BLCKBIRDS

SwiftUI Basics for Beginners

Getting started with intuitive and rapid iOS app development

IMPORTANT:

We strongly recommend you to not only read this book but to try everything yourself immediately. This helps strengthen your skills!

SwiftUI is still pretty new. This means that there can occur some bugs and it's likely that Apple will update its framework more often than usual, which can lead to some problems within the code used in this book. However, whenever this is the case, this book will get updated as fast as possible. You can always contact us if you have some problems, suggestions or wishes!

TABLE OF CONTENT

SHARE YOUR PROGRESS!

Tag **@BLCKBIRDS** or use the **#swiftuibasics** hashtag
for getting shared on Instagram and Twitter!

Chapter 1

PREFACE

- What SwiftUI is
- How SwiftUI differs from the “old" way of coding iOS apps
- The principles and advantages of working with SwiftUI

Thank you! ❤️

First of all, we wanted to thank you for downloading this eBook and taking the time to read it!

In this book, we will give you a detailed introduction into Apple's newest app development framework SwiftUI. After a few general words about SwiftUI and the principles behind it, we'll go briefly through Apple's IDE (Integrated Development Environment). If you have already worked with Xcode, you can feel free to skip this chapter!

Then we'll learn how to use SwiftUI views to create our own user interfaces in a block-by-block approach! Next, we will dig deeper and take a look at the data flow concepts used in SwiftUI. Then, we will learn how to present data within lists and how to connect multiple views hierarchically and finally we learn we’re getting to know @ObservableObjects and how to use them the as the view’s data model

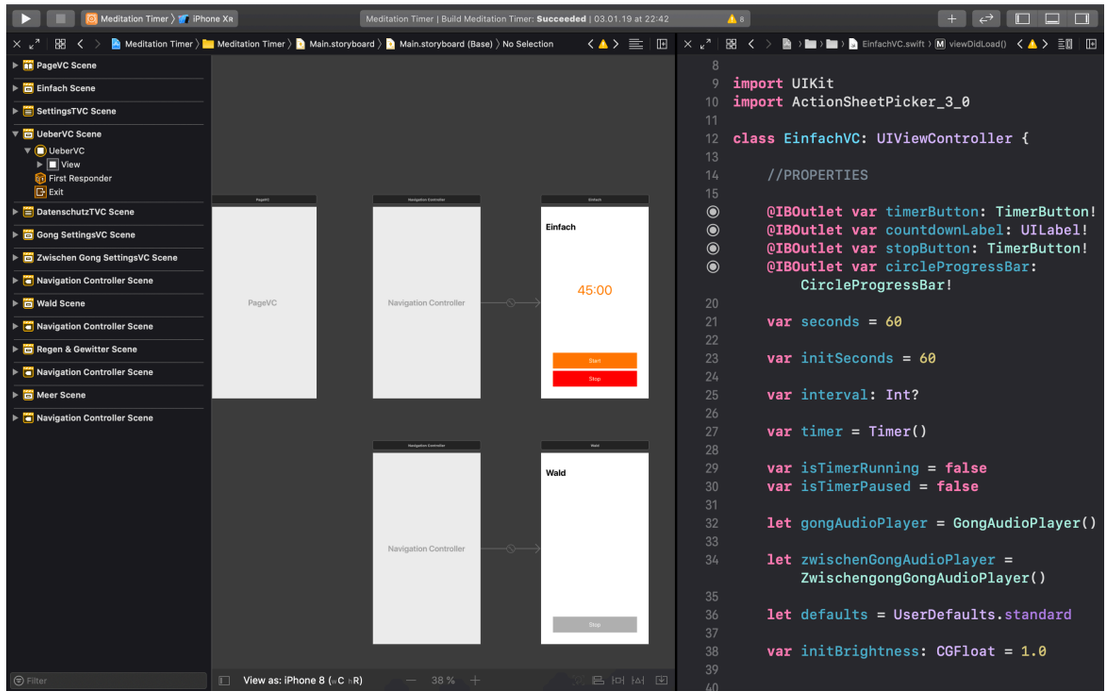
After that, you should already be able to write your own little application with SwiftUI!

What is SwiftUI 🤷‍♂️

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

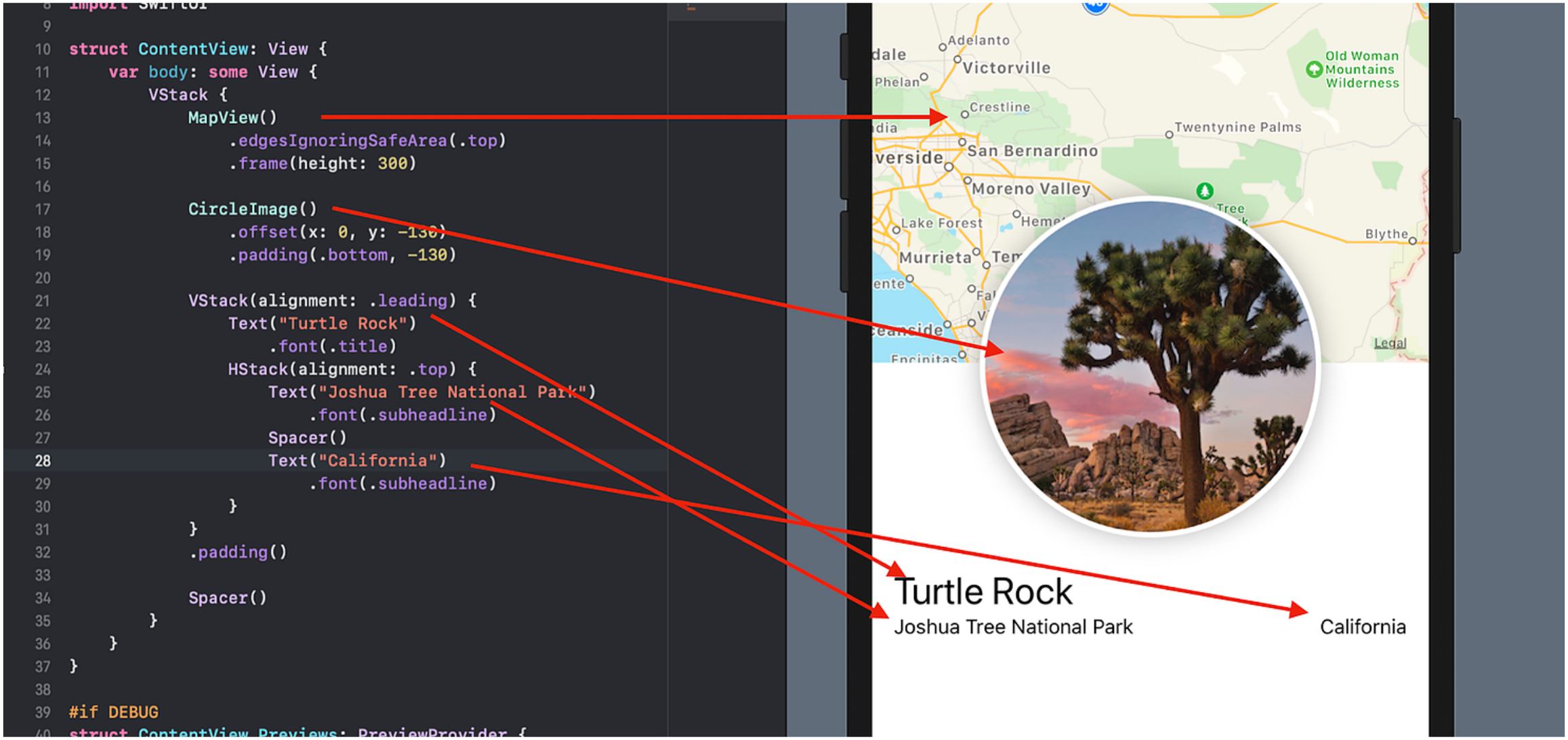
As a comparison: Until now, the most common way to build iOS apps was to use storyboards. Simplified, you created the UI by dragging and dropping, arranging and constraining elements and by connecting them to your code. But to see the final result you were forced to compile and run your app every time you made some changes.

Due to the fact that you had two sides that can easily differ from each other, the storyboard UI on the one hand and the code on the other, this approach often led to bugs or crashes.



Building apps with old-school storyboards often leads to lack of visual feedback, bugs and frustration.

With SwiftUI, Apple introduced a new framework that works completely different. Instead of having two separate layers to work with, the UI and the code, SwiftUI merges them together. This gets clear when you look at the approach of how SwiftUI works.



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

With SwiftUI you use code to describe how your app should look like! With the new syntax, you just describe how your interface should look like and how it should behave. Look at the image above. You see that every part of the UI, the map, the circled image and the texts, is described by code.

And that's one great thing about SwiftUI! Everything you write down gets instantly reflected in your live preview. No time-intense run & compile workflow anymore! Everything you describe within your code get's instantly displayed to you!

And that's not all: You can even manipulate the UI inside the live preview itself. For example, you can drop and drag another text objects or manipulate the color of a certain object, as we'll see later. Every change in the live preview affects your code and vice versa!

This new workflow saves us a lot of time and simplifies app-building tremendously. And because code and UI are always linked and dependent, you don't have to worry about broken outlets and actions anymore.

How does SwiftUI differ from working with UIKit + storyboards ❓

If you never developed iOS apps before, feel free to skip this paragraph. However, if you already worked with UIKit and storyboards you're maybe wondering how the programming approach used in SwiftUI differs from working with UIKit and storyboards. As you saw on the last page, the syntax used in SwiftUI really differs from what you used to know in iOS programming. This is because SwiftUI follows a declarative approach instead of an imperative one. Imperative programming was the common programming approach. But what's the difference? In a nutshell: With imperative programming, we write how we react when the state of our app changes and accordingly execute code. With declarative programming instead, we describe the interface of our app for all possible states in advance. Here's a great article if you want to dive deeper.

Technical requirements for running SwiftUI 🏃

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

Personal requirements 🤓

Although we try to explain everything as easy possible and always step-by-step, we recommend you to be at least familiar with the basics of theSwift programming language. However, you don't have to be a professional programmer to understand and apply what you will learn in this book. If you are a beginner, we recommend you to read our free Swift 5 Programming for Beginners eBook!

By the way, you don't necessarily need to be familiar with building iOS apps using UIKit, although it could give you a slight advantage. But don't worry when you don't know what we're talking about right now!

That's all, let's get our hands dirty and start building some awesome SwiftUI apps!

Chapter 2

INTRODUCTION TO XCODE 11

- Getting to know Apple's in-house IDE
- How Xcode is structured
- How to create app projects

Before we are getting started with creating our first SwiftUI projects, let's take a quick tour through Xcode.

If you are already familiar with Xcode and worked with it before, feel free to skip this chapter. However, if you are completely new to iOS development, make sure you read this chapter carefully since it's crucial for learning iOS app development!

What is Xcode used for?

Xcode is Apple's in-house IDE (integrated development environment). When you develop iOS apps, it is *the* software you work with most often. Therefore, it is very important to have a profound knowledge of Xcode and to master the basics from scratch.

In Xcode you set up the user interface of your app, organize and write the code that makes your app run. Xcode also offers you the possibility to run and test your app in a virtual simulator on your Mac (and of course on a real iOS device).

Creating an Xcode Project and choosing the User Interface

To get in touch with the IDE, just open Xcode and click on "Create a new Xcode Project" (by the way: "Playgrounds", the option below creating a new project, is an option to test new concepts and ideas quickly and easily, here you can read more about it). Next, click on "Single View App", then on "Next" and give your app any name you want.

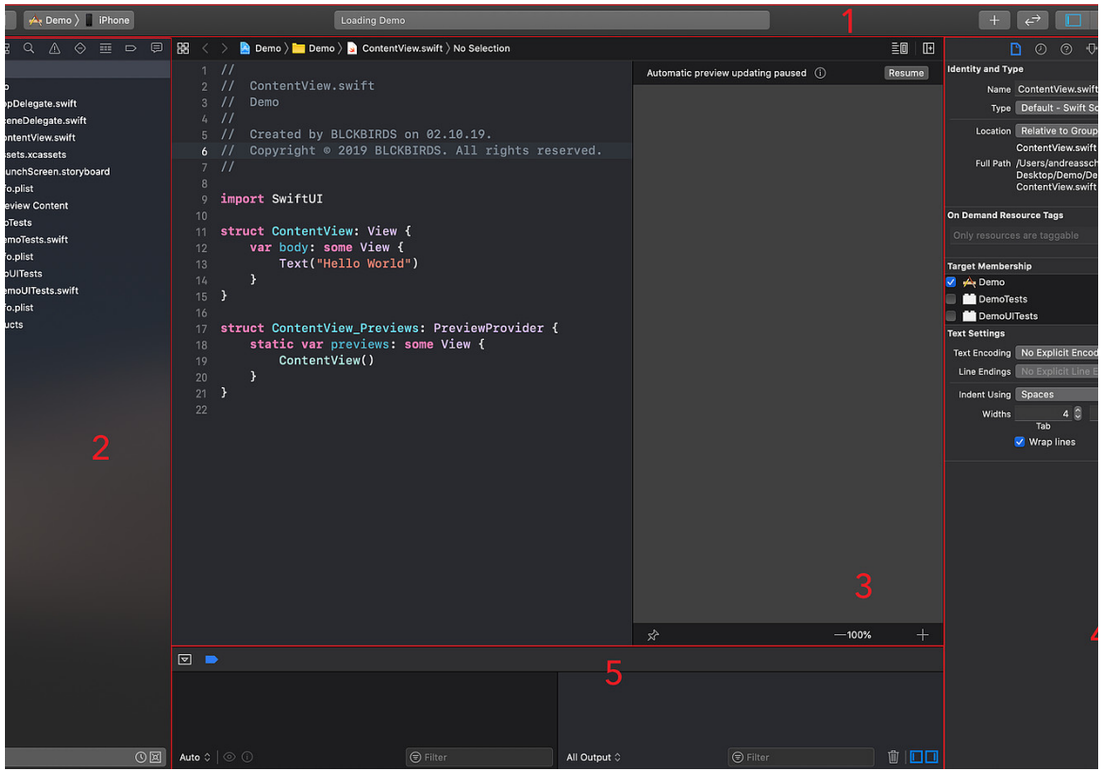
At this point, you can choose which User Interface mode you want to work with. You can either use SwiftUI or Storyboards. Because we are working with SwiftUI in this book, always choose SwiftUI as the User Interface for the following projects.

Understanding the Interface 💡

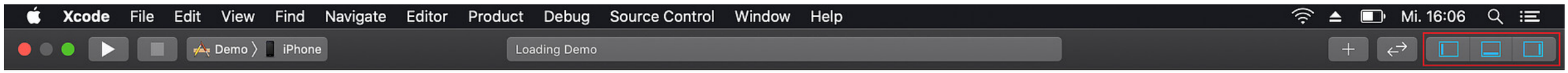
To understand the basic layout of Xcode, create a new Single View app by using SwiftUI, as we described above.

What you see now may feel a bit overwhelming. Don't worry, this feeling is perfectly fine and will pass when you understand the basics of how Xcode is constructed.

Fortunately, the interface is designed pretty straight forward. The interface you see now basically consists of 5 sections:



The five main sections are: The Toolbar(1), the Navigator Area(2), the Editor Area(3), the Utility Area(4), and the Debug Area(5). Let's take a quick look at each of them!



The right three buttons of the toolbar allows you to show/hide the different areas

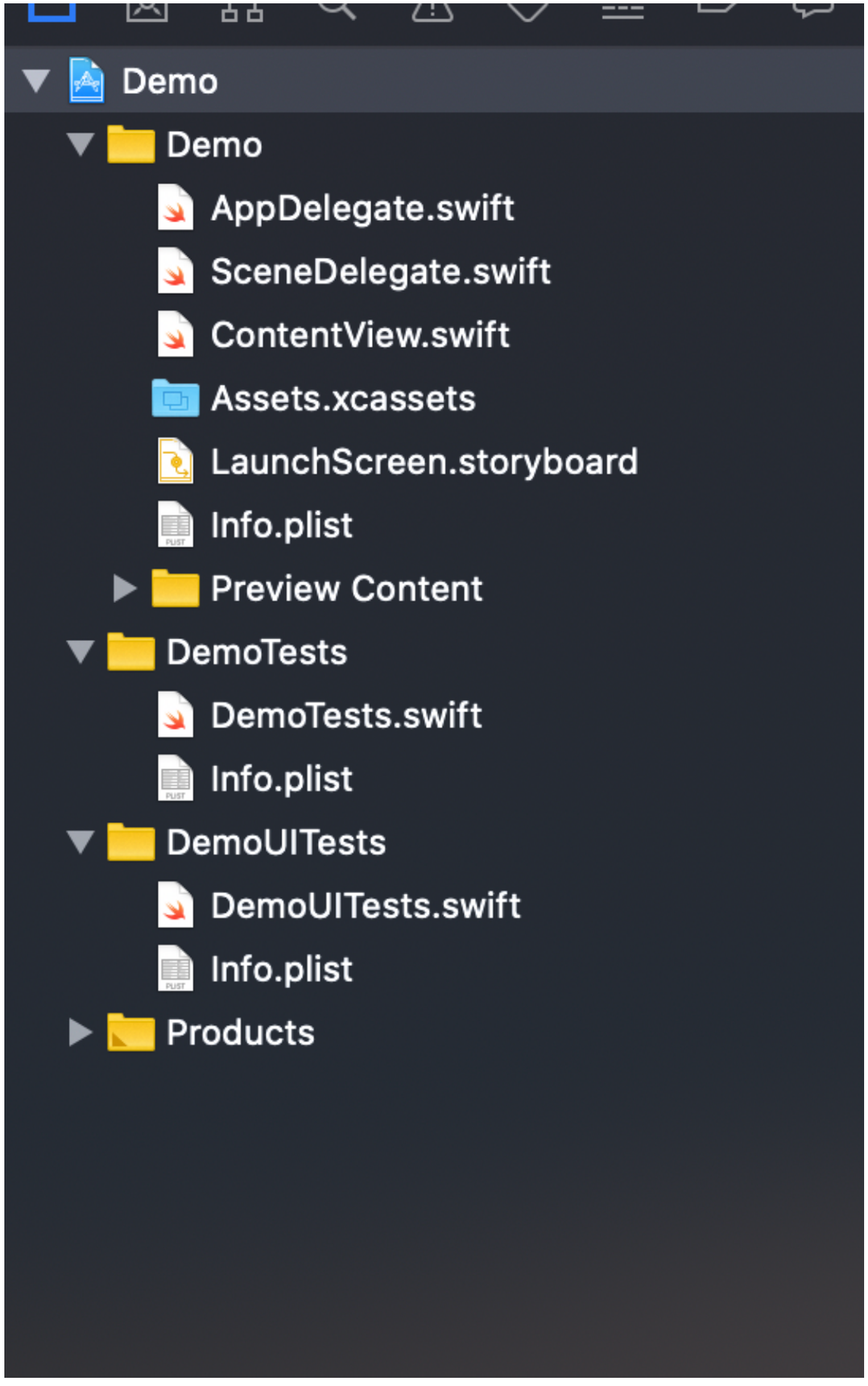
The Toolbar 🛠

The toolbar allows you to access the basic Xcode settings (don't mix this up with your app projects settings) and perform several operations. On the left side of the toolbar, you can select the device on which you want to run your app, for instance on any simulator. The field in the middle tells you when Xcode is working on something. The area on the right is responsible for showing/hiding the different areas of Xcode.

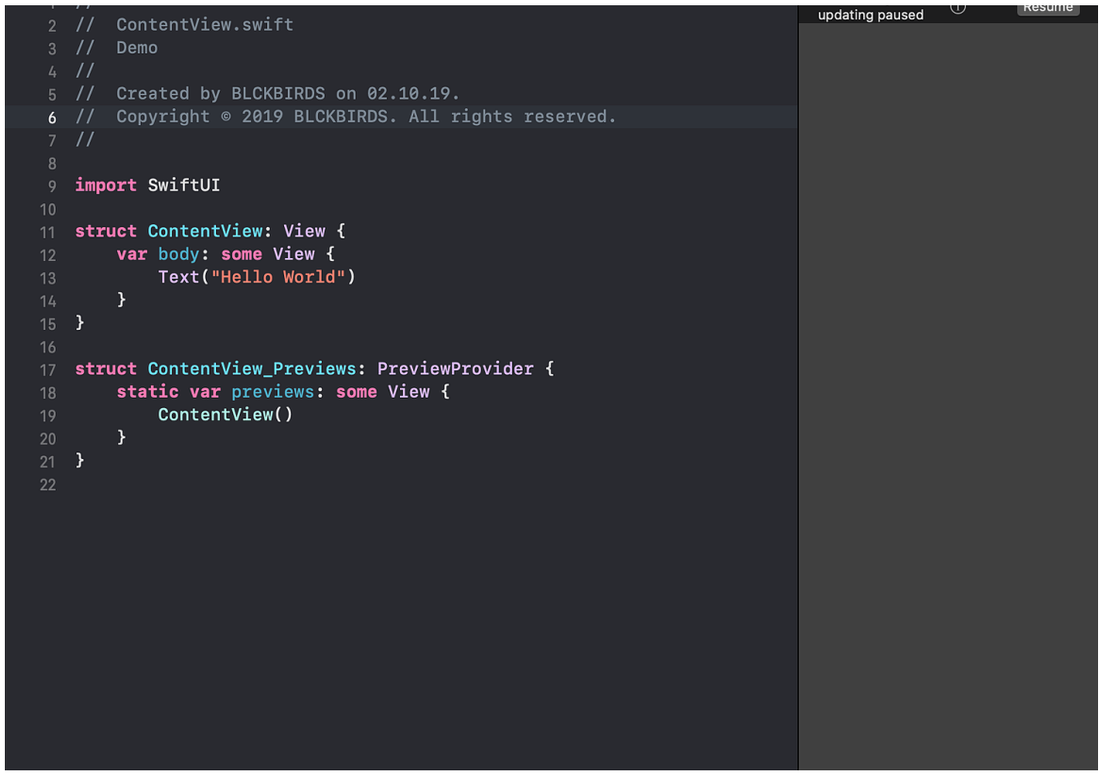
The Navigator Area 🔍

The Navigator Area helps you finding your way around your project and organizing your code and resources. By default, the "Project Navigator" is selected, probably the most important mode of the navigator. Here are the different parts of your app's code listed. The more complex your app becomes, the more files your project will contain. To keep track, you can create „groups“ (Xcode’s name for folders) and move the files as you like. Where you are placing files within your project navigator does not affect the logic of your code or the behavior of the app.

When you click on a file, it will open in the Editor Area where you can, you guessed it, edit it. If not already done, click on "ContentView.swift" to display this file in the Editor Area. Swift files are the heart of every iOS App. In these, you write the code that makes your app run.

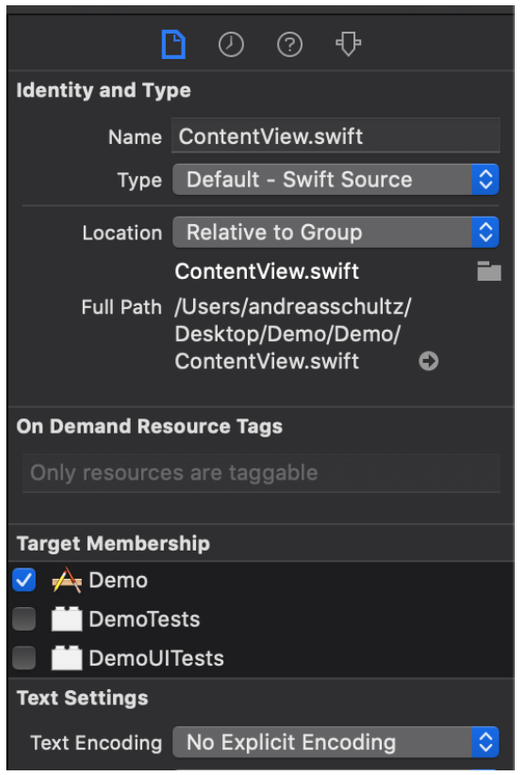


The Editor Area ✍️

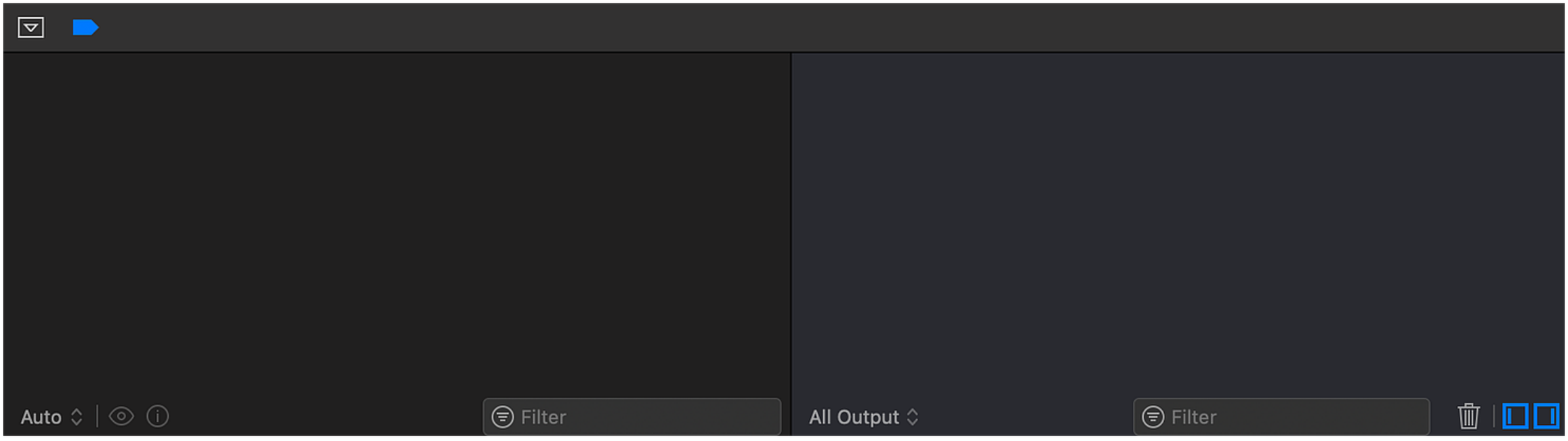


Here you write the code and compose the interface of your app. The appearance of the Editor Area depends on which file type is opened.

The Utility Area 🧰



Similar to the Editor Area, the appearance of the Utility Area depends on what file type you have selected. Here you can access, for instance, metadata, references, etc. of files or/and their components. Especially this area often confuses beginners, because the use of this area depends on the particular situation. But the more you work with Xcode the more you get a feeling for it. You'll see, it's much easier than it looks at first sight.



The Debug Area 👷

When you run your app, you will find all relevant information about errors etc. that Xcode provides you. This area becomes very important when it comes to finding and fixing errors and bugs of your app.

Often the output is very long, so you can use the filter to easily find certain output.

Conclusion 🎉

Congratulations! You now know the basics of Xcode.

Don’t feel insecure if you are a little overwhelmed. You'll see, the more you deal with it, the more comprehensible everything becomes.

In case you have trouble with working something out, do not hesitate to write a message.

Chapter 3

BUILDING USER INTERFACES

- Getting to know Apple's in-house IDE
- How Xcode is structured
- How to create app projects

Creating a SwiftUI Project 🚀

Let's dive right into it and start creating our first SwiftUI project. For doing this, open Xcode 11 and click on "Create a new Xcode project". Select "Single View App" as the application type and click on Next. Now you can give the project any name you want, in this example, we choose "SwiftUIBasics" as the Product Name. Make sure that SwiftUI is selected as the User Interface type. When you click on Next, you can choose where you want to save your project. Now you can finally create the Xcode project!

Xcode should open the default *ContentView.swift* file. If not, just click on it in Xcode's navigator section on the left side.

Now you see a split-screen. The left side is the SwiftUI code we use to build our app's interface. This code consists of two blocks:

- The *View* struct: This struct is the place where we write our code for building up our app. You see that the struct itself contains a body property. Inside this body, we create all the components we need for our UI, arrange and modify them.
- The *Previews* struct: This struct then renders the View with its body and displays it in the Preview Simulator on the right screen.

Every time you make some changes within your View struct, the Previews struct notices and displays the updated the View inside the live preview!

At default, Xcode inserts a Text object inside the View's body, reading "Hello World". Let's modify this object by changing its string to "I love SwiftUI!".

```
Text("I love SwiftUI!")
```

You see that this change gets instantly reflected in the live preview. Awesome!

Note: If the Preview Simulator is not running yet click on the "Resume" button in the upper right corner. Usually, changes are adopted automatically, but sometimes, especially when larger changes are made, clicking the Resume button gets necessary.

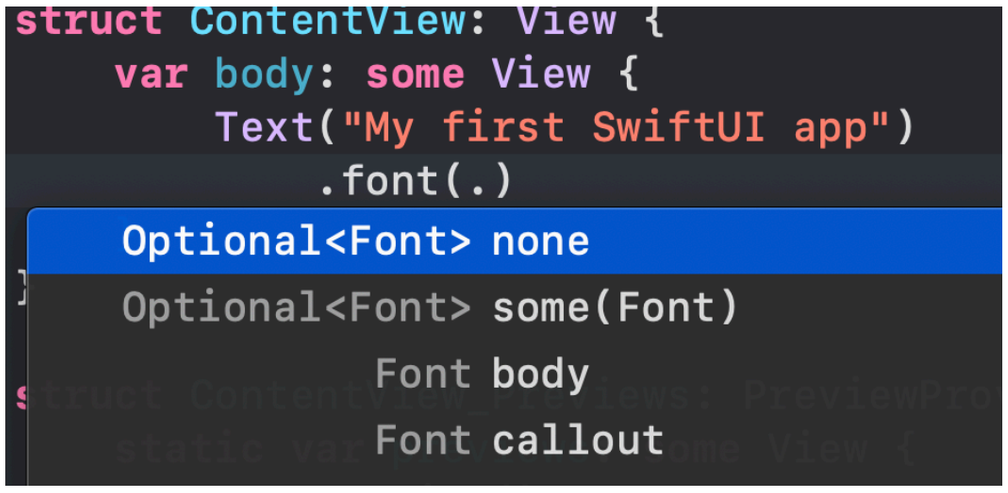
Modifying views 👨‍🎨

Let's say we want our Text to be of a larger font. We can easily do this by applying a modifier to it. As the name says, modifiers are used to modify view objects in SwiftUI. There are a lot of them and by going through this eBook you will meet many of them!

You can add a modifier to an object like this:

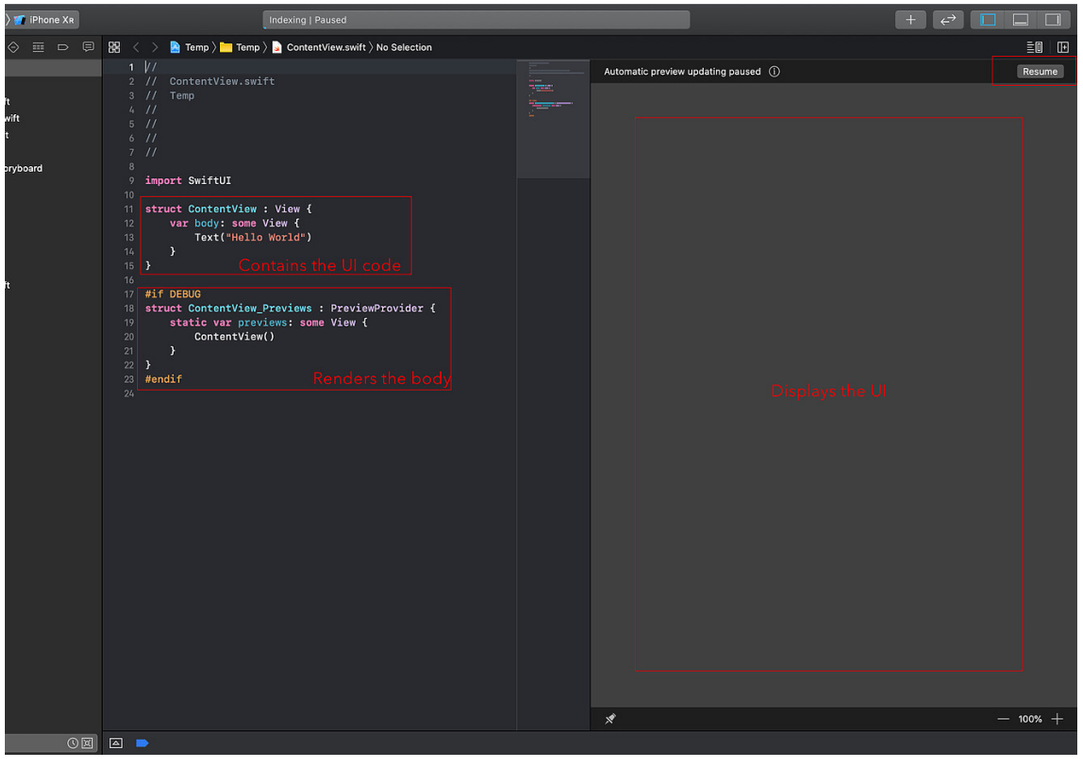
```
Text("My first SwiftUI app")
                .font(.largeTitle)
```

In this example, we choose the *.largeTitle* option. To see all of the other options, you can delete the code inside the braces and just write an ".", which shows you all available text styles. Feel free to try them out!

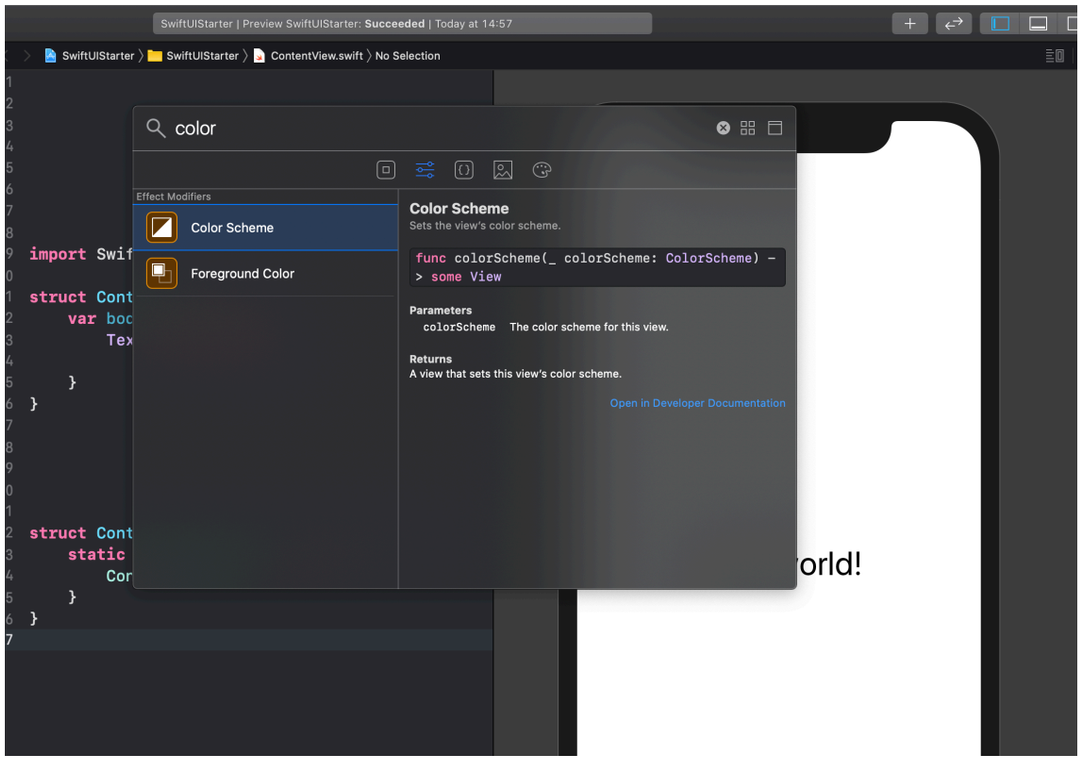


Okay, we saw how we can add modifiers by writing code. But the cool thing in SwiftUI is, that we can also do this directly in our live preview. CMD-click on the Text object in the simulator and click on „Show SwiftUI Inspector". Here we can adjust the object, for instance, change the text type.

We can also apply new modifiers via the preview. To do this, click on the right "+"-symbol on the right side of the Xcode toolbar.



The Preview struct renders the View struct and sends it to the live preview



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

A menu pops up with the different libraries available (represented by the five icons below the search bar). By default, the View library is shown. Click on the modifier symbol (the second one from left) to see all the available modifiers. You can scroll through them to get a feeling of which modifiers are available. Let's say we want to change the color of our Text. For doing this search for "Foreground Color". You can now drag and drop this modifier onto the Text object.

You see that the Text turns blue and also the code gets updated! To choose another color, you can edit the code. To see all available system colors, you can again delete the code inside the braces and just write an ".". But for now, we're fine with the blue color.

```
Text("My first SwiftUI app")
    .font(.largeTitle)
    .foregroundColor(.blue)
```

Stacking views

Next, we want to add another Text view below our current one. For doing this we have to use so-called Stacks. Stacks are used in SwiftUI to stack multiple view objects along a certain axis.

There are three types of Stacks:

- VStacks: All elements wrapped into a VStack get stacked vertically.
- HStack: All elements inside a HStack get aligned horizontally.
- ZStack: All elements inside a ZStack get stacked on top of each other.

Let's see how Stacks work! Let's say we want to place another Text below our current one. Because they should be aligned vertically, we have to use a VStack for this.

For embedding our Text view into a VStack, you can simply CMD-click on it (on the code side) and click on "Embed in VStack".

```
var body: some View {
    VStack {
        Text("I love SwiftUI!")
            .font(.largeTitle)
            .foregroundColor(.blue)
    }
}
```

Now we can insert another Text view and place it below our first one!

```
VStack {
    Text("I love SwiftUI!")
        .font(.largeTitle)
        .foregroundColor(.blue)
    Text("SwiftUI makes developing iOS apps
          super easy and fast.")
}
```

Next, we want to limit the width and height of our Text views. To do this we can use *.frame* modifier. By applying it to the whole VStack, both Text views are affected as a whole.

```
VStack {
        //...
    }
            .frame(width: 300, height: 100)
```

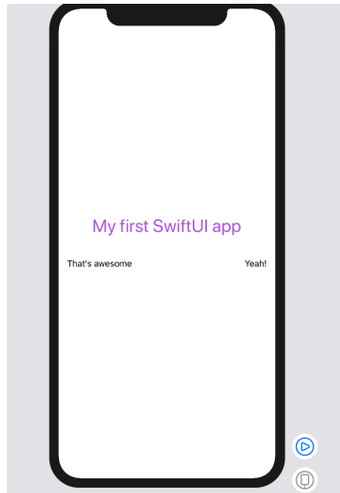
By using the *.frame* modifier, we are positioning the VStack's elements within an invisible frame.

At the moment the texts are centered. Instead, we want to align them on the left side. To do this, we can add braces behind the VStack and use the alignment argument to change the alignment mode of the views wrapped into the VStack. For aligning the objects on the left side we use the *.leading* option. We also want to increase the vertical spacing of the Texts by using the spacing parameter.

```
VStack(alignment: .leading, spacing: 10) {
        //...
    }
        .frame(width: 300, height: 100)
```

Let's add another object below our Text views: a button! But this time, we try doing it by using the live preview. Click on the "+"-symbol, open the view library (first icon) and search for Button. Now drag and drop it below our second Text view (make sure you insert it below the VStack and not into it.

Great, you see that SwiftUI automatically created another VStack and placed the one we created earlier into it. Below this wrapped VStack, we saw our new button view! Again we want to increase the vertical spacing:




```
VStack(spacing: 50) {
    VStack(alignment: .leading, spacing: 10) {
        //...
    }
        .frame(width: 300, height: 100)
    Button(action: {}) {
        Text("Button")
    }
}
```

Inside the curly braces we define the UI of the button. We want our button to read "Show me the logo!"

```
Button(action: {}) {
    Text("Show me the logo!")
 }
```

We want the Button's text view to have a blue background, a white font and rounded corners. To do this, we use the following modifiers:

```
Text("Show me the logo!")
    .background(Color.blue)
    .cornerRadius(10)
    .foregroundColor(.white)
```

We want to enlarge the Button's background. To add spacing around individual views, we use to *.padding* modifier.

```
Text("Show me the logo!")
    .background(Color.blue)
    .cornerRadius(10)
    .foregroundColor(.white)
    .padding()
```

But wait a second: Why did the *.padding* modifier not affect our button's text? This is because the order of the modifiers we append to a particular view matters! To understand this, let's take a look behind the scenes and explore how modifiers in SwiftUI work.

Why the order of modifiers matters 💡

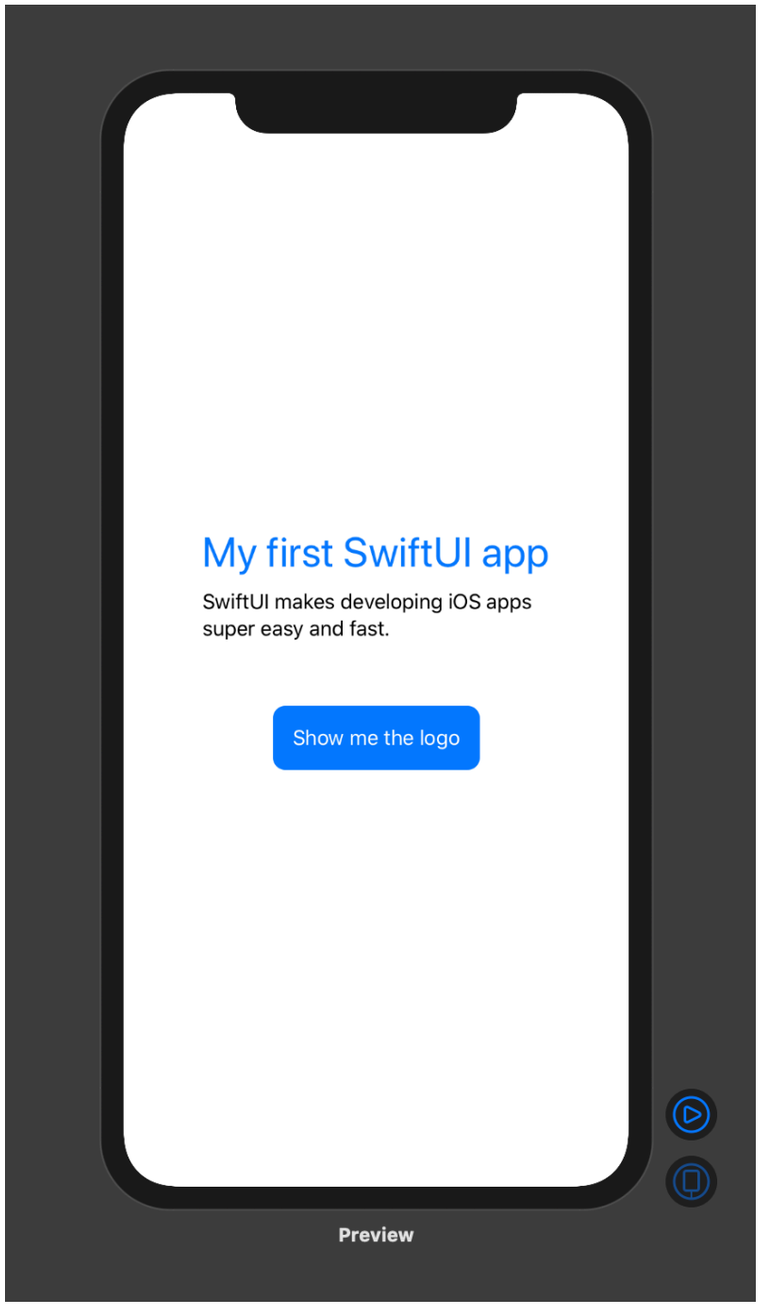
Let's take a close look at our Text view. We initialize the Text view containing the String "Show me the logo!". We added a background to it by applying the *.background* modifier. What happened next is that the modifier grabbed the Text view and created a new view out of it with changing the font type. So the key takeaway to remember is that applying a modifier to a view does not really modify the view itself but rather create a new, customized view!

When applying another modifier, in this case, the *.cornerRadius* modifier, SwiftUI created another new view by grabbing the created view (the view created from the *.background* modifier) and added a background to it. Applying the modifier (*.foregroundColor*) grabbed the last created view and initializes a new view out of it with changing the text font color. And so on!

And this is why applying the *.padding* modifier as the last one does not work as expected. The *.padding* modifier of the Text view grabs the last created view, a Text view without blue background and rounded corners. When we use the .padding modifier as the first modifier, the background modifier uses a Text view that already has some spacing around it and fills it with color! So let's change our button's text view:

```
Text("Show me the logo!")
    .padding()
    .background(Color.blue)
    .cornerRadius(10)
    .foregroundColor(.white)
```

This is how your preview canvas should look so far:



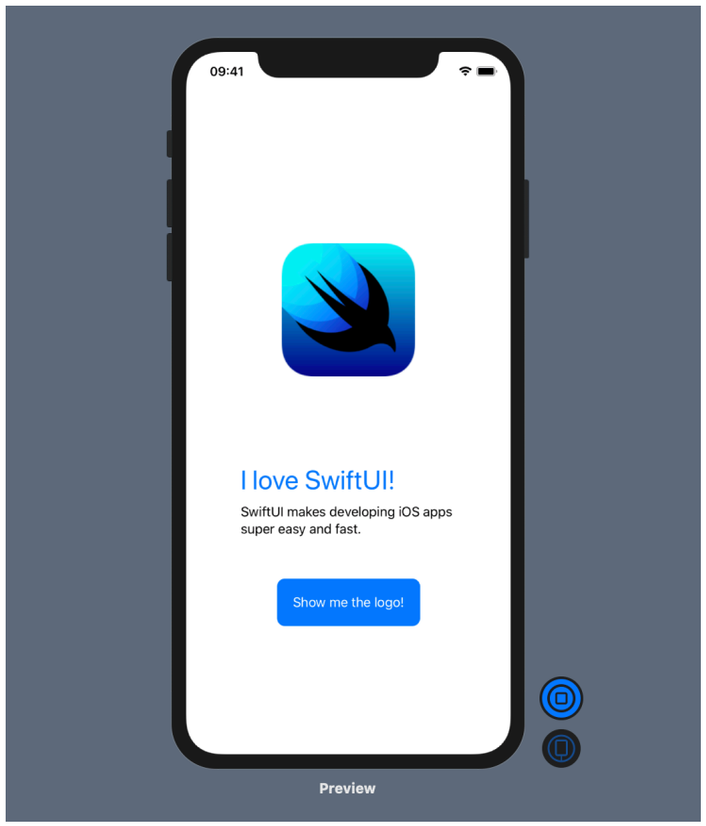
Chapter 4

DATA FLOW IN SWIFTUI

- Using image views in SwiftUI
- Understanding the data flow concept used in SwiftUI
- Understanding @State and @Binding property wrappers

What we want to achieve 🚀

When tapping on the “Show me the logo!” button, we want to show the SwiftUI logo above our Text views. By doing this we are learning the basic data flow concepts used in SwiftUI to react to external events.



Before we move on, we first need to import the SwiftUI icon into our project. To do this, open the *Assets.xcassets* folder and drag and drop the image file into it. Make sure the image is named correctly.

Using images in SwiftUI 🖼️

The SwiftUI icon should be placed right above our two Text views. Therefore, we insert an Image view above our inner VStack.

```
VStack(spacing: 50) {
        Image("logo")
        //...
    }
```

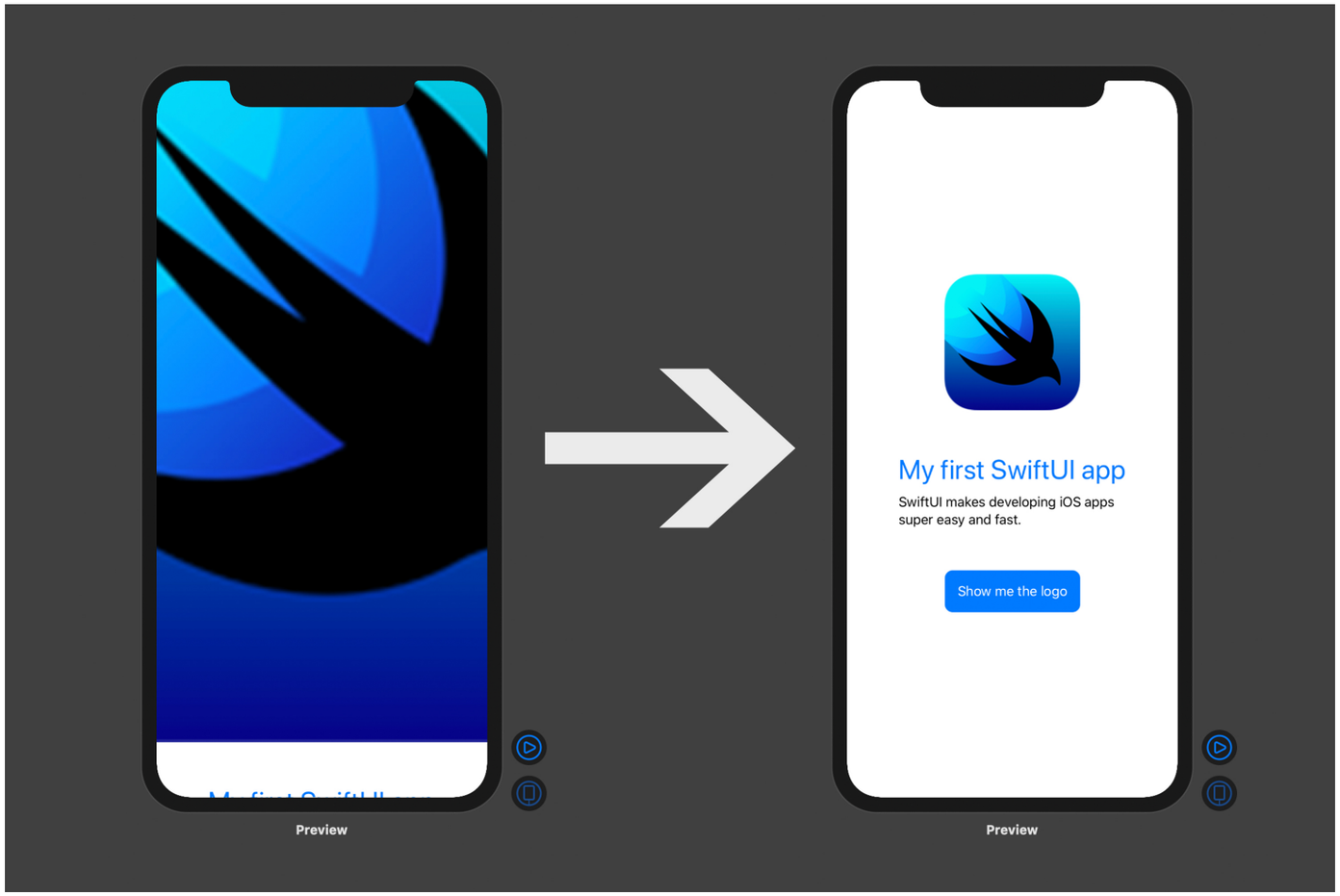
But by using a plain Image view without any modifiers, SwiftUI simply inserts the full-sized image into our *ContentView*'s body. Therefore, we need to give the image a frame which serves as a fixed-sized container for it. We do this by applying a *.frame* modifier again and giving the image's frame a suitable width and height. In order for the image to not exceed the frame, we have to add a *.clipped* modifier. The *.clipped* modifier cuts out the area of the object that exceeds the specified frame.

```
Image("logo")
    .frame(width: 170, height: 170)
    .clipped()
```

Next, we need to resize the image so that it fits into the frame without distorting the dimensions of the original image. To change the scaling of an image, we must use the *.resizable* modifier first. The *.resizable* modifier must be applied to enable changing the scaling of an image. It is important that the *.resizable* modifier is the first modifier of the image object (do you remember why?).

```
Image("logo")
    .resizable()
    .frame(width: 170, height: 170)
    .clipped()
```

Next, we apply the *.aspectRatio* modifier. This modifier scales the image so that it fits into the frame by using the specified dimensions. Since our icon is squared anyway, we could skip this step, but applying the *.aspectRatio* modifier is best practice when working with image views in SwiftUI and we should get used to it. For keeping the original dimensions of the image we can choose between the *.fill* or *.fit* contentMode. "Fill" scales the image so that the entire frame gets filled out with the image. "Fit" scales the image so that the entire image is visible within the frame.



With knowing the required modifiers, its super easy to frame image views in SwiftUI

In our example, we choose *.fill* as the content mode.

```
Image("logo")
    .resizable()
    .aspectRatio(contentMode: .fill)
    .frame(width: 170, height: 170)
    .clipped()
```

Getting familiar with @State properties 🔎

We want the SwiftUI icon to be only displayed after the user tapped on the "Show me the logo button!". But how can we achieve this functionality? To keep track of whether the corresponding image view should be shown or not we declare a showIcon variable above our *ContentView*'s body. At default, the logo shouldn't be shown, so we assign false to it.

```
struct ContentView: View {

    var showIcon = false

    var body: some View {
        //...
    }
}
```

We only want to show the SwiftUI icon, when the *showIcon* property is true. Thus, we conditionally initialise the Image view by using an if-statement inside the outer VStack. Note that if-else-statements can only be used inside Stacks!

```
VStack(spacing: 50) {
        if showIcon {
            Image("logo")
                //...
        }
        //...
    }
```

We could now change the *showIcon* value by using the Button's action parameter. But note the following: Whenever the user taps on the button, the *showIcon* property would get toggled though. But this wouldn't tell the *ContentView* update its body. This means that the *showIcon* would get true when the user taps the button, but the *ContentView* wouldn't notice and therefore wouldn't initialise the Image view by executing the code inside the if-statement.

So how can we get the view to rebuild itself when the user taps the button?

For this purpose, Apple provided @State properties within SwiftUI. These @-keywords (yes, there are more of them), are called property wrappers. Property wrappers in SwiftUI equip variables with a specific logic depending on the type of the property wrapper. This logic is the core of the data flow concept in SwiftUI, so understanding property wrappers is really crucial.

State is probably the most frequently used property wrapper in SwiftUI. We'll get to know most of the other ones throughout this book, but we start with this, basic one.

You can easily declare a State by putting the @State keyword in front of a variable. Let's do this with our *showIcon* property.

```
@State var showIcon = false
```

Hint: State properties must always be related to a view in SwiftUI. So make sure you always declare them inside a View struct (but not inside the View's body)!

We can now toggle the *showIcon* State when the user taps the button. Toggling means that the property gets true when it's false and vice versa.

```
Button(action: {self.showIcon.toggle()}) {
          Text("Show me the logo!")
                //...
      }
```

Again, what does a State property do? Well, you can read out and manipulate data just as you do with regular variables in Swift. But the key difference is that every time the data of a State changes, the related view gets rebuilt.

Let's explain it by taking a look at our app: Every time the user taps the button, the *showIcon* property gets toggled. Because this property is a State, its related view, the *ContentView*'s body, notices and rebuilds itself with checking the showIcon property and based on its value executing the if-statement with eventually showing/hiding the image view!

Try it out in the live preview! To use interactive elements like buttons in the canvas, you need to tap on the "Play" button next to it for starting a live preview. When you now tap the button, the *showIcon* State gets true which causes the whole view to rebuild itself with eventually showing us the SwiftUI. If we tap on the button again, the *showIcon* State gets false again, which causes the ContentView to hide the Image view again!

As said, this topic is very important for mastering SwiftUI, so make sure you really understood it!

Pushing views with Spacers ↔️

When showing and hiding the SwiftUI icon, the position of the remaining views changes. This is because the outer VStack place all of its elements in the center of the screen. Therefore, when the image view is not being shown the remaining views are getting pushed to the center. But how could we tell SwiftUI that the text views and buttons should always be pushed to the bottom, regardless of whether the image view is displayed.

We're going to use a Spacer view for this.

A Spacer creates an invisible space between the contextual objects and therefore pushes these objects to the side.A Spacer inside a VStack pushes the objects along the xAxis and a Spacer inside an HStack along the yAxis.

The image view should always be pushed to the top, the remaining views to the bottom. So let's insert a Spacer between them:

```
VStack(spacing: 50) {
    if showIcon {
        Image("logo")
            //...
    }
    Spacer()
    VStack(alignment: .leading, spacing: 10) {
        //...
    }
    //...
}
```

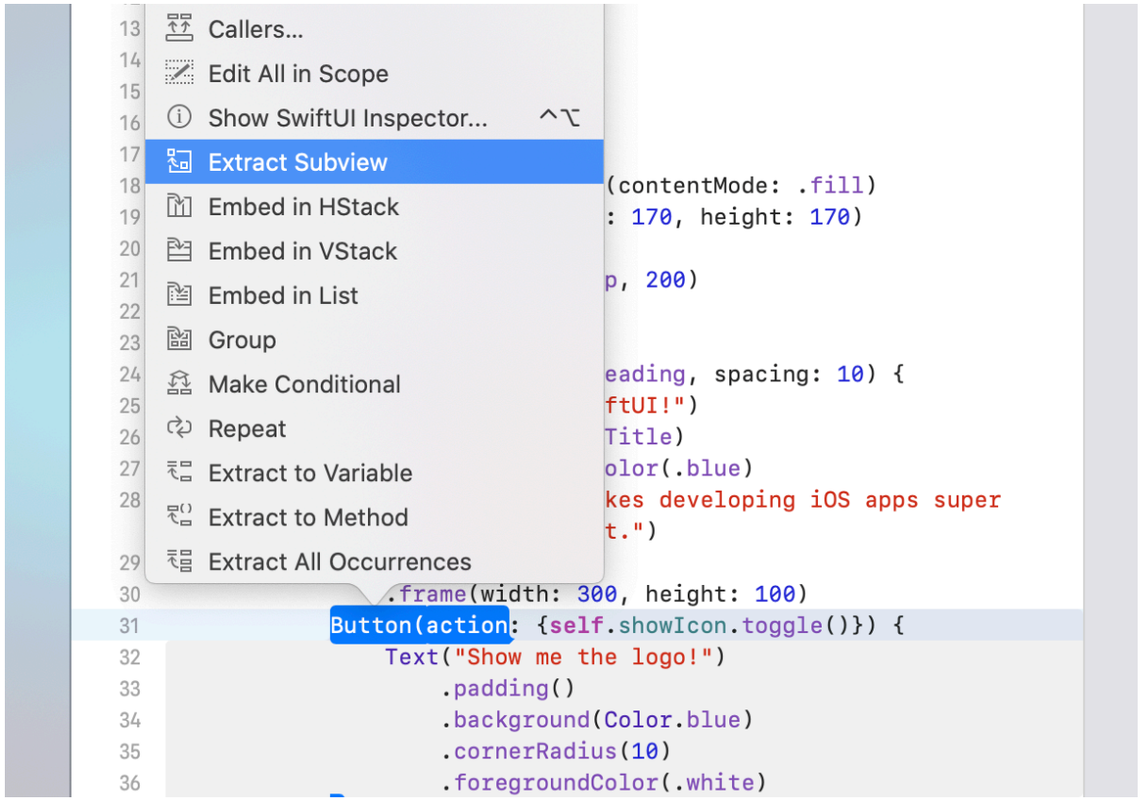
However we want our elements to be a bit more far away from the upper and lower edge. Therefore we add a top padding to the image view …

```
Image("logo")
    //...
    .padding(.top, 180)
```

… and a bottom padding to our button:

```
Button(action: {self.showIcon.toggle()}) {
    //...
}
    .padding(.bottom, 130)
```

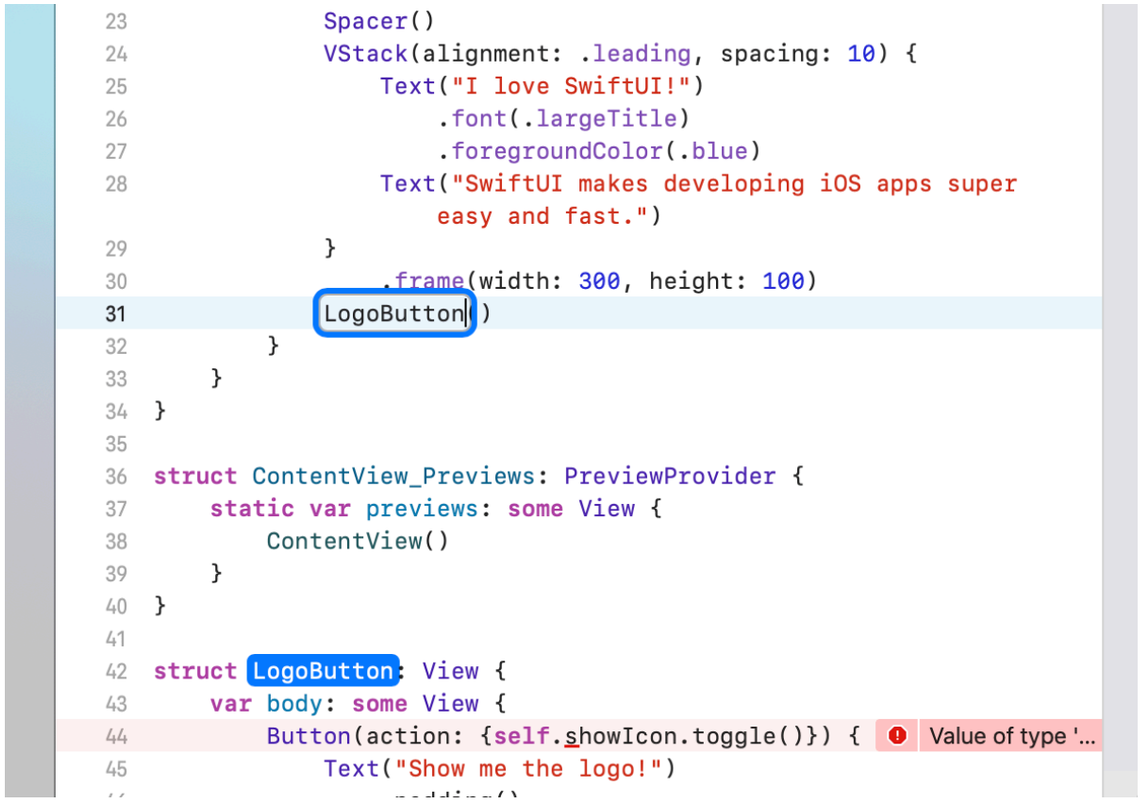
If we run our app again, the Text views and button will always be pushed down to the bottom of the screen (with a certain padding), regardless of whether the SwiftUI icon is being shown or not.



Understanding @Binding

Until now, we only worked inside our *ContentView* struct. However, when working in SwiftUI such structs can easily get confusing, for example when using multiple stacking layers.

Therefore, it's best practice to outsource views in their own structs whenever it's possible. So let's say we want to outsource our button. To do this, we can CMD-click on it and click on "Extract subview". Next, name your outsourced view, for example, "LogoButton".



We get an error saying "Value of type 'LogoButton' has no member 'showIcon'". This is because when outsourcing a view, it gets wrapped into its own struct. But the *LogoButton* struct doesn't own such a property.

To fix this, we need to insert an appropriate property into our outsourced *LogoButton* struct.

But: Instead of creating a State property on its own, we want to have a property that derives its content from the *showIcon* State of our *ContentView* in order to enable data flow between them!

To do this, we declare a @Binding property that's also called *showIcon* above our *LogoButton*'s body and since we want it to derive its content from the *showIcon* State of the *ContentView,* we don't assign any data to it. Bindings are used for creating a data connection between a view (here: our outsourced *LogoButton*) and its source of data (here: our *showIcon* State).

```
struct LogoButton: View {

    @Binding var showIcon: Bool

    var body: some View {
        //...
    }
}
```

To create the "link" between our *ContentView* and our *LogoButton* we initialise the latter by passing the showIcon State of our *ContentView* into it! We use the dollar-sign syntax for binding objects.

```
LogoButton(showIcon: $showIcon)
```

Now our outsourced *LogoButton* is connected to the *showIcon* State of our *ContentView*!

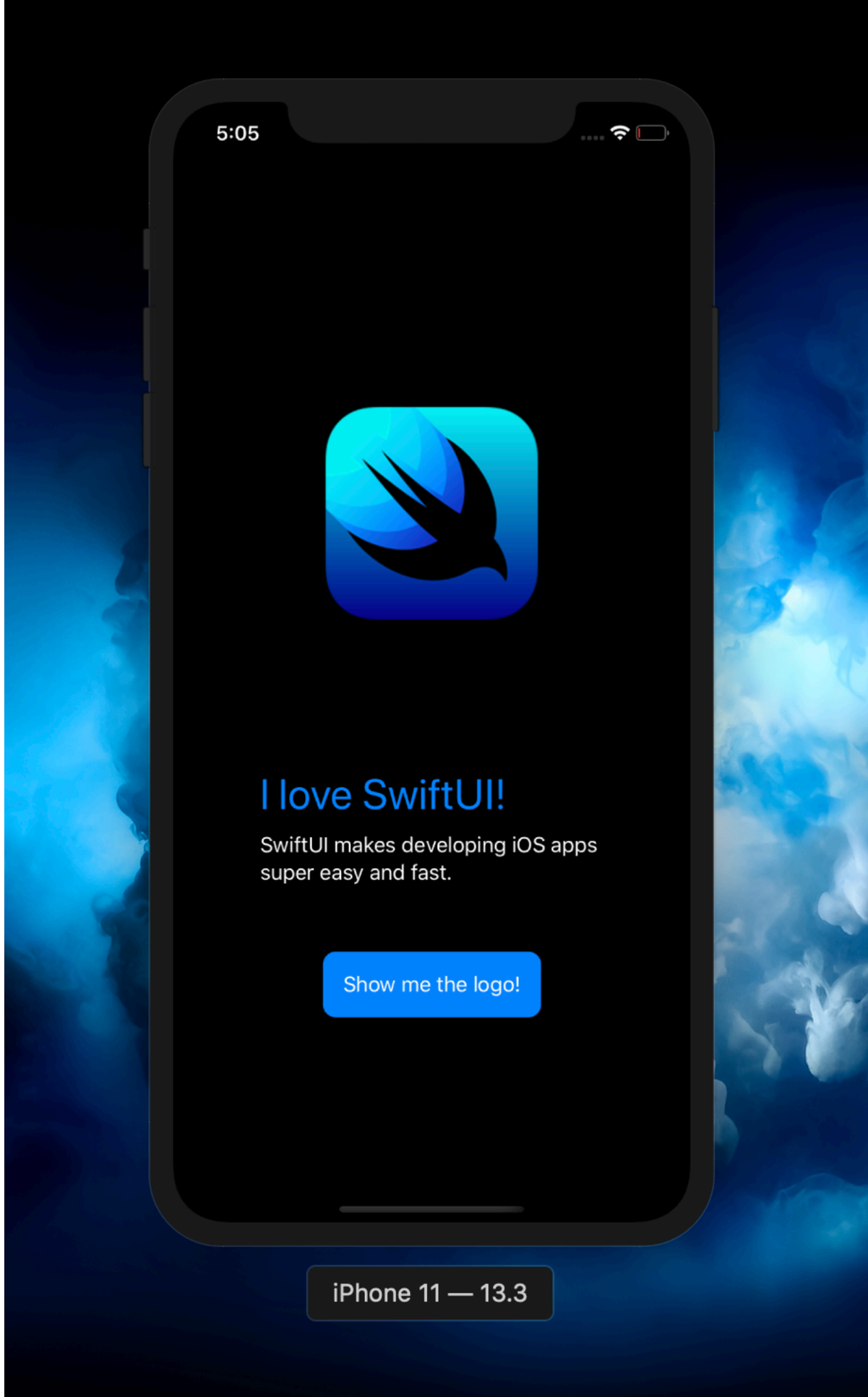
CONCLUSION

That's it! We are finished with creating our first SwiftUI app. You learned a lot of stuff like using several SwiftUI views like Texts, Buttons and Images and how to stack them. You also explored how to use States and Bindings.

In the next chapter, we are going to learn how to present data inside Lists and how to connect multiple SwiftUI views and stack them inside a view hierarchy.

You can find the whole source code here.

To learn more about other views like text fields, pickers, scroll views, toggles, sliders etc. make sure you check out our Mastering SwiftUI book. This eBook contains 14 chapters full of information about developing apps with SwiftUI. By going from basic concepts of SwiftUI, over more advanced topics like creating complex user interfaces to eventually communicating with a backend server you will learn everything you need to know about working with SwiftUI step by step!



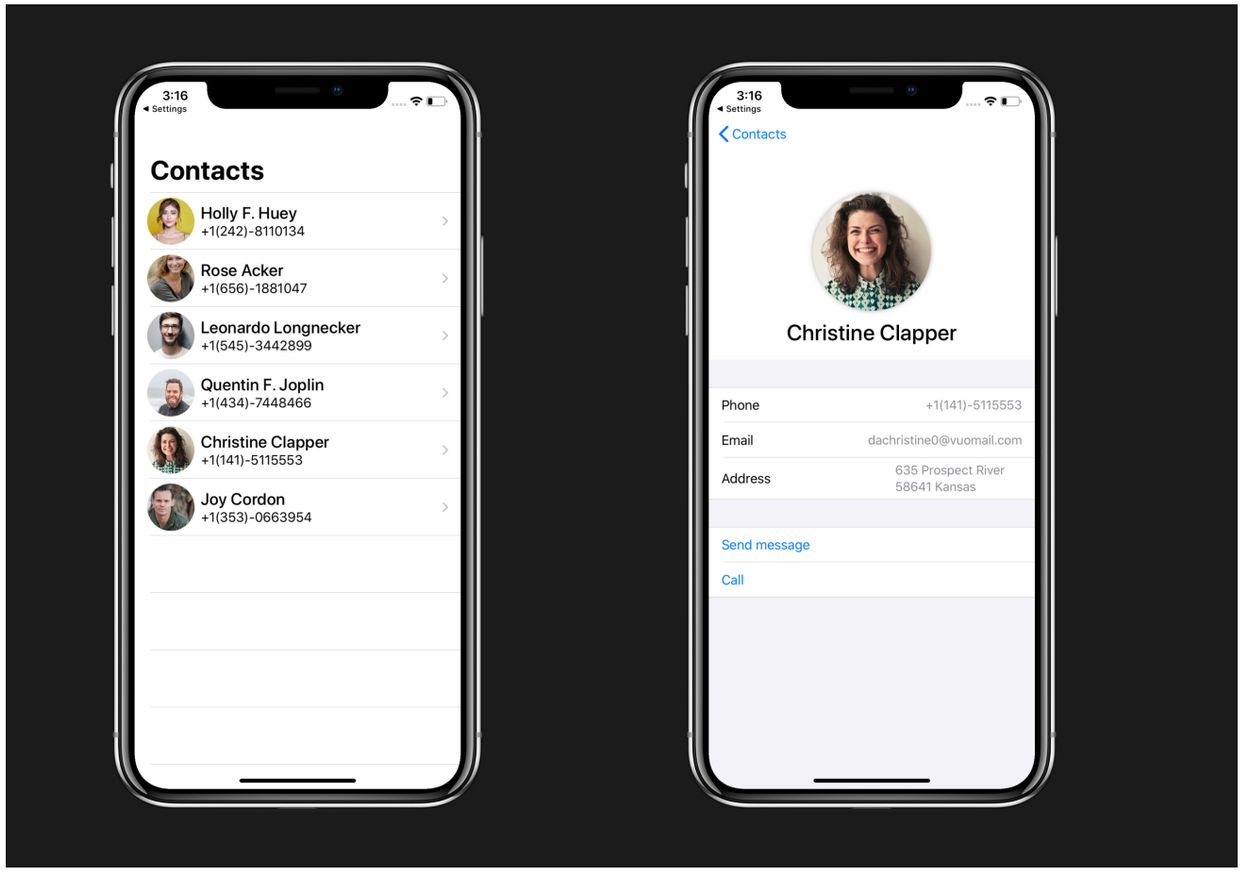
Chapter 5

USING LISTS AND NAVIGATION VIEWS

- How to present data by using Lists
- How to present a linked stack of views inside a navigation hierarchy
- How to cluster views by using Forms and Sections

What we'll achieve 🚀

In this chapter, we are going to create a contacts app. By doing this we'll learn how to present data in rows by using SwiftUI Lists. We'll also learn how to let the user navigate between multiple views, as you see in the preview below:



Setting up our project 👷

Let's start with creating a new project and importing the images we will need later on. Open Xcode 11 and create a new Xcode project. Then select Single View App and click on Next. Give the project a suitable name and make sure "Use SwiftUI" is selected. Then click on Create.

Our Xcode project shows up with presenting the default *ContentView.swift* file.

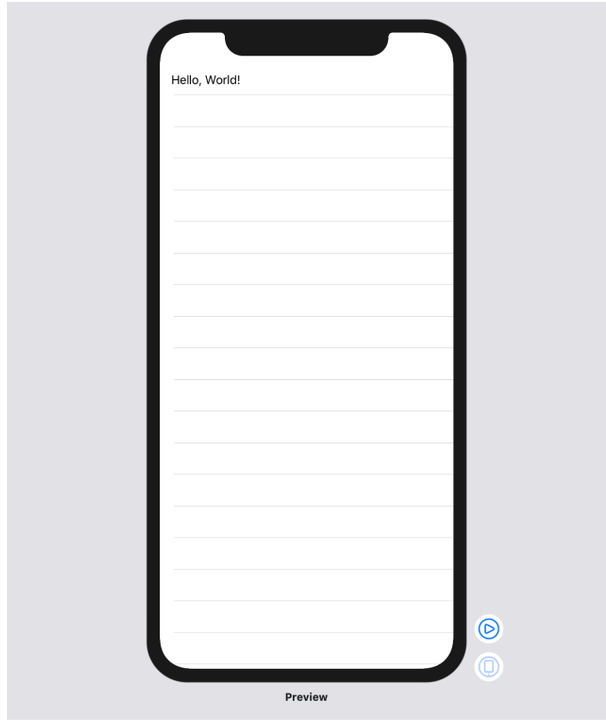
Before we start creating our contacts app, we first need to import the images we'll use for the photos of our contacts. To do this, open the *Assets.xcassets* folder and drag and drop the image files right into it. Make sure the images are named correctly.

Great, we're done with setting up our app's project. So let's dive into coding up our contacts app!

Creating a static list 🔽

A List is like a Table View you perhaps know from UIKit and contains multiple rows of data in a single column. Let's embed the default "Hello World" text into a List to see how it looks like:

```
List {
    Text("Hello, World!")
}
```



You see that SwiftUI automatically created multiple rows for us. Each view inside the List gets its own row, currently that's only our "Hello World" text.

Christine Clapper

+1(141)-5115553

Wisi mattis leo suscipit nec amet, nisl fermentum tempor ac a, augue in eleifend in venenatis, cras sit id in vestibulum felis in, sed ligula.

Each row in our List should contain the contact's photo, its name and its phone number. Let's start composing the row's UI by embedding a Text view into a VStack, replacing its String with a sample contact name and placing another text view reading a sample phone number below it.

```
List {
    VStack {
        Text("Christine Clapper")
        Text("+1(141)-5115553")
    }
}
```

Let's highlight the contact name by applying the *.font* modifier with the *custom* option to the corresponding Text.

```
Text("Christine Clapper")
    .font(.system(size: 21, weight: .medium,
        design: .default))
```

Let's choose the *.leading* alignment mode for our VStack:

```
VStack(alignment: .leading) {
        //...
    }
```

On the left side of the contact's name and number, we want to show its photo. To do this, we need to embed the VStack into a HStack first.

```
List {
        HStack {
            VStack(alignment: .leading) {
            //...
        }
    }
}
```

Next, we can insert a suitable image view above the VStack for creating a sample contact photo.

```
HStack {
    Image("christineClapper")
        .resizable()
        .aspectRatio(contentMode: .fill)
        .frame(width: 60, height: 60)
        .clipped()
        .cornerRadius(50)
        VStack(alignment: .leading) {
            //...
        }
}
```

For keeping our code clean, let's outsource our HStack and by CMD-Clicking on it, choosing "Extract subview" and calling it ContactRow.

Before we can make our List dynamic, we need to define a data model for representing the data for each contact inside the list.

Defining the data model 🛠

Let's create a new Swift file by clicking File - New - File and then selecting Swift File. Let's call this file *Contact*. Make sure the Foundation and SwiftUI Kit is imported. For handling our contacts we create a class called *Contact* inside this file.

```
import Foundation
import SwiftUI

struct Contact {

}
```

We want to know the following information of each contact: Its name, phone number, email, its address and the name of the corresponding image inside our assets folder. Therefore we declare corresponding attributes inside our class.

```
struct Contact {
    let imageName: String
    let name: String
    let phone: String
    let email: String
    let address: String
}
```

Because we will use our *Contact* data model to wrap multiple instances of it into a List, we need it to conform to the *Identifiable* protocol. This is required to pass custom class instances into Lists in SwiftUI (it's also the reason why we imported the SwiftUI framework into our *Contact.swift* file). The *Identifiable* protocol has only one mandatory requirement: It needs the class to contain an attribute to identify every instance by a unique id. Therefore we simply declare an id attribute and assign an *UUID* instance to it when initialising a *Contact*. Using *UUID* instances is automatically creating unique id's for us.

```
struct Contact: Identifiable {
    let id = UUID()
    let imageName: String
    let name: String
    let phone: String
    let email: String
    let address: String
}
```

Below our *Contact* struct, we can insert an array for holding all the contacts for our app.

Feel free to copy and paste the array from here into. Of course, you can edit this array to your wishes!

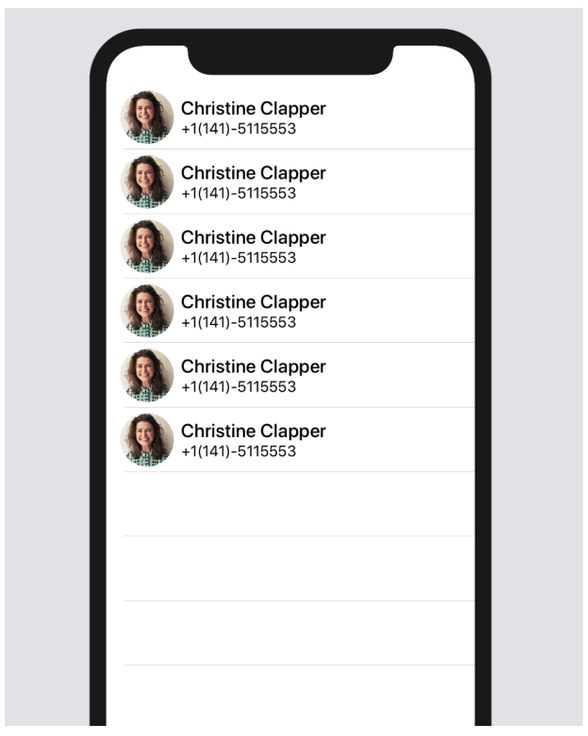
```
let contacts = [
    //...
]
```

Making our List dynamic 🔄

Back to our *ContentView*: We are now ready to pass our *contacts* data set to our List like this:

```
List(contacts) { contact in
        ContactRow()
    }
```

By doing this, our List cycles through our *contacts* array and creates one *ContanctRow* for every *Contact* instance inside it. Your preview canvas should now look like this:



Of course, not every row should display the same sample contact data. Thus, we add a contact variable to our *ContactRow* struct …

```
struct ContactRow: View {

    let contact: Contact

    var body: some View {
        //...
    }
}
```

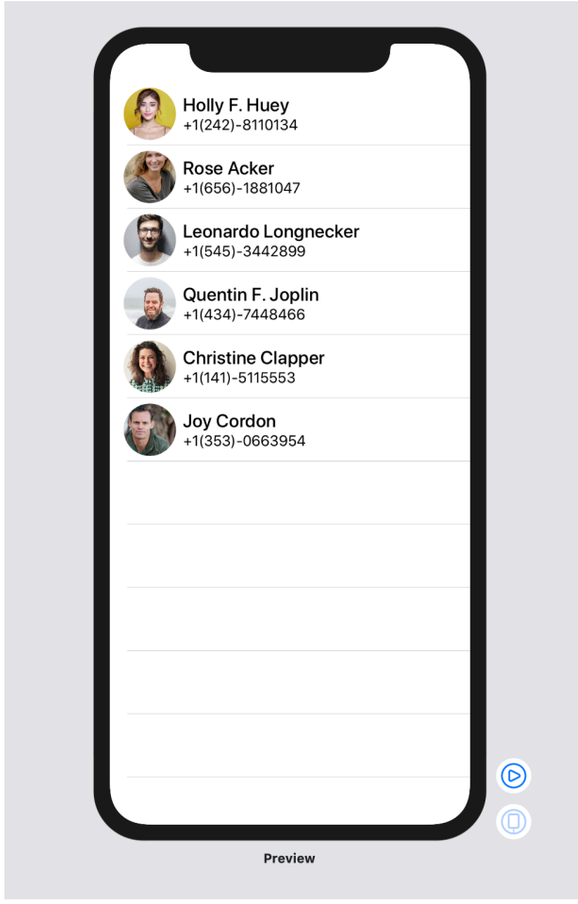
… which we initialise with the particular *Contact* instance in our *ContentView*'s List:

```
List(contacts) { contact in
        ContactRow(contact: contact)
    }
```

We can now replace the Strings in our *ContactRow*'s Text views with the corresponding properties of the passed *contact* property.

```
HStack {
        Image(contact.imageName)
            //...
    VStack(alignment: .leading) {
        Text(contact.name)
            //...
        Text(contact.phone)
    }
}
```

If you take a look at your preview canvas you see that the List successfully cycles through all elements in the *contacts* array and creates one *ContactRow* for each by passing that particular *Contact* instance to the *ContactRow* view.



Composing the DetailView 🖌

When the user taps on a particular contact row, we want to present a new view with detailed information about this contact.

To do this, we create a new SwiftUI view and call it DetailView. As we did in our *ContactRow* view, we declare a contact property which will be initialised in our *ContentView* later on.

```
struct DetailView: View {

    let contact: Contact

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

However, the corresponding previews struct needs to know which contact to use in order to render the preview canvas. For this purpose, we just use the first element of our *contacts* array.

```
struct DetailView_Previews: PreviewProvider {
    static var previews: some View {
            DetailView(contact: contacts[0])
        }
}
```

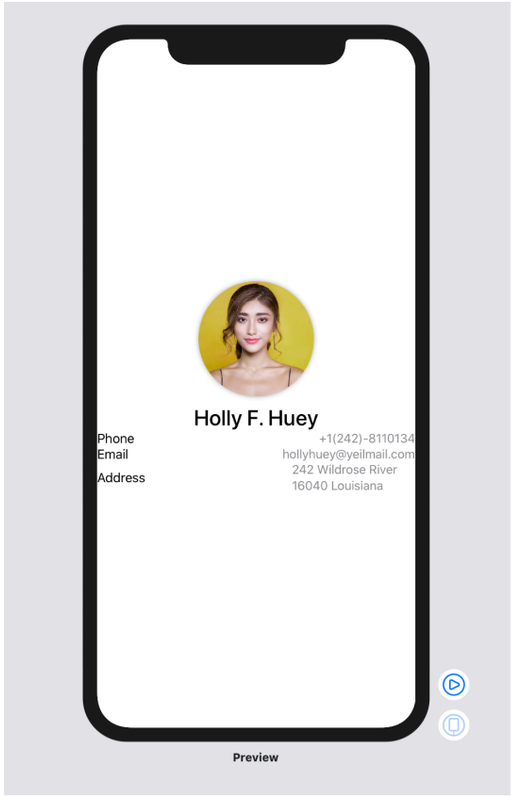
Now we're ready to setup the UI of our *DetailView*. We start with replacing the default "Hello World" Text with an Image view to display the contact's photo.

```
var body: some View {
     Image(contact.imageName)
          .resizable()
          .aspectRatio(contentMode: .fill)
          .frame(width: 150, height: 150)
          .clipped()
          .cornerRadius(150)
          .shadow(radius: 3)
 }
```

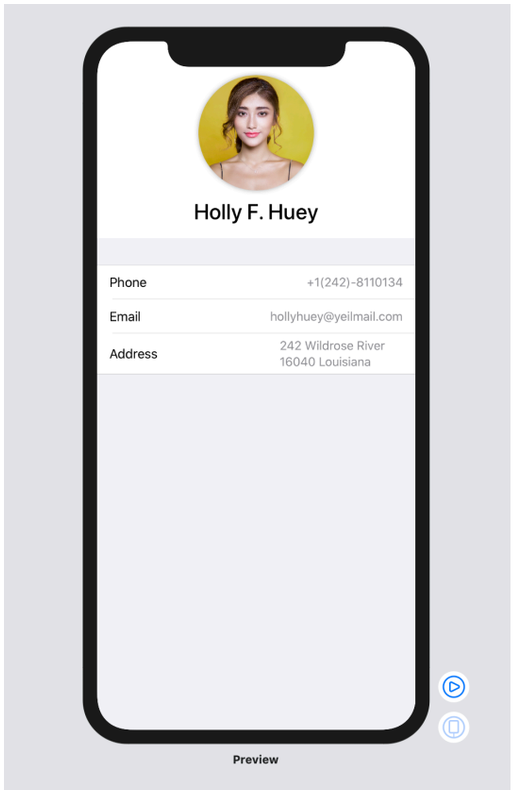
Next, we wrap the image view into a VStack and place a Text view below it for displaying the contact's name.

```
VStack {
          Image(contact.imageName)
               //...
          Text(contact.name)
               .font(.title)
               .fontWeight(.medium)
     }
```

Below the contacts name, we insert three HStacks for presenting the contact's phone number, email and its address:




```
VStack {
        //...
        HStack {
            Text("Phone")
            Spacer()
            Text(contact.phone)
                .foregroundColor(.gray)
                .font(.callout)
        }
        HStack {
            Text("Email")
            Spacer()
            Text(contact.email)
                .foregroundColor(.gray)
                .font(.callout)
        }
        HStack {
            Text("Address")
            Spacer()
            Text(contact.address)
                .foregroundColor(.gray)
                .font(.callout)
                .frame(width: 180)
        }
    }
```



Honestly, that doesn't look very nice. Let's change this by wrapping the three HStacks into a so-called form.

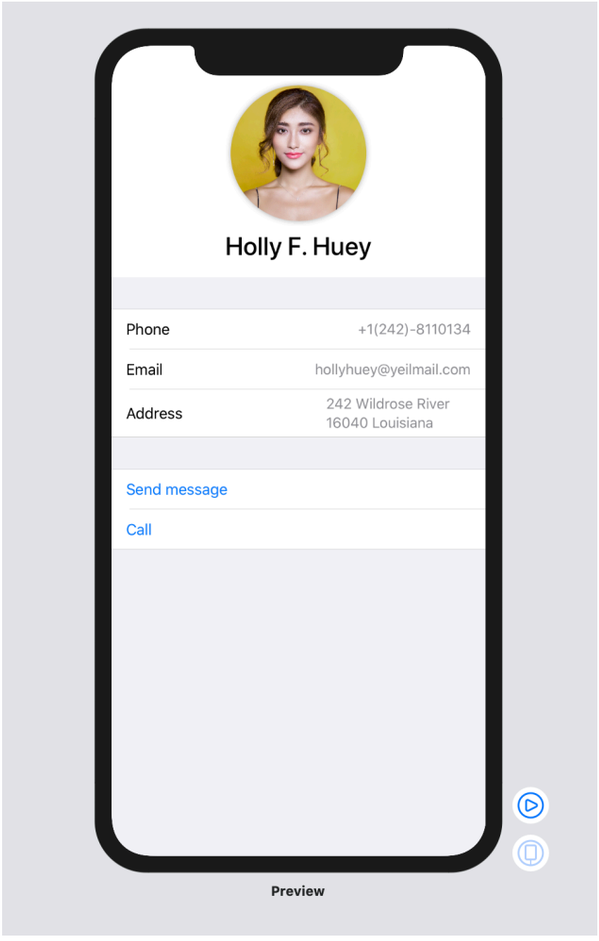
```
Form {
    HStack {
        //...
    }
    HStack {
        //...
    }
    HStack {
        //...
    }
}
```

Looks much better now! But what did the form actually do for us? A form groups views in a manner you know for example from the system's settings app. The cool thing about forms is that SwiftUI automatically adapts the layout and appearance of the views wrapped inside the form for us! Let's say we want to add two views two our row: one button for texting the contact and one for calling him. But we want those two to be separated from the other views inside the form. To do this, we wrap those into Sections like this:

```
Form {
        Section {
        //...
    }
    Section {
        Button(action: {
                print("Send message")
            }) {
            Text("Send message")
            }
            Button(action: {
                print("Call")
            }) {
                Text("Call")
            }
        }
    }
```

Great, we're already finished with composing the UI of our *DetailView*.

The preview of the *DetailedView* should look like this:



Now it's time to connect it to the *ContentView* in a manner that when the user taps on a particular contact row it opens the *DetailView* with showing the detailed information about the selected contact.

Stacking views in NavigationViews

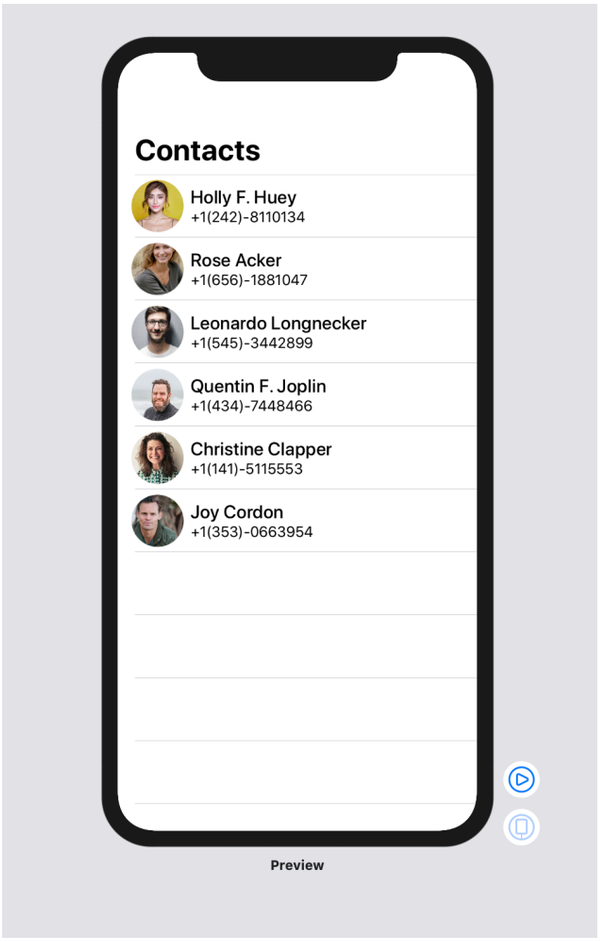
For being able to navigate to the corresponding *DetailView* when we tap on a certain *ContactRow*, we need to embed our ContentView's List into a NavigationView like this:

```
NavigationView {
    List(contacts) { contact in
        ContactRow(contact: contact)
    }
}
```

However, our *ContentView* doesn't have a navigation bar yet. To change this, we append the *.navigationBarTitle* modifier to our List.

```
List(contacts) { contact in
    ContactRow(contact: contact)
}
    .navigationBarTitle("Contacts")
```

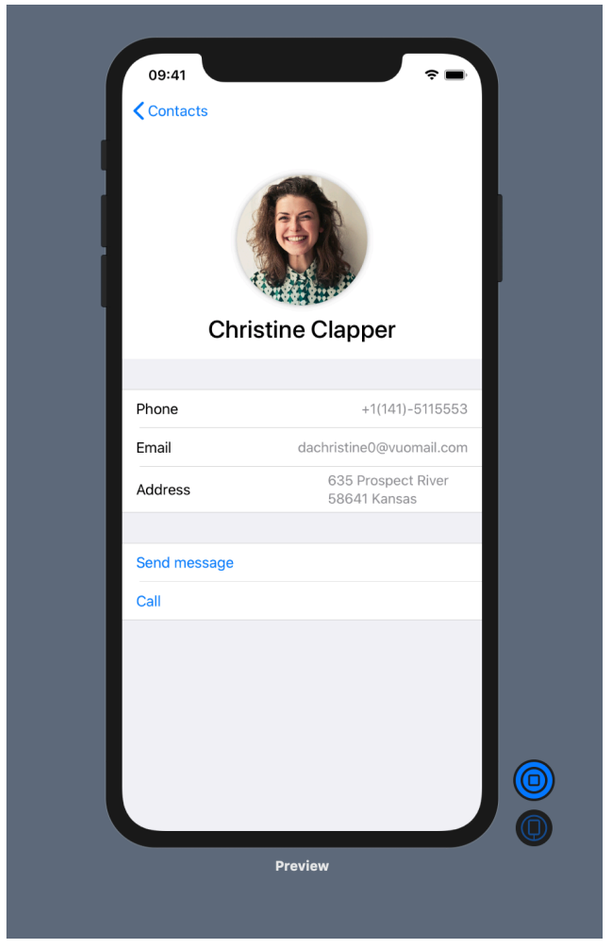
SwiftUI appends a large navigation bar by default. If you want to use the smaller one, you can use the displayMode argument with choosing the *.inline* option. But for now, we are fine with the default navigation bar style.



To navigate to the *DetailView* when tapping on a row, we wrap each *ContactRow* instance that gets created inside the List into a NavigationLink with choosing that instance as the link's destination.

```
List(contacts) { contact in
    NavigationLink(destination: DetailView(contact:
        contact)) {
        ContactRow(contact: contact)
    }
}
```

The NavigationLink tells SwiftUI to push the *DetailView* equipped with the passed *Contact* instance on top of the navigation hierarchy. We can see how this looks like by running our app in the live preview.



Awesome, by using our navigation view we are able to stack our *ContentView* and our *DetailView(s)* in a navigation hierarchy and to let the user navigate between them.

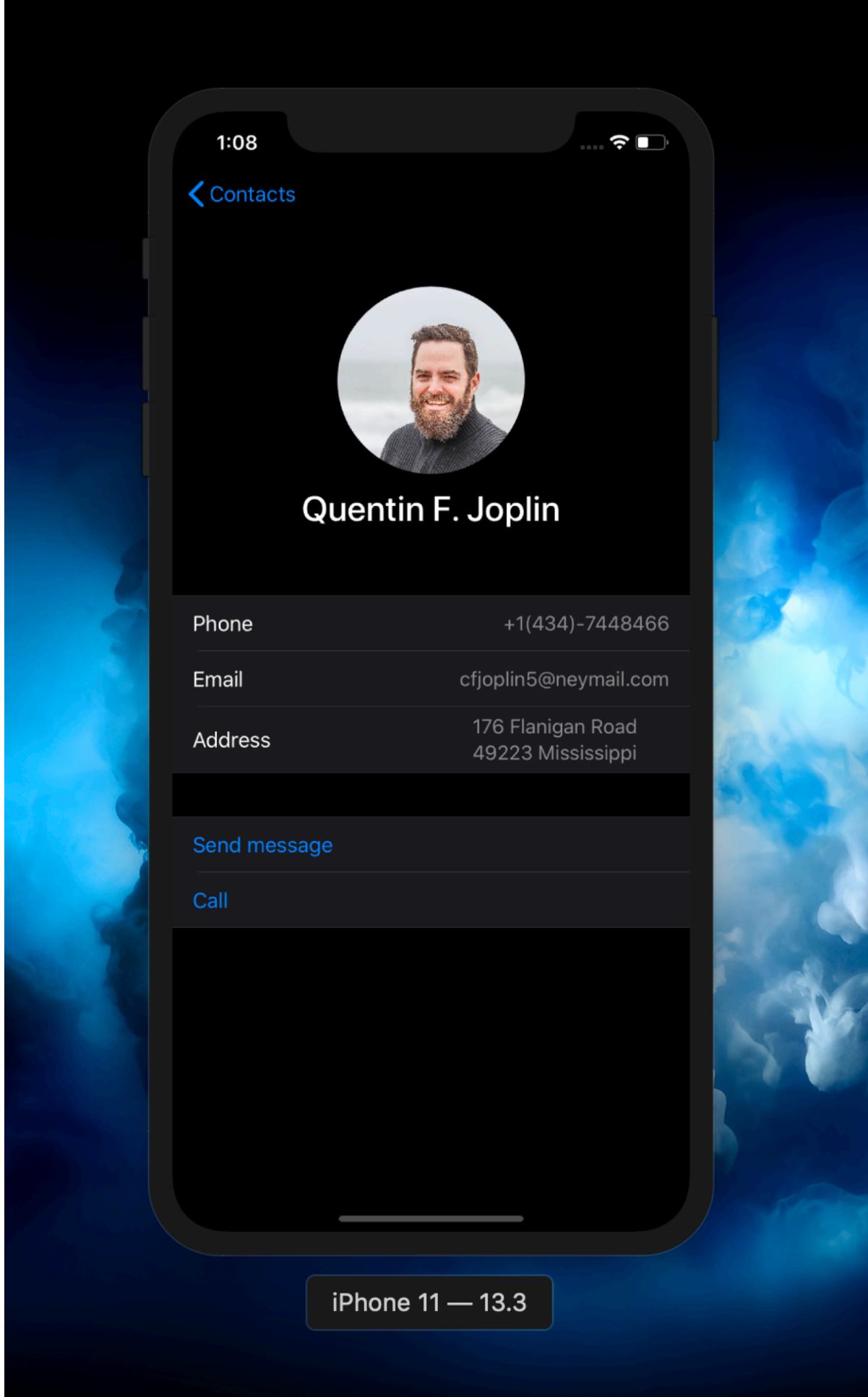
CONCLUSION

That's it! We are finished with creating our own Contacts app. We learned how to present data inside Lists and how to cluster UI pieces inside forms. We also learned how to navigate between several views by using a navigation view hierarchy

You can find the whole source code here.

Maybe you're asking yourself how you can navigate between views independently by not using a navigation view stack. Then take a look at this tutorial!

Or you check out our Mastering SwiftUI book where we cover this topic as well. This eBook contains 14 chapters full of information about developing apps with SwiftUI. By going from basic concepts of SwiftUI, over more advanced topics like creating complex user interfaces to eventually communicating with a backend server you will learn everything you need to know about working with SwiftUI step by step!



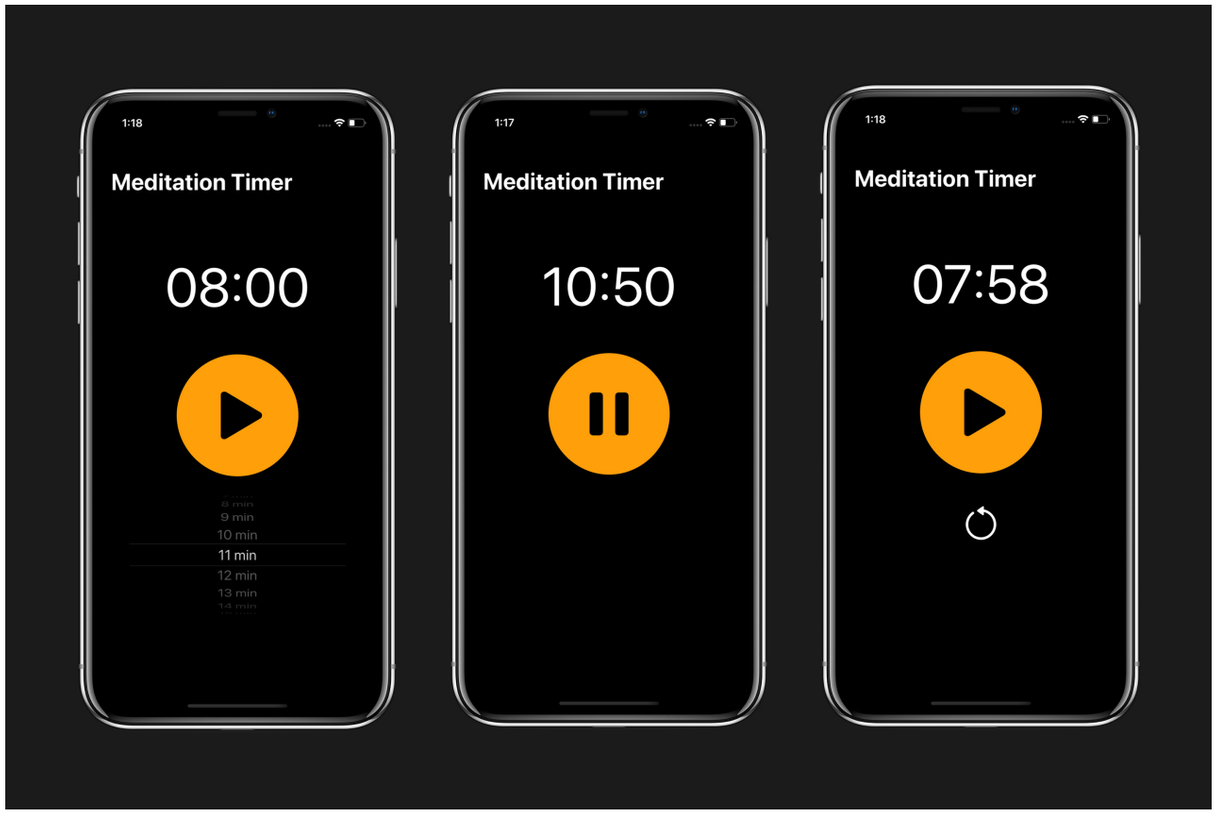
Chapter 6

TIMER APP

- How to create more dynamic UI's
- Using system icons in SwiftUI
- Understanding @ObservableObjects and how to use them as your view models
- Storing user settings in SwiftUI

What we'll achieve 🚀

In this chapter, we're going to create a timer app. By doing this we'll practice creating more complex and dynamics UI's and how to use @ObservableObjects as view models. The timer app will look like this:



Let's get started by creating a new Xcode 11 project, choosing Single View App and calling it "TimerApp".

As you saw in the preview above, the app's UI should depend on whether the timer has not been started, is running or is paused. We'll often refer to these three different scenarios so it's useful to start by creating a suitable enum for this. To do this, create a new Swift file and call it "Helper". Inside this file insert the following enum:

```
enum TimerMode {
    case running
    case paused
    case initial
}
```

For keeping track of the current *TimerMode*, we declare a corresponding State property inside our *ContentView* to adapt the UI, whenever the timer gets started/paused/stopped.

```
struct ContentView: View {

    @State var timerMode: TimerMode = .initial

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

Now we're ready to start composing our UI depending on the current *timerMode*.

Composing the Timer's UI 🎨

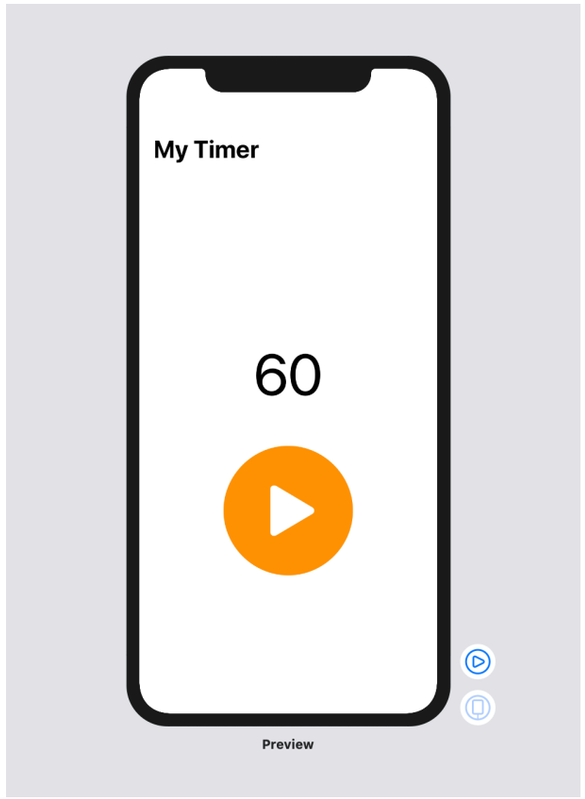
We start by wrapping the "Hello World" Text view into a NavigationView. Then, we wrap the Text into a VStack and append a navigation bar title to it.

```
NavigationView {
    VStack {
        Text("Hello, World!")
    }
        .navigationBarTitle("My Timer")
}
```

Instead of "Hello World" the text view should display the remaining time. For now, we use the temporary and static value 60 for representing that 60 seconds are left before the timer runs off. We highlight the Text view by increasing its size and append some top padding to it.

```
Text("60")
    .font(.system(size: 80))
    .padding(.top, 80)
```

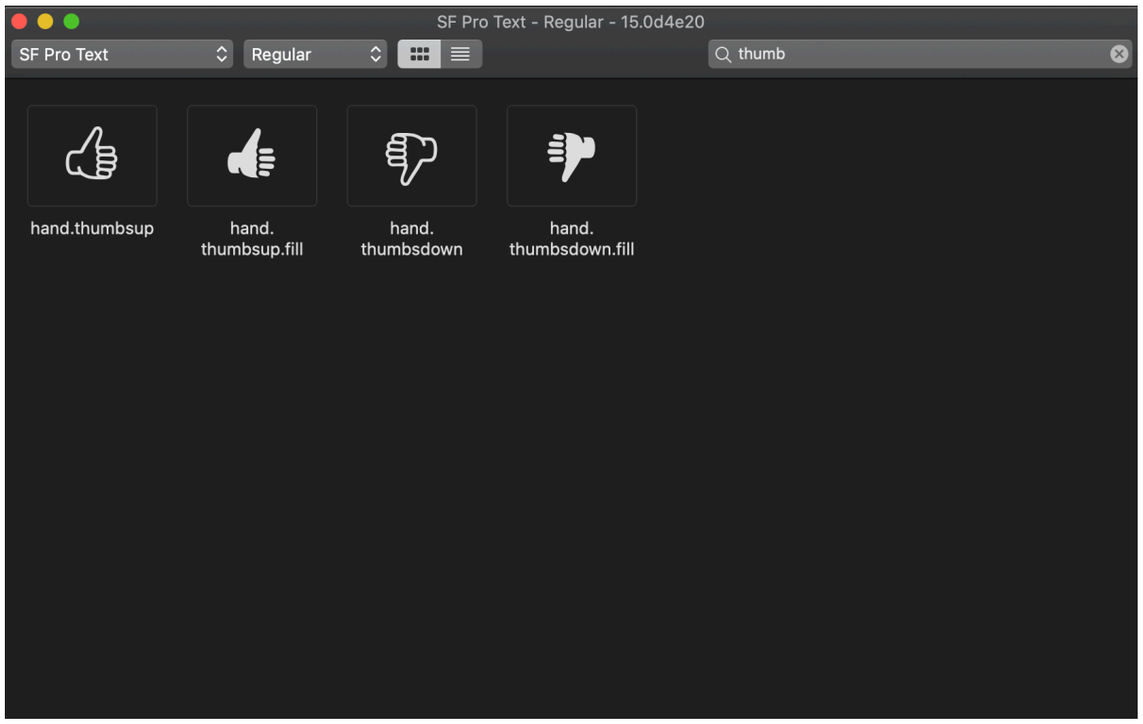
Below our Text view, we want to place a large play icon to start the timer. At this point, we could use Image views with initializing custom icons we had to import into the Assets folder before, but fortunately, there is a much easier way: Using system symbols!



Apple provides us with a bunch of integrated icons we can use for creating our UI. We can initialize such an icon by using an Image view with the system argument. Therefore we write below our Text view:

```
Image(systemName: "play.circle.fill")
    .resizable()
    .aspectRatio(contentMode: .fit)
    .frame(width: 180, height: 180)
    .foregroundColor(.orange)
```

Working with SF Symbols 👨‍💻



At this point, you are probably asking yourself: How I am supposed to know what system icons are available and how to find their names so I can use them within my app?

Well, that's super easy! Just use the free Apple SF symbols app. This macOS app provides you with a set of over 1,500 symbols you can use in your app.

You can download the app here.

Once you open SF symbols, you can scroll through the symbols or you search for a certain term to find suitable symbols for your purpose. The app then shows four different icons we can use by entering their names as the Image's systemName argument!

However, the image view should only display to play icon when the timer is in its initial state. While the timer is running, we want to present a pause icon. Therefore, we adapt the systemName parameter like this:

```
Image(systemName: timerMode == .running ?
   "pause.circle.fill" : “play.circle.fill")
```

Now, while the timer is running, the image views displays the pause icon of SF symbols. Otherwise, it displays the play icon.

When the timer is paused, we want to display a small rewind icon so that the user can reset the timer. To do this, we insert the following below our current Image view:

```
if timerMode == .paused {
    Image(systemName: "gobackward")
        .resizable()
        .aspectRatio(contentMode: .fit)
        .frame(width: 50, height: 50)
        .padding(.top, 40)
}
```

While the timer is in its initial State we want to provide the user with a possibility to set the desired timer length. For this purpose, we’re going to use a Picker.

Accepting user input with pickers

A picker is a piece of UI that includes one or multiple lists displayed as a wheel. The user can use this wheel(s) for selecting a specific value.

Pickers are often found within iOS apps. Take a look at the date selection in the system’s Calendar app or the time selection in the Clock app.

To use a picker inside for our timer app, we need to create a State property for keeping track of the current selection first.

```
@State var selectedPickerIndex = 0
```

Next, we create the data set for our Picker, in our case, it’s an array holding the values 1 to 59 (minutes). Insert this variable below your States.

```
let availableMinutes = Array(1…59)
```

Now we can create a Picker with binding it to the State We want no label, so just insert a Text with an empty String and apply the *.labelIsHidden* modifier to it.

```
if timerMode == .initial {
        Picker(selection: $selectedPickerIndex,
            label: Text("")) {

    }
        .labelsHidden()
}
```

Inside the Picker, we implement a ForEach loop that cycles through every object of our *availableMinutes*. We use the loop's closure for creating one Text object for every minute and assign it to an index.

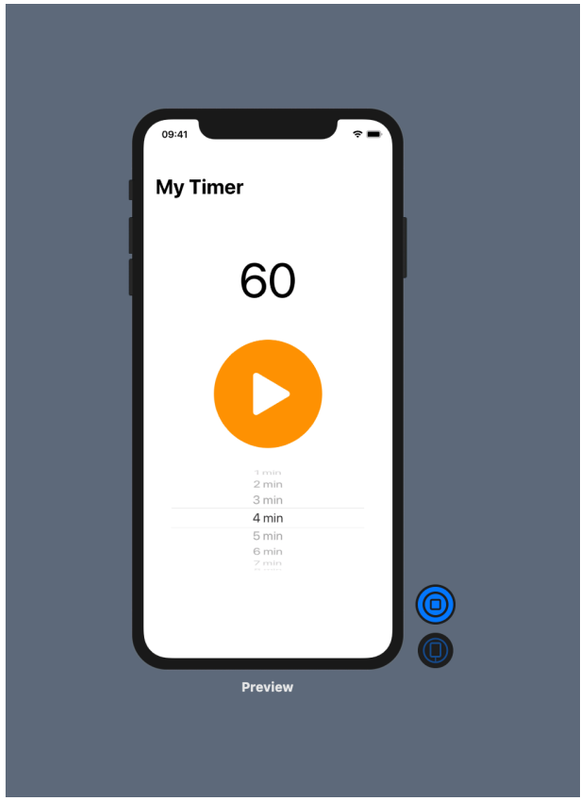
```
Picker(selection: $selectedPickerIndex, label:
    Text("")) {
    ForEach(0 ..< availableMinutes.count) {
        Text("\(self.availableMinutes[$0]) min")
    }
}
```

SwiftUI then knows which index every object should have and updates the State depending on the current selection!

Finally, we push everything to the top by using a Spacer as the last view inside the VStack.

```
VStack {
    //...
    Spacer()
}
```

If you start a live preview, you can scroll through the picker. We'll use this functionality later on the set the timer length.



You can check if your UI acts correctly by changing the timerMode State and taking a look at the preview canvas. When you change the timerMode to *.running*, the picker should disappear and the play icon should be replaced with a pause button. When you change it to *.paused* the rewind icon should show up.

Great, we're done with implementing the UI for our timer app. Let's get started with actually implementing the timer functionality!

Working with @ObservableObjects

You already learned what @State and @Bindings are and what they are used for. For implementing our timer functionality we'll use a so-called @ObservableObject.

ObservableObjects are similar to State properties but with some differences.

- OberservableObjects are used for classes not variables
- These classes can contain data, for example, a string assigned to a variable
- We can bind one or multiple views to the ObservableObject (or better said, we can make these views observe the object).
- The observing views can access and manipulate the data inside the ObservableObject. When a change happens to the ObservableObject's data all observing views get automatically rerendered, similar to when a State changes

Don't worry if you don't understand all of this. You'll understand it by reading on.

Let's create such an ObservableObject by creating a new Swift file and calling it TimerManager. Make sure you import the SwiftUI framework. Next, insert a class called TimerManager as well that adopts the *ObservableObject* protocol:

```
import Foundation
import SwiftUI

class TimerManager: ObservableObject {

}
```

Our *TimerManager* should keep track of whether the timer is running, paused or in its initial state. Therefore declare a variable called *timerMode*.

```
class TimerManager: ObservableObject {

    var timerMode: TimerMode = .initial

}
```

Back to our ContentView: To make it observe the *TimerManager*, we have to initialise an ObservedObject inside it.

```
@ObservedObject var timerManager = TimerManager()
```

We want our *ContentView*'s body to refer to the *timerManager*'s *timerMode* property instead of the *ContentView*'s State.

```
VStack {
    //...
    Image(systemName: timerManager.timerMode
            == .running ? "pause.circle.fill" :
            "play.circle.fill")
        //...
    if timerManager.timerMode == .paused {
        //...
    }
    if timerManager.timerMode == .initial {
        //...
        }
    Spacer()
}
```

Because we don't need it anymore, we can delete the *ContentView's timerMode* State property

Next, our *TimerManager* needs another property for keeping tracks of the remaining seconds.

```
var secondsLeft = 60
```

We want the Text view in our *ContentView* to refer to that value instead of using the static one, there we update it like this:

```
Text("\(timerManager.secondsLeft)")
```

Implementing a Swift timer ⏲️

Implementing a timer itself is really easy in Swift. We just need to add a *Timer* instance to our *TimerManager*.

```
var timer = Timer()
```

Next, we add a function called start to our class that actually triggers the timer. We'll execute this function when the user taps on the play icon. When the function gets called we want to change the timerMode to *.running*.

```
func start() {
    timerMode = .running
}
```

Then we tell our timer to decrease the *secondsLeft* every second until it hits zero.

```
timer = Timer.scheduledTimer(withTimeInterval:
    1.0, repeats: true, block: { timer in
        if self.secondsLeft == 0 {
            self.timerMode = .initial
            self.secondsLeft = 60
            timer.invalidate()
        }
        self.econdsLeft -= 1
    })
```

As said, we want this function to be called when we tap on the play icon. We could do this by wrapping the corresponding Image view into a button, but there's an alternative.

We can achieve the same functionality by simply adding a tap gesture to our Image view that performs that detects when the user touches the Image view.

```
Image(systemName: timerManager.timerMode == .running ?
    "pause.circle.fill" : "play.circle.fill")
    //...
    .onTapGesture(perform: {
        self.timerManager.start()
    })
```

Let's check if our timer works by running our app in live mode and tapping on the play icon. Odd, the time is not running down. Why that?

How to update observing views 🔄

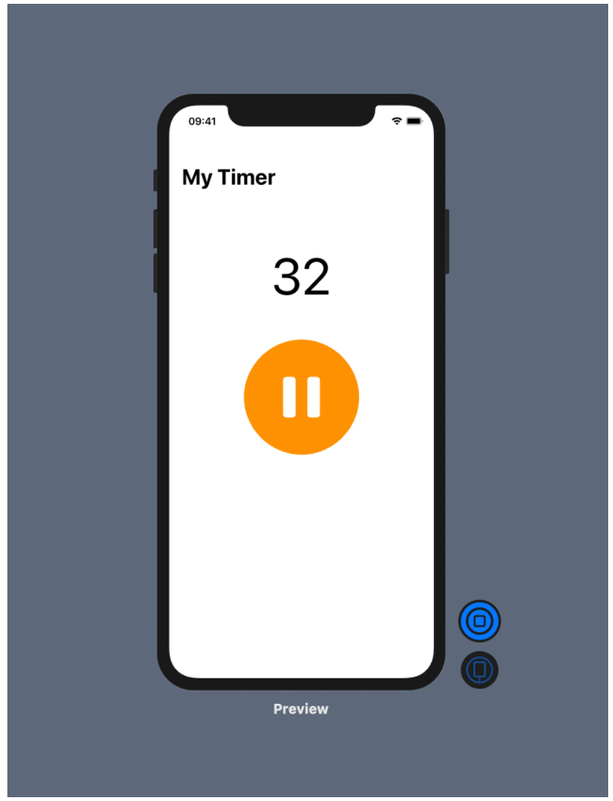
Until now, changes in the *secondsLeft* property don't cause the observing *ContentView* to refresh. To tell SwiftUI that all observing view should get updated whenever the *secondsLeft* is decreasing we have to use the @Published property wrapper.

```
@Published var secondsLeft = 60
```

Every time the *secondsLeft* gets updated now, the *ContentView* will be caused to rebuild itself with eventually showing us the remaining time after tapping on the play icon. At the same time, let's use the @Published property for the *timerMode* as well, so that our *ContentView* gets back to its initial look when the *secondsLeft* are hitting zero.

```
@Published var timerMode: TimerMode = .initial
```

If we run our app now and tap on the play icon our timer starts decreasing every second. When it hits zero, the UI gets back to its default look.



Pausing and resetting the timer ⏸⏮

We already have a function for starting the timer but we still need one for pausing it and resetting it. Therefore, add these two functions to your *TimerManager* class:

```
func reset() {
    self.timerMode = .initial
    self.secondsLeft = 60
    timer.invalidate()
}

func pause() {
    self.timerMode = .paused
    timer.invalidate()
}
```

You maybe noticed that the code inside the *reset* function is the same that we execute in our *start* function when the *secondsLeft* hit zero. To refactor our code, we simply call the *reset* function from our *start* function.

```
if self.secondsLeft == 0 {
    self.reset()
}
```

In our *ContentView*, we want to call the *pause* function when the user taps on the pause icon. To achieve this we update our *.onTapGesture* modifier as follows:

```
.onTapGesture(perform: {
    self.timerManager.timerMode == .running ?
    self.timerManager.pause() :
    self.timerManager.start()
    })
```

We call the *reset* function by appending the corresponding tap gesture to the Image view that shows to rewind symbol.

```
Image(systemName: "gobackward")
    //...
    .onTapGesture(perform: {
            self.timerManager.reset()
    })
```

Great, if we run our app now, we're able to pause and reset our timer!

Setting the timer length by using user defaults

We're already able to start, pause and stop our timer. However, we can't set the timer length by using the picker yet.

To store the chosen time persistently, we're going the use the *UserDefaults*. *UserDefaults* can be used to store small chunks of data, such as the timer length setting.

For this purpose, we add a function called setTimerLength to our *TimerManager*. The function should intake the desired time length as an Integer:

```
func setTimerLength(minutes: Int) {

}
```

We can access the *UserDefaults* like this:

```
func setTimerLength(minutes: Int) {
    let defaults = UserDefaults.standard
}
```

The structure of the *UserDefaults* is very similar to a dictionary. We can store pieces of data under a certain key and retrieve it again by using the specified key. We call the *setTimerLength* function we want to store the given minutes under a key we call “timerLength”. Finally, we set the published *secondsLeft* property to this value when the function gets called.

```
func setTimerLength(minutes: Int) {
    let defaults = UserDefaults.standard
    defaults.set(minutes, forKey: "timerLength")
    secondsLeft = minutes
}
```

The *secondsLeft* property is still 60 when the *TimerManager* gets initialised. Instead, we want to retrieve the stored value from the specified “timerLength” key:

```
@Published var secondsLeft =
    UserDefaults.standard.integer(forKey:
    “timerLength”)
```

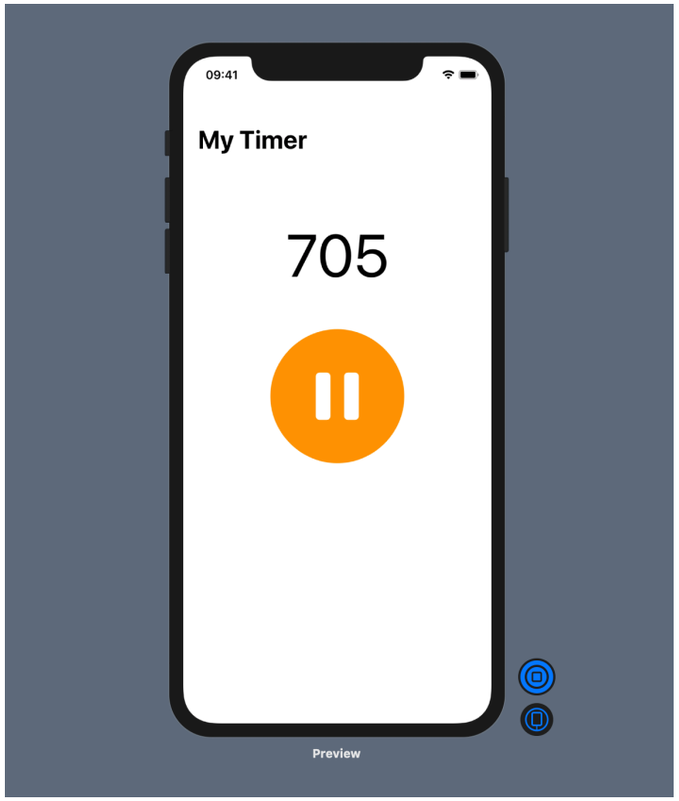
We want to retrieve to timer length from the *UserDefaults* inside our reset function as well:

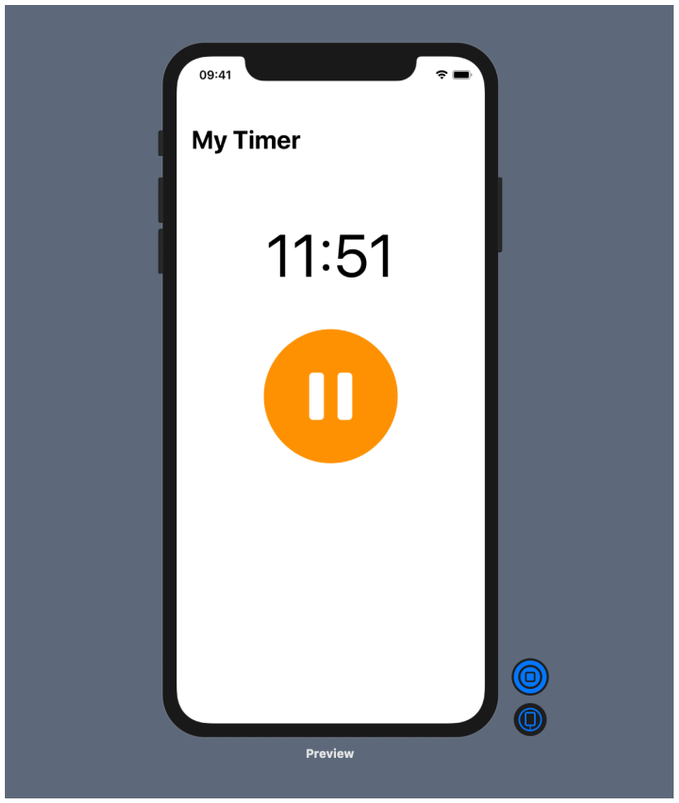
```
func reset() {
    //...
    self.secondsLeft =
          UserDefaults.standard.integer(forKey:
          "timerLength")
    //...
}
```

We want to call to set the timer length whenever the user is starting the timer. Therefore, we call the *setTimerLength* function from the *ContentView*’s Play/Pause Image view but only when the *timerMode* is *.initial*. For the minutes parameter, we use the selected minutes from our picker.

```
.onTapGesture(perform: {
    if self.timerManager.timerMode == .initial {

        self.timerManager.setTimerLength(minutes:
        self.availableMinutes[self.selectedPickerIndex]*60)
    }
    //...
})
```





Converting seconds to a time stamp

If we run our app now, we're able to set the desired timer length by using the picker. However, when selecting for instance 12 minutes our app shows us the plain seconds left. Instead, we want to display a time stamp for better readability. To achieve this, just add this function to your *Helper.swift* file.

```
func secondsToMinutesAndSeconds (seconds : Int) ->
    String {

    let minutes = "\((seconds % 3600) / 60)"
    let seconds = "\((seconds % 3600) % 60)"
    let minuteStamp = minutes.count > 1 ? minutes :
        "0" + minutes
    let secondStamp = seconds.count > 1 ? seconds :
    "0" + seconds

    return "\(minuteStamp):\(secondStamp)"
}
```

In our *ContentView*, we can now use this function to convert the *secondsLeft* Integer into a corresponding time stamp String.

```
Text(secondsToMinutesAndSeconds(seconds:
    timerManager.secondsLeft))
        //...
```

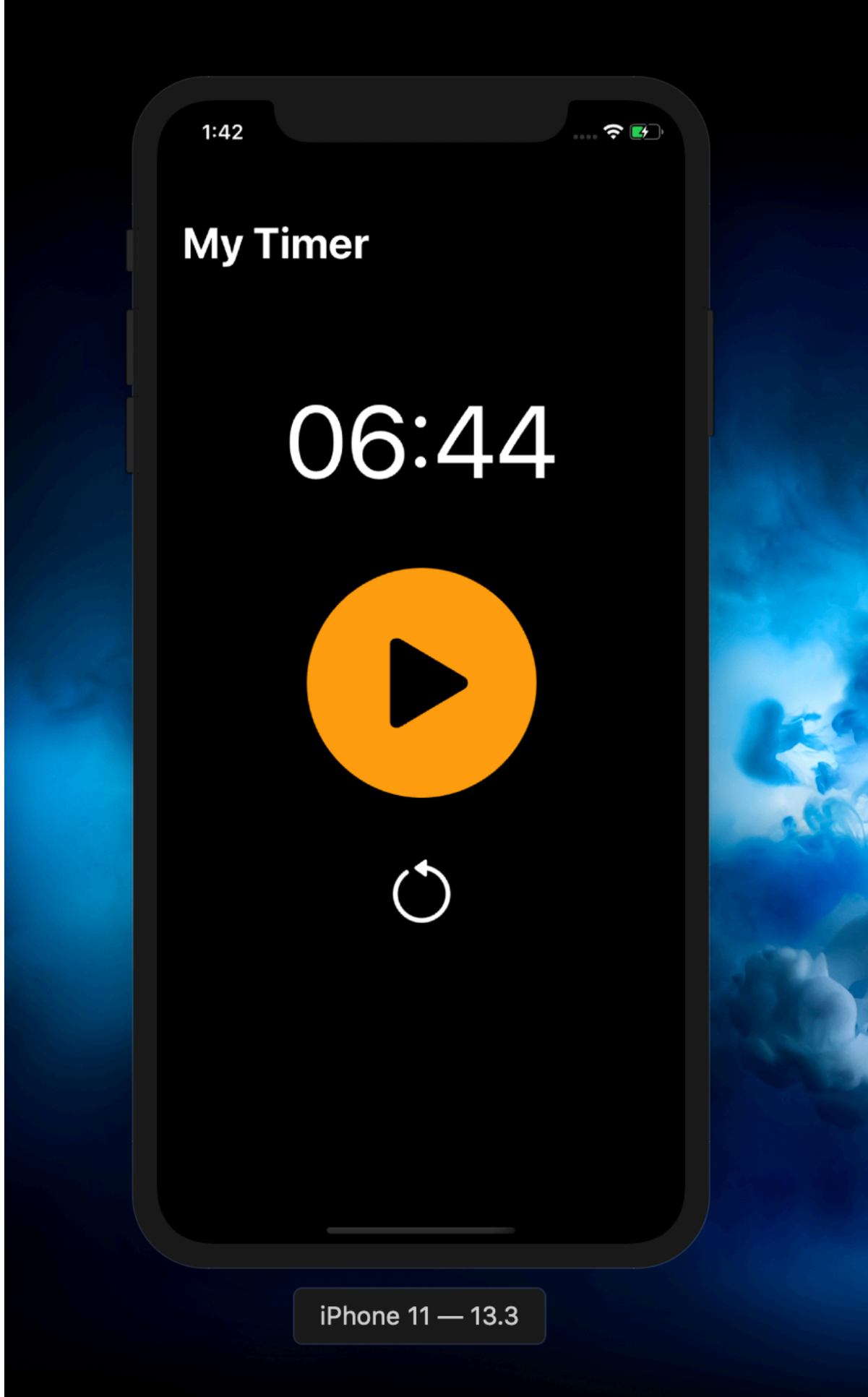
If we start our timer now, we see that the *secondsLeft* get converted into a timestamp string!

CONCLUSION

That's it! We are finished with creating our timer app. We learned how to create more complex and dynamic interfaces with SwiftUI. We also explored @ObservableObjects and how we can use them as our view's model.

Additionally, we learned how we can use Swift timers and how to store small chunks of data persistently by using the UserDefaults!

You can find the whole source code of the app here.



Chapter 7

WHERE TO GO FROM HERE

Congratulations, you just finished this book!

That's it for now, thank you for reading this book! You just got familiar with the basics of SwiftUI. You learned how to compose interfaces step-by-step by using SwiftUI views. You learned how data flow works in SwiftUI and know what @State, @Bindings and @ObservableObjects are. You also explored how to present data by using Lists and how to navigate between screens in a view hierarchy.

If you want to learn more about SwiftUI, visit BLCKBIRDS for many free SwiftUI tutorials!

Want to dive deeper into SwiftUI? Then check out our Mastering SwiftUI book. In over 14 chapters, we're creating multiple useful apps (e.g. a To-do app and a chat messenger) and learn a lot more about SwiftUI. With lifetime-updates and full code support, we teach you how to develop your own iOS apps by using SwiftUI!

If you have any wishes or suggestions on what you would like to learn, make sure you send us a mail or leave a DM on Instagram.

For inspiration, more content about iOS development and tips for iOS developers, follow us on Instagram and Twitter.