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