First Edition

A SwiftUI Kickstart

DANIEL H STEINBERG

Introducing the SwiftUI User Interface Framework

Editors Cut

A SwiftUI Kickstart

UIKit Prequel

by Daniel H Steinberg

Editors Cut

Copyright

"A SwiftUI Kickstart UIKit Prequel", by Daniel H Steinberg

Copyright © 2019 Dim Sum Thinking, Inc. All rights reserved.

ISBN-13: 9 978-1-944994-00-6

This is the prequel to A SwiftUI Kickstart originally released December 29, 2019.

Legal

A SwiftUI Kickstart UIKit Prequel

A UIKit Motivator for SwiftUI

A Simple App
Visual Coding
But…
Code: All in One
Code: Smaller Pieces

A SWIFTUI PREQUEL

UIKit Motivator For SwiftUI

Sections:

This is the former first chapter of A SwiftUI Kickstart.

In the first version of the book I started with this chapter that worked an example in UIKit and then in the next chapter rebuilt that example with SwiftUI to highlight the differences. That was helpful as a transitional chapter but more people are coming to SwiftUI without first working in UIKit so I've removed this chapter but offer it as a stand-alone.

I will not be maintaining or updating this chapter for changes in Xcode but the concepts are still valid and may help your move from UIKit to SwiftUI.

This code was tested with Xcode 11 on Catalina.

So where do we start?

We begin our hero's journey, as all such stories begin: our hero (that would be you), is going about their business in their ordinary world unaware that everything is about to change.

This may seem silly that we begin with an example from UIKit before going on to SwiftUI, but to truly understand what is different about this new world, we need to take one last look at our current one.

In this chapter we start by looking at the way we did things before SwiftUI. I can't keep you from skipping ahead to the next chapter where we start looking at SwiftUI but I encourage you to relax and enjoy this journey. After all, you're the hero.

A Simple App

Suppose we have the simplest app that actually does something. It will have a label and a button. When we tap the button we need to send a message to the label so it changes what it displays.

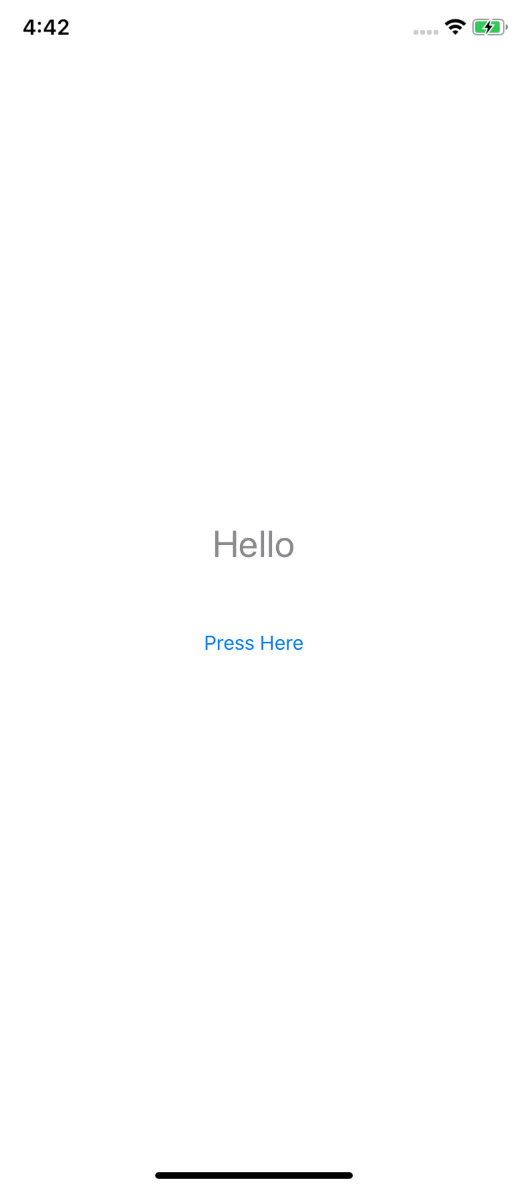
If you are using UIKit instead of SwiftUI, this is the job of the view controller. The view controller knows how to talk to the label using something we call an outlet. The view controller also has a function that gets called when the button is tapped. This function is referred to as an action.

Previewing the app

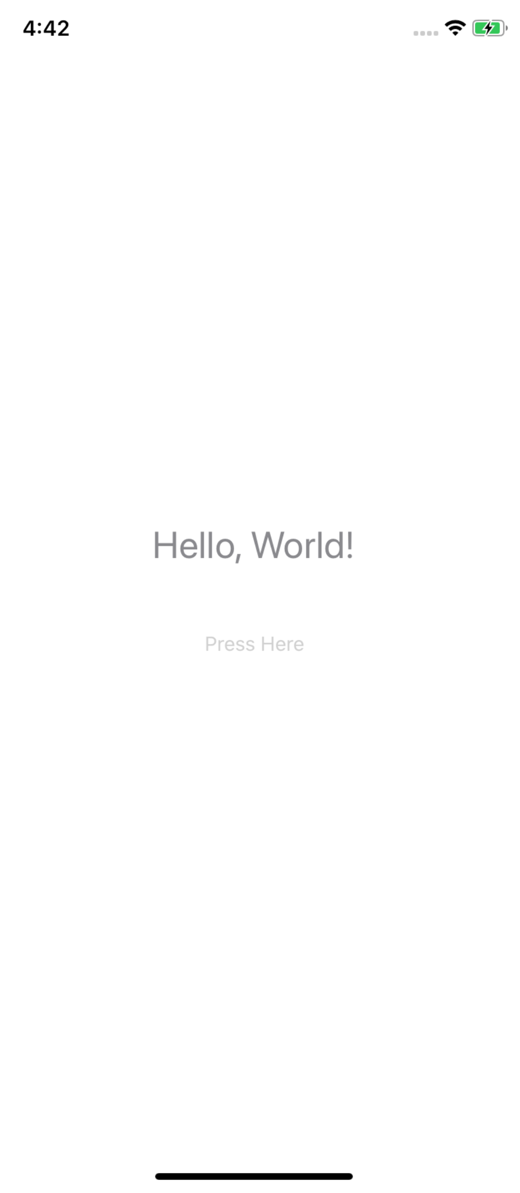
When the app launches we see a label that says "Hello" and a button with the title "Press Here". When we tap the button the label changes to say "Hello, World!" and the button is disabled.

That's it.

Here is what it looks like at launch. Note that throughout this book the screen shots are of light mode as opposed to dark mode.



And this is after the button is tapped.

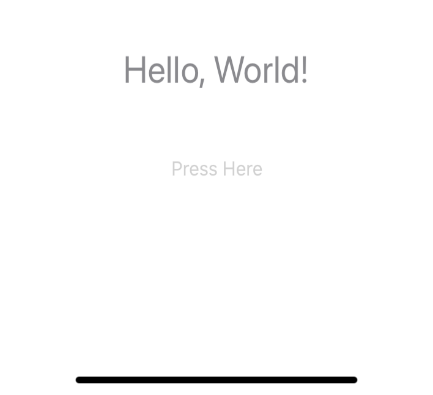


Note the screenshots don't include the phone bezels.

The components

We have a label and a button arranged vertically in the center of the screen. In fact, we want to ensure they remain in the center if our device is in landscape.

We'll arrange our label and button in a stack view and use autolayout.



Less code

In the WWDC sessions introducing SwiftUI in 2019, Apple engineers kept stressing that you'll write less code.

First, I don't always think "less code" is a good argument. Second, I would accomplish everything I've mentioned so far using a storyboard. The only thing I would code would be the action that is triggered when the user taps the button.

In other words, I often write more code in SwiftUI than I did in UIKit and I think more code is ok. The code that I write feels very different.

The plan

Before we get there, in the remainder of this chapter we look at this simple app three ways.

- Using a storyboard
- Doing everything in the ViewController in code
- Splitting the functionality into many files.

Wait. Why are we taking time to look at three different approaches to doing things the old way? Shouldn't we get started doing things the new way?

Here's the thing.

When we start working with SwiftUI, everything is jumbled together. I think it will help if you first consider which parts are for creating and configuring various widgets, which parts are for organizing and presenting them on the screen, and which parts are for working with data and getting things done.

We'll start by working visually and create much of our app in a storyboard.

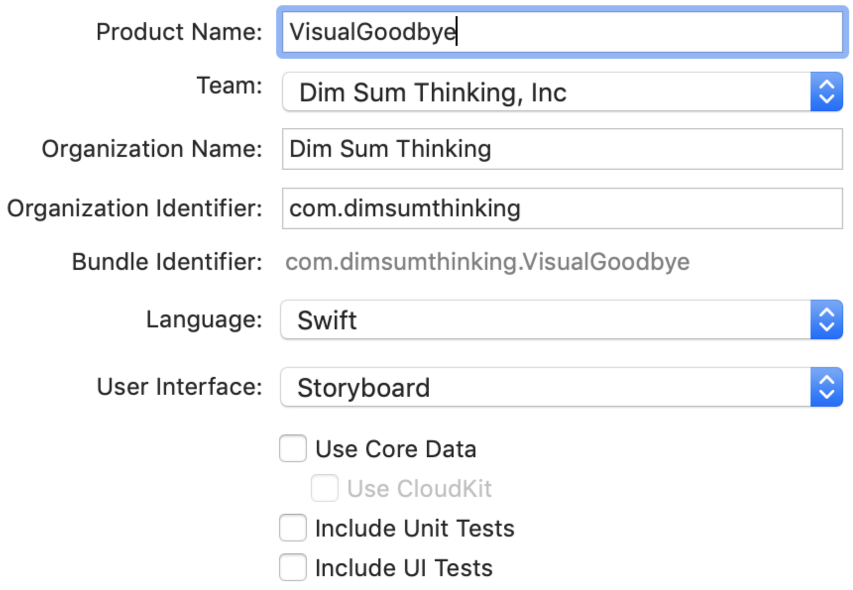
Using Storyboards

In our first version of the app, we'll create our visual elements in the storyboard.

You know that I don't use lines of code as a metric, but if you care about such things, we will create half a dozen lines of code in this version.

Create the project

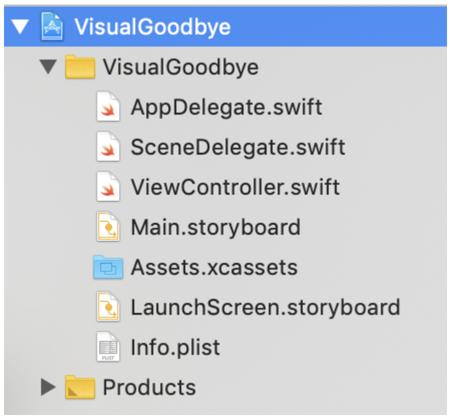
Create a new iOS Single View app. Name it *VisualGoodbye*, choose Swift as the language, and make sure you choose Storyboard as your User Interface.



Product Name:	VisualGoodbye
Team:	Dim Sum Thinking, Inc
Organization Name:	Dim Sum Thinking
Organization Identifier:	com.dimsumthinking
Bundle Identifier:	com.dimsumthinking.VisualGoodbye
Language:	Swift
User Interface:	Storyboard

☐ Use Core Data
☐ Use CloudKit
☐ Include Unit Tests
☐ Include UI Tests

Here are the files generated for this project. The only two that we'll work with are *ViewController.swift* and *Main.storyboard*.

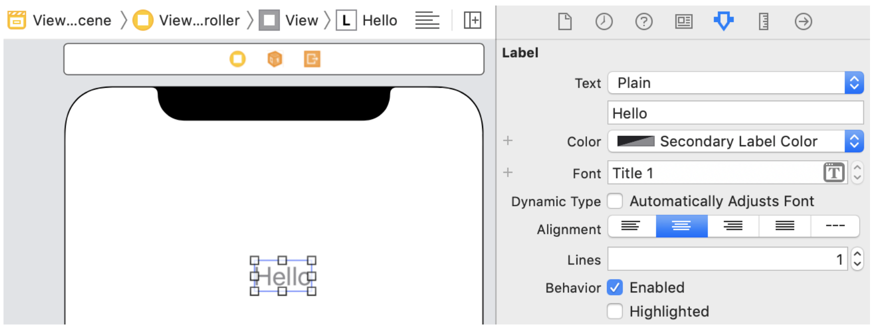


```
▼ VisualGoodbye
  ▼ VisualGoodbye
      AppDelegate.swift
      SceneDelegate.swift
      ViewController.swift
      Main.storyboard
      Assets.xcassets
      LaunchScreen.storyboard
      Info.plist
  ▶ Products
```

Add the visual components

Select the *Main.storyboard* in the project navigator. Click the + button on the right side of Xcode's toolbar or use the keyboard shortcut Command - Shift - L to bring up the Object Library. Drag a `Label` and drop it on the `View`.

Use the Attributes Inspector to set the `Label`'s text to `Hello`, the color to `Secondary Label Color`, and the font to `Title 1`.



View...cene 〉 View...roller 〉 View 〉 L Hello

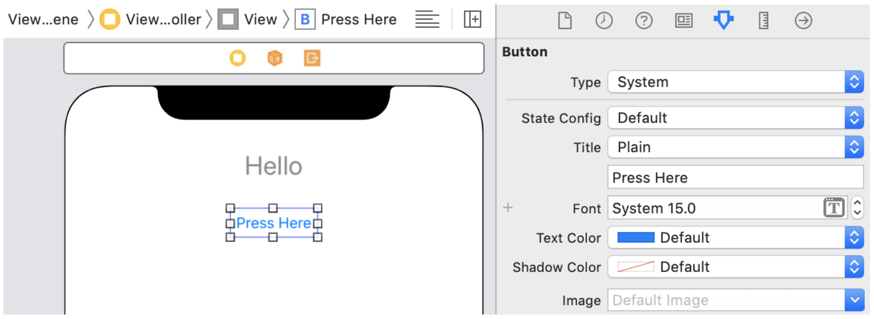
Hello

Label

Text	Plain
	Hello
Color	Secondary Label Color
Font	Title 1
Dynamic Type	☐ Automatically Adjusts Font
Alignment	
Lines	1
Behavior	✓ Enabled
	☐ Highlighted

Now, drag a `Button` from the Object Library and drop it underneath the `Label`.

Set the title to `Press Here`.



View...ene 〉 View...oller 〉 View 〉 B Press Here

Hello

Press Here

Button

Type	System
State Config	Default
Title	Plain
	Press Here
Font	System 15.0
Text Color	Default
Shadow Color	Default
Image	Default Image

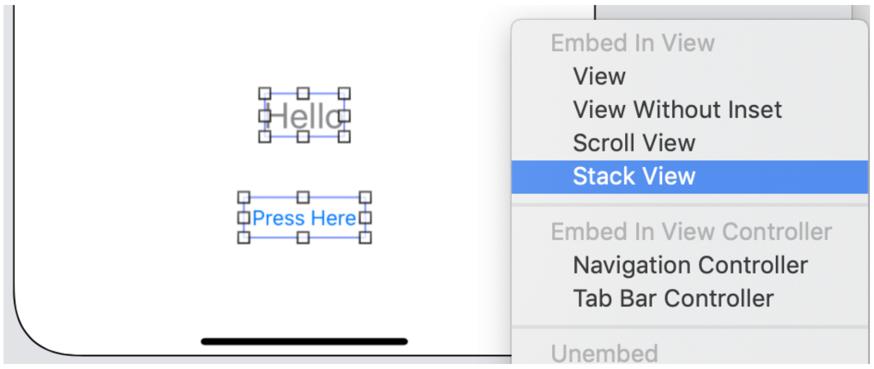
Layout the components

UIKit introduced `StackView`s a while back. They allow you to group other elements vertically or horizontally so that you can organize your screen into pieces that will shrink and grow depending on screen size, orientation, and whether the app is sharing the screen with other apps.

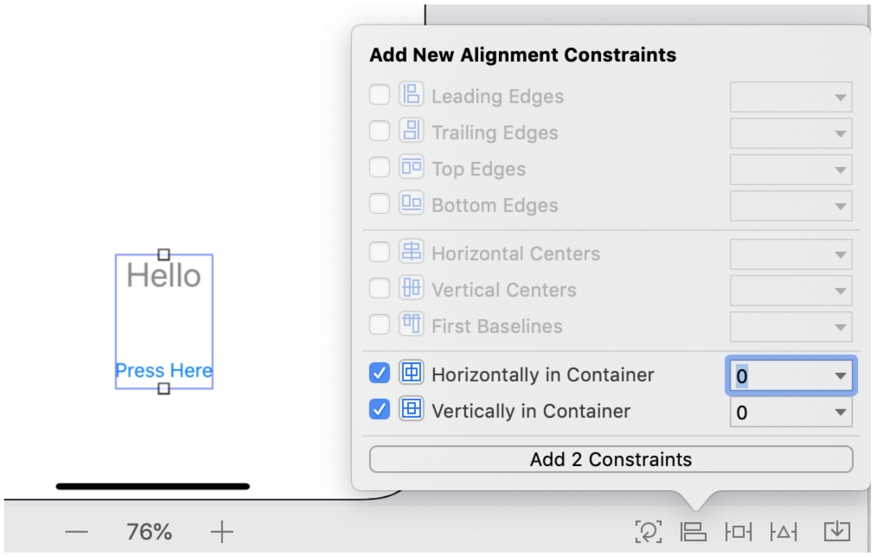
When we get to SwiftUI, we will use vertical stack views called `VStack`s, horizontal stack views called `HStack`s and even stack views whose axis is perpendicular to the screen called `ZStack`s.

Let's group our label and button inside a vertical stack view.

Select both the `Label` and the `Button` and either tap on the button at the bottom right to embed them in a `StackView` or use the menu item Editor > Embed In > Stack View.



Select the StackView and center it horizontally and vertically.



Add New Alignment Constraints

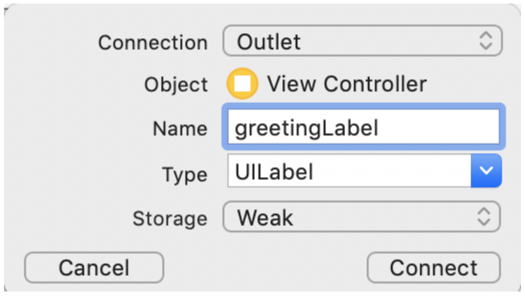
☐	Leading Edges	
☐	Trailing Edges	
☐	Top Edges	
☐	Bottom Edges	
☐	Horizontal Centers	
☐	Vertical Centers	
☐	First Baselines	
✓	Horizontally in Container	0
✓	Vertically in Container	0

Add 2 Constraints

Connect to the code

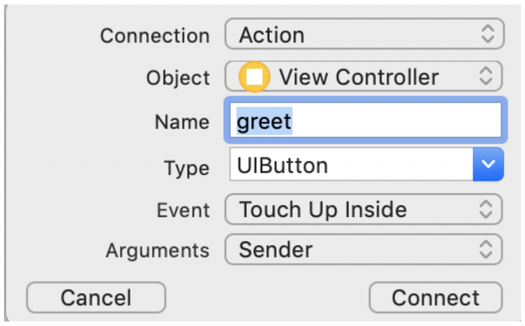
Let's connect our Button and Label to code. Bring up *ViewController.swift* in the Assistant Editor.

Control-Click and Drag from the Label to inside of the ViewController class and release to create an outlet. Set the name to greetingLabel. Its type should be UILabel.



Connection	Outlet
Object	View Controller
Name	greetingLabel
Type	UILabel
Storage	Weak

Also, Control-Click and Drag from the `Button` to inside of the `ViewController` class to create an action. Make sure the connection is `Action`, set the name to `greet`, and the type to `UIButton`.


Connection: Action
Object: View Controller
Name: greet
Type: UIButton
Event: Touch Up Inside
Arguments: Sender

The generated code

Here are the contents of *ViewController.swift*. Note, I specified that both the outlet and action are `private` and I removed the `viewDidLoad()` method.

01/VisualGoodbye/VisualGoodbye/ViewController.swift

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet private weak var greetingLabel: UILabel!

    @IBAction private func greet(_ button: UIButton) {
    }
}
```

Note that in general I present

- existing code like this: `greetingLabel: UILabel!`

- new or featured code that I want to call out or discuss like this: `private`
- comments like this: *// this is a comment*
- and console output like this: `Hello, World!`
- the path to the file the code comes from is given above the listing

Implement the action

Back to our app.

All that remains is to implement the action. When the `Button` is tapped we disable the `Button` and change the `Label`'s contents to `Hello, World!`

01/VisualGoodbye/VisualGoodbye/ViewController.swift

```swift
import UIKit

class ViewController: UIViewController {
    @IBOutlet private weak var greetingLabel: UILabel!

    @IBAction private func greet(_ button: UIButton) {
        button.isEnabled = false
        greetingLabel.text = "Hello, World!"
    }
}
```

If you're counting, that's ten lines of code and we only wrote two of them.

Run the app. Push the button.

Feel good about what we've done for a moment. In the next section I take a contrarian approach that I don't believe in.

But...

In this section we take another look at those ten lines in `ViewController` and categorize them by where they come from, look at other generated code, and take a step back to look at the view hierarchy.

Separating the code

We just built an app where we only wrote these two lines of code.

01/VisualGoodbye/VisualGoodbye/ViewController.swift

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet private weak var greetingLabel: UILabel!

    @IBAction private func greet(_ button: UIButton) {
        button.isEnabled = false
        greetingLabel.text = "Hello, World!"
    }
}
```

But even in this file we *created* all of the other lines of code, we just didn't *explicitly write* them.

These lines were generated for us when we created our app.

01/VisualGoodbye/VisualGoodbye/ViewController.swift

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet private weak var greetingLabel: UILabel!

    @IBAction private func greet(_ button: UIButton) {
        button.isEnabled = false
        greetingLabel.text = "Hello, World!"
    }
}
```

These lines were created when we connected the outlet and action.

01/VisualGoodbye/VisualGoodbye/ViewController.swift

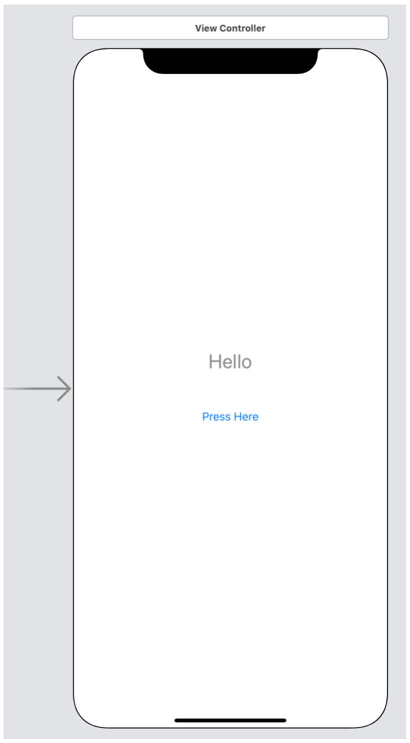
```
import UIKit

class ViewController: UIViewController {
    @IBOutlet private weak var greetingLabel: UILabel!

    @IBAction private func greet(_ button: UIButton) {
        button.isEnabled = false
        greetingLabel.text = "Hello, World!"
    }
}
```

There is a lot of code in other files that was generated when we created the project that we never looked at. I'm not going to worry about those.

But what about our storyboard?

Generated XML

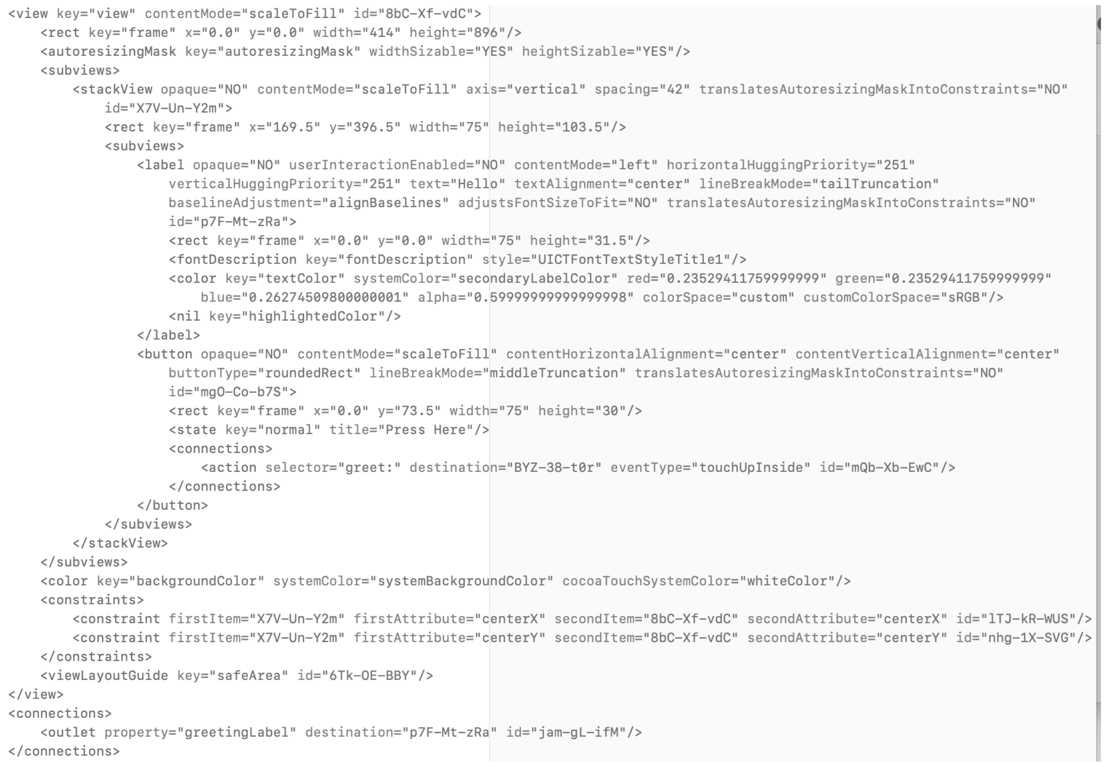
We dragged a label and a button onto the scene in our storyboard and we configured them so they looked like this.



Xcode persists all of our work in the storyboard as XML. You can argue that this is code that we create without writing it.

I disagree with this point of view. True you "create" it, but it is not code that you write. However you feel, let's take a look at the generated XML anyway.

Option - Click on *Main.storyboard* and choose Open As > Source Code and you'll see something like this.



```xml
<view key="view" contentMode="scaleToFill" id="8bC-Xf-vdC">
    <rect key="frame" x="0.0" y="0.0" width="414" height="896"/>
    <autoresizingMask key="autoresizingMask" widthSizable="YES" heightSizable="YES"/>
    <subviews>
        <stackView opaque="NO" contentMode="scaleToFill" axis="vertical" spacing="42" translatesAutoresizingMaskIntoConstraints="NO"
            id="X7V-Un-Y2m">
            <rect key="frame" x="169.5" y="396.5" width="75" height="103.5"/>
            <subviews>
                <label opaque="NO" userInteractionEnabled="NO" contentMode="left" horizontalHuggingPriority="251"
                    verticalHuggingPriority="251" text="Hello" textAlignment="center" lineBreakMode="tailTruncation"
                    baselineAdjustment="alignBaselines" adjustsFontSizeToFit="NO" translatesAutoresizingMaskIntoConstraints="NO"
                    id="p7F-Mt-zRa">
                    <rect key="frame" x="0.0" y="0.0" width="75" height="31.5"/>
                    <fontDescription key="fontDescription" style="UICTFontTextStyleTitle1"/>
                    <color key="textColor" systemColor="secondaryLabelColor" red="0.23529411759999999" green="0.23529411759999999"
                        blue="0.26274509800000001" alpha="0.59999999999999998" colorSpace="custom" customColorSpace="sRGB"/>
                    <nil key="highlightedColor"/>
                </label>
                <button opaque="NO" contentMode="scaleToFill" contentHorizontalAlignment="center" contentVerticalAlignment="center"
                    buttonType="roundedRect" lineBreakMode="middleTruncation" translatesAutoresizingMaskIntoConstraints="NO"
                    id="mgO-Co-b7S">
                    <rect key="frame" x="0.0" y="73.5" width="75" height="30"/>
                    <state key="normal" title="Press Here"/>
                    <connections>
                        <action selector="greet:" destination="BYZ-38-t0r" eventType="touchUpInside" id="mQb-Xb-EwC"/>
                    </connections>
                </button>
            </subviews>
        </stackView>
    </subviews>
    <color key="backgroundColor" systemColor="systemBackgroundColor" cocoaTouchSystemColor="whiteColor"/>
    <constraints>
        <constraint firstItem="X7V-Un-Y2m" firstAttribute="centerX" secondItem="8bC-Xf-vdC" secondAttribute="centerX" id="lTJ-kR-WUS"/>
        <constraint firstItem="X7V-Un-Y2m" firstAttribute="centerY" secondItem="8bC-Xf-vdC" secondAttribute="centerY" id="nhg-1X-SVG"/>
    </constraints>
    <viewLayoutGuide key="safeArea" id="6Tk-OE-BBY"/>
</view>
<connections>
    <outlet property="greetingLabel" destination="p7F-Mt-zRa" id="jam-gL-ifM"/>
</connections>
```

You can see that our view contains a vertical stackview that contains a label and a button. The label has a connection that is an outlet and the button has a connection that is an action.

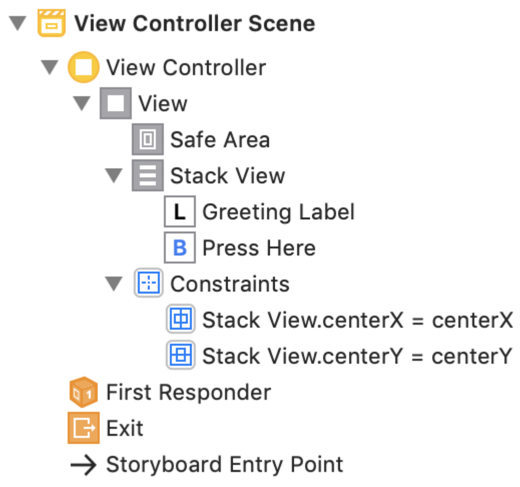
Generally, we're warned against editing the XML by hand. Every item has an ID and we can easily end up in an inconsistent state.

But what if instead of XML, we persisted the storyboard in easy to read and easy to write Swift?

That's actually a big part of what's going on with SwiftUI. If you're on Catalina then you can modify the code and see the Live Preview update in real-time. Or you can add something to the Live Preview and view the changes to the code immediately. Everything will be exposed and you can work in whichever mode you prefer.

The storyboard's hierarchy view

To see the connection between where we are now and at what's coming, take a look at the hierarchy view in the storyboard.



In particular, focus on the part that descends from `View`. Let's write it with the following pseudocode.

```
View {
    StackView (vertical, centered x and y in the view) {
        Label (larger text, lighter color) {
            Text displays "Hello"
        }
        Button {
            Text says "Press Here"
            Action changes what is displayed in the Label
        }
    }
}
```

In the next section, we'll create this in code in Swift using UIKit. In the actual book we do the same using SwiftUI.

Code: All In One

In this section we code up the same example without a storyboard.

Outline the code

Create a new project with all of the same settings as the previous one and name it *ManualGoodbye1*.

We'll do all of our work in the `ViewController`'s `viewDidLoad` method. Here's an outline of what we need to do.

01/ManualGoodbye1/ManualGoodbye1/ViewController.swift

```
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        // create and configure our StackView

        // create and configure our Label

        // create and configure our Button
    }

    // create the button action
}
```

The stack view

Here's how we create the stack view and add it as a subview of the `View`.

01/ManualGoodbye1/ManualGoodbye1/ViewController.swift

```swift
// create and configure our StackView
let stackView = UIStackView()
view.addSubview(stackView)
```

Configure the stack view to be a vertical stack view and to center the items it contains. The horrible line of code with `translatesAutoresizingMaskIntoConstraints = false` is needed to allow our stack view to participate in autolayout.

Wait until you see how much nicer all this is with SwiftUI.

01/ManualGoodbye1/ManualGoodbye1/ViewController.swift

```swift
// create and configure our StackView
let stackView = UIStackView()
view.addSubview(stackView)

stackView.axis = .vertical
stackView.alignment = .center
stackView
    .translatesAutoresizingMaskIntoConstraints = false
```

Finally, center the stack view both horizontally and vertically in the containing view.

01/ManualGoodbye1/ManualGoodbye1/ViewController.swift

```
// create and configure our StackView
let stackView = UIStackView()
view.addSubview(stackView)

stackView.axis = .vertical
stackView.alignment = .center
stackView.translatesAutoresizingMaskIntoConstraints = false

stackView.centerXAnchor
         .constraint(equalTo: view.centerXAnchor)
         .isActive = true
stackView.centerYAnchor
         .constraint(equalTo: view.centerYAnchor)
         .isActive = true
```

Great. We have a view containing a vertical stack view. Run the app and nothing is visible.

Make and display the label

Let's create the label.

01/ManualGoodbye1/ManualGoodbye1/ViewController.swift

```
// create and configure our Label
let greetingLabel = UILabel()
```

Set the label's text size, text color, and text.

01/ManualGoodbye1/ManualGoodbye1/ViewController.swift

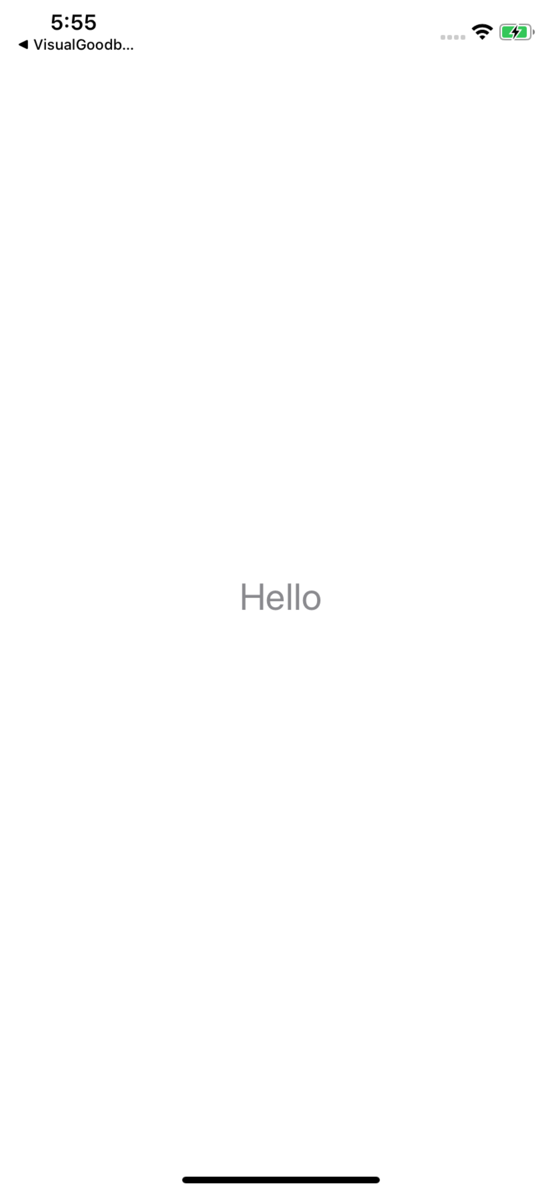
```
// create and configure our Label
let greetingLabel = UILabel()

greetingLabel.textColor = .secondaryLabel
greetingLabel.font
    = UIFont.preferredFont(forTextStyle: .title1)
greetingLabel.text = "Hello"
```

Add the label to the stack view.

01/ManualGoodbye1/ManualGoodbye1/ViewController.swift

```
// create and configure our Label
let greetingLabel = UILabel()

greetingLabel.textColor = .secondaryLabel
greetingLabel.font
     = UIFont.preferredFont(forTextStyle: .title1)
greetingLabel.text = "Hello"

stackView.addArrangedSubview(greetingLabel)
```

Run the app in the simulator and you should see the label containing the word `"Hello"`.



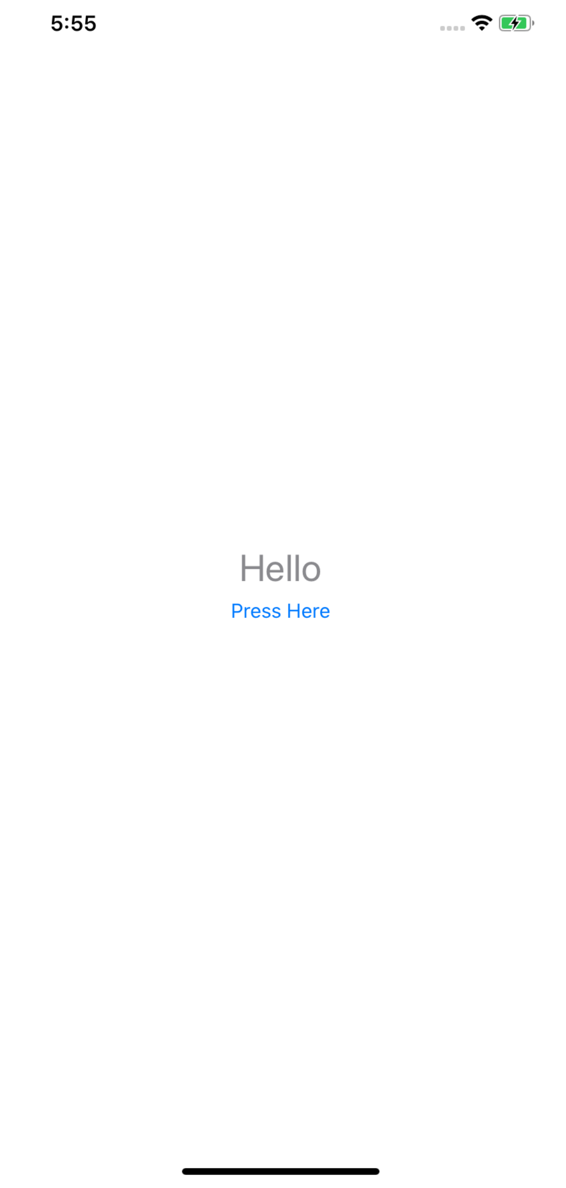
The button

OK, let's create, configure, and add the Button.

01/ManualGoodbye1/ManualGoodbye1/ViewController.swift

```
// create and configure our Button
let greetingButton = UIButton(type: .roundedRect)
greetingButton.setTitle("Press Here", for: .normal)

stackView.addArrangedSubview(greetingButton)
```

Run the app again and the button appears below the label.



Messaging the label

We're about to have a little problem.

Right now the button doesn't do anything. We want it to change the contents of the label.

So what's the problem?

The problem is that the action will be contained in a function that can't see the label. There are several ways to accomplish this. For now we're going to mimic what we did in our visual version. You might remember that `greetingLabel` was a property so it was visible from the action.

Here's the view controller code from the visual version.

01/ManualGoodbye1/ManualGoodbye1/ViewController.swift

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet private weak var greetingLabel: UILabel!

    @IBAction private func greet(_ button: UIButton) {
        button.isEnabled = false
        greetingLabel.text = "Hello, World!"
    }
}
```

Let's create a property for the label in our current version like this.

01/ManualGoodbye1/ManualGoodbye1/ViewController.swift

```
class ViewController: UIViewController {

    private let greetingLabel = UILabel()

    override func viewDidLoad() {
        super.viewDidLoad()

        // ...
        // create and configure our Label
        let greetingLabel = UILabel()

        greetingLabel.textColor = .secondaryLabel   // ...
```

The button action

We can now write our button's action. It takes the sender, which we'll call `button` as an argument and it's labeled `@objc` so that it can be used as an action.

01/ManualGoodbye1/ManualGoodbye1/ViewController.swift

```
// create the button action
@objc private func greet(_ button: UIButton) {
    button.isEnabled = false
    greetingLabel.text = "Hello, World!"
}
```

All that's left is to assemble the pieces. When the user touches up inside of the button, we will call the `greet` method on the view controller. Here's how we express that using the `addTarget` method.

01/ManualGoodbye1/ManualGoodbye1/ViewController.swift

```
// create and configure our Button
let greetingButton = UIButton(type: .roundedRect)
greetingButton.setTitle("Press Here", for: .normal)
greetingButton.addTarget(self,
                         action: #selector(greet),
                         for: .touchUpInside)

stackView.addArrangedSubview(greetingButton)
}
```

The app now works as before. We haven't used a ton of code but it feels like we've just taken the XML and expressed the same information in Swift.

Our view controller

For completeness, here's *ViewController.swift*.

01/ManualGoodbye1/ManualGoodbye1/ViewController.swift

```
import UIKit

class ViewController: UIViewController {

    private let greetingLabel = UILabel()

    override func viewDidLoad() {
        super.viewDidLoad()

        let stackView = UIStackView()
        view.addSubview(stackView)
        stackView.axis = .vertical
        stackView.alignment = .center
        stackView
            .translatesAutoresizingMaskIntoConstraints = false
        stackView.centerXAnchor
            .constraint(equalTo: view.centerXAnchor)
            .isActive = true
        stackView.centerYAnchor
            .constraint(equalTo: view.centerYAnchor).
            isActive = true

        greetingLabel.textColor = .secondaryLabel
        greetingLabel.font
            = UIFont.preferredFont(forTextStyle: .title1)
        greetingLabel.text = "Hello"
        stackView.addArrangedSubview(greetingLabel)

        let greetingButton = UIButton(type: .roundedRect)
        greetingButton.setTitle("Press Here", for: .normal)
        greetingButton.addTarget(self,
                                 action: #selector(greet),
                                 for: .touchUpInside)
        stackView.addArrangedSubview(greetingButton)
    }

    @objc private func greet(_ button: UIButton) {
        button.isEnabled = false
        greetingLabel.text = "Hello, World!"
    }
}
```

Everything is mixed together. It's hard for me to glance at this and see there's a button, a label, and a button action that changes the text in the label. The creation, configuration, layout, and behavior are all mushed together.

In the next section, we'll break it up into more pieces and write it more cleanly.

Code: Smaller Pieces

In this section we break up the code from `viewDidLoad` in `ViewController` into cohesive pieces. When we're done, there will be a lot more code in this version but it will be easier to understand.

Separate components

The components of our app are a `UILabel`, a `UIButton`, and a `UIStackView`. These are all instances of subclasses of `UIView`.

This time when we code up our example we'll create custom subclasses of `UILabel`, `UIButton`, and `UIStackView`.

We won't be able to do this when we use SwiftUI because the corresponding types for buttons, labels, and vertical stacks are structs not classes. You'll see how we handle those differently in the next chapter.

In fact, as you'll see in two chapters, the way we handle data will change dramatically when we move from a single file to multiple files in SwiftUI.

My point is that it might seem like busywork to break the `ViewController` into multiple files, but it will give us context for our discussion in the next two chapters.

Ready?

You can either make another new project with the same settings as the previous two and name it *ManualGoodbye2* or you can refactor

ManualGoodbye1. I'm going to create a new project.

My issue with the previous version of this example was that every time someone has to fix something they have to read through and understand `viewDidLoad()`. It's not that the method was too long so much as that it did too many different things.

The stack view

As a contrast, let's pull out the stack view code into a new file in the *ManualGoodbye2* group that is a subclass of `UIStackView` named `VerticalStackView`.

We'll create a convenience initializer that instantiates a vertical stack view containing the `UIView` subclasses passed in and adds it as a subview of the `UIView` passed in. We add a function named `centered()` that centers the stack view in its super view.

Here's *VerticalStackView.swift*.

01/ManualGoodbye2/ManualGoodbye2/VerticalStackView.swift

```swift
import UIKit

class VerticalStackView: UIStackView {
    convenience init(in view: UIView,
                     containing views: UIView ...)  {
        self.init(arrangedSubviews: views)
        translatesAutoresizingMaskIntoConstraints = false
        alignment = .center
        axis = .vertical
        view.addSubview(self)
    }

    func centered() {
        if let view = superview {
            centerXAnchor
                .constraint(equalTo: view.centerXAnchor)
                .isActive = true
            centerYAnchor
                .constraint(equalTo: view.centerYAnchor)
                .isActive = true
        }
    }
}
```

Add our vertical stack view in the `ViewController`'s `viewDidLoad()` method.

01/ManualGoodbye2/ManualGoodbye2/ViewController.swift

```swift
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        let stackView = VerticalStackView(in: view)
        stackView.centered()
    }
}
```

The label

At the moment, the stack view doesn't contain anything. Let's add a label.

Create a subclass of `UILabel` named `GreetingLabel`. Again, let's put the code that creates and configures our label in a convenience initializer.

Here's *GreetingLabel.swift*.

01/ManualGoodbye2/ManualGoodbye2/GreetingLabel.swift

```
import UIKit

class GreetingLabel: UILabel {
    convenience init(displaying text: String) {
        self.init()
        textColor = .secondaryLabel
        font = UIFont.preferredFont(forTextStyle: .title1)
        self.text = text
    }
}
```

In `ViewController` we add a property for the label and add it to our stack view in `viewDidLoad()`.

01/ManualGoodbye2/ManualGoodbye2/ViewController.swift

```
class ViewController: UIViewController {
    private let greetingLabel
        = GreetingLabel(displaying: "Hello")

    override func viewDidLoad() {
        super.viewDidLoad()
        let stackView
            = VerticalStackView(in: view,
                                containing: greetingLabel)
        stackView.centered()
    }
}
```

The button

A button needs to know two things: what it displays and what it does when tapped. Create a subclass of `UIButton` named `GreetingButton` with a convenience initializer that does this.

01/ManualGoodbye2/ManualGoodbye2/GreetingButton.swift

```
import UIKit

class GreetingButton: UIButton {
    convenience init(title: String,
                     target: AnyObject,
                     selector: Selector) {
        self.init(type: .roundedRect)
        setTitle(title, for: .normal)
        addTarget(target,
                  action: selector,
                  for: .touchUpInside)
    }
}
```

We saw in the previous section that we don't need to create a property for the button, so we'll instantiate it and add it to the stack view. Here's how we add the button and action to *ViewController.swift*.

01/ManualGoodbye2/ManualGoodbye2/ViewController.swift

```
import UIKit

class ViewController: UIViewController {
    private let greetingLabel = GreetingLabel(displaying: "Hello")

    override func viewDidLoad() {
        super.viewDidLoad()
        let stackView
            = VerticalStackView(in: view,
                                containing: greetingLabel,
                                GreetingButton(title: "Press Here",
                                               target: self,
                                               selector: #selector(greet)))
        stackView.centered()
    }

    @objc private func greet(_ button: UIButton) {
        button.isEnabled = false
        greetingLabel.text = "Hello, World!"
    }
}
```

Run the app and it looks and behaves exactly like the previous versions.

Where we are

You can argue that there's a lot more code in this version than in the others but I think it's a lot easier to read.

The code to create and configure the widgets is in *GreetingLabel.swift* and *GreetingButton.swift*.

The code that describes the layout is in *VerticalStackView.swift*.

ViewController.swift contains little more than our visual version and it has no public API.

Now we're ready for our call to action. Now we're ready for SwiftUI. We'll begin with SwiftUI in the next chapter.

The remainder of this chapter contains information about the book that you might find useful.