HACKING WITH SWIFT

SWIFTUI BY EXAMPLE

COMPLETE REFERENCE GUIDE

Get hands-on solution
for common prob

FREE SAMPLE

Paul Hudson

Chapter 1
Introduction

A brief explanation of the basics of SwiftUI

Don’t panic!

Yes, I know SwiftUI is a massive new thing that might seem completely alien at first, but in this guide you’ll get lots of hands-on help to guide you through understanding how SwiftUI works so you can be up and running with it *fast*.

If you’re already familiar with Swift, then everything you know about UIKit is still useful, and will be for a good few years.

Everything you knew about Swift hasn’t changed, so that’s all useful.

Everything you knew about the way iOS apps look and feel hasn’t changed.

What *does* change with SwiftUI is how we make our apps. Yes, Apple introduced a lot of new things to learn, but once you’re past the basics you’ll start to recognize lots of common ground with UIKit.

So, again: don’t panic. If you’re already proficient with UIKit you can pick up the basics of SwiftUI in less than a day. And if you’re approaching SwiftUI without any knowledge of UIKit, that’s okay too because you’ll find SwiftUI gives you an unparalleled app building experience.

So, trust me: you’ve got this.

Before we start: Please make sure you have macOS 11 installed alongside Xcode 12. This combination gives you the most recent version of SwiftUI, which in turn delivers the most functionality and performance.

What is SwiftUI?

SwiftUI is a user interface toolkit that lets us design apps in a declarative way. That's a fancy way of saying that we tell SwiftUI how we want our user interface to look and work, and it figures out how to make that happen as the user interacts with it.

Declarative UI is best understood in comparison to imperative UI, which is what iOS developers were doing before iOS 13. In an imperative user interface we might make a function be called when a button was clicked, and inside the function we'd read a value and show a label – we regularly modify the way the user interface looks and works based on what's happening.

Imperative UI causes all sorts of problems, most of which revolve around *state*, which is another fancy term meaning "values we store in our code." We need to track what state our code is in, and make sure our user interface correctly reflects that state.

If we have one screen with one Boolean property that affects the UI, we have two states: the Boolean might be on or off. If we have two Booleans, A and B, we now have four states:

- A is off and B is off
- A is on and B is off
- A is off and B is on
- A is on and B is on

And if we have three Booleans? Or five? Or integers, strings, dates, and more? Well, then we have lots more complexity.

If you've ever used an app that says you have 1 unread message no matter how many times you try to tell if you've read the darn thing, that's a state problem – that's an imperative UI problem.

In contrast, declarative UI lets us tell iOS about all possible states of our app at once. We might say if we're logged in show a welcome message but if we're logged out show a login button. We don't need to write code to move between those two states by hand – that's the

ugly, imperative way of working!

Instead, we let SwiftUI move between user interface layouts for us when the state changes. We already told it what to show based on whether the user was logged in or out, so when we change the authentication state SwiftUI can update the UI on our behalf.

That's what it means by declarative: we aren't making SwiftUI components show and hide by hand, we're just telling it all the rules we want it to follow and leaving SwiftUI to make sure those rules are enforced.

But SwiftUI doesn't stop there – it also acts as a cross-platform user interface layer that works across iOS, macOS, tvOS, and even watchOS. This means you can now learn one language and one layout framework, then deploy your code anywhere.

SwiftUI vs Interface Builder and storyboards

Every experienced iOS developer is familiar with Interface Builder and storyboards, and perhaps even XIBs too. They might not *like* them, but they are at least *familiar* with them. If you haven't used these before, you should just skip this bit.

Still here? That means you've used IB before and are probably curious how SwiftUI is different. Well, let me ask you this: have you ever edited a storyboard or XIB by hand?

Probably not. Well, apart from that one time once, but broadly the answer is *no* – storyboards and XIBs contain a fairly large amount of XML that isn't easy to read or easy to edit.

Worse, storyboards have a habit of growing larger and larger over time. Sure, they might *start off* small, but then you add another view controller and another, and another, and suddenly you realize that you have ten screens of data in a single file, and any source control changes you make are suddenly quite painful.

But although being a single point of failure isn't great, and it's basically impossible to see what's changed when someone opens a pull request with a storyboard modification, storyboards and XIBs have a bigger problem.

You see, Interface Builder doesn't know much about our Swift code, and our Swift code doesn't know much about Interface Builder. As a result, we end up with lots of unsafe functionality: we Ctrl-drag from IB into our code to connect something to an action, but if we then delete that action the code still compiles – IB really doesn't mind if the code it intends to call no longer exists.

Similarly, when we create view controllers from a storyboard or dequeue table view cells, we use strings to identify important objects in our code – a system so pervasive, it even has its own name: "stringly typed APIs". Even then we need to use typecasts because Swift can't know that the table view cell it got back is actually a **MooncakeTableViewCell**.

These problems exist because IB and Swift are very separate things. This isn't a huge surprise – not only does Interface Builder date from way before the original Mac OS X was a thing, but it's also very much designed around the way Objective-C works.

SwiftUI makes a hard break from that past. It's a Swift-only framework, not because Apple has decided that it's time for Objective-C to die, but because it lets SwiftUI leverage the full range of Swift's functionality – value types, opaque return types, protocol extensions, and more.

Anyway, we'll get onto exactly how SwiftUI works soon. For now, the least you need to know is that SwiftUI fixes many problems people had with the old Swift + Interface Builder approach:

- We no longer have to argue about programmatic or storyboard-based design, because SwiftUI gives us both at the same time.
- We no longer have to worry about creating source control problems when committing user interface work, because code is much easier to read and manage than storyboard XML.
- We no longer need to worry so much about stringly typed APIs – there are still some, but significantly fewer.
- We no longer need to worry about calling functions that don't exist, because our user interface gets checked by the Swift compiler.

So, I hope you'll agree there are lots of benefits to be had from moving to SwiftUI!

Frequently asked questions about SwiftUI

Lots of people are already asking me questions about SwiftUI, and I've done my best to ask other people who know much more to try to find definitive answers as appropriate.

So, here goes…

Which to learn: SwiftUI or UIKit?

This question has been asked so many times I added a dedicated chapter to this book so I could go into more detail: **answering the big question: should you learn SwiftUI, UIKit, or both?**

Where can SwiftUI be used?

SwiftUI runs on iOS 13, macOS 10.15, tvOS 13, and watchOS 6, or any future later versions of those platforms. This means if you work on an app that must support iOS N-1 or even N-2 – i.e., the current version and one or two before that – then it will be a year or so before you can even think of moving to SwiftUI.

However, it's important you don't think of SwiftUI as being a multi-platform framework similar to Java's Swing or React Native. The official line seems to be that SwiftUI is not a multi-platform framework, but is instead a framework for creating apps on multiple platforms.

That might sound the same, but there's an important difference: Apple isn't saying that you can use identical SwiftUI code on every platform, because some things just aren't possible – there's no way to use the Apple Watch's digital crown on a Mac, for example.

Does SwiftUI replace UIKit?

No. Many parts of SwiftUI directly build on top of existing UIKit components, such as **UITableView**. Of course, many other parts don't – they are new controls rendered by SwiftUI and not UIKit.

But the point isn't to what extent UIKit is involved. Instead, the point is that we don't *care*. SwiftUI more or less completely masks UIKit's behavior, so if you write your app for SwiftUI and Apple replaces UIKit with a singing elephant in two years you don't have to care – as long as Apple makes the elephant compatible with the same methods and properties that UIKit exposed to SwiftUI, your code doesn't change.

Does SwiftUI use Auto Layout?

While Auto Layout is certainly being used for some things behind the scenes, it's not exposed to us as SwiftUI developers. Instead, it uses a flexible box layout system that will be familiar to developers coming from the web.

Is SwiftUI fast?

SwiftUI is *screamingly* fast – in all my tests so far it seems to outpace UIKit. Having spoken to the team who made it I'm starting to get an idea why: first, they aggressively flatten their layer hierarchy so the system has to do less drawing, but second many operations can bypass Core Animation entirely and go straight to Metal for extra speed.

So, yes: SwiftUI is incredibly fast, and all without us having to do any extra work.

Why can't I see the preview of my code?

When working with SwiftUI it's helpful to be able to see both the code for your view and a preview of your view – how it looks – side by side. If you can see the code and not the preview, chances are you need to go to the Editor menu and make sure Canvas is enabled.

How closely does the code match the preview?

When you make any change to the preview it will also update the generated code. Similarly, if you change the code it will update the user interface too. So, the code and preview are identical and always stay in sync.

Why do my colors look slightly off?

SwiftUI gives us standard system colors like red, blue, and green, but these aren't the pure red, blue, and green you might be used to from **UIColor**. Instead, these are the new style colors that automatically adapt to light and dark mode, which means they will look brighter or darker depending on your system appearance.

Is UIKit dead?

No! Apple introduced huge amounts of new functionality at both WWDC19 and WWDC20. If Apple are still doing WWDC talks about new features in UIKit, you're quite safe – there's no risk of them retiring it by surprise.

Can you mix views from SwiftUI and UIKit?

Yes! You can embed one inside the other and it works great. s

Answering the big question: should you learn SwiftUI, UIKit, or both?

Of all the SwiftUI questions I've been asked, one comes up more than any other: "I'm learning Swift: should I learn SwiftUI or do I need to learn UIKit as well?"

The answer folks seem to want to hear is "forget that old UIKit thing – you should focus on SwiftUI!" However, the simple truth is that the vast majority of people won't find success with that advice, and it's worth explaining why in a little detail.

Before I get into detail I want to make one thing clear: SwiftUI is a remarkable user interface framework, and is 100% absolutely going to be the future of app development on Apple's platforms. However, if you want to build great apps *today* – or indeed any point in the next one to two years or so – you also 100% absolutely need some knowledge of UIKit, particularly if you intend to make app development your career.

OK, with that out of the way, the problems with focusing on SwiftUI while ignoring UIKit come down to three things:

1. Limited API coverage.
2. Limited adoption.
3. Limited support.

Let's break that down…

Limited API coverage

Regardless of whether you want to work for a company or just build hobby apps in your spare time, one drawback of SwiftUI is that it does not currently have the same broad API coverage as UIKit.

For example, if you want to show rich editable text you would use **UITextView** in UIKit, but SwiftUI's own **TextEditor** will only handle plain strings. Or if you want to show a text field in an alert, it's right there in **UIKit** but not possible with SwiftUI.

This isn't Apple being lazy, and instead seems to be deliberate: rather than releasing wrappers for all their APIs up front then having to make changes later, they are instead taking a much more cautious approach and adding APIs incrementally. This should (I hope!) reduce the number of breaking changes we see in the future, because it gives Apple's engineers more time to hone the subset of APIs they intend to ship.

A lot of the time you'll find workarounds, but honestly it's tiring when you know a particular thing is trivial in UIKit but hard if not impossible in SwiftUI. Sometimes it's even simple things: how can you change the separator insets on a table? Or disable smart quotes in a text field? These are a single line of code in UIKit, but unavailable in SwiftUI.

As each year goes by I fully expect to see more functionality added to SwiftUI to bring it to parity with UIKit, but right now many key components are still missing.

Limited adoption

SwiftUI was only announced at WWDC2019, and is available in iOS 13 devices or later. This immediately means that:

- Almost every app written to date uses UIKit.
- Any app that needs to support iOS n-2 or earlier (e.g. iOS 12) cannot even begin to switch to SwiftUI for a year or more.

This means that if you intend to get a job as an iOS developer in the next two years, UIKit experience is effectively mandatory because that's what existing codebases use. In fact, I fully expect UIKit to still be the dominant UI platform in a year or two. No one – not even Apple, I think! – expects the iOS community to migrate over to SwiftUI at any sort of rapid pace. There's a lot of code, a lot of time, and a lot of money invested in UIKit apps, and it has a long and happy life ahead of it.

Some folks try to draw parallels between adoption of Swift and adoption of SwiftUI, which I don't think is helpful. Adoption of Swift was fast because it worked across every one of the frameworks Apple supported (UIKit, SpriteKit, etc), and also already supported iOS n-1, so many companies could switch to it immediately.

Again, I want to reiterate that SwiftUI is absolutely going to be the future of development for Apple's platforms, but it will take a long time to gain adoption at the level of UIKit.

In the meantime, SwiftUI is the perfect candidate for smaller apps, personal apps, hobby apps, prototype apps, or general experimentation. And if you're lucky enough to join a company that uses SwiftUI exclusively, enjoy it!

Limited support

UIKit has been around over ten years now, which means a) almost every problem you might face has probably already been faced and solved by others, and b) there are lots of libraries out there that provide extensions and customizations.

While some learners might imagine that senior developers hold vast amounts of UIKit in their head, the simple truth is that we all use Google, Stack Overflow, Hacking with Swift, and more to find solutions to problems. When you're desperate that might literally be pasting error messages into a website, but regardless of how you get answers it saves a *lot* of time finding them online.

SwiftUI, simply by virtue of being significantly newer, has significantly fewer solutions available. In fact, it's common to look for things that no one has tried before – you're literally the first person. That can be a lot of fun, but if you have an actual project that you actually want to ship it can also be a frustrating time sink.

So… are you saying I shouldn't learn SwiftUI?

No! SwiftUI is great fun to work with, and you can build marvelous things with it. The whole rest of this book is designed to help you get started with SwiftUI as quickly and efficiently as possible – I wouldn't have written it if I didn't think SwiftUI was awesome.

What I'm trying to say is that the existence of SwiftUI hasn't somehow rendered UIKit obsolete: if you intend to get an iOS development job within the next two years, knowing how to use UIKit will either be a firm requirement or a strong bonus.

So, to answer the question directly: yes should get busy learning SwiftUI because it is the future of app development on Apple's platforms, but you still need to learn UIKit because those skills will be useful for years to come.

As each year goes by, all three problems listed above will become reduced as SwiftUI grows in strength, adoption, and support, and as SwiftUI grows UIKit will start to shrink. However, for now at least, you really do need both.

How to follow this quick start guide

This guide is called SwiftUI by Example, because it focuses particularly on providing as many examples as possible, with each one solving real problems you'll face every day.

I have literally tried to structure this so that almost every entry starts with "How to…" because this is about giving you hands-on code you can use in your own projects immediately. That also means I've tried to get to the point as fast as possible and stay there, so if you're looking for a longer, slower introduction to SwiftUI I'm afraid this isn't it.

Already got some experience?

If you've already grabbed the basics of SwiftUI and just want code that solves your problems, by all means just jump in wherever interests you.

My code examples are specifically written for folks who are following along more or less linearly, so if you're want to make those changes you may need to do a little light editing to make it fit your code.

Just starting out?

If you're just starting out with SwiftUI you should read this guide in a roughly linear order – just keep reading and clicking Next until you're done. As far as possible I've written the guide so that later chapters build on earlier ones, so a linear approach really is a good idea.

If this is you, you should start by creating a new iOS app using the App template. It doesn't matter what you call it, but I would like you to make sure and select SwiftUI for your interface and SwiftUI App for your life cycle, otherwise the rest of this guide will be very confusing indeed.

Migrating from UIKit to SwiftUI

If you've used UIKit before, many of the classes you know and love map pretty much directly to their SwiftUI equivalents just by dropping the **UI** prefix. That doesn't mean they are the same thing underneath, just that they have the same or similar functionality.

Here's a list to get you started, with UIKit class names followed by SwiftUI names:

- **UITableView**: **List**
- **UICollectionView**: **LazyVGrid** and **LazyHGrid**
- **UILabel**: **Text**
- **UITextField**: **TextField**
- **UITextField** with **isSecureTextEntry** set to true: **SecureField**
- **UITextView**: **TextEditor** (plain strings only)
- **UISwitch**: **Toggle**
- **UISlider**: **Slider**
- **UIButton**: **Button**
- **UINavigationController**: **NavigationView**
- **UIAlertController** with style **.alert**: **Alert**
- **UIAlertController** with style **.actionSheet**: **ActionSheet**
- **UIStackView** with horizontal axis: **HStack**
- **UIStackView** with vertical axis: **VStack**
- **UIImageView**: **Image**
- **UISegmentedControl**: **Picker**
- **UIStepper**: **Stepper**
- **UIDatePicker**: **DatePicker**
- **UIProgressView**: **ProgressView** with a value
- **UIActivityIndicatorView**: **ProgressView** without a value
- **NSAttributedString**: Incompatible with SwiftUI; use **Text** instead.

There are many other components that are exclusive to SwiftUI, such as a stack view that lets us build things by depth rather than horizontally or vertically.

What's in the basic template?

Tip: You might think this chapter is totally skippable, but unless you're a Swift genius chances are you should read to the end just to be sure.

The basic App template gives you the following:

1. YourProjectName.swift. This performs an initial set up, then creates and displays your initial view.
2. ContentView.swift. This is our initial piece of user interface. If this were a UIKit project, this would be the **ViewController** class that Xcode gave us.
3. Assets.xcassets. This is an asset catalog, which stores all the images and colors used in our project.
4. Info.plist is a property list file, which in this instance is used to store system-wide settings for our app – what name should be shown below its icon on the iOS home screen, for example.
5. A group called Preview Content, which contains another asset catalog called Preview Assets.

And that's it – it's a pleasingly small amount of code and resources, which means we can build on it.

The part we really care about – in fact, here it's the only part that matters – is ContentView.swift. This is the main piece of functionality for our app, and it's where we can start trying out various SwiftUI code in just a moment.

First, though: what makes ContentView.swift get shown on the screen?

Well, if you remember I said that YourProjectName.swift is responsible for managing the way your app is shown. Obviously it's not actually called that – it will be named according to the project name you chose when creating your poject.

Go ahead and open this file now, and you'll see code like this in there:

```
@main
struct YourProjectName: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

That code creates a new **ContentView** instance (that's the main piece of functionality we'll be looking at soon), and places it inside a window group so it's visible onscreen. It's effectively bootstrapping our app by showing the first instance of **ContentView**, and from there it's over to us – what do you want to do?

Open ContentView.swift and let's look at some actual SwiftUI code. You should see code like this:

```
import SwiftUI

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

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

That's not a lot of code, but it does pack in a great deal.

First, notice how **ContentView** is a struct. Developers familiar with UIKit will know that this is *huge* – we get to benefit from all the immutability and simplicity of values types for our user interface! Folks who *aren't* familiar with UIKit… well, just nod and smile – you never knew the pain we used to have.

Second, **ContentView** conforms to the **View** protocol. Everything you want to show in SwiftUI needs to conform to **View**, and really that means only one thing: you need to have a property called **body** that returns some sort of **View**.

Third, the return type of **body** is **some View**. The **some** keyword was introduced in Swift 5.1 and is part of a feature called **opaque return types**, and in this case what it means is literally "this will return some sort of **View** but SwiftUI doesn't need to know (or care) what."

Important: Returning **some View** means that the **body** property will return something that conforms to the **View** protocol. You can't forget to return anything at all – the Swift compiler will refuse to build your code.

Fourth, inside the **body** property there's **Text("Hello World")**, which creates a label of the text "Hello World".

Fifth, that **Text** view has a **padding()** method call below it. In SwiftUI this actually creates a new view with padding around it, rather than changing the existing **Text** view. As a result, we call these *modifiers* because they create modified content, as opposed to *methods*.

Finally, below **ContentView** is a similar-but-different struct called **ContentView_Previews**. This *doesn't* conform to the **View** protocol because it's specifically there to show view previews inside Xcode as opposed to be on-screen in a real app. This code is only built into the finished product when our app runs in a debug environment because it doesn't make sense in a production app.

We'll look at each of these components in much more detail soon enough, but first let's take a look at that **Text** component…

Dedication

Inside Apple it took an extraordinary amount of effort to design, build, test, document, and ship SwiftUI. As third-party developers we only really see the end result – when a senior Apple staffer gets on stage at WWDC and shows it off to huge applause, when we download the new Xcode to see a huge amount of new functionality, and when we start our own journey of figuring out how to make best use of these incredible new tools.

But SwiftUI started *long* before that as a project from inside the watchOS team – about four years before, from what various folks have said.

Four years.

That's about 1500 days when Apple's engineers were working hard to build something they knew would revolutionize the way we worked, and would be the fullest expression of what Swift is capable of for UI development. If you think how much work it took to build SwiftUI as we know it today, imagine how much change it's seen as Swift itself went from 1.0 through to 5.1!

These engineers weren't allowed to talk to the public about their work, and even inside Apple only a certain number of people were disclosed on SwiftUI's existence. In order to make SwiftUI a reality folks from the UIKit team, the Swift team, the Xcode team, the developer publications team, and more, all had to come together in secret to work on our behalf, and even today you won't find them taking credit for their incredible work.

The simple truth is that SwiftUI wouldn't have been possible without the extraordinary efforts of many, many people. I wish I could list them here and thank them personally, but the only ones I can be sure of are the people who had "SwiftUI engineer" as their job title during a WWDC session or were people I spoke to in the labs.

So, this book is dedicated to Dave Abrahams, Luca Bernardi, Kevin Cathey, Curt Clifton, Nate Cook, Michael Gorbach, John Harper, Taylor Kelly, Kyle Macomber, Raj Ramamurthy, Matt Ricketson, Jacob Xiao, and all the dozens of other folks who worked so hard to make SwiftUI what it is today. We may never know how many more folks from AppKit, UIKit, WatchKit,

Xcode, Swift, developer publications, and beyond helped bring SwiftUI to life, but I hope every one of them feels just blown away by the incredibly positive reactions from our community.

I know WWDC can often be quite the “photo finish” where features land only a day or two before the keynote, but you folks pulled it off and we’re very, very grateful.