

SwiftUI Projects

Build six real-world, cross-platform mobile applications using Swift, Xcode 12, and SwiftUI



Craig Clayton



SwiftUI Projects

Build six real-world, cross-platform mobile applications using Swift, Xcode 12, and SwiftUI

Craig Clayton

Packt

BIRMINGHAM—MUMBAI

SwiftUI Projects

Copyright © 2020 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Ashwin Nair

Acquisition Editor: Divij Kotian

Senior Editor: Keagan Carneiro

Content Development Editor: Aamir Ahmed

Technical Editor: Deepesh Patel

Copy Editor: Safis Editing

Project Coordinator: Kinjal Bari

Proofreader: Safis Editing

Indexer: Tejal Daruwale Soni

Production Designer: Joshua Misquitta

First published: November 2020

Production reference: 1251120

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-83921-466-0

www.packt.com



Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

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

At www.packt.com, 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 Packt books and eBooks.

Contributors

About the author

Craig Clayton is a self-taught, senior iOS engineer at Fan Reach, specializing in building mobile experiences for NBA, NHL, CHL, and NFL teams. He also volunteered as organizer of the Suncoast iOS meetup group in the Tampa/St. Petersburg area for 3 years, preparing presentations and hands-on talks for this group and other groups in the community. He is launching a new site called Design to SwiftUI online, which specializes in teaching developers and designers how to build iOS apps, from design to SwiftUI video courses.

I would like to thank Marc Aupont. He was the technical reviewer of my book and helped me sort out my ideas. It's been a long year, but I appreciate everything you've done. Next, I would like to thank Michaela MJ, Shane Miller, and Thomas Braun for helping review chapters last minute. Finally, I would like to thank Claudia Maciel and Nicolas Philippe who reviewed many chapters and provided me with detailed notes. I really wouldn't have caught many of my errors without your help, so thank you. Thank you, Mustafa A Yusuf, for all your CloudKit help. You are a lifesaver.

About the reviewer

Marc Aupont is a first-generation American born of Haitian immigrant parents. His passion for technology led him to move from Orlando, FL, to NYC 2 years ago. He currently works at Lickability as an iOS engineer, and his hobbies include working on side projects involving electronics and hardware, hosting and organizing tech meetups, as well as weekend road trips to random destinations with his wife and two boys.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight 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.

Table of Contents

Preface

1

SwiftUI Basics

Technical requirements	2	Rounded Rectangle	20
Views and controls	3	View layout and presentation	21
Text	3	VStack	21
TextField	5	VStack with a spacer	22
SecureField	8	HStack	24
Image	9	HStack with spacer	25
Modifying an image	10	ZStack	27
Buttons	13	Group	32
Shapes	16	ForEach	33
Circle	16	List	35
Rectangle	17	ScrollView	36
Ellipse	18	Summary	38
Capsule	19		

2

SwiftUI Watch Tour

Technical requirements	40	Creating a chart Page-View	
Getting started	40	navigation	44
Building out our navigation	41	Creating a SwiftUI watch list	45
Creating a static list	41	Using Swift previews	47
		Charts	48

Bar charts	48	Creating our wedge	59
Activity Ring	57	Summary	60

3

NBA Draft - Watch App

Technical requirements	62	Creating the details prospect header	81
Building our watch UI	62	Creating detailed prospect stats	84
Walking through the design specs	63	Creating detailed prospect info	87
Accessibility and fonts	65	Connecting our views	90
System fonts	65	Refactoring views	90
Custom fonts	66	Refactoring ContentView	90
Designing the menu	68	Refactoring the detail view	95
Designing draft cards	71	Adding watch data	95
Adding our properties	71	Updating the menu with DraftRound	97
Adding our header	72	Updating the draft list with pick data	99
Adding our card contents	73	Updating the Draft List Card	100
Adding the top of the card	73	Adding data to DraftCardView	101
Adding the bottom of the card	74	Challenge - Updating the details view with prospect data	102
Designing the prospect details	78	Summary	102
Creating a details prospect header	79		

4

Car Order Form - Design

Technical requirements	104	The form view	116
The overall design	104	Complete Order design	125
Understanding the structure	105	The Top Order view	127
Car Detail	107	Bottom Order view	130
Creating Basic car info view	107	The Complete Order view	133
Creating our Car Info Detail view	109	Combining our views	135
Creating our Car Info Photos section	111	Cancel Order design challenge	136
Displaying our Car Info view	113	Summary	137
Wrapping up Car Detail	114		

5

Car Order Form – Data

Technical requirements	140	Combine 101 – discussing the basics	141
Understanding State and Binding	140	Three main ingredients	142
Difference between @Binding and @State	141	Networking with Combine	143
		Understanding observable objects	146

6

Financial App – Design

Technical requirements	154	Creating an account	172
Understanding our App design	154	Color button menu	174
Understanding the home view logic	155	Credit card type menu	177
Home header	155	Form view	179
Creating our card view	157	Bringing it all together – CreateAccountView	183
Home submenu view (challenge 1)	162	Create account list (challenge 3)	188
Creating an account summary	162	Summary	189
Creating our home view	170		

7

Financial App – Core Data

Technical requirements	192	Understanding Core Data relationships	201
What is Core Data?	192	Core Data Codegen	202
What is a managed object model?	193	Core Data manager	205
What is an object context?	193	SwiftUI and Core Data optionals	212
What is a persistent store coordinator?	193	Mock account preview service	213
Creating a data model	193	Implementing our View model	215
Creating Core Data entities	194	Core Data challenge	218
Adding model properties	196	Updating ContentView with an environment object	219
		Summary	227

8

Shoe Point of Sale System – Design

Displaying the app container	230	Product details	239
Creating a products header view	231	Creating custom buttons for our	
Creating our main products container	232	product details view	240
Creating the shopping cart header	232	SizeCartItemView	243
Creating a cart total display	233	Creating the shopping cart	245
Displaying our cart view	234	Designing the cart item view	246
Viewing our custom split view	234	Cart content view	248
Creating the products	236	Summary	252
ProductsContentView	237		

9

Shoe Point of Sale System – CloudKit

Understanding the basics of CloudKit	254	Creating brands	278
Turning on CloudKit manually	255	CloudKit helper	278
Creating our first CloudKit record	260	Creating our view model	283
How to index a new record type in CloudKit Dashboard	266	Creating our dummy data	285
Creating CloudKit models	273	Displaying CloudKit models	286
CloudKit extensions	274	Displaying data in the product details view	289
Creating our product model	274	The shopping cart	290
		Updating our product views	295
		Summary	297

10

Sports News App – Design

Creating a sidebar	301	Schedule prototyping	322
Prototyping our app with boxes	305	Prototyping the game details	324
HeaderView	305	Designing dashboard module views	326
Quick challenge	307	Video player with a grid	326
Dashboard	307	The featured news module	328
Roster prototyping	319		

The featured player module	332	RosterView	348
The standings module view	336	Schedule	352
Playoff module design challenge	342	Game detail challenge	356
Roster	342	Summary	356
RosterHeaderView	343		

11

Sports News App – Data

Mockoon	358	Connecting feeds to	
API design	360	FeaturedArticleModuleView	376
Creating the API class	360	Updating FeaturedPlayerModuleView	376
		Updating PlayerCardView	377
Dashboard data	363	Updating RosterView	378
Latest video	363	Standings	379
Model challenge time	365	StandingItem	379
Updating more dashboard data	366		
Creating our View model	368	Final challenge	381
Implementing our View model	372	Summary	381

Other Books You May Enjoy

Index

Preface

SwiftUI Projects is a book that I have been working on since October 2019 and I am excited that you are finally able to read it. I was fortunate enough to be at WWDC when SwiftUI was announced. It was what everyone was talking about the entire week I was there. It is continuing to grow in popularity, so I love the fact that I'm able to bring you a book that will be one of the first of its kind. SwiftUI is so much fun to use and I swear, as soon as I finish with it, I will be finding myself another reason to start a new project.

Who this book is for

SwiftUI Projects is intended for anyone who is already comfortable with Swift. We do not cover Swift topics in detail, so you need to be familiar with these already. All of the SwiftUI topics are taught as if this is the first time you've learned them and will gradually get more difficult.

What this book covers

Chapter 1, SwiftUI Basics, covers the absolute basics of SwiftUI.

Chapter 2, SwiftUI Watch Tour, looks at the basics of SwiftUI, but inside a watch. We will compile some basic charts and graphs to get comfortable.

Chapter 3, NBA Draft – Watch App, explains how to build an NBA Draft app for a watch. We will learn how to take a design and bring it into Xcode, and then we will cover how to make it work with data.

Chapter 4, Car Order Form – Design, covers how to build a custom form design for Tesla for the iPhone.

Chapter 5, Car Order Form – Data, explains how to get the data from our form and send it out using the basics of Combine.

Chapter 6, Financial App – Design, covers how to design a financial app for the iPhone.

Chapter 7, Financial App – Core Data, explains how to integrate Core Data as well as use SwiftUI State for interactions.

Chapter 8, Shoe Point of Sale System – Design, explains how to design and build out a shoe POS system for the iPad.

Chapter 9, Shoe Point of Sale System – CloudKit, integrates CloudKit with our shoe POS system.

Chapter 10, Sports News App – Design, explains how to build out a sports news app design for the iPad and how to work with multiple layouts.

Chapter 11, Sports News App – Data, explains how to get data from an API and integrate it into our sports news app.

To get the most out of this book

Before reading this book, you should already be comfortable with Swift and have a decent understanding. We will not go into any great detail regarding Swift as we will assume that you are already familiar with it. Any new topics will be discussed:

Software/hardware covered in the book	OS requirements
Swift 5.3	macOS X
Xcode 12	macOS X

If you do copy and paste code, please understand that this will cause your code to not be formatted and will make it difficult to read. It may also add extra text and/or strange characters that result in errors within your code. As the author of this book, I can advise you that copying and pasting code with which you are unfamiliar is hard and will cause a lot of problems.

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/SwiftUI-Projects>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://static.packt-cdn.com/downloads/9781839214660_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

Code in text: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Open the CarInfoBasicView file and update previews with the following previewLayout file."

A block of code is set as follows:

```
func save() {  
    let context = persistentContainer.viewContext  
  
    if context.hasChanges {  
        do {  
            try context.save()  
        } catch let error {  
            print(error.localizedDescription)  
        }  
    }  
}
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "We are moving to the **Complete Order** view. This view appears when you tap on the **Complete Order** button."

Tips or important notes

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

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 copyright@packt.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 authors.packtpub.com.

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 Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

1

SwiftUI Basics

When Apple announced SwiftUI at WWDC in June, I was fortunate enough to be there. As soon as Apple presented SwiftUI, I was like a kid in a candy store because one of my biggest passions in iOS is working on the visual side. I love being able to take a design and try to come as close as I can to matching every detail of a designer's composition. The biggest downfall of using storyboards, which I am still a fan of, is that you cannot quickly prototype. I have ways of doing it, but it takes time, and sometimes you want to go in and try something and not spend a lot of time setting things up like Collection views or Table views with data. SwiftUI helps me focus on creating a beautiful design without needing any data, and when I am ready, I can plug in data. I find this to be the best process because I can add the data layer after completing the design.

I enjoyed designing all six apps that we will cover in this book. I tried to cover a wide range of topics with these six apps. We will build two watch apps, two iPhone apps, and two iPad apps. Most of these features are available on any device; I decided to mix up the apps so that there was a little variety. One thing about this book that might be slightly different is that I have set up the book so we cover design first. After that, we'll focus on the data side. If you do not care about design, you can easily skip this part. Do what you feel is best for you.

In this chapter, we will be working with the following:

- Views and controls
- Shapes
- View layout and presentation

Views and controls are a crucial part of SwiftUI. Some of them you'll be familiar with if you have done iOS development before. Shapes in SwiftUI are super easy to create and use, and once you are comfortable with them, we'll add animations to them to bring them to life. We'll look at Rounded Rectangles, Rectangles, Circles, and even creating Activity Rings later in the book. Finally, view layout and presentation is a big part of SwiftUI, and we'll cover a ton of ways to make beautiful designs and recreate them in SwiftUI.

In this chapter's sample files, you will find a project called `SwiftUIBasics`. You can follow along with all of the code examples in that file. Since we will be doing a ton of writing, I designed this chapter for you to follow along with me.

In the `SwiftUIBasics` project, you will see three folders: `Views and Controls`, `Shapes and View Layout`, and `Presentation`. Each folder has all of the completed code, so you can just follow along. Feel free at this time to play around with modifiers. If you are new to programming, just type a `.` at the end of the line, and you will see a list of other modifiers you can add on.

If you want to type out each example, please feel free to follow along by creating a new project and just create a new SwiftUI file for each step.

You will notice `ContentView` in this app, but you can ignore it as we will not be using this file in this chapter. `ContentView` is the default view that you see when you create a new project. We are not going to cover all views and controls in this chapter, but we will cover most of them at some point in this book.

Technical requirements

The code for this chapter can be found here: <https://github.com/PacktPublishing/SwiftUI-Projects/tree/master/Chapter01>

Views and controls

Views and controls make up the things you add to your layouts and presentations. In the book, we focus on just learning what you can do and some of the things you cannot do with SwiftUI, instead of comparing them to UIKit.

Text

`Text` is a view that displays one or more lines of read-only text:

```
struct ViewsAndControlsTextI: View {  
    var body: some View {  
        Text('This is Text')  
    }  
}
```

Try tapping the **Resume** button on the right:

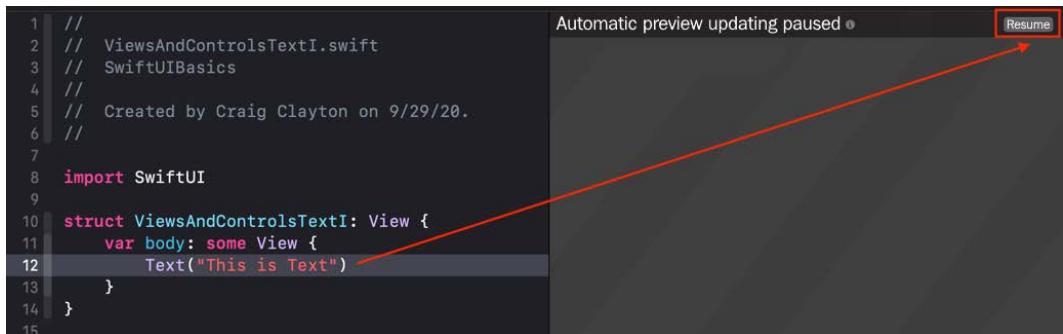


Figure 1.1

After doing so, you should see the following:



Figure 1.2

Text views display using the default font, font size, and the default color of black. In SwiftUI, we can style our views by adding modifiers. Open `ViewsAndControlsTextII` to see how we can add modifiers:

```
struct ViewsAndControlsTextII: View {  
    var body: some View {  
        Text('This is Text')  
            .fontWeight(.bold)  
            .font(.system(size: 24))  
            .foregroundColor(.red)  
    }  
}
```

If you tap the **Resume** button on the left, you will see the following:



Figure 1.3

After adding modifiers to the `Text` view, the text is now red and bold with a font size of 24. Modifiers are used to modify and style a view. `Text` views are used for read-only text; if you want to add a text field that can accept the user's input, you will use `TextField`. Let's take a look at this next.

TextField

In this example, we look at `TextField`. `TextField` is a control that displays an editable text view. Open `ViewsAndControlsTextFieldI`. You will notice we are using `@State` in this example. Since `@State` is a bigger topic, please ignore it for now; we will cover this topic in detail later in the book:

```
struct ViewsAndControlsTextFieldI: View {  
    @State private var username = ''
```

```
var body: some View {  
    TextField('Username', text: $username)  
}  
}
```

If you tap the **Resume** button on the left, you will see the following:



Figure 1.4

The `TextField` we created in this example uses the default look. Let's move to the next example to see how we can modify the `TextField` with a border. Open `ViewsAndControlsTextFieldII` and again ignore `@State`:

```
struct ViewsAndControlsTextFieldIII: View {  
    @State private var username = ''  
  
    var body: some View {  
        TextField('Username', text: $username)  
            .textFieldStyle(RoundedBorderTextFieldStyle())  
    }  
}
```

If you tap the **Resume** button on the left, you will see the following:



Figure 1.5

By using the `.textFieldStyle` modifier, we can give the `TextField` a rounded border. `TextField` views are useful for getting user information such as their username for logging into your app. For a more secure field, such as entering a password, you would use `SecureField`. Let's take a moment to examine `SecureField`.

SecureField

`SecureField` and `TextField` are the same things except that the first one is better for handling sensitive data" if they're both good but the first one is better. The styling is the same for both as well. Open `ViewsAndControlsSecure` and ignore `@State`:

```
struct ViewsAndControlsSecureField: View {  
    @State private var password = ''  
  
    var body: some View {  
        SecureField('Password', text: $password)  
            .textFieldStyle(RoundedBorderTextFieldStyle())  
    }  
}
```

If you tap the **Resume** button on the left, you will see the following:

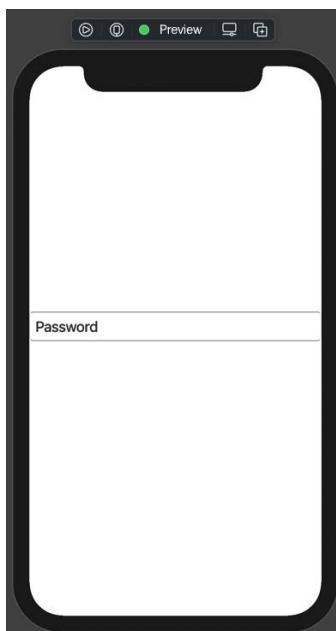


Figure 1.6

Visually, `TextField` and `SecureField` look the same, but as soon as you start to type inside `SecureField`, you immediately see a difference. Now, let's take some time to look at how we can display images in SwiftUI.

Image

We move to our next SwiftUI view, `Image`. Let's look at how we can display local images in SwiftUI:

```
struct ViewsAndControlsImage: View {  
    var body: some View {  
        Image('lebron-james-full')  
    }  
}
```

If you tap the **Resume** button on the left, you will see the following:

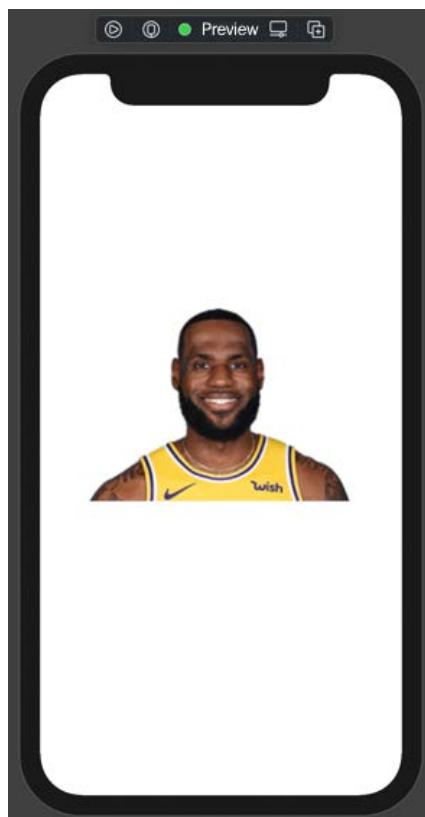


Figure 1.7a

We just looked at how you can display local images in either the Assets Catalog or the project folder. If you need to download the image from a URL, this process takes a bit more code. We do not cover downloading photos from URLs as that is out of scope for this book, mostly because I do not have a place to store them. There are plenty of online resources that cover this topic in detail.

Let's look at how we can modify the size of images.

Modifying an image

In SwiftUI, when you need to resize an image, you have to use the `.resizable` modifier. Let's look at how this works in SwiftUI. Open `ViewsAndControlsResizableImage`:

```
struct ViewsAndControlsResizableImage: View {  
    var body: some View {  
        Image('lebron-james-full')  
            .resizable()  
            .frame(width: 124, height: 92)  
    }  
}
```

Tap the **Resume** button; you'll see the following:



Figure 1.7b

In this example, we use the `.resizable` and `.frame` modifiers to adjust the size of the image. If you are working with images that come from a URL, you will have a different setup. You still have to use the `.resizable` and the `.frame` modifier even if the image comes from a URL. Let's turn our attention to SF Symbols.

In iOS 13, Apple introduced SF Symbols, and with the release of iOS 14, we got even more. These symbols work with accessibility requirements and grow in size when the user changes their system font size. In the following code, we are displaying a rain cloud:

```
struct ViewsAndControlsSFSymbol: View {  
    var body: some View {  
        Image(systemName: 'cloud.heavyrain.fill')  
    }  
}
```

Tap the **Resume** button; you'll see the following:

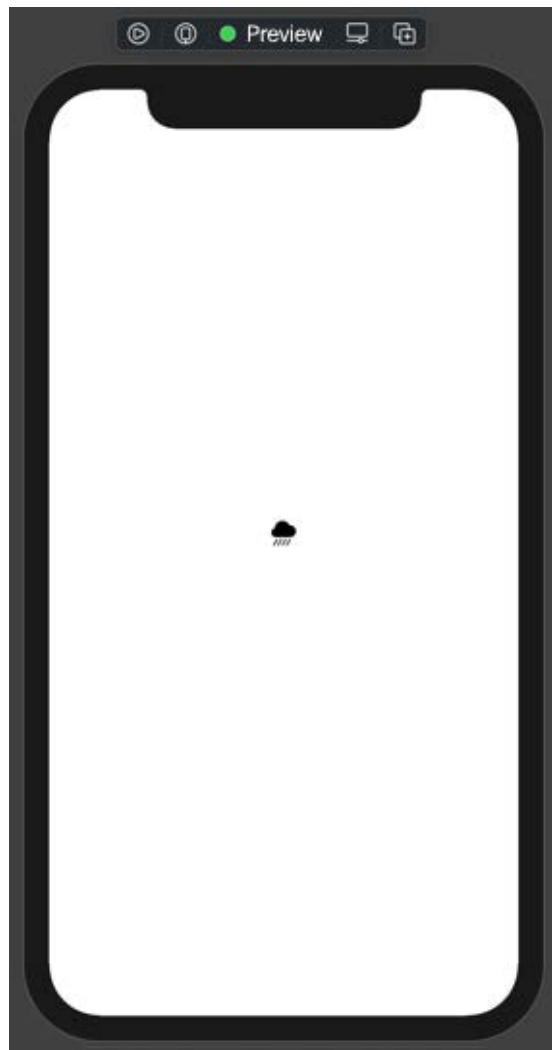


Figure 1.8

If you would like to see the entire library of **SFSymbols v1 and v2**, you can download the app from <https://developer.apple.com/sf-symbols/>. A screenshot of the app looks as follows:

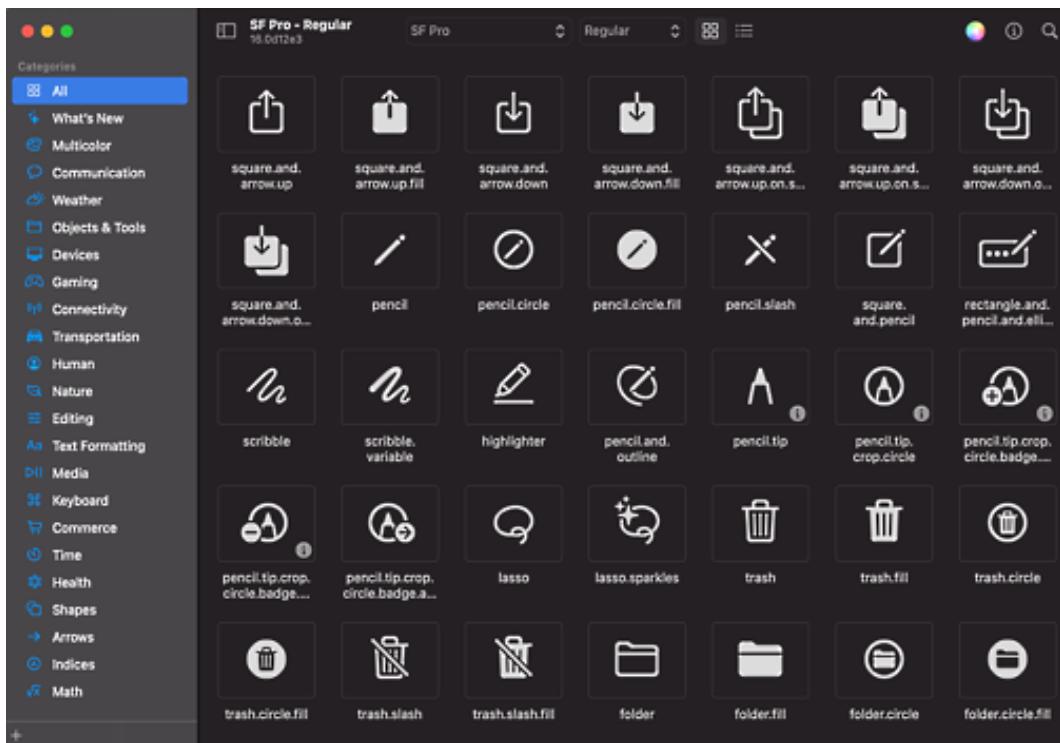


Figure 1.9

With SF Symbols, you have access to over 2,400 configurable symbols that you can use in your Apple apps. We use SF Symbols throughout the book, and you should become familiar with them. Moving to the next example, let's look at how to create buttons.

Buttons

A basic SwiftUI button provides an action and a label. We can use closure blocks or execute a method inside of this action. Let's look at a default SwiftUI button by opening up `ViewsAndControlsButtonI`:

```
struct ViewsAndControlsButtonI: View {
    var body: some View {
        Button(action: { print('Button tapped') }) {
            Text('Button Label')
        }
    }
}
```

If you tap the **Resume** button on the left, you'll see the following:



Figure 1.10

SwiftUI default buttons use a default label that has no style attached. You can add any type of view inside of the label. You can also use modifiers to give a button a certain look and feel.

Let's move to the next example. Open `ViewsAndControlsButtonII` and let's see how we can style a button:

```
struct ViewsAndControlsButtonII: View {  
    var body: some View {  
        Button(action: { print('Button tapped') }) {  
            Text("Tap me")  
        }  
    }  
}
```

```
    Text('Button Label')
}
.padding(10)
.background(Color.red)
.foregroundColor(.white)
}
}
```

If you tap the **Resume** button on the left, you'll see the following:



Figure 1.11

Our button now has a red background with white text and a padding of 10 pixels around the text. You can turn a default button into any design you may need. We'll learn how to create custom buttons and custom button styles that we can share with other buttons in this book. For now, let's move to the next set of examples, shapes.

Shapes

In SwiftUI, we have five preset shapes that you can work with, and they are super easy to create. The Circle, Rectangle, Ellipse, and Capsule are all created the same way. Let's look at each one and stop when you get to Rounded Rectangle.

Circle

Open `ShapesCircle` and let's take a look at how we can create a circle:

```
struct ShapesCircle: View {  
    var body: some View {  
        Circle()  
            .fill(Color.red)  
            .frame(width: 50, height: 50)  
    }  
}
```

If you tap the **Resume** button on the left, you'll see the following:

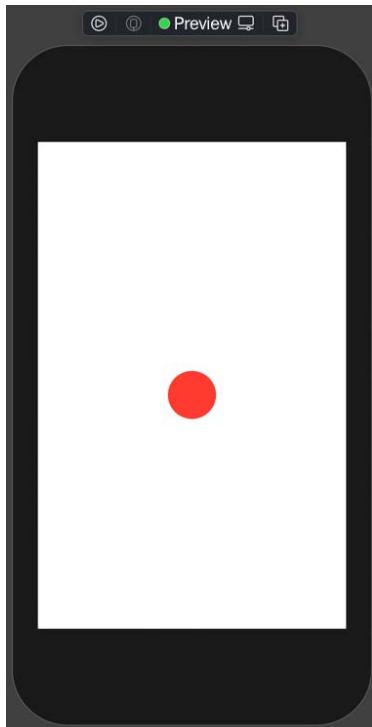


Figure 1.12

Creating shapes is easy in SwiftUI; in the preceding example, our circle is filled with red and is 50x50 in size. We will use custom shapes to create our UI. We can now move on to Rectangle.

Rectangle

We are now going to take a look at the Rectangle. Open `ShapesRectangle` and in our next example, let's take a look at creating a basic rectangle:

```
struct ShapesRectangle: View {  
    var body: some View {  
        Rectangle()  
            .fill(Color.red)  
            .frame(width: 50, height: 50)  
    }  
}
```

Tap on the **Resume** button, and you'll see the following:

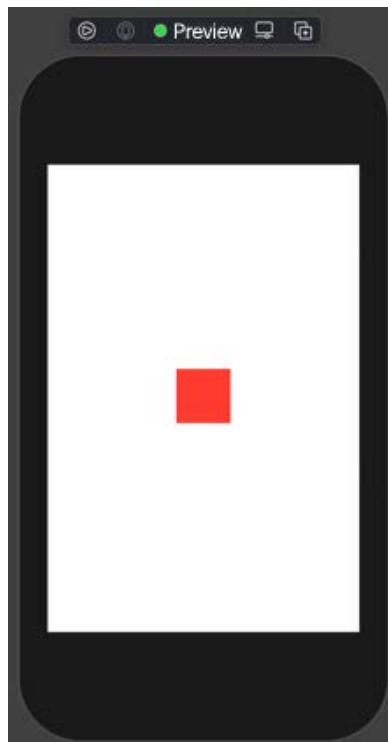


Figure 1.13

In this last example, our Rectangle is filled with red and is 50×50 in size. Let's move on to our next example.

Ellipse

We'll now take a look at an ellipse. Open `ShapesEllipse`, and you will see we applied the same red fill with a size of 100×50 :

```
struct ShapesEllipse: View {  
    var body: some View {  
        Ellipse()  
            .fill(Color.red)  
            .frame(width: 100, height: 50)  
    }  
}
```

Tap the **Resume** button, and you'll see the following:



Figure 1.14

We created an Ellipse, and as you can see, the code is not changing – the shape making, for the most part, has the same pattern. Let's move to the Capsule next.

Capsule

We are onto the next to last shape, the Capsule. Capsules are handy for creating bar charts, which we do later in this book. Let's take a minute and look at a basic Capsule. Open up `ShapesCapsule`:

```
struct ShapesCapsule: View {  
    var body: some View {  
        Capsule()  
            .fill(Color.red)  
            .frame(width: 200, height: 50)  
    }  
}
```

Tap the **Resume** button, and you'll see the following:



Figure 1.15

We now have a capsule sitting in the center of the screen. We can now move on to the final shape, and that's the Rounded Rectangle.

Rounded Rectangle

The Rounded Rectangle is the only shape that has a parameter, `.cornerRadius`, when you create one. Let's open `ShapesRoundedRectangle` and check out our final shape:

```
struct ShapesRoundedRectangle: View {  
    var body: some View {  
        RoundedRectangle(cornerRadius: 25)  
            .fill(Color.red)  
            .frame(width: 200, height: 25)  
    }  
}
```

Tap the **Resume** button, and you'll see the following:



Figure 1.16

We are finished looking at shapes, but remember that all shapes by default have a fill color of black. Next, we'll focus on the view layout and presentation.

View layout and presentation

Let's get into this next section, but keep in mind that you can embed these views inside of other views. We won't cover that in this chapter as it is something we do a ton throughout this book. Let's move to `VStack`.

`VStack`

When you use a `VStack`, it arranges all of its children in a vertical line. Let's take a look at this in operation by opening `ViewLayoutVStack`:

```
struct ViewLayoutVStack: View {  
    var body: some View {  
        VStack {  
            Rectangle()  
                .fill(Color.red)  
                .frame(width: 50, height: 50)  
  
            Rectangle()  
                .fill(Color.red)  
                .frame(width: 50, height: 50)  
        }  
    }  
}
```

Tap the **Resume** button, and you'll see the following:

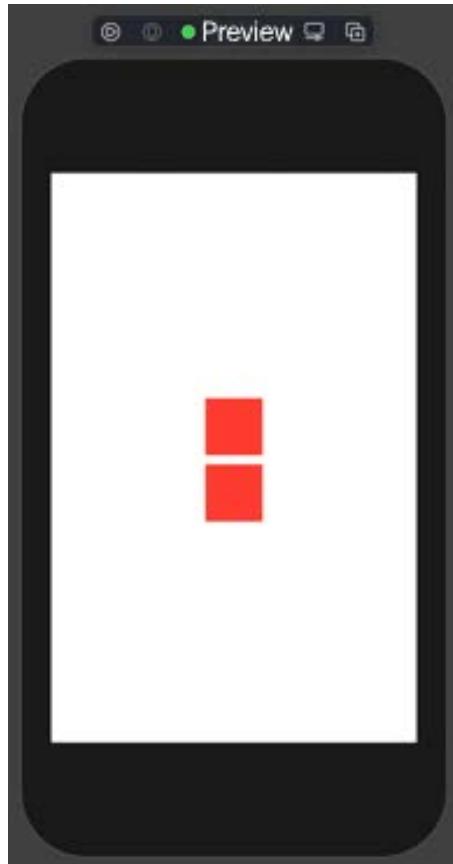


Figure 1.17

In this example, we are displaying two Rectangles on a vertical line in the center of the stack. To make our Rectangle fill all available vertical space, we would need to use a spacer. Let's see what happens when we add a spacer.

VStack with a spacer

Using a spacer allows us to manipulate how our objects respond inside of the `VStack`. In this specific example, we are adding a spacer in between each rectangle. Open `ViewLayoutVStackSpacer`:

```
struct ViewLayoutVStackSpacer: View {  
    var body: some View {  
        VStack {
```

```
    Rectangle()
        .fill(Color.red)
        .frame(width: 50, height: 50)
    Spacer()
    Rectangle()
        .fill(Color.red)
        .frame(width: 50, height: 50)
    }
}
}
```

Tap the **Resume** button, and you'll see the following:

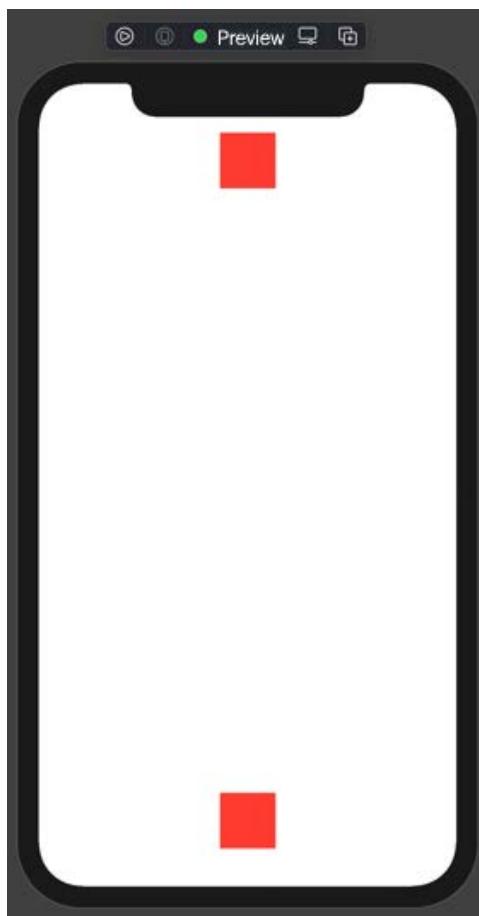


Figure 1.18

Putting our spacer in between each rectangle pushes our rectangles to the top and bottom, respectively. If you move the spacer below the two Rectangles, this moves both Rectangles to the top of the screen. You would get the opposite if you moved the spacer above both Rectangles.

Take a minute to move the spacer around inside of the `VStack`, to see how it behaves. When finished, let's move to the `HStack`.

HStack

Our next container is called an `HStack`, and you probably guessed it – the `HStack` displays its children on a horizontal line:

```
struct ViewLayoutHStack: View {  
    var body: some View {  
        HStack {  
            Rectangle()  
                .fill(Color.red)  
                .frame(width: 50, height: 50)  
  
            Rectangle()  
                .fill(Color.red)  
                .frame(width: 50, height: 50)  
        }  
    }  
}
```

Tap the **Resume** button, and you'll see the following:

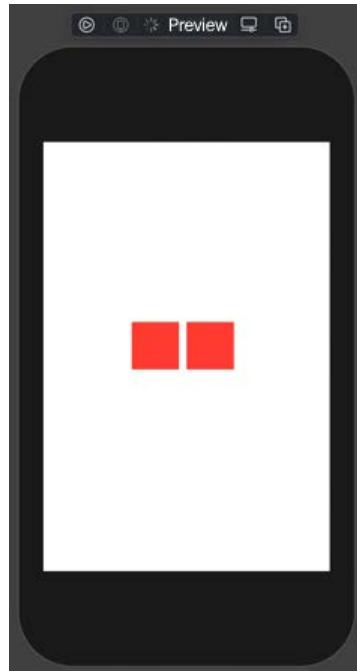


Figure 1.19

We pretty much have the same code as we did for the `VStack` example, except using an `HStack` as the main container. `HStack`, by default, is aligned in the center of the screen. Now, just as we did with `VStack` and a spacer, we can do the same by manipulating the `Rectangles` here to get the layout we need.

HStack with spacer

To illustrate a spacer in an `HStack`, we use two spacers, instead of one, along with three rectangles. Open `ViewLayoutHStackSpacer`, and let's see it in action:

```
struct ViewLayoutHStackSpacer: View {
    var body: some View {
        HStack {
            Rectangle()
                .fill(Color.red)
                .frame(width: 50, height: 50)
            Spacer()
            Rectangle()
```

```
        .fill(Color.red)
        .frame(width: 50, height: 50)

    Spacer()

    Rectangle()
        .fill(Color.red)
        .frame(width: 50, height: 50)
    }

}
```

Tap the **Resume** button, and you'll see the following:

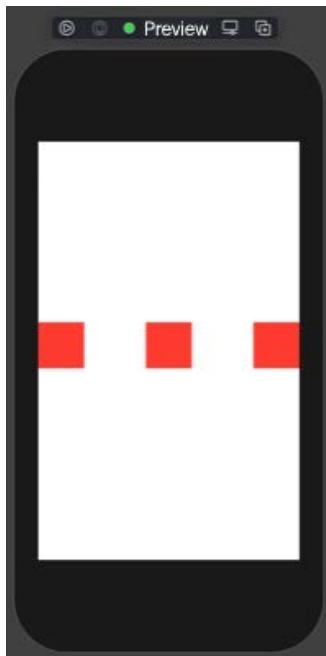


Figure 1.20

By adding two spacers, we get rectangles on the right, on the left, and directly in the middle. Take some time and move the spacers around, so you get a feel for how it works as we use it throughout the book. Finally, in our last stack example, we'll take a look at the `ZStack`.

ZStack

`ZStack` is a bit different than `VStack` and `HStack` because instead of its children aligning along a particular axis when they are added to the container, with `ZStack` they are stacked on top of each other. Open `ViewLayoutZStack` to see this in action:

```
struct ViewLayoutZStack: View {  
    var body: some View {  
        ZStack {  
            Color.black  
            Text('Craig Clayton')  
                .foregroundColor(.white)  
        }  
    }  
}
```

Tap the **Resume** button, and you'll see the following:



Figure 1.21

In this `ZStack` example, we add the color black to our `ZStack` along with a `Text` view. You might be asking how we can add a color to our view. Well, colors are nothing more than views, which means they can be added just like other views. Right now, you might be thinking that `ZStack` is nothing special, but it is, as you will see soon. Let's look at another example by opening `ViewLayoutZStack`:

```
struct ViewLayoutZStack: View {  
    var body: some View {  
        ZStack {  
            Color.black  
                .edgesIgnoringSafeArea(.all)  
  
            Text('Craig Clayton')  
                .foregroundColor(.white)  
  
            Text('Craig Clayton')  
                .foregroundColor(.white)  
                .offset(x: 0, y: 100)  
        }  
    }  
}
```

Tap the **Resume** button, and you'll see the following:



Figure 1.22

In the preceding example, we set the background color to black. Then we extend the color to the edges and ignore the safe areas on all edges of the device. Next, we have two `Text` views, but one of them has an offset. If you remove the offset, the `Text` views get stacked on top of each other, just as you would expect when adding views to a `ZStack`. Using the offset, we can move our views around on a `ZStack`. Let's look at one more `ZStack` example; this time, we will use an alignment with a `ZStack`.

Maybe this example will show you the reason for my excitement over using `ZStack`. Open `ViewLayoutZStackAlignment`:

```
struct ViewLayoutZStackAlignment: View {  
    var body: some View {  
        ZStack(alignment: Alignment(horizontal: .trailing,  
                                     vertical: .top)) {  
            Color.black  
                .edgesIgnoringSafeArea(.all)  
  
            Text('Another Example')  
                .foregroundColor(.white)  
                .offset(y: 25)  
  
            Text('Craig Clayton')  
                .foregroundColor(.white)  
                .offset(y: 50)  
  
            Rectangle()  
                .fill(Color.red)  
                .frame(width: 100, height: 25)  
        }  
    }  
}
```

Tap the **Resume** button, and you'll see the following:

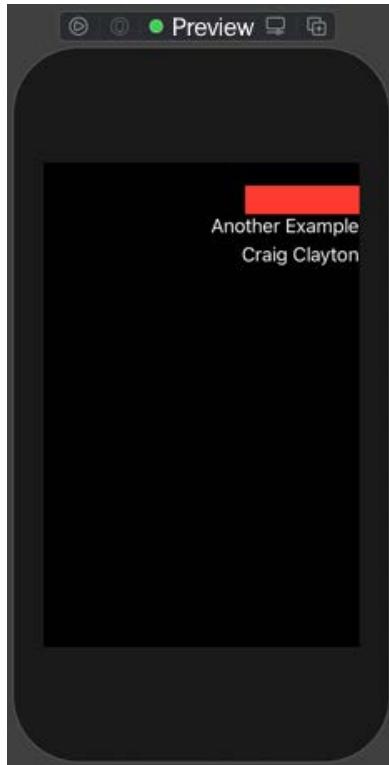


Figure 1.23

HStacks have vertical alignments, and VStacks have horizontal alignments. ZStacks can utilize both horizontal and vertical alignments. A ZStack with an alignment helps get the views in the general direction required, and we can then fine-tune the placement using x and y offsets. As we work through more and more designs in this book, this will make more sense to you.

ZStack is one of my favorite features in SwiftUI, and I use it a ton in this book; the more you get familiar with it, the more you'll understand why.

Take some time messing with ZStack, and when you are ready, move on to the next example.

Group

A `Group` in SwiftUI is a container that you can use to group view elements without any special alignment. Open `ViewLayoutGroup`, and let's take a look at how `Group` works:

```
struct ViewLayoutGroup: View {  
    var body: some View {  
        VStack {  
            Group {  
                Text('Gabriel Lang')  
                Text('John Brunelle')  
                Text('Matthew Arieta')  
                Text('Ralph Dugue')  
            }  
            .foregroundColor(.red)  
            .font(.largeTitle)  
  
            Group {  
                Text('Alex Burnett')  
                Text('Craig Heneveld')  
                Text('Bill Munsell')  
                Text('Wayne Ohmer')  
            }  
            .foregroundColor(.red)  
            .font(.largeTitle)  
        }  
    }  
}
```

Tap the **Resume** button, and you'll see the following:

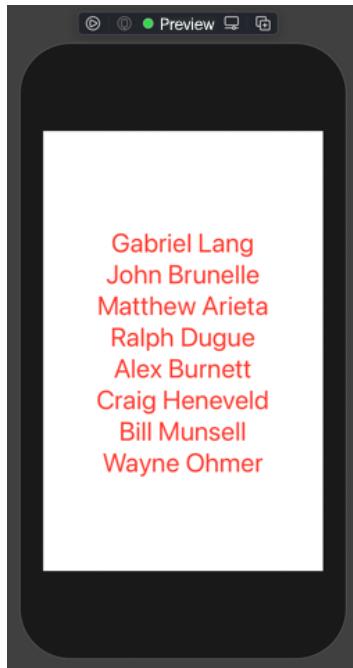


Figure 1.24

In this example, we have text views that are inside of a `VStack`, and instead of adding `foregroundColor` to each text, I added it to the `Group`. Also note that you can use this technique with `VStack`, `HStack`, and `ZStack`. Grouping is also great for applying animations to the entire `Group` or if you want to manipulate the `Group` differently based on the device in use. Let's move to the next example.

ForEach

A `ForEach` struct is a bit different than the `forEach()` you might be accustomed to. SwiftUI's `ForEach` is a view struct, which allows us to add it directly to the body. We can create views using the `ForEach` struct because it takes an array of unique items. Open `ViewLayoutForEach`, and let's take a few moments to see how `ForEach` works:

```
struct ViewLayoutForEach: View {  
    let coworkers = ['Gabriel Lang', 'John Brunelle', 'Matthew  
        Arieta', 'Wayne Ohmer', 'Ralph Dugue', 'Alex  
        Burnett', 'Craig Heneveld', 'Bill Munsell']  
    var body: some View {  
        VStack {  
            ForEach(coworkers, id: \.self) { name in
```

```
        Text(name.uppercased())
    }
}
.foregroundColor(.blue)
.font(.headline)
}
}
```

Tap the **Resume** button, and you'll see the following:

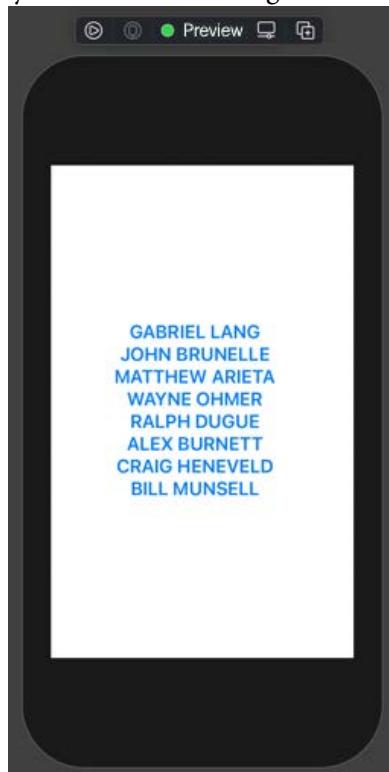


Figure 1.25

In this example, we use a `ForEach` struct to loop through the array of coworkers. Each time it loops through, it grabs the `id` (which needs to be unique; we are using each name as our unique identifier). During the loop, it also sets the name to uppercase, sets the text foreground color to blue, and finally sets the font to headline. We can use `ForEach` as a way to work with an array of data. Let's now take the time to look at `List`.

List

A `List` is a container that displays a row of data in a single column. Open `ViewLayoutList` and let's see a `List` in action:

```
struct ViewLayoutList: View {  
    var body: some View {  
        List {  
            Text('1')  
            Text('2')  
            Text('3')  
        }  
    }  
}
```

Tap the **Resume** button, and you'll see the following:

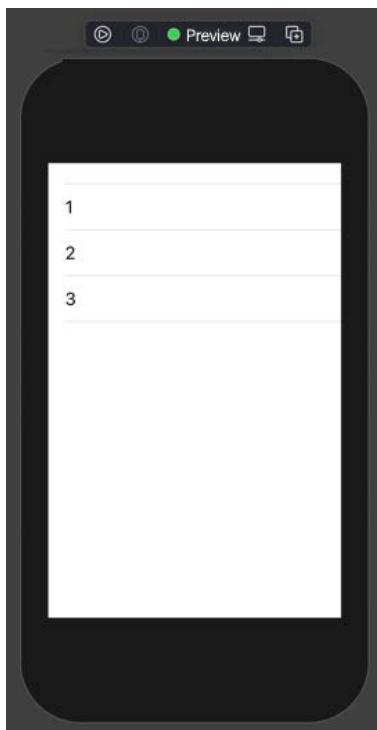


Figure 1.26

Here, we are displaying a list of text views that are embedded into a `List`. Now, let's take a minute and understand the difference between `ForEach` and `List`.

Differences between ForEach and List

When you are working with a `List`, you can display mixed content as well as scroll. `List` also utilizes the reusing cell pattern, which is super-efficient. Now, as far as design goes, it is much harder to customize `List` view defaults. When you use a `ForEach` struct, it works only with a collection.

ScrollView

A `ScrollView` allows you to scroll content either horizontally or vertically. Open `ViewLayoutScrollHorizontal` to move to the next example:

```
struct ViewLayoutScrollHorizontal: View {
    var body: some View {
        ScrollView(.horizontal) {
            HStack(spacing: 15) {
                ForEach(0..<10) { _ in
                    Rectangle()
                        .fill(Color.red)
                        .frame(width: 50, height: 50)
                }
            }
        }
    }
}
```

Tap the **Resume** button, and you'll see the following:

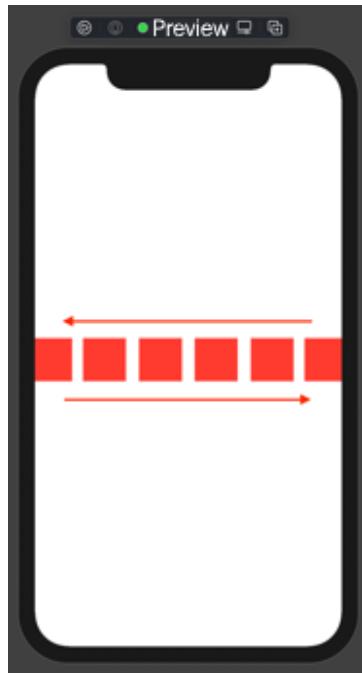


Figure 1.27

For this example, we have a `ScrollView` wrapped around an `HStack`, which means its contents will scroll horizontally. In our `ForEach`, we are creating 10 Rectangles, which are red and 50×50 . Open `ViewLayoutScrollVertical`, and let's see how we can do this vertically:

```
struct ViewLayoutScrollVertical: View {
    var body: some View {
        ScrollView {
            VStack(spacing: 15) {
                ForEach(0..<20) { _ in
                    Rectangle()
                        .frame(width: 50, height: 50)
                }
            }
        }
    }
}
```

Tap the **Resume** button, and you'll see the following:

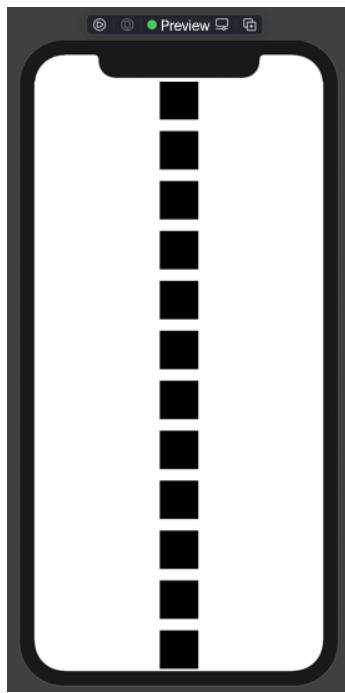


Figure 1.28

In this final example, we changed the `HStack` to a `VStack` from our last example. SwiftUI makes it easy to create a specific UI design. We have looked at views and controls, shapes, and view layouts. We now understand that we can use modifiers to update how they look visually and where they will be laid out on the screen. By the end of this book, you will have learned how to break down designs specifically for SwiftUI apps.

Summary

In this chapter, we looked at the basics of SwiftUI, focusing on views and controls, shapes, and finally, view layout. We learned about how to add styles to our views and controls. This chapter served as a useful introduction; as we progress through the book, we will look at even more SwiftUI views and controllers.

In the next chapter, we will go a step further and build a watch app using SwiftUI. We will not be focused on the design of a specific app, but we will be focused on making sure we get better at understanding the basics. We covered a lot in this chapter, but we will see each of these views repeatedly throughout the book.

2

SwiftUI Watch Tour

We've looked at things SwiftUI can do; now, we will examine how they look on each device. In this chapter and the next, we will work exclusively with watchOS and SwiftUI. Even if you are not doing watchOS development, everything can be applied to the iPhone, iPad, and macOS. There are some subtle differences, but overall everything is the same.

Since there is no specific design for this chapter, we'll look at the default behaviors that we get out of the box. WatchOS has many default looks that are harder to customize than on the other devices. We'll use this chapter to focus on execution, and in the next chapter, we'll work with a specific design. Let's get started working with SwiftUI.

In this chapter, we'll cover the following:

- Creating a SwiftUI PageView in watchOS
- Creating a Bar Chart, Wedge Chart, and Activity Ring
- Creating a list in watchOS

This chapter has a starter file called `SwiftUIWatchTour` inside the `Chapter 2` folder.

Technical requirements

The code for this chapter can be found at <https://github.com/PacktPublishing/SwiftUI-Projects/tree/master/Chapter02>.

Getting started

We are going to build a small app that will just get your feet wet. We will create a List which we will use to navigate us through our small app. Charts will link to charts and Colors will link to another colors list. One will show you Charts, and the other will show you a List from an Array. You will create three charts in this chapter – Bar, Activity Ring, and Wedge. We will build out a PageView, which will allow us to swipe from left to right to see each one. For the List, you will create a list and display it using an array. Here are what the screens will look like when we are finished:



Figure 2.1

Building out our navigation

SwiftUI is now fully supported in watchOS 7, which means we do not have to work with some of the older controllers pre watchOS 6. We will first create our entire navigation, and then we will go back and build out each view. Let's get started by opening ContentView first.

Creating a static list

Our List, in this view, is going to be a static list with just two links. One will present a modal, and the other will do the standard push to detail view. Add the following after the struct declaration:

```
@State private var isPresented = false
```

In the last chapter, we skipped `@State` because it was not in scope, and I will do it again as I cover this in greater detail in *Chapter 5, Car Order Form – Data*. But I will, for now, just say we are creating a Boolean that keeps track of the modal presentation. Next, inside the body variable, replace `Text ('Content View')` with the following:

```
List {  
    Button('Charts') { // Step 1  
        self.isPresented.toggle() // Step 2  
    }  
    .fullScreenCover(isPresented: $isPresented, content:  
        ChartsView.init) // Step 3  
  
    // Add next step here  
}
```

We just added a few lines of code; let's break down each step:

1. As we covered in the last chapter, we have a button with its label set to `Charts`. In watchOS, buttons have a default look to them, so as long as you give it a label and an action, you will get all of the stylings. You might also notice we created our button slightly differently. There are a couple of different ways to create buttons, and this is just another implementation of it.
2. We call the `toggle()` method on the `isPresented` boolean.
3. In this step, we are using the `fullScreenCover` modifier and passing it the `isPresented` variable and the destination view, which in this case is `ChartsView`.

Now let's add our last bit of code by replacing `// Add next step here` with the following:

```
NavigationLink(destination: ColorsView()) { // Step 1
    Text('Colors') // Step 2
}.navigationTitle('Home') // Step 3
```

Let's discuss the code we just added:

1. We are using a `NavigationLink` and this requires a destination view and also a label.
2. The label is a view, and, in this case, we are using a text view.
3. We set the title of the page here by using `.navigationTitle` set to `Home`.

We are done with our static `List`, so build and run the project, and you will be able to hit each item and see the following:



Figure 2.2

You now have the main navigation done but let's look at how we can create a page view navigation in the next section.

Creating a chart Page-View navigation

Next, we are going to create a Page View navigation controller. Just as you would expect on a phone, you will be able to swipe from the right to left and see different content on each page. You will also see some dots at the bottom of the page, which shows which page is being displayed currently.

Open `ChartsView` and let's get started. Inside the `body` variable, replace `Text ('Charts View')` with the following:

```
TabView { // Step 1
    BarChartView() // Step 2
    WedgeChartView()
    RingView()
}.tabViewStyle(PageTabViewStyle(indexDisplayMode: .automatic))
// Step 3
```

We just added all of the code for `ChartsView`, so let's look at what we did:

1. We wrap all of our views inside a `TabView`.
2. Next, we set up each page by just listing them inside the `TabView`.
3. We set the `.tabViewStyle()` modifier to `PageTabViewStyle(indexDisplayMode: .automatic)`, which sets `TabView` to a page control.

If you build and run now, and then select **Charts** from the List, you will see the following:



Figure 2.3

We are done with getting all of our navigation set up. Next, we will create the first List using an array in the next section.

Creating a SwiftUI watch list

We are going to display a SwiftUI List view. Our List is going to display a list of colors. First, we need to create a color model.

Open the `ColorModel` file inside the `Model` folder and add the following:

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

This struct has two properties: `id` and `name`. We have also set our model so that it conforms to `Identifiable`. When using a List in SwiftUI, our List is required to be unique, and there are two ways to handle this. We can either pass data, for example, the name as our unique ID, or we can use `UUID` and use this as our ID. The more you work with SwiftUI, the more ways you will encounter to handle `Identifiable`. If your data was coming from a feed, then you could use `id` if it were unique.

Open `ColorsView.swift` and add the following code inside the `ColorsView` struct, before the body:

```
@State var colors: [ColorModel] = [ ColorModel(name: 'Red'),
                                      ColorModel(name: 'White'),
                                      ColorModel(name: 'Blue'),
                                      ColorModel(name: 'Black'),
                                      ColorModel(name: 'Pink'),
                                      ColorModel(name: 'Yellow')
]
```

Inside the body variable, replace `Text ('Hello World')` with the following code:

```
List {
    ForEach(colors) { color in // Step 1
        NavigationLink(destination: EmptyView()) { // Step 2
            Text(color.name) // Step 3
        }
}
```

```
    }  
}
```

We just added another list; let's break down the rest of the code:

1. We have a `ForEach` loop inside our `List`, and as it loops through the array, it gives us a color.
2. Our `ForEach` loop creates a `NavigationLink` every time it loops through. `NavigationLink` is used to link to another page, but you might be wondering why you don't see `EmptyView()` in the folders. `EmptyView()` is an empty view that we get with SwiftUI and is excellent for prototyping. Using this allows us to deploy a simple placeholder that we can use until we create our actual detail view. For this example, we won't change it, but in later chapters, we will.
3. `Text(color.name)` is our button label.

We are finished setting up our Colors View. Hit *Command + R* and you should now see a list of colors where, when you tap on an item, it goes to a blank screen with a back button:



Figure 2.4

Instead of displaying an empty view, we can use a `Text` view to display our detail view. Update `NavigationLink(destination: EmptyView())` to `NavigationLink(destination: Text(color.name))`. Now, if you build and run, you can tap on the color and will see it in the detail view.

Using Swift previews

We covered this in the last chapter, but before we write any SwiftUI code, we will look at a SwiftUI file. Open `BarChartView` and you'll see our struct view, which we looked at in the previous chapter. If you scroll to the bottom, you will see **static var previews**. Previews are used to preview our design without having to launch the simulator. You should see a blank space to the right of your code, and at the top of this, you will see a button named **Resume**:

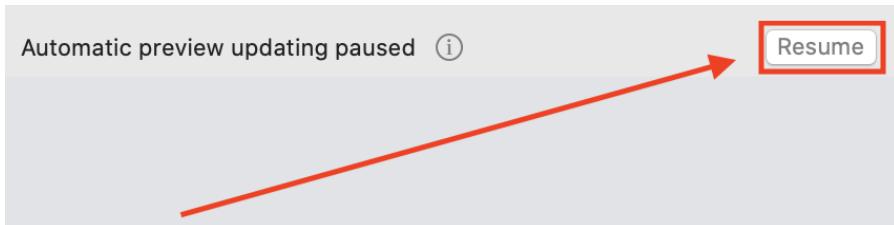


Figure 2.5

Click **Resume**, and you will see the preview appear:

```
//
//  BarChartView.swift
//  SwiftUIWatchTour WatchKit Extension
//
//  Created by Craig Clayton on 12/6/19.
//  Copyright © 2019 Cocoa Academy. All rights
//  reserved.
//

import SwiftUI

struct BarChartView: View {
    var body: some View {
        Text("Bar Chart")
    }
}

struct ShowsView_Previews: PreviewProvider {
    static var previews: some View {
        BarChartView()
    }
}
```



Figure 2.6

Let's move on to building some charts.

Charts

Building charts is pretty fun in SwiftUI because it requires very little code. Here is an example of the three screens that we are looking to create in this section:



Figure 2.7

We will start on the Bar Chart first using the Capsule shape.

Bar charts

Bar charts are a great way to display information to users. In this example, we will create a static Bar Chart that you can use to display data on the watch. We can also take all of these examples and show them on the larger screens of other devices. Whenever we work in SwiftUI, we will use Swift Previews to review our work, making our workflow go much faster than having to wait for the simulator to launch and display. Please note that to use Swift Previews, you must be on macOS Catalina. If you are not, then just run the simulator.

Creating a header

First, we will start by creating our header for our view. This view is just two text views stacked horizontally next to each other. Open `BarChartView`, and we will start by first replacing `Text ('Bar Chart')` with the following code:

```
HStack(spacing: 0) {
    Text('BAR')
        .fontWeight(.heavy)
    Text('CHART')
        .fontWeight(.thin)
```

```
}

.foregroundColor(Color.red)

// Add next step here
```

In this code, we have an `HStack` with two `Text` views, which also has a spacing of 0. We apply a `foregroundColor` of `red` to the `HStack`; this gives a red color to both `Text` views.

You should see the following in the **Preview**:



Figure 2.8

Next, we need to create the container that our bar chart will go in. Replace `// Add the next step here` with the following:

```
VStack(spacing: 0) { // Step 1
    // Header
    HStack(spacing: 0) { // Step 2
        Text('BAR')
            .fontWeight(.heavy)
        Text('CHART')
```

```
        .fontWeight(.thin)
    }
    .foregroundColor(Color.red) // Step 3

    // Add next step here
}
```

We are adding our Header first; let's break down the code:

1. We use a `VStack` as our main container with 0 spacing.
2. Inside the `VStack`, we have an `HStack` that we are using for a header.
3. We set the foreground to red for the entire `HStack`. Both **BAR** and **CHART** will now be red.

You should now see the following in the Swift **Preview**:

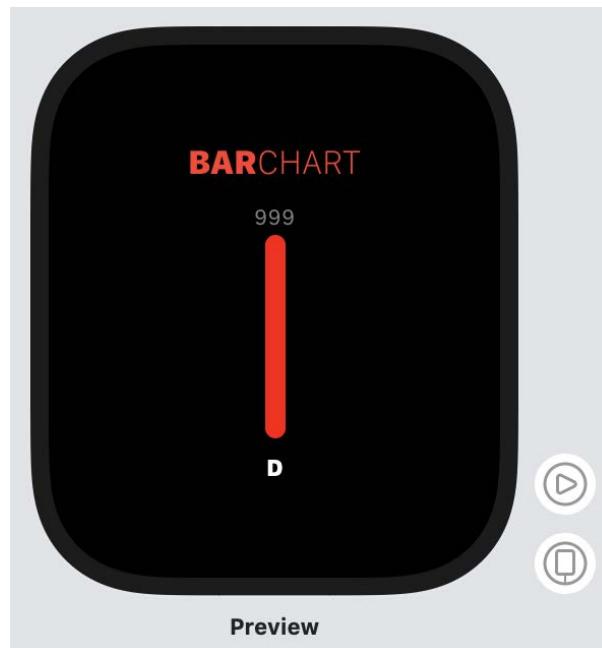


Figure 2.9

Now that we have our header, let's add our capsule. Replace `// Add next step here` with the following code:

```
HStack(alignment: .bottom, spacing: 2) { // Step 1
    VStack { // Step 2
        VStack(spacing: 2) { // Step 3
            Text('99')
                .font(.system(size: 11))
                .foregroundColor(Color(.gray))
            Capsule()
                .frame(width: 10, height: 100)
                .foregroundColor(Color(.red))
        }
    }
    Text('M') // Step 4
        .font(.system(size: 12))
        .fontWeight(.black)
        .padding(.top, 0)
    }
}
.padding(.top, 10) // Step 5
```

We have a lot going on inside this `HStack` so let's go through each step together:

1. We are using an `HStack` for alignment purposes only. The alignment is set to `bottom` with the spacing set to 2. I am putting the alignment to the bottom because I want my graph to start bottom-up. If I weren't trying to get everything aligned to the bottom, I wouldn't use the `HStack`.
2. Next, we have a `VStack` that is our actual main container.
3. Inside the main container, we have another `VStack` that holds the `Capsule` shape and the `Text` view for the value.
4. Here, we use a `Text` view to display the day of the week.
5. We add a 10-pixel padding to the top to create a bit of space between the `Capsule` and the title.

When you are done, you will see the following in the **Preview**:

```
struct BarChartView: View {
    var body: some View {
        VStack(spacing: 0) {
            // Header
            HStack(spacing: 0) {
                Text("BAR")
                    .fontWeight(.heavy)
                Text("CHART")
                    .fontWeight(.thin)
            }
            .foregroundColor(Color.red)

            // Add next step here
            HStack(alignment: .bottom, spacing: 2) {
                VStack {
                    VStack(spacing: 2) {
                        Text("99")
                            .font(.system(size: 11))
                            .foregroundColor(Color(.gray))
                        Capsule()
                            .frame(width: 10, height: 100)
                            .foregroundColor(Color(.red))
                    }
                    Text("M")
                        .font(.system(size: 12))
                        .fontWeight(.black)
                        .padding(.top, 0)
                }
                .padding(.top, 10)
            }
        }
    }
}
```

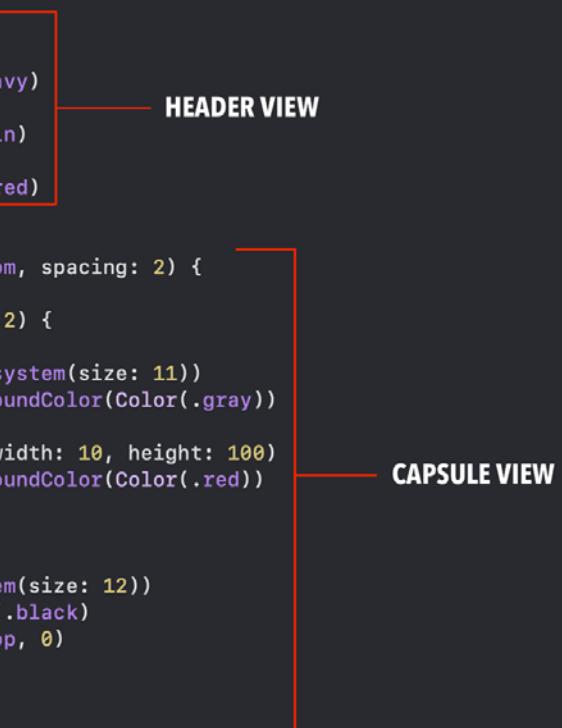


Figure 2.10

We need to make this a bit more dynamic because we want to create more than one instance of our Capsule shape. Let's refactor this code so that we can reuse this.

Cleaning up our code

Creating child views is a great way to clean up our code, and we can clean up two things in this view. First, our Header code will be used for multiple views, so we can move the headers into a child view. Lastly, the Capsule code we just added can also be made into a view to be dynamic. Let's check out our code and see what I mean:

```
//  
// HeaderView.swift  
//  
//  
// Created by Craig Clayton on 11/15/19.  
// Copyright © 2019 Cocoa Academy. All rights reserved.  
//  
  
import SwiftUI  
  
struct HeaderView: View {  
  
    var body: some View {  
        HStack(spacing: 0) {  
            Text("BAR")  
                .fontWeight(.heavy)  
            Text("CHART")  
                .fontWeight(.thin)  
        }  
        .foregroundColor(Color.red)  
    }  
}  
  
struct HeaderView_Previews: PreviewProvider {  
    static var previews: some View {  
        HeaderView()  
    }  
}
```

Figure 2.11

We want to take the `HStack` that holds our header contents and move it to `HeaderView`. We also want to take the other `HStack` and move it to `CapsuleView`.

Creating a reusable header view

First, highlight and cut the first `HStack` that we have inside our `VStack`. Then, open the `HeaderView` class, and paste it into the `body` variable in place of `Text ('Header View')`. Save the file and you should see the following inside `HeaderView`:

```
struct HeaderView: View {  
    var body: some View {  
        HStack(spacing: 0) {  
            Text('BAR')  
                .fontWeight(.heavy)
```

```
        Text('CHART')
            .fontWeight(.thin)
    }
    .foregroundColor(Color.red)
}
}
```

Next, go back to `BarChartView` and type `HeaderView()` on the line where we just removed the code. Then hit **Resume** in **Swift Previews**, and you should still see the bar chart just as you did before. Let's do the same with our `Capsule` code.

Creating a reusable bar view

Find the remaining `HStack` that is inside the `VStack`, then select and cut the code. Open `CapsuleView`, which is inside the `Views` folder, and inside the `body` variable, paste the `Capsule` code in place of `Text ('Capsule View')`. Save the file, go back to `BarChartView`, and under `HeaderView()`, add `CapsuleView()`.

When you are done, your `BarChartView` code will now look like the following:

```
struct BarChartView: View {
    var body: some View {
        VStack(spacing: 0) {
            HeaderView()
            CapsuleView()
        }
    }
}
```

Your code is now refactored, but we need to allow each view to take data. First, let's update our `HeaderView` so that it can take a title and subtitle.

HeaderView with dynamic text

Open the `HeaderView` file again and above the `body` variable but below the `struct` declaration, add the following variables:

```
let title: String
let subtitle: String
```

Then, under `Swift Previews`, update the preview variable by replacing `HeaderView()` with the following: `HeaderView(title: 'BAR', subtitle: 'CHART')`.

That now gets rid of our errors, so let's update the two `Text` views to use both the `title` and `subtitle` variables. Update `BAR` inside our `body` variable with `title.uppercased()`.

Then, update `CHART` inside our `body` variable with `subtitle.uppercased()`. Lastly, back in `BarChartView`, update `HeaderView()` with `HeaderView(title: 'BAR', subtitle: 'CHART')`.

We should still see what we saw before, but now we can reuse this view in our next two examples. Lastly, we want to make it so that we display seven capsules (covering the week from Sunday to Saturday). Let's do this now.

Capsule with dynamic data

For our Capsule, we need to pass two values, one for `day` and another for `value` that we use at the top of the bar, which we also use for the height. Setting this up is similar to what we did with `HeaderView`. Open the `Capsule` view and file and, above the `body` variable but below the struct declaration, add the following variables:

```
let value: Int
let day: String
```

Then under `SwiftPreviews`, update the preview variable by replacing `HeaderView()` with the following:

```
CapsuleView(value: 75, day: 'S')
```

Next, let's update our `Text('99')` view with `Text('\'(value)')`. Then, update the Capsule height with `CGFloat(value)`.

Finally, update the `Text('M')` view with `Text(day.uppercased())`, then back in `BarChartView`, update `CapsuleView()` with `CapsuleView(value: 75, day: 'S')`.

Now, we want to display seven of these `CapsuleViews`, so press *Command + click* on the `CapsuleView` text and select **Embed in HStack**. Sometimes, holding *Command* and clicking doesn't work, so you might have to restart Xcode or do it manually.

Now, copy and paste seven CapsuleViews into the `HStack`, as follows:

```
HStack {  
    CapsuleView(value: 75, day: 'S')  
    CapsuleView(value: 100, day: 'M')  
    CapsuleView(value: 50, day: 'T')  
    CapsuleView(value: 25, day: 'W')  
    CapsuleView(value: 40, day: 'T')  
    CapsuleView(value: 25, day: 'F')  
    CapsuleView(value: 40, day: 'S')  
}
```

Mix up the data however you'd like. When you've finished, notice that our data is not lining up correctly in the **Preview**:

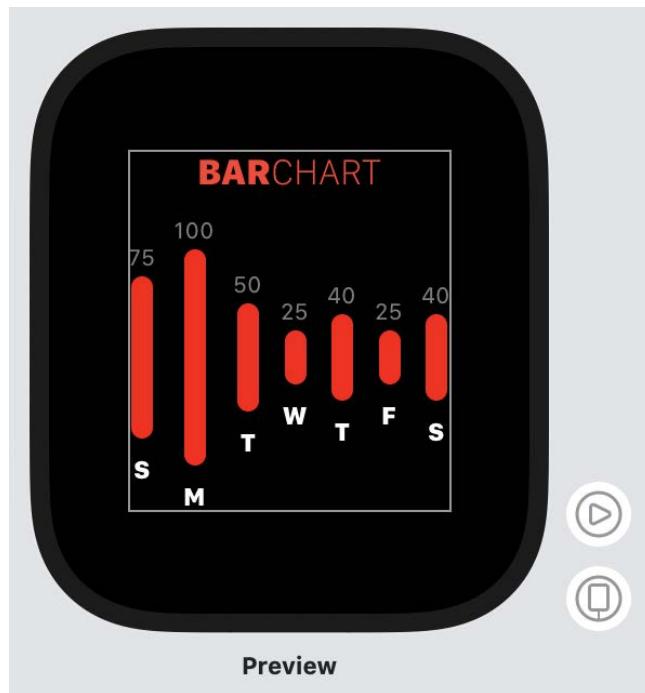


Figure 2.12

The `HStack` container we are using needs the alignment set. Add `HStack(alignment: .bottom)`. You should now see that all of the capsules, no matter their heights, line up along the bottom of our view:



Figure 2.13

We are done with Bar Charts, so we can now move on to Activity Rings next.

Activity Ring

In this section, we want to create an **Activity Ring**. First, we need to create our container and HeaderView. Open the RingView file and find the following line inside the body variable:

```
Text('Ring View ')
```

Once found, replace it with the following code:

```
 VStack {  
     HeaderView(title: 'ACTIVITY', subtitle: 'RING')  
     // Ring goes here  
 }
```

We now just need to add the code for our **Activity Ring**. Find the following comment:

```
// Ring goes here
```

Replace this with the following code:

```
ZStack { // Step 1
    Circle() // Step 2
        .stroke(lineWidth: 20)
        .fill(Color(.darkGray))

    Circle() // Step 3
        .trim(from: 0.5, to: 1)
        .stroke(Color(.red), style: StrokeStyle(lineWidth: 12,
            lineCap: .round, lineJoin: .round))
        .rotationEffect(.degrees(180))
        .rotation3DEffect(.degrees(180), axis: (x: 1, y: 0, z:
    0))
}
.frame(width: 130, height: 130) // Step 4
.rotationEffect(.degrees(90), anchor: .center)
.padding(.top, 10)
```

We now have an activity ring, but let's discuss what we just added:

1. We are using a `ZStack` as our container. Since the `ZStack` lets us stack views on top of each other, it's a no-brainer what to use here.
2. We have two Circles in a `ZStack` (which allows us to overlap the Circles). The first circle in our `ZStack` has a stroke line width of 20 and a fill color of Dark Gray.
3. In the other Circle, we have `trim` set to 0.5, which specifies how much of the ring is filled. In this case, 0.5 equals 50% of the ring. We added a `StrokeStyle`, which lets us add a round `lineCap` and `lineJoin` with a `lineWidth` of 12. We have two rotation effects applied to this circle. We use this to flip the inner ring, filling it from the top and rotating clockwise. If you remove one of the rotation effects, it flips and makes the ring go counterclockwise.
4. Our `ZStack` also has a rotation of 90 degrees, which means that it starts from the center and top of the circle at the 12 o'clock position.

In the **Preview**, you should see the following:



Figure 2.14

We have completed our Activity Ring, and we can now move on to our last chart, the Wedge.

Creating our wedge

In our final view, we are going to make a wedge shape. To simplify this next shape, I have already created the Wedge shape for you. Open `WedgeView` in `Views`. Now, just like we did for the **Bar Chart** and **Activity Ring**, let's add our `HeaderView` inside a `VStack`. Add the following code inside the `body` variable, replacing `Text ('Wedge Chart')`:

```
VStack {  
    HeaderView(title: 'WEDGE', subtitle: 'CHART')  
    // Last step  
}
```

Good, now we have our title displaying before we create our wedge. We need to create an array of wedges. Above the `body` variable, add the following array:

```
let wedges = [
    Wedge(startAngle: -43, endAngle: 43, color: Color.blue),
    Wedge(startAngle: 43, endAngle: 150, color: Color.green),
    Wedge(startAngle: 150, endAngle: -43, color: Color.red)
]
```

Our array creates three wedges we can use to make our `WedgeShape`. You can mess with the start and end angles and change the color of each wedge. Finally, replace the comment 'Last step' with the following code:

```
ZStack {
    ForEach(0 ..< wedges.count) {
        WedgeShape(
            startAngle: Angle(degrees: self.wedges[$0].startAngle),
            endAngle: Angle(degrees: self.wedges[$0].endAngle),
            lineWidth: 24
        )
        .foregroundColor(self.wedges[$0].color)
    }
}.frame(width: 140)
```

In the preceding code, we use a `ZStack` to hold all of the wedges. We then use a `ForEach` loop and loop through our `wedges` array. Each time we loop through, we create a new `Wedge` shape, taking the start and end angle along with the color from our array.

Summary

In this chapter, we got to work on some basic SwiftUI views. This chapter was a primer to help you get a bit more comfortable using SwiftUI. From here on out, our UIs will become more complex. In the next chapter, we will build an NBA Draft app for the Apple Watch. We will work with animations as well as loading data from a plist. I hope you're excited as I am because this is the kind of stuff that made me want to write this book.

3

NBA Draft – Watch App

Anyone that knows me knows that I am a huge sports fan. My first love is basketball. I have played and watched basketball for as long as I can remember. When I first decided to write this book, I knew that making a basketball app was going to happen. I built an NFL watch app for the New England Patriots, but I never got around to doing one for the NBA, so here we are. We are going to build a watch app in this chapter. We'll work off a custom design, which means we will set up our app to use custom colors and fonts.

There might be a lot thrown at you in this chapter, so take your time—a lot of these same techniques are used throughout the book. The more we do, the more comfortable you should get.

In this chapter, we'll work with the following:

- SwiftUI design in watchOS
- Custom list view background
- View modifiers
- Loading data from a property list
- Digital crown scrolling

Technical requirements

The code for this chapter can be found here: <https://github.com/PacktPublishing/SwiftUI-Projects/tree/master/Chapter03>

Building our watch UI

As we start to build our first SwiftUI app together, we will stick to the same format in each app we make. We'll work with the design first and then go back and add the data. Our app, in this chapter, uses a property list (.plist) for the data. We'll cover this once we get to it, but for now, let's look at our design. We have our menu, draft by round, and the draft card details from left to right:



Figure 3.1

One thing that is different about this book is we will be working with design specs. SwiftUI is excellent for prototyping apps. We can build out the app before we add any real data. To me, this is massive, and something I love about SwiftUI. I also feel some developers may not have a ton of experience working with a designer, so I wanted to give my apps a more accurate look and feel.

In this chapter, we will work together with the design specs. I have included different screenshots for you to point out features. In the future, you can challenge yourself to go back and redo the designs without my help. You have some reusability from this book, and I hope you use it and let me know on Twitter (@thedevme).

Walking through the design specs

Every app in this book has design specs for you, which means you can access font sizes, spacing, images, colors, and anything else relevant to the app. You will see that my numbers will be exact, and you will also know where I changed them. Sometimes, the spacing is off, or the design was not right on the device and required a tweak. Design specs are a guideline, but you can take the designs and recreate them into something different; once you understand how to do it with someone else's designs, you will be able to change them and make them your own easily.

Open the Chapter 3 project file, and you will see the same structure from now on. Every design chapter will have a specs folder, a starter folder, and a completed folder. Open the specs folder, and you will see the following:

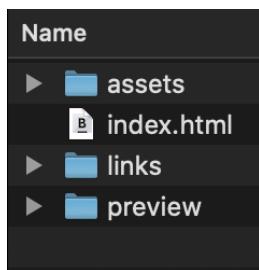


Figure 3.2

All you need to do is open `index.html` in a browser of your choice, and you will see the sketch designs:

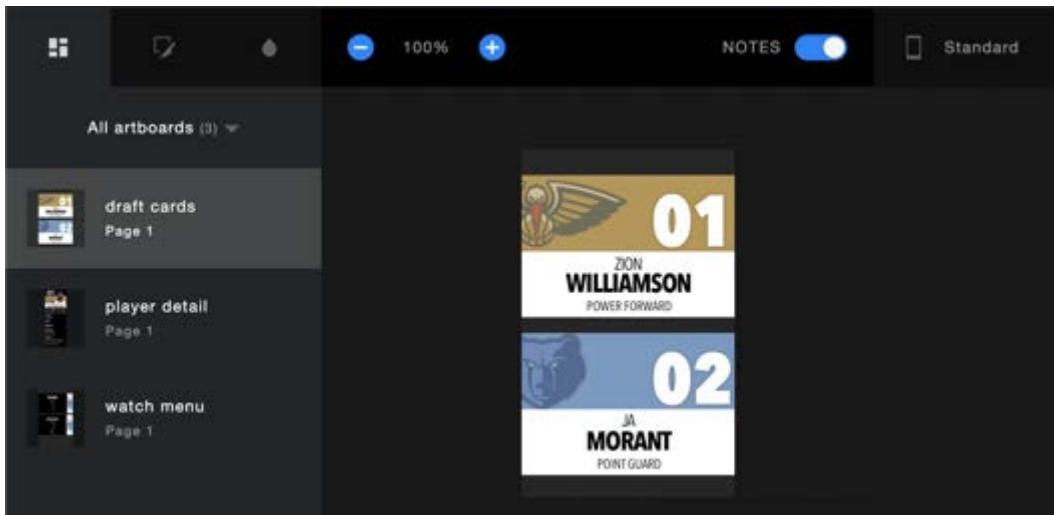


Figure 3.3

Inside `index.html`, you will see every layout for the app, and you can interact with these designs. For example, click on **01** above Zion Williamson's name, and you will see the following:

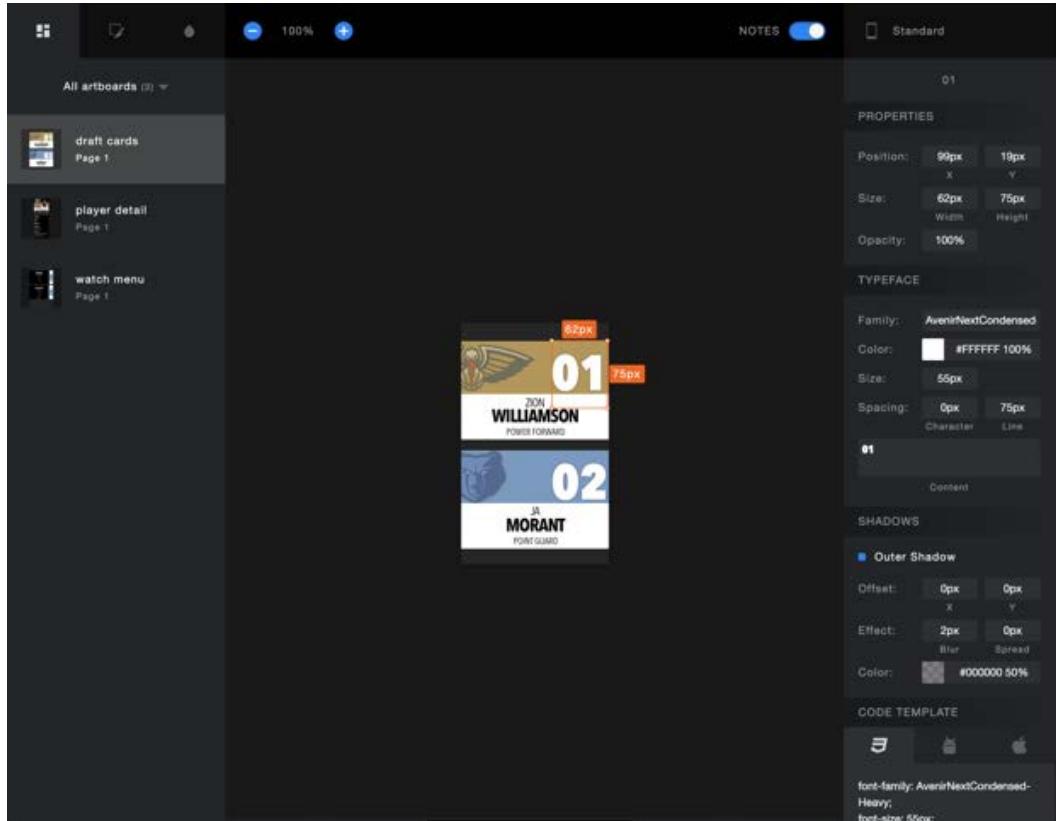


Figure 3.4

When you select **01**, you see the font size, color, font, family, and more. Take a few minutes to browse the design—it's a small app—so that you are familiar with what I am referring to.

Before we dive into the design, let's discuss accessibility and fonts. These are things that are typically ignored or left until the last minute, but I wanted to bring them up as something that we will focus on throughout the book.

Accessibility and fonts

In every app we work on from now on, we are going to use a custom font. The fonts we'll be using throughout this book are in each project file for you. But I wanted to go over fonts in general and how we'll use this class.

We used system fonts in the previous chapter, but I did not cover them in depth. Apple would prefer fonts that can scale, and even though we are making a watch app in this chapter, I would like to make it a habit to use scalable fonts in every app.

Let's look at how system fonts work first.

System fonts

In Storyboards, we had a panel to set your fonts manually, but in SwiftUI, it is effortless to append a font to `Text ("Craig")`. Let's look at a couple of examples again of system fonts:

```
Text ("Craig Clayton") .font (.headline)
Text ("Craig Clayton") .font (.subheadline)
Text ("Craig Clayton") .font (.title)
Text ("Craig Clayton") .font (.largeTitle)
Text ("Craig Clayton") .font (.body)
Text ("Craig Clayton") .font (.callout)
Text ("Craig Clayton") .font (.footnote)
```

In this example, we are using a system font by text style. These text styles use a combination of different predefined font sizes and weights. These predefined text styles update their font size when a user changes their device's font size; they adjust accordingly.

Now, we can use individual font weights by doing the following:

```
Text ("Craig Clayton") .fontWeight (.black)
Text ("Craig Clayton") .fontWeight (.bold)
Text ("Craig Clayton") .fontWeight (.semibold)
Text ("Craig Clayton") .fontWeight (.ultraLight)
Text ("Craig Clayton") .fontWeight (.light)
Text ("Craig Clayton") .fontWeight (.regular)
Text ("Craig Clayton") .fontWeight (.medium)
Text ("Craig Clayton") .fontWeight (.thin)
Text ("Craig Clayton") .fontWeight (.heavy)
```

These are all of the options you have available by font weight. Lastly, we can update the font weight and font size together by doing the following:

```
Text ("Craig Clayton") .font (.system(size: 60))
```

If you are like me and work with designers, you probably do not use system fonts too much. Let's look at how we can do the same thing but with custom fonts.

Custom fonts

In iOS, custom fonts and accessibility have not been easy to work with, but Apple releases a new update every year to make it better. We now can use **Dynamic Type** with custom fonts by just setting some initial values and letting Apple take care of the rest. If you would like to use custom font text styles, you can create your own custom class to declare your text styles. Open `CustomTextStyle`, which is located inside the `_theming` folder, inside of the `Utilities` folder. I have created some styles that you can use:

```
case body
case callout
case caption1
case caption2
case footnote
case headline
case subheadline
case title1
case title2
case largeTitle
case custom(CustomFont, Int)
```

The first 10 text styles are just like what we used with Apple, but we can assign a custom font and weight style to each style. Below these cases, we have a variable that sets the size for each style:

```
var size: Int {
    switch self {
        case .body:           return 16
        case .headline:        return 24
        case .subheadline:     return 24
        case .largeTitle:      return 34
        case .title1:          return 30
    }
}
```

```
        case .title2:      return 24
        case .callout:     return 16
        case .caption1:    return 12
        case .caption2:    return 11
        case .footnote:    return 13
        case .custom(_, let size): return size
    }
}
```

Next, under `size`, you see the name, and this assigns the font weight for my custom font. If you open up the `ScaledFont` file, look for the `CustomFont` enum, and you will see that each text style has a font assigned to it. I also added a few others that you might want to use for messing around with different fonts.

Since I am not a designer, I didn't create a set of styles for each app. I just adjusted fonts to my liking, instead of trying to stick with `.body` or `.headline`. I like setting it up so that I can use any font and font size I want. You might have noticed in `CustomTextStyle` that there is a `custom` case, and that is what we're using in this book. So, when you use custom fonts in SwiftUI, the syntax looks like the following:

```
Text("Craig Clayton").font(.custom("NameOfFont", size: 50))
```

There are many reasons I do not like this, and the main one is that I would have to remember the exact spelling of each font. Since I am the worst at spelling, I created a class so that I can quickly type it out and get code hints:

```
Text("Craig Clayton").custom(font: .bold, size: 50)
Text("Craig Clayton").custom(font: .demibold, size: 50)
Text("Craig Clayton").custom(font: .medium, size: 50)
Text("Craig Clayton").custom(font: .ultralight, size: 50)
Text("Craig Clayton").custom(font: .heavy, size: 50)
Text("Craig Clayton").custom(font: .regular, size: 50)
```

Now that we've got fonts out of the way, let's have some fun by designing our first menu with some custom fonts.

Designing the menu

We are going to build out the menu first. Our menu has a couple of labels and a background graphic for our button:



Figure 3.5

We need a list here to create our menu. Let's start by adding a list that creates two items. Inside of `ContentView`, replace `Text ("Content View")` with the following:

```
List { // Step 1  
  
    ForEach(0..<2) { _ in // Step 2  
        // Add next step here  
    }  
}  
.listStyle(CarouselListStyle())  
.navigationBarTitle(Text ("NBA Draft"))
```

We have a few lines of code added here, but let's go through each step to make sure you understand what we are doing:

1. List has `listStyle` set to `CarouselListStyle()`, and this gives us a carousel effect out of the box. We are also setting `navigationBarTitle` to NBA Draft.
2. We are using `ForEach` to create two list items.

We have `List` set up, and we now want our menu buttons, but we will need to create a background for it first. After the `body` variable's last curly brace, add the following variable:

```
var background: some View {  
    Rectangle().fill(Color("CardBackground"))  
}
```

We are creating a view that we will add to our cards. I am making it this way as adding the entire code as a modifier would be a bit confusing. Next, let's add the remaining code. Add the following code by replacing the `// Add next step here` comment with the following:

```
NavigationLink(destination: EmptyView()) { // Step 1  
    ZStack { // Step 2  
        Image("draft-menu-background").resizable() // Step 3  
  
        VStack(spacing: -15) { // Step 4  
            Text("ROUND")  
                .custom(font: .bold, size: 16) // Step 5  
            Text("1")  
                .custom(font: .ultralight, size: 70)  
            }.offset(x: -10, y: 10)  
    }  
} // Step 6  
.listRowInsets(EdgeInsets(top: 0, leading: 0, bottom: 0,  
    trailing: 0))  
.listRowBackground(background)
```

We just added a bit of code here, and you should be familiar with everything you see here, but let's still cover each step:

1. Here, we are creating a `NavLink` for each link we make. Inside of the parentheses, we set our destination to `EmptyView()`. Using `EmptyView()` is excellent for prototyping as we can use this until we create the view we are going to. Our other style, `listRowBackground`, is used to set the background to an image called `draft-menu-background`. Finally, we add `listRowInsets` for padding on all sides of our list row.
2. We have `VStack`, in which we set the alignment to `.center` and spacing to `-15`. We added padding to the beginning and top of `VStack` and set the height to `90`.
3. Inside of `VStack`, we have two `Text` views. Each `Text` view has a custom font set to it.

You should now see the following in Swift Previews:



Figure 3.6

We now have our menu created; later in this chapter, we will update this view to use dynamic data from a property list (`.plist`). Let's move on to adding the next view, which is the draft round cards.

Designing draft cards

We now need to build the next screen that the user sees. After a user selects a draft round, they will see every player drafted in that round. A team drafts each player, displaying the team logo, team color, player name, position, and when they were picked. Let's take another look at the design to familiarize ourselves with what we are doing next:



Figure 3.7

You will notice that each card is slightly tilted, and each card flips down. Let's first add the title and the line above the cards.

Adding our properties

Open `DraftList`, located inside of the `Views` folder. Just above the `body` variable, there is an `// add properties here` comment. Replace it with the following:

```
@State private var currentIndex = 0.0 // Step 1
@State private var isShowingDetail = false // Step 2
private let numberOfVisibleCards = 3 // Step 3
private let testPicks = 10 // Step 4
```

We just added four variables we will need in this view. Let's discuss what each one does:

1. `currentIndex`: Keeps track of the current index card we are on.
2. `isShowingDetail`: Determines whether or not we are showing the details view.
3. `numberOfVisibleCards`: We use this variable to set how many cards you can see at once when they are stacked.
4. `testPicks`: A temporary number we use to determine how many picks we have. Later, this changes to a dynamic property.

We have added all of our properties. Let's move on to the header next.

Adding our header

The more you work with SwiftUI, the more you will love how you can break code up into small chunks. As I am editing this book, I find myself editing my code into even smaller pieces. We will do this quite a bit in this book, and I hope it will help you start doing the same in your code. We can take this first view and break it down into a few parts, but the one we need to focus on now is the header.

Let's replace `Text ("Draft List View")` inside of the `body` variable with the following:

```
 VStack(spacing: 10) { // Step 1
    VStack(spacing: 0) { // Step 2
        Text("ROUND 1")
            .custom(font: .bold, size: 20)
        Divider()
    }

    // Add next step here
}

.navigationBarTitle(Text("By Round"))
```

We created a header in the last chapter; this header is slightly different:

1. We are using `VStack` as a primary container, and the spacing is set to 10. Our `VStack` modifier, `navigationBarTitle`, sets our title to `By Round`.
2. Inside of `VStack`, we have a `Text` view and `Divider()`. We are doing this so that we can control the spacing between round 1 and the divider.

Hit the **Resume** button inside of **Previews**, and you should see the following:

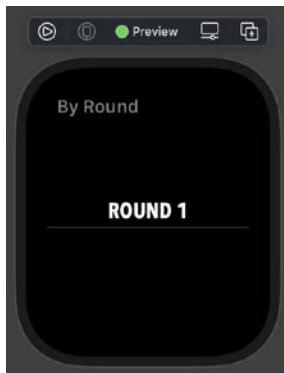


Figure 3.8

Our header is complete now; let's work on the card next.

Adding our card contents

We are going to create the draft card next. The card has a lot of code, so we will break it up into small chunks. We will work backward and build each element; then, we will bring it all together. In this section, we will be writing a lot before we see the code compile. Be patient in this section, and I will let you know when we should see something in Swift Previews. Let's create the top of our card first.

Adding the top of the card

We work on the top of the card first by replacing `// Add next step here` with the following code:

```
var topCard: some View {
    HStack {
        Image("pelicans").frame(height: 56)
        Spacer()
        Text(String(format: "%02d", 1))
            .custom(font: .bold, size: 50)
    }
    .frame(height: 48)
    .background(Color("pelicans"))
}
// Add bottomCard
```

Let's go over what we just added:

1. We have an `HStack`, and this is the top of our card. It has a height of 48, and the background color is `pelicans`, which is a color created in our asset catalog.
2. Inside of `HStack`, we have an `Image` and `Text` with `Spacer` in between each item. We discussed `Spacer` in detail, but it pushes `Image` and `Text` to the opposite sides of `HStack`. The image we are using is `pelicans` for now, as this is just for prototyping. Finally, we have some formatted text that takes any number between 1 and 9 and appends 0 in front of it. We use our custom font and set the style to bold with a font size of 50.

Our card needs a bottom; let's add it now.

Adding the bottom of the card

We are moving on to the bottom of the card. Find // Add `bottomCard` here, add the following variable:

```
var bottomCard: some View {
    VStack(spacing: -5) { // Step 1
        Text("ZION") // Step 2
            .custom(font: .ultralight, size: 13)
        Text("WILLIAMS")
            .custom(font: .bold, size: 20)
        Text("POWER FORWARD")
            .custom(font: .ultralight, size: 10)
    }
    .frame(minWidth: 0, maxWidth: .infinity) // Step 3
    .frame(height: 60, alignment: .center) // Step 4
    .background(Color.white)
    .foregroundColor(.black)
}
// Add fullCard here
```

Let's break down this code:

1. We are using a `VStack` that has a spacing of -5. The `VStack` has a height of 60 and the alignment set to `.center`. We also have the background set to `.white` and the foreground set to `.black`.

2. Inside of `VStack`, we have three `Text` views, which each have a custom font set.
3. We set the frame's `minWidth` value to `0` and `maxWidth` to `.infinity`, and doing this tells `SwiftUI` to fill the screen.
4. We set the frame height here and make sure that we are aligned to the center.

Now that we have a top and bottom, we can create a variable that combines them both. Replace `// Add fullCard here` with the following variable:

```
var fullCard: some View {
    VStack(alignment: .leading, spacing: -3) { // Step 1
        topCard // Step 2
        bottomCard
    }
}
```

Here, we have a lot going on, and we only have two containers. Let's break it all down:

1. First, we have a `VStack`, and we set the alignment to `.leading` and the spacing to `-3`. Setting the spacing to `-3` is done to get the gap to be a bit smaller.
2. I am adding each `topCard` and `bottomCard`.
3. We created all of the variables we need for this section. Look back inside of the `body` variable. Then, replace `// Add next step here`, add the following code:

```
ZStack {
    // Add next step here
}
.focusable(true)
.digitalCrownRotation(
    $currentIndex.animation(),
    from: 0.0,
    through: Double(testPicks - 1),
    by: 1.0,
    sensitivity: .low
)
```

1. First, we are using a `ZStack` as the container for our card, which is the main container for our stack cards. Let's look at each modifier we added to the `ZStack`.

2. `focusable` makes sure that our view is in focus. We are setting it to `true`.
3. Next, we are adding `digitalCrownRotation` to the `ZStack`. `digitalCrownRotation` takes a few parameters. In our first parameter, we track the current index and give it the card flip rotation as we use the digital crown. Next, we get the minimum value, which is `0.0`. Then, we get the `through` value, which is the total number of items minus 1. We then set the stride to `1.0` and, finally, set the sensitivity to `.low`.
4. We want to create all of our cards, so we will use `ForEach` to create a `fullCard`. Later in this chapter, we will update this code with real data. Replace `// Add next step here` with the following code:

```
ForEach((0...testPicks).reversed(), id: \.self) { index in
    fullCard
        .cardTransformed(self.currentIndex, card: index)
        .onTapGesture {
            self.isShowingDetail.toggle()
        }
        .sheet(isPresented: self.$isShowingDetail) {
            DraftPlayerDetailView()
        }
}
```

Because we added all of the code inside of variables, we were able to clean up the code. Let's break down the `ForEach` we just added:

1. Here, we use a `ForEach` to create our cards. For prototyping, we use `testPicks` just for testing purposes. When we use real data, this variable is replaced. When using `ForEach`, each item needs to have a unique identifier. In this case, we are using the object itself for the unique value.
2. `cardTransformed` is a custom view modifier. If you open `CardStackModifier`, you can check out how it works. Overall, this class rotates each card and does the card flip animation.
3. `onTapGesture` allows us to add a tap gesture to the card.
4. When `onTapGesture` is tapped, it will toggle our `self.isShowingDetail.toggle()` method.

5. We have attached `.sheet` to the `VStack`. When the user taps the card, we show a details view; for now, we will use `EmptyView()`. You will see `self.$isShowingDetail`, which uses `State`. I cover `State` and `Binding` in more detail later, in *Chapter 5, Car Order Form – Data*.
6. When `self.isShowingDetail` toggles, it presents `DraftPlayerDetailView()`.

When you finish, you will see the following in Swift Previews by hitting **Resume**:



Figure 3.9

Note

If you have experience in iOS development, you might be wondering why I have not built the project yet. I have, in a way, but I have just adjusted my workflow to match SwiftUI. What I mean is that instead of always building the iPhone to see the progress, I do this with Previews. Now, sometimes, Previews does not allow you to do this all the time, but I find that I work much faster using this method.

We have one more view to design before we move on to the data. Let's move on to that next.

Designing the prospect details

We are going to create the details view next. When you tap on each card, we want to display a detailed view that shows more information about the prospect. Let's look at what this screen looks like:



Figure 3.10

I divided the details view into three parts, so it's easier to know what we are working on first. Here are the three sections:

- Header
- Stats
- Info



Figure 3.11

Breaking up a design before coding is what I have been doing since UIKit, but I do it even more now in SwiftUI. We will start with the header first.

Creating a details prospect header

Now that we have a plan, let's jump into `DraftPlayerDetailView` and get started. After the last curly brace for the `body` variable, you are going to add the following:

```
var header: some View {
    Text("header here")
}

var info: some View {
    Text("info here")
```

```
}

var stats: some View {
    Text("stats here")
}
```

We have our three sections done, just like our design. Next, replace `Text ("Draft Player Detail View")` with the following:

```
ScrollView {
    VStack {
        header
        info
        stats
    }
}
```

We just added all of our content into a `ScrollView` view.

When you are done adding the code, your view should now be structured the same as mine:

```
struct DraftPlayerDetailView: View {
    var body: some View {
        ScrollView {
            VStack {
                header
                stats
                info
            }
        }
    }

    var header: some View {
        Text("header here")
    }

    var info: some View {
        Text("info here")
    }

    var stats: some View {
        Text("stats here")
    }
}
```

Figure 3.12

Now that we have our structure, let's work on the header.

Creating the details prospect header

We are going to work on the header, but before we do this, let's look at precisely what we are going to be making first:

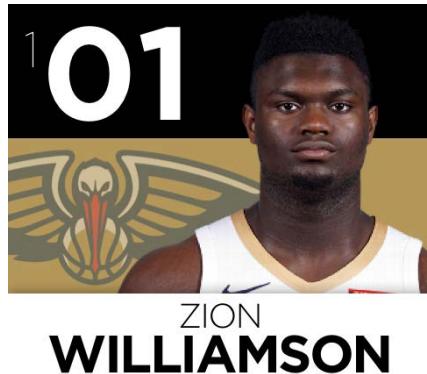


Figure 3.13

This design is similar to the card we created earlier; let's break this down into two sections. The top area has the round, and the pick—the player—went in the draft. Add the following, replacing `Text ("header here")`, which is located in the new header variable we just created:

```

VStack(alignment: .leading, spacing: 0) { // Step 1
    VStack(spacing: -35) { // Step 2
        HStack(alignment:.top, spacing: 0) { // Step 3
            Text("1")
                .custom(font: .ultralight, size: 19)
                .offset(x: -2, y: 15)
            Text(String(format: "%02d", 1))
                .custom(font: .bold, size: 70)
        }
        .padding(.leading, 12)
        .offset(x: -45, y: -17)
    }
    // add bottom Card here
}
}

```

Let's break down what we just added:

1. We are using a `VStack` as our main container, which has an alignment of `.leading` and `spacing` set to 0.
2. First, put our header into a `VStack` as our container. This container has `spacing` set to -35.
3. Inside of `VStack`, we have an `HStack` with the alignment set to `.top` and its `spacing` set to 0. We also have the `leading padding` set to 12 and the `x` and `y` offsets set to -45 and -17, respectively.
4. Now, inside of the `HStack`, we have two `Text` views that have custom fonts. As we did earlier, we formatted the text, which takes any number between 1 and 9 and appends 0 in front of it.

The top of our card is done. Let's move on to the bottom half. Replace `// add bottom Card here` with the following code:

```
VStack(spacing: -1) { // Step 1
    HStack { // Step 2
        Image("pelicans").offset(x: 29) // Step 3
        Image("zion-williamson").offset(x:0, y: -25)
    }
    .frame(height: 58)
    .frame(minWidth: 0, maxWidth: .infinity)
    .background(Color("pelicans"))

    VStack(spacing: -5) { // Step 4
        Text("ZION") // Step 5
            .custom(font: .ultralight, size: 13)
        Text("WILLIAMSON")
            .custom(font: .bold, size: 20)
    }
    .frame(minWidth: 0, maxWidth: .infinity)
    .background(Color.white)
    .foregroundColor(.black)
}
```

Let's go over the code we just added:

1. We wrap the bottom of our card inside of a `VStack`, which has a spacing of `-1`.
2. Inside of `VStack`, we have an `HStack`, which has its height set to `58`. We are also using `minWidth` and `maxWidth`, which are set to `0` and `.infinity`, respectively. By doing this, it puts our stack's width to infinity. Finally, we set the background color to `pelicans`, which is a color in the asset catalog.
3. Inside of the `HStack`, we have two `Image` views that both have offsets.
4. Outside of the `HStack`, we have another `VStack`, which has a spacing of `-5`. This `VStack` has its frame set to `minWidth` and `maxWidth`, which are set to `0` and `.infinity`, respectively. Lastly, we have the background color set to `white`, and the foreground color set to `black`.
5. Inside of the `VStack`, we have two `Text` views that contain our prospect's first and last names. These `Text` views both use custom fonts.

When you're finished, you will see the following in Swift Previews:



Figure 3.14

We are one-third of the way there. Now, let's move on to the stats section of our detailed view.

Creating detailed prospect stats

We have completed the header, and now we are going to add the stats section. Let's take a look at what we need to create first:

PTS	REB	AST	EFG
22.6	8.9	2.1	70.8

Figure 3.15

Now that we have seen our design, let's move on by replacing `Text ("stats here")`, which is inside of the `stats` variable, with the following:

```
HStack {  
    ForEach(0..<4) { _ in  
        HStack {  
            Spacer(minLength: 4)  
            VStack {  
                Text("PTS")  
                    .custom(font: .bold, size: 12)  
                    .offset(y: 1)  
                Text("99.9")  
                    .custom(font: .ultralight, size: 18)  
            }  
            .background(Image("stat-bg-small"))  
            Spacer(minLength: 4)  
        }  
    }  
    .padding(.top, 10)  
    .padding(.horizontal, 5)
```

Let's break down the stats section:

1. We use an `HStack` for our stats container. This container has padding on top set to 10 and horizontal (left and right side) padding set to 5.
2. Inside of the `HStack`, we have a `ForEach` that creates four stat items.
3. Inside of `ForEach`, we have an `HStack` that is our stat container.
4. Inside of the `HStack`, we have two `Spacer` (`minLength: 4`), which pads our `VStack` with 4 pixels on each side.
5. Inside of the `HStack`, we also have a `VStack`. This `VStack` has two `Text` views, which display the stat name and the stat value. Both `Text` views have a custom font set to each of them.

If you hit **Resume** in Swift Previews, you'll see the following:



Figure 3.16

You'll notice that the stats are cut off. If you hit the play button next to the device, you can scroll the view:



Figure 3.17

After you hit play, you will be able to scroll down in the watch, and you should see the following:



Figure 3.18

We are now done with the stats section. Let's move on to the info section.

Creating detailed prospect info

In our last section, we will display the prospect info:

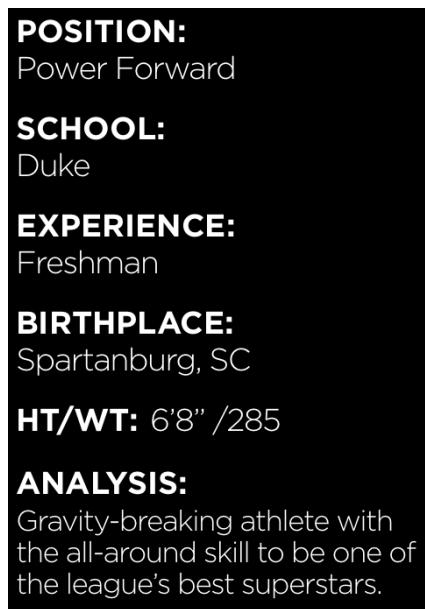


Figure 3.19

The info section has all the Text views. Replace `Text ("info here")` with the following:

```
Group { // Step 1
    HStack {
        VStack(alignment: .leading) {
            Text("POSITION:").custom(font: .bold, size: 16)
            Text("PF").custom(font: .ultralight, size: 16)
        }
        Spacer()
    }

    HStack {
        VStack(alignment: .leading) {
            Text("SCHOOL:").custom(font: .bold, size: 16)
```

```
        Text("School").custom(font: .ultralight, size: 16)
    }
    Spacer()
}

// Add next step here

}

.padding(.top, 10)
.padding(.horizontal, 5)
```

We added the first two player info sections:

1. We use Group as our container. This group has a top padding set to 10 and the horizontal padding set to 5.
2. Inside of the Group container, we have two VStack that have two Text views, both inside an HStack with a spacer. The reason for having them in the HStack is to push the text to the left side. The alignment is set to .leading, and each Text view has a custom font.
3. This HStack has its spacing set to 5, and it also contains two Text views with custom fonts.

We have more stats that are formatted the same. Replace // Add next step here with the following:

```
HStack {
    VStack(alignment: .leading) {
        Text("EXPERIENCE:").custom(font: .bold, size: 16)
        Text("2").custom(font: .ultralight, size: 16)
    }
    Spacer()
}

HStack {
    VStack(alignment: .leading) {
        Text("BIRTHPLACE:").custom(font: .bold, size: 16)
        Text("Cleveland, OH").custom(font: .ultralight, size:
16)
```

```

    }
    Spacer()
}

// Add the last step here

```

We just added two more sections that are the same as what we did before. Replace // Add the last step here with the following:

```

HStack {
    HStack(spacing: 5) {
        Text("HT/WT:").custom(font: .bold, size: 16)
        Text("6'0 / 999").custom(font: .ultralight, size: 16)
    }
    Spacer()
}

HStack { // Step 1
    VStack(alignment: .leading) { Step 2
        Text("ANALYSIS:").custom(font: .bold, size: 16)
        Text("Analysis goes here")
            .lineLimit(nil)
            .multilineTextAlignment(.leading)
            .fixedSize(horizontal: false, vertical: true)
            .custom(font: .ultralight, size: 14)
    }
    Spacer()
}

```

We are adding two more sections; the first one is the same as the other, but let's talk about the last one as it's slightly different:

1. Our last container is a `VStack` that has a `.leading` alignment.
2. It has two `Text` views with custom fonts but one that supports multiline text. We set the line limit to `nil`, which lets us have a dynamic length. `multilineTextAlignment` is set to `fixedSize`, with `horizontal` set to `false` and `vertical` set to `true`. Setting `vertical` to `true` means we want to fix the height of the `Text` view.

We are done with the player details views. If you hit **Resume** in Swift Previews and then play, you should be able to scroll the entire view. Now that we have all of the views prototyped, we need to connect them all.

Connecting our views

We need to get everything connected. First, inside of `ContentView`, please find `EmptyView()` and replace it with `DraftList()`. Now, our app goes from the menu to the draft list.

You can now launch the app and go from the menu to the draft list to a details view. Now that our design is complete, we need to bring in the data. Next, we need to do some cleanup.

Refactoring views

Our views have too much code, and we should refactor them a bit. Dealing with small chunks of code makes it easier to manage as well as being able to reuse views. We refactor each of the three main views, but if you see a place where you can refactor, have at it.

Refactoring `ContentView`

The first view we are going to refactor is inside of `ContentView`. Open `ContentView` and highlight everything inside of `ForEach`, and then hit *Command + X* (cut). Then, open `DraftRoundItemView` and highlight `Text ("Draft Round Item View")` and hit *Command + V* (paste) to replace this `Text` view. Go back to `ContentView()` and cut the `background` variable from `ContentView()` and paste it after the `body` variable in `DraftRoundItemView()`. Hit **Save**.

Back inside of `ContentView` again, add `DraftRoundItemView()` inside of `ForEach`. When you're finished, `ContentView` should look like the following:

```
struct ContentView: View {  
    var body: some View {  
        List {  
            ForEach(0..<2) { _ in  
                DraftRoundItemView()  
            }  
        }.listStyle(CarouselListStyle())  
        .navigationBarTitle(Text ("NBA Draft"))  
    }  
}
```

```

    }
}

```

There was not much code for this section, but you want to keep them inside of their files when you have list items. Let's move to `DraftList` and refactor this file.

Refactoring `DraftList`

We want to move everything after the divider, our card, and move it into its file. Open `DraftList` and highlight `fullCard`, `topCard`, and `bottomCard`. Then, open `DraftCardView`, and after the `body` variable, paste in all three variables. When you are finished, `DraftCardView` should look like the following:

```

struct DraftCardView: View {
    var body: some View {
        Text("Draft Card View")
    }

    var fullCard: some View {
        VStack(alignment: .leading, spacing: -3) {
            topCard
            bottomCard
        }
    }

    var topCard: some View {
        HStack {
            Image("pelicans").frame(height: 56)
            Spacer()
            Text(String(format: "%02d", 1))
                .custom(font: .bold, size: 50)
        }
        .frame(height: 48)
        .background(Color("pelicans"))
    }

    var bottomCard: some View {
        VStack(spacing: -5) {
            Text("ZION")
                .custom(font: .ultralight, size: 13)
            Text("WILLIAMS")
                .custom(font: .bold, size: 20)
            Text("POWER FORWARD")
                .custom(font: .ultralight, size: 10)
        }
        .frame(minWidth: 0, maxWidth: .infinity)
        .frame(height: 60, alignment: .center)
        .background(Color.white)
        .foregroundColor(.black)
    }
}

```

Figure 3.20

Now, we will remove all the code inside of `fullCard` and move it to the body. When you are done, you will only have `topCard` and `bottomCard`. Here is what `DraftCardView` should look like when you are finished:

```
struct DraftCardView: View {
    var body: some View {
        VStack(alignment: .leading, spacing: -3) {
            topCard
            bottomCard
        }
    }

    var topCard: some View {
        HStack {
            Image("pelicans").frame(height: 56)
            Spacer()
            Text(String(format: "%02d", 1))
                .custom(font: .bold, size: 50)
        }
        .frame(height: 48)
        .background(Color("pelicans"))
    }

    var bottomCard: some View {
        VStack(spacing: -5) {
            Text("ZION")
                .custom(font: .ultralight, size: 13)
            Text("WILLIAMS")
                .custom(font: .bold, size: 20)
            Text("POWER FORWARD")
                .custom(font: .ultralight, size: 10)
        }
        .frame(minWidth: 0, maxWidth: .infinity)
        .frame(height: 60, alignment: .center)
        .background(Color.white)
        .foregroundColor(.black)
    }
}
```

Figure 3.21

Inside of `DraftList`, replace `fullCard` with `DraftCardView`. Now, we want to move the following code:

```

struct DraftList: View {
    @State private var currentIndex = 0.0
    @State private var isShowingDetail = false
    private let numberOfVisibleCards = 3
    private let testPicks = 10

    var body: some View {
        VStack(spacing: 10) {
            VStack(spacing: 0) {
                Text("ROUND 1")
                .custom(font: .bold, size: 20)
                Divider()
            }
        }
        Cut This ZStack
        ZStack {
            ForEach((0...testPicks).reversed(), id: \.self) { index in
                DraftCardView()
                .cardTransformed(self.currentIndex, card: index)
                .onTapGesture {
                    self.isShowingDetail.toggle()
                }
                .sheet(isPresented: self.$isShowingDetail) {
                    DraftPlayerDetailView()
                }
            }
            }
            .focusable(true)
            .digitalCrownRotation(
                $currentIndex.animation(,
                from: 0.0,
                through: Double(testPicks - 1),
                by: 1.0,
                sensitivity: .low
            )
        }
        }
        .navigationBarTitle(Text("By Round"))
    }

```

Cut this also

Cut This ZStack

Figure 3.22

Move all of the highlighted code into `DraftRoundCardView` when you are done with `DraftRoundCardView`, and you should have the following:

```
struct DraftRoundCardView: View {
    @State private var currentIndex = 0.0
    @State private var isShowingDetail = false
    private let numberOfVisibleCards = 3
    private let testPicks = 10

    var body: some View {
        ZStack {
            ForEach(0...testPicks).reversed(), id: \.self) { index in
                DraftCardView()
                    .cardTransformed(self.currentIndex, card: index)
                    .onTapGesture {
                        self.isShowingDetail.toggle()
                    }
                    .sheet(isPresented: self.$isShowingDetail) {
                        DraftPlayerDetailView()
                    }
            }
        }
        .focusable(true)
        .digitalCrownRotation(
            $currentIndex.animation(),
            from: 0.0,
            through: Double(testPicks - 1),
            by: 1.0,
            sensitivity: .low
        )
    }
}
```

Figure 3.23

After refactoring, `DraftList` should look like the following:

```
struct DraftList: View {

    var body: some View {
        VStack(spacing: 10) {
            VStack(spacing: 0) {
                Text("ROUND 1")
                    .custom(font: .bold, size: 20)
                Divider()
                DraftRoundCardView()
            }
        }
        .navigationBarTitle(Text("By Round"))
    }
}
```

Figure 3.24

Build and run the project, and there should be no difference in what you had before.

Refactoring the detail view

In this section, you have the challenge of refactoring the code yourself. Try to refactor this view into three new views.

Open the project files for this chapter, then open the refactor complete folder to see how I refactored it. But as a quick summary, I created three new files:

`DraftPlayerDetailHeaderView`, `DraftPlayerDetailStatsView`, and `DraftPlayerDetailInfoView`. We already split them up into three sections, so it should be an easy refactor.

My updated `DraftPlayerDetailView` file looks like the following:

```
struct DraftPlayerDetailView: View {
    var body: some View {
        ScrollView {
            VStack(alignment: .leading, spacing: 0) {
                DraftPlayerDetailHeaderView()
                DraftPlayerDetailStatsView()
                DraftPlayerDetailInfoView()
            }
        }
    }
}
```

The preceding code is so much easier to read versus what we had earlier in the chapter. Now that we have our views refactored, let's move on to the data. If you open the `Chapter03` folder, you will find the complete folder with everything done up to this point.

Adding watch data

A plist drives our data for this app. There is not much difference when you get data from a plist versus JSON. If I were building this app for the store, I would use a JSON feed. To make this easier, we are not going to do any of the data models together. Please open the chapter files, and you will see a folder called `chapter_assets`, and inside of that folder, you will see `Models`, which will have all of the data we will need for this project. Drag and drop the files from the folder into your project now and make sure you have the extension selected.

Inside the `Models` folder, open the `draft.plist` file and you will see the following:

▼ Root	Array	(2 items)
▼ Item 0	Dictionary	(4 items)
round	String	Round 1
headline	String	Round
subheadline	String	1
► picks	Array	(30 items)
▼ Item 1	Dictionary	(4 items)
round	String	Round 2
headline	String	Round
subheadline	String	2
► picks	Array	(30 items)

Figure 3.25

We have objects that we will use for our round data. We also have picks for each round, and each pick has prospect data:

▼ picks	Array	(30 items)
▼ Item 0	Dictionary	(1 item)
▼ prospect	Dictionary	(14 items)
firstName	String	Zion
lastName	String	Williamson
position	String	Power Forward
image	String	zion-williamson
ht	Number	79
wt	Number	285
experience	String	Freshman
school	String	Duke
birthPlace	String	Spartanburg, SC
analysis	String	Gravity-breaking athlete with the all-around skill to be one of the league's best superstars.
draftPosition	Number	1
round	Number	1
▼ team	Dictionary	(2 items)
name	String	Pelicans
market	String	New Orleans
▼ stats	Array	(4 items)
► Item 0	Dictionary	(2 items)
value	String	22.6
name	String	pts
► Item 1	Dictionary	(2 items)
► Item 2	Dictionary	(2 items)
► Item 3	Dictionary	(2 items)

Figure 3.26

Since what we will cover next is more about Swift rather than SwiftUI, I will skip breaking down the code I am writing for parsing a plist file. Take a minute and look at the Model folder. Everything is all ready to use; we just need to implement it.

Updating the menu with DraftRound

Our plist has the round data, which is mapped to a DraftRound object. Let's update ContentView to get the DraftRound data first. Open ContentView and find the following code:

```
ForEach(0..<2) { _ in
    DraftRoundItemView()
}
```

Replace it with this ForEach:

```
ForEach(draftData) { round in
    DraftRoundItemView(round: round)
}
```

Then, inside of DraftRoundItemView, add the following above the body variable:

```
let round: DraftRound
```

You get an error because we added a variable to our file, and Swift Previews needs the missing argument to be added. Replace DraftRoundItemView_Previews with the following:

```
struct DraftRoundItemView_Previews: PreviewProvider {
    static var previews: some View {
        DraftRoundItemView(round: sampleRound)
    }
}
// Update the extension
extension DraftRoundItemView_Previews {
    static var sampleRound: DraftRound {
        return MockDraftPreviewService.draftRound
    }
}
```

MockDraftPreviewService is a class that creates dummy data for us to use in Swift Previews. Now that our error is gone, let's update DraftRoundItemView so that the data comes from the plist.

Please locate the following Text view:

```
Text("ROUND").custom(font: .bold, size: 16)  
Text("1").custom(font: .ultralight, size: 70)
```

Replace it with the following views:

```
Text(round.headline.uppercased()).custom(font: .bold, size: 16)  
Text(round.subheadline.uppercased()).custom(font: .ultralight,  
size: 70)
```

If you run your project, you will now see our data is working:



Figure 3.27

Now, we want to pass the picks to the next view, so let's work on that next.

Updating the draft list with pick data

The next thing we need to do is pass all of the draft picks to the following view.

To pass data, we need to update `DraftRoundItemView`. We can pass data through a `NavigationLink`. Let's open `DraftRoundItemView`.

Inside of the `body` variable, please find the following:

```
NavigationLink(destination: DraftList())
```

We need to pass our `DraftRound` data, so update `NavigationLink` with the following:

```
NavigationLink(destination: DraftList(data: round))
```

You get an error because we have to set up `DraftList` to take in `DraftRound`. Open `DraftList` and add the following variable above the `body` variable:

```
let data: DraftRound
```

The error you were getting is gone. Next, the error you get is that we need to update `DraftList_Previews`. Add the following extension at the bottom of the `DraftList` file:

```
extension DraftList_Previews {
    static var sampleRound: DraftRound {
        return MockDraftPreviewService.draftRound
    }
}
```

`MockDraftPreviewService` will give us sample data we can use to see our design work in `Previews`. Now, inside of `DraftList_Previews`, update the following:

```
static var previews: some View {
    DraftList()
}
```

Add in the new `data` parameter and pass in `sampleRound`:

```
static var previews: some View {
    DraftList(data: sampleRound)
}
```

All of our errors have been taken care of, and we want `DraftList` to display data from the `plist`. On this screen, we are displaying **ROUND 1**, and we need this to be dynamic. Locate the following Text view:

```
Text ("ROUND 1")
```

Update it with the following Text view:

```
Text (data.round)
```

Now, our header is dynamic and displays the appropriate round.

Updating the Draft List Card

We now need to have the data for each pick passed to `DraftRoundCardView()`. Locate this code:

```
DraftRoundCardView()
```

Update it with the following:

```
DraftRoundCardView(picks: data.picks)
```

You'll see another error. Let's fix this now by opening `DraftRoundCardView`. Delete `private let testPicks = 10`, which gives us more errors, with the following:

```
let picks: [Pick]
```

Next, we need to update the `ForEach` here:

```
ForEach((0...testPicks).reversed())
```

Replace it with the following:

```
ForEach((0...picks.count-1).reversed())
```

Next, inside of `digitalCrownRotation`, update `through: Double(testPicks - 1)` with `through: Double(picks.count - 1)`.

Finally, update Swift Previews with the following:

```
struct DraftRoundCardView_Previews: PreviewProvider {
    static var previews: some View {
        DraftRoundCardView(picks: samplePicks)
```

```

    }
}

extension DraftRoundCardView_Previews {
    static var samplePicks: [Pick] {
        return MockDraftPreviewService.picks()
    }
}

```

Now, we need to pass each prospect as the `ForEach` loops through the data and add our cards. `DraftCardView` is the next view that needs to be updated. Inside of `DraftRoundCardView`, update `DraftCardView` to the following:

```
DraftCardView(prospect: self.picks[index].prospect)
```

Let's move on to the draft card view next as we need to update it now to accept prospect data.

Adding data to DraftCardView

We now have our data passing through, but again, we are getting an error because we have not set up `DraftCardView` to take any parameters. Above the `body` variable, add the following:

```
let prospect: Prospect
```

We now have prospect data passed through. Fix our error, which is `DraftCardView_Previews`. Add the following extension:

```

extension DraftCardView_Previews {
    static var prospect: Prospect {
        return MockDraftPreviewService.currentProspect()
    }
}

```

Now, update `DraftCardView_Previews` with the following:

```
DraftCardView(prospect: prospect)
```

We are done with working on `DraftCardView`. Let's talk about what your challenge will be for this chapter.

Challenge – Updating the details view with prospect data

Now that you've seen how this is done, please take some time and finish updating `DraftCardView`, and then take the time to update `DraftPlayerDetailView` on your own. If you get stuck, please look at the completed files for help, but you should be able to do it independently without the project files.

Summary

In this chapter, we created an entire watch app using SwiftUI. We set up our design first and then learned how to refactor our views into smaller views. Lastly, we learned how to take `plist` data and pass data to each view.

In the next chapter, we will take things a step further by working with SwiftUI inside an iPhone. There are not many changes, but we'll have more UI on the screen at the same time. After the first few chapters, I hope you have seen how amazing SwiftUI is, and you are excited to learn even more in the upcoming chapters.

4

Car Order Form – Design

A passion for cars was something that I never had as a kid and into my adult life. My first car was a hand-me-down from my mom, a 1984 Chevy Nova. Yes, I know it was nothing pretty, but it was better than walking. The first car that caught my attention was a Chrysler 300. After I saw it, I became obsessed, and it was my first goal. I bought a die-cast version of this car and kept it on my desk as a goal.

Since then, I have owned two cars. I never really thought I would want another car as much as the Chrysler 300 until I sat in a Tesla. I do not own a die-cast version of this car yet, but I'll probably have one by the time this book is published. It is a goal for a few years down the road, but goals keep me focused.

When I designed the app for this chapter, I thought it would be fun to use Tesla as the car for my design. In this chapter, we are going to create a fake car-service order form. We'll learn how to take a generic form and customize it.

In this chapter, we'll work with the following:

- SwiftUI custom form design
- Vibrancy blur effects
- Custom modals

Technical requirements

The code for this chapter can be found here: <https://github.com/PacktPublishing/SwiftUI-Projects/tree/master/Chapter04>

The overall design

In this chapter, we are going to work on a custom form design in SwiftUI. As we progress through the book, you'll notice some design elements in SwiftUI and others to dig deeper into the framework. No matter how we have to do it in the end, you'll learn how to take a basic template and turn it into a beautiful design.

This is what we are going to work on in the next two chapters:

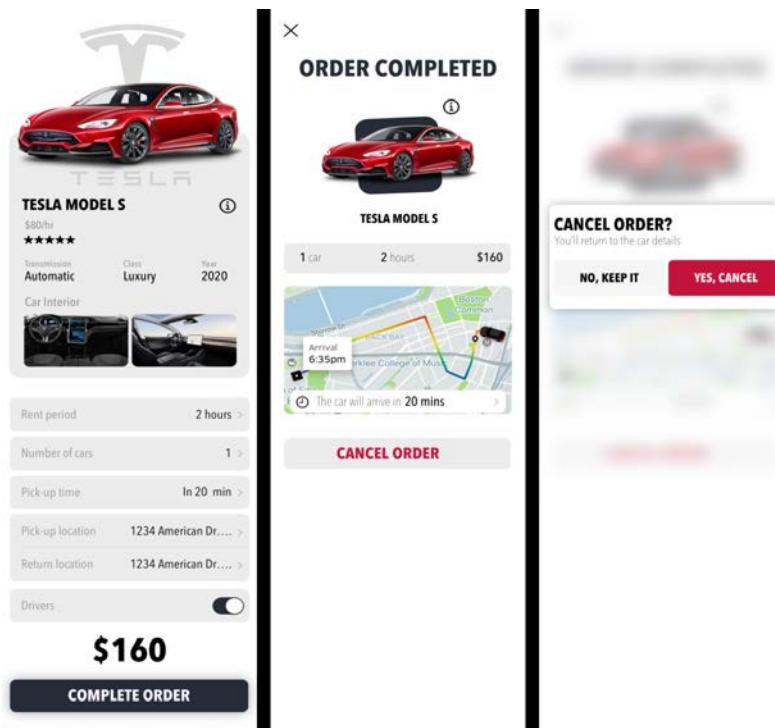


Figure 4.1

Our app does not contain a ton of views, but it does have a ton of features. Let's walk through this app's overall structure to better understand what app we will build and then build the app in SwiftUI. As stated in the last chapter, inside the folder for this chapter, you also have the design specs, and you can see everything from spacing to colors to sizing.

Understanding the structure

You might notice in this book that I like to plan how I will break up the design into smaller views. This is the best approach before I work on an app in code, especially SwiftUI. The reason I do this is that the code can get very long and overwhelming. I find it much easier to have smaller views and bring them into the main view. Doing this also helps me spot places where I can reuse code before I build it out. Let's take another look at the design, but look at the main view first:

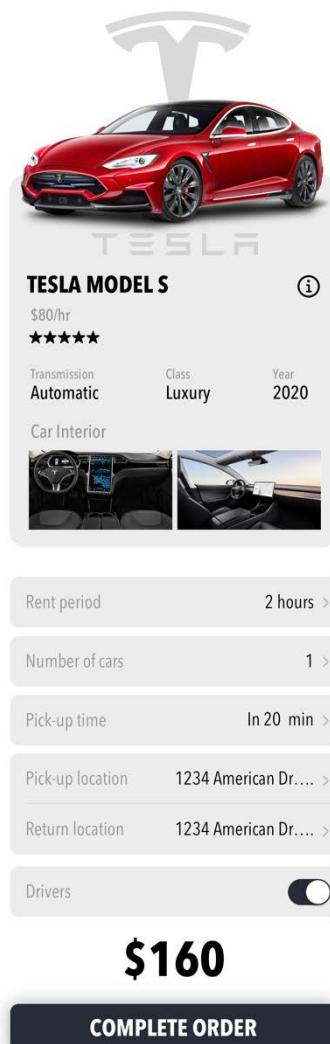


Figure 4.2

In this view, there are two main sections: the top portion of the form and the bottom part. Let's call them **Car Detail** and **Form View**:

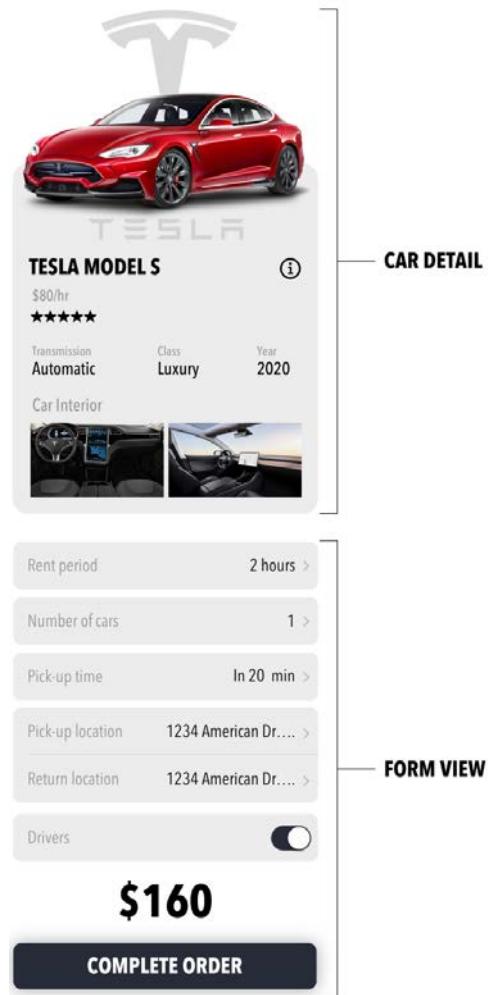


Figure 4.3

These two views are a great way to separate the two sections from each other. I know that I can make them all one view, but smaller views make it easier to work with; let's work on Car Detail first.

Car Detail

As I stated earlier, I break views down into smaller views. I did the same for Car Detail as well. I want to say that breaking up the views in this section was done more for cleaner code than for its reusability. This is how I broke up Car Detail:

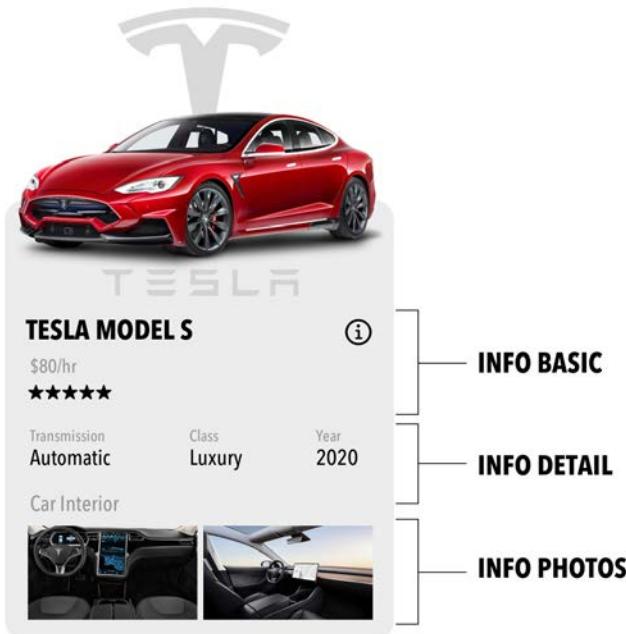


Figure 4.4

We'll create three smaller views, **Basic**, **Detail**, and **Photos**, and then add them to Car Detail. Let's get started on the Basic view first.

Creating Basic car info view

Now that we have an idea of what we will do in this chapter, let's get started:

1. Open the `CarInfoBasicView` file and update the previews with the following `previewLayout`:

```
CarInfoBasicView()  
    .previewLayout(.fixed(width: 400, height: 100))
```

We have changed our preview layout to be a fixed size. Adding a fixed size helps me focus on the size instead of being inside a giant phone.

2. Now, add the following code inside of the `body` variable by replacing `Text ("Car Info Basic View")`:

```
 VStack(alignment: .leading, spacing: 0) {
    HStack {
        Text ("TESLA MODEL S")
            .custom(font: .bold, size: 28)
        Spacer()
        Image(systemName: "info.circle")
            .font(.system(size: 28))
            .offset(y: -2)
    }

    VStack(alignment: .leading, spacing: 4) {
        Text ("$80/hr")
            .custom(font: .medium, size: 19)
            .foregroundColor(.baseDarkGray)

        HStack {
            ForEach(0..<5) { _ in
                Image(systemName: "star.fill")
                    .font(.system(size: 15))
            }
        }
    }
}
```

Now, let's break down the code we just added:

1. We use `VStack` as our main container with `alignment` set to `.leading` and `spacing` set to 0.
2. Inside `VStack`, we have two stacks. Firstly, inside `HStack`, we have a `Text` view and an SF symbol.
3. Finally, we have a `VStack` with a `Text` view and a `HStack`.

4. We are using an `HStack` that uses a `ForEach` to create the star rating.

Note

Since this is just a prototype, we are using `ForEach`, but for a real app, you would not use `ForEach`.

When you finish, you will see the following:



Figure 4.5

We are finished with `CarInfoBasicView`, and we will move on to the next view, `CarInfoDetailView`.

Creating our Car Info Detail view

We are now going to focus on the Info Detail view. In this view, broken down into small views, we have another simple layout:

1. Open `CarInfoDetailView` next and update the previews with the following by adding `previewLayout`:

```
CarInfoDetailView()
    .previewLayout(.fixed(width: 400, height: 100))
```

Again, we set `previewLayout`, with `width` set to 400 and `height` set to 100.

Next, we need to add three columns of text. We will add those variables first, then add them to the body once we are done.

2. Add the first column after the `body` variable's last curly brace:

```
var column1: some View {
    VStack(alignment: .leading) {
        Text("Transmission")
            .custom(font: .medium, size: 16)
            .foregroundColor(.baseDarkGray)
```

```
        Text ("Automatic")
            .custom(font: .medium, size: 22)
        }
    }

// Add column 2 here
```

We have our first column of text; let's add the second column.

3. Replace // Add column 2 with the following:

```
var column2: some View {
    VStack(alignment: .leading) {
        Text ("Class")
            .custom(font: .medium, size: 16)
            .foregroundColor(.baseDarkGray)

        Text ("Luxury")
            .custom(font: .medium, size: 22)
    }
}

// Add column 3 here
```

Our second column is done; let's add one more.

4. Next, replace // Add column 3 with the following:

```
var column3: some View {
    VStack(alignment: .leading) {
        Text ("Year")
            .custom(font: .medium, size: 16)
            .foregroundColor(.baseDarkGray)
        Text ("2020")
            .custom(font: .medium, size: 22)
    }
}
```

We are done adding columns; now let's get them into our body variable.

- Inside the body variable, replace Text ("Car Info Detail View") with the following:

```
HStack {
    column1
    Spacer()
    column2
    Spacer()
    column3
}.padding(.top, 15)
```

We are done with the Car Info Detail view. When you are finished, you will see the following in the preview:



Figure 4.6

We have completed the detail section, and now we can move on to the photos section.

Creating our Car Info Photos section

We are on the last section of this screen. In this section, we are going to display two pictures:

- Open `CarInfoPhotosView` and update the previews with `previewLayout`:

```
CarInfoPhotosView()
.previewLayout(.fixed(width: 400, height: 150))
```

- Now that we have the preview size set up, add the following code inside of the body variable by replacing `Text ("Car Info Photos View")`:

```
VStack(alignment: .leading, spacing: 4) { // Step 1
    Text("Car Interior") // Step 2
        .custom(font: .medium, size: 22)
        .foregroundColor(.baseDarkGray)
```

```
HStack { // Step 3
    Image("pic1")
        .frame(width: 170, height: 94)
        .cornerRadius(10)

    Image("pic2")
        .frame(width: 170, height: 94)
        .cornerRadius(10)
    }.frame(height: 94)
}.padding(.top, 10)
```

We are done adding code, but let's break down what we just added:

1. We are using `VStack` as our main container with a `.leading` alignment and a spacing of 4.
2. We add a `Text` view inside the `VStack` but outside of the following `HStack` so that it is separate from the two images.
3. We wrap our two image views inside `HStack`, which allows them to align horizontally together. If this were a production app, I would also make this a horizontal scroller.

This section added two photos, and you will notice that our images are not sizing correctly:



Figure 4.7

To fix this, you need to add the `resizable()` property above the `height` property. When you are done, you will have the following for your images:

```
Image("pic1")
    .resizable()
    ....
```

```
Image("pic2")
    .resizable()
    ...
```

After you add `resizable()`, you'll see that the images scale and fit the screen:



Figure 4.8

Our photo section is complete; let's put this all together inside `CarInfoView`.

Displaying our Car Info view

We are almost finished with our top section. Now that we have all of our children's views completed, we will bring them all together inside `CarInfoView`:

1. Open `CarInfoView` and update the previews with `previewLayout`:

```
CarInfoView()
    .previewLayout(.fixed(width: 400, height: 350))
```

2. Next, add the following inside the `body` variable by replacing `Text ("Car Info View")`:

```
 VStack(alignment: .leading, spacing: 5) {
    CarInfoBasicView()
    CarInfoDetailView()
    CarInfoPhotosView()
    Spacer()
}
.frame(height: 320)
.padding(.horizontal, 20)
```

When you are finished, you'll see the following:



Figure 4.9

We have completed `CarInfoView`, and now we are going to combine it inside our `CarDetailView`.

Wrapping up Car Detail

The top section of our main view is almost complete. We need to update our Car Detail section, and we will be finished:

1. Open `CarDetailView` and update the previews with `previewLayout`:

```
CarInfoPhotosView()
    .previewLayout(.fixed(width: 400, height: 650))
```

2. Now that we have our preview layout set up, let's update the body with the following:

```
ZStack {
    RoundedRectangle(cornerRadius: 20)
        .fill(Color.baseGray)
        .frame(height: 442)

    VStack(spacing: 10) {
        VStack {
            Image("tesla-logo").offset(y: 120)
            Image("tesla-s")
            Image("tesla-text-logo").offset(y: -10)
        }
    }
}
```

```
        }
        CarInfoView()
    }.offset(y: -155)
}
.padding(.horizontal, 12)
.offset(y: 100)
.frame(height: 250)
```

1. We are using a `ZStack` as our main container. Remember `ZStacks` gives us layers.
2. We are creating a `RoundRectangle` and using it as our background.
3. We are wrapping the logo, car and text logo into a `VStack`. I am using offsets to fine tune the positioning of the logo. We want to push the logo further down behind the car. I am tweaking the offset of the text logo so that it moves close to the car. Offsets are a great way to fine tune the design.
4. We add the `CarInfoView` here so that it appears below are car and logos just like the design.

When you are done, you'll see the following:

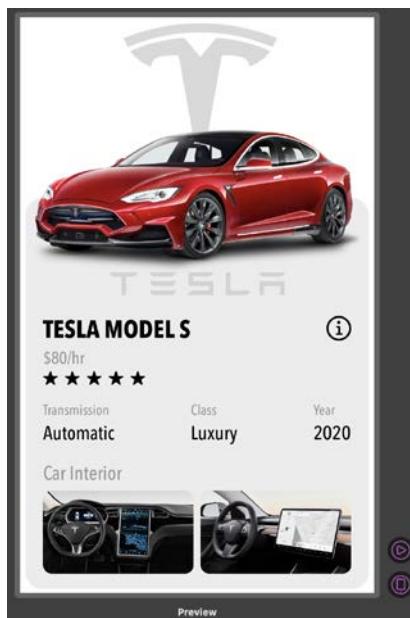


Figure 4.10

Our Car Detail is complete, and we can now move on to the form section.

The form view

To wrap up the main view, we need to work on the bottom section, which contains the form. You will learn how to customize a generic form. Since SwiftUI is new, we will not be able to customize our form using only SwiftUI. SwiftUI forms are built on top of `UITableView`, so a lot of our customizations will be made to the table view itself. Let's get started by opening `FormView`, and before we get started, you will notice this:

```
@ObservedObject var order = OrderViewModel()
```

You will see this in a few files throughout this chapter. I have added some basic defaults for this file in `OrderViewModel.swift` that we will use to get our form working and some other logic we need in this app. This file and the default values are not necessary. We are only using it for prototyping and nothing more. In *Chapter 5, Car Order Form – Data*, we will rewrite this code. For now, we are going to just focus on design in this chapter.

Creating the form sections

1. Inside `FormView`, right before the last curly brace, add a new line and add the following variable:

```
var rentalPeriod: some View {
    Section {
        Picker(selection: $order.prototypeAmt, label:
            Text("Rental period")) {
            ForEach(0 ..< order.prototypeArray.count,
                id: \.self) { value in
                Text("\(self.order.prototypeArray[value])")
                    .tag(value)
            }
        }
    }.listRowBackground(Color.baseGray)
}
```



```
// Add numberOfCars here
```

We created a section that, when we add it into a form, it gives us some cool features such as grouped cells, and when a picker is inside of a section, it goes to a detail list where you can pick the value and come back to the main form. All of this is free with SwiftUI, and we are going to take advantage of it.

2. Let's add our next section. Replace // Add numberOfCars here with the following code:

```
var numberOfCars : some View {
    Section {
        Picker(selection: $order.prototypeAmt, label:
            Text("Number of cars")) {
            ForEach(0 ..< order.prototypeArray.count,
                id: \.self) { value in

                Text("\(self.order.prototypeArray[value])")
                    .tag(value)
            }
        }
    }.listRowBackground(Color.baseGray)
}

// Add Pickup times
```

Our numberOfCars section, the same as the first section. Let's move on to the next form section.

3. Next, replace // Add Pickup times with the following code:

```
var pickupTime: some View {
    Section {
        Picker(selection: $order.prototypeAmt, label:
            Text("Pick-up time")) {
            ForEach(0 ..< order.prototypeArray.count,
                id: \.self) { value in

                Text("In \(self.order.prototypeArray
                    [value]) mins").tag(value)
            }
        }
    }.listRowBackground(Color.baseGray)
}

// Add location here
```

We have added our third picker in a row; let's keep going. We are going to add our next section, which is a bit longer.

4. Replace `// Add location here` with the following code:

```
var location: some View {
    Section {
        Picker(selection: $order.prototypeAmt, label:
            Text("Pick-up location")) {
            ForEach(0 ..< order.prototypeArray.count,
                id: \.self) { value in
                Text("\(self.order.prototypeArray
                    [value])").tag(value)
            }
        }

        Picker(selection: $order.prototypeAmt, label:
            Text("Return location")) {
            ForEach(0 ..< order.prototypeArray.count,
                id: \.self) { value in
                Text("\(self.order.prototypeArray
                    [value])").tag(value)
            }
        }
    }
    .listRowBackground(Color.baseGray)
}

// Add drivers here
```

We added a new section, but this section is different because we have two pickers in the same section—one for the pick-up location and the other for the return location. Because we put them together, they are part of the same form group as well. Let's add the next section.

5. Next, replace // Add drivers here with the following code:

```
var drivers: some View {
    Section {
        Toggle(isOn: $order.prototypeBoolean) {
            Text("Drivers")
        }.toggleStyle(SwitchToggleStyle(tint:
            .baseGreen))
    }.listRowBackground(Color.baseGray)
}

// Add orderComplete here
```

In this section, we are using a toggle instead of a picker, and that's it; we can move to our last section.

6. Finally, replace // Add orderComplete here with the following code:

```
var orderComplete: some View {
    Group {
        Section {
            HStack(alignment: .center) {
                Spacer()
                Text("$160")
                    .custom(font: .bold, size: 60)
                Spacer()
            }
        }
    }

    Section {
        Button(action: {
            self.order.isOrderCompleteVisible.
            toggle()
        }) {
            Text("COMPLETE ORDER")
        }
        .custom(font: .bold, size: 28)
        .frame(minWidth: 0, maxWidth: .infinity)
        .frame(height: 60)
    }
}
```

```
        .foregroundColor(.white)
        .background(Color.baseGreen)
        .cornerRadius(10)
    }
}
```

In our last variable, we have two sections wrapped inside a group. One of the sections just has a text field centered on the screen with the two spacer. In the last section, we have a button that is our **Complete Order** button. This button has a ton of straightforward modifiers.

Now that our sections are all created, we can now put them inside the form and display them in the preview. Inside the `body` variable, add the following:

```
 VStack {
    Form {
        rentalPeriod
        numberOfRows
        pickupTime
        location
        drivers
        orderComplete
    }
}.frame(minWidth: 0, maxWidth: .infinity)
```

`VStack` will be our main container, and inside this container, we have our `Form` view. Inside our form, we have added all of our variables and when you finish, you'll see the following in the preview:

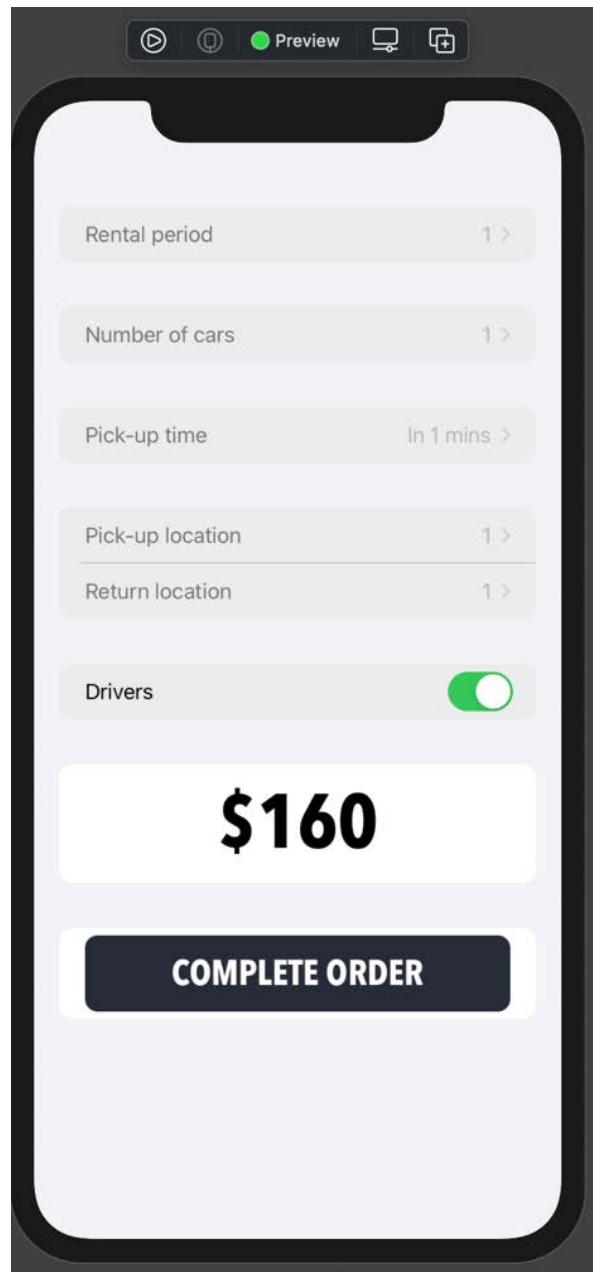


Figure 4.11

We now have a basic form, but let's look at how we can update our form to match the design. Let's look at the preview and the design side by side:

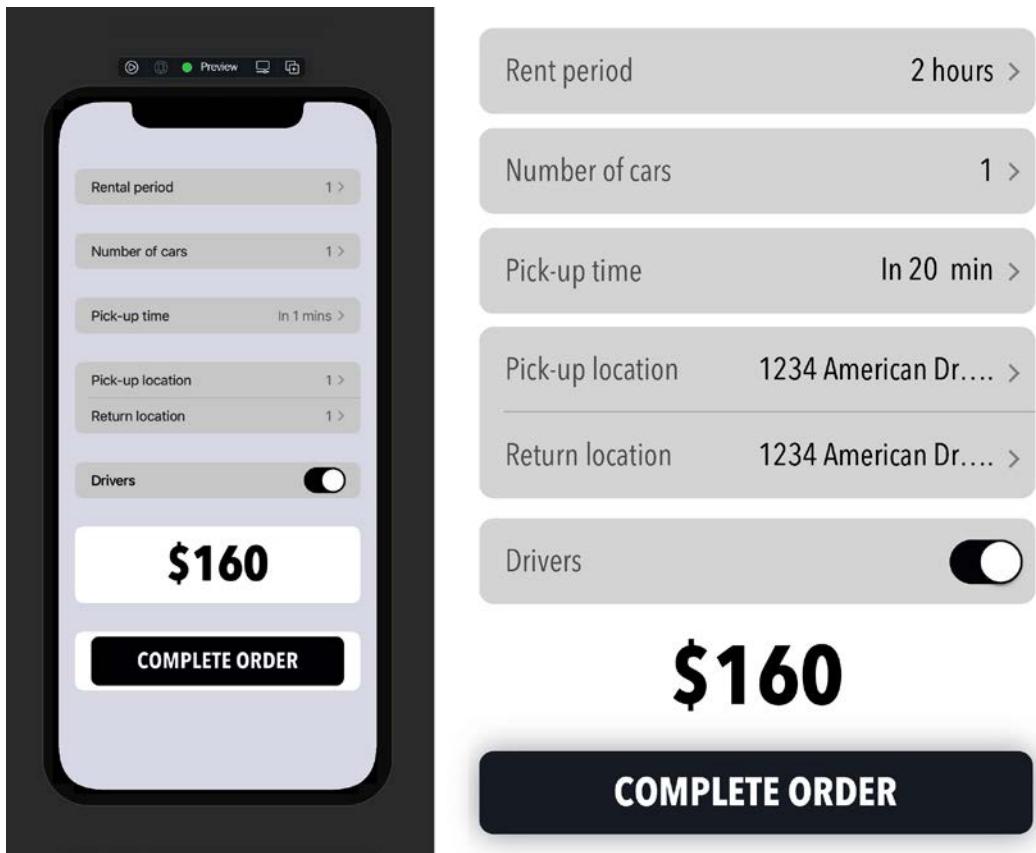


Figure 4.12

The overall design is missing a few things. The spacing between each section doesn't match, the complete order button and \$160 text view is inside a white box:

1. Let's fix the spacing by adding the following inside of the `init` function above the `body` variable:

```
UITableView.appearance().sectionHeaderHeight = 0
UITableView.appearance().sectionFooterHeight = 10
// Add next step here
```

When you are done, you will see that our spacing is now fixed:

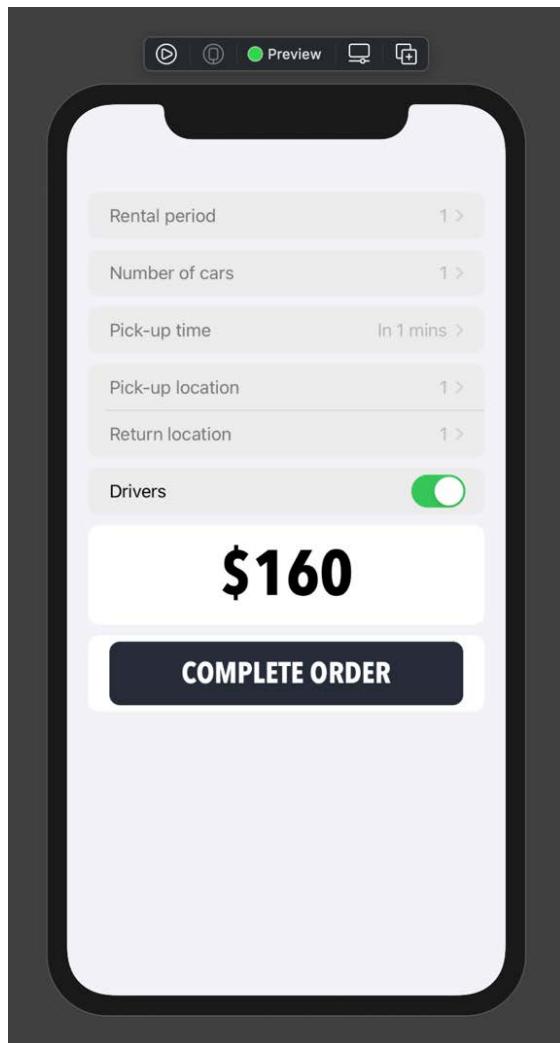


Figure 4.13

Let's fix the white background by getting rid of the background color.

2. Replace // Add next step with the following code:

```
UITableViewController.appearance().backgroundColor = .clear  
// Add next step here
```

You will now see the following in **Preview**:

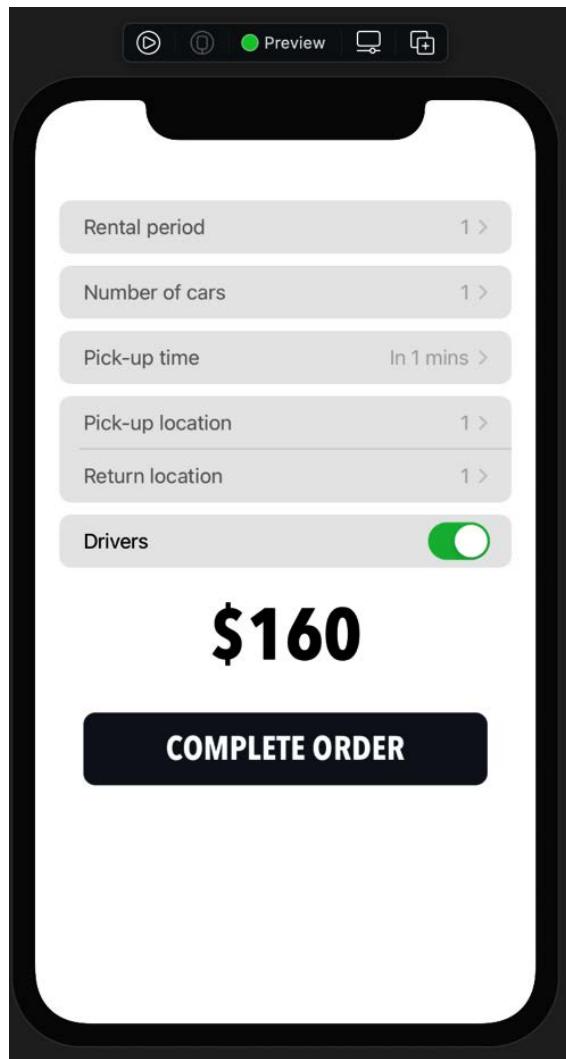


Figure 4.14

Finally, we have finished our form, and the design is coming along nicely. The next screen we will focus on now is the **Complete Order** view.

Complete Order design

We are moving to the **Complete Order** view; this view appears when you tap on the **Complete Order** button. This view will be a modal that lays on top of our main view:

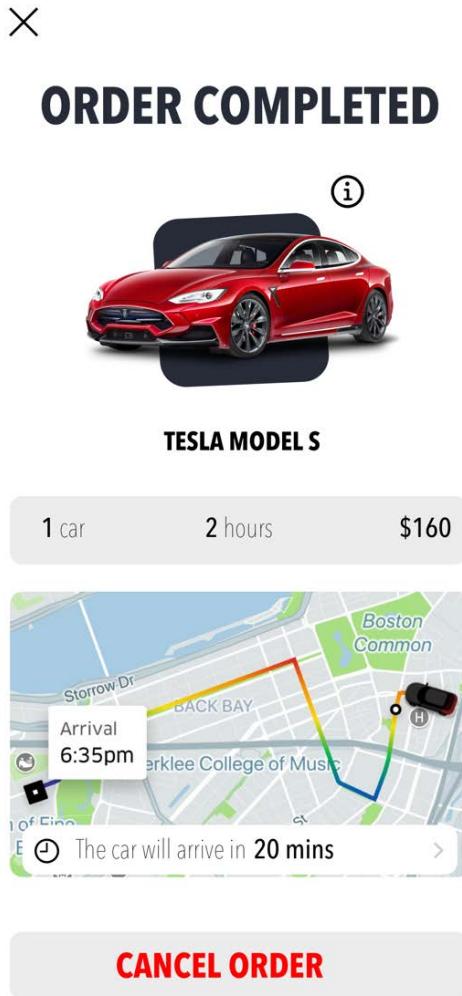


Figure 4.15

We'll break this view up into two sections, **Top Order** and **Bottom Order**:

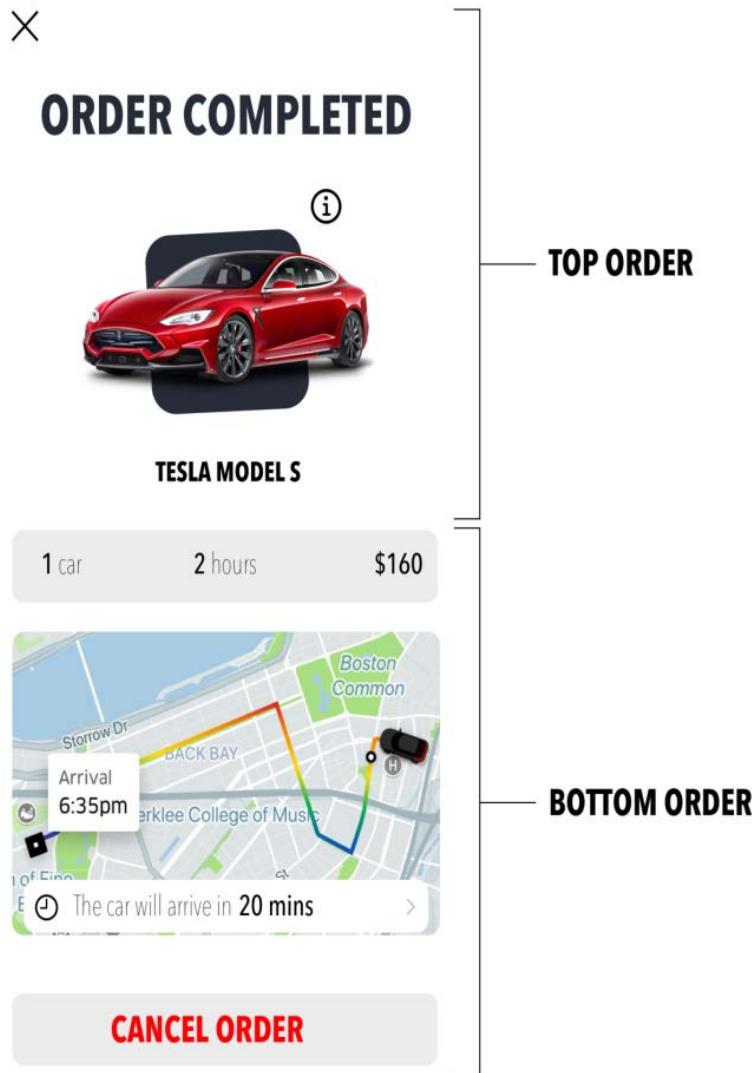


Figure 4.16

We could break this up into more; for example, if we were using MapKit here, we could create a view with all the code for displaying the map and the arrival time. Since we are just displaying an image, we are keeping it simple. We will do a few more screens together, and at the end of this chapter, you will get to try it out on your own.

Let's work on the order view, starting with the **Top Order** view.

The Top Order view

Now that we have a better idea of the direction, let's open the `TopOrderView` file; before the last curly brace, add the following variable:

```
var closeBtn: some View {
    Group {
        HStack { // Step 1
            Button(action: {
                self.order.isOrderCompleteVisible.toggle() }) {
                Image("close-btn")
            }
            Spacer()
        }
        .buttonStyle(PlainButtonStyle())
    }

    Text("ORDER COMPLETED")
        .custom(font: .bold, size: 42)
    }
}

// Add carInfo here
```

We are creating a variable for our close button and the ORDER COMPLETED text. Let's look at each step:

1. We add an `HStack` that has our close button. The `Spacer()` is used to push the close button to the left.
2. We have a text view that displays **Order Complete**.

Next, replace `// Add carInfo here` with the following code:

```
var carInfo: some View {
    VStack(spacing: 0) {
        HStack {
            Spacer()

            Image(systemName: "info.circle")
                .font(.system(size: 28))
        }
    }
}
```

```
        .offset(x: -30)
    }

ZStack {
    Image("car-bg-shape")
    Image("tesla-s")
    .resizable()
    .frame(width: 268, height: 125)
}

Text("TESLA MODEL S")
    .custom(font: .bold, size: 21)
    .padding(.top, 30)
}

}
```

We have a `VStack` container, and inside that container we use a `ZStack` container to stack the car and the square shape behind it. We can now add our newly created variables into the `body` variable. Replace `body` variable with the following code:

```
var body: some View {
    VStack {
        closeBtn
        carInfo
    }.padding(.horizontal, 10)
}
```

When you finish, you'll see the following:

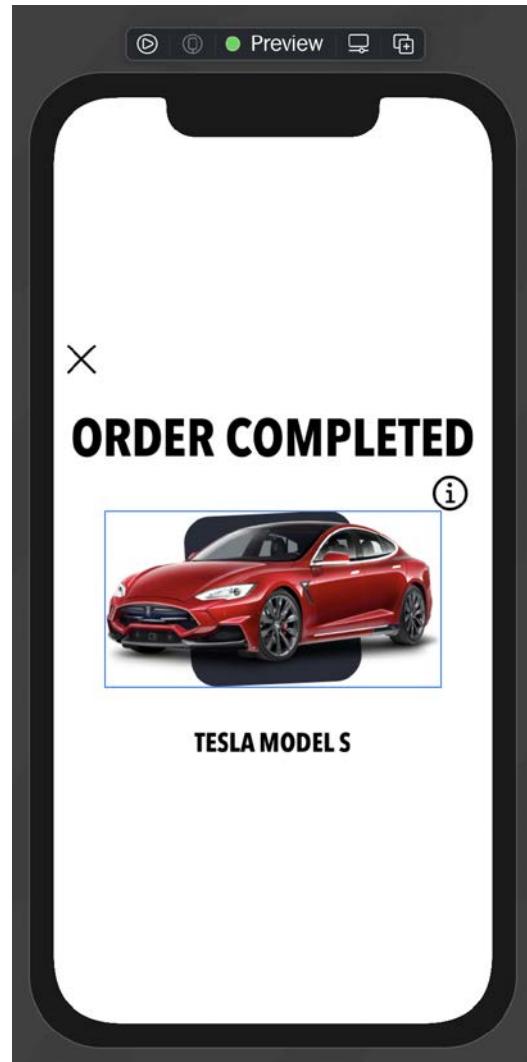


Figure 4.17

Now that `TopOrderView` is finished, we can move on to `BottomOrderView`.

Bottom Order view

The Bottom Order view has a summary of the order and when the car will arrive. Let's get started:

1. Open BottomOrderView, add the following variable above the last curly brace and a new line, and then add the following:

```
var info: some View {
    HStack {
        HStack(spacing: 4) {
            Text("1")
                .custom(font: .medium, size: 22)
            Text("car")
                .custom(font: .ultralight, size: 22)
        }

        Spacer()

        // Add next step here
    }
    .padding(.horizontal, 15)
    .frame(height: 55)
    .frame(minWidth: 0, maxWidth: .infinity)
    .background(Color.baseGray)
    .cornerRadius(10)
}

// Add map here
```

We are adding an order summary, and we are putting the information displayed at the top.

2. Replace // Add next step here with the following:

```
HStack(spacing: 4) {
    Text("2")
        .custom(font: .medium, size: 22)
    Text("hours")
        .custom(font: .ultralight, size: 22)
```

```

    }

Spacer()

HStack(spacing: 4) {
    Text("$160")
        .custom(font: .medium, size: 22)
}

```

The remaining parts of the order summary have been added.

3. Replace // Add map here with the following:

```

var map: some View {
    ZStack(alignment: Alignment(horizontal: .center,
        vertical: .bottom)) {
        Image("sample-map")
            .resizable()
            .scaledToFit()
            .padding(.bottom, 30)

        // Add next step here
    }
    .frame(maxWidth: 370)
}

// Add button here

```

The map variable will be what shows where the car is located and when it will arrive.

4. Replace //Add next step here with the following:

```

HStack {
    Image(systemName: "clock")
    HStack(spacing: 4) {
        Text("The car will arrive in")
            .custom(font: .ultralight, size: 22)
        Text("20 mins")
            .custom(font: .medium, size: 22)
    }
}

```

```
Spacer()
Image("disclosure-indicator")
}
.frame(height: 40)
.padding(.horizontal, 5)
.background(Color.white)
.cornerRadius(5)
.offset(y: -35)
.padding(.horizontal, 5)
```

We have just added the view that shows when the car will arrive and some text describing it.

5. Now, we can add our last variable, which is `button`. Replace `// Add button` here with the following code:

```
var button: some View {
    Button(action: {
        self.order.isCancelOrderVisible.toggle() }) {
        Text("CANCEL ORDER")
    }
    .frame(height: 55)
    .frame(minWidth: 0, maxWidth: .infinity)
    .background(Color.baseGray)
    .buttonStyle(PlainButtonStyle())
    .cornerRadius(10)
    .foregroundColor(.baseCardinal)
    .custom(font: .bold, size: 28)
}
```

We have now created our **Cancel Order** button, and we can move on to putting it all together.

6. Replace `body` variable with the following:

```
var body: some View {
    VStack {
        info
        map
        button
```

```

    Spacer()
    }.padding(.horizontal, 10)
}

```

When you finish, you'll see the following in the **Preview**:

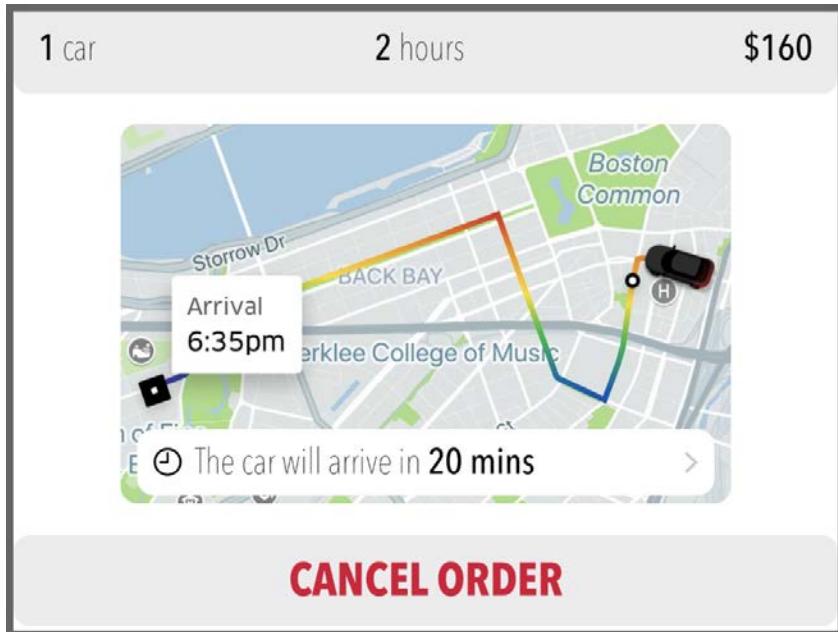


Figure 4.18

We are done with the Bottom Order view; let's move on to the Complete Order view next.

The Complete Order view

Now that both the Top Order and Bottom Order are complete, let's put them together to create the **Complete Order** view. Open `CompleteOrderView` and replace `Text ("Complete Order View")` with the following code:

```

VStack {
    TopOrderView().padding(.top, 20)
    BottomOrderView()
}
.background(Color.white)
.edgesIgnoringSafeArea(.all)

```

In the preceding code, we are combining `TopOrderView` and `BottomOrderView`. We added them to a `VStack`. When you are done adding this code, you'll see the following:

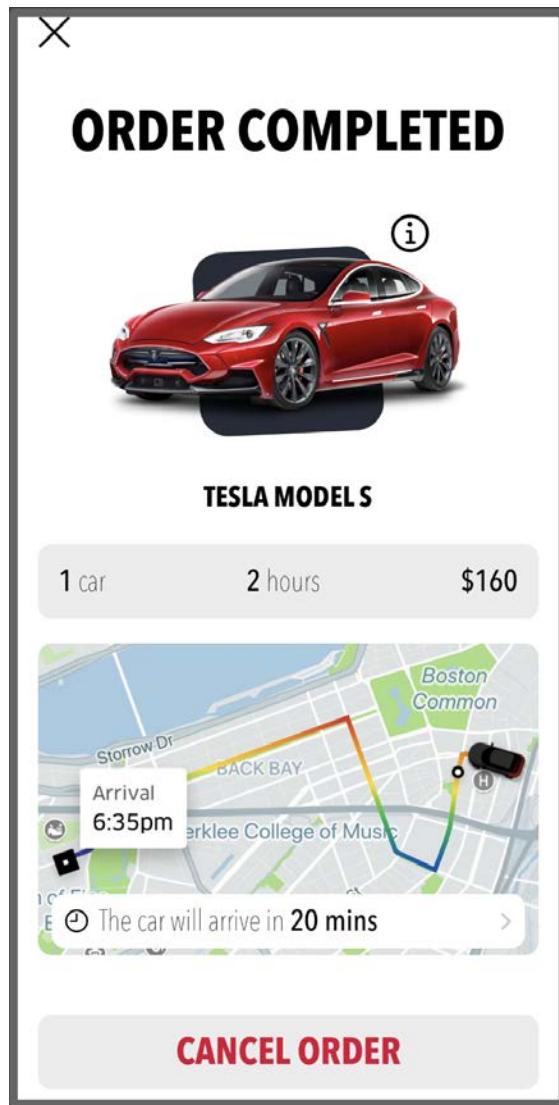


Figure 4.19

The Complete Order view is done, but we still need to finish the Content view. Let's do this next.

Combining our views

We have created `CarDetailView`, `FormView`, and `CompleteOrderView`, and we need to combine them all together. Open `ContentView` and replace `Text ("Content view")` with the following code:

```
ZStack {
    NavigationView {
        ScrollView(.vertical) {
            VStack(spacing: 0) {
                CarDetailView()
                    .frame(height: 600)
                FormView()
                    .environmentObject(order)
                    .frame(height: 450)
            }
            .padding(.top, 40)
        }
        .hideNavigationBar()
    }

    CompleteOrderView()
        .environmentObject(order)
        .opacity(order.isOrderCompleteVisible ? 1 : 0)
        .animation(.default)
}
```

We are using a `ZStack` as our main container. Using `NavigationView` will allow us to go to the list view for our forms, and wrapping them in `ScrollView` gives us scrolling. As you can see, breaking up all of our views makes our main views relatively clean.

Anytime you can separate your views, do so, but just remember you might not always be able to. For example, splitting up the form into multiple forms is a bit harder, especially with the design, but other designs might let you split it up.

We have one more view remaining, and it will be your challenge.

Cancel Order design challenge

When you tap on **CANCEL ORDER** in CompleteOrderView, you will see the following:

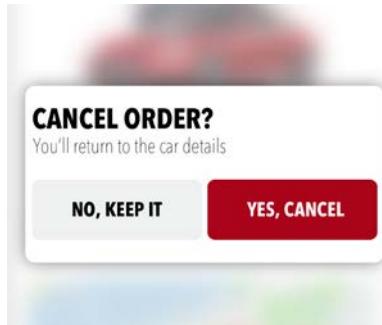


Figure 4.20

Before you start working on this view, we will create a blurred view together first. We are going to create our first `UIViewRepresentable`. Open `VibrancyBackground` and add the following:

```
struct VibrancyBackground: UIViewRepresentable {  
  
    var style: UIBlurEffect.Style = .light  
  
    func makeUIView(context: UIViewRepresentableContext<VibrancyBackground>) -> UIVisualEffectView {  
        return UIVisualEffectView(effect: UIBlurEffect(style: style))  
    }  
  
    func updateUIView(_ uiView: UIVisualEffectView, context: Context) {  
        uiView.effect = UIBlurEffect(style: style)  
    }  
}
```

`VibrancyBackground` will be used as a background, and it can be used as follows:

```
Rectangle()  
    .fill(Color.clear)  
    .background(VibrancyBackground())
```

The preceding code sets `Rectangle` to `.clear`, and then we attach `VibrancyBackground()` to `.background`. Now it is time for you to create the modal view design. You do not need to worry about making it appear visually, as we will cover that in the next chapter.

Here are the design challenges:

1. Create a `CancelOrderView` design.
2. Integrate `VibrancyBackground`.
3. Bonus: Create a button style for the two buttons.

We haven't covered the bonus challenge in this chapter, but we will, in *Chapter 6, Financial App – Design*. For now, do the best you can and see if you can figure it out. All the code for this challenge is available in the `Chapter 4 completed` folder. Remember, you have the design specs with all of the spacing, fonts, and so on.

Remember that our code may not be the same, so focus on the result. If yours matches the design, then you did fine. There are numerous ways to accomplish this task, so even if your code doesn't match mine, it does not mean you did it wrong.

Summary

In this chapter, we worked on our first iPhone app using SwiftUI. We learned how to take a basic form in SwiftUI and customize the design. We also learned how to break up our views into smaller views. Smaller views make it great for reuse, but they are also great for cleaner code. Finally, we created our first `UIViewRepresentable` to create a blurry effect on our modal.

In the next chapter, we will get data into our pickers, learn how to get data from our form, and send data to an API.

5

Car Order Form – Data

When Apple first announced SwiftUI, I spent the week working on designs. What was fascinating to me was having so much more flexibility to do what we want and the simplicity it took to do it. The iOS navigational hierarchy is pretty rigid, which makes it not tricky but quite troublesome to use. One thing I didn't expect to enjoy learning was `State` and `Binding`, along with `Combine`.

In my job, we use reactive programming, and honestly, I am not a fan. The learning curve is steep, but I feel people force certain reactive programming parts because they think reactive programming is cool. I enjoy taking a simple button and making it so that you do not have to create a delegate and all that comes with it. From that standpoint, I enjoy it, but I feel that the SwiftBond library, which is now Bond, does a great job of making it super simple to use.

I feel Apple has done the same thing with using `State` and `Binding`, and the more I work with them, along with `Combine`, the more I love it. We will see how we can leverage this in our app and use it so that we do not have to pass data around like how you would do it in a non-SwiftUI app. We worked on the design of our app in the last chapter, and now we are going to learn how to leverage `State` and `Binding` along with `Combine`.

In this chapter, we will be working with the following:

- Understanding State and Binding
- Combine basics
- @StateObject
- Networking with Combine

Technical requirements

The code files for this chapter can be found here: <https://github.com/PacktPublishing/SwiftUI-Projects/tree/master/Chapter05>.

Understanding State and Binding

State is a topic that I wanted to cover once you had some understanding of SwiftUI. Using State in SwiftUI simplifies our apps because we will now have one source of truth. When using State, persistent storage is created by SwiftUI for each of our views. Here is an example of a State property:

```
@State private var isDriverEnabled: Bool = false
```

In this book, you have seen something similar to this, but let's take the time to break it down thoroughly. By adding @State, we tell the system that the isDriverEnabled variable changes over time, and that views will depend on this value. Changes to @State-wrapped properties initiate a re-rendering of the view when the values are updated. Every change is dispersed to all of the views of the children.

If you have two views inside your main view, which also needs to react to a change, these views will not use the @State property. The topmost view always owns the State-wrapped properties. All of the child views of the topmost view would use the @Binding property instead. For best practice, @State properties should always be private.

When you need to pass State through to child views, you will use @Binding. Binding has read and write access to the value without any ownership. When using Binding, you do not need to worry about having default values because their values are passed in from the parent. To create a binding, you just need to pass a State property using the \$ prefix, which provides a reference of the property to the child view. Here is an example of where we create a binding with the child view by passing the State property over:

```
@State private var isDriveEnabled: Bool = false
var body: some View {
```

```
    ChildView(driverEnabled: $isDriveEnabled)  
}
```

Inside of the child view, you would just use the following:

```
@Binding driverEnabled: Bool
```

@Binding-wrapped properties give a reference to the app's `state`. When values change on `Binding`-wrapped properties, they trigger changes across the app to all the views dependent on that `State` property's values.

Now that we understand both `State` and `Binding`, we can see that ownership is one of the primary differences. Let's take the time to understand what the other differences are in the next section.

Difference between `@Binding` and `@State`

Every view marked with `State` has ownership and properties, and using `Binding` has read and write access but no ownership.

An excellent example of this would be when you are renting an apartment. You do not own the apartment; you just have an agreement (aka your renter's agreement) that says you live there. If something happens to the building or anything inside the apartment, the complex management will take ownership and fix those issues.

We will use `Binding` and `State` throughout the book, but `Combine` is a topic that is a whole book by itself. Even though we will not use `Combine` a ton in this book, we need to discuss the `Combine` basics. There is no way I can cover everything, so please take the time to look into more detailed information on this topic.

Combine 101 – discussing the basics

`Combine` is a framework introduced in iOS 13, and it brings a native approach to reactive programming. Before iOS 13, if you wanted a reactive framework, you used RxSwift, ReactiveCocoa, or others. You may be familiar with reactive programming either from Swift or another programming language. Reactive programming typically has a higher learning curve, but I think Apple has done an excellent job of simplifying the topic.

If you are new to reactive programming, you are probably asking why you should learn Combine. Using Combine helps you with synchronous and asynchronous tasks. For example, let's say you have a sign-up form. You typically have a username, two password fields, and a **Submit** button. Let's say that your usernames are unique, and you need to verify that any new usernames don't already exist. The username will need to check the server to verify, and with passwords, you want to check that both passwords match. While this is happening, you want to disable the **Submit** button until all the criteria are met. You can code these checks, but using a framework such as Combine makes your life easier. Let's see how, in the next section.

Three main ingredients

Combine is composed of three main ingredients:

- Publishers
- Subscribers
- Operators

Publishers transmit a sequence of values over time. The best way to think about publishers and signals is that their pattern is similar to a notification center. There are four kinds of messages that a publisher can transmit:

- **Subscription:** You cannot have a publisher without a subscriber. The connection between the publisher and subscriber would be the subscription.
- **Value:** The value can be any data type that you might want to be sent and received.
- **Error:** You can transmit an error when one occurs, and the subscriber can respond accordingly.
- **Completion:** This value is an optional value but transmits a signal when the stream has successfully ended, and no more data will be transmitted.

A publisher can be denoted as follows:

```
PublisherName<Output, Failure>
```

Subscribers declare a type that they can receive from a publisher. If your publisher is transmitting a string type, then your publisher must receive a string as well. There are two parts to a subscriber:

- **Input:** The data type it can receive
- **Failure:** The error type it can receive

A subscriber can be denoted as follows:

```
SubscriberName<Input, Failure>
```

A subscriber's three essential functions are receiving a subscription, obtaining a value, and receiving a completion or failure (error) from a publisher.

Operators are the middleman between the publisher and the subscriber. They convert the value into the correct type.

As I stated earlier, this book is more on SwiftUI and not the Combine framework. Understanding the basics is all you will need for this book, and if we cover anything outside of this, I will explain in detail what is going on. Before we move back to our project, let's examine one more topic, `ObservableObject`, which we will use in this chapter.

Networking with Combine

We now have some of the basics; let's create a networking layer that uses Combine. Most of this code structure is just a networking layer, but we will use `URLSession`. Let's get started.

Open the API file inside of the `View Model` folder and add the following:

```
struct API {
    enum Error: LocalizedError, Identifiable {
        var id: String { localizedDescription }

        case addressUnreachable(URL)
        case invalidResponse

        var errorDescription: String? {
            switch self {
                case .invalidResponse: return 'The server
                    responded with garbage.'
                case .addressUnreachable(let url): return
                    '\(url.absoluteString)'
            }
        }
    }
}
```

```
// Add next step here
}
```

Remember that when using subscribers, you can send errors, and here we are creating a custom error that we will use.

Next, we need to set up our EndPoint enum. The EndPoint enum will be where we create our URL and URL request. Replace // Add next step here with the following:

```
enum EndPoint {
    static let base = URL(string: 'https://regres.in/')!

    case post

    var url: URL {
        switch self {
        case .post:
            return EndPoint.base.appendingPathComponent
                ('api/tesla-order')
        }
    }

    static func request(with url: URL,
        and order:OrderViewModel) -> URLRequest {
        guard let encoded = try?
            JSONEncoder().encode(order) else {
            fatalError('Invalid')
        }

        var request = URLRequest(url: url)
        request.setValue('application/json',
            forHTTPHeaderField: 'Content-Type')
        request.httpMethod = 'POST'
        request.httpBody = encoded

        return request
    }
}
```

```
// Add next step here
```

Our endpoint is using a `post` method, and the URL we are using just sends back whatever we send, so we will send it our form data, and it will send it back to us. The `request` method sets up the encoding and passes it to our URL request.

If you've used `URLSession` pre-iOS 13, you might be familiar with `dataTask`, but with the release of iOS 13, `dataTaskPublisher` was added, which is part of Combine.

Replace `// Add next step here` with the following:

```
private let decoder = JSONDecoder() // Step 1
func post(with order: OrderViewModel) ->
    AnyPublisher<OrderViewModel, Error> { // Step 2
    URLSession.shared.dataTaskPublisher(for:
        EndPoint.request(with: EndPoint.post.url, and: order))
        // Step 3
        .map { $0.data } // Step 4
        .decode(type: OrderViewModel.self, decoder: decoder)
        // Step 5
        .mapError { error in // Step 6
            switch error {
            case is URLError:
                return Error.addressUnreachable
                    (EndPoint.post.url)
            default: return Error.invalidResponse
            }
        }
        .print() // Step 7
        .map { return $0 } // Step 8
        .eraseToAnyPublisher() // Step 9
}
```

The code we just added has a lot going on, so let's break down each step:

1. We are creating an instance of `JSONDecoder()`.
2. The `post` method takes an instance of `OrderViewModel` and returns a publisher, an output of `OrderViewModel`, and a failure of error (our custom error recreated earlier).

3. In this step, we are using `dataTaskPublisher` and passing in our URL endpoint and order.
4. Here, we use the `map` function to map the response JSON data. We then take that data and pass it down to the `decode` method.
5. We use the `decode` method next, and this maps our JSON data to our model object. We pass `JSONDecoder` into this method as well.
6. The `.mapError` method is used when we encounter an error.
7. `.print()` is great for debugging when you are having issues. It is handy to use, especially when you are not getting any data.
8. Next, we use the `.map` method again, which takes the object—in this case, `OrderViewModel`—and returns it to whoever calls this function.
9. Finally, `.eraseToAnyPublisher()` allows us to make the publisher an instance of `AnyPublisher`.

We will call this API a bit later, but for now, we are ready to make a *post* request to our API and get a response mapped to `OrderViewModel`. We haven't updated this file yet, so let's work on that next. Please note that you will have an error inside of the API class. The class expects `ViewOrderModel` to conform to `Codable`, which requires a few things to do in the next step.

Understanding observable objects

Earlier, we discussed `State` and `Binding`, and you use `State` and `Binding` when you are working inside of the view. When you move properties into a model object, the name changes slightly, but overall it is the same. `ObservableObject` is part of both the Foundation and Combine frameworks. `ObservableObject` is a custom class that keeps track of the state. Using `ObservableObject` allows you to create environment objects, which means you can have one source of truth. Instead of using `@State` and `@Binding`, you will create an item that uses `ObservableObject` using `@Published` instead of `@Binding`.

You might wonder when to use one over the other; a lot of this would depend on your architecture, but a general rule of thumb would be if you need to keep track of the state outside of the view, you should create `ObservableObject`. When you have a state that lives only in one view, then use `State` and `Binding`. Let's work on our first `ObservableObject`.

OrderViewModel

Inside of `OrderViewModel`, we have some necessary prototyping code that we can delete. Now, we need to add all of the State-driven code next. Add the following variables inside of `OrderViewModel`:

```
@Published var rentalAmount = 0
@Published var amountOfCars = 0
@Published var location = 0
@Published var returnLocation = 0
@Published var pickupTime = 0
@Published var specialDriver = true
@Published var isOrderCompleteVisible = false
@Published var isCancelOrderVisible = false
@Published var isModalVisible = false

let pickupTimes = Array(stride(from: 60, through: 480,
    by: 10))
let rentalPeriods = Array(1..<5)
let numberOfCars = Array(1..<4)
let locations = ['MIA Inter. Airport',
    'Ft. Lauderdale Inter. Airport',
    'Palm Beach Inter. Airport']

let returnLocations = ['MIA Inter. Airport',
    'Ft. Lauderdale Inter. Airport',
    'Palm Beach Inter. Airport']

private let api = API()
private var subscriptions = Set<AnyCancellable>()

// Add next step here
```

All of this code drives the data for our switch and pickers inside of our form. Now, this is all the code you need to drive that form part. We are now going to work on making our code work with Codable.

Conforming to Codable

Codable allows us to work with JSON, and we will make it conform to JSON both for sending and receiving data:

1. Inside of OrderViewModel, update class OrderViewModel: ObservableObject to class OrderViewModel: ObservableObject, Codable.
2. Then, add the following by replacing // Add Next step here:

```
init() {}

enum CodingKeys: String, CodingKey {
    case rentalAmount, amountOfCars, selectedDate,
        location, specialDriver, name, pickupTime
}

// Add Next step here
```

The enums we just added are CodingKey; we use this for encoding and decoding.

3. Let's move on to the next step by replacing // Add Next step here with the following:

```
required init(from decoder: Decoder) throws {
    let values = try decoder.container(keyedBy:
        CodingKeys.self)

    rentalAmount = try values.decode(Int.self, forKey:
        .rentalAmount)
    amountOfCars = try values.decode(Int.self, forKey:
        .amountOfCars)
    location = try values.decode(Int.self, forKey:
        .location)
    pickupTime = try values.decode(Int.self, forKey:
        .pickupTime)
    specialDriver = try values.decode(Bool.self, forKey:
        .specialDriver)
}

// Add Next step here
```

We just created our decoder and next we will add our encode method:

```
func encode(to encoder: Encoder) throws {
    var container = encoder.container(keyedBy:
        CodingKeys.self)

    try container.encode(rentalAmount, forKey: .rentalAmount)
    try container.encode(amountOfCars, forKey: .amountOfCars)
    try container.encode(location, forKey: .location)
    try container.encode(pickupTime, forKey: .pickupTime)
    try container.encode(specialDriver, forKey:
        .specialDriver)
}

// Add Next step here
```

These last two methods are basics for working with models and JSON.

Finally, we need to create our `sendOrder()` method for sending our form data. Replace `// Add Next step here` with the following method:

```
func sendOrder() {
    api.post(with: self)
        .receive(on: DispatchQueue.main)
        .sink(receiveCompletion: { response in
            print(response)
            print('=====')
        }, receiveValue: { value in
            print('Received response from Combine publisher')
            print(value)
        }).store(in: &subscriptions)
}
```

Finally, we have our `sendOrder()` method to pass `OrderViewModel` into the `post` method. We are using more Combine methods here, and the `receive` method makes sure that this call is on the main thread. We are using the `.sink` way, which handles receiving events from the publisher using Combine.

Now that our data is set up, we can update our UI to use the newly created data.

TeslaOrderFormApp

In SwiftUI, we can set up an environment object that allows us to have one source of truth. In our case, we will use `OrderViewModel` as our environment object.

Open `TeslaOrderFormApp` and you'll see the following on the line above the `body` variable:

```
@StateObject private var order = OrderViewModel()
```

You will also see the following attached to `ContentView()`:

```
ContentView().environmentObject(order)
```

The `@StateObject` property wrapper is used when you want your state management to create a reference type inside one of your views, and you want to make sure it stays alive in that view and others you share it with. We now have an environment object set up. We already have our object set up in each of our views, but we need to update them to use the new properties we just added.

Updating FormView

Inside `FormView`, we are using prototype variables, and now that we have the actual variables set up, we need to update our pickers to use the correct data:

1. Update `rentalPeriod` to use `rentalAmount` and `rentalPeriods`:

```
Picker(selection: $order.rentalAmount, label:
    Text('Rental period')) {
    ForEach(0 ..< order.rentalPeriods.count, id: \.self)
    {
        Text('\(self.order.rentalPeriods[$0])').
        tag(value)
    }
}
```

Note

Inside `ForEach`, I added `id`. If you do not add `id`, you will get a warning message in the console.

2. In the `numberOfCars` view, use `amountOfCars` and `numberOfCars`:

```
Picker(selection: $order.amountOfCars, label:
Text('Number of cars')) {
    ForEach(0 ..< order.numberOfCars.count, id: \.self) {
        Text('\'\($self.order.numberOfCars[$0])\').tag(value)
    }
}
```

3. Let's move to the `pickupTime` view and use `pickupTime` and `pickupTimes`:

```
Picker(selection: $order.pickupTime, label: Text('Pick-up
time')) {
    ForEach(0 ..< order.pickupTimes.count, id: \.self) {
        Text('In \'\($self.order.pickupTimes[$0])
mins\').tag(value)
    }
}
```

4. Move to the `location` view and use `location` and `locations`:

```
Picker(selection: $order.location, label: Text('Pick-up
location')) {
    ForEach(0 ..< order.locations.count, id: \.self) {
        Text('\'\($self.order.locations[$0])\').tag(value)
    }
}

Picker(selection: $order.location, label: Text('Return
location')) {
    ForEach(0 ..< order.locations.count, id: \.self) {
        Text('\'\($self.order.locations[$0])\').tag(value)
    }
}
```

5. Update the drivers `Toggle` to use `specialDriver`:

```
Toggle(isOn: $order.specialDriver) {
    Text('Drivers')
}
```

We're done updating our app; you can now build and run it. Feel free to add even more to this project. If you do, make sure you share it with me on Twitter (@thedevme).

Summary

In this chapter, we have looked at `State` and `Binding` and how they work inside our views. We also looked at `Published` and how it works inside of `ObservableObject`. We looked at some `Combine` basics and how we can do networking with `Combine`. Finally, we created an environment object for our project to have one source of truth.

In the next chapter, we will build a financial app, and the app will be driven with Core Data. We will set up the app prototype in one chapter and learn about SwiftUI and Core Data together.

6

Financial

App – Design

SwiftUI is a fantastic new way to build apps. SwiftUI is excellent on the iPhone and iPad but still has some way to go on the Apple Watch. Building apps for the iPhone and iPad using SwiftUI is most exciting to me. When I think of SwiftUI, I think of a blank canvas, and I can do whatever I want to design beautiful apps. Yes, some things are much harder to do in SwiftUI, but I find if you take the approach of *I want to take a storyboard app and move it to SwiftUI*, you might struggle a bit. The best approach with SwiftUI is to use all of the things that make SwiftUI great and build from there. Yes, SwiftUI has its flaws, and most things do, especially when they are new, but the more you play with SwiftUI, the more you will see that it is a great tool to use.

In this chapter, we will be working on the following:

- Using SwiftUI to create graphics
- Custom button styles
- Custom forms

Technical requirements

The code files for this chapter can be found here: <https://github.com/PacktPublishing/SwiftUI-Projects/tree/master/Chapter06>.

Understanding our App design

In this chapter, we are going to be building a financial app. In the next chapter, we will work on the data side of this app. Let's take a look at the app design we will be working on in this chapter:

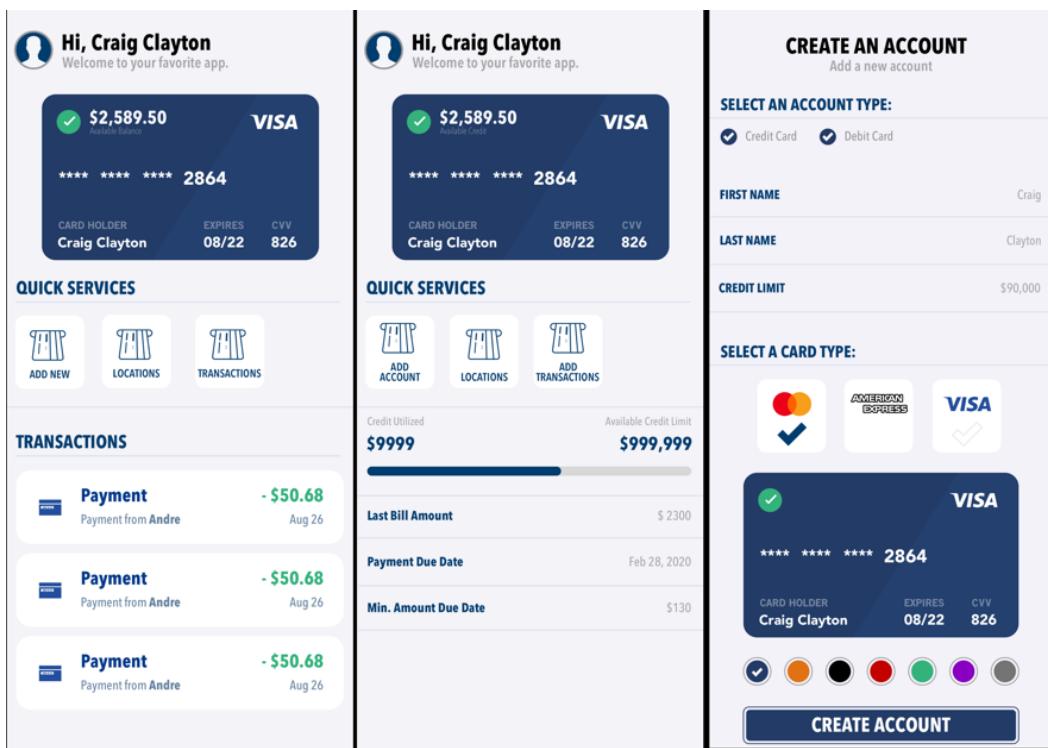


Figure 6.1

In this design, we have three main screens that we are going to design. We have an account list screen, an account home screen, and a create account screen. The account home screen will display slightly different data depending on the account type. When the app launches, we will check to see if we have any accounts, and if not, we will show the create account screen instead of the list screen. We will worry about this logic later; we will get started by designing the home screen.

Understanding the home view logic

Let's look at our first screen now, the home view. Let's take a look at how we will break down the home view into smaller views:

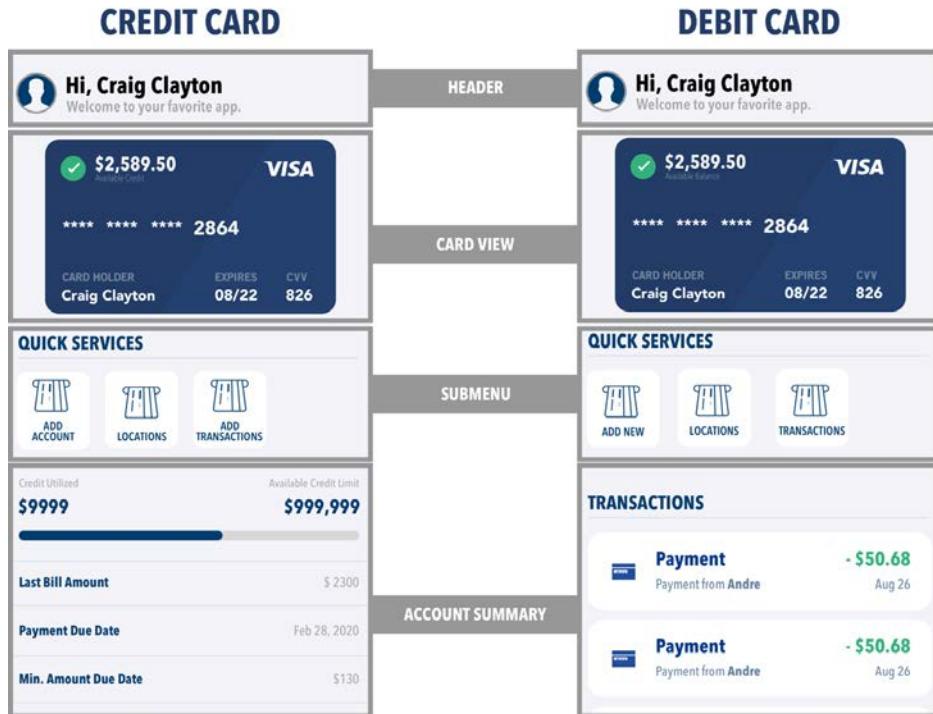


Figure 6.2

If you look at the preceding screen, you will notice that we have a header at the top, followed by our card view, representing the account we selected. Next, we have our submenu, which contains quick links to other services. Finally, we have basic account information, and this information is based on whether it's a credit card or debit card. Let's get started by opening the starter project.

Home header

You can find the starter project for this chapter in the Chapter06 folder called `starter`. To get started, open `HomeHeaderView`. Update the variable previews inside of `HomeHeaderView_Previews` with the following:

```
HomeHeaderView()
    .previewLayout(.fixed(width: 600, height: 80))
```

Now, inside of this file, add the following inside of the body variable:

```
HStack(alignment: .center) { // Step 1
    VStack(alignment: .leading, spacing: -8) { // Step 2
        Text("Hi, Craig Clayton") // Step 3
            .customFont(.custom(.bold, 24))
        Text("Welcome to your favorite app.")
            .customFont(.custom(.demibold, 16))
    }.foregroundColor(.basePrussianBlue)
    Spacer() // Step 4
    Image("avatar") // Step 5
}.padding(.horizontal, 20)
```

The preceding code block is explained as follows

1. The main container is `HStack`, which has a `center` alignment.
2. `VStack` is our `Text` view container with `foregroundColor` set to `.basePrussianBlue`.
3. Inside of `VStack`, there are two `Text` views with `custom` fonts.
4. `Spacer()` forces our `VStack` to be floated left and our image to be floated right.
5. Finally, inside of our `HStack` container, we have an `Image` view.

When creating components first, I find it easier to size the component to be longer or taller than the original design. Creating the component first lets me know that it will resize to any device. Click **Resume** if you need to, and in **Preview** you should see the following:

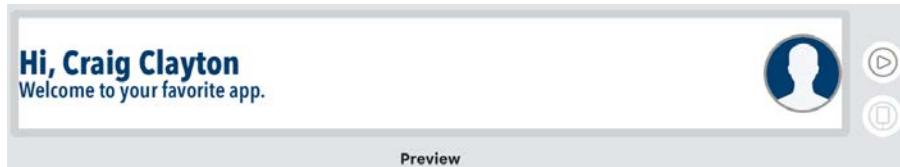


Figure 6.3

Our header is now complete; let's move on to the next component.

Creating our card view

Let's create our CardView first, which looks like the following screenshot:



Figure 6.4

We are going to first create the card background. Open CardView and update the variable previews inside of `CardView_Previews` with the following:

```
CardView()
    .previewLayout(.fixed(width: 300, height: 180))
```

Let's get to work on the design now.

Card background

We are going to set up our card background to be dynamic. We will allow users to change the color of their background when they create an account. Let's add the following to `CardView`:

```
ZStack { // Step 1
    HStack(spacing: -116) { // Step 2
        Image("left-card") // Step 3
            .renderingMode(.template)
            .foregroundColor(.black)
        Image("right-card")
            .renderingMode(.template)
            .foregroundColor(.black)
            .opacity(0.94)
    }
    // Add next step here
}
```

Let's break down this code:

1. We are using `ZStack` as our container. We want to have layers, and our background will be the layer that will appear on the bottom.
2. We use `HStack` to hold our background images. The spacing is used to bring the images together to create our card.
3. Here, we have two images left and right card. Each card has its `.renderingMode` set to `.template`, which allows us to change our images' colors. Finally, we set `.opacity` to `.94` to give the right side a lighter color.

Now that we have our background set, let's move on to add the card contents. We have three sections we need to add. Let's add the first one now.

Card balance and logo

Our card displays the balance and a logo, either American Express, Visa, or MasterCard. Let's create that now by adding the following, by replacing `// Add next step here` in the previous code block with the following:

```
 VStack { // Step 1
    HStack { // Step 2
        Image("checkmark") // Step 3
        VStack(alignment: .leading, spacing: -6) {
            Text("$99,999").customFont(.custom(.bold, 20))
            Text("Available Balance").
                customFont(.custom(.ultralight, 10))
        }
        Spacer()
        Image("visa-logo")
    }.padding(.horizontal, 20)
    // Add next step here
}
.frame(width: 280, height: 160)
.padding(10)
.foregroundColor(.baseWhite)
```

Let's go over the code we added:

1. We first use `VStack` as our container to hold all of the card details. We add a `.frame` here, so our contents don't go outside of the card. `.padding` is used to give us a little padding around the entire card. Finally, we set `.foregroundColor` to `.baseWhite`, which sets all text to this color.
2. We add the first of three sections into `HStack`.
3. Inside of `HStack`, we have the checkmark `Image`, a `VStack` that displays "Available Balance". All of this is being floated left because of the `Spacer()` added after everything. Finally, we have an image that displays our card logo and is floated right.

Our first section is complete; let's move on to add our card number.

Card number

We now need to display the card number next, replace `// Add next step here` with the following:

```
HStack { // Step 1
    HStack { // Step 2
        Text("****")
            .customFont(.custom(.black, 17))
        Text("****").customFont(.custom(.black, 17))
        Text("****").customFont(.custom(.black, 17))
        Text("9999").customFont(.custom(.black, 20))
    }
    Spacer() // Step 3
}.padding(.top, 15)
.padding(.horizontal, 20)

Spacer() // Step 4
// Add next step here
```

Let's break down the code now:

1. We use `HStack` as our container with `.padding` of 15 added to the `.top`.
2. Inside of `HStack`, we have another `HStack` that holds our `Text` views. Each `Text` view uses a custom font.
3. We use a `Spacer()` that floats our `HStack` container to the right.
4. Here, we use another `Spacer()` after the card number and card balance information and push it to the top.

Finally, we just need to add the rest of the card information.

Card information

In this section, we need to add the cardholder name, expiration, and CVV number. Replace `// Add next step here` with the following:

```
HStack { // Step 1
    VStack(alignment: .leading) { // Step 2
        Text("CARD HOLDER")
            .customFont(.custom(.bold, 11))
            .foregroundColor(Color.baseRockBlue)
        Text("Craig Clayton").customFont(.custom(.black, 16))
    }
    Spacer() // Step 3
    VStack(alignment: .leading) { // Step 4
        Text("EXPIRES")
            .customFont(.custom(.bold, 11))
            .foregroundColor(Color.baseRockBlue)
        Text("08/22").customFont(.custom(.black, 16))
    }
    VStack(alignment: .leading) { // Step 5
        Text("CVV")
            .customFont(.custom(.bold, 11))
            .foregroundColor(Color.baseRockBlue)
        Text("999").customFont(.custom(.black, 16))
    }
}
```

Now that our card is complete, let's go over what we just added:

1. We use another `HStack` as our container.
2. Here, we use a `VStack` to add the cardholder and the name of the cardholder.
3. The `Spacer()` here floats the cardholder information to the left and the rest of the card information to the right.
4. Inside of this `VStack`, we have two `Text` views that display the card expiration date. The `VStack` also sets `.alignment` to `.leading`.
5. Finally, we have another `VStack`, which holds the CVV information.

We are done with the card design. If you haven't already, hit the **Resume** button and you will see the following in **Preview**:

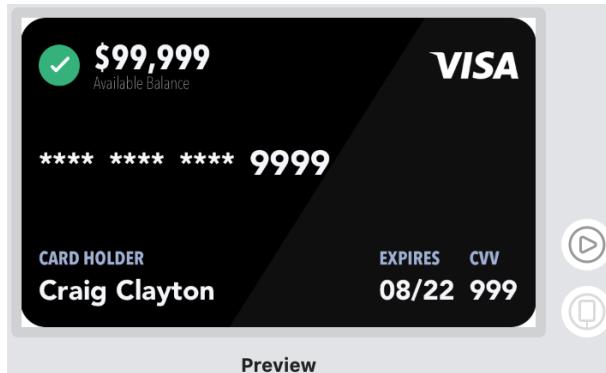


Figure 6.5

We have our card view created. Next, we need to make the account summary based on a credit card or a debit card. Let's move on to this section next.

If you want to check your code with my code, you can find the completed code for Step 1 inside the `Chapter06/steps` files called `step 1 - complete`.

Home submenu view (challenge 1)

On the home screen, we have some buttons that will take users quickly to a different part of the app. Your challenge will be to try and create the home submenu view yourself:

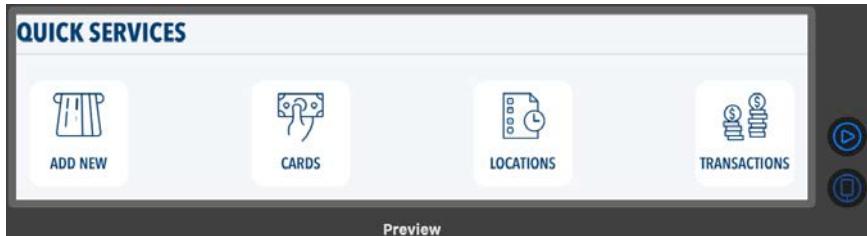


Figure 6.6

If you get stuck, you can see my `HomeSubMenuView` inside of `challenge 1` inside of the `challenge` folder to guide you. Remember that these are all buttons, and they do not need any actions set at this time. Set your preview layout to `600 x 140`. The custom blue color being used is `.basePrussianBlue`. The images used in this section are named the following: **Add New** (icon-send), **Cards** (icon-receive), **Locations** (icon-invoices), and **Transactions** (icon-bills).

Creating an account summary

In this section, we are going to create the account summary. Let's take a look at the account summary:

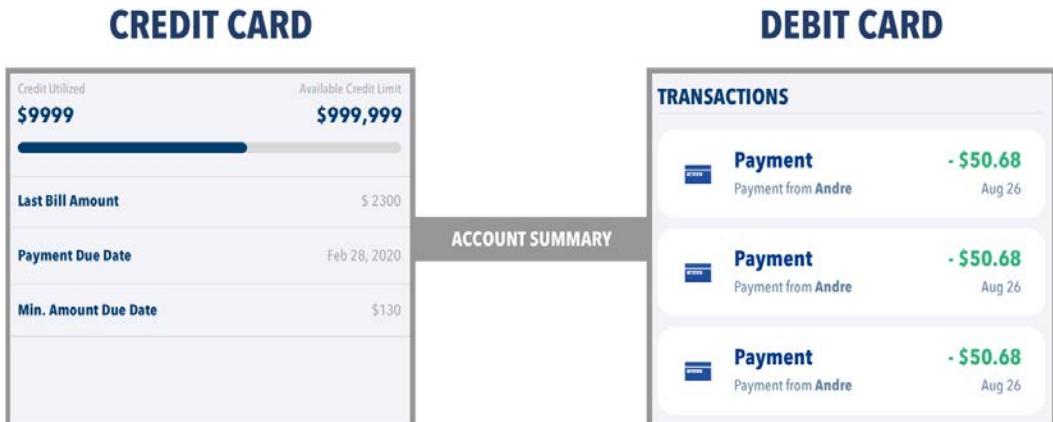


Figure 6.7

The account summary section has two different views depending on the card type. Let's get started by creating a new SwiftUI file inside of the Home folder named AccountSummaryView. Next, update the preview layout size to be 600 x 310. When you are finished, you will have the following:

```
static var previews: some View {
    AccountSummaryView()
        .previewLayout(.fixed(width: 600, height: 310))
}
```

Next, above the body variable, add the following:

```
private var type = "creditcard"
```

We will use this temporary variable for controlling which account summary is shown. In AccountSummaryView, we have a lot of code, so we will break each view into smaller chunks using an extension. You could also just create new views, but we will keep all the code in the same file – either way would work.

After the AccountSummaryView_Previews struct, add the following:

```
private extension AccountSummaryView {
    var creditcard: some View {
        // Add credit code here
    }
}
```

Account summary view – credit card

We are set up now; let's work on the account summary credit card view's logic. Inside of the body variable, add the following:

```
vStack {
    if type == "creditcard" {
        creditcard
    } else {
        Text("Debit code here")
    }
}.padding(.horizontal, 10)
```

Next, inside of the `creditcard` variable, replace `// Add credit code here` with the following:

```
 VStack { // Step 1
    VStack { // Step 2
        VStack(spacing: 0) { // Step 3
            HStack { // Step 4
                Text("ACCOUNT")
                    .customFont(.custom(.bold, 20))
                    .foregroundColor(.basePrussianBlue)
                Spacer()
                Text("*****  ****  ****  2864")
                    .customFont(.custom(.medium, 15))
                    .foregroundColor(.gray)
            }
        }
        Divider() // Step 5
    }
    // Add next step here
}
.padding(.horizontal, 10)

// Add Divider here
}
.padding(.top, 10)
.background(Color.baseWhite)
```

Let's break down the code we just added:

1. Our main container is a `VStack` container that has `.padding` of 10 and `.background` is set to `.baseWhite`.
2. Inside the main container, we have another `VStack` container, which is the container we use to hold the account information, the credit utilized, and the bar chart. The horizontal `.padding` is used to match the `List` padding below it.
3. We have another `VStack` container, which has a spacing of 0.

4. Next, we have an `HStack`, which holds the current account number.
5. We add a `Divider()` to the `Vstack` container, which floats our account information to the top.

We now have our **Account Number** displayed. Next, we need to show the current credit used versus the credit remaining. Replace `// Add next step here` with the following code:

```
HStack { // Step 1
    VStack(alignment: .leading, spacing: 0) { // Step 2
        Text("Credit Utilized")
            .customFont(.custom(.medium, 12))
            .foregroundColor(.gray)
        Text("$9999")
            .customFont(.custom(.bold, 20))
            .foregroundColor(.basePrussianBlue)
    }
    Spacer() // Step 3
    VStack(alignment: .trailing, spacing: 0) { // Step 4
        Text("Available Credit Limit")
            .customFont(.custom(.medium, 12))
            .foregroundColor(.gray)
    }
    Text("$999,999")
        .customFont(.custom(.bold, 20))
        .foregroundColor(.basePrussianBlue)
}
.padding(.top, 10)
// Add next step here
```

You will now see our credit text. Let's quickly look over it:

1. We have an `HStack` with `.padding` set to `.top`.
2. Inside of the `HStack`, we have a `VStack` container, which contains our two `Text` views.
3. We use `Spacer()` to float our contents right and left.
4. Finally, in our last `Vstack` container, we are displaying two more `Text` views.

All that's remaining is displaying the bar chart. Replace `// Add next step here` with the following:

```
ZStack(alignment: .leading) {
    Rectangle()
        .fill(Color.baseMediumGray)
        .cornerRadius(4.5)
        .frame(minWidth: 0, maxWidth: .infinity)
        .frame(height: 10)

    Rectangle()
        .fill(Color.basePrussianBlue)
        .cornerRadius(4.5)
        .frame(width:120, height: 10)
}
.padding(.bottom)
```

In the preceding code, we are using a `ZStack` to stack two `Rectangle()` on top of each other. We also add some padding to the bottom of our `ZStack`.

We have to display a list of three items and their values. Instead of doing this by hand, we are going to use a `List`. `List` will make it easy to add more items if need be or make it dynamic. You are going to replace `// Add divider here` with the following code:

```
Divider()

List {
    HStack {
        Text("Last Bill Amount")
            .customFont(.custom(.bold, 14))
            .foregroundColor(.basePrussianBlue)
```

```
Spacer()
Text("$ 2300")
    .customFont(.custom(.medium, 14))
    .foregroundColor(.gray)
}.listRowBackground(Color.baseWhite)

HStack {
    Text("Payment Due Date")
        .customFont(.custom(.bold, 14))
        .foregroundColor(.basePrussianBlue)
    Spacer()
    Text("Feb 28, 2020")
        .customFont(.custom(.medium, 14))
        .foregroundColor(.gray)
}.listRowBackground(Color.baseWhite)

HStack {
    Text("Min. Amount Due Date")
        .customFont(.custom(.bold, 14))
        .foregroundColor(.basePrussianBlue)
    Spacer()
    Text("$ 130")
        .customFont(.custom(.medium, 14))
        .foregroundColor(.gray)
}.listRowBackground(Color.baseWhite)
}

.frame(height: 150)
```

There is a lot of code, but we are repeating the information. First, we added a divider. Next, we have a List that just displays some data with values. Later, when we add the data, we can refactor this dynamically and reduce the amount of code.

When you are finished, you will see the following in **Preview**:

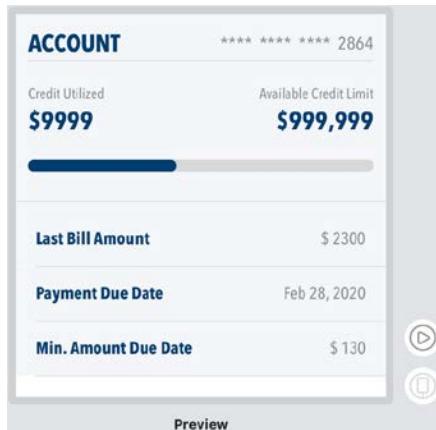


Figure 6.8

Now that we have the account summary done for a credit card, let's work on the account summary for a debit card next.

Account summary view – debit card

The next design we need to build is the account summary for the debit card. Let's take a look at what we need to design next:

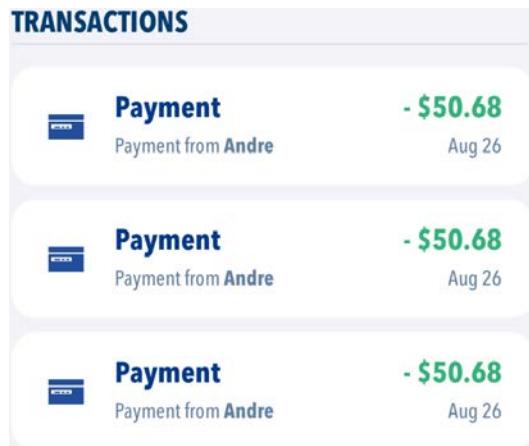


Figure 6.9

In this layout, we need to display a header and the last three transactions that the card made. We can create an item that we can reuse for each transaction.

Transaction item view (challenge 2)

Take a minute and see if you can create this item on your own:



Figure 6.10

Now that `TransactionItemView` is complete, we can go back to `AccountSummaryView` and finish the debit card code. If you want to compare your code with mine, you can find my completed `TransactionItemView` in the challenge 2 folder, located in the challenge folder.

Finishing up account summary

Now that our `TransactionItemView` is done. Open `AccountSummaryView` and let's add a new variable into our extension called `debitcard`:

```
var debitcard: some View {
    // Add Debit code here
}
```

Now, replace `// Add Debit code here` with the following code:

```
vStack {
    VStack(spacing: 0) {
        HStack {
            Text("TRANSACTIONS")
                .customFont(.custom(.bold, 20))
                .padding(.bottom, 2)
                .foregroundColor(.basePrussianBlue)
            Spacer()
        }

        Divider().padding(.bottom, 10)
    }

    VStack {
        ForEach(0 ..< 3) { _ in
```

```
        TransactionItemView()
    }
}.padding(.horizontal, 10)
}.padding(.horizontal, 10)
```

Next, replace `Text ("Debit code here")` with `debitcard`. Now we have two different account views. You can change the type to be an empty string at the top of the file, and you will see we are now showing the debit account summary instead. We are using this just for testing purposes and it will be removed in the next chapter. account summary is complete. Now we need to focus on the last item in the home view. Let's add our submenu next.

Creating our home view

If you had any problems with the challenge, please make sure you use the completed project to be all at the same spot. Next, we need to put everything together to create our home view. Open `AccountHomeView`. Add the following code inside of the `body` variable:

```
ZStack {
    Color(.baseLightWhite)
    .edgesIgnoringSafeArea(.all)

    ScrollView {
        VStack{
            HomeHeaderView()
            CardView()
            Divider()
            AccountSummaryView()
            HomeSubmenuView()
        }
    }
}
```

When you are done change your phone to iPhone 11 and run **Preview**, you will notice that if you are displaying the account summary for credit cards, there is a white bar between **Min. Amount Due Date** and **QUICK SERVICES**:

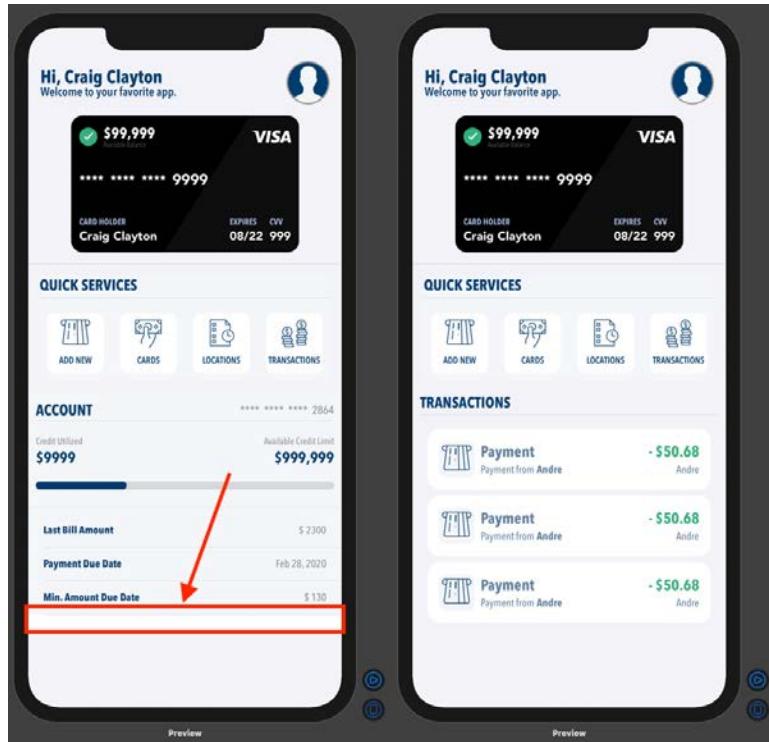


Figure 6.11

List is built on top of UITableView, and we can fix this by going back into AccountSummaryView and adding the following code after the padding modifier:

```
.onAppear {
    UITableView.appearance().tableFooterView = UIView()
    UITableView.appearance().backgroundColor = UIColor.clear
}
```

Let's talk about what we just added:

1. In this code, we are setting the table footer view to an empty UIView. Adding this gets rid of all of the empty cells you would see.
2. We are setting the background color of our table view to clear.

If you rerun **Preview**, you will see the white bar is fixed now. This is great; we now have the first view of our app complete.

Creating an account

We will work on one more screen, and that is where a user would create an account. As stated earlier, we will work on data, state-driven elements, and animations in *Chapter 7, Financial App – Core Data*. In this view, we will be saving the data entered into Core Data.

I broke this screen into five views – one we have already done (the card view). At the end of this section, you will be challenged to build another screen similar to this one:



Figure 6.12

Let's look at each view:

- Account type view
- Color button menu
- Credit card type menu

- Card view (previously done)
- Form view

We will work with the account type view first. Let's get started there now.

Account type view

The account type view contains custom radio buttons. When the user selects **Credit Card**, we will make sure that credit limit is shown and if the account is a debit card, then we will hide CREDIT LIMIT. Open `AccountTypeView`. Then add the following inside of the `body` variable:

```
HStack(spacing: 15) { // Step 1
    ForEach(0..<2) { index in // Step 2
        Button(action: {}) {
            HStack { // Step 3
                ZStack {
                    Circle()
                        .fill(Color.basePrussianBlue)
                        .frame(width: 18, height: 18)
                    Image("checkmark-selector")
                        .resizable()
                        .renderingMode(.template)
                        .frame(width: 10, height: 8)
                        .foregroundColor(.white)
                }
            }
            Text("Account Types") // Step 4
                .customFont(.custom(.medium, 14))
                .foregroundColor(.baseDustyGray)
            }
        }.buttonStyle(PlainButtonStyle())
    }
    Spacer()
}
```

Now that we have our code, let's break down each line:

1. Our main container is an `HStack` with its spacing set to 15.
2. Inside of the `HStack`, we have a `ForEach` that creates two buttons.
3. In our button, we have an `HStack` that contains a `ZStack`. In the `ZStack`, we have a `Circle` and an `Image`.
4. Next to our `ZStack`, we have a `Text` view to display the label for our radio button.

We will move on to the `Color` component next, and we will use this component shortly.

Color button menu

Our color button menu is a custom radio button; you can quickly turn the checkmark into a circle and repurpose it. The color button menu will change our card's color, but we will do that in *Chapter 7, Financial App – Core Data*. In this chapter, we will only be creating our color buttons. The first thing we need to do is create what we are going to use for each color.

Color view

We need to make a view that represents a color that is selected. When the color is selected, we want the checkmark to be visible. We will not add our checkmark inside of this view, though. Open `ColorView`, then add the following:

```
struct ColorView: View {  
    let color: Color  
  
    var body: some View {  
        ZStack {  
            Circle()  
                .fill(color)  
                .frame(width: 24, height: 24)  
            Circle()  
                .stroke(lineWidth: 1)  
                .frame(width: 30, height: 30)  
        }  
    }  
}
```

Besides passing in color, we are taking two `Circle` shapes and stacking them on each other. We give one a `fill` color and one a `stroke`. Next, update previews to the following:

```
struct ColorView_Previews: PreviewProvider {
    static var previews: some View {
        ColorView(color: .black)
            .previewLayout(.fixed(width: 50, height: 50))
    }
}
```

When you are finished, you will see the following:

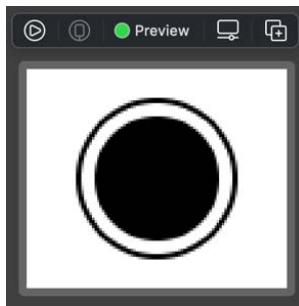


Figure 6.13

Color button menu

We just created our color view; now we need to use that to make all of our colors. We will keep track of the selected color and segment (basically an ID). Open `ColorButtonMenu` and then add the following above the `body` variable:

```
private var selectedSegment: Int = 0
private var selectedColor: Color = .baseEndeavourBlue

private var colors: [Color] = [.baseEndeavourBlue,
    .orange,
    .black,
    .red,
    .green,
    .purple,
    .gray
]
```

Here, we have three variables: `selectedSegment` is the currently selected ID. We have `selectedColor`, which is the currently selected color, and finally, an array of colors. You can change any of these colors to whatever you'd like. Ultimately, we want to create seven color views. Add the following inside of the `body` variable:

```
HStack {  
    ForEach(0..<colors.count) { index in  
        Button(action: {  
            }) {  
            ZStack {  
                ColorView(color: self.colors[index])  
                Image("checkmark-selector")  
                .resizable()  
                .renderingMode(.template)  
                .opacity(self.selectedSegment == index ? 1 : 0)  
                .frame(width: 12.0, height: 10.0)  
                .foregroundColor(.white)  
            }  
        }  
        .buttonStyle(PlainButtonStyle())  
    }  
}
```

Here, we are creating each button using a `ForEach`. Inside of the `Button`, we use a `ZStack` and stack our checkmark over it. We cannot use this code until we have some state in the next chapter. Update your previews to the following code:

```
struct ColorButtonMenu_Previews: PreviewProvider {  
    static var previews: some View {  
        ColorButtonMenu()  
        .previewLayout(.fixed(width: 300, height: 50))  
    }  
}
```

When you are done, you will see the following:

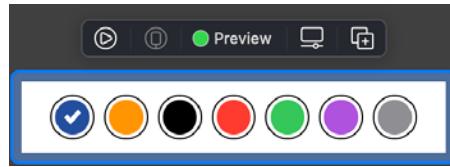


Figure 6.14

When you view these in **Preview**, they may render wrongly, but they will render correctly when seen on a device or in the simulator, so do not worry about it. Next, we will work on our credit card type menu.

Credit card type menu

We have another section that has buttons that we need to work on now:

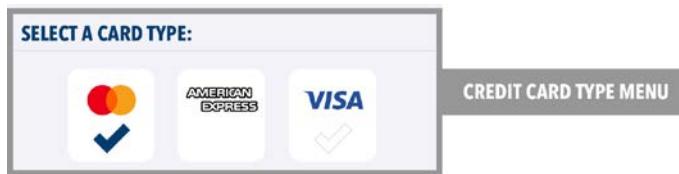


Figure 6.15

As you can see, each button has the same look but with a different logo. In the last section, we used `PlainButtonStyle()`, and now we will create a custom button style for these buttons that we can reuse repeatedly.

Custom button style

It is pretty easy to create a `ButtonStyle`. I created this one for you, please open `CreditCardStyle` and you should see the following:

```
struct CreditCardStyle: ButtonStyle {
    func makeBody(configuration: Self.Configuration) -> some View {
        configuration.label
            .frame(width: 75, height: 80)
            .background(Color.white)
            .cornerRadius(10)
            .buttonStyle(PlainButtonStyle())
    }
}
```

Make sure you change the import from Foundation to SwiftUI. When using `ButtonStyle`, we use the `makeBody` function and pass our modifiers to the `configuration.label`. We now have a new `ButtonStyle` called `CreditCardStyle`.

Let's see how we can use this style. Open `CreditCardTypeMenuView` and then above the `body` variable, add the following array:

```
let logos = [
    "mc-logo-selector",
    "visa-logo-selector",
    "am-logo-selector"
]
```

We just created an array for our image logos. Now add the following:

```
VStack { // Step 1
    VStack(alignment: .leading, spacing: 3) { // Step 2
        Text("SELECT A CARD TYPE")
            .customFont(.custom(.bold, 18))
            .foregroundColor(.basePrussianBlue)
            .padding(.leading, 10)

        Divider()
    }.padding(.top, 15)

    HStack { // Step 3
        ForEach(0..<logos.count) { index in
            Button(action:{ }) {
                VStack {
                    Image(self.logos[index])
                    Image("checkmark-outline-selector")
                }
            }
        }
        .buttonStyle(CreditCardStyle())
    }
}.padding(.top, 20)
}.background(Color.clear)
```

Let's quickly break down each step we did here:

1. Our main container is a `VStack` container.
2. Inside of the `VStack` container, we have another `VStack` container, the alignment is set to `.leading`, and spacing is set to 3. Inside of the `VStack` container, we have a `Text` view and a `Divider`.
3. We have an `HStack` inside of the `VStack` container, which is the container for our buttons. We use a `ForEach`, which creates our buttons.

Update previews to the following:

```
struct CreditCardTypeMenuView_Previews: PreviewProvider {
    static var previews: some View {
        CreditCardTypeMenuView()
            .previewLayout(.fixed(width: 400, height: 180))
    }
}
```

We are finished with our custom buttons, and we now know how to create a custom button style. Finally, we will develop our form section, and this will be the final section that we need to complete before we bring it all together.

Form view

We need to create the form section, and instead of using a form, we will make a custom text field for our form. Although we could use a form, I decided to take the custom route instead. Let's take a look at the form section again:

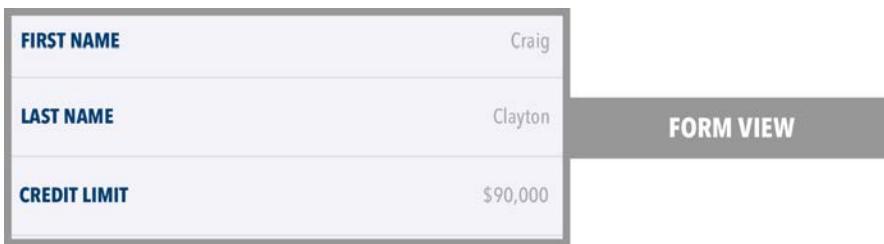


Figure 6.16

Since this is custom, we are going to create a custom form text field. Let's start this now.

Form text field

Let's start by opening `FormTextField`. Then, above the `body` variable, add the following variables:

```
@State private var value = ""
var text = ""
```

Ignore the `@State` property – we will cover this in *Chapter 7, Financial App – Core Data*. Next, inside of the `body` variable, add the following:

```
VStack(alignment: .center, spacing: 10) { // Step 1
    HStack(alignment: .center, spacing: 0) { // Step 2
        Text(text.uppercased()) // Step 3
            .customFont(.custom(.bold, 14))
            .foregroundColor(.basePrussianBlue)

        TextField(text.uppercased(), text: self.$value,
            onEditingChanged: { _ in
        }, onCommit: {
            print("\u{1f49}(\u{1f49}value) ")

        }).multilineTextAlignment(.trailing)
            .customFont(.custom(.medium, 14))
            .padding(.trailing, self.value == "" ? -5 : 10)

        Button(action: {
            self.value = ""
        }) {
            Image(systemName: "xmark.circle.fill")
                .opacity(self.value == "" ? 0 : 1)
                .offset(x: self.value == "" ? 35 : 0)
                .padding(.trailing, self.value == "" ? 0 : 5)
        }
    }.padding(.horizontal, 0)
}
.frame(height: 40)
.padding(.top, 10)
```

Let's break down the code we just added:

1. We are using a `VStack` container for our main container.
2. Inside of the `VStack` container, we have an `HStack`.
3. Inside of the `HStack`, we have another `HStack` with a `Text` view, a `TextField`, and a `Button`. The `TextField` is used to take the user's information.

For the logic, we initially hide the button, and it will only appear if it has text. The button is used to clear out text inside of the text field.

We need to update previews by adding the following:

```
struct FormTextField_Previews: PreviewProvider {  
    static var previews: some View {  
        FormTextField(text: "First Name")  
            .previewLayout(.fixed(width: 600, height: 50))  
    }  
}
```

When you are finished, you will see the following:

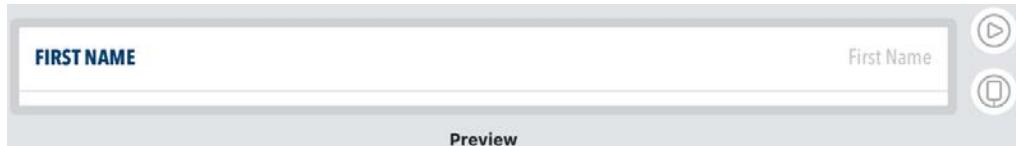


Figure 6.17

Now, let's move on to creating our form.

Account form view

Open `AccountFormView` and above the `body` variable, add the following variables:

```
private let type = "creditcard"
```

These variables are going to be placeholders until the next chapter so for now, just ignore them. We need them for the `TextField` and the `type` variable is used for logic that we will later update as well. Under the `type` variable, add the following `init` method:

```
init() {
    UITableView.appearance().backgroundColor = UIColor.
baseWhite
    UITableView.appearance().tableFooterView = UIView(frame:
    CGRect(x: 0, y: 0, width: 0, height:
    Double.leastNonzeroMagnitude))
    UITableView.appearance().tableHeaderView = UIView(frame:
    CGRect(x: 0, y: 0, width: 0, height:
    Double.leastNonzeroMagnitude))
    UITableView.appearance().separatorStyle = .none
}
```

Here, we are just clearing out some of the design defaults that Apple adds to `UITableView`. Feel free to mess with each one so that you can see how it affects `List`. Next, inside of the `body` variable, add the following:

```
List { // Step 1
Section { // Step 2
    FirstNameView()
    LastNameView()
    CardLimitView()
}
.listRowBackground(Color.baseWhite)
.listRowInsets(EdgeInsets(top: 0, leading: 10, bottom: 0,
trailing: 0))
}
.listStyle(GroupedListStyle())
.background(Color.baseWhite)
```

Let's break down the code we just added:

1. We use a `List` as our main container.
2. Inside of our `List`, we add a `Section`, and inside of the section, we add our `FormTextField`.
3. Finally, if the user is creating a credit card, we will show the **CREDIT LIMIT** field. Otherwise, it is hidden.

Finally, update preview to the following:

```
struct AccountFormView_Previews: PreviewProvider {
    static var previews: some View {
        AccountFormView().previewLayout(.fixed(width: 600,
height: 200))
    }
}
```

When you are done, you will see the following:

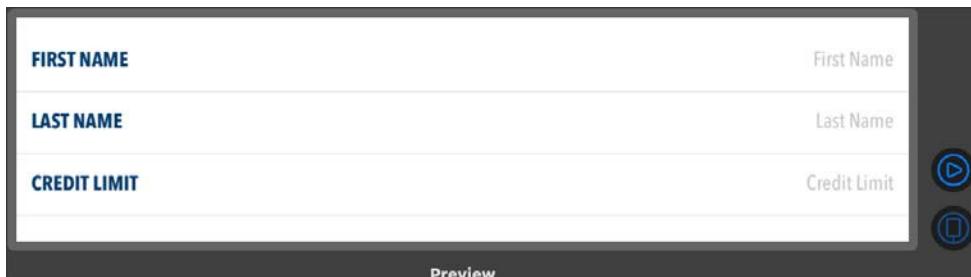


Figure 6.18

We are now done creating all of the views we need to create an account. We need to make way for the user to switch between creating a debit card account and a credit card account. Now, let's create this view.

Bringing it all together – CreateAccountView

Open `CreateAccountView` and look for the extension created for you:

```
private extension CreateAccountView {
    // Add code here
}
```

Inside of the extension, we are going to create some variables to store our smaller views. Let's create the header by adding the following variable:

```
var header: some View {
    VStack(spacing: -8) {
        Text("CREATE AN ACCOUNT")
            .foregroundColor(.basePrussianBlue)
            .customFont(.custom(.bold, 24))

        Text("Add a new account")
            .foregroundColor(.baseDustyGray)
            .customFont(.custom(.demibold, 16))
    }.padding(.bottom, 20)
}

// Add next step here
```

We are creating a VStack with two Text views. Next, let's create our account button by replacing // Add next step here with the following code:

```
var createAccountBtn: some View {
    Button(action: { }) {
        ZStack {
            Text("CREATE ACCOUNT")
                .customFont(.custom(.bold, 22))
                .frame(width: 294, height: 34)
                .background(Color.basePrussianBlue)
                .cornerRadius(4)
                .foregroundColor(.white)

            RoundedRectangle(cornerRadius: 6)
                .stroke(lineWidth: 2)
                .foregroundColor(.basePrussianBlue)
                .frame(width: 300, height: 40)
        }
    }.padding(.top, 50)
```

```
.padding(.bottom, 20)
}

// Add next step here
```

Inside of `createAccountBtn`, we create a custom button using a `ZStack`, which contains a `Text` view and a `RoundedRectangle`. We need to add our account selector next. Replace `// Add next step here` with the following:

```
var accountSelector: some View {
    VStack {
        VStack(alignment: .leading, spacing: 3) {
            Text("SELECT AN ACCOUNT TYPE")
                .customFont(.custom(.bold, 18))
                .foregroundColor(.basePrussianBlue)
                .padding(.leading, 10)

            Divider()
        }.padding(.top, 15)

        HStack(alignment: .center) {
            AccountTypeView()
            Spacer()
        }.padding(.leading, 10)
    }
}

// Add next step here
```

We added a header to our `AccountTypeView` with a bit of padding. Next, let's add the guts of our form by replacing `// Add next step here` with the following code:

```
var main: some View {
    ScrollView {
        VStack {
            accountSelector
            AccountFormView()
        VStack {
```

```
        CreditCardTypeMenuView()
        CardView().padding(.vertical, 20)
        ColorButtonMenu()
        Spacer()
    }.padding(.top, 0)

    createAccountBtn
}
}
}
```

In main, we are adding all of our newly created views into a `ScrollView` that holds a `VStack`. `main` is complete, and we are now done with our variables. Let's get them into the body. Update the `body` variable by replacing `Text ("Hello World")` with the following:

```
var body: some View {
    ZStack {
        Color(.baseWhite)
        .edgesIgnoringSafeArea(.all)

        VStack {
            header
            ScrollView {
                main
            }
        }
    }
}
```

Using the same technique that we used in `AccountFormView`, we can easily see we have a header for our `ScrollView`, which holds our main content.

You should have the following in previews:

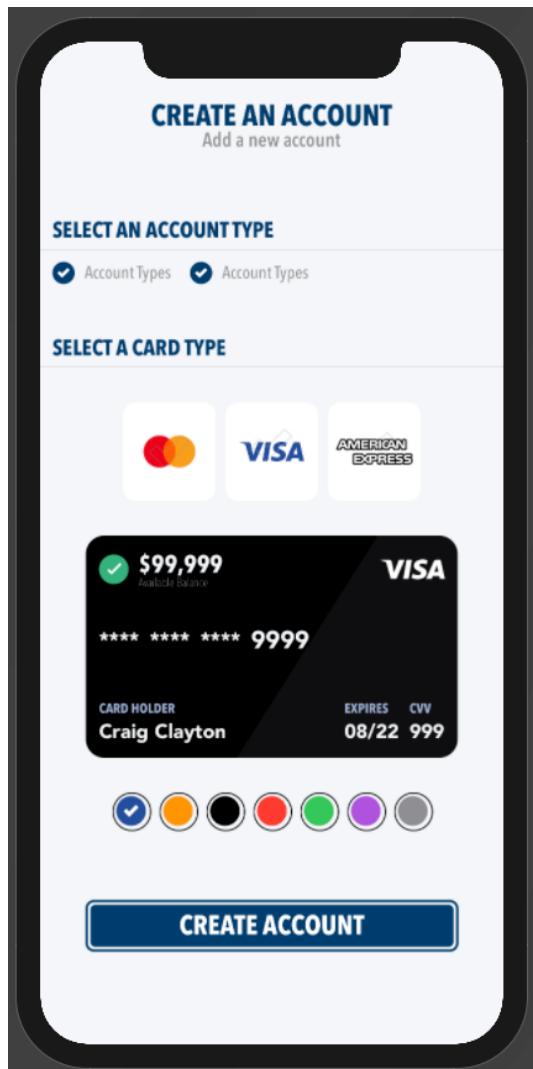


Figure 6.19

We are finished with `AccountFormView` and there is one more screen that needs to be completed.

Create account list (challenge 3)

Before moving on to the next chapter, we want to create a list of cards for each account created. When you tap on the card, we want to go to the account home screen. For now, just create a screen that only lists cards, and in the next chapter, we will look at how to drive that data using Core Data:

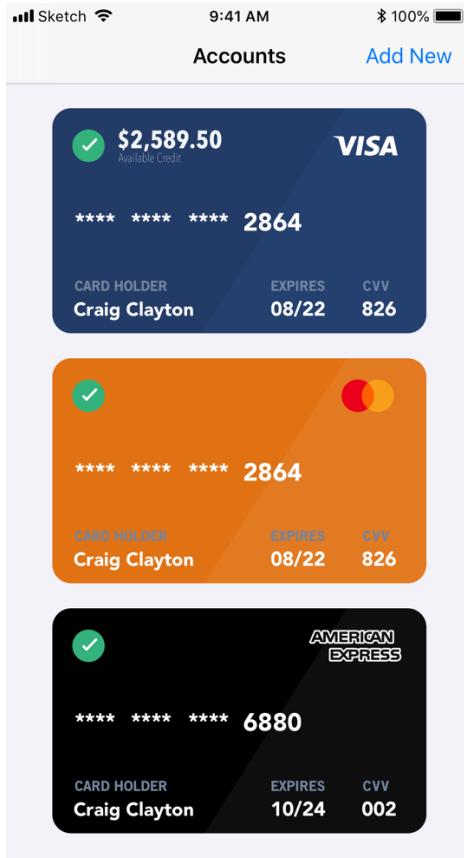


Figure 6.20

Follow the steps given below:

- Use a `NavigationView` to create your list.
- `NavigationView` must have the title **Accounts**.
- Create a new `CardView` and rename it `CardListRow` and save or move it into the `Misc.` folder.
- Duplicate `CardView` and put code into `CardListRow`.

You have all of the designs in the `specs` folder if you need them.

Important note

We cannot reuse our `CardView` because of what we will be doing in the next chapter. Your list view may have arrows in it (this is fine as it is the default behavior for List views).

If you get stuck on this last challenge, you can look at the files in the `challenge_3` folder for this chapter or in the completed folder to see what you missed. You will need these files in the next chapter, so please make sure your project is up to date.

Summary

In this chapter, we designed a few screens using SwiftUI. We created our first button modifier, which allows us to create custom buttons. We also created a custom text field for our form to reuse in our form. We got to see a process for creating views by creating smaller views and then combining them to make our main views.

In the next chapter, we will bring it all to life by adding state and working with data and Combine.

7

Financial App – Core Data

In the last chapter, we set up our design for our financial app. In this chapter, we are going to work with the data side. We will use Core Data to store our data. Core Data is pretty popular and is one of those hot topics among the iOS community. On the one hand, you have those who hate it and haven't used it for years, and then you have those who have used it since iOS 10 and can see what Apple has done to it over the course of the last few years. Core Data has come a long way from what it was pre-iOS 10. I do not use Core Data in my everyday work, but I find it not as bad as it used to be. I know that if I need an easy system that I want to get up and running, I will use Core Data without hesitation.

In this chapter, we will be covering the following topics:

- What is Core Data?
- Integrating Core Data with SwiftUI
- Displaying Core Data in a list

We are going to cover a lot of Core Data basics in this chapter. Take your time, even if it means going back and doing it a few times. Let's get started and learn about Core Data. If you are already comfortable with Core Data, treat this as a reminder.

Technical requirements

The code files for this chapter can be found here: <https://github.com/PacktPublishing/SwiftUI-Projects/tree/master/Chapter07>.

What is Core Data?

Let's start by taking a quote directly from Apple: "Core Data is a framework for managing and persisting an object graph." Core Data is not a database, but it allows us to use the API to read and write them from permanent storage. The Core Data framework shines at handling complex object graphs. Core Data shines when working with massive datasets because it is excellent when sorting and filtering data and performing undo and redo operations.

Core Data pre-iOS 10 was more complicated than it needed to be, and every year since iOS 10, it has gotten a bit easier to work with in your app. In this chapter, we will stick to the basics of saving and retrieving data from Core Data. Before we get into that, we need first to understand how Core Data works and some of the vocabulary associated with Core Data. When working with Core Data, you should be familiar with the managed object model, the managed object context, and the persistent store coordinator. Here is a diagram of each one and how they are all connected:

CORE DATA STACK

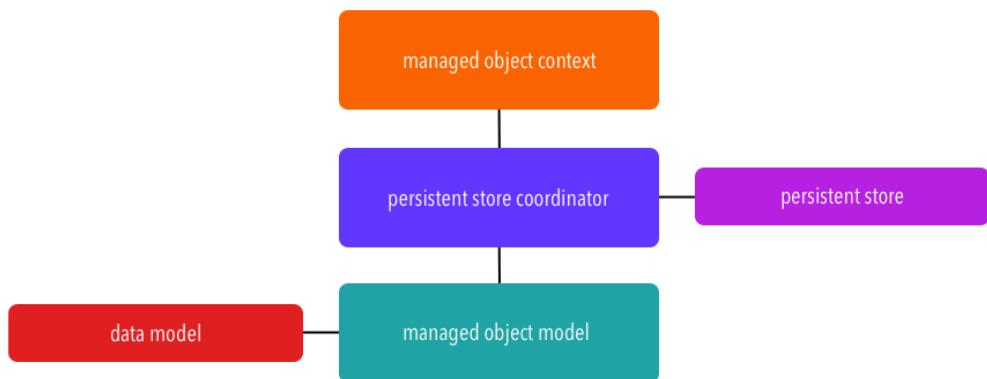


Figure 7.1

What is a managed object model?

The managed object model (`NSManagedObjectModel`) represents the data model of your Core Data application. The object model interacts with all of your entities (models) that you create within your app. The object model will know of any relationships that your data has in your app. The object model interacts with your data model as well as the persistent store coordinator.

What is an object context?

The managed object context manages a collection of model objects, which it receives from the persistent store coordinator. The object context is responsible for all **CRUD (Creating, Reading, Updating, Deleting)** work. The object context is what you will be interacting with the most.

What is a persistent store coordinator?

The persistent store coordinator has a reference to both the object model and the managed object context. The persistent store coordinator communicates with the persistent object store. The persistent store coordinator will interact with an object graph. This graph is where you will create your entities and set up relationships within your app.

Let's look at how we can add Core Data to our SwiftUI project. Setting up Core Data is similar to another Swift project. Once we have our Core Data set up, that is where things differ slightly between a Swift and a SwiftUI project.

Creating a data model

The data model is where we create our app's model objects and their properties. In this project, we are going to create two model items—account and card. We will create one together, and then you will make the second one on your own.

In the **Navigator** panel, right-click on the **Model** folder and click **New File**. Inside **Choose a template for your new file**, select **iOS** at the top, and then scroll down to the **Core Data** section and select **Data Model**. Then, hit **Next**:

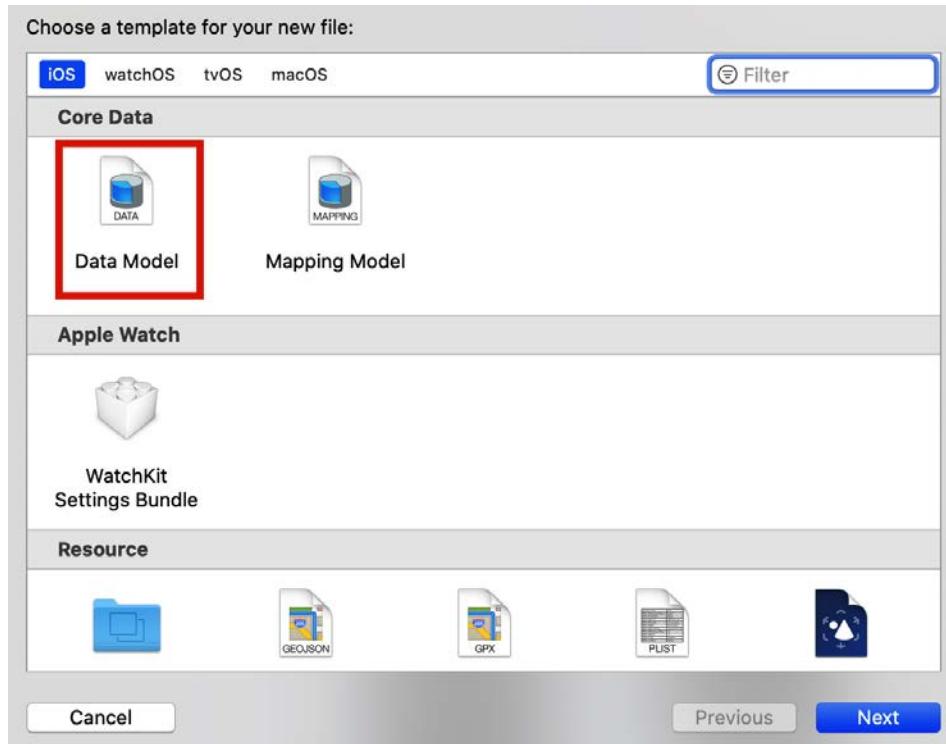


Figure 7.2

Name the file **FinancialApp** and click **Create**. We have created our first Core Data data model. We now need to develop entities inside our data model. We will make our first one together next.

Creating Core Data entities

We just made our data model, and now we need to add some entities. You should have your data model open and, in the bottom-left corner, click on the **Add Entity** button:

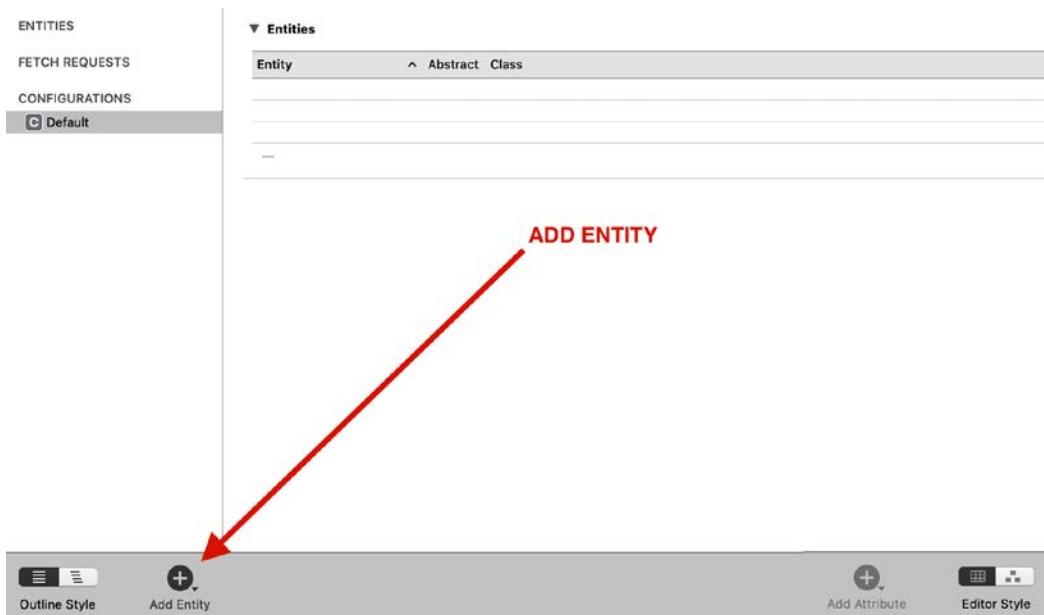


Figure 7.3

Next, change **Editor Style** to **Graph Style**:

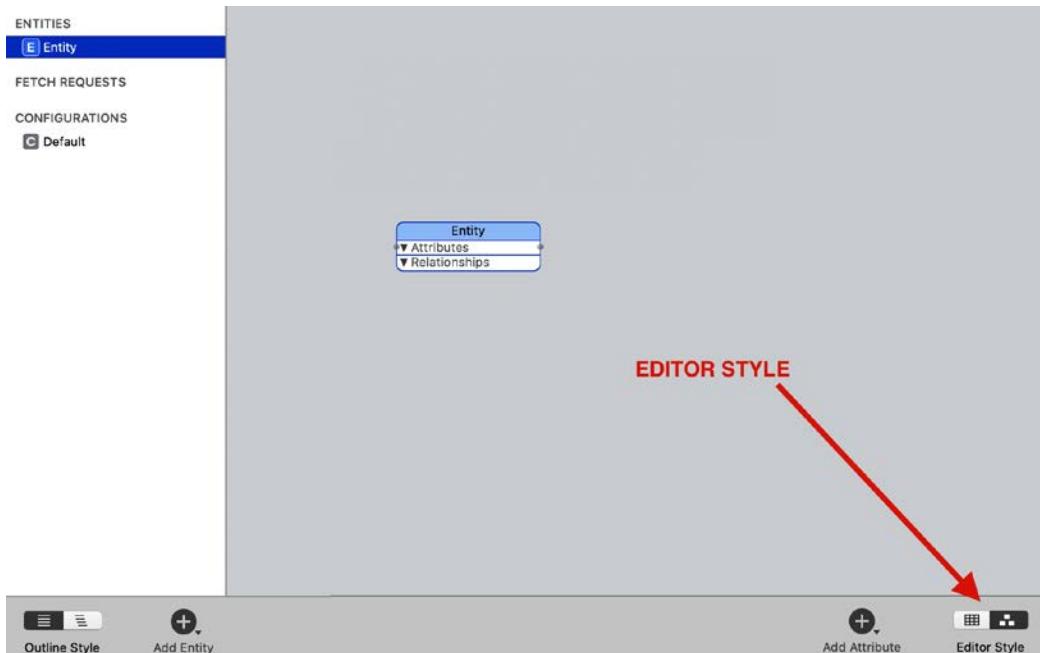


Figure 7.4

In **Graph Style**, double-click on **Entity** inside the box to change the name:

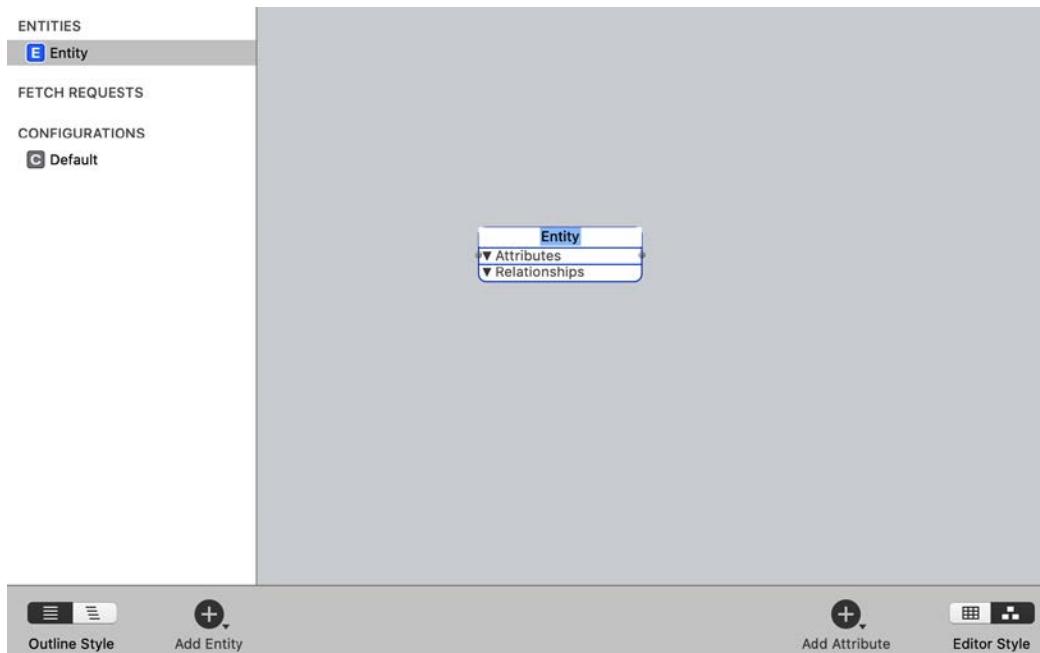


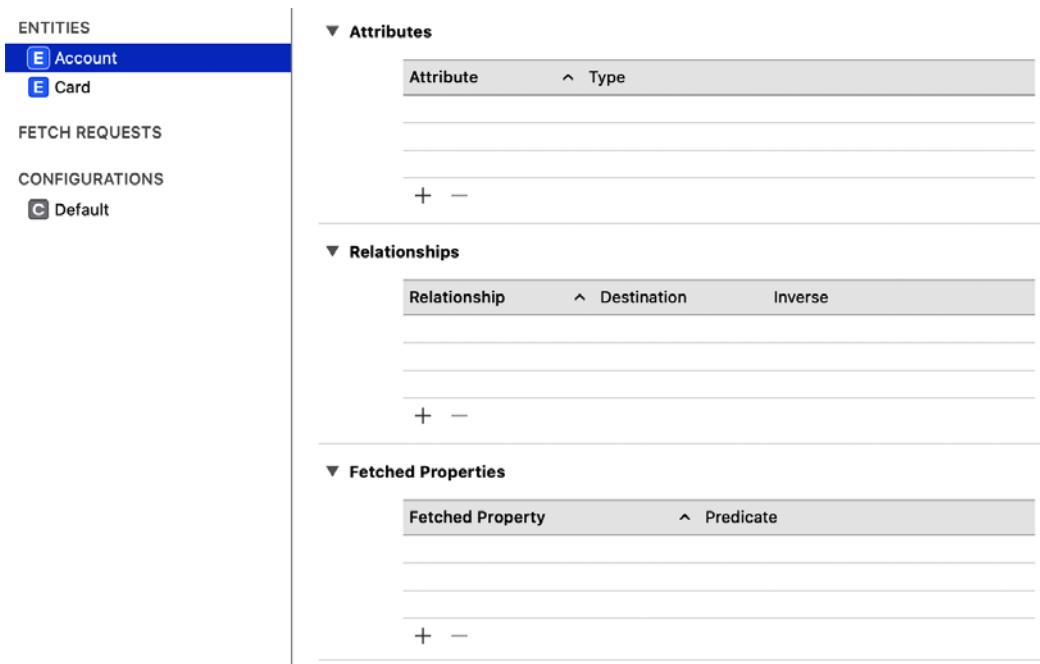
Figure 7.5

Update the text to say **Account** and then hit **Enter**. Repeat the steps and add one more entity named **Card**. We now have our entities created. Next, we need to add our attributes.

Adding model properties

Properties or attributes can be added in multiple ways. You can add them in our current view, Graph view, or add them in a Table view. Since we just used the Graph view, let's change to Table view and see how that looks.

In the bottom-right corner of Xcode, click the icon that looks like a table. Then select **Account**, under **Entities**. When you are finished, you will see the following in Xcode:



The screenshot shows the Xcode interface for configuring an entity named 'Account'. The left sidebar lists 'ENTITIES', 'FETCH REQUESTS', and 'CONFIGURATIONS'. The 'Account' entity is selected in the 'ENTITIES' list. The main area is divided into three sections: 'Attributes', 'Relationships', and 'Fetched Properties'. Each section has a table with a header row and a '+' and '-' button for adding or removing rows.

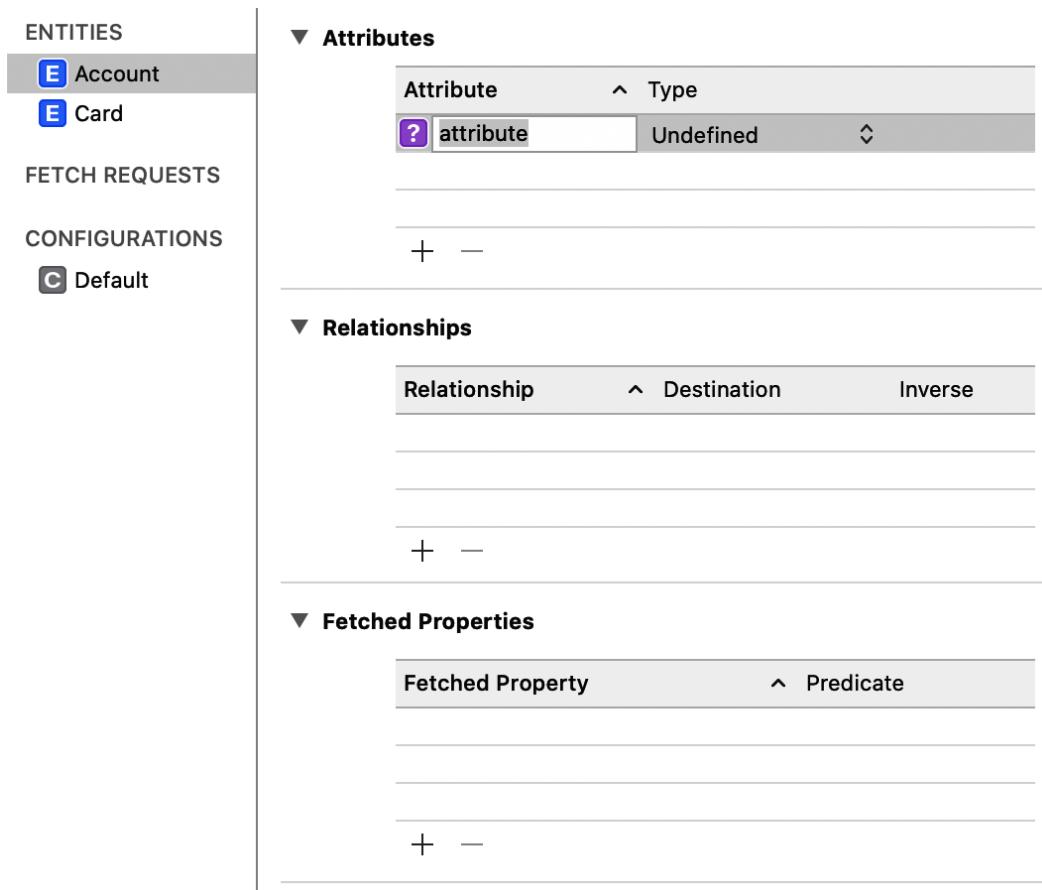
Attribute	Type
+	-

Relationship	Destination	Inverse
+	-	

Fetched Property	Predicate
+	-

Figure 7.6

Next, click on the **Add Attribute** button, which is next to the **Graph view** button. You will see the following:



The screenshot shows the Xcode Core Data editor. On the left, there is a sidebar with three sections: **ENTITIES**, **FETCH REQUESTS**, and **CONFIGURATIONS**. Under **ENTITIES**, there are two entities: **Account** (selected) and **Card**. Under **CONFIGURATIONS**, there is one configuration: **Default**. The main area is titled **Attributes** and contains a table with one row. The table has three columns: **Attribute** (containing **attribute**), **Type** (containing **Undefined**), and a dropdown arrow. Below the table are **+** and **-** buttons. There are also three sections below the table: **Relationships**, **Fetched Properties**, and **Relationships** again, each with a table and **+** and **-** buttons.

Figure 7.7

Update **Attribute** to be `acctNumber`, and then press **Enter**. Next, set **Type** in the dropdown to **String**. When you are done, you will see the following:

The screenshot shows the Xcode Core Data editor with the following structure:

- ENTITIES** section on the left:
 - Account** (highlighted in grey)
 - Card**
- FETCH REQUESTS** section:
 - Default**
- CONFIGURATIONS** section:
 - Default**
- Attributes** section (under the Account entity):

Attribute	Type
acctNumber	String

 Buttons: + - (below the table)
- Relationships** section (under the Account entity):

Relationship	Destination	Inverse

 Buttons: + - (below the table)
- Fetched Properties** section (under the Account entity):

Fetched Property	Predicate

 Buttons: + - (below the table)

Figure 7.8

We now need to add the rest of the attributes for the account as well as the card. Add the following attributes and types for **Account**:

- **Attribute:** lastName; **Type:** String
- **Attribute:** type; **Type:** String
- **Attribute:** balance; **Type:** Float
- **Attribute:** dateCreated; **Type:** Date
- **Attribute:** firstName; **Type:** String

When you are done, you should have the following:

▼ **Attributes**

Attribute	Type	▼
S acctNumber	String	▼
S lastName	String	▼
S type	String	▼
N balance	Float	▼
D dateCreated	Date	▼
S firstName	String	▼

+

-

Figure 7.9

Next, add the following attributes for **Card**:

- **Attribute:** color; **Type:** String
- **Attribute:** cvv; **Type:** String
- **Attribute:** dateCreated; **Type:** Date
- **Attribute:** expirationDate; **Type:** Date
- **Attribute:** id; **Type:** String
- **Attribute:** logo; **Type:** String
- **Attribute:** number; **Type:** String

When you are done with **Card**, you will see the following:

▼ **Attributes**

Attribute	Type	▼
S color	String	▼
S cvv	String	▼
D dateCreated	Date	▼
D expirationDate	Date	▼
S id	String	▼
S logo	String	▼
S number	String	▼

+

-

Figure 7.10

We are done with the attributes. Next, we need to add a relationship. We will need to add a one-to-one relationship between our account and card. Let's look at how to do that next.

Understanding Core Data relationships

In Core Data, we have four different types of relationships. We will not be using all four, but let's see how each one works:

- A one-to-one relationship means that an entity is linked to another entity.
- A one-to-many relationship means that an entity is linked to many entities.
- A many-to-one relationship means that many entities are linked to one entity.
- A many-to-many relationship means that many entities are linked to many entities.

In our app, we have an account entity, which will be linked to our card entity using a one-to-one relationship. Typically, an account can have multiple cards, but we will just have one card for the sake of simplicity.

Let's go back to **Table view**, and with **Account** selected, click the plus icon under the **Relationships** table. Set **Relationship** to **Card**, and **Destination** to **Card**. Now, choose **Card** and set **Relationship** to **Account**, and **Destination** to **Account**, and then set **Inverse** to **Card**. By selecting **Inverse**, we let Core Data understand that the link goes both ways.

We have completed setting up our entities, and now we need to understand how our code is generated.

Core Data Codegen

When Xcode 8 was released, Apple added a new **Codegen** setting to the Xcode data model editor. To view the **Codegen** dropdown, select the **Account** entity, and then open **Data Model Inspector** in the right pane:

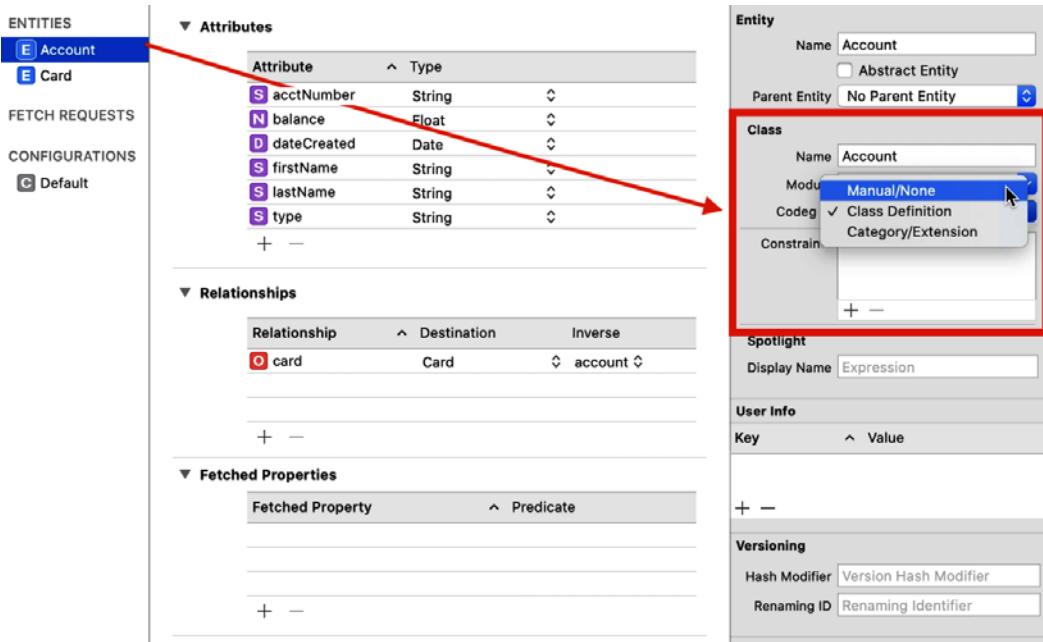


Figure 7.11

The **Codegen** setting helps you manage and maintain `NSManagedObject` subclasses. Three possible settings come with the **Codegen** configuration:

- Manual/None
- Class Definition
- Category/Extension

Let's take some time to break down each setting.

Manual/None

The **Manual/None** setting was the default setting pre-Xcode 8. Using this setting means you create your entity files. Using this setting is more of an advanced feature for those who are comfortable with Core Data.

Class Definition

The **Class Definition** setting is now the default setting since Xcode 8, and this setting will create the files for you from behind the scenes. Once you make your entities, you can hit *Command + B*, and as long as you have everything set up, Xcode will create the files. You will not see any project files, but there are actual files in the finder. You shouldn't edit these files because Xcode will overwrite them.

Class Definition is what we will use in this chapter. Since using Class Definition, the code is autogenerated. We can create extensions for our entities to add extra functionality, precisely what we will do later in the chapter.

Category/Extension

The **Category/Extension** setting is a mix of the first two settings. Xcode will only autogenerated `Card+CoreDataProperties` for you. Class Definition is my preferred way to work with Core Data classes.

If you haven't already done so, hit *Command + B*, and your project should build successfully. If not, make sure that all of your properties have types. You cannot produce the project without each property having a type. While we are here, let's create an extension for both `Account` and `Card`.

Account extension

For `Account`, we want to display the balance in currency format, and we also want to create an instance of our `Card`. Open `Account+Extension` and then add the following code:

```
public var accountCard: Card {  
    card ?? Card()  
}  
  
public var displayBalance: String {  
    let currencyFormatter = NumberFormatter()  
    currencyFormatter.usesGroupingSeparator = true  
    currencyFormatter.numberStyle = .currency
```

```
currencyFormatter.locale = Locale.current
currencyFormatter.maximumFractionDigits = 0
if let priceString = currencyFormatter.string(from:
    NSNumber(value: self.balance)) {
    return priceString
} else { return "$0.00" }

}

static var accountFetchRequest: NSFetchedRequest<Account> {
    let request: NSFetchedRequest<Account> = Account.
        fetchRequest()
    request.sortDescriptors = [NSSortDescriptor(key:
        "dateCreated", ascending: true)]
    return request
}
```

Here, we just created an instance of Card, which we will use later in the book. We now have a formatter to be able to get our account displaying in currency format. Let's create an extension for Card next.

Card extension

Open Card+Extension and then add the following:

```
extension Card {
    public var cardExpDate: String {
        let formatter = DateFormatter()
        formatter.dateFormat = "MM/YY"
        return formatter.string(from: self.expirationDate ?? Date())
    }
}
```

We can now display the expiration date inside our card's incorrect format. Now that we understand a bit more about Core Data, let's create our Core Data manager next.

Core Data manager

We now need to set up our Core Data manager. Open `CoreDataManager` and add the following class declaration under the `import` statement:

```
class CoreDataManager: ObservableObject {  
    // Add next step here  
}
```

Our Core Data manager will be a singleton. A singleton is a class that can only be created once, but we will subclass this file using `ObservableObject`. Replace `// Add next step here` with the following code:

```
static var shared = CoreDataManager()  
  
private init() {  
    persistentContainer.viewContext.  
        automaticallyMergesChangesFromParent = true  
}  
  
// Add next step here
```

In this code, we are creating the standard `static` variable. We also have a `private init` that gets run, which calls `automaticallyMergesChangesFromParent`. `automaticallyMergesChangesFromParent`, as the name suggests, automatically merges the changes from the parent on the child context. Now that we have our singleton code and the `init` method is complete, let's create some Core Data variables. Replace `// Add next step here` with the following code:

```
public var context: NSManagedObjectContext { // (1)  
    get {  
        return self.persistentContainer.viewContext  
    }  
}  
  
var persistentContainer: NSPersistentContainer = { // (2)  
    let container = NSPersistentContainer(name:  
        "FinancialApp")  
    container.loadPersistentStores { (storeDescription,  
        error) in
```

```
        guard error == nil else {
            print(error?.localizedDescription as Any)
            return
        }
    }
    return container
}()

// Add next step here
```

We added two new variables to our Core Data manager. Let's now go over each one:

1. The `context` variable is where we create an instance of `NSManagedObjectContext`. We will use this the most when we are doing stuff with Core Data.
2. The `persistentContainer` variable is creating our `NSPersistentContainer`. You see boilerplate code and all you have to do is update the name, so it matches the word we used for `xcodemodel`, which we created earlier.

We are almost done with our Core Data manager. We just have one more method to add. Replace `// Add next step here` with the following code:

```
func save() {
    let context = persistentContainer.viewContext

    if context.hasChanges {
        do {
            try context.save()
        } catch let error {
            print(error.localizedDescription)
        }
    }
}
```

We added a `save()` method to our Core Data manager. When we go to save, update, or delete an object, we must always call the `save()` method. We are now done with the Core Data manager and can move on to setting up our app to use Core Data.

Setting up our EnvironmentObject

We want to set up our Core Data manager inside our `FinancialApp`. We can set up Core Data as an `@StateObject` variable and pass it over to `ContentView`. Open `FinancialApp.swift`, and you should see the following:

```
@StateObject private var model = CreateAccountViewModel()
```

Next, update `ContentView()` after the `@StateObject` variable and add the following:

```
let manager = CoreDataManager.shared
```

Then, after `.environmentObject(model)`, add the following modifier:

```
.environment(\.managedObjectContext, manager.context)
```

We are done updating `FinancialApp`. Now that we see how our Core Data manager is set up, let's cover one more topic before digging into our app, and that's CRUD.

How to use Core Data manager

Earlier, we created our Core Data manager. This manager is responsible for creating and fetching objects. We will not cover updating and deleting an item in this app, but it is similar to saving objects. I want to make sure that you know how to do all of these operations even if we do not delete objects in the app. The following examples can be used with any entity. Just swap out `EntityNameHere` with yours and your properties.

Creating steps (aka save)

The most common task you will perform inside Core Data will be saving objects. In order to save an object, you would do the following:

```
CoreDataManager.shared.context.perform { // (1)
    let item = EntityNameHere(context:
        CoreDataManager.shared.context) // (2)
    item.name = "Name Here"
}

CoreDataManager.shared.save() // (3)
}
```

Let's break down each part of this code:

1. Wrap everything inside the `perform` block.
2. Create a new `EntityNameHere` (`NSManagedObject` object) in the context.
3. Save the context.

Next, we take a look at how to read Core Data objects.

Reading steps (fetching an object)

Fetching data from Core Data requires a few steps, but it is not hard to do. To pull or fetch data, you would do the following:

```
var item: EntityNameHere?  
CoreDataManager.shared.context.perform {  
    let request: NSFetchedRequest<EntityNameHere> =  
        EntityNameHere.fetchRequest()  
    request.predicate = NSPredicate(format: "color = %@",  
        "red")  
    request.fetchLimit = 1  
  
    item = try? CoreDataManager.shared.context.fetch(request)  
}
```

We are not adding this code to any files as this is just for reference. Let's understand what we did in this code:

1. Create an `NSFetchRequest` for your entity.
2. Create an array to store the entities.
3. Create a `do-catch` and use an instance of your **Core Data Manager** context to call `fetch` on the context.
4. Then, loop through the fetched items and add them to your array.
5. Add `fatalError` for any errors that might occur.

Fetching for an item or items is the same process. If you wanted to fetch all items for an entity, you would remove the `fetchLimit`. You would also remove `.first` from the `fetch`:

```
var entities: [Card] = []  
CoreDataManager.shared.context.perform {
```

```
let request: NSFetchedResultsController<EntityNameHere> =  
    EntityNameHere.fetchRequest()  
if let items = try?  
    CoreDataManager.shared.context.fetch(request) {  
        for item in items {  
            entities.append(item)  
        }  
    }  
}  
print(entities)
```

As you can see, fetching one item versus many items is very similar. Now that we know how to fetch an item, let's look at how to update an item next.

Updating steps

Even though we do not actually update items, you may still want to. In order to update an entity, you would do the following:

```
CoreDataManager.shared.context.perform { // (1)  
    let request: NSFetchedResultsController<EntityNameHere> =  
        EntityNameHere.fetchRequest() // (2)  
    request.predicate = NSPredicate(format: "id = %@", id)  
    // (3)  
    let item = try?  
        CoreDataManager.shared.context.fetch(request).first  
    // (4)  
    item.name = "Ruby" // (5)  
  
    CoreDataManager.shared.save() // (6)  
}
```

We are not adding this code to any files as this is just for reference. Let's now break down the steps involved in updating an entity:

1. Perform everything within the `perform` block.
2. Create an `NSFetchRequest` with your entity.
3. Set up the `NSPredicate`; in this example, we search for the ID that is passed.

4. Create an `if-let` for your fetch. This can be done inside a `do-catch` block. I used `if-let` instead because, in this chapter, I am returning an optional object. If nothing comes back, it will return `nil`. A `do-catch` block would do the same. If you wish to use a `do-catch` block instead, this will have the same outcome.
5. Update the `name` property. The `name` property can be any property that you need to update. In this example, we just updated `name`.
6. Use Core Data manager's `save()` method to save all the changes.

Now that you are familiar with updating, let's move on to deleting an object.

Deleting steps (single object)

We will not be deleting a card from the app, but if you needed to, you would want to get the ID of the card you want to delete and write the following:

```
CoreDataManager.shared.context.perform { // (1)
    let request: NSFetchedResultsController<EntityNameHere> =
        EntityNameHere.fetchRequest() // (2)
    request.predicate = NSPredicate(format: "id = %@", id)
    // (3)

}

if let item = try?
    CoreDataManager.shared.context.fetch(request).first {
    // (4)
    CoreDataManager.shared.context.delete(item) // (5)
}
}
```

We are not adding this code to any files as this is just for reference. Let's break down what we are doing in the preceding code:

1. Perform everything within the `perform` block.
2. Create an `NSFetchRequest` with your entity.
3. Set up the `NSPredicate`. In this example, we search for the ID that is passed.

4. Create an `if let` for your fetch. This can be done inside a `do-catch` block. I used `if let` instead because, in this chapter, I am returning an optional object. If nothing comes back, it will return `nil`. A `do-catch` block would do the same. If you wish to use a `do-catch` block instead, this will have the same outcome.
5. Pass the entity to the `delete` method.

Deleting an object is pretty simple and deleting multiple objects is very similar. Let's now take a look at how to delete multiple items at once.

Batch delete

We will not be deleting objects at all, but if you wanted to delete all the entities from Core Data, you would write the following code:

```
CoreDataManager.shared.context.perform { // (1)
    let request: NSFetchedRequest<EntityNameHere> =
        EntityNameHere.fetchRequest() // (2)

    if let items = try?
        CoreDataManager.shared.context.fetch(request) { // (3)
            for item in items { // (4)
                CoreDataManager.shared.context.delete(item) // (5)
            }
        }
    }
}
```

We are not adding this code to any files as this is just for reference. Let's break down each step of batch deletion:

1. Perform everything within the `perform` block.
2. Create an `NSFetchRequest` with your entity.
3. Create an `if let` for your fetch. This can be done inside a `do-catch` block. I used `if let` instead because, in this chapter, I am returning an optional object. If nothing comes back, it will return `nil`. A `do-catch` block would do the same. If you wish to use a `do-catch` block instead, this will have the same outcome.
4. Run a `for` loop on each entity found.
5. Pass the entity to the `delete` method.

Now that we have covered all of the different scenarios, we can now move on to how SwiftUI and Core Data deals with optionals.

SwiftUI and Core Data optionals

If you have used Swift before, you are familiar with `Optionals`, but it is handled much differently when it comes to Core Data. If you uncheck the **Optional** checkbox in **Attributes inspector**, it will still be generated as an optional Swift property. Core Data only cares that properties have values when saved and can be `nil` at other times. There are different ways to handle this, but I found the most elegant way was to create an extension in the `Optional` class. If you look inside the `Utilities` folder, you will see three files — `IntOptional.swift`, `BoolOptional.swift`, and `StringOptional.swift`. Each of these files is structured the same way, but they work for `Int`, `Boolean`, and `String`, respectively.

Let's look at `StringOptional.swift` in a little more detail using the following code:

```
extension Optional where Wrapped == String {
    var _value: String? {
        get {
            return self
        }
        set {
            self = newValue
        }
    }

    public var value: String {
        get {
            return _value ?? ""
        }
        set {
            _value = newValue.isEmpty ? nil : newValue
        }
    }
}
```

Figure 7.12

Now with this file, we can make our code a bit shorter:

1. We do not have to do something like this in our code:

```
Image(self.model.logo ?? "visa-logo")
```

2. Instead, we can do this:

```
Image(self.model.logo.value)
```

3. For Booleans, we use boolValue:

```
if model.isCardVisible.boolValue { }
```

4. Finally, for integers, we use intValue:

```
self.model.age.intValue
```

We will be using these extensions throughout the rest of the chapter. When you see a value for a property, our optional class helps us. Next, we need to set up a file that will allow us to see fake data from Core Data. If we add core data without this, we will either be writing the same helper methods repeatedly or deleting the previews altogether.

Mock account preview service

In SwiftUI, I love having previews, but they will not always work. We can use previews with Core Data. Setting up a mock preview service helps me to create Core Data objects and use them in preview. We need a checking account, a credit account, and at least two cards in this app. We only need two cards, one for each type of account, for now.

Open `MockAccountPreviewService` inside the `Models` folder and add the following code:

```
struct MockAccountPreviewService {  
  
    static let moc = CoreDataManager.shared.context  
  
    // Checking Account  
  
    // Credit Account  
  
    // Visa Card  
  
    // AMEX Card  
  
}
```

We have our `struct` and an instance of the managed object context, which we have named `moc`. We will pass `moc` to each item we need to create with Core Data.

Next, let's add the cards as we will need them for our accounts. Add the following after the `// Visa Card` comment:

```
static var visaCard: Card = {  
    let card = Card(context: moc)  
    card.color = "#000000"  
    card.csv = "999"  
    card.dateCreated = Date()  
    card.expirationDate = Date()  
    card.logo = "visa-logo"  
  
    return card  
}()
```

Here, we are creating an instance of a card and, as stated earlier, we need the `moc` (managed object context) whenever we create Core Data objects. Let's create one for AMEX by adding the following under `// AMEX Card`:

```
static var amexCard: Card = {  
    let card = Card(context: moc)  
    card.color = "#000000"  
    card.csv = "999"  
    card.dateCreated = Date()  
    card.expirationDate = Date()  
    card.logo = "amex-logo"  
  
    return card  
}()
```

We have added our AMEX card and now we can create one checking and one credit account. We are creating two accounts because visually they look different. Swift previews for different states are amazing because you no longer have to run code to check each required state. Let's add the following code after `// Checking Account`:

```
static var checkingAccount: Account = {  
    let account = Account(context: moc)  
    account.acctNumber = "99999"
```

```
account.balance = 9000.00
account.firstName = "Test"
account.lastName = "Last"
account.type = "Checking"

account.card = visaCard

return account
}()
```

Finally, add the following code after // Credit Account:

```
static var creditAccount: Account = {

let account = Account(context: moc)
account.acctNumber = "99999"
account.balance = 99000.00
account.firstName = "Test"
account.lastName = "Last"
account.type = "Credit Card"

account.card = amexCard

return account
}()
```

All of our mock data is complete. Now we can create our View model.

Implementing our View model

We need to create a few methods that will save data to Core Data and pull data out of Core Data. I have started `CreateAccountViewModel` for you. Everything you will add will be focused on things you need to know. Please take a minute to review all of the properties I added for you to use later.

First, let's create a method for creating an account. After the `uiColors` variable, add the following method:

```
func createAccount() {
    let currentAccount = Account(context:
        CoreDataManager.shared.context) // (1)

    CoreDataManager.shared.context.perform { // (2)

        let accountNo = UUID().uuidString.suffix(6) // (3)
        let currentCard = Card(context:
            CoreDataManager.shared.context) // (4)
        currentCard.expirationDate = self.expDate
        currentCard.number = self.ccNumber
        currentCard.cvv = self.cvv
        currentCard.id = String(accountNo)
        currentCard.dateCreated = Date()
        currentCard.color = self.selectedCardColor.hexString
        currentCard.logo =
            self.cardLogos[self.selectedCardType]

        // Add next step here
    }
}
```

We are finally using `CoreDataManager`, so let's break down each step, which we are doing a lot in this method:

1. We are creating an instance of `Account` from Core Data.
2. Next, we use the `perform` block from our context in Core Data.
3. When saving to Core Data, it is best practice to have a unique ID for it, so we use `UUID()` to create a unique ID and save the last six characters of the string. We will also use this same ID as the account number.
4. Finally, we create an instance of `Card` from Core Data.

After we set everything for the card, we then proceed by saving all of the information we need in order to create an account. Now, replace `// Add next step here` with the following:

```
if self.accountTypes[self.selectedAccountType] ==  
    AccountType.creditcard.rawValue { // (1)  
    currentAccount.balance = self.creditLimit // (2)  
} else {  
    currentAccount.balance =  
        Float(self.createRandomBalance()) // (3)  
}  
  
currentAccount.acctNumber = String(accountNo)  
currentAccount.firstName = self.firstName  
currentAccount.lastName = self.lastName  
currentAccount.dateCreated = Date()  
currentAccount.type =  
    self.accountTypes[self.selectedAccountType]  
  
currentAccount.card = currentCard // (4)  
  
// Add next step here
```

Let's look at what is happening in the code:

1. In this `if` statement, we are checking whether the user selected a credit card.
2. If the user has selected a credit card, we take the value the user entered into the form we created.
3. If they select a debit card, we create a random balance. I created a helper method that creates a random balance.
4. After we have set everything up for the new account, we pass it `currentCard`, so it's associated with the account.

As we learned earlier, when saving to Core Data, we must call the `save` method. Replace `// Add next step here` with the following:

```
CoreDataManager.shared.save() // 1  
self.clear() // 2
```

We are done adding to the `createAccount()` method. Now let's look at each line:

1. We use Core Data manager's `save()` method.
2. We clear all of the values used to create an account.

We need a way to see whether there are any accounts created. Let's add the following code:

```
func hasAccounts() -> Bool {
    let request: NSFetchedResultsController<Account> =
        Account.fetchRequest()
    var accounts: [Account] = []
    do {
        for data in try
            CoreDataManager.shared.context.fetch(request) {
            accounts.append(data)
        }
        if accounts.count > 0 { return true }
        return false
    } catch let error as NSError {
        fatalError("Unresolved error \(error),
            \(error.userInfo)")
    }
}
```

In this function, we are fetching Core Data, and if it gets an account back, it returns as `true`, and if not, it returns as `false`. Now it is time for you to challenge yourself.

Core Data challenge

We are done writing methods for Core Data, but if you want to go further and make it so that the app can fetch or delete cards, try taking the time to write the following methods:

```
func fetchCard(with id: String) -> Card?
func fetchCards() -> [Card]?
func deleteCard(with id: String)
func deleteCards()
```

If you get stuck, I have included the code for you inside the `Challenge 1` folder inside the `challenge` folder. Just open `CreateAccountViewModel`.

We will now go through the app and update it to use our View model instead of static text.

The first update we need to make is to ContentView. Let's get started.

Updating ContentView with an environment object

We want our app to show `CreateAccountView()` if there are no accounts created. Open `ContentView` and above the `body` variable, add the following variable:

```
@EnvironmentObject var model: CreateAccountViewModel
```

Earlier, we added an `EnvironmentObject` to our content view in our **Scene Delegate**, and this now means that `CreateAccountViewModel` is accessible in all of the children's views. Next, replace `Text ('Hello Financial App')` with the following:

```
var body: some View {
    ZStack {
        CreateAccountView()

        if model.hasAccounts() {
            AccountListView()
        }
    }
}
```

Here, we are using `ZStack` to stack `CreateAccountView` and `AccountListView` on top of one another. Remember, `AccountListView` was your challenge at the end of the last chapter. If you skipped this challenge, you could grab this file and `CardListRow` from the project starter folder.

If there are no accounts found in Core Data, we will see `CreateAccountView`, and if we do have accounts in Core Data, we will see `AccountListView` instead.

Finally, update `ContentView_Previews` to the following:

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

We just added `environmentObject` to `ContentView` for previews to continue to work.

If you build and run your project, you will now see `CreateAccountView`. Let's work on this next so that we can create accounts and save them to Core Data. To create an account and get everything working, we will be working inside many files before building the project again. The next few pages will guide you to getting everything done, and when you are finished, you will be able to save new accounts to Core Data. Let's get started.

Creating an account view

We have `CreateAccountView` showing up on launch, but we now need to get our UI hooked into Core Data. Open `CreateAccountView` and, above the `body` variable, add the following code:

```
@Environment(\.presentationMode) var presentationMode:  
    Binding<PresentationMode> // (1)  
@EnvironmentObject var model: CreateAccountViewModel // (2)
```

Let's look at the variables we just added:

1. We just created `Environment` and `EnvironmentObject` variables. We will use the `Environment` variable for modals, which allows us to dismiss `CreateAccountView` when done.
2. You are already familiar with `EnvironmentObject` as we just added this inside `ContentView`.

Next, find the following `AccountFormView()` inside the `main` variable and add the following `.frame`:

```
.frame(height: self.model.selectedAccountType == 1 ? 182 : 120)
```

We are setting the height based on `selectedAccountType`. Next, find the `createAccountBtn` variable and add the following methods inside the action:

```
self.onCreateTapped()  
self.presentationMode.wrappedValue.dismiss()
```

Our button will now call the `onCreateTapped()` method and the method to dismiss this view. We have not created our `onCreateTapped()` method yet, so ignore the error.

Finally, we need to add `onAppear` to our `ZStack`. Inside the `body` variable, find the closing curly brace for `ZStack` and add the following:

```
.onAppear {
    self.model.initAccount()
}
```

When `CreateAccountView` appears, it will call the `initAccount` method inside `CreateAccountViewModel`. The `initAccount` method creates an expiration date, a credit card number, and a CVV for our card.

Next, we need to create our `onCreateTapped()` method. Add the following after the last curly brace for the `body` variable:

```
func onCreateTapped() {
    model.createAccount()
}
```

The `onCreateTapped()` method calls the `createAccount` method, and inside this method, we save all of the info from `CreateAccountView` to Core Data.

Finally, we need to update `CreateAccountView_Previews`. Update `CreateAccountView_Previews` with the following command:

```
struct CreateAccountView_Previews: PreviewProvider {
    static var previews: some View {
        CreateAccountView()
            .environmentObject(CreateAccountViewModel())
    }
}
```

We are now finished updating `CreateAccountView` and now we have to update all of the views inside it.

AccountHomeView

Inside `AccountHomeView`, we will create an `ObservableObject` that we will pass to all the children. Whenever this object updates, all of the children will update as well. Add the following code after the struct declaration:

```
@ObservedObject var account: Account
```

Next, update `previews` with the following:

```
struct AccountHomeView_Previews: PreviewProvider {
    static var previews: some View {
        AccountHomeView(account: MockAccountPreviewService.
            checkingAccount)
    }
}
```

At the end of the last chapter, you were faced with a few challenges. One of these challenges was to create a new `CardListRow` from `CardView`. If you did not complete this challenge, you can find the file inside the `Chapter 6/challenge 3` folder.

Next, replace `CardView()` with `CardListRow()`.

Before we pass the account to all of the children, we need to first update the following:

1. Change `HomeHeaderView()` to `HomeHeaderView(account: account)`.
2. Update `AccountSummaryView()` to `AccountSummaryView(account: account)`.

When you are finished with all of these updates, you should see the following:

```
VStack {
    HomeHeaderView(account: account)
    CardListRow(account: account)
    HomeSubmenuView()
    AccountSummaryView(account: account)
}
```

I got rid of some of the things we did in design, so that is why this view is different. I removed the divider and also reordered HomeSubmenuView with AccountSummaryView.

You will have some errors because these views do not currently have the account variable added. To get rid of the errors, add `@ObservedObject var account: Account` to the following files:

1. HomeHeaderView
2. CardListRow
3. AccountSummaryView

When you are done, return to HomeHeaderView and update previews with the following:

```
struct HomeHeaderView_Previews: PreviewProvider {
    static var previews: some View {
        HomeHeaderView(account: MockAccountPreviewService
                      .creditAccount)
    }
}
```

Inside HomeHeaderView, we have a static text view that we need to update in this file. Replace `Text("Hi, Craig Clayton")` with the following:

```
Text("Hi, \((account.firstName.value) \((account.lastName.
    value)\)).
```

Let's open AccountSummaryView and fix some errors in this file:

1. Update previews with the following:

```
AccountSummaryView(account: MockAccountPreviewService.
    creditAccount)
```

2. Delete `var type = "creditcard"` and replace it with the following:

```
    @ObservedObject var account: Account
```

3. Then, update `if`-statement in the `VStack` to the following:

```
    if account.type == AccountType.creditcard.rawValue {  
        creditcard  
    } else {  
        debitcard  
    }
```

We are finished with Account Summary View and now, depending on the account type, we will see a different summary.

Overall, we have a few places that need to be hooked up to our observable object. Currently, we are only capturing a few items. We need to capture and save the account type; we also need to make it so that the user can select the credit card type and the color of their card. Each require minimal code. Before we update these files, there are a couple of files that we will need that require a bit more code. Since I cannot cover every aspect, I prepared these files for you. Inside your project files, you should see a chapter assets folder that contains a number of files:

1. `FormNameView`
2. `CardView`
3. `AccountTypeView`
4. `CreditCardTypeMenuView`
5. `ColorButtonMenu`
6. `AccountListView`

Swap these files out with yours. However, if you feel comfortable setting up these files on your own, by all means take on the challenge. Inside each file, I will leave the static code commented out near the updated code so that you can compare it with your code and see what I did to update the file. When you are done, we will now be able to save new accounts in Core Data. Tap on the account and view the account on the home screen. Try creating an account now and, when you are finished, you should see the following:

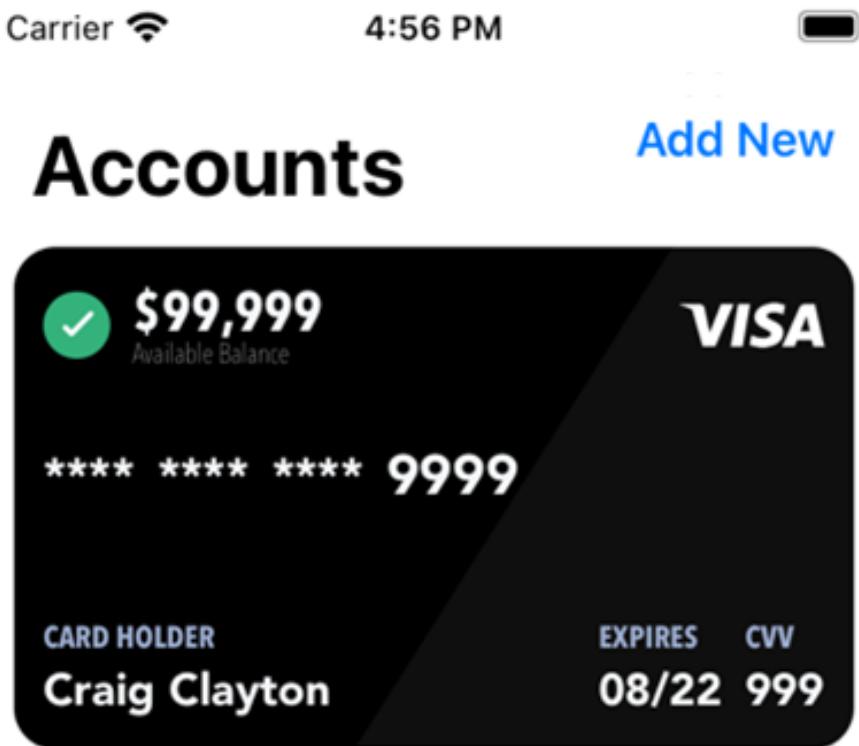


Figure 7.13

If you tap on the card, you will see our dashboard:

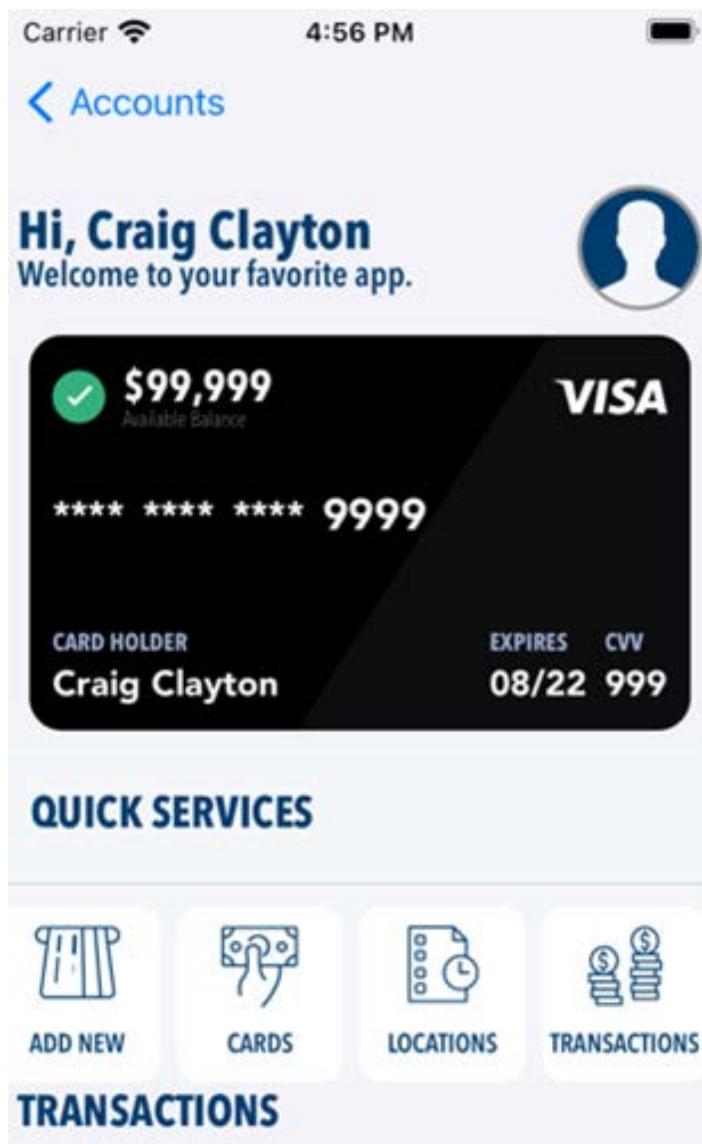


Figure 7.14

There is so much that was done in this chapter and, with regard to certain aspects, I couldn't go into further detail, but you have all of the project files. If you are still unsure, go over the chapter again and it will start to click.

We have a static text view that we need to update in this file. Replace `Text ("Hi, Craig Clayton")` with the following: `Text ("Hi, \$(account.firstName.value) \$(account.lastName.value)")`.

When you are done, return to `AccountHomeView`.

Card view and card list row (challenge)

In the previous chapter, you created `CardListRow`, and since these files are similar, I would like you to update them both. These files are very similar in terms of their setup, except that `CardListRow` uses `@ObservedObject var account: Account`. You want to make sure that you display the currently selected account information.

`CardView` uses `@EnvironmentObject var model: CreateAccountViewModel`, and the information displayed is the information being entered into the form. When we create an account, `CreateAccountViewModel` creates new account information, a credit card number, expiration date, CVV, and so on.

See whether you can set up each file on your own. Take your time, and if you get stuck, you can find a finished version of `CardListRow` and `CardView` in the project files for this chapter.

When you have finished, you should be able to build and run the project. Create a new account and be taken to the Account home screen. From there, you can create another account or view other created accounts. If you are having any problems at this point that you can't figure out, check out the completed files.

Summary

In this chapter, we learned about Core Data and how to create a Core Data model. We also learned how to make Core Data an environment object and created observable objects. Finally, we went through all of the files and updated the prototype app to work with Core Data.

In the next chapter, we will create a shoe **Point of Sale (POS)** system using a number of Core Data advanced techniques such as CloudKit.

8

Shoe Point of Sale System – Design

When SwiftUI was announced and I finally decided to write again, this app idea was number one on my list. I feel like you can really see the power of reactive programming in an app that has a lot going on on one screen. I am also a huge fan of shoes, so it was a no-brainer for me.

In this chapter, we will be building a **Point of Sale (POS)** system for shoes. We are going to ignore multi-tasking because my thinking for this app is that it would be used on company iPads and at the cash registers used for checking out customers' items only. I wanted to focus on particular features, instead of this app being universal in application. In the last two chapters, the app we built was more adapted for use with multi-tasking. Here, we will be building an app that is connected to CloudKit. You will be designing and coding the shoe app, and you will also be able to see the store's inventory. Since this app will be more about data, I limited the design so that we can focus on the good stuff. I cover a lot of design in this book and this will be the one chapter that will be really light on design.

In this chapter, we will be working with the following:

- Creating a POS system for the iPad
- Working with LazyHGrid
- Creating a custom split-view iPad app

Before we jump into the design, let's take some time to get familiar with what we are building in this chapter. As I said earlier, we do not have a lot to design in this chapter, but we are going to cover a lot of information between this chapter and the next. The design we are going to be working on is as follows.

We have two main sections: the products grid view and the shopping cart view. There are definitely other things we could add to the design, but we are keeping it simple. Let's get started by jumping in and getting our shell built out first.

Displaying the app container

We are going to approach the design of this app a little differently than we have before. Since we are creating our own custom split view, we are going to set up each side first, then add in the other elements after.

Let's look at what we are going to create first:



Figure 8.1

We have two sides to our app, along with a couple of icons. Let's get started by creating our custom split view first.

Creating a products header view

We are going to create a custom split view starting with the view on the left, which is what will hold our products. Create a new SwiftUI file named `ProductsHeaderView` and save it in the `Products` folder inside `Supporting Views`.

In the `body` variable, replace the `Text` view with the following:

```
HStack {  
    Spacer()  
    Text("KICKZ STORE")  
        .custom(font: .heavy, size: 30)  
    Spacer()  
  
    // Add next step here  
}  
.frame(height: 60)  
.background(Color.white)  
.offset(x: -1)  
.border(Color.baseLightGrey, width: 1)
```

Next, we need to add our filter button. Replace `// Add next step here` with the following:

```
Button(action: { }) {  
    Image(systemName: "line.horizontal.3.decrease.circle")  
        .font(Font.system(size: 24, weight: .thin))  
}  
.buttonStyle(PlainButtonStyle())  
.foregroundColor(Color.black)  
.padding(.trailing, 15)
```

Our header is all set – we have a header that has a button on the right side. Next, let's create our products container.

Creating our main products container

We now just need to use our header and create a container for our products. Create a new SwiftUI file called `ProductsView` and save it to the `Products` folder which is located inside of the `Supporting Views` folder. Next, replace `Text ("")` with the following:

```
 VStack {  
     ProductsHeaderView()  
     Text("Products go here")  
     Spacer()  
 } .background(Color.baseLightGrey)
```

We just created a `VStack` container with a `baseLightGrey` background and added a `Text` placeholder for now, with a `Spacer()` below it. Pretty easy so far; let's move on to the cart side of the app.

Creating the shopping cart header

As we did for the products side, we need to create a header with an icon on the right. Create a new SwiftUI file called `CartHeaderView` and save it to the `Cart` folder inside the `Supporting Views` folder. Then, replace `Text ("Hello! World")` with the following:

```
 HStack {  
     Spacer()  
     Text("CART (0)")  
         .custom(font: .demibold, size: 24)  
     Spacer()  
     Image(systemName: "trash")  
         .font(Font.system(size: 24, weight: .thin))  
 }  
 .padding(EdgeInsets(top: 0, leading: 15, bottom: 0, trailing:  
 15))  
 .frame(height: 60)  
 .background(Color.white)  
 .border(Color.baseLightGrey, width: 1)  
 .offset(x: -1)
```

We have an `HStack` container that contains a `Text` view and an `Image`. It's a pretty simple layout here. Next, we want to add the cart total display.

Creating a cart total display

Next, we are going to create the cart total display. Create a new SwiftUI file called `CartTotalView` and save it to the `Cart` folder, which is inside the `Supporting Views` folder. Next, replace `Text ("Hello, World!")` with the following:

```
HStack {  
    ZStack {  
        RoundedRectangle(cornerRadius: 10)  
            .foregroundColor(.baseLightBlue)  
            .frame(height: 70)  
        // Add next step here  
    }  
    .padding(EdgeInsets(top: 0, leading: 15, bottom: 0,  
        trailing: 15))  
}  
.frame(height: 93)  
.background(Color.clear)
```

We are using an `HStack` container as our main container, and inside this we have a `ZStack` container. Our `ZStack` container has a `RoundedRectangle` that is used as our cart display background. All we need to add now is our `Text` views. Let's add these next by replacing `// Add next step here` with the following:

```
HStack {  
    Text("Total:")  
    .foregroundColor(.white)  
    .custom(font: .demibold, size: 28)  
    Spacer()  
    Text("$9999.99")  
        .foregroundColor(.white)  
        .custom(font: .bold, size: 47)  
}  
.padding(EdgeInsets(top: 0, leading: 15, bottom: 0, trailing:  
    15))
```

We are using an `HStack` container so that we can add both `Text` views with a `Spacer` in between them. Our cart view is done, so let's display it next.

Displaying our cart view

Now that we have our cart view elements created, let's display them with our products container that we created earlier. Create a new SwiftUI file inside the `Views` folder called `CartView` and replace `Text ("Hello, world!")` with the following:

```
 VStack {  
     CartHeaderView()  
     Text("Cart goes here")  
     Spacer()  
     CartTotalView()  
 }  
 .background(Color.baseWhite)  
 .frame(width: 417)
```

We are using a `VStack` component as our main container and inside, we have our newly created `CartHeaderView`, a `Text` view, `Spacer()`, and `CartTotalView()`. We are setting the width of the cart, but our products will be displayed bigger on larger devices.

Now, we just need to put everything together.

Viewing our custom split view

We now just need to display what we have been working on. Open `ContentView` and replace `Text ("Hello, world!") .padding()` with the following:

```
 HStack(spacing: -5) {  
     ProductsView()  
     Spacer()  
     CartView()  
 }
```

Now, if you open Previews, you will see that our device is in portrait mode. Update caps with the following:

```
ContentView().previewLayout(.fixed(width: 1024, height: 768))
```

Now, you will see the following:

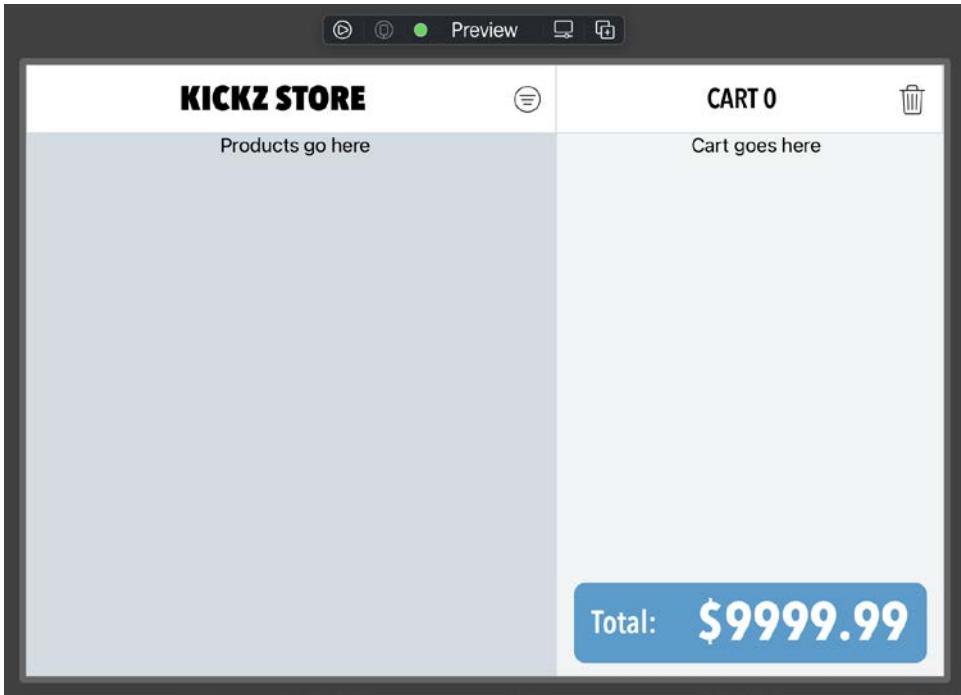


Figure 8.2

We have completed the basic structure of our app. We now need to create the products and also add all of the views that we need for the cart.

Creating the products

Now that we have completed our basic structure, let's move on to getting the products to display next. Let's take a look at what we need to create for our products section:



Figure 8.3

For each product, we have the product image, the name of the shoe, and the price. Let's create this now by creating a new SwiftUI file called `ProductView` and save it to the `Product` folder, inside the `Supporting Views` folder. Next, open `ProductView` and replace `Text("Hello, world!")` with the following:

```
ZStack(alignment: .bottom) {
    Rectangle()
        .foregroundColor(Color.white)
        .frame(minWidth: 200)
        .frame(height: 110)
        .cornerRadius(10)
    VStack(spacing: 0) {
        Image("shoe-1")
            .resizable()
            .frame(width: 188, height: 100)
        VStack(spacing: -8) {
            Text("Nike Fury")
                .custom(font: .demibold, size: 18.0)
            Text("$125.00")
                .foregroundColor(.baseLightBlue)
                .custom(font: .bold, size: 29.0)
        }
    }
}
```

```

    .frame(height: 100)
    .padding(.bottom, 25)
    .padding(.horizontal, 5)
    .background(Color.clear)
}

```

ProductsContentView

We now just want to display our products. One thing that's different about this section is that we want to make sure that, depending on the width of the area, we will display as many items as can fit horizontally until it wraps. If we want to accomplish this, then we will need to code this section slightly differently.

We will look at how we can make our grid adapt to any size we throw at it. Let's take a look at the products view section one more time, but this time, we'll look at the previews on a couple of different devices:

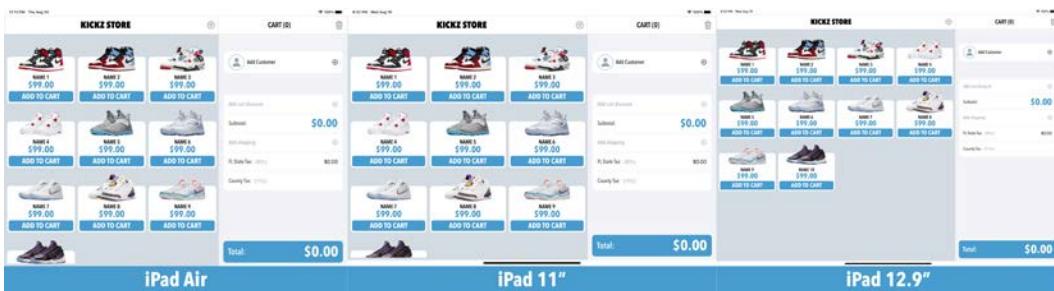


Figure 8.4

Alright, create a new file named `ProductsContentView` and save it to the `Products` folder inside the `Supporting Views` folder. Replace `Text` ("Hello, World!") with the following:

```

ScrollView(showsIndicators: false) {
    LazyVGrid(columns: [GridItem(.adaptive(minimum: 200),
    spacing: 10)], spacing: 34) {
        ForEach(0..<20) { _ in
            ProductView()
        }
    }
    .padding(.top, 20)
    .padding(.horizontal, 10)
}

```

We are creating a horizontal scroller that uses `LazyVGrid`. Our grid gets a `.adaptive` item set to a minimum of 200 pixels. This means it will make the product item bigger if it needs to, but the smallest it will be is 200 pixels. Our products are now flexible in size, along with showing more items depending on the size of the iPad display. Now that our products side is complete, let's move over to the cart side.

Finally, in `ProductsView`, replace `Text ("Products go here")` with `ProductsContentView()`. If you build and run the project, you will see the following:

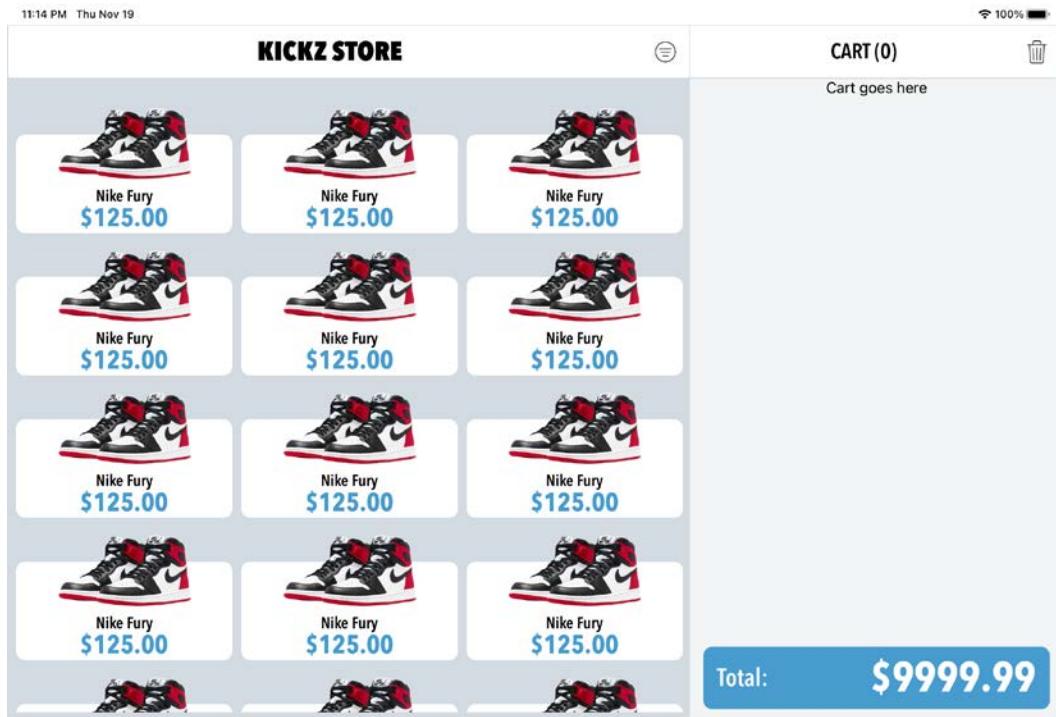


Figure 8.5

We can now move on to the product details.

Product details

When you tap on a product, we want to display a custom modal over the products section. The modal will display the selected product as well as the sizes for each shoe and the quantity in stock. In this chapter, we are going to focus strictly on the design. In the next chapter, we will add the functionality to view ProductDetail. For now, just get it to match the design and we will use it in the next chapter. Let's look at the design:

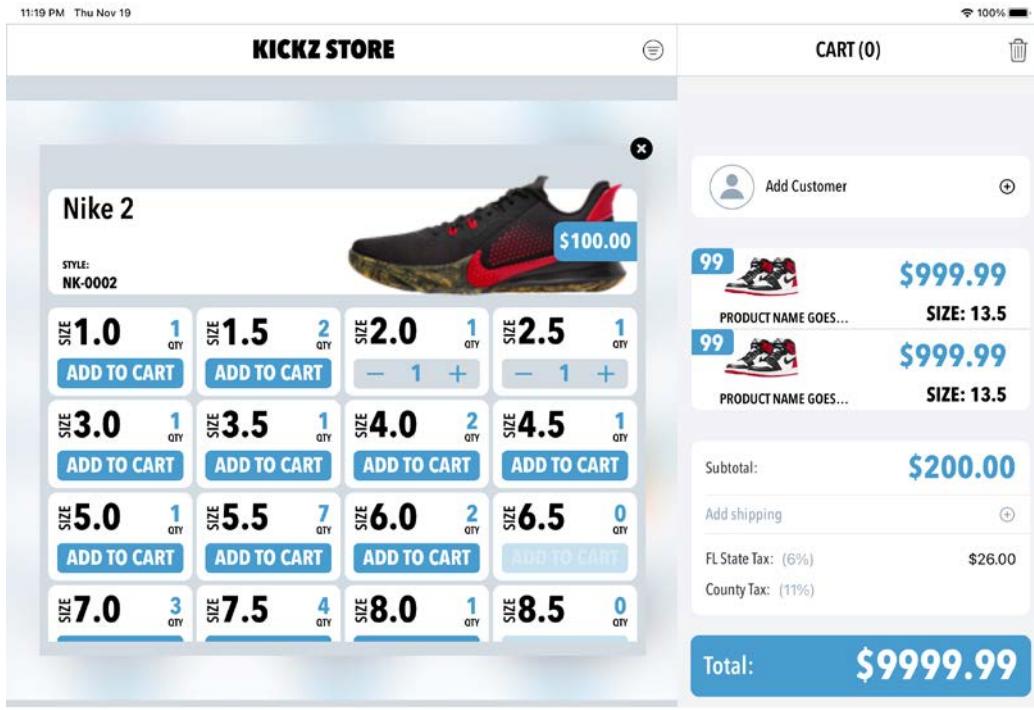


Figure 8.6

You will actually be creating this view on your own, but before you dig in, we will work on the custom buttons together. Let's work on the **ADD TO CART** and stepper buttons next. We will not be using Product Details until the next chapter.

Creating custom buttons for our product details view

Let's take a closer look at what the size view will look like:

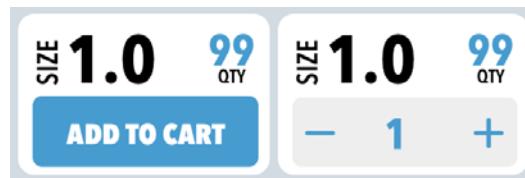


Figure 8.7

The functionality for this is pretty simple – you will see **ADD TO CART** first, and when you tap it, you will see the stepper view. We will cover the coding functionality in detail in the next chapter. All you need to know for now is that **ADD TO CART** will appear on top of the stepper view. Since the stepper view is at the bottom, let's work on that first.

Custom stepper view

The reason we are creating a custom stepper is because Apple's `UIStepper` does not allow us to do much customization. We will drive this button using state and hopefully, in the future, we will have more control over the look and feel of Apple's stepper.

Create a new SwiftUI file named `StepperView` and save it to the `Products` folder inside the `Supporting Views` folder. Next, replace `Text ("Hello, world!")` with the following `ZStack`:

```
ZStack {
    Rectangle()
        .foregroundColor(.white)
        .cornerRadius(6)
        .frame(height: 34)

    // Add next step here

    Text("1")
        .custom(font: .bold, size: 27)
        .foregroundColor(.baseLightBlue)
}
```

We are using a `ZStack` component as our main container, and inside of the `ZStack` container, we first added a `Rectangle()` for our custom background and a `Text` view. The `Text` view will display the number of items added for the product. Now, we just need to add our buttons, so replace `// Add next step here` with the following `HStack`:

```
HStack { // (1)
    Button(action: { }) { // (2)
        HStack {
            Image(systemName: "minus")
                .font(Font.system(size: 24, weight: .medium))
                .foregroundColor(.baseLightBlue)
        }
        .frame(width: 50, height: 34)
    }

    Spacer() // (3)

    Button(action: { }) { // (4)
        HStack {
            Image(systemName: "plus")
                .font(Font.system(size: 24, weight: .medium))
                .foregroundColor(.baseLightBlue)
        }
        .frame(width: 50, height: 34)
    }

}
.frame(height: 34)
.background(Color.baseLightGrey)
.cornerRadius(6)
```

We have added the last bit of code to our custom stepper. Let's look at what we just added (refer to the numbered steps in the preceding code):

1. We are using an `HStack` container as our container.
2. We add a button that will represent the minus button, using a system font.
3. Here, we have a spacer so that we can push each button to each edge of the view.
4. Finally, we have another button, which represents our plus button.

We are done with our custom stepper button. Let's create an add to cart button next.

Creating an add to cart button

Next, we need to create the add to cart button for our product view. Create a new SwiftUI file named ShoePOSButton and save it to the Products folder inside the Supporting Views folder. Above the body variable, add the following variable:

```
let title: String
```

Next, replace `Text ('Hello, world!')` with the following `ZStack`:

```
ZStack {
    Rectangle()
        .foregroundColor(.baseLightBlue)
        .cornerRadius(6)
        .frame(height: 34)
    Text(title)
        .custom(font: .bold, size: 24)
        .foregroundColor(.white)
}
```

We have a `ZStack` container, which just contains a `Rectangle` and a `Text` view. We could also make this a button, but instead, we will use a tap gesture just to mix it up. If I were building this for a real app, I would just make it a button instead, but I wanted to show some variety. Finally, update `previews` with the following:

```
struct ShoePOSButton_Previews: PreviewProvider {
    static var previews: some View {
        ShoePOSButton(title: "ADD TO CART")
    }
}
```

Now that you have the two buttons that you need for the `Size` item, it is time for your next challenge.

SizeCartItemView

Before you get into your next challenge, I wanted to work with you on one more view. This view will be used inside your product detail view, which is what you will create in your next challenge. But before we work on that view, we need to create a new one. Create a new SwiftUI file named `SizeCartItemView` and save it to the `Cart` folder inside the `Supporting Views` folder.

Next, replace `Text ("Hello, world!")` with the following `VStack`:

```
 VStack {  
     // Add next step here  
 }  
 .frame(height: 100)  
 .padding(.horizontal, 10)  
 .background(Color.white)  
 .cornerRadius(10)
```

We are using a `VStack` component as our main container, which has `height` set to 100 along with some horizontal padding. Now, replace `// Add next step here` with the following:

```
 VStack(spacing: 0) { // (1)  
     HStack {  
         HStack(spacing: -6) { // (2)  
             Text("SIZE")  
                 .rotationEffect(.degrees(-90))  
                 .custom(font: .demibold, size: 18)  
  
             Text("13.0")  
                 .custom(font: .bold, size: 42)  
         }.offset(x: -5)  
  
         Spacer() // (3)  
  
         VStack(spacing: -8) { // (4)  
             Text("99")  
                 .custom(font: .bold, size: 27)  
                 .foregroundColor(.baseLightBlue)
```

```
Text("QTY")
    .custom(font: .demibold, size: 12)
}

ShoePOSButton(title: "ADD TO CART") // (5)
.opacity(0.3)

Spacer()
}
```

Alright, we just added a `VStack` container that has a lot inside it. Let's break everything down:

1. We are using a `VStack` component with 0 spacing as our main container.
2. Inside of our `VStack` container, we have an `HStack` container, which is a container for our size and quantity values. We use this `HStack` container to display the size text along with the size value. It has a slight offset of -5.
3. We are utilizing a `Spacer` so that we can separate the size and quantity.
4. In this `VStack` container, we display the quantity value along with the quantity text.
5. Finally, we add `ShoePOSButton`. In the next chapter, we will update this to work with the custom toggle we created earlier.

Now that `SizeCartItemView` has been created, you can move on to your challenge.

Product details challenge

You will be creating the rest of the product details view on your own. The blurred background is something we did back in *Chapter 4, Car Order Form – Design*, so you can reuse this file or create a new one if you remember how to do this. You can find the design specs in the folder for this chapter, so please review them and have fun. Remember, the point of the design specs is for you to practice replicating a design as if you had to do this for a job. Once you've replicated it, duplicate the project and play around with the design however you see fit.

Creating the shopping cart

Now, that the products side is complete, let's look at what we need to do for the cart view. We have a couple of things to do for our shopping cart. Let's take a look before we dig in:

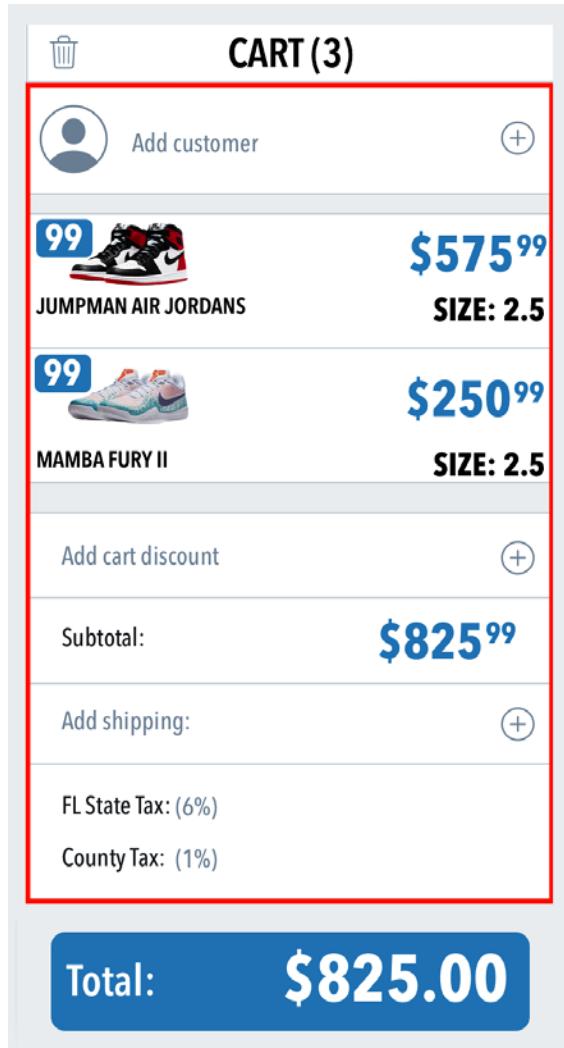


Figure 8.8

We need to update our cart so that we can take products and also see the total value of the transaction. First, we will create the cart item, and then we will work on the cart content after that. Let's get started.

Designing the cart item view

We are going to work on the cart item view as we will need this in the next step. Create a new SwiftUI file named `CartItemView` and save it to the `Cart` folder, which is inside the `Supporting Views` folder. Update previews to the following:

```
CartItemView()
    .previewLayout(.fixed(width: 375, height: 100))
```

Next, replace `Text ("Hello, world!")` with the following `HStack`:

```
HStack {
    HStack(spacing: 10) {
        // Add next step here
    }
    Spacer()

    // Add last step here
}.offset(x: 0)
```

We are using an `HStack` container as our main container. Inside the `HStack` container, we add another `HStack` container and a `Spacer()`. Next, replace `//Add next step here` with the following `VStack`:

```
 VStack(alignment: .leading) { // (1)
    ZStack(alignment: Alignment(horizontal: .leading, vertical:
        .top)) { // (2)

        Image("shoe-1")
            .resizable()
            .frame(width: 107, height: 57)

        ZStack {
            Rectangle()
                .fill(Color.baseLightBlue)
                .frame(width: 40, height: 30)
                .cornerRadius(5, corners: [.topRight,
                    .bottomRight])
            Text("99")
        }
    }
}
```

```

        .custom(font: .bold, size: 26)
        .foregroundColor(.white)
        .offset(y: 2)
    }
    .offset(x: -10, y: -5)
}
.padding(.leading, 10)
Text("PRODUCT NAME GOES HERE") // (3)
    .custom(font: .demibold, size: 16)
    .padding(.leading)
}

```

We added the majority of our requirements to our cart item view in this step, so let's look at what we just added:

1. We are using a `VStack` container as our main container.
2. Next, we have a `ZStack` container inside of the `VStack` container. It contains the product image and cart quantity.
3. Lastly, we have a `Text` view at the bottom of our `VStack` container.

The left side of our `CartItemView` is complete now, so let's add the right side. Replace `// Add last step here` with the following code:

```

VStack(alignment: .trailing, spacing: 0) { // (1)
    Text("$999.99") // (2)
        .custom(font: .bold, size: 36)
        .foregroundColor(.baseLightBlue)
        .offset(y: 5)
    Spacer()
    Text("SIZE: 13.5").custom(font: .bold, size: 23)
}
.padding(.trailing, 10)

```

Let's break down what we just added:

1. We are using a `VStack` component as our main container. The alignment is set to `.trailing` and the spacing is set to 0.

2. Inside the `VStack` container, we have two `Text` views separated by a `Spacer` . `Spacer` pushes the two `Text` views to the top and bottom of the `VStack` container.

Our cart item view is complete, so we can now move on to our cart content view.

Cart content view

Our cart content view is broken up into three main sections. We have the header or **Add Customer** section, and then we have the **Cart** section, where we will list each cart item view. Finally, we have the section containing the subtotal, tax, and so on. Create a new SwiftUI file named `CartContentView` and save it to the `Cart` folder inside the `Supporting Views` folder. Next, replace `Text ("Hello, world!")` with the following `Form`:

```
Form {  
    // Add section here  
}
```

When working with forms, we can utilize `Section` to separate each section of the shopping cart. Let's add our first section next.

Creating the Add Customer section

We will create our first section by replacing `// Add section here` with the following:

```
Section { // (1)  
    HStack { // (2)  
        Image(systemName: "person.circle")  
            .foregroundColor(.baseMediumGrey)  
            .font(Font.system(size: 54, weight: .ultraLight))  
        Text("Add Customer")  
            .custom(font: .medium, size: 18)  
        Spacer()  
        Image(systemName: "plus.circle")  
    }  
    .frame(height: 58)  
    // Add next section here
```

Let's break down what we just added:

1. We wrap everything in a section.
2. Inside of our `Section`, we are using an `HStack` container to hold an `Image` and `Text` with another `Image` that's separated by a `Spacer`.

Now that our first section is complete, let's add the next section.

Cart item view

Our next section is where we will display our cart items. When the app initially opens, this section will not be visible. As we add products, it will add them as new items one at a time. Replace `// Add next section here` with the following code:

```
Section { // (1)
    ForEach(0..<3) { _ in // (2)
        CartItemView()
    }
}
.frame(height: 80)
// Add last section here
```

Now that our new code is added, let's break it down:

1. We are using `Section` as our main container.
2. Inside `Section`, we are using a `ForEach` loop to create three `CartItemView()` views as placeholders. In the next chapter, we will update this code so that it works with our shopping cart.

Our second section is now complete, so now we can move on to our final section and the last bit of design we need to do for this app.

Subtotal and taxes

We just created two sections together and only have one more to complete. I like to think of this section as the footer since it's at the bottom of our `Form`. Let's jump in and get started. Replace `// Add last section here` with the following:

```
Section {
    // Add next step here
}
```

We have set up our main container as a `Section`; now, let's move on to the next step. Replace `// Add next step here` with the following:

```
HStack { // (1)
    HStack(spacing: 10) { // (2)
        Text("Subtotal:")
            .foregroundColor(.baseDarkGrey)
            .custom(font: .medium, size: 18)
    }
    Spacer()
    Text("$999.99")
        .custom(font: .bold, size: 36)
        .foregroundColor(.baseLightBlue)
}
// Add next step here
```

We just added a new `HStack` container, so now let's break it down:

1. We are using an `HStack` container as our main container.
2. Inside the `HStack` container, we have another `HStack` container holding a `Text` view that displays the `Subtotal` label.
3. After the `Spacer`, we have another `HStack` container containing a `Text` view that displays the subtotal value.

We have completed our first row, so let's move on to the next row and display the shipping costs. Replace `// Add next step here` with the following:

```
HStack { // (1)
    Text("Add shipping") // (2)
        .custom(font: .medium, size: 18)
        .foregroundColor(.baseMediumGrey)
    Spacer()

    Button(action: { }) { // (3)
        Image(systemName: "plus.circle")
            .foregroundColor(.baseMediumGrey)
    }.buttonStyle(PlainButtonStyle())
}

```

```
// Add next step here
```

Again, we have added another `HStack` container, so let's discuss what we just added:

1. Our main container is an `HStack` container.
2. Inside of our `HStack` container, we have a `Text` view to display our `Add Shipping` label.
3. After the `Spacer`, we have a `Button`.

We just need to add one more chunk of code. Add the last bit by replacing `// Add next step here` with the following:

```
 VStack(alignment: .leading) { // (1)
    Spacer()
    HStack { // (2)
        HStack {
            Text("FL State Tax:")
                .foregroundColor(.baseDarkGrey)
                .custom(font: .medium, size: 18)
            Text("(6%)")
                .foregroundColor(.baseMediumGrey)
        }
        Spacer()
        Text("$0.00")
    }
    Spacer()
    HStack { // (3)
        Text("County Tax:")
            .foregroundColor(.baseDarkGrey)
            .custom(font: .medium, size: 18)
        Text("(11%)")
            .foregroundColor(.baseMediumGrey)
    }
    Spacer() // (4)
}.frame(height: 80)
```

We just added a lot of code, so let's talk through it now:

1. We are using a `VStack` container as our main container with its alignment set to `.leading`.
2. After the `Spacer`, we added an `HStack` container, which displays our state tax label and the state tax amount.
3. Then, after another `Spacer`, we use another `HStack` container to display our county tax label as well as the county tax value.

Note that we are using a `Spacer` in between each one and that is because we want it to be evenly spaced across the device, no matter how big the screen. We just need to make an update inside `CartView` in order to see what we have created.

Open `CartView` and replace `Text ("Cart goes here")` with `CartContentView()`. If you build and run it, you will now see that our cart view is complete, and we can now move on over to the next chapter and learn about CloudKit integration.

Summary

In this chapter, we built a custom split screen that displays products on the left and a shopping cart on the right. Even though we have our app locked in landscape mode, we could easily use it on all device displays.

We also learned how to structure a shopping cart design using forms. In the next chapter, we will take this app to the next level by getting our app to work with CloudKit data, as well as building out our shopping cart functionality.

9

Shoe Point of Sale System – CloudKit

In this chapter, we are going to work with CloudKit to manage our data. In order to work through the examples in this chapter, you will need to have a developer account. If you do not have one, you will not be able to do much in this chapter. We will work with CloudKit without using Core Data. We do not need Core Data, so there is no reason for us to add that extra layer. We set up our **Point of Sale (POS)** system design in the previous chapter, and now we will jump in and start learning about how CloudKit works.

In this chapter, we will be working with the following:

- CloudKit basics
- CloudKit Dashboard
- Manually setting up CloudKit

There is a lot to cover in this chapter, so we will take our time and make sure you fully understand what you are doing. We need to understand the basics of CloudKit first, and then we can start setting up our CloudKit helpers.

Understanding the basics of CloudKit

If you have not used CloudKit before, then you might be more familiar with the name most iPhone users are accustomed to, which is iCloud. iCloud allows applications to synchronize data across devices using Apple servers. It provides three basic services:

- **Key-value storage:** Stores single values
- **Document storage:** Stores files
- **CloudKit storage:** Stores structured data in public and private databases

We will be using the third option in this chapter – CloudKit storage.

In CloudKit, the structure is a bit different than you might be accustomed to. Apple puts all of a user's data into a container. The most common structure is having a single container per app, but your app can use multiple containers across multiple apps. In CloudKit, a container is represented as `CKContainer`. You can access the default container by doing the following:

```
let container = CKContainer.default()
```

It is recommended to use a custom container instead of using the default. By creating your own custom container, you can share your CloudKit databases between your apps and extensions, or even between your app on multiple platforms, such as iOS, watchOS, and macOS.

Inside the container, there are three different types of databases: private, public, and shared. You do not actually create a database; every container comes with those three by default. In CloudKit, databases are represented as `CKDatabase`. Let's look at the differences between each database type:

- **Private:** Used when data should only be accessible to the user. Best to use when saving a user's private data.
- **Public:** Used when data should be accessible to every user running the app. The data used in this database can be created in a custom app or CloudKit Dashboard. Even though this database is public, you can also restrict access to records.
- **Shared:** Used when a user wants to share with others.

Turning on CloudKit manually

Before we go any deeper into CloudKit, let's take the time to set up CloudKit for this chapter.

First, you will need to create a container name. Your container name cannot be changed, so make sure you get your pattern correct before you add it to your project. For example, all my test apps get something like this:

```
iCloud.io.designtoswiftui.test.HelloCloudKit
```

I use tests as a way to make sure I stick to a format. I am not saying you should use a test, but I just want you to think about using a naming convention with your domain that you will stick to.

Now, we need to add our newly created container name to our project. In your project, select the project and make sure you select the target:

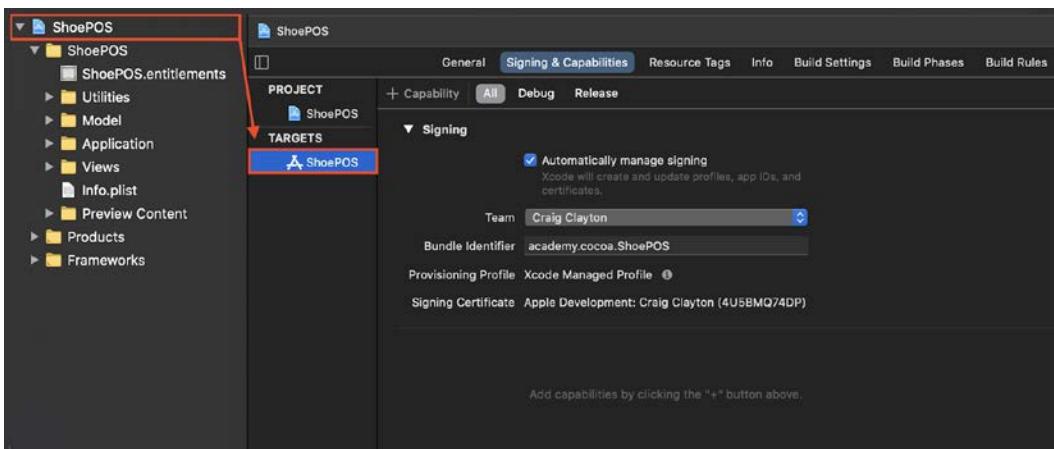


Figure 9.1

Now, select **Signing & Capabilities**, and then click on the **+** Capability button here:

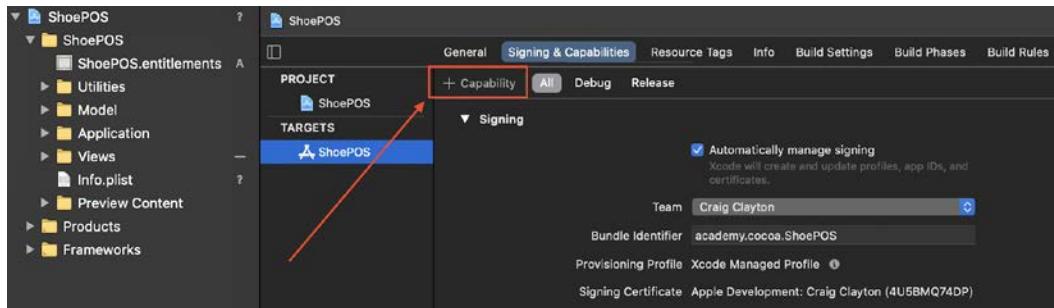


Figure 9.2

When the modal appears, type Cloud and double-click on CloudKit:

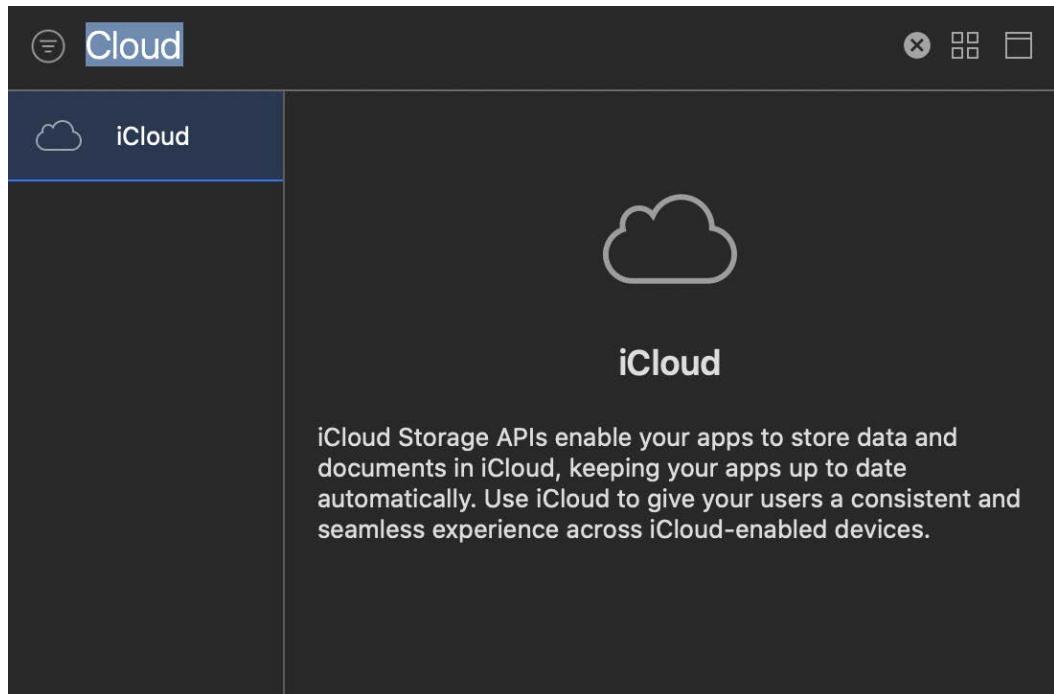


Figure 9.3

When you are done, you will see the following:

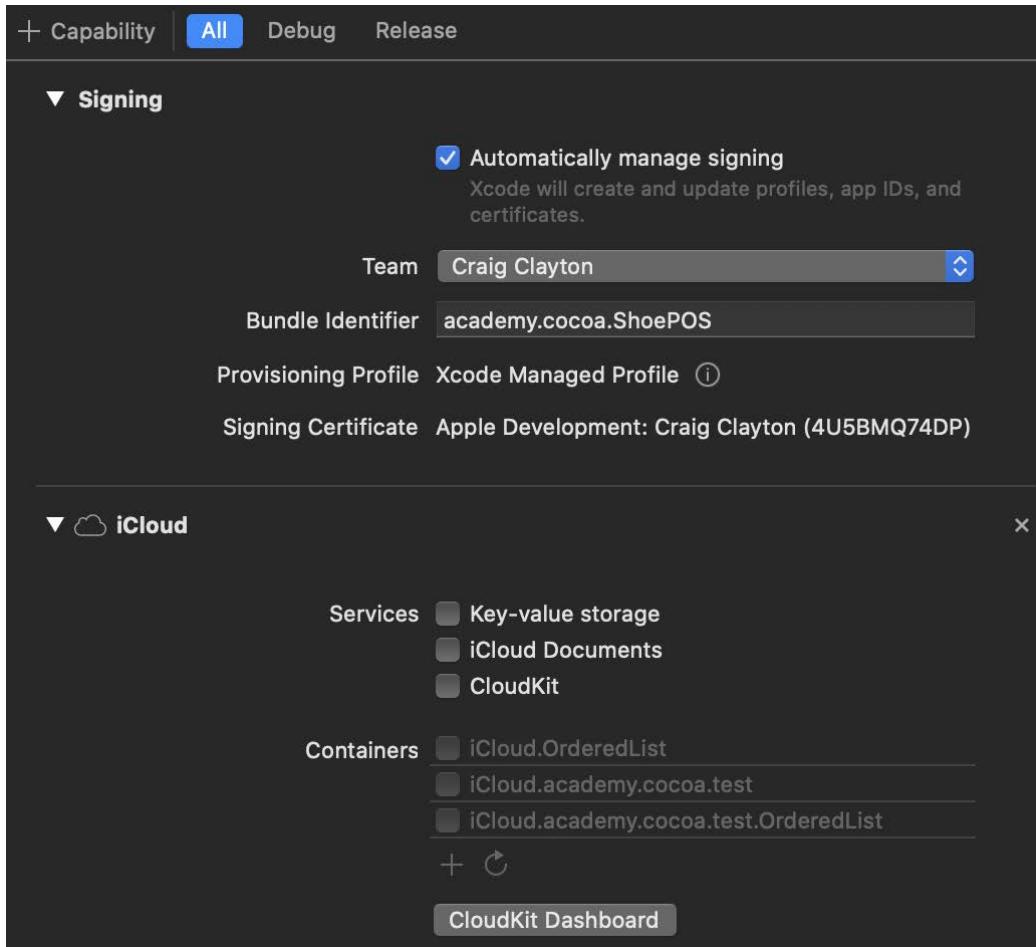


Figure 9.4

When working with a CloudKit project, make sure you change your **Team** and **Bundle Identifier** properties. The Bundle identifier must be unique so please make sure that you use a unique identifier. Typically, you would use your reverse domain name since domains are unique. If you do not have a domain then buy one and use that but if you try to submit an app with a domain already used you will have problems. Next, click the **CloudKit** checkbox and then hit the **+** button under **Containers**. Then, you will see a box appear:

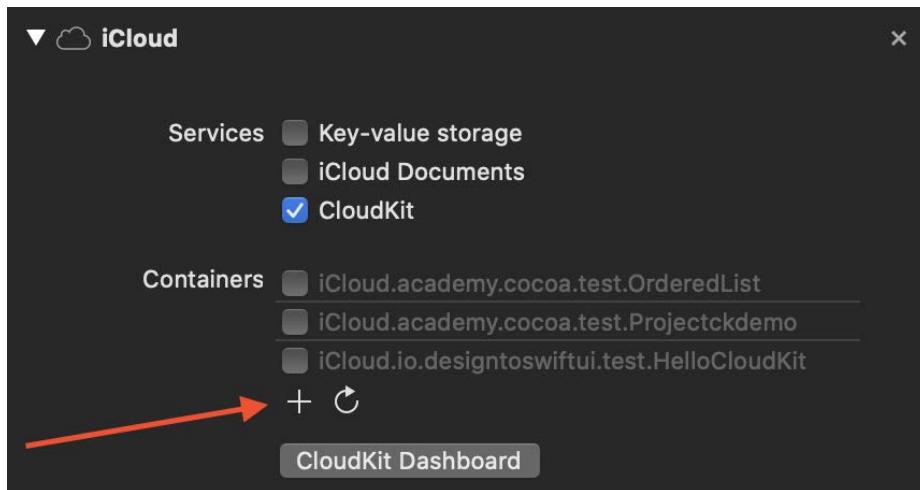


Figure 9.5

In this box, add the container name you created and hit **OK**:



Figure 9.6

When we activate CloudKit, Xcode automatically activates **Push Notifications**. You will see it appear underneath the **iCloud** section:

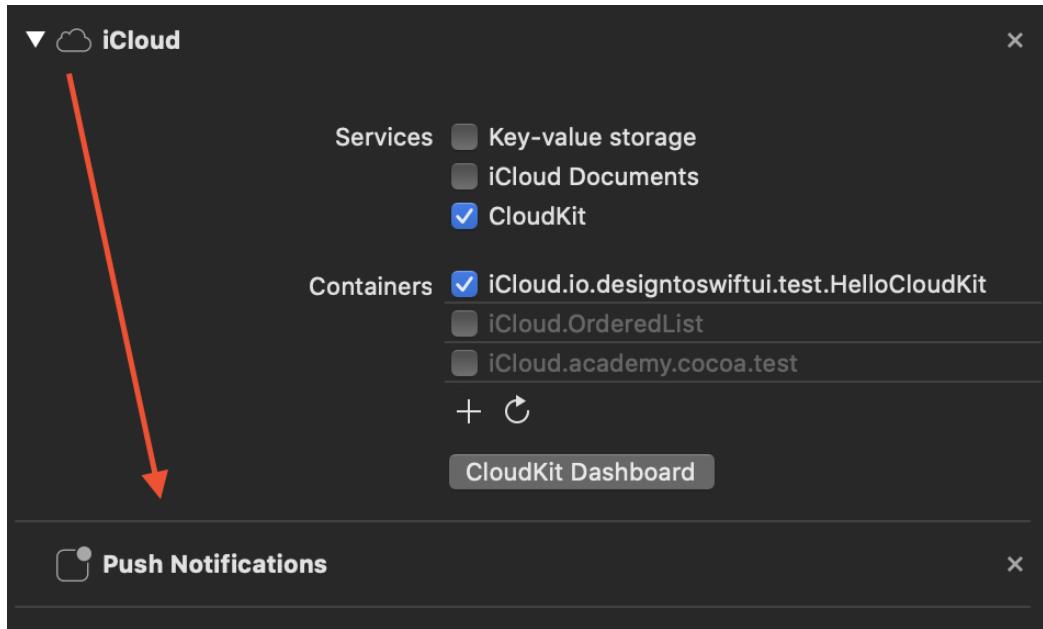


Figure 9.7

Next, we need to add another capability. Click on the **+** **Capability** button again and this time search for **Background Modes** and double-click on it. You will now see the following:

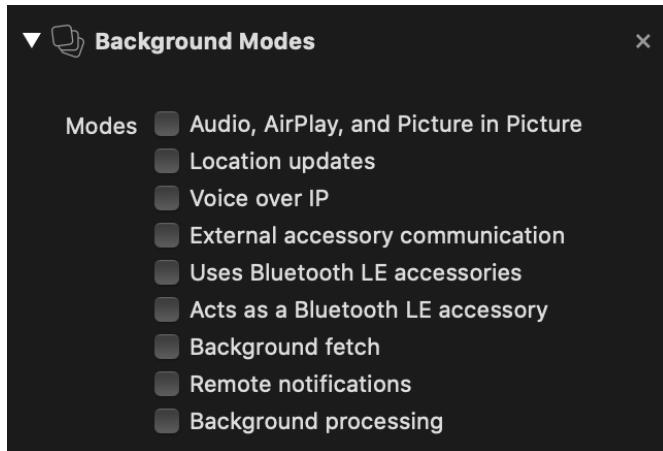


Figure 9.8

Go ahead and enable both **Background fetch** and **Remote notifications**.

Let's take a few minutes to just mess around with CloudKit first before we dig a bit deeper.

Creating our first CloudKit record

Before we dig into all the details, let's first just take the time to see how we can create a record. For now, think of a record as a type. We will go into this in more depth a bit later in this chapter. Let's say we want to create a shoe record. Let's see how we can accomplish that now.

Open ShoePOSApp.swift and, under `import SwiftUI`, add `import CloudKit`. Now, above the `body` variable, add the following:

```
init() {
    // Add next step here
}
```

Next, replace `// Add next step here` with the following:

```
let container = CKContainer(identifier: "//add your container
name here") // (1)
let db = container.publicCloudDatabase // (2)
let recordID = CKRecord.ID(recordName: "shoe-pos-test") // (3)
```

Alright, let's break down each bit of code we added line by line.

Earlier, you created a container name and added it to your project for iCloud. If you do not add the “iCloud” in front of the name, CloudKit will automatically add it. Now, grab that same name and put it into your identifier. As I stated earlier, we could use the default container, but custom containers provide us with more features.

Every container comes with three databases. We are going to use the public database.

Every record needs a unique identifier. We can either have CloudKit create one for us or we can create it ourselves. In this case, we will create a custom one so that we can see it on the database.

Alright, now we need to create our first record. After `recordID`, add the following:

```
let record = CKRecord(recordType: "Shoe", recordID: recordID)
// (1)
let imageURL = Bundle.main.url(forResource: "nike-mamba-fury",
```

```
withExtension: "png")! // (2)
record["name"] = "Mamba Fury" as NSString // (3)
record["price"] = 125.99 as Double // (4)
record["image"] = CKAsset(fileURL: imageURL) // (5)
```

Now, that we have our first record, let's break it all down:

1. We are creating an instance of `CKRecord` and giving it a record type and ID. In this case, the record type is `Shoe` with the ID we created in the preceding code.
2. We will use an image from our folders. You can grab the image from the project files and drag it into your project. Make sure that the file is in your project and not inside the `Assets` folder.
3. Here, we are creating a name and setting it to an `NSString` type.
4. Next, we are creating a price and setting it to a `Double` type.
5. Finally, we are creating an image and setting it to a `CKAsset` type.

All we have to do is save this to CloudKit by adding the following after everything we just added:

```
db.save(record) { (record, error) in
    if let rec = record {
        print("record saved \(rec)")
    } else {
        print("error saving \(error!)")
    }
}
```

We are now using the `save()` method to save our record to CloudKit. Then, we are waiting to get either the record back or an error. Build and run the project and you should see the following error:

```
error saving <CKError 0x600002861d10: "Permission Failure"
(10/2007); server message = "Operation not permitted"; uuid =
8E75ED04-EC6A-4EE7-A38B-7071D9FC62CD; container ID = "iCloud.
io.designtoswiftui.test.HelloCloudKit">
```

You are seeing this error because you have to be logged in to iCloud on the simulator. Go to the **Settings** app in the simulator and tap on **Sign in to your iPad**. Once you add your credentials, re-run the app and you should now see the following:

```
record saved <CKRecord: 0x7fa912506980; recordID=shoe-pos-test:(_defaultZone:_defaultOwner_), recordChangeTag=kfiwefz4, values={ name=Mamba Fury, price=125.99, image=<CKAsset: 0x7fa912508520; path=/Users/craigclayton/Library/Developer/CoreSimulator/Devices/2DC8A5DF-1859-4D3A-BC3F-EBBE7ACCFDE7/data/Containers/Bundle/Application/87A5781C-7F84-4C2A-B853-70A8B0876512/ShoePOS.app/nike-mamba-fury.png, size=201036, UUID=7F19CEA2-BE2B-4F2C-A9CC-45238554C439, signature={length = 21, bytes = 0x019ad956d47d08492d7f7f26cadfdfd9107f8976c8}> }, recordType=Shoe>
```

Now that we have a record saved to CloudKit, let's go check it out on the dashboard. Click on the **CloudKit Dashboard** button on the project TARGETS page:

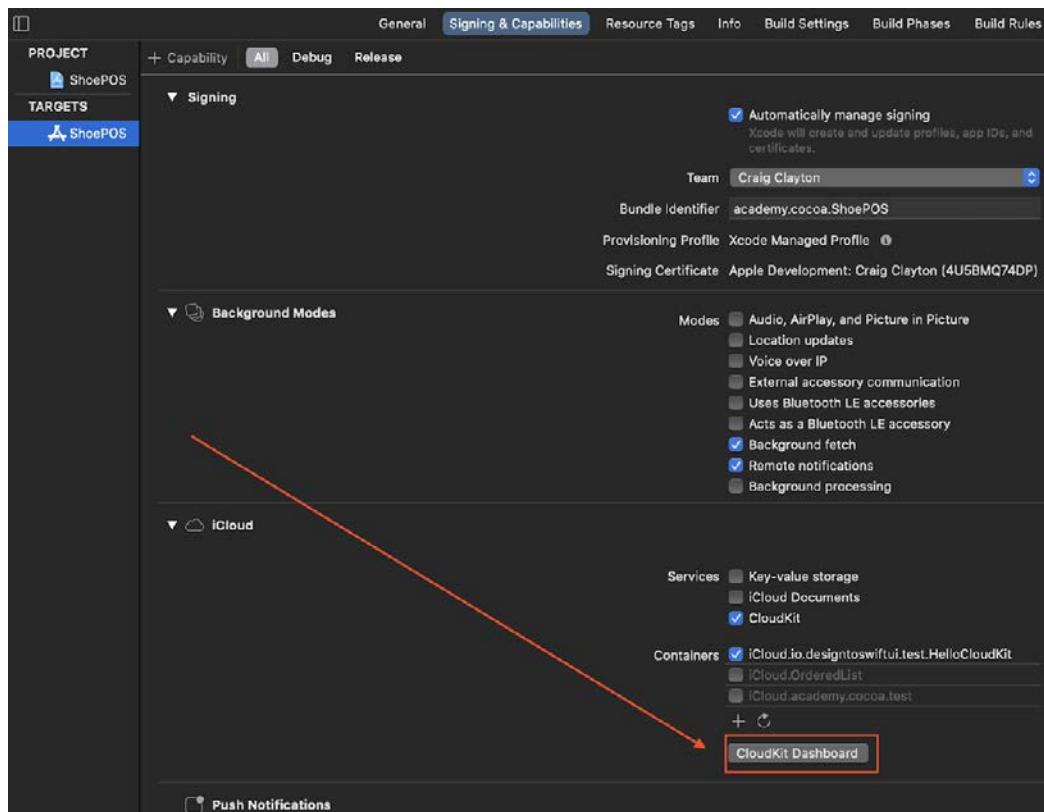


Figure 9.9

When you click on the button, you will be taken to Apple's CloudKit Dashboard. Sign in with your developer account information (make sure it's the developer account you are using for the development of the ShoePOS app and not your iCloud account).

On the **Home** page, you should see all of your containers. For most of you, there will just be one, but you might see others:

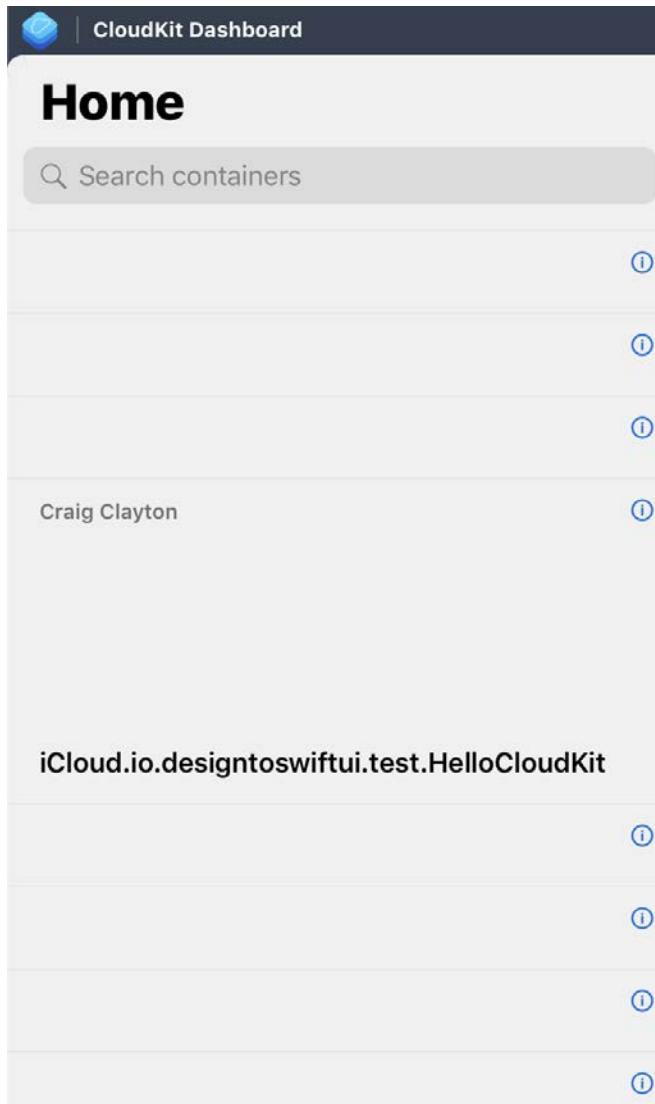


Figure 9.10

Next, click on your container, and then click on the **Data** button:

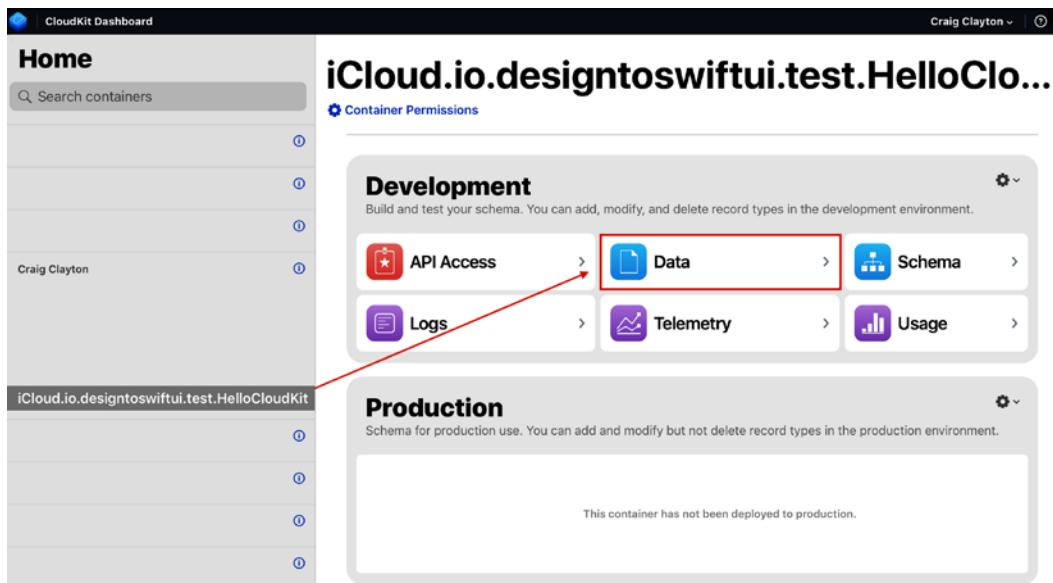


Figure 9.11

When you get to the **Data** screen, in the dropdown under **Database**, change it to **Public Database** and change **Type** to **Shoe**:

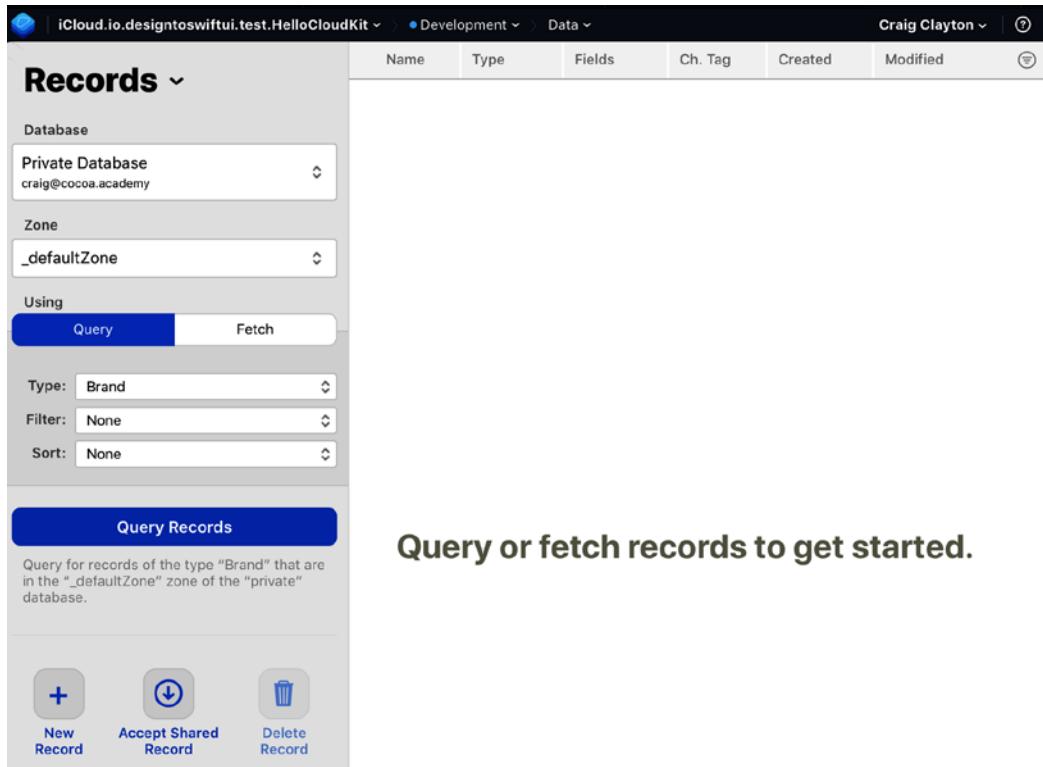


Figure 9.12

Then, hit **Query Records** and you will see that we get a popup with a message:

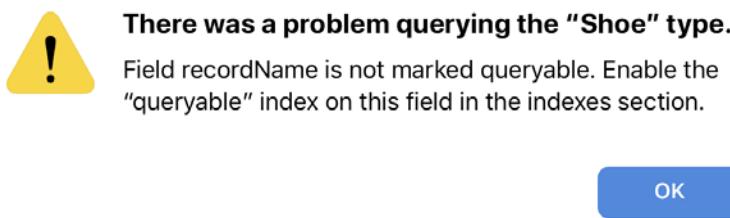
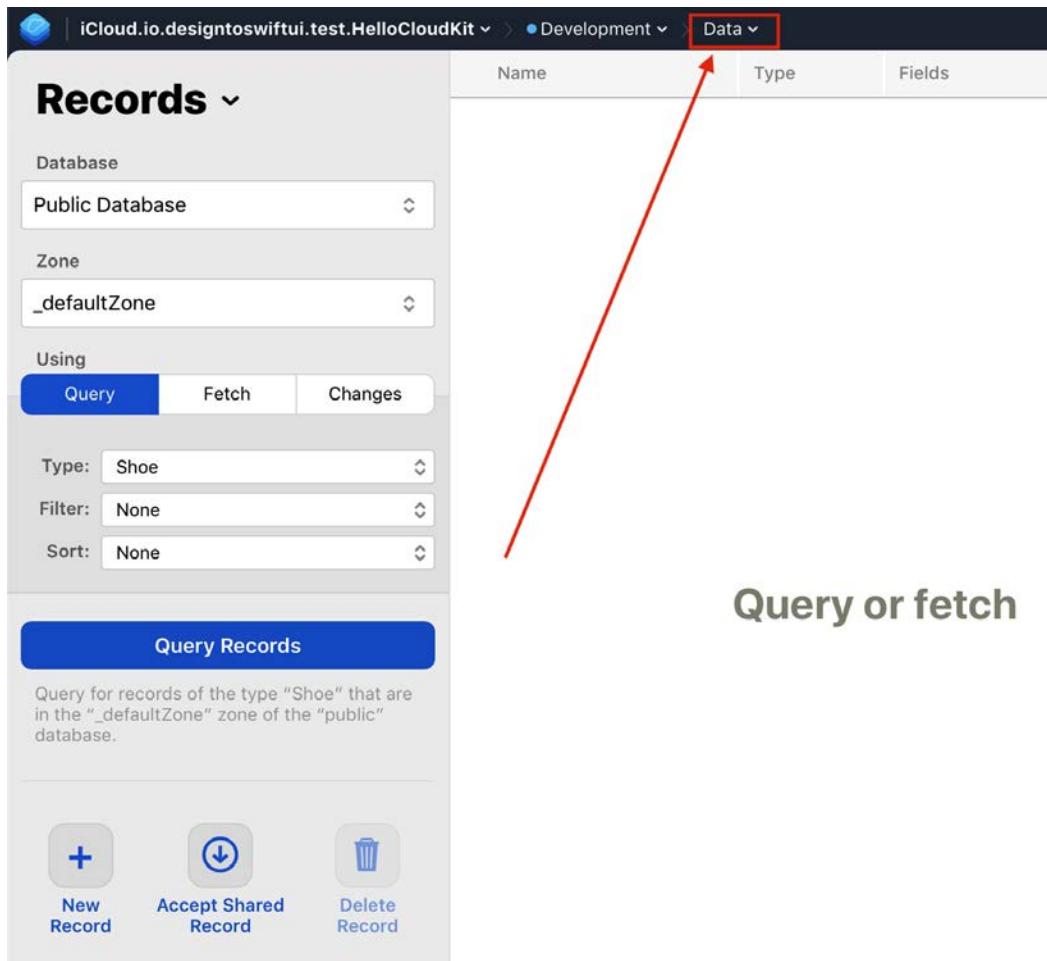


Figure 9.13

The message states that `recordName` is not **queryable**. You get this kind of message every time you add a new record type. Since this happens every time, you will have to remember these steps and the more you do it, the faster you will get at doing this.

How to index a new record type in CloudKit Dashboard

In order to fix indexing new record types, we have to do a few steps. First, start by tapping **Data** at the top of the window:



The screenshot shows the CloudKit Dashboard interface. At the top, there is a navigation bar with the URL "iCloud.io.designtoswiftui.test.HelloCloudKit" and a dropdown for "Development". A red arrow points to the "Data" dropdown menu, which is currently selected. The main area is titled "Records" and shows a table with columns "Name", "Type", and "Fields". On the left, there is a sidebar with filters for "Database" (set to "Public Database"), "Zone" (set to "_defaultZone"), and "Using" (set to "Query"). Below these filters, there are dropdowns for "Type" (set to "Shoe"), "Filter" (set to "None"), and "Sort" (set to "None"). A large blue button labeled "Query Records" is present. Below this button, a description reads: "Query for records of the type "Shoe" that are in the "_defaultZone" zone of the "public" database." At the bottom of the sidebar, there are three buttons: "New Record" (with a plus sign icon), "Accept Shared Record" (with a downward arrow icon), and "Delete Record" (with a trash bin icon).

Query or fetch

Figure 9.14

From the dropdown, select **Schema**:

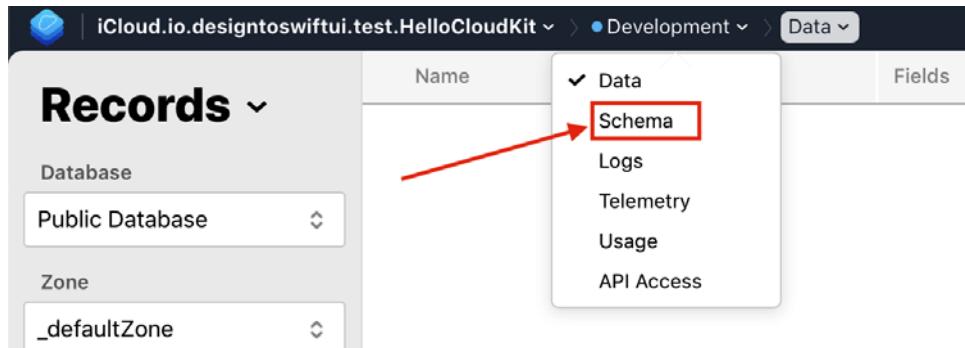


Figure 9.15

Then, click on **Record Types**:

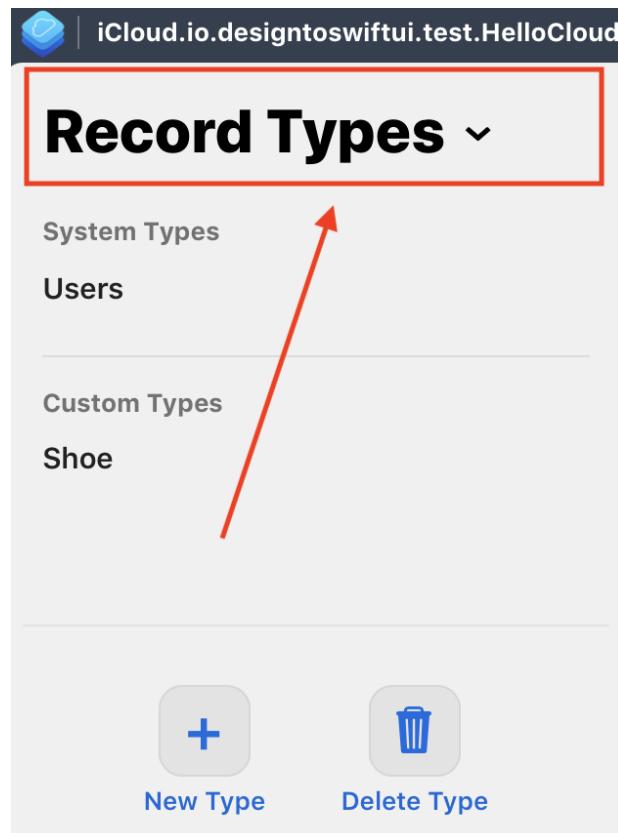


Figure 9.16

Now, click on **Indexes**:

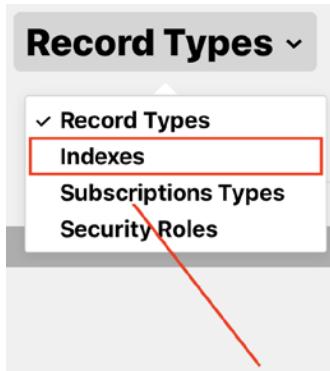


Figure 9.17

Next, select **Shoe** under **Custom Record Types**:

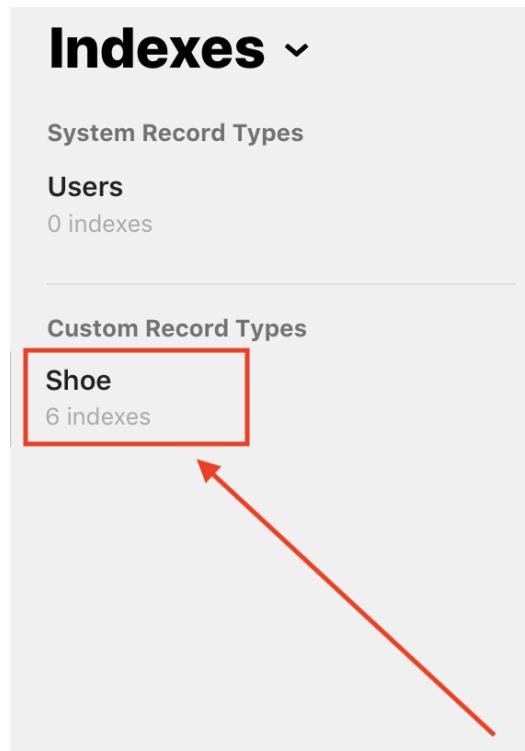


Figure 9.18

Then, click on **Add Index**:

Fields	Index Type	
recordName	QUERYABLE	✖
key	QUERYABLE	✖
key	SEARCHABLE	✖
key	SORTABLE	✖
name	QUERYABLE	✖
name	SEARCHABLE	✖
name	SORTABLE	✖
+ Add Index		

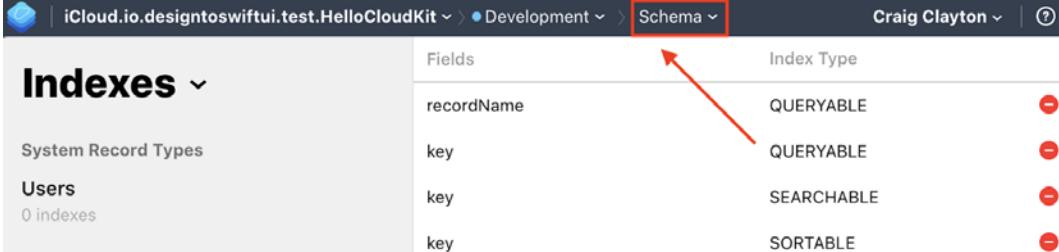
Figure 9.19

Make sure **Index Type** is set to **QUERYABLE**, and then tap the **Save Changes** button:

Fields	Index Type	
recordName	QUERYABLE	✖
key	QUERYABLE	✖
key	SEARCHABLE	✖
key	SORTABLE	✖
name	QUERYABLE	✖
name	SEARCHABLE	✖
name	SORTABLE	✖
recordName	QUERYABLE	✖
+ Add Index		
Remove All Indexes	Revert	Save Changes

Figure 9.20

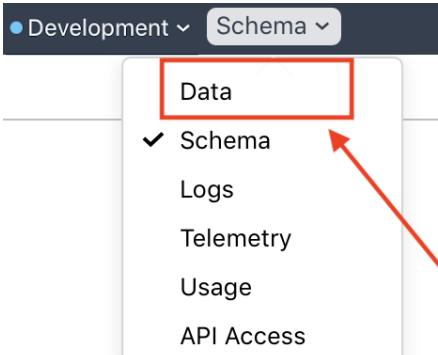
Now, we need to go back and do a query again. At the top of the screen, select **Schema**:



The screenshot shows the CloudKit developer interface. At the top, there is a navigation bar with the project name "iCloud.io.designtoswiftui.test.HelloCloudKit", a "Development" dropdown, and a "Schema" dropdown. The "Schema" dropdown is highlighted with a red box and has a red arrow pointing to it from the text above. To the right of the dropdown, there is a user name "Craig Clayton" and a help icon. Below the navigation bar, on the left, is a sidebar titled "Indexes" with a dropdown arrow. It shows "System Record Types" and "Users" (0 indexes). The main area is titled "Fields" and lists four entries: "recordName" (Index Type: QUERYABLE), "key" (Index Type: QUERYABLE), "key" (Index Type: SEARCHABLE), and "key" (Index Type: SORTABLE). Each entry has a red minus sign icon to its right.

Figure 9.21

Then, select **Data**:



The screenshot shows the CloudKit developer interface with a dropdown menu open. The menu items are "Development" (selected), "Schema" (selected), "Data" (selected and highlighted with a red box), "Logs", "Telemetry", "Usage", and "API Access". A red arrow points to the "Data" item from the text above.

Figure 9.22

Now, hit **Query** again. Double-check and make sure you have **Database** set to **Public Database** and **Type** as **Shoe**:

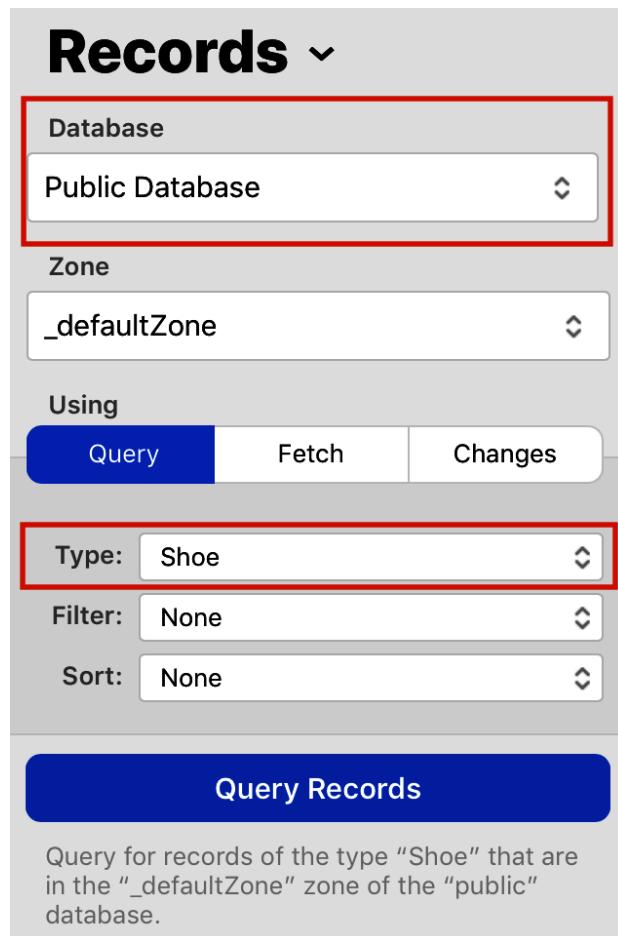


Figure 9.23

If so, hit **Query Records** and you should see the record we added:

Name	Type	Fields	Ch. Tag	Created	Modified	≡
► shoe-pos-test	Shoe	3: image, price, name	kfiwefz4	2020/09/25, 19:50	2020/09/25, 19:50	

Figure 9.24

We now have a better idea of how to create a new record type, as well as how to make sure our record type is indexed. You can take a minute and check out your record type. You should also see the image you uploaded, as well as the other information we saved:

Name	Type	Fields	Ch. Tag	Created	Modified	≡	Editing Record	≡
▼ shoe-pos-test	Shoe	<p>image 196.3kB Asset</p> <p>price 125.99</p> <p>name Mamba Fury</p>	kfiwefz4	2020/09/25, 19:50	2020/09/25, 19:50		<p>Metadata</p> <p>Name: shoe-pos-test</p> <p>Type: Shoe</p> <p>Database: public</p> <p>Zone: _defaultZone</p> <p>Created: Sep 25 2020 7:50 PM by _6e87f15b587eb4d384e04c00a9345e44</p> <p>Modified: Sep 25 2020 7:50 PM by _6e87f15b587eb4d384e04c00a9345e44</p> <p>Change Tag: kfiwefz4</p> <p>References None</p> <p>Custom Fields 3 of 3</p> <p>image Asset 196.3kB (Binary File) Details Download Choose File</p> <p>name String Mamba Fury</p> <p>price Double 125.99</p>	

Figure 9.25

From this point, you can mess around with this record a bit more, but keep in mind that you can edit the name and price and can even upload a different image.

Now that we are a bit more familiar with CloudKit, let's create some new record types that we will use for this chapter. You can delete the record or keep it for reference, but we won't be using it from this point forward:

The screenshot shows the CloudKit Record Editor for a Shoe record named "shoe-pos-test". The record has three fields: "image" (a binary file asset), "price" (125.99), and "name" (Mamba Fury). The "Editing Record" tab is selected. The "Metadata" section shows the record's type as "Shoe", database as "public", and zone as "_defaultZone". The "Custom Fields" section shows the "image" field as an asset. The "Delete" button in the bottom right corner is highlighted with a red box and an arrow.

Name	Type	Fields	Ch. Tag	Created	Modified
shoe-pos-test	Shoe	image 196.3kB Asset ⓘ	kfiwefz4	2020/09/25, 19:50	2020/09/25, 19:50
		price 125.99			
		name Mamba Fury			

DELETE RECORD

Editing Record

Metadata

Name: shoe-pos-test
Type: Shoe
Database: public
Zone: _defaultZone
Created: Sep 25 2020 7:50 PM
by _6e87f15b587ab4d384e04c00a9345e44
Modified: Sep 25 2020 7:50 PM
by _6e87f15b587ab4d384e04c00a9345e44
Change Tag: kfiwefz4

References None ⓘ

Custom Fields 3 of 3 ⓘ

image Asset
196.3kB (Binary File) Details Download Choose File ⓘ

name String
Mamba Fury

price Double
125.99

Delete **Reload** **Save**

record listed

Figure 9.26

Before we move on, please delete the entire `init()` function we created inside `StorePOSApp`.

Creating CloudKit models

We will create one model together and then you will need to create the last two models needed for this chapter. When you are done creating each model, we will move on to creating our `create` and `fetch` methods, which we will use in the app. Before we create our models, let's create a couple of extensions that will help simplify a few things.

CloudKit extensions

If you open the project files for this chapter, in the `step 1` folder, you will see a couple of files that we need. Drag all of these files into the `Model` folder. Open `CKContainer+Extension` and update the identifier to match your CloudKit container name:

```
extension CKContainer {
    static var shared: CKContainer {
        return CKContainer(identifier: "Cloudkit container name
here")
    }
}
```

If you take a look in `CKConstant`, you will see the following:

```
struct CKConstant {
    struct RecordType {
        static let Product = "Product"
        static let Brand = "Brand"
        static let Size = "Size"
    }
}
```

We have a static variable that points to each of our CloudKit models. Let's create our product model together.

Creating our product model

The product model is the model that we will use for every product we have in the app. Using `Product` means that you can easily flip out shoes and make the product any kind of product you want. Right-click on the `Model` folder and select **Create a new file named Product**. Select **swift file** and replace `import Foundation` with the following:

```
import CloudKit
import UIKit
```

Now, under the import statement, add the following:

```
struct Product: Identifiable {
    var id = UUID().uuidString
    var recordID: CKRecord.ID?
    var productName: String
    var price: Double
    var productNo: String
    var record: CKRecord
    var brand: Brand?
    let imageFileURL: URL?
    var sizes: [Size] = []
}

// Add next step here
}

// Add extensions here
```

We haven't added anything new. We are making our `Product` struct conform to `Identifiable`, which is used with SwiftUI and our `ForEach` statements. So far, this is just a simple struct, but we do have a reference to `Brand` and `Size` here. You will get errors for these. You might have some errors, ignore them for now; they will go away once you have added all of your models.

Now, replace `// Add next step here` with the following:

```
static let `default` = Self(record: defaultProduct) // (1)
static var defaultProduct: CKRecord { // (2)
    let recordID = CKRecord.ID(recordName: "nike-product")

    let productRecord = CKRecord(recordType: CKConstant.
        RecordType.Product, recordID: recordID)
    productRecord[.productName] = "Nike" as NSString
    productRecord[.price] = 199.99 as Double
    productRecord[.productNo] = "NK1234" as NSString

    let imageURL = Bundle.main.url(forResource: "nike-mamba-
        fury", withExtension: "png")!
    productRecord[.productImage] = CKAsset(fileURL: imageURL)
```

```
        return productRecord
    }
    // Add next step here
```

All of this should be familiar to you already, but I will explain it again.

Here, we are creating a static variable for previews named `default` and we are passing a generic `CKRecord` to it.

Our `defaultProduct` is used for Swift previews and inside of this `static var`, we are creating a fake product. We are setting the name, price, product number, and image.

There will be additional errors, but you can continue to ignore them. We will fix them soon. Let's add another variable:

```
var productImage: UIImage {
    if let fileURL = self.imageFileURL {
        if let image = UIImage(contentsOfFile: fileURL.path) {
            return image
        }
    }
    return UIImage(named: "nopicture")!
}
// Add next step here
```

Here, we are adding a helper variable for `image`. Since this is being saved as a `CKAsset`, we need to get the image file URL and pass it to `UIImage(contentsOfFile:)`. If there is an issue with the image, we set it to a default image. Next, we need to add a custom `init` method. Replace `// Add next step here` with the following `init` method:

```
init(record: CKRecord) {
    self.record = record
    productName = record[.productName] as! String
    productNo = record[.productNo] as! String
    self.price = record[.price] as! Double

    if let asset = record[.productImage] as? CKAsset {
        imageFileURL = asset.fileURL
    } else {
        imageFileURL = nil
    }
}
```

```
    }  
}
```

Creating this custom `init` method will be useful later when we are working with Products. Let's add one more helper `init` method by adding the following after the last `init` method:

```
init(product: ProductItem) {  
    let record = CKRecord(recordType: CKConstant.RecordType.  
        Product)  
    record[.name] = product.name as NSString  
    record[.productNo] = product.productNo as NSString  
    record[.price] = product.price as Double  
  
    let url = Bundle.main.url(forResource: product.image,  
        withExtension: "png")!  
    record[.image] = CKAsset(fileURL: url)  
  
    self.init(record: record)  
}
```

You probably noticed errors for the following:

```
productRecord[.productName]
```

We are using enums for safe typing, rather than the following method:

```
productRecord["productName"]
```

Replace // Add extensions here with the following:

```
extension Product {  
    enum RecordKey: String {  
        case id  
        case recordID  
        case productName = "name"  
        case price  
        case productNo  
        case record  
        case brand
```

```
        case productImage = "image"
        case sizes
    }
}

extension CKRecord {
    subscript(key: Product.RecordKey) -> Any? {
        get {
            return self[key.rawValue]
        }
        set {
            self[key.rawValue] = newValue as? CKRecordValue
        }
    }
}
```

We now have a safe type for our keys instead of using strings. We need two more models: `Brand` and `Size`. You can find both models in the project files for this chapter. Now that you have `Brand` and `Size`, we can move on to creating some brands manually.

Creating brands

We are going to manually create some brands, and then you can create something similar for products. First, we are going to set up our CloudKit helper before we add any code.

CloudKit helper

We are going to create a CloudKit helper, which will be used to make calls to the CloudKit database. We will have all of this code in a central location:

1. Create a new folder called `CloudKit`.
2. Then, right-click on the `CloudKit` folder and select **New File...**
3. Select **Swift** file and name it `CloudKitHelper`, and then hit **Save**.
4. First, after the `import` statement, add the following:

```
import CloudKit
```

Next, let's get started by adding a few variables we will need.

5. Add the following struct after the `import` statement:

```
struct CloudKitHelper {  
  
    // Databases  
    static let publicDB = CKContainer.shared.  
        publicCloudDatabase // (1)  
  
    // Add next step here  
}
```

We are creating a variable to access the public database of our container. We are going to create a method to get records from CloudKit Dashboard. We will make this method work with any `CKRecord`. Replace `// Add next step here` with the following:

```
static func getRecords(recordType: CKRecord.RecordType =  
    CKConstant.RecordType.Brand,  
    cursor: CKQueryOperation.Cursor? = nil,  
    completion: @escaping (([CKRecord]) ->  
        Void)) {  
  
    let operation: CKQueryOperation // 1  
    if let cursor = cursor {  
        operation = CKQueryOperation(cursor: cursor) // 2  
    } else {  
        operation = CKQueryOperation(query: CKQuery(recordType:  
            recordType, predicate: .init(value: true))) // 3  
    }  
    var records: [CKRecord] = [] // 4  
    operation.recordFetchedBlock = { record in // 5  
        records.append(record)  
    }  
    operation.queryCompletionBlock = { cursor, error in // 6  
        if let cursor = cursor {  
            getRecords(cursor: cursor) { fetchedRecords in // 7  
                records.append(contentsOf: fetchedRecords) // 8  
                completion(records)  
            }  
        } else { completion(records) } // 9  
    }  
}
```

```
    }
    CloudKitHelper.publicDB.add(operation) // 10
}

// Add next step here
```

We now can use this method to get products or brands from the dashboard. Let's break down what we just added:

1. First, we create a `CKQueryOperation` variable.
2. We then initialize `CKQueryOperation` using a cursor. Basically, when we fetch data from CloudKit Dashboard, all of the data might not come back when we call it. Using a cursor will allow us to call our method until we get all of the data.
3. We initialize another `CKQueryOperation` variable and check whether the cursor is nil. If it's nil, we use a basic `CKQueryOperation`.
4. Now, we create a new `CKRecord` array that we can store `CKRecord` items inside of.
5. Next, we use `recordFetchedBlock`, which has an enclosure that's called each time a record is loaded (fetched). The new record is passed into the enclosure and is appended to our `results` array.
6. We are using the `queryCompletionBlock` from `queryOperation`, which is called once all the records have been loaded. If an error occurs, the enclosure is passed the details of that error.
7. Next, if the cursor is not nil, we rerun the function again to fetch more data.
8. Then, we add items to the array as they come back.
9. Finally, when the cursor comes back nil, we call the completion block.
10. We call the `add` operator to begin the fetch.

We need to add a couple more helper methods. The current method is used with products, but we need a couple of methods to handle `Size`. Replace `// Add next step here` with the following:

```
static func fetchAllSizes(products: [Product], completion: @escaping ([Product], Error?) -> Void) {
    let recordIDs = products.map { $0.record[.sizes] as!
        [CKRecord.Reference] }.reduce([], +).map { $0.recordID }
    var prod = products
```

```

let operation = CKFetchRecordsOperation(recordIDs:
    recordIDs)

// Add next step here

CloudKitHelper.publicDB.add(operation)
}

// Add next method here

```

Inside the method, we are getting all of the record IDs for sizes and we are going to use those so that we can get each size CKRecord. Replace // Add next step here with the following:

```

operation.fetchRecordsCompletionBlock = { result, error in
    if let _ = result?.values {
        prod.enumerated().forEach { (index, product) in
            let productSizes = product.record[.sizes] as!
                [CKRecord.Reference]
            let x = productSizes.map { $0.recordID }
            CloudKitHelper.fetchSizes(x) { (sizes) in
                prod[index].sizes = sizes
                completion(prod, error)
            }
        }
    }
}

```

We are adding an operation for fetching records. We have seen this before except instead of a simple fetch, we are mapping our sizes to objects and making sure that they go with each product. Let's add one more helper method. Replace // Add next method here with the following:

```

static func fetchSizes(_ recordIDs: [CKRecord.ID], completion:
    @escaping ([Size] -> Void) {
    var recordSizes:[Size] = []
    let fetchOperation = CKFetchRecordsOperation(recordIDs:
        recordIDs)

```

```
// Add next step here

CloudKitHelper.publicDB.add(fetchOperation)
}
```

We are creating a `fetchSize` method in order to help get reference IDs and convert them into records. We have an empty `Size` array that we will use to store all of our sizes. Replace `// Add next step here` with the following:

```
fetchOperation.fetchRecordsCompletionBlock = {
    records, error in
    if error != nil {
        print("\(error!)")
    } else {
        if let records = records {
            for record in records {
                recordSizes.append(Size(record: record.value))
            }
        }
    }

    DispatchQueue.main.async {
        var sorted = recordSizes.sorted(by: { $0.size < $1.size })
        completion(sorted)
        sorted = []
    }
}
```

Lastly, we are looping through the records and adding them to the array. Once they are finished, we sort them in order from smallest to largest. Now that we have our helper methods created, let's move on to our view model.

Creating our view model

We are going to create our view model next. We will add a few methods that will help us create brands and products, as well as fetching brands and products from CloudKit Dashboard. Create a new Swift file inside the Model folder. Name this file `ShoePOSViewModel` and add the following `import` statement after `import Foundation`:

```
import CloudKit
```

Next, add the following:

```
class ShoePOSViewModel: ObservableObject {  
    @Published var brands: [Brand] = []  
    @Published var products: [Product] = []  
    @Published var isProductDetailVisible = false  
    @Published var selectedProduct: Product?  
  
    // Add next step here  
}
```

We have two array variables that we will use in the app and two others that we will use a bit later. Our class is an `ObservableObject`, which allows us to tie it into our UI. Since we do not want to have to create brands or products by hand, we are going to create them once so that we have data to use. Typically, this data would be created in another app and you would just load it in. In the project files folder, in step 2, drag and drop `ShoeData` and `ShoePOSViewModel+Extension` into your Model folder. I have created some helper methods to create products and brands, as well as given you some dummy data. We imported `BrandItem` and `ProductItem` earlier. Back in `ShoePOSViewModel`, replace `// Add next step here` with the following method:

```
func createBrands() {  
    let records = ShoeData.brandsItems.map { Brand(brand: $0)  
    }.map { getRecord(brand: $0) }  
    save(records: records)  
}  
  
// Add next step here
```

Our `createBrands` method is for creating brands. We are using data from the `ShoeData` file and then saving the brands to CloudKit Dashboard. Replace `// Add next step here` with the following method:

```
func createProducts() {
    CloudKitHelper.getRecords() { records in
        DispatchQueue.main.async {
            self.brands = []
            self.brands = records.map { Brand(record: $0) }

            for item in ShoeData.productItems {
                let brandID = self.filterBrands(by: item.brand)
                self.save(product: item, brandID: brandID?
                    recordID)
            }
        }
    }
    // Add next step here
}
```

In order to create products, we need brands. First, we grab the brands from CloudKit Dashboard, and then we look through our array of products. We first find the brand for each shoe as we loop through the array, and then we get the brand ID and save it with the product. If you wanted to, you can now fetch products by brand. Let's move on to the next method by replacing `// Add next step here` with the following method:

```
func fetchAllBrands() {
    CloudKitHelper.getRecords() { records in
        DispatchQueue.main.async {
            self.brands = []
            self.brands = records.map { Brand(record: $0) }
        }
    }
    // Add next step here
}
```

We are using this method to fetch all brands from CloudKit Dashboard. When we get the records back, we map them to our Brand object. Again, replace // Add next step here with the following:

```
func fetchAllProducts() {
    CloudKitHelper.getRecords(recordType: CKConstant.
        RecordType.Product) { records in
        let prods = records.map { Product(record: $0) }
        CloudKitHelper.fetchAllSizes(products: prods) { (items,
            error) in
            DispatchQueue.main.async {
                self.products = []
                self.products = items
            }
        }
    }
}
```

In our last method, we are fetching all product records, and then we map the record to our Product object. Now we have everything we need to display products. Let's get our products data loaded next.

Creating our dummy data

We already have our brands saved, and now we want to save products. To make it easier, I already set up the code you need to do this. ShoePOSViewModel will loop through an array of ProductItems. Each item will have the product image, name, price, product number, and brand name. We will use the brand name to search the CloudKit manager for the brand record. When we find the record, we will return it and associate the shoe with the brand. What this means is that we can do a search in CloudKit for shoes by brand and get all of the Nike shoes only. I have also set it up so that when a shoe is added, it will generate a random amount of sizes for your inventory. When we tap on a product in the app, it will show sizes by inventory, and this makes it easy to fake this data for our purpose. Open ShoePOSApp and add the following above the body variable:

```
let model = ShoePOSViewModel()
```

Then, underneath the variable, add the following:

```
init() {
    self.model.createBrands()
}
```

Build and run the project, and we will now have brands created. Go ahead and delete the `init` method as we do not need it anymore. Make sure that you go into CloudKit Dashboard and make sure the brand `recordName` is `queryable`. Then, replace `ContentView()` with the following:

```
ContentView()
    .environmentObject(model)
    .onAppear {
        model.createProducts()
}
```

Again, build and run the project; you won't see anything yet because we need to add a few things, but we at least have data. Once you are done building the project, replace `model.createProducts()` with `model.fetchAllProducts()`. We no longer need the create methods. Also, make sure that you aren't logged out of CloudKit. Go back onto the dashboard and make sure both the `Size` and `Product` `recordName` are `queryable`.

Now that we have products, we can display them in the app.

Displaying CloudKit models

We just set up our data in the CloudKit database and we have our new model attached to our `ContentView`. To get data to display in `ProductsContentView`, we need to do the following:

1. Open `ProductsView` and add the following variable above the `body` variable:

```
@EnvironmentObject var model: ShoePOSViewModel
```

2. Then, find `ProductsContentView()` and update it to the following:

```
ProductsContentView().environmentObject(model)
```

We can now access products inside `ProductsContentView`. Let's get that set up now.

Setting up products

We are now going to get our products to display in the products area. Since all of our data is in CloudKit Dashboard, this is pretty easy to do:

1. Open `ProductsContentView` and add the following variable above the body variable:

```
@EnvironmentObject var model: ShoePOSViewModel
```

2. Then, update `ForEach` to the following:

```
ForEach(model.products) { product in
    ProductView(product: product)
}
```

3. Next, open `ProductView` and add the following variable above the variable:

```
let product: Product
```

4. Now, update the `VStack` container that contains all of your product info with the following:

```
vStack(spacing: 0) {
    Image(uiImage: product.productImage)
        .resizable()
        .frame(width: 188, height: 100)
    VStack(spacing: -8) {
        Text(product.productName.uppercased())
            .custom(font: .demibold, size: 18.0)
        Text("$\\"(product.price, specifier: "%.2f")\"")
            .foregroundColor(.baseLightBlue)
            .custom(font: .bold, size: 29.0)
    }
}
```

5. Finally, update previews with the following:

```
ProductView(product: Product.default)
```

Build and run the project and you should now see the following:

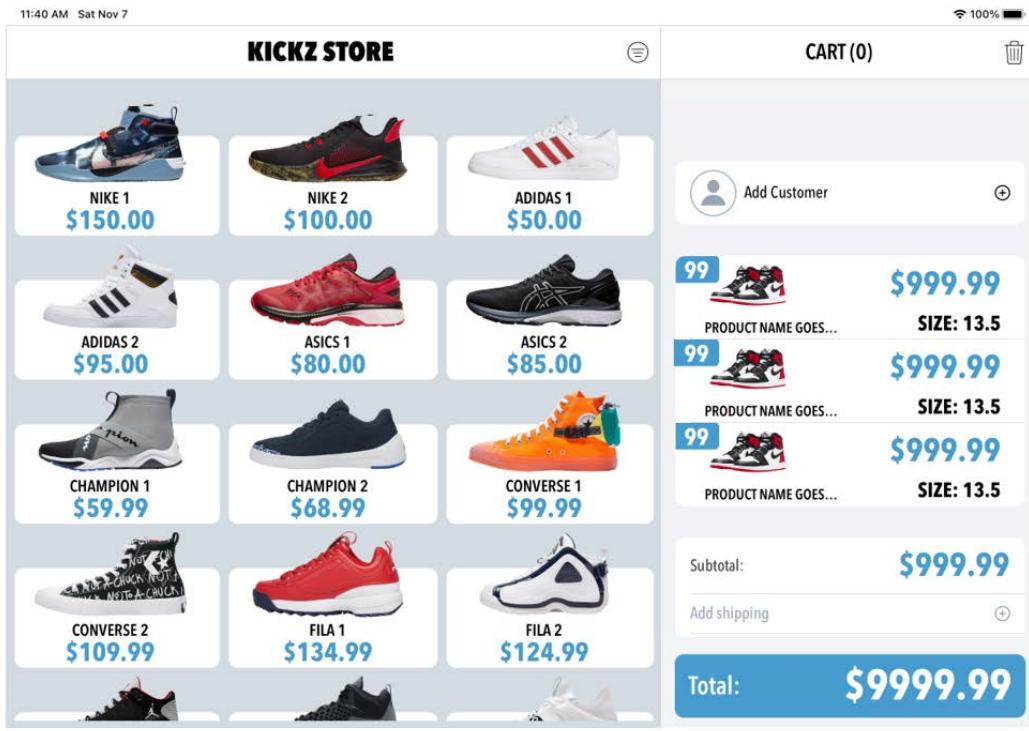


Figure 9.27

Next, we want to be able to tap on a product and see a product details view. Let's work on the product details view next.

Displaying a product details view modal

We are now displaying products inside `ProductsContentView()`. Next, we need to display `ProductDetail` when you tap on a product, which is what we need to set up next:

1. Open `ProductsContentView`, then update `ProductView()` in the `ForEach` with the following:

```
ProductView(product: product).onTapGesture {
    model.isProductDetailVisible.toggle()
    model.selectedProduct = product
}
```

2. Next, *Command* + click on `ScrollView` and select embed in `VStack`. Then, change the `VStack` to a `ZStack`.
3. Outside of the `ScrollView` but inside of the `ZStack`, add the following:

```
if let product = model.selectedProduct {
    ProductDetail(product: product)
        .environmentObject(model)
        .opacity(model.isProductDetailVisible ? 1 : 0)
        .animation(.default)
}
```

Now, we need to update `ProductDetail` before we can run the project. Let's update this next.

Displaying data in the product details view

We are displaying `ProductDetail` as a modal, but we need to be able to close `ProductDetail` when you tap the close button and also display the actual product we tap on, open `ProductDetail`:

1. Add the following variable above the `body` variable:

```
@EnvironmentObject var model: ShoePOSViewModel
let product: Product
```

2. Inside the `Button` action, add the following:

```
model.isProductDetailVisible.toggle()
```

We can now close the modal. Next, we need to get our data to display inside `ProductDetail`:

1. Find `Text ("JUMPMAN AIR JORDANS")` and replace it with the following:

```
Text(product.productName)
```

2. Then, find `Text ("CJ9999-001")` and replace it with the following:

```
Text(product.productNo)
```

3. Now, let's update Image ("shoe-1") with the following:

```
Image(uiImage: product.productImage)
```

4. Then, we will update Text ("\$999.99") to the following:

```
Text("$\$(product.price, specifier: "%.2f")")
```

5. Now, update ForEach(0..<29) { _ in } with the following:

```
ForEach(product.sizes) { item in
    SizeCartItemView()
}
```

6. Finally, update Previews with the following:

```
ProductDetail(product: Product.default).previewLayout(.fixed(width: 1112, height: 834))
```

Build and run the project and you will be able to see the product details display over the products. We can close the modal and select another product.

We are finished updating `ProductDetail` and we can now turn our attention to the shopping cart next. The shopping cart is the last thing we need before we start hooking everything up.

The shopping cart

We need to get our shopping cart together so that we can add products. We want to be able to tap on a product and then display all of the sizes for each product. Then, we want to be able to select a size and add the shoe/size to the cart. We first need to create a `CartItem` before we get into the shopping cart.

CartItem

We need a model object for representing a `CartItem`. Because we are working with shoes, we will need to include `Size`. Our app will not actually keep track of the inventory. What I mean by that is that this app does everything up to that point. We cover how to do that inside of the app, so if you want to add inventory updating, it wouldn't take much to add this feature. It is just beyond the scope of this book.

Right-click on the Cart folder and create a new Swift file. Then, after the `import` statement, add the following:

```
struct CartItem: Identifiable {
    var id = UUID().uuidString

    var product: Product
    var size: Size
    var quantity: Int
    var maxQuantity: Int

    init(product: Product, size: Size, quantity:Int) {
        self.product = product
        self.size = size
        self.quantity = quantity
        self.maxQuantity = size.quantity
    }
}
```

Our cart item is straightforward. Next, let's move on to our shopping cart. One of the amazing things about SwiftUI and using `ObservableObject` is that it really makes using Core Data and CloudKit and doing basic interactions so much easier. We are going to create a shopping cart that subclasses `ObservableObject`. Let's right-click on the Model folder and add a new Swift file named `ShoppingCart`. Replace the `import` statement with the following:

```
import SwiftUI
import Combine
```

After the `import` statements, you are going to add the following class declaration:

```
class ShoppingCart: ObservableObject {
    @Published var quantity: Int = 0 // (1)
    @Published var cartTotal: Double = 0 // (2)
    @Published var items: Dictionary<String, CartItem> = [:] // (3)
    @Published var isShippingAdded: Bool = false // (4)
    // (5)
    @Published var selectedProduct: Product? // (6)
```

```
    @Published var selectedSize: Size? // (7)
    // Add next step here
}
```

Our class declaration is subclassing `ObservableObject` and we also have a number of variables that we will need for our shopping cart. Let's go over each one:

1. `quantity`: This is used for the number of items in our cart.
2. `cartTotal`: This is used for the running total amount of everything in the cart.
3. `items`: We are using a dictionary array for each cart item we add to the array.
4. `isShippingAdded`: We are using this value to control our form. We will look at this further when we use it.
5. `isProductDetailVisible`: Used for presenting the detail view modal.
6. `selectedProduct`: Keeps track of the selected products.
7. `selectedSize`: Keeps track of the selected size.

We will now need a couple of helper variables. First, we will start with `total`. Replace `// Add next step here` with the following:

```
var total: Double {
    if items.count > 0 {
        var amount: Double = 0

        for item in items {
            let price = item.value.product.price
            let quant = item.value.quantity
            amount += price * Double(quant)
        }

        return amount
    } else {
        return 0
    }
}

// Add next step here
```

We are using `total` as a helper. When an item gets added to the cart, we loop through the array and combine the total with the new quantity for each product. Let's move on to our next helper method. Replace `// Add next step here` with the following:

```
func toggleCart(item: CartItem) {
    if items[item.size.id] == nil {
        items[item.size.id] = CartItem(product: item.product,
            size: item.size, quantity: 1)
    } else {
        items[item.size.id] = nil
    }
}
// Add next step here
```

The `toggleCart` method checks to see whether the product size ID has been added to the cart and if the ID has a value, whether we will add a new one to the cart. Otherwise, we set the item id to nil. Our next method is for updating tax:

```
func getTaxSum() -> Double {
    var GST: Double = 0.0

    if (total > 0) {
        GST = total * 0.13
    } else {
        return GST
    }

    return GST
}
// Add next step here
```

Now, as our `total` method only does pre-tax totals, we need to actually have something where we can combine totals with tax. In the preceding method, we take the total and multiply it by `0.13` (sales tax). If you wanted to update the tax, you could just replace `0.13` with whatever you want. Now, we need a function to add the actual `CartItem` to the shopping cart array. Replace `// Add next step here` with the following function:

```
func shoppingCartItems() -> [CartItem] {
    var products: [CartItem] = []
    DispatchQueue.main.async {
        self.quantity = self.items.values.map { $0.quantity
            }.reduce(0, +)
    }
    for item in items.values {
        products.append(item)
    }
    return products
}
// Add next step here
```

Our `shoppingCartItems` function is easy. We use this method to display the items in our shopping cart. I am using `DispatchQueue` so we run this code on the main thread. Finally, we have two more functions, and then we are done. Replace `// Add next step here` with the following:

```
func getOrderTotal() -> Double {
    return Double(total + getTaxSum())
}

func inCart(item: CartItem) -> Bool {
    return items[item.size.id] != nil
}
```

We have a `getOrderTotal`, which combines `total` and tax together, and finally, we have an `inCart` method, which just returns `true` or `false` depending on whether it can find the element in the cart. Our shopping cart is done, and it really didn't need a lot of code to get it up and running. Now it is time to go through all of the files and just update each one to have the helpers added to them.

Updating our product views

Now we'll move on to updating our prototype and making it work with iCloud, as well as building out the interactivity. Before we jump into updating all of our files, we need to set up our `ObservableObject`. Open `ShoePOSApp` and add the following variables above the `body` variable:

```
@StateObject private var cart = ShoppingCart()
```

Next, update `ContentView()` with the following:

```
ContentView()
    .environmentObject(model)
    .environmentObject(cart)
    .onAppear {
        model.fetchAllProducts()
    }
```

We now have our `environmentObject` set up and the model fetching the products and brands when the app launches.

Next, we will update all of our views for the products, and then we will turn our attention to the cart views. Since `SizeCartItemView` is something we use in both views, we are going to start with this first:

1. Open `SizeCartItemView` and above the `body` variable, please add the following:

```
@EnvironmentObject var shoppingCart: ShoppingCart
let item: CartItem
```

2. Next, update previews from `SizeCartItemView()` to the following:

```
SizeCartItemView(item: CartItem.init(product: Product.default,
    size: Size.default, quantity: 1))
```

Let's update our text views first.

3. Find `Text("13.0")` and change it to the following:

```
Text("\u{1d64}(item.size.size, specifier: "%.1f")")
```

4. Finally, update `Text("99")` to the following:

```
Text("\u{1d64}(item.size.quantity)")
```

Now, find `ShoePOSButton()` and `Spacer()`:

```
ShoePOSButton(title: "ADD TO CART")
    .opacity(0.3)
Spacer()
```

Replace this code with the following:

```
if item.size.quantity > 0 { // 1
    if shoppingCart.inCart(item: item) { // 2
        StepperView(items: self.$shoppingCart.items, item:
            item) // 3
    } else {
        ShoePOSButton(title: "ADD TO CART") // 4
            .onTapGesture { // 5
                self.shoppingCart.toggleCart(item: item)
            }
    }
} else {
    ShoePOSButton(title: "ADD TO CART")
        .opacity(0.3) // 6
}

Spacer()
```

Now, that we've added a few lines of code, let's take this chunk of code step by step:

1. We first check that the quantity is greater than 0 (make sure the product is available in that size!). If so, we move to *step 2*.
2. Next, we use the `inCart (item:)` method that we created in the shopping cart class. The `inCart (item:)` method checks to see whether the item was already added to the shopping cart. If `true`, we move to *step 3*, and if `false`, we move to *step 4*.
3. If we see `StepperView ()`, this means we already have that item in the cart.
4. If we see the **ADD TO CART** button, then that means this item is not in the cart.

5. The tap gesture attached to the button uses `toggleCart(item:)` to add the item to the cart.
6. If the size quantity equals 0, then we set the opacity to 30%.

For `SizeCartItemView`, I deleted Previews for simplicity's sake. It is not something I do a ton, but I did delete it inside this view. We are done with `SizeCartItemView` and we are doing with the app. Shopping Cart currently has a few files that need to be updated. If you want to build out the Shopping Cart feel free to do so now. Before you start open the `step 3` folder and replace your `StepperView` with mine. You can find `StepperView` inside of `Products` folder in the Supporting Views folder.

We now have products from CloudKit loading in our content area. We now just need to make a few updates to get our cart to work. Open the project files folder and drag the four remaining files(`CartContentView`, `CartItemView`, `CartHeaderView` and `CartTotalView`) and replace all of the files inside the `step 3` folder. Then you just need to make one more update. Update `ProductDetail` and update `SizeCartItemView()` to the following:

```
SizeCartItemView(item: CartItem(product: product, size: item,  
quantity: 0))
```

You can now build and run the project and add products to the cart.

Summary

You are done setting up this project. Now it's time to create products, and remember that you will need to update the index in the CloudKit manager online. However you decide to add products, you have all the tools you need to make this work, either using an array, adding products in CloudKit Dashboard, or creating a small form that does it for you.

In this chapter, you learned the basics of CloudKit. You also learned about how to use Cloudkit Dashboard to view your models as well as edit and delete them. We created a shopping cart that you can now take to the next level with CloudKit.

In the next chapter, we will design our sports news app.

10

Sports News App – Design

We have come a long way and, hopefully, by now you are comfortable with designing. In this chapter, we will follow along with another design, but you will have plenty of challenges to complete before you get to your biggest design challenge in the book. We will work on making our iPad app work at any orientation and see how we can make our lives easier by prototyping our layout first.

In this chapter, we will be working with the following:

- Sidebar navigation
- Resizable content
- The video player
- A multi-tasking iPad app

As we have discussed in every design chapter, you can find all of the design specs in the project files:

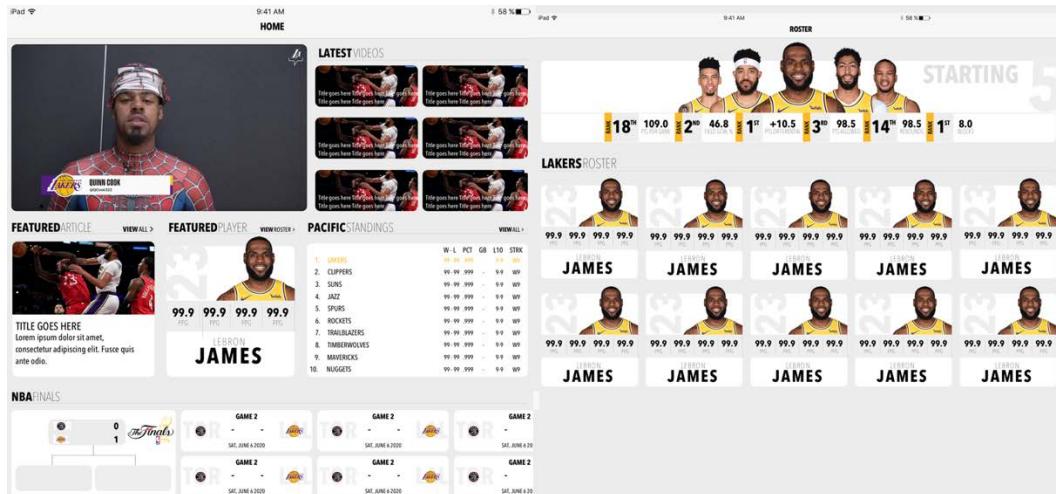


Figure 10.1

All of the assets, colors, and fonts are already set up for you:

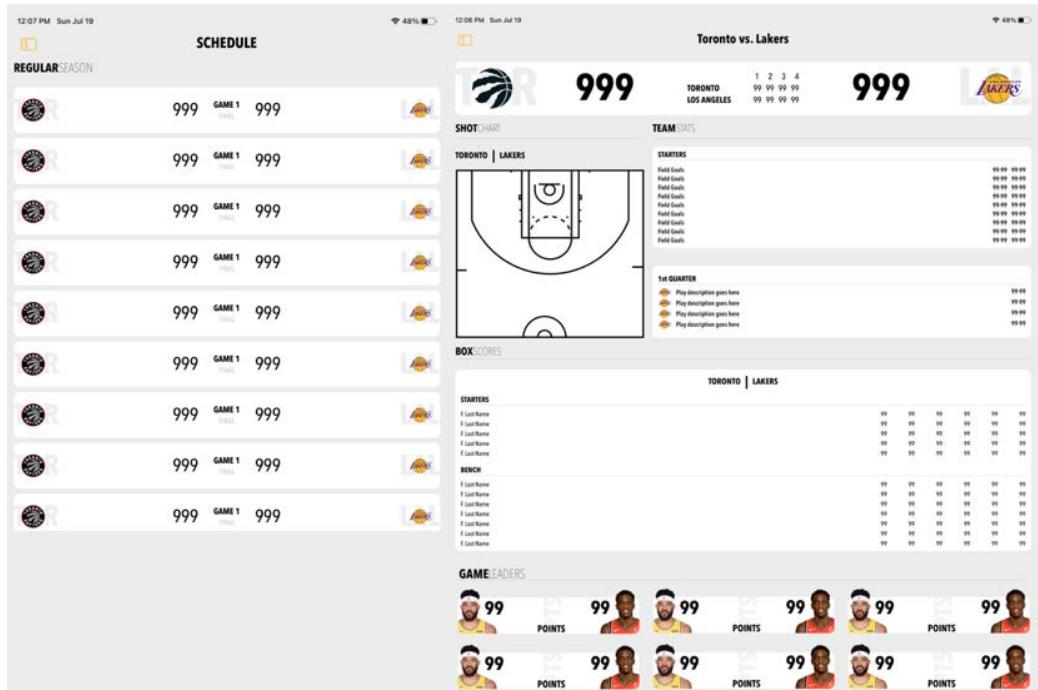


Figure 10.2

Take a few minutes and check out the colors in the `_theme` folder in the `Assets` folder. When you are ready, let's just jump in and get started by setting up our navigation first.

Navigation

In this app, we are going to use a sidebar as our navigation. Even though we won't be covering macOS development, utilizing a sidebar will make your iPad app compatible if you ever wanted to support macOS. Now, please keep in mind that you will still have to work; it would automatically work, but it will be close. Let's create our navigation using the following steps.

Creating a sidebar

Our navigation will only have three pages: the **dashboard**, **roster**, and **schedule**. Inside of the `Navigation` folder, open `AppSidebarNavigation` and let's first create an enum to represent each of these pages.

Inside the struct, above the `body` variable, add the following `enum`:

```
enum NavigationItem {  
    case dashboard  
    case roster  
    case schedule  
}
```

Our enum will allow us a way to keep track of the currently selected navigation item when using the sidebar. From the designs, we know that the dashboard will be the first page we want to load.

We need to create a variable to keep track of the selected `NavigationItem`. After the enum, add the following `private` variable:

```
@State private var selection: Set<NavigationItem> =  
[.dashboard]
```

We now have what we need to create a sidebar. Let's create our sidebar now by adding the following after the `body` variable:

```
var sidebar: some View {  
    List(selection: $selection) { // (1)  
        // Add next step here  
    }
```

```
    .listStyle(SidebarListStyle()) // (2)  
}
```

We now have our sidebar created, but before we move on, let's break down the code we just added:

1. First, we create a new view called `sidebar`. We will add this to our main code shortly. Here, we are creating a list and setting the selection to our `selection` variable.
2. In order to make our list be a sidebar, we need to set `.listStyle` to `SidebarListStyle()`.

When creating a sidebar, we need to add a `NavLink` for each link we need. Let's add our first one now by replacing `// Add next step here` with the following:

```
NavLink(destination: HomeDashboardView()) { // (1)  
    Label("DASHBOARD", systemImage: "rectangle.3.offgrid.fill")  
    // (2)  
    .custom(font: .bold, size: 18)  
}  
.accessibility(label: Text("Dashboard")) // (3)  
.tag(NavigationItem.dashboard) // (4)  
  
// Add next step here
```

Now, that we have this added, let's discuss the steps:

3. Here, we have a `NavLink` that has its destination set to `HomeDashboardView()`.
4. `Label` is set to "DASHBOARD" and we set `systemImage` to an off-grid rectangle. We also set the font to `.bold` with a font size of 18.
5. For accessibility, we set the label to read "Dashboard".
6. Finally, we set our tag to `NavigationItem.dashboard`.

We now need to add two more `NavLink`. Let's add the roster link next. Replace `// Add next step here` with the following:

```
NavLink(destination: RosterView()) {  
    HStack {
```

```

        Image("jersey")
            .renderingMode(.template)
            .padding(.leading, 5)
            .padding(.trailing, 10)
        Text("ROSTER")
            .custom(font: .bold, size: 18)
    }
}

.accessibility(label: Text("Roster"))
.tag(NavigationItem.roster)

// Add next step here

```

We have a similar setup except we are using a custom image instead of `systemImage`. Our custom image is inside an `HStack` component with a `Text` view. Let's add our last `NavigationLink` by replacing `// Add next step here` with the following code:

```

NavigationLink(destination: ScheduleView()) {
    Label("SCHEDULE", systemImage: "calendar")
        .custom(font: .bold, size: 18)
}
.accessibility(label: Text("Schedule"))
.tag(NavigationItem.schedule)

```

We now have our sidebar created. Let's add it to our body next. Inside of the `body` variable, replace `"Text("App Sidebar Navigation")"`:

```

NavigationView { // (1)
    sidebar // (2)
        .navigationBarTitle(Text("LAKERS"), displayMode:
        .inline) // (3)
        .custom(font: .bold, size: 24)

    HomeDashboardView() // (4)
}

```

Our code is added for our sidebar. Let's discuss what we added inside of the `body` variable:

1. We need to wrap a `NavigationView` around our `sidebar` in order for our links to work.
2. Here, we are adding the `sidebar` variable that we created earlier.
3. We are adding a `.navigationBarTitle` to our `sidebar` variable. Here, we set the title to `LAKERS` and `displayMode` to `inline`. We also set our font to `.bold` and the size to `24`.
4. Finally, we add `HomeDashboardView()` inside of our `NavigationView`.

We need to do one more thing before we can run the simulator. Open `ContentView()` and replace `Text ("Content View")` with `AppSidebarNavigation()`. Now, run the app and you should see the **Home Dashboard View** text:

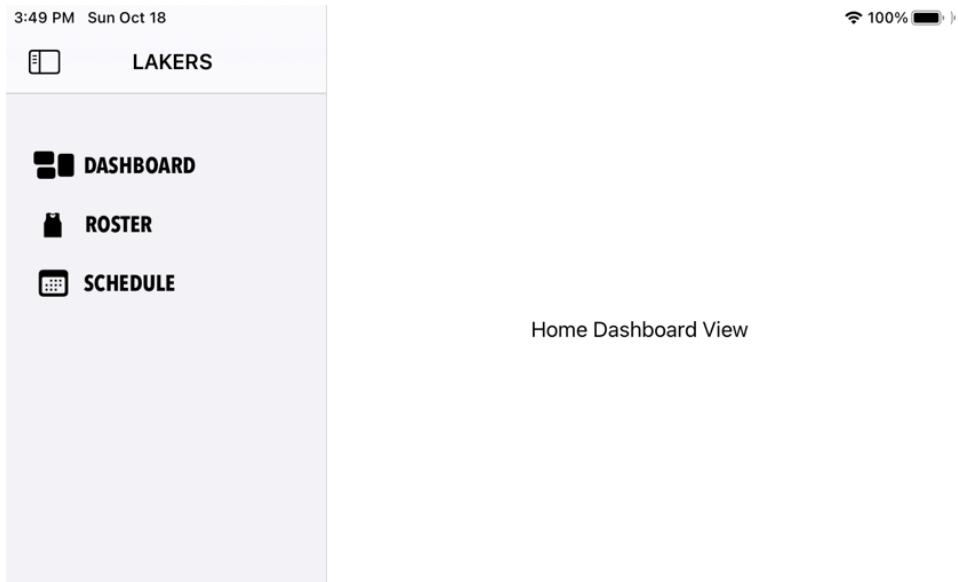


Figure 10.3

Note

At the time of writing this book, all of the code that was run was with Xcode Beta. Adding `HomeDashboardView()` inside of `NavigationView` is the only way I can get the dashboard to load at first launch. You might not need to add this when Xcode is finally released. Run the app without that line and check to see whether **Home Dashboard View** appears.

Prototyping our app with boxes

Now, we are going to do something a bit different. When working on this app for the book, I found that before creating each view, it was easier to create each section represented by a simple box. When I first built the app, I did not use this approach and I had to go back and redo each one. I find this approach helps with understanding how the iPad works with each size. In the project settings, I disabled portrait mode, but the app will work just fine if you want to enable it. Before we start adding our boxes, let's create our `HeaderView` so that we can see each section with their titles.

HeaderView

We are going to work on the header view, but before we code, let's take a minute and look at the design:



Figure 10.4

One thing to note is that our headers sometimes say **VIEW ALL** and sometimes they don't. Let's make our `HeaderView` work so that we can hide and show the **VIEW ALL** button.

Open `HeaderView`, which you can find inside of the `Supporting Views` folder inside of the `Misc` folder/. Above the `body` variable, add the following variables:

```
let title: String
let subtitle: String
var showViewAll: Bool = true
```

You will now see that `previews` is giving us an error. Update `HeaderView()` with the following:

```
static var previews: some View {
    Group {
        HeaderView(title: "Featured", subtitle: "Article",
                   showViewAll: true)
        HeaderView(title: "Featured", subtitle: "Player",
                   showViewAll: false)
    }.previewLayout(.fixed(width: 800, height: 80))
}
```

Now, we have two `HeaderView` inside `previews`, so we can see when `showViewAll` is visible and when it's not.

Right now, we are still seeing “**Header View**”. Let's update this by replacing `Text ("Header View")` with the following:

```
HStack(alignment: .bottom) {  
    // Add next step here  
}
```

Now, we need to add our text inside of the `HStack` component. Replace `// Add next step here` with the following:

```
HStack(spacing: 0) {  
    Text(title.uppercased()).custom(font: .bold, size: 20)  
    Text(subtitle.uppercased())  
        .custom(font: .light, size: 20)  
    Spacer()  
}  
  
// Add next step here
```

We added two `Text` views and used a spacer to push them to the left. We now just need to add the **VIEW ALL** logic.

Replace `// Add next step here` with the following code:

```
if showViewAll { // (1)  
    HStack {  
        HStack(spacing: 0) { // (2)  
            Text("VIEW").custom(font: .bold, size: 12)  
            Text("ALL").custom(font: .light, size: 12)  
            Image("viewall-arrow")  
        }  
        .offset(y: -2)  
    }  
}
```

We are now done with adding our code for our `HeaderView`. Let's discuss this code:

1. Here, we are checking whether `showViewAll` is true. If it is true, we show **VIEW ALL**.
2. We use an `HStack` component for our main container and add two `Text` views with an `Image` view.

Run Previews and you should now see two headers: one with **VIEW ALL** and the other without:

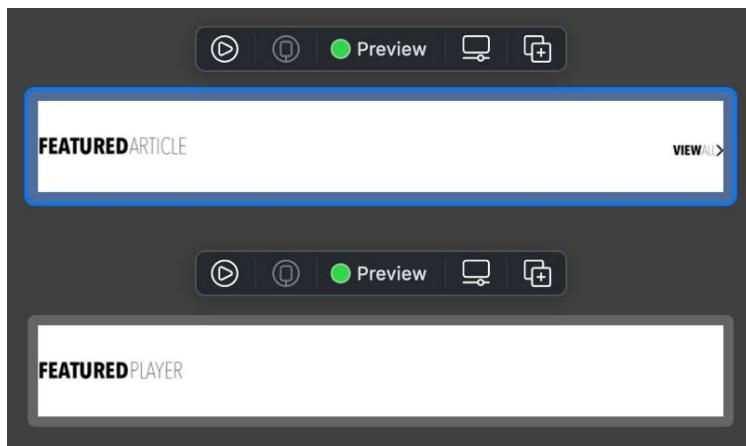


Figure 10.5

Now, before we move on to prototyping our app, here is a quick challenge for you.

Quick challenge

We set up our `HeaderView` to show and hide the **VIEW ALL** button, but what if our designer comes back and says that some of the headers should not always say **VIEW ALL**? Update your `HeaderView` so that you can customize the **VIEW ALL** text. Remember to update previews as well. You can find the solution for `HeaderView` in Challenge 1 in the Challenge folder or in the Completed folder for this chapter.

Dashboard

We completed the header view; we now need to prototype the app with boxes so that we can work with and see our layout. I think this is really helpful for the iPad, especially since users can drag in other apps and multi-task. It is also helpful to know how to make your iPad app adapt to different sizes. We won't cover every aspect, but this app example should help you become much more comfortable with dealing with multi-tasking and resizing.

We saw the designs earlier, but let's take a look at `HomeDashboardView` as a prototype instead and see what we can do first:

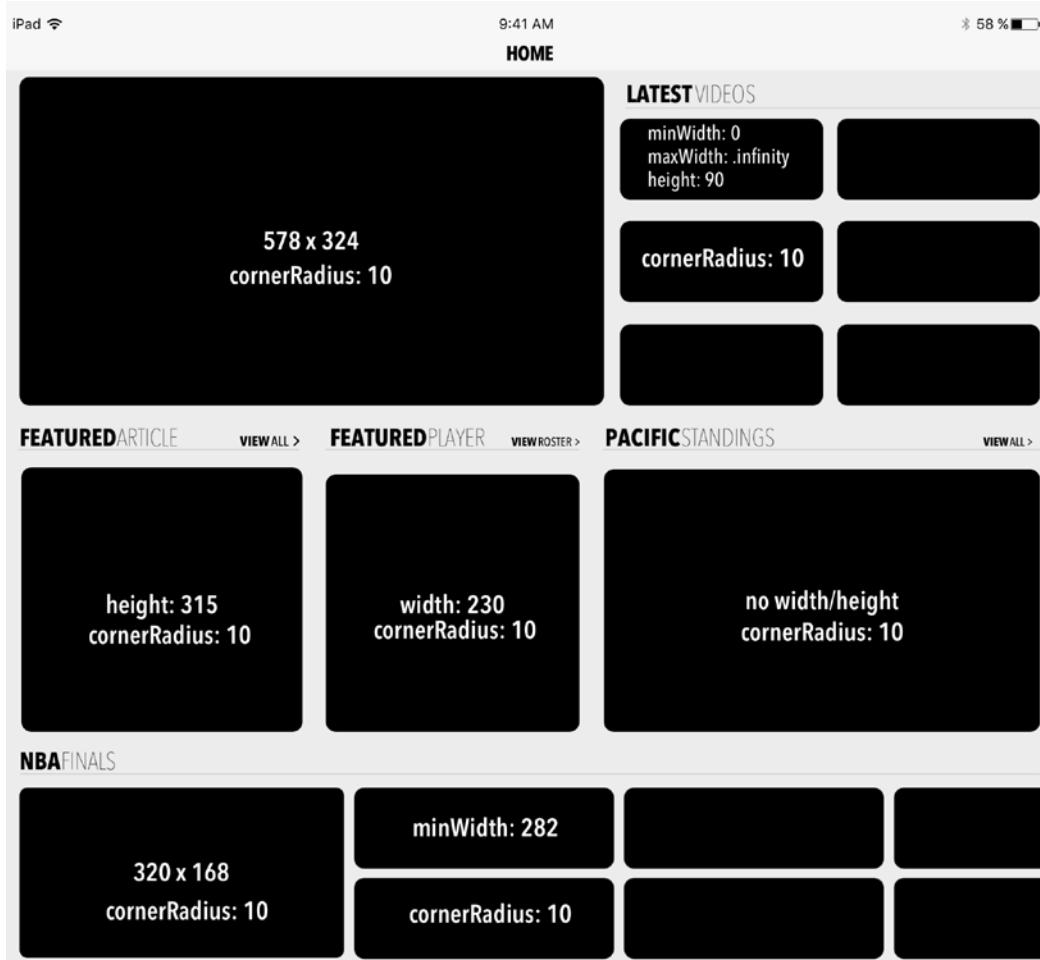


Figure 10.6

First of all, we can separate the dashboard into three sections – top, middle, and bottom. Let's begin by setting up those sections. In `HomeDashboardView`, inside of the `body` variable, replace `Text ("Home Dashboard View")` with the following code:

```
ZStack { // (1)
    Color(.baseGrey) // (2)
        .edgesIgnoringSafeArea(.all)
```

```
// Add next step here
}
.navigationTitle(Text("HOME")) // (3)
.navigationBarTitleDisplayMode(.inline) // (4)
.background(CustomNavigationBar { navigationController in // (5)
    navigationController.navigationBar.titleTextAttributes =
    [NSAttributedString.Key.font: UIFont(name: CustomFont.bold.
        rawValue, size: 24)!]
})
```

Let's go over what we just added to `HomeDashboardView`:

1. We are using a `ZStack` component as our main container.
2. Inside of our `ZStack` container, we set the background color of our app.
3. We add a navigation title.
4. We set the display mode of the title to `.inline`. This means we won't have a big title to start and when you scroll, it will animate into the title bar.
5. Finally, we have our background set to `CustomNavigationBar`, which lets us have custom fonts in our navigation bar. `CustomNavigationBar` was added to your starter project as a simple class.

Next, we need containers for our three rows. Before we add anything inside of the `ZStack` container, let's create three row containers. After the body declaration, add the following variables with a `Text` view inside each:

```
var topRow: some View {
    Text("Top row")
}

var middleRow: some View {
    Text("Middle Row")
}

var bottomRow: some View {
    Text("Bottom Row")
}
```

We now have our three containers. Let's get them to display. Inside of the body variable, replace `// Add next step here` with the following code:

```
ScrollView {  
    VStack(alignment: .leading) {  
        topRow  
        middleRow  
        bottomRow  
    } .padding()  
}
```

We added a `ScrollView` that scrolls vertically and made sure that the alignment for each row is `.leading` inside of the `VStack` component. Finally, we just added some padding to our `VStack` component. If you open Previews, you will see the following:

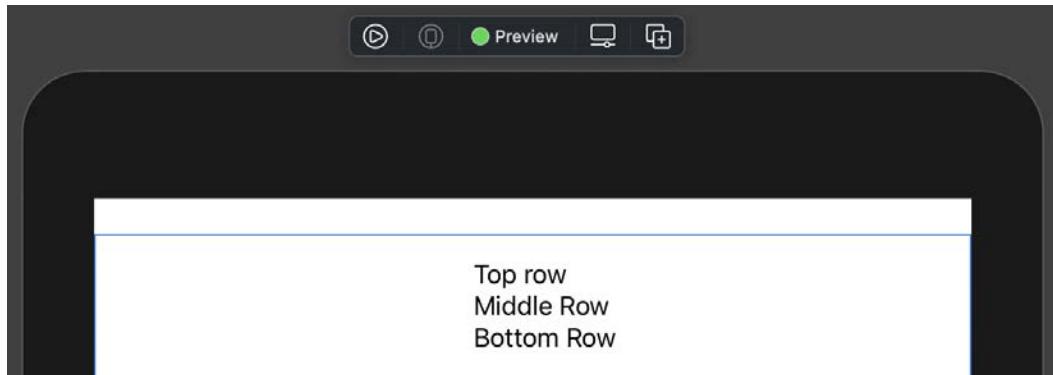


Figure 10.7

Currently, Xcode only supports portrait mode in Previews, but we can force it to be in landscape mode. Inside of the `previews` variable, add the following modifier:

```
.previewLayout(.fixed(width: 1187, height: 1034))
```

Now, if you click resume in Previews, you should see the following:

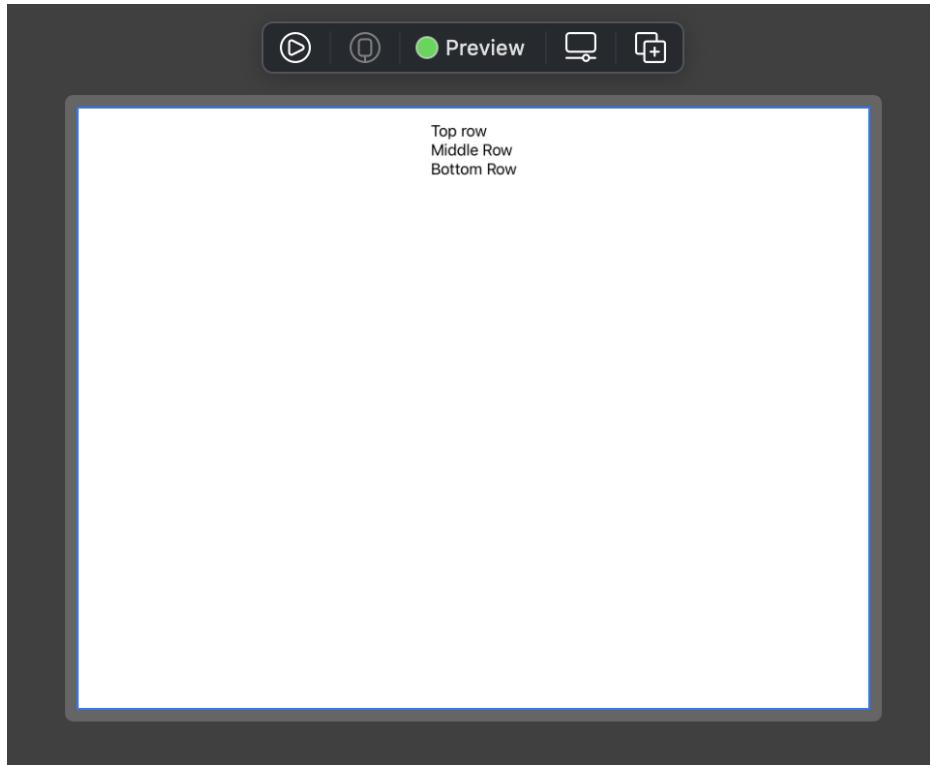


Figure 10.8

We now have our three containers set up. Let's move on to updating each one.

Creating our top row

We first need to create our first row. Before we jump in and start coding again, let's look at it one more time:

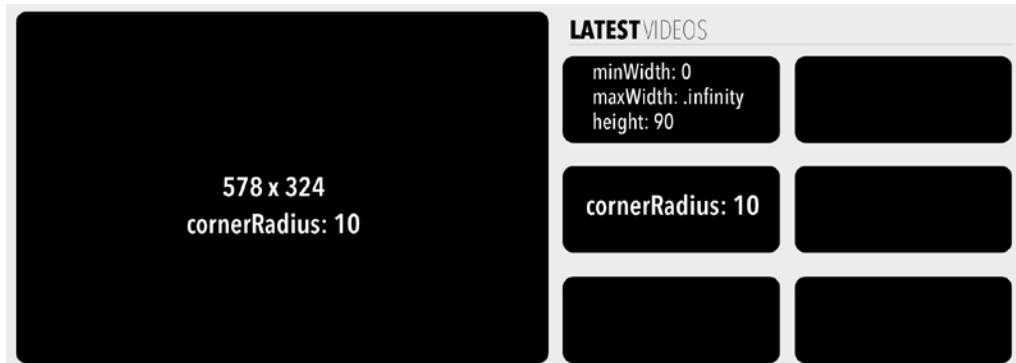


Figure 10.9

We need to create a spot for our video player and a spot for the thumbnail grid. Our video player size will not change but the thumbnail grid will adjust according to the remaining space. Inside of the `topRow` variable, replace `Text ("Top row")` with the following `VStack`:

```
VStack { // (1)
    HStack { // (2)
        Rectangle() // (3)
            .cornerRadius(10)
            .frame(width: 578, height: 324)

        VStack { // (4)
            VStack(spacing: 0) { // (5)
                HeaderView(title: "LATEST", subtitle: "VIDEOS",
                showViewAll: false)
                Divider().padding(.bottom, 3)
            }

            LazyVGrid(columns: Array(repeating: .init(
                flexible()), count: 2), spacing: 6) { // (6)
                ForEach(0..<6) { _ in
                    Rectangle() // (7)
                        .cornerRadius(10)
                        .frame(minWidth: 0, maxWidth:
                            .infinity)
                        .frame(height: 90)
                }
            }
        }
    }
}
```

Let's look at what we did in this code snippet:

1. We are using a `VStack` component as our main container.
2. Inside of the `VStack` container, we have an `HStack` component that holds our video player and the grid.

3. Here, we are using this rectangle to represent our video player.
4. We create another `VStack` component, which holds the header view and the grid.
5. Inside of the `VStack` component, we created another `VStack` component, which holds our header view and divider.
6. `LazyVGrid` is creating a vertical grid that is flexible. Our grid has two columns with a spacing of 6 pixels. By setting our grid to be flexible, we let the thumbnails adjust their size based on the available space.
7. Our rectangle represents our video thumbnails. We set the frame height to 90 but we also set `.minWidth` to 0 and `.maxWidth` to `.infinity`. Setting these values makes it so that the thumbnails can adjust their size.

When you click resume in Previews, you will see the following:

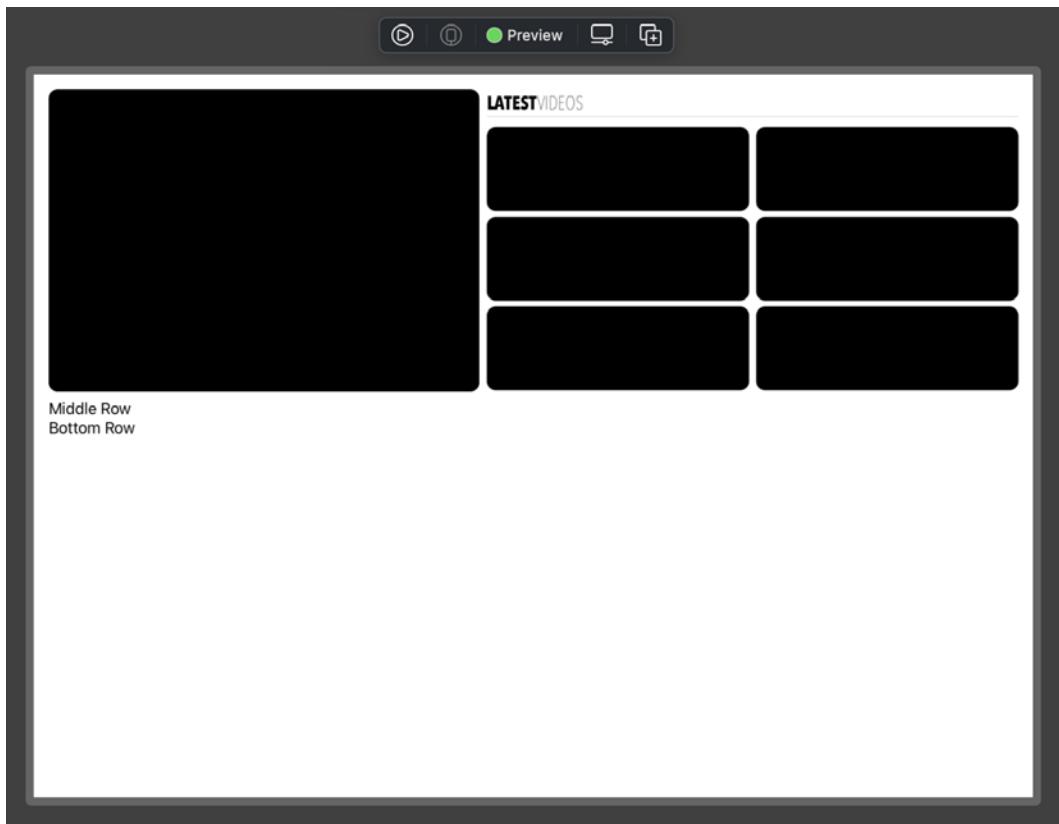


Figure 10.10

You can also run this in the simulator, and you will see that when the sidebar is open, our thumbnails will shrink in size. When you close the sidebar, you will notice that they will get bigger. We have the behavior we want, but take the time to really mess with the values that we entered and see how things adjust.

We now need to move on to our middle row.

Creating our middle row

Our middle row is next, but before we start, let's take a quick look at what we need this time:

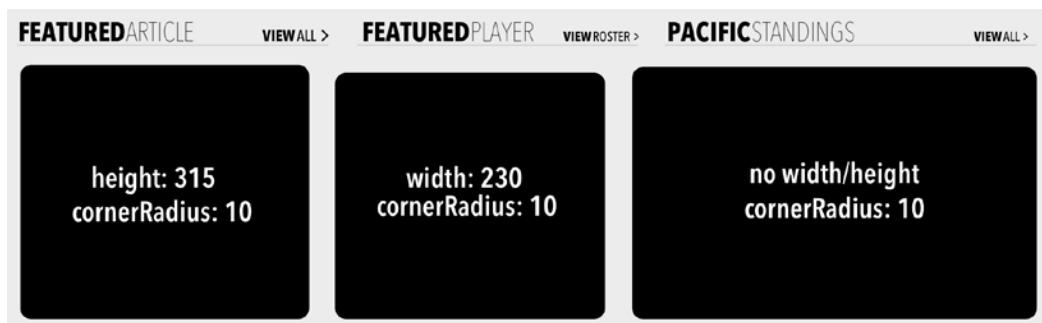


Figure 10.11

We have three sections, each with slightly different specs. We want the **STANDINGS** section to be the one to have a flexible width and the other two to always be the same. In the middle row code, replace `Text ("Middle Row")` with the following:

```
HStack(alignment: .top, spacing: 20) {
    VStack {
        HeaderView(title: "FEATURED", subtitle: "ARTICLE",
        showViewAll: false)
        Rectangle()
            .frame(height: 315)
            .cornerRadius(10)
    }

    VStack {
        HeaderView(title: "FEATURED", subtitle: "PLAYER",
        showViewAll: false)
        Rectangle()
    }

    VStack {
        HeaderView(title: "PACIFIC", subtitle: "STANDINGS",
        showViewAll: false)
        Rectangle()
    }
}
```

```
    .cornerRadius(10)
}
.frame(width: 230)

VStack {
    HeaderView(title: "PACIFIC", subtitle: "STANDINGS",
        showViewAll: false)
    Rectangle()
        .cornerRadius(10)
}
}
```

We do not have a lot of code here but let's just talk about what we added:

1. We are using an `HStack` component as our main container.
2. Inside of our `HStack` container, we have a `VStack` component, which contains a `HeaderView` and a `Rectangle` with its height set to 315.

We have another `VStack` component, which has no frame set for the rectangle, but you will see that we have a frame set on the actual `VStack` component. By adding the frame to the container, it allows us a flexible row and will make **STANDINGS** grow based on the available space.

3. Finally, we have `Standings`, which will take whatever space and has a flexible width.

Click resume in Previews and you will see the following:

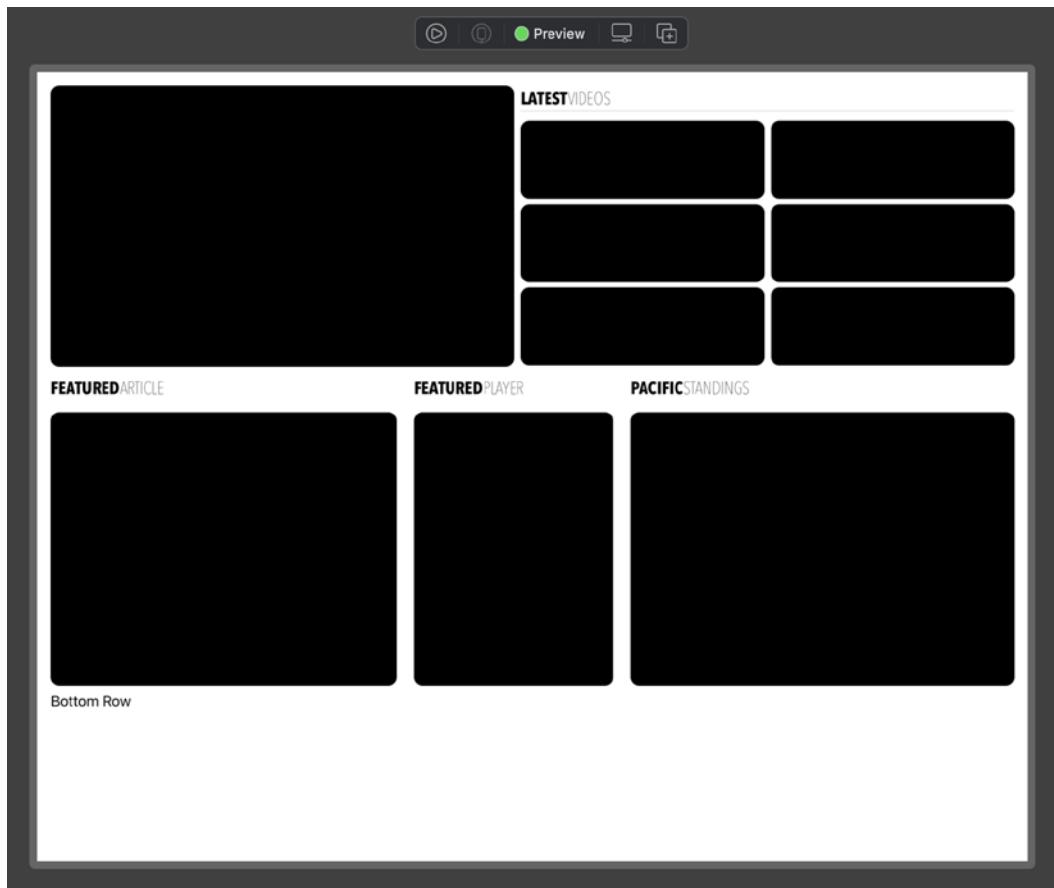


Figure 10.12

We just need to do our bottom row. We will walk through this one together because it is a bit tricky.

Creating our bottom row

We are moving on to our last row. Let's take a look at what we are going to prototype this time:



Figure 10.13

Our bottom row looks simple because it looks like we have a box and then a grid next to it. Well, there is one caveat here and that is that a large box will act as a sticky header, which means all of the smaller boxes will scroll underneath the header. It is actually easy once you see the code. Inside `bottomRow`, replace `Text ("Bottom Row")` with the following:

```

VStack(alignment: .leading) { // (1)
    VStack(spacing: 0) { // (2)
        HeaderView(title: "NBA", subtitle: "FINALS",
        showViewAll: false)
        Divider().padding(.bottom, 5)
    }

    ScrollView(.horizontal) { // (3)
        HStack(alignment: .top) {
            LazyHGrid(rows: [GridItem(.adaptive(minimum: 80))],
            spacing: 6, pinnedViews: [.sectionHeaders]) { // (4)
                Section(header: Rectangle()
                    .frame(width: 320, height: 168)
                    .cornerRadius(10)) { // (5)
                    ForEach(0..<7) { _ in
                        Rectangle()
                            .frame(width: 282) // (6)
                            .cornerRadius(10)
                    }
                }
            }
        }.frame(height: 168) // (7)
    }
}

```

Let's discuss what we just added inside our bottom row:

1. We are using a `VStack` component as our main container.
2. Inside of the `VStack` container, we have our `HeaderView` and `Divider`.
3. We create a `ScrollView`, which will hold our playoff module and will allow the user to scroll horizontally.
4. Here, we create a `LazyHGrid`, which has an adaptive minimum size of 80. We set the spacing to 6 pixels and finally, we set `pinnedViews` to `[.sectionHeaders]`. We have now told our `LazyHGrid` that we will have pinned section headers. If you wanted to add sticky or pinned footers, you would just add `.sectionFooters` to the array of `pinnedViews`.
5. We add a `Section` inside of our `LazyHGrid` and set `header` to our rectangle.
6. Next, we set the width of each item in the grid.
7. Finally, we set the height of the `HStack` component to 168.

When you are ready, hit resume in Previews and you will see the following:

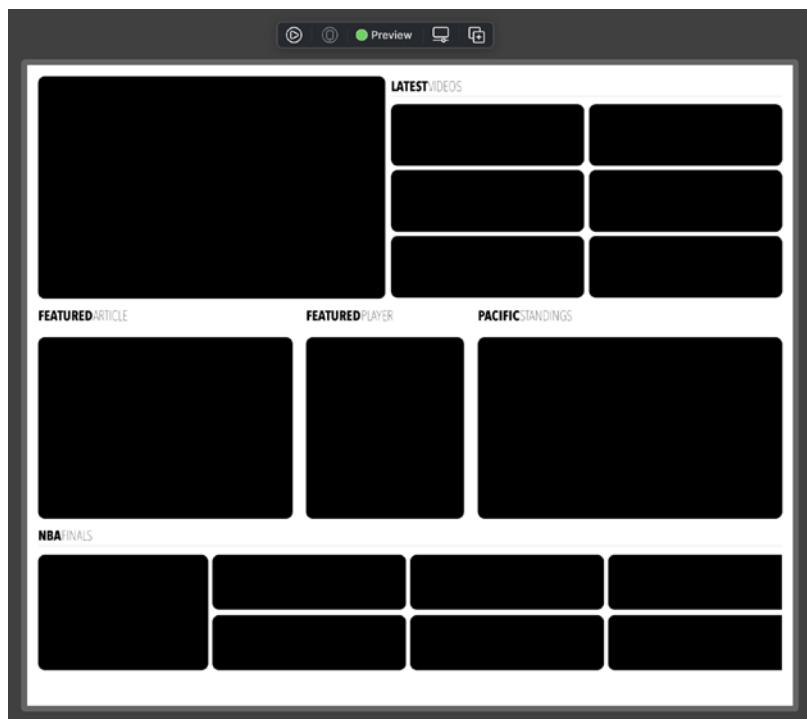


Figure 10.14

Before we move on to the next challenge, let's do a couple more to make sure you have gotten the hang of it.

Roster prototyping

We need to prototype the roster view next. Let's take a look at what we are going to prototype in `RosterView`:

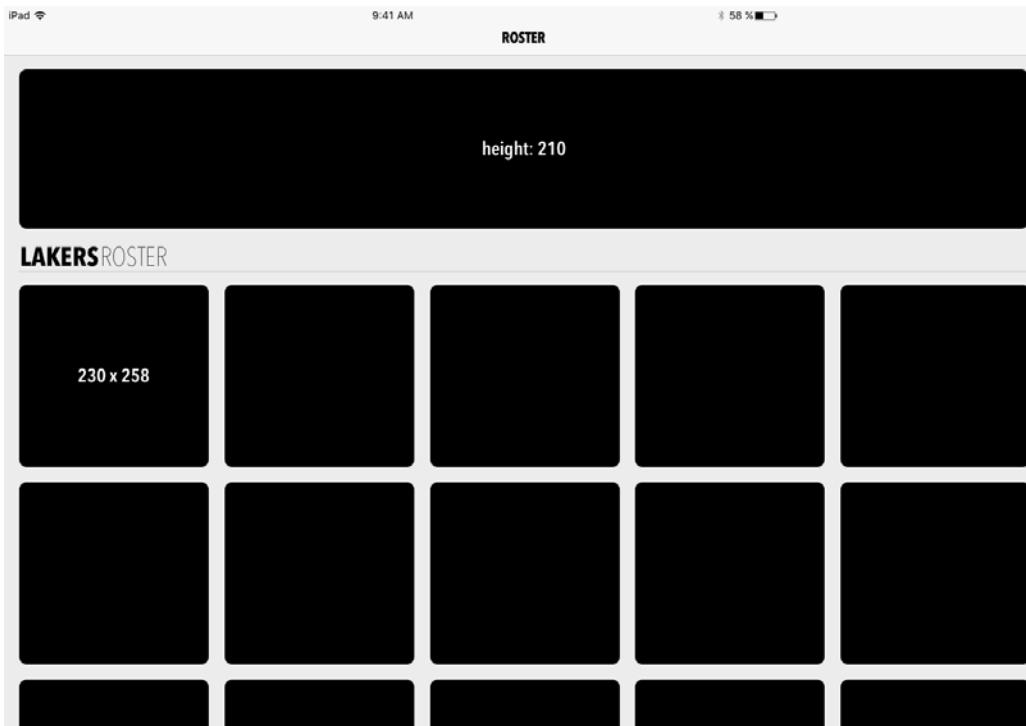


Figure 10.15

We have a section that expands the entire width and then under that, we have a grid of players. We will not be pinning the header in this section; the entire view will scroll together. Let's open `RosterView` and get started. Replace `Text ("Roster View")` with the following:

```
zStack {
    Color(.baseGrey)
    .edgesIgnoringSafeArea(.all)

    // Add next step here
```

```
}

.navigationTitle(Text("ROSTER"))
.navigationBarTitleDisplayMode(.inline)
```

We are using a `ZStack` component as our main container. We also set the title to `"ROSTER"` and `displayMode` as `.inline`. Now, replace `// Add next step here` with the following:

```
ScrollView {
    VStack {
        // Add next step here
    }
}
```

As we saw earlier, we want the entire view to scroll vertically. Now, go ahead and replace `// Add next step here` with the following:

```
Rectangle() // (1)
    .cornerRadius(10)
    .frame(height: 210)
    .padding(.horizontal, 10)

VStack(spacing: 0) { // (2)
    HeaderView(title: "Lakers", subtitle: "Roster",
    showViewAll: false)
    .padding(.horizontal, 10)
    Divider()
    .padding(.horizontal, 10)
}

// Add next step here
```

Let's talk about what we just added here:

1. We are adding our header section, which expands the entire width of the screen. We only needed to set the height and that's all we need to make this have a flexible width.
2. Here, we have a `VStack` container that holds our `HeaderView` and `Divider`.

We just need to add our grid now. Replace // Add next step here with the following code:

```
LazyVGrid(columns: [GridItem(.adaptive(minimum: 250), spacing: 4)], spacing: 34) {
    ForEach(0..<15) { _ in
        Rectangle()
            .cornerRadius(10)
            .frame(width: 230, height: 258)
    }
}
.padding(.horizontal, 5)
.padding(.vertical, 30)
```

We now have our `LazyVGrid` setup and we have a `Rectangle`, which will be our player cards. Before we look at the preview, let's update `preview` to display the roster in landscape mode by adding the following to `RosterView()` in `previews`:

```
.previewLayout(.fixed(width: 1187, height: 1034))
```

Now, hit resume and you will see the following:

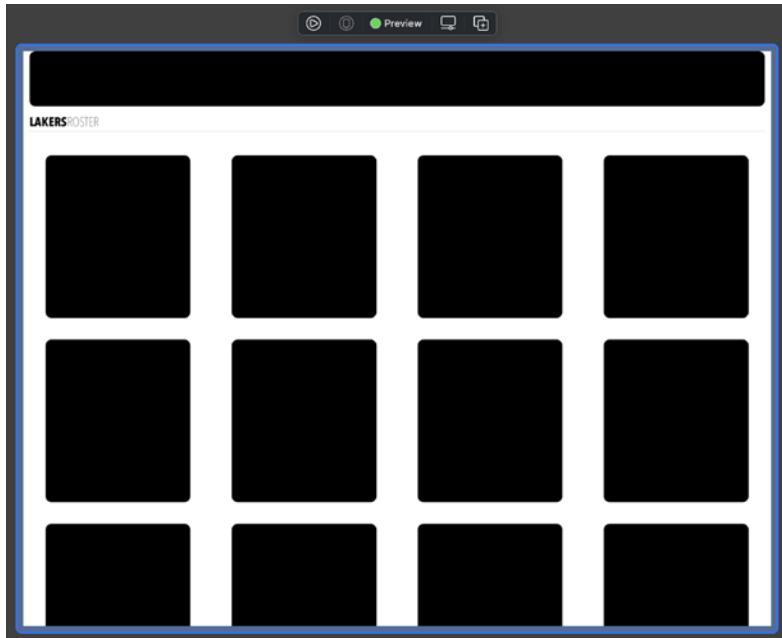


Figure 10.16

Our roster section is complete. Let's do one more section together and then I will give you another challenge.

Schedule prototyping

Our last view that we will prototype together will be the **SCHEDULE** section. Let's take a minute and look at what we need to do:

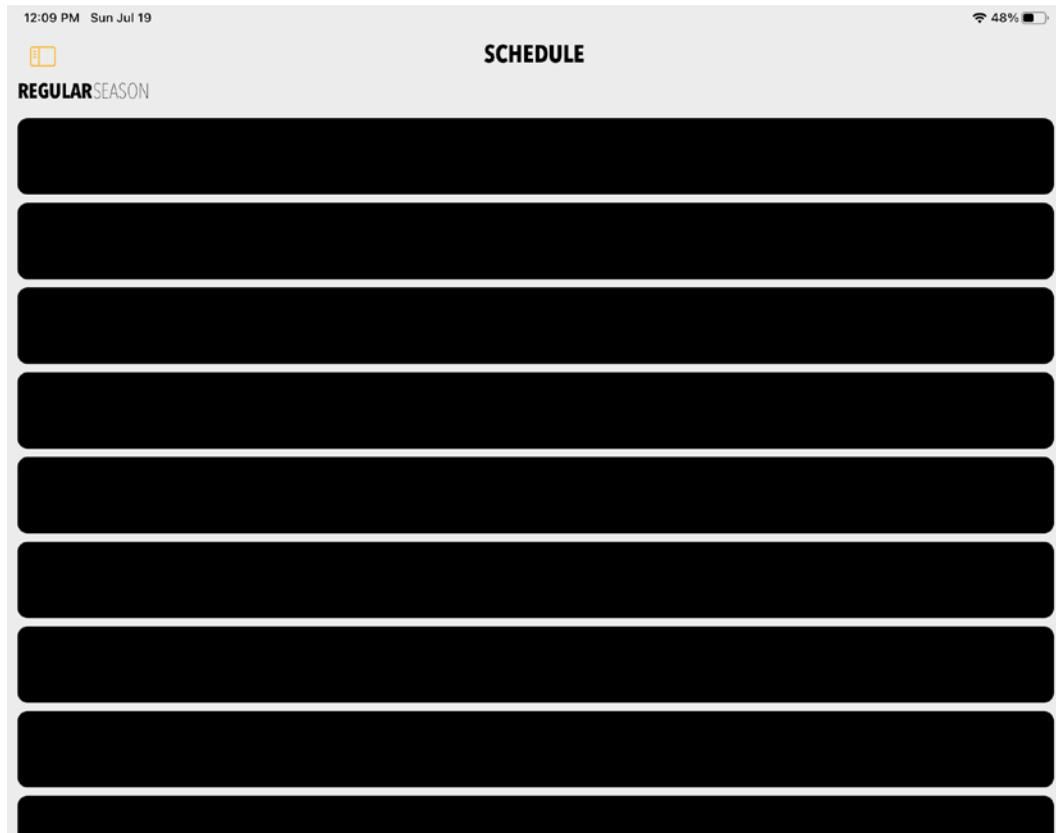


Figure 10.17

We have a vertical list that scrolls with a header. Let's get started by opening up `ScheduleView` and update `preview` to display in landscape mode by adding the following to `ScheduleView()` inside of `previews`:

```
.previewLayout(.fixed(width: 1187, height: 1034))
```

Next, replace `Text ("Schedule View")` with the following code:

```
ZStack {
    Color(.baseGrey)
    .edgesIgnoringSafeArea(.all)

    ScrollView {
        LazyVStack {
            // Add next step here
        }.padding(.horizontal, 10)
    }
    .navigationTitle(Text("SCHEDULE"))
    .navigationBarTitleDisplayMode(.inline)
}
```

We have done this a few times already, but we have a `ZStack` and `ScrollView` component. Now, let's add a section by replacing `// Add next step here` with the following:

```
Section(header: HeaderView(title: "REGULAR", subtitle:
    "SEASON", showViewAll: false)) {
    ForEach(0..<10) { _ in
        Rectangle()
            .cornerRadius(10)
            .frame(height: 74)
    }
}
```

We added a `HeaderView` to our section and used a `ForEach` to create 10 rectangles. We do not need to do anymore with this section until we have data to use. Let's move on to the next challenge.

Prototyping the game details

Now that we have set up a few sections together, you will now prototype the game details screen on your own. Use the skills we covered earlier to set up the game details screen. If you get stuck, you can find `GameDetailView()` in the `Challenge_2` folder in the `Challenge` folder. Here is a look at the game details prototype view:



Figure 10.18

I broke the view down into four rows, so here is the info you need for doing this on your own. All rectangles will have a `cornerRadius` value of 10. Here is the info for each row.

The following are the details for row 1:

- height: 100

The following are the details for row 2:

- **Section 1:**

width: 355

title: SHOT

subtitle: CHART

- **Section 2:**

title: TEAM

subtitle: STATS

Rectangle 1: height: 190

Rectangle 2: height: 135

The following are the details for row 3:

- height: 340

The following are the details for row 4:

- title: GAME
- subtitle: LEADERS
- .adaptive (minimum: 235)
- spacing: 8
- height: 80

There is a lot going on but I am throwing a lot at you so that you can see how much you are able to do on your own. Before you look at the solution, take your time and try and work it out. I did not get this right on my first attempt and I do not expect you to do it on your first try either. If you are able to get it, then that is great. Remember, you can use the design specs if you feel more comfortable with this section.

Designing dashboard module views

We have prototyped our entire app and now it is time to slowly add in each design element. As we have done throughout the book, we will first do this together and then I will challenge you to do it on your own. Let's first move to `HomeDashboardView` and work on this view first.

Video player with a grid

We know from our design that our `HomeDashboardView` has a video player with a grid. We already have the base of this created; we just need to swap out a couple of things to make this work.

Open `HomeDashboardView` and copy the `VStack` component out of the `topRow` variable and paste it into the `body` variable in `VideoModuleView`. Now, update `previews` to the following:

```
.previewLayout (.fixed(width: 1024, height: 350))
```

Inside of `VideoModuleView()`, locate the following `Rectangle()`, which we just copied over from `HomeDashboardView`:

```
Rectangle()  
    .cornerRadius(10)  
    .frame(width: 578, height: 324)
```

Replace it with the following code:

```
CustomVideoPlayer(urlPath: $path)  
    .frame(width: 578, height: 324)  
    .mask(RoundedRectangle(cornerRadius: 8)  
        .background(Color.clear)  
        .frame(width: 578, height: 324))
```

Apple does have a video player that works fine on a device running iOS 14 but there are some issues with it in the simulator. I added `CustomVideoPlayer` to your files so that you can see how I created a video player using `UIViewControllerRepresentable`. In *Chapter 11, Sports News App – Data*, we will make the video player work with the data.

Open `HomeDashboardView()` and delete `topRow` as this will get rid of one of the errors. Next, in the `body` variable, replace `topRow` with `VideoModuleView()`. If you build the project, you will see that the video player is working:

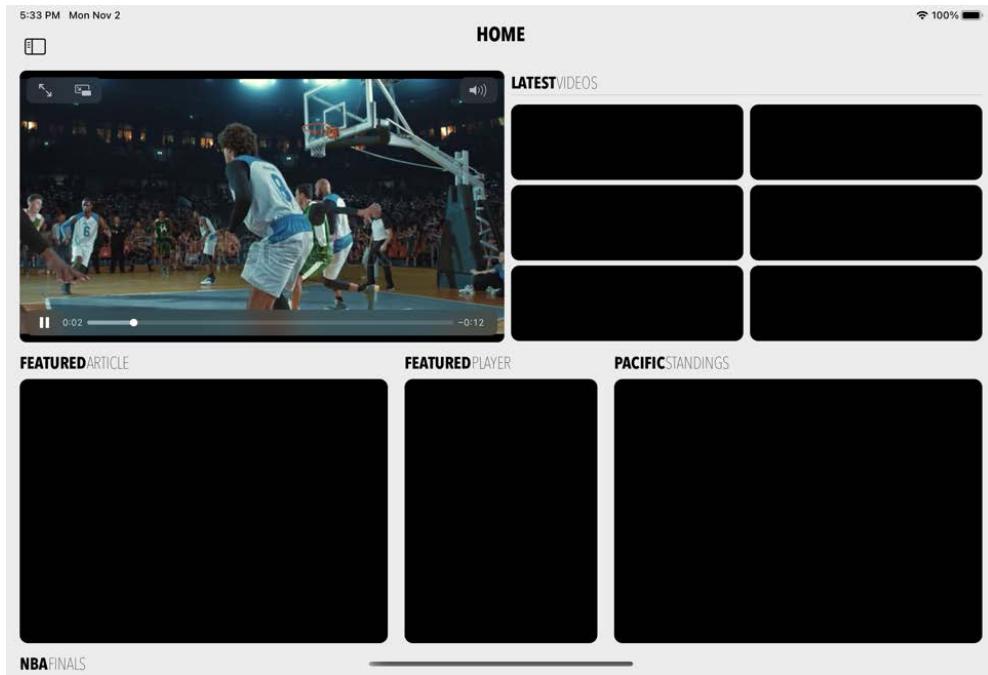


Figure 10.19

We created a video player, but we also added a mask around it so that we could add rounded corners to it. Let's create our video item next.

Creating a video item

Now, we need to work on creating video items. Open `VideoItem`, which is in the `Home` folder inside `Supporting Views`. Then, replace `Text ("Video Item")` with the following code:

```
Image ("news-sample")
    .resizable()
    .cornerRadius(10)
    .aspectRatio(contentMode: .fill)
    .frame(minWidth: 0, maxWidth: .infinity)
    .frame(height: 90)
    .mask(Rectangle()
        .cornerRadius(10)
        .background(Color.clear)
        .frame(height: 80))
```

Go back to `VideoModuleView` and update the following:

```
Rectangle()  
    .cornerRadius(10)  
    .frame(minWidth: 0, maxWidth: .infinity)  
    .frame(height: 90)
```

You can find this rectangle inside of our `ForEach`; replace it with the following:

```
VideoItem()  
    .frame(minWidth: 0, maxWidth: .infinity)
```

When you are finished, click resume and you will see the following:

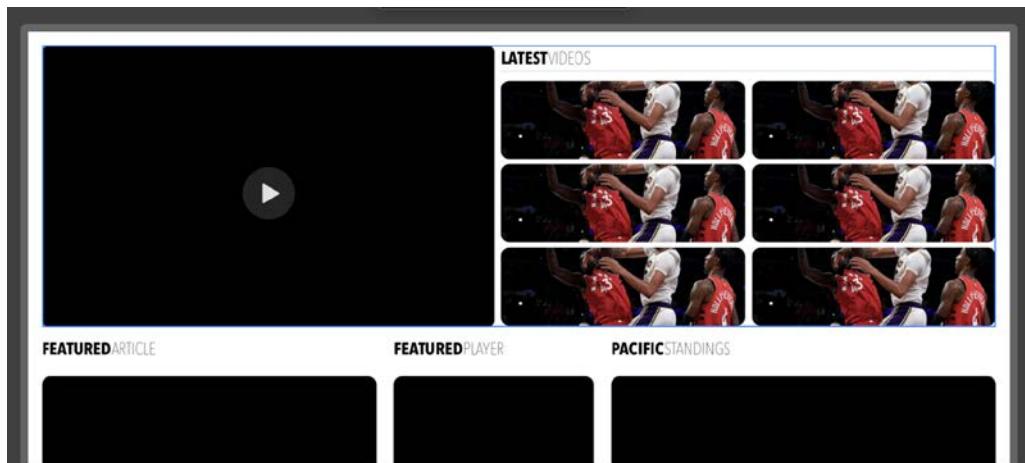


Figure 10.20

We have now completed the first row. Let's move on to the next row before you start your next challenge.

The featured news module

Let's take a look at the featured news module design before we get started:

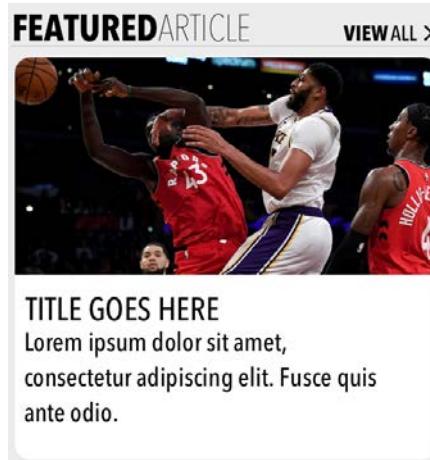


Figure 10.21

Open `FeaturedArticleModuleView`, which is located inside of the `Home` folder in the `Supporting Views` folder. Inside of the `body` variable, replace `Text ("Featured Article Module View")` with the following:

```
 VStack {
    VStack(spacing: 0) {
        HeaderView(title: "FEATURED", subtitle: "ARTICLE")
        Divider().padding(.bottom, 5)
    }

    // Add next step here

}.frame(height: 315)
```

Nothing here is new; we just have our header inside of a `VStack` component and have set the entire container's height to 315. Now, replace `// Add next step here` with the following:

```
 ZStack(alignment: .top) {
    Rectangle()
        .fill(Color.white)
        .cornerRadius(8)

    // Add next step here
}
```

We have added a `ZStack` component with a white rectangle, which is our background. Now, let's add the module content to it by replacing `// Add next step here` with the following:

```
 VStack { // (1)
    Image("news-sample") // (2)
        .resizable()
        .aspectRatio(contentMode: .fill)
        .frame(height: 140)
        .mask(Rectangle()
            .cornerRadius(10, corners: [.topLeft,
            .topRight])
            .background(Color.clear)
            .frame(height: 140))
}

VStack(alignment: .leading) { // (3)
    Text("TITLE GOES HERE")
        .custom(font: .bold, size: 21)
    Text("Lorem ipsum dolor sit amet, consectetur
        adipiscing elit. Fusce quis ante odio.
        Lorem ipsum dolor sit amet, consectetur
        adipiscing elit. Fusce quis ante odio. Lorem ipsum dolor sit
        amet, consectetur adipiscing elit. Fusce quis ante odio.
        Lorem ipsum dolor sit amet, consectetur
        adipiscing elit. Fusce quis ante odio.")
        .multilineTextAlignment(.leading)
    .lineLimit(nil)
        .custom(font: .medium, size: 16)
    }.padding(.horizontal, 10)
}
```

Let's talk about the code we just added inside of our `ZStack` component:

1. We are using a `VStack` component as our main container.
2. Next, we are adding an image and applying a mask to it. Our mask is a bit different because we are adding a corner radius only to the top right and left of the image. We are using a custom class called `RoundedCorners` to get this effect.

- Finally, we have a `VStack` component that holds our title and a brief description. We also have the description text set to have a flexible line limit.

Now, update `previews` to the following:

```
.previewLayout (.fixed(width: 400, height: 315))
```

Hit `resume` and you should see the following:

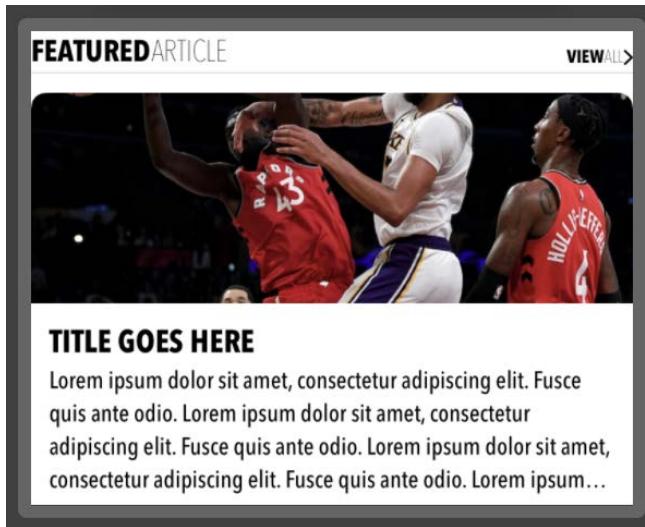


Figure 10.22

Our news module is done. Next, open `HomeDashboardView` and in the `middleRow` variable, locate the following `VStack` component:

```
VStack {
    HeaderView(title: "FEATURED", subtitle: "ARTICLE",
    showViewAll: false)
    Rectangle()
        .frame(height: 315)
        .cornerRadius(10)
}
```

Replace it with the following:

```
FeaturedArticleModuleView()
```

When you are done swapping those out, you will see the following in Previews:

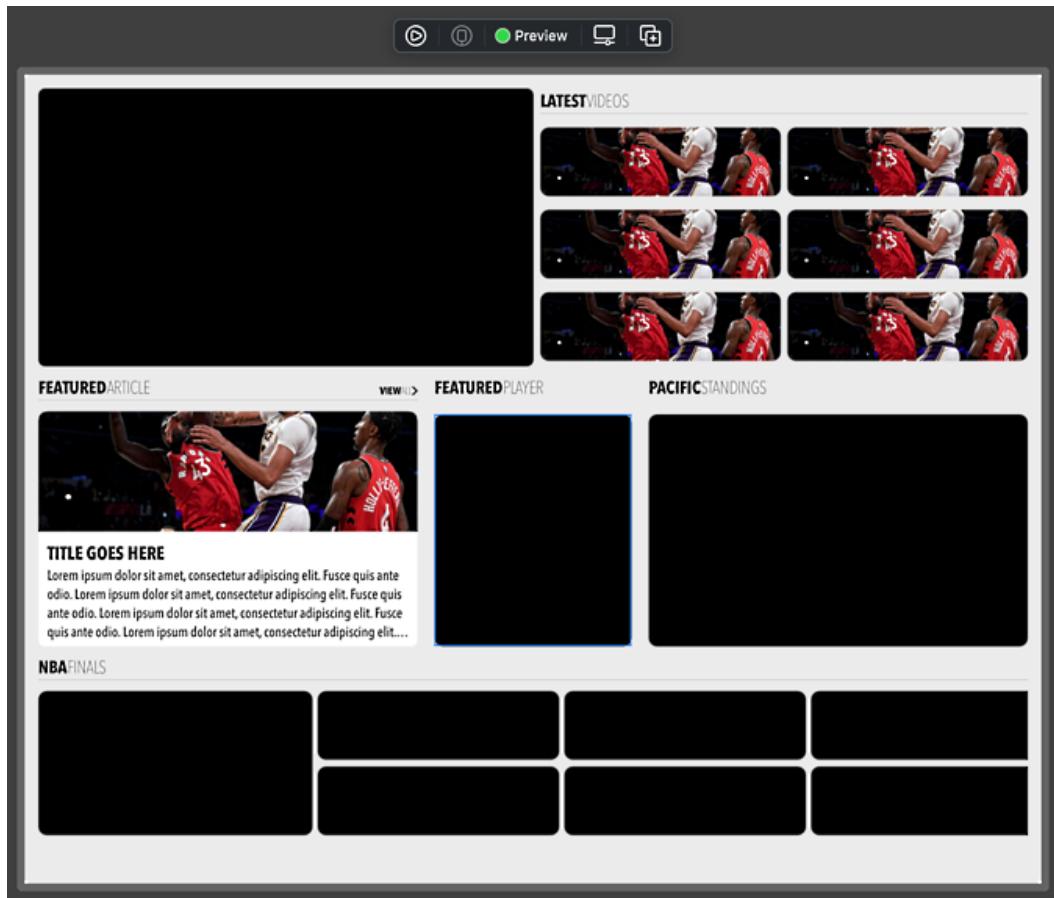


Figure 10.23

Now, let's move on to our next module and design it.

The featured player module

We are going to work on the featured player module now. Let's take a look at our design to see what we need to do:

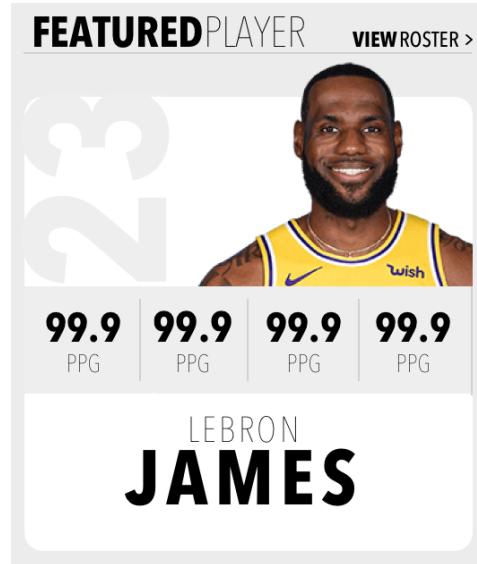


Figure 10.24

In this design, we have the player extending past the background. We also have the text displayed vertically. Let's go ahead and get started. First, open `PlayerCardView` and replace `Text ("Player Card View")` with the following:

```
ZStack {
    // Add next step here

    Image("lebron-james-full")
        .resizable()
        .frame(width: 180, height: 132)
        .offset(x: 0, y: -77)
}
```

We are using a `ZStack` component as our main container and we added our `Image` inside of it. Instead of masking this image, we are just going to add an image that's already masked. We will update this image in *Chapter 11, Sports News App – Data*. Replace `// Add next step here` with the following:

```
ZStack { // (1)
    HStack { // (2)
        Text("23")
            .custom(font: .bold, size: 112)
```

```
    .foregroundColor(Color.baseGrey)
}
.rotationEffect(.degrees(-90)).offset(x: -84, y: -70)
// (3)
// Add next step here
}
.frame(width: 230, height: 240)
.background(Color.white)
.cornerRadius(10)
```

Let's discuss what we just added:

1. We created another `ZStack` container.
2. Inside of the `ZStack` container, we added an `HStack` component, which contains a `Text` view.
3. We are using `rotationEffect` to rotate the `HStack` component.

Next, let's add the player stats. Replace `// Add next step here` with the following code:

```
 VStack { // (1)
    Spacer()

    HStack { // (2)
        Spacer()
        ForEach(0 ..< 4) { item in // (3)
            VStack(spacing: -5) {
                Text("99.9").custom(font: .bold, size: 23)
                Text("PPG").custom(font: .light, size: 14)
            }.padding(.trailing, 8)
        }

        Spacer()
    }
    .frame(height: 60)
    .background(Color.baseGrey)
```

```
// Add next step here  
}
```

We do not have a lot going on here, but let's cover a couple of things:

1. We use a `VStack` component as our main container.
2. Inside of the `VStack` container, we have an `HStack` component.
3. We are using a `ForEach` to create four stats for our featured player.

We just need to add the player's name. Replace `// Add next step here` with the following code:

```
VStack(spacing: -15) {  
    Text("LEBRON").custom(font: .light, size: 20)  
    Text("JAMES").custom(font: .bold, size: 43)  
}.frame(height: 72)
```

We are done with `PlayerCardView`.

Let's open `FeaturedPlayerModuleView()` and replace `Text ("Featured Player Module View")` with the following code:

```
VStack(alignment: .center, spacing: 0) {  
    HeaderView(title: "Featured", subtitle: "Player")  
    Divider().padding(.bottom, 30)  
    PlayerCardView()  
  
}.frame(width: 230)
```

We now have a `HeaderView` with our `PlayerCardView` and `FeaturedPlayerModuleView`. Now, open `HomeDashboardView` and locate `middleRow`:

```
VStack {  
    HeaderView(title: "FEATURED", subtitle: "PLAYER",  
    showViewAll: false)  
    Rectangle()  
        .cornerRadius(10)  
}  
.frame(width: 230)
```

Replace it with the following code:

```
FeaturedPlayerModuleView()
```

When you are finished, you will see the following in Previews:

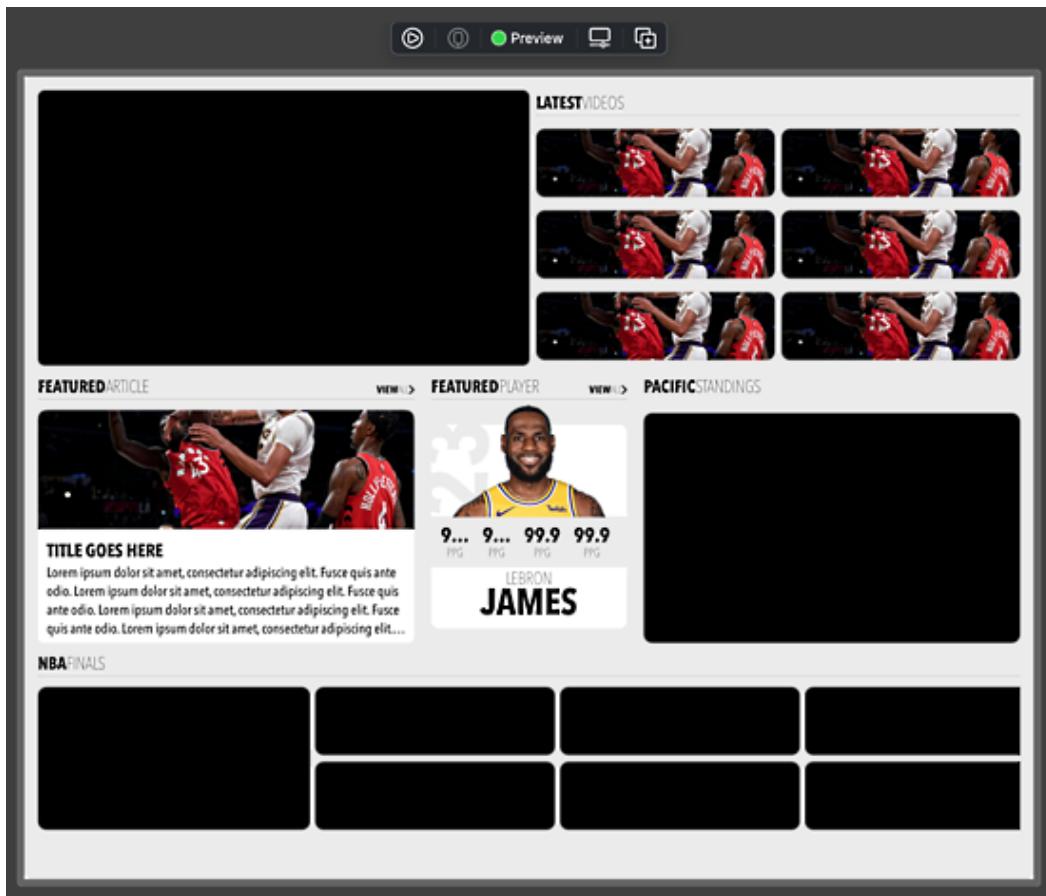


Figure 10.25

The standings module view

We are going to work on one more view together and after that, you will proceed with your last challenge of the chapter. Before we do anything, let's take a look at the design and what we need to do:

PACIFIC STANDINGS						VIEW ALL >
		W - L	PCT	GB	L10	STRK
1.	LAKERS	99 - 99	.999	-	9-9	W9
2.	CLIPPERS	99 - 99	.999	-	9-9	W9
3.	SUNS	99 - 99	.999	-	9-9	W9
4.	JAZZ	99 - 99	.999	-	9-9	W9
5.	SPURS	99 - 99	.999	-	9-9	W9
6.	ROCKETS	99 - 99	.999	-	9-9	W9
7.	TRAILBLAZERS	99 - 99	.999	-	9-9	W9
8.	TIMBERWOLVES	99 - 99	.999	-	9-9	W9
9.	MAVERICKS	99 - 99	.999	-	9-9	W9
10.	NUGGETS	99 - 99	.999	-	9-9	W9

Figure 10.26

One thing to remember is that this module will be flexible in width, so we won't be setting width to anything in these views.

We want to work on the header first. What is tricky about the header is that we want to make sure it aligns with the content below it, but this is not easy to do when resizing. Let's get started by opening `StandingsHeaderView()` and replace `Text ("Standings Header View")` with the following:

```
HStack {
    // Add next step here
}.padding(.bottom, 5)
```

We now have our main container and we want to make sure that the data always matches the headers, so we will add some empty text views and give them `minWidth` and `maxWidth` parameters. When we do this, we are able to keep the headers aligned. Replace `// Add next step here` with the following code:

```
HStack {
    ZStack {
        VStack(alignment: .leading) {
            Text("")
                .custom(font: .medium, size: 14)
```

```
        .foregroundColor(.white)
    }
}

.frame(maxWidth: 10)
.frame(height: 20)

ZStack {
    Text("")
        .frame(maxWidth: .infinity, alignment: .leading)
        .multilineTextAlignment(.leading)
        .custom(font: .medium, size: 14)
        .foregroundColor(.black)
    }
    .frame(minWidth: 80)
    .frame(height: 20)
}

// Add next step here
```

We created a couple of Text views that we are setting inside of an HStack component. These two views represent the rank and the team name. Our first Text view doesn't need to be big since it will only display numbers of no more than two digits. The other Text view is where the team will lie and this will be no less than 80 pixels in width. Now, replace // Add next step here with the following code:

```
HStack {
    ZStack {
        Text("W-L")
            .frame(maxWidth: .infinity, alignment: .center)
            .multilineTextAlignment(.center)
            .custom(font: .bold, size: 12)
            .foregroundColor(.black)
    }
    .frame(minWidth: 40)
    .frame(height: 20)

    ZStack {
```

```
Text("PCT")
    .frame(maxWidth: .infinity, alignment: .center)
    .multilineTextAlignment(.center)
    .custom(font: .bold, size: 12)
    .foregroundColor(.black)
}
.frame(minWidth: 35)
.frame(height: 20)

ZStack {
    Text("GB")
        .frame(maxWidth: .infinity, alignment: .center)
        .multilineTextAlignment(.center)
        .custom(font: .bold, size: 12)
        .foregroundColor(.black)
}
.frame(minWidth: 20)
.frame(height: 20)
}
```

We have a lot going on but it is the same code. We are creating a `Text` view for each stat. We are not doing anything special here. We need to do exactly the same thing but for the data we will display.

We are done; we just need to bring it all together. Open `StandingsModuleView`, then find `Text ("Standings Module View")` and replace it with the following:

```
 VStack(alignment: .leading, spacing: 0) {
    HeaderView(title: "Western", subtitle: "Conference")
    Divider().padding(.bottom, 8)

    // Add next step here
}
```

We now have our container with a header and divider. Next, we need to create our module container. Replace `// Add next step here` with the following code:

```
HStack {  
    VStack {  
        StandingsHeaderView()  
        VStack(spacing: 2) {  
            ForEach(0..<10) { item in  
                StandingItem()  
            }  
        }  
    }  
    .padding()  
    .background(Color.white)  
    .cornerRadius(10)
```

We've now added our `StandingsHeaderView` and we created 10 `StandingItem()`, which were provided for you.

Let's go back to `HomeDashboardView`, delete everything in `middleRow`, and replace it with the following:

```
HStack(alignment: .top, spacing: 20) {  
    FeaturedArticleModuleView()  
    FeaturedPlayerModuleView()  
    StandingsModuleView()  
}
```

When you are done, you will see the following:



Figure 10.27

Playoff module design challenge

Let's take a look at the playoff module design:

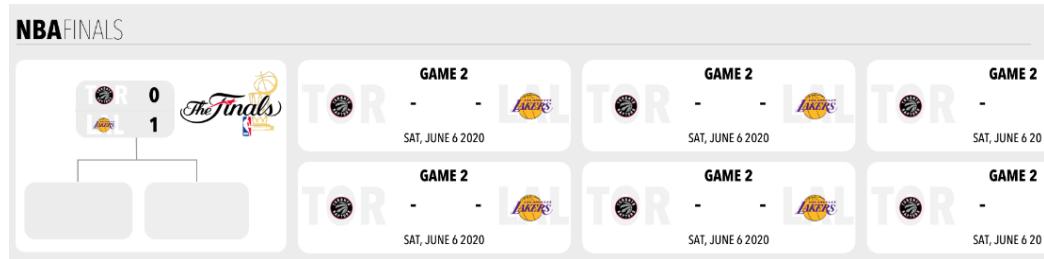


Figure 10.28

We just set up all of the modules on the home dashboard together. Now it is your turn to do the same for this design. I put all of the game playoff design inside of the PlayoffGridView and then I combined this with PlayoffModuleView together with the title. Each file is created for you all you have to do is code it. You have all of the sizing and specs you need in the design spec folder. Have fun and take your time, and as always, you can find the files in the challenge folder inside challenge 3.

Roster

We now need to get the roster up and running; it will start to come together fairly easily as some of the components were used in the dashboard as well. Let's get the roster done next but before we do that, let's look at the design spec and see exactly what we need to do:



Figure 10.29

We only need to build out the header for this section because we already created the cards for the dashboard.

The header has a 3D stacking effect with team stats under the starting five. The starting five will always be centered, as will the stats, and when the screen gets smaller, the stats will scroll.

Looking at this design, we can break up the header into two pieces: the header (which contains the starting five) and the stats section, which scrolls. Let's start with the header part first.

RosterHeaderView

Let's get started by opening `RosterHeaderView` first.

In the design, we see that the players' heads peek over the rounded background. In order to mimic this effect in SwiftUI, we will start by creating a `ZStack` container. Find `Text ("Roster Header View")` and replace it with the following:

```
ZStack {  
    // Add next step here  
}  
.padding(24)  
.padding(.top, 50)
```

We are using a `ZStack` component as our main container. Next, replace `// Add next step here` with the following:

```
ZStack { // (1)  
    HStack { // (2)  
        Spacer()  
        ZStack(alignment: .topTrailing) { // (3)  
            Text("STARTING")  
                .custom(font: .bold, size: 65)  
                .foregroundColor(Color(.baseWhite))  
                .offset(x: -80, y: 20)  
            Text("5")  
                .custom(font: .bold, size: 200)  
                .foregroundColor(Color(.baseWhite))  
                .offset(x: 15, y: -30)  
        }  
    }  
}  
.frame(height: 210)  
.background(Color.white)  
.cornerRadius(10)
```

```
// Add next step here
```

Let's look at what we just added:

1. We created a `ZStack` container for the **STARTING 5** text.
2. We wrap the text inside of an `HStack` component, which allows us to use a spacer and make our text appear on the right side of the container.
3. We use another `ZStack` container, which actually holds our `Text` views.

Now that we have our text in place, let's add our player images. Replace `// Add next step here` with the following:

```
ZStack { // (1)
    Image("lebron-james-full")
        .zIndex(4) // (2)
    Image("anthony-davis-full")
        .zIndex(3)
        .offset(x: 130, y: -3)
    Image("javale-mcgee-full")
        .zIndex(1)
        .offset(x: -130, y: -4)
    Image("danny-green-full")
        .zIndex(0)
        .offset(x: -250, y: -4)
    Image("avery-bradley-full")
        .zIndex(0)
        .offset(x: 250, y: 1)
}.offset(y: -62)

// Add RosterStatsView here
```

We have some new code. Let's discuss it:

1. We are using a `ZStack` container for our starting players' images.
2. The key to our header is `zIndex`. The lower the number, the farther back the view will be. In our case, Avery Bradley and Danny Green are both pushed all the way back.

We are just about done with our header. We now need to add our stats view. Open `TeamStatItem` and replace `Team Stat Item` with the following:

```
HStack(spacing: 0) {  
    // Add next step here  
}
```

We are using an `HStack` component for our main container. Now, let's replace `// Add next step here` with the following:

```
 VStack {  
    Text("Rank")  
        .custom(font: .bold, size: 16)  
        .frame(width: 35)  
        .rotationEffect(.degrees(-90))  
    }  
    .frame(width: 20, height: 75)  
    .background(Color.baseYellow)  
  
    // Add next step here
```

We are using a `VStack` component and then rotating our text -90 degrees.

Let's add the stat ranking next by replacing `// Add next step here` with the following:

```
HStack(alignment: .top, spacing: 0) {  
    Text("18")  
        .custom(font: .bold, size: 46)  
    Text("TH")  
        .custom(font: .bold, size: 16).offset(y:10)  
}  
    .frame(width: 70, height: 75)  
    .background(Color.baseGrey)  
  
    // Add next step here
```

Now, we are using an `HStack` component with zero spacing and it is our main container. Inside of the `HStack` container, we have two `Text` views.

Let's add our final container by replacing `// Add next step here` with the following:

```
 VStack(spacing: -8) {
    Text("109.0")
        .custom(font: .bold, size: 27)
    Text("PTS PER GAME")
        .custom(font: .light, size: 16)
}
.frame(width: 95, height: 75)
.background(Color.white)
```

We are using a `VStack` component as a container to wrap the stats `Text` view.

Now that we are done, open `RosterStatsView` next and replace `Roster Stats View` with the following:

```
VStack {
    ScrollView(.horizontal, showsIndicators: false) {
        HStack {
            ForEach(0 ..< 6) { item in
                TeamStatItem()
            }
        }
    }
}.offset(y: 68)
```

We just created a horizontal scroller for `TeamStatItem`. We are just creating six items for our scroller. Now, go back to `RosterHeaderView()` and replace `// Add RosterStatsView here` with `RosterStatsView()`.

When you are finished, you will see the following in Previews:



Figure 10.30

We are not done with our roster header. Let's get the rest of RosterView updated.

RosterView

Inside RosterView, we need to get rid of some of the prototyping that we did earlier. Before we do that, let's add a couple of variables first:

Add the following after the body variable:

```
var header: some View {  
    RosterHeaderView()  
}
```

We are adding RosterHeaderView inside of this variable. No, it's not really needed, but I find it gives a cleaner look. You can choose whether to create it or not.

Now, find the following code:

```
Rectangle() // (1)  
.cornerRadius(10)  
.frame(height: 210)  
.padding(.horizontal, 10)
```

Replace it with header. When you are done, hit resume in the preview and you will see the following:

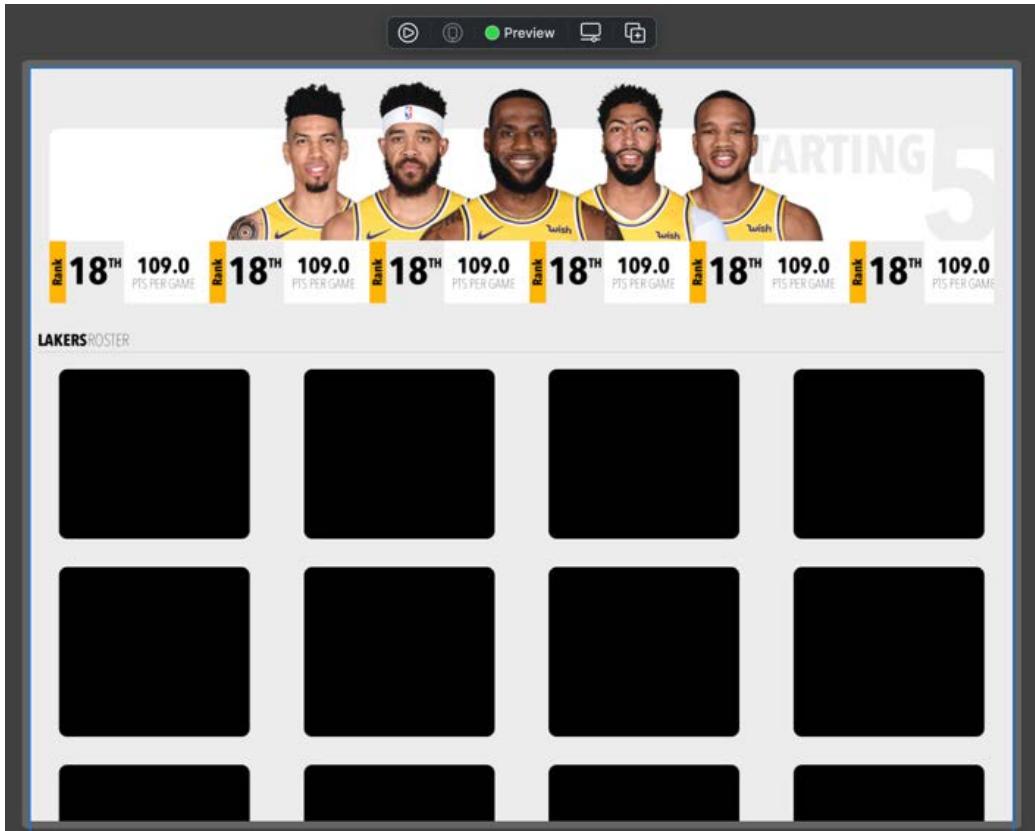


Figure 10.31

Next, after the `header` variable, add the following. We already have the code for the title created, so you can just move it into the `title` variable:

```
var title: some View {
    VStack(spacing: 0) {
        HeaderView(title: "Lakers", subtitle: "Roster",
        showViewAll: false)
            .padding(.horizontal, 10)
        Divider()
            .padding(.horizontal, 10)
    }
}
```

Make sure you delete the `VStack` component we created for our title and replace it with `title`. We are just missing our grid with cards now. We are going to use the same cards that we used in the dashboard, so after the `title` variable, add the following. Again, you can cut and paste this code from `body` and place it into the `grid` variable. All you have to do is replace `Rectangle()` with `PlayerCardView()` and delete both modifiers:

```
var grid: some View {
    LazyVGrid(columns: [GridItem(.adaptive(minimum: 250),
        spacing: 4)], spacing: 34) {
        ForEach(0..<15) { _ in
            PlayerCardView()
        }
    }
    .padding(.horizontal, 5)
    .padding(.vertical, 30)
}
```

We are officially done with the roster view. If you build and run, you'll see the same behavior you saw when we were prototyping, but now we have our header with every card showing LeBron James:

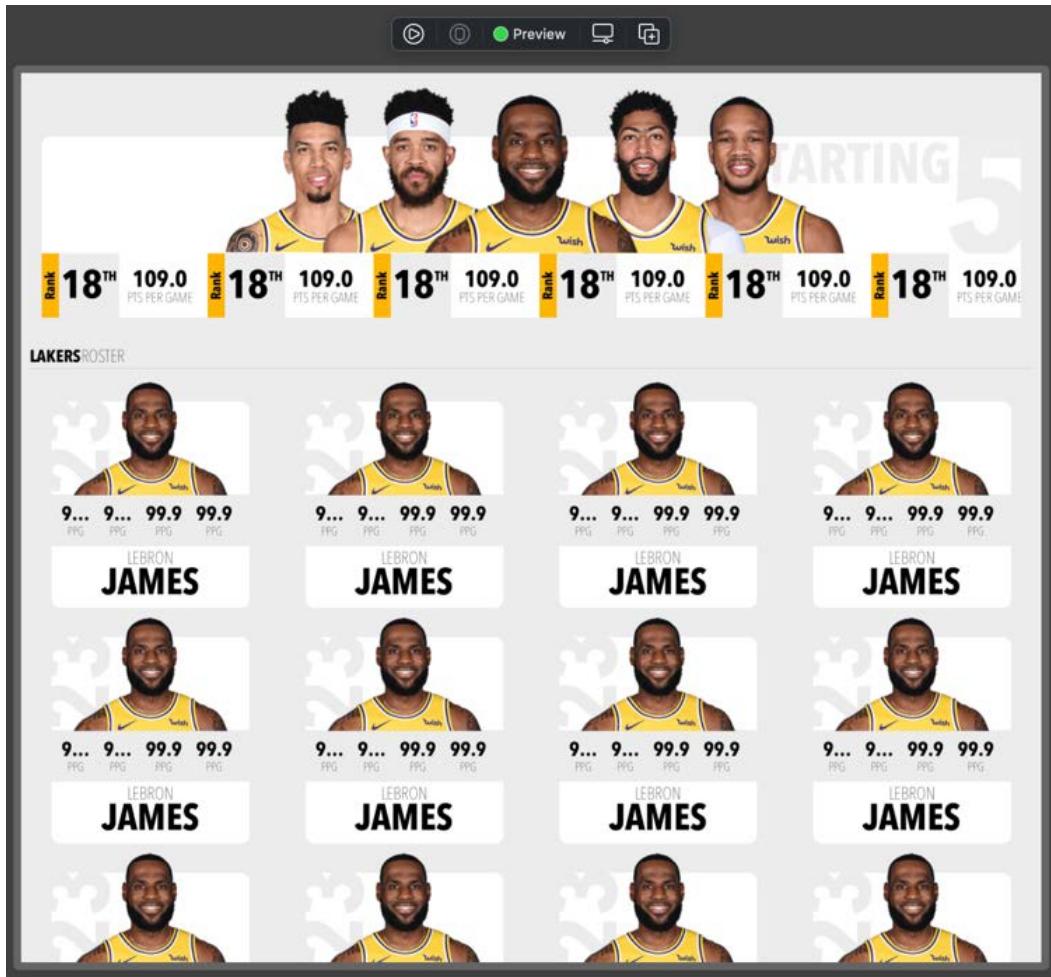


Figure 10.32

We are going to work on one more section together – schedule – and then you will get on to your final challenge of the book.

Schedule

We do not have much to do for the schedule section. Let's look at the design specs for schedule and see what we need to do. We have to just create a view for each game.

Open GameItemView and update previews with the following:

```
.previewLayout(.fixed(width: 290, height: 80))
```

Now that we have previews updated, let's move on to getting our GameItemView set up. Replace Text ("Game Item View") with the following:

```
ZStack {  
    // Add next step here  
}  
.frame(height: 74)  
.background(Color.white)  
.cornerRadius(10)
```

We are using a ZStack component as our main container.

Next, let's add the team names by replacing // Add next step here with the following:

```
HStack {  
    Text("TOR")  
        .custom(font: .bold, size: 47)  
        .foregroundColor(Color.baseGrey)  
  
    Spacer()  
  
    Text("LAL")  
        .custom(font: .bold, size: 47)  
        .foregroundColor(Color.baseGrey)  
}  
  
// Add next step here
```

We now have our team names added and we are displaying them horizontally using a `Spacer()` so that they are pushed to the edges. We need to do the same thing with the team logos.

Replace `// Add next step here` with the following:

```
HStack {  
    Image("raptors-logo")  
    Spacer()  
    Image("lakers-logo")  
}.padding(.horizontal, 12)  
  
// Add next step here
```

Our logos are now added; we just need to add the scores and game information.

Replace `// Add next step here` with the following:

```
HStack(spacing: 24) {  
    Text("999")  
    .custom(font: .medium, size: 29)  
    VStack {  
        Text("GAME 1")  
        .custom(font: .bold, size: 14)  
        Text("FINAL")  
        .custom(font: .medium, size: 12)  
    }  
    Text("999")  
    .custom(font: .medium, size: 29)  
}
```

Our GameItemView is now complete. In Previews, you should see the following:

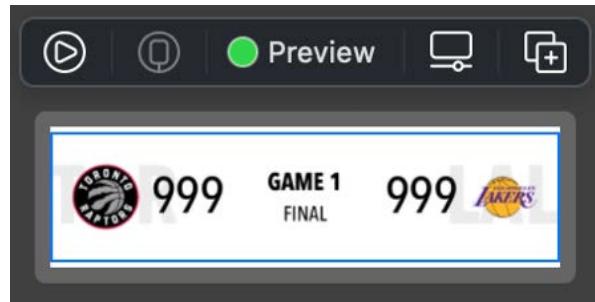


Figure 10.33

Next, open ScheduleView and replace Rectangle() in the ForEach with GameItemView. Then, copy the entire section and paste it underneath Section so that you now have two of them. Then, change HeaderView() to say “POST” instead of “REGULAR”. When you are finished, you should have the following:

```
Section(header: HeaderView(title: "REGULAR", subtitle:  
    "SEASON", showViewAll: false)) {  
    ForEach(0..<10) { _ in  
        GameItemView()  
            .cornerRadius(10)  
            .frame(height: 74)  
            .id(UUID())  
    }  
}  
  
Section(header: HeaderView(title: "POST", subtitle: "SEASON",  
    showViewAll: false)) {  
    ForEach(0..<10) { _ in  
        GameItemView()  
            .cornerRadius(10)  
            .frame(height: 74)  
            .id(UUID())  
    }  
}
```

We now have two sections for our schedule: **REGULAR SEASON** and **POST SEASON**. In order to keep the schedule from crashing, we are assigning an `.id(UUID())` to each `GameItemView()` so that each is unique. If you do not add the `.id` modifier the app will not run. In Previews, you will see the following:

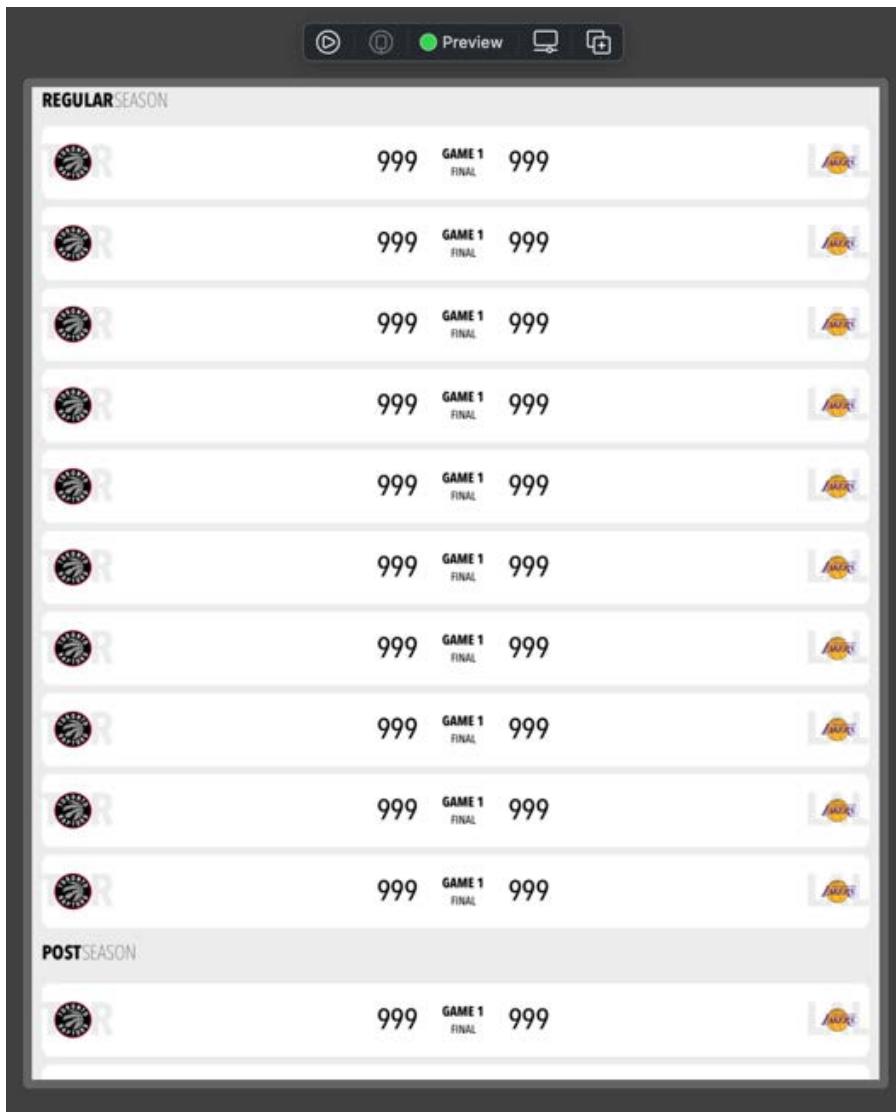


Figure 10.34

You are done following along with me on this design. Now it is time for your final design challenge.

Game detail challenge

Well, we are now at the point where we have gone back and forth between you following along with me and you designing on your own. You have a big challenge now – to work on an entire view on your own. You'll get to use the design specs in the project file. You have everything you need to get it done. If you get stuck, you can find the files you need in the challenge folder in challenge 4.

Summary

In this chapter, we covered some different things that we did not get to cover when we looked at designing for the iPhone. Our iPad can now flexibly work in landscape and portrait mode. Using a sidebar is a great way to utilize space and also make it flexible to move to a Mac app if you ever decide to. Now that we have our design done, all we need to do is add data.

In the next chapter, we are going to learn how to use an API to drive our app's data. We will also see how we can use Combine for our data as well.

11

Sports News

App – Data

In this book, we have yet to really work with feed data. In *Chapter 5, Car Order Form – Data*, we did some form posting that returned a small JSON response. In this chapter, we are going to work with feeds on a much larger scale. Working with feeds is unreliable, especially when writing a book, so we will be using a macOS app called Mockoon. Mockoon allows us to run mock APIs locally. Once we have everything set up in Mockoon, we will be able to work as if we were getting data from actual feeds. We'll get started with Mockoon first before we add any code.

In this chapter, we will cover the following topics:

- Learning how to use API structures using Combine
- Learning more about JSON and Decodable
- Learning how to load and play videos in a video player

Before we can get started with working on feeds, we need to understand how to get and use Mockoon. Let's do that now.

Mockoon

Mockoon is a Mac app and is very similar to Postman, though to me, Mockoon is much easier to get started with. If you are more familiar with Postman, by all means use that, though in this chapter, we will be using Mockoon. You can download Mockoon from the official website at <https://mockoon.com>. Once you have installed Mockoon, you will see that some demo paths have already been created for you:

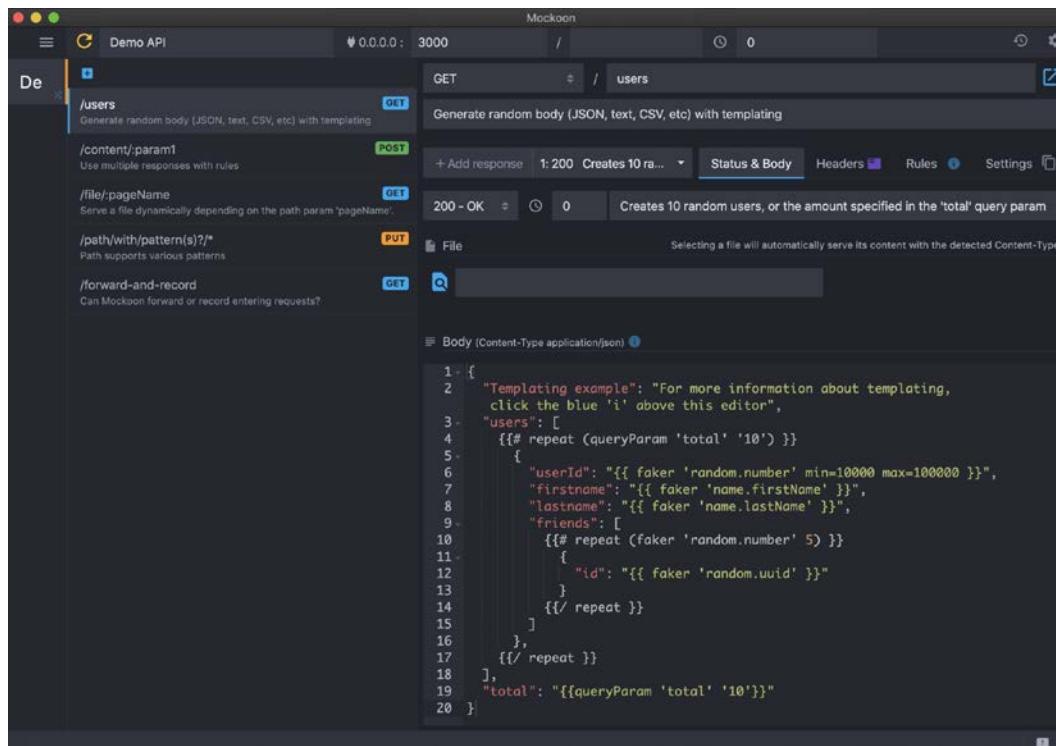


Figure 11.1

If you open the project files for this chapter, you will see a folder called `feeds`. Inside the `feeds` folder, you will see the `sportsnews.json` file. Move this file to your desktop so that it's easier to find. Then, inside Mockoon, go to the **Import/export** menu and select **Mockoon's format**:

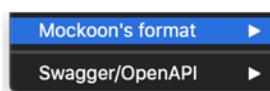


Figure 11.2

Then, select **Import from a file (JSON)**:

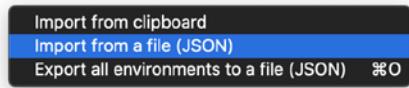


Figure 11.3

Select the `sportsnews.json` file wherever you saved it. Once you have done this, you will see that all the feeds have been loaded into Mockoon. Let's quickly go over the setup before we go back to Xcode:

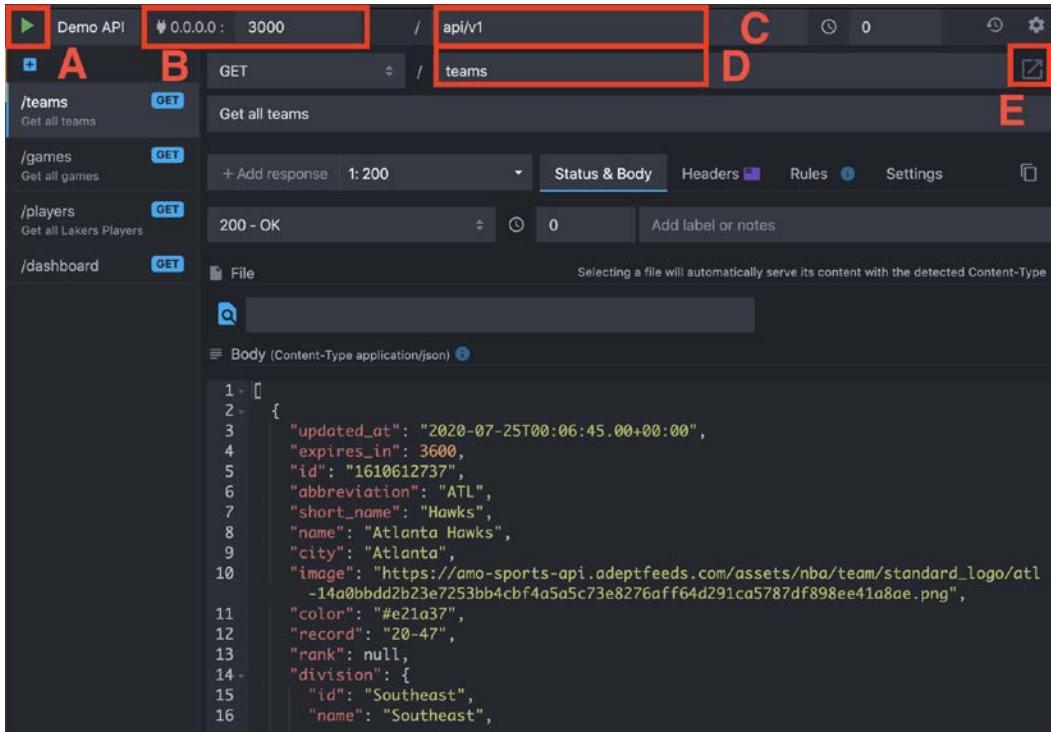


Figure 11.4

A. The **Play/Stop/Refresh** button (starts, stops, and refreshes the server)

B. **Port**

C. **Routes** (we are using `api/v1`)

D. **Path**

E. **View in browser**

Before you go back to Xcode, make sure you hit the play button, otherwise you will drive yourself crazy wondering why you are not getting data.

With that, your feeds have been set up and are ready to go. If you want to know how to do more with Mockoon, please visit their website.

API design

In this section, we are going to set up our API class. Once we have the respective file, we will use it minimally. I always try to use a simple API design and I feel this is really to the point. Even though we'll be using Combine here a lot more than we have the rest of this book, this won't be excessive and it will be fairly easy for you to understand what's going on. Here, 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 API class

Here, we need to create an API for every page in this app, which will include the dashboard, roster, schedule, and game details dashboard. First, create a new folder called `Networking` and then create a new file named `API`. Inside the `Networking` folder, update the import statements to the following:

```
import Combine
import SwiftUI
```

Next, add the following struct:

```
struct API {
    // Add next step here
}
```

We want to make sure we create a custom `Error` so that we're covered for when we get bad data. Replace `// Add next step here` with the following:

```
enum Error: LocalizedError, Identifiable { // (1)
    var id: String { localizedDescription }

    case addressUnreachable(URL) // (2)
    case invalidResponse

    var errorDescription: String? { // (3)
        switch self {
            case .invalidResponse: return "The server responded
```

```
        with garbage."
    case .addressUnreachable(let url): return "\\(url.
        debugDescription)"
    }
}
}

// Add next step here
```

Let's go over what we just added:

1. First, we created an `Error` enum that conforms to `LocalizedError` and `Identifiable`. `LocalizedError` gives us localized messages telling us about the error and why it happened.
2. Next, we added two case statements, `addressUnreachable` (URL) and `invalidResponse`. These two errors allow us to make sure we have a valid URL or that we get an invalid response from the feed otherwise.
3. After that, we created a description for both enum case statements we added. You can customize these descriptions however you like. You can also use `url.debugDescription` to get the server error that comes back.

From this point on, you will not be doing anything with `Error` enum. Now, let's move on to the next enum we need, `EndPoint`. Replace `// Add next step here` with the following:

```
enum EndPoint {
    static let baseURL = URL(string: "http://127.0.0.1:3000/")!
    // (1)

    case games // (2)
    case players
    case dashboard

    // Add next step here
}

// Add private variables here
```

We just added another enum called `EndPoint`. Let's discuss what we added:

1. We added our base URL here. Since we are using Mockoon, we used our localhost, which is typically `localhost:3001/`. However, Xcode will not allow us to use this path, so we used `http://127.0.0.1:3000/` instead. You might need to change this a bit on your machine, but this is pretty standard on most machines. If you need help with this, please check out Mockoon's website.
2. We then created an enum for each feed we need.

Now that our `EndPoint` section is done, we can add our URLs for our enums. Replace

`// Add next step here` with the following variable:

```
var url: URL {
    switch self {
        case .games:
            return EndPoint.baseURL.appendingPathComponent ("/api/
                v1/games")
        case .players:
            return EndPoint.baseURL.appendingPathComponent ("/api/
                v1/players")
        case .dashboard:
            return EndPoint.baseURL.appendingPathComponent ("/api/
                v1/dashboard")
    }
}

// Add next step here
```

The `url` variable we added here is used to map the path to each feed and append that path to the base url. We just need to add one more function to the `EndPoint` enum. Replace `// Add next step here` with the following code:

```
static func request(with url: URL) -> URLRequest {
    var request = URLRequest(url: url)
    request.httpMethod = "GET"
    request.setValue("application/json", forHTTPHeaderField:
        "Content-Type")

    return request
```

```
}
```

Lastly, we created a request, where we'll pass a `url` through the parameter. We are using a "GET" method to get our data here. Whatever comes back from the feed will need to be parsed through that. With that, we are done with the basic setup. However, we need to create a few more things before we can add anything else to this file. Let's move on and create our model data.

Dashboard data

Our dashboard has a lot of data inside of it and this is all stuff you should be familiar with if you have worked with iOS before. I am not going to go step by step and cover every file for the dashboard, but I will go over one section and let you set up the rest. If you have any issues with this process, you can look inside the project files for my data models. We are going to work on getting the Latest Video section set up so that we can gather data, as well as get it to play videos.

Latest video

Next, we are going to set up the latest video section so that you can set up the remaining modules. Let's take a look at the JSON structure for our `dashboard.json` in the `feeds` folder:

```
"videogallery_mod" : {  
    "id" : "20000900",  
    "title" : "LATEST",  
    "sub_title" : "VIDEOS",  
    "videos" : [...]  
}
```

Our `videogallery_mod` has four main components: `id`, `title`, `sub_title`, and an array of `videos`. Each video looks as follows:

```
{  
    "id": "20000903",  
    "title": "Staying Healthy with Team Physicians Dr. Jones",  
    "image": "video3-thumb",  
    "url": "video2",  
    "date": "2020-07-10"  
}
```

Each video object has five properties: `id`, `title`, `image`, `url`, and `date`. Now that we know what we need, let's set up our `Video` object. Open `Video.swift` inside the `Model` folder and add the following code:

```
struct Video: Decodable, Identifiable {
    var id: String
    var title: String
    var image: String
    var url: String
    var date: String

    static let `default` = Self(id: "1234", title: "Video
title", image: "video1-thumb", url: "", date: "10/10/2020")
}
```

We now have a `Video` struct that conforms to `Decodable` and `Identifiable`. `Decodable` is used to decode JSON data. In our case, all the `json` keys match precisely what we have in the feed, so we do not have to do anything special. `Identifiable` is used when we create a list, and it must contain a unique `id` that we get from the feed. We also added a default variable that we can use for our View model later.

Finally, just like we did with `Video`, we want a struct that conforms to `Decodable` and `Identifiable`. Let's make `VideoGalleryMod` match the properties we have provided in JSON. Create the `VideoGalleryMod` file, save it inside the `Model` folder, and then add the following code:

```
struct VideoGalleryMod: Decodable, Identifiable {
    let id: String
    let title: String
    let subtitle: String
    let videos: [Video]

    enum CodingKeys: String, CodingKey {
        case id
        case title
        case subtitle = "sub_title"
        case videos
    }
    static let `default` = Self(id: UUID().uuidString, title:
```

```
    "LATEST", subtitle: "VIDEOS", videos: [Video.default])  
}
```

VideoGalleryMod is similar to our Video object, except we have CodingKeys in this file. We created a subtitle instead of `sub_title`. When we create them, the properties don't match exactly, so we have to create a CodingKeys so that Decodable knows how to map the data. Now that you know how we configured the video, it is time for your first challenge.

Model challenge time

If you have worked with Decodable before, then this won't be a difficult challenge. Here, the following model objects need to be created:

- Game
- Player
- Stats
- Article
- Team (used for Standings)
- GameTeam (used for Schedule)

We also need to create the following module data models:

- PlayoffMod (ignore the playoff grid – only worry about games)
- FeaturedPlayerMod
- FeaturedArticleMod
- StandingsMod

When you are finished, we can move on to creating our dashboard data. If you need help, you can find the model objects inside the `challenge` folder inside `challenge 1`.

Updating more dashboard data

Now that all our data has been created for the dashboard, we are going to create a `Dashboard` object that will hold each module. Create a `Dashboard` file, save it in the `Models` folder, and then add the following to it:

```
struct Dashboard: Decodable {
    let featuredArticleMod: FeaturedArticleMod
    let videoGalleryMod: VideoGalleryMod
    let playoffMod: PlayoffMod
    let featuredPlayerMod: FeaturedPlayerMod
    let standingsMod: StandingsMod

    static let `default` = Self(featuredArticleMod:
        FeaturedArticleMod.default, videoGalleryMod:
        VideoGalleryMod.default, playoffMod: PlayoffMod.default,
        featuredPlayerMod: FeaturedPlayerMod.default,
        standingsMod: StandingsMod.default)

    enum CodingKeys: String, CodingKey {
        case featuredArticleMod = "article_mod"
        case videoGalleryMod = "videogallery_mod"
        case playoffMod = "playoff_mod"
        case featuredPlayerMod = "featured_player_mod"
        case standingsMod = "standings_mod"
    }
}
```

With that, we have created a variable for each module and also mapped our data to the JSON keys. Now that we have created our `Dashboard` file, we can create the method that will grab the JSON from the feed. Open `API`, find `// Add private variables here`, and replace it with the following:

```
private let decoder = JSONDecoder()

// Add next step here.
```

We just created a private `JSONDecoder` to parse JSON. Now, we need to create a function that will fetch our dashboard JSON data. Replace `// Add next step here` with the following:

```
func fetchDashboard() -> AnyPublisher<Dashboard, Error> {
    // (1)

    URLSession.shared.dataTaskPublisher(for: EndPoint.
        request(with: EndPoint.dashboard.url)) // (2)
        .map { return $0.data } // (3)
        .decode(type: Dashboard.self, decoder: decoder) // (4)
        .print() // (5)
        .mapError { error in // (6)
            switch error {
            case is URLError:
                return Error.addressUnreachable(EndPoint.
                    dashboard.url)
            default: return Error.invalidResponse
            }
        }
        .print() // (7)
        .map { $0 } // (8)
        .eraseToAnyPublisher() // (9)
}
```

Let's walk through what we just added:

1. Once everything has been run, we expect to get back a publisher that will contain a dashboard and an error.
2. We are using `dataTaskPublisher` instead of just `dataTask` here because `Publisher` gives us access to Combine features. We are also passing in the dashboard URL for this request.
3. `.map` maps our JSON data.
4. `.decode` decodes our data to the `Dashboard` object.
5. `.print()`: Using `print` is a great way to debug errors. I have added it here for demonstration purposes. You would remove this statement when you are done debugging.

6. `.mapError` will be called if there are any errors.
7. `.print()`: Another spot where you can add print and see what your streams are doing. You would remove this statement when you are done debugging.
8. `.map` is used to map our data one last time.
9. `.eraseToAnyPublisher` is required when returning publishers.

We need two more methods before we can move on to the next part.

API fetch data challenge

We need one method that will fetch the roster and one that will fetch the schedule. Earlier, you created the model objects for `Player` and `Game`. Now, you need to create two methods called `fetchPlayers` and `fetchGames`. These two methods will look similar to the method we just created together, except for their paths and model objects. If you get stuck, you can find both methods inside the `challenge` folder, inside the `challenge 2` folder.

Now, let's move on to creating our View model.

Creating our View model

Next, we need a View model for this app. First, create a new `View Model` folder inside the `SportsNews` folder. Then create a new file called `SportsNewsViewModel`, and save it inside the `View Model` folder. Next, update the `import` statements inside it to the following:

```
import SwiftUI
import Combine
```

Then, add the following after the `import` statements:

```
class SportsNewsViewModel: ObservableObject {
    // Add next step here
}
```

Next, we need to add a couple of variables to get started. Replace `// Add next step here` with the following:

```
private let api = API() // (1)
private var subscriptions = Set<AnyCancellable>() // (2)
```

```

@Published var error: API.Error? = nil    // (3)
@Published var regSeasonGames: [Game] = [] // (4)
@Published var preSeasonGames: [Game] = []  // (5)
@Published var players: [Player] = []     // (6)
@Published var selectedVideo: Video = Video.default // (7)
@Published var dashboard: Dashboard =
    Dashboard(FeaturedArticleMod: FeaturedArticleMod.default,
              VideoGalleryMod: VideoGalleryMod.default, PlayoffMod:
              PlayoffMod.default, FeaturedPlayerMod: FeaturedPlayerMod.
              default, StandingsMod: StandingsMod.default) // (8)

// Add fetch method here

```

Let's go over each variable as it is important to know what each one is for. They have been labeled correspondingly in the code:

1. First, we created an instance of our API() class.
2. Then, we created an instance of Set<AnyCancellable>(). When using subscribers and publishers, as we are doing for our feeds, AnyCancellable provides a **cancellation token** and automatically calls cancel() when it is de-initialized.
3. Our Error variable is used when our feeds return an error.
4. This is an array to hold regular season games.
5. This is an array to hold preseason games.
6. This is an array to hold players.
7. In the video module, we need to keep track of which video is playing. selectedVideo allows us to set the variable. When it changes, the video player will reload the newly selected video.
8. Finally, we created a dashboard variable, which we can use inside each model to display our data.

Earlier, we created a `fetchDashboard()` method that gets our JSON data and decodes it to our Dashboard object. Now, we are going to use it. Replace //Add `fetch` method here with the following:

```

func fetchDashboard() {
    api

```

```
    .fetchDashboard()
    .receive(on: DispatchQueue.main)
    .sink(receiveCompletion: { completion in
        if case .failure(let error) = completion {
            self.error = error
        }
    }, receiveValue: { dashboard in
        self.dashboard = dashboard
        self.error = nil
    }).store(in: &subscriptions)
}

// Add fetchPlayers here
```

Here, we are creating another `fetchDashboard()` method located inside our View model. We are mapping the dashboard object to the `@Published` variable, which will allow us to get automatic updates once the data returns.

We are using Combine's `sink` method to determine whether we got an error or whether everything comes back fine. If we get an error, we set the error to `self.error`, and if everything comes back fine, we set it to our `dashboard` variable and set `error` to `nil`. Since we are using `@Published`, whatever view is using the `dashboard` variable will have its data automatically update when the feed is done. We do not have to add a refresh or any other method to get our data. Combine might be a difficult topic to grasp, but once you have grasped it, it is very powerful.

Now, we need to set up two more functions. They are going to have the same pattern, except we will be fetching different data. Let's create a method for fetching a player and replace `// Add fetchPlayers here` with the following:

```
func fetchPlayers() {
    api
    .fetchPlayers()
    .receive(on: DispatchQueue.main)
    .sink(receiveCompletion: { completion in
        if case .failure(let error) = completion {
            self.error = error
        }
    }, receiveValue: { players in
```

```
    self.players = []
    self.players = players

    self.error = nil
}).store(in: &subscriptions)
}

// Add fetchGames here
```

The code we just added does exactly what the fetch dashboards did. However, we still need one more that will fetch games. Replace `// Add fetchGames here` with the following:

```
func fetchGames() {
    api
    .fetchGames()
    .receive(on: DispatchQueue.main)
    .sink(receiveCompletion: { completion in
        if case .failure(let error) = completion {
            self.error = error
        }
    }, receiveValue: { games in
        self.regSeasonGames = []
        self.preSeasonGames = []
        self.preSeasonGames = games.filter({ $0.seasonType
            == "PRE" })
        self.regSeasonGames = games.filter({ $0.seasonType
            == "REG" })

        self.error = nil
}).store(in: &subscriptions)
}
```

Now, we want to make sure our `ViewModel` is set up as an environment variable.

Implementing our View model

We have our View model created and now we need to set it up to use in our app:

1. Open `SportsNewsApp.swift`, and add the following variable:

```
@StateObject var viewModel = SportsNewsViewModel()
```

2. Next, update `ContentView()` to the following:

```
ContentView().environmentObject(viewModel)
```

3. Now that `SportsNewsApp` has been set up, open `ContentView()` and update it to the following:

```
@EnvironmentObject var model: SportsNewsViewModel
var body: some View {
    AppSidebarNavigation().environmentObject(model)
}
```

4. Update `previews` as well with the following:

```
static var previews: some View {
    ContentView().environmentObject(SportsNewsViewModel())
}
```

If you were to try and run `previews` without adding this environment object, you would get an error. When working with environment objects, remember that you will need to add them to your `previews` in order for them to work.

Lastly, we need to make sure all our views also have access to the environment object. Let's take a look at how to do this:

1. Open `AppSidebarNavigation()` and add the following variable after the `selection` variable:

```
@EnvironmentObject var model: SportsNewsViewModel
```

2. Inside the `body` variable, change `HomeDashboardView()` to the following:

```
HomeDashboardView().environmentObject(model)
```

- Now, do the same with `HomeDashboardView()` inside the `sidebar` variable:

```
HomeDashboardView().environmentObject(model).
```

- Next, inside the navigation link, update `RosterView()` and `ScheduleView()` so that each has `.environmentObject(model)`:
- Finally, update previews for `AppSidebarNavigation()` with the following:

```
struct AppSidebarNavigation_Previews: PreviewProvider {
    static var previews: some View {
        AppSidebarNavigation()
            .environmentObject(SportsNewsViewModel())
    }
}
```

We now have previews working again. We are done with `AppSidebarNavigation()`, so let's move on to `HomeDashboardView()`.

Adding our View model to HomeDashboardView

Next, open `HomeDashboardView()`:

- Add the following variable above the body declaration:

```
@EnvironmentObject var model: SportsNewsViewModel
```

- Look for the `background` modifier and add the following `fetch` after it:

```
.onAppear {
    model.fetchDashboard()
}
```

- Next, update the preview for `HomeDashboardView()` to the following:

```
static var previews: some View {
    HomeDashboardView()
        .previewLayout(.fixed(width: 1187, height: 1034))
        .environmentObject(SportsNewsViewModel())
}
```

4. Next, we need to update `RosterView()` and `ScheduleView()` by adding the following:

```
@EnvironmentObject var model: SportsNewsViewModel
```

5. You can update the preview with the following:

```
.environmentObject (SportsNewsViewModel())
```

Now that we are done with `RosterView` and `ScheduleView`, we can move on to `VideoModuleView`.

Updating `VideoModuleView`

Let's open `VideoModuleView` and start by adding our View model:

1. Above the `body` variable, add the following:

```
@EnvironmentObject var model: SportsNewsViewModel
```

2. Then, find `CustomVideoPlayer(urlPath: $path)` and replace it with the following:

```
CustomVideoPlayer(urlPath: $model.selectedVideo.url)
```

When the selected video gets updated, it will trigger our video player, making it load the new video URL.

You can now delete the private `var path` variable.

3. Next, find `HeaderView()` inside `VideoModuleView` and update the Header title and subtitle:

```
HeaderView(title: model.dashboard.videoGalleryMod.title,  
          subtitle: model.dashboard.videoGalleryMod.subtitle,  
          showViewAll: false)
```

Now, our title is coming from the feed instead of it being hardcoded. Let's update our `ForEach` next so that it's using data from the feed as well.

4. Replace `ForEach` with the following:

```
ForEach(model.dashboard.videoGalleryMod.videos) { item in
```

Now, we are loading videos from the feed. However, `VideoItem` needs to be updated. Let's update `VideoItem` next.

Updating VideoItem

We are moving on to `VideoItem` in order to make a couple of quick updates:

1. Open `VideoItem` and add the following variable above the body:

```
var item: Video
```

2. Next, fix `VideoItem_Previews` with the following:

```
struct VideoItem_Previews: PreviewProvider {
    static var previews: some View {
        VideoItem(item: Video.default)
    }
}
```

3. Then, fix `Image("news-sample")` to the following:

```
Image(item.image)
```

That's all we need to do inside `VideoItem`. Finally, let's finish `VideoModuleView()`.

Finishing VideoModuleView

We have to make a change in `VideoModuleView()`. Update `ForEach` to the following:

```
ForEach(model.dashboard.videoGalleryMod.videos) { item in
    VideoItem(item: item)
        .frame(minWidth: 0, maxWidth: .infinity)
        .onTapGesture {
            model.selectedVideo = item
        }
}
```

You can now build and run and when we tap a video thumbnail in the grid, it will load a new video to be played. Let's work on the `FeaturedArticle` module next.

Connecting feeds to FeaturedArticleModuleView

We are going to get our `FeaturedArticle` module working next:

1. Open `FeaturedArticleModuleView()` and add the following variable:

```
@EnvironmentObject var model: SportsNewsViewModel
```

2. Now, update `HeaderView` so that we are using the titles from the feeds:

```
HeaderView(title: model.dashboard.featuredArticleMod.  
title, subtitle: model.dashboard.featuredArticleMod.  
subtitle)
```

3. Next, we need to update the image, article title, and article summary by making the following changes:

```
Image(model.dashboard.featuredArticleMod.article.image)
```

4. Now, let's update the title:

```
Text(model.dashboard.featuredArticleMod.article.title.  
uppercased())
```

5. Finally, we will update the summary:

```
Text(model.dashboard.featuredArticleMod.article.summary)
```

Now that our featured article is complete, build and run the project and you will see that `FeaturedArticle` is now pulling data from the feed. Let's now move on to `FeaturedPlayer` and get it working with feed data.

Updating FeaturedPlayerModuleView

We are now moving on to the `FeaturedPlayer` module view:

1. Open `FeaturedPlayerModuleView` and add the following above the `body` variable:

```
@EnvironmentObject var model: SportsNewsViewModel
```

2. Next, update the Header view so that our title is coming from the feeds instead of being hardcoded:

```
HeaderView(title: model.dashboard.featuredPlayerMod.
    title, subtitle: model.dashboard.featuredPlayerMod.
    subtitle)
```

3. We also need to change PlayerCardView() to the following:

```
PlayerCardView(player: model.dashboard.featuredPlayerMod.
    featuredPlayer)
```

Next, we need to update PlayerCardView first before we can finish up in FeaturePlayerModuleView. Once we are finished, we will come back and make the necessary changes.

Updating PlayerCardView

Updating PlayerCardView causes RosterView to error because we are using PlayerCardView in both the dashboard and RosterView. By updating PlayerCardView, we are going to kill two birds with one stone:

1. Open PlayerCardView and add the following variable:

```
var player: Player
```

2. Next, let's update the jersey text to the following:

```
Text(player.jersey)
```

3. Now, we need to make an update to the player stats. So, here, update ForEach to the following:

```
ForEach(player.quickStats) { item in
    VStack(spacing: -5) {
        Text(item.value).custom(font: .bold, size: 23)
        Text(item.name).custom(font: .light, size: 14)
    }.padding(.trailing, 8)
}
```

Now, we are getting the stats from the feed. When we go to Roster, we will see each player with their stats as well.

4. Next, let's update the player's name by replacing the hardcoded Lebron James text with the following:

```
 VStack(spacing: -15) {  
     Text(player.firstName.uppercased()).custom(font:  
         .light, size: 20)  
     Text(player.lastName.uppercased()).custom(font:  
         .bold, size: 43)  
 } .frame(height: 72)
```

5. And let's do the same for the image:

```
 Image(player.image)
```

With that, we have updated everything we needed to inside `PlayerCardView`.

6. To get a preview of this, update `PlayerCardView()` with the following:

```
 PlayerCardView(player: Player.default).previewLayout(.  
     fixed(width: 400, height: 600))
```

Now, we are done making updates to `PlayerCardView`. Let's move on to `RosterView`.

Updating RosterView

Next, let's jump over to `RosterView` and fix the error:

1. Now, above the `body` variable, add the following:

```
 @EnvironmentObject var model: SportsNewsViewModel
```

2. Look inside the `grid` variable and update `ForEach` to the following:

```
 ForEach(model.players) { player in  
     PlayerCardView(player: player)  
 }
```

3. We need to fetch the roster feed when this view appears, so find the following modifier – `.navigationBarTitleDisplayMode(.inline)`. Add a new line after the modifier and add the following:

```
.onAppear {
    model.fetchPlayers()
}
```

Now, you will see the player cards for each player. The only remaining module we need to update is the `Standings` module. Let's work on this now. With that, we are done with `RosterView()`.

Standings

We will work on one more module together before you get to work on the next challenge. Before we jump into `StandingsModuleView`, we need to update `StandingItem`.

StandingItem

We have a few things we need to add to `StandingItem`:

1. First, we will add the following `Team` variable:

```
var item: Team
```

2. Next, we'll update the team name with the following code:

```
Text(item.shortName)
```

3. Now, add the following for the team's record:

```
Text(item.record())
```

4. We need to update the winning percentage with the following updates:

```
Text(item.winPct)
```

5. Let's now update the games behind:

```
Text(item.gamesBehind)
```

6. Finally, make sure you update the preview with the following:

```
static var previews: some View {  
    StandingItem(item: Team.default).previewLayout(.  
        fixed(width: 600, height: 25))  
}
```

With that, we've made all the changes we need to `StandingItem`. Let's wrap up our dashboard by making a few changes to `StandingsModuleView`.

StandingsModuleView

We are just about done with the dashboard. All we have left to do is perform a few updates in `Standings`:

1. Open `Standings` and add the following variable:

```
@EnvironmentObject var model: SportsNewsViewModel
```

2. Next, update `HeaderView` with the following:

```
HeaderView(title: model.dashboard.standingsMod.title,  
        subtitle: model.dashboard.standingsMod.subtitle)
```

3. Lastly, update `ForEach` with the following:

```
ForEach(model.dashboard.standingsMod.standings) { item in  
    StandingItem(item: item)  
}
```

4. The preview also needs to be updated with the following:

```
StandingsModuleView()  
    .environmentObject(SportsNewsViewModel())
```

With that, we've finished updating `StandingsModuleView`. We have done a lot together, but now it is time for you to work on something on your own. Let's look at what your next challenge will be. At this point, you should be able to build and run the project and see data in the home dashboard.

Final challenge

There is actually one more module that needs to be created: the `Playoff` module. Since you will be creating `Schedule`, you will actually be creating a `Playoff` module at the same time. You can ignore the grid on the left and just focus on the games. Your task is to get `Schedule` set up so that it shows both the preseason and regular season. When you are done, your `PlayoffGridView` module also needs to be updated. We have created sufficient modules together that you can do all of this on your own. If you get stuck, feel free to look at the completed files to see what I did. Remember to add the feed call we set up for `ScheduleView` in the `onAppear()` method, just like we did for `RosterView`.

We are not going to worry about loading data into the game details dashboard since this requires a feed that can load data based on the game. Game details is beyond the scope of this book as it would require a lot of work to get it working.

If you get stuck with `ScheduleView` or the `Playoff` module, take a look in the completed folder as the completed project will have all of the working code for this last challenge. You can also find the solutions in the `challenge` folder, inside `challenge 3`.

Summary

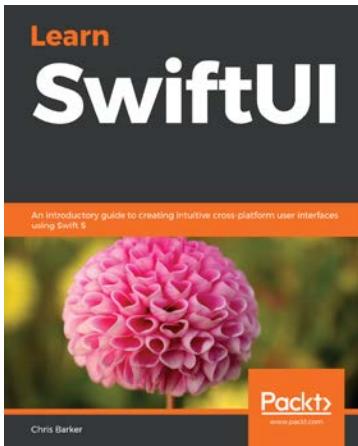
In this chapter, we worked on adding an API to our SwiftUI app. We set up our own API using Combine. By completing this chapter, we have worked with model objects that are decodable. We also set up a video player that can play videos.

At this stage, you should feel comfortable with SwiftUI on all devices. You should be comfortable with being able to take any design that you see and recreating it in SwiftUI. Even though we did not design an app that works on all devices, you have all the skills you need to make this happen.

SwiftUI is a great tool once you start using it on a regular basis. Writing this book has made me fall in love with it even more because I enjoy designing with SwiftUI and then tying in interactivity with `Binding` and `State`. Because of this book, I have created video courses on learning how to design with SwiftUI. This should be sufficient proof to demonstrate how much I enjoy working with it.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Learn SwiftUI

Chris Barker

ISBN: 978-1-83921-542-1

- Explore the fundamentals of SwiftUI and compare it with existing UI frameworks
- Write SwiftUI syntax and understand what should and shouldn't be included in SwiftUI's layer
- Add text and images to a SwiftUI view and decorate them using SwiftUI's modifiers
- Create basic forms, and use camera and photo library functions to add images to them
- Understand the core concepts of Maps in iOS apps and add a MapView in SwiftUI
- Design extensions within your existing apps to run them on watchOS
- Handle networking calls in SwiftUI to retrieve data from external sources



SwiftUI Cookbook

Giordano Scalzo, Edgar Nzokwe

ISBN: 978-1-83898-186-0

- Explore various layout presentations in SwiftUI such as HStack, VStack, LazyHStack, and LazyVGrid
- Create a cross-platform app for iOS, macOS, and watchOS
- Get up to speed with drawings in SwiftUI using built-in shapes, custom paths, and polygons
- Discover modern animation and transition techniques in SwiftUI
- Add user authentication using Firebase and Sign in with Apple
- Handle data requests in your app using Core Data
- Solve the most common SwiftUI problems, such as integrating a MapKit map, unit testing, snapshot testing, and previewing layouts

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

A

account, financial app
 account form view 181-183
 account list, creating 188, 189
 account type view 173
 color button menu 174-177
 color view 174
 CreateAccountView 183-186
 credit card type menu 177
 custom button style 177-179
 form text field 180, 181
 form view 179
AccountHomeView 222-226
account summary view, financial app
 credit card 163-168
 debit card 168
account view
 creating 220, 221

B

bar charts, SwiftUI Watch Tour
Capsule, with dynamic data 55-57
code cleanup 52
header, creating 48-52
HeaderView, with dynamic text 54, 55

reusable bar view, creating 54
reusable header view, creating 53
Binding
 about 140, 141
 versus State 141
Buttons 13-15

C

Capsule 19, 20
card list row 227
card view 227
card view, financial app
 card background 157, 158
 card balance 158, 159
 card information 160, 161
 card logo 158, 159
 card number 159, 160
 creating 157
car order form design
 Basic car info view, creating 107-109
 Bottom Order view 130-133
 Cancel Order design challenge 136, 137
 Car Detail 107
 Car Detail, updating 114, 115
 Car Info Detail view, creating 109-111

-
- Car Info Photos section, creating 111-113
 - Car Info view, displaying 113, 114
 - Complete Order design 125, 126
 - Complete Order view 133, 134
 - form sections, creating 116-124
 - form view 116
 - overview 104
 - structure 105, 106
 - Top Order view 127, 128
 - views, combining 135
 - Category/Extension setting, Core Data
 - Account extension 203, 204
 - Card extension 204
 - circle
 - creating 16, 17
 - CloudKit
 - basics 254
 - turning on, manually 255-259
 - CloudKit Dashboard
 - record type, indexing 266-273
 - CloudKit extensions 274
 - CloudKit helper 278-282
 - CloudKit models
 - creating 273
 - displaying 286
 - product model, creating 274- 278
 - CloudKit record
 - creating 260-265
 - CloudKit storage 254
 - Combine
 - about 141, 142
 - networking with 143-146
 - operators 143
 - publisher 142
 - subscribers 142
 - ContentView
 - refactoring 90, 91
 - updating, with environment object 219, 220
 - controls
 - about 3
 - Buttons 13-15
 - SecureField 8, 9
 - TextField 5-8
 - Core Data
 - about 192
 - challenge 218
 - relationships 201
 - Core Data Codegen
 - about 202
 - Category/Extension 203
 - Class Definition 203
 - Manual/None 203
 - Core Data entities
 - creating 194-196
 - Core Data manager
 - about 205, 206
 - batch delete 211
 - steps, creating 207, 208
 - steps, deleting 210, 211
 - steps, reading 208, 209
 - steps, updating 209, 210
 - using 207
 - CRUD (Creating, Reading, Updating, Deleting) 193

D

- dashboard data, Sports news app
- API fetch data challenge 369
- latest video section 364-366
- model challenge time 366
- updating 366-368

dashboard module views, Sports news app
 featured news module 328- 332
 featured player module 332-336
 playoff module design 342
 standings module view 336-340
 video item, creating 327, 328
 video player, with grid 326
 dashboard, Sports news app
 bottom row, creating 316-318
 middle row, creating 314-316
 top row, creating 311-313
 databases, CloudKit
 private 254
 public 254
 shared 254
 data model
 creating 193, 194
 detail view
 refactoring 95
 document storage 254
 DraftCardView
 data, adding to 101
 DraftList
 refactoring 91-95
 Draft List Card
 updating 100, 101
 DraftRound
 used, for updating menu 97, 98

E

ellipse
 creating 18, 19
 environment object
 ContentView, updating 219, 220
 EnvironmentObject
 setting up 207

F

financial app
 account, creating 172
 account summary, creating 162, 163
 account summary, finishing up 169, 170
 app design 154
 card view, creating 157
 home view, building 170, 171
 home view logic 155
 transaction item view 169
 ForEach struct
 about 33, 34
 versus List 36
 FormView
 updating 150-152

G

Group 32, 33

H

home view, financial app
 building 154, 170, 171
 header 155, 156
 home submenu view 162
 logic 155
 HStack
 about 24, 25
 with spacer 25, 26

I

image
 modifying 10-12
 Image view 9, 10

K

key-value storage 254

L

List

- versus ForEach 36
- working 35

M

managed object model 193

mock account preview service 213-215

Mockoon

- about 358
 - demo paths 358, 359
 - URL 358
- model properties
- adding 196-201

N

navigation, SwiftUI Watch Tour

- building 41
- static list, creating 41- 43

networking

- with Combine 143-146

O

object context 193

observable objects

- about 146
- conforming, to Codable 148, 149
- FormView, updating 150-152
- OrderViewModel 147
- TeslaOrderFormApp 150

Optionals 212

OrderViewModel 147

P

persistent store coordinator 193

R

record type

- indexing, in CloudKit
- Dashboard 266-273

rectangle

- creating 17, 18

Rounded Rectangle 20

S

ScrollView 36-38

SecureField 8, 9

SFSymbols

- reference link 12

shapes

- about 16
- Capsule 19, 20
- circle 16, 17
- ellipse 18, 19
- rectangle 17

Rounded Rectangle 20

Shoe Point of Sale System

Add Customer section, creating 248, 249

add to cart button, creating 242

app container, displaying 230

brands, creating 278

cart content view 248

CartItem 290-294

cart item view 249

cart item view, designing 246, 247

cart total display, creating 233
 cart view, displaying 234
 custom buttons, creating for
 product details view 240
 custom split view, viewing 234, 235
 custom stepper view 240, 241
 data, displaying in product
 details view 289, 290
 dummy data, creating 285, 286
 main products container, creating 232
 product details challenge 244
 product details view modal,
 displaying 288-290
 ProductsContentView 237, 238
 products, creating 236
 products header view, creating 231
 products, setting up 287, 288
 product views, updating 295-297
 shopping cart 290
 shopping cart, creating 245
 shopping cart header, creating 232
 SizeCartItemView 243, 244
 subtotal and taxes 249-252
 view model, creating 283-285

Sports news app
 API class, creating 360-364
 API design 360
 dashboard 307-310
 dashboard data 364
 dashboard module views, designing 326
 game details, prototyping 324, 325
 HeaderView 305-307
 navigation 301
 prototyping, with boxes 305
 RosterHeaderView 343-347
 roster prototyping 319-322
 roster section 342
 RosterView 348-351

schedule prototyping 322, 323
 schedule section 352-355
 sidebar, creating 301-304
 Standings 380
 View model, creating 369-372
 Standings, Sports news app
 StandingItem 380, 381
 StandingsModuleView 381
 State
 about 140, 141
 versus Binding 141
 SwiftUI Watch Tour
 Activity Ring, creating 57-59
 bar charts 48
 chart Page-View navigation,
 creating 44, 45
 charts, building 48
 navigation, building 41
 Swift previews, using 47
 SwiftUI watch list, creating 45, 46
 wedge, creating 59, 60

T

TeslaOrderFormApp 150
 TextField 5-8
 Text view 3, 4

V

View model
 implementing 215-218
 View model, Sports news app
 adding, to HomeDashboardView 374
 creating 369-372
 FeaturedPlayerModuleView,
 updating 377, 378

feeds, connecting to
 FeaturedArticleModuleView 377
implementing 372, 374
PlayerCardView, updating 378, 379
RosterView, updating 379
VideoItem, updating 376
VideoModuleView, finishing 376
VideoModuleView, updating 375
views
 about 3
 Image 9, 10
 layout 21
 presentation 21
 Text 3, 4
 VStack
 using 21, 22
 with spacer 22, 23

W

watch UI
 accessibility 65
 building 62
 card contents, adding 73
 challenge 102
 custom fonts 66, 67
 data, adding 95-97
 data, adding to DraftCardView 101
 data, updating for
 DraftRoundCardView 100, 101
 design specs 63, 64
 draft cards, designing 71
 draft list, updating with
 pick data 99, 100
 fonts 65
 header, adding 72, 73
 menu, designing 68-70
 menu, updating with DraftRound 97, 98

properties, adding 71
prospect details, designing 78, 79
system fonts 65, 66
views, refactoring 90
watch UI, card contents
 bottom of card, adding 74-78
 top of card, adding 73, 74
watch UI, prospect details
 details prospect header, creating 79-83
 detailed prospect info, creating 87-90
 detailed prospect stats, creating 84-87
 views, connecting 90
watch UI, views
 ContentView, refactoring 90, 91
 detail view, refactoring 95
 DraftList, refactoring 91-95

Z

ZStack 27-31

