to detect the touch event.

Modify the code in the `.gesture` modifier like this to implement the `LongPressGesture`:

```
.gesture(
    LongPressGesture(minimumDuration: 1.0)
        .onEnded({ _ in
            self.isPressed.toggle()
        })
)
```

Run the project in the preview canvas to see what it does. Now you have to press and hold the star image for at least a second before it toggles its size.

The @GestureState Property Wrapper

When you press and hold the star image, the image doesn't give the user any response until the long press event is detected. Obviously, there is something we can do to improve the user experience. What I want to do is to give the user immediate feedback when he/she taps the image. Any kind of feedback will help to improve the situation. Let's dim the image a bit when the user taps it. This just lets the user know that our app captures the touch and is doing work. Figure 3 illustrates how the animation works.

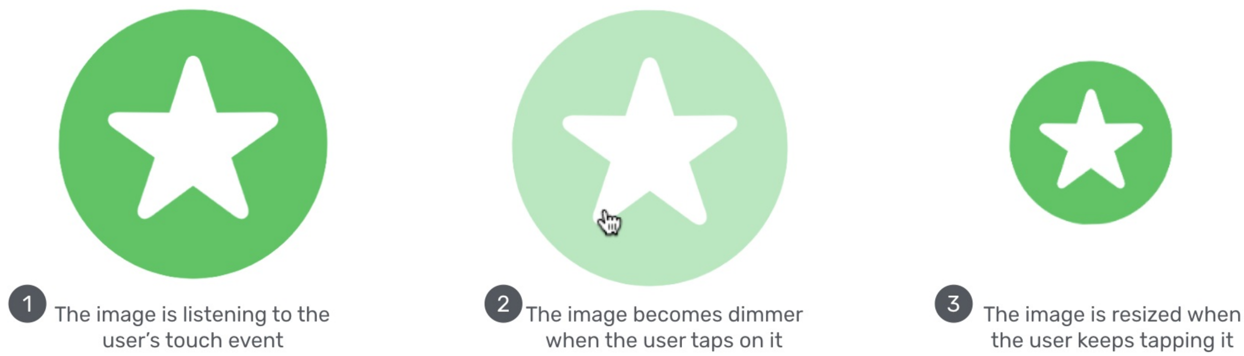


Figure 3. Applying a dimming effect when the image is tapped

To implement the animation, you need to keep track of the state of gestures. During the performance of the long press gesture, we have to differentiate between tap and long press events. So, how do we do that?

SwiftUI provides a property wrapper called `@GestureState` which conveniently tracks the state change of a gesture and lets developers decide the corresponding action. To implement the animation we just described, we can declare a property using `@GestureState` like this:

```
@GestureState private var longPressTap = false
```

This gesture state variable indicates whether a tap event is detected during the performance of the long press gesture. Once you have the variable defined, you can modify the code of the `Image` view like this:

```
Image(systemName: "star.circle.fill")
    .font(.system(size: 200))
    .opacity(longPressTap ? 0.4 : 1.0)
    .scaleEffect(isPressed ? 0.5 : 1.0)
    .animation(.easeInOut, value: isPressed)
    .foregroundColor(.green)
    .gesture(
        LongPressGesture(minimumDuration: 1.0)
            .updating($longPressTap, body: { (currentState, state, transaction) in
                state = currentState
            })
            .onEnded({ _ in
                self.isPressed.toggle()
            })
    )
```

We only made a couple of changes in the code above. First, we added the `.opacity` modifier. When the tap event is detected, we set the opacity value to `0.4` so that the image becomes dimmer.

Second, we added the `updating` method of the `LongPressGesture`. During the performance of the long press gesture, this method will be called. It accepts three parameters: *value*, *state*, and *transaction*:

- The *value* parameter is the current state of the gesture. This value varies from gesture to gesture, but for the long press gesture, a `true` value indicates that a tap is detected.
- The *state* parameter is actually an in-out parameter that lets you update the value of the `longPressTap` property. In the code above, we set the value of `state` to `currentState`. In other words, the `longPressTap` property always keeps track of the latest state of the long press gesture.
- The `transaction` parameter stores the context of the current state-processing update.

After you make the code change, run the project in the preview canvas to test it. The image immediately becomes dimmer when you tap it. Keep holding it for one second and then the image resizes itself.

The opacity of the image is automatically reset to normal when the user releases the long press. Do you wonder why? This is an advantage of `@GestureState`. When the gesture ends, it automatically sets the value of the gesture state property to its initial value, `false` in our case.

Using Drag Gesture

Now that you understand how to use the `.gesture` modifier and `@GestureState`, let's look into another common gesture: *Drag*. What we are going to do is modify the existing code to support the drag gesture, allowing a user to drag the star image to move it around.

Replace the `ContentView` struct like this:

```
struct ContentView: View {
    @GestureState private var dragOffset = CGSize.zero

    var body: some View {
        Image(systemName: "star.circle.fill")
            .font(.system(size: 100))
            .offset(x: dragOffset.width, y: dragOffset.height)
            .animation(.easeInOut, value: dragOffset)
            .foregroundColor(.green)
            .gesture(
                DragGesture()
                    .updating($dragOffset, body: { (value, state, transaction) in

                        state = value.translation
                    })
            )
    }
}
```

To recognize a drag gesture, you initialize a `DragGesture` instance and listen for an update. In the `update` function, we pass a gesture state property to keep track of the drag event. Similar to the long press gesture, the closure of the `update` function accepts three parameters. In this case, the *value* parameter stores the current data of the drag including the translation. This is why we set the `state` variable, which is actually the `dragOffset`, to `value.translation`.

Run the project in the preview canvas, you can drag the image around. But, when you release it, the image returns to its original position.

Do you know why the image returns to its starting point? As explained in the previous section, one advantage of using `@GestureState` is that it resets the value of the property to its original value when the gesture ends. Therefore, when you end the drag and release the press, the `dragOffset` is reset to `.zero`, which is its original position.

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

Since the `@GestureState` property wrapper will reset the property to its original value, we need another state property to save the final position. Therefore, let's declare a new state property like this:

```
@State private var position = CGSize.zero
```

Next, update the `body` variable like this:

```
var body: some View {
    Image(systemName: "star.circle.fill")
        .font(.system(size: 100))
        .offset(x: position.width + dragOffset.width, y: position.height + dragOff
set.height)
        .animation(.easeInOut, value: dragOffset)
        .foregroundColor(.green)
        .gesture(
            DragGesture()
                .updating($dragOffset, body: { (value, state, transaction) in

                    state = value.translation
                })
                .onEnded({ (value) in
                    self.position.height += value.translation.height
                    self.position.width += value.translation.width
                })
        )
}
```

We have made a couple of changes to the code:

1. We implemented the `onEnded` function which is called when the drag gesture ends. In the closure, we compute the new position of the image by adding the drag offset.
2. The `.offset` modifier was also updated, such that we take the current position into account.

Now when you run the project and drag the image, the image stays where it is even after the drag ends.

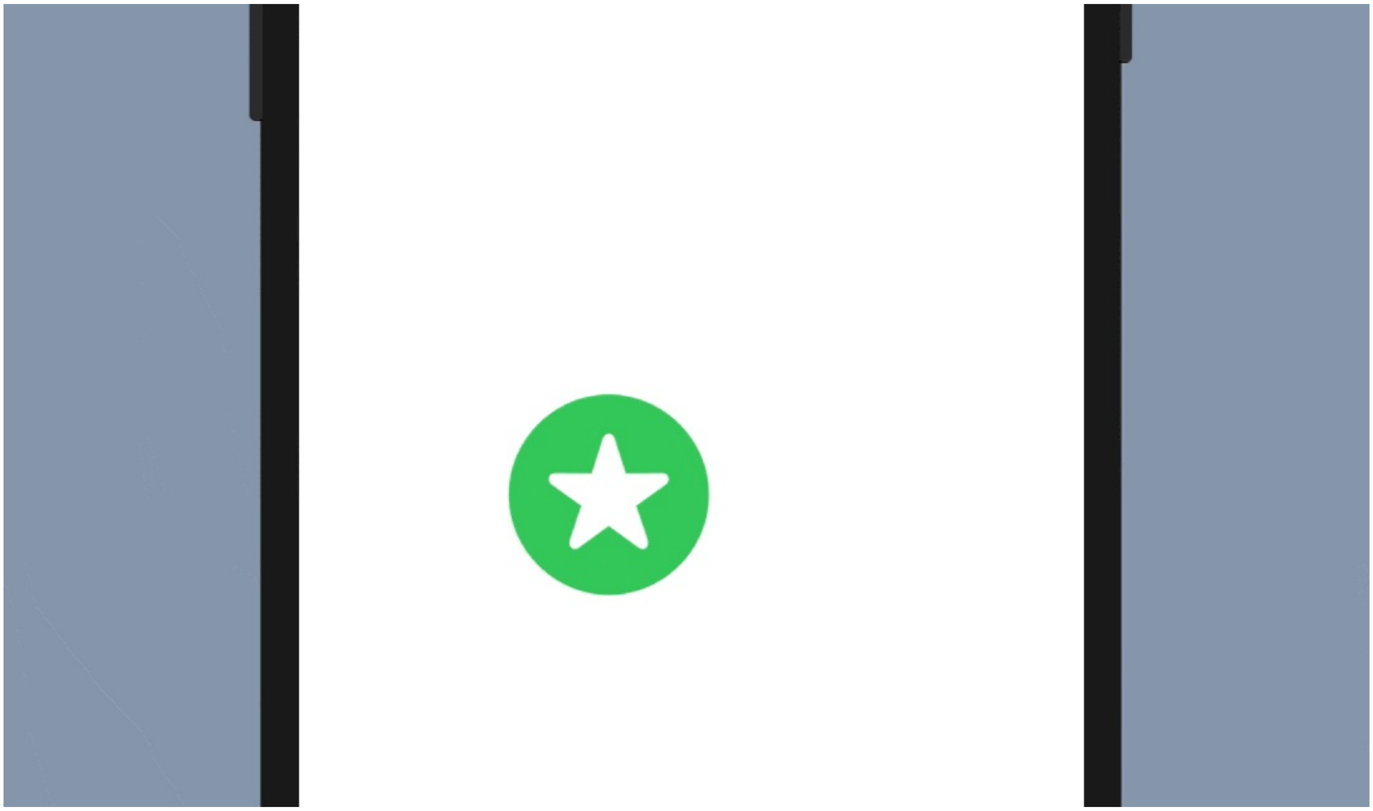


Figure 4. Drag the image around

Combining Gestures

In some cases, you need to use multiple gesture recognizers in the same view. Let's say, we want the user to press and hold the image before starting the drag, we have to combine both long press and drag gestures. SwiftUI allows you to easily combine gestures to perform more complex interactions. It provides three gesture composition types including *simultaneous*, *sequenced*, and *exclusive*.

When you need to detect multiple gestures at the same time, you use the *simultaneous* composition type. When you combine gestures using the *exclusive* composition type, SwiftUI recognizes all the gestures you specify but it will ignore the rest when one of the gestures is detected.

As the name suggests, if you combine multiple gestures using the *sequenced* composition type, SwiftUI recognizes the gestures in a specific order. This is the type of the composition that we will use to sequence the long press and drag gestures.

To work with multiple gestures, you update the code like this:

```
struct ContentView: View {
    // For long press gesture
    @GestureState private var isPressed = false

    // For drag gesture
    @GestureState private var dragOffset = CGSize.zero
    @State private var position = CGSize.zero

    var body: some View {
        Image(systemName: "star.circle.fill")
            .font(.system(size: 100))
            .opacity(isPressed ? 0.5 : 1.0)
            .offset(x: position.width + dragOffset.width, y: position.height + dra
gOffset.height)
            .animation(.easeInOut, value: dragOffset)
            .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 some parts of the code snippet because we are combining the long press gesture that we have built with the drag gesture.

Let me explain the code in the `.gesture` modifier line by line. We require the user to press and hold the image for at least one second before he/she can begin the dragging. So, we start by creating the `LongPressGesture`. Similar to what we have implemented before, we have a `isPressed` gesture state property. When someone taps the image, we will alter the opacity of the image.

The `sequenced` keyword is how we link the long press and drag gestures together. We tell SwiftUI that the `LongPressGesture` should happen before the `DragGesture`.

The code in both `updating` and `onEnded` functions looks pretty similar, but the `value` parameter now actually contains two gestures (i.e. long press and drag). We have the `switch` statement to differentiate between the gestures. You can use the `.first` and `.second` cases to find out which gesture to handle. Since the long press gesture should be recognized before the drag gesture, the *first* gesture here is the long press gesture. In the code, we do nothing but just print the *Tapping* message for your reference.

When the long press is confirmed, we will reach the `.second` case. Here, we pick up the *drag* data and update the `dragOffset` with the corresponding translation.

When the drag ends, the `onEnded` function will be called. Similarly, we update the final position by figuring out the drag data (i.e. `.second` case).

Now you're ready to test the gesture combination. Run the app in the preview canvas using the debug preview, so you can see the message in the console. You can't drag the image until holding the star image for at least one second.

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

Refactoring the Code Using Enum

A better way to organize the drag state is by using Enum. This allows you to combine the `isPressed` and `dragOffset` state into a single property. Let's declare an enumeration called `DragState`.

```
enum DragState {
    case inactive
    case pressing
    case dragging(translation: CGSize)

    var translation: CGSize {
        switch self {
        case .inactive, .pressing:
            return .zero
        case .dragging(let translation):
            return translation
        }
    }

    var isPressing: Bool {
        switch self {
        case .pressing, .dragging:
            return true
        case .inactive:
            return false
        }
    }
}
```

We have three states here: *inactive*, *pressing*, and *dragging*. These states are good enough to represent the states during the performance of the long press and drag gestures. For the *dragging* state, we associate it with the translation of the drag.

With the `DragState` enum, we can modify the original code like this:

```
struct ContentView: View {
    @GestureState private var dragState = DragState.inactive
    @State private var position = CGSize.zero

    var body: some View {
        Image(systemName: "star.circle.fill")
            .font(.system(size: 100))
            .opacity(dragState.isPressing ? 0.5 : 1.0)
            .offset(x: position.width + dragState.translation.width, y: position.h
eight + dragState.translation.height)
            .animation(.easeInOut, value: dragState.translation)
            .foregroundColor(.green)
            .gesture(
                LongPressGesture(minimumDuration: 1.0)
                .sequenced(before: DragGesture())
                .updating($dragState, body: { (value, state, transaction) in

                    switch value {
                    case .first(true):
                        state = .pressing
                    case .second(true, let drag):
                        state = .dragging(translation: drag?.translation ?? .zero)
                    default:
                        break
                    }

                })
                .onEnded({ (value) in

                    guard case .second(true, let drag?) = value else {
                        return
                    }

                    self.position.height += drag.translation.height
                    self.position.width += drag.translation.width
                })
            )
    }
}
```

We now declare a `dragState` property to track the drag state. By default, it's set to `DragState.inactive`. The code is nearly the same as the previous code except that it's modified to work with `dragState` instead of `isPressed` and `dragOffset`. For example, for the `.offset` modifier, we retrieve the drag offset from the associated value of the dragging state.

The result of the code is the same. However, it's always good practice to use Enum to track complicated states of gestures.

Building a Generic Draggable View

So far, we have built a draggable image view. What if we want to build a draggable text view? Or what if we want to create a draggable circle? Should you copy and paste all the code to create the text view or circle?

There is a better way to implement that. Let's see how we can build a generic draggable view.

In the project navigator, right click the `SwiftUIGesture` folder and choose *New File...*. Select the *SwiftUI View* template and name the file `DraggableView`.

Declare the `DragState` enum and update the `DraggableView` struct like this:

```
enum DraggableState {
    case inactive
    case pressing
    case dragging(translation: CGSize)

    var translation: CGSize {
        switch self {
        case .inactive, .pressing:
            return .zero
        case .dragging(let translation):
            return translation
        }
    }

    var isPressing: Bool {
        switch self {
```

```
        case .pressing, .dragging:
            return true
        case .inactive:
            return false
        }
    }
}

struct DraggableView<Content>: View where Content: View {
    @GestureState private var dragState = DraggableState.inactive
    @State private var position = CGSize.zero

    var content: () -> Content

    var body: some View {
        content()
            .opacity(dragState.isPressing ? 0.5 : 1.0)
            .offset(x: position.width + dragState.translation.width, y: position.h
eight + dragState.translation.height)
            .animation(.easeInOut, value: dragState.translation)
            .gesture(
                LongPressGesture(minimumDuration: 1.0)
                .sequenced(before: DragGesture())
                .updating($dragState, body: { (value, state, transaction) in

                    switch value {
                    case .first(true):
                        state = .pressing
                    case .second(true, let drag):
                        state = .dragging(translation: drag?.translation ?? .zero)
                    default:
                        break
                    }

                })
                .onEnded({ (value) in

                    guard case .second(true, let drag?) = value else {
                        return
                    }

                    self.position.height += drag.translation.height
```

```
                    self.position.width += drag.translation.width
                })
            )
    }
}
```

All of the code is very similar to what you've written before. The tricks are to declare the `DraggableView` as a generic view and create a `content` property. This property accepts any `View`. We power this `content` view with the long press and drag gestures.

Now you can test this generic view by replacing the `DraggableView_Previews` like this:

```
struct DraggableView_Previews: PreviewProvider {
    static var previews: some View {
        DraggableView() {
            Image(systemName: "star.circle.fill")
                .font(.system(size: 100))
                .foregroundColor(.green)
        }
    }
}
```

In the code, we initalize a `DraggableView` and provide our own content, which is the star image. In this case, you should achieve the same star image which supports the long press and drag gestures.

So, what if we want to build a draggable text view? You can replace the code snippet with the following code:

```
struct DraggableView_Previews: PreviewProvider {
    static var previews: some View {
        DraggableView() {
            Text("Swift")
                .font(.system(size: 50, weight: .bold, design: .rounded))
                .bold()
                .foregroundColor(.red)
        }
    }
}
```

In the closure, we create a text view instead of the image view. If you run the project in the preview canvas, you can drag the text view to move it around (remember to long press for 1 second). Isn't it cool?

Figure 6. A draggable text view

If you want to create a draggable circle, you can replace the code like this:

```
struct DraggableView_Previews: PreviewProvider {
    static var previews: some View {
        DraggableView() {
            Circle()
                .frame(width: 100, height: 100)
                .foregroundColor(.purple)
        }
    }
}
```

That's how you create a generic draggable. Try to replace the circle with other views to make your own draggable view and have fun!

Exercise

We've explored three built-in gestures including tap, drag, and long press in this chapter. However, there are a couple of them we haven't checked out. As an exercise, try to create a generic scalable view that can recognize the `MagnificationGesture` and scale any given view accordingly. Figure 7 shows you a sample result.

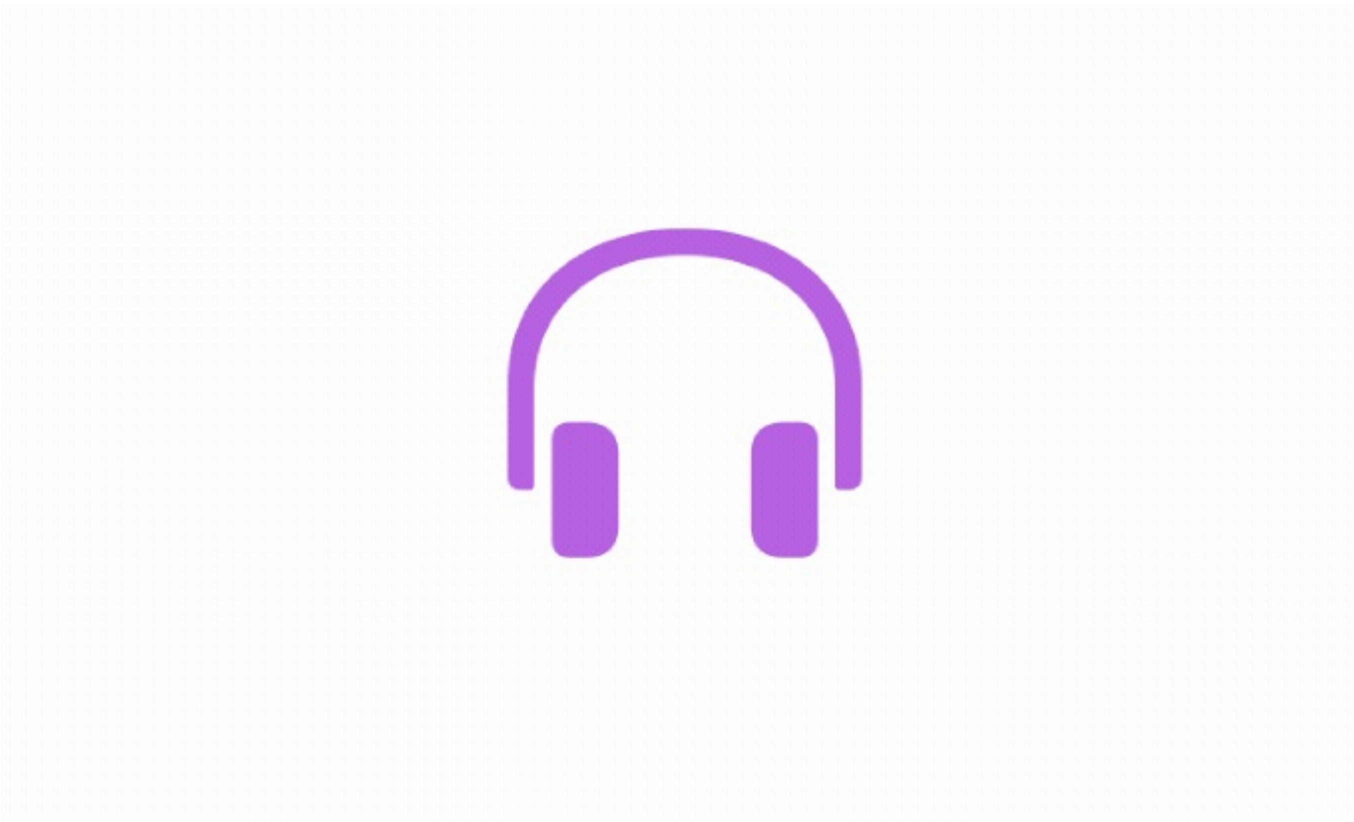


Figure 7. A scalable image view

Summary

The SwiftUI framework has made gesture handling very easy. As you've learned in this chapter, the framework has provided several ready to use gesture recognizers. To enable a view to support a certain type of gesture, all you need to do is attach the `.gesture` modifier to it. Composing multiple gestures has never been so simple.

It's a growing trend to build gesture-driven user interfaces for mobile apps. With the easy to use API, try to power your apps with some useful gestures to delight your users.

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

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.

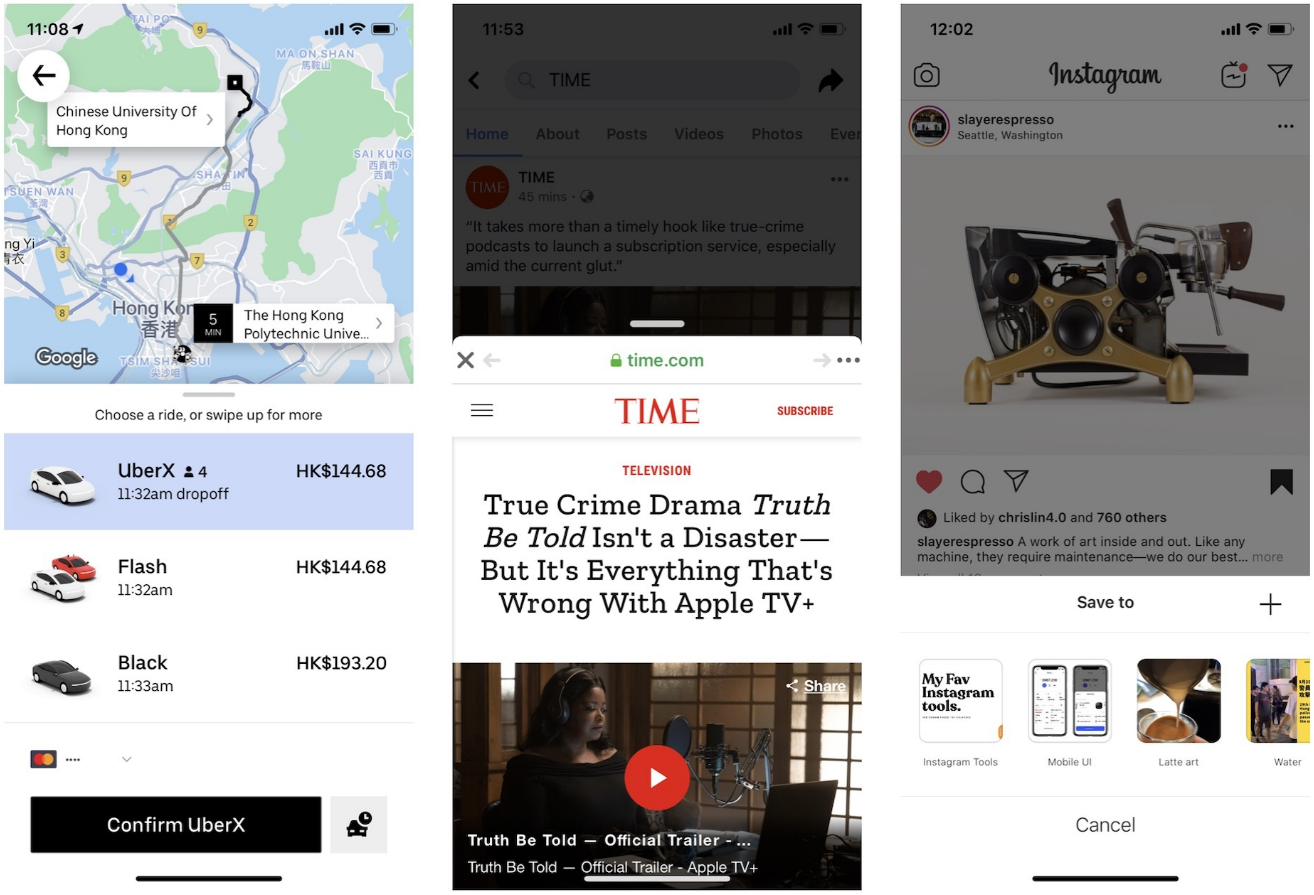


Figure 1. Uber, Facebook and Instagram all use bottom sheets in their apps

In this chapter, we will build a similar expandable bottom sheet using SwiftUI gestures. The demo app shows a list of restaurant 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.

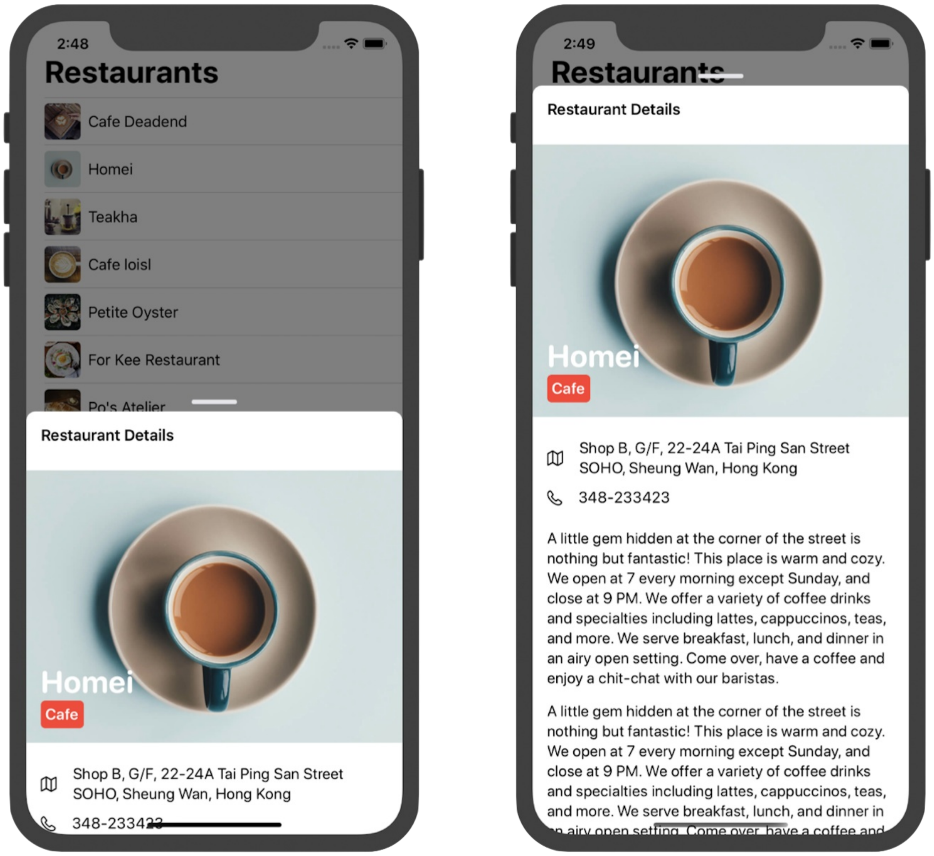


Figure 2. Building a expandable bottom sheet

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

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.

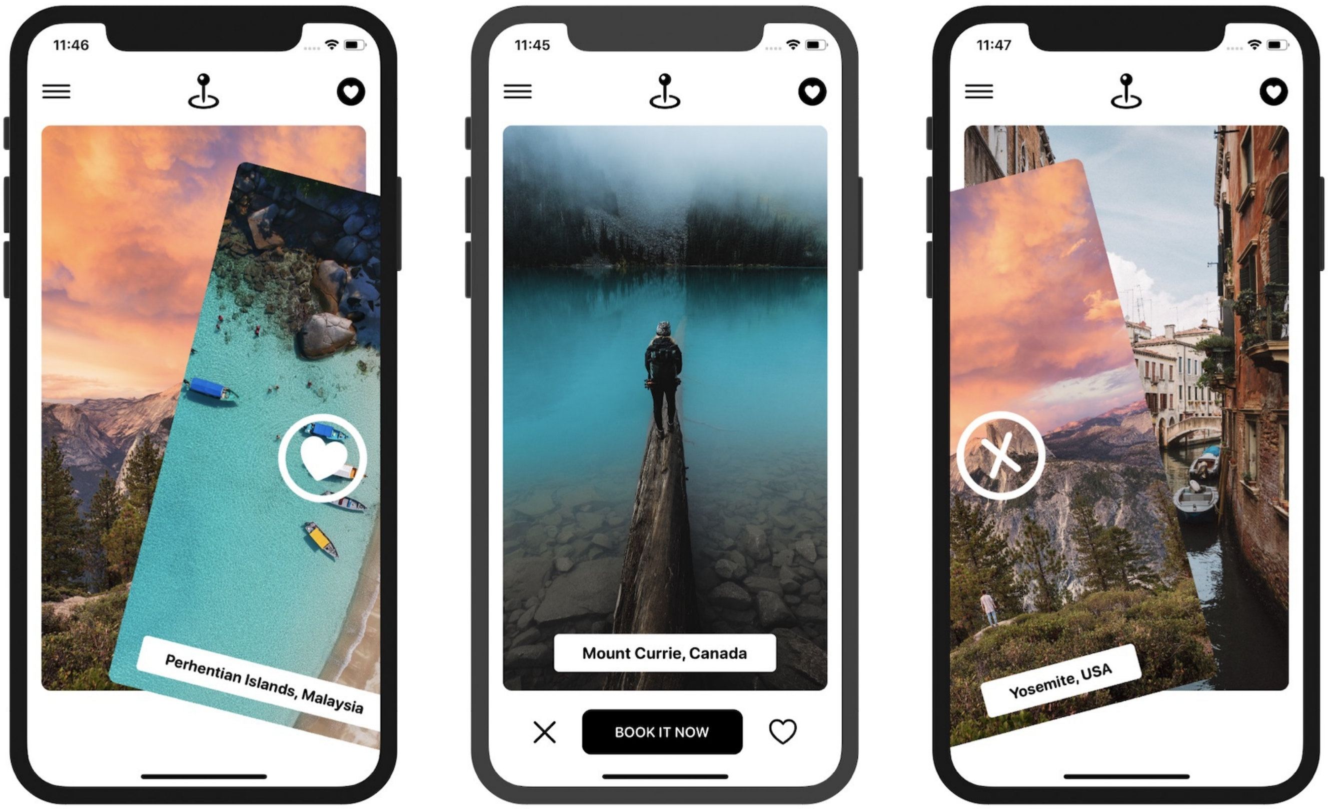


Figure 1. Building a tinder-like user interface

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

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

Chapter 20
Creating an Apple Wallet like Animation and View Transition

Do you use Apple's Wallet app? I believe so. In the previous chapter, we built a simple app with a Tinder-like UI. What we're going to do in this chapter is to create an animated UI similar to the one you see in the Wallet app. When you tap and hold a credit card in the wallet app, you can use the drag gesture to rearrange the cards. If you haven't used the app, open Wallet and take a quick look. Alternatively, you can visit this URL (https://link.appcoda.com/swiftui-wallet) to check out the animation we're going to build.

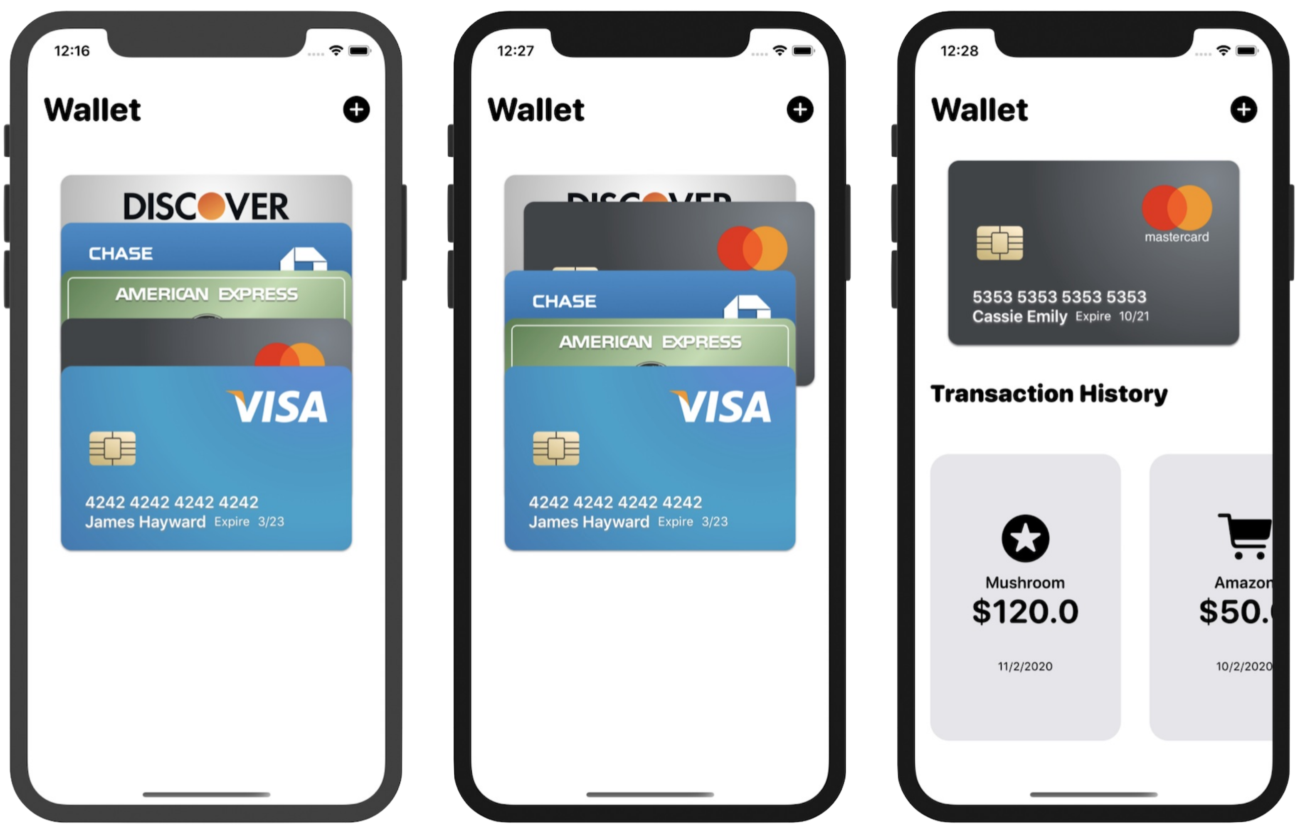


Figure 1. Building a Wallet-like animations and view transitions

In the Wallet app, tapping one of the cards will bring up the transaction history. We will also create a similar animation, so you'll better understand view transitions and horizontal scroll view.

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

Chapter 21
Working with JSON, Slider and Data Filtering

JSON, short for JavaScript Object Notation, is a common data format for data interchange in client-server application. Even though we are mobile app developers, it's inevitable to work with JSON since nearly all web APIs or backend web services use JSON as the data exchange format.

In this chapter, we will discuss how you can work with JSON while building an app using the SwiftUI framework. If you do not have any ideas about JSON, I would recommend to check out this free chapter from our Intermediate programming book. It will explain to you in details the two different approaches in handling JSON in Swift.

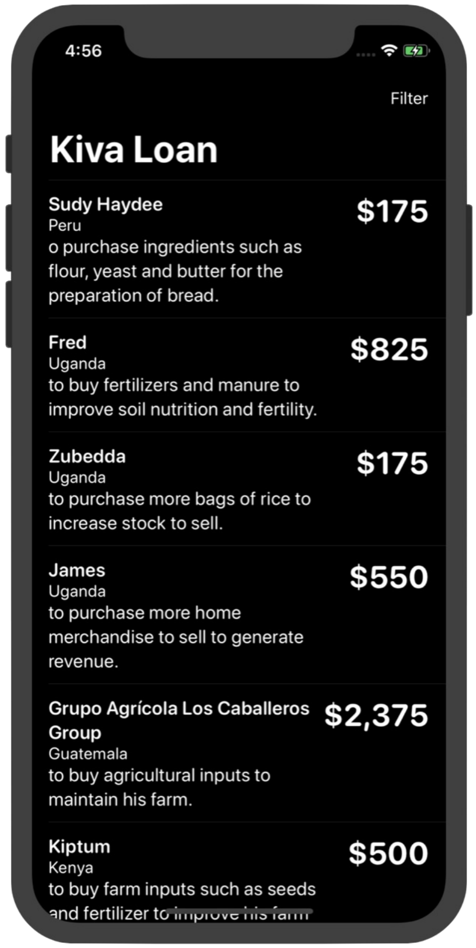


Figure 1. The demo app

As usual, in order to grasp the knowledge of JSON and its related APIs, you will build a simple JSON app that utilize a JSON-based API provided by Kiva.org. If you haven't heard of Kiva, it is a non-profit organization with a mission to connect people through lending to alleviate poverty. It lets individuals lend as little as $25 to help create opportunities around the world. Kiva provides free web-based APIs for developers to access their data. For our demo app, we'll call up a free Kiva API to retrieve the most recent fundraising loans and display them in a list view as shown in figure 1.

On top of that, we will demonstrate the usage of a Slider, which is one of the many built-in UI controls provided by SwiftUI. You will implement a data filtering option in the app so that users can filter the loan data in the list.

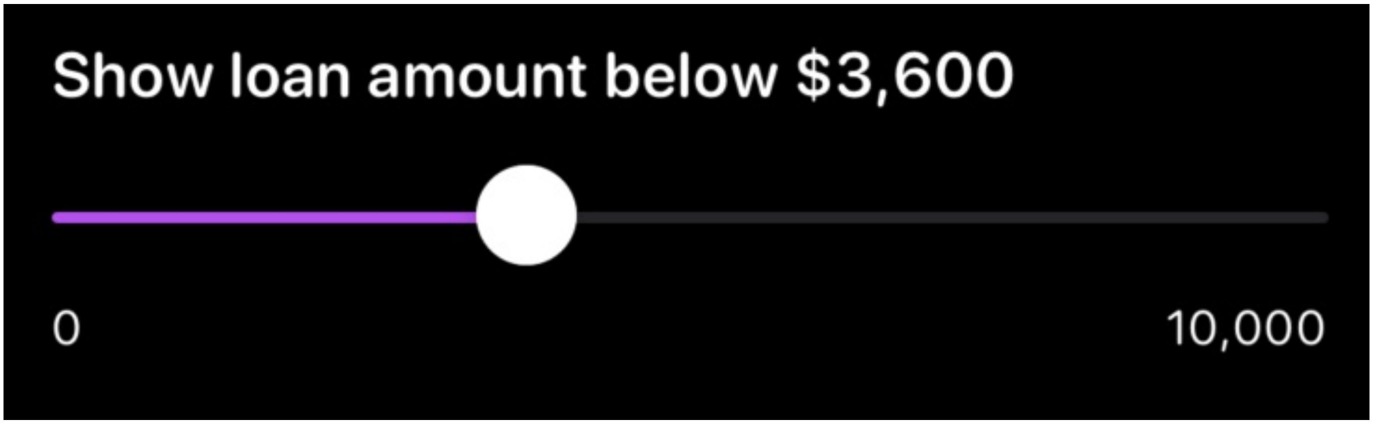


Figure 2. A slider control

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

Chapter 22
Building a ToDo App with Core Data

One common question of iOS app development is how we can work with Core Data and SwiftUI to save data permanently in the built-in database. In this chapter, we will look into it by building a ToDo app.

Since the ToDo demo app makes use of List and Combine to handle the data presentation and sharing, I assume that you've read the following chapters:

- Chapter 7 - Understanding State and Binding
- Chapter 10 - Understanding Dynamic List, ForEach and Identifiable
- Chapter 14 - Data Sharing with Combine and Environment Objects

If you haven't done so or already forgot what Combine and Environment Objects are, go back and read the chapters again.

So, what are we going to do this chapter in order to understand Core Data? Instead of building the ToDo app from scratch, I already built the core parts of the app. That said, it can't save the data permanently. To be more specific, it can only save the to-do items in an array. Whenever the user closes the app and starts it again, all the data is gone. We will modify the app and convert it to use Core Data for saving the data permanently to the local database. Figure 1 shows some sample screenshots of the ToDo app.

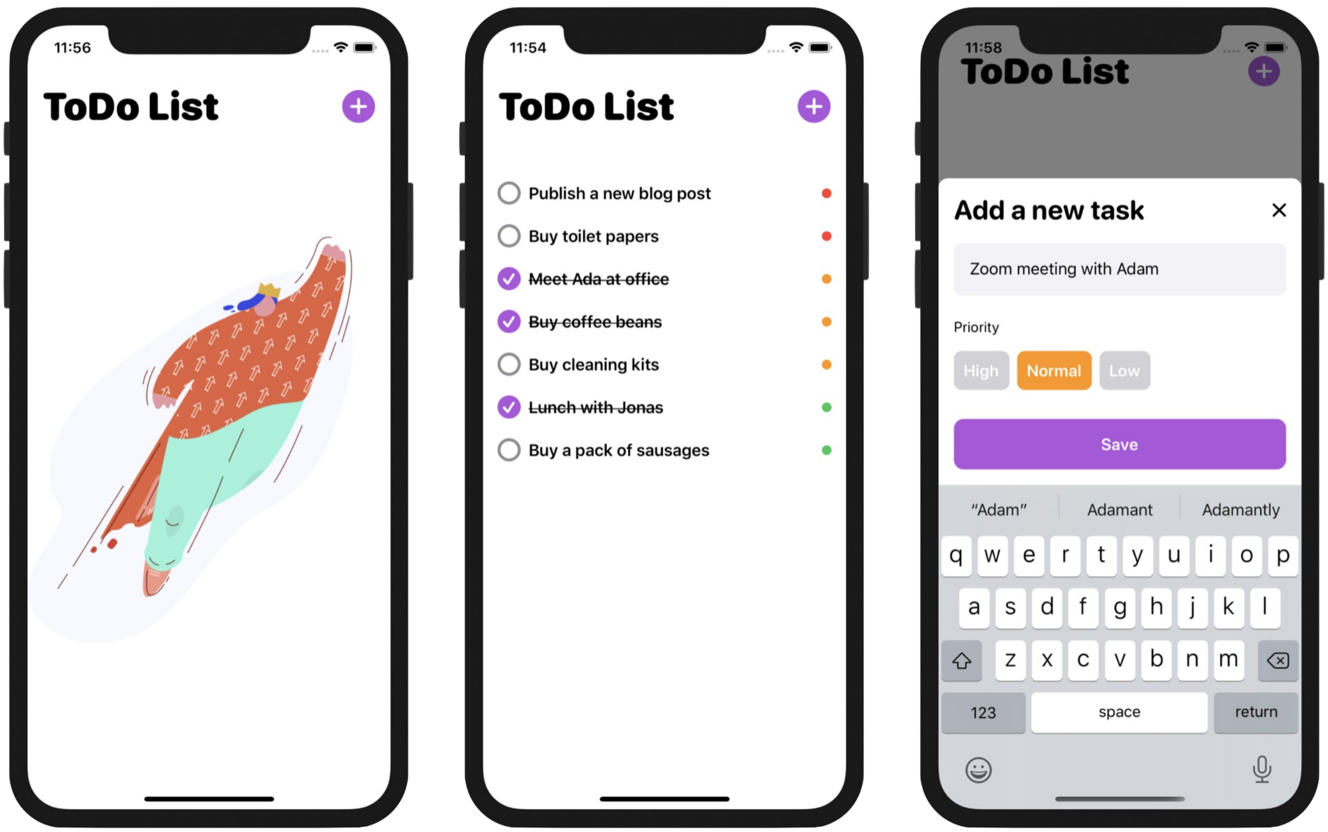


Figure 1. The ToDo demo app

No worries. Before we perform the modification, I will walk you through the starter project so you fully understand how the code works. Other than Core Data, you will also learn how to customize the style of a toggle. Take a look at the screenshots above. The checkbox is actually a toggle view of SwiftUI. I will show you how to create these checkboxes by customizing the Toggle's style.

We got a lot to cover in this chapter, so let's get started.

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

Chapter 23
Integrating UIKit with SwiftUI Using UIViewRepresentable

In the previous chapter, we addressed a common question about Core Data and SwiftUI. The other common question is how to work with UIKit views in SwiftUI projects. In this chapter, you will learn this technique by integrating a UISearchBar in the Todo app.

If you are new to UIKit, UISearchBar is a built-in component of the framework that allows developers to present a search bar for data search. Figure 1 shows you the standard search bar in iOS. For SwiftUI, however, it doesn't come with this standard UI component. To implement a search bar in a SwiftUI project (say, our ToDo app), one approach is to make use of the `UISearchBar` component in UIKit.

So, how can we interface with UIKit views or controllers in SwiftUI?

For the purpose of backward compatibility, Apple introduced a couple of new protocols, namely `UIViewRepresentable` and `UIViewControllerRepresentable` in the iOS SDK. With these protocols, you can wrap a UIKit view (or view controller) and make it available to your SwiftUI project.

To see how it works, we will enhance our Todo app with a search function. We will add a search bar right below the app title and let users filter the to-do items by entering a search term.

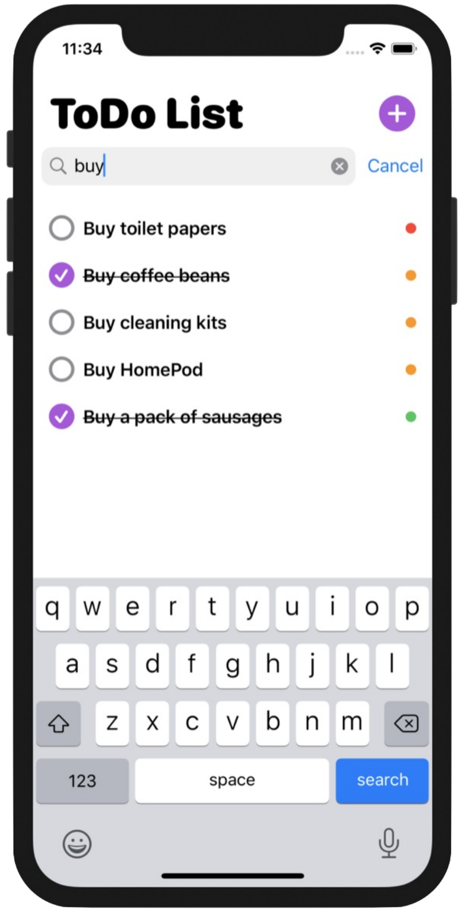


Figure 1. Adding a search bar in the ToDo app

To get started, please download the ToDo project. We will build on top of the existing project. In case you haven't read chapter 22, I recommend you to check it out first. This will help you better understand the topic we are going to discuss below, especially you have no idea about Core Data.

Understanding UIViewRepresentable

To use a UIKit view in SwiftUI, you can wrap the view with the `UIViewRepresentable` protocol. Basically, you just need to create a `struct` in SwiftUI that adopts the protocol to create and manage a `UIView` object. Here is the skeleton of the custom wrapper for a UIKit view:

```
struct CustomView: UIViewRepresentable {

    func makeUIView(context: Context) -> some UIView {
        // Return the UIView object
    }

    func updateUIView(_ uiView: some UIView, context: Context) {
        // Update the view
    }
}
```

In the actual implementation, you replace `some UIView` with the UIKit view you want to wrap. Let's say, we want to use `UISearchBar` in UIKit. The code can be written like this:

```
struct SearchBar: UIViewRepresentable {

    func makeUIView(context: Context) -> UISearchBar {

        return UISearchBar()
    }

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

        // Update the view
    }
}
```

In the `makeUIView` method, we return an instance of `UISearchBar`. This is how you wrap a UIKit view and make it available to SwiftUI. To use the `SearchBar`, you can treat like any SwiftUI view and create it like this:

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

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

Chapter 24
Creating a Search Bar View and Working with Custom Binding

Previously, we showed you how to implement a search bar by reusing the `UISearchBar` component of the old UIKit framework. Have you ever thought of building one from scratch? If you look at the search bar carefully, it's not too difficult to implement. So, let's try to build a SwiftUI version of search bar in this chapter.

Not only can you learn how to create the search bar view, we will show you how to work with custom binding. We've discussed binding before, but haven't showed you how to create a custom binding. Custom binding is particularly useful when you need to insert additional program logic while it's being read and write. On top of all that, you will learn how to dismiss the software keyboard in SwiftUI.

Figure 1 gives you an idea about the search bar we're going to build. The look & feel is the same as that of `UISearchBar` in UIKit. We will also implement the *Cancel* button which only appears when the user starts typing in the search field.

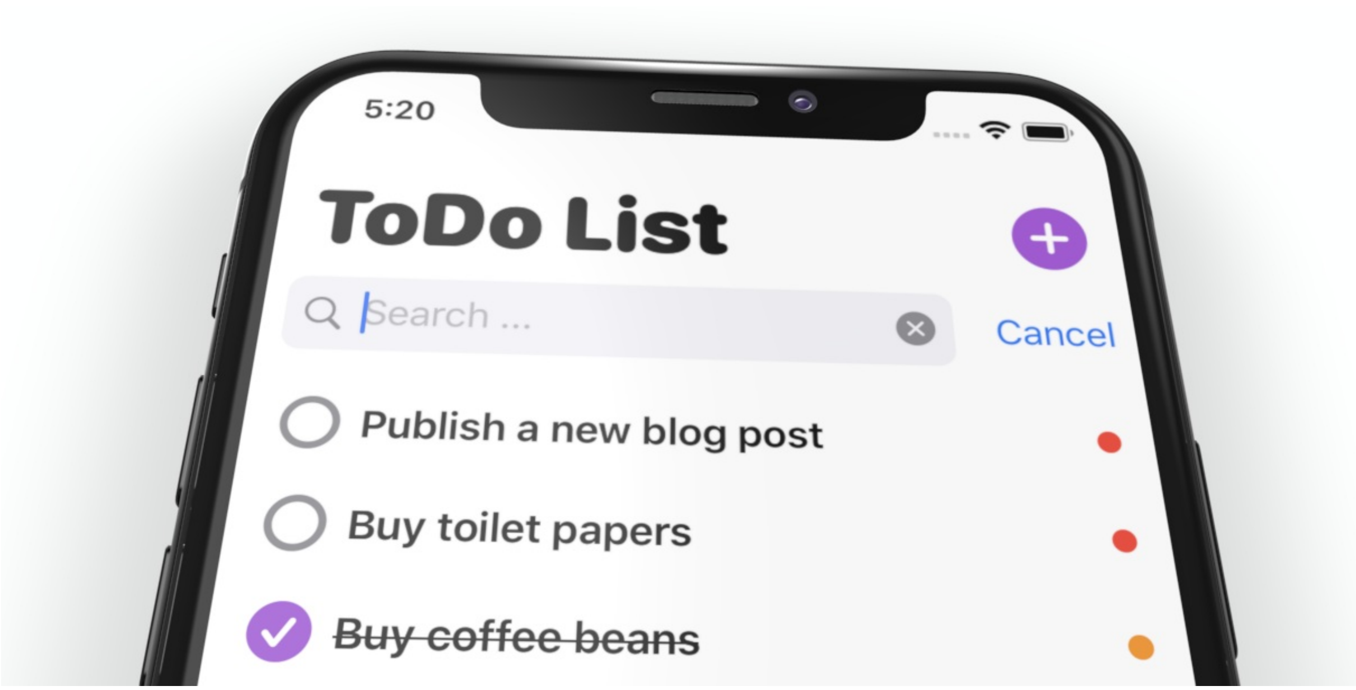


Figure 1. Building a search bar view entirely using SwiftUI

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

Chapter 25
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
6. How to use DatePicker for date selection

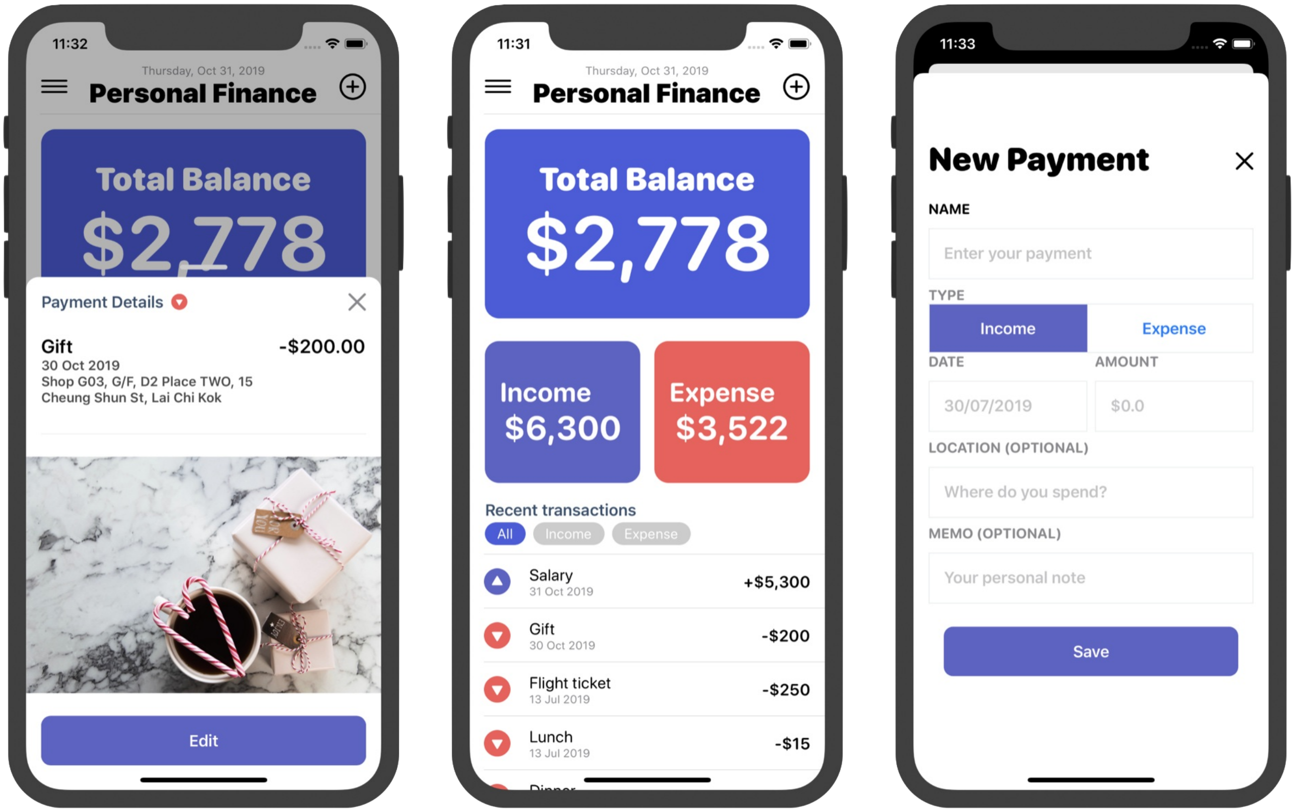


Figure 1. The Personal Finance App

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

Chapter 26
Creating an App Store like Animated View Transition

I believe you should have used the built-in *App Store* app before. In the *Today* section, it presents users with various articles and app recommendations. What interests me and many of you is the animated view transition. As you can see in figure 1, the articles are listed in a card like format. When you tap it, the card pops out to reveal the full content. To dismiss the article view and return to the list view, you can simply drag down the article to minimize it. If you don't understand what I mean, the best way is to open the *App Store* app on your iPhone to try it out.

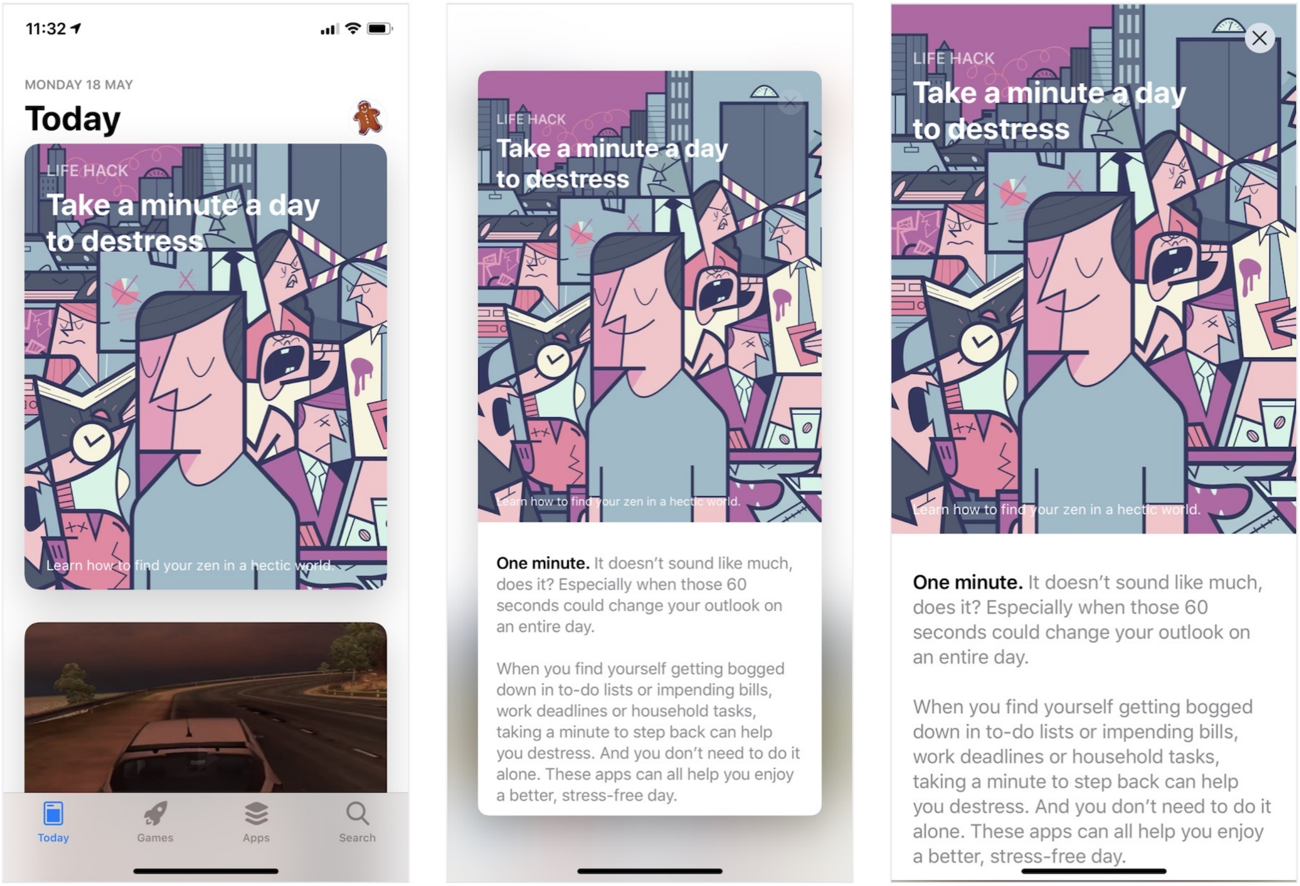


Figure 1. Apple's App Store app

In this chapter, we will build a similar list view and implement the animated transition using SwiftUI. In particular, you will learn the following techniques:

- How to use GeometryReader to detect screen sizes
- How to create a variable-sized card view
- How to implement an App Store like animated view transition
- How to implement a drag-to-dismiss view animation

Let's get started.

Introducing the Demo App

As usual, we will build a demo app together. The app looks very similar to the *App Store* app but without the tab bar. It only has a list view showing all the articles in card format. When a user taps any of the articles, the card expands to full screen and display the article details. To return to the list view, the user can either tap the close button or drag down the article view to collapse it.

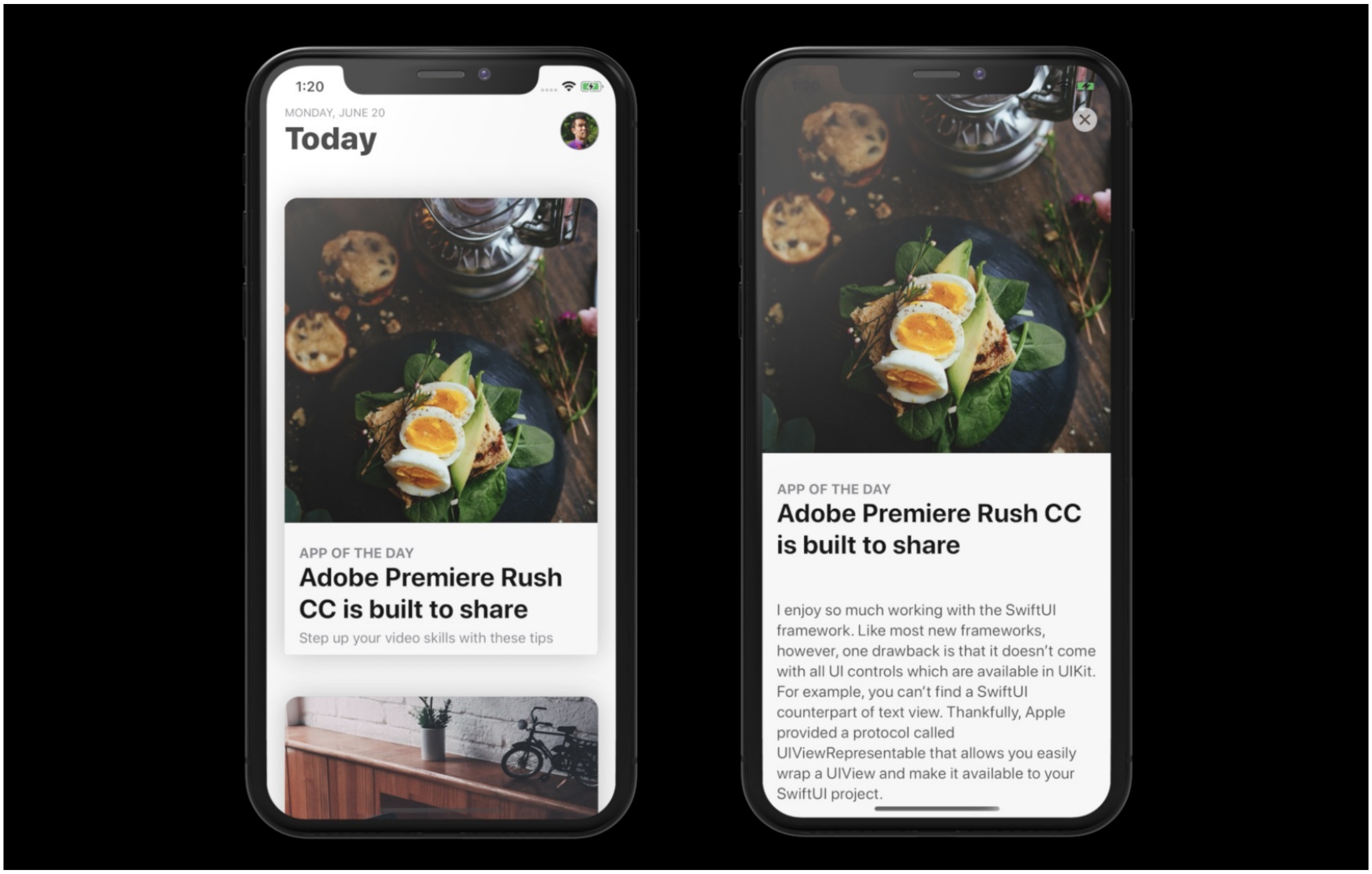


Figure 2. The demo app

We will build the app from scratch. But to save you time from typing some of the code, I have prepared a starter project [only available in the full version of the book] for you. After downloading the project, unzip it and open `SwiftUIAppStore.xcodeproj` to take a look.

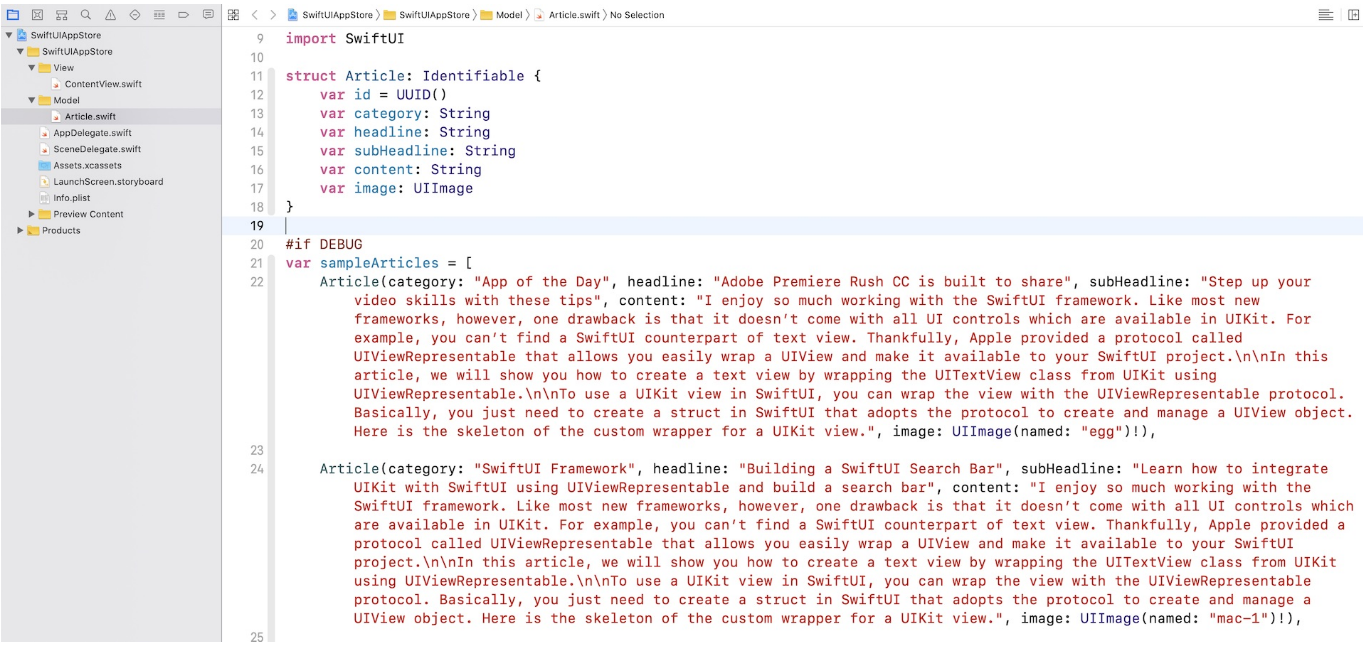


Figure 3. The starter project

The starter projects comes with the following implementation:

1. It already bundles the required images in the asset catalog.
2. The `ContentView.swift` file is the default SwiftUI view generated by Xcode.
3. The `Article.swift` file contains the `Article` struct, which represents an article in the app. For testing purpose, this file also creates the `sampleArticles` array which includes some test data. You may modify its content if you want to change the article data.

Understanding the Card View

You've learned how to create a card-like UI before. This card view is very similar to that implemented in chapter 5, but it will be more flexible to support scrollable content. In other words, it has two modes: *excerpt* and *full content*. In the excerpt mode, it only displays the image, category, headline and sub-headline of the article. As its name suggests, the full content will display the article details as shown before in figure 2.

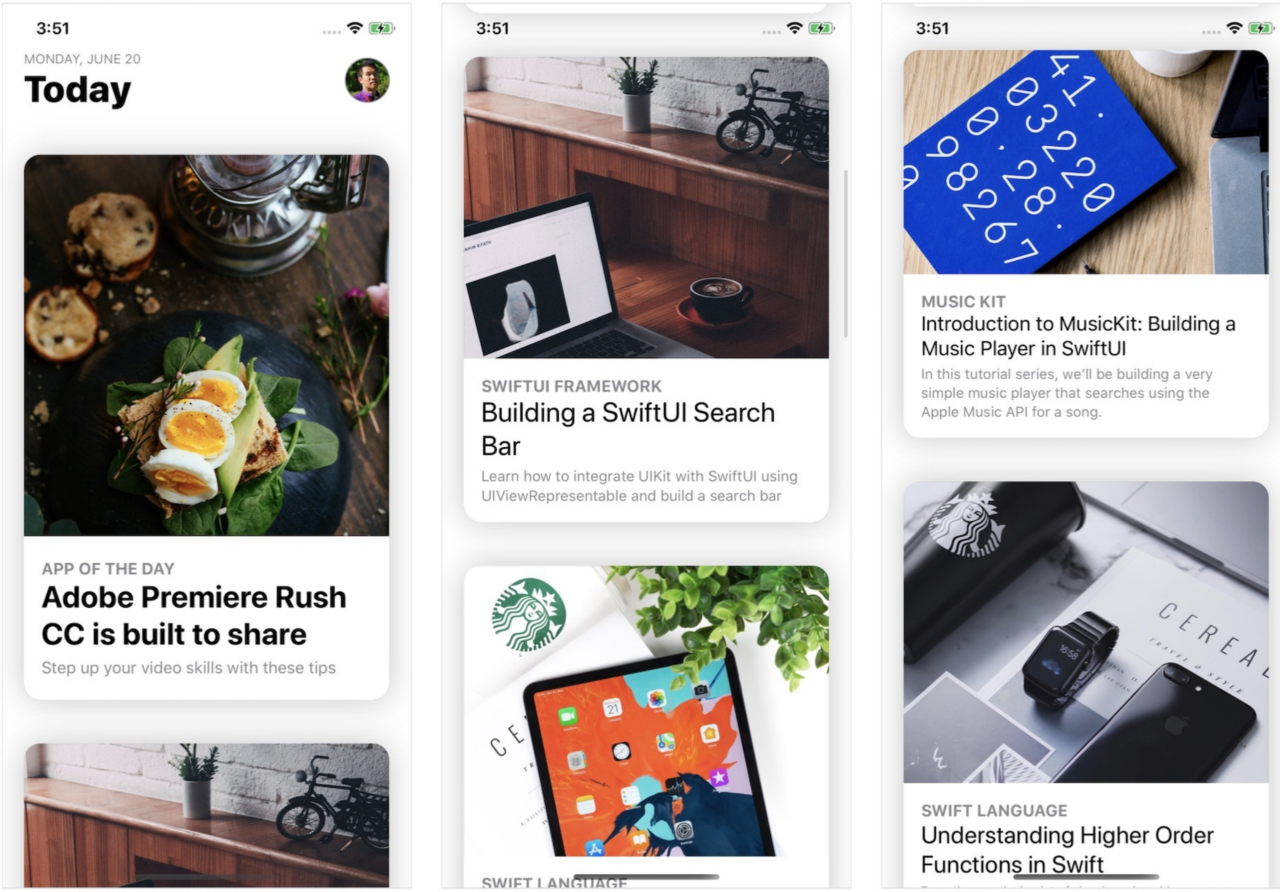


Figure 4. The sample card views

If you look a bit closer into the card views shown in figure 4, you should find that the size of card views varies according to the height of the image. However, the height of the card will not exceed 500 points.

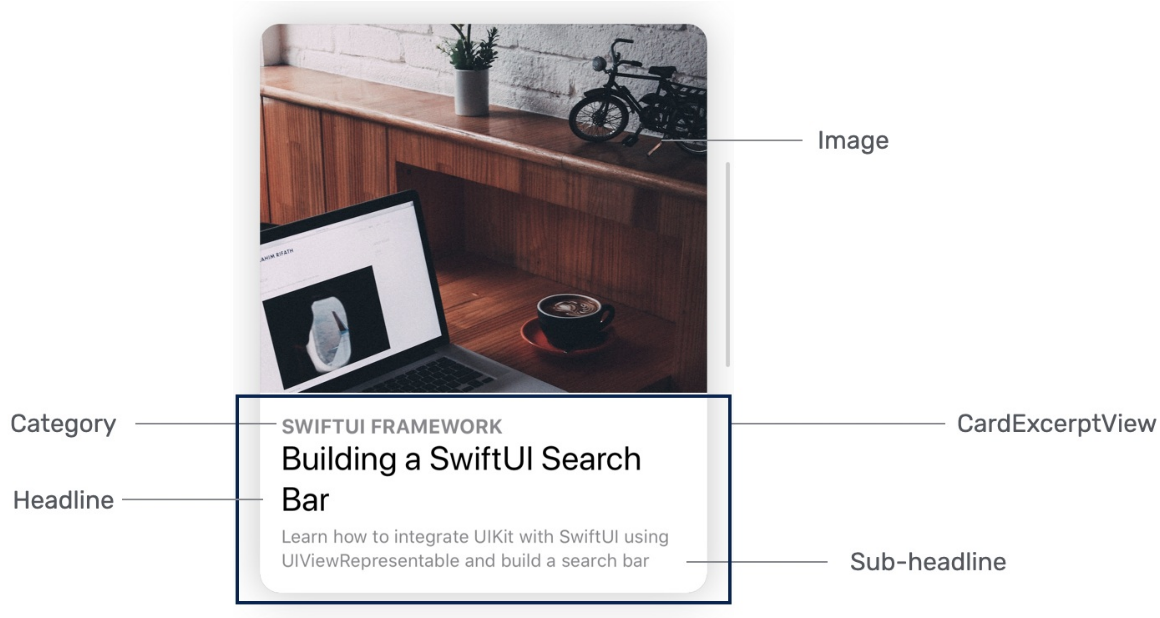


Figure 5. The components of a card view in excerpt mode

Let's also take a look at how the card view looks like in full content mode. As you can see in the figure below, the card view expands to full screen that displays the content. Other than that, the image is a little bit bigger and the sub-headline is hidden. Further, the close button appears on screen for users to dismiss the view. Please also take note that this is a scrollable view.

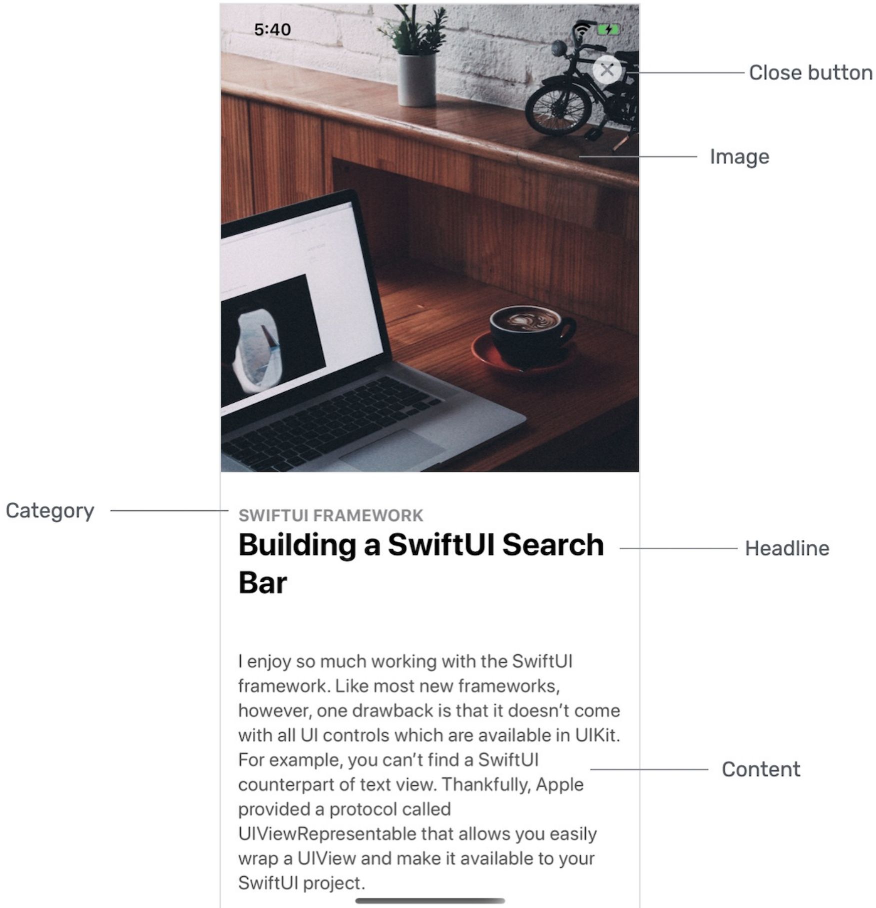


Figure 6. The components of a card view in full content mode

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

Chapter 27
Building an Image Carousel

Carousel is one of the common UI patterns that you can find it in most of the mobile and web apps. Some people refer it as an image slider or rotator. However, whatever name you call it, a carousel is designed to display a set of data in finite screen space. Say, for an image carousel, it may show a single image from its collection with a navigation control suggesting additional content. Users can swipe the screen to navigate through the image set. This is how Instagram presents multiple images to users. You can also find similar carousels in many other iOS apps such as Music and App Store.

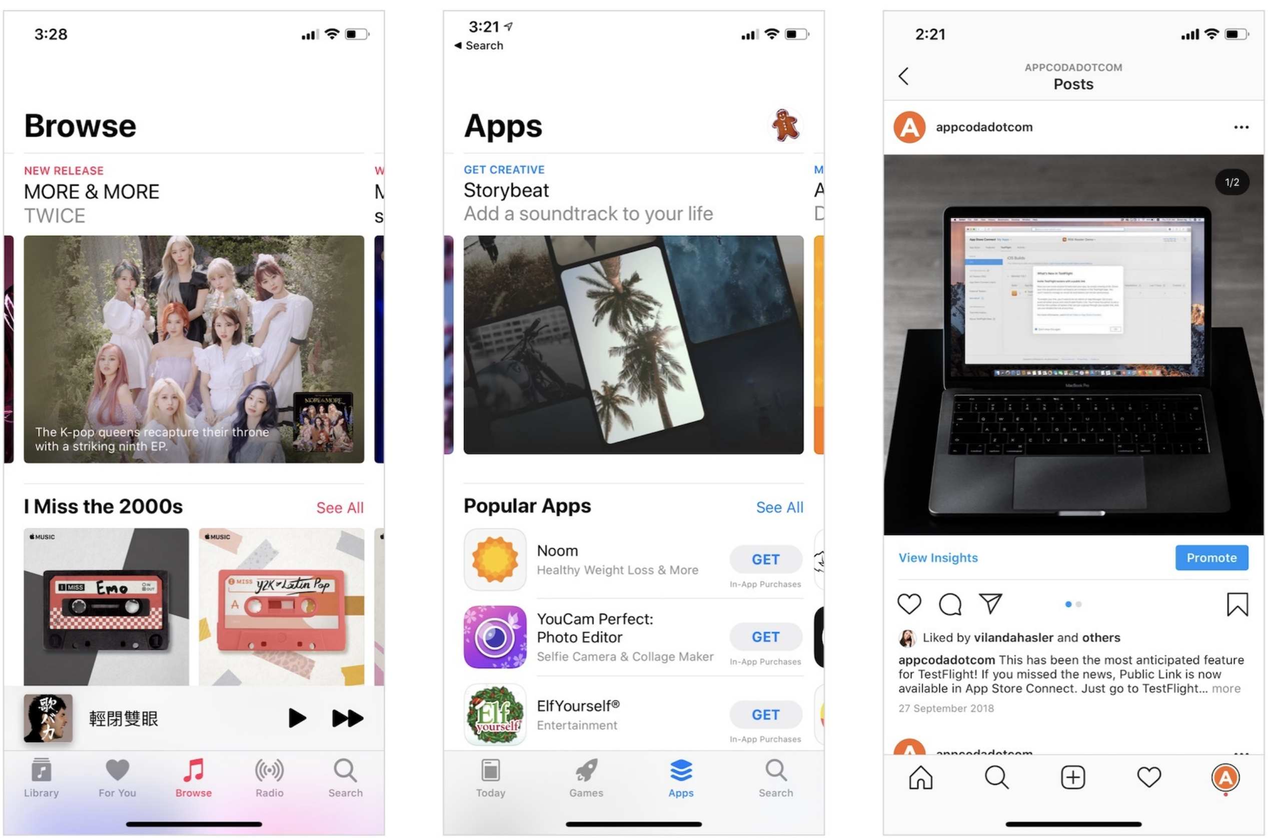


Figure 1. Sample carousel in the Music, App Store, and Instagram app

In this chapter, you will learn how to build an image carousel entirely in SwiftUI. There are various ways to implement a carousel. One approach is to integrate with the UIKit component called `UIPageViewController` and make use of it to create the carousel. However, we will explore an alternative approach and create the carousel without using the UIKit framework but build it completely in SwiftUI.

Let's get started.

Introducing the Travel Demo App

Likes other chapters, I love to walk you through the implementation by building a demo app. The app displays a collection of travel destinations in the form of a carousel. To browse through the trips, the user can swipe right to view the subsequent destination or swipe left to check out the previous trip. To make this demo app more engaging, the user can tap a destination to see the details. So, other than the implementation of a carousel, you will also learn some animation techniques that can be applied in your own apps. Figure 2 shows you some sample screenshots of the demo app. To see it in action, you can check out the video at https://link.appcoda.com/carousel-demo.

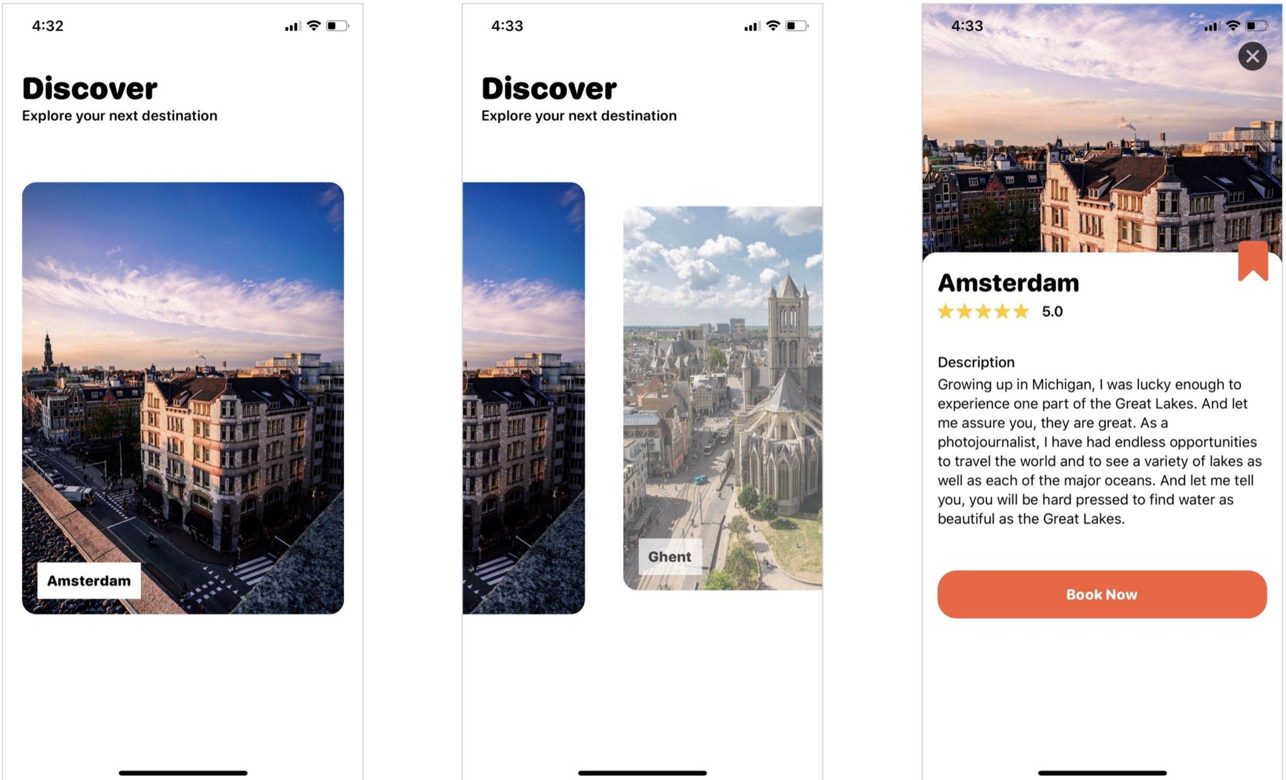


Figure 2. The demo app

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

Chapter 28
Building an Expandable List View Using OutlineGroup

SwiftUI list is very similar to UITableView in UIKit. In the first release of SwiftUI, Apple's engineers made list view construction a breeze. You do not need to create a prototype cell and there is no delegate/data source protocol. With just a few lines of code, you can build a list view with custom cells. Starting from iOS 14, Apple continued to improve the `List` view and introduced several new features. In this chapter, we will show you how to build an expandable list / outline view and explore the inset grouped list style.

The Demo App

First, let's take a look at the final deliverable. I'm a big fan of La Marzocco, so I used the navigation menu on its website as an example. The list view below shows an outline of the menu. Users can tap the disclosure button to expand the list.

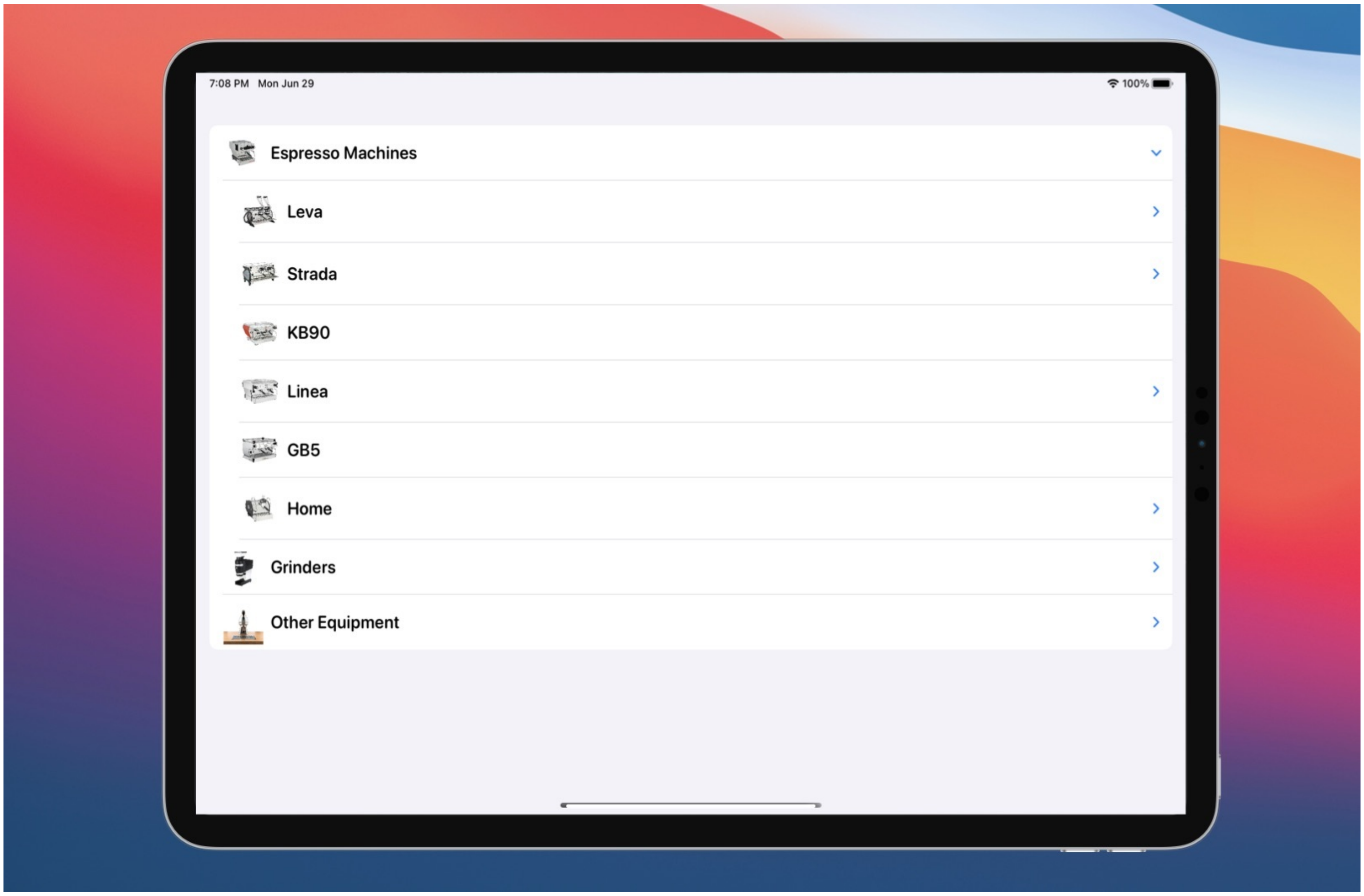


Figure 1. The expandable list view

Of course, you can build this outline view using your own implementation. Starting from iOS 14, Apple made it simpler for developers to build this kind of outline view, which automatially works on iOS, iPadOS, and macOS.

Creating the Expandable List

In order to follow this chapter, please download these image assets from https://www.appcoda.com/resources/swiftui/expandablelist-images.zip. Then create a new SwiftUI project using the *App* template. I named the project *SwiftUIExpandableList* but you are free to set the name to whatever you want.

Once the project is created, unzip the image archive and add the images to the asset catalog.

In the project navigator, right click *SwiftUIExpandableList* and choose to create a new file. Select the *Swift File* template and name it *MenuItem.swift*.

Setting up the data model

To make the list view expandable, all you need to do is create a data model like this. Insert the following code in the file:

```
struct MenuItem: Identifiable {
    var id = UUID()
    var name: String
    var image: String
    var subMenuItems: [MenuItem]?
}
```

In the code above, we have a struct that models a menu item. The key to making a nested list is to include a property that contains an optional array of child menu items (i.e. `subMenuItems`). Note that the children are of the same type (MenuItem) as their parent.

For the top level menu items, we create an array of `MenuItem` in the same file like this:

```
// Main menu items
let sampleMenuItems = [ MenuItem(name: "Espresso Machines", image: "linea-mini", s
ubMenuItems: espressoMachineMenuItems),
                        MenuItem(name: "Grinders", image: "swift-mini", subMenuIte
ms: grinderMenuItems),
                        MenuItem(name: "Other Equipment", image: "espresso-ep", su
bMenuItems: otherMenuItems)
                    ]
```

For each of the menu item, we specify the array of the sub-menu items. If there are no sub-menu items, you can omit the `subMenuItems` parameter or pass it a `nil` value. We define the sub-menu items like this:

```swift
// Sub-menu items for Espressco Machines
let espressoMachineMenuItems = [ MenuItem(name: "Leva", image: "leva-x", subMenuItems: [ MenuItem(name: "Leva X", image: "leva-x"), MenuItem(name: "Leva S", image: "leva-s") ]),
                                 MenuItem(name: "Strada", image: "strada-ep", subMenuItems: [ MenuItem(name: "Strada EP", image: "strada-ep"), MenuItem(name: "Strada AV", image: "strada-av"), MenuItem(name: "Strada MP", image: "strada-mp"), MenuItem(name: "Strada EE", image: "strada-ee") ]),
                                 MenuItem(name: "KB90", image: "kb90"),
                                 MenuItem(name: "Linea", image: "linea-pb-x", subMenuItems: [ MenuItem(name: "Linea PB X", image: "linea-pb-x"), MenuItem(name: "Linea PB", image: "linea-pb"), MenuItem(name: "Linea Classic", image: "linea-classic") ]),
                                 MenuItem(name: "GB5", image: "gb5"),
                                 MenuItem(name: "Home", image: "gs3", subMenuItems: [ MenuItem(name: "GS3", image: "gs3"), MenuItem(name: "Linea Mini", image: "linea-mini") ])
                                ]

// Sub-menu items for Grinder
let grinderMenuItems = [ MenuItem(name: "Swift", image: "swift"),
                         MenuItem(name: "Vulcano", image: "vulcano"),
                         MenuItem(name: "Swift Mini", image: "swift-mini"),
                         MenuItem(name: "Lux D", image: "lux-d")
                        ]

// Sub-menu items for other equipment
let otherMenuItems = [ MenuItem(name: "Espresso AV", image: "espresso-av"),
                         MenuItem(name: "Espresso EP", image: "espresso-ep"),
                         MenuItem(name: "Pour Over", image: "pourover"),
                         MenuItem(name: "Steam", image: "steam")
                        ]
```

Presenting the List

With the data model prepared, we can now create the list view. The `List` view has an optional `children` parameter. If you have any sub items, you can provide their key path. SwiftUI will then look up the sub menu items recursively and present them in outline form. Open `ContentView.swift` and insert the following code in `body` :

```
List(sampleMenuItems, children: \.subMenuItems) { item in
    HStack {
        Image(item.image)
            .resizable()
            .scaledToFit()
            .frame(width: 50, height: 50)

        Text(item.name)
            .font(.system(.title3, design: .rounded))
            .bold()
    }
}
```

In the closure of the `List` view, you describe how each row looks. In the code above, we layout an image and a text description using `HStack`. If you've added the code in `ContentView` correctly, SwiftUI should render the outline view as shown in figure 2.

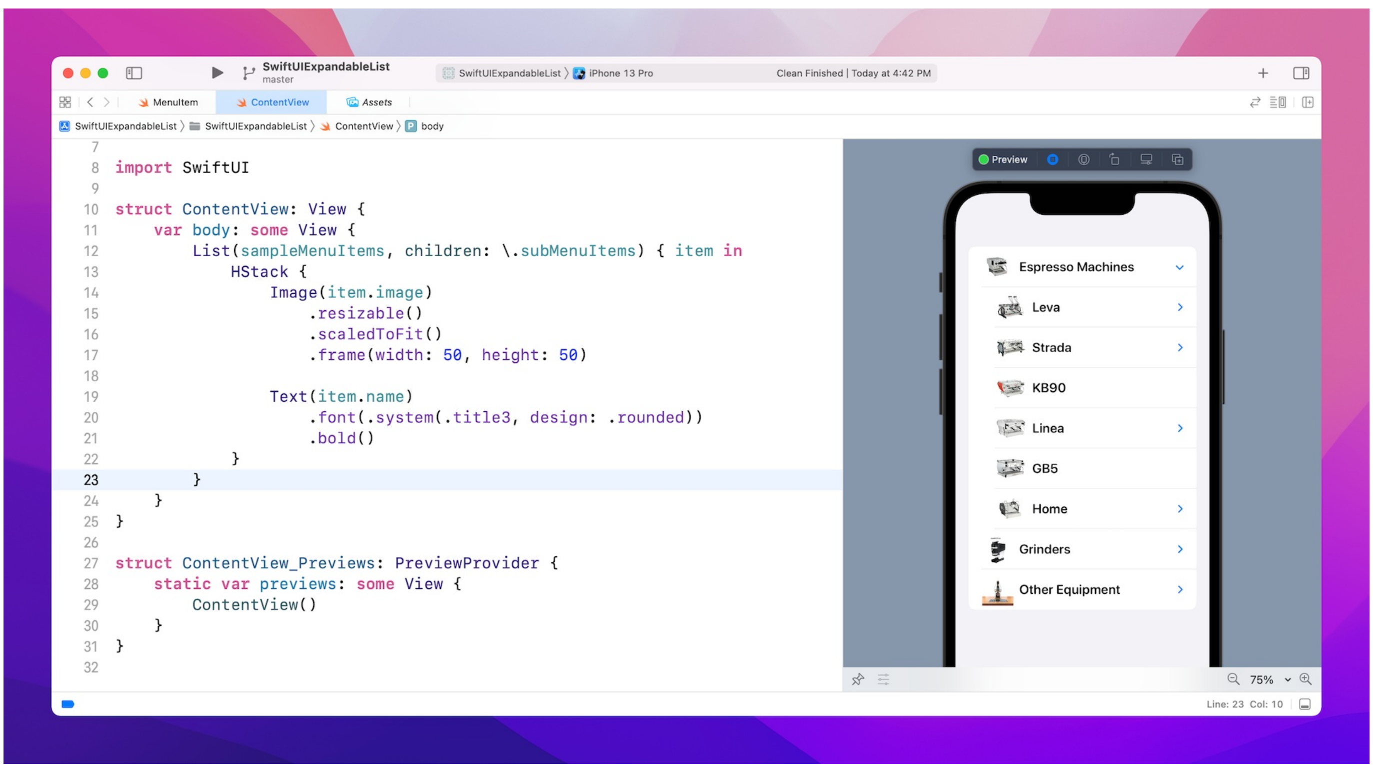


Figure 2. The expandable list view

To test the app, run it in a simulator or the preview canvas. You can tap the disclosure indicator to access the submenu.

Using the Plain List Style

In iOS 15, Apple sets the default style of the list view to Inset Grouped, where the grouped sections are inset with rounded corners. If you want to switch it back to the plain list style, you can attach the `.listStyle` modifier to the `List` view and set its value to `.plain` like this:

```
List {
  ...
}
.listStyle(.plain)
```

If you've followed me, the list view should now change to the plain style.

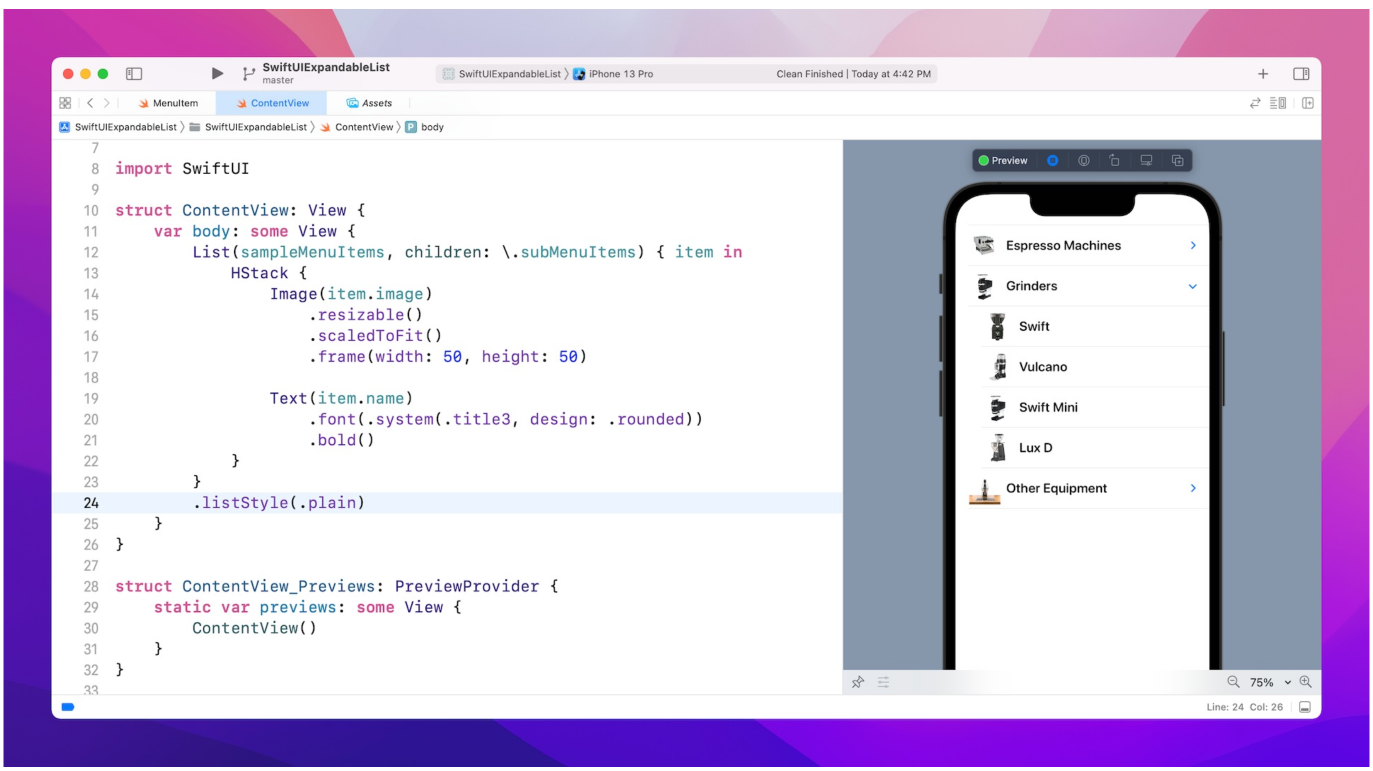


Figure 3. Using inset grouped list style

Using OutlineGroup to Customize the Expandable List

As you can see in the earlier example, it is pretty easy to create an outline view using the `List` view. However, if you want to have a better control of the appearance of the outline view (e.g. adding a section header), you will need to use `OutlineGroup`. This new view is introduced in iOS 14 for you to present a hierarchy of data.

If you understand how to build an expandable list view, the usage of `OutlineGroup` is very similar. For example, the following code allows you to build the same expandable list view like the one shown in figure 1:

```
List {
    OutlineGroup(sampleMenuItems, children: \.subMenuItems) {  item in
        HStack {
            Image(item.image)
                .resizable()
                .scaledToFit()
                .frame(width: 50, height: 50)

            Text(item.name)
                .font(.system(.title3, design: .rounded))
                .bold()
        }
    }
}
```

Similar to the `List` view, you just need to pass `OutlineGroup` the array of items and specify the key path for the sub menu items (or children).

With `OutlineGroup`, you have better control on the appearance of the outline view. For example, we want to display the top-level menu items as the section header. You can write the code like this:

```
List {
    ForEach(sampleMenuItems) { menuItem in

        Section(header:
            HStack {

                Text(menuItem.name)
                    .font(.title3)
                    .fontWeight(.heavy)

                Image(menuItem.image)
                    .resizable()
                    .scaledToFit()
                    .frame(width: 30, height: 30)

            }
            .padding(.vertical)

        ) {
            OutlineGroup(menuItem.subMenuItems ?? [MenuItem](), children: \.subMen
uItems) {  item in
                HStack {
                    Image(item.image)
                        .resizable()
                        .scaledToFit()
                        .frame(width: 50, height: 50)

                    Text(item.name)
                        .font(.system(.title3, design: .rounded))
                        .bold()
                }
            }
        }
    }
}
.ignoresSafeArea()
```

In the code above, we use `ForEach` to loop through the menu items. We present the top-level items as section headers. For the rest of the sub menu items, we rely on `OutlineGroup` to create the hierachy of data. If you've made the change in

`ContentView.swift` , you should see an outline view like that shown in figure 4.

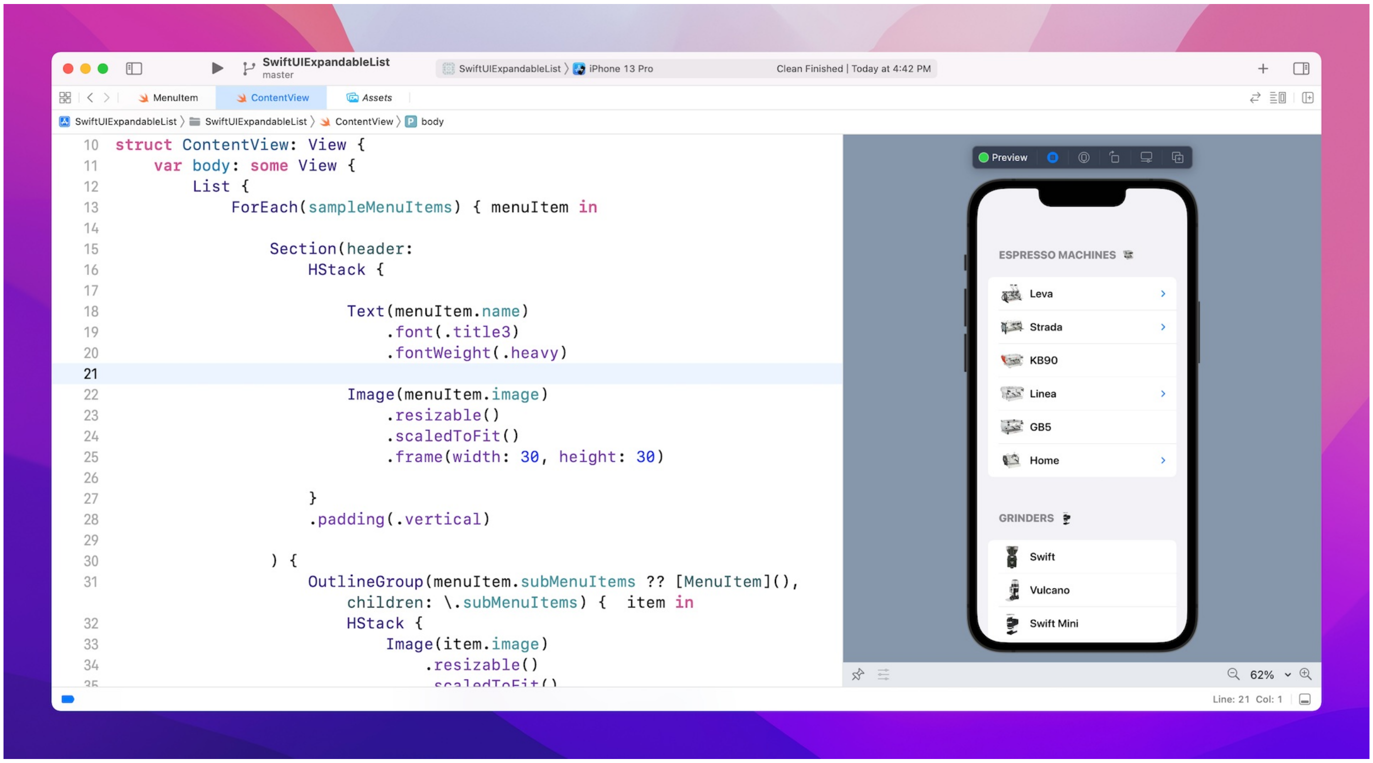


Figure 4. Building the outline view using OutlineGroup

Similarly, if you prefer to use the plain list style, you can attach the `listStyle` modifier to the `List` view:

```
.listStyle(.plain)
```

You then achieve an outline view like figure 5.

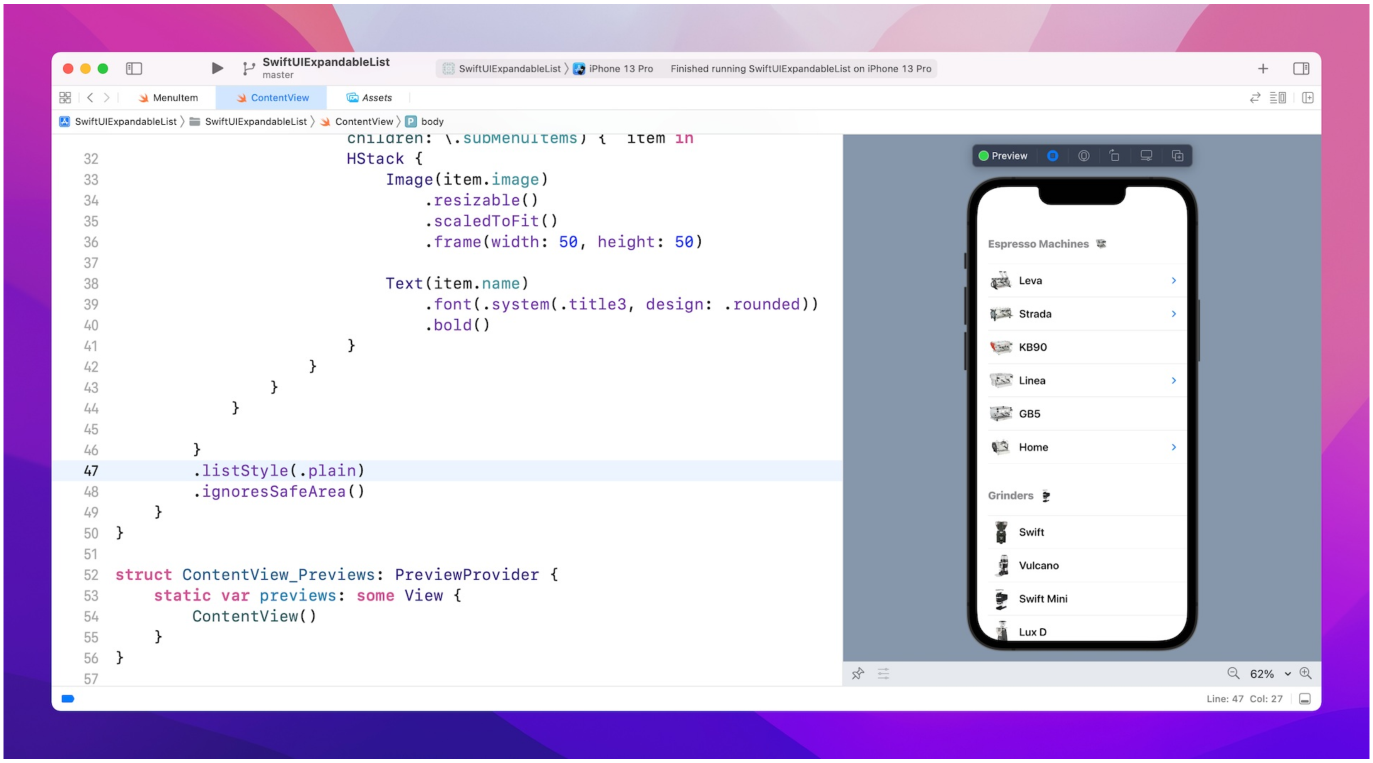


Figure 5. Applying the inset grouped list style

Understanding DisclosureGroup

In the outline view, you can show/hide the sub menu items by tapping the disclosure indicator. Whether you use `List` or `OutlineGroup` to implement the expandable list, this "expand & collapse" feature is supported by a new view called `DisclosureGroup`, introduced in iOS 14.

The disclosure group view is designed to show or hide another content view. While `DisclosureGroup` is automatically embedded in `OutlineGroup`, you can use this view independently. For example, you can use the following code to show & hide a question and an answer:

```
DisclosureGroup(
    content: {
        Text("Absolutely! You are allowed to reuse the source code in your own pro
jects (personal/commercial). However, you're not allowed to distribute or sell the
 source code without prior authorization.")
            .font(.body)
            .fontWeight(.light)
    },
    label: {
        Text("1. Can I reuse the source code?")
            .font(.body)
            .bold()
            .foregroundColor(.black)
    }
)
```

The disclosure group view takes in two parameters: *label* and *content*. In the code above, we specify the question in the `label` parameter and the answer in the `content` parameter. Figure 6 shows you the result.

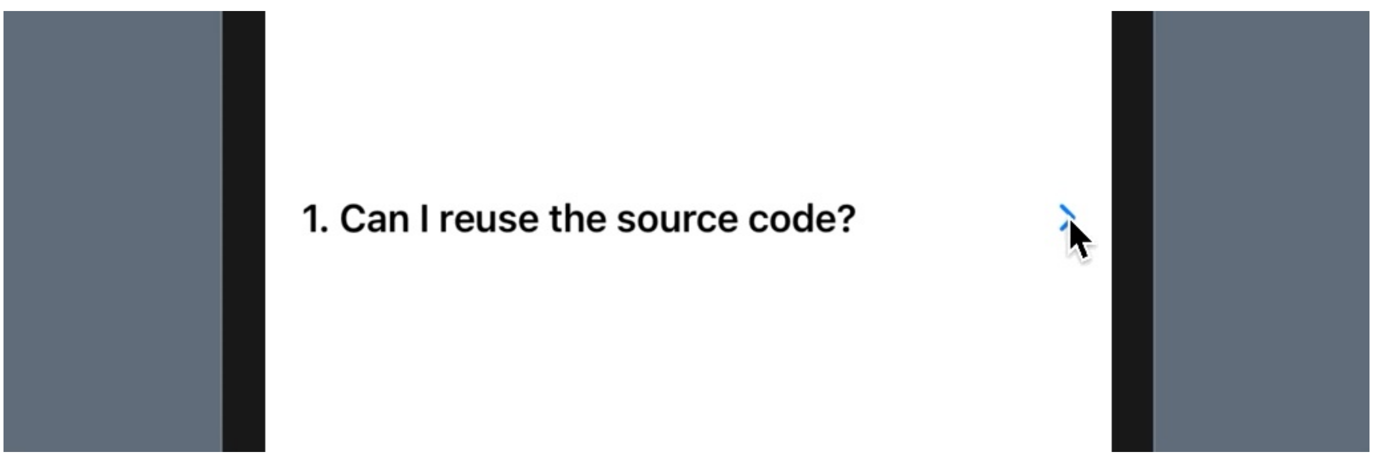


Figure 6. Using DisclosureGroup for showing and hiding content

By default, the disclosure group view is in hidden mode. To reveal the content view, you tap the disclosure indicator to switch the disclosure group view to the "expand" state.

Optionally, you can control the state of `DisclosureGroup` by passing it a binding which specifies the state of the disclosure indicator (expanded or collapsed) like this:

```
struct FaqView: View {
    @State var showContent = true

    var body: some View {
        DisclosureGroup(
            isExpanded: $showContent,
            content: {
                ...
            },
            label: {
                ...
            }
        )
        .padding()
    }
}
```

Exercise

The `DisclosureGroup` view allows you to have finer control over the state of the disclosure indicator. Your exercise is to create a FAQ screen similar to the one shown in figure 7.

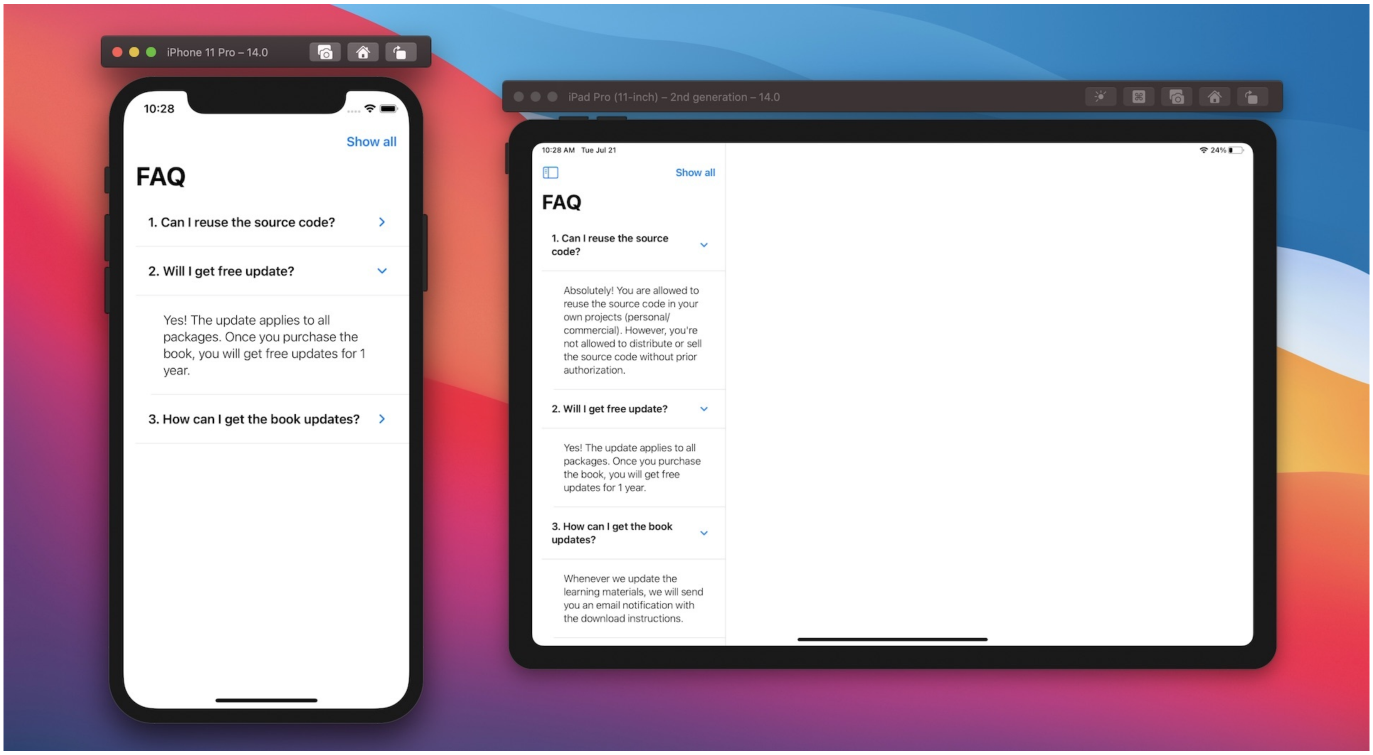


Figure 7. Your exercise

Users can tap the disclosure indicator to show or hide an individual question. Additionally, the app provides a "Show All" button to expand all questions and reveal the answers at once.

Summary

In this chapter, I've introduced a couple of new features of SwiftUI. As you can see in the demo, it is very easy to build an outline view or expandable list view. All you need to do is define a correct data model. The List view handles the rest, traverses the data structure, and renders the outline view. On top of that, the new update provides `OutlineGroup` and `DisclosureGroup` for you to further customize the outline view.

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

Chapter 29
Building Grid Layouts Using LazyVGrid and LazyHGrid

The initial release of SwiftUI didn't come with a native collection view. You can either build your own solution or use third party libraries. In WWDC 2020, Apple introduced tons of new features for the SwiftUI framework. One of them is to address the need for grid views. SwiftUI now provides developers two new UI components called *LazyVGrid* and *LazyHGrid*. One is for creating vertical grids and the other is for horizontal grids. The word *Lazy*, as mentioned by Apple, refers to the grid view not creating items until they are needed. What this means to you is that the performance of these grid views are already optimized by default.

In this chapter, I will walk you through how to create both horizontal and vertical views. Both *LazyVGrid* and *LazyHGrid* are designed to be flexible, so that developers can easily create various types of grid layouts. We will also look into how to vary the size of grid items to achieve different layouts. After you manage the basics, we will dive a little bit deeper and create complex layouts like that shown in figure 1.

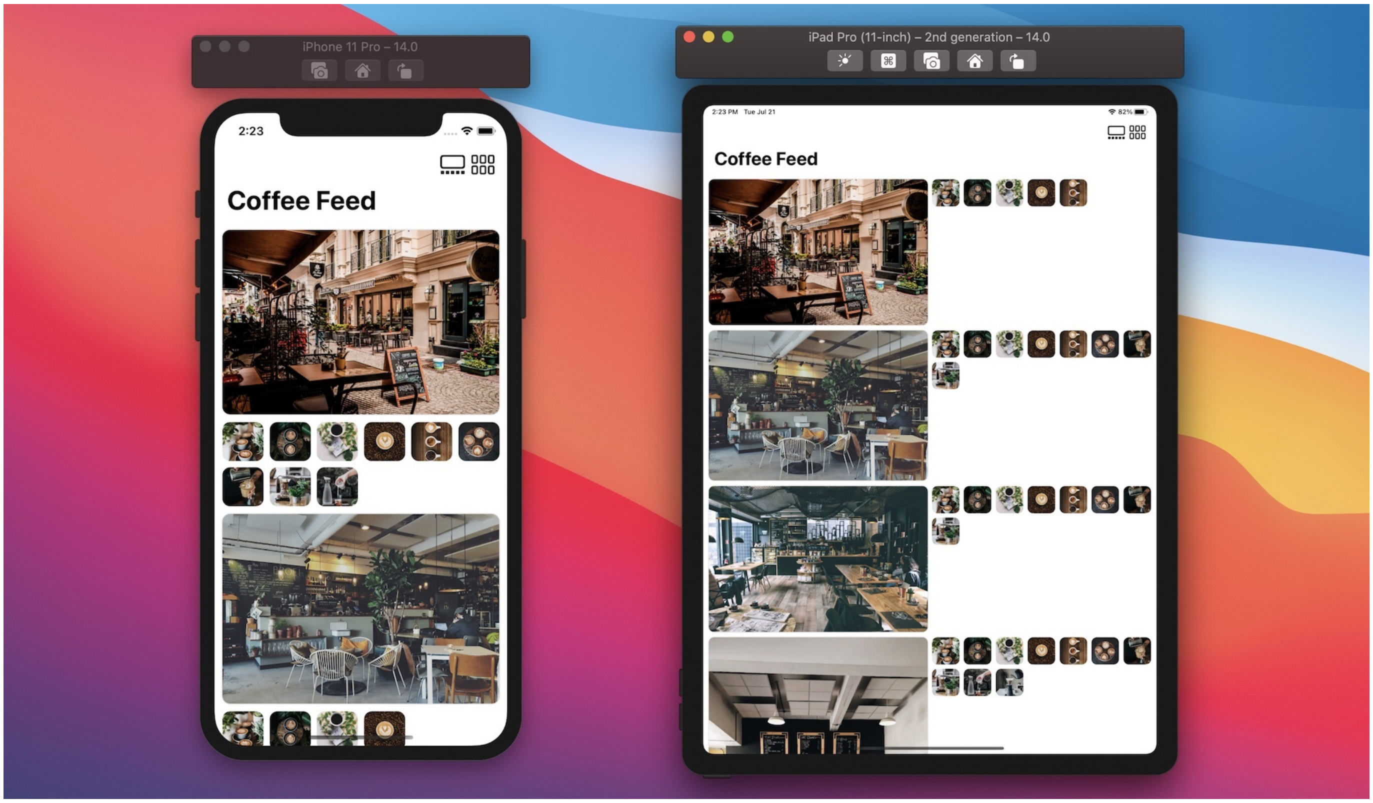


Figure 1. Sample grid layouts

The Essential of Grid Layout in SwiftUI

To create a grid layout, whether it's horizontal or vertical, here are the steps you follow:

1. First, you need to prepare the raw data for presentation in the grid. For example, here is an array of SF symbols that we are going to present in the demo app:

   ```
   private var symbols = ["keyboard", "hifispeaker.fill", "printer.fill", "tv.fill",
   "desktopcomputer", "headphones", "tv.music.note", "mic", "plus.bubble", "video"]
   ```

2. Create an array of type `GridItem` that describes what the grid will look like. Including, how many columns the grid should have. Here is a code snippet for describing a 3-column grid:

```
private var threeColumnGrid = [GridItem(.flexible()), GridItem(.flexible()), GridI
tem(.flexible())]
```

3. Next, you layout the grid by using `LazyVGrid` and `ScrollView`. Here is an example:

```
ScrollView {
    LazyVGrid(columns: threeColumnGrid) {
        // Display the item
    }
}
```

4. Alternatively, if you want to build a horizontal grid, you use `LazyHGrid` like this:

```
ScrollView(.horizontal) {
    LazyHGrid(rows: threeColumnGrid) {
        // Display the item
    }
}
```

Using LazyVGrid to Create Vertical Grids

With a basic understanding of the grid layout, let's put the code to work. We will start with something simple by building a 3-column grid. Open Xcode and create a new project with the *App* template. Please make sure you select *SwiftUI* for the Interface option. Name the project *SwiftUIGridLayout* or whatever name you prefer.

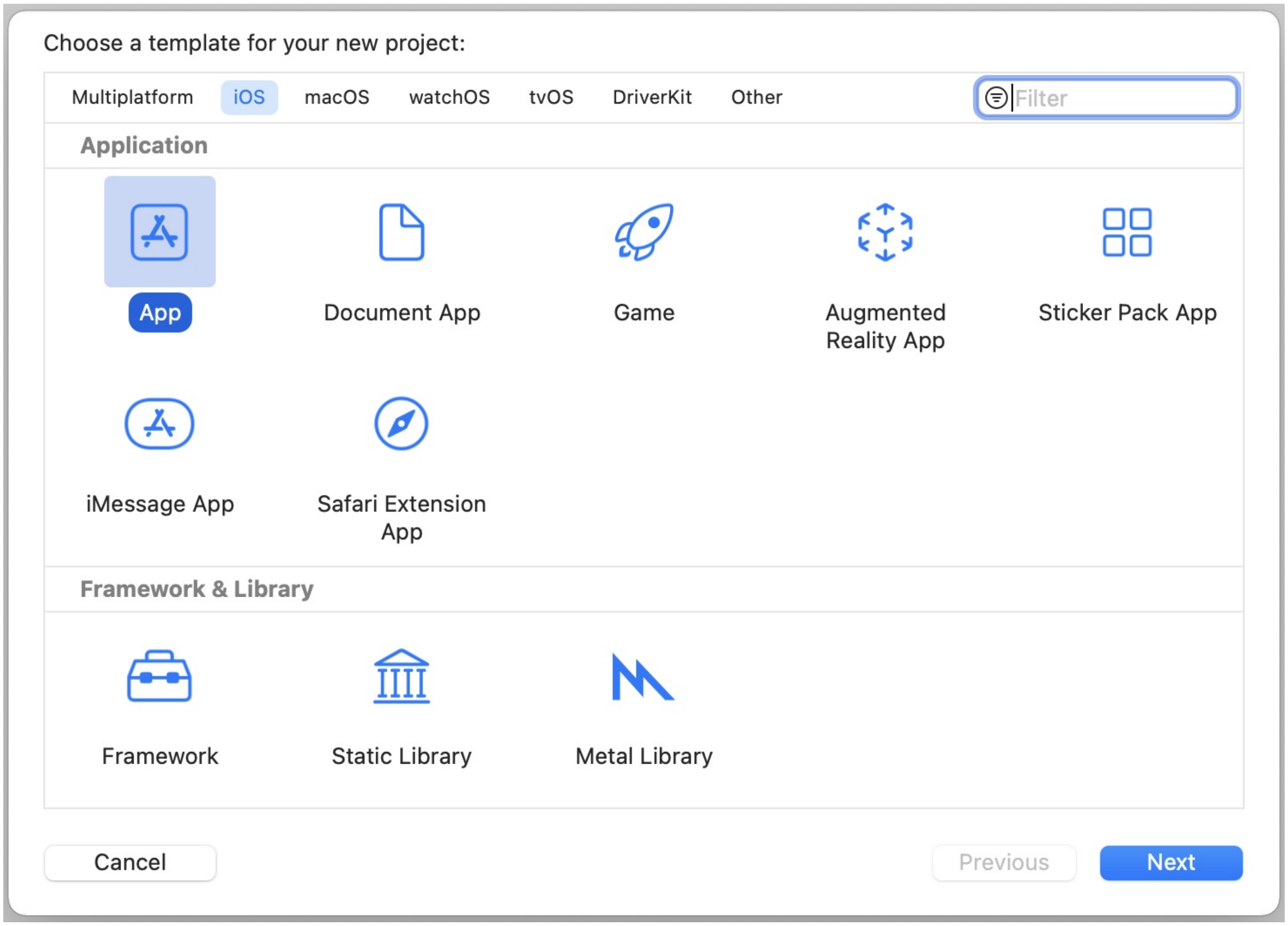


Figure 2. Creating a new project using the App template

Once the project is created, choose `ContentView.swift`. In `ContentView`, declare the following variables:

```
private var symbols = ["keyboard", "hifispeaker.fill", "printer.fill", "tv.fill",
"desktopcomputer", "headphones", "tv.music.note", "mic", "plus.bubble", "video"]

private var colors: [Color] = [.yellow, .purple, .green]

private var gridItemLayout = [GridItem(.flexible()), GridItem(.flexible()), GridIt
em(.flexible())]
```

We are going to display a set of SF symbols in a 3-column grid. To present the grid, update the `body` variable like this:

```
var body: some View {
    ScrollView {
        LazyVGrid(columns: gridItemLayout, spacing: 20) {
            ForEach((0...9999), id: \.self) {
                Image(systemName: symbols[$0 % symbols.count])
                    .font(.system(size: 30))
                    .frame(width: 50, height: 50)
                    .background(colors[$0 % colors.count])
                    .cornerRadius(10)
            }
        }
    }
}
```

We use `LazyVGrid` and tell the vertical grid to use a 3-column layout. We also specify that there is a 20 point space between rows. In the code block, we have a `ForEach` loop to present a total of 10,000 image views. If you've made the change correctly, you should see a three column grid in the preview.

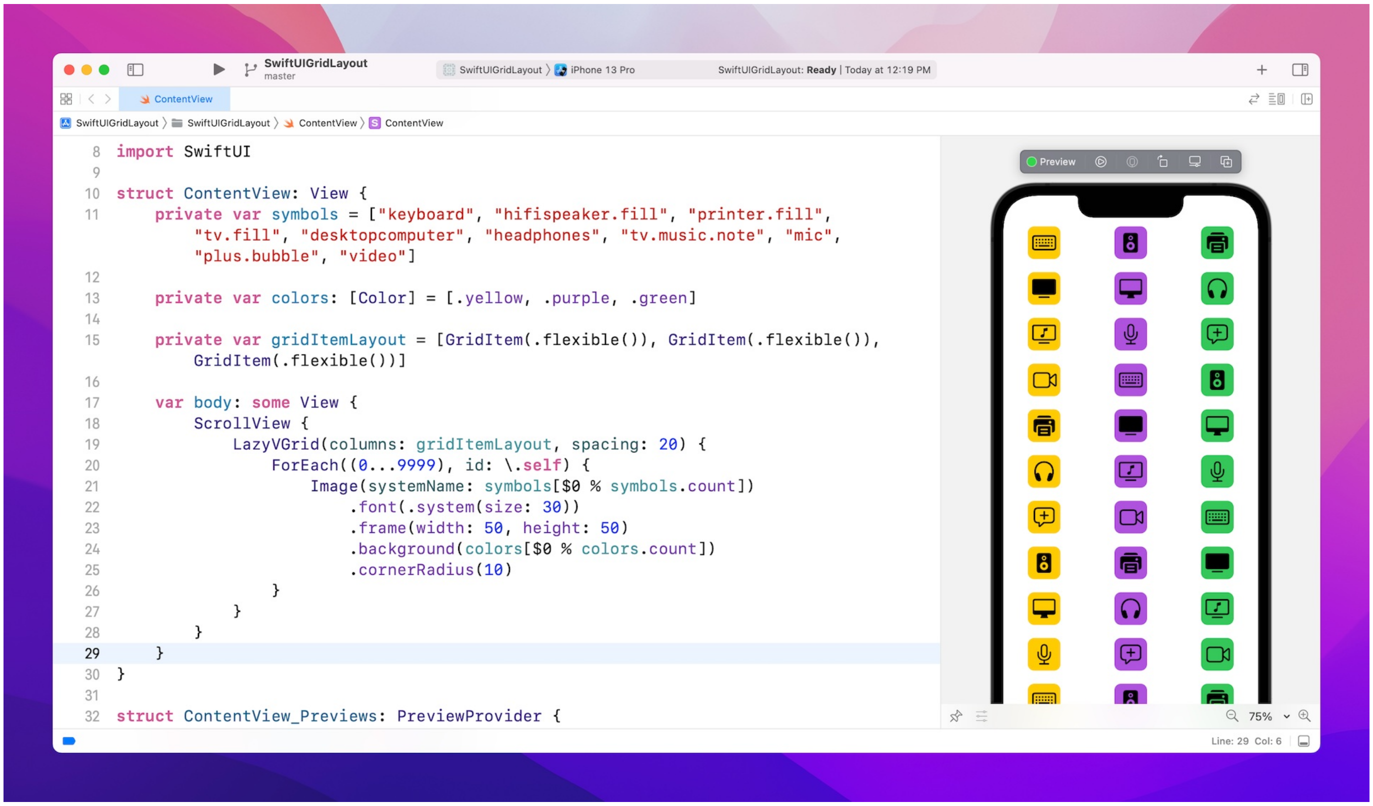


Figure 3. Displaying a 3-column grid

This is how we create a vertical grid with three columns. The frame size of the image is fixed to 50 by 50 points, which is controlled by the `.frame` modifier. If you want to make a grid item wider, you can alter the frame modifier like this:

```
.frame(minWidth: 0, maxWidth: .infinity, minHeight: 50)
```

The image's width will expand to take up the column's width like that shown in figure 4.

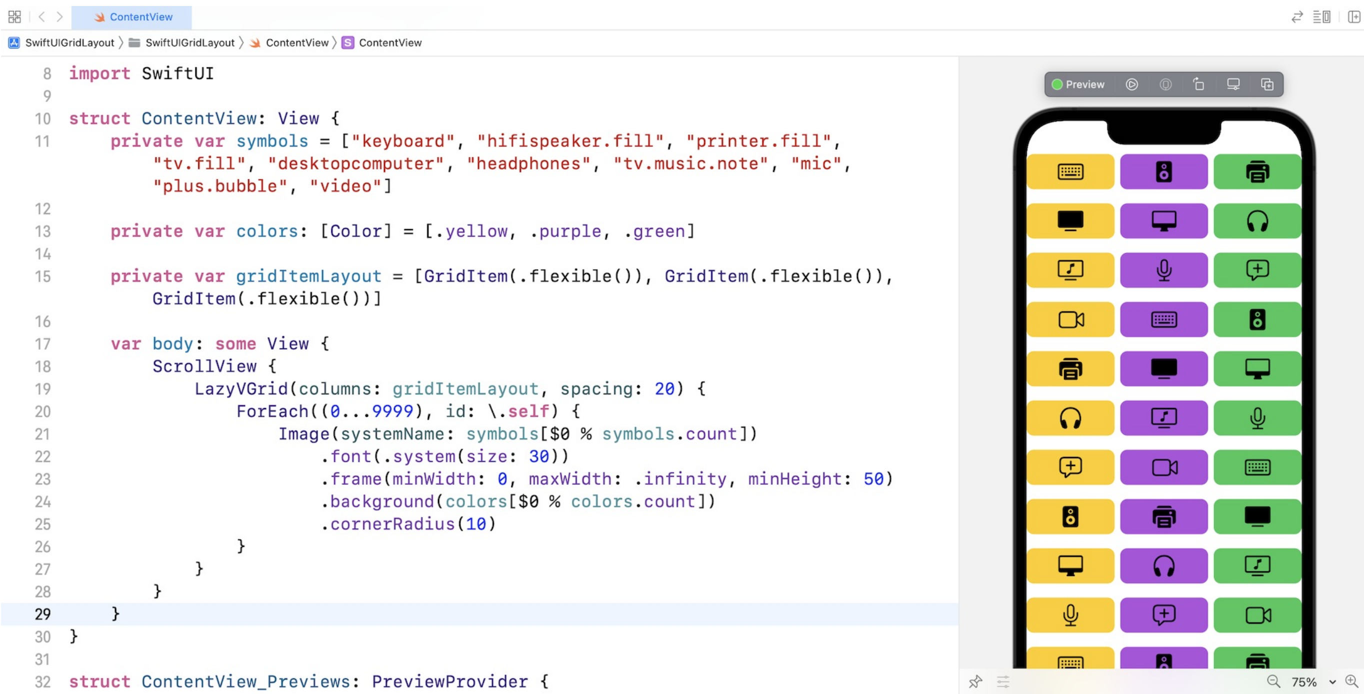


Figure 4. Changing the frame size of the grid items

Note that there is a space between the columns and rows. Sometimes, you may want to create a grid without any spaces. How can you achieve that? The space between rows is controlled by the `spacing` parameter of `LazyVGrid`. We have set its value to `20` points. You can simply change it to `0` such that there is no space between rows.

The spacing between grid items is controlled by the instances of `GridItem` initialized in `gridItemLayout`. You can set the spacing between items by passing a value to the `spacing` parameter. Therefore, to remove the spacing between rows, you can initialize the `gridLayout` variable like this:

```
private var gridItemLayout = [GridItem(.flexible(), spacing: 0), GridItem(.flexible(), spacing: 0), GridItem(.flexible(), spacing: 0)]
```

For each `GridItem`, we specify to use a spacing of zero. For simplicity, the code above can be rewritten like this:

```
private var gridItemLayout = Array(repeating: GridItem(.flexible(), spacing: 0), count: 3)
```

If you've made both changes, your preview canvas should show you a grid view without any spacing.

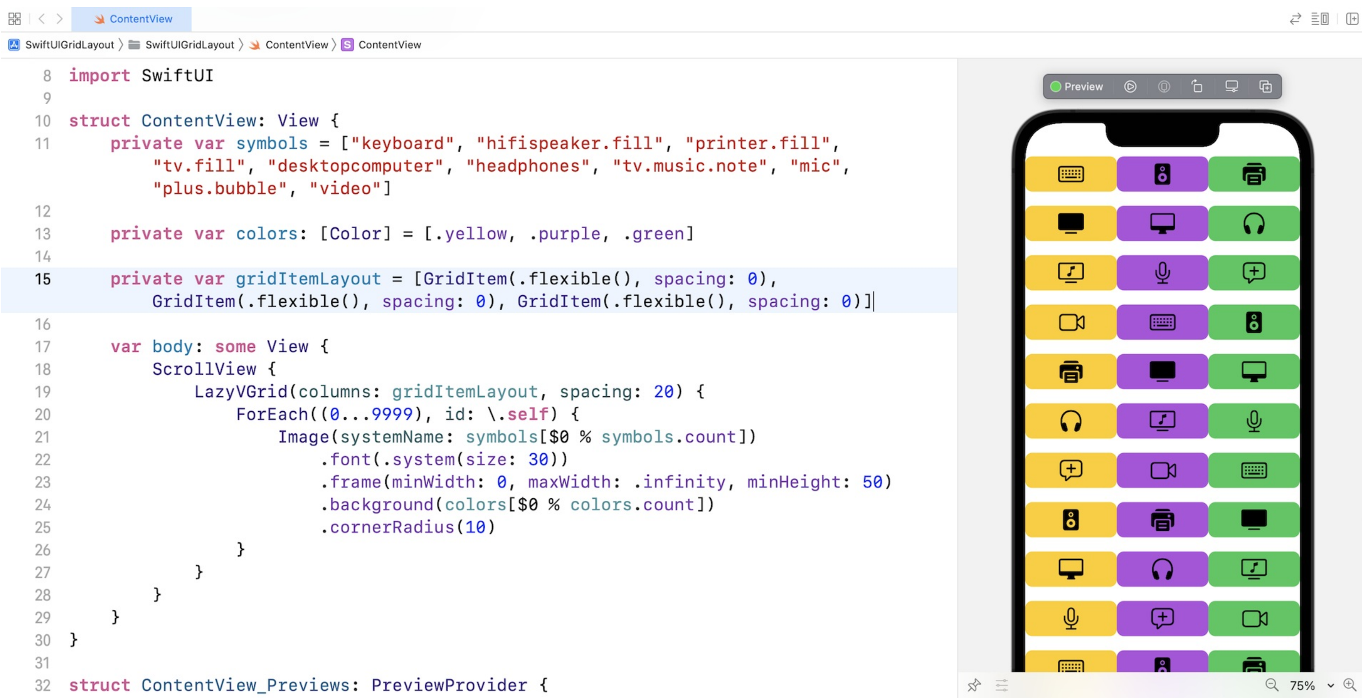


Figure 5. Removing the spacing between columns and rows

Using GridItem to Vary the Grid Layout (Flexible/Fixed/Adaptive)

Let's take a further look at `GridItem`. You use `GridItem` instances to configure the layout of items in `LazyHGrid` and `LazyVGrid` views. Earlier, we defined an array of three `GridItem` instances, each of which uses the size type `.flexible()`. The flexible size type enables you to create three columns with equal size. If you want to describe a 6-column grid, you can create the array of `GridItem` like this:

```
private var sixColumnGrid: [GridItem] = Array(repeating: .init(.flexible()), count
: 6)
```

`.flexible()` is just one of the size types for controlling the grid layout. If you want to place as many items as possible in a row, you can use the *adaptive* size type:

```
private var gridItemLayout = [GridItem(.adaptive(minimum: 50))]
```

The *adaptive* size type requires you to specify the minimize size for a grid item. In the code above, each grid item has a minimum size of 50. If you modify the `gridItemLayout` variable as above and set the spacing of `LazyVGrid` back to `20`, you should achieve a grid layout similar to the one shown in figure 6.

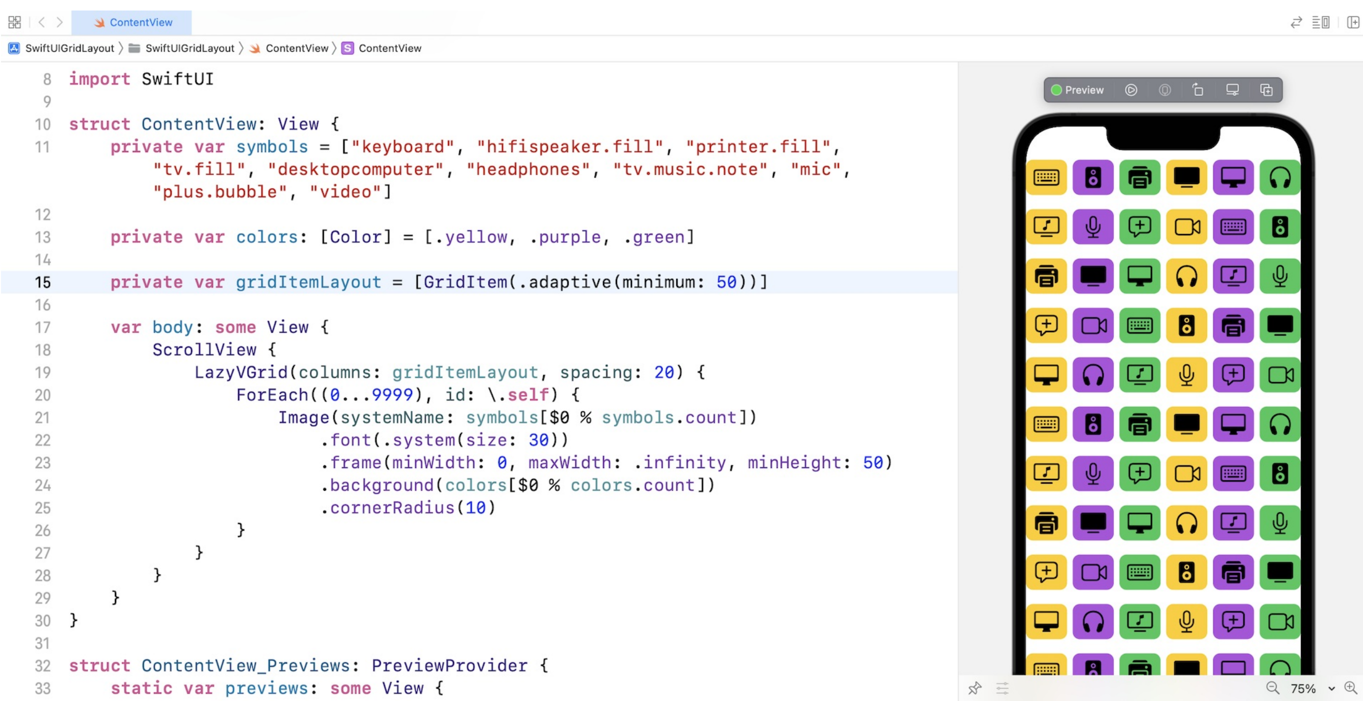


Figure 6. Using adaptive size to create the grid

By using `.adaptive(minimum: 50)`, you instruct `LazyVGrid` to fill as many images as possible in a row such that each item has a minimum size of 50 points.

Note: I used iPhone 13 Pro as the simulator. If you use other iOS simulators with different screen sizes, you may achieve a different result.

In addition to `.flexible` and `.adaptive`, you can also use `.fixed` if you want to create fixed width columns. For example, you want to layout the image in two columns such that the first column has a width of 100 points and the second one has a width of 150 points. You write the code like this:

```
private var gridItemLayout = [GridItem(.fixed(100)), GridItem(.fixed(150))]
```

Update the `gridItemLayout` variable as shown above, this will result in a two-column grid with different sizes.

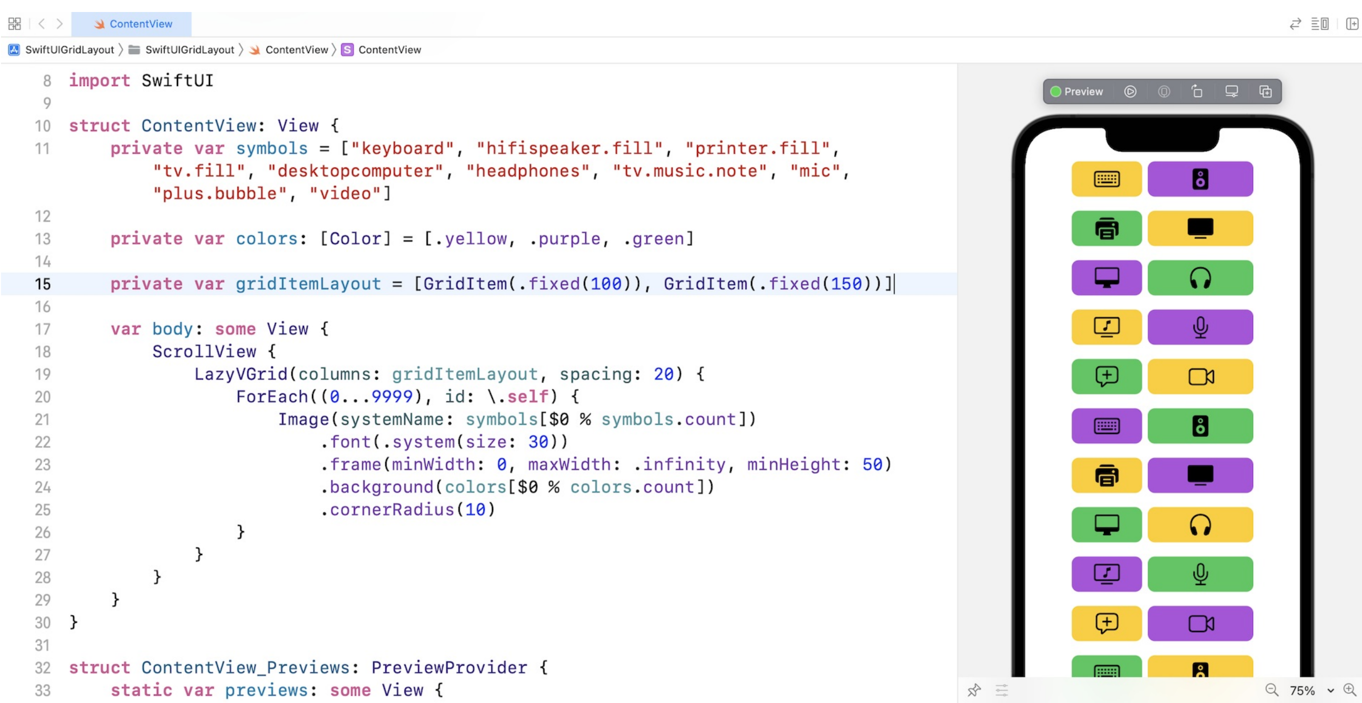


Figure 7. A grid with fixed-size items

You are allowed to mix different size types to create more complex grid layouts. For example, you can define a fixed size `GridItem`, followed by a `GridItem` with an adaptive size like this:

```
private var gridItemLayout = [GridItem(.fixed(150)), GridItem(.adaptive(minimum: 50
))]
```

In this case, `LazyVGrid` creates a fixed size column of 100 point width. And then, it tries to fill as many items as possible within the remaining space.

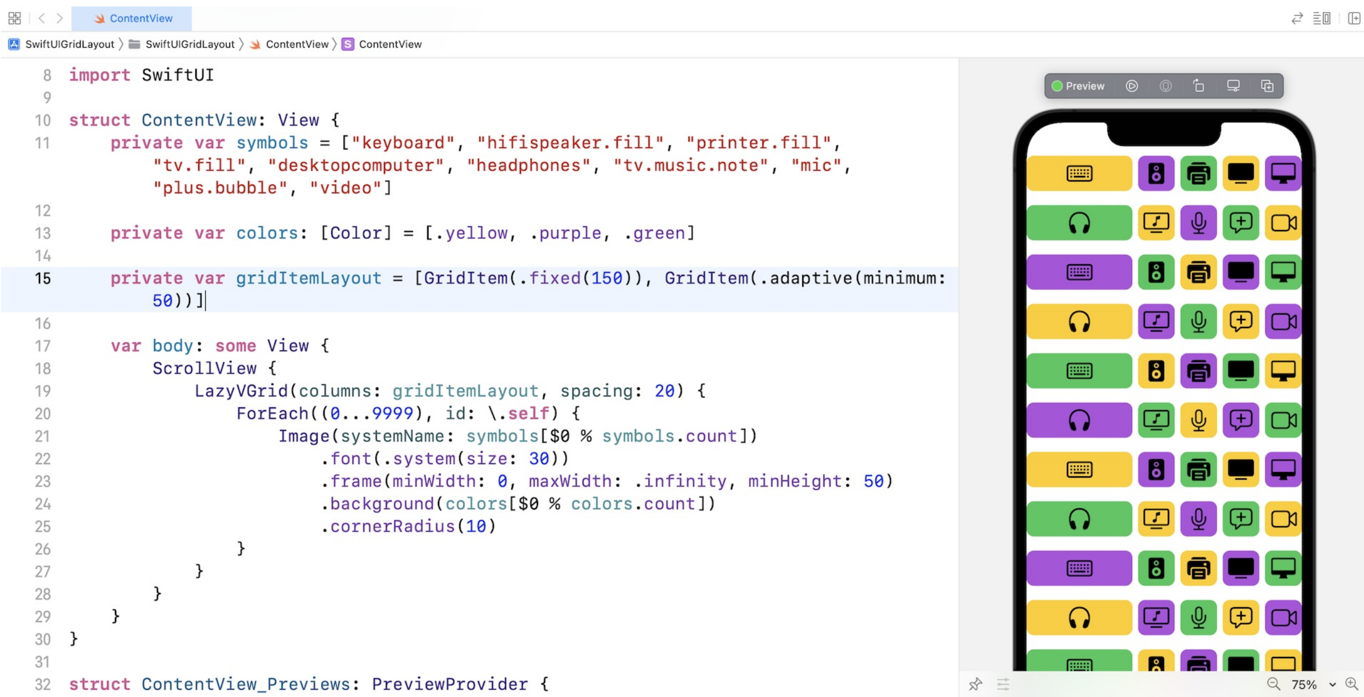


Figure 8. Mixing a fixed-size item with adaptive size items

Using LazyHGrid to Create Horizontal Grids

Now that you've created a vertical grid, `LazyHGrid` has made it so easy to convert a vertical grid to a horizontal one. The usage of horizontal grid is nearly the same as `LazyVGrid` except that you embed it in a horizontal scroll view. Furthermore, `LazyHGrid` takes in a parameter named `rows` instead of `columns`.

Therefore, you can rewrite a couple lines of code to transform a grid view from vertical orientation to horizontal:

```
ScrollView(.horizontal) {
    LazyHGrid(rows: gridItemLayout, spacing: 20) {
        ForEach((0...9999), id: \.self) {
            Image(systemName: symbols[$0 % symbols.count])
                .font(.system(size: 30))
                .frame(minWidth: 0, maxWidth: .infinity, minHeight: 50, maxHeight: .infinity)
                .background(colors[$0 % colors.count])
                .cornerRadius(10)
        }
    }
}
```

Run the demo in the preview or test it on a simulator. You should see a horizontal grid.

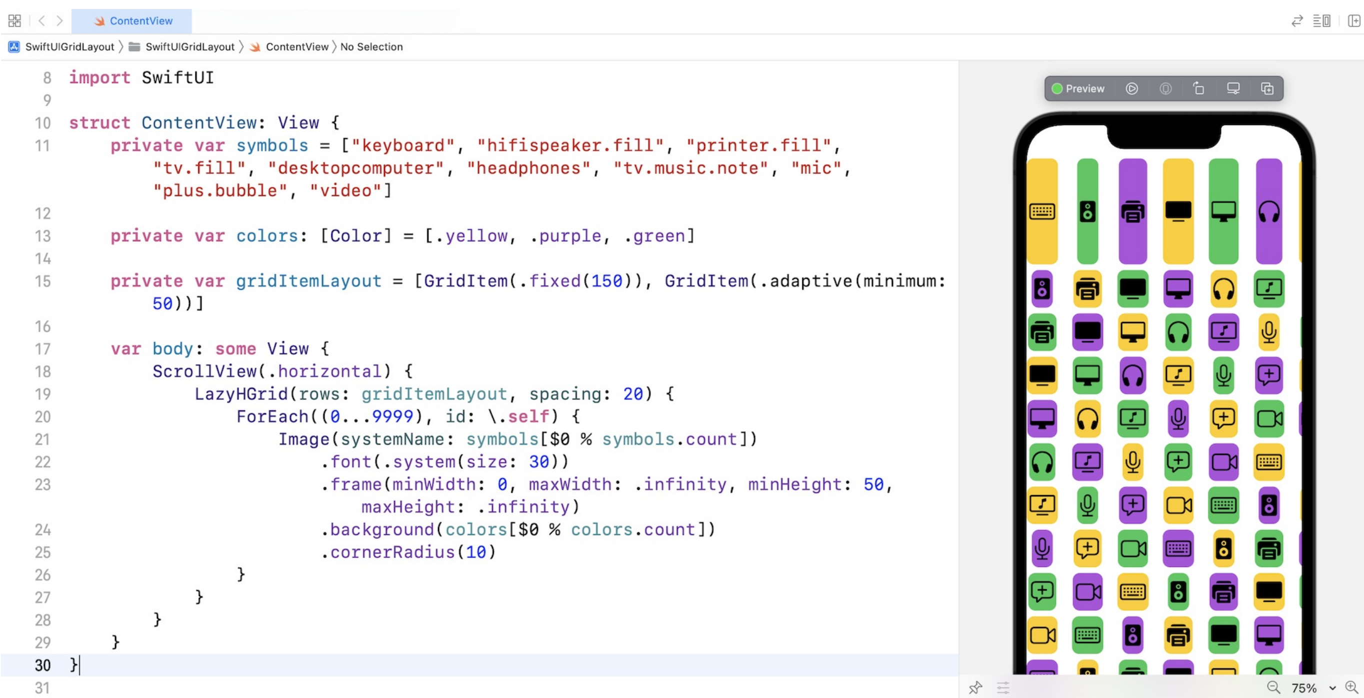


Figure 9. Creating a horizontal grid with LazyHGrid

Switching Between Different Grid Layouts

Now that you have some experience with LazyVGrid and LazyHGrid, let's create something more complicated. Imagine you are going to build a photo app that displays a collection of coffee photos. In the app, it provides a feature for users to change the layout. By default, it shows the list of photos in a single column. The user can tap a *Grid* button to change the list view to a grid view with 2 columns. Tap the same button again for a 3-column layout, followed by a 4-column layout.

Figure 10. Creating a horizontal grid with LazyHGrid

Create a new project for this demo app. Again, choose the *App* template and name the project *SwiftUIPhotoGrid*. Next, download the image pack at https://www.appcoda.com/resources/swiftui/coffeeimages.zip. Unzip the images and add them to the asset catalog.

Before creating the grid view, we will create the data model for the collection of photos. In the project navigator, right click *SwiftUIGridView* and choose *New file...* to create a new file. Select the *Swift File* template and name the file *Photo.swift*.

Insert the following code in the `Photo.swift` file to create the `Photo` struct:

```
struct Photo: Identifiable {
    var id = UUID()
    var name: String
}

let samplePhotos = (1...20).map { Photo(name: "coffee-\($0)") }
```

We have 20 coffee photos in the image pack, so we initialize an array of 20 `Photo` instances. With the data model ready, let's switch over to `ContentView.swift` to build the grid.

First, declare a `gridLayout` variable to define our preferred grid layout:

```
@State var gridLayout: [GridItem] = [ GridItem() ]
```

By default, we want to display a list view. Other than using `List`, you can actually use `LazyVGrid` to build a list view. We do this by defining the `gridLayout` with one grid item. By telling `LazyVGrid` to use a single column grid layout, it will arrange the items like a list view. Insert the following code in `body` to create the grid view:

```
NavigationView {
    ScrollView {
        LazyVGrid(columns: gridLayout, alignment: .center, spacing: 10) {

            ForEach(samplePhotos.indices) { index in

                Image(samplePhotos[index].name)
                    .resizable()
                    .scaledToFill()
                    .frame(minWidth: 0, maxWidth: .infinity)
                    .frame(height: 200)
                    .cornerRadius(10)
                    .shadow(color: Color.primary.opacity(0.3), radius: 1)

            }
        }
        .padding(.all, 10)
    }

    .navigationTitle("Coffee Feed")
}
```

We use `LazyVGrid` to create a vertical grid with a spacing of 10 points between rows. The grid is used to display coffee photos, so we use `ForEach` to loop through the `samplePhotos` array. We embed the grid in a scroll view to make it scrollable and wrap it with a navigation view. Once you have made the change, you should see a list of photos in the preview canvas.

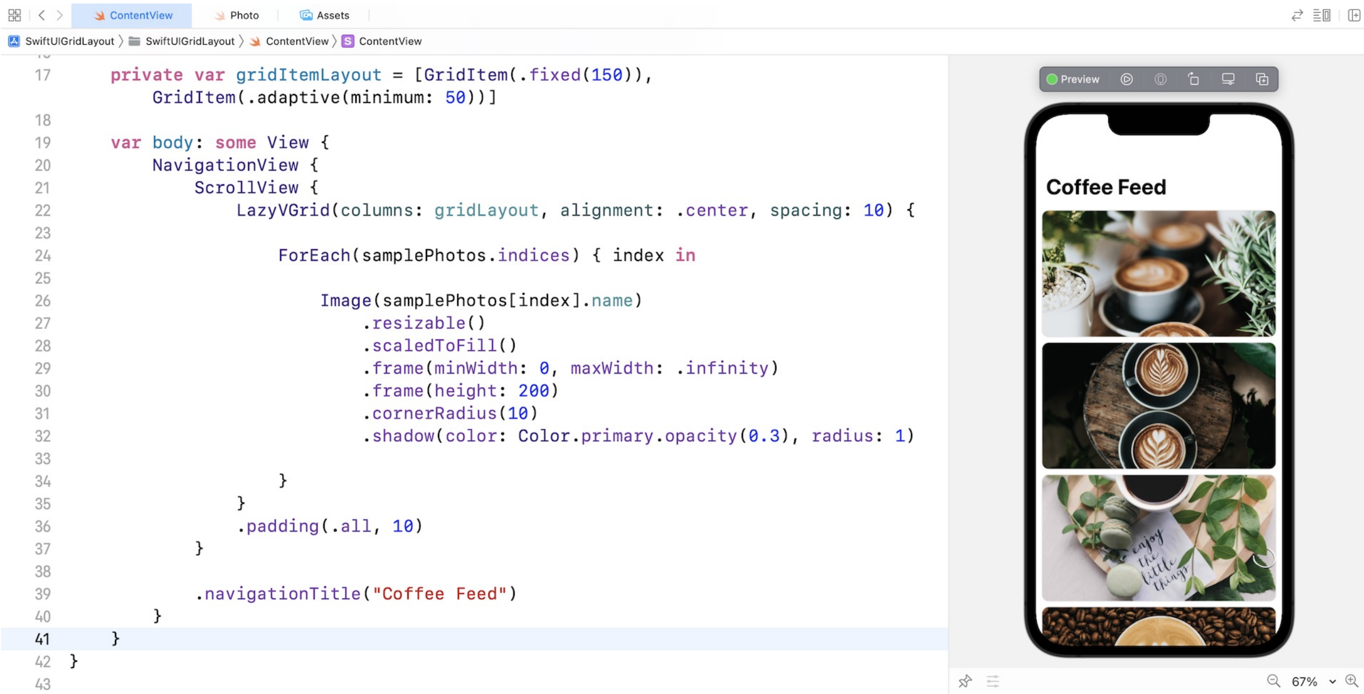


Figure 11. Creating a list view with LazyVGrid

Now we need to a button for users to switch between different layouts. We will add the button to the navigation bar. In iOS 14, Apple introduced a new modifier called `.toolbar` for you to populate items within the navigation bar. Right after `.navigationTitle`, insert the following code to create the bar button:

```
.toolbar {
    ToolbarItem(placement: .navigationBarTrailing) {
        Button(action: {
            self.gridLayout = Array(repeating: .init(.flexible()), count: self.gri
dLayout.count % 4 + 1)
        }) {
            Image(systemName: "square.grid.2x2")
                .font(.title)
                .foregroundColor(.primary)
        }
    }
}
```

In the code above, we update the `gridLayout` variable and initialize the array of `GridItem`. Let's say the current item count is one, we will create an array of two `GridItem`s to change to a 2-column grid. Since we've marked `gridLayout` as a state variable, SwiftUI will render the grid view every time we update the variable.

Figure 12. Adding a bar button for switching the grid layout

You can run the app to have a quick test. Tapping the grid button will switch to another grid layout.

There are a couple of things we want to improve. First, the height of the grid item should be adjusted to 100 points for grids with two or more columns. Update the `.frame` modifier with the `height` parameter like this:

```
.frame(height: gridLayout.count == 1 ? 200 : 100)
```

Second, when you switch from one grid layout to another, SwiftUI simply redraws the grid view without any animation. Wouldn't it be great if we added a nice transition between layout changes? To do that, you just add a single line of code. Insert the following code after `.padding(.all, 10)` :

```
.animation(.interactiveSpring(), value: gridLayout.count)
```

This is the power of SwiftUI. By telling SwiftUI that you want to animate changes, the framework handles the rest and you will see a nice transition between the layout changes.

Figure 13. SwiftUI automatically animates the transition

Building Grid Layout with Multiple Grids

You are not limited to using a single `LazyVGrid` or `LazyHGrid` in your app. By combining more than one `LazyVGrid`, you are able to build some interesting layouts. Take a look at figure 14. We are going to create this kind of grid layout. The grid displays a list of cafe photos. Under each cafe photo, it shows a list of coffee photos. When the device is in landscape orientation, the cafe photo and the list of coffee photos will be arranged side by side.

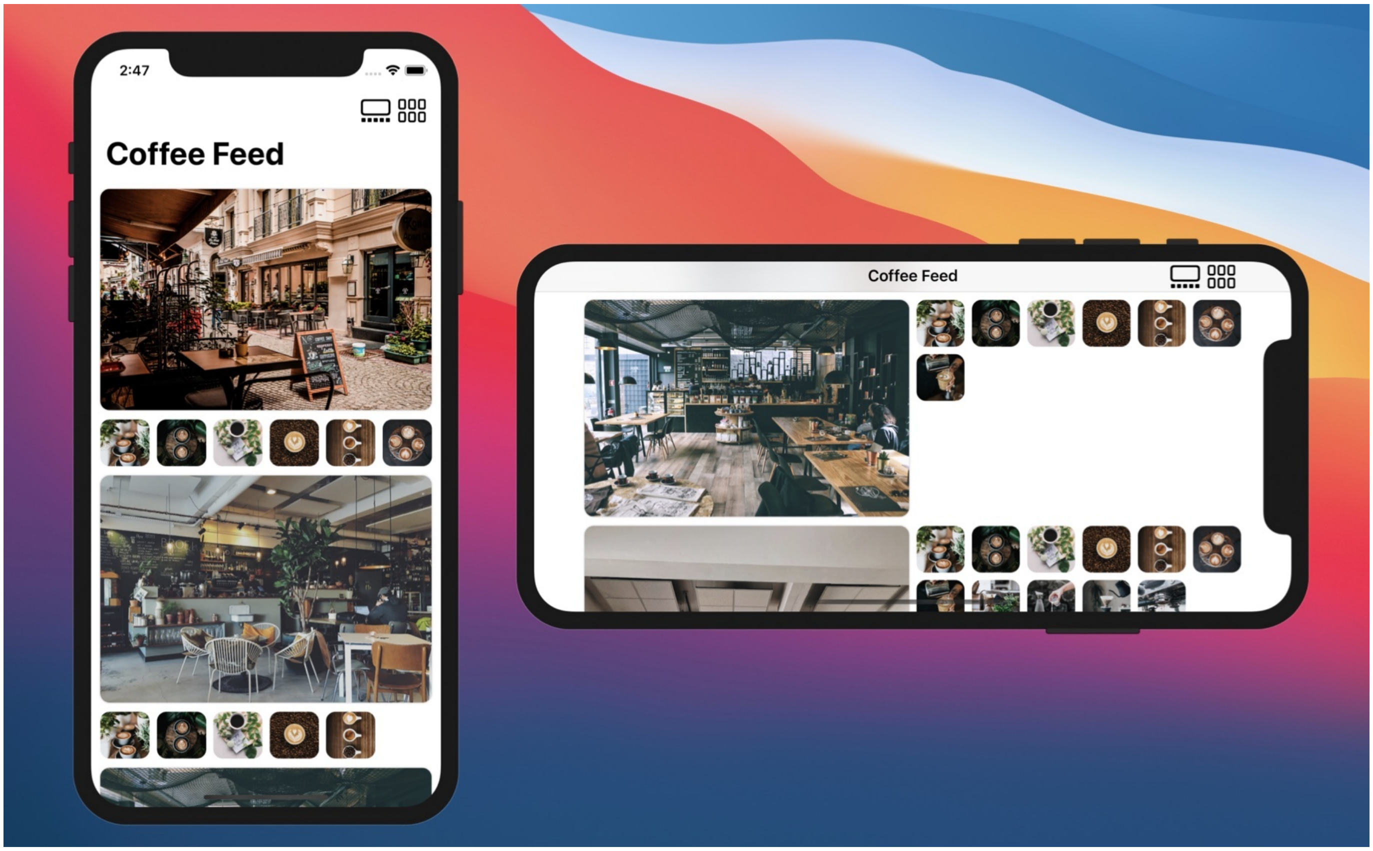


Figure 14. Building complex grid layout with two grids

Let's go back to our Xcode project and create the data model first. The image pack you downloaded earlier comes a set of cafe photos. So, create a new Swift file and name it *Cafe.swift*. In the file, insert the following code:

```
struct Cafe: Identifiable {
    var id = UUID()
    var image: String
    var coffeePhotos: [Photo] = []
}

let sampleCafes: [Cafe] = {

    var cafes = (1...18).map { Cafe(image: "cafe-\($0)") }

    for index in cafes.indices {
        let randomNumber = Int.random(in: (2...12))
        cafes[index].coffeePhotos = (1...randomNumber).map { Photo(name: "coffee-\
($0)") }
    }

    return cafes
}()
```

The `Cafe` struct is self explanatory. It has an `image` property for storing the cafe photo and the `coffeePhotos` property for storing a list of coffee photos. In the code above, we also create an array of `Cafe` for demo purposes. For each cafe, we randomly pick some coffee photos. Please feel free to modify the code if you have other images you prefer.

Instead of modifying the `ContentView.swift` file, let's create a new file for implementing this grid view. Right click *SwiftUIPhotoGrid* and choose *New File...*. Select the *SwiftUI View* template and name the file *MultiGridView*.

Similar to the earlier implementation, let's declare a `gridLayout` variable to store the current grid layout:

```
@State var gridLayout = [ GridItem() ]
```

By default, our grid is initialized with one `GridItem`. Next, insert the following code in `body` to create a vertical grid with a single column:

```
NavigationView {
    ScrollView {
        LazyVGrid(columns: gridLayout, alignment: .center, spacing: 10) {

            ForEach(sampleCafes) { cafe in
                Image(cafe.image)
                    .resizable()
                    .scaledToFill()
                    .frame(minWidth: 0, maxWidth: .infinity)
                    .frame(maxHeight: 150)
                    .cornerRadius(10)
                    .shadow(color: Color.primary.opacity(0.3), radius: 1)
            }

        }
        .padding(.all, 10)
        .animation(.interactiveSpring(), value: gridLayout.count)
    }
    .navigationTitle("Coffee Feed")
}
```

I don't think we need to go through the code again because it's almost the same as the code we wrote earlier. If your code works properly, you should see a list view that shows the collection of cafe photos.

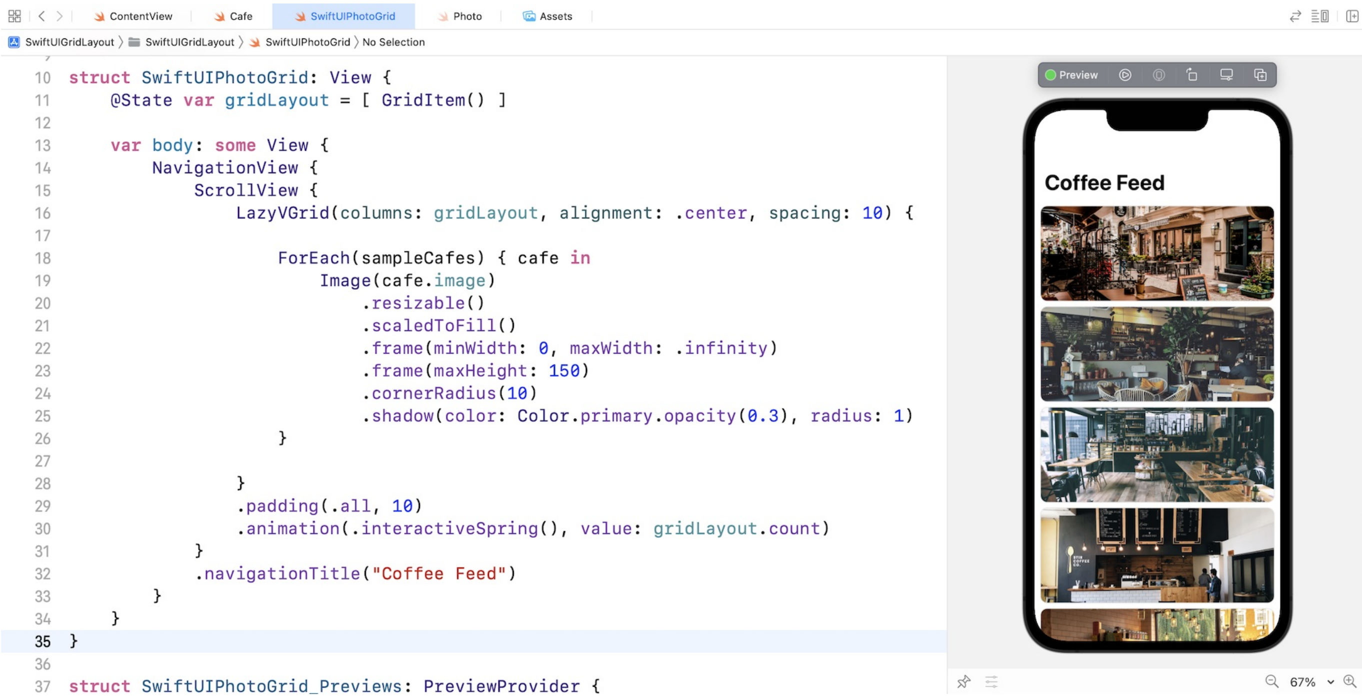


Figure 15. A list of cafe photos

Adding an Additional Grid

How do we display another grid under each of the cafe photos? All you need to do is add another `LazyVGrid` inside the `ForEach` loop. Insert the following code after the `Image` view of the loop:

```
LazyVGrid(columns: [GridItem(.adaptive(minimum: 50))]) {
    ForEach(cafe.coffeePhotos) { photo in
        Image(photo.name)
            .resizable()
            .scaledToFill()
            .frame(minWidth: 0, maxWidth: .infinity)
            .frame(height: 50)
            .cornerRadius(10)
    }
}
.frame(minHeight: 0, maxHeight: .infinity, alignment: .top)
.animation(.easeIn, value: gridLayout.count)
```

Here we create another vertical grid for the coffee photos. By using the *adaptive* size type, this grid will fill as many photos as possible in a row. Once you make the code change, the app UI will look like that shown in figure 16.

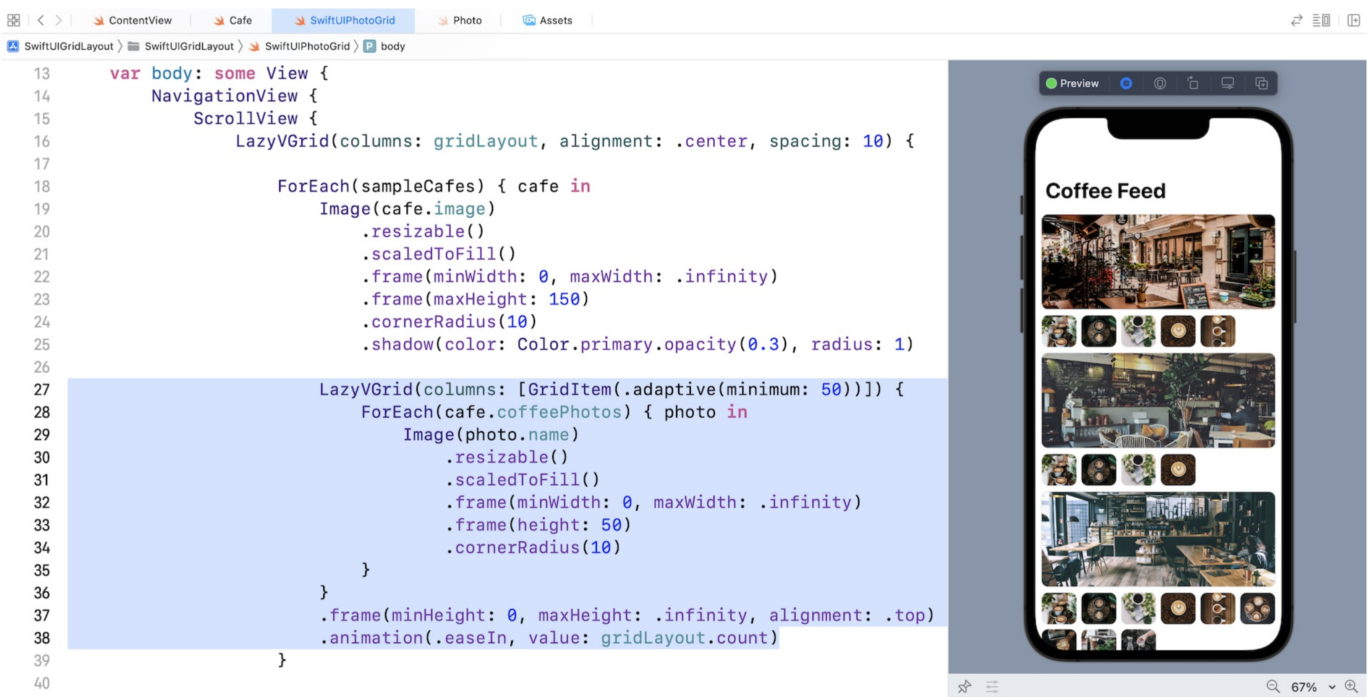


Figure 16. Adding another grid for the coffee photos

If you prefer to arrange the cafe and coffee photos side by side, you can modify the `gridLayout` variable like this:

```
@State var gridLayout = [ GridItem(.adaptive(minimum: 100)), GridItem(.flexible())
 ]
```

As soon as you change the `gridLayout` variable, your preview will be updated to display the cafe and coffee photos side by side.

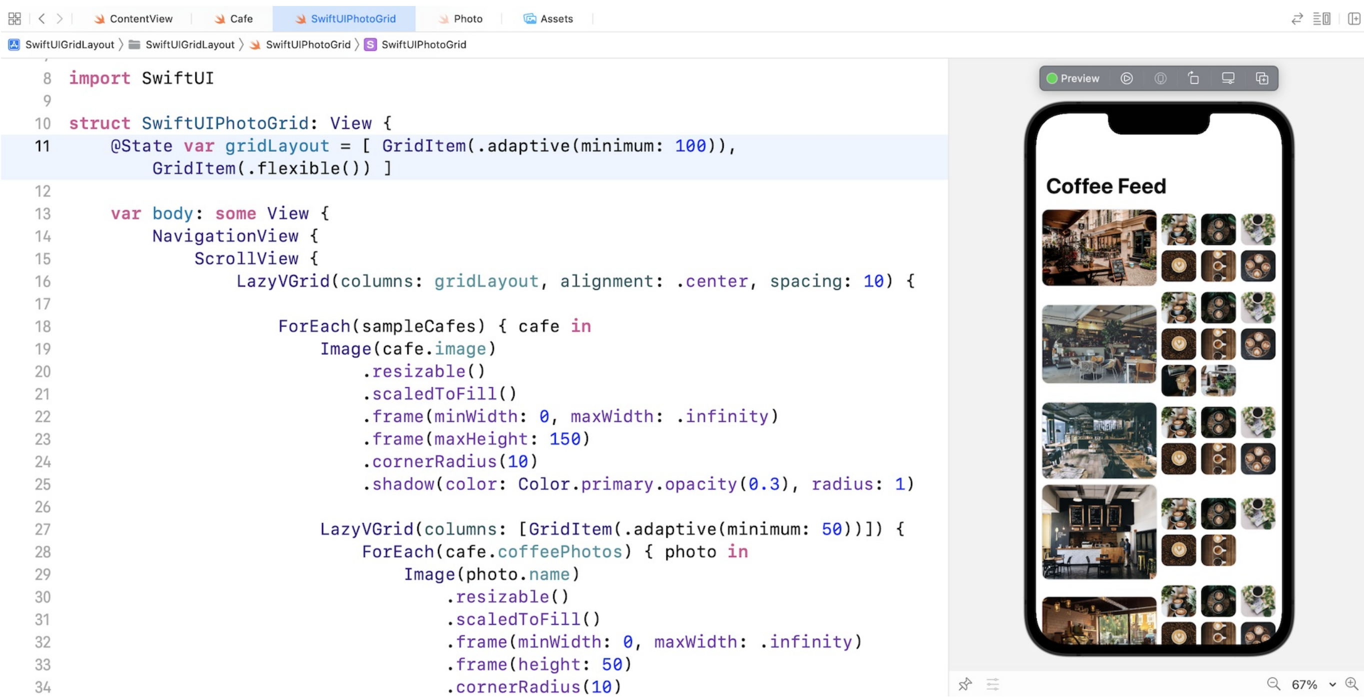


Figure 17. Arrange the cafe and coffee photos side by side

Handling Landscape Orientation

To test the app in landscape orientation, you need to run it on a simulator. The preview canvas doesn't allow you to rotate the device yet.

Before you run the app, you will need to perform a simple modification in `SwiftUIGridLayoutApp.swift`. Since we have created a new file for implementing this multi-grid, modify the view in `WindowGroup` from `ContentView()` to `MultiGridView()` like below:

```
struct SwiftUIGridLayoutApp: App {
    var body: some Scene {
        WindowGroup {
            SwiftUIPhotoGrid()
        }
    }
}
```

Now you're ready to run the app in an iPhone simulator. It works great in the portrait orientation just like the preview canvas. However, if you rotate the simulator sideways by pressing command-left (or right), the grid layout doesn't look as expected. What we expect is that it should look pretty much the same as that in portrait mode.

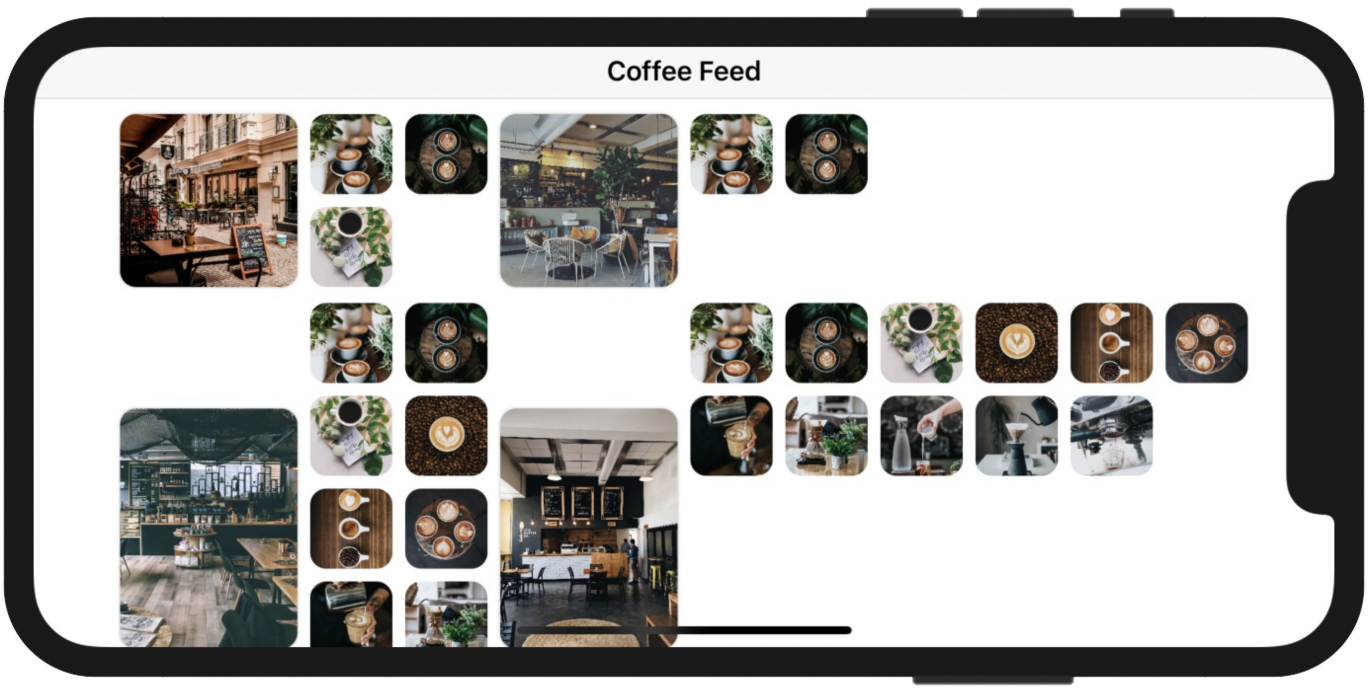


Figure 18. The app UI in landscape mode

To fix the issue, we can adjust the minimum width of the adaptive grid item and make it a bit wider when the device is in landscape orientation. The question is how can you detect the orientation changes?

In SwiftUI, every view comes with a set of environment variables. You can find out the current device orientation by accessing both the horizontal and vertical size class variables like this:

```
@Environment(\.horizontalSizeClass) var horizontalSizeClass: UserInterfaceSizeClass
?
@Environment(\.verticalSizeClass) var verticalSizeClass: UserInterfaceSizeClass?
```

The `@Environment` property wrapper allows you to access the environment values. In the code above, we tell SwiftUI that we want to read both the horizontal and vertical size classes, and subscribe to their changes. In other words, we will be notified whenever the device's orientation changes.

If you haven't done so, please make sure you insert the code above in `SwiftUIPhotoGrid`.

The next question is how do we capture the notification and respond to the changes? In iOS 14 (or later), Apple provided a new modifier called `.onChange()`. You can attach this modifier to any views to monitor any state changes. In this case, we can attach the modifier to `NavigationView` like this:

```
.onChange(of: verticalSizeClass) { value in
    self.gridLayout = [ GridItem(.adaptive(minimum:  verticalSizeClass == .compact
  ? 100 : 250)), GridItem(.flexible()) ]
}
```

We monitor the change of both `horizontalSizeClass` and `verticalSizeClass` variables. Whenever there is a change, we will update the `gridLayout` variable with a new grid configuration. The iPhone has a *compact* height in landscape orientation. Therefore, if the value of `verticalSizeClass` equals `.compact`, we alter the minimum size of the grid item to 250 points.

Now run the app on an iPhone simulator again. When you turn the device sideways, it now shows the cafe photo and coffee photos side by side.

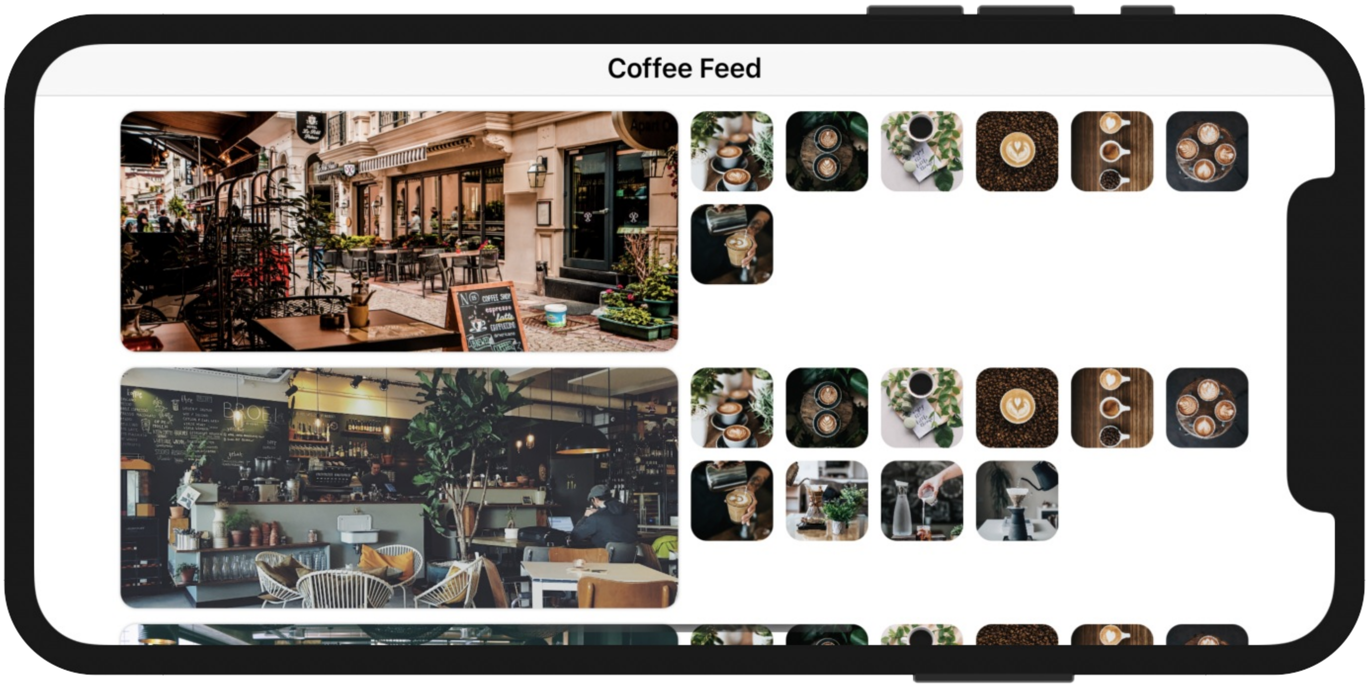


Figure 19. The app UI in landscape mode now looks better

Understanding Navigation View Style

Earlier, I used the iPhone 13 Pro as the simulator. The app works perfectly in both portrait and landscape mode. But when you run the app on iPhone Max models, it looks a bit weird in landscape orientation. iOS automatically turns the view into a primary detail split view. You can only access the grid view by tapping a navigation button at the top-left corner of the screen.

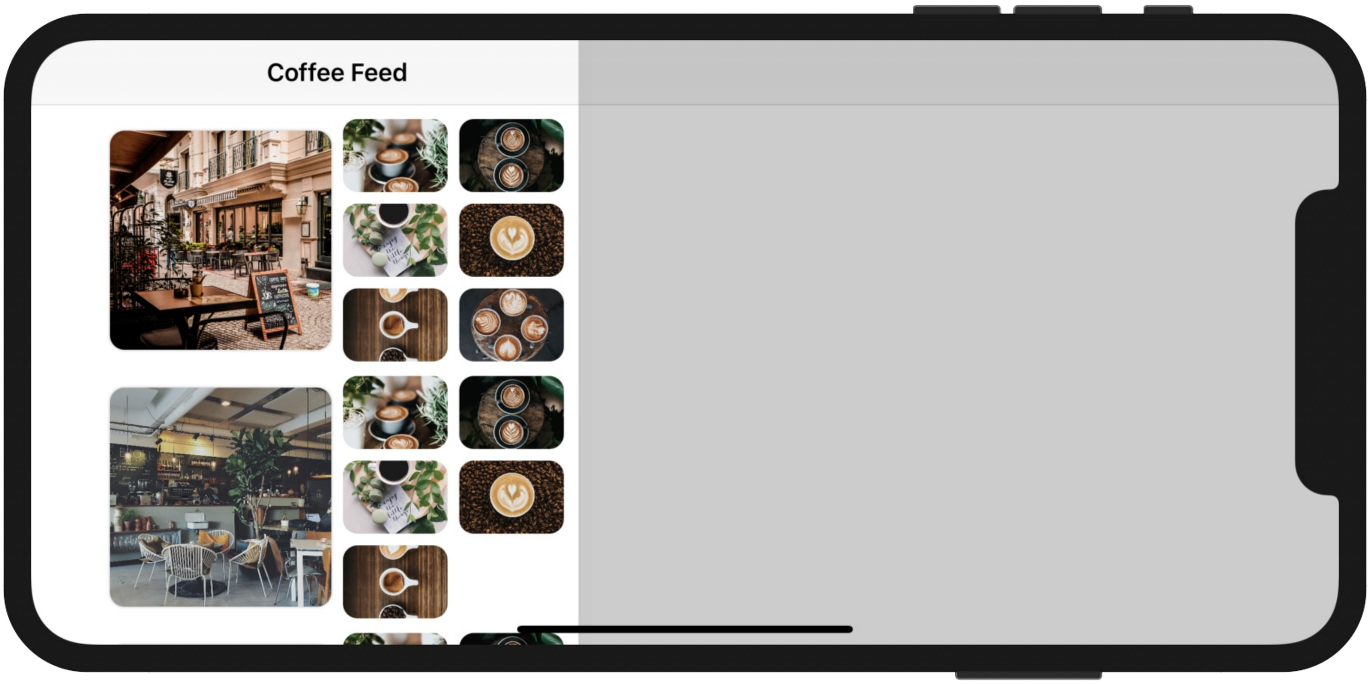


Figure 20. Split view on iPhone 11 Max

This is a default behavior of `NavigationView` on large devices like the iPhone 13 Pro Max. To solve the issue and disable the split-view behavior on iPhone Max models, you can attach the following modifier to the navigation view:

```
.navigationViewStyle(StackNavigationViewStyle())
```

By using the `.navigationViewStyle` modifier, we explicitly instruct `NavigationView` to use the stack navigation view style, regardless of the screen size. Now test the app again on iPhone 13 Max Pro. The app UI should look well even in landscape orientation just like that shown in figure 19.

Exercise

I have a couple of exercises for you. First, the app UI doesn't look good on iPad. Modify the code and fix the issue such that it only shows two columns: one for the cafe photo and the other for the coffee photos.

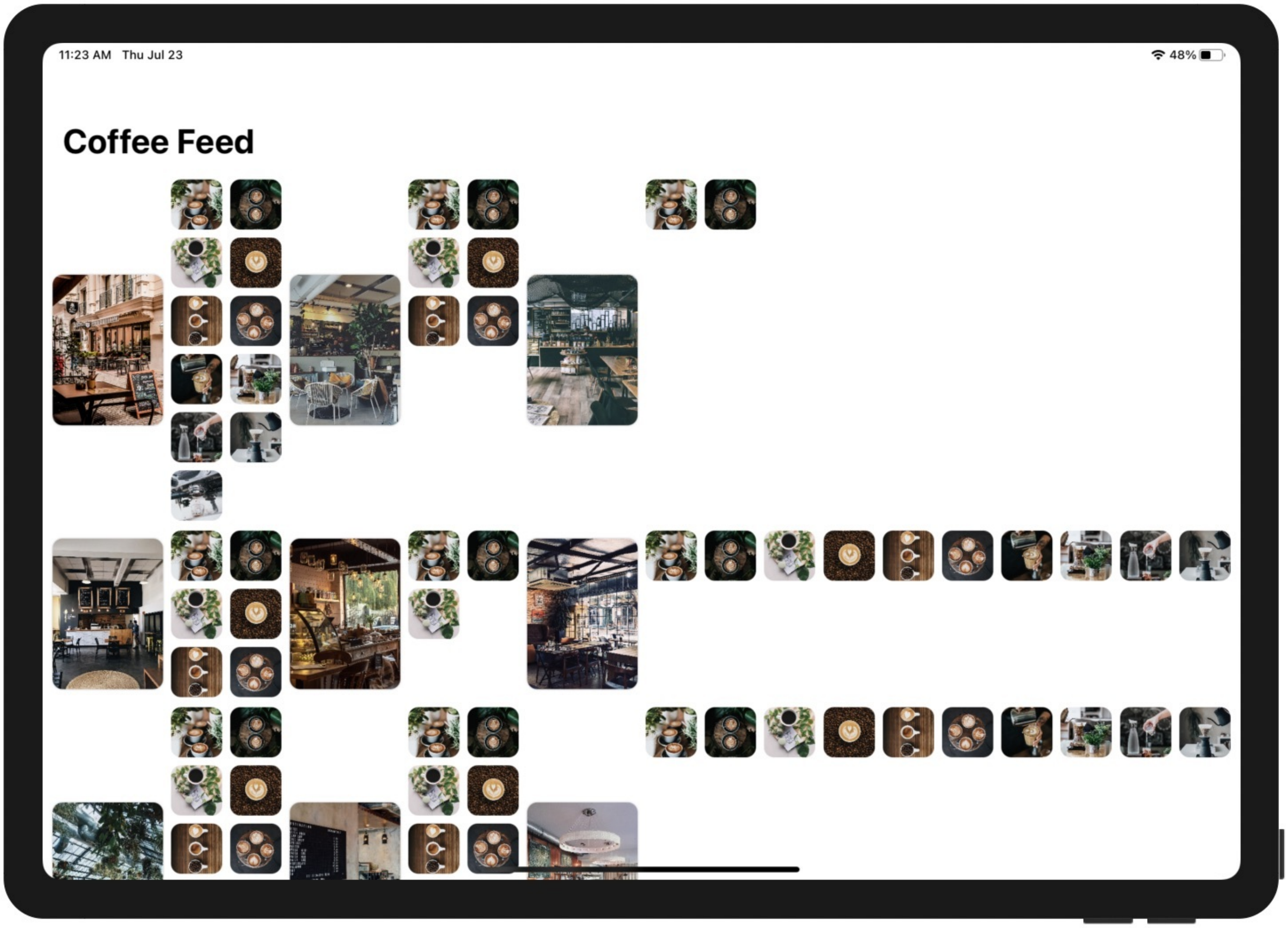


Figure 21. App UI on iPad

The next exercise is more complicated with a number of requirements:

1. **Different default grid layout for iPhone and iPad** - When the app is first loaded up, it displays a single column grid for iPhone in portrait mode. For iPad and iPhone landscape, the app shows the cafe photos in a 2-column grid.
2. **Show/hide button for the coffee photos** - Add a new button in the navigation bar for toggling the display of coffee photos. By default, the app only shows the list of cafe photos. When this button is tapped, it shows the coffee photo grid.
3. **Another button for switching grid layout** - Add another bar button for toggling the grid layout between one and two columns.

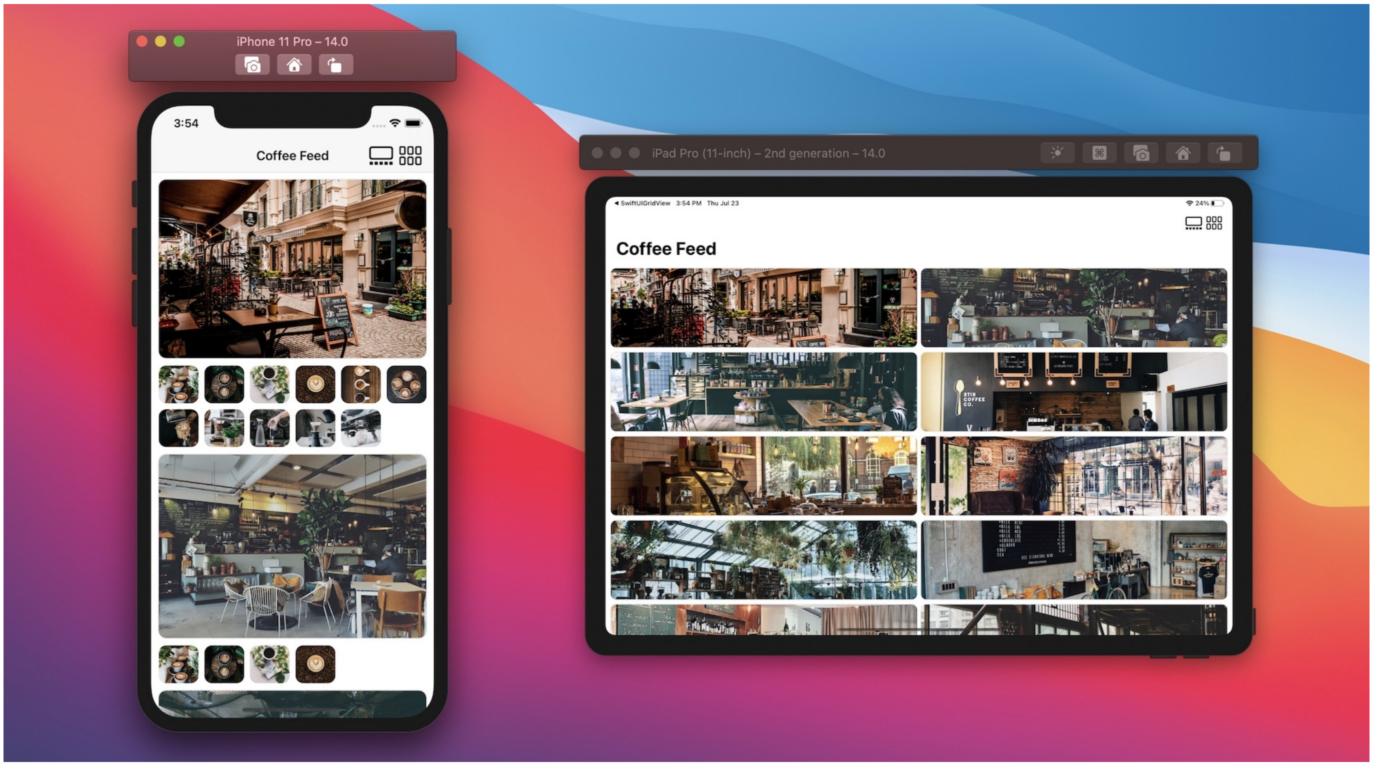


Figure 22. Enhancing the app to support both iPhone and iPad

To help you better understand what the final deliverable looks like, please check out this video demo at https://link.appcoda.com/multigrid-demo.

Summary

The missing collection view in the first release of SwiftUI is now here. The introduction of `LazyVGrid` and `LazyHGrid` in SwiftUI lets developers create different types of grid layouts with a few lines of code. This tutorial is just a quick overview of these two new UI components. I encourage you to try out different configurations of `GridItem` to see what grid layouts you can achieve.

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

Chapter 30
Creating an Animated Activity Ring with Shape and Animatable

The built-in Activity app uses three circular progress bars to show your progress of *Move*, *Exercise*, and *Stand*. The kind of progress bars is usually known as activity ring. Take a look at figure 1 if you haven't used the Activity app or you don't know what an activity ring is. Apple Watch has certainly played a big part that this round progress bar becomes a popular UI pattern.

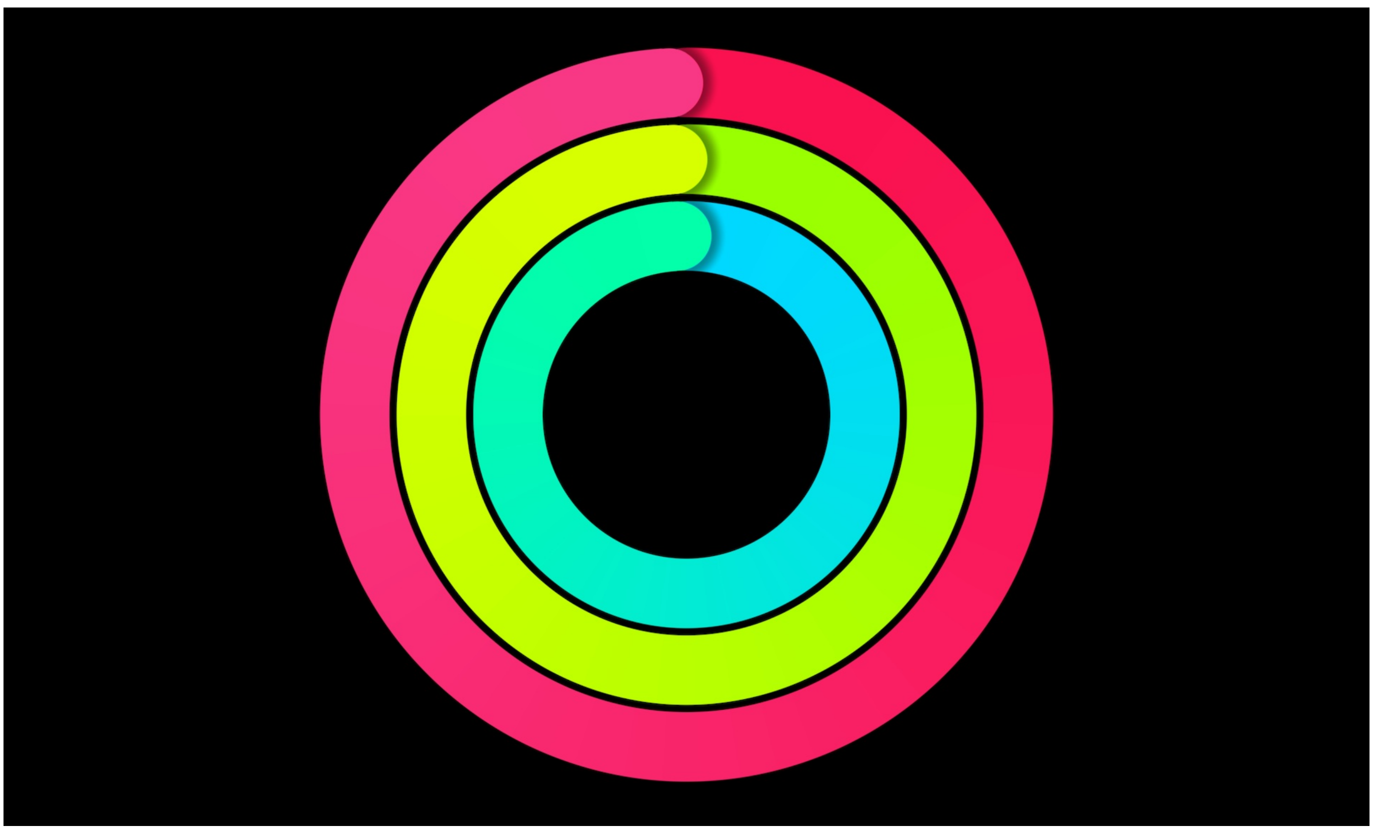


Figure 1. A sample activity ring

In this chapter, we will look into its implementation and build a similar activity ring in SwiftUI. Our goal is not just to create a static activity ring. It will be an animated one that animates the progress change like that shown in figure 2. Or you can check out the demo video at https://link.appcoda.com/progressring.

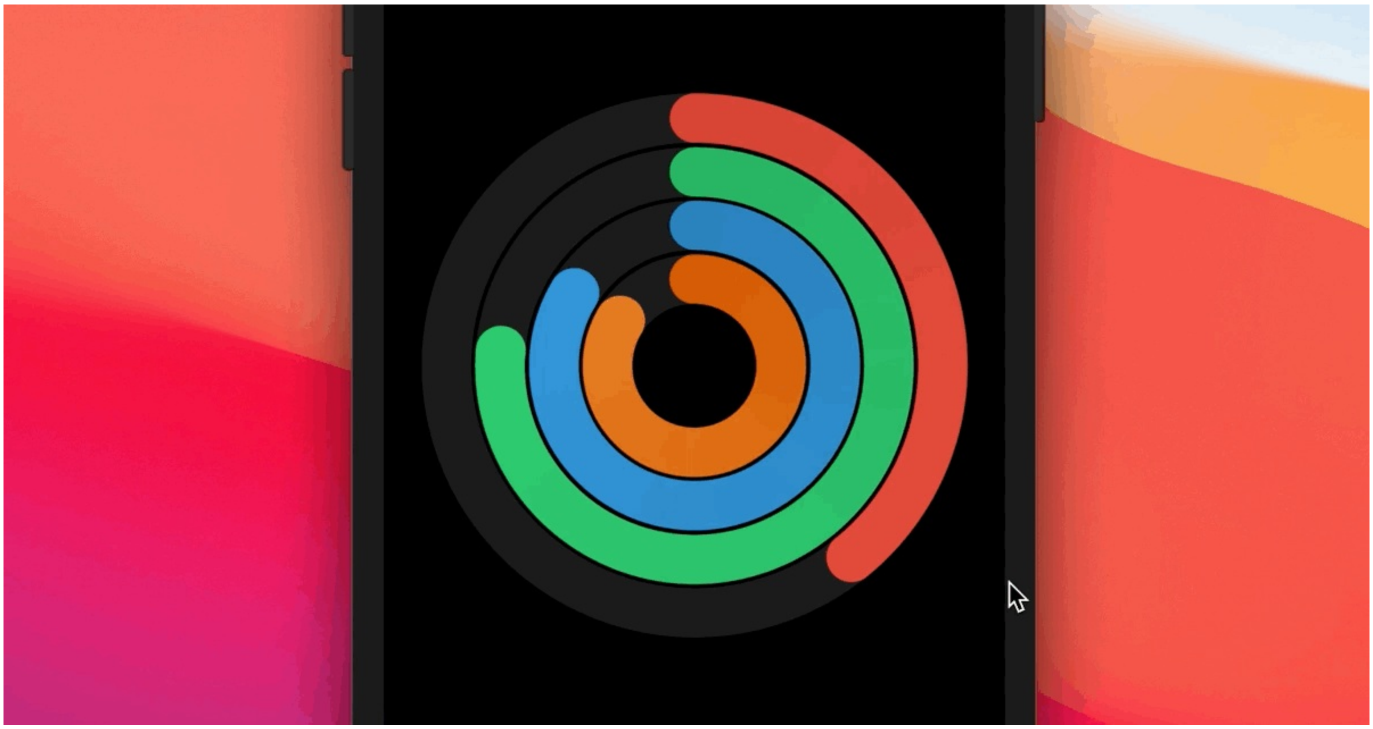


Figure 2. Animated progress ring

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

Chapter 31
Working with AnimatableModifier and LibraryContentProvider

Earlier, you have learned how to animate a shape by using `Animatable` and `AnimatableData`. In this chapter, we will take this further and show you how to animate a view using another protocol called `AnimatableModifier`. On top of that, I will walk you through a new feature of SwiftUI introduced in Xcode 12 that will allow developers to easily share a custom view to the View library and make it easier for reuse. Later, I will show you how to take the progress ring view to the View library for reuse. As a sneak peek, you can take a look at figure 1 or check out this demo video (https://link.appcoda.com/librarycontentprovider) to see how it works.

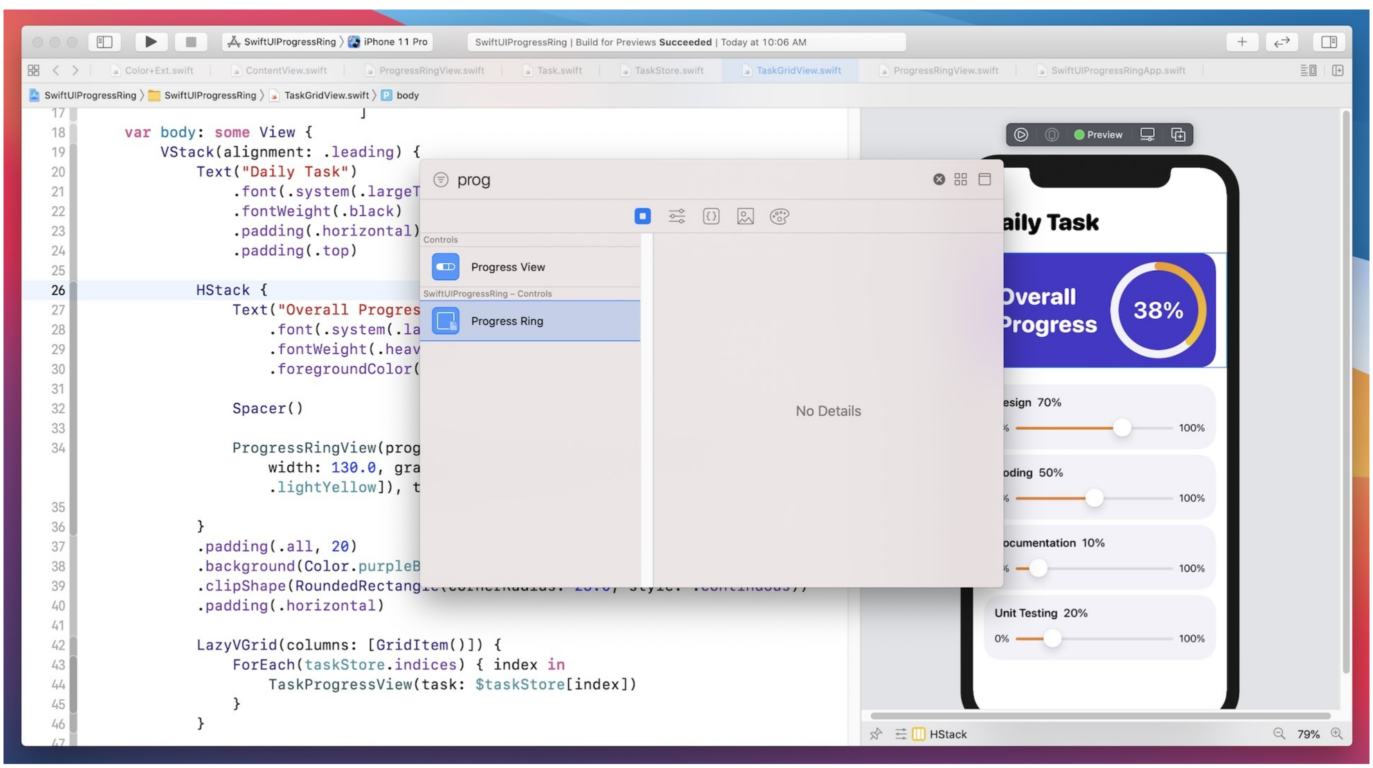


Figure 1. Using a custom view in the View library

Understanding AnimatableModifier

Let's first look into the `AnimatableModifier` protocol. As its name suggests, `AnimatableModifier` is a view modifier and it conforms to the `Animatable` protocol. This makes it very powerful to animate value changes for different types of views.

```
protocol AnimatableModifier : Animatable, ViewModifier
```

So, what are we going to animate? We will build on top of what we've implemented in the previous chapter and add a text label at the center of the progress ring. The label shows the current percentage of the progress. As the progress bar moves, the label will be updated accordingly, following the animation. Figure 2 shows you how the label looks like.

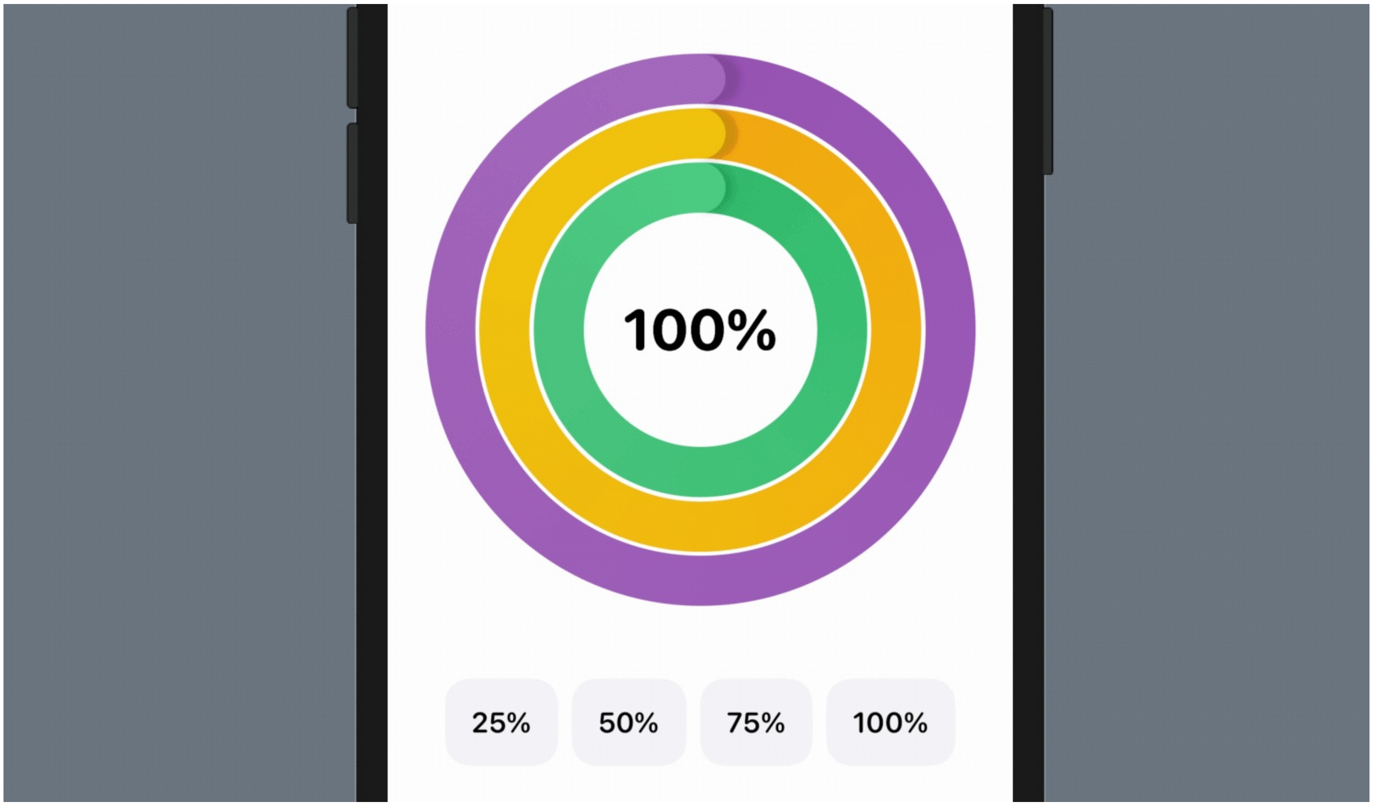


Figure 2. Animating the progress label

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

Chapter 32
Working with TextEditor to Create Multiline Text Fields

The first version of SwiftUI, released along with iOS 13, doesn't come with a native UI component for multiline text field. To support multiline input, you will need to wrap a `UITextView` from the UIKit framework and make it available to your SwiftUI project by adopting the `UIViewRepresentable` protocol. In iOS 14, Apple introduced a new component called `TextEditor` for the SwiftUI framework. This `TextEditor` enables developers to display and edit multiline text in your apps. In this chapter, we will show you how to use `TextEditor` for multiline input.

Using TextEditor

It is very easy to use `TextEditor`. You just need to have a state variable to hold the input text. Then create a `TextEditor` instance in the body of your view like this:

```
struct ContentView: View {
    @State private var inputText = ""

    var body: some View {
        TextEditor(text: $inputText)
    }
}
```

To instantiate the text editor, you pass the binding of `inputText` so that the state variable can store the user input.

You can customize the editor like any SwiftUI view. For example, the below code changes the font type and adjust the line spacing of the text editor:

```
TextEditor(text: $inputText)
    .font(.title)
    .lineSpacing(20)
    .autocapitalization(.words)
    .disableAutocorrection(true)
    .padding()
```

Optionally, you can enable/disable the auto-capitalization and auto-correction features.

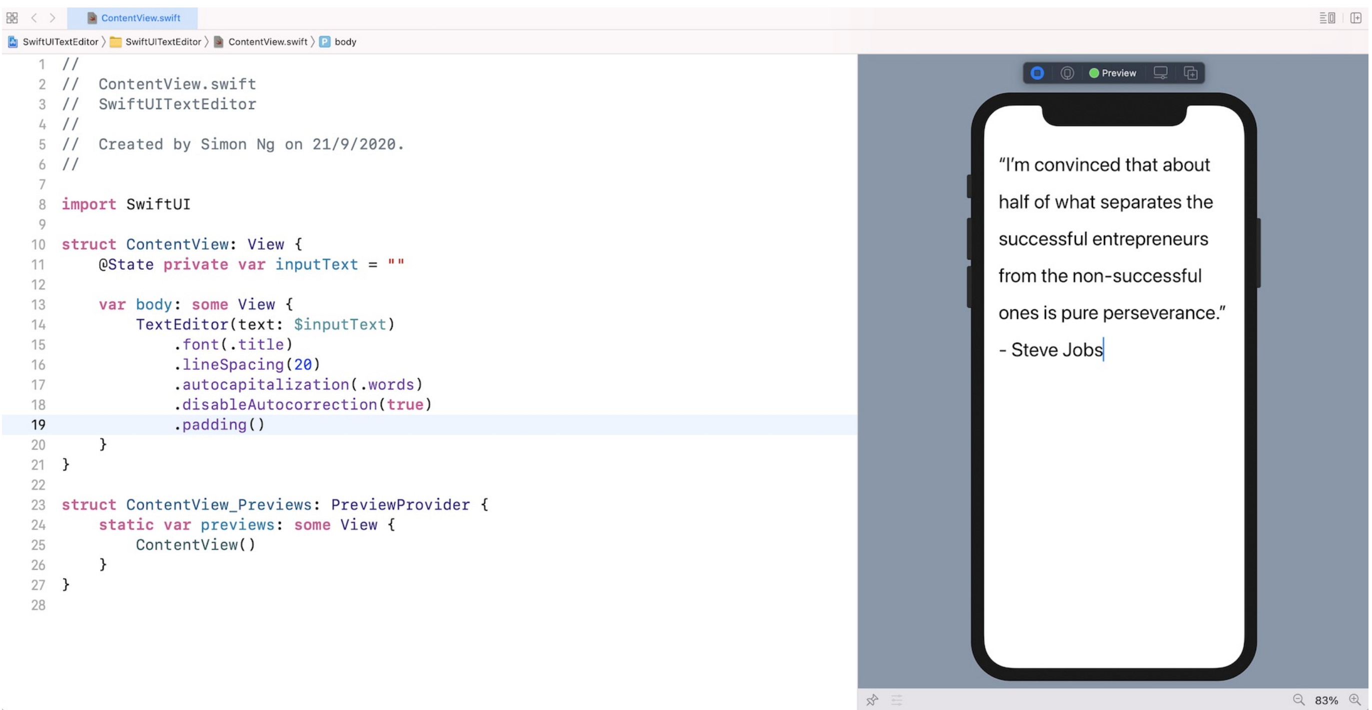


Figure 1. Using TextEditor

Using the onChange() Modifier to Detect Text Input Change

For `UITextView` of the UIKit framework, it works with the `UITextViewDelegate` protocol to handle the editing changes. So, how about `TextEditor`? How can we detect the change of user input and perform further processing?

The new version of SwiftUI introduces an `onChange()` modifier which can be attached to `TextEditor` or any other view. Let's say, if you are building a note application using `TextEditor` and need to display a word count in real time, you can attach the `onChange()` modifier to `TextEditor` like this:

```
struct ContentView: View {
    @State private var inputText = ""
    @State private var wordCount: Int = 0

    var body: some View {
        ZStack(alignment: .topTrailing) {
            TextEditor(text: $inputText)
                .font(.body)
                .padding()
                .padding(.top, 20)
                .onChange(of: inputText) { value in
                    let words = inputText.split { $0 == " " || $0.isNewline }
                    self.wordCount = words.count
                }

            Text("\(wordCount) words")
                .font(.headline)
                .foregroundColor(.secondary)
                .padding(.trailing)
        }
    }
}
```

In the code above, we declare a state property to store the word count. And, we specify in the `onChange()` modifier to monitor the change of `inputText`. So, whenever a user types a character, the code inside the `onChange()` modifier will be invoked. In the closure, we compute the total number of words in `inputText` and update the `wordCount` variable accordingly.

If you run the code in a simulator, you should see a plain text editor and it displays the word count in real time.

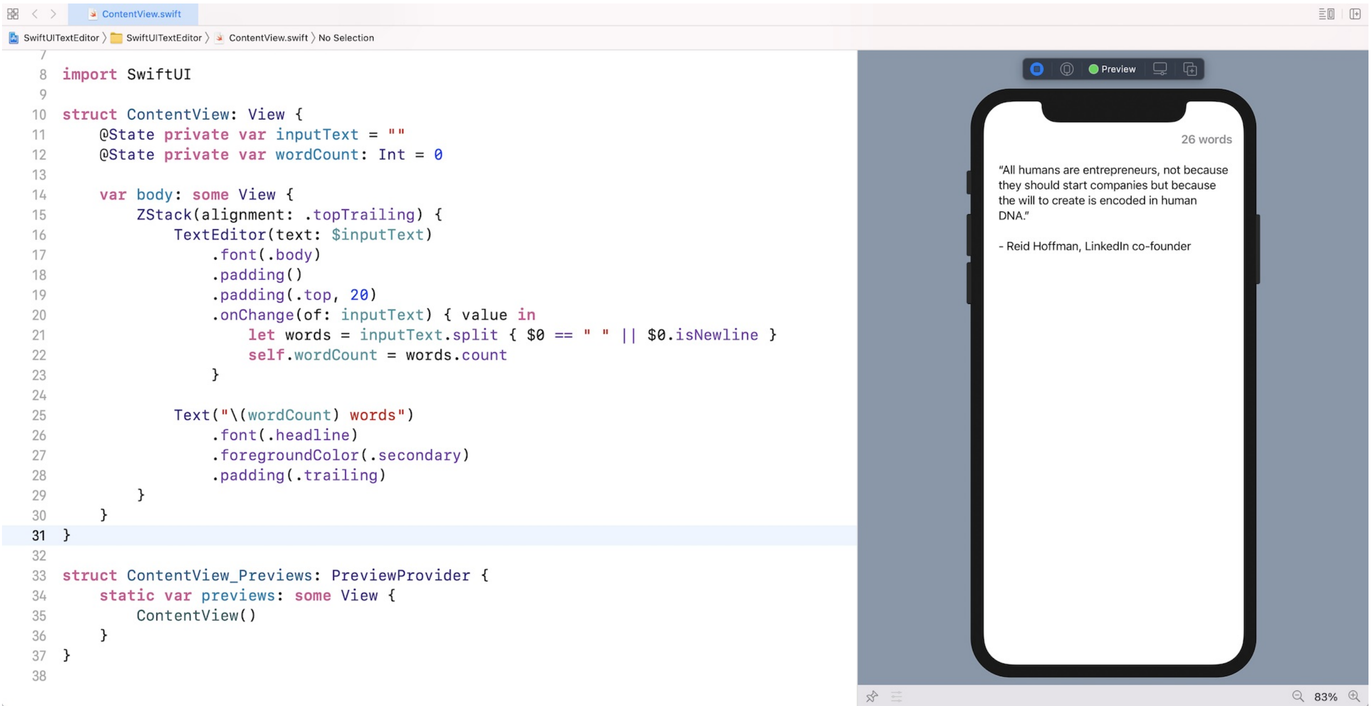


Figure 2. Using onChange() to detect text input and display the word count

Summary

`TextEditor` is one of the most anticipated UI components, which was missing in the initial release of SwiftUI. You can now use this native component to handle multiline input on iOS 14. However, if you still need to support older versions of iOS, you may need to tap into UIKit and bring `UITextView` to your SwiftUI project using the `UIViewRepresentable` protocol.

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

Chapter 33
Using matchedGeometryEffect to Create View Animations

In iOS 14, Apple introduced a lot of new additions to the SwiftUI framework like LazyVGrid and LazyHGrid. But `matchedGeometryEffect` is the new one that really caught my attention because it allows developers to create some amazing view animations with a few lines of code. In the earlier chapters, you have already learned how to create view animations. `matchedGeometryEffect` just takes the implementation of view animations to the next level.

For any mobile apps, it is very common that you need to transit from one view to another. Creating a delightful transition between views will definitely improve the user experience. With the `matchedGeometryEffect` modifier, all you need is describe the appearance of two views. The modifier will then compute the difference between those two views and automatically animates the size/position changes.

Feeling confused? No worries. You will understand what I mean after going through the demo apps.

Revisiting SwiftUI Animation

Before I walk you through the usage of `matchedGeometryEffect`, let's take a look at how we implement animation using SwiftUI. Figure 1 shows the beginning and final states of a view. When you tap the circle view on your left, it should grow bigger and move upward. Conversely, if you tap the one on the right, it returns to the original size and position.

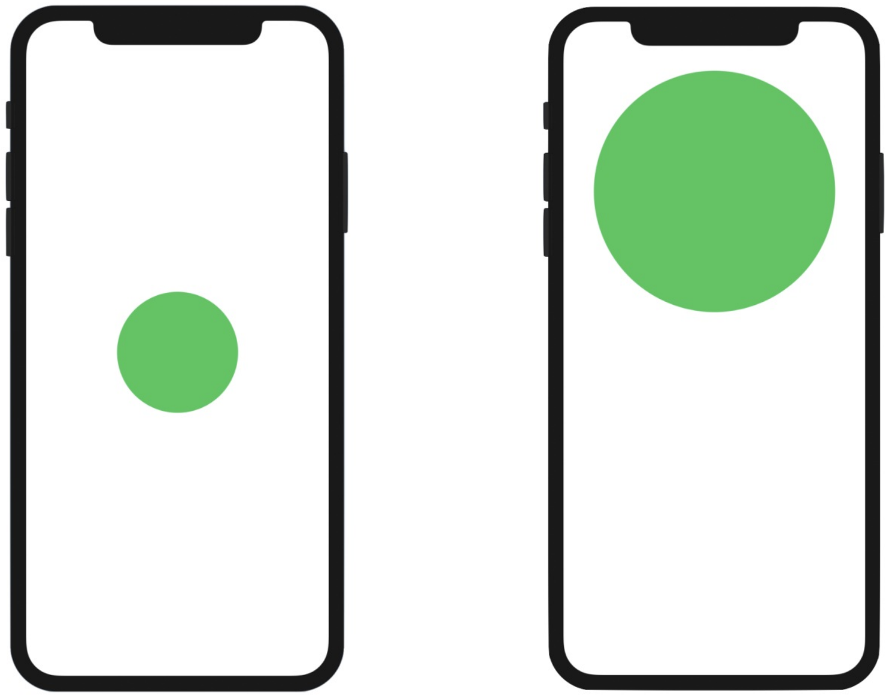


Figure 1. The Circle view at the start state (left), The Circle view at the end state (right)

The implementation of this tappable circle is very straightforward. Assuming you've created a new SwiftUI project, you can update the `ContentView` struct like this:

```
struct ContentView: View {

    @State private var expand = false

    var body: some View {
        Circle()
            .fill(Color.green)
            .frame(width: expand ? 300 : 150, height: expand ? 300 : 150)
            .offset(y: expand ? -200 : 0)
            .animation(.default)
            .onTapGesture {
                self.expand.toggle()
            }
    }
}
```

We have a state variable `expand` to keep track of the current state of the `Circle` view. In both the `.frame` and `.offset` modifiers, we vary the frame size and offset when the state changes. If you run the app in the preview canvas, you should see the animation when you tap the circle.

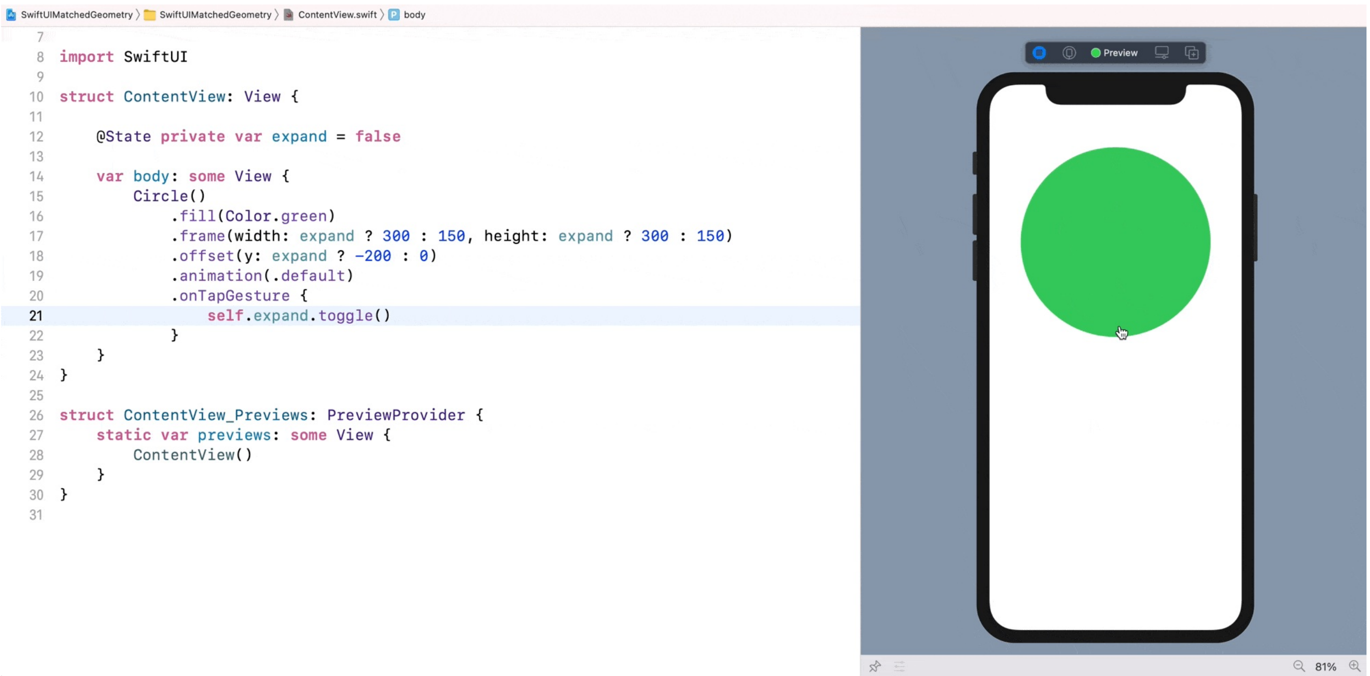


Figure 2. The Circle view animation

Understanding the matchedGeometryEffect Modifier

So, what is `matchedGeometryEffect` ? How does it simplify the implementation of the view animation? Take a look at figure 1 and the code of the circle animation again. We have to figure out the exact value change between the start and the final state. In the example, they are the frame size and the offset.

With the `matchedGeometryEffect` modifier, you no longer need to figure out these differences. All you need to do is describe two views: one represents the start state and the other is for the final state. `matchedGeometryEffect` will automatically interpolate the size and position between the views.

Chapter 34 ScrollViewReader and Grid Animation

Earlier, I introduced you the new `matchedGeometryEffect` modifier and showed you how to create some basic view animations. In this chapter, let's see how to use the modifier and animate item selection of a grid view. On top of that, you will learn another brand new UI component called `ScrollViewReader`.

The Demo App

Before we step into the implementation, let me show you the final deliverable. This should give you an idea about what you are going to build. When developing some real world apps, you may need to display a grid of photo items and let users select some of the items.

One way of presenting the item selection is to have a dock at the bottom of the screen. When an item is selected, it is removed from the grid and insert it into the dock. As you select more items, the dock will hold more items. You can swipe horizontally to navigate through the items in the dock. If you tap an item in the dock, that item will be removed and insert it back to the grid. Figure 1 illustrates how the insertion and removal of an item works.

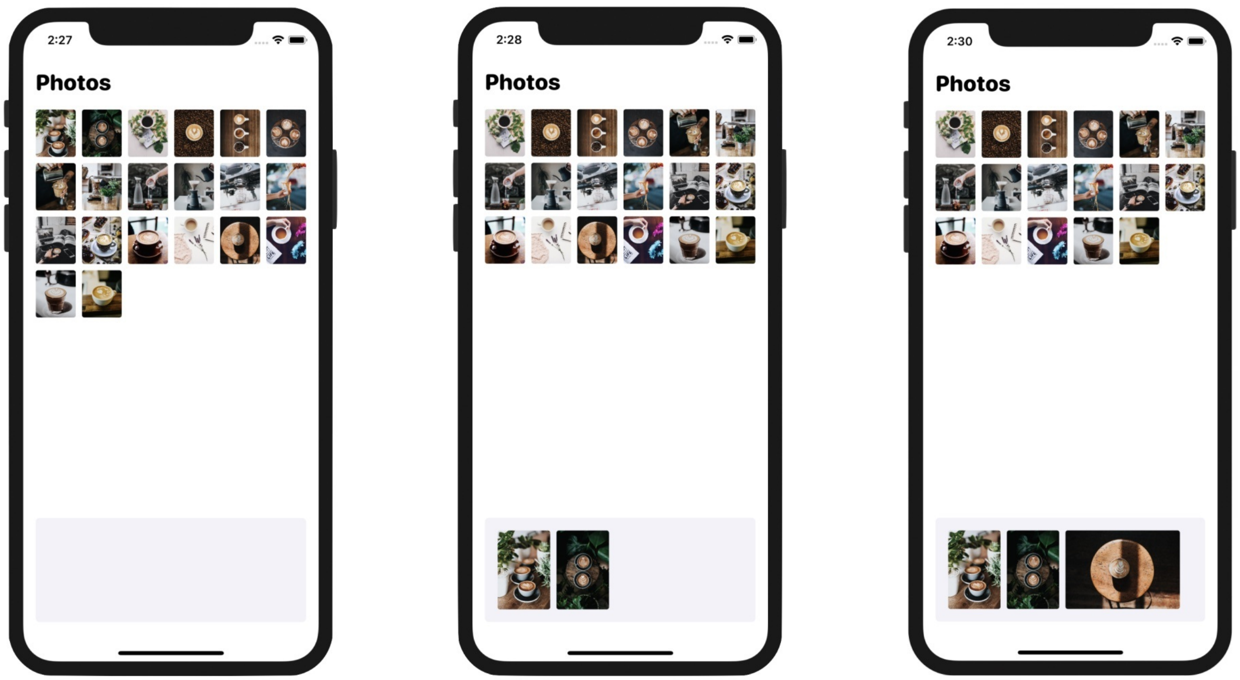


Figure 1. The demo app

We will implement the grid view and the item selection. Needless to say, we will use the `matchedGeometryEffect` modifier to animate the selection.

To access the full content and the source code, please get the official version of the book at https://www.appcoda.com/swiftui.

Chapter 35
Working with Tab View and Tab Bar Customization

The tab bar interface appears in some of the most popular mobile apps such as Facebook, Instagram, and Twitter. A tab bar appears at the bottom of an app screen and let users quickly switch between different functions of an app. In UIKit, you use the `UITabBarController` to create the tab bar interface. The SwiftUI framework provides a UI component called `TabView` for developers to display tabs in the app.

In this chapter, we will show you how to create a tab bar interface using `TabView`, handle the tab selection, and customize the appearance of the tab bar.

Using TabView to Create the Tab Bar Interface

Assuming you've created a SwiftUI project using Xcode 13, let's start with a simple text view like this:

```
struct ContentView: View {
    var body: some View {
        Text("Home Tab")
            .font(.system(size: 30, weight: .bold, design: .rounded))
    }
}
```

To embed this text view in a tab bar, all you need to do is wrap it with the `TabView` component and set the tab item description by attaching the `.tabItem` modifier like this:

```
struct ContentView: View {
    var body: some View {
        TabView {
            Text("Home Tab")
                .font(.system(size: 30, weight: .bold, design: .rounded))
                .tabItem {
                    Image(systemName: "house.fill")
                    Text("Home")
                }
        }
    }
}
```

This creates a tab bar with a single tab item. In the sample code, the tab item has both image and text, but you are free to remove either one of the those.

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

Chapter 36
Using AsyncImage in SwiftUI for Loading Images Asynchronously

In WWDC 2021, Apple announced tons of new features for the SwiftUI framework to make developers' life easier. `AsyncImage` was definitely one of the most anticipated features, introduced in iOS 15. If your app needs to retrieve and display images from remote servers, this new view should save you from writing your own code to handle asynchronous download.

`AsyncImage` is a built-in view for loading and displaying remote images asynchronously. All you need is to tell it what the image URL is. `AsyncImage` then does the heavy lifting to grab the remote image and show it on screen.

In this chapter, I will show you how to work with `AsyncImage` in SwiftUI projects.

The Basic Usage of AsyncImage

The simplest way to use AsyncImage is by specifying the image URL like this:

```
AsyncImage(url: URL(string: imageURL))
```

AsyncImage then connects to the given URL and download the remote image asynchronously. It also automatically renders a placeholder in gray while the image is not yet ready for display. Once the image is completely downloaded, `AsyncImage` displays the image in its intrinsic size.

To access the full content and the complete source code, please get your copy at https://www.appcoda.com/swiftui.

Chapter 37
Implementing Search Bar Using Searchable

Prior to iOS 15, SwiftUI didn't come with a built-in modifier for handling search in `List` views. Developers have to create your own solution. In earlier chapters, I have shown you how to implement a search bar in SwiftUI using `TextField` and display the search result. With the release of iOS 15, the SwiftUI framework brings a new modifier named `searchable` to `List` views.

In this chapter, we will look into this modifier and see easily it is to implement search for a list.

Basic Usage of Searchable

To demonstrate the usage of `Searchable`, please download the demo project (To access the demo, please purchase the book here).

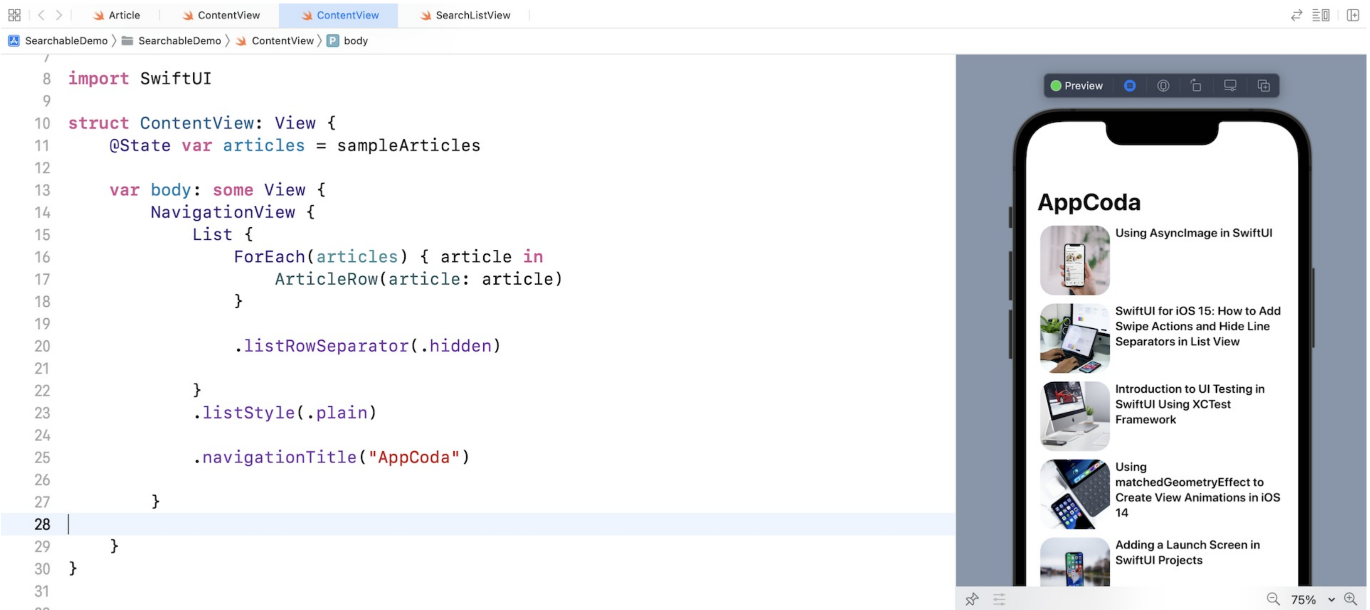


Figure 1. The starter project

The starter project already implements a list view to display a set of article. What I want to enhance is provide a search bar for filtering the articles. To add a search bar to the list view, all you need to do is declare a state variable (e.g. `searchText`) to hold the search text and attach a `searchable` modifier to the `NavigationView` like this:

```
struct ContentView: View {

    @State var articles = sampleArticles
    @State private var searchText = ""

    var body: some View {
        NavigationView {
            .
            .
            .
        }
        .searchable(text: $searchText)
    }
}
```

SwiftUI automatically renders the search bar for you and put it under the navigation bar title. If you can't find the search bar, try to run the app and drag down the list view. The search box should appear.

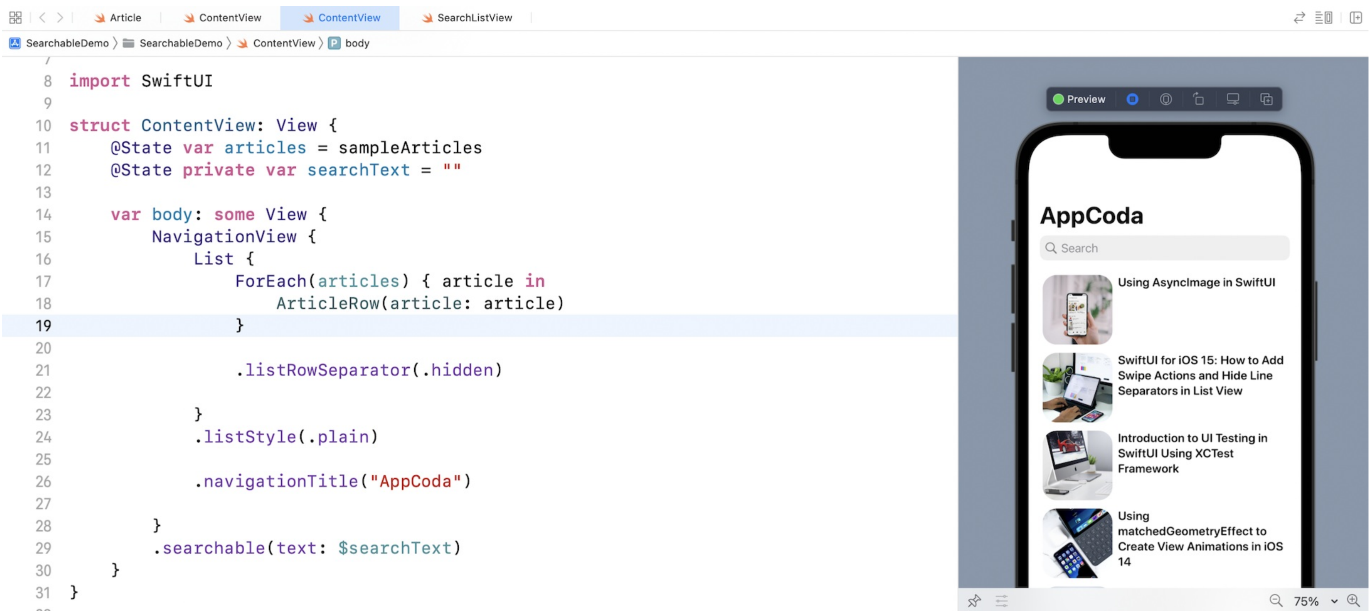


Figure 2. The searchable modifier automatically renders a search field

By default, it displays the word *Search* as a placeholder. In case if you want to change it, you can write the `.searchable` modifier like this and use your own placeholder value:

```
.searchable("Search articles...", text: $searchText)
```

Search Bar Placement

The `.searchable` modifier has a `placement` parameter for you to specify where to place the search bar. By default, it's set to `.automatic`. On iPhone, the search bar is placed under the navigation bar title. When you scroll up the list view, the search bar will be hidden.

If you want to permanently display the search field, you can change the `.searchable` modifier and specify the `placement` parameter like this:

```
.searchable(text: $searchText, placement: .navigationBarDrawer(displayMode: .alway
s))
```

So far, we attach the `.searchable` modifier to the navigation view. You can actually attach it to the `List` view and achieve the same result on iPhone.

Having that said, the placement of the `.searchable` modifier affects the position of the search field when using Splitview on iPadOS. Take a look at the sample code again:

```
NavigationView {
    List {
        ForEach(articles) { article in
            ArticleRow(article: article)
        }

        .listRowSeparator(.hidden)

    }
    .listStyle(.plain)

    .navigationTitle("AppCoda")

}
.searchable(text: $searchText, placement: .navigationBarDrawer(displayMode: .alway
s))
```

As usual, we attach the `.searchable` modifier to the navigation view. If you run the app on iPad, the search bar is displayed on the sidebar of the split view.

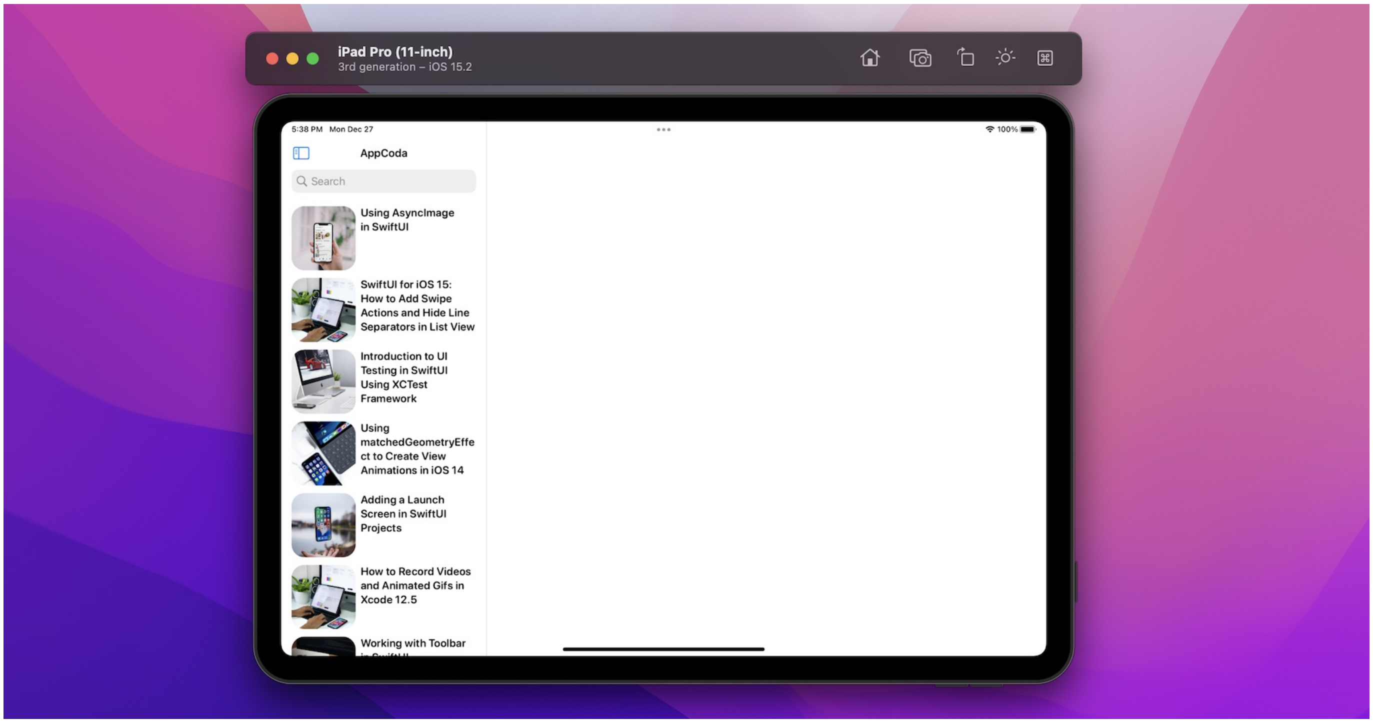


Figure 3. Search bar on iPad

What if you want to place the search field in the detail view? You can try to insert the following code right above the `.navigationTitle` modifier:

```
Text("Article Content")
    .searchable(text: $searchText)
```

iPadOS will then render an additional search bar at the top right corner of the detail view.

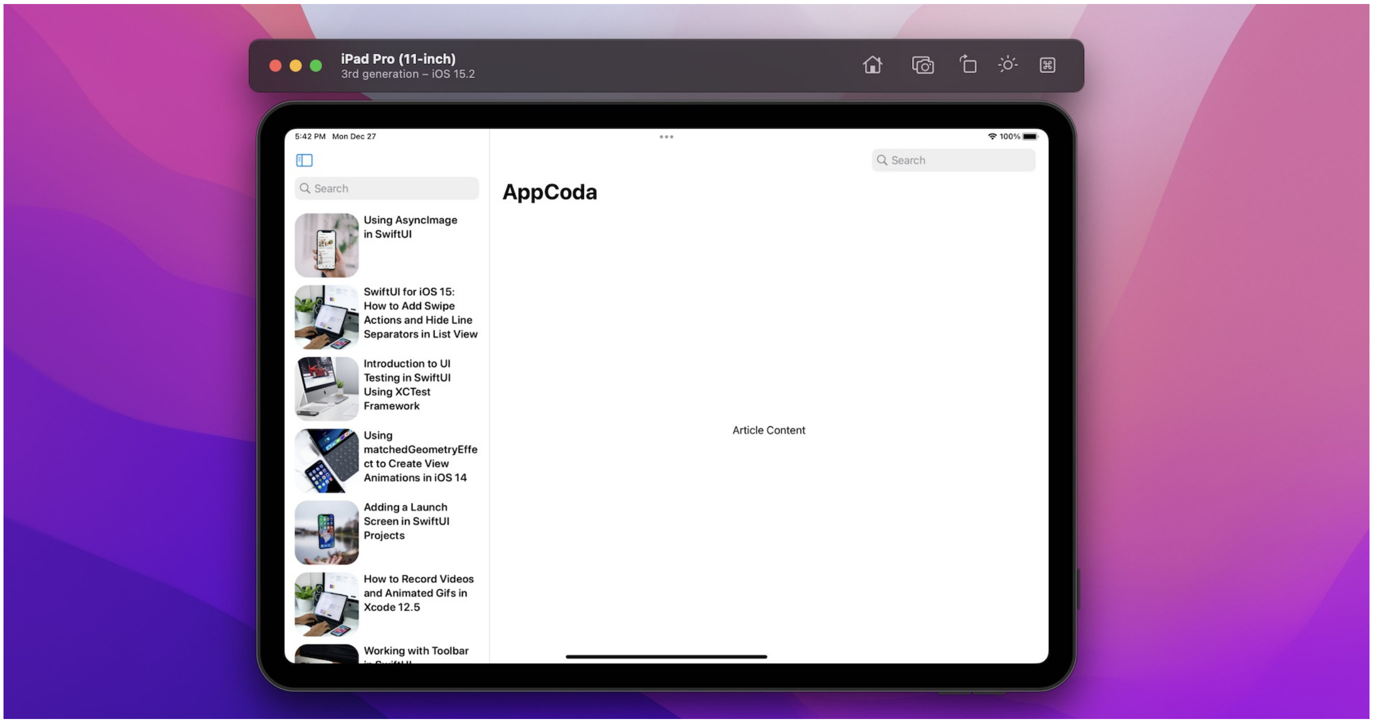


Figure 4. Adding a search bar to the detail view

Again, you can further change the placement of the search bar by adjusting the value of the `placement` parameter. Here is an example:

```
.searchable(text: $searchText, placement: .navigationBarDrawer)
```

By setting the `placement` parameter to `.navigationBarDrawer`, iPadOS places the search field beneath the navigation bar title.

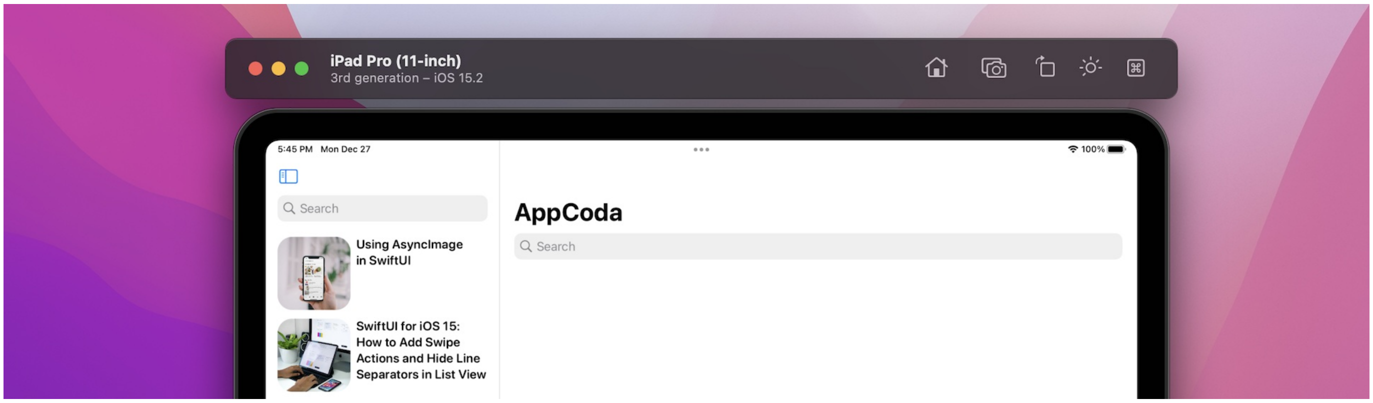


Figure 5. Using .navigationBarDrawer to adjust the position of the search field

Performing Search and Displaying Search Results

There are different ways to filter the list of data. You may create a computed property that performs the data filtering in real-time. Alternatively, you can attach the `.onChange` modifier to keep track of the change of the search field. Update the code of `NavigationView` like below:

```
NavigationView {
    List {
        ForEach(articles) { article in
            ArticleRow(article: article)
        }

        .listRowSeparator(.hidden)

    }
    .listStyle(.plain)

    .navigationTitle("AppCoda")
}
.searchable(text: $searchText)
.onChange(of: searchText) { searchText in

    if !searchText.isEmpty {
        articles = sampleArticles.filter { $0.title.contains(searchText) }
    } else {
        articles = sampleArticles
    }
}
```

The `.onChange` modifier is called whenever the user types in the search field. We then perform the search in real-time by using the `filter` method.

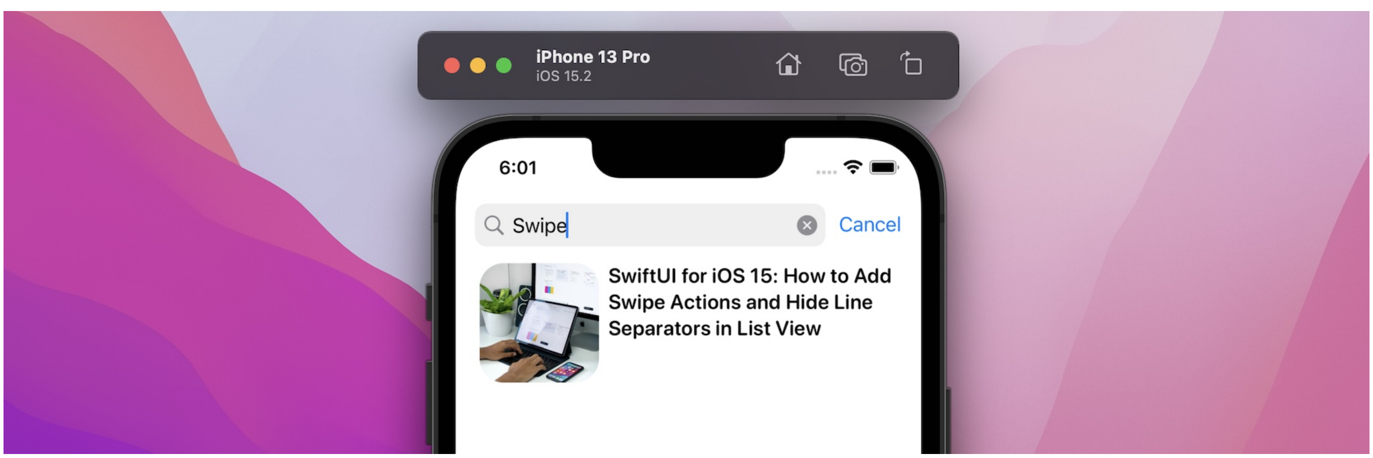


Figure 6. Performing search

Adding Search Suggestions

The `.searchable` modifier lets you add a list of search suggestions for displaying some commonly used search terms or search history. For example, you can create tappable search suggestion like this:

```
.searchable(text: $searchText) {
    Text("SwiftUI").searchCompletion("SwiftUI")
    Text("iOS 15").searchCompletion("iOS 15")
}
```

This displays a search suggestion with two tappable search terms. Users can either type the search keyword or tap the search suggestion to perform the search.

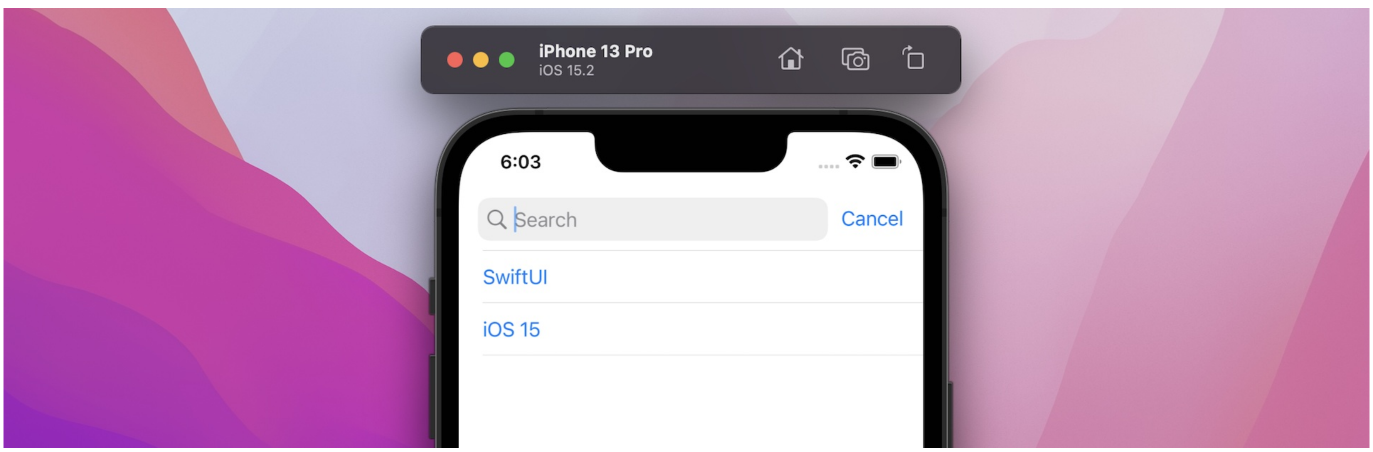


Figure 7. Displaying search suggestions

Summary

iOS 15 brings another welcome feature for the SwiftUI framework. The `.searchable` modifier simplifies the implementation of search bar and saves us time from creating our own solution. The downside is that this feature is only available on iOS 15 (or later). If you are building an app that needs to support the older versions of iOS, you still need to build your own search bar.

To access the demo project, please purchase the book here).