# Beginning SwiftUI

Updated to SwiftUI 2.0 and iOS 14

Greg Lim

# Beginning SwiftUI

Updated to SwiftUI 2.0 and iOS 14

Greg Lim

# Beginning SwiftUI

Greg Lim

First Edition: March 2021

## Table of Contents

# Preface

## About this book

In this book, we take you on a fun, hands-on and pragmatic journey to learning iOS 14 development using SwiftUI. You'll start building your first SwiftUI app within minutes. Every section is written in a bite-sized manner and straight to the point as I don’t want to waste your time (and most certainly mine) on the content you don't need. In the end, you will have the skills to create an app and submit it to the app store.

In the course of this book, we will cover:

- Chapter 1: Introduction
- Chapter 2: Body Mass Index Calculator
- Chapter 3: To Do List App Using *List*
- Chapter 4: Persistent Data Using Core Data
- Chapter 5: Extending Core Data to CloudKit
- Chapter 6: Getting Data from an API: GitHub Users
- Chapter 7: Machine Learning with Core ML
- Chapter 8: C.R.U.D. Notes App with Firebase/Firestore
- Chapter 9: Building Cross Platform Apps in SwiftUI

The goal of this book is to teach you SwiftUI development in a manageable way without overwhelming you. We focus only on the essentials and cover the material in a hands-on practice manner for you to code along.

## Requirements

No previous knowledge of iOS development required, but you should have basic programming knowledge.

## Getting Book Updates

To receive updated versions of the book, subscribe to our mailing list

by sending a mail to support@i-ducate.com . I try to update my books to use the latest version of software, libraries and will update the codes/content in this book. So do subscribe to my list to receive updated copies!

## Contact and Code Examples

Visit *www.greglim.net/swiftui* to obtain the source codes used in this book. Comments or questions concerning this book can also be directed to the same.

# Chapter 1: Introduction

Welcome to Beginning SwiftUI iOS 14 Development! I'm Greg and I'm so excited that you decided to come along for this. With this book, you will go from absolute beginner to having your app submitted to the App Store and along the way, equip yourself with valuable iOS app development skills.

SwiftUI makes it easier to build user interfaces across all Apple platforms with Swift. It uses a declarative syntax where you simply state what your user interface should do.

Before SwiftUI, developers used UIKit and Storyboard to design user interfaces. We drag and drop UI controls on to View Controllers and connect them to outlets and actions on the View Controller classes. We then update view controls and handle events that occur through delegates. If you would like to learn iOS development using UIKit and Storyboard, check out my best-selling book or contact me at *support@i-ducate.com* .

SwiftUI in contrast is a state-driven, declarative framework. There is no more dragging and dropping in the storyboard. Layouts are specified declaratively using code. For example, you can state that you want a list of items consisting of text views and then describe the value for each view.

```
struct ContentView: View {
    var body: some View {
        List{
            Text("Write SwiftUI book")
            Text("Read Bible")
            Text("Bring kids out to play")
            Text("Fetch wife")
            Text("Call mum")
        }
    }
}
```

Your code is easier to read and write than before, saving you time and maintenance.

## Working Through This Book

This book is purposely broken down into brief chapters where the development process of each chapter will center it on different essential topics to develop apps for the iPhone. The book takes a practical hands-on approach to learning through practice. You learn best when you code along with the examples in the book. Along the way, if you encounter any problems, drop me a mail at support@i-ducate.com where I will try to answer your query.

## Get a Mac

Before we proceed on, you will need to have a Mac running on at least macOS version of 10.15.4 (Catalina) to run Xcode 12. This is because to use Live Preview and design canvas features in Xcode, you need at least macOS Catalina (10.15).

If you do not yet have a Mac, the cheapest option is to get a Mac Mini and if you have a higher budget, get a higher model or iMac with more processing power. You might have heard of the option to run Mac on Windows machines for iOS development, but I do not recommend it. Unexpected problems will arise in development and publishing to the App store that can be avoided by just using a Mac. If you are serious about developing iOS apps and publishing them on the App Store, getting a Mac is a worthwhile investment.

## Downloading Xcode

Next, there is an essential piece of software you need to have on your computer before we can move forward. It's called Xcode and is an integrated development environment (IDE) provided by Apple to write Swift code and make iOS apps. It includes the code editor, graphical user interface editor, debugging tools, an iPhone/iPad simulator (to test our apps without real devices) and much more. Let ' s get it downloaded before proceeding.

Download the latest version of Xcode 12 (at time of writing) from the Mac App Store (fig. 1).

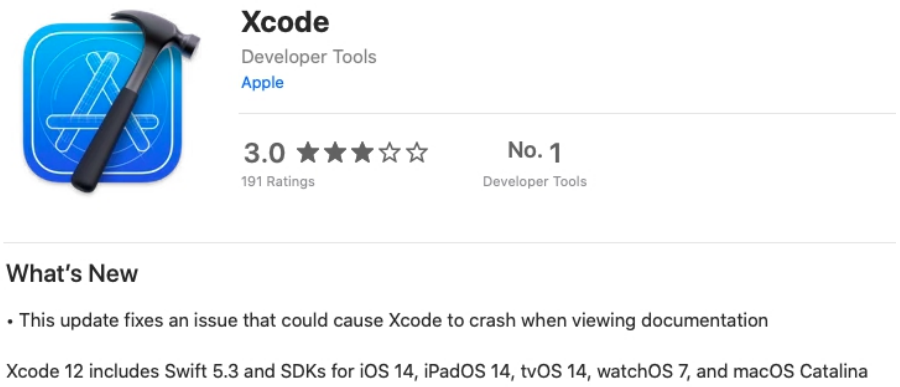



Figure 1

You will need an Apple ID to login and download apps from the Mac App store. If you do not already have one, sign up for an account (https://appleid.apple.com/account). You will also need an Apple ID to be able to deploy your app to a real iPhone/iPad device for testing.
The installation of Xcode might require you to update your version of MacOS. At this book ' s time of writing, the MacOS required is Catalina version 10.15.4.

## *Installing Xcode*

Just like any other Mac App, Mac App store will take care of the downloading and installation of Xcode for you. Do note that installation of Xcode 12 requires 20-30 GB of space available for the installation to proceed and installation takes quite some time. Once the installation is complete, you should see the Xcode icon on your computer.

## Swift and Xcode

I'm going to be introducing you to two terms that you're going to encounter throughout this book. One of those is Swift and the other one is Xcode. Swift is the programming language we use to make iPhone apps. Swift came out in 2014. Previous to that, the programming language used to make iPhone apps was Objective C. But Objective C was complicated. Many developers new to the space of iOS development found that it was hard to read and write. Swift then was introduced. Swift is specifically designed with

beginners in mind and even experienced programmers think of Swift as a really clean and beautiful language.

Xcode is the program that allows us to make iPhone apps. We're going to type Swift into Xcode and also use Xcode for designing the visual side of our app like where we want a button, what color do we want it to be, where do we want to place our table view, etc. Throughout this book, these are the two skills that we will be improving upon step by step.

This book is written for a beginner in iOS development. So, you don ' t need to have any prior iOS development experience. But if you have some iOS development experience, you're going to feel pretty familiar with what's going on.

It will also be best if you have some basic programming experience. But if you do not have it, it ' s alright and I will try my best to explain certain programming concepts.

## Xcode Walkthrough

Now in this section, I want you to become acquainted with Xcode. Open Xcode.

At the time of writing, this book uses Xcode 12. But make sure you're using the latest official version of Xcode from the Mac App Store.

In the ' Welcome to Xcode ' screen (fig. 2), you can choose to either get started with a playground which is a great way to explore the Swift language. The next option is creating a new Xcode project where you create an app for iPhone, iPad, Mac, Apple Watch or Apple TV.


Figure 2

You also have a third option to clone an existing project but we will not be covering this option in this book.

For now, let's create a new project. When you do so, it's going to bring up a page (fig. 3) that asks what kind of project do you want to make, whether iOS, watchOS, tvOS, macOS or Multiplatform.

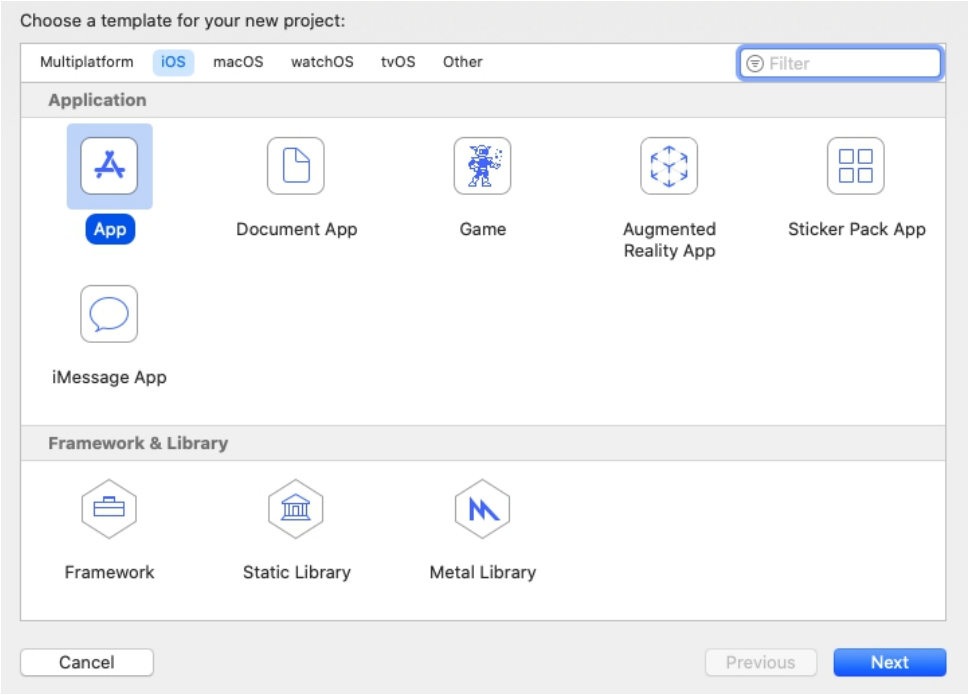



Figure 3

*iOS* includes apps for the iPhone, iPad, iPod Touch. *watchOS* is for

Apple watch apps. *tvOS* is for Apple TV apps. *macOS* is for Mac apps on the desktop and *Multiplatform* is if you want to make an app that works across multiple platforms. For us, we focus on *iOS* apps.

For an iOS app, there are lots of different templates that you can start with. The templates help you get started with some boilerplate code. For us, we want the *App* option. This is essentially the blank starting point for almost every app that we're going to make. So let's go ahead and double click on that.

You will then have to input the below fields for your project (fig. 4):

*Product Name* : (as this is our first project, we will name it *HelloSwiftUI* )
*Team* :
*Organization Identifier* : (normally the reverse of your website e.g. com.iducate.calculator. If you do not have a website, com.firstname.lastname will do fine)
*Interface:* select *SwiftUI*
*Life Cycle:* select SwiftUI App
*Language* : select *Swift* (only option available)

For *Use Core Data, Include Tests* : leave all the boxes unchecked

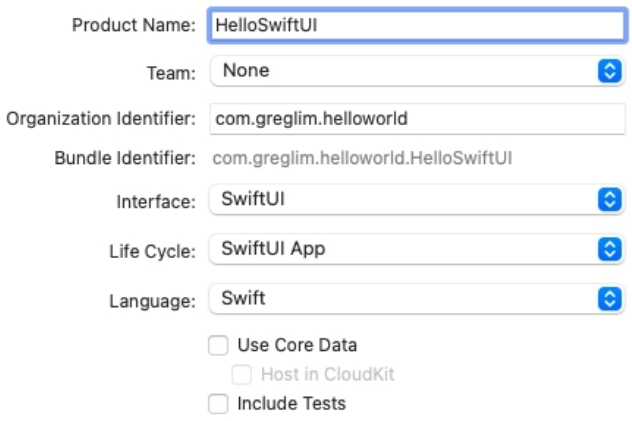



Figure 4

Go ahead and fill in the fields. You can change the field values later in your project, so don ’ t worry if you have inputted a wrong value.

When you have the fields filled up, hit the *Next* button. It's going to ask you where you want to save this new project. I'm going to put ours under ‘ Documents ’ .

There will also be a checkbox to ‘ Create Git repository on my Mac ’ . This will make a GitHub repository for your app which helps you save different versions of your app and if you want to collaborate with people. Git is outside the scope of this book but just go ahead and leave this checked.

You should see the project created for you (fig. 5a).

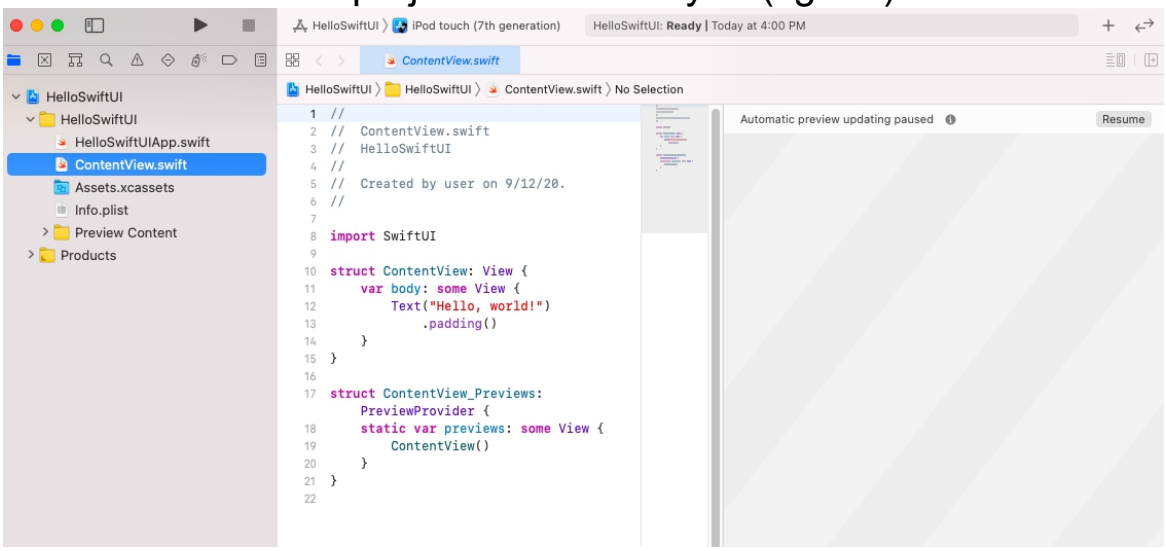



Figure 5a

You can see that a new folder has been added called *HelloSwiftUI* . On the left side of Xcode, you can see the folder-file structure of the project (fig. 5b).

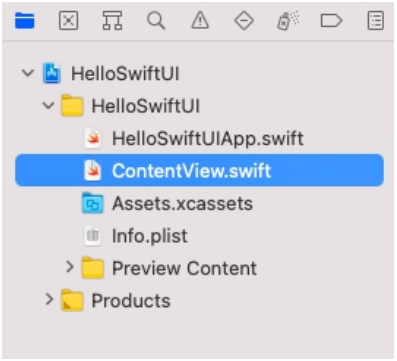



Figure 5b

We have our *HelloSwiftUI* folder at the root and another *HelloSwiftUI*

folder with more files inside of that. Xcode will take the name of your project and put that in a folder which consists of another folder with that same name. It will also have a project file *HelloSwiftUI.xcodeproj* which is the project file to open our project. If we close out Xcode and wanted to open up your project to work on it again, double click on *HelloSwiftUI.xcodeproj* and it will open up Xcode with all your project folders and files.

And then we have folders like *Assets.xcassets* to store resources that our app uses. For example, images, sounds, fonts, and videos. The apps in this book will have *Assets.xcassets* mainly storing images.

*Info.plist* (Information Property List) contains metadata settings for your app e.g. app name, version, and other fields that we have entered in our initial project setup form.

Lastly, if you click back on the root project file, you can see the General settings for your app (fig. 6).

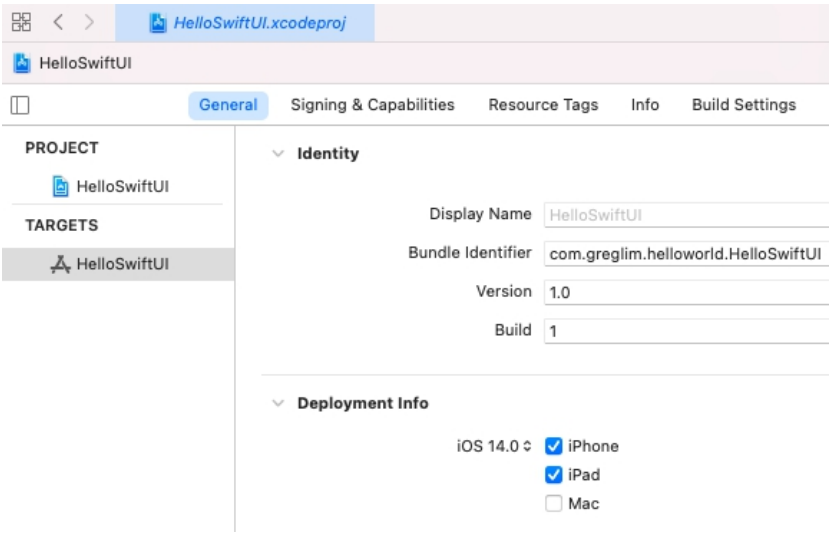



Figure 6

This is where previously you had filled in the fields at the beginning of the project. You can come back here and change it.

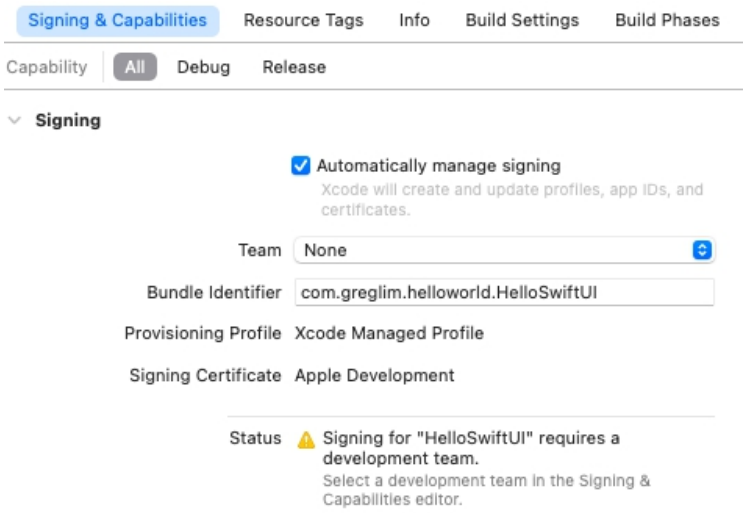



Figure 7

Under the ‘ Signing & Capabilities ’ tab (fig. 7) is also where we had to specify the ‘ Team ’ field. Again, you can come back here and change it. For example, we have not specified *Team* . Here, you can go back and change when you want to submit your app to the App Store.

The main file we will work with for now is *ContentView.swift. ContentView.swift* contains the UI for our application's initial main screen.

There is a canvas on the right side of Xcode which lets you preview the UI of your application without needing to run the application on the Simulator or a real device.

Click 'Resume' to preview your UI (fig. 8a, fig. 8b).

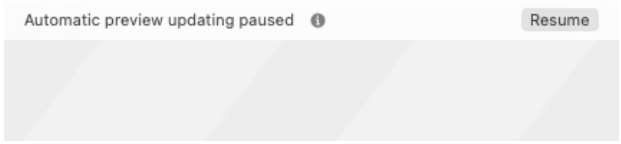



Figure 8

*if you don ’ t see ‘ Resume ’ , make sure you are running at least macOS Catalina.

Figure 8b

*ContentView.swift* contains the following:

```
import SwiftUI

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

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

## *Code Explanation*

```
struct ContentView: View {
```

The *ContentView* struct defines the UI of your screen. It conforms to the *View* protocol (everything you want to show in SwiftUI needs to conform to *View* ). Thus, it must declare a property called *body* which returns some sort of View.

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

```
    }
```

*body* currently consists of a *Text* View. A *Text* View is a graphical view that displays one or more lines of read-only text. In other words, it is a text label.
Note: the *some* keyword in the above code literally means “ this will return some sort of View but SwiftUI doesn ’ t need to care what. ”

Change the text in the *Text* View:

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

The preview should automatically update to reflect the text change (fig. 9).

Figure 9

Sometimes, the automatic preview will pause when we make some changes. What that happens, click 'Resume' and you should see the preview again. Occasionally, you might face an error where the preview doesn't work. You might have to click 'Try Again' to rebuild the preview.

Now, what produces the preview? The next section explains it.

## *ContentView_Previews*

The next struct *ContentView_Previews* conforms to the *PreviewProvider* protocol which produces the view previews.

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

For example, if we want to preview how our UI looks on an iPhone SE, we do the following:

```
struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView(). previewDevice("iPhone SE (2nd generation)")
    }
}
```

The preview canvas now displays our UI on an iPhone SE (2nd generation). With this, you can test to see how your app will look different on different devices, e.g. iPhone 12, iPhone 12 Pro Max, iPad etc.

We cover more on *ContentView_Previews* later in this book. But for now, realize the declarative nature of SwiftUI ensures we get the same consistent UI layout across different devices. In the past, with storyboards, to ensure the app looks the same across devices, we had to manually center the label by adding constraints.

To run your app on the simulator, simply click the  icon from the panel on the top left of Xcode. The simulator should start running and soon display your app.

## Buttons

Other than Texts, buttons are another visual control that you see in almost all apps. Users click on buttons to call specific code to

perform some actions.

Fill in the following to create a button:

```
struct ContentView: View {
    var body: some View {
        Button(action:{

        }){
            Text("Press Me")
        }


    }
}
```

The *Button* View has a parameter *action* where you provide the code to be performed when the button is clicked.

Below the button code block, you also have the code block which describes the look of the button. In our case, our button simply contains a *Text* View. We will later see how to customize the look of the button with *modifiers* .

To create an action for our button, we simply put the code we want to run when the button is pressed.

Let ' s add a *print* statement in the action:

```
Button(action:{
    print("button tapped")
}){
    Text("Press Me")
}
```

Now when you run the app in the simulator and you click on the button, you should have " button tapped " printed in the console log. To show your console, go to ' View ' , ' Debug Area ' , ' Activate Console ' (fig. 10).

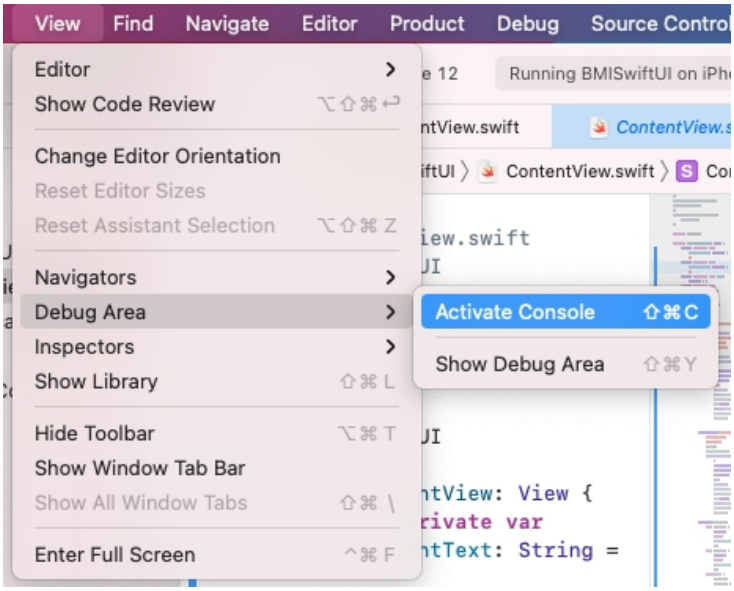


Figure 10

## Stacking the Views

So far, we have just displayed a single Text, or a single Button. But UIs often have multiple and different types of views together. How do we group and position them? This is where we use stacks. There are three kinds of stacks we can use to group our UI:

HStack - arranges its children views horizontally
VStack - arranges its children views vertically
ZStack - arranges its children views depth based (e.g. back to front)

So, if we want to stack a Text view above a Button view, I would have:

```
var body: some View {
    VStack{
        Text("Hello")
        Button(action:{
            print("button tapped")
        }){
            Text("Press Me")
        }
    }
}
```

And this would give me (fig. 11):


Figure 11

Note: by default, a Stack places little or no spacing between two views. But we can control the spacing by providing a parameter when we create the stack:

```
var body: some View {
    VStack(spacing: 20){
            …
    }
}
```

A VStack also by default aligns its views centered. But we can change that with the ‘ alignment ’ property. For e.g.

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

This aligns the views to the leading edge, i.e. aligned left.

## *HStack*

If I change VStack to HStack:

```
    HStack{
        Text("Hello")
        Button(action:{
            print("button tapped")
        }){
            Text("Press Me")
        }
    }
```

We get (fig. 12):

Figure 12

HStack has the same syntax as VStack including specifying spaces and alignment.

We can combine VStack, HStack views together to create more complex arrangements. For e.g.:

```
HStack {
    ZStack {
        Circle()
            .fill(Color.yellow )

        Button(action:{
            print("button tapped")
        }){
            Text("Press Me")
        }
    }
    .frame(width: 100.0, height: 100.0)

    VStack(alignment: .leading, spacing: 4) {
        Text("Beginning SwiftUI")
        Text("Greg Lim, 2021")
    }
}
```

This gives us (fig. 13):


Figure 13

## *Code Explanation*

We have a ZStack and a VStack nested in a HStack. The VStack itself contains two Text views. The ZStack contains a circle and a button overlaying the front of it.

We call the *frame* modifier of the ZStack to set its width and height to 100.

VStack(alignment: .leading, spacing: 4)

Remember that by default the views wrapped in a VStack are aligned in the center. We change this with the *alignment* parameter. In our case, we use the *leading* property to align to the left. Other values for alignment are *center* (default) and *trailing* .

Now, how do I know what other parameters or properties are available in SwiftUI? Do I have to memorize all of them? Fortunately, the answer is no. In fact, you can have the above generated for you in Xcode.

For example, to fill in yellow color for the circle, just select the circle and then add the ‘ Fill ’ modifier in the *Attributes Inspector* and specify the value you want (fig. 14). It is essentially a WYSIWYG editor where you specify what you want and Xcode will then generate the corresponding Swift code for you.

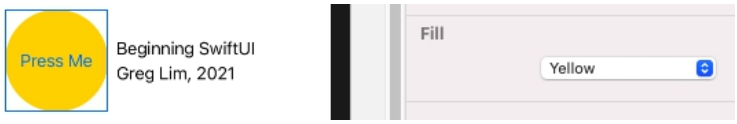



Figure 14

So suppose I want “ Beginning SwiftUI ” to be larger, I simply select it, and in Attributes Inspector, specify *Font* to be ‘ Large Title ’ . And I will get (fig. 15):

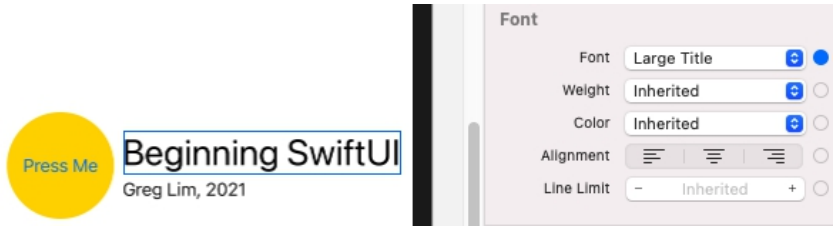



Figure 15

Xcode generates the code for me:

```
VStack(alignment: .leading, spacing: 4) {
    Text("Beginning SwiftUI")
        .font(.largeTitle)
    Text("Greg Lim, 2021")
```

```
}
```

See how convenient it is?

You can chain specify multiple modifiers to a View also by specifying in Attributes Inspector (fig. 16) (or in the code directly):

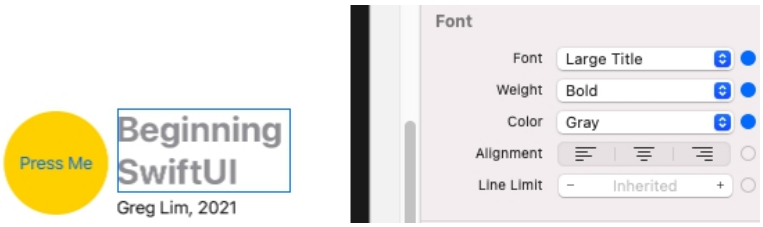



Figure 16

which generates the following code:

```
Text("Beginning SwiftUI")
    .font(.largeTitle)
    .fontWeight(.bold)
    .foregroundColor(Color.gray)
```

Now that we know how to create visual controls from code and buttons to call an action, let ' s modify our app to create a counter app. A counter app is useful especially when you organize events and want to count the number of attendees. You might have seen a physical one before (fig. 17).


Figure 17

In our counter app, there should be a label which shows how many times the button has been clicked. And each time the button is clicked, the count in the label should be incremented by one. We will then have another button which resets the count to zero. You will also have a variable to store how many times the button has been clicked.

## Solution

First, we will need a State variable to store how many times the button has been clicked. Declare a state variable *count* and initialize it to 0 as shown below:

```
struct ContentView: View {

    @State var count: Int = 0
```

```
var body: some View {
    …
```

Second, we increment *count* by one each time we tap on our button. Let ' s change " Press Me " to " Increment " .

```
var body: some View {
    HStack {
        ZStack {
            Circle()
                .fill(Color.yellow)

            Button(action:{
                self.count += 1
            }){
                Text("Increment")
            }
        }
    …
```

Next, we use String interpolation to display *count* .

```
        VStack(alignment: .leading, spacing: 4) {
            Text("Count: \(count)")
                .font(.largeTitle)
                .fontWeight(.bold)
                .foregroundColor(Color.gray)

            Text("Greg Lim, 2021")
        }
```

The app should look something like (fig. 18):


Figure 18

And each time you click on the ' Increment ' button, the counter increments by one!

The final code should look like:

```
import SwiftUI

struct ContentView: View {

    @State var count: Int = 0

    var body: some View {
        HStack {
            ZStack {
                Circle()
                    .fill(Color.yellow)

                Button(action:{
                    self.count += 1
                }){
                    Text("Increment")
                }
            }
            .frame(width: 100.0, height: 100.0)
            VStack(alignment: .leading, spacing: 4) {
                Text("Count: \(count)")
                    .font(.largeTitle)
                    .fontWeight(.bold)
                    .foregroundColor(Color.gray)

                Text("Greg Lim, 2021")
            }
        }

    }
}
```

# *@State*

You might be asking, what is *@State* ? The *@State* keyword declares it is a state variable. If a view is dependent on a state variable, SwiftUI ensures that the view is updated whenever the value of the state variable changes. In our case, whenever state variable count changes, the *Text* view *Text("Count: \(count)")* automatically updates to reflect the new value of *count* .

## *Try it yourself:*

Now, can you implement the ‘ Reset ’ button functionality which resets count to zero when pressed? Remember to create an action for the reset button and in it set *count* to 0.

## *Summary*

In this chapter, we explained what SwiftUI is and showed how SwiftUI enables us to quickly create user interfaces for our iOS application. We downloaded and installed Xcode and had a walkthrough of the interface. We saw how to layout our views using the two common stacking views, *VStack* and *HStack* . We created our first iOS application using SwiftUI and learned how the various components in our project work together.

# Chapter 2: Body Mass Index Calculator

In this chapter, we make a Body Mass Index (BMI) Calculator app. In it, we will reinforce the concepts we have learnt in the previous chapter. We will also learn new concepts like using *TextFields* to get input from users. If you are unfamiliar with BMI, it helps to see if we are at a healthy weight. To work out your BMI:

- divide your weight in kilograms (kg) by your height in meters (m)
- then divide the answer by your height again to get your BMI

For example: If you weigh 70kg and you're 1.75m tall, divide 70 by 1.75 – the answer is 40
then divide 40 by 1.75 – Your BMI is 22.9.

We then have the following classifications:
Underweight: Your BMI is less than 18.5.
Healthy weight: Your BMI is 18.5 to 24.9.
Overweight: Your BMI is 25 to 29.9.
Obese: Your BMI is 30 or higher.

We will be creating a new project for our BMI Calculator. So, close the current one, and once again go through the steps needed to create a new project: ‘ Create a New Xcode Project ’ , select ‘ App ’ , and name your project. I have named my ‘ BMISwiftUI ’ . Uncheck all the boxes as we won ’ t be using them yet (fig. 1).

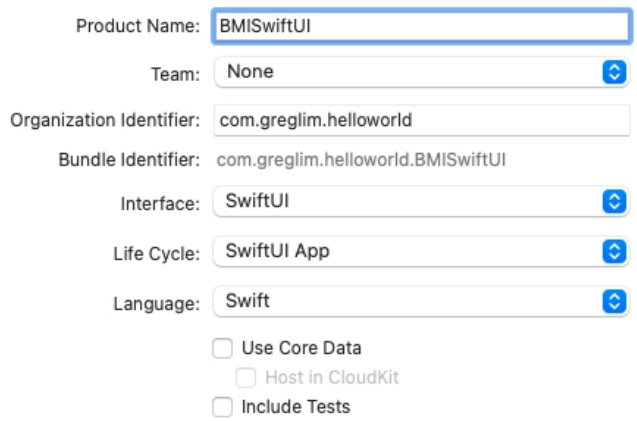



Figure 1

In *ContentView.swift* , we use two TextField views for our weight and height inputs. When you need users to enter some text or numbers, you can use the TextField view.
We also have a Text view that displays “ BMI Calculator ” on the top. We embed all of them in a VStack.

```
struct ContentView: View {
    @State private var weightText: String = ""
    @State private var heightText: String = ""

    var body: some View {
        VStack{
            Text("BMI Calculator:").font(.largeTitle)
            TextField("Enter Weight (in kilograms)",text: $weightText)
                .textFieldStyle(RoundedBorderTextFieldStyle())
                    .border(Color.black)

            TextField("Enter Height (in meters)",text: $heightText)
                .textFieldStyle(RoundedBorderTextFieldStyle())
                    .border(Color.black)

        }.padding()
    }
}
```

## *Code Explanation*

TextField("Enter Weight (in kilograms)",text: $weightText)

The first argument to the TextField view is the placeholder text where we provide hints to the user on how to enter input. In our case, we prompt to the user to enter weight in kilograms e.g. 85 (meaning 85 kg).

```
@State private var weightText: String = ""
  @State private var heightText: String = ""
```

At the top, we have two private state variables *weightText* and *heightText* . And each of them is bound to a *TextField* object through the *text* parameter.

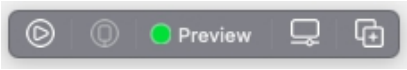
```
TextField("Enter Weight (in kilograms)",text: $weightText )
TextField("Enter Height (in meters)",text: $heightText )
```

Note that we have the *$* prefix to bind the state variable to the view.

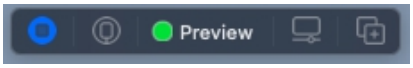
When the user types into the TextField, the value of the state variable is updated. Conversely, when we update the state variable, the TextField will also be updated.

## *Running our App*

Let ' s try running our app in the preview. Click the " Live Preview " play button on the far left.




The button turns into a blue ' stop ' and the " Live Preview " runs.




You should get something like (fig. 2):

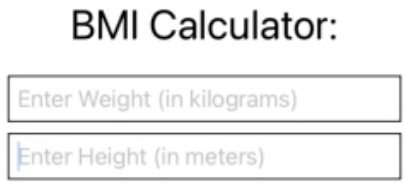



Figure 2

## *Modifiers*

Notice that we have added some modifiers to our TextField. For e.g.:

```
TextField("Enter Weight (in kilograms)",text: $weightText)
            .textFieldStyle(RoundedBorderTextFieldStyle())
                .border(Color.black)
```

Try removing them and see what happens. For e.g., if you remove .border, there won ' t be any border for the *TextField* and it will be difficult for the user to locate the view.

And *.textFieldStyle ( **RoundedBorderTextFieldStyle( )** )* makes the corners of the TextField rounded. With modifiers, we can style *TextFields* (and other views) to our liking.

## Calculate and Output BMI Button

Next, we add a " Calculate BMI " button with the following code:

```
@State private var weightText: String = ""
@State private var heightText: String = ""
 @State private var bmi: Double = 0

var body: some View {
    VStack{
            …

        Button(action:{
            let weight = Double(self.weightText)!
            let height = Double(self.heightText)!
            self.bmi = weight/(height * height )

        }){
            Text("Calculate BMI")
                .padding()
                .foregroundColor(.white)
                .background(Color.blue)
        }

            Text("BMI: \(bmi)").font(.title).padding()
        }.padding()
```

In the above code, we assign *weight* and *height* with the user inputted value from the TextField. We convert their inputted value from String to Double using *Double ( … )* because we will be

executing mathematical operations with it.
You might be asking, what is the use of the exclamation mark “ ! ” at the end of the line? We will be explaining this in a section later entitled, “ Optionals ” .

Below the button, we have a Text view that references the state variable *bmi* and displays it so that the user can see it. This is the reason why we have to declare *bmi* as a *@State* . So that when *bmi* changes in value, the Text view recomputes and updates its appearance.

## *Running our App*

When you run your app in the simulator, fill in your weight in kilograms and height in meters (e.g. weight: 85, height: 1.7) and when you click ‘ Calculate BMI ’ , you have your BMI index printed in the console (fig. 3)! How did you fare? My BMI isn ’ t too good. I had better exercise more!

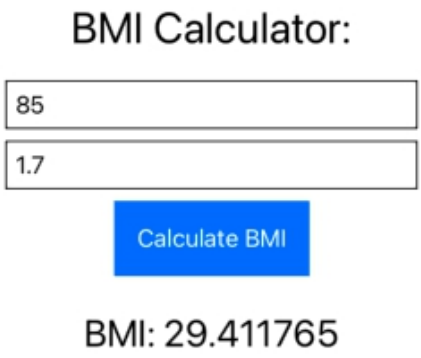



Figure 3

## *Formatting*

We should however only display the result to 1 decimal place. To format our result to 1 d.p., we simply add the *specifier* property:

```
Text("BMI: \(bmi, specifier: "%.1f" )").font(.title).padding()
```

and our BMI should now display a nicely formatted to 1 d.p. result.

The *specifier* parameter is a specialized string interpolation formatter. “ %.1f ” is the format code for “ floating-point number with

one digit after the decimal point ”  .

## BMI Classification

Next, we want to append the BMI classification to the BMI for e.g.: “ BMI: 29.4, Overweight  ”

We have to implement the below logic in our Swift code:
Underweight: Your BMI is less than 18.5.
Healthy weight: Your BMI is 18.5 to 24.9.
Overweight: Your BMI is 25 to 29.9.
Obese: Your BMI is 30 or higher.

To do so, add the following code:

```
struct ContentView: View {
    @State private var weightText: String = " "
    @State private var heightText: String = ""
    @State private var bmi: Double = 0
    @State private var classification: String = ""

    var body: some View {

        VStack{
                …

            Button(action:{
                let weight = Double(self.weightText)!
                let height = Double(self.heightText)!
                self.bmi = weight/(height * height)

                if self.bmi < 18.5{
                    self.classification = "Underweight"
                }
                else if self.bmi < 24.9{
                    self.classification = "Healthy weight"
                }
                else if self.bmi < 29.9{
                    self.classification = "Overweight"
                }
                else{
```

```
                    self.classification = "Obese"
                }
            }){
                Text("Calculate BMI")
                    .padding()
                    .foregroundColor(.white)
                    .background(Color.blue)
            }

            Text("BMI: \(bmi, specifier:"%.1f"),\(classification) ")
                        .font(.title)
                        .padding()
        }.padding()
    }
}
```

## *Code Explanation*

```
@State private var classification: String = ""
```

We first declare a state variable *classification* to store the classification result e.g. Underweight.

```
if self.bmi < 18.5{
    self.classification = "Underweight"
}
else if self.bmi < 24.9{
    self.classification = "Healthy weight"
}
else if self.bmi < 29.9{
    self.classification = "Overweight"
}
else{
    self.classification = "Obese"
}
```

We next then have a series of *if-else* statements to determine the classification as per the BMI classification logic.

```
Text("BMI: \(bmi, specifier:"%.1f"), \(classification) ")
    .font(.title)
    .padding()
```

Again, we use Swift string interpolation “ *\(classification )* ” to

append text before and after the calculated result.

When you run your app now, the BMI classification should be nicely appended to the calculated BMI (fig. 4).

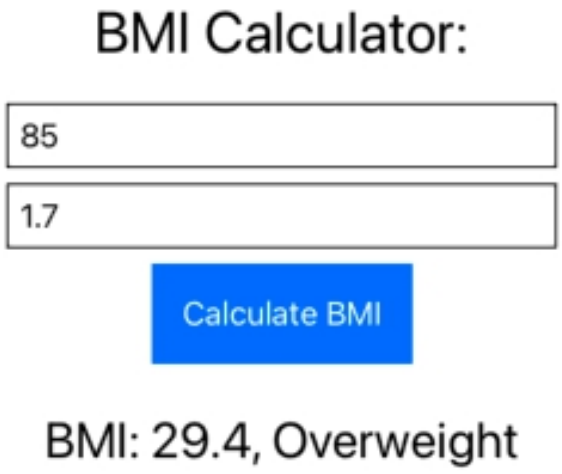



Figure 4

## *var* vs *let*

Now is a good time to revisit our code and look into when we should use *var* and when to use *let.* For example in our code, we declare *weight* , *height* and *bmi* using *let* .

```
    let weight = Double(self.weightText)!
let height = Double(self.heightText)!
self.bmi = weight/(height * height)

if self.bmi < 18.5{
    self.classification = "Underweight"
}
…
```

But we have declared *classification* using *var* .

The *let* keyword defines a constant, that is, the value won ’ t be changed afterward. In our case, *weight* , *height* after being inputted by the user and *bmi* after being calculated won ’ t be changed, thus we use *let* . We can of course use *var* for these variables and the code will still work. But Xcode will prompt you a warning to use *let*

instead (fig. 5).

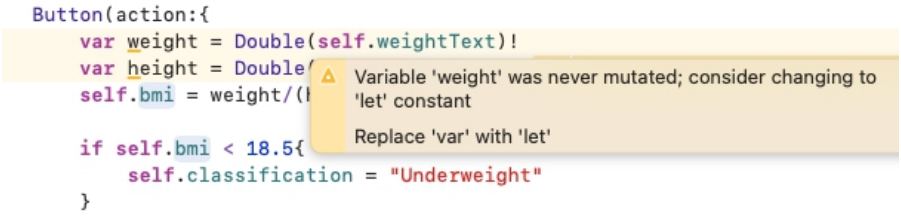


Figure 5

It is good practice to use *let* for variables which value never changes. *var* however defines an ordinary variable whose value changes during runtime. For example, *classification* changes during runtime.

## Optionals

Earlier, we had the below code with an exclamation mark at the end:

```
let weight = Double(self.weightText)!
let height = Double(self.heightText)!
```

So why do we need it? If you remove the ‘ ! ’ , you will get the following error (fig. 6).

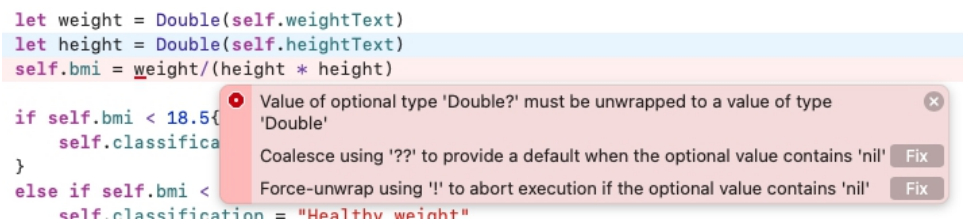


Figure 6

That is, Double(<String>) actually returns a ‘ Double? ’ which is a Double optional. In fact, any variable type with a ‘ ? ’ as suffix will make an existing type an optional. This is to specify that either this variable has a value, or it is nil. For example, *Double?* would either have a double value or nil. *Int?* would either have an integer value or nil.

You might ask, why should Double(self.weightText) return an optional? Shouldn ’ t it just return a double? Now when a user enters a string such as “ Hello World ” , this cannot convert into a double. So, when

a value cannot be converted into a double, it returns nil.

Therefore, Double(self.weightText ) returns an optional, that is, it returns the double value entered by the user and returns nil when the user does not enter anything in the textfield or enters an incompatible value e.g. letters.

To unwrap an optional, that is, we say get the value directly, we add " ! " as a suffix to a variable. But as you might have guessed, it is quite dangerous to do so especially when there ' s the possibility that our variable indeed has a nil value. In fact, if you run the BMI calculator app and leave the field blank and calculate BMI, the app will crash with an error like:

*" Fatal error: Unexpectedly found nil while unwrapping an Optional valu e "*

To avoid our app crashing when a textfield is left blank, we should unwrap the optional using ' *if le t* ' . '*if let* ' first checks if an optional variable contains an actual value and bind the non-optional form to a temporary variable. This is the safe way to "unwrap" an optional or in other words, access the value contained in the optional.

```
Button(action:{
    var weight: Double = 0
    var height: Double = 0

    if let weightDouble = Double(self.weightText) {
        weight = weightDouble
    }

    if let heightDouble = Double(self.heightText) {
        height = heightDouble
    }
     …
```

In the above code, we are saying, only move forward if *Double(self.weightText )* is not nil (i.e. it ' s a valid double numerical value) and in such a case, assign the value double to temporary

variable *weightDouble* (*weightDouble* does not exist outside the scope). Then, assign *weightDouble* to *weight* .

We then do the same for height.

## Previewing in Light and Dark Modes

So far, we have been previewing our app in the default Light mode. SwiftUI also supports Dark mode in iOS. You can test both modes during run time and during design time.

### *During run time*

When the application is running on the simulator, in the simulator, to go ‘ Settings ’ , ‘ Developer ’ and turn on ‘ Dark Appearance ’ . Your device will now run in dark mode and you can see your app ’ s dark mode UI (fig. 7).

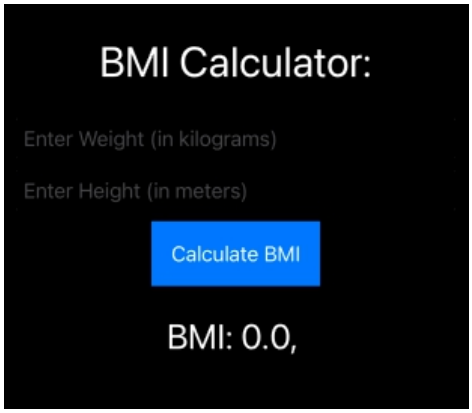



Figure 7

### *During design time*

To preview your UI in dark mode during design time, under the preview code block, use the *.environment()* modifier. For e.g.,

```
struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView().environment(\.colorScheme, .dark)
    }
}
```

If in dark mode, the text views change to white, but background remains white. (Running on simulator or device won ’ t have this issue). Try wrapping ContentView in a ZStack and set background to black:

```
static var previews: some View {
    ZStack{
        Color(.black)
        ContentView().environment(\.colorScheme,.dark)
    }
}
```

## *Summary*

In this chapter, we illustrated the basic structure of a SwiftUI view through the *Text* view and *Button* view. We were introduced to input views which allow us to collect input from users. Applying that knowledge, we went to create a Body Mass Index calculator app. In the next chapter, we will explore how to use *List* views in our app.

# Chapter 3: To Do List App Using List

In this chapter, we will be building the classic To-Do app where you have a list of to-dos in a List view (fig. 1). The List view is a container that displays rows of items.
We will see how to dynamically populate a list, add items to a list, remove items from a list, as well as arrange the position of items in the list.

When you tap on a to-do, you get to a to-do details page which shows the todo name plus an enlarged icon of the todo category.

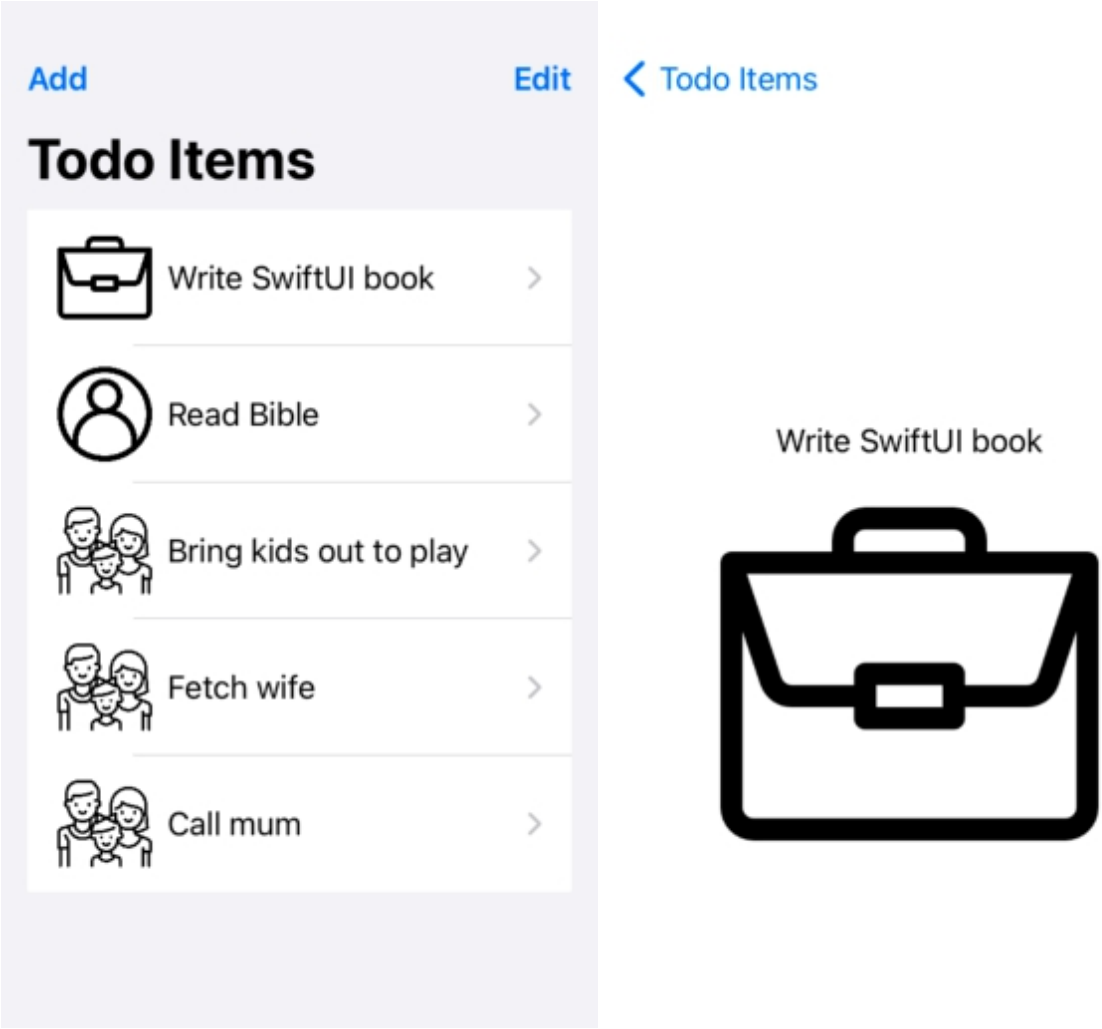


Figure 1

First, begin a new ‘ App ’ project in Xcode, call it *TodoSwiftUI* . Under ‘ Language ’ choose ‘ Swift ’ , under ‘ Interface ’ choose ‘ SwiftUI ’ , under ‘ Life Cycle ’ choose ‘ SwiftUI App ’ (fig. 2).
Leave ‘ Use Core Data ’ unchecked. We won ’ t be using Core Data for now, but we will implement it in the next chapter for data persistency.

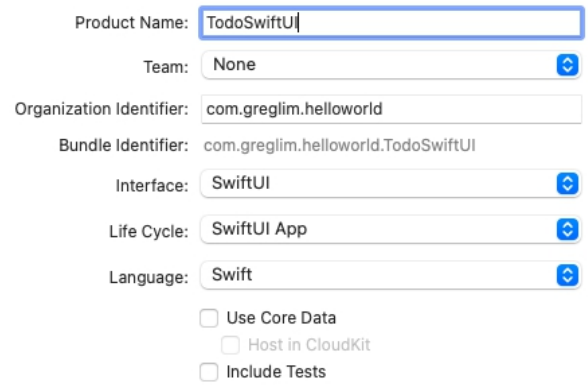



Figure 2

To have a List view displaying a list of to-dos, enter the following code in *ContentView* :

```
struct ContentView: View {
    var body: some View {
        List{
            Text("Write SwiftUI book")
            Text("Read Bible")
            Text("Bring kids out to play")
            Text("Fetch wife")
            Text("Call mum")
        }
    }
}
```

When the preview updates, you will see the five rows of text in a list (fig. 3):


Figure 3

Currently, each row displays a single Text view. But we display custom views to display more information.

Let ' s use the *HStack* to have each row contain an image and a text horizontally (fig. 4).


Figure 4

In our app, each image represents the category the todo belongs too (i.e.: ' work ' , ' personal ' or ' family ' ). The code to do this is:

```
var body: some View {
    List{
        HStack{
            Image("work").resizable().frame(width: 50, height: 50)
            Text("Write SwiftUI book")
        }
        HStack{
            Image("personal").resizable().frame(width: 50, height: 50)
```

```
            Text("Read Bible")
        }
        HStack{
            Image("family").resizable().frame(width: 50, height: 50)
            Text("Bring kids out to play")
        }
        HStack{
            Image("family").resizable().frame(width: 50, height: 50)
            Text("Fetch wife")
        }
        HStack{
            Image("family").resizable().frame(width: 50, height: 50)
            Text("Call mum")
        }
    }
}
```

The string in the *Image* view e.g. Image("family") refers to an image I have added to *Assets.xcassets* (fig. 5).

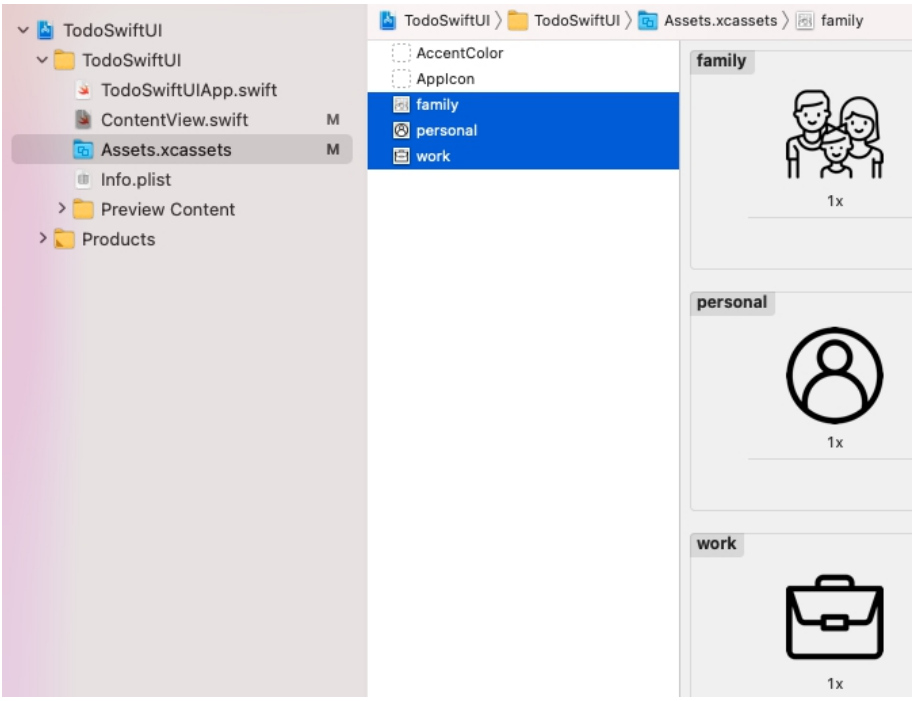



Figure 5

I got my images from [www.flaticon.com](http://www.flaticon.com) but you can use your own images. Just remember to drag and drop the images to *Assets.xcassets* and name them ‘ work ’ , ‘ personal ’ and ‘

family ’ .

Because images come in different sizes, big images would extend beyond the screen size and only a portion of it is shown. To fix that, we apply the *.resizeable* modifier to make the image fit in the available area:

```
Image("personal").resizable().frame(width: 50, height: 50)
```

We then apply the *.frame* modifier to restrict the size of the image to a custom frame.

## Populating Rows from an Array

Currently, we have a static list view where we have five pieces of fixed data. Let ’ s now see how to populate each row with values from an array.

Because each row now consists of a category and a text, we will create a *struct* to store them.
Structs allow us to create complex data types that are made up of multiple values. We can then create an instance of the struct and fill in its values to pass it around our code.

In *ContentView.swift* at the top, add:

```
struct Todo {
    let name: String
    let category: String
}
```

We have defined a *Todo* struct type that contains two properties: *name* and *category* .

Next, declare a state array of Todo structs in *ContentView.swift* :

```
struct ContentView: View {

    @State private var todos = [
        Todo(name:"Write SwiftUI book",category: "work"),
        Todo(name:"Read Bible",category: "personal"),
```

```
    Todo(name:"Bring kids out to play",category: "family"),
    Todo(name:"Fetch wife",category: "family"),
    Todo(name:"family",category: "Call mum")
]
    …
```

We use a state variable *todos* so that the items in the List view can be updated dynamically. Notice how Swift makes it easy to create instances of our *Todo* structs. We simply pass in initial values for *name* and *category* .

Next, fill in the below into *body* :

```
var body: some View {
  List{
    ForEach(todos, id:\.name){ (todo) in
      HStack{
        Image(todo.category) .resizable().frame(width: 50, height: 50)
        Text(todo.name)
      }
    }
  }
}
```

In the List view, we use a *ForEach* which receives an array and then creates multiple subviews. We have to supply *id* to uniquely identify each item. For now, we supply *.name* to use the todo name as the identifier for each row.

When you preview your app, it should show the same as before, only this time, your rows are populated programmatically.

## Displaying the List within a NavigationView

We will next implement a todo details screen. That is, when a user taps on a todo, we will go to a separate todo details screen. We achieve this by wrapping our List in a *NavigationView* .

```
var body: some View {
    NavigationView{
        List{
```

```
                ForEach(todos, id:\.name){ (todo) in
                        …
                }
            }.navigationBarTitle("Todo Items")
        }
    }
```

We also set the title in the navigation bar using the *navigationBarTitle* method. Notice that the *navigationBarTitle* modifier belongs to the List view and not to the Navigation View. This is because Navigation view displays new screens by pushing them on top from the right edge. Each screen has its own title. If the title is attached to the Navigation View, the title is fixed. By attaching the title to what ' s inside the Navigation view, the title can change as its content changes.


Figure 6

As you can see, a navigation bar has been added to the top of our List (fig. 6).

## *Making the items tappable*

Wrapping the List view in a NavigationView allows us to navigate to another page when a row is tapped. To navigate to the todo detail screen. Wrap the row item in a *NavigationLink* function:

```
NavigationView{
    List{
        ForEach(todos, id:\.name){ (todo) in
            NavigationLink(destination:
                VStack{
                    Text(todo.name)
                    Image(todo.category)
                            .resizable()
                            .frame(width: 200, height: 200)
                }
             ){
                HStack{
                            …
                }
            }
        }
    }.navigationBarTitle("Todo Items")
}
```

Notice that we have to provide an argument into NavigationLink which is the view that is displayed when an item is tapped. That is, the view will consist of:

```
NavigationLink(destination:
    VStack{
        Text(todo.name)
        Image(todo.category)
                .resizable()
                .frame(width: 200, height: 200)
    }
 )
```

When you run your app and tap on an item, the selected item details will be presented on another screen. The top bar in the new screen will also show ‘ Items ’ with a back symbol (fig. 7).


Figure 7

## Deleting a Todo Item

In iOS, to delete a particular row, we typically swipe to the left to show a Delete button displayed on the row (fig. 8).

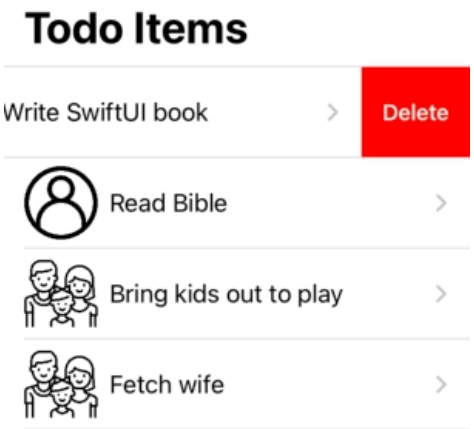



Figure 8

To enable this, we add the *.onDelete()* modifier at the end of the *ForEach* view:

```
var body: some View {
    NavigationView{
```

```
            List{
                ForEach(todos, id:\.name){ (todo) in
                    NavigationLink(destination:
                                …
                    }
                }
                .onDelete(perform: { indexSet in
                    todos.remove(atOffsets: indexSet)
                })
            }.navigationBarTitle("Todo Items")
        }
    }
```

*onDelete* provides us with the argument *indexSet* which contains all the positions of the items in the *ForEach* view to be removed. We pass *indexSet* to the *remove* function of *todos* to remove the specific rows.

## Rearranging Rows

We can allow users to rearrange rows in a List view with the *.onMove()* modifier at the end of the *ForEach* view.

```
    var body: some View {
        NavigationView{
            List{
                ForEach(todos, id:\.name){ (todo) in
                    NavigationLink(destination:
                        VStack{
                            Text(todo.name)
                            Image(…)
                        }
                        ){
                        HStack {
                            Image(…)
                            Text(todo.name)
                        }
                    }
                }
                .onDelete(perform: { indexSet in
                    todos.remove(atOffsets: indexSet)
                })
```

```
                .onMove(perform: { indices, newOffset in
                    todos.move(fromOffsets: indices, toOffset: newOffset)
                })
            }.navigationBarTitle("Todo Items")
            .navigationBarItems(trailing: EditButton())
        }
    }
```

*onMove* provides the *indices* argument which contains the indices of the items to move. *onMove* also provides the *newOffset* argument which is the index of the position to move to.

Note that the items can only be moved when the user enters 'Edit' mode. Thus, we need to have an *EditButton* added to the navigation bar. When the user taps on 'Edit', she can then proceed to move items (fig. 9).

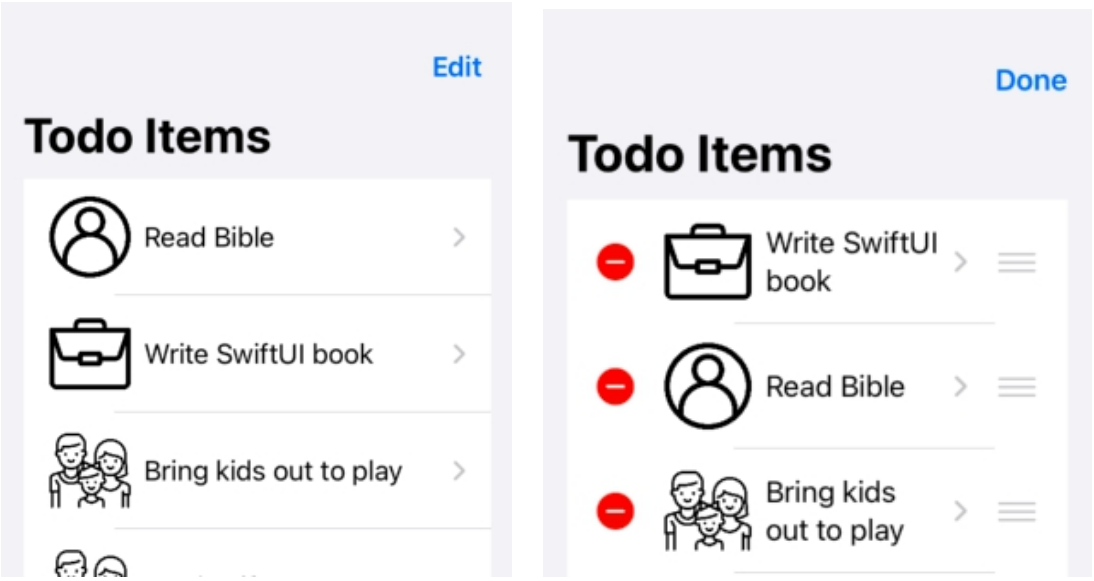



Figure 9

You would also notice that in Edit mode, each item displays a Delete button where users can quickly delete items by tapping on it. This is automatically enabled in Edit mode without us having to add any code!

## Creating Identifiers for our Todos

Now, what if we have multiple todo items that have the same name? For e.g. “do homework”, “do homework”. Remember that in our *ForEach* :

```
ForEach(todos, id:\.name)
```

If there are multiple todos with the same name, this will cause problems when we try to delete rows because List view doesn’t know which exact row to delete. To resolve this, we will add a unique identifier to our *Todo* struct.

```
struct Todo: Identifiable {
    let id = UUID()
    let name: String
    let category: String
}
```

We make *Todo* conform to the *Identifiable* protocol which requires the property *id* . *Identifiable* as its name suggests simply means “this type can be identified uniquely”. We use the *UUID()* function to general a unique identifier for each new todo.

Because we have an id, we can remove from *ForEach* :

```
ForEach(todos , id:\.name )
```

Because List view will automatically use *id* as the identifier for the rows.

## Adding a new todo item

To add rows to our List view, we add a new todo item to the *todos* array. We will add a navigation bar button item to the upper left corner (*leading* ) of the *NavigationView* for adding *todos* :

```
    var body: some View {
        NavigationView{
            List{
                        …
            }.navigationBarTitle("Todo Items")
                    .navigationBarItems(
```

```
                leading: Button(action: {},
                  label: {
                      Text("Add")
                  }),
                trailing: EditButton()
            )
        }
    }
```

We have to specify a function in the button's action to add a new Todo to *todos* .

```
        NavigationView{
            List{
                        …
            }.navigationBarTitle("Todo Items")
                    .navigationBarItems(
                     leading: Button(action: addTodo,
                       label: {
                           Text("Add")
                       }),
                     trailing: EditButton()
                )
        }
```

Implement the *addTodo()* function:

```
    func addTodo(){
        todos.append(Todo(name: "newTodo", category: "work"))
    }
```

Run your app now and when you tap on 'Add', a new todo item 'newTodo' is added to the bottom of the List view (fig. 10).

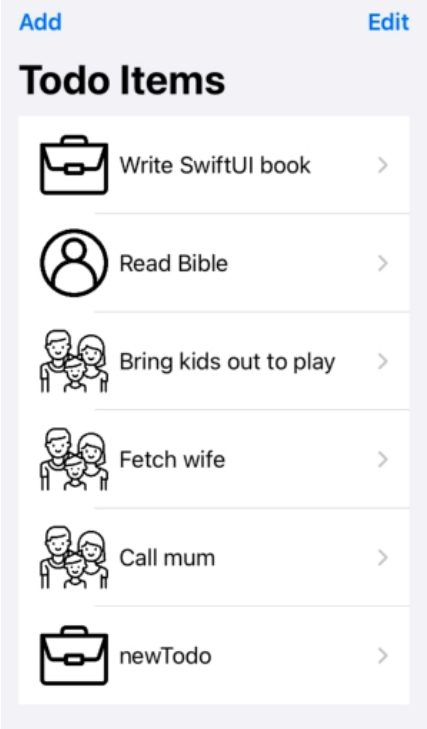


Figure 10

Our current new todo's name and category is hard-coded as "newTodo" and "work" which shouldn't be the case.

So let's comment out the *addTodo()* function and instead have a *AddTodoView* presented over *ContentView* for users to add a todo. Create our *AddTodoView* by adding the below at the end of *ContentView.swift* :

```
struct AddTodoView: View {
    var body: some View {
        Text("Add Todo view")
    }
}
```

Back in *ContentView* , add the below line:

```
struct ContentView: View {

    @State private var showAddTodoView = false
```

We have a state variable *showAddTodoView* to determine if we show *AddTodoView* . It is initially default to false (not showing).

Next, add the below codes in **bold** :

```
            ...
        }.navigationBarTitle("Todo Items")
        .navigationBarItems(
            leading:
                Button(action: {
                    self.showAddTodoView.toggle()
                },
                label: {
                    Text("Add")
                }).sheet(isPresented: $showAddTodoView){
                    AddTodoView()
                },
            trailing: EditButton()
        )
```

## *Code Explanation*

```
                Button(action: {
                    self.showAddTodoView.toggle()
                },
```

In the button's action, we call *toggle()* of *showAddTodoView* to toggle it from false to true.

```
            }).sheet(isPresented: $showAddTodoView){
                AddTodoView()
            },
```

To display the sheet, we use the *sheet* modifier and attach it to the Button view. We bind *showAddTodoView* state variable to the *isPresented* parameter of the *.sheet()* modifier. When *showAddTodoView* is true, the sheet is displayed (fig. 11).

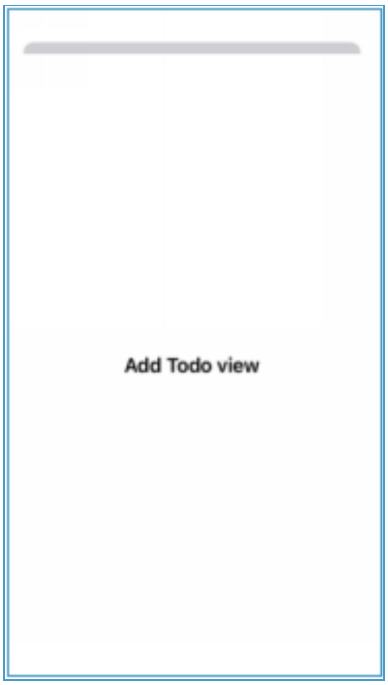


Figure 11

A user can drag the sheet downward to dismiss it. But in our case, we want to have two text fields and a Add button that when clicked, programmatically dismisses the screen.

## *@Binding*

To do so, let's first add a *Button* view and a *binding* variable *showAddTodoView* in *AddTodoView* (we will later explain what is *binding* ):

```
struct AddTodoView: View {
    @Binding var showAddTodoView: Bool

    var body: some View {
        Text("Add Todo view")

        Button(action: {
            self.showAddTodoView = false
        } ,
        label: {
            Text("Done")
        })
    }
```

}

Back in *ContentView* , add:

```
            …
        }.navigationBarTitle("Todo Items")
        .navigationBarItems(
            leading:
                Button(action: {
                    self.showAddTodoView.toggle()
                },
                label: {
                    Text("Add")
                }).sheet(isPresented: $showAddTodoView){
                    AddTodoView(showAddTodoView: self.$showAddTodoView )
                },
         …
```

## *Code Explanation*

```
struct AddTodoView: View {
        @Binding var showAddTodoView: Bool
        …
```

By declaring *showAddTodoView* as *@Binding* , we are declaring that its value will come from elsewhere and will be shared between *AddTodoView* and that other place.

That other place in our case is *ContentView* where we pass in the *showAddTodoView* value when instantiating *AddTodoView* :

```
                    …
                }).sheet(isPresented: $showAddTodoView){
                    AddTodoView(showAddTodoView: self.$showAddTodoView )
                },
```

So *ContentView* and *AddTodoView* share the *showAddTodoView* value. When *showAddTodoView* in *AddTodoView* is changed, the change is also reflected back in *ContentView* which will then dismiss the sheet.

# Adding User Inputs

Next, we will add a *TextField* for Todo name, a Picker for user to select a Todo category (“work”, “family”, “personal”) and a ‘Done’ button (fig. 12).

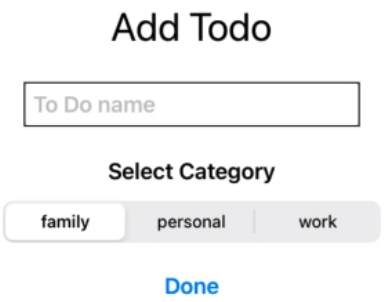



Figure 12

First, add the below into *AddTodoView* :

```
struct AddTodoView: View {
    @Binding var showAddTodoView: Bool

    @State private var name: String = ""
    @State private var selectedCategory = 0
    var categoryTypes = ["family","personal","work"]
```

The array *categoryTypes* will store the pre-defined categories we will display in the pickerview. State variable *selectedCategory* will store the array index of the category picked.

Next, add a *VStack* containing our input controls similar to what we did earlier in our BMI calculator.

```
    var body: some View {
        ~~Text("Add Todo view")~~

      VStack{
          Text("Add Todo").font(.largeTitle)
          TextField("To Do name",text: $name)
              .textFieldStyle(RoundedBorderTextFieldStyle())
              .border(Color.black).padding()

          Text("Select Category")
          Picker("",selection: $selectedCategory){
```

```
            ForEach(0 ..< categoryTypes.count){
                Text(self.categoryTypes[$0])
            }
        }.pickerStyle(SegmentedPickerStyle())

    }.padding()
            …
```

The *Text* and *TextField* views should be familiar to you. What is new is that we have used a new view called the Picker view.

```
Picker("",selection: $selectedCategory){
    ForEach(0 ..< categoryTypes.count){
        Text(self.categoryTypes[$0])
    }
}.pickerStyle(SegmentedPickerStyle())
```

The *Picker* view allows users to select an item from a set of defined values. A *ForEach* loops through the *categoryTypes* array and populates the picker. Notice at the end of the picker view we attach the modifier *.pickerStyle(SegmentedPickerStyle())* . This actually displays our Picker view as a *SegmentedControl* .
Notice what happens if we remove the modifier (fig. 13):


Figure 13

We get the classic picker view but which can be too large for

selecting from just a set of three values. It would be more appropriate to use the classic picker view if we have more values to offer the user.

## *Adding our Todo item to **todo** s array*

That we have fields to accept user input, let's implement adding our todo item to *todos* array.

To do so, *AddTodoView* needs to have access to *todos* in *ContentView* . So back in *ContentView* , we pass in todos to *AddTodoView* as highlighted in **bold** :

```
.navigationBarItems(
    leading:
        Button(action: {
            self.showAddTodoView.toggle()
        },
        label: {
            Text("Add")
        }).sheet(isPresented: $showAddTodoView){
          AddTodoView(showAddTodoView: self.$showAddTodoView,
                                                    todos:self.$todos )
        },
    trailing: EditButton()
)
```

And in *AddTodo* View, we receive *todos* with *@Binding* :

```
struct AddTodoView: View {
    @Binding var showAddTodoView: Bool


    @State private var name: String = ""
    @State private var selectedCategory = 0
    var categoryTypes = ["family","personal","work"]

    @Binding var todos: [Todo]
        …
```

Remember that *@Binding* allows us access to *todos* in *ContentView* . In the Button's action, we then create a *Todo* and then append it to *todos* :

```
Button(action: {
    self.showAddTodoView = false
    todos.append(Todo(name: name,
                      category: categoryTypes[selectedCategory]))
},
label: {
    Text("Done")
})
```

When you run your app now, you will have a fully functioning Todo app!

Note that when we use *append()* , the new item is added to the end of the array, meaning that the new todo is displayed at the last row. If we want the item to be added to the first of the array, we instead use *insert()* . Try it out!

As we complete this chapter, I wish to highlight an important SwiftUI development principle. SwiftUI adopts a philosophy of a “single source of truth”. That is, the data behind a view has only a single source. In our case, there is a single source for *todos* which is in *ContentView* . *todos* in *AddTodoView* refers to *todos* in *ContentView* through the *@Binding* keyword. Using *@Binding* allows us to bind data between views. This prevents multiple or copies of the same data which leads to data inconsistency.

## *Summary*

In this chapter, we have built the classic ‘To Do List’ app. We showed how to display a list of items using a *List* view, a container that displays rows of items in a single column. We explained how to dynamically populate a List, add items, remove items from it, and how to rearrange the position of items in the list.

The problem with our app now is that the data is not persistent. If we add new to-do items, they get displayed. But if we close the app and open it again, the data disappears. In the next chapter, we will use Core Data to make our data persistent.

# Chapter 4: Persistent Data Using Core Data

In this chapter, we will be using Core Data to keep our data persistent i.e., we will be able to save our to-do items and retrieve them even after we close and open our app. Currently, when we close our app, all data is gone. Core Data allows us to take the objects we have created and saved them into an internal database. We later retrieve the objects from the database when we need to.

Don't worry if you did not 'checked' the 'Use Core Data' checkbox when you created the project (fig. 1). We will add the usage of Core Data into our existing project from scratch.

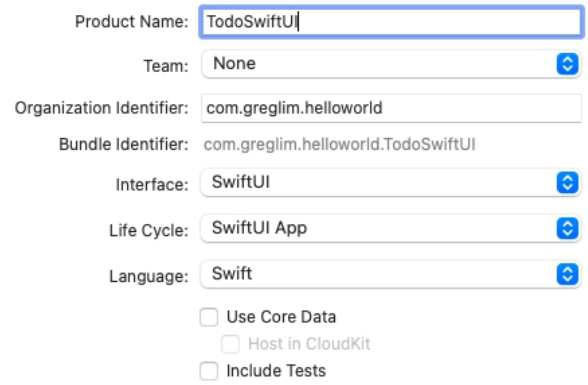



Figure 1

## Create a Data Model

Let's first create our data model file where we define our app's object structure e.g. their object type, properties and relationships.

In your project, go to 'File', 'New', 'File', and under 'Core Data', select 'Data Model' and hit 'Next' (fig. 2).

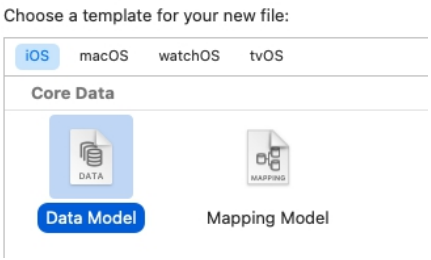



Figure 2

Save the model as ‘ Todo ’ and hit ‘ Create ’ . You will have a file called *Todo.xcdatamodeld* (fig. 3).

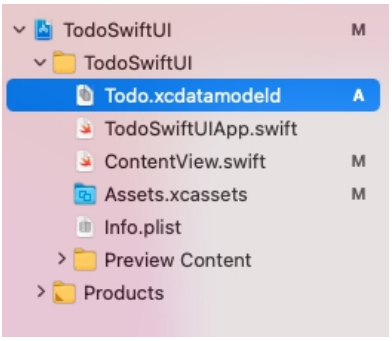
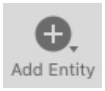



Figure 3

This is the file where we create ‘ entities ’ to be saved into Core Data. You can think of a Core Data ‘ entity ’ as the same as a ‘ Class ’ . Currently we have our struct *Todo* to represent a to-do object. But now, we will have an entity to represent our to-do object in Core Data.

Select the *Todo.xcdatamodeld* file and create a new entity by clicking on the ‘ Add Entity ’ button 

at the bottom of *Todo.xcdatamodeld* . An entity named ‘ Entity ’ will be created. Rename it to ‘ *TodoC D* ’ (CD abbreviation for CoreData).
Next in ‘ Attributes ’ , add the attributes *name* and *category* and specify their type to be *String* (fig. 4).

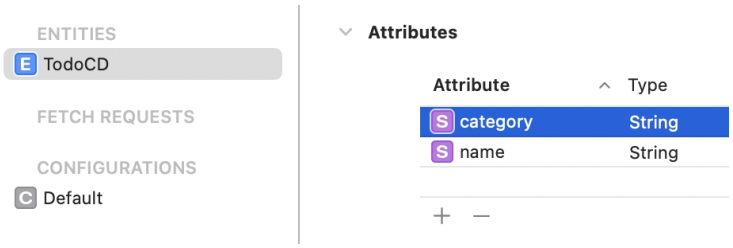


Figure 4

Behind the scenes, Xcode automatically creates a *NSManagedObject* class for *TodoCD* . Instead of our current *Todo* struct, we have the *TodoCD* class which conforms to the *NSManagedObject* class. In Core Data, a stored object is an instance of *NSManagedObject* . *NSManagedObject* conforms to the *ObservableObject* protocol so that it can be observed directly by a SwiftUI ' s view. In other words, when the *NSManagedObject* instance changes, the views also get updated automatically. We will explore more about *ObservableObject* in the next chapter.

Having created our Data Model, we will be adding Core Data functionality to our project in the following sections.

## Applying Core Data functionality

In this section, we attempt to encapsulate the Core Data stack in our app by first having a *PersistentController* struct. This struct will wrap the Core Data framework to simplify managing the Persistent store.

Create a new swift file *Persistence.swift* . Fill it with the following code:

```
import CoreData

struct PersistentController{
    static let shared = PersistentController()
    let container: NSPersistentContainer


    init(){
        container = NSPersistentContainer(name: "Todo" )
        container.loadPersistentStores(completionHandler: {(storeDescription, error)
in
            if let error = error as NSError? {
                fatalError("unresolved error:\(error)")
            }
        })
    }
}
```

## *Code Explanation*

```
static let shared = PersistentController()
```

We declare a static instance of *PersistentController* called *shared* . This is a singleton pattern which means that there is only one instance of *PersistentController* to be shared by other files.

```
init(){
    container = NSPersistentContainer(name: "Todo")
```

*init()* provides you different ways to create your struct (if you are familiar with Java, it is similar to a constructor). We access the container by providing the name of the *.xcdatamodeld* file " Todo " to *NSPersistentContainer* . *NSPersistentContainer* contains the core data stack.

```
        container.loadPersistentStores(completionHandler: {(storeDescription, error)
in
            if let error = error as NSError? {
                fatalError("unresolved error:\(error)")
            }
        })
```

We then instruct the persistent container to load the persistent store with *loadPersistentStores().* We have to provide a completion handler which is called when *loadPersistentStores()* completes. In the completion handler, *NSError* value will be populated if there is an error in the loading of the persistent store.

## *TodoSwiftUIApp.swift*

Next in *TodoSwiftUIApp.swift* , add in the following code in **bold** :

```
import SwiftUI

@main
struct TodoSwiftUIApp: App {


    var body: some Scene {
        let persistentContainer = PersistentController.shared

        WindowGroup {
```

```
            ContentView().environment(\.managedObjectContext,
                        persistentContainer.container.viewContext)
        }
    }
}
```

We first access the persistent container from the shared instance. *persistentContainer.container* being our *NSPersistentContainer* instance contains the *NSManagedObjectContext* instance in its *viewContext* property. *NSManagedObjectContex t* ' s *viewContext* acts like a doorway to allow users to save/fetch the managed objects from the core data stack.

We put *viewContext* in the environment keypath *\.managedObjectContex t* so that it can be injected into the View environment. *viewContext* allows us to store items into the Core Data with its *save()* method. We will revisit this later.

## *ContentView.swift*

Back in *ContentView.swift* , we create a property wrapper

```
struct ContentView: View {

    @Environment(\.managedObjectContext) private var viewContext
    …
```

We can then access *managedObjectContext* in *ContentView* by retrieving *viewContext* from the environment.

We can remove the previous *Todo* struct since we are no longer using it.

```
~~struct Todo: Identifiable{~~
    ~~let id = UUID()~~
    ~~let name: String~~
    ~~let category: String~~
~~}~~
```

Remove the dummy data as well:

```
    ~~@State private var todos = [~~
        ~~Todo(name:"Write SwiftUI book",category: "work"),~~
```

```
        Todo(name:"Read Bible",category: "personal"),
        Todo(name:"Bring kids out to play",category: "family"),
        Todo(name:"Fetch wife",category: "family"),
        Todo(name:"Call mum",category: "family")
    ]
```

We are no longer using them since we are using Core Data.

## *@FetchRequest*

Instead, we fetch all the todos stored in Core Data using the *@FetchRequest* property wrapper to display them in a List.

```
struct ContentView: View {


  @Environment(\.managedObjectContext) private var viewContext
  @FetchRequest(sortDescriptors:[]) private var todosCD:
FetchedResults<TodoCD>
   …
```

We can use the *sortDescriptors* argument to specify our sort criteria of the retrieved results e.g. by name, date, ascending, descending. Internally, *@FetchRequest* retrieves an object from *managedObjectContext* and expects it in *@Environment(\.managedObjectContext)* . We have earlier in *struct TodoSwiftUIAp p* injected *managedObjectContext* i nto *environment(\.managedObjectContext)* so our code will work without any further configuration needed:

```
struct TodoSwiftUIApp: App {


    var body: some Scene {
        let persistentContainer = PersistentController.shared


        WindowGroup {
            ContentView().environment(\.managedObjectContext,
                          persistentContainer.container.viewContext)
        }
    }
}
```

We then display them in the List:

```
var body: some View {
    NavigationView{
        List{
            ForEach(todosCD, id:\.self ){ todo in
                NavigationLink(destination:
                    VStack{
                        Text(todo.name ?? "untitled" )
                        Image(todo.category ?? "" )
                                .resizable().frame(width: 200, height: 200)
                    }
                    ){
                    HStack{
                        Image(todo.category ?? "" )
                                .resizable().frame(width: 50, height: 50)
                        Text(todo.name ?? "untitled" )
                    }
                }
            }
             …
```

Note that because *@FetchRequest* returns *NSManagedObject* instances which are optionals, i.e. they can possibly be null. We use ‘ *? ? ’* in the event that the values are null to give them a default value, e.g. “ untitled ” .

And because we are no longer using the *todos* array, we have to remove the *.onMove* modifier since *todosCD* does not support the *move()* method.

```
ForEach(todosCD, id:\.self){ todo in
        …
}
.onDelete(perform: { indexSet in
    …
})
.onMove(perform: { indices, newOffset in
    todos.move(fromOffsets: indices, toOffset: newOffset)
})
```

## Saving into Core Data

Now let ’ s implement adding a to-do item to Core Data. Because we are no longer adding to the *todos* array, but to Core Data itself, we

don ’ t have to pass it to *AddTodoView* . Thus, remove:

```
.navigationBarItems(
    leading:
        Button(action: {
            self.showAddTodoView.toggle()
        },
        label: {
            Text("Add")
        }).sheet(isPresented: $showAddTodoView){
           AddTodoView(showAddTodoView: self.$showAddTodoView
                    ~~, todos:self.$todos~~ )
        },
    trailing: EditButton()
)
```

And in *AddTodoView* , we can remove:

```
struct AddTodoView: View {

    @Binding var showAddTodoView: Bool

    @State private var name: String = ""
    @State private var selectedCategory = 0
    var categoryTypes = ["family","personal","work"]

    ~~@Binding var todos: [Todo]~~
```

Instead, add the following to retrieve the *managedObjectContext* from environment:

```
struct AddTodoView: View {
    @Environment(\.managedObjectContext) private var viewContext
```

And in the button ’ s action, we add the following codes:

```
        Button(action: {
            self.showAddTodoView = false
            let newTodoCD = TodoCD(context: viewContext)
            newTodoCD.name = name
            newTodoCD.category = categoryTypes[selectedCategory]
            ~~todos.append(Todo(name:name,category:
categoryTypes[selectedCategory])~~ )
            do{
```

```
            try viewContext.save()
        }
        catch{
            let error = error as NSError
            fatalError("unresolved error:\(error)")
        }
    }
```

In the above, we create a new *TodoCD* instance in *managedObjectContext* and then save it into Core Data with:

```
viewContext.save()
```

This tells *viewContext* what has changed and save the current progress. Note that we add *try* before the *viewContext.save()* with a *do/catch* to handle possible exceptions thrown e.g. there is not enough space on the phone. As an exercise, add something to warn the user that the *todo* isn ' t saved in case of an error, and maybe try again.

## Deleting a Todo from Core Data

In the *.onDelete* modifier of the *ForEach* component in *ContentView* , we will call a new method *deleteTodo* :

```
.onDelete(perform: { indexSet in
    todos.remove(atOffsets: indexSet)
    deleteTodo(offsets: indexSet)
})
```

Remember that the argument *indexSet* is a collection of row indexes that should be deleted:

Implement *deleteTodo* with the following code:

```
private func deleteTodo(offsets: IndexSet){
    for index in offsets{
        let todo = todosCD[index]
        viewContext.delete(todo)
    }

    do {
        try viewContext.save()
```

```
        } catch {
            let error = error as NSError
            fatalError("unresolved error:\(error)")
        }
    }
```

We loop through the indexes that ought to be deleted, retrieve the exact *ManagedObject* at that index and call the Managed Object Context ' s *delete()* . After the *for* loop, we save the Core Data context.

## *Running your App*

When you run your app now, upon swiping the cells to the left and hitting the red Delete button, the todos will be permanently deleted.

## Updating in Core Data

We will go through the essential code to perform an update in Core Data. Let ' s allow the user to add a smiley face on to the todo name whenever she does a long press gesture on a todo, indicating that the todo is done.

Add a *.onLongPressGesture* modifier on the *NavigationLink* :

```
NavigationLink(destination:
                …
}.onLongPressGesture(perform: {
    updateTodo(todo: todo)
})
```

Create a function *updateTodo* with the following code:

```
private func updateTodo(todo: FetchedResults<TodoCD>.Element){
    todo.name = "  ��  " //press control+commmand+space for emoticon editor
    do {
        try viewContext.save()
    } catch {
        let error = error as NSError
        fatalError("unresolved error:\(error)")
    }
```

}

As you can see, updating a Managed Object in Core Data involves just retrieving it, changing the value of its properties and then saving the context. You can extend this to updating the todo ' s name and category.

## *Running your App*

When you run your app and do a long press on a *todo* , the name will change to a smiley face (fig. 5)! And all these data are kept persistent.


Figure 5

## *Sorting in Core Data*

Additionally, you can sort the data from *@FetchRequest* by specifying a *SortDescriptor* . For example:

@FetchRequest(sortDescriptors: **[NSSortDescriptor(keyPath:\TodoCD.name, ascending:false)]** ) private var todosCD: FetchedResults<TodoCD>

## *Summary*

In this chapter, we learned how to use Core Data to keep our data persistent, save our to-do items and retrieve them even after we close and open our app. We learned that a stored object in Core Data is an instance of *NSManagedObject* which conforms to the *ObservableObject* protocol so that it can be observed directly by a

SwiftUI's view.

We used *@FetchRequest* property wrapper to fetch objects for the Core Data repository and load data in a view. We learned how to perform the basic persistence operations, creating, reading, updating and deleting persistent objects in Core Data and SwiftUI.

In the next chapter, we will explore how to extend Core Data to store our data in CloudKit.

# Chapter 5: Extending Core Data to CloudKit

In this short chapter, we will improve our To-Do list by storing on CloudKit via the CloudKit persistent container. Storing data in CloudKit allows data management and synchronization across multiple devices. We will see how Core Data (local storage) and CloudKit (cloud storage) work together seamlessly without us needing to write a lot of code to synchronize them.

In your project, under ‘ Signing & Capabilities ’ , make sure that ‘ Automatically manage signing ’ has been checked (fig. 1).

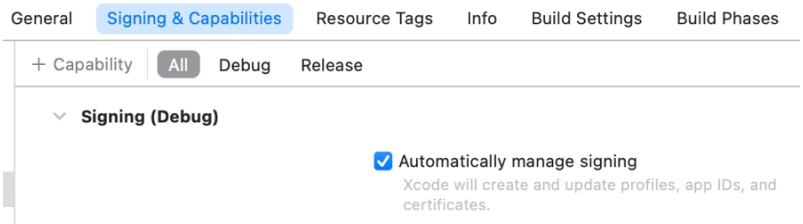



Figure 1

Next, go to ‘ + Capability ’ , search for ‘ iCloud ’ and select it. The ‘ iCloud ’ capability will then be added. In it, check ‘ CloudKit ’ . Then, add a ‘ Container ’ (fig. 2).

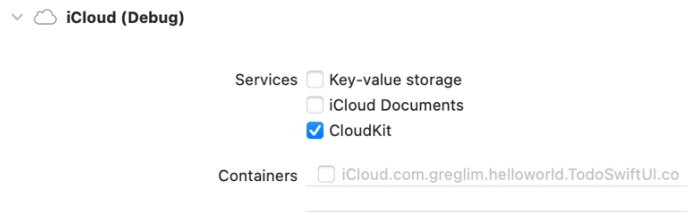



Figure 2

At this point, the ‘ Push Notification ’ capability will automatically be added for you.


Now, we need to add another capability called ‘ Background Modes ’ . So add it and check the ‘ Remote Notifications ’ check box (fig. 3).

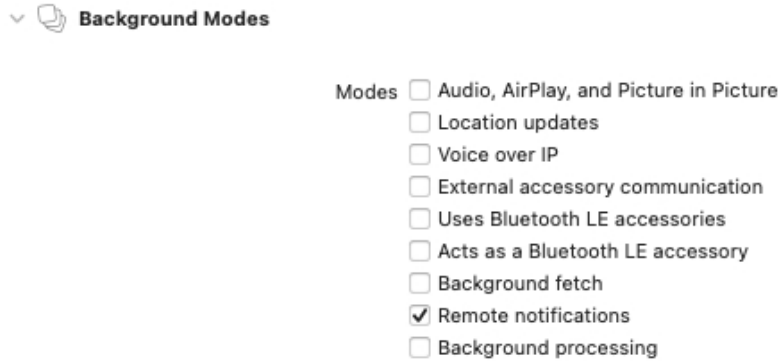



Figure 3

By enabling ‘ Remote Notifications ’ , multiple devices connected to the same iCloud account can be notified when the data in iCloud changes.

Now, let ’ s go to *Persistent.swift* . Add the following code changes:

```
import CoreData

struct PersistentController{
    static let shared = PersistentController()
    let container: NSPersistentContainer


    init(){
        container = NSPersistent**CloudKit** Container(name: "Todo")
        container.loadPersistentStores(completionHandler: {(storeDescription, error)
in
            if let error = error as NSError? {
                fatalError("unresolved error:\(error)")
            }
        })

        **container.viewContext.automaticallyMergesChangesFromParent = true**
```

```
        container.viewContext.mergePolicy =
NSMergeByPropertyObjectTrumpMergePolicy
    }
}
```

## *Code Explanation*

First, we rename:

container = NSPersistentContainer(name: "Todo")

to,

container = NSPersistent**CloudKit** Container(name: "Todo")

NSPersistentCloudKitContainer is a subclass of NSPersistentContainer. It provides us with additional functionality to mirror Core Data to CloudKit ' s private database.

Notice that we don ' t have to change the declaration:

```
let container: NSPersistentContainer
```

Because NSPersistent **CloudKit** Container is of NSPersistentContainer type. We then have:

container.viewContext.automaticallyMergesChangesFromParent = true

which automatically manages changes once the silent notification comes in. We also have

container.viewContext.mergePolicy =
NSMergeByPropertyObjectTrumpMergePolicy

which merges conflicts between the persistent store in iCloud and the current in-memory version by individual property. The in-memory changes trump the persistent store ' s version.

## *Running your App*

To test out if our app works, run the app on two different simulators e.g. iPhone 11 and iPhone 12. Alternatively, you can deploy the app on two different actual devices. Be sure to login in with the same Apple id (with iCloud enabled) onto both devices under ' Settings ' .

When you have deployed your app on both devices, try adding a todo on one device. If you have implemented the code in this chapter correctly, the other device will have the todo appearing as well. And when do any update or delete, it will be reflected on the other as well.

Notice how Apple has made it quite easy to implement CloudKit. There wasn ’ t a lot of code changes transiting from Core Data to CloudKit.

# Chapter 6: Getting Data from an API: GitHub Users

In this chapter, we will learn how to connect our app with the Internet. We will be connecting to an API to get the data of GitHub users.

Start a new ‘App’ project and name it ‘GitHub Users’. Choose the options for SwiftUI as we have done before. As we won’t be using Core Data, leave that check box unchecked.

## GitHub RESTful API

We will illustrate by connecting to the GitHub RESTful API to retrieve and manage GitHub content. You can know more about the GitHub API at

https://docs.github.com/en/free-pro-team@latest/rest

But as a quick introduction, we can get GitHub users data with the following url,

https://api.github.com/search/users?q=<search term>

We simply specify our search term in the url to get GitHub data for user with name matching our search term. An example is shown below with search term *gre g* .

https://api.github.com/search/users?q=greg

When we make a call to this url, we will get the following json objects as a result (fig. 1).

fig. 1

Note that the value of the *items* key is an array of users, each containing the details of each user. In each user, we want to retrieve the following:
*login* – User's GitHub login
*url* – User's GitHub json url
*avatar_url* – User's avatar image
*html_url* - User's GitHub webpage

## *Structs*

In *ContentView.swift* , create the following structs:

```
struct Result: Codable{
    var items: [User]
}

struct User: Codable{
    public var login: String
    public var url: String
    public var avatar_url: String
    public var html_url: String
}
```

Both structs conform to the *Codable* protocol so that we can later pass them to JSON Decoder to decode the JSON response string back into the respective structs. Why do we need two separate structs? We use the *Result* struct to encapsulate the first level of the JSON results.

We specify that we just want access to the *items* property, which is an array of *User* type. We then define the *User* struct with the properties that we want from the JSON results.

In our case, we have just accessed two levels of the JSON response, *items* and its property, *users* . But there is no limit to the number of levels Codable will go through. Just ensure that your structs should match the JSON structure.

Also, note that the variable names have to be exactly like in the JSON file. For e.g. it has to be strictly *avatar_url* instead of *avatarUrl* .

## *ObservableObject FetchUsers class*

Next, create a class *FetchUsers* which conforms to the *ObservableObject* class with the following code:

```
class FetchUsers: ObservableObject {
  @Published var items = [User]()


    init() {
        let url = URL(string: "https://api.github.com/search/users?q=greg")!
```

```
        URLSession.shared.dataTask(with: url) {(data, response, error) in
            do {
                if let data = data {
                    let decodedData = try JSONDecoder().decode(Result.self, from:
                                                                    data)
                    DispatchQueue.main.async {
                        self.items = decodedData.items
                    }
                } else {
                    print("No data")
                }
            } catch {
                print("Error: \(error.localizedDescription ?? "unknown error")")
            }
        }.resume()
    }
}
```

## *Code Explanation*

```
class FetchUsers: ObservableObject {
  @Published var items = [User]()
```

Our *FetchUsers* class conforms to the *ObservableObject* . Remember we have used the *@Binding* property wrapper to bind state variables of primitive types to properties in another view. To bind more complex objects, we use *@ObservableObject* instead. *@ObservableObject* allows the view to track changes made to members of a class.

We have a *items* variable to store the decoded JSON User content. The *@Published* prefix means that any view bound to this property will automatically be notified when the property value changes.

```
let url = URL(string: "https://api.github.com/search/users?q=greg")!
```

We create a *URL* object from the API url string (we can force unwrap the string since its hardcoded). For now, our url string has a hardcoded search term ‘greg’. As an exercise, try implement retrieving a user entered search term from a Text field and concatenate it into the url string.

```
URLSession.shared.dataTask(with: url) {(data, response, error) in
```

We then input the url into *URLSession.shared.dataTask* which creates the task for calling a web service endpoint on a remote server. *URLSession* is used to create the session. With the session, we then call *dataTask* to create a data task.

*URLSession.shared.dataTask* expects us to provide a completion handler which is the function that gets called when the request completes. When the request completes, we are provided the data retrieved, the response and any errors thrown.

```
    do {
        if let data = data {
            let decodedData = try
                    JSONDecoder().decode(Result.self, from: data)
            DispatchQueue.main.async {
                self.items = decodedData.items
            }
        } else {
            print("No data")
        }
    } catch {
        print("Error: \(error.localizedDescription ?? "unknown error")")
    }
}.resume()
```

In the completion handler function we provide, we check if *data* is not nil, in which case our request got connected successfully and we proceed to retrieve the data. Else, we print “No data”.

When the JSON content is downloaded, we use the *JSONDecoder()* object’s *decode()* function to decode the JSON content into the *Result* struct we defined earlier. When the conversion is done, we assign the result to the *items* variable.

```
            DispatchQueue.main.async {
                self.items = decodedData.items
            }
```

Notice that we require a DispatchQueue.main.asyn c . Without this. Our

retrieval code will run on a separate thread to the main thread. Thus, to run code on the main thread, we enclose the retrieval in DispatchQueue.main.async. Whatever is enclosed in *DispatchQueue.main.async* is run on the main thread.

Because tasks are created in a suspended state, we start the task with *resume()* .

## Defining the View

Next, let's define the view. We will use a List view to display the list of users:

```
struct ContentView: View {
    @ObservedObject var fetch = FetchUsers()

    var body: some View {
        List(fetch.items, id: \.login) { user in
            VStack(alignment: .leading) {
                Text(user.login)
                Text("\(user.url)")
                    .font(.system(size: 11))
                    .foregroundColor(Color.gray)
            }
        }
    }
}
```

We first create an instance of *FetchUsers()*

With the *@ObservedObject* prefix, we indicate that *fetch* will be observed for changes to it and the views that are bounded to it will be automatically notified when that happens. (see how ObservableObject and ObservedObject go hand in hand?)

The List view is bound to *fetch* 's *items* property and we use a VStack to display the user's GitHub login id and url. Here's how the app looks like when you run it (fig. 2):

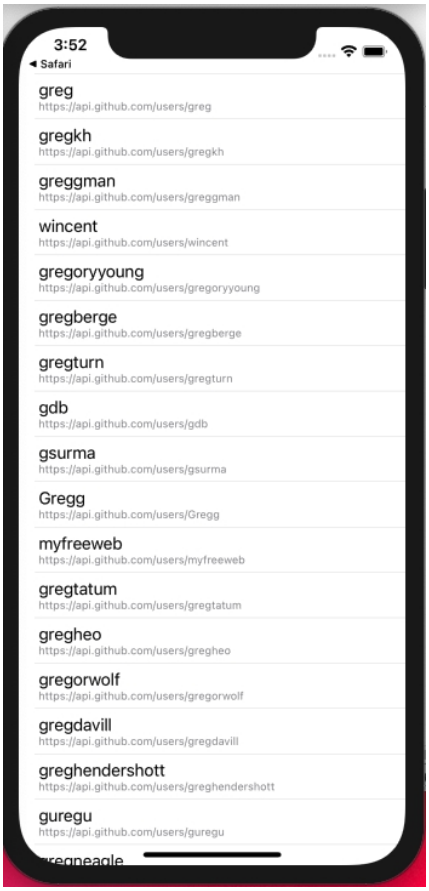


Figure 2

## Display Images Remotely

So far, we have displayed our GitHub users data nicely in a List view. Next, we will explore how to show the user's avatar image.

We have previously used the Image view to display images. The problem with the Image view is it can only display images locally bundled with our application.

To display an image from the web, we will use solutions that others have created to load images remotely and asynchronously. We will make use of the *URLImage* view developed by https://github.com/dmytro-anokhin/url-image .

To use it, we need to add its package to our project. In Xcode, go to 'File', 'Swift Packages', 'Add Package Dependency'. Enter the url:

https://github.com/dmytro-anokhin/url-image (fig. 3)

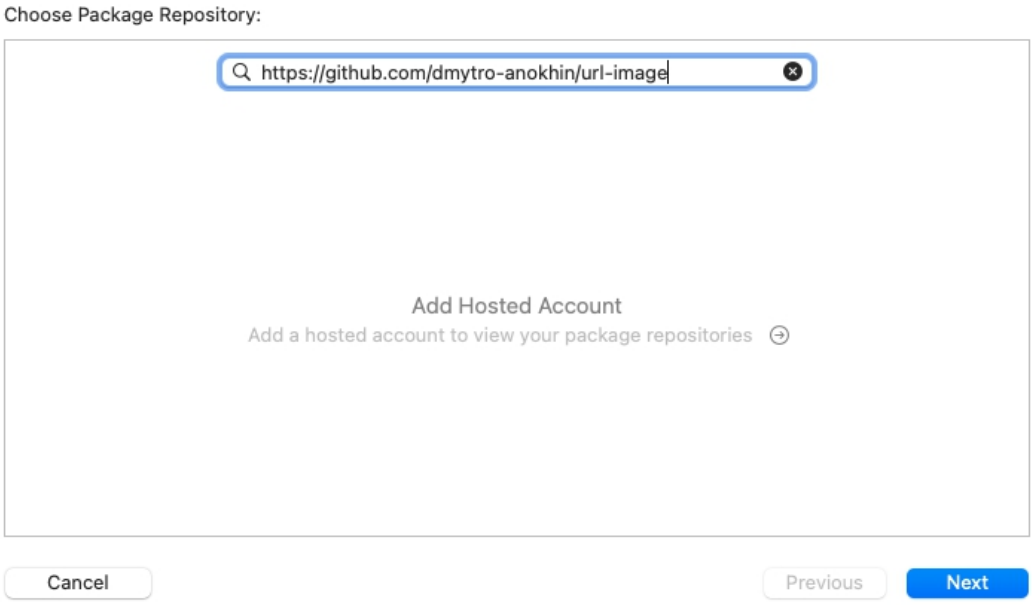



Figure 3

Click ‘Next’, ‘Next’ and ‘Finish’. The package will then be added. Now, add in *ContentView* :

```
import SwiftUI
import URLImage
…

    var body: some View {
        List(fetch.items, id: \.login) { user in
            HStack(alignment: .top) {

                URLImage(url:URL(string:user.avatar_url)!){ image in
                    image.resizable().frame(width: 50, height: 50)
                }
                VStack(alignment: .leading) {
                        …
                }
            }
        }
    }
```

We add the *URLImage* view to each row in the List

view. The image url is stored in *user.avatar_url* . When you run your app, the avatar image is displayed next to their details (fig. 4).

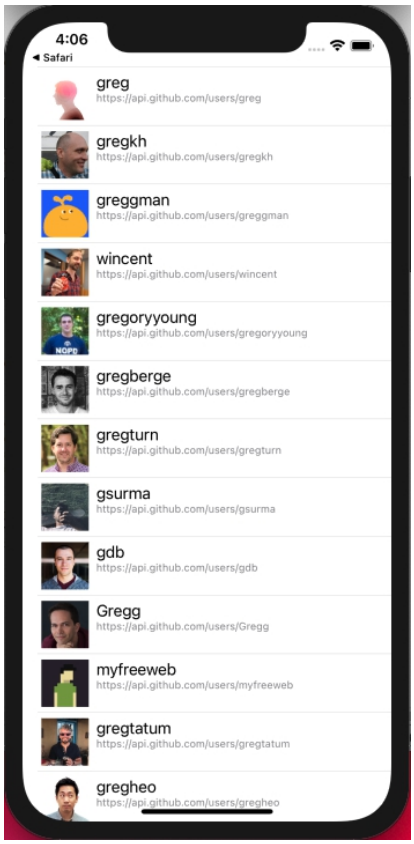


Figure 4

Wrapping each User in a Link View

Next, we implement that upon tapping a row, we will be brought to the GitHub webpage of the user in a web browser. The url is stored in the *html_url* property of the User struct. So, add the below:

```
var body: some View {
    List(fetch.items, id: \.login) { user in
        Link(destination: URL(string: user.html_url)!){
            HStack(alignment: .top){
                URLImage(url:URL(string:user.avatar_url)!){ image in
                    image.resizable().frame(width: 50, height: 50)
```

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

When you run the app, tap on a row and the user's web page will be loaded in the web browser (fig. 5).

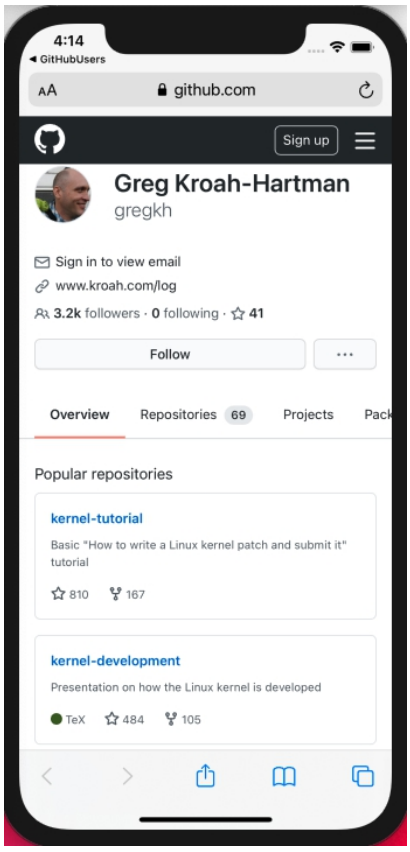


Figure 5

SwiftUI provides a dedicated *Link* view that opens a URL in Safari when pressed. You just have to provide the destination URL to show.

## Showing a Loader Icon

While getting content from a server, it is often useful to show a loading icon to the user  .

In *FetchUsers* , we are currently fetching data in *init* (). Let's instead have a separate function *loadData()* to do so. So rename *init* () into a new function *loadData* ().

```
init() {
    …
}

func loadData(){
    …
}
```

In *body* of *ContentView* , we check if *fetch.items* has no items (indicating that retrieval is still in progress). If so, show the *ProgressView* and call *loadData* in the *ProgressView* 's .*onAppear()* .

```
var body: some View {
    if(fetch.items.count == 0){
        ProgressView().onAppear(){
            fetch.loadData()
        }
    }
    else{
        List(fetch.items, id: \.login) { user in
            Link(destination: URL(string: user.html_url)!){
             …
            }
        }
    }
}
```

After data is loaded, *fetch.items* will have items. The view will be notified since *fetch* is an *ObservedObject* type and the List view will show the results.

When you run your app now, it will show the progress view while

data is being loaded.

The entire *ContentView.swift* is reproduced below:

```
import SwiftUI
import URLImage

struct Result: Codable{
    var items: [User]
}

struct User: Codable{
    public var login: String
    public var url: String
    public var avatar_url: String
    public var html_url: String
}

class FetchUsers: ObservableObject {
  @Published var items = [User]()


    init() {

    }


    func loadData(){
        let url = URL(string: "https://api.github.com/search/users?q=greg")!
        URLSession.shared.dataTask(with: url) {(data, response, error) in
            do {
                if let data = data {
                    let decodedData = try JSONDecoder().decode(Result.self, from:
data)
                    DispatchQueue.main.async {
                        self.items = decodedData.items
                    }
                } else {
                    print("No data")
                }
            } catch {
                print("Error: \(error.localizedDescription ?? "unknown error")")
            }
        }.resume()
    }
}

struct ContentView: View {
```

```
    @ObservedObject var fetch = FetchUsers()

    var body: some View {

        if(fetch.items.count == 0){
            ProgressView().onAppear(){
                fetch.loadData()
            }
        }
        else {
            List(fetch.items, id: \.login) { user in
                Link(destination: URL(string: user.html_url)!){
                    HStack(alignment: .top){

                        URLImage(url:URL(string:user.avatar_url)!){ image in
                            image.resizable().frame(width: 50, height: 50)
                        }
                        VStack(alignment: .leading) {
                            Text(user.login)
                            Text("\(user.url)")
                                .font(.system(size: 11))
                                .foregroundColor(Color.gray)
                        }
                    }
                }
            }
        }
    }
}

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

## *Summary*

In the chapter, we learned how to implement a GitHub User Search application by connecting our iOS app to the GitHub RESTful API to get the data of different GitHub users. We processed retrieved JSON data from the API using the *Codable* protocol and *JSONDecoder* .

We also added a third-party package to aid us in displaying images remotely.

In the next chapter, we will build a text classification app using machine learning!

# Chapter 7: Machine Learning with Core ML

In this chapter, we will learn about Core ML and Create ML, Apple ' s library to make machine learning easy for us.

Machine Learning taps on the idea that if you train a machine to do something repeatedly, it will eventually begin to make appropriate choices on its own. For example, if we show a machine thousands of text messages of spam and non-spam messages, we eventually train the machine to be able to look at a message and identify if it's spam or not. The set of messages which a machine is trained on is called a model. You can either create your own model (from a dataset) or use an existing model created by others. In this chapter, we will go through using a dataset provided by Kaggle (the world's largest data science community) that identifies if a text message is spam or not, and then use Create ML to create a model we will use in our app (fig. 1).

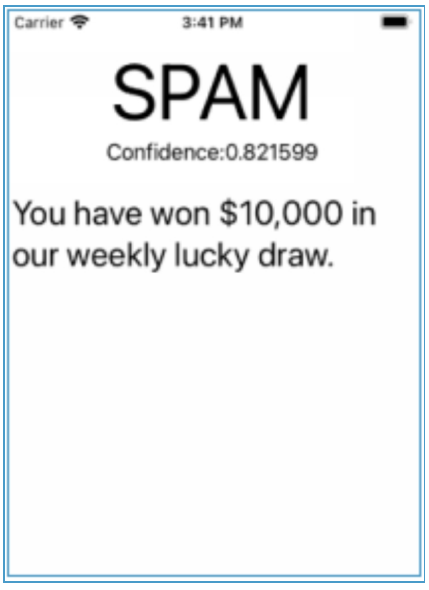


Figure 1

Having gone through this example, you can easily search for more datasets on Kaggle (or else where or even use your own datasets) and create more complex and interesting machine learning projects. For example:

- if a picture contains nudity, filter out such images
- auto answering of email/sms. Gmail provides the function where we can choose to reply with provided answers upon looking at the context of the email, for e.g. ‘Sounds good’, ‘Thanks’, etc.
- classifying newspaper articles into categories e.g. business, technology, politics

All these help in either automation or saving humans a bit of time in making decisions that humans would normally be making. Now, let’s go on to make our spam message recognizer app.

## Downloading the SMS Spam Collection Dataset

Go to https://www.kaggle.com/uciml/sms-spam-collection-dataset (fig. 2), or just Google ‘ Kaggle SMS Spam Collection Dataset ’

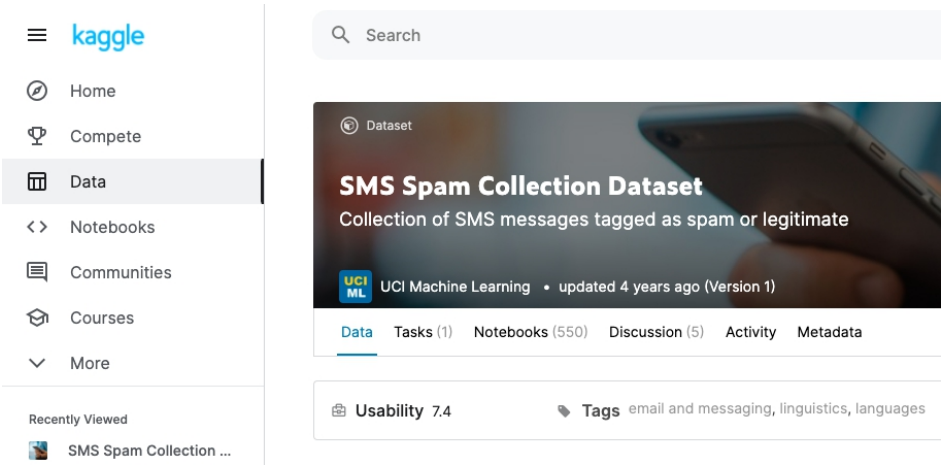



Figure 2

I have chosen this dataset also because it contains around 5,000 plus rows of data, which will be quite manageable for your laptop to train the model. (if it ' s around hundreds of thousands or millions of data, your laptop will literally be on fire)

Download the *spam.csv* file. The dataset has two columns:

' v1 ' which contains the label (ham or spam)
' v2 ' contains the raw text.

For e.g., if I open the csv file in Excel (fig. 3):

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| 1 | v1 | v2 | | | | |
| 2 | ham | Go until jurong point, crazy.. Available only in bugis n great world la | | | | |
| 3 | ham | Ok lar... Joking wif u oni... | | | | |
| 4 | spam | Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005. T | | | | |
| 5 | ham | U dun say so early hor... U c already then say... | | | | |
| 6 | ham | Nah I don't think he goes to usf, he lives around here though | | | | |
| 7 | spam | FreeMsg Hey there darling it's been 3 week's now and no word back! | | | | |

Figure 3

In column *v1* , spam messages will be labelled as ' spam ' , and non-spam messages as ' ham ' .

CreateML can take JSON, CSV and text files as training data. But importantly, we need to use the column headers ' label ' and ' text ' explicitly.

So, in the file, change column headers ‘ v1 ’ and ‘ v2 ’ to ‘ label ’ and ‘ text ’ (fig. 4). Save your file.

| | A | B | C |
|---|---|---|---|
| 1 | label | text | |
| 2 | ham | Go until jurong point, crazy.. | |
| 3 | ham | Ok lar... Joking wif u oni... | |

Figure 4

# Training our Model

Next, let’s create and train our model. We can choose to do this in code via Playground. But included in Xcode 11 and up, we have the CreateML application to help us do so.

So open the CreateML application (fig. 5).

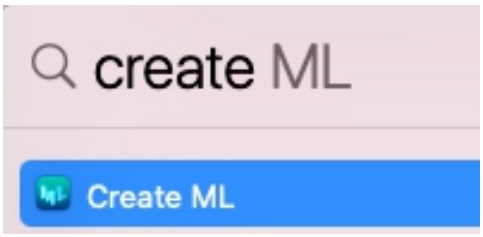



Figure 5

You get a wide range of project types (image classifiers, text classifiers, object detection – fig. 6) .

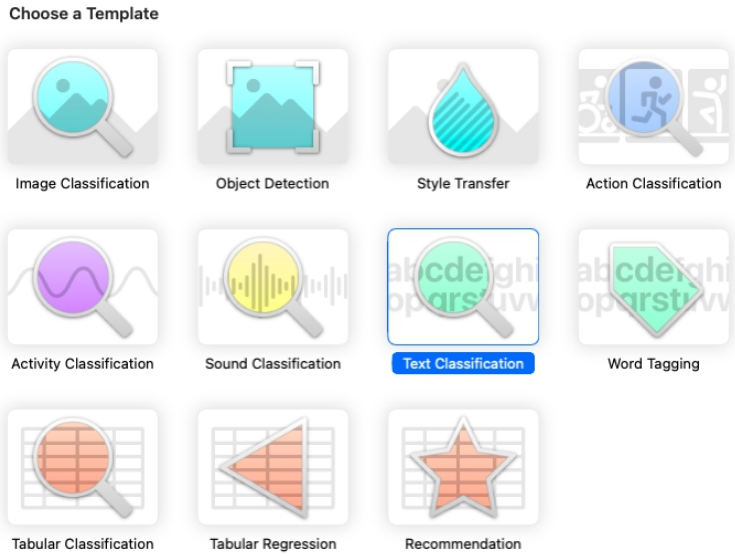



Figure 6

Choose ‘Text Classification’ and name your project (fig. 7).

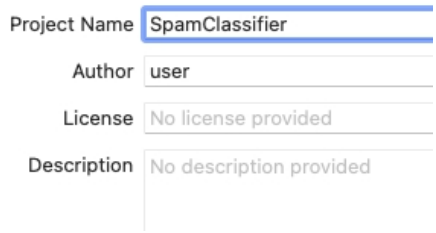



Figure 7

I have named my project ‘SpamClassifier’. Hit ‘Next’ and select a location for your project folder.

Next, select the *spam.csv* for ‘Training Data’ and leave ‘Validation Data’ as automatic (fig. 8). This will auto split our data set into ‘training data’ and ‘testing data’. Alternatively, you can create a smaller csv file using texts you didn’t use in the training data to act as testing data.

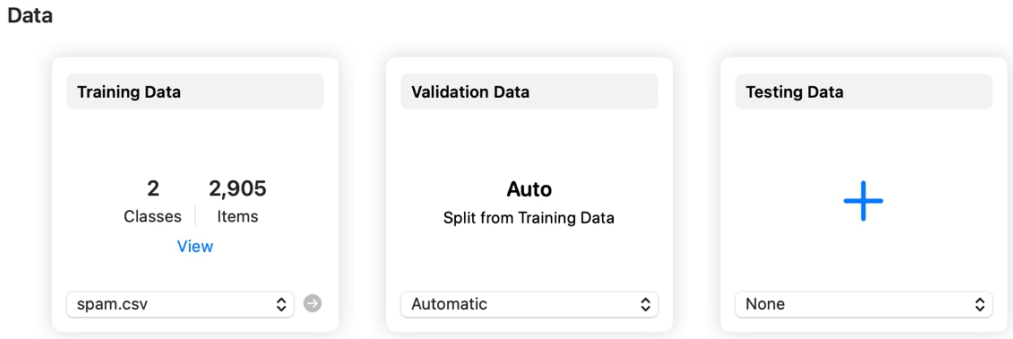



Figure 8

CreateML will correctly identify that we have two classes, ‘ham’ and ‘spam’.

We typically split a dataset into two sets. A training set and a testing set. 80% for training and 20% for testing.
You train the model using the training set.
You test the model using the testing set.
When we choose ‘Validation Data’ as automatic, CreateML does the above automatically for us.

Next, we have to select which algorithm to train our model as shown

in fig. 9.


Figure 9

Different algorithms are best suited to different uses. Which algorithm to best use for which case is beyond the scope of this book. But for our simple example, we will use the default ‘Maximum entropy’.

Next, hit the ‘Train’ button 

in the top left corner. This will train the model with the data and the algorithm we selected (fig. 10).


Figure 10

This might take a while, but you can see the accuracy improving as it iterates through the data more times.

When the training is done, you can try out the model in the ‘Preview’ tab. Try typing in a message and see its classification and how confident it is (fig. 11).

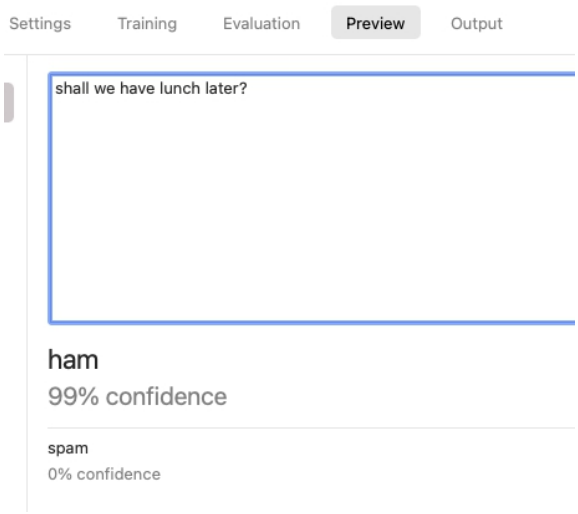



Figure 11

And if I change my message to (fig. 12):

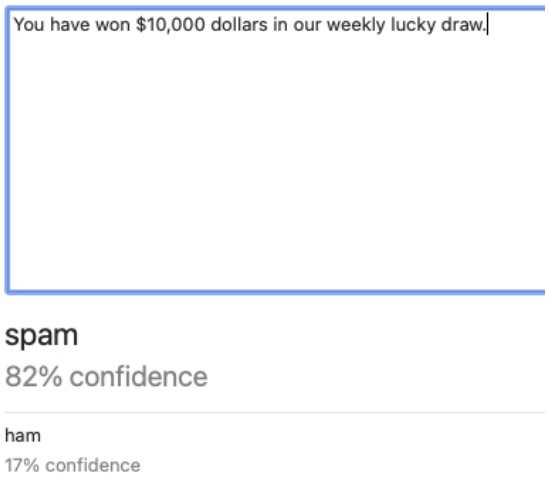



Figure 12

See how straightforward it is to train a text-classifier with CreateML? Having created our model, let's go ahead and build our app.

## Building the App

Start Xcode and create a new App project called 'SpamClassifier'. You can uncheck the 'Core Data' check box since we will not be using it.

When we create a model using CreateML, it outputs a *.mlmodel* file which includes a class that represents the data model. We interact

with the model through this class. To include the model in our project, in the ‘Output’ tab of CreateML, click on the Xcode icon (fig. 13).

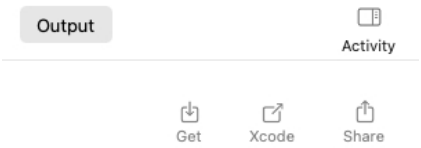



Figure 13

There will be a popup showing the details of the generated CoreML class (fig. 14).

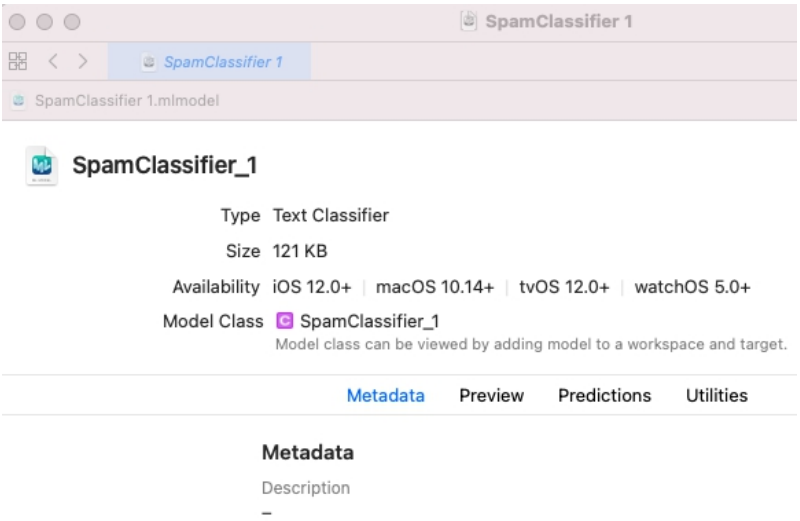



Figure 14

Download and drag the *.mlmodel* file into the project files to add the model class to our app (fig. 15).

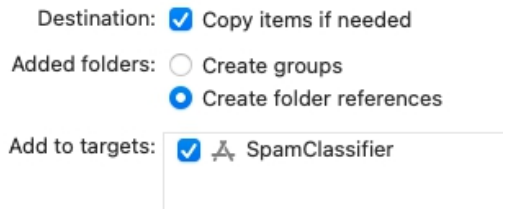



Figure 15

In the popup, check the ‘Copy items if needed’ and ‘Add to targets’ checkbox (fig. 16).

*SpamClassifier 1.mlmodel* will then be added to your project where

you can see its details.

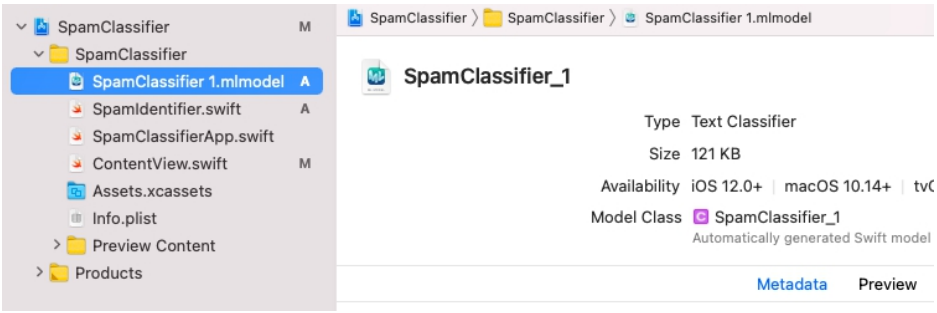



Figure 16

The more important part is the section ‘Predictions’ (fig. 17). It contains the input/output of the model. For example, the model expects a string as input and will then output a *label* of *String* type i.e. ‘ham’ or ‘spam’. For an image classifier model, it would typically expect an image as input and output a classification string.

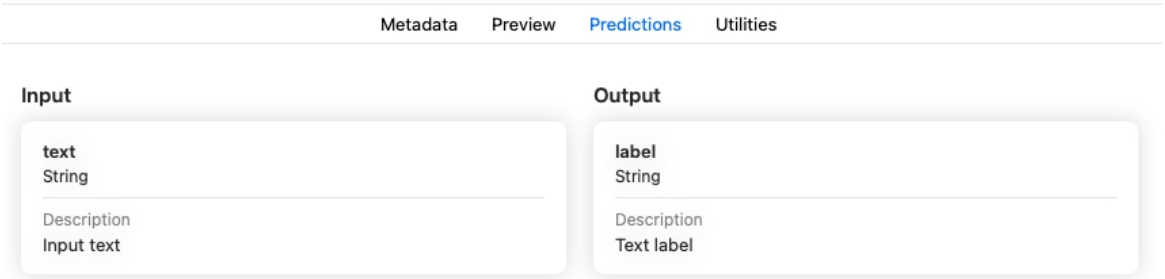



Figure 17

Next, create a class *SpamIdentifier.swift* to handle our model. Fill it with the following code:

```
import Foundation
import SwiftUI
import CoreML
import NaturalLanguage

class SpamIdentifier : ObservableObject{
    @Published var prediction = ""
    @Published var confidence = 0.0

    var model = MLModel()
    var spamPredictor = NLModel()


    init(){
```

```
        do{
            self.spamPredictor = try NLModel(mlModel: SpamClassifier_1().model)

        }catch{
            print(error)
        }
    }

    func predict(_ text: String){
        self.prediction = spamPredictor.predictedLabel(for: text) ?? ""
        let predictionSet = spamPredictor.predictedLabelHypotheses(for: text,
maximumCount: 1)
        self.confidence = predictionSet[prediction] ?? 0.0
    }
}
```

## *Code Explanation*

```
import Foundation
import SwiftUI
import CoreML
import NaturalLanguage
```

Other than *Foundation* and *SwiftUI* , we have to import *CoreML* and *NaturalLanguage* .

```
class SpamIdentifier : ObservableObject{
    @Published var prediction = ""
    @Published var confidence = 0.0
```

*SpamIdentifier* conforms to the *ObservableObject* protocol as we will publish the output of the predictions for the views to update themselves with.

We have two published properties to update in the views. The prediction (‘ham’, ‘spam’) and the confidence of the prediction.

```
    init(){
        do{
            self.spamPredictor = try NLModel(mlModel: SpamClassifier_1().model)

        }catch{
            print(error)
        }
    }
```

In *init()* , we instantiate an *NLModel* instance with the SpamClassifier model class. We use *NLModel* to make predictions with its built-in methods.

```
    func predict(_ text: String){
        self.prediction = spamPredictor.predictedLabel(for: text) ?? ""
```

*predict()* takes the text message we want to classify and uses our *NLModel (spamPredictor)* to make a prediction.

```
let predictionSet = spamPredictor.predictedLabelHypotheses(for: text,
maximumCount: 1)
```

To get the confidence of the prediction, we use predictedLabelHypotheses . It returns a *String* : *Double* dictionary containing the label (as key) and the confidence for its label.

```
        self.confidence = predictionSet[prediction] ?? 0.0
    }
```

Thus, to extract the confidence, we supply *prediction* as key.

## Linking to our View

We will create our view and link it to *SpamIdentifier* . In *ContentView* , fill it with the below codes:

```
struct ContentView: View {


    @ObservedObject var identifier = SpamIdentifier()
    @State private var input = "Enter Message"

    var body: some View {
        VStack(alignment: .center) {
            Spacer()
            Text(self.identifier.prediction=="spam" ? "SPAM":"NOT SPAM")
                    .font(.system(size: 60))


            Text("Confidence:\(self.identifier.confidence)")

            TextEditor(text: $input)
                .font(.title)
                .onChange(of: input){_ in
                    if input.last == " "{
```

```
                    self.identifier.predict(input)
                }
            }

            Spacer()
        }
    }
}
```

## *Code Explanation*

```
@ObservedObject var identifier = SpamIdentifier()
@State private var input = "Enter Message"
```

We have an *ObservedObject* instance of *SpamIdentifier* class to make predictions and listen to the outcome.
We have a state variable *input* to allow user to type in some text to a *TextEditor* and then call *identifier.predict* ().

```
Text(self.identifier.prediction=="spam" ? "SPAM":"NOT SPAM")
    .font(.system(size: 60) )


Text("Confidence:\(self.identifier.confidence)")
```

If the prediction returns “ spam ” , we return “ SPAM ” and “ NOT SPAM ” otherwise.
At the same time, we also display the confidence of the prediction.

```
TextEditor(text: $input)
    .font(.title)
    .onChange(of: input){_ in
        if input.last == " "{
            self.identifier.predict(input)
        }
    }
```

We use a *TextEditor* view to allow the user to text in a message that spans multiple lines. Rather than a user pressing a button once they ’ ve finished typing, we call *predict* each time a user finishes typing a word, i.e. the latest character of the input is a space. We use the *.onChange* modifier of *TextEditor* to do so. The *.onChange* modifier lets us listen to state changes and perform actions on a view accordingly.

## *Running Your App*

Now, let ' s run our app and try entering some messages (fig. 18):


Figure 18

It works pretty well right?

In the above sections, we learned about Create ML to create a model from a data set. We then applied the model in our spam classification app. Go on to try different data sets and explore the different Machine Learning apps you can possibly create!

# Chapter 8: C.R.U.D. Notes App with Firebase/Firestore

In this chapter, we will cover how to implement full C.R.U.D. operations in SwiftUI with a backend server. A typical iOS application architecture consists of the server side and client side. Typically, an iOS app serves as the client side. It may talk to a backend server to get or save data via RESTful http services built using server-side frameworks like ASP.NET, Node.js and Ruby on Rails. We had explored this when we obtained data from the GitHub server in chapter six.

Building the server side however is often time-consuming and not within the scope of this course. In this chapter, however, we will explore using Firebase as our backend server. Firebase is Google ' s real-time database which offers a powerful backend platform for building fast and scalable real-time apps.

With Firebase, we don't have to write server-side code or design relational databases. Firebase provides us with a real-time, fast and scalable database in the cloud and also a library to talk to this database. This allows us to focus on building our application according to requirements rather than debugging server-side code.

Firebase stores our data in objects called *documents* which are grouped into *collections* . Within these collections, you can have more sub-collections up to hundred levels deep. You can then query for the *documents* with *where* clauses.

This chapter aims to illustrate create, read, update and delete functionality with SwiftUI and Firebase integrated so that you can go on and create a fully working app.

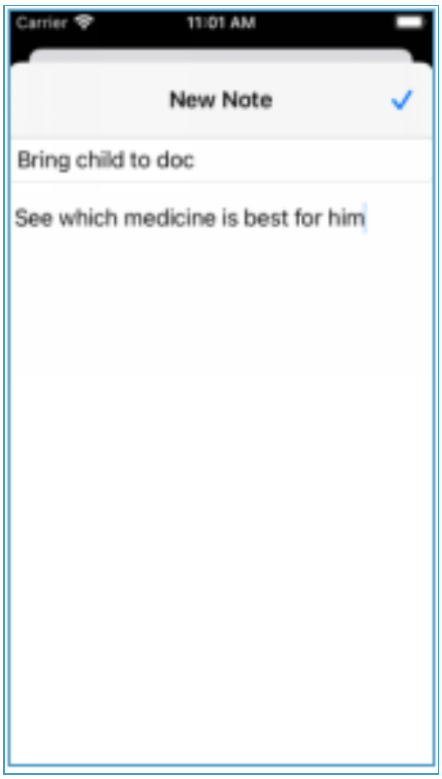
## *More on Firebase*

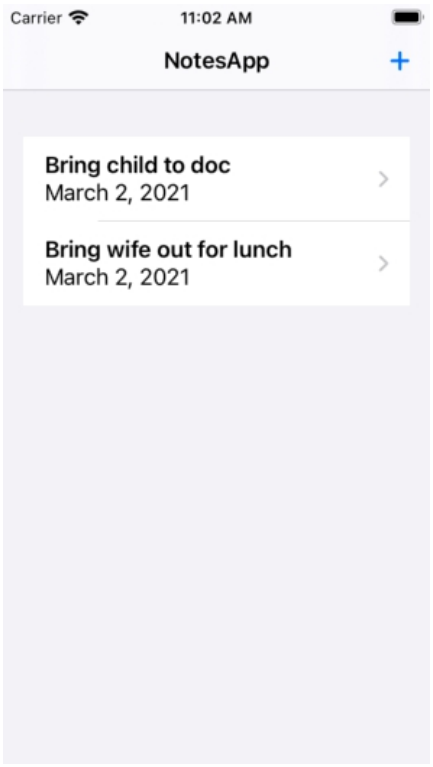
Firebase is a real-time database. which means that as data is modified, all connected clients are automatically refreshed in an optimized way. If one user adds a new item either through a browser or a mobile app, another user (again either through a browser or mobile app) sees the addition in real-time without refreshing the page. Firebase of course provides more than just a real-time database. It offers other services like Authentication, cloud messaging, disk space, hosting analytics. You not only can develop iOS apps with Firebase as backend but also Android and web applications.

Firebase offers two types of databases. The realtime database is the original one, and Cloud Firestore is the newer and more powerful implementation. In this chapter, we will use Cloud Firestore. It allows our app to save data in the repository, and also sends events when it is updated by another client, allowing our app to react to these changes seamlessly. This asynchronous feature works well with SwiftUI.

## *Our App*

We will implement a simplified version of the default Notes app in iOS (fig. 1a).



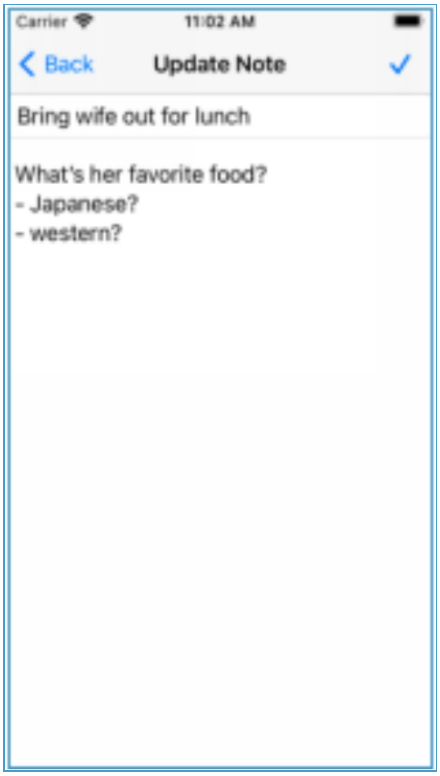




Figure 1a

We will save our notes in a Firestore collection.

## *Setting our Environment*

We will first import the Firebase framework by installing Cocoapods (https://cocoapods.org/), a standard dependency manager for iOS.

Install Cocoapods in the Terminal with the command:

```
sudo gem install cocoapods
```

Next, in Xcode, create a new SwiftUI app called *NotesApp* . In the next section, we will go through the setting up of Firebase.

## Using Firebase

We can use Firebase features for free and only pay when our application grows bigger. You can choose between a subscription-based or ‘pay as you use’ model. Find out more at firebase.google.com/pricing.

Before adding Firebase to our iOS project, we need to first create a Firebase account. Go to firebase.google.com and sign in with your Google account.

In the Firebase console (https://console.firebase.google.com/), click on ‘Add Project’ (figure 1b)

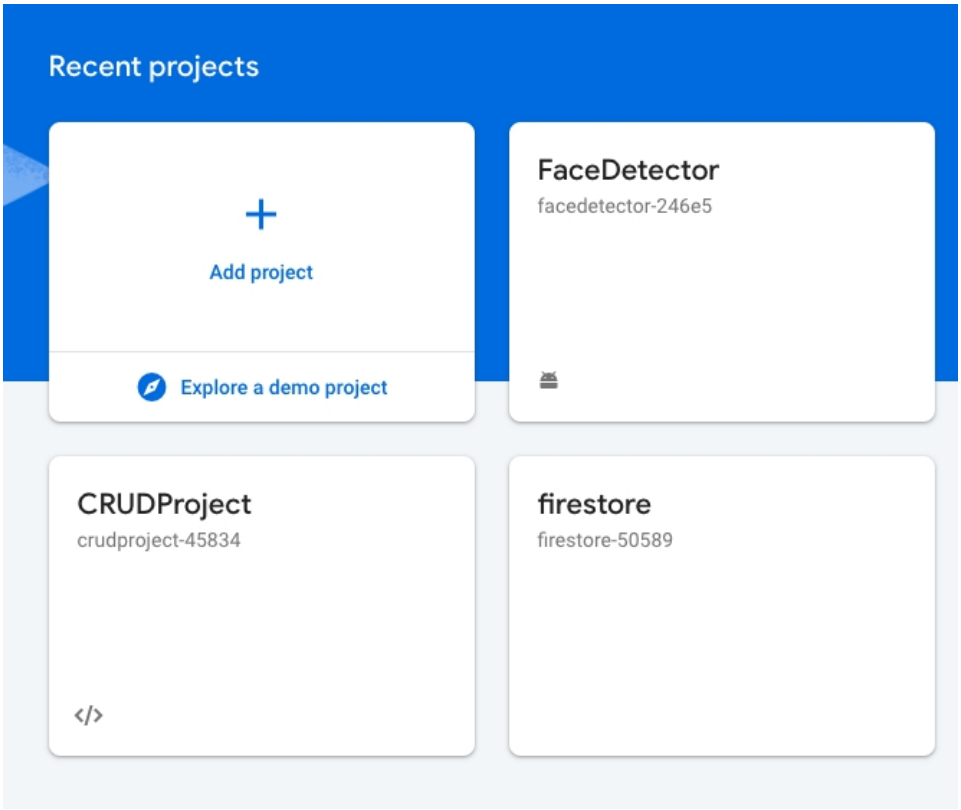


figure 1b

Fill in the project name, optionally enable Google Analytics for your project and click ‘Create Project’.

When your project is created, in the project page, under ‘’ Get started by adding Firebase to your app ’ , click on the ‘ iOS ’ icon to add firebase to our iOS app (figure 2).

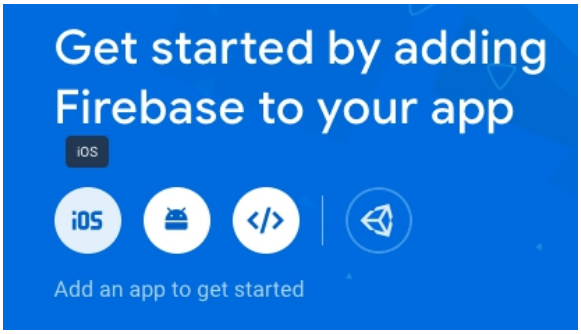


figure 2

To prepare the Firebase configuration file, register the app with your iOS bundle ID e.g. (fig. 3):

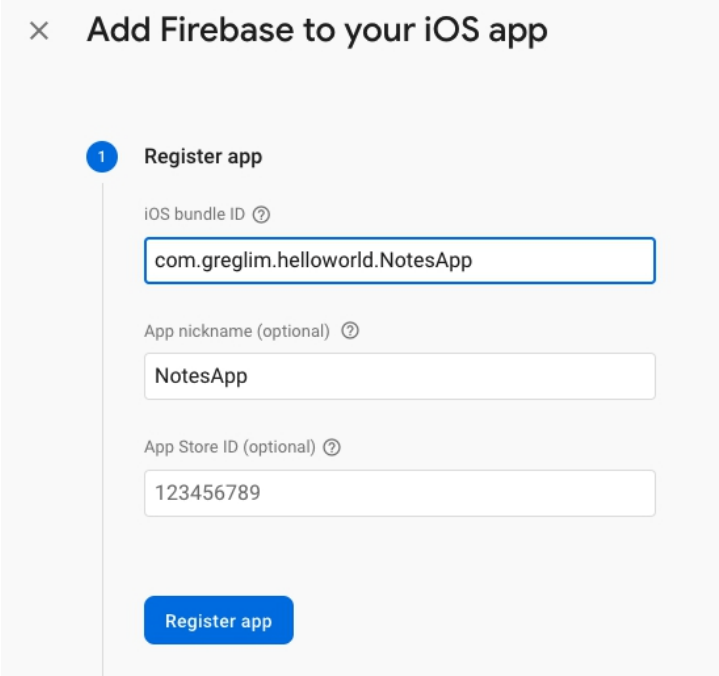



Figure 3

Download the generated *GoogleService-Info.plist* (fig. 4)

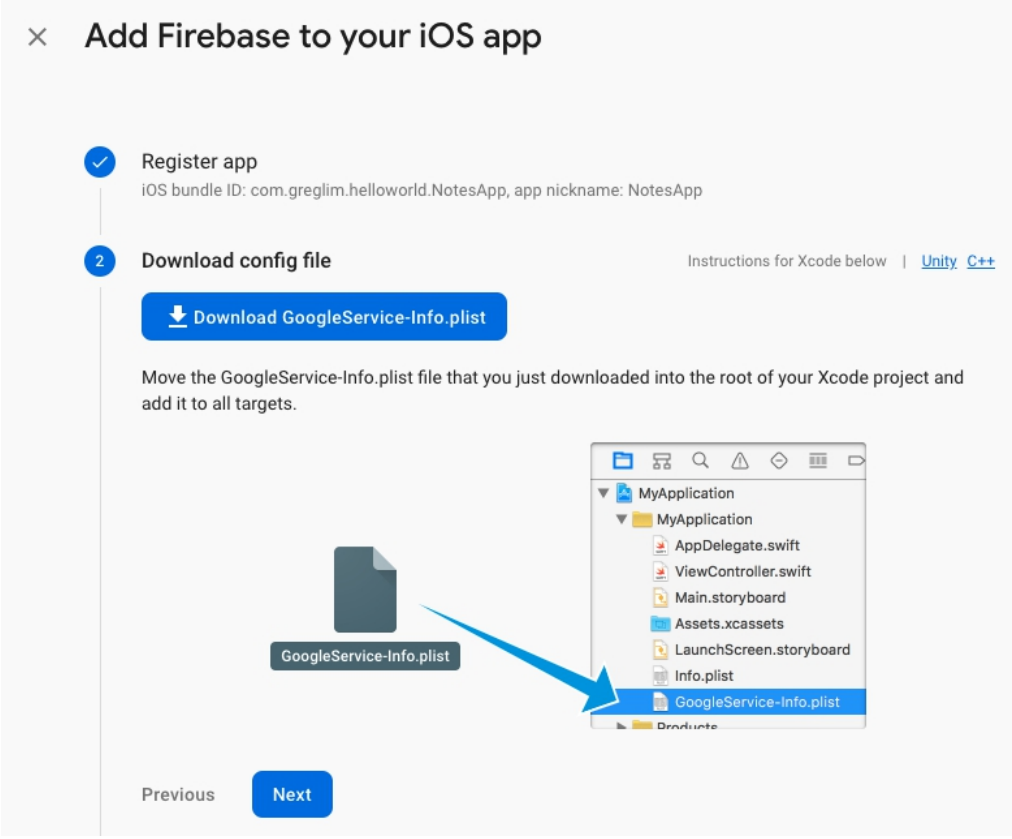



Figure 4

Drag the downloaded *GoogleService-Info.plist* into the iOS project and ensure that you select ‘Copy items if needed’ (fig. 5).

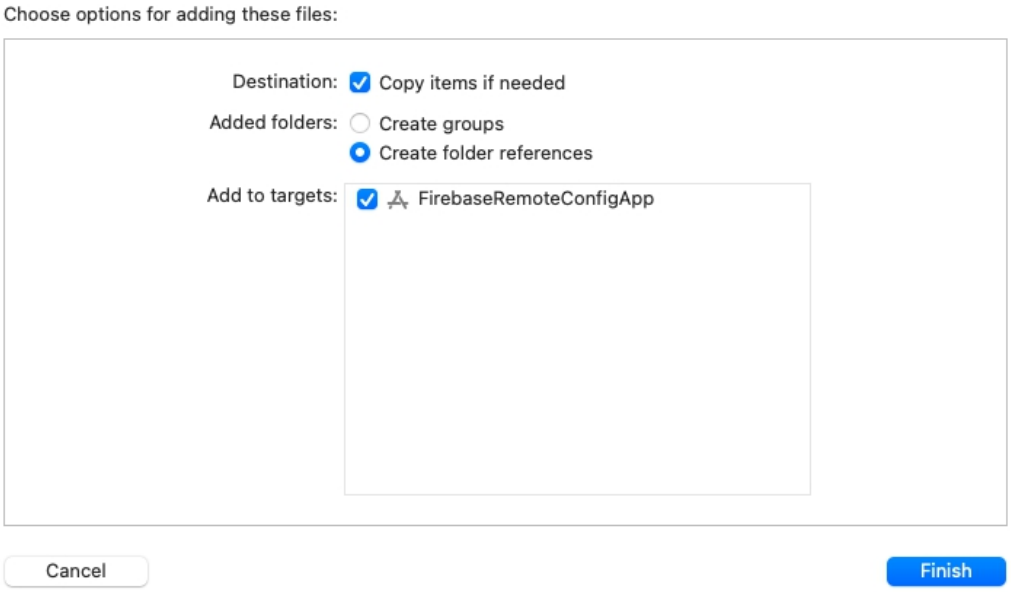



Figure 5

The *.plist* file should be in your project (fig. 6):

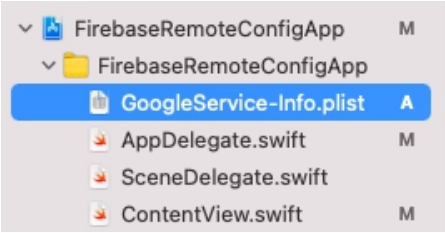



Figure 6

## *Installing Cocoapods Dependencies*

Next, we have to create the Cocoapods configuration by going to our project folder in Terminal and running the following command:

```
pod init
```

This creates a *Podfile* file where we add dependencies. Open that file in an editor and add the dependencies as shown in **bold** :

```
target 'FirebaseRemoteConfigApp' do
  use_frameworks!
```

```
  pod 'Firebase/Firestore'
  pod 'FirebaseFirestoreSwift'
end
```

Now, install the pods by running in Terminal:

```
pod install
```

Cocoapods will install the dependencies and create a workspace that contains our project and the frameworks we just installed. Being a workspace, we must now open *NotesApp.xcworkspace* instead of *NotesApp.xcodeproj* .

Next, open the app in Xcode using the *.xcworkspace* file. In *AppDelegate* , initialize Firebase by adding the below:

```
import UIKit
import Firebase

@main
class AppDelegate: UIResponder, UIApplicationDelegate {

    func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
        FirebaseApp.configure()
        return true
    }
    …
}
```

# Working with a Firebase Database

Now let’s look at our Firebase database. Go to console.firebase.google.com. Click on your project, and under ‘Build’, click on ‘Cloud Firestore’, and click on ‘**Create database** ’ as shown in fig. 7.

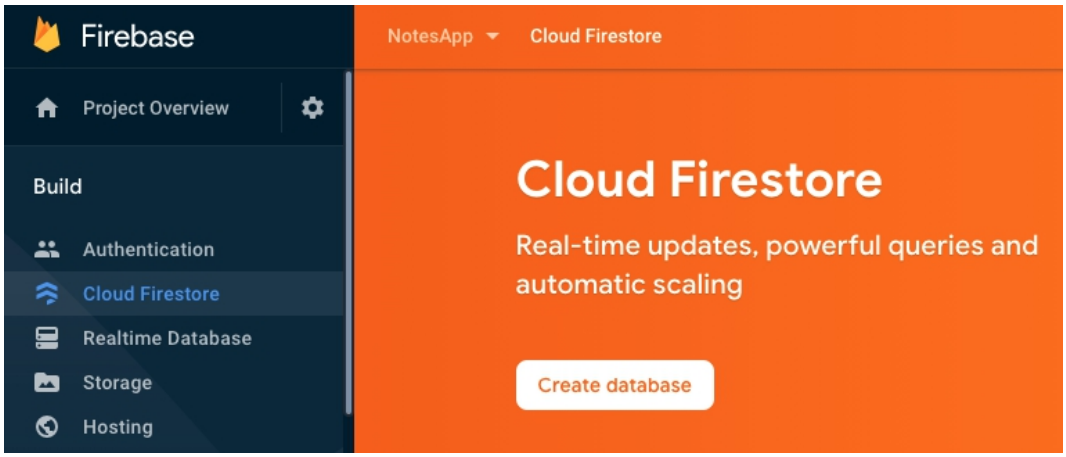



figure 7

Because Firebase's security rules are beyond the scope of this book, in the next screen shown below (fig. 8), choose 'test mode' and then click 'Next'.

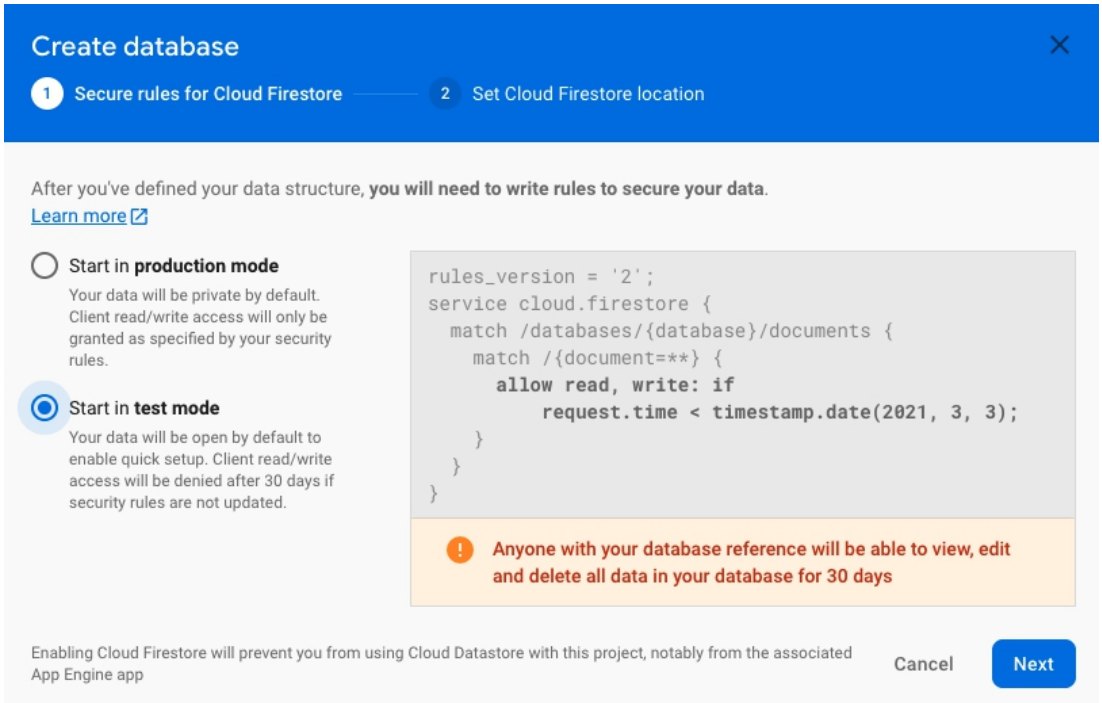



Figure 8

Choose the default settings given for 'Cloud Firestore location', and we will have created the database.

## Beginning our NotesApp Project

## *Note.swift*

First, we will create a *Note* struct. Create *Note.swift* in your project with the following code.

```
import Foundation

struct Note: Identifiable{
    let id: String
    let title: String
    let date: Date
    let body: String
}
```

Our note consists of a title, body and date. Because we will show the notes in a *List* View, we make it *Identifiable* .

## *NotesRepository.swift*

Next, we create a *NotesRepository* class which will hold the functions to create, delete and update a note. Create *NotesRepository.swift* with the following method headers:

```
import Foundation
import Firebase

class NotesRepository : ObservableObject{

    private let db = Firestore.firestore()

    @Published
    var notes: [Note] = []

    init(){
    }

    func newNote(title: String, date: Date, body: String){
    }

    func updateNote(id: String, title: String, date: Date, body: String){
    }
```

```
    func remove(at index:Int){
    }
}
```

## *Code Explanation*

```
    private let db = Firestore.firestore()
```

We have our Firestore instance *db* . Previously, we have copied *GoogleService-Info.plist* into our project and called *Firebase.configure()* . That configures our Firestore instance to know which database to connect to.

```
    @Published
    var notes: [Note] = []
```

With the *@Published* prefix, any view bound to *notes* will automatically be notified when it changes.

## *loadAll()*

After every *newNote* , *updateNote* or *remove* function call, we have to load all the notes again to reflect the change. Let's implement a *loadAll* function to load all the notes:

```
class NotesRepository : ObservableObject{
        …
    private func loadAll(){
        db.collection("notes").getDocuments{(snapshot, error) in
            if let error = error{
                print(error)
                return
            }
            guard let documents = snapshot?.documents else{
                return
            }
            self.notes = documents.compactMap{
                document in let data = document.data()
                guard let title = data["title"] as? String,
                        let timestamp = data["date"] as? Timestamp,
                        let body = data["body"] as? String else{
                    return nil
```

```
            }
            return Note(id: document.documentID,
                        title: title,
                        date: timestamp.dateValue(),
                        body: body)
        }
    }
  }
}
```

## *Code Explanation*

```
db.collection("notes").getDocuments{(snapshot, error) in
    if let error = error{
        print(error)
        return
    }
```

We access the “notes” collection and fetch all the *note* documents in it with *getDocuments()* . If there’s any error, we will print it and exit.

```
    guard let documents = snapshot?.documents else{
        return
    }
```

We unwrap the optional with *guard let* and if it finds *nil* , we exit the function.

```
    self.notes = documents.compactMap{
        document in let data = document.data()
        guard let title = data["title"] as? String,
              let timestamp = data["date"] as? Timestamp,
              let body = data["body"] as? String else{
            return nil
        }
        return Note(id: document.documentID,
                    title: title,
                    date: timestamp.dateValue(),
                    body: body)
    }
```

*compactMap* let us *transform* the elements in the array. In the transformation, we retrieve the title, timestamp and body from the

Firestore document and return a *Note* struct instance. Any nil values get discarded.

Now, let's add the *loadAll()* function call in the init, create, delete and update operations:

```
class NotesRepository : ObservableObject{
        …
    init(){
        loadAll()
    }

    func newNote(title: String, date: Date, body: String){
        loadAll()
    }

    func updateNote(id: String, title: String, date: Date, body: String){
        loadAll()
    }

    func remove(at index:Int){
        loadAll()
    }
        …
}
```

## Creating a New Note

In *newNote* , we call the *addDocument()* function with the data to create a new Note in Firestore.

```
class NotesRepository : ObservableObject{
        …
    func newNote(title: String, date: Date, body: String){
        db.collection("notes").addDocument(data: [
            "title": title,
            "date": date,
            "body": body,
        ])
        loadAll()
    }
```

```
}
```

To be able to add an object to firebase, we need to have write permission. Earlier on, we had set this to be *true* in the firebase console when we selected the ‘test mode’ option.

## *Deleting a Note*

To delete a note, we get a reference to the document using the collection path ‘ notes ’ and the document ’ s id, and call *delete()* .

```
class NotesRepository : ObservableObject{
    …
    func remove(at index:Int){
        let noteToDelete = notes[index]
        db.collection("notes").document(noteToDelete.id).delete()
        loadAll()
   }
   …
}
```

## *Updating a Note*

Like *delete* , updating a document in Firestore requires knowing its path and document id.

```
class NotesRepository : ObservableObject{
        …
    func updateNote(id: String, title: String, date: Date, body: String){
        db.collection("notes").document(id).updateData([
            "title": title,
            "date": date,
            "body": body
        ]){ (error) in
            if error != nil{
              print("Error")
            }
            else {
              print("succesfully updated" )
            }
        }
        loadAll()
```

```
    }
            ...
}
```

With this, we have completed our *NotesRepository* . In the next section, we will focus on creating the Views of the app.

## Creating the Views for NoteApp

We will have *ContentView* to list the notes, *NewNote* to add a note and *ShowNote* to show and edit it. Though typing out code is a good way to learn, if you fell it too cumbersome, refer to my source codes at *www.greglim.net/swiftui* .

## *ContentView*

In *ContentView* , fill it with the following:

```
import SwiftUI

struct ContentView: View {
    static let taskDateFormat:DateFormatter={
        let formatter = DateFormatter()
         formatter.dateStyle = .long
         return formatter
    }()

    @ObservedObject
    var repository:NotesRepository = NotesRepository()
    @State
    var isNewNotePresented = false


    var body: some View {
        NavigationView{
            List {
                ForEach(repository.notes){ note in
                    NavigationLink(destination: ShowNote(
                                    id: note.id,
                                    title: note.title,
                                    bodyText: note.body,
                                    repository: repository)){
                        VStack(alignment: .leading){
```

```
                        Text(note.title).font(.headline)
                        Text("\(note.date, formatter:Self.taskDateFormat)")
                    }
                }
            }
            .onDelete{ indexSet in
                if let index = indexSet.first {
                    repository.remove(at: index)
                }
            }
        }
        .navigationBarTitle("NotesApp",displayMode: .inline)
        .navigationBarItems(trailing:
            Button{
                isNewNotePresented.toggle()
            } label:{
                Image(systemName: "plus").font(.headline)
            }
        )
        .sheet(isPresented: $isNewNotePresented){
            NewNote(isNewNotePresented:      $isNewNotePresented,repository:
repository)
        }
    }
  }
}
```

## *Code Explanation*

```
struct ContentView: View {
    static let taskDateFormat:DateFormatter={
       let formatter = DateFormatter()
        formatter.dateStyle = .long
        return formatte r
    }()
          …
```

We first have a *DateFormatter* instance to format our note ’ s date.

```
    @ObservedObject
    var repository:NotesRepository = NotesRepository()
```

We then watch the *repository* property for changes.

```
var body: some View {
    NavigationView{
        List{
            ForEach(repository.notes){ note in
                NavigationLink(destination: ShowNote(
                                id: note.id,
                                title: note.title,
                                bodyText: note.body,
                                repository: repository)){
```

In *body* , we have a *NavigationView* with a *List* to present the notes. When a user clicks on a row, they will be directed to a *ShowNote* view to show the selected note. We will implement *ShowNote* later.

```
            List{
        …
            }
            .onDelete{ indexSet in
                if let index = indexSet.first {
                    repository.remove(at: index)
                }
            }
        }
```

We add the *.onDelete()* modifier to remove a note.

```
            List{
                        …
            }
            .navigationBarTitle("NotesApp",displayMode: .inline )
            .navigationBarItems(trailing:
                Button{
                    isNewNotePresented.toggle()
                } label:{
                    Image(systemName: "plus").font(.headline)
                }
            )
            .sheet(isPresented: $isNewNotePresented){
                NewNote(isNewNotePresented:      $isNewNotePresented,repository:
repository)
            }
```

We add a ‘ plus ’ button to the navigation bar that toggles the

*isNewNotePresented* state variable, which then presents the *NewNote* view. We implement *NewNote* in the next section.

## *NewNote View*

Create a new SwiftUI file called *NewNote.swift* with the following code:

```
import SwiftUI

struct NewNote: View{
    @State private var title: String = ""
    @State private var bodyText: String = ""

    @Binding var isNewNotePresented: Bool
    var repository: NotesRepository

    var body: some View{
        NavigationView{
            VStack(){
                TextField("Title",text: $title)
                    .textFieldStyle(RoundedBorderTextFieldStyle())
                TextEditor(text: $bodyText)
                    .textFieldStyle(RoundedBorderTextFieldStyle())
            }
            .navigationBarTitle("New Note",displayMode: .inline)
            .navigationBarItems(trailing: Button {
                repository.newNote(title: title, date: Date(), body: bodyText)
                isNewNotePresented = false
            } label:{
                Image(systemName: "checkmark")
                    .font(.headline)
            }.disabled(title.isEmpty))
        }
    }
}
```

## *Code Explanation*

```
            VStack(){
                TextField("Title",text: $title)
                    .textFieldStyle(RoundedBorderTextFieldStyle())
```

```
        TextEditor(text: $bodyText)
            .textFieldStyle(RoundedBorderTextFieldStyle())
    }
```

Our *NewNote* view will have a *VStack* containing a *TextField* to add a title and a *TextEditor* (supports multi-line text) to add the note ' s body.

```
NavigationView{
    VStack(){
                …
    }
        …
}
```

We then encapsulate the *VStack* in a *NavigationView* .

```
NavigationView{
    VStack(){
                …
    }
        .navigationBarTitle("New Note",displayMode: .inline)
    .navigationBarItems(trailing: Button{
        repository.newNote(title: title, date: Date(), body: bodyText)
        isNewNotePresented = fals e
    } label:{
        Image(systemName: "checkmark")
            .font(.headline)
    }.disabled(title.isEmpty))
}
```

The *NavigationView* has a checkmark bar button item to add our note. The checkmark button is disabled if the *title* field is empty. Note also that we set *isNewNotePresented* to false after we call *repository.newNote()* to dismiss the *NewNote* View and return to *ContentView* .

## *ShowNote View*

Next, we implement the *ShowNote* View to show existing notes to the user for them to edit. Create a new SwiftUI file called *ShowNote.swift* . *ShowNote.swift* will contain similar code to

*NewNote.swift* . We can in fact combine them and add conditional logic to differentiate between adding a note and editing a note. But for simplicity, we will just create a separate *ShowNote* View. Fill in *ShowNote.swift* with the following code:

```
import SwiftUI

struct ShowNote: View {
    @Environment(\.presentationMode) var presentationMode: Binding<PresentationMode>
    var id: String = ""
    @State var title: String = ""
    @State var bodyText: String = ""

    var repository: NotesRepository

    var body: some View {
        VStack(){
            TextField("Title",text:$title)
                        .textFieldStyle(RoundedBorderTextFieldStyle())
            TextEditor(text: $bodyText)
                        .textFieldStyle(RoundedBorderTextFieldStyle() )
        }
        .navigationBarTitle("Update Note",displayMode: .inline)
        .navigationBarItems(trailing: Button{
            repository.updateNote(id: id, title: title, date: Date(), body: bodyText)
            self.presentationMode.wrappedValue.dismiss()
        } label:{
            Image(systemName: "checkmark")
                .font(.headline)
        }.disabled(title.isEmpty))
    }
}
```

## *Code Explanation*

Remember that *ShowNote* is instantiated in *ContentView* with the inputs id, title, *bodyText* and repository.

```
    var body: some View {
        NavigationView{
            List{
```

```
ForEach(repository.notes){ note in
    NavigationLink(destination: ShowNote(
                        id: note.id,
                        title: note.title,
                        bodyText: note.body,
                        repository: repository) ){
```

We retrieve these passed-in values by declaring the below variables:

```
struct ShowNote: View {
        …
    var id: String = ""
    @State var title: String = ""
    @State var bodyText: String = ""

      var repository: NotesRepository
```

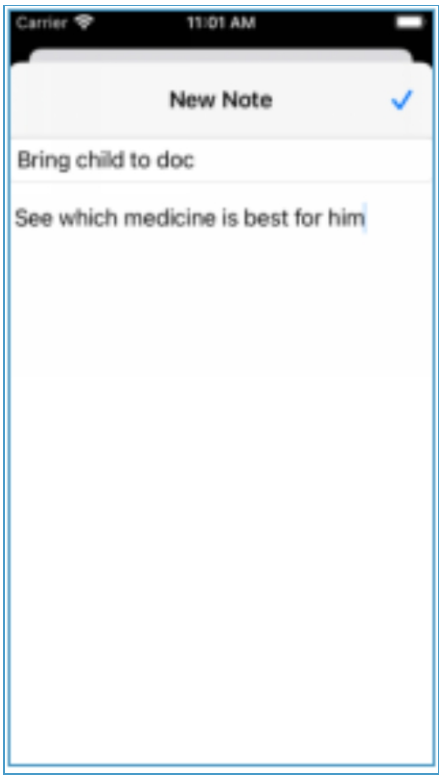
For the rest of the code, it should be similar to *NewNote* , except that in the checkmark button, we call *repository.updateNote()* :

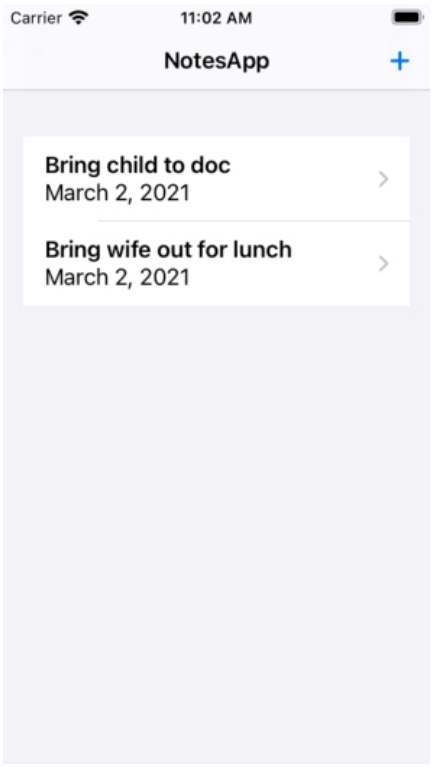
```
.navigationBarItems(trailing: Button{
  repository.updateNote(id: id, title: title, date: Date(), body: bodyText)
  self.presentationMode.wrappedValue.dismiss()
}
```

After calling *updateNote()* , we dismiss the *ShowNote* View with *self.presentationMode.wrappedValue.dismiss()* declared in the beginning of the struct:

```
struct ShowNote: View {
    @Environment(\.presentationMode) var presentationMode:
Binding<PresentationMode>
```

With this, our app is now complete (fig. 9a).



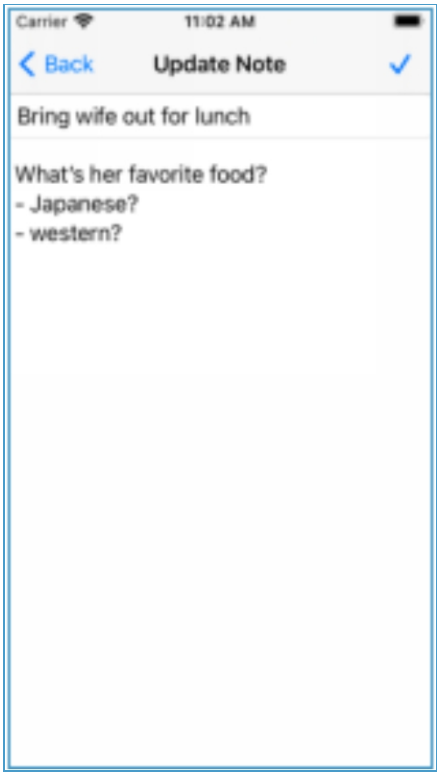




Figure 9a

Try it out by adding/editing/deleting notes! After that, go to the Firebase console where you can see the changes made to the data

in the database (fig. 9b).

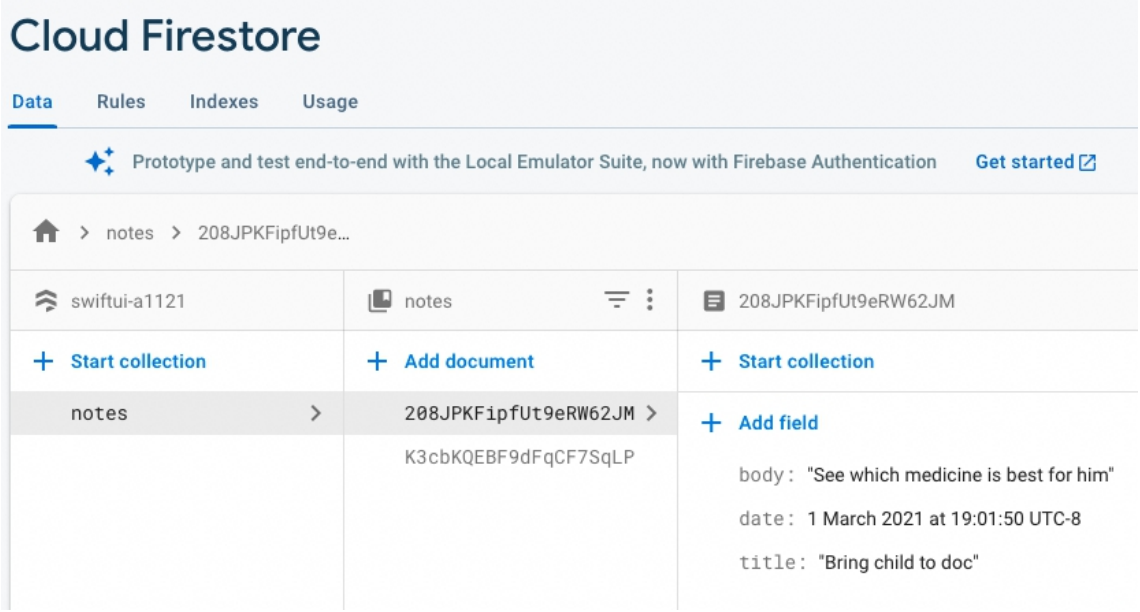



Figure 9b

As you can see, the data is stored as a collection (‘notes’) of documents. Each document, representing a note, has an id (a unique key generated by Firebase), body, date and title. This is modelled after the *Note* struct we have defined. In *NotesApp* , we experimented with String and Date types. But you can experiment with other primitive types like Boolean, number and even complex objects.

I hope this chapter shows how you fit Firestore into a SwiftUI app

## *Summary*

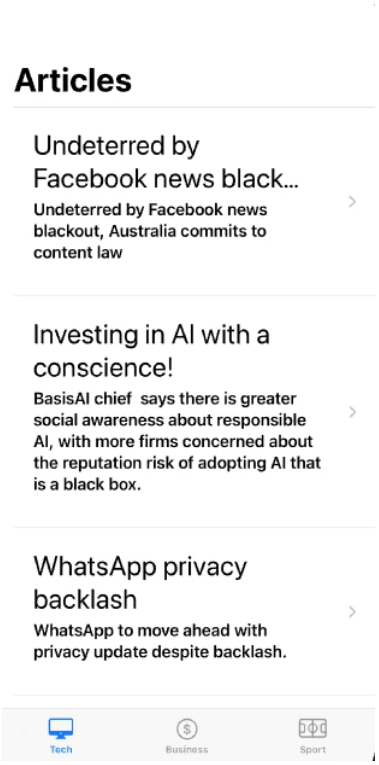
In this chapter, we learned how to implement C.R.U.D. operations using Firestore as our backend. We learned how to add Firestore to our application, how to work with a Firestore collection, how to display a list of notes, how to add a new note, how to delete a note and retrieve a single Firestore document to prepare our note for edit and how to update a note.

# Chapter 9: Building Cross Platform Apps in SwiftUI

With SwiftUI, we can use the code written for one platform and extend it to many platforms. In this chapter, we create an iOS app and reuse some of its components to create an iPad, macOS and watchOS app.

We will see how to share common resources between platforms (e.g. models) while having other resources that are platform specific (e.g. views). Having platform specific views let us follow platform-specific design guidelines and improve the user experience provided by our apps.

By the end of this chapter, we will have created an article reader app for different Apple platforms (fig. 1, 2):



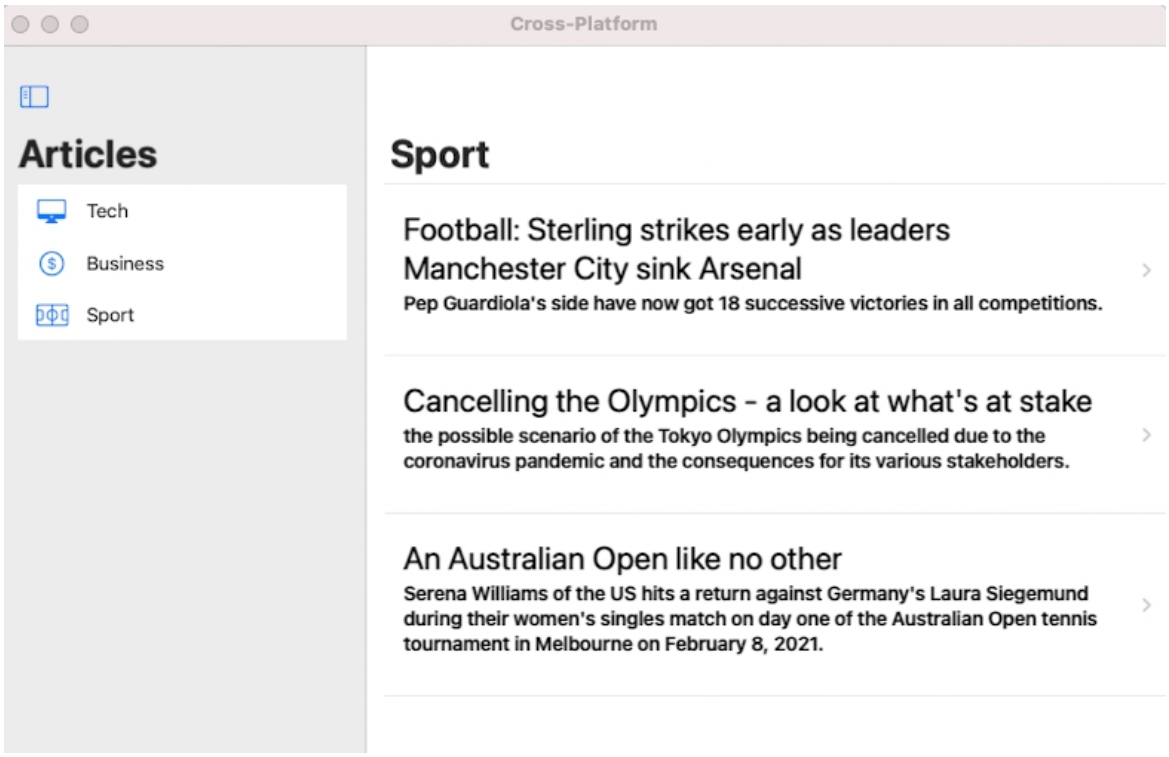

Cross-Platform
Articles
Tech
Business
Sport
Sport
Football: Sterling strikes early as leaders Manchester City sink Arsenal
Pep Guardiola's side have now got 18 successive victories in all competitions.
Cancelling the Olympics – a look at what's at stake
the possible scenario of the Tokyo Olympics being cancelled due to the coronavirus pandemic and the consequences for its various stakeholders.
An Australian Open like no other
Serena Williams of the US hits a return against Germany's Laura Siegemund during their women's singles match on day one of the Australian Open tennis tournament in Melbourne on February 8, 2021.

Figure 1


Business
1:03
Bitcoin hits US$1 trillion value as crypto leads other assets
Bitcoin's market value reached US$1 trillion for the first time, a surge that's

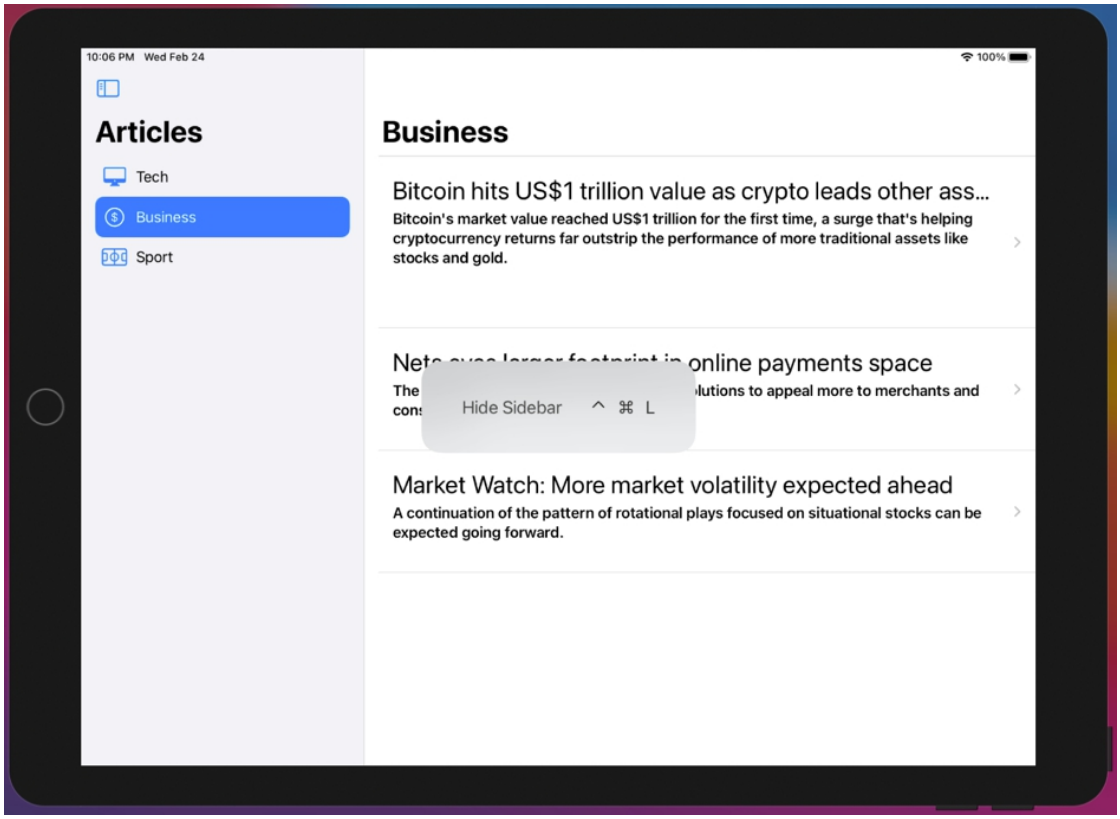


Figure 2

## *Creating a new Xcode Project*

First, create a new ‘ Multiplatform ’ Xcode project. So, choose ‘ Multiplatform ’ and ‘ App ’ (fig. 3).

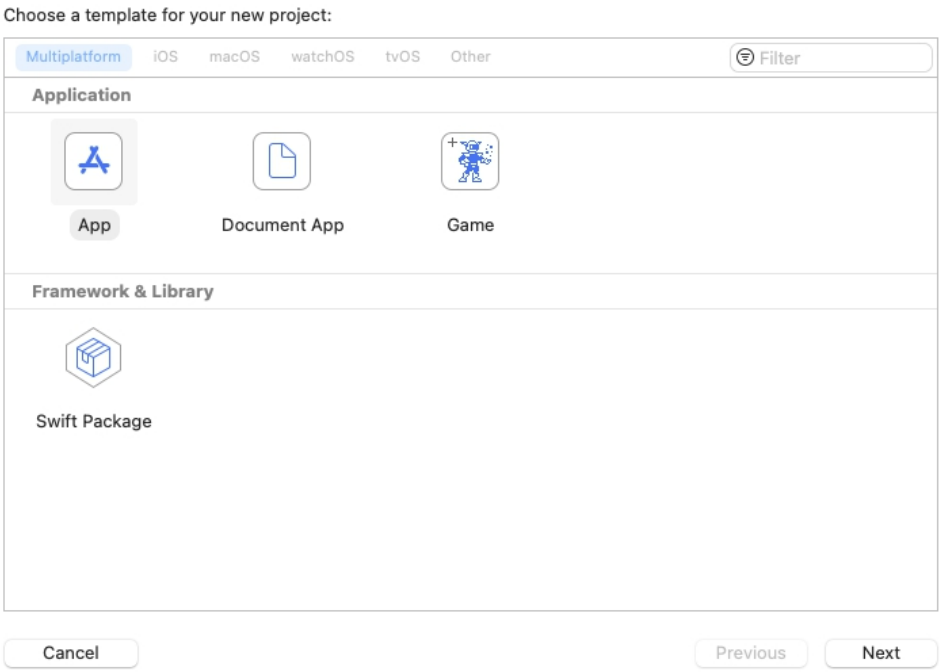



Figure 3

Select ‘ Next ’ . Give the project any name you want. I will name mine ‘ Cross-Platform ’ (fig. 4).

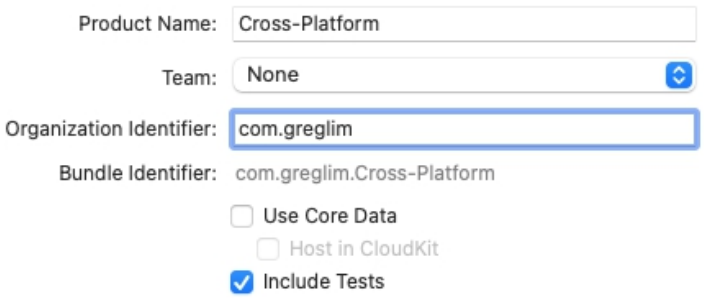
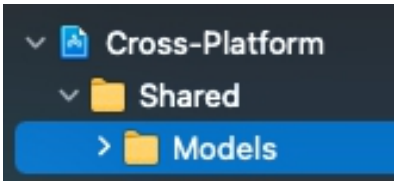



Figure 4

# Creating our Data Model

To stay organized, let ’ s create a group to store our models. Because models are shared across all platforms, we will create the group under the ‘ Shared ’ folder. So right-click on the ‘ Shared ’ folder in the navigation pane, select ‘ New Group ’ and name the folder *Models* .




In the ‘ Models ’ folder, create a new Swift file and name it *Article* . Under ‘ Targets ’ , check both ‘ Cross-Platform (iOS) ’ , ‘ Cross-Platform (macOS) ’ (fig. 5) and select ‘ Create ’ .

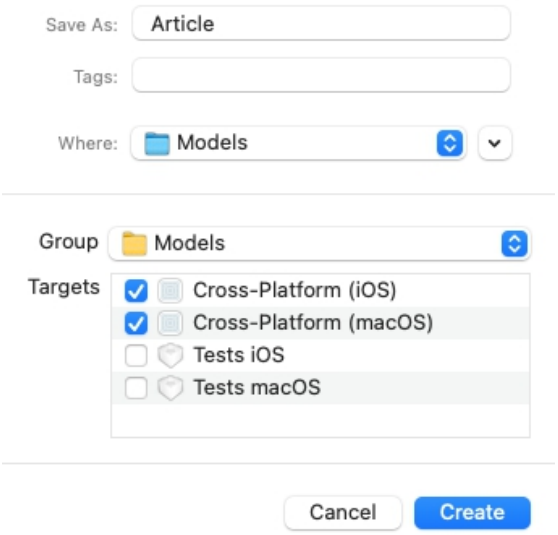



Figure 5

Next in *Article.swift* , create the *Article* struct:

```
import Foundation

struct Article: Identifiable {

    var id = UUID()
    let title: String
    let description: String
    let type: String
}
```

Below the struct, let ’ s create three arrays each representing a category of dummy *Article* instances (you can of course use your own dummy data).

```
let techArticles = [
```

```
    Article(title: "Undeterred by Facebook news blackout", description: "Undeterred by Facebook news blackout, Australia commits to content law", type: "Tech"),
    Article(title: "Investing in AI with a conscience!", description: "BasisAI chief says there is greater social awareness about responsible AI, with more firms concerned about the reputation risk of adopting AI that is a black box.", type: "Tech"),
    Article(title: "WhatsApp privacy backlash", description: "WhatsApp to move ahead with privacy update despite backlash.", type: "Tech")
]

let businessArticles = [
    Article(title: "Bitcoin hits US$1 trillion value as crypto leads other assets", description: "Bitcoin's market value reached US$1 trillion for the first time, a surge that's helping cryptocurrency returns far outstrip the performance of more traditional assets like stocks and gold.", type: "Business"),
    Article(title: "Nets eyes larger footprint in online payments space", description: "The CEO wants to boost the firm’s online solutions to appeal more to merchants and consumers.", type: "Business"),
    Article(title: "Market Watch: More market volatility expected ahead", description: "A continuation of the pattern of rotational plays focused on situational stocks can be expected going forward.", type: "Business")
]

let sportArticles = [
    Article(title: "Football: Sterling strikes early as leaders Manchester City sink Arsenal", description: "Pep Guardiola's side have now got 18 successive victories in all competitions.", type: "Sport"),
    Article(title: "Cancelling the Olympics - a look at what's at stake", description: "the possible scenario of the Tokyo Olympics being cancelled due to the coronavirus pandemic and the consequences for its various stakeholders.", type: "Sport"),
    Article(title: "An Australian Open like no other", description: "Serena Williams of the US hits a return against Germany's Laura Siegemund during their women's singles match on day one of the Australian Open tennis tournament in Melbourne on February 8, 2021.", type: "Sport"),
]
```

You can get the source code at *www.greglim.net/swiftui* .

## *ArticleView*

In *SharedFolder* , let ’ s create an SwiftUI *ArticleView* to display a single article. Again, under ‘ Targets ’ , check both ‘ Cross-Platform (iOS) ’ and ‘ Cross-Platform (macOS) ’ . Fill it with the below:

```
import SwiftUI

struct ArticleView: View {

    let article: Article
    var body: some View {
        VStack(alignment: .leading, spacing: 5) {
            Text(article.title)
                .font(.title)

            Text(article.description)
                .font(.headline)

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

As you can see, *ArticleView* is a straightforward view that displays the article ' s title and description.

## *ArticlesListView*

Next, in *Shared* , we create a SwiftUI View called *ArticlesListView* to list the articles. Fill it with the following:

```
import SwiftUI

struct ArticlesListView: View {

    let articles: [Article ]

    var body: some View {
        #if os(macOS)
        return
            view
                .frame(minWidth: 400, minHeight: 600)
        #else
        return view
        #endif
    }
```

```
    @ViewBuilder
    private var view: some View {
        List(articles) { article in
            NavigationLink(destination: ArticleView(article: article)) {
                ArticleView(article: article)
            }
        }
        .navigationTitle("\(articles[0].type)")
    }
}
```

Here, we check if the app is compiled for a Mac device using the compiler directive *#if os(macOS )* .
Build directives let us perform certain checks and custom logic when our code is being compiled. This is handy for a cross-platform code base where we have custom compilations based on different platforms. In our case, for a Mac app which has a larger screen, we adjust the size of the frame with the *frame* method.

## *TabBar*

We have three categories of articles: tech, business and sports. In an iPhone, we want these three categories to be three different tabs in a tabbar (fig. 6).

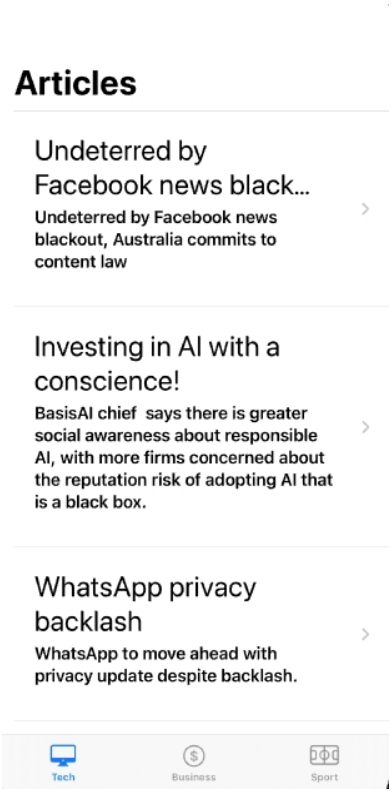



Figure 6

For iPad and Mac, we want the three categories to be listed in a sidebar (fig. 7).

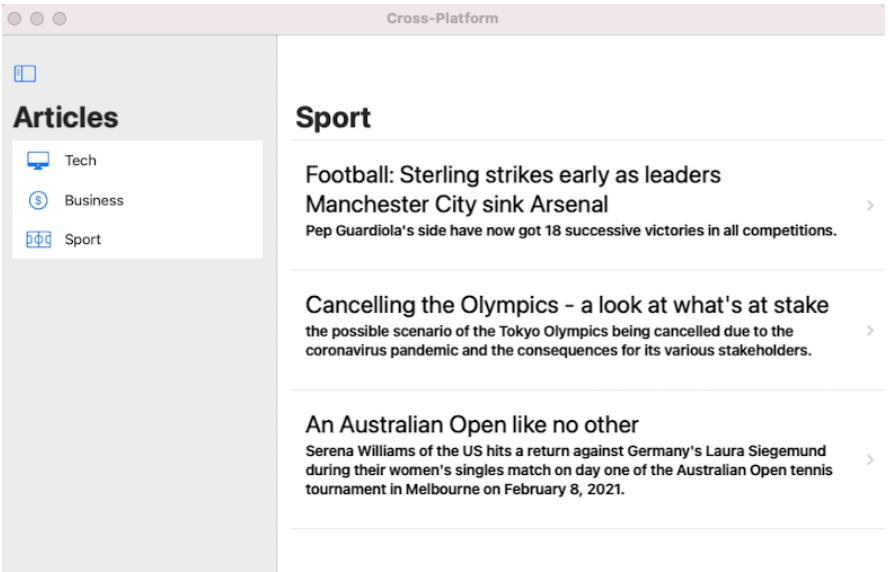



Figure 7

In the *iOS* folder, create a new SwiftUI view file *TabBarView.swift* . Note that we don ' t have to target ' Cross-Platform (macOS) ' for

this file since its meant only for iOS. Fill it with the following code:

```
import SwiftUI

struct TabBarView: View {

    var body: some View {
        TabView {
            ArticlesListView(articles: techArticles)
                .tabItem {
                    Image(systemName: "desktopcomputer")
                    Text("Tech")
                }
                .tag(0)

            ArticlesListView(articles: businessArticles)
                .tabItem {
                    Image(systemName: "dollarsign.circle")
                    Text("Science")
                }
                .tag(1)

            ArticlesListView(articles: sportArticles)
                .tabItem {
                    Image(systemName: "sportscourt")
                    Text("Design")
                }
                .tag(2)
        }
        .navigationTitle("Articles")
    }
}
```

We create three tabs by putting three *ArticlesListViews* each with a different *Article* array as defined previously in *Article.swift* (*techArticles* , *businessArticles* and *sportArticles* ) inside the *TabView* .

We use the *.tabItem* modifier to style each tab ’ s image and text. The image is created using *systemName* which lets us load images from the built-in SF Symbols icon set which has over 2,400 icons that Apple designed for apps to use

(*https://developer.apple.com/design/human-interface-guidelines/sf-symbols/overview/* ).
We use *tag()* to position their tab order.

Now, let ' s try running the iPhone version of our app. In *ContentView* , add:

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

Note that we need to enclose *TabBarView* in a *NavigationView* to ensure that the *NavigationLink* s in *ArticlesListView* can work. When we run the app in an iPhone simulator, we get (fig. 8):

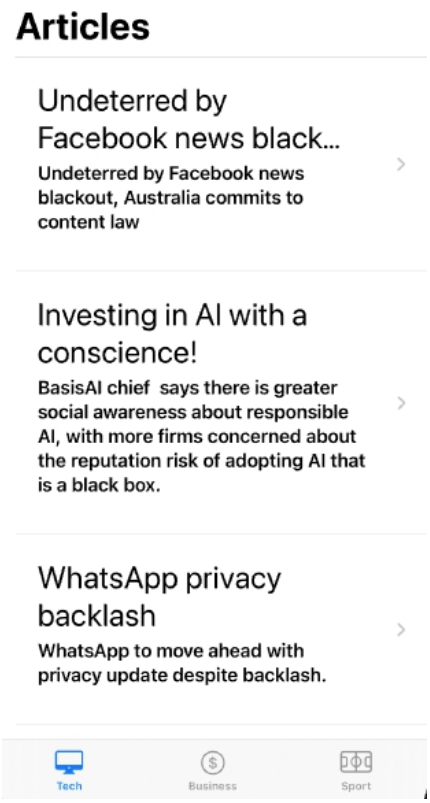


Figure 8

As you navigate to a different tab category, that category ' s articles

are listed.

## *SideBar*

Next, let ' s see how to implement the *SideBar* version of our app for the Mac and iPad (fig. 9).

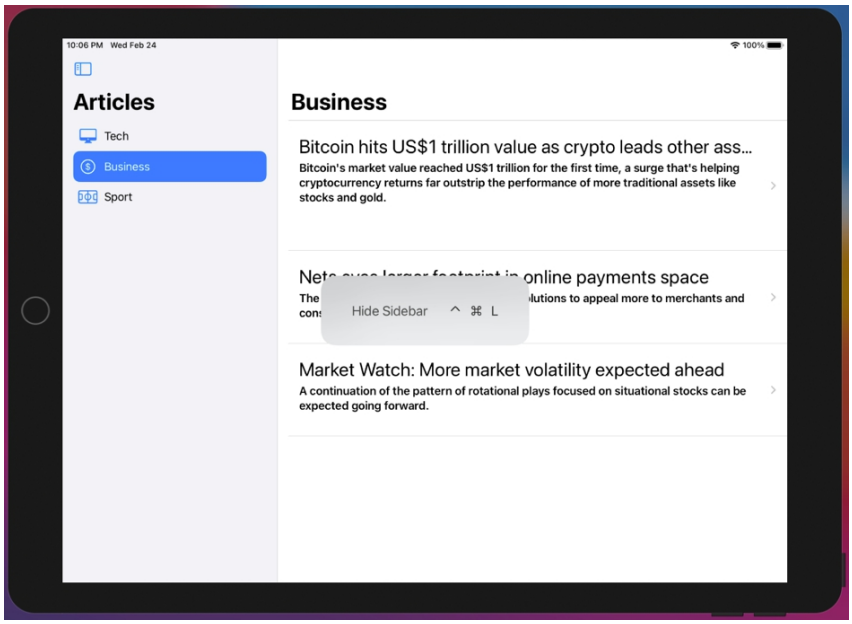


Figure 9

The SideBar navigation is a three-column navigation that provides quick access to top-level collections.

So, under *macOS* folder, create a new SwiftUI view file, *SideBarView.swift* . It has to target both iOS and Mac since we are using it for the iPad and Mac. Fill it in with the following:

```
import SwiftUI

struct SideBarView: View {


    @ViewBuilder
    var body: some View {
        List() {
            NavigationLink(
                destination: ArticlesListView(articles: techArticles),
                label: {
                    Label("Tech", systemImage: "desktopcomputer")
                }
```

```
            )

            NavigationLink(
                destination: ArticlesListView(articles: businessArticles) ,
                label: {
                    Label("Business", systemImage: "dollarsign.circle")
                }
            )

            NavigationLink(
                destination: ArticlesListView(articles: sportArticles),
                label: {
                    Label("Sport", systemImage: "sportscourt")
                }
            )
        }
        .navigationTitle("Articles")
        .listStyle(SidebarListStyle())
    }
}
```

We use the new *SideBarListStyle* list styling available on iOS 14 and macOS Big Sur.

*SideBar* uses a *List* to list the three categories of article types. Let ' s try running our Mac app. In *ContentView* , comment out *TabBarView* and add in *SideBarView* :

```
struct ContentView: View {
    var body: some View {
        NavigationView{
            //TabBarView()
            SideBarView()
        }
    }
}
```

And when you run the app on a Mac, you get (fig. 10) a three-level navigation article reader app:

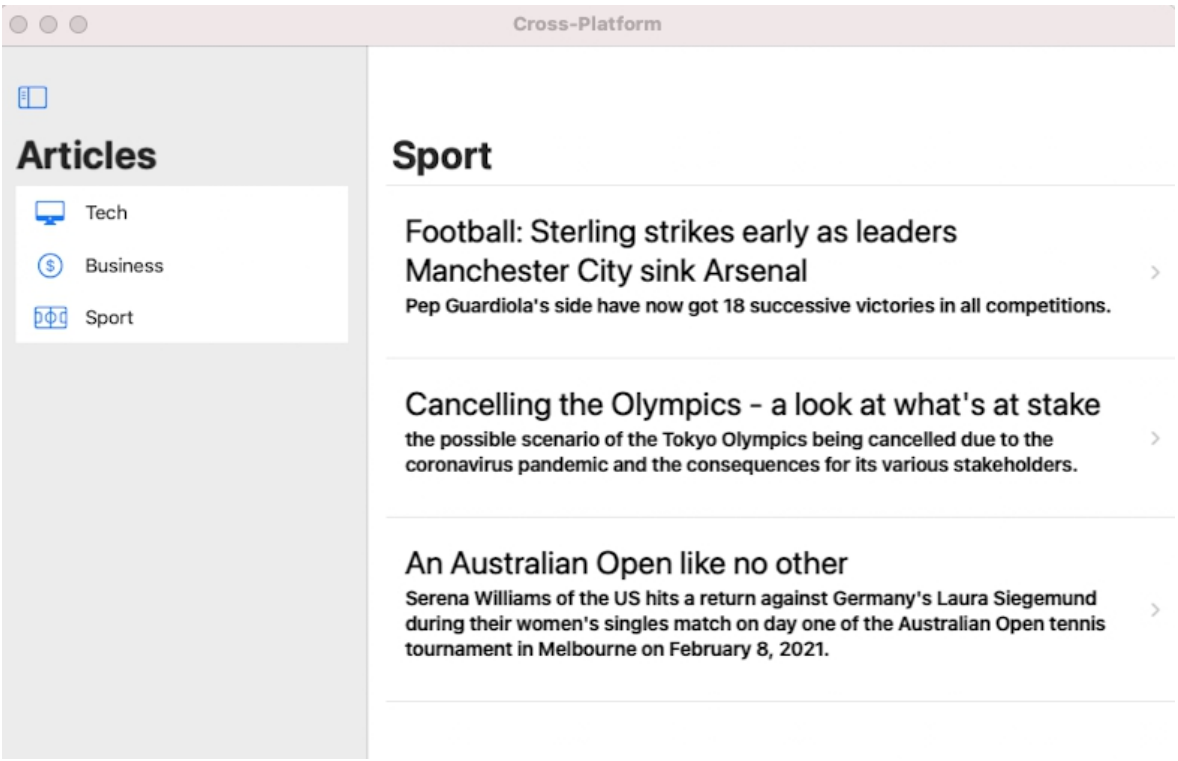


Figure 10

Now, how do we run *TabBarView* when running on iPhone and run *SideBarView* when running on Mac and iPad?

We will use *#if* to detect which OS the app is running on and also the *horizontalSizeClass* property to detect if the device is an iPad or iPhone. Fill in the below into *ContentView* :

```
import SwiftUI

struct ContentView: View{
    #if os(iOS)
    @Environment(\.horizontalSizeClass) var horizontalSizeClass:
UserInterfaceSizeClass?
    #endif


    @ViewBuilder
    var body: some View {
        NavigationView {
            #if os(iOS)
            if horizontalSizeClass == .compact {
                TabBarView()
            } else {
```

```
                SideBarView()
            }
            #else
            SideBarView()
            #endif
        }
    }
}
```

We create a *@Environment* object to store the size class value. We later check the size class value if its ‘ .compact ’ size class. Generally, iPhones have ‘ .compact ’ size, whereas iPads have ‘ .regular ’ size. Thus, if size is *.compact* (iPhone), we show the *TabBarView* . Else (iPad or Mac), we show the *SideBarView* .

So if we run the app on an iPad, it will show (fig. 11):

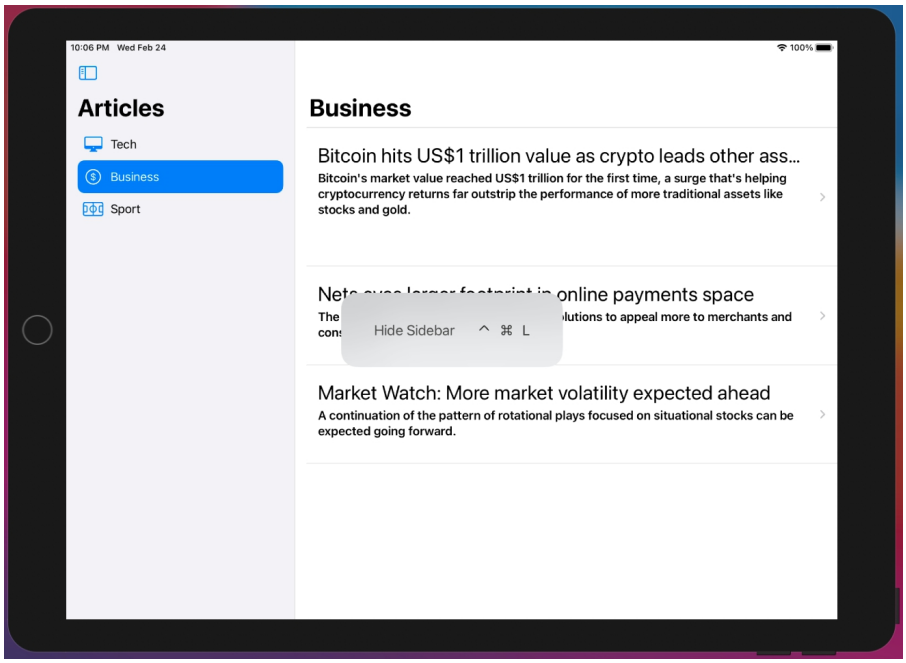


Figure 11

# Creating the watchOS version

So far, we have created an iOS app and a macOS version of our app. Let ’ s now create the watchOS version. Keep in mind that the screen on the watch is much smaller.

Let ’ s start by adding a watchOS target to our project. First, create a

new ‘ Target …’ (fig. 12):

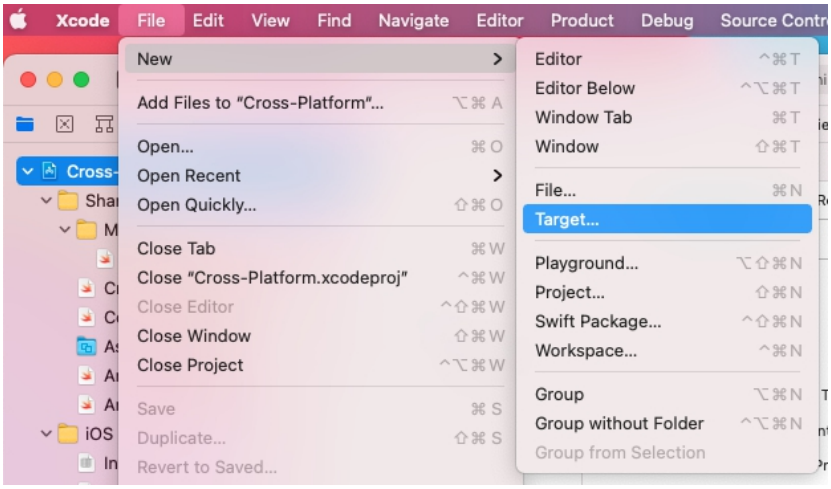



Figure 12

Choose the watchOS template, scroll down and select ‘ Watch App for iOS App ’ and click ‘ Next ’ (fig. 13).

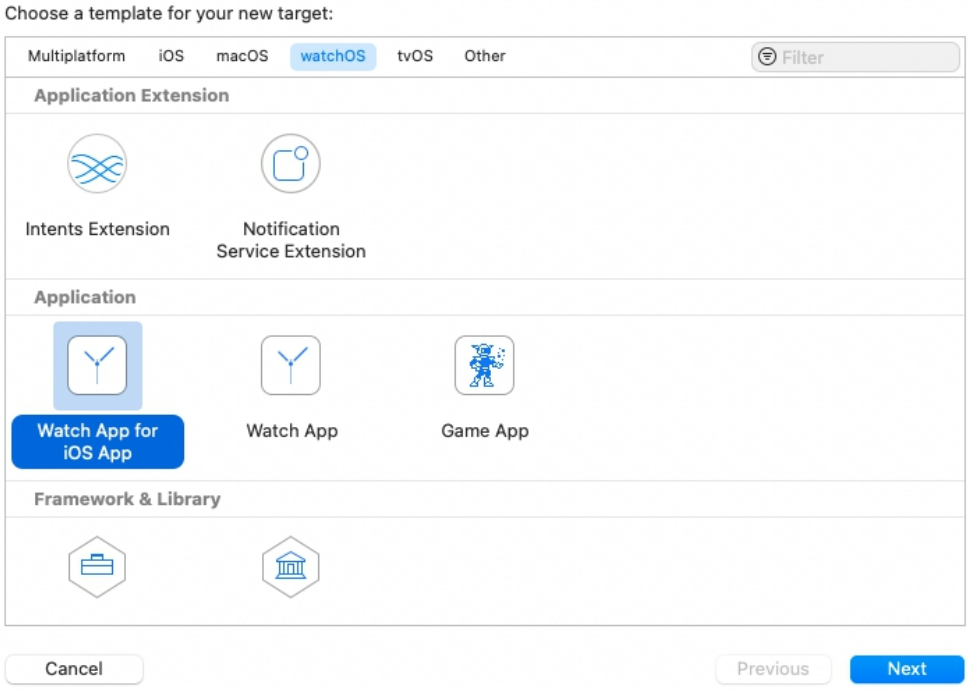



Figure 13

In the next screen, enter the product name ‘ watchOS-Cross-Platform ’ and select ‘ Finish ’ .

In the popup window, select ‘ Activate ’ (fig. 14).

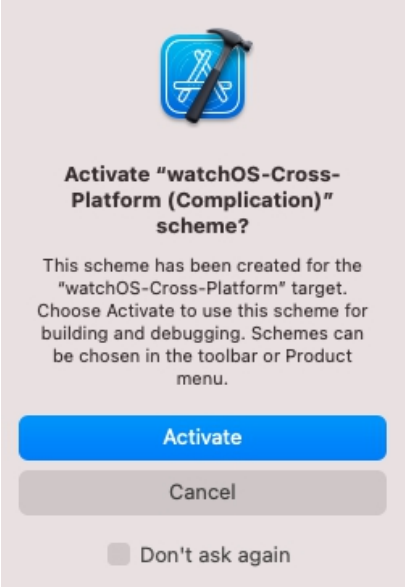


Figure 14

The *watchOS* folders should appear in the navigation pane (fig. 15). Xcode adds groups and files for the watchOS app to our project, along with the schemes to build and run the app.

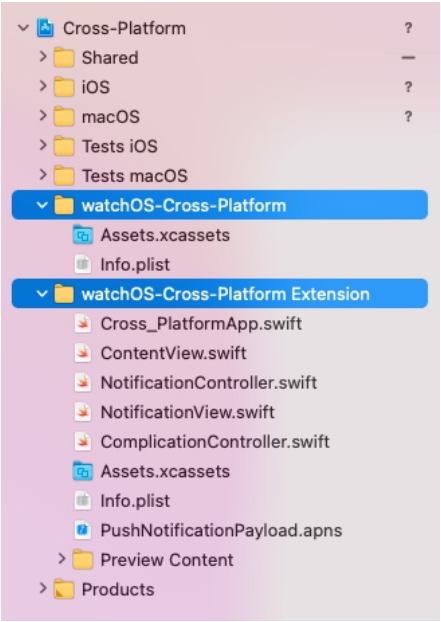


Figure 15

Select *ContentView.swift* in the *watchOS-Cross-Platform-Extension* folder. The preview should look something like (fig. 16):

Figure 16

Next, we will share the *Article.swift* , *ArticleView.swift* , *ArticlesListView.swift* with our watchOS platform. Select them, and in the *Inspector* pane, check the ‘ watchOS-Cross-Platform Extension ’ checkbox (fig. 17) in the ‘ Target Membership ’ section:

Figure 17

# Creating the Watch View

The view for the watch app should use smaller font sizes to accommodate the smaller display size. To do so, let ’ s edit *ArticleView* as shown in **bold** :

```
import SwiftUI

struct ArticleView: View {


    let article: Article
    var body: some View {
        #if os(watchOS)
        return
```

```
            VStack(alignment: .leading, spacing: 5) {
                Text(article.title)
                Text(article.description)
                Spacer()
            }
            .padding()
        #else
        return
            VStack(alignment: .leading, spacing: 5) {
                Text(article.title)
                    .font(.title)

                Text(article.description)
                    .font(.headline)

                Spacer()
            }
            .padding()
        #endif
    }
}
```

For the watch app, we simply remove the *.title* and *.headline* specifications.

Since watchOS supports *TabView* , let ' s share *TabBarView* () with *watchOS-Cross-Platform-Extension* (fig. 18).

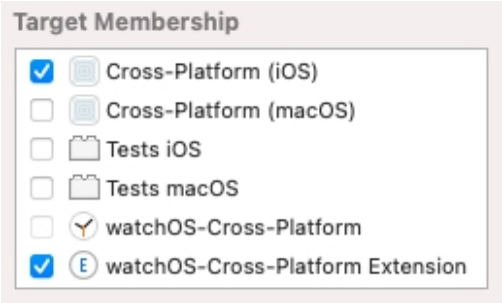



Figure 18

And in *ContentView* of *watchOS-Cross-Platform-Extension* folder, add:

```
struct ContentView: View {
    var body: some View {
```

```
        TabBarView()
    }
}
```

If you run the watch app now, it will throw a blank screen. That is because in *TabBarView.swift* , watchOS doesn ’ t yet support the *.navigationTitle ( … )* modifier.

```
struct TabBarView: View {
    var body: some View {
        TabView {
                …
        }
        //.navigationTitle("Articles")
    }
}
```

So, comment it out. Now when you run your app (fig. 19):


Figure 19

You are able to swipe left and right between the categories of articles, scroll down the list and navigate to article details.

With that, our cross-platform app is completed!

Notice that we shared files like *Article* , *ArticleView* , *ArticlesListView*

between our iPhone, iPad, Mac and watchOS app. But yet, each platform has different views (*TabBarView* and *SideBarView* ) for the different screen sizes.

## *Summary*

We have gone through quite a lot of content to equip you with the skills to create an iOS app and submit it to the app store.

Hopefully, you have enjoyed this book and would like to learn more from me. I would love to get your feedback, learning what you liked and didn't for us to improve.

Please feel free to email me at support@i-ducate.com if you encounter any errors with your code or to get updated versions of this book.

If you didn't like the book, or if you feel that I should have covered certain additional topics, please email us to let us know. This book can only get better thanks to readers like you.

If you like the book, I would appreciate if you could leave us a review too. Thank you and all the best for your learning journey in iOS development!

# About the Author

Greg Lim is a technologist and author of several programming books. Greg has many years in teaching programming in tertiary institutions and he places special emphasis on learning by doing.

Contact Greg at support@i-ducate.com or find out more at www.greglim.net