Hacking with iOS - SwiftUI

# Project 01 – WeSplit

Some notes about creating a new SwiftUI project:

* Example of how to create a basic project:
* For Product Name please enter the name of the project, like “WeSplit”.
* For Organization Identifier you can enter whatever you want, but if you have a website you should enter it with the components reversed: “hackingwithswift.com” would be “com.hackingwithswift”. If you don’t have a domain, make one up – “me.yourlastname.yourfirstname” is perfectly fine.
* For Interface please select SwiftUI.
* For Language please make sure you have Swift selected.
* Make sure all the checkboxes at the bottom are not checked.
* When creating a new project, you can check the target iOS by selecting the project in the project navigation bar, the one at the top, select Project list, and you can see the iOS Deployment Target, also if you select the Targets list, you can see the minimum Deployments target.

These are some of the files that are created by default, following the example:

* The WeSplitApp.swift contains code for launching your app.
* ContentView.swift contains the initial user interface for your program.
* Assets.xcassets is an asset catalog.
* Preview Content is a group with preview assets, another catalog, for example, for images you want to use when you’re designing your UI, to have an idea of how they might look.

The ContentView.swift file has a default code like this:

**import** SwiftUI

**struct** ContentView: View {

**var** body: **some** View {

VStack {

Image(systemName: "globe")

.imageScale(.large)

.foregroundColor(.accentColor)

Text("Hello, world!")

}

.padding()

}

}

**struct** ContentView\_Previews: PreviewProvider {

**static** **var** previews: **some** View {

ContentView()

}

}

The View protocol has one requirement: to have a computed property called body that returns some View.

The padding method is also called a modifier. They are like regular methods but they return a new view that contains both your original data plus the extra modification you asked for.

The ContentView struct won’t be part of the final app that goes to the App Store, it is specifically for Xcode to use so it can show a preview of your UI design.

With the preview of the UI visible, press Option+Cmd+P to update the canvas.

If you need to add more than 10 elements in a Form you can use Groups. Groups don’t actually change the way your user interface looks, they just let us work around SwiftUI’s limitation of ten child views inside a parent.

If you want your form to look different when splitting items into chunks, you should use the Section view instead. This splits your form into discrete visual groups, just like the Settings app does.

## Navigation Bar

By default SwiftUI ensures components are placed in an area where they can’t be covered up by system UI or device rounded corners – an area known as the safe area.

If you have a Form and you swipe around in the simulator, you will find you can move the row up so it goes under the clock. You can fix this by using a navigation bar. Example:

**import** SwiftUI

**struct** ContentView: View {

**var** body: **some** View {

NavigationView {

Form {

Section {

Text("Hello, world!")

}

}

.navigationTitle("SwiftUI")

.navigationBarTitleDisplayMode(.inline)

}

}

}

navigationTitle: add a navigation title, this uses a large font.

navigationBarTitleDisplayMode allows you to get a small font for the navigation bar.

## Modifying Program State

SwiftUI’s views are a function of their state, that means that the way your user interface looks – the things people can see and what they can interact with – are determined by the state of your program. For example, they can’t tap Continue until they have entered their name in a text field.

When creating struct methods that want to change properties, we need to add the mutating keyword: **mutating func doSomeWork()**, for example. However, Swift doesn’t let us make mutating computed properties, which means we can’t write **mutating var body: some View** – it just isn’t allowed.

Fortunately, Swift gives us a special solution called a *property wrapper*: a special attribute we can place before our properties that effectively gives them super-powers. In the case of storing simple program state like the number of times a button was tapped, we can use a property wrapper from SwiftUI called **@State**, like this:

**struct** ContentView: View {

@State private **var** tapCount = 0

**var** body: **some** View {

Button("Tap Count: \(tapCount)") {

**self**.tapCount += 1

}

}

}

**@State** allows us to work around the limitation of structs: we know we can’t change their properties because structs are fixed, but **@State** allows that value to be stored separately by SwiftUI in a place that *can* be modified.

There are several ways of storing program state in SwiftUI, and you’ll learn all of them. **@State** is specifically designed for simple properties that are stored in one view. As a result, Apple recommends we add **private** access control to those properties, like this: **@State private var tapCount = 0**.

## Binding State to User Interface Controls

Views are a function of their state – that text field can only show something if it reflects a value stored in your program. What SwiftUI wants is a string property in our struct that can be shown inside the text field, and will also store whatever the user types in the text field.

In the case of a text field to handle a name, Swift needs to make sure whatever is in the text is also in the **name** property, so that it can fulfill its promise that our views are a function of their state – that everything the user can see is just the visible representation of the structs and properties in our code.

This is what’s called a *two-way binding*: we bind the text field so that it shows the value of our property, but we also bind it so that any changes to the text field also update the property.

In Swift, we mark these two-way bindings with a special symbol so they stand out: we write a dollar sign before them. This tells Swift that it should read the value of the property but also write it back as any changes happen.

**struct** ContentView: View {

@State **private** **var** name = ""

**var** body: **some** View {

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

Form {

Text("Hello, \(name)")

}

}

}

## Creating views in a loop

In SwiftUI you have the ForEach to loop over arrays and ranges , creating as many views as needed. Example:

Form {

ForEach(0..<100) { number in

Text("Row \(number)")

}

}

We can use shorthand syntax for the parameter name:

Form {

ForEach(0 ..< 100) {

Text("Row \($0)")

}

}

// 1. Has an array of possible student names.

// 2. Has an @State property storing the currently selected student.

// 3. Creates a Picker view asking users to select their favorite, using a two-way binding to the @State property.

// 4. Uses ForEach to loop over all possible student names, turning them into a text view.

//

**import** SwiftUI

**struct** ContentView: View {

**let** students = ["Harry", "Hermione", "Ron"]

@State **private** **var** selectedStudent = "Harry"

**var** body: **some** View {

NavigationView {

Form {

Picker("Select your student", selection: $selectedStudent){

ForEach(students, id: \.**self**){

Text($0)

}

}

}

}

}

}

## Reading text from the user in TextField

When you need to have text fields for double values, you can add a format to the TextField to use Locale (Locale is a massive struct built into iOS that is responsible for storing all the user’s region settings – what calendar they use, how they separate thousands digits in numbers, whether they use the metric system, and more). But you can also use the keyboardType with .decimalPad to limit the user to enter just numeric values:

Form {

Section{

TextField("Amount", value: $checkAmount,

format: .currency(code: Locale.current.currency?.identifier ?? "USD"))

.keyboardType(.decimalPad)

}

Section{

Text(checkAmount, format: .currency(code: Locale.current.currency?.identifier ?? "USD"))

}

}

One of the great things about the **@State** property wrapper is that it automatically watches for changes, and when something happens it will automatically re-invoke the **body** property.

This synchronization happens because:

1. Our text field has a two-way binding to the checkAmount property.

2. The checkAmount property is marked with @State, which automatically watches for

changes in the value.

3. When an @State property changes SwiftUI will re-invoke the body property (i.e., reload

our UI)

4. Therefore the text view will get the updated value of checkAmount.

## Creating pickers in a Form

SwiftUI’s pickers serve multiple purposes, and exactly how they look depends on which device you’re using and the context where the picker is used.

The value of a picker is in fact an index.

**struct** ContentView: View {

@State **private** **var** checkAmount = 0.0

@State **private** **var** numberOfPeople = 2

@State **private** **var** tipPercentage = 20

**let** tipPercentages = [10, 15, 20, 25, 0]

**var** body: **some** View {

VStack {

NavigationView {

Form {

Section{

TextField("Amount", value: $checkAmount,

format: .currency(code: Locale.current.currency?.identifier ?? "USD"))

.keyboardType(.decimalPad)

}

Picker("Number of people", selection: $numberOfPeople){

ForEach(2 ..< 100){

Text("\($0) people")

}

}

}

.navigationTitle("We Split")

}

}

.padding()

}

}

## Adding a segmented control for tip percentages

A segmented control is a specialized kind of picker that shows a handful of options in a horizontal list, and it works great when you have only a small selection to choose from.

SwiftUI lets us add views to the header and footer of a section, which in this instance we can use to add a small explanatory prompt.

Section {

Picker("Tip percentage", selection: $tipPercentage) {

ForEach(tipPercentages, id: \.**self**) {

Text($0, format: .percent)

}

}

.pickerStyle(.segmented)

} header: {

Text("How much tip do you want to leave?")

}

## Calculating the total per person

We can create properties, for example, a property called totalPerPerson where we can calculate the amount to pay per person according to the values selected and entered in the other properties:

// Property that calculates the total per person

**var** totalPerPerson: Double {

// The picker starts with 2, so we need to add 2 to the value

**let** peopleCount = Double(numberOfPeople + 2)

**let** tipSelection = Double(tipPercentage)

**let** tipValue = checkAmount / 100 \* tipSelection

**let** grandTotal = checkAmount + tipValue

**let** amountPerPerson = grandTotal / peopleCount

**return** amountPerPerson

}

This one can be displayed in the Text view below:

Section{

Text(totalPerPerson,

format: .currency(code: Locale.current.currency?.identifier ?? "USD"))

}

## Hiding the keyboard

To hide the keyboard:

1. We need to give SwiftUI some way of determining whether the check amount box should currently have focus – should be receiving text input from the user.
2. We need to add some kind of button to remove that focus when the user wants, which will in turn cause the keyboard to go away.

To solve the first one you need to meet your second property wrapper: @FocusState. This is exactly like a regular @State property, except it’s specifically designed to handle input focus in our UI.

For example, you can create a property like this:

@FocusState **private** **var** amountIsFocused: Bool

And attach it to the text field with the modifier focused:

TextField("Amount", value: $checkAmount,

format: .currency(code: Locale.current.currency?.identifier ?? "USD"))

.keyboardType(.decimalPad)

.focused($amountIsFocused)

The second part of our solution is to add a toolbar to the keyboard when it appears, so we can place a Done button in there. To make this work really well you need to meet several new SwiftUI views:

.toolbar {

ToolbarItemGroup(placement: .keyboard) {

Button("Done") {

amountIsFocused = **false**

}

}

This is a modifier to be applied to the Form.

1. The toolbar() modifier lets us specify toolbar items for a view. These toolbar items might appear in various places on the screen – in the navigation bar at the top, in a special toolbar area at the bottom, and so on.
2. ToolbarItemGroup lets us place one or more buttons in a specific location, and this is where we get to specify we want a keyboard toolbar – a toolbar that is attached to the keyboard, so it will automatically appear and disappear with the keyboard.
3. The Button view we’re using here displays some tappable text, which in our case is “Done”. We also need to provide it with some code to run when the button is pressed, which in our case sets amountIsFocused to false so that the keyboard is dismissed.

The complete code looks like this:

**import** SwiftUI

**struct** ContentView: View {

@State **private** **var** checkAmount = 0.0

@State **private** **var** numberOfPeople = 2

@State **private** **var** tipPercentage = 20

@FocusState **private** **var** amountIsFocused: Bool

**let** tipPercentages = [10, 15, 20, 25, 0]

// Property that calculates the total per person

**var** totalPerPerson: Double {

// The picker starts with 2, so we need to add 2 to the value

**let** peopleCount = Double(numberOfPeople + 2)

**let** tipSelection = Double(tipPercentage)

**let** tipValue = checkAmount / 100 \* tipSelection

**let** grandTotal = checkAmount + tipValue

**let** amountPerPerson = grandTotal / peopleCount

**return** amountPerPerson

}

**var** body: **some** View {

VStack {

NavigationView {

Form {

Section{

TextField("Amount", value: $checkAmount,

format: .currency(code: Locale.current.currency?.identifier ?? "USD"))

.keyboardType(.decimalPad)

.focused($amountIsFocused)

Picker("Number of people", selection: $numberOfPeople){

ForEach(2 ..< 100){

Text("\($0) people")

}

}

}

Section {

Picker("Tip percentage", selection: $tipPercentage) {

ForEach(tipPercentages, id: \.**self**) {

Text($0, format: .percent)

}

}

.pickerStyle(.segmented)

} header: {

Text("How much tip do you want to leave?")

}

Section{

Text(totalPerPerson,

format: .currency(code: Locale.current.currency?.identifier ?? "USD"))

}

}

.navigationTitle("We Split")

.toolbar {

ToolbarItemGroup(placement: .keyboard) {

Spacer()

Button("Done") {

amountIsFocused = **false**

}

}

}

}

}

.padding()

}

}

**struct** ContentView\_Previews: PreviewProvider {

**static** **var** previews: **some** View {

ContentView()

}

}

# Project 02 – Guess the Flag

## Using stacks to arrange views

When we return some View for our body, SwiftUI expects to receive back some kind of view that can be displayed on the screen. That might be a navigation view, a form, a text view, a picker, or something else entirely, but it must conform to the View protocol so that it can be drawn on the screen.

If we want to return multiple things we have various options, but three are particularly useful. They are HStack, VStack, and ZStack, which handle horizontal, vertical, and, er, zepth.

The previous stacks allow a maximum of 10 views as well, so if you need to have more items, you will need to group them using other stacks. The VStack and HStack have some properties like spacing or alignment that allow you to modify the way the views inside are displayed. For Example:

**var** body: **some** View {

VStack (alignment: .leading, spacing: 20) {

Text("Hello, world!")

Text("This is another text view!")

}

}

You can also use Spacer(), that is a kind of view to add some space between other views.

In the ZStack, you don’t have spacing property, but you can use alignment.

## Colors and Frames

When using stacks, if you do something like this:

ZStack {

Text("Your content")

}

.background(.red)

You will see the are painted in red, is just the text, because the stacks take fit their content. You can use Color.red which will take the whole screen size and it is a view by itself, but you can also use a frame to limit the width and height.

ZStack {

Color.red

.frame(width: 200, height: 200)

Text("Your content")

}

You can also specify minimum and maximum widths and heights, depending on the layout you want. For example: we could say we want a color that is no more than 200 points high, but for its width must be at least 200 points wide but can stretch to fill all the available width that’s not used by other stuff:

ZStack {

Color.red

.frame(minWidth: 200, maxWidth: .infinity, maxHeight: 200)

Text("Your content")

}

Beyond red, green and other colors, you can use semantic color, like primary, secondary, that refers to the text color. You can also use custom colors:

Color.primary

Color(red: 1, green: 0.8, blue: 0)

By default, the colors don’t include the safe area (the top and bottom parts of the screen. You can indicate to ignore the safe area.

ZStack {

Color.red

Text("Your content")

}

.ignoresSafeArea()

Keep anything important inside the safe area.

SwiftUI gives us an alternative for the foreground color that provides a very slightly different effect: change the foregroundColor() modifier to foregroundStyle().

## Gradients

SwiftUI gives us three kinds of gradients to work with. Gradients are made up of several components:

• An array of colors to show

• Size and direction information

• The type of gradient to use

LinearGradient(gradient: Gradient(colors: [.white, .black]),

startPoint: .top, endPoint: .bottom)

You can use stoppers to specify both a color and how far along the gradient the color should be used:

LinearGradient(gradient: Gradient(stops: [

Gradient.Stop(color: .white, location: 0.45),

Gradient.Stop(color: .black, location: 0.55),

]), startPoint: .top, endPoint: .bottom)

As an alternative, radial gradients move outward in a circle shape, so instead of specifying a direction we specify a start and end radius – how far from the center of the circle the color should start and stop changing. For example:

RadialGradient(gradient: Gradient(colors: [.blue, .black]),

center: .center, startRadius: 20, endRadius: 200)

The last gradient type is called an angular gradient, although you might have heard it referred to elsewhere as a conic or conical gradient.

AngularGradient(gradient: Gradient(colors: [.red, .yellow, .green, .blue, .purple, .red]), center: .center)

## Buttons and Images

The buttons allow you to execute some actions in different ways, for example:

**struct** ContentView: View {

**var** body: **some** View {

VStack{

Button("Delete selection") {

print("Now deleting...")

}

//Call a function when click the button

Button("Delete Items", action: executeDelete)

}

}

**func** executeDelete(){

print("Now deleting...")

}

}

You can also give some style to the buttons with some predefined properties, like buttonStyle and role:

VStack {

Button("Button 1") { }

.buttonStyle(.bordered)

Button("Button 2", role: .destructive) { }

.buttonStyle(.bordered)

Button("Button 3") { }

.buttonStyle(.borderedProminent)

.tint(.mint)

Button("Button 4", role: .destructive) { }

.buttonStyle(.borderedProminent)

}

You can customize the button, like this, showing a label with different properties:

Button {

print("Button was tapped")

} label: {

Text("Tap me!")

.padding()

.foregroundColor(.white)

.background(.red)

}

You can add images using some built-in assets catalog images. For example:

Image(systemName: "pencil")

You can use them as decorative images for the buttons.

Button {

print("Button was tapped")

} label: {

Label("Edit", systemImage: "pencil")

}

## Showing Alert Messages

In SwiftUI you create the alert and set the conditions under which it should be shown.

A basic SwiftUI alert has a title and a button that dismisses it, but the more interesting part is how we present that alert: we don’t assign the alert to a variable then write something like myAlert.show(), because that would be back to the old “series of events” way of thinking.

Instead, we create some state that tracks whether our alert is showing, like this:

@State private var showingAlert = false

We then attach our alert somewhere to our user interface, telling it to use that state to determine whether the alert is presented or not. SwiftUI will watch showingAlert, and as soon as it becomes true it will show the alert.

**struct** ContentView: View {

@State **private** **var** showingAlert = **false**

**var** body: **some** View {

VStack {

Button ("Show Alert"){

showingAlert = **true**

}.alert("Important message", isPresented: $showingAlert){

Button("OK"){ }

}

}

}

}

In the previous example, the isPresented property is bound to the showingAlert property, this way, when the value of the showingAlert changes, it will present the alert.

Any button insider the alert will dismiss the alert.

You can add as many buttons as you need and user roles for them:

@State **private** **var** showingAlert = **false**

**var** body: **some** View {

VStack {

Button ("Show Alert"){

showingAlert = **true**

}.alert("Important message", isPresented: $showingAlert){

Button("Delete", role: .destructive){ }

Button("Cancel", role: .cancel) {}

} message: {

Text("Please read this")

}

}

}

## Notes from Project

When you have several asset images, you just need to select the Assets.xcassets in the project and drag and drop the images there.

.shuffled() method can be used in an array to shuffle or re-arranged the items.

The **renderingMode(.original)** modifier tells SwiftUI to render the original image pixels rather than trying to recolor them as a button.

We can control the size and style of text using the font() modifier, which lets us select from one of the built-in font sizes on iOS. As for adjusting the weight of fonts – whether we want super-thing text, slightly bold text, etc – we can get fine-grained control over that by adding a weight() modifier to whatever font we ask for.

For the flags, and for any other view you can give a number of modifiers to affect the way views are presented, and we’re going to use two here: one to change the shape of flags, and one to add a shadow.

Making our image capsule shaped is as easy as adding the .clipShape(Capsule()) modifier, like this:

.clipShape(Capsule())

The final code is this:

**import** SwiftUI

**struct** ContentView: View {

@State **private** **var** showingScore = **false**

@State **private** **var** scoreTitle = ""

@State **private** **var** userScore = 0

@State **private** **var** questionsCount = 1

@State **private** **var** countries = ["Estonia", "France", "Germany", "Ireland", "Italy", "Nigeria",

"Poland", "Russia", "Spain", "UK", "US"].shuffled()

@State **private** **var** correctAnswer = Int.random(in: 0...2)

@State **private** **var** showingAlert = **false**

@State **private** **var** showingResultAlert = **false**

**var** body: **some** View {

ZStack {

RadialGradient(stops: [

.init(color: Color(red: 0.1, green: 0.2, blue: 0.45), location: 0.3),

.init(color: Color(red: 0.76, green: 0.15, blue: 0.26), location: 0.3)

], center: .top, startRadius: 200, endRadius: 700)

.ignoresSafeArea()

VStack{

Spacer()

Text("Guess the Flag")

.font(.largeTitle.weight(.bold))

.foregroundColor(.white)

VStack(spacing: 15){

VStack {

Text("Tap the flag of")

.foregroundStyle(.secondary)

.font(.subheadline.weight(.heavy))

Text(countries[correctAnswer])

.font(.largeTitle.weight(.semibold))

}

ForEach(0..<3){number **in**

Button {

flagTapped(number)

} label: {

Image(countries[number])

.renderingMode(.original)

.clipShape(Capsule())

.shadow(radius: 5)

}

}

}

.frame(maxWidth: .infinity)

.padding(.vertical, 20)

.background(.regularMaterial)

.clipShape(RoundedRectangle(cornerRadius: 20))

Spacer()

Spacer()

Text("Score: \(userScore)")

.foregroundColor(.white)

.font(.title.bold())

Spacer()

}

.padding()

}

.alert(scoreTitle, isPresented: $showingScore){

Button("Continue", action: askQuestion)

} message: {

Text("Your score is \(userScore)")

}

.alert("Results", isPresented: $showingResultAlert){

Button("Restart the Game", action: resetGame)

} message: {

Text("Your final score is \(userScore)")

}

}

**func** flagTapped(\_ number: Int){

**if** number == correctAnswer {

scoreTitle = "Correct"

userScore += 1

} **else** {

scoreTitle = "Wrong! That's the flag of \(countries[number])"

}

showingScore = **true**

questionsCount += 1

**if** questionsCount == 9 {

showingResultAlert = **true**

}

}

**func** askQuestion() {

countries.shuffle()

correctAnswer = Int.random(in: 0...2)

}

**func** resetGame(){

userScore = 0

questionsCount = 1

showingScore = **false**

askQuestion()

}

}

**struct** ContentView\_Previews: PreviewProvider {

**static** **var** previews: **some** View {

ContentView()

}

}

Some important things to remember:

* The Spacers allow you to add spaces to segment the elements in a VStack or HStack
* There are some predefined modifiers for the text elements, like foregroundStyle(.secondary) of .subheadline.bold()
* A button can have an image that fits the entire button or just a part along with a text.
* To keep the original color of the image in a button, use .renderingMode(.original)
* You can give different shapes to a view using clipShape
* You can use different properties to modify how to display a view, like .frame or .padding and .background.
* An alert uses the isPresented to know when it should be displayed.
* An Alert can have as many buttons as you need.
* You can define methods inside a Struct

# Project 03 – Views and Modifiers

## Why does SwiftUI use structs for views?

There are several reasons why in SwiftUI we use structs instead of classes:

1. Performance: Structs are simpler and faster than classes. In UIKit, every view descended from a class called UIView that had many properties and methods. There were lots of these, and every UIView and UIView subclass had to have them, because that’s how inheritance works. As structs cannot inherit, they contain what you see and nothing more, it doesn’t load anything more.
2. Structs forces us to think about isolating state in a clean way. SwiftUI encourages us to move to a more functional design approach: our views become simple, inert things that convert data into UI, rather than intelligent things that can grow out of control.

## What’s behind the main SwiftUI view?

This if the starting code you see when working with SwiftUI:

struct ContentView: View {

var body: some View {

Text("Hello, world!")

.padding()

}

}

You should try to get into the mindset that there is nothing behind our view – that what you see is all we have.

Now, right now at least there is something behind our content view called a UIHostingController: it is the bridge between UIKit (Apple’s original iOS UI framework) and SwiftUI. However, if you start trying to modify that you’ll find that your code no longer works on Apple’s other platforms, and in fact might stop working entirely on iOS at some point in the future.

One way, for example, to make the Text view to use all the available space and color it red, is this:

Text("Hello, world!")

.frame(maxWidth: .infinity, maxHeight: .infinity)

.background(.red)

## Why modifiers order matters?

Whenever we apply a modifier to a SwiftUI view, we actually create a new view with that change applied – we don’t just modify the existing view in place.

You can peek into the underbelly of SwiftUI by asking for the type of our view’s body. Modify the button to this:

Button("Hello, world!") {

print(type(of: self.body))

}

.background(.red)

.frame(width: 200, height: 200)

Swift’s type(of:) method prints the exact type of a particular value, and in this instance it will print the following:

ModifiedContent<ModifiedContent<Button<Text>, \_BackgroundStyleModifier<Color>>, \_FrameLayout>

So, if you have a code like this:

Text("Hello, world!")

.padding()

.background(.red)

.padding()

.background(.blue)

.padding()

.background(.green)

.padding()

.background(.yellow)

You will see how every color is stack up over the color defined before.

## Why SwiftUI use “some View” for its type?

The “some View” allow us to define whatever view we want inside, otherwise, we should define the exact type to return having in mind the modifiers applied to the view to return.

First, using some View is important for performance: SwiftUI needs to be able to look at the views we are showing and understand how they change, so it can correctly update the user interface. If SwiftUI didn’t have this extra information, it would be really slow for SwiftUI to figure out exactly what changed.

1. How does VStack work – it conforms to the View protocol, but how does it fill the “what kind of content does it have?” hole if it can contain lots of different things inside it?

2. What happens if we send back two views directly from our body property, without wrapping them in a stack?

To answer the first question first, if you create a VStack with two text views inside, SwiftUI silently creates a TupleView to contain those two views – a special type of view that holds exactly two views inside it.

As for the second question, Swift silently applies a special attribute to the body property called @ViewBuilder. This has the effect of silently wrapping multiple views in one of those TupleView containers, so that even though it looks like we’re sending back multiple views they get combined into one TupleView.

## Conditional modifiers

You can use a ternary conditional operator in the properties of a View, for example:

struct ContentView: View {

@State private var useRedText = false

var body: some View {

Button("Hello World") {

// flip the Boolean between true and false

useRedText.toggle()

}

.foregroundColor(**useRedText ? .red : .blue**)

}

}

You can often use regular if conditions to return different views based on some state, but this actually creates more work for SwiftUI – rather than seeing one Button being used with different colors, it now sees two different Button views, and when we flip the Boolean condition it will destroy one to create the other rather than just recolor what it has. So, this kind of code might look the same, but it’s actually less efficient:

var body: some View {

if useRedText {

Button("Hello World") {

useRedText.toggle()

}

.foregroundColor(.red)

} else {

Button("Hello World") {

useRedText.toggle()

}

.foregroundColor(.blue)

}

}

## Environment modifiers

Many modifiers can be applied to containers, which allows us to apply the same modifier to many views at the same time. For example:

VStack {

Text("Gryffindor")

Text("Hufflepuff")

Text("Ravenclaw")

Text("Slytherin")

}

.font(.title)

The font title will be applied to all the Text elements. This is called an environment modifier and is different from a regular modifier that is applied to a view. However, you can use font property in one of the Text views and this last one will have the priority over the container property.

However, not all the modifiers work the same way.

## Views as properties

There are several ways to create complex views:

We could create two text views like this as properties, then use them inside a VStack:

struct ContentView: View {

let motto1 = Text("Draco dormiens")

let motto2 = Text("nunquam titillandus")

var body: some View {

VStack {

motto1

motto2

} }

}

You can even apply modifiers to the properties.

You can create computed properties if you want, like this:

var motto1: some View {

Text("Draco dormiens")

}

This is often a great way to carve up your complex views into smaller chunks, but be careful: unlike the body property, Swift won’t automatically apply the @ViewBuilder attribute here, so if you want to send multiple views back you have three options.

First, you can place them in a stack, like this

var spells: some View {

VStack {

Text("Lumos")

Text("Obliviate")

}

}

If you don’t specifically want to organize them in a stack, you can also send back a Group. When this happens, the arrangement of your views is determined by how you use them elsewhere in your code:

var spells: some View {

Group {

Text("Lumos")

Text("Obliviate")

}

}

The third option is to add the @ViewBuilder attribute yourself,

@ViewBuilder var spells: some View {

Text("Lumos")

Text("Obliviate")

}

## View composition

SwiftUI lets us break complex views down into smaller views without incurring much if any performance impact.

struct CapsuleText: View {

var text: String

var body: some View {

Text(text)

.font(.largeTitle)

.padding()

.foregroundColor(.white)

.background(.blue)

.clipShape(Capsule())

}

}

We can then use that CapsuleText view inside our original view, like this:

struct ContentView: View {

var body: some View {

VStack(spacing: 10) {

CapsuleText(text: "First")

CapsuleText(text: "Second")

} }

}

And you can use more properties to the CapsuleText views inside the body view.

## Custom modifiers

To create a custom modifier, create a new struct that conforms to the **ViewModifier** protocol. This has only one requirement, which is a method called body that accepts whatever content it’s being given to work with, and must return **some View**. For Example:

struct Title: ViewModifier {

func body(content: Content) -> some View {

content

.font(.largeTitle)

.foregroundColor(.white)

.padding()

.background(.blue)

.clipShape(RoundedRectangle(cornerRadius: 10))

}

}

We can now use that with the modifier() modifier – yes, it’s a modifier called “modifier”, but it lets us apply any sort of modifier to a view, like this:

Text("Hello World")

.modifier(Title())

When working with custom modifiers, it’s usually a smart idea to create extensions on View that make them easier to use. For example, we might wrap the Title modifier in an extension such as this:

extension View {

Text("Hello World")

.modifier(Title())

}

We can now use the modifier like this:

Text("Hello World")

.titleStyle()

Custom modifiers can do much more than just apply other existing modifiers – they can also create new view structure, as needed. Remember, modifiers return new objects rather than modifying existing ones, so we could create one that embeds the view in a stack and adds another view:

struct Watermark: ViewModifier {

var text: String

func body(content: Content) -> some View {

ZStack(alignment: .bottomTrailing) {

content

Text(text)

.font(.caption)

.foregroundColor(.white)

.padding(5)

.background(.black)

}

}

}

extension View {

func watermarked(with text: String) -> some View {

modifier(Watermark(text: text))

}

}

With that in place, we can now add a watermark to any view like this:

Color.blue

.frame(width: 300, height: 200)

.watermarked(with: "Hacking with Swift")

## Custom containers

You can create custom containers using a stack called GridStack. What we want to say is that there is a new struct called GridStack that conforms to the View protocol and has a set number of rows and columns, and that inside the grid will be lots of content cells that themselves must conform to the View protocol.

struct GridStack<Content: View>: View {

let rows: Int

let columns: Int

let content: (Int, Int) -> Content

var body: some View {

VStack {

ForEach(0..<rows, id: \.self) { row in

HStack {

ForEach(0..<columns, id: \.self) { column in

content(row, column)

}

}

}

}

}

}

The first line – struct GridStack<Content: View>: View – uses a more advanced feature of Swift called generics, which in this case means “you can provide any kind of content you like, but whatever it is it must conform to the View protocol.” After the colon we repeat View again to say that GridStack itself also conforms to the View protocol.

Take particular note of the let content line – that defines a closure that must be able to accept two integers and return some sort of content we can show.

Tip: When looping over ranges, SwiftUI can use the range directly only if we know for sure the values in the range won’t change over time.

struct ContentView: View {

var body: some View {

GridStack(rows: 4, columns: 4) { row, col in

Text("R\(row) C\(col)")

} }

# Project 04 – Better Rest

All iPhones come with a technology called Core ML built right in, which allows us to write code that makes predictions about new data based on previous data it has seen. We’ll start with some raw data, give that to our Mac as training data, then use the results to build an app able to make accurate estimates about new data – all on device, and with complete privacy for users.

Using a technique called regression analysis we can ask the computer to come up with an algorithm able to represent all our data. This in turn allows it to apply the algorithm to fresh data it hasn’t seen before, and get accurate results.

## Entering numbers with Stepper

A stepper is a control with a – and a + button you can tap to select a specific number. You can use a Stepper with Int, double and more.

In the following example, we create a numeric variable that will be bound to a Stepper and we set some optional parameters to define things like, a range of numbers the user can select, the step to increase or decrease the values and we also show the value in a formatted way, to avoid a lot of zeros after the number.

**struct** ContentView: View {

@State **private** **var** sleepAmount = 8.0

**var** body: **some** View {

Stepper("\(sleepAmount.formatted()) hours",

value: $sleepAmount,

in: 4...12,

step: 0.25)

}

}

## Selecting dates and times with DatePicker

In SwiftUI you can have DatePicker too, for example:

**struct** ContentView: View {

@State **private** **var** wakeUp = Date.now

**var** body: **some** View {

DatePicker("Please enter a date", selection: $wakeUp)

.labelsHidden()

}

}

The labelsHidden is to hide the label in case you need it, which is better than just leaving a blank space in the first parameter.

You can also specify that you only want to see the date part or the hour and minutes like this:

**var** body: **some** View {

DatePicker("Please enter a date", selection: $wakeUp,

displayedComponents: .date)

}

You can work with date ranges, too. For example, you could define a function to have a range of dates from today to tomorrow:

**func** exampleDates() {

// create a second Date instance set to one day in seconds from now

**let** tomorrow = Date.now.addingTimeInterval(86400)

// create a range from those two

**let** range = Date.now...tomorrow

}

You can define a date range from today on, so the user cannot select past dates:

**var** body: **some** View {

DatePicker("Please enter a date", selection: $wakeUp,

in: Date.now...

)

}

## Working with Dates

When working with dates, you must try to use the Apple’s framework to do calculations, instead of doing your own calculations, like for example:

let now = Date.now

let tomorrow = Date.now.addingTimeInterval(86400)

let range = now...tomorrow

In the previous example, we are getting now and tomorrow by adding 86400 seconds, which is what you have in 24 hours, but you must try to avoid using this kind of calculous.

There are some challenges when working with dates like:

1. Choosing a sensible default “wake up” time.

2. Reading the hour and minute they want to wake up.

3. Showing their suggested bedtime neatly formatted.

For first challenge, we can use DateComponents, a special type that allows you read or write different parts of the date instead the whole thing:

var components = DateComponents()

components.hour = 8

components.minute = 0

let date = Calendar.current.date(from: components) ?? Date.now

In the previous example, we say, we want a date with 8 hours and 0 minutes, and if getting that date fails, give me the current date.

For the second challenge, to get a date or hours from a DatePicker or the current date, we can use DateComponents too:

let components =

Calendar.current.dateComponents([.hour, .minute], from:

someDate)

let hour = components.hour ?? 0

let minute = components.minute ?? 0

someDate can be a DatePicker or Date.now, for example.

For the third challenge, for example, if we just wanted the time from a date, we would write this:

Text(Date.now, format: .dateTime.hour().minute())

When we write day().month().year() we’re asking for that data, not arranging it, and iOS will automatically format that data using the user’s preferences.

As an alternative, we can use the formatted() method directly on dates, passing in configuration options for how we want both the date and the time to be formatted, like this:

Text(Date.now.formatted(date: .long, time: .shortened))

## Training a model with Create ML

iOS 11 came with a Machine Learning framework called Core ML, a second framework was introduced 1 year later called Create ML.

Core ML is capable of handling a variety of training tasks, such as recognizing images, sounds, and even motion, but in this instance we’re going to look at tabular regression.

Machine learning is done in two steps: we train the model, then we ask the model to make predictions. Training is the process of the computer looking at all our data to figure out the relationship between all the values we have, and in large data sets it can take a long time – easily hours, potentially much longer. Prediction is done on device: we feed it the trained model, and it will use previous results to make estimates about new data.

To start training the model, open Create ML by going to the Xcode menu and choosing Open Developer Tool > Create ML. The first thing Create ML app will do is ask you to create a project or open a previous one.

With a sample file called BetterRest.csv, where we have 4 columns (wake, estimatedSleep, coffee and actualSleep), we select this file (to load it) in the Training Data box, for the target we define the actualSleep, which is the value we want to predict and to do this, we will use select the other 3 columns.

A screenshot of a computer

Description automatically generated

In the Algorithm field, there are several options, but for this example, we will use Automatic and click on the Train button.

Once the training has finished, you can go to the Evaluation tab and click on the Validation tab to see some result metrics. The value Root Mean Square Error means on average the model was able to predict suggested accurate sleep time with an error of only 170 seconds, or three minutes.

Even better, if you go to the Output tab you’ll see an our finished model has a file size of 544 bytes or so. Create ML has taken 180KB of data, and condensed it down to just 544 bytes – almost nothing.

To export the trained model, go to the Output tab and click on Get button to save the model and use it on Xcode.

## Building a basic layout

**struct** ContentView: View {

@State **private** **var** wakeUp = Date.now

@State **private** **var** sleepAmount = 8.0

@State **private** **var** coffeeAmount = 1

**var** body: **some** View {

NavigationView{

VStack {

Text("When do you want to wake up?")

.font(.headline)

DatePicker("Please enter a time", selection: $wakeUp,

displayedComponents: .hourAndMinute)

.labelsHidden()

Text("Desired amount of sleep")

.font(.headline)

Stepper("\(sleepAmount.formatted()) hours", value: $sleepAmount, in: 4...12,

step: 0.25)

Text("Daily coffee intake")

.font(.headline)

Stepper(coffeeAmount == 1 ? "1 cup" : "\(coffeeAmount) cups", value:

$coffeeAmount, in: 1...20)

}

.navigationTitle("BetterRest")

.toolbar{

Button("Calculate", action: calculateBedtime)

}

}

}

**func** calculateBedtime(){

}

}

## Connecting SwiftUI to Core ML

To use the model you have created and trained, first you need to add it to the project. To do this, just drag and drop the mlmodel file in the project navigation bar, where the other files are:

A screenshot of a computer

Description automatically generated

Make sure you select “Copy files if needed” and finish.

Xcode will create a class (that you don’t see in the project) with the same name as the file.

In the example, the model receives 3 arguments, all of them are double values.

So the function calculateBedtime will make an instance of the MLModelConfiguration to pass it to the instance of SleepCalculator (the model we have created and we have renamed to this).

We will get the wake up time, which is in the sum of hours and minutes taken from the wakeup calendar. To get this values, we use the Calendar.current.dateComponents and convert them to seconds multiplying by 60.

To get the prediction, we use the object model.prediction and pass the values we have. The prediction will have the actualSleep time in seconds. We use this time to calculate the time the user should go to bed, by subtracting this value from the wakeUp value. Then we give it a format and display the value in an alert.

**func** calculateBedtime(){

**do**{

**let** config = MLModelConfiguration()

**let** model = **try** SleepCalculator(configuration: config)

// Get the hour and minutes from the wake up time selected by the user

**let** components = Calendar.current.dateComponents([.hour, .minute], from: wakeUp)

// Get the hour and minutes in seconds

**let** hour = (components.hour ?? 0) \* 60 \* 60

**let** minute = (components.minute ?? 0) \* 60

// Feed the values to the CoreML module

**let** prediction = **try** model.prediction(wake: Double(hour + minute),

estimatedSleep: sleepAmount, coffee: Double(coffeeAmount))

// Get the time the user needs to go to sleep by substracting the actualSleep from the wakeup value

// The actualSleep is in seconds

**let** sleepTime = wakeUp - prediction.actualSleep

alertTitle = "Your ideal bedtime is..."

alertMessage = sleepTime.formatted(date: .omitted, time: .shortened)

} **catch** {

alertTitle = "Error"

alertMessage = "Sorry, there was a problem calculating your bedtime"

}

showAlert = **true**

}

## Cleaning up the UI

In the example, we add a computed property to the ContentView struct, that contains a Date value referencing 7am of the current day:

**static** **var** defaultWakeTime: Date {

**var** components = DateComponents()

components.hour = 7

components.minute = 0

**return** Calendar.current.date(from: components) ?? Date.now

}

So, the wakeUp property, now looks like this:

@State **private** **var** wakeUp = defaultWakeTime

The defaultWakeTime property is static to avoid accessing one property from inside another one. In other words, it belongs to the ContentView struct itself instead a single instance of that struct.

For the style, we can use a Form instead the VStack and separate every title and data field into a VStack:

**var** body: **some** View {

NavigationView{

Form {

VStack(alignment: .leading, spacing: 0) {

Text("When do you want to wake up?")

.font(.headline)

DatePicker("Please enter a time", selection: $wakeUp,

displayedComponents: .hourAndMinute)

.labelsHidden()

}

VStack(alignment: .leading, spacing: 0) {

Text("Desired amount of sleep")

.font(.headline)

Stepper("\(sleepAmount.formatted()) hours", value: $sleepAmount, in: 4...12,

step: 0.25)

}

VStack(alignment: .leading, spacing: 0) {

Text("Daily coffee intake")

.font(.headline)

Stepper(coffeeAmount == 1 ? "1 cup" : "\(coffeeAmount) cups", value:

$coffeeAmount, in: 1...20)

}

}

.navigationTitle("BetterRest")

.toolbar{

Button("Calculate", action: calculateBedtime)

}

.alert(alertTitle, isPresented: $showAlert){

Button("OK") { }

} message: {

Text(alertMessage)

}

}

}

# Project 05 – Word Scramble

The next project will show players a random eight-letter word and ask them to make words out of it. Along the way you’ll meet List, onAppear(), Bundle, fatalError(), and more – all useful skills that you’ll use for years to come. You’ll also get some practice with @State, NavigationView, and more.

## Introduction to List

The job of **List** is to provide a scrolling table of data. Just like Form, you can provide List a selection of static and dynamic views:

List {

Section("Section 1") {

Text("Static row 1")

Text("Static row 2")

}

Section("Section 2") {

ForEach(0..<5) {

Text("Dynamic row \($0)")

}

}

Section("Section 3") {

Text("Static row 3")

Text("Static row 4")

}

}

You’ll notice that this list looks similar to the form we had previously, but we can adjust how the list looks using the **listStyle()** modifier, like this:

.listStyle(.grouped)

One thing List can do that Form can’t is to generate its rows entirely from dynamic content without needing a ForEach. Example:

List(0..<5) {

Text("Dynamic row \($0)")

}

Lists work well with arrays because SwiftUI can identify each row uniquely based on its position in the range. When working with an array of data, SwiftUI still needs to know how to identify each row uniquely, so if one gets removed it can simply remove that one rather than having to redraw the whole list. This is where the id parameter comes in, and it works identically in both List and ForEach – it lets us tell SwiftUI exactly what makes each item in the array unique.

When working with arrays of strings and numbers, the only thing that makes those values unique is the values themselves. When working with this kind of list data, we use id: \.self like this:

**struct** ContentView: View {

**let** people = ["Finn", "Leia", "Luke", "Rey"]

**var** body: **some** View {

List(people, id: \.**self**) {

Text($0) }

}

}

## Loading resources from your app bundle

When Xcode builds your iOS app, it creates something called a “bundle”. It allows the system to store all the files for a single app in one place – the binary code.

When you want to look in a bundle for a file you placed there, you can use a new data type called URL. If we want to read the URL for a file in our main app bundle, we use Bundle.main.url(). If the file exists, it will be sent back, otherwise, we’ll get nil.

What’s inside the URL doesn’t really matter, because iOS uses paths that are impossible to guess – our app lives in its own sandbox, and we shouldn’t try to read outside of it.

**func** loadFile(){

**if** **let** fileURL = Bundle.main.url(forResource: "some-file", withExtension: "txt"){

**if** **let** fileContents = **try**? String(contentsOf: fileURL){

//fileContents is a String

}

}

}

Once we have a URL, we can load it into a string with a special initializer: **String(contentsOf:)**. We give this a file URL, and it will send back a string containing the contents of that file if it can be loaded. If it *can’t* be loaded it throws an error, so you you need to call this using **try** or **try?**

## Working with Strings

Swift gives us a method called **components(separatedBy:)** that can converts a single string into an array of strings by breaking it up wherever another string is found. For example, this will create the array **["a", "b", "c"]**:

let input = "a b c"

let letters = input.components(separatedBy: " ")

Once you have created the array with the components function, you can get a random element from it by using randomElement():

let letter = letters.randomElement()

This method returns an optional string, so you must either unwrap or use it with nil coalescing.

Another useful string method is **trimmingCharacters(in:)**, which asks Swift to remove certain kinds of characters from the start and end of a string. This uses a new type called **CharacterSet**, but most of the time we want one particular behavior: removing whitespace and new lines – this refers to spaces, tabs, and line breaks, all at once.

let trimmed =

letter?.trimmingCharacters(in: .whitespacesAndNewlines)

Another string functionality we can use is the spell checker, provided by the class UITextChecker. The class comes from UIKit and it is written in Objective-C.

**func** testCheckSpell(){

// 1. You must crete a word to check and an instance of UITextChecker

**let** word = "swift"

**let** checker = UITextChecker()

// 2. Tell the checker ow much of our string we want to check

**let** range = NSRange(location: 0, length: word.utf16.count)

// 3. Ask our text checker to report where it found any misspellings in our word

**let** misspelledRange = checker.rangeOfMisspelledWord(in: word, range: range, startingAt: 0, wrap: **false**, language: "en")

// 4. Check that there are no spelling mistake

**let** allGood = misspelledRange.location == NSNotFound

}

## Adding to a list of Words

In the following code, we have a method called addNewWord that will:

1. Lowercase newWord and remove any whitespace
2. Check that it has at least 1 character, otherwise exit.
3. Insert that word at position 0 in the usedWords array.
4. Set newWord back to be an empty string

**struct** ContentView: View {

@State **private** **var** usedWords = [String]()

@State **private** **var** rootWord = ""

@State **private** **var** newWord = ""

**var** body: **some** View {

List {

Section {

TextField("Enter your word", text: $newWord)

// Do not allow capitalization for the word

.autocapitalization(.none)

}

Section{

ForEach(usedWords, id: \.**self**) { word **in**

HStack{

// This line allows you to show the count of letters of the word in a circle

Image(systemName: "\(word.count).circle")

Text(word)

}

}

}

}

.navigationTitle(rootWord)

// This modifier adds an action to perform when the user submits a value to this view.

.onSubmit(addNewWord)

}

**func** addNewWord(){

// Lowercase and trim the word, to make sure we don't add duplicate words with case differences

**let** answer = newWord.lowercased().trimmingCharacters(in: .whitespacesAndNewlines)

// exit if the remaining string is empty

**guard** answer.count > 0 **else** { **return** }

// Extra validation

withAnimation{

usedWords.insert(answer, at: 0)

}

newWord = ""

}

}

A couple of notes about the previous code:

* The modifier autocapitalization can be used when you need to autocapitalice a sentence or word, or avoid it by using the .none value.
* The .onSubmit adds an action to perform when the user submits a value. In the example is added to the List, but it can be added to the TextField or in any other element in the screen.
* The withAnimation block allows you to add an animation to the action of inserting the new word to the usedWord which eventually, will update the list.

## Running code when our app launches

For this project, once you have downloaded the start.txt file, drag it to the project pane, and use the following options:

A screenshot of a computer

Description automatically generated

In the following function, we define the behavior for the start of the game:

**func** startGame(){

**if** **let** startWordsURL = Bundle.main.url(forResource: "start", withExtension: "txt"){

// Load start text into a string

**if** **let** startWords = **try**? String(contentsOf: startWordsURL){

// Get all words by spliting starWords

**let** allWords = startWords.components(separatedBy: "\n")

// Get a random word from the array

rootWord = allWords.randomElement() ?? "silkworm"

**return**

}

}

// If the start file was not found or cannot be loaded, a fatal error should

// have happened

fatalError("Could not load start.txt from bundle")

}

In the previous function:

* We find the start.txt with the Bundle.main.url method.
* Load the file into a string.
* Split that string into array of strings, where each element is a word.
* Pick one random word to the assigned as rootWord or use a default word if array is empty.

In the last part of the function, if the file was not found or there was an error loading it, we show a fatalError which will crash the app.

To call this function from the start, we make use of the onAppear function as a modifier for the List, to start the game:

.onAppear(perform: startGame)

## Validating words with UITextChecker

In the project, we will include 4 more functions:

* isOriginal will validate the word has not been used
* isPossible, will validate the word sent is possible to use based on the letter from the rootWord.
* isReal, will validate that the word exists or is not misspelled using UITextChecker. If there are no misspells in the word, it exists.
* A function to show the alert with the error title and error message.

Finally, we use these function from the addNewWord function.

This is the full code:

@State **private** **var** usedWords = [String]()

@State **private** **var** rootWord = ""

@State **private** **var** newWord = ""

@State **private** **var** errorTitle = ""

@State **private** **var** errorMessage = ""

@State **private** **var** showingError = **false**

**var** body: **some** View {

NavigationStack{

List {

Section {

TextField("Enter your word", text: $newWord)

// Do not allow capitalization for the word

.autocapitalization(.none)

}

Section{

ForEach(usedWords, id: \.**self**) { word **in**

HStack{

// This line allows you to show the count of letters of the word in a circle

Image(systemName: "\(word.count).circle")

Text(word)

}

}

}

}

.navigationTitle(rootWord)

// This modifier adds an action to perform when the user submits a value to this view.

.onSubmit(addNewWord)

// This modifier performs a function before the view appears

.onAppear(perform: startGame)

.alert(errorTitle, isPresented: $showingError){

Button("OK", role: .cancel) {}

} message: {

Text(errorMessage)

}

}

}

**func** addNewWord(){

// Lowercase and trim the word, to make sure we don't add duplicate words with case differences

**let** answer = newWord.lowercased().trimmingCharacters(in: .whitespacesAndNewlines)

// exit if the remaining string is empty

**guard** answer.count > 0 **else** { **return** }

**guard** isOriginal(word: answer) **else** {

wordError(title: "Word used already", message: "Be more original")

**return**

}

**guard** isPossible(word: answer) **else** {

wordError(title: "Word not possible", message: "You can't spell that word from '\(rootWord)'!")

**return**

}

**guard** isReal(word: answer) **else** {

wordError(title: "Word not recognized", message: "You can't just make them up, you know!")

**return**

}

withAnimation{

usedWords.insert(answer, at: 0)

}

newWord = ""

}

**func** startGame(){

**if** **let** startWordsURL = Bundle.main.url(forResource: "start", withExtension: "txt"){

// Load start text into a string

**if** **let** startWords = **try**? String(contentsOf: startWordsURL){

// Get all words by spliting starWords

**let** allWords = startWords.components(separatedBy: "\n")

// Get a random word from the array

rootWord = allWords.randomElement() ?? "silkworm"

**return**

}

}

// If the start file was not found or cannot be loaded, a fatal error should

// have happened

fatalError("Could not load start.txt from bundle")

}

// This function validates if the word has been already used

**func** isOriginal(word: String) -> Bool{

!usedWords.contains(word)

}

// This function validates if the word passed uses all letter available in the rootWord

**func** isPossible(word: String) -> Bool {

**var** tempWord = rootWord

**for** letter **in** word {

**if** **let** pos = tempWord.firstIndex(of: letter){

tempWord.remove(at: pos)

} **else** {

**return** **false**

}

}

// Every letter in the word was found

**return** **true**

}

// This function will validates if the word entered is real

**func** isReal(word: String) -> Bool {

**let** checker = UITextChecker()

**let** range = NSRange(location: 0, length: word.utf16.count)

**let** misspelledRange = checker.rangeOfMisspelledWord(in: word, range: range, startingAt: 0, wrap: **false**, language: "en")

// If the word is not misspelled, then is a real word

**return** misspelledRange.location == NSNotFound

}

// This will show the alert with the error message passed as parameters

**func** wordError(title: String, message: String){

errorTitle = title

errorMessage = message

showingError = **true**

}

}

# Project 06 - Animation

## Implicit Animations

An implicit animation is the simplest type in SwiftUI, you tell the view, if someone wants to animate you, here’s how you should respond. In the following example, we define a rounded button, and we will use the scaleEffect along with a state property to increase the size of the button. The value in this scaleEffect goes from 0 to 1.0 which represents the 100% of the normal size of the button.

To make this size increasing process to be animated, we use animation modifier. The implicit animation takes effect on all properties of the view that change, so by adding another animation modifier, like the blur, you will see how the button scales and blurs smoothly.

**struct** ContentView: View {

@State **private** **var** animationAmount = 1.0

**var** body: **some** View {

Button("Tap me"){

animationAmount += 1

}

.padding(50)

.background(.red)

.foregroundColor(.white)

.clipShape(Circle())

.scaleEffect(animationAmount)

.blur(radius: (animationAmount - 1) \* 3)

.animation(.default, value: animationAmount)

}

}

## Customizing Animations

We can control the type of animation used by passing in different values to the modifier. For example, we could use **.easeOut** to make the animation start fast then slow down to a smooth stop:

.animation(.easeOut, value: animationAmount)

Remember that implicit animations always need to watch a particular value, otherwise they will be triggered for every small change, even rotating the device.

There are even spring animations, that cause the movement to overshoot then return to settle at its target. You can control the initial stiffness of the spring (which sets its initial velocity when the animation starts), and also how fast the animation should be “damped”

.animation(.interpolatingSpring(stiffness: 50, damping: 1),

value: animationAmount)

You can also specify the number of seconds for the animation’s duration:

.animation(.easeInOut(duration: 2), value: animationAmount)

You can also use the modifier .delay to set a specific time before the animation starts.

For continuous animations, there is a **repeatForever()** modifier that can be used like this

.animation(

.easeInOut(duration: 1)

.repeatForever(autoreverses: true),

value: animationAmount

)

We can use these **repeatForever()** animations in combination with **onAppear()** to make animations that start immediately and continue animating for the life of the view.

struct ContentView: View {

@State private var animationAmount = 1.0

var body: some View {

Button("Tap me"){

//animationAmount += 1

}

.padding(50)

.background(.red)

.foregroundColor(.white)

.clipShape(Circle())

.overlay(

Circle()

.stroke(.red)

.scaleEffect(animationAmount)

.opacity(2 - animationAmount)

.animation(

.easeInOut(duration: 1)

.repeatForever(autoreverses: false),

value: animationAmount

)

)

.onAppear{

animationAmount = 2

}

}

}

That makes a stroked red circle over our button, using an opacity value of **2 - animationAmount** so that when **animationAmount** is 1 the opacity is 1 (it’s opaque) and when **animationAmount** is 2 the opacity is 0 (it’s transparent).

## Animation Bindings

The **animation()** modifier can be applied to any SwiftUI binding, which causes the value to animate between its current and new value. This also applies to Boolean data. For example:

**struct** ContentView: View {

@State **private** **var** animationAmount = 1.0

**var** body: **some** View {

VStack {

Stepper("Scale amount", value: $animationAmount.animation(), in: 1...10)

Spacer()

Button("Tap me"){

animationAmount += 1

}

.padding(50)

.background(.red)

.foregroundColor(.white)

.clipShape(Circle())

.scaleEffect(animationAmount)

}

}

}

As you can see, the stepper can move **animationAmount** up and down, and tapping the button will add 1 to it – they are both tied to the same data, which in turn causes the size of the button to change. However, tapping the button changes **animationCount** immediately, so the button will just jump up to its larger size. In contrast, the stepper is bound to **$animationAmount.animation()**, which means SwiftUI will automatically animate its changes.

SwiftUI is examining the state of our view before the binding changes, examining the target state of our views *after* the binding changes, then applying an animation to get from point A to point B.

This is why we can animate a Boolean changing: Swift isn’t somehow inventing new values between false and true, but just animating the view changes that occur as a result of the change.

These binding animations use a similar **animation()** modifier that we use on views, so you can go to town with animation modifiers if you want to:

Stepper("Scale amount", value: $animationAmount.animation(

.easeInOut(duration: 1)

), in: 1...10)

**Tip:** With this variant of the **animation()** modifier, we don’t need to specify which value we’re watching for changes – it’s literally attached to the value it should watch!

## Creating explicit animations

You can also use 3D animation with some modifiers SwiftUI offers, like rotation3DEffct():

* If we skewer the view through the X axis (horizontally) then it will be able to spin forwards and backwards.
* If we skewer the view through the Y axis (vertically) then it will be able to spin left and right.
* If we skewer the view through the Z axis (depth) then it will be able to rotate left and right.

**struct** ContentView: View {

@State **private** **var** animationAmount = 0.0

**var** body: **some** View {

Button("Tap Me") {

withAnimation(.interpolatingSpring(stiffness: 5, damping: 1)){

animationAmount += 360

}

}

.padding(50)

.background(.red)

.foregroundColor(.white)

.clipShape(Circle())

.rotation3DEffect(.degrees(animationAmount), axis: (x: 0, y: 1,

z: 0))

}

}

Along with the rotation3DEffect modifier, we need to use withAnimation at the moment of changing the value for the animationAmount, if not, the change will happen immediately.

The interpolatingSpring add another effect to use a spring animation.

## Controlling the animation stack

It is important to remember that the order of modifiers matters, and it also applies to animation modifiers.

Besides that, you can have as many animation modifiers as you want, and the order of these modifiers will matter.

Now for the fun part: if we apply multiple **animation()** modifiers, each one controls everything before it up to the next animation. This allows us to animate state changes in all sorts of different ways rather than uniformly for all properties.

**struct** ContentView: View {

@State **private** **var** enabled = **false**

**var** body: **some** View {

Button("Tap Me") {

enabled.toggle()

}

.frame(width: 200, height: 200)

.background(enabled ? .blue : .red)

.animation(.default, value: enabled)

.foregroundColor(.white)

.clipShape(RoundedRectangle(cornerRadius: enabled ? 60 : 0))

.animation(.interpolatingSpring(stiffness: 10, damping: 1), value: enabled)

}

}

For even more control, it’s possible to disable animations entirely by passing **nil** to the modifier.

Button("Tap Me") {

enabled.toggle()

}

.frame(width: 200, height: 200)

.background(enabled ? .blue : .red)

.animation(nil, value: enabled)

.foregroundColor(.white)

.clipShape(RoundedRectangle(cornerRadius: enabled ? 60 : 0))

.animation(.interpolatingSpring(stiffness: 10, damping: 1),

value: enabled)

## Animating gestures

SwiftUI allows you to add and animate gestures, like the drag gesture in the following example:

**struct** ContentView: View {

@State **private** **var** dragAmount = CGSize.zero

**var** body: **some** View {

LinearGradient(gradient: Gradient(colors: [.yellow, .red]), startPoint: .topLeading, endPoint: .bottomTrailing)

.frame(width: 300, height: 200)

.clipShape(RoundedRectangle(cornerRadius: 10))

.offset(dragAmount)

.gesture(

DragGesture()

.onChanged { dragAmount = $0.translation }

.onEnded { \_ **in** dragAmount = .zero }

)

.animation(.spring(), value: dragAmount)

}

}

In the previous example, besides adding the drag gesture to the “card”, the object is animated with a spring animation.

You can also animate just the onEnded method:

LinearGradient(gradient: Gradient(colors: [.yellow, .red]), startPoint: .topLeading, endPoint: .bottomTrailing)

.frame(width: 300, height: 200)

.clipShape(RoundedRectangle(cornerRadius: 10))

.offset(dragAmount)

.gesture(

DragGesture()

.onChanged { dragAmount = $0.translation }

**.onEnded { \_ in**

**withAnimation{**

**dragAmount = .zero**

**}**

**}**

)

You can also add a delay to the animations:

struct ContentView: View {

let letters = Array("Hello, SwiftUI")

@State private var enabled = false

@State private var dragAmount = CGSize.zero

var body: some View {

HStack(spacing: 0){

ForEach(0..<letters.count) { num in

Text(String(letters[num]))

.padding(5)

.font(.title)

.background(enabled ? .blue : .red)

.offset(dragAmount)

.animation(

**.default.delay(Double(num) / 20),**

value: dragAmount)

}

}

.gesture(

DragGesture()

.onChanged{ dragAmount = $0.translation }

.onEnded{ \_ in

dragAmount = .zero

enabled.toggle()}

)

}

}

## Showing and hiding Views with transitions

You can animate when a view is hiding or showing by using the withAnimation modifier along with the transition modifier. This transition allows simple like .scale or asymmetric transitions when you define the type of transition when it is inserting the element and when it is removing it:

**struct** ContentView: View {

@State **private** **var** isShowingRed = **false**

**var** body: **some** View {

VStack{

Button("Tap Me") {

withAnimation{

isShowingRed.toggle()

}

}

**if** isShowingRed {

Rectangle()

.fill(.red)

.frame(width: 200, height: 200)

.transition(.asymmetric(insertion: .scale, removal: .opacity))

}

}

}

}

## Building custom transitions using ViewModifier

SwiftUI allows you to create new transitions by adding or removing views using custom animations. This can be made by using the .modifier transition, which accepts any view modifier we want, but we need to instantiate the modifier, in other words, we need to create it.

So, in the following example, we will be using a custom modifier, by creating a CornerRotateModifier struct that has an anchor point to control where the rotation should take place, and an amount to control how much rotation should be applied.

The .clipped means that when the view rotates the parts that are lying outside the natural rectangle won’t get drawn.

The use of the CornerRotateModifier is made using an extension to AnyTransition, and making it to rotate from -90 to 0 on its top leading corner.

The pivot animation is attached to any view using the .transition modifier.

**struct** CornerRotateModifier: ViewModifier {

**let** amount: Double

**let** anchor: UnitPoint

**func** body(content: Content) -> **some** View {

content

.rotationEffect(.degrees(amount), anchor: anchor)

.clipped()

}

}

**extension** AnyTransition {

**static** **var** pivot: AnyTransition{

.modifier(active: CornerRotateModifier(amount: -90, anchor: .topLeading),

identity: CornerRotateModifier(amount: 0, anchor: .topLeading))

}

}

**struct** ContentView: View {

@State **private** **var** isShowingRed = **false**

**var** body: **some** View {

ZStack{

Rectangle()

.fill(.blue)

.frame(width: 200, height: 200)

**if** isShowingRed {

Rectangle()

.fill(.red )

.frame(width: 200, height: 200)

.transition(.pivot)

}

}

.onTapGesture {

withAnimation{

isShowingRed.toggle()

}

}

}

}

# Project 07 – iExpense

## Introduction

SwiftUI’s **State** property wrapper is designed for simple data that is local to the current view, but as soon as you want to share data between views it stops being useful.

For example:

struct ContentView: View {

@State private var user = User()

var body: some View {

VStack {

Text("Your name is \(user.firstName) \(user.lastName).")

TextField("First name", text: $user.firstName)

TextField("Last name", text: $user.lastName)

}

} }

With a struct, when a value changes, the whole struct changes.

For SwiftUI developers, what this means is that if we want to share data between multiple views – if we want two or more views to point to the same data so that when one changes they all get those changes – we need to use classes rather than structs.

So, instead of having a struct, we use class:

**class** User {

**var** firstName = "Bilbo"

**var** lastName = "Baggins"

}

When we use **@State**, we’re asking SwiftUI to watch a property for changes. So, if we change a property’s value, SwiftUI will re-invoke the body property of the view. Basically, a change in a property creates a new instance of the struct. @State was able to spot that change, and automatically reloaded our view. Now that we have a class, that behavior no longer happens: Swift can just modify the value directly.

For structs we need the mutating keyword for struct methods that modify properties, because the struct itself is constant. Swift needs to be able to destroy and recreate the whole struct when a property changes, and that is not possible for constant structs. Classes don’t need the mutating keyword, because even if the class instance is marked as constant Swift can still modify variable properties.

## Sharing SwiftUI state with @StateObject

To share data in SwiftUi, you have 3 property wrappers to help you with this: @StateObject, @ObervedObject and @EnvironmentObject.

The @State is designed to track local structs rather than classes. When using classes, you need to tell SwiftUI what views should be reloaded when they change, by using property wrappers. In the following example we have our class User1 with 2 properties. Whenever they change, we want to notify any view watching our class that a change has happened. We can do this by using @Published property observer.

Then, the @StateObject property wrapper tells SwiftUI that we’re creating a new class instance that should be watched for any change announcement.

**class** User1 : ObservableObject {

**@Published** **var** firstName = "Bilbo"

**@Published** **var** lastName = "Baggins"

}

**struct** StateObjectExample: View {

**@StateObject** **var** user = User1()

**var** body: **some** View {

VStack {

Text("Your name is \(user.firstName) \(user.lastName)")

TextField("First Name", text: $user.firstName)

TextField("Last Name", text: $user.lastName)

}

}

}

To use the @StateObject we need to conform the ObservableObjet protocol in the class.

As you’ve seen, rather than just using @State to declare local state, we now take three steps:

* Make a class that conforms to the ObservableObject protocol.
* Mark some properties with @Published so that any views using the class get updated when they change.
* Create an instance of our class using the @StateObject property wrapper.

The end result is that we can have our state stored in an external object, and, even better, we can now use that object in multiple views and have them all point to the same values.

However, there is something important: @StateObject tells SwiftUI that we’re creating a new class instance that should be watched for any change announcements, but that should only be used when you’re *creating* the object like we are with our **User1** instance.

When you want to use a class instance elsewhere – when you’ve created it in view A using **@StateObject** and want to use that same object in view B – you use a slightly different property wrapper called **@ObservedObject**.

## Showing and hiding views

One of the ways to show views in SwiftUI is using a sheet: a new view presented on top of our existing one. Sheets work like alerts, but we define the conditions under which sheet should be shown and when those conditions become true or false.

**import** SwiftUI

**struct** SecondView: View {

@Environment(\.dismiss) **var** dismiss

**let** name: String

**var** body: **some** View {

VStack {

Text("Hello, \(name)!")

Button("Dismiss") {

dismiss()

}

}

}

}

**struct** FirstView: View {

@State **private** **var** showingSheet = **false**

**var** body: **some** View {

Button("Show Sheet") {

showingSheet.toggle()

}

.sheet(isPresented: $showingSheet){

SecondView(name: "Esteban")

}

}

}

#Preview {

FirstView()

}

First thing to note is to have a state to track whether the sheet is showing or not (showingSheet var).

Then, we toggle its value with the button.

Third, we need to attach our sheet somewhere to our view hierarchy. We do it with the .sheet(isPresented…) modifier.

We then define the second view. In this view, we can pass parameters and make the view dismiss itself. To dismiss another view, we need another property wrapper: @Environment. It allows us to create properties that store values provided to us externally.

We need to ask the environment to dismiss our view, because it might have been presented in any number of different ways. So, we’re effectively saying “hey, figure out how my view was presented, then dismiss it appropriately.”

To try it out add this property to **SecondView**, which creates a property called **dismiss** based on a value from the environment.

## Deleting items using onDelete()

In SwiftUI we can use the onDelete() to remove objects from a collection. Mostly exclusive for List and ForEach, but there is another place we can use it.

**struct** OnDeleteExample: View {

@State **private** **var** numbers = [Int]()

@State **private** **var** currentNumber = 1

**var** body: **some** View {

NavigationView {

VStack {

List {

ForEach(numbers, id: \.**self**) {

Text("Row \($0)")

}

.onDelete(perform: removeRows)

}

Button("Add Number") {

numbers.append(currentNumber)

currentNumber += 1

}

}

.navigationTitle("onDelete()")

.toolbar {

EditButton()

}

}

}

**func** removeRows(at offsets: IndexSet){

numbers.remove(atOffsets: offsets)

}

}

In the previous example, we have a view where we can add items to a list and they are displayed using a ForEach, which is the one that has the onDelete method attached. We can add the items to the list without using the ForEach, but the onDelete modifier only exists on ForEach.

In order to make **onDelete()** work, we need to implement a method that will receive a single parameter of type **IndexSet**. This is a bit like a set of integers, except it’s sorted, and it’s just telling us the positions of all the items in the **ForEach** that should be removed. This method is the removeRows.

And finally, we tell SwiftUI to call this method when it wants to delete data from ForEach.

In the example, we also added a NavigationView to add the modifier .toolbart to the VStack and be able to use an Edit button that will show the option to delete the items from the list and not only by swiping the item from right to left.

## Storing user settings with UserDefaults

One common way to store small amount of data is called UserDefaults. The recommendation is not to store more than 512KB in there.

**UserDefaults** is perfect for storing things like when the user last launched the app, which news story they last read, or other passively collected information.

In the following code, we use UserDefaults.standard to set and get the value stored, using a string for a key name.

**struct** StoringUserSettings: View {

@State **private** **var** tapCount = **UserDefaults.standard.integer(forKey: "Tap")**

**var** body: **some** View {

Button("Tap count: \(tapCount)"){

tapCount += 1

**UserDefaults.standard.setValue(tapCount, forKey: "Tap")**

}

}

}

So, the value will be kept after the user closes the app and launches it again. But it is important to note that it is important to have a default value. Second, it takes iOS a little time to write your data to permanent storage – to actually save that change to the device. They don’t write updates immediately because you might make several back to back, so instead they wait some time then write out all the changes at once. How much time is another number we don’t know, but a couple of seconds ought to do it.

SwiftUI provides an @AppStorage property wrapper around UserDefaults, and in simple situations like this one it’s really helpful:

**struct** StoringUserSettings: View {

@AppStorage("tapCount") **private** **var** tapCount = 0

**var** body: **some** View {

Button("Tap count: \(tapCount)"){

tapCount += 1

}

}

}

There are three important things to note:

1. Our access to the **UserDefaults** system is through the **@AppStorage** property wrapper. This works like **@State**: when the value changes, it will reinvoked the **body** property so our UI reflects the new data.

2. We attach a string name, which is the **UserDefaults** key where we want to store the data. I’ve used “tapCount”, but it can be anything at all – it doesn’t need to match the property name.

3. The rest of the property is declared as normal, including providing a default value of 0. That will be used if there is existing value saved inside **UserDefaults**.

## Archiving Swift objects with Codable

When you need to store custom Swift types, you need to use something else than the @AppStorage.

For example:

**struct** UserObj: **Codable** {

**let** firstName: String

**let** lastName: String

}

**struct** CodableExample: View {

@State **private** **var** user = UserObj(firstName: "John", lastName: "Smith")

**var** body: **some** View {

Button("Save User"){

**let** encoder = JSONEncoder()

**if** **let** data = **try**? encoder.encode(user){

UserDefaults.standard.setValue(data, forKey: "UserData")

}

}

}

}

In the previous example, we defined a UserObj to store. In order to do this, we use the Codable protocol when defining the struct, which allows you to convert objects into plain text and back again.

Swift will automatically generate some code for us that will archive and unarchive **User** instances for us as needed, but we still need to tell Swift *when* to archive and what to do with the data.

This part of the process is powered by a new type called **JSONEncoder**. Its job is to take something that conforms to **Codable** and send back that object in JSON format. The Codable protocol doesn’t require that we use JSON, and in fact other formats are available, but it is by far the most common.

To convert the user data we use the encode() method of JSONEncoder, this might throw errors, so it should be called with try or try? To handle errors. That accesses UserDefaults directly rather than going through @AppStorage

## The project

First, we need to define a struct that will be used to handle the expense items.

The first one is the ExpenseItem that will represent every item in the list. This will be using a UUID (Universally Unique Identifier) to identify every item. The struct will conform the protocol called Identifiable, which only requirement is to have a property of type UUID. With this, we are telling SwiftUI that this type can be identified uniquely, this will help us when we need to delete the item from the list:

**struct** ExpenseItem: Identifiable, Codable {

// Generate a UUID automatically for every item

**let** id = UUID()

**let** name: String

**let** type: String

**let** amount: Double

}

The previous struct also conforms Codable, so we can encode and decode the item in order to save it to the user defaults.

Then, we define a class called Expenses that will conform the ObservableObject:

**class** Expenses: ObservableObject {

@Published **var** items = [ExpenseItem](){

**didSet** {

**if** **let** encoded = **try**? JSONEncoder().encode(items) {

UserDefaults.standard.set(encoded, forKey: "Items")

}

}

}

// Have an initializer that will read the values and load them into the items array

**init**(){

// Read the data from user defaults, if it exists

**if** **let** savedItems = UserDefaults.standard.data(forKey: "Items"){

// If the data exists, try to decode it

**if** **let** decodedItems = **try**? JSONDecoder().decode([ExpenseItem].**self**, from: savedItems){

items = decodedItems

**return**

}

}

// If it was not able to load the saved data, load an empty array

items = []

}

}

In the previous class, we create an array of ExpenseItems that we will use to store all the items of the list. This object is created here and shared with other views, like the AddView and the ContentView and that is why it has the @Published.

We’re going to leverage four important technologies to help us save and load data in a clean way:

* The **Codable** protocol, which will allow us to archive all the existing expense items ready to be stored.
* UserDefaults, which will let us save and load that archived data.
* A custom initializer for the Expenses class, so that when we make an instance of it we load any saved data from **UserDefaults**
* A **didSet** property observer on the items property of Expenses, so that whenever an item gets added or removed we’ll write out changes.

The AddView has the following code:

**struct** AddView: View {

// Observed an expense object passed to the view which will not be

// created here

@ObservedObject **var** expenses: Expenses

@Environment(\.dismiss) **var** dismiss

@State **private** **var** name = ""

@State **private** **var** type = "Personal"

@State **private** **var** amount = 0.0

**let** types = ["Business", "Personal"]

**var** body: **some** View {

NavigationView {

Form {

TextField("Name", text: $name)

Picker("Type", selection: $type) {

ForEach(types, id: \.**self**){

Text($0)

}

}

TextField("Amount", value: $amount, format: .currency(code: "USD"))

.keyboardType(.decimalPad)

}

.navigationTitle("Add new expense")

.toolbar {

Button("Save"){

**let** item = ExpenseItem(name: name, type: type, amount: amount)

expenses.items.append(item)

dismiss()

}

}

}

}

}

#Preview {

// For the preview, pass a dummy value

AddView(expenses: Expenses())

}

This is the screen we use to add a new item to the list. It will be shown every time the user clicks on plus button in the main screen (ContentView).

Some things to note in this view:

* We add an @ObservedObject to be able to work with the shared array.
* The @Environment property wrapper is used to execute the dismiss action of the view. This causes the **showingAddExpense** Boolean in **ContentView** to go back to false, and hides the **AddView**. To execute this, we just need to call the property dismiss() on the save button.
* We define a Form for the values of the expense item, and we add a Button to save it.
* In the Save button code, we call the dismiss property which will call the dismiss action for the view.
* The Preview code needs that we supply a dummy expenses object to be able to show the preview of the screen.

Finally, the main screen (ContentView) looks like this:

**struct** ContentView: View {

// Make an instance of expenses. The @StateObject asks SwiftUI

// to watch the object for any change announcements

@StateObject **var** expenses = Expenses()

@State **private** **var** showingAddExpense = **false**

// Get only business items

/\*var businessItems: [ExpenseItem] {

return expenses.items.filter { $0.type == "Business" }

}\*/

**var** body: **some** View {

NavigationView {

List {

Section("Business") {

ForEach(expenses.items , id: \.id) { item **in**

**if** item.type == "Business"{

HStack {

VStack {

Text(item.name)

.font(.headline)

.multilineTextAlignment(.leading)

Text(item.type)

.multilineTextAlignment(.leading)

}

.frame(maxWidth: .infinity, alignment: .leading)

Spacer()

Text(item.amount, format: .currency(code: Locale.current.currency?.identifier ?? "USD"))

.foregroundColor(getAmountColor(item.amount))

}

}

}

.onDelete(perform: removeItems)

}

Section("Personal") {

ForEach(expenses.items, id: \.id) { item **in**

**if** item.type == "Personal" {

HStack {

VStack {

Text(item.name)

.font(.headline)

.multilineTextAlignment(.leading)

Text(item.type)

.multilineTextAlignment(.leading)

}

.frame(maxWidth: .infinity, alignment: .leading)

Spacer()

Text(item.amount, format: .currency(code: Locale.current.currency?.identifier ?? "USD"))

.expenseStyle(for: item)

//.foregroundColor(getAmountColor(item.amount))

}

}

}

.onDelete(perform: removeItems)

}

}

.navigationTitle("iExpense")

.toolbar {

Button {

showingAddExpense = **true**

} label: {

Image(systemName: "plus")

}

}

.sheet(isPresented: $showingAddExpense){

AddView(expenses: expenses)

}

}

}

**func** getAmountColor(\_ amount: Double) -> Color {

**if** amount <= 10.0 {

**return** .green

} **else** **if** amount > 10 && amount <= 100 {

**return** .black

} **else** {

**return** .red

}

}

**func** removeItems(at offsets: IndexSet){

expenses.items.remove (atOffsets: offsets)

}

}

/\*\*

ViewModifier to define a specific style depending of the amount of the expense item

\*/

**struct** ExpenseStyle: ViewModifier {

**let** expenseItem: ExpenseItem

**func** body(content: Content) -> **some** View {

**switch** expenseItem.amount {

**case** 0..<10:

content

.foregroundColor(.green)

.bold()

**case** 10..<100:

content.foregroundColor(.black)

**default**:

content

.foregroundColor(.red)

.bold()

}

}

}

/\*\*

Extension to be able to use the expsnseStyle

\*/

**extension** View {

**func** expenseStyle(for expenseItem: ExpenseItem) -> **some** View {

modifier(ExpenseStyle(expenseItem: expenseItem))

}

}

Important things to note in this screen:

* The var expenses has a @StateObject because we are asking SwiftUI to watch for this object for any change.
* The button to add a new item is in the toolbar and what it does is to turn the showingAddExpense to true, this way, the .sheet is presented allowing the user to add the new item and we pass the expenses variable, the shared object that was created on the Expenses class.
* There are two foreach as we are separating the expense items in two sections for each type (business and personal). The list items are filtered by a simple if as if we use the filter modifier, we would be creating two different lists or arrays and the removeItem method will have issues locating the correct item to delete based on the index.
* The amounts have a different style based on the amount and using a ViewModifier and an extension.

# Project 08 - Moonshot

This section will give you more experience with Codable, Lists, Text and more while you learn more about making an image fit its space correctly.

## Resizing Images to Fit the Screen using GeometryReader

When we create an **Image** view in SwiftUI, it will automatically size itself according to the dimensions of its contents. This is sometimes what you want, but mostly you’ll want to show the image at a lower size.

When you load an image with the Image view, and even if you use the frame modifier to set a width and height, it could not fit and parts of the image may not be seen depending of the size of the image. If you use the resizable modifier makes thing better, but not enough, as the image may look squashed.

To make an image resize itself proportionally, you can use scaleToFit and scaleToFill modifiers.

All this works great if we want fixed-sized images, but very often you want images that automatically scale up to fill more of the screen in one or both dimensions. That is, rather than hard-coding a width of 300, what you *really* want to say is “make this image fill 80% of the width of the screen.” SwiftUI gives us a dedicated type for this called GeometryReader.

**GeometryReader** is a view just like the others we’ve used, except when we create it we’ll be handed a **GeometryProxy** object to use. This lets us query the environment: how big is the container? What position is our view? Are there any safe area insets? And so on.

In principle that seems simple enough, but in practice you need to use **GeometryReader** carefully because it automatically expands to take up available space in your layout, then positions its own content aligned to the top-left corner.

**struct** GeometryReaderExample: View {

**var** body: **some** View {

GeometryReader { geo **in**

Image("Landscape")

.resizable()

.scaledToFit()

.frame(width: geo.size.width \* 0.8)

// This second frame will contain the first one and will align it in the center

.frame(width: geo.size.width, height: geo.size.height)

}

}

}

We’ve given SwiftUI enough information that it can automatically figure out the height.

## How ScrollView lets us work with Scrolling Data

Scroll views can scroll horizontally, vertically, or in both directions, and you can also control whether the system should show scroll indicators next to them. When we place views inside scroll views, they automatically figure out the size of that content so users can scroll from one edge to the other.

**struct** CustomText: View {

**let** text: String

**var** body: **some** View {

Text(text)

}

**init**(\_ text: String){

print("Creating a new CustomText")

**self**.text = text

}

}

**struct** ScrollViewExample: View {

**var** body: **some** View {

ScrollView {

VStack(spacing: 10){

ForEach(0..<100){

CustomText("Item \($0)")

.font(.title)

}

}

.frame(maxWidth: .infinity)

}

}

}

With this code you can see you can drag the scroll view around freely, and if you scroll to the bottom you’ll also see that **ScrollView** treats the safe area just like **List** and **Form.** This all seems really straightforward, however there’s an important catch that you need to be aware of: when we add views to a scroll view they get created immediately.

If you want to avoid this happening, there’s an alternative for both **VStack** and **HStack** called **LazyVStack** and **LazyHStack** respectively. These can be used in exactly the same way as regular stacks but will load their content on-demand – they won’t create views until they are actually shown, and so minimize the amount of system resources being used.

**struct** ScrollViewExample: View {

**var** body: **some** View {

ScrollView(.vertical) {

LazyVStack(spacing: 10){

ForEach(0..<100){

CustomText("Item \($0)")

.font(.title)

}

}

.frame(maxWidth: .infinity)

}

}

}

Lazy stacks always take up as much as room as is available in our layouts, whereas regular stacks take up only as much space as is needed. One last thing: you can make horizontal scrollviews by passing .horizontal as a parameter when you make your ScrollView. Once that’s done, make sure you create a horizontal stack or lazy stack, so your content is laid out as you expect.

## Pushing new views onto the stack using NavigationLink

SwiftUI’s **NavigationView** shows a navigation bar at the top of our views, but also does something else: it lets us push views onto a view stack.

This view stack system is very different from the sheets we’ve used previously. Yes, both show some sort of new view, but there’s a difference in the *way* they are presented that affects the way users think about them.

**struct** NavigationLinkExample: View {

**var** body: **some** View {

NavigationView {

List(0..<100) { row **in**

NavigationLink {

Text("Detail \(row)")

} label: {

Text("Row \(row)")

}

}

.navigationTitle("SwiftUI")

}

}

}

With NavigationLink, the user is going to see a list of 100 elements that indicates the Row and #, and it will show a chevron icon to show the user that this is a clickable item and when the user clicks on this icon, it will show another screen with Detail and row number.

We can use NavigationLink with any kind of destination view. Even better, you’ll see that the “SwiftUI” title animates down to become a back button, and you can tap that or swipe from the left edge to go back.

Both **sheet()** and **NavigationLink** allow us to show a new view from the current one, but the *way* they do it is different and you should choose them carefully:

* **NavigationLink** is for showing details about the user’s selection, like you’re digging deeper into a topic.
* **sheet()** is for showing unrelated content, such as settings or a compose window.

## Working with hierarchical Codable data

If you want to decode a complex JSON object, where you can find arrays inside other arrays, the codable protocol is capable of decoding everything but it is important to create separate types for each level you have.

In the next example you’ll see a button which will decode a simple JSON with the JSONDecoder but the important thing is that we have two structs, one for each level and they should conform the Codable protocol:

**struct** User: Codable {

**let** name: String

**let** address: Address

}

**struct** Address: Codable {

**let** street: String

**let** city: String

}

**struct** HierarchicalCodableData: View {

**var** body: **some** View {

Button("Decode JSON") {

**let** input = """

{

"name": "John Smith",

"address": {

"street": "555, Ever Green Avenue",

"city": "Nashville"

}

}

"""

**let** data = Data(input.utf8)

**if** **let** user = **try**? JSONDecoder().decode(User.**self**, from: data){

print(user.address.street)

}

}

}

}

## How to lay out views in a scrolling grid

When you need to show scrolling rows of data in columns, you need to use two views: LazyHGrid or LazyVGrid. Creating a grid is done in two steps. First, we need to define the rows or columns we want – we only define one of the two, depending on which kind of grid we want.

In the next example, we have a vertically scrolling grid, with three columns defined with 80 points wide each one.

**struct** ScrollingGridExample: View {

**let** layout = [

GridItem(.fixed(80)),

GridItem(.fixed(80)),

GridItem(.fixed(80))

]

**var** body: **some** View {

ScrollView {

LazyVGrid(columns: layout) {

ForEach(0..<1000) {

Text("Item \($0)")

}

}

}

}

}

Once you have your lay out defined, you should place your grid inside a **ScrollView**, along with as many items as you want. Each item you create inside the grid is automatically assigned a column in the same way that rows inside a list automatically get placed inside their parent.

Grids have the ability to work across a variety of screen sizes. You can also specify a maximum range for even more control and rely on the adaptive layouts. In the following example we are using an horizontal grid using rows instead of columns:

**struct** ScrollingGridExample: View {

**let** layout = [

GridItem(.adaptive(minimum: 80, maximum: 120))

]

**var** body: **some** View {

ScrollView(.horizontal){

LazyHGrid(rows: layout) {

ForEach(0..<1000) {

Text("Item \($0)")

}

}

}

}

}

## Loading a specific kind of Codable data

As part of the project, in this chapter we will load a couple of JSON files: one for the astronauts and the other for the mission’s information. Also, several images will be uploaded into the Assets folder.

To be able to work with the astronauts file, we have created the following Austronaut struct:

**struct** Astronaut: Codable, Identifiable {

**let** id: String

**let** name: String

**let** description: String

}

To load the files, we will create a Bundle-Decodable extension for the Bundle class. This is similar to the previous way to load JSON files, but with one difference: previously we used String(contentsOf:) to load files into a string, but because Codable uses Data we are instead going to use Data(contentsOf:). It works in just the same way as String(contentsOf:): give it a file URL to load, and it either returns its contents or throws an error.

**extension** Bundle {

// Function that reads a file and returns a dictionary of type [string, astronaut]

**func** decode(\_ file: String) -> [String: Astronaut] {

// Find that file in our installed app bundle

**guard** **let** url = **self**.url(forResource: file, withExtension: **nil**) **else** {

fatalError("Failed to locate \(file) in bundle.")

}

// If file was found, try to load it

**guard** **let** data = **try**? Data(contentsOf: url) **else** {

fatalError("Failed to load \(file) from bundle.")

}

// If it was loaded, decode it

**let** decoder = JSONDecoder()

**guard** **let** loaded = **try**? decoder.decode([String: Astronaut].**self**, from: data) **else** {

fatalError("Failed to decode \(file) from bundle.")

}

**return** loaded

}

}

With this extension, it takes only one code line to load a json file in the ContentView:

**struct** ContentView: View {

**let** astronauts = Bundle.main.decode("astronauts.json")

**var** body: **some** View {

Text("\(astronauts.count)")

.padding()

}

}

## Using Generics to Load any kind of Codable Data

The Bundle extension was specific for loading the JSON file for Astronauts, however, if we want to load any other kind of file, like the missions.json file, we need to do some changes. As the other file has a different structure, we will first start by creating a struct for the Crew Role and another one for the Mission:

**struct** Mission: Codable, Identifiable {

// Nested struct cause this is only being used by Mission struct

**struct** CrewRole: Codable {

**let** name: String

**let** role: String

}

**let** id: Int

**let** launchDate: String? //Optional property

**let** crew: [CrewRole]

**let** description: String

}

In the previous struct, we defined the properties for the Mission along with the nested struct for the crew role. It is important to note that the launchDate is optional as not all the missions have it.

For the Bundle extension, we start making the method generic by adding a placeholder for certain types, this is the <T> part:

**extension** Bundle {

// Function that reads a file and returns a dictionary of type [string, astronaut]

**func** decode<T: Codable>(\_ file: String) -> T {

// Find that file in our installed app bundle

**guard** **let** url = **self**.url(forResource: file, withExtension: **nil**) **else** {

fatalError("Failed to locate \(file) in bundle.")

}

// If file was found, try to load it

**guard** **let** data = **try**? Data(contentsOf: url) **else** {

fatalError("Failed to load \(file) from bundle.")

}

// If it was loaded, decode it

**let** decoder = JSONDecoder()

**guard** **let** loaded = **try**? decoder.decode(T.**self**, from: data) **else** {

fatalError("Failed to decode \(file) from bundle.")

}

**return** loaded

}

}

Then we need to replace the parts where we had the [String: Astronaut] with just T. If you try compiling this code, you’ll see an error in Xcode: “Instance method 'decode(\_:from:)' requires that 'T' conform to 'Decodable’”. What it means is that **T** could be anything: it could be a dictionary of astronauts, or it could be a dictionary of something else entirely. The problem is that Swift can’t be sure the type we’re working with conforms to the **Codable** protocol.

Fortunately we can fix this with a *constraint*: we can tell Swift that **T** can be whatever we want, as long as that thing conforms to **Codable**. That way Swift knows it’s safe to use, and will make sure we don’t try to use the method with a type that *doesn’t* conform to **Codable**.

Finally we need to update the ContentView to use a type annotation when we try to load the astronauts file and also for the missions file:

**struct** ContentView: View {

**let** astronauts: [String: Astronaut] = Bundle.main.decode("astronauts.json")

**let** missions: [Mission] = Bundle.main.decode("missions.json")

We just need to define the type of dictionary or type we will be using when we decode the files.

## Formatting our mission view

For this project, we will use some computed properties in the Mission struct to make it easier to display the values we want:

struct Mission: Codable, Identifiable {

// Nested struct cause this is only being used by Mission struct

struct CrewRole: Codable {

let name: String

let role: String

}

let id: Int

let launchDate: Date? //Optional property

let crew: [CrewRole]

let description: String

var displayName: String {

"Apollo \(id)"

}

var image: String {

"apollo\(id)"

}

var formattedLaunchDate: String {

launchDate?.formatted(date: .abbreviated, time: .omitted) ?? "N/A"

}

}

The displayName and image properties just make use of simple interpolation to display a string with a format we need. For the date, we first changed to Date type to use the formatted modifier. The formattedLaunchDate basically tries to give an abbreviate format to the launch date if it is not nil, otherwise, it will return “N/A”.

We will also create an extension of ShapeStyle for the color we want to use in the screen:

**import** Foundation

**import** SwiftUI

// Creates an extension of ShapeStyle only when we are extending Color

**extension** ShapeStyle **where** **Self** == Color {

**static** **var** darkBackground: Color {

Color(red: 0.1, green: 0.1, blue: 0.2)

}

**static** **var** lightBackground: Color {

Color(red: 0.2, green: 0.2, blue: 0.3)

}

}

**Color** conforms to a bigger protocol called **ShapeStyle** that is what lets us use colors, gradients, materials, and more as if they were the same thing. This ShapeStyle protocol is what the background() modifier uses, so what we really want to do is extend Color but do it in a way all the SwiftUI modifiers using ShapeStyle work too. That is why we use the where Self == Color, which means “we want to add functionality to ShapeStyle, but only for times when it’s being used as a color”.

You can also add colors to the asset catalog with specific names, which lets you work visually, but the extensions make it easier to monitor changes using something like GitHub.

The ContentView will look like this now:

**struct** ContentView: View {

**let** astronauts: [String: Astronaut] = Bundle.main.decode("astronauts.json")

**let** missions: [Mission] = Bundle.main.decode("missions.json")

**let** columns = [

GridItem(.adaptive(minimum: 150))

]

**var** body: **some** View {

NavigationView {

ScrollView {

LazyVGrid(columns: columns) {

ForEach(missions) { mission **in**

NavigationLink {

Text("Detail view")

} label: {

VStack {

Image(mission.image)

.resizable()

.scaledToFit()

.frame(width: 100, height: 100)

VStack {

Text(mission.displayName)

.font(.headline)

.foregroundColor(.white)

Text(mission.formattedLaunchDate)

.font(.caption)

.foregroundColor(.white.opacity(0.5))

}

.padding(.vertical)

.frame(maxWidth: .infinity)

.background(.lightBackground)

}

.clipShape(RoundedRectangle(cornerRadius: 10))

.overlay(

RoundedRectangle(cornerRadius: 10)

.stroke(.lightBackground)

)

}

}

}

.padding([.horizontal, .bottom])

}

.navigationTitle("Moonshot")

.background(.darkBackground)

.preferredColorScheme(.dark)

}

}

}

Things to note there:

* We are using a LazyVGrid and we define the columns to have adaptive grid items with a minimum 150 points of size.
* For each mission, we have a VStack as a label where we define an Image using the image computed property and defining a specific size of 100 x 100
* Then we define another VStack for the mission display name and formatted launch date. For them we are defining a specific font and foreground color.
* For the previous VStack we define a padding vertical, and use the lightBackground color we defined in the extension.
* For the main VStack we use the darkBackground and the dark scheme color.

## Showing Mission Details with ScrollView and GemetryReader

For the details view, we will display a screen with the information about the mission: its mission badge, mission description and all the astronauts that were on the crew along with their roles.

For the first two things, we have the following code:

**struct** MissionView: View {

**let** mission: Mission

**var** body: **some** View {

GeometryReader { geometry **in**

ScrollView {

VStack {

Image(mission.image)

.resizable()

.scaledToFit()

//Use only 60% of available width

.frame(maxWidth: geometry.size.width \* 0.60)

.padding(.top)

VStack(alignment: .leading) {

Text("Mission Highlights")

.font(.title.bold())

.padding(.bottom, 5)

Text(mission.description)

}

.padding(.horizontal)

}

.padding(.bottom)

}

}

.navigationTitle(mission.displayName)

.navigationBarTitleDisplayMode(.inline)

.background(.darkBackground)

}

}

We have a scrolling VStack with a resizable image for the mission badge, then a Text view for the highlights and another one for the description. We use GeometryReader to set the maximum width of the mission image, in this case, the 60% of the available width.

To be able to see the preview, we replace the #Preview chunk of code with this:

**struct** MissionView\_Previews: PreviewProvider {

**static** **let** missions: [Mission] = Bundle.main.decode("missions.json")

**static** **var** previews: **some** View {

MissionView(mission: missions[0])

.preferredColorScheme(.dark)

}

}

Basically, we load the missions and display the first one.

**Tip:** This view will automatically have a dark color scheme because it’s applied to the **NavigationView** in **ContentView**, but the **MissionView** preview doesn’t know that so we need to enable it by hand.

## Merging Codable structs

Below the mission description, we will show the information of the crew members.

As the astronauts information is in a separate file, we need to make use of the astronaut id to link the information to show.

In the MissionView, we have the next code:

**struct** MissionView: View {

**struct** CrewMember {

**let** role: String

**let** astronaut: Astronaut

}

**let** mission: Mission

**let** crew: [CrewMember]

**var** body: **some** View {

GeometryReader { geometry **in**

ScrollView {

VStack {

Image(mission.image)

.resizable()

.scaledToFit()

//Use only 60% of available width

.frame(maxWidth: geometry.size.width \* 0.60)

.padding(.top)

VStack(alignment: .leading) {

Rectangle()

.frame(height: 2)

.foregroundColor(.lightBackground)

.padding(.vertical)

Text("Mission Highlights")

.font(.title.bold())

.padding(.bottom, 5)

Text(mission.description)

Rectangle()

.frame(height: 2)

.foregroundColor(.lightBackground)

.padding(.vertical)

Text("Crew")

.font(.title.bold())

.padding(.bottom, 5)

}

.padding(.horizontal)

ScrollView(.horizontal, showsIndicators: **false**) {

HStack {

ForEach(crew, id: \.role) { crewMember **in**

NavigationLink {

Text("Astronat details")

} label: {

HStack {

Image(crewMember.astronaut.id)

.resizable()

.frame(width: 104, height: 72)

.clipShape(Capsule())

.overlay(

Capsule()

.strokeBorder(.white, lineWidth: 1, antialiased: **false**)

)

VStack(alignment: .leading) {

Text(crewMember.astronaut.name)

.foregroundColor(.white)

.font(.headline)

Text(crewMember.role)

.foregroundColor(.secondary)

}

}

.padding(.horizontal)

}

}

}

}

}

.padding(.bottom)

}

}

.navigationTitle(mission.displayName)

.navigationBarTitleDisplayMode(.inline)

.background(.darkBackground)

}

// Initialize the mission and crew structures

**init**(mission: Mission, astronauts: [String: Astronaut]){

**self**.mission = mission

**self**.crew = mission.crew.map { member **in**

// Find the astronauts from the crew in the astronauts info by name

**if** **let** astronaut = astronauts[member.name] {

**return** CrewMember(role: member.role, astronaut: astronaut)

} **else** {

fatalError("Missing \(member.name)")

}

}

}

}

**struct** MissionView\_Previews: PreviewProvider {

**static** **let** missions: [Mission] = Bundle.main.decode("missions.json")

**static** **let** astronauts: [String: Astronaut] = Bundle.main.decode("astronauts.json")

**static** **var** previews: **some** View {

MissionView(mission: missions[0], astronauts: astronauts)

.preferredColorScheme(.dark)

}

}

Things to note in this view:

* We created a nested struct for the CrewMember for the role and the data of the astronaut.
* A property called crew : [CrewMember] was created to store an array of CrewMember objects. To pair the data, we can loop over the mission crew, and for each crew member, look in the dictionary to find the one with the matching ID. If there is no match, then a fatal error is displayed. This is made on the init method.
* After that change, the preview struct is also updated to read the astronauts.json and pass that information into the mission view constructor.
* With the information available, we create a couple of “dividers” using the Rectangle view before the Mission Highlights and Crew titles.
* For the crew information, we have another ScrollView along with a HStack.
* We use another HStack to display the image and a VStack for the name and role of the astronaut. For the image, we use a specific size using frame and the clipShape and overlay to give it a specific form and highlight the border.
* The ScrollView is after the VStack because they work best when they take full advantage of the available screen space.

In the ContentView, for the NavigationLink, now we have the following ode to show the MissionView screen:

NavigationLink {

MissionView(mission: mission, astronauts: astronauts)

}

## Finishing up with one last view

To finish the application, we will add a third view to display astronaut details. This is the AstronautView:

**struct** AstronautView: View {

**let** astronaut: Astronaut

**var** body: **some** View {

ScrollView {

VStack {

Image(astronaut.id)

.resizable()

.scaledToFit()

Text(astronaut.description)

.padding()

}

}

.background(.darkBackground)

.navigationTitle(astronaut.name)

.navigationBarTitleDisplayMode(.inline)

}

}

**struct** AstronautView\_Previews: PreviewProvider {

**static** **let** astronauts: [String: Astronaut] = Bundle.main.decode("astronauts.json")

**static** **var** previews: **some** View {

AstronautView(astronaut: astronauts["armstrong"]!)

}

}

This is a simpler screen than the others as it only has a ScrollView with a VStack to show the image of the astronaut, using the available space and the description of the astronaut.

Again, for the preview, we need to decode the information from astronauts.json and in this case, select a specific item and as it is marked as optional, we use the !.

# Project 09 - Drawing

In this technique project we’re going to take a close look at drawing in SwiftUI, including creating custom paths and shapes, animating your changes, solving performance problems, and more – it’s a really big topic, and deserves close attention.

SwiftUI uses the same drawing system that we have on the rest of Apple’s frameworks: Core Animation and Metal. Most of the time Core Animation is responsible for our drawing, whether that’s custom paths and shapes or UI elements such as **TextField**, but when things really get complex we can move down to Metal – Apple’s low-level framework that’s optimized for complex drawing. One of the neat features of SwiftUI is that these two are almost interchangeable: we can move from Core Animation to Metal with one small change.

## Creating custom paths with SwiftUI

SwiftUI gives us a dedicated **Path** type for drawing custom shapes. It’s very low level. Just like colors, gradients, and shapes, paths are views in their own right. This means we can use them just like text views and images.

**struct** CustomPathEx: View {

**var** body: **some** View {

Path { path **in**

path.move(to: CGPoint(x: 200, y: 100))

path.addLine(to: CGPoint(x: 100, y: 300))

path.addLine(to: CGPoint(x: 300, y: 300))

path.addLine(to: CGPoint(x: 200, y: 100))

path.closeSubpath() //Indicates this is the end of the path

}

//.fill(.red) //This is a way to color the figure

//.stroke(.blue, lineWidth: 10)

.stroke(.blue, style: StrokeStyle(lineWidth: 10, lineCap: .round, lineJoin: .round))

}

}

In the example above, we have built a triangle using a Path and defining an initial starting point with the move modifier and adding lines with the addLine modifier. Paths have lots of methods for creating shapes with squares, circles, arcs, and lines.

CG is short for Core Graphics, which provides a selection of basic types that lets us reference X/Y coordinates (**CGPoint**), widths and heights (**CGSize**), and rectangular frames (**CGRect**).

We can fill the triangle with the .fill modifier. We can also use the stroke() modifier to draw around the path rather than filling it in. The closeSubpath() is used to indicate that it is the end of the path.

An alternative is to use SwiftUI’s dedicated **StrokeStyle** struct, which gives us control over how every line should be connected to the line after it (line join) and how every line should be drawn when it ends without a connection after it (line cap). This is particularly useful because one of the options for join and cap is **.round**, which creates gently rounded shapes

## Paths vs shapes in SwiftUI

SwiftUI enables custom drawing with two types: paths and shapes:

* A path is a series of drawing instructions such as “start here, draw a line to here, then add a circle there”, all using absolute coordinates.
* A shape has no idea where it will be used or how big it will be used, but instead will be asked to draw itself inside a given rectangle.

SwiftUI implements Shape as a protocol with a single required method: path to draw.

**struct** Triangle: Shape {

**func** path(in rect: CGRect) -> Path {

**var** path = Path()

path.move(to: CGPoint(x: rect.midX, y: rect.minY))

path.addLine(to: CGPoint(x: rect.minX, y: rect.maxY))

path.addLine(to: CGPoint(x: rect.maxX, y: rect.maxY))

path.addLine(to: CGPoint(x: rect.midX, y: rect.minY))

**return** path

}

}

**struct** Arc: Shape {

**let** startAngle: Angle

**let** endAngle: Angle

**let** clockwise: Bool

**func** path(in rect: CGRect) -> Path {

**let** rotationAdjustment = Angle.degrees(90)

**let** modifiedStart = startAngle - rotationAdjustment

**let** modifiedEnd = endAngle - rotationAdjustment

**var** path = Path()

path.addArc(center: CGPoint(x: rect.midX, y: rect.midY), radius: rect.width / 2, startAngle: modifiedStart, endAngle: modifiedEnd, clockwise: !clockwise)

**return** path

}

}

**struct** ShapesAndPaths: View {

**var** body: **some** View {

Arc(startAngle: .degrees(0), endAngle: .degrees(110), clockwise: **true**)

.stroke(.blue, lineWidth: 10)

.frame(width: 300, height: 300)

Triangle()

//.fill(.red)

.stroke(.red, style: StrokeStyle(lineWidth: 10, lineCap: .round, lineJoin: .round))

.frame(width: 300, height: 300)

}

}

In the previous example, we have 2 Shapes, one for a Triangle and one for an Arc.

By using the CGRect you have some helpful properties like minx (smalles X value in the rectangle), maxX (largest X value in the rectangle), midX (mid-point between minX and maxX)

By calling the Triangle shape, we can use the stroke modifier and give it a specific size with the frame modifier.

In the case of the Arc shape, we define 3 properties (startAngle, endAngle and clockwise to specify the rotation). There are a couple of things to take into consideration:

1. In the eyes of SwiftUI, 0 degrees is not straight upwards, but instead directly to the right.
2. Shapes measure their coordinates from the bottom-left corner rather than the top-left corner, which means SwiftUI goes the other way around from one angle to the other. This is, in my not very humble opinion, extremely alien.

So, to create an Arc starting from the top of the rectangle, we defined the rotationAdjustment and the modifiedStart and modifiedEnd.

## Adding strokeBorder() support with InsettableShape

If you create a shape without a specific size, it will automatically expand to occupy all available space.

If you use stroke modifier without defining the size of the shape, you may see the edges of the border are cut off, you can avoid this by using .strokeBorder modifier.

When you use a custom shape, like the Arc shape defined previously, and try to use strokeBorder, it will not be found, because this modifier is part of the InsettableShape protocol. This protocol allows a shape to be inset (reduced inwards), by a certain amount.

**struct** Arc: **InsettableShape** {

**let** startAngle: Angle

**let** endAngle: Angle

**let** clockwise: Bool

**var** insetAmount = 0.0

**func** path(in rect: CGRect) -> Path {

**let** rotationAdjustment = Angle.degrees(90)

**let** modifiedStart = startAngle - rotationAdjustment

**let** modifiedEnd = endAngle - rotationAdjustment

**var** path = Path()

path.addArc(center: CGPoint(x: rect.midX, y: rect.midY), radius: rect.width / 2 - insetAmount, startAngle: modifiedStart, endAngle: modifiedEnd, clockwise: !clockwise)

**return** path

}

**func** inset(by amount: CGFloat) -> **some** InsettableShape {

**var** arc = **self**

arc.insetAmount += amount

**return** arc

}

}

So, to make the previous Arc shape an insettable shape, we must add a property (insetAmount), for this shape, use it for the radius, add the inset method which will use a CGFloat parameter and conform the InsettableShape.

Then, we can use it like this:

Arc(startAngle: .degrees(-90), endAngle: .degrees(90), clockwise: **true**)

.strokeBorder(.blue, lineWidth: 40)

## Transforming shapes using CGAffineTransform and even-odd fills

SwiftUI offers the CGAffineTransform that describes how a path or view should be rotated, scaled or sheared.

There are also the even-odd fills, that allows us to control how overlapping shapes should be rendered.

In the following example, we are going to create a Flower shape out of several rotated ellipse petals, with each ellipse positioned around a circle. There is one important note about CGAffineTransform: it measures angles in radians rather than degrees but 3.141 radians is equal to 180 degrees, so 3.141 radians multiplied by 2 is equal to 360 degrees.

This is the code:

**struct** Flower: Shape {

**var** petalOffset = -20.0

**var** petalWidth = 100.0

**func** path(in rect: CGRect) -> Path {

**var** path = Path()

**for** number **in** stride(from: 0, to: Double.pi \* 2, by: Double.pi / 8){

**let** rotation = CGAffineTransform(rotationAngle: number)

**let** position = rotation.concatenating(CGAffineTransform(translationX: rect.width / 2, y: rect.height / 2))

**let** originalPetal = Path(ellipseIn: CGRect(x: petalOffset, y: 0, width: petalWidth, height: rect.width / 2))

**let** rotatedPetal = originalPetal.applying(position)

path.addPath(rotatedPetal)

}

**return** path

}

}

**struct** TransformShapes: View {

@State **private** **var** petalOffset = -20.0

@State **private** **var** petalWidth = 100.0

**var** body: **some** View {

VStack {

Flower(petalOffset: petalOffset, petalWidth: petalWidth)

//.stroke(.red, lineWidth: 1)

.fill(.red, style: FillStyle(eoFill: **true**))

Text("Offset")

Slider(value: $petalOffset, in: -40...40)

.padding([.horizontal, .bottom])

Text("Width")

Slider(value: $petalWidth, in: 0...100)

.padding(.horizontal)

}

}

}

This is what it does:

* A new path is created.
* We count from 0 up to pi multiplied by 2 to complete the circle, counting in one eighth of pi each time – this will give us 16 petals.
* We create a rotation transform equal to the current number.
* Add to that rotation a movement equal to half the width and height of our draw space, so each petal is centered in our shape.
* Create a *new* path for a petal, equal to an ellipse of a specific size.
* Apply our transform to that ellipse so it’s moved into position.
* Add that petal’s path to our main path.

1. Rotating something then moving it does not produce the same result as moving then rotating, because when you rotate it first the direction it moves will be different from if it were not rotated.

2. To really help you understand what’s going on, we’ll be making our petal ellipses use a couple of properties we can pass in externally.

3. Ranges such as 1...5 are great if you want to count through numbers one a time, but if you want to count in 2s, or in our case count in “pi/8”s, you should use stride(from:to:by:) instead.

In the fill modifier for the Flower shape in the main struct, we use the FillStyle eoFill in true, this is the even-odd rule, which decides whether part of a path should be colored depending on the overlaps it contains: It works like this:

* If a path has no overlaps it will be filled.
* If another path overlaps it, the overlapping part won’t be filled.
* If a third path overlaps the previous two, then it will be filled.
* ...and so on.

Only the parts that actually overlap are affected by this rule.

## Creative borders and fills using ImagePaint

SwiftUI gives us a dedicated type that wraps images in a way that we have complete control over how the images should be rendered. This allows us to use them for borders and fills without problem.

The type is called **ImagePaint**, and it’s created using one to three parameters. At the very least you need to give it an **Image** to work with as its first parameter, but you can also provide a rectangle within that image to use as the source of your drawing specified in the range of 0 to 1 (the second parameter), and a scale for that image (the third parameter).

If you want to try using the **sourceRect** parameter, make sure you pass in a **CGRect** of relative sizes and positions: 0 means “start” and 1 means “end”. For example, this will show the entire width of our example image, but only the middle half:

Text("Hello World")

.frame(width: 300, height: 300)

.border(ImagePaint(image: Image("mountain"), sourceRect: CGRect(x: 0, y: 0.25, width: 1, height: 0.5), scale: 0.1), width: 30)

It’s worth adding that **ImagePaint** can be used for view backgrounds and also shape strokes. For example, we could create a capsule with our example image tiled as its stroke:

Capsule()

.strokeBorder(ImagePaint(image: Image("mountain"), sourceRect: CGRect(x: 0, y: 0.25, width: 1, height: 0.5), scale: 0.2), lineWidth: 20)

.frame(width: 300, height: 200)

**ImagePaint** will automatically keep tiling its image until it has filled its area – it can work with backgrounds, strokes, borders, and fills of any size.

## Enabling high-performance Metal rendering with drawingGroup()

SwiftUI uses Core Animation for its rendering by default, which offers great performance out of the box. However, for complex rendering you might find your code starts to slow down. The next example will demonstrate t by creating a color-cycling view that renders concentric circles in a range of colors.

We can get a color cycling effect by using the **Color(hue:saturation:brightness:)** initializer: hue is a value from 0 to 1 controlling the kind of color we see – red is both 0 and 1, with all other hues in between. To figure out the hue for a particular circle we can take our circle number (e.g. 25), divide that by how many circles there are (e.g. 100), then add our color cycle amount (e.g. 0.5). So, if we were circle 25 of 100 with a cycle amount of 0.5, our hue would be 0.75.

One small complexity here is that hues don’t automatically wrap after we reach 1.0, which means a hue of 1.0 is equal to a hue of 0.0, but a hue of 1.2 is *not* equal to a hue of 0.2. As a result, we’re going to wrap the hue by hand: if it’s over 1.0 we’ll subtract 1.0, to make sure it always lies in the range of 0.0 to 1.0.

**struct** ColorCyclingCircle: View {

**var** amount = 0.0

**var** steps = 100

**var** body: **some** View {

ZStack {

ForEach(0..<steps) { value **in**

Circle()

.inset(by: Double(value))

.strokeBorder(

LinearGradient(gradient: Gradient(colors: [

color(for: value, brightness: 1),

color(for: value, brightness: 0.5)

]), startPoint: .top, endPoint: .bottom),

lineWidth: 2

)

}

}

}

**func** color(for value: Int, brightness: Double) -> Color {

**var** targetHue = Double(value) / Double(steps) + amount

**if** targetHue > 1 {

targetHue -= 1

}

**return** Color(hue: targetHue, saturation: 1, brightness: brightness)

}

}

**struct** DrawingGroupEx: View {

@State **private** **var** colorCycle = 0.0

**var** body: **some** View {

VStack {

ColorCyclingCircle(amount: colorCycle)

.frame(width: 300, height: 300)

Slider(value: $colorCycle)

}

}

}

That now renders a gentle gradient, showing bright colors at the top of the circle down to darker colors at the bottom. And *now* when you run the app you’ll find it runs much slower –

SwiftUI is struggling to render 100 gradients as part of 100 separate views.

We can fix this by using the modifier drawingGroup() in the VStack:

**var** body: **some** View {

ZStack {

ForEach(0..<steps) { value **in**

Circle()

.inset(by: Double(value))

.strokeBorder(

LinearGradient(gradient: Gradient(colors: [

color(for: value, brightness: 1),

color(for: value, brightness: 0.5)

]), startPoint: .top, endPoint: .bottom),

lineWidth: 2

)

}

}

**.drawingGroup()**

}

This tells SwiftUI it should render the contents of the view into an off-screen image before putting it back onto the screen as a single rendered output, which is significantly faster. This is powered by Metal, an Apple’s framework for working directly with the GPU for extremely fast graphics.

**Important:** The **drawingGroup()** modifier is helpful to know about and to keep in your arsenal as a way to solve performance problems when you hit them, but you should *not* use it that often. Adding the off-screen render pass might slow down SwiftUI for simple drawing, so you should wait until you have an actual performance problem before trying to bring in **drawingGroup()**.

## Special effects in SwiftUI: blurs, blending, and more

You can apply blend modes to control the way one view is rendered on top of another. For example:

**struct** SpecialEffectsEx: View {

**var** body: **some** View {

ZStack {

Image("mountain")

Rectangle()

.fill(.green)

.**blendMode**(.multiply)

}

}

}

In the previous example, we use the blendMode(.multiply). It multiplies each source pixel color with the destination pixel color – in our case, each pixel of the image and each pixel of the rectangle on top. Each pixel has color values for RGBA, ranging from 0 (none of that color) through to 1 (all of that color), so the highest resulting color will be 1x1, and the lowest will be 0x0.

This is so common, that it has a shortcut:

Image("mountain")

.colorMultiply.(blue)

Another popular effect is called *screen*, which does the opposite of multiply: it inverts the colors, performs a multiply, then inverts them again, resulting in a brighter image rather than a darker image.

VStack {

ZStack {

Circle()

.fill(Color(red: 1, green: 0, blue: 0))

.frame(width: 200 \* amount)

.offset(x: -50, y: -80)

.blendMode(.screen)

Circle()

.fill(Color(red: 0, green: 1, blue: 0))

.frame(width: 200 \* amount)

.offset(x: 50, y: -80)

.blendMode(.screen)

Circle()

.fill(Color(red: 0, green: 0, blue: 1))

.frame(width: 200 \* amount)

.blendMode(.screen)

}

.frame(width: 300, height: 300)

Slider(value: $amount)

.padding()

}

.frame(maxWidth: .infinity, maxHeight: .infinity)

.background(.black)

.ignoresSafeArea()

SwiftUI use adaptive by default, which are designed to look good in both dark mode and light mode, so they are custom blend, but if you want the full effect, you need to use the Color view and specify the values for red, green and blue:

Color(red: 1, green: 0, blue: 0))

There is another effect called saturation, which adjusts how much color is used inside a view:

@State **private** **var** amount = 0.0

**var** body: **some** View {

VStack {

Image("mountain")

.resizable()

.frame(width: 200, height: 200)

.**saturation**(amount)

.**blur**(radius: (1 - amount) \* 20)

Slider(value: $amount)

.padding()

}

}

With that code, having the slider at 0 means the image is blurred and colorless, but as you move the slider to the right it gains color and becomes sharp – all rendered at lightning-fast speed.

## Animating simple shapes with animatableData

struct Trapezoid: Shape {

var insetAmount: Double

var animatableData: Double {

get { insetAmount }

set { insetAmount = newValue }

}

func path(in rect: CGRect) -> Path {

var path = Path()

path.move(to: CGPoint(x: 0, y: rect.maxY))

path.addLine(to: CGPoint(x: insetAmount, y: rect.minY))

path.addLine(to: CGPoint(x: rect.maxX - insetAmount, y: rect.minY))

path.addLine(to: CGPoint(x: rect.maxX, y: rect.maxY))

path.addLine(to: CGPoint(x: 0, y: rect.maxY))

return path

}

}

struct AnimatableData: View {

@State private var insetAmount = 50.0

var body: some View {

Trapezoid(insetAmount: insetAmount)

.frame(width: 200, height: 100)

.onTapGesture {

withAnimation{

insetAmount = Double.random(in: 10...90)

}

}

}

}

In the example above, we define a Trapezoid shape, which uses a property called insetAmount to define the longitude of the top line of the trapezoid.

In the body of the View, where we define the shape, we are modifying that property every time the users taps on it with the onTapGesture modifier. Every time you tap the trapezoid, insetAmount gets set to a new value, causing the shape to be redrawn.

The animation is not working without the animatableData property because as soon as **insetAmount** is set to a new random value, it will immediately jump to that value and pass it directly into **Trapezoid** – it won’t pass in lots of intermediate values as the animation happens. This is why our trapezoid jumps from inset to inset; it has no idea an animation is even happening.

The property anitmatableData fixes this. when we use **withAnimation()**, SwiftUI immediately changes our state property to its new value, but behind the scenes it’s also keeping track of the changing value over time as part of the animation. As the animation progresses, SwiftUI will set the **animatableData** property of our shape to the latest value, and it’s down to us to decide what that means – in our case we assign it directly to **insetAmount**, because that’s the thing we want to animate.

## Animating complex shapes with AnimatablePair

AnimatableData property allows us to animate changes to shapes, but this a property, which means it must always be one value, however we can use a special wrapper called AnimatablePair to work with a pair of values.

Having this code, where we build a Checkerboard of 4x4 initially, we want to make it of 8 rows by 16 columns when we tap on it, with an animation of 3 seconds long:

struct Checkerboard: Shape {

var rows: Int

var columns: Int

func path(in rect: CGRect) -> Path {

var path = Path()

// figure out how big each row/column needs to be

let rowSize = rect.height / Double(rows)

let columnSize = rect.width / Double(columns)

// loop over all rows and columns, making alternating squares colored

for row in 0..<rows {

for column in 0..<columns {

// Decide if the square is colored by identifying if it is multiple of 2

if (row + column).isMultiple(of: 2) {

// this square should be colored; add a rectangle here

let startX = columnSize \* Double(column)

let startY = rowSize \* Double(row)

// Draw the square

let rect = CGRect(x: startX, y: startY, width:

columnSize, height: rowSize)

path.addRect(rect)

}

} }

return path }

}

struct AnimatablePairEx: View {

@State private var rows = 4

@State private var columns = 4

var body: some View {

Checkerboard(rows: rows, columns: columns)

.onTapGesture {

withAnimation(.linear(duration: 3)){

rows = 8

columns = 16

}

}

}

}

However, when we tap on the checkerboard, the change is made at once, without any animation. There are two problems here:

1. We have two properties that we want to animate, not one.

2. Our row and column properties are integers, and SwiftUI can’t interpolate integers.

For the first problem, we use AnimatablePair. We can read individual values from the pair using .first and .second.

To resolve the *second* problem we’re just going to do some type conversion: we can convert a **Double** to an **Int** just by using **Int(someDouble)**, and go the other way by using **Double(someInt)**.

So, we just need to add a property to the Checkerboard shape for solve these two problems:

struct Checkerboard: Shape {

var rows: Int

var columns: Int

**var animatableData: AnimatablePair<Double, Double> {**

**get {**

**AnimatablePair(Double(rows), Double(columns))**

**}**

**set {**

**rows = Int(newValue.first)**

**columns = Int(newValue.second)**

**}**

**}**

# Project 10 – Cupcake Corner

In this project we’re going to build a multi-screen app for ordering cupcakes. This will use a couple of forms, which are old news for you, but you’re also going to learn how to make classes conform to Codable when they have @Published properties, how to send and receive the order data from the internet, how to validate forms, and more.

## Adding Codable conformance for @Published properties

If all the properties of a type already conform to **Codable**, then the type itself can conform to **Codable** with no extra work – Swift will synthesize the code required to archive and unarchive your type as needed. However, this *doesn’t* work when we use property wrappers such as **@Published**, which means conforming to **Codable** requires some extra work on our behalf.

To fix this, we need to implement **Codable** conformance ourself.

**class** User: ObservableObject, Codable {

@Published **var** name = "Esteban"

}

In the previous code, the property wrapper comes from the fact that our name property is automatically wrapped inside another type that adds some additional functionality. In the case of @Published, that’s a struct called Published that can store any kind of value.

You cannot make an instance of Published by itself, but instead make an instance of Published<String> for example. It is the same that happens with Set, arrays and dictionaries.

Swift already has rules in place that say if an array contains **Codable** types then the whole array is **Codable**, and the same for dictionaries and sets. However, SwiftUI *doesn’t* provide the same functionality for its **Published** struct

We need to tell Swift which properties should be loaded and saved, and how to do both of those actions.

This is done using an enum that conforms to a special protocol called **CodingKey**, which means that every case in our enum is the name of a property we want to load and save. The convention is to call the enum CodingKeys.

Then, we need to create a custom initializer that will be given some sort of container and use that to read values for all our properties. First, this initializer is handed an instance of a new type called Decoder. This contains all our data, but it’s down to us to figure out how to read it.

Second, anyone who subclasses our **User** class must override this initializer with a custom implementation to make sure they add their own values. We mark this using the **required** keyword.

Third, inside the method we ask our **Decoder** instance for a container matching all the coding keys we already set in our **CodingKey** struct by writing **decoder.container(keyedBy: CodingKeys.self)**. This means “this data should have a container where the keys match whatever cases we have in our **CodingKeys** enum.

Finally, we can read values directly from that container by referencing cases in our enum – **container.decode(String.self, forKey: .name)**. This provides really strong safety in two ways: we’re making it clear we expect to read a string and we’re also using a case in our CodingKeys enum rather than a string.

There is one more task to do before User class conforms to Codable: we need to tell Swift how to encode this type, how to archive it ready to write to JSON. In this function we need an encoder instance to write to, ask it to make a container using our CodingKeys enum for keys, then write our values attached to each key.

**class** User: ObservableObject, Codable {

**enum** CodingKeys: CodingKey {

**case** name

}

@Published **var** name = "Esteban"

**required** **init**(from decoder: Decoder) **throws** {

**let** container = **try** decoder.container(keyedBy: CodingKeys.**self**)

name = **try** container.decode(String.**self**, forKey: .name)

}

**func** encode(to encoder: Encoder) **throws** {

**var** container = encoder.container(keyedBy: CodingKeys.**self**)

**try** container.encode(name, forKey: .name)

}

}

## Sending and receiving Codable data with URLSession and SwiftUI

With the built-in tools for sending and receiving data from internet combined with the Codable support, we can convert Swift objects to JSON and vice versa and assign the data immediately to the UI.

**struct** Response: Codable {

**var** results: [Result]

}

**struct** Result: Codable {

**var** trackId: Int

**var** trackName: String

**var** collectionName: String

}

**struct** \_2\_UrlSessionExample: View {

@State **private** **var** results = [Result]()

**var** body: **some** View {

List(results, id: \.trackId) { item **in**

VStack(alignment: .leading) {

Text(item.trackName)

.font(.headline)

Text(item.collectionName)

}

}

.task {

**await** loadData()

}

}

**func** loadData() **async** {

// 1. Define the url

**guard** **let** url = URL(string: "https://itunes.app.com/search=mudvayne&entity=song")

**else** {

// If it fails to get the information, print message and exit. It should not fail

print("Invalid URL")

**return**

}

// 2. Fetch data from url

**do** {

**let** (data, \_) = **try** **await** URLSession.shared.data(from: url)

**if** **let** decodedResponse = **try**? JSONDecoder().decode(Response.**self**, from: data){

results = decodedResponse.results

}

} **catch** {

print("Invalid data")

}

}

}

The Result struct represents part of the information returned by Apple’s API, the Response struct will store an array of results.

The content view (\_2\_UrlSessionExample) shows a list with the track name and the collection name. To load this list, we need to use a **task**.

An Asynchronous function allows you to leave some code running while the main app code carries on working. This is useful when you have networking work to do, like downloading data from internet. While we are downloading the data, the rest of the app doesn’t freeze up.

The loadData function uses the **async** keyword to denote that this function might want to go to sleep in order to complete its work.

We want that to be run as soon as our **List** is shown, but we can’t just use **onAppear()** here because that doesn’t know how to handle sleeping functions – it expects its function to be synchronous. To do this, we use task(). The task is used to call functions that might go to sleep for a while. To call an async function inside task, we need to use **await** before the task.

Tip: Think of **await** as being like try – we’re saying we understand a sleep might happen, in the same way try says we acknowledge an error might be thrown.

Inside loadData() we have three steps we need to complete:

1. Creating the URL we want to read. For this, we use guard let… to guarantee that the URL is well constructed, if it is not, an error will be printed, and the process will end.
2. Fetching the data for that URL. Here is where the sleep is likely to happen. Every time a sleep is possible, we need to use await and as an error might happen too, you need try too.

There are some important things in the part:

* 1. The data(from:) method, takes a URL and returns the Data object at that URL. This method belongs to the **URLSession** class, which you can create and configure by hand if you want, but you can also use a shared instance that comes with sensible defaults.
  2. The return value from data(from:) is a tuple containing the data at the URL and some metadata describing how the request went. We don’t use the metadata, but we do want the URL’s data, hence the underscore – we create a new local constant for the data, and toss the metadata away.
  3. When using both try and await at the same time, we must write try await, in that order.

1. Decoding the result of that data into a Response struct. We use the JSONDecoder for this.

## Loading an image from a remote server

When you need to load an image from Internet, instead of using an Image view, you need to use a AsyncImage. This view use an image URL instead of a asset name.

The simplest image we can create looks like this:

AsyncImage(url: URL(string: "https://hws.dev/img/logo.png"))

As the image has to be downloaded, SwiftUI knows nowthing about it, so, it is not able to size it properly ahead of time. We can use the scale property to tell it to scale the image, for example, to 3x:

AsyncImage(url: URL(string: "https://hws.dev/img/logo.png"), scale: 3)

If you want a specific size, you need a more advance form of the AsyncImage that passes the final image view once it’s ready, which we can then customize as needed.

AsyncImage(url: URL(string: "https://hws.dev/img/logo.png"))

{ image **in**

image

.resizable()

.scaledToFit()

} placeholder: {

Color.red

}

.frame(width: 200, height: 200)

The red color for the place holder is just to allow us to see the placeholder before it loads the image. A resizable image and Color.red both automatically take up all available space, which means the frame() modifier actually works now.

Another way of creating a AsyncImage that gives you more control is the following, where we can show an error if something doesn’t allow us to load the image

struct \_3\_AsyncImageExample: View {

var body: some View {

AsyncImage(url: URL(string: "https//hws.dev/img/logo.png")) { phase **in**

**if** **let** image = phase.image {

image

.resizable()

.scaledToFit()

} **else** **if** phase.error != **nil** {

Text("There was an error loading the image.")

} **else** {

ProgressView()

}

}

.frame(width: 200, height: 200)

}

}

## Validating and disabling forms

The modifier disabled() allow us to use a condition to disable a view. You can use simple properties or methods for this condition. In the following example, we define a Form for a username and an email, and a Button that will be available only if both texts have more than 5 characters:

**struct** \_4\_ValidateForms: View {

@State **private** **var** username = ""

@State **private** **var** email = ""

**var** body: **some** View {

Form {

Section {

TextField("Username", text: $username)

TextField("Email", text: $email)

}

Section {

Button("Create account") {

print("Creating account...")

}

}

//.disabled(username.isEmpty || email.isEmpty)

.disabled(disableForm)

}

}

**var** disableForm: Bool {

username.count < 5 || email.count < 5

}

}

In the previous example, we are using a method but we chan use a validation in the same disabled modifier.

## Taking basic order details - Project

The first thing we will do is to create an ordering screen. First, we need to define a data model using @State for simple values and @StateObject for reference types. We will have a single class that stores all the data, which will be passed from screen to screen. This means all screens in the app share the same data.

The first class to be created is the Order class:

**class** Order: ObservableObject {

**static** **let** types = ["Vanilla", "Strawberry", "Chocolate", "Rainbow"]

@Published **var** type = 0

@Published **var** quantity = 3

@Published **var** specialRequestEnabled = **false** {

**didSet** {

**if** specialRequestEnabled == **false** {

extraFrosting = **false**

addSprinkles = **false**

}

}

}

@Published **var** extraFrosting = **false**

@Published **var** addSprinkles = **false**

}

This class conforms to ObservableObject and it has several properties like the types of cakes (an static array with the possible options), a type which will use an index of the types, the quantity, whether the user wants to make special requests (in this case, when the value is set to false, we will do the same for the other two properties (extraFrosting and add Sprinkles)), whether the user wants extra frosting and whether the user wants to add sprinkles. Each of these properties need to update the UI when changed, which means we need to mark them with @Published.

Now, in the ContentView we create a single instance of Order.

**struct** ContentView: View {

// This will be the only place in the application where the object will be created

@StateObject **var** order = Order()

**var** body: **some** View {

NavigationView {

Form {

Section {

Picker("Select your cake type", selection: $order.type) {

ForEach(Order.types.indices) {

Text(Order.types[$0])

}

}

Stepper("Number of cakes: \(order.quantity)", value: $order.quantity, in: 3...20)

}

Section {

Toggle("Any special requests?", isOn: $order.specialRequestEnabled.animation())

**if** order.specialRequestEnabled {

Toggle("Add extra frosting", isOn: $order.extraFrosting)

Toggle("Add extra sprinkles", isOn: $order.addSprinkles)

}

}

Section {

NavigationLink {

AddressView(order: order)

} label: {

Text("Deliver details")

}

}

}

.navigationTitle("Cupcake Corner")

}

}

}

The ContentView has 3 sections:

* The first one allows the user to select the type of cake and the number of cakes. We use the type property and the indices from types to display and select the type of cake.
* The second has a toggle for the special requests and if this one is on, it will show the extra frosting and sprinkles toggles.
* The last section hast a navigatin link for the AddressView.

There’s a small speed bump here: our cupcake topping list is an array of strings, but we’re storing the user’s selection as an integer – how can we match the two? One easy solution is to use the indices property of the array, which gives us a position of each item that we can then use with as an array index. This is a bad idea for mutable arrays because the order of your array can change at any time, but here our array order won’t ever change so it’s safe.

The AddresView will have an order observed object property like this:

**struct** AddressView: View {

@ObservedObject **var** order: Order

**var** body: **some** View {

Text("Hello, World”)

}

}

The AddressView will be updated later.

## Checking for a valid address - Project

The second step in the project is to let the user enter a valid address.

The first thing to do is add a CheckoutView, which will be the next step once the address is ready. We will come to this view with another navigation link. This will have so far, just the same Order observed object:

**struct** CheckoutView: View {

@ObservedObject **var** order: Order

**var** body: **some** View {

Text("Hello, World!")

}

}

For the Address, we need to add 4 new properties to Order class:

// Address properties

@Published **var** name = ""

@Published **var** streetAddress = ""

@Published **var** city = ""

@Published **var** zip = ""

Now, the AddressView will have the following structure:

**struct** AddressView: View {

@ObservedObject **var** order: Order

**var** body: **some** View {

Form {

Section {

TextField("Name", text: $order.name)

TextField("Street Address", text: $order.streetAddress)

TextField("City", text: $order.city)

TextField("Zip", text: $order.zip)

}

Section {

NavigationLink {

CheckoutView(order: order)

} label: {

Text("Check out")

}

}

}

.navigationTitle("Delivery details")

.navigationBarTitleDisplayMode(.inline)

}

}

Now there are 3 views sharing the same data. The data is not lost even when you go to the third screen and go back to the first one. this is the natural side effect of using a class for our data, but it’s an instant feature in our app without having to do any work.

Now, we will add some validations.

In the Order class, add this property at the end:

// Validation property for Address

**var** hasValidAddress: Bool {

**if** name.isEmpty || streetAddress.isEmpty || city.isEmpty ||

zip.isEmpty {

**return** **false**

}

**return** **true**

}

We can now use that condition in conjunction with SwiftUI’s disabled() modifier. For the project, we can use it after the second section in the AddressView:

Section {

NavigationLink {

CheckoutView(order: order)

} label: {

Text("Check out")

}

}

.**disabled**(order.hasValidAddress == false)

Now if you run the app you’ll see that all four address fields must contain at least one character in order to continue. Even better, SwiftUI automatically grays out the button when the condition isn’t true, giving the user really clear feedback when it is and isn’t interactive.

## Preparing for checkout - Project

The first part is to build the CheckoutView.

First, in the Order class we will add a computed property called cost that will use the following logic for the price:

* There’s a base cost of $2 per cupcake.
* We’ll add a little to the cost for more complicated cakes.
* Extra frosting will cost $1 per cake.
* Adding sprinkles will be another 50 cents per cake.

This is represented in the following code in the Order class:

**var** cost: Double {

// $2 per cacke

**var** cost = Double(quantity) \* 2

// complicated cakes cost more

cost += (Double(type) / 2)

// $1/cake for extra frosting

**if** extraFrosting {

cost += Double(quantity)

}

// $0.50/cake for sprinkles

**if** addSprinkles {

cost += Double(quantity) / 2

}

**return** cost

}

For the CheckoutView we use a ScrollView with a Vstack, and inside this, we add a AsyncImage with a scale of 3x and a specific height of 233.

We also add a text for the total and a button to place the order.

**struct** CheckoutView: View {

@ObservedObject **var** order: Order

**var** body: **some** View {

ScrollView {

VStack {

AsyncImage(url: URL(string: "https://hws.dev/img/cupcakes@3x.jpg"),

scale: 3) { image **in**

image

.resizable()

.scaledToFit()

} placeholder: {

ProgressView()

}

.frame(height: 233)

Text("Your total is \(order.cost, format: .currency(code: "USD"))")

.font(.title)

Button("Place Order", action: {})

.padding()

}

}

.navigationTitle("Check out")

.navigationBarTitleDisplayMode(.inline)

}

}

## Encoding an ObservableObject class - Project

In this project we have used one Order object that is shared between all the screens, however there is a problem, we have used the @Published property wrapper and with this, we have lost support for automatic Codable conformance.

When you try to conforme Codable in the Order class, it will fail, so now you need to do manually. Swift doesn’t understand how to encode and decode published properties and we need Codable protocol to encode and decode JSON.

First, we add an enum that conforms CodingKeys, listing all the properties we want to save.

Then, we write an encode(to:) method that creates a container using the coding keys and writes out all the properties attached to their respective keys.

Because that method is marked with **throws**, we don’t need to worry about catching any of the errors that are thrown inside – we can just use **try** without adding **catch**, knowing that any problems will automatically propagate upwards and be handled elsewhere.

Our final step is to implement a required initializer to decode an instance of **Order** from some archived data. This is pretty much the reverse of encoding, and even benefits from the same **throws** functionality

The problem *now* is that we just created a custom initializer for our **Order** class, **init(from:)**, and Swift wants us to use it everywhere , so we can create another initializar, this time empty:

**class** Order: ObservableObject, Codable {

// Add a CodingKeys enum to declare the properties we want to encode and decode

**enum** CodingKeys: CodingKey {

**case** type, quantity, extraFrosting, addSprinkles, name, streetAddress, city, zip

}

**static** **let** types = ["Vanilla", "Strawberry", "Chocolate", "Rainbow"]

@Published **var** type = 0

@Published **var** quantity = 3

@Published **var** specialRequestEnabled = **false** {

**didSet** {

**if** specialRequestEnabled == **false** {

extraFrosting = **false**

addSprinkles = **false**

}

}

}

@Published **var** extraFrosting = **false**

@Published **var** addSprinkles = **false**

// Address properties

@Published **var** name = ""

@Published **var** streetAddress = ""

@Published **var** city = ""

@Published **var** zip = ""

// Validation property for Address

**var** hasValidAddress: Bool {

**if** name.isEmpty || streetAddress.isEmpty || city.isEmpty ||

zip.isEmpty {

**return** **false**

}

**return** **true**

}

**var** cost: Double {

// $2 per cacke

**var** cost = Double(quantity) \* 2

// complicated cakes cost more

cost += (Double(type) / 2)

// $1/cake for extra frosting

**if** extraFrosting {

cost += Double(quantity)

}

// $0.50/cake for sprinkles

**if** addSprinkles {

cost += Double(quantity) / 2

}

**return** cost

}

**init**() { }

**func** encode(to encoder: Encoder) **throws** {

**var** container = encoder.container(keyedBy: CodingKeys.**self**)

**try** container.encode(type, forKey: .type)

**try** container.encode(quantity, forKey: .quantity)

**try** container.encode(extraFrosting, forKey: .extraFrosting)

**try** container.encode(addSprinkles, forKey: .addSprinkles)

**try** container.encode(name, forKey: .name)

**try** container.encode(streetAddress, forKey: .streetAddress)

**try** container.encode(city, forKey: .city)

**try** container.encode(zip, forKey: .zip)

}

**required** **init**(from decoder: Decoder) **throws** {

**let** container = **try** decoder.container(keyedBy: CodingKeys.**self**)

type = **try** container.decode(Int.**self**, forKey: .type)

quantity = **try** container.decode(Int.**self**, forKey: .quantity)

extraFrosting = **try** container.decode(Bool.**self**, forKey: .extraFrosting)

addSprinkles = **try** container.decode(Bool.**self**, forKey: .addSprinkles)

name = **try** container.decode(String.**self**, forKey: .name)

streetAddress = **try** container.decode(String.**self**, forKey: .streetAddress)

city = **try** container.decode(String.**self**, forKey: .city)

zip = **try** container.decode(String.**self**, forKey: .zip)

}

}

## Sending and receiving orders over the internet - Project

iOS comes with fantastic functionality for handling networking: the URLSession to send and receive data class combined with Codable to convert objects to and from JSON, and the URLRequest struct t configure how data should be sent.

If we have an async function like this one:

**func** placeOrder() **async** {

}

And we try to call it from a buttion like this:

Button("Place Order", action: {

placeOrder()

})

.padding()

It won’t work, because the button expects to be able to run its action immediately, and doesn’t understand how to wait for something, even with the await placeOrder() code.

We can create a new task and just like the task() modifier, this will run any kind of asyncrhonous code. We just need to add the await cal inside the Task like this:

Button("Place Order", action: {

Task {

**await** placeOrder()

}

})

.padding()

Inside **placeOrder()** we need to do three things:

1. Convert our current **order** object into some JSON data that can be sent.

2. Tell Swift how to send that data over a network call.

3. Run that request and process the response.

For the URLRequest, we need to attach the data in a very specific way, which means we need to provide 2 extra pieces of data:

1. The HTTP method of a request determines how data should be sent. There are several HTTP methods, but in practice only GET (“I want to read data”) and POST (“I want to write data”) are used much. We want to write data here, so we’ll be using POST.
2. The content type of a request determines what kind of data is being sent, which affects the way the server treats our data. This is specified in what’s called a MIME type, which was originally made for sending attachments in emails, and it has several thousand highly specific options.

The complete code of CheckoutView is this:

**struct** CheckoutView: View {

@ObservedObject **var** order: Order

@State **private** **var** confirmationMessage = ""

@State **private** **var** showingConfirmation = **false**

**var** body: **some** View {

ScrollView {

VStack {

AsyncImage(url: URL(string: "https://hws.dev/img/cupcakes@3x.jpg"),

scale: 3) { image **in**

image

.resizable()

.scaledToFit()

} placeholder: {

ProgressView()

}

.frame(height: 233)

Text("Your total is \(order.cost, format: .currency(code: "USD"))")

.font(.title)

Button("Place Order", action: {

Task {

**await** placeOrder()

}

})

.padding()

}

}

.navigationTitle("Check out")

.alert("Thank you!", isPresented: $showingConfirmation){

Button("OK") { }

} message: {

Text(confirmationMessage)

}

.navigationBarTitleDisplayMode(.inline)

}

**func** placeOrder() **async** {

// 1. Convert the current order object into JSON data

**guard** **let** encoded = **try**? JSONEncoder().encode(order) **else** {

print("Failed to encode order")

**return**

}

// 2. Tell Swift how to send that data over a network call

// Define the url to send the data to

// Force unwrap for the URL

**let** url = URL(string: "https://reqres.in/api/cupcakes")!

// Define the request

**var** request = URLRequest(url: url)

request.setValue("application/json", forHTTPHeaderField:

"Content-Type")

request.httpMethod = "POST"

**do** {

// Do the network call

**let** (data, \_) = **try** **await** URLSession.shared.upload(for: request, from: encoded)

// Decode the data from the result

**let** decodedOrder = **try** JSONDecoder().decode(Order.**self**, from:

data)

// Set up the confirmation message

confirmationMessage = "Your order for \(decodedOrder.quantity) x \(Order.types[decodedOrder.type].lowercased()) cupcakes is on its way!"

showingConfirmation = **true**

} **catch** {

print("Checkout failed.")

}

}

}

# Project 11 - Bookworm

## Creating a custom component with @Binding

You’ve already seen how SwiftUI’s **@State** property wrapper lets us work with local value types, and how **@StateObject** lets us work with shareable reference types. Well, there’s a third option, called **@Binding**, which lets us connect an **@State** property of one view to some underlying model data.

@**Binding** it lets us store a mutable value in a view that actually points to some other value from elsewhere. This is extremely important whenever you want to create a custom user interface component.

Example:

struct PushButton: View {

let title: String

**@Binding var isOn: Bool**

var onColors = [Color.red, Color.yellow]

var offColors = [Color(white: 0.6), Color(white: 0.4)]

var body: some View {

Button(title){

**isOn**.toggle()

}

.padding()

.background(

LinearGradient(

gradient: Gradient(colors: **isOn** ? onColors : offColors),

startPoint: .top,

endPoint: .bottom)

)

.foregroundColor(.white)

.clipShape(Capsule())

.shadow(radius: **isOn** ? 0 : 5)

}

}

struct ContentView: View {

@State private var rememberMe = false

var body: some View {

VStack {

***PushButton(title: "Remember Me", isOn: $rememberMe)***

Text(rememberMe ? "On" : "Off")

}

.padding()

}

}

ContentView has the rememberMe boolean, used to create the PushButton – the button has an initial value provided by ContentView. Once the button was created it takes over control of the value: it toggles the isOn property between true or false internally to the button, but to pass that change back on to the ContentView you need the property to use Binding.

Binding allows us to create a two-way connection between PushButton and whatever is using it, so when one value changes the other does too.

To use Binding, you need a property with @Binding and to add the $ to the value you use for this property (isOn in the example)

## Accepting multi-line text input with TextEditor

For long pieces ot text you should switch the TextField to use a TextEditor. It also expects to be given a two-way binding to a text string, but it has the additional benefit of allowing multiple lines of text.

Mostly because it has nothing special in the way of configuration options, using **TextEditor** is actually easier than using **TextField** – you can’t adjust its style or add placeholder text, you just bind it to a string. However, you *do* need to be careful to make sure it doesn’t go outside the safe area, otherwise typing will be tricky; embed it in a **NavigationView**, a **Form**, or similar.

struct TextEditor\_Example: View {

@AppStorage("notes") private var notes = ""

var body: some View {

NavigationView {

TextEditor(text: $notes)

.navigationTitle("Notes")

.padding()

}

}

}

Tip: @AppStorage is not designed to store secure information, so never use it for anything private.

## How to combine Core Data and SwiftUI

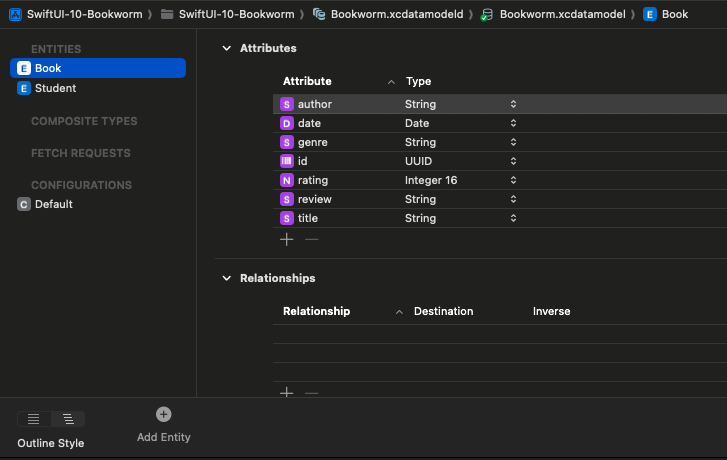
Core Data is an object graph and persistence framework, which is a fancy way of saying it lets us define objects and properties of those objects, then lets us read and write them from permanent storage.

Core Data is capable of sorting and filtering of our data, and can work with much larger data – there’s effectively no limit to how much data it can store. Even better, Core Data implements all sorts of more advanced functionality for when you really need to lean on it: data validation, lazy loading of data, undo and redo, and much more.

To setup Core Data by hand, it takes 3 steps:

* + 1. Define the data we want to use in the app. This is done in a Data Model, a file with extension “xcdatamodeld”.

To create one: press Cmd+N to make a new file, select Data Model from the list of templates, then name your model Bookworm.xcdatamodeld. When it is created, you open it and add an Entity and rename it. Next, click on the + button below the attributes and add the attributes.



* + 1. Set up Core Data. Write a little Swift code to load that model and prepare it for us to use.

Create a Swift file called DataController.swift that conforms ObservableObject and enter this:

**import** CoreData

**class** DataController: ObservableObject {

// Core Data type responsible for loading a data model

**let** container = NSPersistentContainer(name: "Bookworm")

**init**() {

container.loadPersistentStores { description, error **in**

**if** **let** error = error {

print("Core Data failed to load: \(error.localizedDescription)")

}

}

}

}

Data models don’t contain our actual data, just the definitions of properties and attributes.

To actually load the data model we need to call **loadPersistentStores()** on our container, which tells Core Data to access our saved data according to the data model in Bookworm.xcdatamodeld.

The initializer loads our stored data immediately and if something goes wrong, it will print a message to Xcode debug log.

* + 1. Create an instance of DataController and send it into SwiftUI’s environment.

Most apps work with only one Core Data store at a time, so rather than every view trying to create their own store individually we instead create it once when our app starts, then store it inside the SwiftUI environment so everywhere else in our app can use it.

In the BookworkApp.swift add this:

**import** SwiftUI

**@main**

**struct** SwiftUI\_10\_BookwormApp: App {

// Creates the data controller

@StateObject **private** **var** dataController = DataController()

**var** body: **some** Scene {

WindowGroup {

ContentView()

.environment(\.managedObjectContext,

dataController.container.viewContext)

}

}

}

The managed object context are the “live” version of your data, when you load objects and change them, those changes only exist in memory until you specifically save them back to persistent store.

The job of the view context is to let us work with all our data in memory, which is much faster than constantly reading and writing data to disk.

Now we need to read and write data.

Retrieving information from Core Data is done using a *fetch request* – we describe what we want, how it should sorted, and whether any filters should be used, and Core Data sends back all the matching data. We need to make sure that this fetch request stays up to date.

For that we use a property wrapper called @FetchRequest and it takes at least one parameter describing how we want the result to be sorted.

In the ContentView add this:

@FetchRequest(sortDescriptors: [

SortDescriptor(\.title),

SortDescriptor(\.author, order: .reverse)

]) **var** books: FetchedResults<Book>

When we defined the entity Core Data created a class for us that inherits from one of its own classes: **NSManagedObject**. This one cannot be seen in our code. These objects are called *managed* because Core Data is looking after them.

All our managed objects live inside a *managed object context*, one of which we created earlier. Placing it into the SwiftUI environment meant that it was automatically used for the **@FetchRequest** property wrapper

when it comes to adding and saving objects, we need access to the managed object context that it is in SwiftUI’s environment. add this property to **ContentView** now

@Environment(\.managedObjectContext) **var** moc

To ask our managed object context to save itself, means it will write its changes to the persistent store. This is a throwing function call, because in theory it might fail. In practice, nothing about what we’ve done has any chance of failing, so we can call this using **try?** – we don’t care about catching errors.

try? moc.save()

## Creating books with Core Data

# Project 12 – Core Data

This technique project is going to explore Core Data in more detail, starting with a summary of some basic techniques then building up to tackling some more complex problems.

## Why does .self work for ForEach?

Previously we looked at the various ways **ForEach** can be used to create dynamic views, but they all had one thing in common: SwiftUI needs to know how to identify each dynamic view uniquely so that it can animate changes correctly.

There are several ways, like the object conforming the Identifiable protocol and use the id property, or use a keypath for a property that we know is unique and also using the \.selft.

Example:

List {

ForEach([2, 4, 6, 8, 10], id: \.self) {

Text("\($0) is even")

}

}

When we use **\.self** as an identifier, we mean “the whole object”, but in practice that doesn’t mean much. So what actually happens is that Swift computes the *hash value* of the struct, which is a way of representing complex data in fixed-size values, then uses that hash as an identifier.

Hash values can be generated in any number of ways, but the concept is identical for all hash- generating functions:

1. Regardless of the input size, the output should be the same fixed size.

2. Calculating the same hash for an object twice in a row should return the same value.

Hashes are commonly used for things like data verification.

Xcode generates a class for our managed objects, it makes that class conform to **Hashable**, which is a protocol that means Swift can generate hash values for it, which in turn means we can use **\.self** for the identifier. This is also why **String** and **Int** work with **\.self**: they also conform to **Hashable**.

**Hashable** is a bit like **Codable**: if we want to make a custom type conform to **Hashable**, then as long as everything it contains also conforms to **Hashable** then we don’t need to do any work.

**struct** Student: Hashable {

**let** name: String

}

**struct** \_1\_HashableExample: View {

**let** students = [Student(name: "Harry Potter"), Student(name: "Louse Lane")]

**var** body: **some** View {

List(students, id: \.**self**) { student **in**

Text(student.name)

}

}

}

We can make **Student** conform to **Hashable** because all its properties already conform to **Hashable**, so Swift will calculate the hash values of each property then combine them into one hash that represents the whole struct. Of course, if we end up with two students that have the same name we’ll hit problems.

When using Core Data, the objects it creates for us actually have a selection of other properties beyond those we defined in our data model, including one called the object ID. These IDs are similar to **UUID**, although Core Data generates them sequentially as we create objects.

So \.self works for Core Data’s objects because they already conform to Hashable. So, if you want to use a specific identifier that’s awesome, but you don’t need to because **\.self** is also an option.

**Warning:** Although calculating the same hash for an object twice in a row should return the same value, calculating it between two runs of your app – i.e., calculating the hash, quitting the app, relaunching, then calculating the hash again – can return different values.

## Creating NSManagedObject subclasses

When we create a new Core Data entity, Xcode automatically generates a managed object class for us when we build our code. We can then use that in a SwiftUI **@FetchRequest** to show data in our user interface, but it is a quite painful because there are lots of optionals to unwrap.

There are 2 solutions. First, we can create an entity to work with: open your data model and create an entity with some attributes. In the View mene, choose Inspectors> Show Data Model Inspector and if you choose the entity, you can see a property called Codegen. This controls how Xcode generates the entity as a managed object class when we build our project. For that property, select Manual/None. With this value, Xcode will no longer generate the entity class for you to use in code.

Then you need to go to the Editor menu and choose Create NSManagedObject Subclass, make sure “CoreDataProject” is selected then press Next, then make sure your entity is selected and press Next again. Xcode will convert its generated code into actual Swift files that we can see and change, but keep in mind if you change the files Xcode generated and you re-generate those files, your changes will be lost.

Xcode will have generated two files for us, but we only care about one of them: Movie+CoreDataProperties.swift. You will something like this example:

extension Movie {

@nonobjc public class func fetchRequest() -> NSFetchRequest<Movie> {

return NSFetchRequest<Movie>(entityName: "Movie")

}

@NSManaged public var title: String?

@NSManaged public var director: String?

@NSManaged public var year: Int16

}

extension Movie : Identifiable {

}

Here is where the optional is set for the properties which use @NSManaged for all of them. **@NSManaged** is *not* a property wrapper. When we read or write the value of a property that is **@NSManaged**, Core Data catches that and handles it internally. However changing this class, like removing the optional (?) from the properties might not work because Core Data is lazy.

Instead of removing the optional, you might want to consider add computed properties to access optional values safely, while also letting us store your nil coalescing code all in one place:

public var wrappedTitle: String {

title ?? "Unknown Title"

}

## Conditional saving of NSManagedObjectContext

We’ve been using the **save()** method of **NSManagedObjectContext** to flush out all unsaved changes to permanent storage, but there is a way to check wheter any changes actually need to be saved.

Every managed object is given a **hasChanges** property, that is true when the object has unsaved changes. And, the entire context also contains a **hasChanges** property that checks whether any object owned by the context has changes.

So, rather than call **save()** directly you should always wrap it in a check first, like this:

if moc.hasChanges {

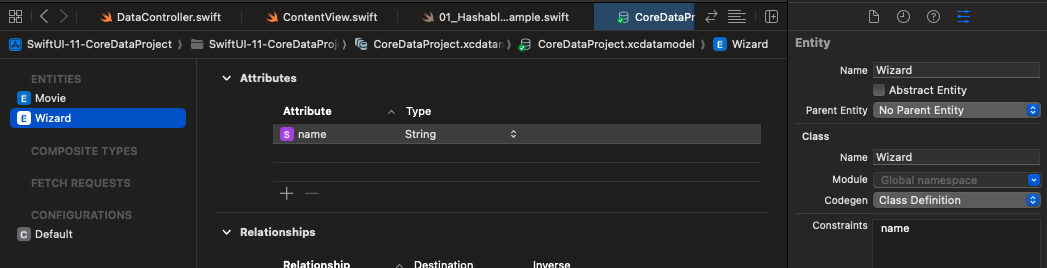
try? moc.save()

}

## Ensuring Core Data objects are unique using constraints

Core Data gives us *constraints*: we can make one attribute constrained so that it must always be unique. We can then go ahead and make as many objects as we want, unique or otherwise, but as soon as we ask Core Data to save those objects it will resolve duplicates so that only one piece of data gets written. Even better, if there was some data already written that clashes with our constraint, we can choose how it should handle merging the data.

For example, for a specif entity, you can go to View > Inspectors > Data Model, in the right panel, you will see a Constraints section where you can add a new element. If you add a new one and change it to use the name of a specific entity’s property, like in this example:



If you have a code like the following, when you can add a new Wizard element to a list with the same name (which is the field used as a constraint, or in other words, it must be unique), you can add all the elements you want, but as soon as you hit the Save button, you will receive an error, Core Data has detected the collision and is refusing to save the changes:

**struct** \_\_ConstraintExample: View {

@Environment(\.managedObjectContext) **var** moc

@FetchRequest(sortDescriptors: []) **var** wizards: FetchedResults<Wizard>

**var** body: **some** View {

VStack{

List(wizards, id: \.**self**) { wizard **in**

Text(wizard.name ?? "Unknown")

}

Button("Add") {

**let** wizard = Wizard(context: moc)

wizard.name = "Harry Potter"

}

Button("Save") {

**do** {

**try** moc.save()

} **catch** {

print(error.localizedDescription)

}

}

}

}

}

If you *want* Core Data to write the changes, you need to open DataController.swift and adjust the loadPersistentStores() completion handler to specify how data should be merged in this situation:

container.loadPersistentStores { description, error in

if let error = error {

print("Core Data failed to load: \(error.localizedDescription)")

return

}

self.container.viewContext.mergePolicy = NSMergePolicy.mergeByPropertyObjectTrump

}

That asks Core Data to merge duplicate objects based on their properties.

## Filtering @FetchRequest using NSPredicate

When we use SwiftUI’s **@FetchRequest** property wrapper, we can provide an array of sort descriptors to control the ordering of results, but we can also provide an **NSPredicate** to control *which* results should be shown.

In the following example, we have an Entity called Ship with two string attibutes (name and universe) and we have the following code:

**struct** \_3\_NSPredicateExample: View {

@Environment(\.managedObjectContext) **var** moc

@FetchRequest(sortDescriptors: [], predicate:

NSPredicate(format: "universe == %@", "Star Wars")

) **var** ships: FetchedResults<Ship>

**var** body: **some** View {

VStack {

List(ships, id: \.**self**) { ship **in**

Text(ship.name ?? "Unknown name")

}

}

Button("Add Examples"){

**let** ship1 = Ship(context: moc)

ship1.name = "Enterprise"

ship1.universe = "Star Trek"

**let** ship2 = Ship(context: moc)

ship2.name = "Defiant"

ship2.universe = "Star Trek"

**let** ship3 = Ship(context: moc)

ship3.name = "Millennium Falcon"

ship3.universe = "Star Wars"

**let** ship4 = Ship(context: moc)

ship4.name = "Executor"

ship4.universe = "Star Wars"

**try**? moc.save()

}

}

}

The predicate we have it is basically asking to show only the ships which universe is Star Wars, the %@ is used to indicate a value will be provided, in this case, ‘Star Wars’.

As well as **==**, we can also use comparisons such as **<** and **>** to filter our objects. For example

this will return Defiant, Enterprise, and Executor:

NSPredicate(format: "name < %@", "F"))

We could use an **IN** predicate to check whether the universe is one of three options from an array, like this:

NSPredicate(format: "universe IN %@", ["Aliens", "Firefly","Star Trek"])

We can also use predicates to examine part of a string, using operators such as **BEGINSWITH** and **CONTAINS**. For example,

NSPredicate(format: "name BEGINSWITH %@", "E"))

You can flip predicates around using **NOT**, to get the inverse of their regular behavior

NSPredicate(format: "NOT name BEGINSWITH[c] %@", "e"))

## Dynamically filtering @FetchRequest with SwiftUI

Fetch requests are created as a property, so if you try to make them reference another property Swift will refuse. To solve this, we should carve off the functionality we want into a separate view, then inject values into it.

As an example, we have the following SwitfUI view

**struct** \_4\_DynamicFiltering: View {

@Environment(\.managedObjectContext) **var** moc

@State **private** **var** lastNameFilter = "T"

**var** body: **some** View {

VStack {

// list of matching singers

\_4\_FilteredList(filter: lastNameFilter)

}

Button("Add Examples"){

**let** bjorn = Singer(context: moc)

bjorn.firstName = "Bjorn"

bjorn.lastName = "Speed"

**let** wayne = Singer(context: moc)

wayne.firstName = "Wayne"

wayne.lastName = "Static"

**let** corey = Singer(context: moc)

corey.firstName = "Corey"

corey.lastName = "Taylor"

**try**? moc.save()

}

Button("Show T" ){

lastNameFilter = "T"

}

Button("Show S" ){

lastNameFilter = "S"

}

}

}

}

The previous UI class has a property called lastNameFilter with a default value, that can be changed dynamically and it is used to called another SwiftUI that will receive that value to load a list of elements, in this case, a list of Singer elements, but applying the filter in the initialization of the UI.

**struct** \_4\_FilteredList: View {

@FetchRequest **var** fetchRequest: FetchedResults<Singer>

**var** body: **some** View {

List(fetchRequest, id: \.**self**){ singer **in**

Text("\(singer.wrappedFirstName) \(singer.wrappedLastName)")

}

}

**init**(filter: String) {

// Initialize the view with a fetch request with a predicate that uses the filter

// by using the \_ we are requesting new results to load

\_fetchRequest = FetchRequest<Singer>(sortDescriptors: [], predicate: NSPredicate(format: "lastName BEGINSWITH %@", filter))

}

}

The FilterList view is the one using the FetchRequest. That will run a fetch request using the current managed object context. Because this view will be used inside **ContentView**, we don’t even need to inject a managed object context into the environment.

The reason for the underscore in the \_fetchRequest is beause you are not writing to the fetched object inside the fetch request, but instead writing a wholly new fetch request.

You might look at our code and think that every time the view is recreated – which is every time any state changes in our container view – we’re also recreating the fetch request, which in turn means reading from the database when nothing else has changed. But Core Data won’t do that, it will only re-run the database query when the filter string changes, even if the view is recreated.

For a more flexible FilteredList, we can work with generics:

1. Rather than specifically referencing the **Singer** class, we’re going to use generics with a constraint that whatever is passed in must be an **NSManagedObject**.

2. We need to accept a second parameter to decide which key name we want to filter on, because we might be using an entity that doesn’t have a **lastName** attribute.

3. Because we don’t know ahead of time what each entity will contain, we’re going to let our containing view decide. So, rather than just using a text view of a singer’s name, we’re instead going to ask for a closure that can be run to configure the view however they wan

The FilteredList will look like this:

**import** CoreData

**import** SwiftUI

**struct** \_4\_FilteredList<T: NSManagedObject, Content: View>: View {

@FetchRequest **var** fetchRequest: FetchedResults<T>

// Accept whatever T represents and return some Content

**let** content: (T) -> Content

**var** body: **some** View {

List(fetchRequest, id: \.**self**){ item **in**

**self**.content(item)

}

}

**init**(filterKey: String, filterValue: String, @ViewBuilder content: **@escaping** (T) -> Content) {

// Initialize the view with a fetch request with a predicate that uses the filter

// by using the \_ we are requesting new results to load

\_fetchRequest = FetchRequest<T>(sortDescriptors: [], predicate: NSPredicate(format: "&K BEGINSWITH %@", filterKey, filterValue))

**self**.content = content

}

}

Some important things here:

* The View now receives a generic type (T) and a Swift UI view as Content.
* The content is the closure, this will be called once for each item in the list.
* For the predicate, when we use %@, Core Data automatically inserts quote marks for us, so for the filterKey it doesn’t work, instead we need to use %K for “key”.

For the closure that decides the content of each list row we need to use:

**• @ViewBuilder** lets our containing view (whatever is using the list) send in multiple views if they want.

**• @escaping** says the closure will be stored away and used later, which means Swift needs to take care of its memory.

Now, in the ContentView that calls the FilteredList we can have:

\_4\_FilteredList(filterKey: "lastName", filterValue: lastNameFilter){ (singer: Singer) **in**

Text("\(singer.wrappedFirstName) \(singer.wrappedLastName)")

}

The (singer: Singer) is required so that Swift understands how FilteredList is being used, to make Swfit understand what exactly type of managed object we are using.

## One-to-many relationships with Core Data, SwiftUI, and @FetchRequest

Core Data allows us to link entities together using relationships, and when we use **@FetchRequest** Core Data sends all that data back to us for use. However we need to make a custom NSManagedObject subclass.

In the example, we will use 2 Core Data entities: one to track candy bars and one to track countries where those bars come from.

Relationships come in four forms:

• A one to one relationship means that one object in an entity links to exactly one object in another entity.

• A one to many relationship means that one object in an entity links to many objects in another entity. In our example, this would mean that one type of candy could have been introduced simultaneously in many countries, but that each country still could only make one type of candy.

• A many to one relationship means that many objects in an entity link to one object in another entity. In our example, this would mean that each type of candy has one country of origin, and that each country can make many types of candy.

• A many to many relationship means that many objects in an entity link to many objects in another entity. In our example, this would mean that one type of candy had been introduced simultaneously in many countries, and each country can make many types of candy.

In the example, we will work with one to many relationship. We add a Candy entity with a name attribute and a Country entity with fullName and shortName attributes.

To add a relationship between entityes, select one, start with Country and press + under Relationships table. Call the relationship “candy”, change its destination to Candy, then over in the data model inspector change Type to To Many.

Now select Candy, and add another relationship there. Call the relationship “origin”, change its destination to “Country”, then set its inverse to “candy” so Core Data understands the link goes both ways.

Press Cmd+S to force Xcode to save the changes and generate the code for us. Select both Candy and Country and set their Codegen to Manual/None, then go to the Editor menu and choose Create NSManagedObject Subclass to create code for both our entities – remember to save them in the CoreDataProject group and folder.

In the Country+CoreDataProperties we have a candy property that is a NSSet (Objective-C data type for Set). To fix this we need to modify the files Xcode generated for us, adding convenience wrappers that make SwiftUI work well.

For the candy wrapper, we need first to convert it from NSSet to a Set<Candy> and then convert that Set<Candy> into an array, so that ForEach can read individual values from there and then sort the values.

That class would look like this:

extension Country {

    @nonobjc public class func fetchRequest() -> NSFetchRequest<Country> {

        return NSFetchRequest<Country>(entityName: "Country")

    }

    @NSManaged public var fullName: String?

    @NSManaged public var shortName: String?

    @NSManaged public var candy: NSSet?

    public var wrappedShortName: String {

        shortName ?? "Unknown Country"

    }

    public var wrappedFullName: String {

        fullName ?? "Unknown Country"

    }

    public var candyArry: [Candy] {

        // Try to convert a NSSet to a Set of Candy

        let set = candy as? Set<Candy> ?? []

        return set.sorted {

            $0.wrappedName < $1.wrappedName

        }

    }

}

As we have a custom type, we cannot just use sorted(), instead we have to provide a closure that accepts two candy bars and returns true if the frist candy should be sorted before the second.

In the corresponding view, add the following code:

struct \_5\_RelationshipsExample: View {

@Environment(\.managedObjectContext) var moc

@FetchRequest(sortDescriptors: []) var countries: FetchedResults<Country>

var body: some View {

VStack {

List {

ForEach(countries, id: \.self) { country in

Section(country.wrappedFullName) {

ForEach(country.candyArry, id: \.self) { candy in

Text(candy.wrappedName)

}

}

}

}

Button("Add Examples") {

let candy1 = Candy(context: moc)

candy1.name = "Mars"

candy1.origin = Country(context: moc)

candy1.origin?.shortName = "UK"

candy1.origin?.fullName = "United Kingdom"

let candy2 = Candy(context: moc)

candy2.name = "KitKat"

candy2.origin = Country(context: moc)

candy2.origin?.shortName = "UK"

candy2.origin?.fullName = "United Kingdom"

let candy3 = Candy(context: moc)

candy3.name = "Toblerone"

candy3.origin = Country(context: moc)

candy3.origin?.shortName = "CH"

candy3.origin?.fullName = "Switzerland"

let candy4 = Candy(context: moc)

candy4.name = "Tapita Gallito"

candy4.origin = Country(context: moc)

candy4.origin?.shortName = "CRC"

candy4.origin?.fullName = "Costa Rica"

try? moc.save()

}

}

}

}

Notice how we don’t need to specify anything about the relationships in our fetch request – Core Data understands the entities are linked, so it will just fetch them all as needed.

# Project 13 - Instafilter

The next app will allow the user to import photos from their library, and modify them, using Apple's Core Image framework integrating with UIKit.

## How Propery Wrappers Become Structs

You’ve seen how SwiftUI lets us store changing data in our structs by using the @State property wrapper, how we can bind that state to the value of a UI control using $, and how changes to that state automatically cause SwiftUI to reinvoke the body property of our struct

Property wrappers have that name because they wrap our property inside another struct. What this means is that when we use @State to wrap a string, the actual type of property we end up with is a State<String>. Similarly, when we use @Environment and others we end up with a struct of type Environment that contains some other value inside it.

We know properties cannot be modified in our views because they are structs and therefore, they are fixed.

Note: Cmd+Shift+O lets you to open the Open Quickly to look for ay file or type in the project or frameworks you have imported.

In the following example:

struct \_1\_PropWrapperExe: View {

@State private var blurAmount = 0.0 {

didSet {

print("New value is \(blurAmount)")

}

}

var body: some View {

VStack{

Text("Hello world!")

.blur(radius: blurAmount)

Slider(value: $blurAmount, in: 0...20)

Button("Random Blur") {

blurAmount = Double.random(in: 0...20)

}

}

}

}

You will see that when the value of the blurAmount is changed, we don't see the text printed as expected, except when you press the button. This happens because @State actually wraps its contents, what it’s actually saying is that when the State

struct that wraps blurAmount changes, print out the new blur amount.

You’ve just seen how State wraps its value using a non-mutating setter, which means neither blurAmount or the State struct wrapping it are changing – our binding is directly changing the internally stored value, which means the property observer is never being triggered.

So, changing the property directly using a button works fine, because it goes through the nonmutating setter and triggers the didSet observer, but using a binding doesn’t because it bypasses the setter and adjusts the value directly.

## Responding to state changes using onChange()

Because of the way SwiftUI sends binding updates to property wrappers, assigning property observers used with property wrappers often won’t work.

To fix this we need to use the onChange() modifier, which tells SwiftUI to run a function of our choosing when a particular value changes. SwiftUI will automatically pass in the new value to whatever function you attach, or you can just read the original property if you prefer:

struct \_1\_PropWrapperExe: View {

@State private var blurAmount = 0.0

var body: some View {

VStack{

Text("Hello world!")

.blur(radius: blurAmount)

Slider(value: $blurAmount, in: 0...20)

.onChange(of: blurAmount) { newValue in

print("New value is \(newValue)")

}

Button("Random Blur") {

blurAmount = Double.random(in: 0...20)

}

}

}

}

you can do whatever you want inside the onChange() function: you can call methods, run an algorithm to figure out how to apply the change, or whatever else you might need.

## Showing multiple options with confirmationDialog()

SwiftUI gives us alert() for presenting important announcements with one or two buttons, and sheet() for presenting whole views on top of the current view, but it also gives us confirmationDialog(): an alternative to alert() that lets us add many buttons.

The following is an example of the confirmationDialog:

struct \_2\_MultipleOptsExe: View {

@State private var showingConfirmation = false

@State private var backgroundColor = Color.white

var body: some View {

Text("Hello, World!")

.frame(width: 300, height: 300)

.background(backgroundColor)

.onTapGesture {

showingConfirmation = true

}

.confirmationDialog("Change background", isPresented: $showingConfirmation){

Button("Red") { backgroundColor = .red}

Button("Green") { backgroundColor = .green }

Button("Blue") { backgroundColor = .blue }

Button("Cancel", role: .cancel){ }

} message: {

Text("Select a new color")

}

}

}

As with the alert message, this is a modifier for the text and it accepts two parameters: a binding that decides whether the dialog is currently presented or not and a closure that provides the buttons that shoulde be shown.

## Integrating Core Image with SwiftUI