Introduction

Welcome 5

How to Use These Books 8

What's Not in this Book Series 14

Book One: Accessibility 16

- Developing an Accessibility Mindset 16
- The 10,000 Foot View: Basics 21
- VoiceOver 34
- The Rotor Control 50
- Custom and System Actions 76
- The Switch Control 88
- Voice Control 99
- Customized Accessibility Content 112
- Behaviors Overview 124
- Focus 126
- Reading Content 136
- Image Adjustments 152
- Textual Context 162
- Navigation and Interactions 172
- Dynamic Type 191
- Adaptable Interfaces 227
- Posting and Consuming Accessibility Notifications 246

User Preferences in SwiftUI 270
Convenience Functions 289
Contextual Scenarios 302
SwiftUI Techniques. 318
Accessibility Settings 344
Assistive Access 355
Accessibility Inspector 356
Accessibility Shortcuts 378
Epilogue: Are You Done? 384

Book Two: Design 387

Can You Learn Design?. 387
What, Exactly, Is Your App? 398
Embracing Scattering Thoughts 416
The Fidelity Problem 428
Finding Your App's Voice 438
Navigation 450
Bars 469
Modality 492
Feedback and Reinforcement 505
Color 520
Copy 541
Playfulness 551
Gestures 563
Device Considerations 572
Feeling at Home 585

Keep or Cut 597
Epilogue: An Endless Cycle of Tiny Improvements 603

Book Three: User Experience 606

Easy Flying 606
Onboarding 612
Launch Time 627
On-Demand Resources 641
Scrolling Performance 653
Quick Look Preview 675
In-App Purchases and Subscriptions 688
Search 706
Loading States 722
State Restoration 736
Custom View Controller Transitions 752
Localization and Internationalizations 774
Micro Animations 791
Haptics 802
Undo and Redo 812
Context Menus 831
Reloading Data 856
Leveraging Hardware Keyboards 877
The Pointer on iPadOS 897
Handoff 913
Text that Works 924
SF Symbols 945

Getting Data In and Out 968
Controller Considerations 987
Your App's Website 999
Epilogue: The Feeling of Feeling Right 1010

Book Four: iOS Technologies and Frameworks 1012

The Heart of iOS 1012
App Clips 1018
AirPrint 1031
AirDrop 1042
CloudKit 1055
CKSyncEngine 1078
Dark Mode 1102
Drag and Drop 1115
Reacting to Low Power Mode 1140
Live Photos 1149
Machine Learning 1163
Maps 1188
Multiple Windows and Scenes 1205
Notifications 1222
PencilKit 1239
Security and Privacy 1257
Safari Services 1275
Siri Shortcuts 1291
Share and Action Extensions 1308
Spotlight Search 1325

Quick Actions 1338
VisionKit 1356
WidgetKit 1378
URL Schemes 1406
Always Onward 1419
The Journal: Quick Tips from over a Decade of iOS Development . . . 1421
Environment Variables and Launch Arguments 1422
Property List Keys 1430
simctl 1435
vmmap 1440
plutil 1445
Leveraging Schemes 1448
Development Assets 1453
How to Focus 1456

Welcome

Welcome to a Best-in-Class iOS App. Here, you will find nearly all the things I've learned from having more than a decade of experience creating iOS apps. I've made award-winning apps, terrible apps, and some in-between that spectrum. The ones that were great, though, all shared common approaches and characteristics - and that's what I want to share with you.

You're reading this because you care: You want to make a great app. But what does that look like now? Surely, the definition has changed over the years.

At first, it was ensuring you filled the iPhone's original 320 by 480 pixel screen with enough information to make the task at hand quick and efficient. Then, it shifted to adapting to new screen sizes. Today, the term "iOS app" is nearly a misnomer - your binary at this moment could run on iOS, tvOS, iPadOS, macOS, visionOS and watchOS.



What used to be a single app can now appear in several places in the Apple ecosystem.

Oh - and SwiftUI. Now we've got a completely new way to not only make an iOS app, but one that also runs across the entire Apple ecosystem. Where does that fit in? Do you still use UIKit, a mix - just one or the other?

This book series is here to help you make sense of all of those questions. The world of iOS development went from big to gigantic, and it's not stopping. That's great news for the platform, but it does make keeping up with the thought of "What makes this great and how do I do that?" a bit harder to comprehend, much less do.

Who Am I?

As I mentioned above, I started out in the iOS industry with some misses. Over the years, I dedicated myself to learning more about what makes iOS software amazing. I've started with solid ideas for apps, even ones people found valuable and solved a problem they had well.

After I launched them, I found out that they didn't stick the landing in so many ways. Hidden gestures were the only means to navigate, colors felt off and many of the things users expected to be present in an iOS app simply weren't there. Discovering early on that design was paramount, and my coding skills could only take me so far, was probably the line in the sand for my career. There were apps I made before it, and after.

As of this writing, my last app (Spend Stack) had critical acclaim, regular press coverage, was featured year-round by Apple's App Store editors, and was even selected as a retail demo app to be used throughout stores on iPads and iPhones all over, eventually leading to its acquisition. At my job at Buffer, applying many of the principles I learned helped us achieve a Webby award for Mobile Apps: Best Practices. None of that happened by accident, though I will be forever thankful for the praise that my work, and things I worked on with other talented engineers, received.

What I have found is this: design isn't the only thing, but it's likely the most important one. And you can prescribe whichever definition you have for "design" here, and there are many. But to me, design is what it looks like, feels like, the emotions it invokes with users, the technology it leverages and more. So in that sense, this book is primarily about design - because design can stem from the way you leverage an API as much as it can from using the right typography.

My hope is that from getting a sense of who I am and where I've been, you can feel confident that someone is writing these words who is in a good position to help you achieve the things you want to with your future iOS endeavors.

With that, allow me to welcome you to A Best-in-Class iOS App. Let's get started.

How to Use These Books

I've organized the book series to be leafed through, though you can certainly approach it in a linear fashion if you'd like. Keep in mind that there is simply a massive amount of information to digest within it. For that reason, I suggest you find topics that are either relevant to a task at hand or an API that simply interests you. Skipping around to what you need, when you need it is what I had in mind when writing this.

There are five main topics covered, and those are:

1. Accessibility
2. Design
3. User Experience
4. iOS Technologies and Frameworks
5. General Tips

I suspect the majority of readers will naturally gravitate towards section 4, as that is where the bulk of what many consider to be core 'iOS Development' topics are found. Though, I encourage you to dissect all the topics, in due time, as best as you can.

With that said, you'll find that each section can be written in one of two ways; Either a traditional, long-form explanation to explain the current section or topic, or a

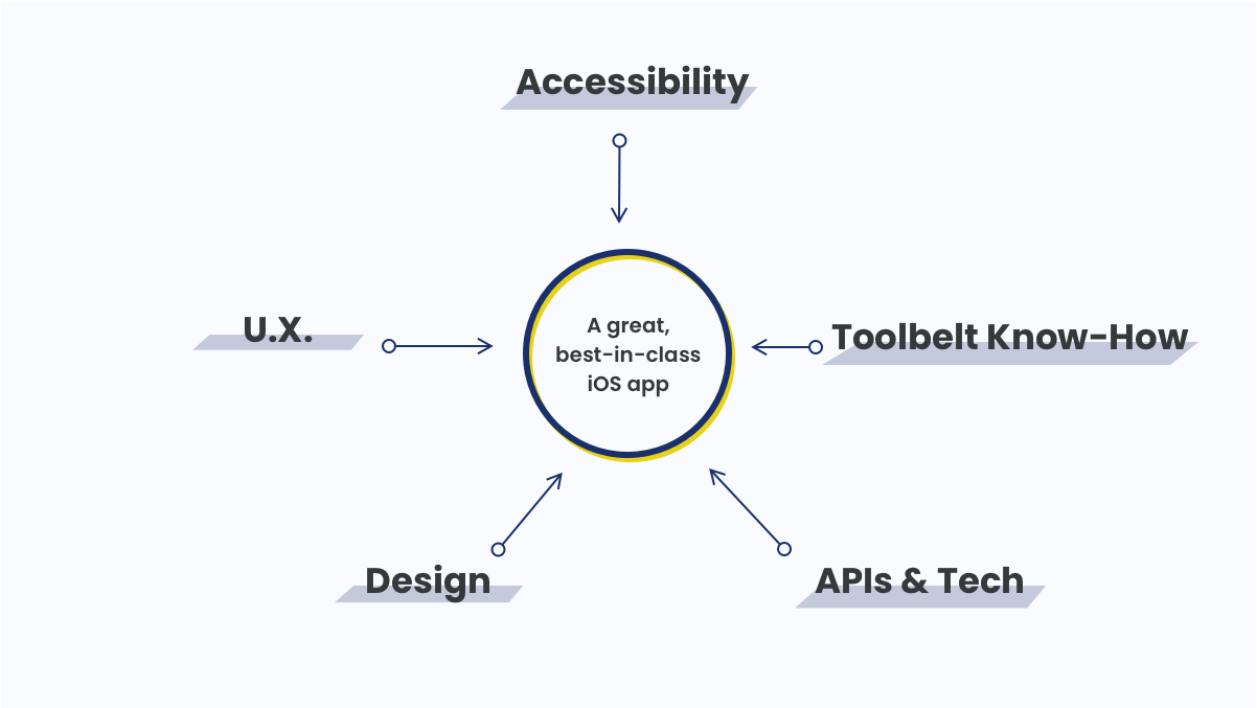
three-pronged approach covering the what, how and some tips on the subject. This is not so much a tutorial book, though you'll certainly find plenty of code samples.

For experienced developers, this book is more about how to use the technologies you're likely familiar with in ways that can help make your app great. For newer ones, it's an introduction to the wide world of iOS and how to use the tools within. Or, just as importantly, it'll help you discover new ones you might not have known about while also paying close attention to design and user experience.

Why these topics?

I've found that knowing about technology or APIs is never enough. It's inherently exciting to engineers such as myself, because these are the things we long to know more about. But unless we apply tact, care and even a hint of resiliency to our software, we won't get far.

That's where design, user experience and accessibility start to enter the picture - and we'll cover each in earnest. Then, knowing how to use the tools available to you to debug difficult problems helps you become a well-rounded engineer, equipped to deal with many of the issues you'll encounter along the way.



A wide range of skills is required to make something great.

A best-in-class app requires all of these things. My goal is to bring your apps to everyone (accessibility), make sure it gets covered by the press and loved by users (design and user experience), ultimately get Apple's eyes on it (iOS technologies and frameworks) while allowing you to be ready for any roadblocks along the way (the tips journal).

To that end, here's how you can think about this book series. It's like the human interface guidelines and documentation combined into short, applicable posts. My hope is that you can thumb through any part of it and find something to take away from the text to make your software shine.

SwiftUI and UIKit

SwiftUI is Apple's most recent foray into making a cross-platform (for Apple's ecosystem, at least), declarative user interface framework. Its nascent nature means it's lacking in some areas, but make no mistake - this is the future of software engineering on Apple's platforms.

It also means it has over a decade of holes to fill that are plugged in nicely by UIKit, Apple's existing user interface framework primarily used by iOS. If you bring in macOS's toolkit, AppKit, then you've got over two decades of catching up to do.

Why mention all of this? Simply to set expectations, I think developers should be familiar with both. The situation harkens back to the early days of the Swift programming language itself when it lacked many features developers relied on such as error handling. Over time, Swift has accommodated all of those shortcomings, and SwiftUI will, too.

But today - using and being comfortable with both frameworks will serve you well. There are simply things you cannot do in SwiftUI, and dipping down into UIKit becomes necessary. To that end, you'll see both frameworks frequently covered.

Where to Find What

Lastly, you may be surprised to find certain topics in sections you may not have expected. For example, take Apple's dynamic type technology. If you're unfamiliar, this API allows text to scale in size according to user preferences. Peeling this back, it deals with:

- Text and any text-based control, such as UIKit's `UILabel` or SwiftUI's `Text` control.

- Accessibility, and allowing text to be read by all users of your app.

So where should it go? In a section that covers the controls that house text itself, or accessibility? The answer will vary from topic to topic, but if you find yourself looking in one section for something you expected to find there and it's not, it may be covered in another tangentially related section.

Chapter Makeup

Almost every chapter in this book is written in the same format. It has three parts:

1. A brief, one to two paragraph, introduction on the topic.
2. A "How it Works" kickoff. Again, I assume you have iOS experience for technical topics, this book's focus is how to take these APIs to their fullest extent underneath the "Tips" section, but I still include the "How it Works" to get you up to speed in case you need a little refresher. In some chapters, this may be the majority of the content because the bulk of the API or topic really *is* solely about how it works and how to use it.
3. Finally, a bullet list style breakdown of APIs, properties, and techniques.

For other topics, a more traditional, long-form chapter makes more sense. For example, some of the design topics.

Corrections or Suggestions

If you find any errors, or there is a topic you'd like to see added or expanded upon - please reach out. Keep in mind this book is never finished, and I look forward to

adding topics that people are passionate about, need help with or want to see expanded upon.

Please use the private Discord server to suggest these types of things. You can find out how to get setup there by reading the "READ FIRST" .pdf included in your package.

If for any reason you opted not to join the community, you can also use other channels. Please send longer form messages by email to **jordan@swiftjectivec.com** and shorter, quick suggestions via X to **@jordanmorgan10**.

What I Assume About You

In short - nothing. The only thing that would be required is that you have a knowledge of programming for the technical aspects of this book. This book will not be focused on teaching you how to program, and instead focuses directly on how to use Apple's many APIs and use them well.

Of course, that works in the other direction as well. Nearly 40% of this book series is focused on design and user experience. So, if you've got a design background some of the content may seem obvious to you. Or, if you've been developing on iOS for a decade, some chapters or elements may come across as elementary.

In short, I write as if I'm teaching these concepts for the first time to whoever may be reading. Experienced developers - enjoy the little nuggets of information about things you are familiar with but can find something new to learn about. New developers, take it all in and realize it takes years to learn all of this. The same goes for all of the design topics.

As a last aside, I know many readers just want to "get to it" and skip the introductions and pleasantries. No matter if you've purchased the entire book series or simply one book, you can skip around as you please to get to the topics that matter to you.

What's Not in this Book Series

I've made every effort to include APIs and topics that will help you make a better app. That's my North Star when I sat down and brainstormed what this book could look like. While there is a large surface area covered, I'd like to be upfront about what's not included. There may be some API here and there that isn't included, but from a major frameworks standpoint, I've opted to not include the following:

- AirPlay
- Game Frameworks (Apple Arcade, Games, Game Center, SceneKit and SpriteKit)
- ARKit
- Apple News Format
- Apple Pay
- Business Chat
- Bonjour
- CareKit, ClassKit, ResearchKit or HealthKit
- Catalyst
- CarPlay
- Exposure Notifications
- MusicKit

- HomeKit
- iBeacon
- Game Center
- Keyboard Extensions
- Wallet and Passkit

In future editions of the series, that list may change with frameworks being added or removed.

Also, please keep in mind that the version of this book series you are currently reading is the shortest version you'll ever have. I continue to update it annually, and when new frameworks and tooling ship - they will also appear in this book accordingly. This is much less a traditional book, it helps to think of it more like versioned software. A living, breathing document that will stay at your side throughout your iOS journey.

Book One: Accessibility

Developing an Accessibility Mindset

Do you remember the very first time you used an iPhone? I certainly do. I bought the first iPhone available back in 2007 when I was a server at a local restaurant. Taking a few double shifts, I stashed away enough to preorder the mythical device from Cupertino. I remember getting the shipping notification on my email address at a local Internet café (remember those?) and I rushed home to meet the delivery man who was just pulling out of my driveway.

"I know what's in this, I couldn't believe you weren't home to get it!" That's exactly what he said to me, in near disbelief that anyone would miss the opportunity to own the nascent piece of hardware. I almost did, but I signed for it and ran to my friend's house who activated it via iTunes on his Mac. Then I turned it on for the very first time, and…

Nothing happened. The screen was black and it wouldn't power on. I sat there in shock, I had waited for this moment and I wanted to peer into the future of personal computing! Can you imagine my disappointment?

Well, you don't have to, because that last paragraph was a lie. But I hope you felt the sense of sorrow creeping over me. Fortunately, my iPhone powered right on - and my world changed in an instant. But too many times, users who absolutely require accessibility features hit that same wall when they try to use iOS apps. Even though

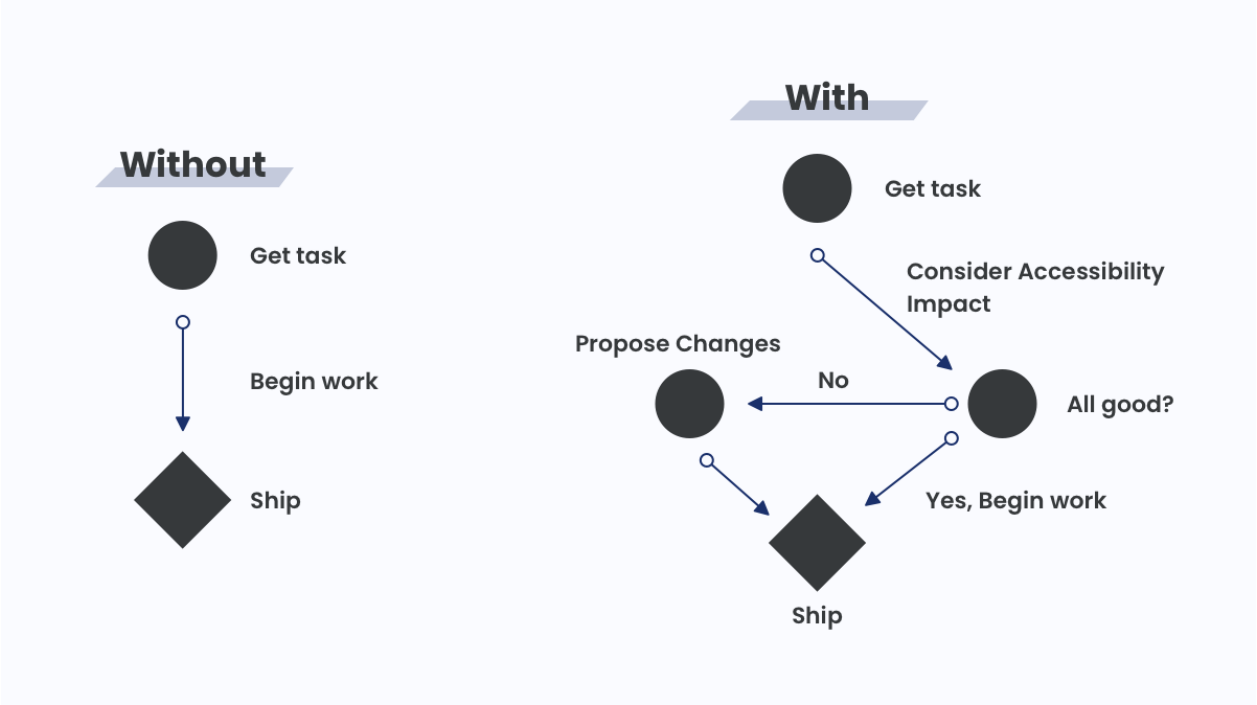
Apple's frameworks are among the best in the industry and give you so much for free, there is still work to be done and many haven't done it. We've got to stop having users turned away at the door simply because they can't use our apps, and we're the only ones who can hold it open.

Where We Are Today

You'd be hard-pressed to find anyone who would advocate against including accessibility features in an iOS app. When speaking to other developers, nearly all of them want to do it and do it right. Yet, many apps lack basic accessibility features and remain that way for years.

So where is the disconnect? Why aren't we doing better as an industry? There are many reasons why, and they range from the obvious ('I don't know what to do') to the bottom line ('It's hard to sell the work as a product manager'). Be that as it may, we can still affect change with a few simple tweaks to how we think of accessibility.

If we're going to have accessible apps, we need to start by developing an accessible mindset. That means you can take yourself as an engineer, designer, manager or anything else in-between to shift your thinking from "It would be nice to do" to "I'm going to get it done". You'll start to notice common pitfalls to avoid, questions to ask earlier and different approaches to take when writing code.



As you can see, developing with an accessible mindset takes longer - but results in better products that work for a much wider audience.

Above we can see the difference. Whereas you may have thought about jumping straight into "Feature A", a person with an accessibility mindset will ask more of themselves before diving into the code, design or specification. The process is much better served when it's full spectrum across disciplines (i.e. everyone is onboard with accessibility) but that's often not the case - which is why there's no better place to start than with you.

Why Bother? Several Reasons.

If you think about the "why" here, there are many practical reasons to have an accessible app. The first, which is nearly impossible to argue against, is simply because it's the right thing to do. Users who have accessibility needs often face an unnecessary amount of friction when using software, and they shouldn't. Simply ensuring everyone

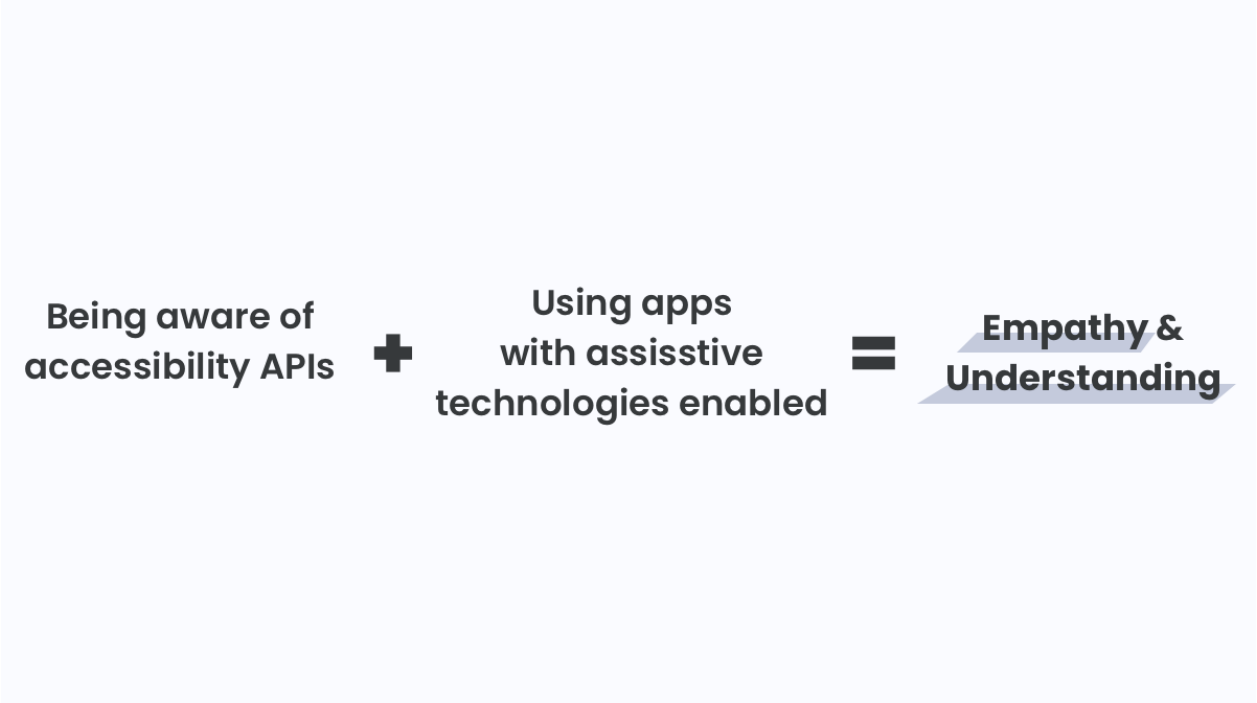
can use your app means you let everyone in, and everyone should get to experience the joy of well-built software. I truly believe that. It has to be a conviction you develop personally, and then hold yourself to it as a standard.

Though, not every organization can (or even wants to) afford to spend engineering effort on something because it's "the right thing to do". Fine, but that's far from the only reason to do accessibility programming and features. At this point, it's a competitive advantage. There are so many popular apps that simply don't do much of it at all. You can literally sway users from them, to you, by simply opening the door. You're supporting their needs while still supporting your app as a business. I can't think of a more apt win-win.

The Elephant in the Room

Another common roadblock with getting accessible apps out the door is simply not knowing where to start. That's okay, the world seems big. You need to start small and build up both your confidence and know-how to get good at it. And the beauty of that? It's absolutely understandable! Programming will always be a moving target and as such, you're always learning new things. That's why this book is truly never "finished", because new APIs will emerge, some will change and new techniques arise. It has to adapt with it. Accessibility is no different in that regard.

So that's what we'll do here, we'll start small. By getting your foundation under you, you can move on to more advanced needs that are available to support. As part of that process, you'll start to get comfortable with the technologies and tools you need to know about to further your understanding. One of the best methodologies to understanding what to support is to simply use your app with those technologies enabled such as screen curtain, Voice Over or Voice Control.



Developers who are effective at accessibility programming have empathy that's fostered by using those same technologies in their own apps.

With that, let's start turning the page on accessibility - maybe starting with you and your future work, we'll begin to make accessibility programming the baseline instead of the exception.

Three Key Takeaways

1. An accessible app is practical from every viewpoint - it's moral, increases usage and leads to more users.
2. You can't cover all the ground you need at once, but you can cover some of the ground over time.
3. An important piece of developing an accessibility mindset is to use the tools you're building for.

The 10,000 Foot View: Basics

Accessibility on iOS covers four main topics, and the technology you'll craft with care into your apps will generally fall into one of these categories:

1. Vision: i.e. "This text is hard to read."
2. Hearing: i.e. "I don't know the context because there are no captions in this video."
3. Physical and Motor Skills: i.e. "I can't progress because I can't touch a button."
4. Literacy and Learning: i.e. "I can't read this, I don't know what's conveyed."

This chapter is all about giving you a kickstart. The rest that follows will be shorter and succinct, with quick wins to apply. Before you know where to use them, it helps to gain a large view of the field so you can make informed decisions on where to start.

As we continue, keep in mind that many of the technologies have SwiftUI counterparts, and those are covered along the way and then recapped in the chapter "SwiftUI Techniques".

UIAccessibility

This is where it all starts. `UIAccessibility` is an informal protocol[1] that most of UIKit (and technically, SwiftUI) leverage to make most of your app accessible out of the box. Using it, you can support several hallmark assistive technologies such as Voice Over, Voice Control and more.

Consider a label in your app, or specifically the `UILabel` class. Imagine we've used one like so:

```
// Xcode -> TenKFig1ViewController.swift
override func viewDidLoad() {
  super.viewDidLoad()

  let label = UILabel(frame: .zero)
  label.text = "Welcome to A Best-in-Class iOS App: The Book"
  view.addSubview(label)
  label.frame = CGRect(x: view.safeAreaInsets.left,
                       y: 100,
                       width: view.bounds.width,
                       height: 54)
}
```

[1] An informal protocol is nothing more than a way to have a type or object extended via an Objective-C category to allow for optional implementation of some functionality. In contrast, a formal protocol is more contractual. These have explicit @optional or @required annotations to define its conformer's responsibilities.

If you were to turn on Voice Over for this control, without doing anything, it would read out the text by default, "Welcome to A Best-in-Class iOS App: The Book". The same is true of SwiftUI:

```
// Xcode -> TenKFig1.swift
var body: some View {
  Text("Welcome to A Best-in-Class iOS App: The Book")
}
```

Voice Over would give us the same result. Whether you leverage SwiftUI or UIKit, most of the standard controls get you 90% of the way there. That's amazing news as a developer! Coincidentally, I believe this is also why accessibility work often gets overlooked. Most of the time, what you have is nearly good enough. That's a testament to the work Apple has done for us, but we've got to meet them the rest of the way.

This is where apps can fall short - the developers may simply not know what's expected of them from here. What is there to do from here? To answer that question, we need a high-level understanding of UIAccessibility's responsibilities. What can it do?

Quite simply - a lot. Taking all of it in is slightly overwhelming, so let's look at the parts that give us quick wins right away:

1. `isAccessibilityElement: Bool` - Controls whether or not the receiver should be recognized by accessibility APIs.
2. `accessibilityTraits: UIAccessibilityTraits` - Helps assistive APIs understand the role of the element (i.e. this is a button or this is static text?)
3. `accessibilityFrame: CGRect` - Communicates where the element is, in screen coordinates, to assistive APIs.

4. `accessibilityLabel: String` - A short, to-the-point string that identifies the element. It should be localized.
5. `accessibilityValue: String` - The element's value if it has one. For example, a text field could have the text "Score" but its value would be the actual score of some sporting event. This should be localized.
6. `accessibilityHint: String` - Briefly describes what would happen if the user interacted with the element. This should be localized.

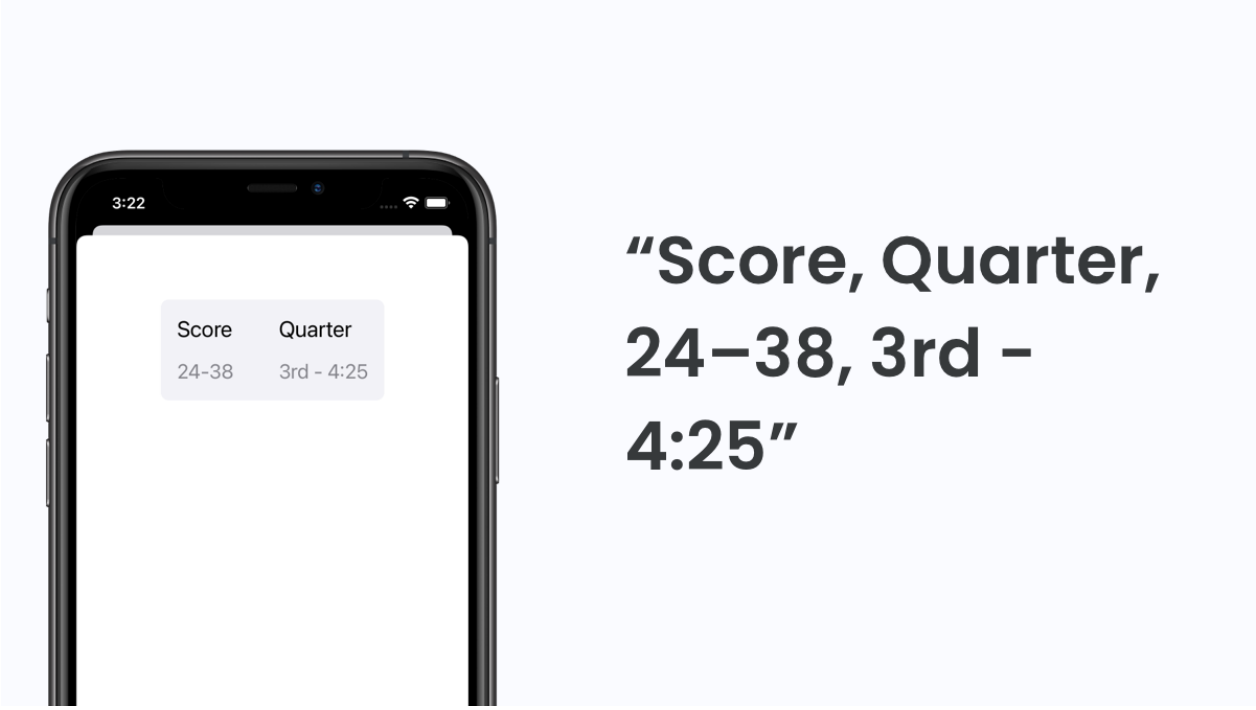
If you can grasp these six elements of `UIAccessibility`, you're going to do well. Each of them opens up your app to nearly every user, regardless of their needs. Of course, there are more advanced features, such as the rotor control, and we'll cover those as well. But keep in mind, this chapter serves to give you a solid foundation.

And really, this list is it - those six things should be your foundation. Try to commit them to memory, and when you think about accessibility this is often where you should start. Though, those won't always be given to us for free. Almost every app has custom user interface elements, and in those cases, we might need to supply some (or all) of these values. We can do that using `UIAccessibilityElement`.

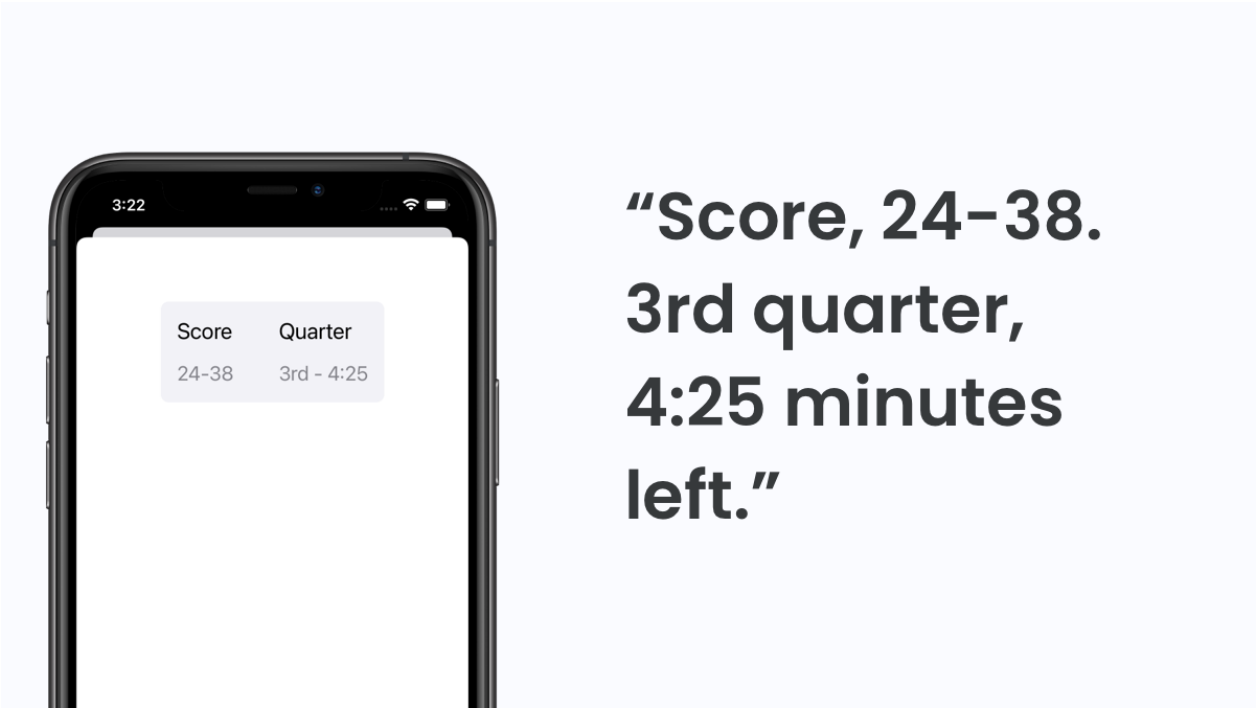
UIAccessibilityElement

When you reach for `UIAccessibilityElement`, you're saying to the system that what you're exposing is something custom, isn't accessible by default and you're going to make it accessible yourself. By looking at the class itself, you may think this is only required when you don't directly inherit from `UIView` or `UIResponder`, but that's not always the case.

For example, consider a view that aggregates several others together as one. In these cases, you may want to consolidate the accessible aspects of it into one item. You can do that using a custom accessibility element. Below, without any work, Voice Over would read out this user interface in Voice Over incorrectly:



But, with some tweaks using `UIAccessibilityElement`, we can get to:



So how would we go about this? Consider that Voice Over reads directionally according to the device’s locale. For example, in many English speaking territories that would mean it reads left to right. Voice Over sees these labels and starts at the top left, progresses right and then down, repeating the same pattern much like how you’d read a book.

And that’s our problem. This way, Voice Over doesn’t quite pull together the right context to read this interface correctly. Though, it does promote some accessibility for free. However, when you let the context get muddled, it can do more harm than good. This is a perfect time to use `UIAccessibilityElement` to patch things up.

Before we look at the code sample, recall what this object can do. It gives us free rein to identify what the accessibility values should be. We can decide where in the interface we say it is, what the labels and values should be and more. With that in mind, I can spot two opportunities to use `UIAccessibilityElement` here.

The first is the score and the actual score label. Second, the quarter and the time left in the quarter labels. If we group these labels together as one, Voice Over will read them the same way that many of us see it and understand it:

```
// Xcode -> TenKFig2ViewController.swift
var customElements: [UIAccessibilityElement] = []

override func viewDidLoad() {
  super.viewDidLoad()

  // Group both score labels together and let Voice Over
  // Know its coordinates
  let groupedScore = UIAccessibilityElement(accessibilityContainer:
view!)
  groupedScore.accessibilityLabel = "\(scoreLbl.text!), \
(scoreAmountLbl.text!)"
  groupedScore.accessibilityFrameInContainerSpace =
scoreLbl.frame.union(scoreAmountLbl.frame)
  customElements.append(groupedScore)

  // Repeat with the quarter labels
  let groupedQuarter =
UIAccessibilityElement(accessibilityContainer: view!)
  groupedQuarter.accessibilityLabel = "Third quarter, \
(quarterActualLbl.text!) minutes left."
  groupedQuarter.accessibilityFrameInContainerSpace =
quarterLbl.frame.union(quarterActualLbl.frame)
  customElements.append(groupedQuarter)

  // Assign them to the view's element
  view.accessibilityElements = customElements
}
```

Using this technique, you'll know when to reach for it if some assistive technologies aren't quite giving you the results you're after. There are other times where it's necessary for elements that have no accessible properties by default such as if you created

interface elements that are not backed by a UIView (i.e. a custom drawn icon). Spinning up an accessibility element for it as we've done above is the right answer in these situations.

Keep in mind, you can get similar results with SwiftUI and its assistive modifiers. Those are covered as well, but I've saved them for the chapter, "SwiftUI Techniques". Going forward, unless the API is easily conveyed with an example for both - any accessibility topics for SwiftUI are saved for its own dedicated chapter.

The Big Three: Labels, Hints and Values

Before closing out on the main concepts, let's finish on the biggest impact properties. We've listed these above, but to refresh your memory they are the `UIAccessibilityLabel`, `UIAccessibilityValue` and `UIAccessibilityHint`. If you know what these are and when to set them, Voice Over support is going to shine. I keep mentioning this, but again - many times you won't even have to do anything, iOS will get sensible values for you the majority of the time.

That thought, though, really embodies the challenge with accessibility programming: Most don't know when to go further and if they do, they're not confident on which properties to set or the APIs to use. So, let's ensure you've got a solid grasp on these properties and what they represent to assistive technologies.

UIAccessibilityLabel

This one will be used most often. In UIKit controls, it's derived from the title property by default. In SwiftUI, it's set automatically a majority of the time but also has its own dedicated modifier to set one yourself.

Consider the code below:

```
// UIKit
let demoLabel = UILabel()
demoLabel.text = "Welcome to the Accessibility Chapter!"

// SwiftUI
var body: some View {
    Text("Welcome to the Accessibility Chapter!")
}
```

Voice Over will simply read out “Welcome to the Accessibility Chapter!”, and we can see that’s also true of SwiftUI text-based views. In short, this property identifies the element and often without any work from us.

So, that begs the question - when would you manually change this if it’s right most of the time? Consider this:

```
// Xcode -> TenKFig3ViewController.swift
import UIKit

class TicTacToeImageView: UIImageView {
  // Code to set tic tac toe image
}

class TenKFig3ViewController: UIViewController {
  override func viewDidLoad() {
    super.viewDidLoad()

    // Add the custom image view to the view
    let ticTacToeImgView = TicTacToeImageView(frame: view.bounds)
    view.addSubview(ticTacToeImgView)
  }
}
```

If you run this with Voice Over…nothing would happen. Why is that? If you think it’s because its accessibility label isn’t set, that’s true, but it’s also only half of the story. In

this case, Voice Over sees an image and by default, it doesn't report to assistive technologies since its isAccessibilityElement boolean is false by default.

If we wanted to communicate to the user that this is a tic-tac-toe board that's empty, then we'd set both the accessibility label and change the boolean property to true:

```
// Xcode -> TenKFig3ViewController.swift
override init(frame: CGRect) {
  super.init(frame: frame)
  isAccessibilityElement = true
  accessibilityLabel = "Tic-Tac-Toe board, empty."
}
```

This would result in Voice Over reporting "Tic-Tac-Toe board, empty" which may be what you're after when the game starts. Always recall that this property is what Voice Over reads out by default, so as long as it reflects what you intend to - you're good.

UIAccessibilityHint

Used less often than the `UIAccessibilityLabel` property, this one is used to give users an understanding of what will happen when they interact with the control. In short, it helps set expectations.

Consider an interface that streams a broadcast, which has an image view acting as a button with a play icon in it:

```
// UIKit, Xcode -> TenKFig3ViewController.swift
let imageView = UIImageView(image: UIImage(systemName: "play.fill"))
imageView.contentMode = .scaleAspectFit
imageView.frame = CGRect(x: 44, y: 164, width: 44, height: 44);
imageView.tintColor = .blue
view.addSubview(imageView)

// SwiftUI, Xcode -> TenKFig2.swift
var body: some View {
```

```
    Image(systemName: "play.fill")
        .aspectRatio(contentMode: .fit)
        .frame(width: 44, height: 44)
        .foregroundColor(.blue)
}
```

Voice Over will have some problems here, especially since there are no textual inferences that can be made about what this "button" does. In fact, all Voice Over know is that this is an image view, but not what it's for or what it will do if it was triggered. The same would be true even if we used a `UIButton`, or SwiftUI's Button view here.

This is where a hint can help out the user. By adding one (along a sensible accessibilityLabel, such as "Start Stream"), Voice Over users will know that it starts a stream. Since an image view isn't available to assistive APIs by default, we'd also need to opt in:

```
// UIKit, Xcode -> TenKFig3ViewController.swift
let imageView = UIImageView(image: UIImage(systemName: "play.fill"))
imageView.contentMode = .scaleAspectFit
imageView.frame = CGRect(x: 44, y: 164, width: 44, height: 44);
imageView.tintColor = .blue
imageView.accessibilityLabel = "Start Stream"
imageView.accessibilityHint = "Start streaming the broadcast"
imageView.isAccessibilityElement = true
imageView.accessibilityTraits = .button
imageView.isUserInteractionEnabled = true
view.addSubview(imageView)

// SwiftUI, Xcode -> TenKFig3.swift
var body: some View {
    Image(systemName: "play.fill")
        .aspectRatio(contentMode: .fit)
        .frame(width: 44, height: 44)
        .foregroundColor(.blue)
        .accessibility(hidden: false)
        .accessibility(hint: Text("Start streaming the broadcast"))
```

```
        .accessibility(label: Text("Start Stream"))
}
```

Whereas the accessibility label answers the question of what the element is or presents, the hint signifies what will happen if it's used.

UIAccessibilityValue

Now that we know how to identify what elements are and what they will do - there is still one critical piece of information we can convey. What's this element's value?

The role of this property is to let the user know what an element's literal value or meaning is - but this isn't always needed since not all elements have a value. So what does it do? Things like a slider would need one, since the slider would be on a scale of 1 to 10, so the `accessibilityValue` would represent what value the slider currently is set at. Another example, which you'll find below, would be something less obvious like a text field to send a message. The accessibilityLabel might be "Message Text" and its `accessibilityValue` would be the text if any was entered, like below:

```
// UIKit, Xcode -> TenKFig3ViewController.swift
let textField = UITextField(frame: .zero)
textField.placeholder = "Tap to enter a message"
textField.text = "Let's chat accessibility!"
textField.accessibilityValue = textField.text

// SwiftUI, Xcode -> TenKFig3.swift
@State private var chatMessage: String = "Let's chat accessibility!"

var body: some View {
    TextField("Tap to enter a message", text: $chatMessage)
        .accessibility(value: Text(chatMessage))
}
```

Keep in mind, you'd need a more robust approach to keep things in sync for the value. In instances like this, frameworks like Combine or patterns like closures to delegates become appropriate places to update any accessibility properties.

So, given all of this information - why not merge them all together? How come we don't just toss all of that into the accessibilityLabel property? While that might seem easier, keep in mind iOS has incredible heuristics and capabilities to know when to communicate what. Context is key, and iOS is the one who has it so long as we provide the right accessibility information. Further, some of these properties aren't used in many cases (the hint and value). So, don't fight the framework - which is generally a good practice regardless.

Three Key Takeaways

1. `UIAccessibility` plays a pivotal role in accessibility on iOS, and when you implement standard controls much of the heavy lifting is done for you.
2. By understanding the basics, you'll know when the time comes to implement something more custom such as using `UIAccessibilityContainer` or creating your own `UIAccessibilityElement`.
3. By knowing "The Big Three", you'll have taken major strides right away to foster an accessible app.

VoiceOver

VoiceOver is the marquee element of accessibility on iOS. Many users outside of blind folks rely on it for several reasons, such as combating motion sickness. Your accessibility journey usually starts here, so let's see how to make it shine.

At its core, VoiceOver is a screen reading technology that allows you to navigate your device and its apps without necessarily requiring the screen to do so. Users navigate via gestures and taps, which we'll cover below. Your job with VoiceOver is to vend information and ensure your app is navigable.

How it Works

When VoiceOver is enabled, it begins auditing the current view for accessibility elements. These could be either individual elements (i.e. a label) or grouped elements aggregated together (i.e. an accessibility container or a custom element). Within those elements, it'll also look for things to help describe its role and contents:

```
// Xcode -> VOFig1ViewController.swift
let button = UIButton(frame: .zero)
button.setTitle("Hello World", for: .normal)

/*
button.accessibilityTraits would be UIAccessibilityTraitButton
button.accessibilityLabel is set to "Hello World"
*/
```

With this information, VoiceOver understands the element's role and purpose and eventually conveys that to the user when it's focused.

When an element is selected by VoiceOver, it'll have a black rectangle around it[2]. Users of VoiceOver navigate through your app by way of moving this "cursor" throughout your interface. With sensible accessibility programming, you'll allow them to make informed decisions on what steps to take. VoiceOver users should not encounter surprises. Each element they choose to interact with should provide enough context to reinforce their decisions.

---

[2] There are ways to change this though, as we'll discover later using UIAccessibilityPath.

Tips

Navigation with VoiceOver

Learning how to navigate with VoiceOver will speed up your ability to test for it and understand how people are actually using it. To that end, here are the gestures and taps you should get familiar with to be able to use VoiceOver on your own:

One Finger:

1. Tap → Select element
2. Double tap → Activate element (i.e. perform its target action handler or `UIAction`)
3. Slide from bottom + first vibration → Go to the home screen
4. Slide from bottom + second vibration → Open App Switcher
5. Slide from top → first vibration → Open Control Center
6. Slide from top → second vibration → Open Notification Center
7. Swipe right → Navigate to the next accessibility element
8. Swipe left → Navigate to the previous accessibility element

Two Fingers:

1. Tap → Pause or continue speaking
2. Double tap → Start, or stop, the current action
3. Double tap + hold → Edit an item's label
4. Rotation → Activate the Rotor control
5. Triple tap → Opens the Item Chooser, allows for quicker navigation
6. Scrub (a Z motion "escape") → Return to the previous screen or dismiss a modal

7. Swipe up → Speak the entire screen from the top
8. Swipe down → Speak the entire screen from the bottom

Three Fingers:

1. Tap → Get additional information (i.e. whether the text is selected, position, etc.)
2. Double tap → Toggle VoiceOver mute
3. Triple tap → Toggle Screen Curtain
4. Swipe up → Scroll down one page
5. Swipe down → Scroll up one page
6. Swipe right → Scroll left one page
7. Swipe left → Scroll right one page

Four Fingers:

1. Tap at the top of the screen → Select first accessibility element
2. Tap at the bottom of the screen → Select the last accessibility element

Context Changes

While navigating your app, a user might dismiss a modal alert or view. This means the layout changes and the context often shifts. For someone viewing your app, these context changes are apparent and welcome. For VoiceOver users, they aren't always so obvious.

For times when your content dynamically changes, or the interface is altered in a dramatic fashion (i.e. a whole piece of the content goes away or changes significantly) - let VoiceOver know about it by using these notifications:

1. `UIAccessibilityLayoutChangedNotification`: Lets the user know something in the interface has changed and is a cue to reset their current focus or expectations.
2. `UIAccessibilityScreenChangedNotification`: Lets the user know that the screen context has changed.

So when do you use which?

Post layout changes when your interface changes dynamically. Consider filling out a signup form with a switch to say "I am 18+ years old", and doing so shows up another date control to enter in a birthday. Here, we've revealed a control that was not present before, making this a great use case to post this notification:

```
// Xcode -> VOFig1ViewController.swift

// Simply post that the layout has changed
UIAccessibility.post(notification: .layoutChanged, argument: nil)

// Or, post it with more context on what's specifically changed
UIAccessibility.post(notification: .layoutChanged, argument: "Select
birthday toggle now showing.")
```

If there is a new element that would be sensible to focus by default, you pass it as well:

```
UIAccessibility.post(notification: .layoutChanged, argument:
birthdayPicker)
```

Though, keep in mind that focusing on another element should be used sparingly. Making choices for a user, especially in terms of navigation, can be jarring. Only provide this if it's nearly certain that this is a path most users would naturally take given the context.

On the other hand, **post screen change notifications when a new view is presented that acts as the focus for nearly the whole screen**. Consider a browser, which has multiple tabs. In engineering terms, the view controller likely wasn't dismissed - but the container now represents an entirely different context when switching from one tab to another:

```
// With announcement
UIAccessibility.post(notification: .screenChanged,
                     argument: "On tab 1 of 2")

// Focus a particular element
UIAccessibility.post(notification: .screenChanged,
                     argument: tab2)

// If you pass nil, it'll focus the top-left most accessibility
element
UIAccessibility.post(notification: .screenChanged,
                     argument: nil)
```

Just like the previous notification, you can either pass a string to convey context or you can pass an element that should be the focus for the user. If you omit both, the accessibility framework will focus on the first accessibility element it finds.

Accessibility Traits

Accessibility traits exist to answer one fundamental question, "What kind of element is this?" Any iOS app has a number of interface elements, and when VoiceOver knows the primary role of each one it can stitch together a more cohesive experience for users.

Like many other accessibility properties, iOS assigns this automatically for you most of the time. Much like we saw with `UIAccessibilityElement`, you'll mostly leave these alone unless you're creating a custom control or the context or meaning

of an existing one changes. As we saw above, the system will set sensible values for these most of the time, so a `UIButton` would have its trait set to reflect that it's, of course, a button.

In some cases, you'll need your traits to reflect more than one role. That's supported by virtue of this property being setup as a bitmask, meaning you can assign multiple values to the property:

```
// UIKit, Xcode -> VOFig1ViewController.swift
let myTappableLabel = UILabel(frame: .zero)
myTappableLabel.text = "Change Folder"
myTappableLabel.isUserInteractionEnabled = true

let tapGesture = UITapGestureRecognizer(target: self, action:
#selector(testFunc))
myTappableLabel.addGestureRecognizer(tapGesture)

// Indicate this acts as both a header, and could be tapped
myTappableLabel.accessibilityTraits = [.button, .header]

// SwiftUI, Xcode -> VOFig1.swift
var body: some View {
    Text("Change Folder")
        .onTapGesture {
          // Change a folder
        }
        .accessibilityRemoveTraits([.isStaticText])
        .accessibility(addTraits: [.isHeader, .isButton])
}
```

Note that in the case of SwiftUI, you may need to remove any implicit traits that no longer apply. Further, in many cases you will want to simply *add* to the traits that al-

ready exist[3]:

```
myAccessibilityElement.accessibilityTraits.insert(.button)

// Or, add multiple values at once
myAccessibilityElement.accessibilityTraits.formUnion([.button, .adju
stable])
```

Screen Curtain

To test things out with no visual aids, triple tap the screen while VoiceOver is active. This turns the screen off, forcing you to navigate your app using only VoiceOver. There's no better way to test how accessible your app is, especially from a navigation standpoint.

[3] In these examples, we've told the system that we only desire the traits we assigned to - overwriting any previous values that were already there (if any).



Toggle Screen Curtain with a three finger triple tap while VoiceOver is active.

As you use your app with Screen Curtain activated, try to think of a task to carry out in your app. For example, if your app is a grocery shopping list, attempt to add an item to your list. Then, see if you can edit the item as well as you normally would.

Screen Curtain is the defacto barometer to measure how much work you may have ahead of you to create a fully VoiceOver compliant app. Don't get discouraged if you can't get far, but also realize that this is the experience that blind users would otherwise have in your app if you weren't to address the issues you find.

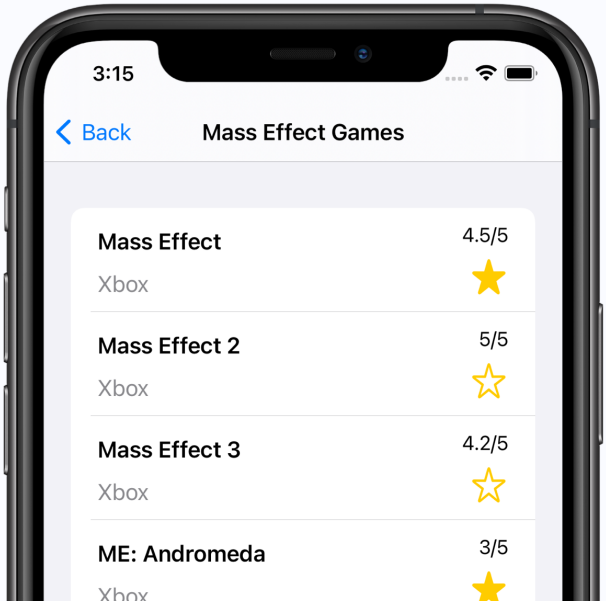
The Big Three

In an earlier chapter, "The 10,000 Foot View: Basics", we covered what I call "The Big Three". For a reminder, those were the accessibility label, value and hint properties or modifiers. These all play a critical role with VoiceOver. Remember that the label describes what the element is, the value conveys what the control's current value is (if

any) and the hint describes what the outcome of using that control would be. If you need more of a refresher, be sure to revisit that chapter.

Flattening Hierarchy

When VoiceOver is active, it can be useful to "flatten" your hierarchy from an accessibility standpoint. This means you aggregate multiple pieces of information together to describe things in a more succinct manner. A great example of this is a table view cell. Consider the following:



By default, VoiceOver will read all of these items out, but there's a few ways this is cumbersome:

1. The user has to flick a few times to get all of the information
2. Context can get easily lost, and the user may forget which value the current element corresponds with. Here, if VoiceOver reads out "not favorited" and

they had to flick a few times to get to that point, they may have already lost the association to the item that was even being favorited. Worse, they'll have to reverse their work to get back to the title to get that information.

So how can flattening hierarchy help us out? By promoting a stronger association between the items. Here's how we can fix it using what we've covered so far:

```
// UIKit, Xcode -> VOFig2ViewController.swift

// First, mark existing elements out of accessibility systems
[titleLbl, ratingLbl, platformLbl, favoriteImgView].forEach {
    $0.isAccessibilityElement = false
}

// When data is set, make one cohesive accessibility label
func setData(from game: VideoGame) {
    titleLbl.text = game.name
    ratingLbl.text = game.rating
    platformLbl.text = game.platform
    favoriteImgView.image = game.isFavorite ? UIImage(systemName:
"star.fill") : UIImage(systemName: "star")

    // Flatten hierarchy
    accessibilityLabel = "\(titleLbl.text!), on \
(platformLbl.text!). Rated \(ratingLbl.text!) - \(game.isFavorite ?
"favorited." : "Not favorited.")"
}

// SwiftUI, Xcode -> VOFig2.swift
VideoGameCell(game: game)
    .accessibilityLabel("\(game.name), on \(game.platform). Rated \
(game.rating) - \(game.isFavorite ? "favorited." : "Not
favorited.")")
```

Here, we've made VoiceOver navigation more intuitive by doing two things:

1. Decreasing the amount of swipes the user has to perform to get things done and
2. We've provided all of context and information needed for the current item in the list

At first, combining elements like this may not seem like an intuitive solution - but that's because we tend to think of things in the context of someone who has their full vision or may not rely on VoiceOver[4]. Consider the alternative, though - imagine if every app you used made you do twice the work for half the benefit. If we don't audit our hierarchy specifically for VoiceOver users and flatten things where we need to, that's the exact situation users will find themselves in.

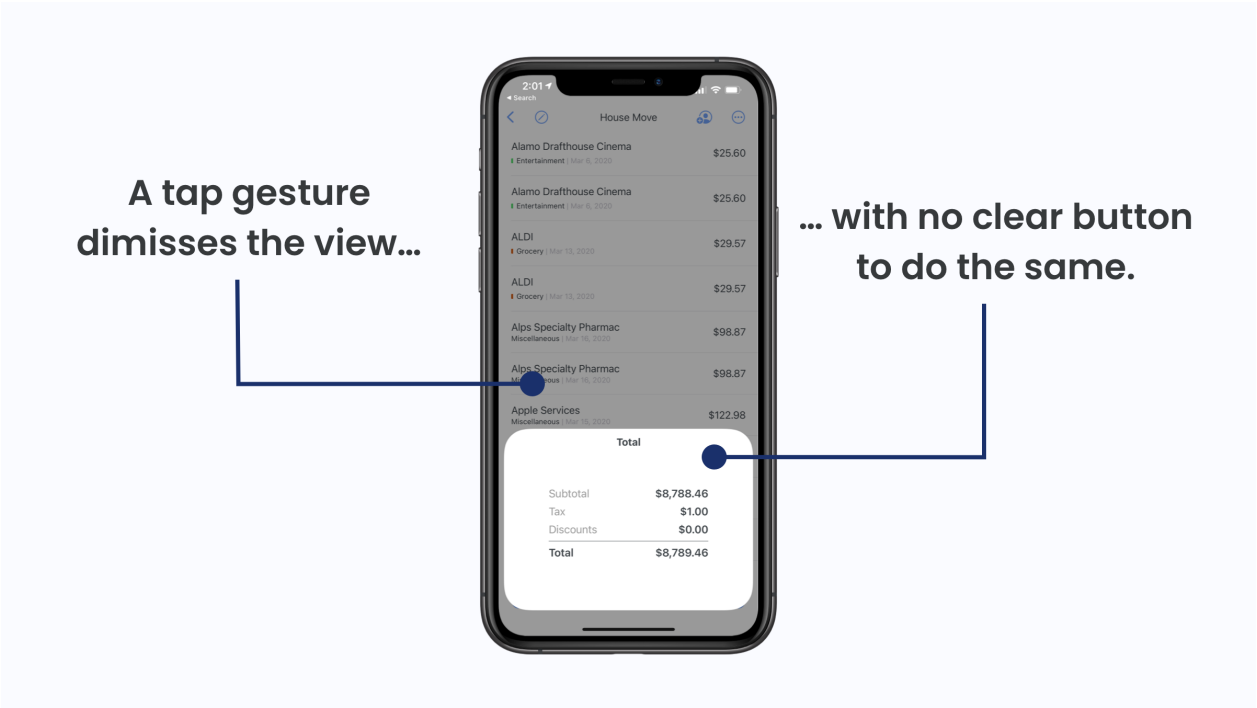
Common Pitfalls

Let's wrap up by covering some things that might get in the way of a solid VoiceOver experience.

Modality

It's common in many apps to have some sort of modal view or bottom sheet that solely relies on a gesture to dismiss. If that's the case, ensure that VoiceOver users have an obvious way to dismiss this kind of view:

[4] As we'll see later, we'd also need to provide a way for the user to toggle a title as their favorite.



From a visual standpoint it may seem obvious how to leave this screen, but those relying on VoiceOver could be stuck here.

I found myself in this situation in my own apps where I had created a custom view controller transition to create such an interface. Immediately, I received reports from VoiceOver users letting me know there was no way to exit this view. Here's how we can fix it:

1. The obvious way - add a button in the interface to dismiss it.
2. Override `accessibilityPerformEscape`, which the two-finger escape gesture will invoke, button to dismiss it:

```
func accessibilityPerformEscape() -> Bool {
   dismiss(animated: true)
   return true
}
```

Implied Modality

The above illustrates a situation that can arise from a custom view controller presentation, though the need to use `accessibilityPerformEscape` is not exclusive to such scenarios. There are many times you may have a simple `UIView` whose presentation mimics a view controller presentation.

In these cases, it'll help VoiceOver see what's going on by setting `accessibilityViewIsModal` to reflect its modality.

Consider the same interface in the image above under the "Modality" section. If I had implemented such an interface using just a `UIView` and *not* as a custom view controller, then it would make sense for me to override this property and set it to true.

This way, iOS wouldn't pick up all of the accessibility elements below the view (in this case, the table view from the presenting view). Instead, the system would just focus on the view presented modally, which is correct since that's the presentation context we're conveying - and now VoiceOver will treat it the same way we intended.

Information Overload

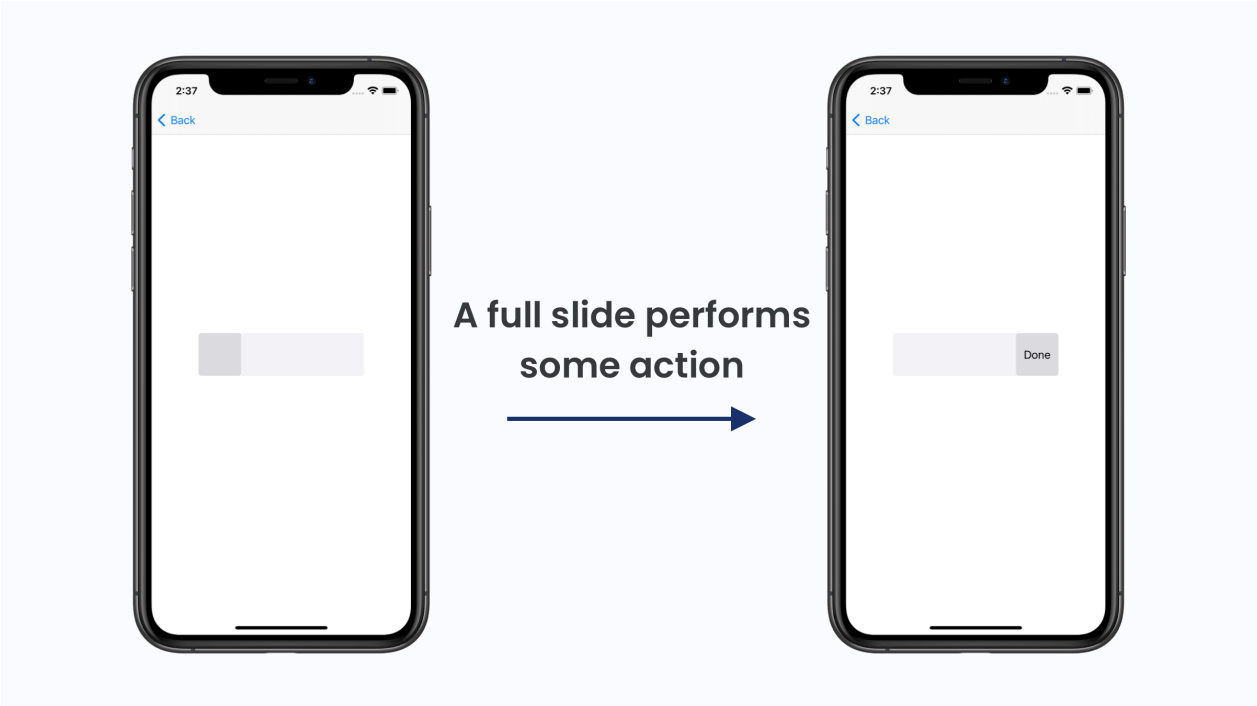
As we saw above when we talked about flattening your hierarchy, if you've got several elements grouped together - consider consolidating that information and reporting that the container is the only accessibility element in the interface. Look for places where you've got a container and three or four elements within it. We want to make VoiceOver give succinct information to get a task done, and swiping through several labels discourages that.

As we'll see in a later in the "Customized Accessibility Content" chapter, you can also give users the choice if they do want that additional information by using the `AXCustomContentProvider` protocol.

Missing out on `accessibilityActivate()`

There are places in our apps where it makes sense to promote a gesture to complete a task. Consider the very first version of iPhoneOS[5] and its now-famous "Slide to Unlock" gesture. If there are similar user experience patterns in your app, VoiceOver users may not be able to complete those gestures.

Here, you'll want to leverage `accessibilityActivate()` - which allows you to perform some logic when a user double taps the element. I've recreated a rudimentary version of a slide to unlock control below. When the user slides the full width, a label pops up saying "Done":



Here, we can ensure that VoiceOver users get the same benefit by performing the same logic when a user has moved the slider the full width within `accessibilityActivate()`:

[5] It wasn't even called iOS at this point!

```
// Xcode -> VOFig3ViewController.swift
class SlideControl: UIView {

    // MARK: Accesiblity Activate
    override func accessibilityActivate() -> Bool {
        fullSlidePerformed()
        return true
    }

    // Abbreviated
    @objc func updateSlide(_ sender:UIPanGestureRecognizer) {
        // If they've moved to the end...
        fullSlidePerformed()
    }

    private func fullSlidePerformed() {
        messageLbl.text = "Done"
    }
}
```

More to the Story

There are a lot of components required to make an incredible VoiceOver experience, but none of them are complicated. Instead, the trick is to simply *know* about them so you can identify when it's time to apply them. There are chapters ahead that cover other critical concepts such as the switch control, custom actions and more - but make no mistake, your accessibility journey on iOS typically starts with VoiceOver.

Three Key Takeaways

1. VoiceOver is usually your first stop when thinking about accessibility on iOS.
2. Ensuring you've addressed "The Big Three" is always the first step.
3. From there, look for issues with navigation or too much information density that would make your app cumbersome to use for VoiceOver users.

The Rotor Control

The rotor control helps make navigation quicker. That's what it does at its very core. Consider the user interface below:



Visual cues play an important role to navigation.

Visually, we get the benefit of context fairly quickly. We can pick out a few main headings, reason about content sections and generally get a feel for how we want to navigate the view. For example, we may not be interested in the top “Next Five Days” section since we intended to revisit the “Last Five Days” section right away.

We can do that because we can see it. And when we can see it, we reason about where we want to go. In this case, we’ve seen that there are two main categories here, and based off of that - we chose to “navigate” to the second one.

The Rotor Control can give VoiceOver users that same affordance. One of its many benefits is that it can tell VoiceOver to only navigate by, or to, certain elements (such as headers). In our example above, that would allow VoiceOver users to navigate with the same efficiency as users who aren’t visually impaired might.



The Rotor Control on iOS.

The Rotor Control has several ways to help with navigation. In fact, its capabilities shift with the context. There are options to change the speaking rate of VoiceOver, move to only misspelled words in text, change input methods and more.

So, how do developers fit into the Rotor Control? Primarily, two ways.

First, we can create our own rotors to hand off to the system's Rotor Control to make custom categorical navigation possible for VoiceOver users. This is what we focus on in this chapter. If you find yourself in a situation where you've got an interface that would make sense to navigate to categorically, and the system's default rotors don't cover it - then you've found a great opportunity to supply your own custom rotor to fill that gap.

Secondly, we can make sure we're using the correct `accessibilityTraits` in our apps to make sure the system provided rotor controls behave as users expect. If we've built a custom header-like element but we haven't indicated to the system that *it*

is a header-like element, then we're essentially taking away functionality from VoiceOver users.

How it Works

Let's kick things off with UIKit. Any `NSObject` has a custom rotors property we can assign to:

```
open var accessibilityCustomRotors:[UIAccessibilityCustomRotor]?
```

When we assign to it, those rotors become available to VoiceOver. Each rotor needs to know a few things:

1. The name or type of the rotor
2. The next element that it should navigate to

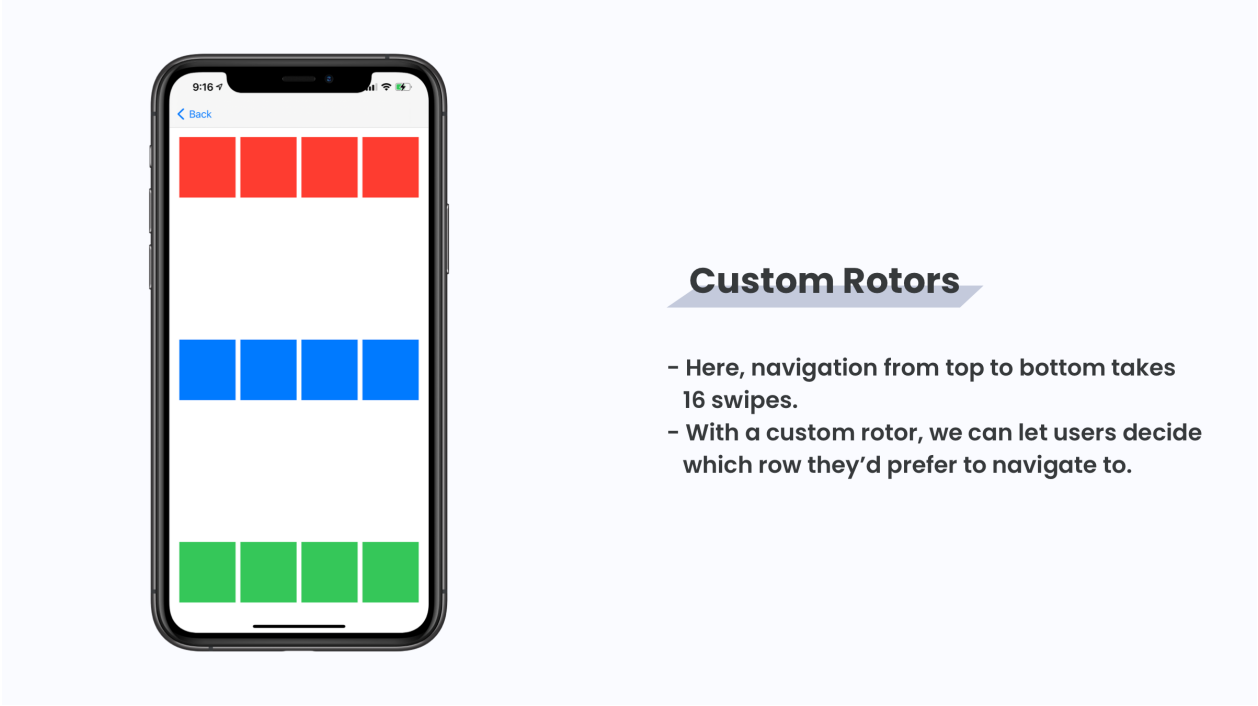
To vend that information, you'll be dealing with three classes and one type aliased closure:

1. `UIAccessibilityCustomRotor`
2. `UIAccessibilityCustomRotorItemResult`
3. `UIAccessibilityCustomRotorSearchPredicate`
4. `public typealias Search = (UIAccessibilityCustomRotorSearchPredicate) -> UIAccessibilityCustomRotorItemResult?`

The **rotor** houses all of the information. The **result** is returned by us to let the active rotor know *which* element to go to next. The **search predicate** exposes what element is focused and which direction the user is navigating (in Rotor Control terms, either up

(`.previous`) or down (`.next`). Finally, the **search closure** gives you the last active predicate while returning the next rotor item.

Let's look at an end-to-end example. Consider a row that has four square views in it, and that a view controller is showing three of these rows. Each row has a different color, and each square within the row has a `.button` accessibility trait and the user can swipe through each one.



Without doing anything, if they want to see the colors in the last row, they'd have to swipe towards it several times to get there.

With a custom rotor, we could simplify things two ways:

1. Provide a custom rotor that toggles each active row in of themselves (i.e. flick up and down to switch rows) or

2. Provide three custom rotors, one for each color, where flicking up and down navigates to each square in the row.

Remember, when a rotor is active - the primary navigation gesture is swiping up or down to *select items within that rotor's category*. Users can, and commonly do, still swipe left and right to navigate through the view's hierarchy still. A rotor is used in tandem with typical VoiceOver navigation.

We'll tackle example one in SwiftUI further down. So for example two, here's what an implementation might look like:

```
// UIKit -> Xcode -> RotorControlFig1ViewController.swift
private func colorRowRotor(forColor color:String, stack:UIStackView)
-> UIAccessibilityCustomRotor {
    return UIAccessibilityCustomRotor(name: color) { searchPredicate
in

        // Ensure we've got a square that's focused
        guard let currentFocusedSquare =
searchPredicate.currentItem.targetElement as? UIView else {
            return nil
        }

        // Find the index of the square in the current stack view
        let indexOfCurrentSquare =
stack.arrangedSubviews.firstIndex(of: currentFocusedSquare)
        let nextIndex: Int

        // Did the user swipe up, or down?
        switch searchPredicate.searchDirection {
        case .next:
            nextIndex = (indexOfCurrentSquare ?? 1) - 1
        case .previous:
            nextIndex = (indexOfCurrentSquare ?? -1) + 1
        @unknown default:
            fatalError()
```

```
            }

            // Ensure selecting the next square won't crash, this
            // Is basically signaling to VoiceOver we've either
            // Reached the end or the beginning of the elements
            guard 0..<stack.arrangedSubviews.count ~= nextIndex else {
                return nil
            }

            // VoiceOver will focus next based off of this result
            let result =
UIAccessibilityCustomRotorItemResult(targetElement:
stack.arrangedSubviews[nextIndex],

targetRange: nil)
            return result
        }
    }
```

Using this custom rotor function to create three rotors, the user could toggle to either "Red", "Blue" or "Green" and the rotor would focus to that particular row with the next swipe up or down. Sequential swipes would then navigate through the squares in the row itself.

If we had implemented a single rotor instead of three (one for each row), we could have one single rotor called something like "Color Rows", where each swipe up or down would take you to the next row and swipes left and right would navigate within them. It's up to you to figure out which ways to implement these custom rotors - but try to create them in such a way that they navigate how users who aren't visually impaired would scan and use your user interface.

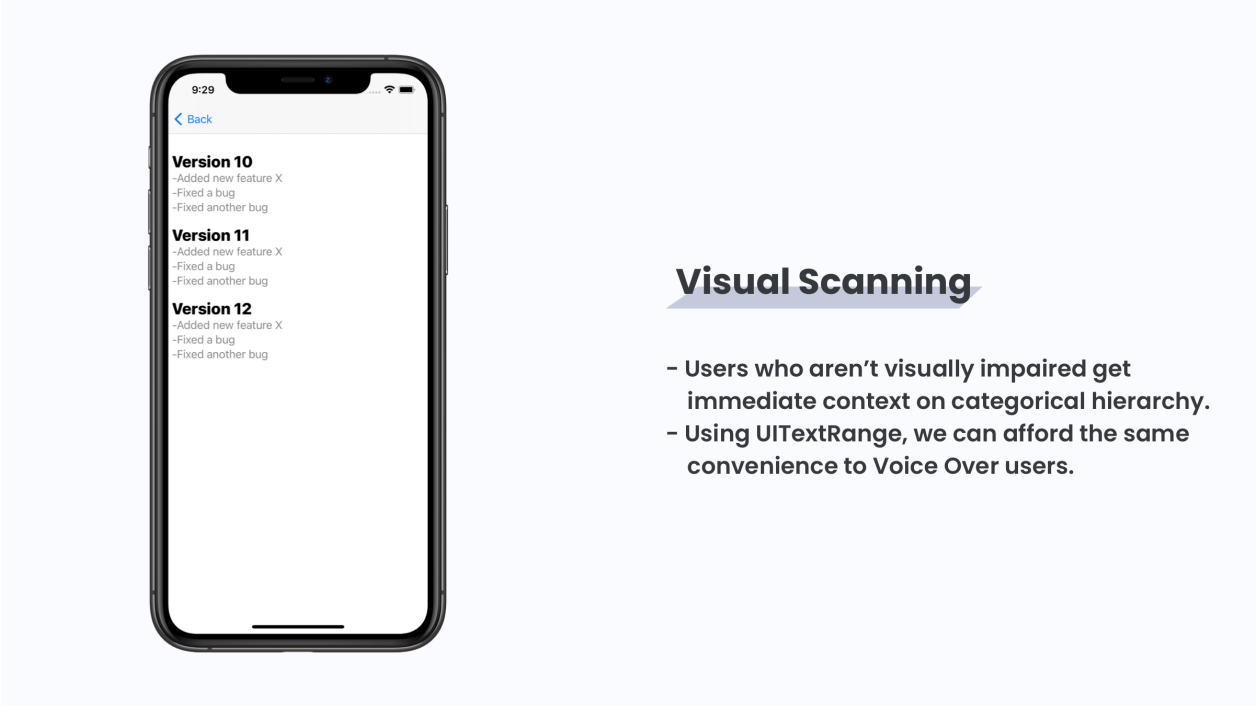
When creating the implementation for a custom rotor, you're essentially responsible for:

1. Tracking the user's search direction

2. Returning the next item that belongs to the rotor based off of that

Another common way VoiceOver users rely on rotors is for long form text. Consider release notes for an app. If you or I were to implement a custom view, regardless of whether or not we used SwiftUI or UIKit, we might have one or more text controls listing everything out.

Visually, we'd likely make each release more distinct from the rest of the text. That means when folks view it, they are likely scanning the interface version by version.



Since VoiceOver users don't get that by default, one way to solve this would be with a custom rotor. Thankfully, the rotor control has a initializer specifically for text ranges:

```
// UIKit -> Xcode -> RotorControlFig2ViewController.swift
private func versionReleaseRotor() -> UIAccessibilityCustomRotor {
    return UIAccessibilityCustomRotor(name: "Releases") { [unowned
self] searchPredicate in
```

```
        guard let currentTextView =
searchPredicate.currentItem.targetElement as? UITextView else {
            return nil
        }

        var nextTextView: UITextView?
        let swipedNext = searchPredicate.searchDirection == .next

        if currentTextView == firstReleaseTextView {
            nextTextView = swipedNext ? secondReleaseTextView : nil
        } else if currentTextView == secondReleaseTextView {
            nextTextView = swipedNext ? thirdReleaseTextView :
firstReleaseTextView
        } else {
            nextTextView = swipedNext ? nil : secondReleaseTextView
        }
        guard let textView =  nextTextView else { return nil }

        let versionTextRange = versionTextPosition(in: textView)
        return UIAccessibilityCustomRotorItemResult(targetElement:
textView,
                                                    targetRange:
versionTextRange)
    }
}
```

Notice that the logic and flow is extremely similar, but now we're dealing with where in *text* the rotor should go along with the text control that contains it instead of in terms of a simple object[6]. This implementation requires a bit more tact that the one above, so if you can reconfigure your view setup to support the previous way of supporting a rotor control - by all means, do so.

---

[6] For example, the previous examples sent nil for the text range parameter. Sending an object is non-negotiable though, it's not a nullable type.

However, I'd invite you not to be intimidated by this approach. Apple supplied it for a reason, and it's built specifically for text-based navigation. It's mostly a matter of translating a range of text into one `UITextPostition` object, so be sure to comb through the sample code to get a feel for it.

SwiftUI

Moving on, SwiftUI has robust support for the Rotor Control as well. It houses the same ideas that its UIKit counterpart has used over the years, but they are expressed primarily via modifiers and the `AccessibilityRotorEntry` struct.

Rotor APIs in SwiftUI rely heavily on a view's *identity*. That is, it matches a rotor, and where to navigate to using that rotor, to a view whose identifier matches the same one the rotor has. Or, it can rely on some arbitrary data's identifiable member. Finally, you can utilize a namespace for more complex view hierarchies using either of the previous approaches to signify identity.

Using Identifiable Entries

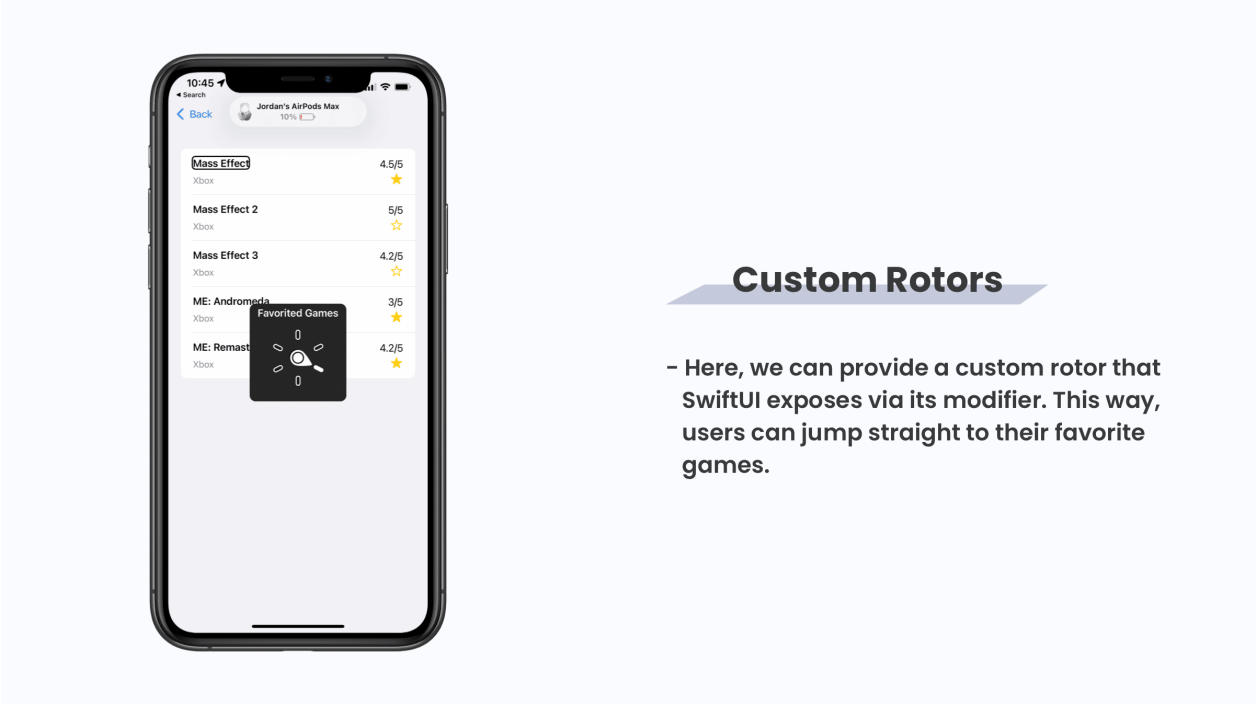
Consider an interface of video games that can be favorited. We could make a custom rotor to navigate *only* to favorited games. Looking at the model for video game, we could leverage the `isFavorited` property to handle this.

```
// SwiftUI -> Xcode -> RotorControlFig1View.swift

extension Array where Element == VideoGame {
    var favorited: [VideoGame] { filter { $0.isFavorite }}
}

struct RotorControlFig1View: View {
    private let games: [VideoGame] = VideoGame.data

    var body: some View {
        List(games){ game in
            VideoGameCell(game: game)
```

```
            }
            .accessibilityRotor("Favorited Games", entries:
games.favorited, entryLabel: \.name)
        }
}
```

The `.accessibilityRotor` modifier here allows us to supply a name for the rotor, signify what data to match against and finally a key path to a member of that identifiable data for using-facing purposes. Currently, that last part only applies for macOS.

With that, the user can now navigate exclusively to their favorited games:



A custom rotor in SwiftUI provides all of the same benefits as its UIKit counterpart.

The Rotor Control APIs for SwiftUI are very flexible. We could achieve the exact same effect by leveraging the `AccessibilityRotorEntry` struct directly. The fact that each item in a `List` must be identifiable already means they all should have a unique, stable identifier we can use for the rotor:

```
// SwiftUI -> Xcode -> RotorControlFig2View.swift

struct RotorControlFig2View: View {
    private let games: [VideoGame] = VideoGame.data

    var body: some View {
        List(games){ game in
            VideoGameCell(game: game)
        }
        .accessibilityRotor("Favorited Games") {
            ForEach(games, id: \.id) { game in
                if game.isFavorite {
                    AccessibilityRotorEntry(game.name, id: game.id)
                }
            }
        }
    }
}
```

This approach has us leveraging a `ViewBuilder` to construct our custom rotor and its contents. In this case, we use the same data the list is using to create a rotor for each favorited game.

Regardless of the approach, you may have already spotted a similarity beyond the need for an identifier. Each of these techniques used a ForEach or a ScrollView with explicit identifiers. What do we do when we don't have that?

For example, what if your view is constructed from a `VStack`. Let's revisit the color rows example we created in UIKit, and create a rotor to navigate to each of them. Here's how the view is constructed:

```
// SwiftUI -> Xcode -> RotorControlFig3View.swift

// Abbreviated.

var body: some View {
    VStack {
```

```
            HStack {
                ColorSquareRow(colors: redColors.colors)
            }
            Spacer()
            HStack {
                ColorSquareRow(colors: blueColors.colors)
            }
            Spacer()
            HStack {
                ColorSquareRow(colors: greenColors.colors)
            }
        }
}
```

Now we've got a problem. If we only had the first two approaches at our disposal, we'd have to change up the view hierarchy to support a custom rotor. Currently, this code isn't built using a `List`, `ForEach` or `ScrollView` with views represented using identifiers.

To address this, we can explicitly state which element should be matched to a custom rotor by using the `.accessibilityRotorEntry` modifier. In fact, these views could be across many other views because they'll use two things to pair rotor control entries to the correct custom rotor:

1. An identifier, as we've used in the previous examples.
2. A namespace, which tells the rotor control *where* that entry is. It could be in the same view or an entirely different view from where the rotor control modifier was used.

Let's put all of those ideas together to recreate the color row custom rotor. Only this time, we'll switch it up from the UIKit example and create just one rotor to navigate to each color row as opposed to creating one for each:

```
// SwiftUI -> Xcode -> RotorControlFig3View.swift

struct RotorControlFig3View: View {
    let redColors: ColorsModel = ColorsModel(color: .red)
    let blueColors: ColorsModel = ColorsModel(color: .blue)
    let greenColors: ColorsModel = ColorsModel(color: .green)

    @Namespace private var customRotorNamespace

    var body: some View {
        VStack {
            HStack {
                ColorSquareRow(colors: redColors.colors)
            }
            .id(redColors.id)
            .accessibilityRotorEntry(id: redColors.id, in:
customRotorNames)
            Spacer()
            HStack {
                ColorSquareRow(colors: blueColors.colors)
            }
            .id(blueColors.id)
            .accessibilityRotorEntry(id: blueColors.id, in:
customRotorNames)
            Spacer()
            HStack {
                ColorSquareRow(colors: greenColors.colors)
            }
            .id(greenColors.id)
            .accessibilityRotorEntry(id: greenColors.id, in:
customRotorNames)
        }
        .accessibilityRotor("Color Rows") {
            let rowID: [UUID] =
[redColors.id,blueColors.id,greenColors.id]
            ForEach(rowID, id: \.self) { model in
                AccessibilityRotorEntry("Color Rows", model, in:
customRotor)
            }
```

```
        }
    }

    private struct ColorSquareRow: View {
        let colors: [ColorRowModel]

        var body: some View {
            ForEach(colors) { colorModel in
                ColorSquare(model: colorModel)
            }
        }
    }

    private struct ColorSquare: View {
        let model: ColorRowModel

        var body: some View {
            Rectangle()
                .fill(model.color)
                .frame(width: 64, height: 64)
                .accessibilityAddTraits(.isButton)
        }
    }
}
```

By giving each `HStack` an associated identifier and a namespace, we can then pair it with a rotor entry modifier explicitly instead of creating one later on in a `ViewBuilder`. Now, SwiftUI can match it up with a rotor control defined from pretty much anywhere. In our case, that's just at the root of the parent view – but it doesn't have to be. There's no need for a `List`, or something that already implicitly demands identity, or even for it to be in the same view definition.

Each approach ties its utility to the amount of scale you require. For most views, the first two approaches should suffice. But for cases where the views don't have an

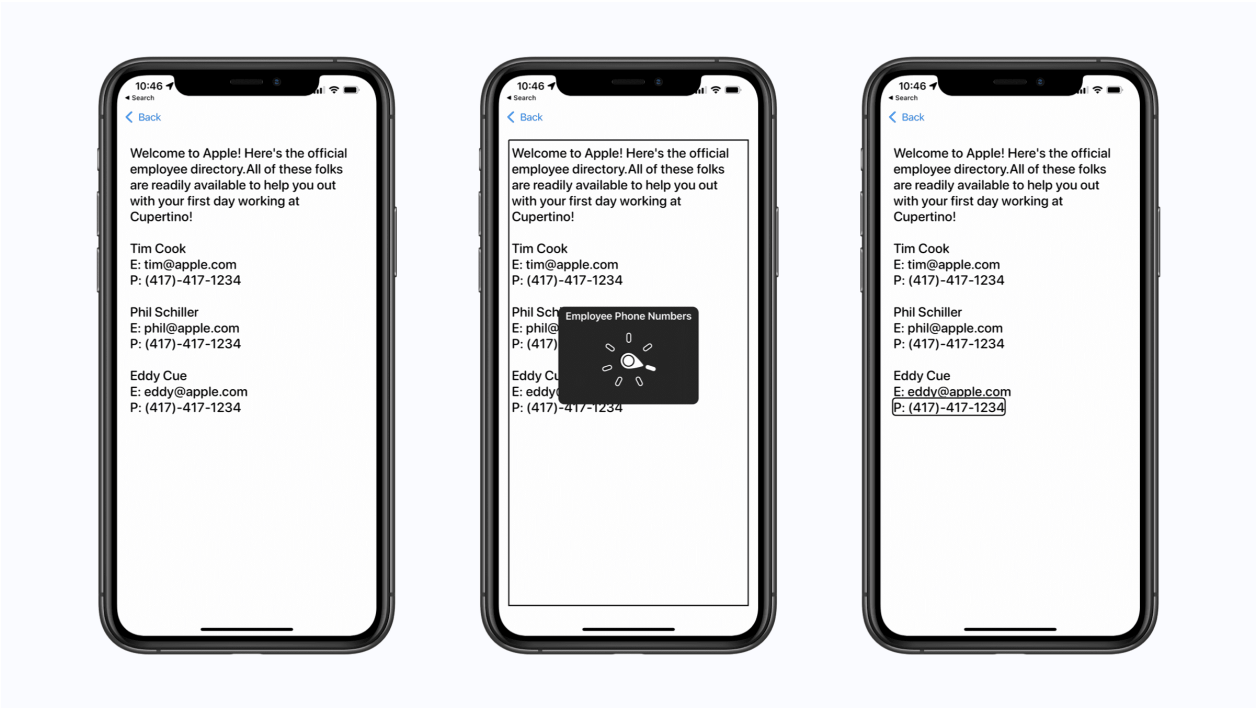
implicit container using identity or they might span multiple views due to SwiftUI's inherent knack for composition – the last approach has you covered.

To recap the amount of utility this API provides:

1. The first example used implicitly made rotor control entries.
2. The second example used explicitly made rotor control entries for *all* views that needed one by matching identifiers to data in a `List`.
3. The third example also used explicitly made rotor control entries, but they were created by each individual view that needed one via a modifier. Then, those were matched together by a rotor control modifier and a namespace.

Text Ranges

Finally, SwiftUI also supports text ranges like UIKit does. In fact, it's a bit simpler to boot as it just needs an array of Swift's `Range<String.Index>` type. Consider an interface which listed employee phone numbers. We could add a rotor to simply move straight to them:



Once the rotor is activated, it can skip to certain parts of the text marked via text ranges.

The code looks like the previous modifiers, except instead of identifiers we use text ranges. In particular, notice how `phoneRanges()` hands off the phone number ranges to the `accessibilityRotor` modifier:

```
// SwiftUI -> Xcode -> RotorControlFig4View.swift

struct RotorControlFig4View: View {
    @State private var welcomeText = """
Welcome to Apple! Here's the official employee directory.All of
these folks are readily available to help you out with your first
day working at Cupertino!

Tim Cook
E: tim@apple.com
P: (417)-417-1234

Phil Schiller
E: phil@apple.com
```

```
P: (417)-417-1234

Eddy Cue
E: eddy@apple.com
P: (417)-417-1234
"""

    var body: some View {
        TextEditor(text: $welcomeText)
            .font(.system(size: 22, weight: .medium,
design: .rounded))
            .padding()
            .accessibilityRotor("Employee Phone Numbers",
textRanges: self.phoneRanges())
    }

    private func phoneRanges() -> [Range<String.Index>] {
        let phoneRegex: String = "[(][0-9]{3}[)][-][0-9]{3}[-][0-9]
{4}"
        guard let regexPattern = try? NSRegularExpression(pattern:
phoneRegex, options: .caseInsensitive) else {
            return []
        }

        let welcomeTextRange = NSRange(welcomeText.startIndex...,
in: welcomeText)
        let phoneNumMatches = regexPattern.matches(in: welcomeText,
options: [], range: welcomeTextRange)
        return phoneNumMatches.compactMap { Range($0.range, in:
welcomeText) }
    }
}
```

I've also included a similar example in this file showing how you might do the same thing, only for emails instead of phone numbers. Feel free to switch out the modifier argument to use that instead of `phoneRanges()` to see how it works.

Tips

Know Where to Assign Rotors

When you assign to the `accessibilityCustomRotors` - make sure you do it in the right place. Any `UIView` can have these custom rotors, so when you assign some to any particular view, those are activated and used when that particular view is in focus.

If you need to, you can also aggregate several rotors into a view's rotor array:

```
self.view.accessibilityCustomRotors =
[view1.accessibilityCustomRotors, view2.accessibilityCustomRotors,
view3.accessibilityCustomRotors].flatMap { $0 }
```

Using the Correct Traits and System Types

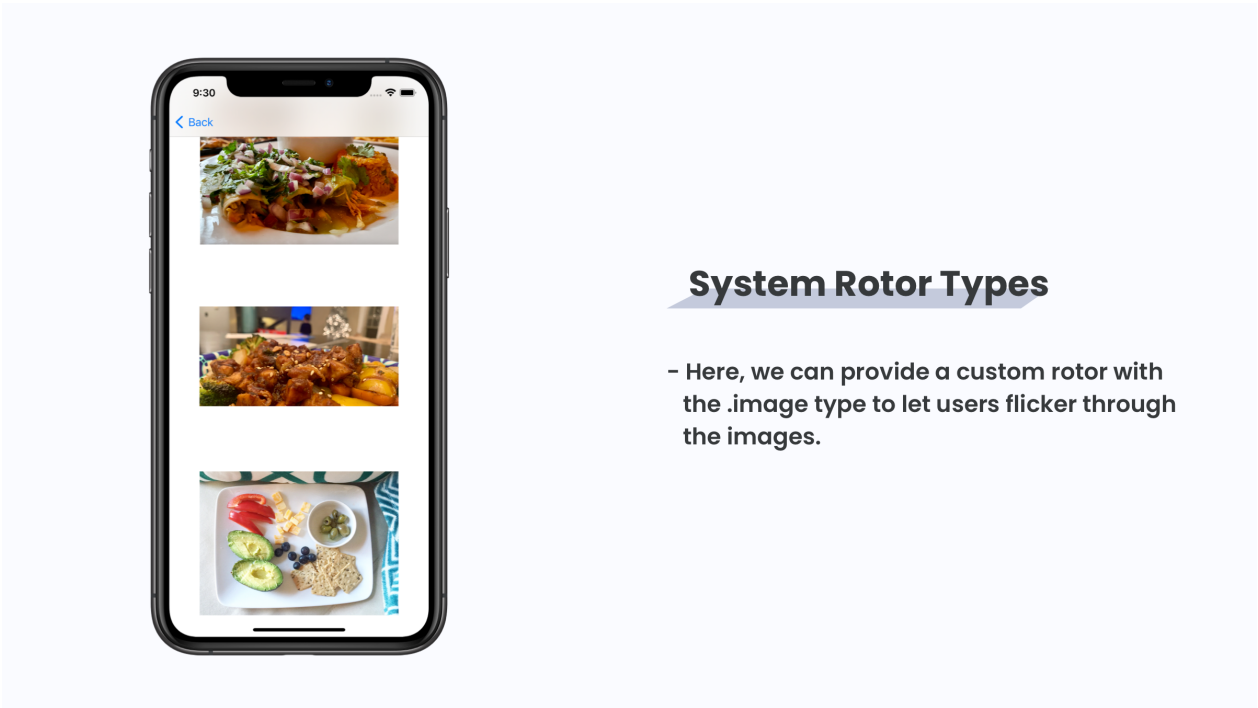
The rotor, in a sense, identifies where to go next categorically. For example, in most interfaces when the rotor is activated you'll likely see "Heading" as an option. As I pointed out above in "How it Works", that means you need heading level elements to have that accessibility trait so it'll be exposed to the rotor control.

Conceptually, thinking in header level elements is quite trivial. But consider all of the other types of rotors available:

```
public enum SystemRotorType : Int {
    case none = 0
    case link = 1
    case visitedLink = 2
    case heading = 3
    case headingLevel1 = 4
    case headingLevel2 = 5
    case headingLevel3 = 6
    case headingLevel4 = 7
    case headingLevel5 = 8
```

```
    case headingLevel6 = 9
    case boldText = 10
    case italicText = 11
    case underlineText = 12
    case misspelledWord = 13
    case image = 14
    case textField = 15
    case table = 16
    case list = 17
    case landmark = 18
}
```

On the other hand, look at that list and try and see if there's any elements you need to supply to the system that might would make sense to show but aren't exposed as a rotor by default. Consider the interface below:



By default, there is no "Image" rotor that's vended by the system, but using the initializer for custom rotor to take in a `SystemRotorType`, we can create one:

```
// UIKit -> Xcode -> RotorControlFig3ViewController.swift

private func imageRotor() -> UIAccessibilityCustomRotor {
    return UIAccessibilityCustomRotor(systemType: .image) { [unowned
self] predicate in
        guard let currentImage = predicate.currentItem.targetElement
as? UIImageView else { return nil }
        let nextIndex: Int
        let currentIndex =
self.stackView.arrangedSubviews.firstIndex(of: currentImage)

        switch predicate.searchDirection {
        case .next:
            nextIndex = (currentIndex ?? 1) - 1
        case .previous:
            nextIndex = (currentIndex ?? -1) + 1
        @unknown default:
            fatalError()
        }

        guard 0..<self.stackView.arrangedSubviews.count ~= nextIndex
else {
            return nil
        }

        return UIAccessibilityCustomRotorItemResult(targetElement:
self.stackView.arrangedSubviews[nextIndex],
                                                    targetRange:
nil)
    }
}
```

And now, activating the rotor control will show an “Images” option that will cycle through just the images within the interface. Notice that the initializer for the custom rotor takes in the `SystemRotorType` instead of a string, and we passed in `.image`.

Returning Results

Since we leverage `UIAccessibilityCustomRotorItemResult` to return the next item to select a rotor, it helps to know all of the ways you can package them up to the system. There are really only two simple responsibilities to remember:

1. You'll *always* return an item, or put differently - an object that the accessibility engine can select. If your logic dictates that you don't have one, then you'd return nil from the rotor and not deliver a `UIAccessibilityCustomRotorItemResult` instance. Recall that the block you use to build a custom rotor asks you to return a nullable instance of that class - so indicating that you don't have one is fine and in many cases the right call. It indicates to VoiceOver users that they've reached some sort of beginning or end.
2. Once you've got an object, you can also return a text range if you're dealing with text.

That's really all VoiceOver needs from your custom rotor to navigate. On the other hand, you'll also receive one of these objects from the search predicate (more on that directly below) when constructing custom rotors. This proves useful as you'll be able to inspect the last focused item or text range to help you vend the next accessible item to that the rotor should navigate to.

Search Predicates

Leveraging the search predicate appears intimidating at first, but I've found it helps to rename it in your head to something like "Previous Rotor Item" since that's what it often represents. Hearing the word predicate may have you draw comparisons to

`NSPredicate` which isn't accurate in this case. This is simply an object with some useful information to help you decide what to do next.

You'll typically use two critical pieces of information from the predicate:

1. The last item that was focused by looking at `predicate.currentItem.targetElement`
2. The direction the user swiped to reason if they want the next or previous item:

```
switch predicate.searchDirection {
case .next:
    // User swiped for next item
case .previous:
    // User swiped for previous item
@unknown default:
    fatalError()
}
```

Custom Attributed String Keys

Since rotor results can be used in tandem with one or more text views you'll be dealing with ranges of matched text quite often. Working with a text range is tricky enough, but you can make your life a bit easier in those situations by extending the attributed string API's key type:

```
extension NSAttributedString.Key {
    static let versionHeader =
NSAttributedString.Key.init("versionHeader")
}
```

Why do this? You can tack that key into your attributed text to later find its range in a trivial fashion when you're creating custom rotors dealing with text:

```
// Note the Version header attribute added last
let attributes: [NSAttributedString.Key:Any] = [.font:
UIFont.systemFont(ofSize: 24, weight: .heavy),
    .foregroundColor: UIColor.label,
    .versionHeader: NSNumber(booleanLiteral: true)]

attributedText.addAttributes(attributes, range:
                             rangeOfVersion(in: textview))
```

Then, when you're looking for that text to translate into a `UITextRange` instance, you can look for the specific attribute without having to match raw text instead:

```
// Search the text view by our custom attribute
textView.attributedText.enumerateAttribute(.versionHeader, in:
NSMakeRange(0,
textView.text.count), options: []) { valueAttribute, matchedRange,
stop in
    guard valueAttribute != nil else { return }
    // Use matchedRange to get a UITextPosition from the text view
    // Then stop iterating
    stop.pointee = true
}
```

Avoiding Dead Rotors

If you find yourself making a custom rotor, assigning to an object's custom rotor property and it *doesn't* show up it's likely because the item you're returning isn't an accessible item by default.

If this happens, be sure to check that the object has `isAccessibilityItem` set as `true` and that the accessibility traits it has lend itself to navigational purposes. For

example, `.staticText` isn't a navigational item so it wouldn't do anything for a rotor. Always remember - the rotor is there to make navigation snappy. As such, it stands to reason that the items we vend to it help accomplish that goal.

SwiftUI Views and Accessibility Container

Rotor controls look for the container accessibility trait. Some views in SwiftUI are inherently using that trait, such as `LazyVStack` or a `List`. If you trying to opt a view into the rotor that doesn't have that trait, you might need to use the `accessibilityElement(children:)` modifier to allow the rotor control to pick it up:

```
MyCustomView()
.accessibilityElement(children: .combine)
```

SwiftUI and Scrolling

If you're using a `ScrollViewReader` along with a custom rotor, the content you need to show could be offscreen at the moment the rotor is used. If that's the case, you can scroll to it using the Rotor Control APIs.

In our example where we had a custom rotor used to scroll to different color rows – we might add in an explicit scroll in case the user had a large list of colors and some of them could be offscreen. In this adapted example, look at the `AccessibilityRotorEntry` initializer which now uses a trailing closure to enable a scroll:

```
var body: some View {
    ScrollViewReader { reader in
        VStack {
            HStack {
                ColorSquareRow(colors: redColors.colors)
            }
            .id(redColors.id)
```

```
            .accessibilityRotorEntry(id: redColors.id, in:
customRotorNamespace)
            Spacer()
            HStack {
                ColorSquareRow(colors: blueColors.colors)
            }
            .id(blueColors.id)
            .accessibilityRotorEntry(id: blueColors.id, in:
customRotorNamespace)
            Spacer()
            HStack {
                ColorSquareRow(colors: greenColors.colors)
            }
            .id(greenColors.id)
            .accessibilityRotorEntry(id: greenColors.id, in:
customRotorNamespace)
        }
        .accessibilityRotor("Color Rows") {
            let rowID: [UUID] =
[redColors.id,blueColors.id,greenColors.id]
            ForEach(rowID, id: \.self) { model in
                // Ensure we scroll to the row
                AccessibilityRotorEntry("Color Rows", model, in:
customRotorNamespace) {
                    reader.scrollTo(model)
                }
            }
        }
    }
}
```

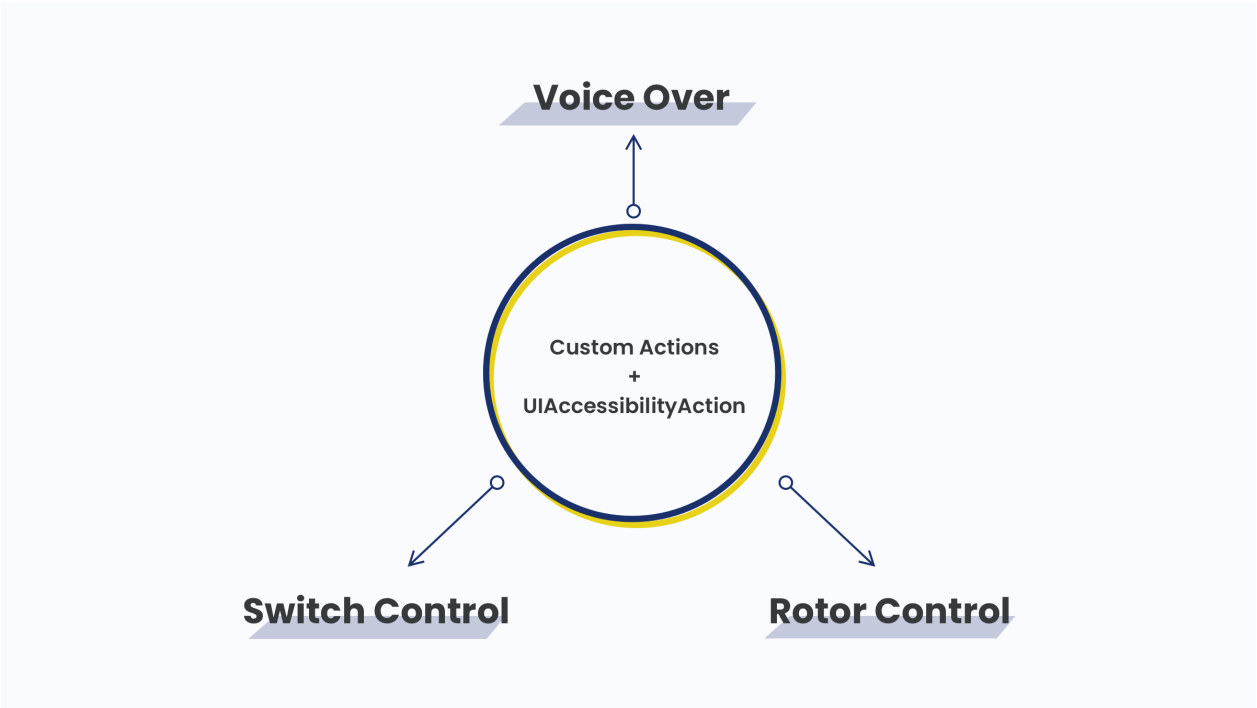
Three Key Takeaways

1. The Rotor Control helps VoiceOver users navigate their device efficiently.
2. We can extend the system rotor controls and provide our own.
3. Be sure to use the correct accessibility traits to ensure your existing interface works great with the system-provided rotors.

Custom and System Actions

Custom and system actions help users do common things quicker. This is key to the iOS ecosystem since Apple has long prided itself in its efficiency when it comes to interacting with software. Think about how cumbersome Messages would be to use if we couldn't swipe left or right to delete conversations or pin one.

When it comes to accessibility considerations, some of these actions are difficult for users to perform due to various disabilities. With custom and system actions, we get to expose those same affordances to everyone. Even better, when you provide custom actions the accessibility framework can use them in more than one place. Custom actions could show up in a rotor control, Voice Control, while VoiceOver is announcing content and even in the Switch Control:



Custom actions show up in several places in iOS.

There are also several common actions users take with VoiceOver, and we'll refer to these as system actions. From an API standpoint, they are using exposed overrides as opposed to dedicated objects that you must create. The `UIAccessibilityAction` informal protocol houses all of the actions developers can use. We saw an example of this earlier in the VoiceOver chapter by leveraging `accessibilityActivate`.

Currently, there are six actions we can implement:

1. `accessibilityActivate() -> Bool`: Activates the accessibility element and returns a boolean indicating if the action was successful or not.
2. `accessibilityIncrement()`: Signals that the accessibility element should increment its value.
3. `accessibilityDecrement()`: Signals that the accessibility element should decrement its value.

4. `accessibilityScroll(UIAccessibilitySCrollDirection) -> Bool`: Scroll the accessibility element according to the passed in direction and returns a boolean indicating if the action was successful or not.
5. `accessibilityPerformEscape() -> Bool`: Dismisses the current modal view and returns a boolean indicating if the action was successful or not.
6. `accessibilityPerformMagicTap() -> Bool` : Performs a context-specific action. This is reserved for the most critical action the user would take in the interface, and it shouldn't always be used. For example, for the phone app on iOS magic tap either answers or ends the call.

When you implement these, the accessibility engine will take them into account when their particular actions are performed. Or, they give the system the correct context to begin with, such as knowing whether or not a control is scrollable, and in what should happen in a control like a picker when the value is either increased or decreased.

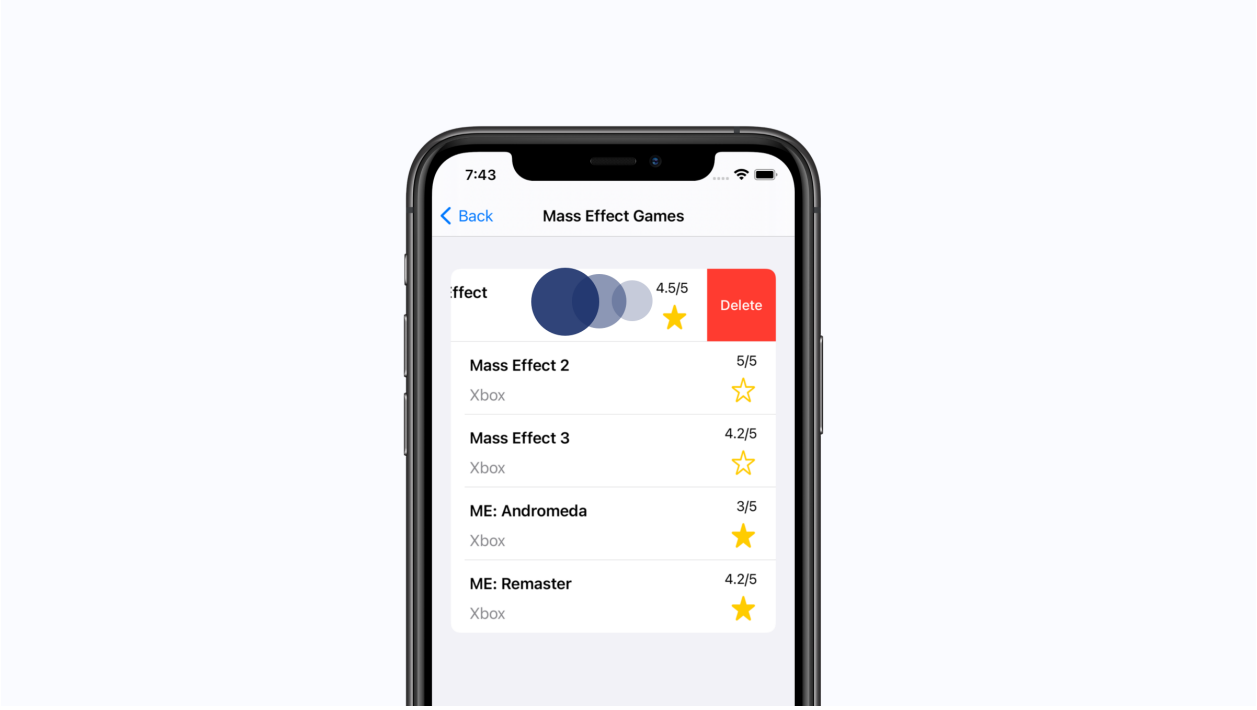
To recap - we'd use custom actions to expose things that are specific to our controls and user interface flows. We'd use the `UIAccessibilityAction` informal protocol for things that that fit within the expressed overrides (increment, decrement, scrolling, etc.). In many cases, you'll find yourself using both within the same interface.

How it Works

To create a custom action, you'll leverage the `UIAccessibilityCustomAction` class. Each instance represents an individual action you'd like to expose to the accessibility system. You do so by assigning them the accessibility container's array of custom actions:

```
var accessibilityCustomActions: [UIAccessibilityCustomAction]? { get
set }
```

Consider the interface below, where swiping from right to left shows a delete option:



By default, if we're using iOS' APIs[7] we'll get these actions reported for us for free when VoiceOver is active. To try it out, you could either focus the cell and VoiceOver would say "Actions Available" and then you'd swipe up or down to toggle through them. Also, there's a rotor just for this listed under "Actions".

However, what if we *didn't* have swipe actions and instead relied upon the little star in the interface to toggle a favorite item[8]? Situations like this are perfect for cus-

---

7 Here, that'd be the UITableViewDelegate methods for return either a UISwipeConfiguration object for either trailing or leading swipes.

8 There's also a lesson here - rely on UIKit and SwiftUI as much as possible. When it comes to accessibility, there's just too much you get for free that makes it hard to pass up without a good reason. Further, it means you'll have to implement these things yourself. That's not a bad thing, in some cases a custom interface is certainly warranted, but it does mean more development cost.

tom actions. Here's how we could expose those actions using `UIAccessbilityCustomAction`:

```
// Xcode -> CustomAndSystemActionsFig1ViewController.swift
extension FlattenCell {
    func addCustomAction() {
        let favoriteAction = UIAccessibilityCustomAction(name:
"Toggle Favorite") { ac in
            // dataModel.toggleFavoriteStatus()
            return true
        }

        accessibilityCustomActions = [favoriteAction]
    }
}

// Then later when creating the cell
lazy var datasource: TableDataSource = {
    let datasource = TableDataSource(tableView: tableView,
cellProvider: { (tableView, indexPath,
model) -> UITableViewCell? in
        let cell = tableView.dequeueReusableCell(withIdentifier:
FlattenCell.cellID, for: indexPath)
as! FlattenCell
        cell.setData(from: model)
        // Add the custom action
        cell.addCustomAction()
        return cell
    })

    return datasource
}()
```

Much like our discussion over rotor controls, you'll notice the API for custom actions is quite similar. Our responsibilities include:

1. Identifying what the action does with the `name` parameter.

2. Firing off our logic when it's used.
3. And returning a `boolean` value to indicate whether or not it was successful.

In SwiftUI, the flow is much the same:

```
// Xcode -> CustomAndSystemActionsFig1View.swift
VideoGameCell(game: game)
    .accessibilityLabel( "\(game.name), on
    \(game.platform). Rated \(game.rating) -
    \(game.isFavorite ? "favorited." : "Not
favorited.")")
    .accessibilityAction(named: "Toggle
Favorite") {
        // dataModel.toggleFavoriteStatus()
    }
```

However, its API more closely mirrors the responsibilities of the `UIAccessiblityAction` informal protocol than it does custom actions. It also doesn't return a boolean from its closure.

Speaking of system actions, in SwiftUI the line between the custom and system actions are more blurred than they are in UIKit. In UIKit, the two are distinct in their implementations with one requiring an object you create and vend to the system while another (the system actions) manifest themselves as overrides.

For example, in SwiftUI, to add a magic tap action you use a similar modifier as above:

```
// Xcode -> CustomAndSystemActionsFig2View.swift
var body: some View {
    Text("Magic Tap installed.\nDouble tap with two
fingers to\nperform magic tap action.")
        .background(bgColorIsBlue ? Color.blue :
Color.green)
```

```
        .accessibilityAction(.magicTap, {
            self.bgColorIsBlue.toggle()
        })
}
```

But in UIKit, Magic Tap is tapped into via an override:

```
// Xcode -> CustomAndSystemActionsFig2ViewController.swift
class CustomAndSystemActionsFig2ViewController:
BaseSampleViewController {

    private var bgColorIsBlue: Bool = true
    private let textLabel: UILabel = {
        let lbl = UILabel(frame: .zero)
        lbl.text = "Magic Tap installed.\nDouble tap with two
fingers to\nperform magic tap action."
        lbl.backgroundColor = UIColor.blue
        lbl.numberOfLines = 0
        return lbl
    }()

    override func viewDidLoad() {
        super.viewDidLoad()
        view.addSubview(textLabel)

        textLabel.translatesAutoresizingMaskIntoConstraints = false
        textLabel.widthAnchor.constraint(equalToConstant:
200).isActive = true
        textLabel.centerYAnchor.constraint(equalTo:
view.centerYAnchor).isActive = true
        textLabel.centerXAnchor.constraint(equalTo:
view.centerXAnchor).isActive = true
    }

    private func toggleBackgroundColorForText() {
        bgColorIsBlue.toggle()
        let newColor = bgColorIsBlue ? UIColor.blue : UIColor.green
        textLabel.backgroundColor = newColor
    }
```

```
    override func accessibilityPerformMagicTap() -> Bool {
        toggleBackgroundColorForText()
        return true
    }
}
```

Tips

Provide Imagery

Even though you may be quick to associate custom actions with VoiceOver users, that's not always the case. Some assistive technologies such as the Switch Control can benefit from custom actions providing an image. When you do, they can show up in the switch control.

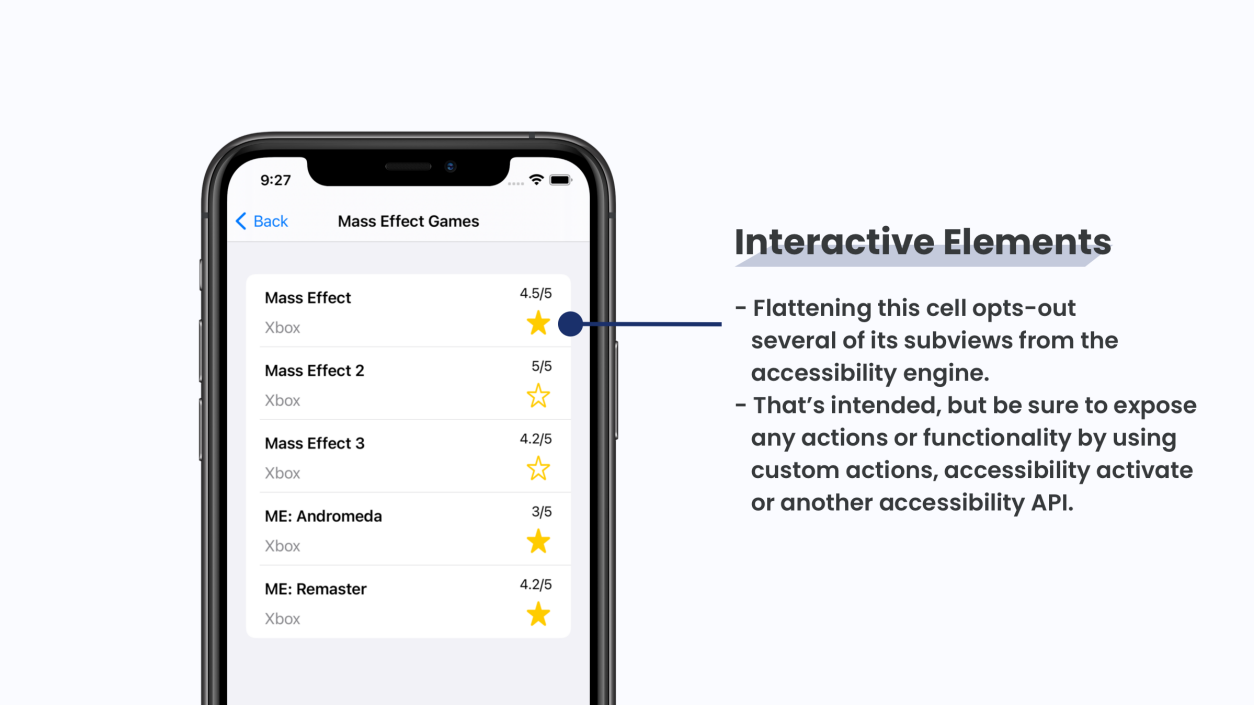
Further, you can even leverage SF Symbols for the task:

```
private func customActionWithImage() {
    let trashImg = UIImage(systemName: "trash")
    let actionDelete = UIAccessibilityCustomAction(name: "Delete",
image:
trashImg) { action in
        // Delete
        return true
    }
}
```

Flattening Hierarchy: Round Two

In the VoiceOver chapter, we talked about this concept of "flattening" view hierarchy to make navigation easier for VoiceOver users. It's often the right call, but be sure to

expose functionality that might've been housed within your user interface as custom actions:



Audit a flattened hierarchy for functionality you may need to expose to assisstive technologies.

When we opt-out objects as an accessibility element for flattening purposes, we gain simpler navigation but lose important functionality if we don't remember to use custom actions (or any other applicable accessibility API) to expose them.

Audit Gesture Recognizers

If there are some views, or user experiences, that rely solely on gestures to accomplish a task, you're almost guaranteed to need to do some work for assistive technologies. If you find yourself in a scenario with something like we saw in the VoiceOver chapter for our Slider View…

```
// Slide logic
```

```
let slideGesture = UIPanGestureRecognizer(target: self, action:
#selector(updateSlide(_:)))
sliderView.addGestureRecognizer(slideGesture)
```

…be sure to open things up to the accessibility engine. In this case, we've got some functionality that relies solely on a gesture recognizer for interaction. You've got several options here, but some great ways to address this for accessibility considerations could be by using the system actions.

In this particular scenario, `accessibilityActivate` is a great choice as it will trigger the element with a two-finger tap:

```
override func accessibilityActivate() -> Bool {
    fullSlidePerformed()
    return true
}
```

Or, possibly `accessibilityPerformMagicTap` is the better choice if the action is critical within the current context as a whole.

D.R.Y.

We can debate the merits of code architecture until the end of time, but I'd be remiss if I didn't point out when it comes to accessibility code it typically pays to follow D.R.Y.[9] principles. If we read back through this chapter, you'll see that many of these accessibility APIs when using UIKit follow the same pattern:

1. Do some logic and
2. Return a boolean to indicate the outcome.

[9] If you're unfamiliar with the term, this acronym stands for "Don't Repeat Yourself", promoting the idea that you shouldn't duplicate logic in your codebase.

Though, many of your flows that you've originally authored might not care about any success or failure states by nature. If the logic you depend on doesn't involve a network request then maybe there is no concept of failing or succeeding, the function simply does something and that's that:

```
private func toggleFavorite() {
    thingToFavorite.isFavorited.toggle()
}
```

However, when we go to hook this up with something like magic tap - we're now in a situation where a boolean state *does* matter because accessibility services need to indicate whether something "worked" or not even if the concept didn't previously apply to the function:

```
override func accessibilityPerformMagicTap() -> Bool {
    toggleFavorite()
    return true
}
```

Since we're likely to find ourselves in this situation, it stands to reason that we don't want to duplicate the logic of `func toggleFavorite()` but we also don't need to return a boolean all the time. The one thing *not* to do here is this:

```
override func accessibilityPerformMagicTap() -> Bool {
    // Rewritten logic from the original method
    thingToFavorite.isFavorited.toggle()
    return true
}
```

Duplicating the logic for situations like this invites future bugs. In between your meetings, sprint planning, customer support requests, coding tasks for the day and whatev-

er else our 9-5 requires of us - we'll forget to update one or the other when the time comes to change how the favoriting logic should work.

With that in mind, whatever your solution is, try to opt towards it leading to you *not* rewriting the same logic twice. Be sure to survey your architecture to decide what the best way forward might be, but writing the same thing twice rarely pays dividends.

When to use Magic Tap

At first glance, it might be ambiguous to know when to use `accessibilityActivate()` versus `accessibilityMagicTap` - but the answer is, thankfully, easy to surmise.

The magic tap gesture should be used for the most critical, core pieces of your user experience. If you're making a messaging app, it sends a message. If it's a VoIP app, it answers and hangs up calls. Podcast players would likely play and pause playback.

Another thing to think about is how the API itself works. VoiceOver users just have to two-finger tap *anywhere* to fire off magic tap - which, in of itself, should tell you all you need to know: This action is a holistic representation of the entire view the user is on, and not as concerned or associated with one single element itself.

Three Key Takeaways

1. Custom and system actions level the playing field for accessibility users.
2. We can create custom actions to expose functionality that may be difficult for accessibility users to otherwise access.
3. Similarly, we can ensure common actions such as scrolling a page or dismissing a view, is easy to do for everyone by utilizing `UIAccessibility-Action`.

The Switch Control

The Switch Control gives users with limited mobility an elegant way to use their iOS devices. The technology works by responding to a user's interactions with switch hardware hooked up to their iPhone or iPad, and in turn translates those actions to specific interactions displayed on screen. They could change the volume, perform scrolling and several other actions.

These switches are often mounted onto a chair or wheelchair and could be used by tapping on it, breathing through a special draw and more. When the focused element is activated, a modal view opens which gives them actions to perform on it:



The Switch Control running on iOS.

The Switch Control is often used with an auto scanning feature, which means the focused element changes automatically after a set interval (i.e. stay on an element for one second, then go to the next one and repeat). Starting top-to-bottom, and then left-to-right - the Switch Control travels through all of the accessible elements. When the element the user wants to interact with is chosen, they activate it via the switch control.

It can also be used in a manual scanning fashion. This input method requires at least two switches. The Switch Control focus changes by being manually triggered.

Finally, there is also single switch step scanning. The users triggers a switch to move the focus from item to item. If they don't take an action after a certain interval, then the focused item gets activated.

Due to the nature of any of these input methods and a reliance on timing the cursor, it's also important to make sure your logic and interface is tolerant of errors when

the Switch Control is on, because they may have accidentally tapped the wrong element:

```
if UIAccessibility.isSwitchControlRunning {
    // Consider any interface tweaks to make things
    // More error tolerant, such as double opt-in
    // Actions (i.e. action -> confirm action)
}

// Or, change things dynamically from a notification
let switchToggleNote =
UIAccessibility.switchControlStatusDidChangeNotification
NotificationCenter.default
    .publisher(for: switchToggleNote).sink { note in
    // Query value and respond
}.store(in: &subs)
```

For example, I typically recommend *always* confirming a destructive action in your interface but this is doubly important when the Switch Control is on. Why? Because of that error tolerance we're talking about. A user might accidentally choose "Delete" if it's next to a "Save" button. If we can give them another prompt to confirm that action, we reduce the risk that they might carry out a consequential action without intending to.

One of the best parts of Switch Control is that it will work very well if you're already putting the work in for a magnificent VoiceOver experience. Things like custom actions, system actions, sensible accessibility labels, values and hints all work together to make the Switch Control not only a viable way to use an app - but *the* way to use mobile software, on any platform, for folks with limited mobility.

How it Works

From an API standpoint, there isn't anything "new" to learn here. In fact, as far I've found, there is only one API that was specifically created for Switch Control[10] There is no "UIAccessibilitySwitchControl(X or Y)" to demonstrate. As I mentioned above, it's more about ensuring you've got a strong VoiceOver experience, which will translate to a strong switch control experience.

With that said, there can be implementation details to consider. Let's take a look at the "Tips" section to see what we can do to make sure our apps handle the Switch Control great.

Tips

`UIAccessibilityNavigationStyle`

This property only is considered when the Switch Control API is running. There are three options:

1. `automatic`: The default setting. In most cases, the accessibility system will traverse your interface as it normally would: Top-to-bottom, and left-to-right.
2. `separate`: This indicates that the element should have any elements it contains navigated as separate entities. So, if a view with several subviews had its property set like so:

   ```
   view.accessibilityNavigationStyle = .separate
   ```

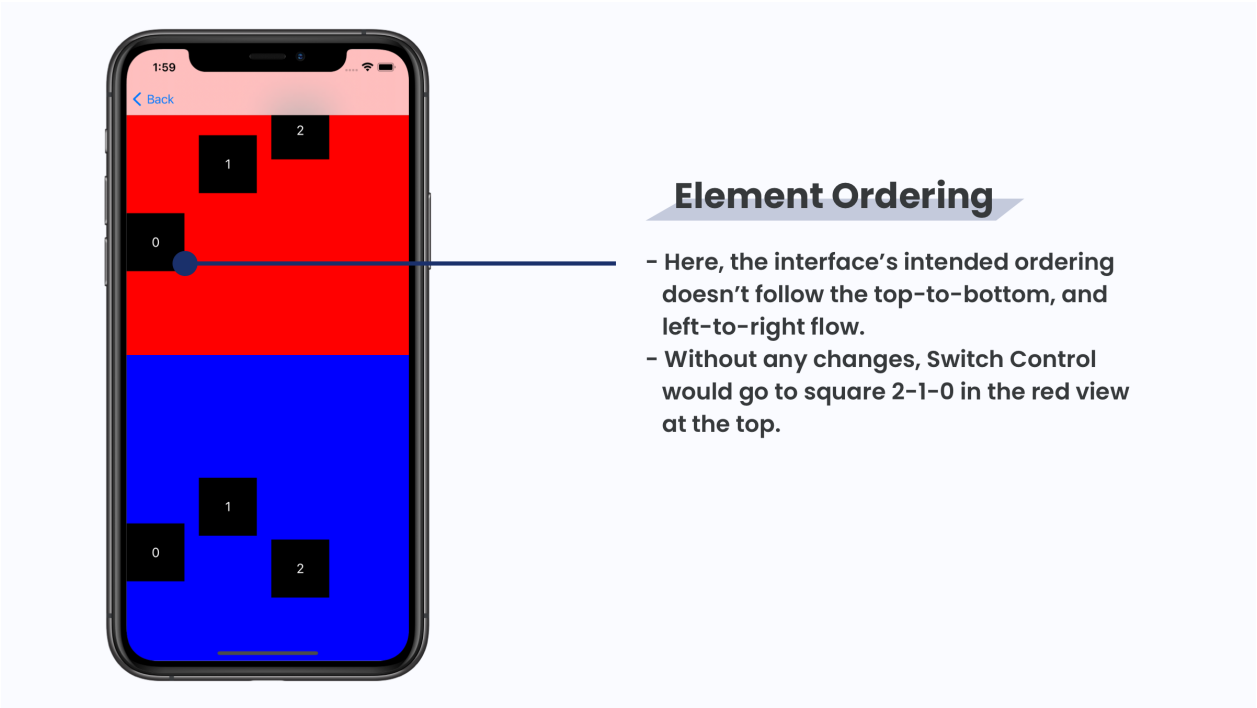
[10] And that's UIAccessibilityNavigationStyle, covered below.

It's indicating to the Switch Control API that each view should be navigable on their own.

3. `combined`: If this is chosen, the containing element is indicating that no matter how many subviews or accessible elements it has, this element should be treated as one entity when it comes to navigation.

Let's consider this interface below. The top red section and bottom blue section have their three black buttons appear at a randomly chosen point on their Y axis. That means the first button could be much lower than the far right one. In this case, the VoiceOver engine would choose "button 2" first, which isn't the intent of the interface.

To address it, we can use `UIAccessibilityNavigationStyle`:



Here, the elements may not be in a sensible order for Switch Control by default. By leveraging the navigation style property and ordering the elements, we can make sure things work as Switch Control users would expect.

```
// Xcode -> SwitchControlFig1ViewController.swift

[topContainer, bottomContainer].forEach {
    for idx in 0..<buttonsPerContainer {
        let yPosition = randomYPositionIn(container: $0)
        let xPosition = xPositionFor(index: idx)
        let button = createButtonWithText(String(idx))
        button.frame = CGRect(x: xPosition, y: yPosition, width: 80,
height: 80)
        $0.addSubview(button)
    }

    // Switch Control considerations
    // .combined means the elements are navigated by
    // single level sections (the top and bottom individually)
    $0.accessibilityNavigationStyle = .combined
```

```
    // Precise ordering (1,2,3 and then 4,5,6 - no matter of their
visual order)
    $0.accessibilityElements = $0.subviews
}
```

Using `.combined`, along with ordering the accessibility elements in numerical order, would ensure that the switch navigates the top and bottom section correctly. Now, instead of slowly focusing each and every element (out of order, no less!) a Switch Control user can choose the top or bottom container, and then sensibly navigate each button in order after it's selected.

Again, if we hadn't done anything here, we'd get the default behavior of going top-to-bottom, and left-to-right which doesn't match the intent of the interface.

`accessibilityRespondsToUserInteraction`

This property, added in iOS 13, can help give the accessibility engine a better of idea of which elements are interactive. A common use case for this is perhaps a text control which has a gesture recognizer attached to it. By default, this interactivity isn't known to the accessibility engine, but we can set that to change this:

```
myLabel.accessibilityRespondsToUserInteraction = true
```

This property is derived from other properties on its own, but sometimes you'll want to change it to get across the intent of the element. For example, if the accessibility traits had `.notEnabled` in it:

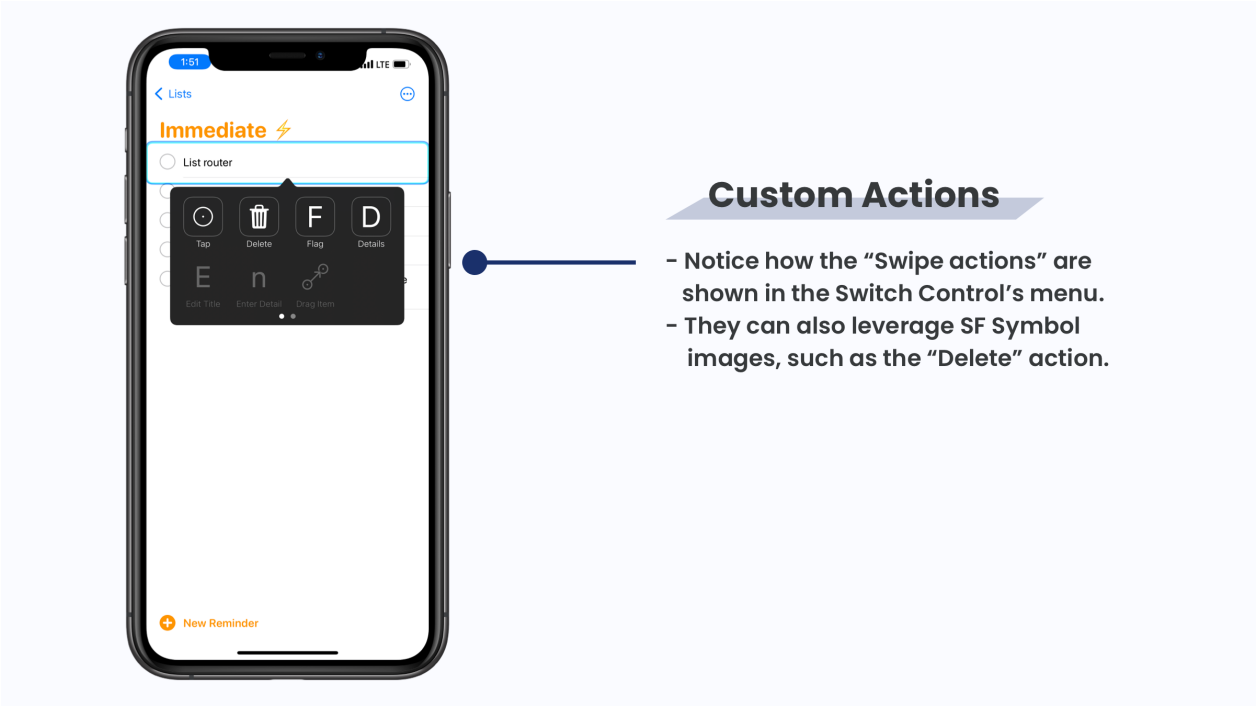
```
myLabel.accessibilityTraits = [.notEnabled]
```

Then this property automatically would be false. Note that this property specifies interactivity, not *just* being an accessible element. You could achieve a similar outcome by tweaking a few other properties:

```
myLabel.isAccessibilityElement = true
myLabel.accessibilityTraits = [.button]
```

Gestures and Actions

As the user navigates with a switch, they'll bring up menus that look like this one:



The switch control showing the available actions they can take.

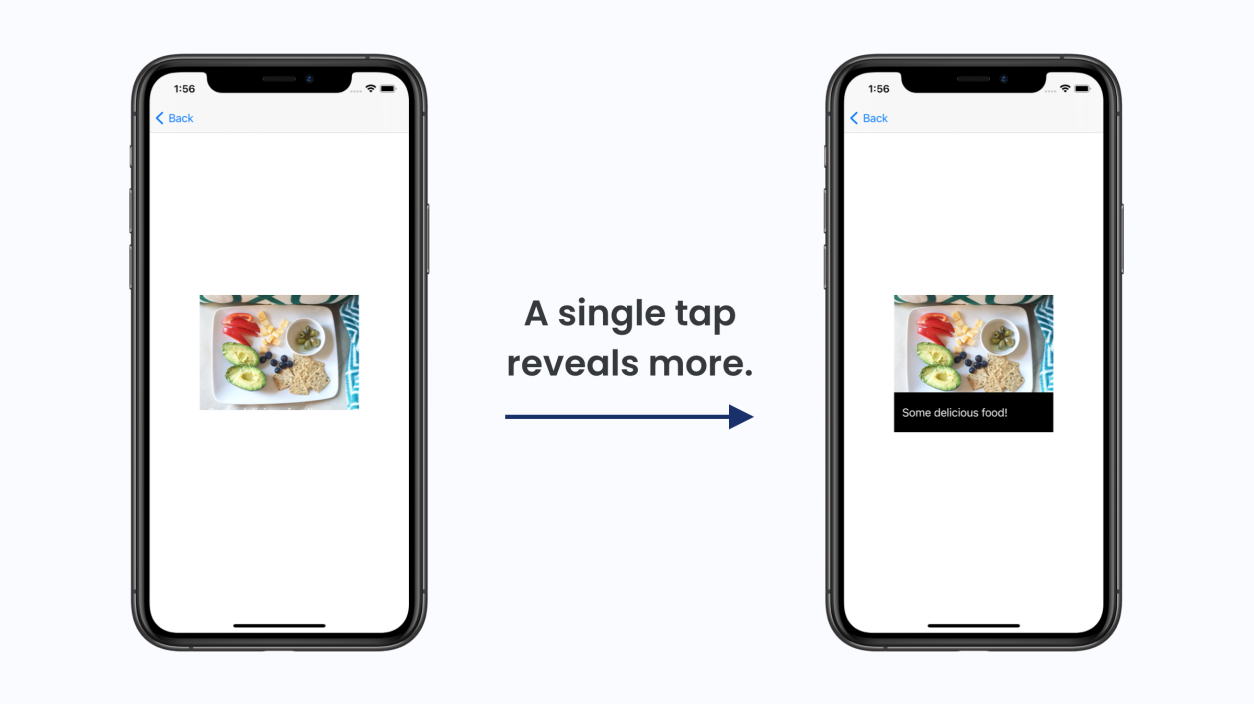
As you can see, they can do a wide variety of actions - it nearly covers anything you could think to do on an iPhone such as execute a gesture, perform a custom action and more. This is the reason things like a custom action pay off *early* and then pay off *often*. Sure, you may have made it easy for someone who relies on VoiceOver (a rewarding thing to do as a developer!) but that custom action could also be crucial to a switch control user as well.

Throughout the accessibility chapters, we keep coming back to this idea of making navigation quick and efficient. No matter how delightful you might assume your

app is, if it takes an accessibility user five steps or actions to do something we could do with one swipe, I promise you - it's not. We've got all we need in our tool belts to address these things, and no matter if it's an honest mistake in your own app - making it right is typically a matter of only a few lines of code.

Automatically Performing Actions

Some Switch Control users may want have certain actions performed automatically on their behalf to promote information flow and intuitive navigation. Consider this image below:



Some actions such as this may be good candidates to perform automatically for Switch Control users.

Since tapping the image to reveal more information is likely something a lot of users would want to do, and it's a low-impact decision - we can have it happen when the image receives and loses focus from the accessibility engine:

```
// Xcode -> SwitchControlFig2ViewController.swift

// MARK: Detail Label
@objc private func toggleDetailsTapHandler() {
    revealDetails()
}

private func revealDetails(forceHidden:Bool? = nil) {
    if let forcedValue = forceHidden {
        dimView.isHidden = forcedValue
    } else {
        dimView.isHidden.toggle()
    }
}

// MARK: Accessibility Focus
override func accessibilityElementDidBecomeFocused() {
    revealDetails(forceHidden:false)
}
override func accessibilityElementDidLoseFocus() {
    revealDetails(forceHidden:true)
}
```

By overriding `accessibilityElementDidBecomeFocused` and `accessibility-ElementDidLoseFocus` we can make sure this navigation flow isn't as cumbersome for accessibility users. With the code above, the extra details just show up automatically, eliminating another action a switch control user would need to otherwise make. What if they were scanning ten, or even hundreds of images? This simple tweak could save them hours.

After you create an interface, or receive a design or whatever it is - take a second and stop to think if some actions should just *happen* when an element gets focused. There is no golden rule here, but what I've found to be a close bet is this set of questions:

1. Is the action I'm considering to perform automatically "low cost" (i.e. it doesn't delete something, or go to a new screen, etc).
2. If so, if it happened every time - would that be helpful or obnoxious?
3. If the answer is, it's helpful - then it's a good choice to do what we've done above.

Consider the Navigation Medium

Switch Control users may not be able to navigate an iOS device with the same physical dexterity or efficiency as you or I, no matter how well we'd like for that to happen. Because of that, consider what that means for user experience purposes.

If you know the user can't respond to an interaction as quickly as you're used to, then take extra steps to make your interface resilient to that fact. Do you have a two factor authentication view? If the switch control is on, employ a more relaxed timeout policy so they've got plenty of time to enter in their code. Is their sensitive information you might have showing? Consider a toggle to hide or show it so that Switch Control users can be in complete control of whether or not it's appearing.[11]

By exercising a high degree of empathy, we can make a complete experience for *everyone* who wants to use your app. And remember - that shouldn't be your bar, that should be your baseline.

---

[11] Recall that many of these switch hardware units could be mounted on a wheelchair, so it's easy for anyone around to see the screen of their iOS device.

Three Key Takeaways

1. The Switch Control feature ensures those with limited mobility are in no way limited by their iOS device.
2. That said, it's on us to make sure it works correctly and if we've done a great job with VoiceOver, we're likely 99% of the way there.
3. But, there are a few times we need to a bit more for navigational purposes or to expose elements to accessibility APIs.

Voice Control

Voice Control, introduced in iOS 13, gives users a way to interact with their iPhone using only their voice. While it can be used by many, it proves especially useful for those with motor disabilities who might have difficulty touching or interacting with the physical hardware of their device.



Voice Control activated on iOS.

Before we continue, it's easy to get lost in vernacular here. VoiceOver and Voice Control are both accessibility technologies, but how they work and who they are (mostly) intended for are different. VoiceOver helps users drive the interface even if they can't see it, while Voice Control helps users drive the interface even if they can't touch it.

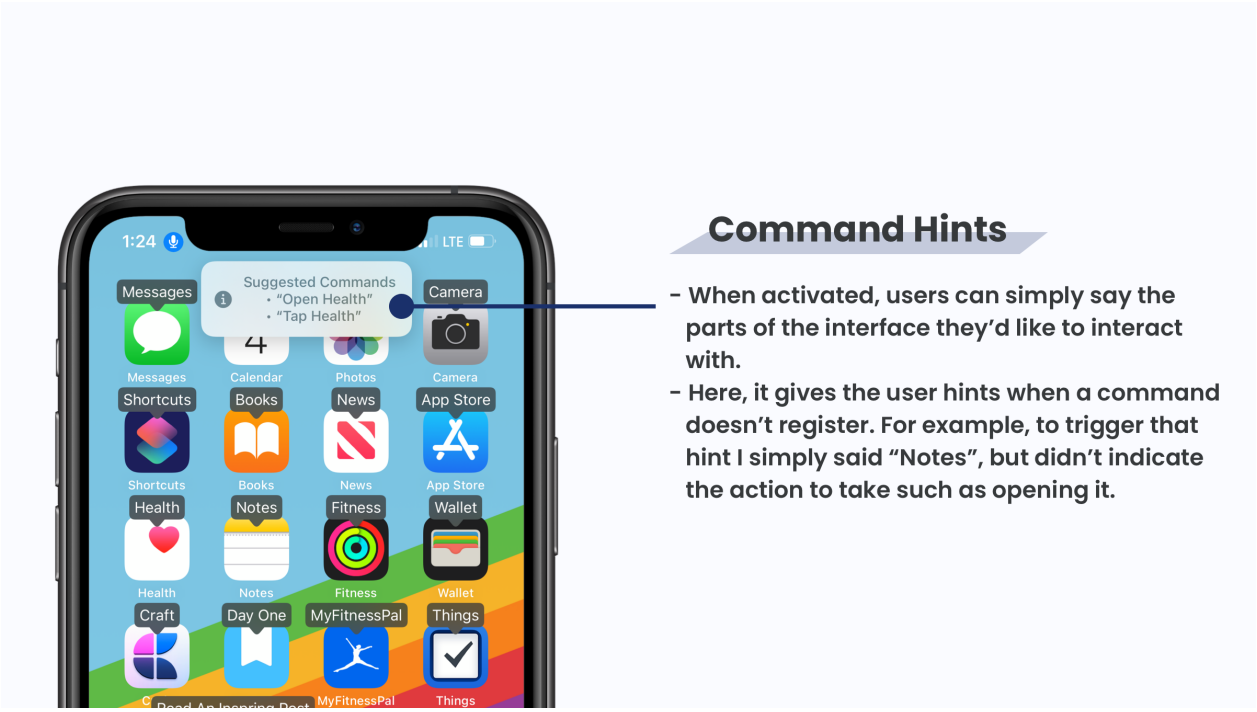
In addition, it is a binary choice. Users cannot have VoiceOver running alongside Voice Control, and vice-versa.

Setup requires a one-time file download from Apple, so to get ready for development it's a good idea to go ahead and get that done on each device you wish to use for testing. Visit the "Accessibility" section within the Settings app and choose "Voice Control" to get going. In addition to allowing navigation and interaction, it also boasts a wide array of options for users to tweak so it'll behave virtually any way they'd want. They can change the:

- Language.

- Customizable commands.
- The ability to teach new words to Voice Control.
- Toggle hints to show within the Voice Control interface elements.
- A grid overlay option to section out the screen (i.e. “Tap square 24”).
- Attention Aware activation. If the device has Face ID, Voice Control kicks off right when a user looks at their phone and disables itself when the device goes to sleep.
- Confirmation toggle options, which makes the system confirm any action with an alert before they are activated. Recall the importance of this “error tolerance” in our discussion in the chapter “The Switch Control” to learn more.

Once you have the one-time configuration file downloaded, activating it is as simple as instructing Siri to turn it on, “Turn on Voice Control.” And, unsurprisingly, you can turn it off the same way, “Turn off Voice Control.” Once activated, using iOS becomes remarkably intuitive by solely relying on your voice. You simply tell it what you want to do and the system will likely pick it up so long as it follows a verb-noun pattern, “Tap Settings” and so on:



Voice Control displays hints when the command is ambigious.

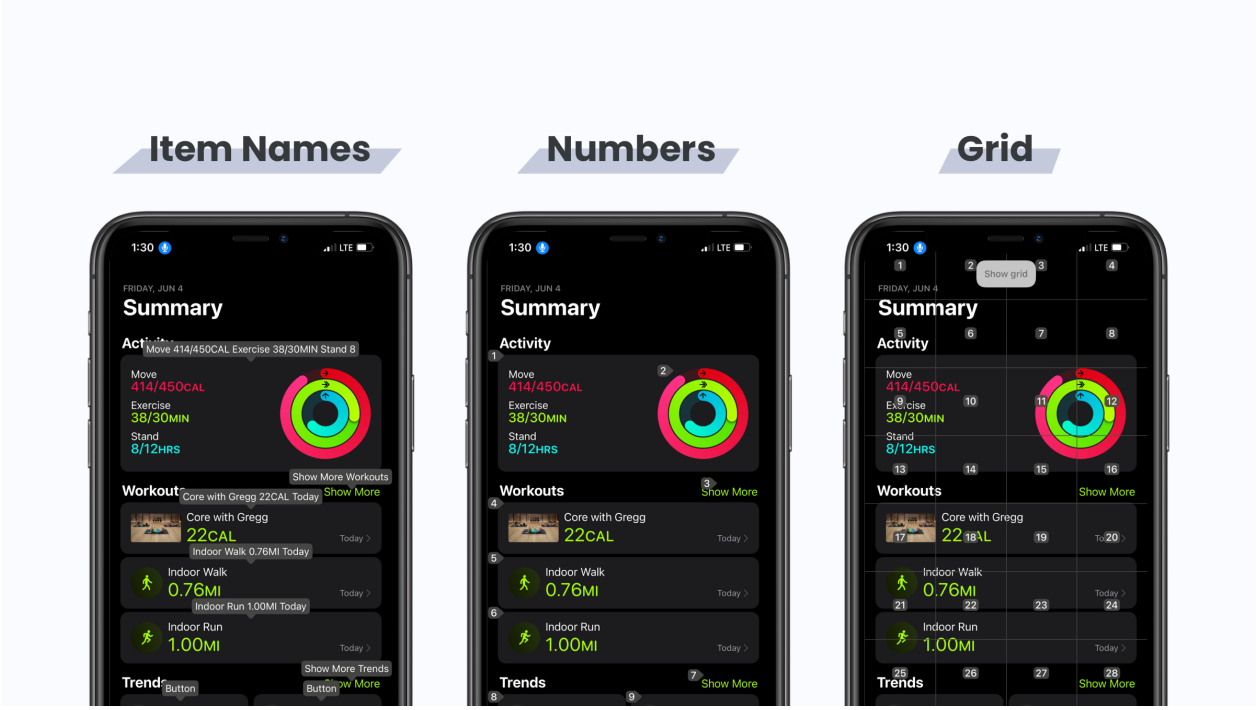
Voice Control also has two modes that prove useful for long-form text entry. Those are Dictation mode (the default when a text control is activated) and Command mode. Dictation mode enters in any words that the user speaks into the active text control *unless* it's a specific Voice Control command. Users would want this mode when composing an email, writing a text message or any other task which requires significant text entry. Command mode is what's used to navigate and interact with iOS, and unless text entry is occurring, this is what Voice Control defaults to using. Regardless, you can switch between the two modes at any point by simply calling out their name, "Command mode" or "Dictation mode."

There are three main ways that Voice Control can decorate the interface to allow for easier navigation, referred to as screen overlays. They are:

1. Item names
2. Numbers

3. Grid

Users can switch between these by saying “Show (screen overlay option)” when Voice Control is active.



All of Voice Control's screen overlay options.

Item names will display a tooltip like bubble around elements, and the user says which of them they’d like to interact with. The numbers option works similarly, except instead of item names the interface will be decorated with numbers for each element.

The grid is another unique feature of Voice Control that works much differently than using item names or numbers. While Voice Control makes navigation quick for most cases, other times the interface might be expansive and the options available to take could number in the tens, twenties or even higher. For example, think of Maps and all of the landmarks someone might want to interact with.

Further, if there are several elements, then there are also several tool tips floating all over the screen. At this point, Voice Control becomes too cluttered to use with the item names or numbers option.

In these cases, Voice Control users would likely activate the grid. This sections out the entire interface into squares, each with a corresponding number. The user simply refers to whatever section of the grid they want to interact with, "Double tap 25."

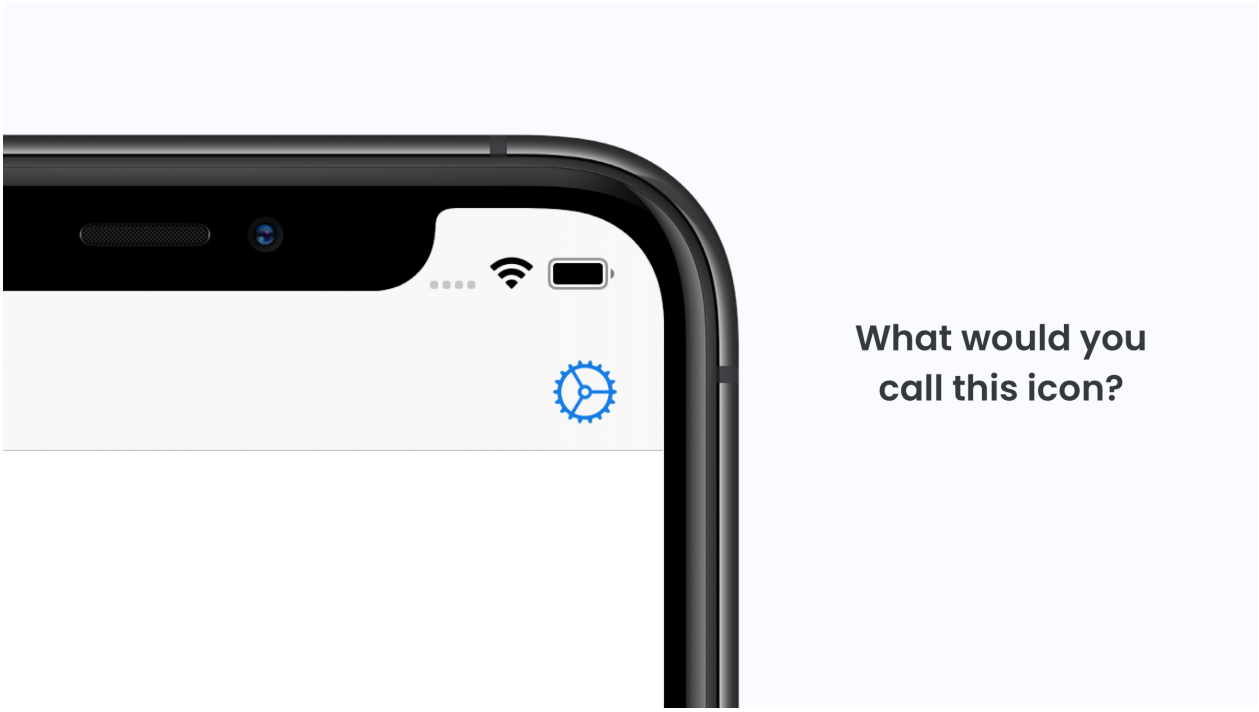
How it Works

Like many of iOS accessibility technologies, the great thing is we get that "it just works" magic if we've already setup sensible values for "The Big Three" properties we discussed in the chapter "The 10,000 Foot View: Basics".

Voice Control will pull from your accessibility label's value to display text when the screen overlay is using the item names option. For the most part, this is exactly what you want. However, if you do need to make things simpler or provide more options, the `UIAccessibility` protocol provides us with this property:

```
open var accessibilityUserInputLabels: [String]!
```

With it, you can provide an array of strings that accurately describe an element. But, why is it an array? Consider this element:



If someone asked you what that was, what would you say? A setting icon? A gear, perhaps a cog? All of those make sense, and I would say each of them do, in fact, describe the element.

Now consider that icon in the context of Voice Control while using the item name screen overlay option. Wouldn't it be convenient if the user could refer to that element by *any* of those?

That's exactly what this property affords to us. Even better, it's available in SwiftUI too. Following our example above, here's what it would look like:

```
// UIKit, Xcode -> VoiceControlFig1ViewController.swift
let cogBarButtonItem = UIBarButtonItem(title: "Settings", image:
UIImage(systemName: "gear"), primaryAction: nil, menu: nil)
cogBarButtonItem.accessibilityUserInputLabels = ["Settings", "Cog",
"Gear"]

// SwiftUI, Xcode -> VoiceControlFig1View.swift
private let cogInputLabels: [Text] = [Text("Settings"),
```

```
                                        Text("Cog"),
                                        Text("Gear")]
Image(systemName: "gear")
  .resizable()
  .foregroundColor(.blue)
  .accessibilityInputLabels(cogInputLabels)
```

With that setup, Voice Control users could interact with the cog by announcing any of the names in the array. The first item, however, will get precedence within the interface. It's what will show if the "Item names" screen overlay is activated.

What this property gives us are two main advantages:

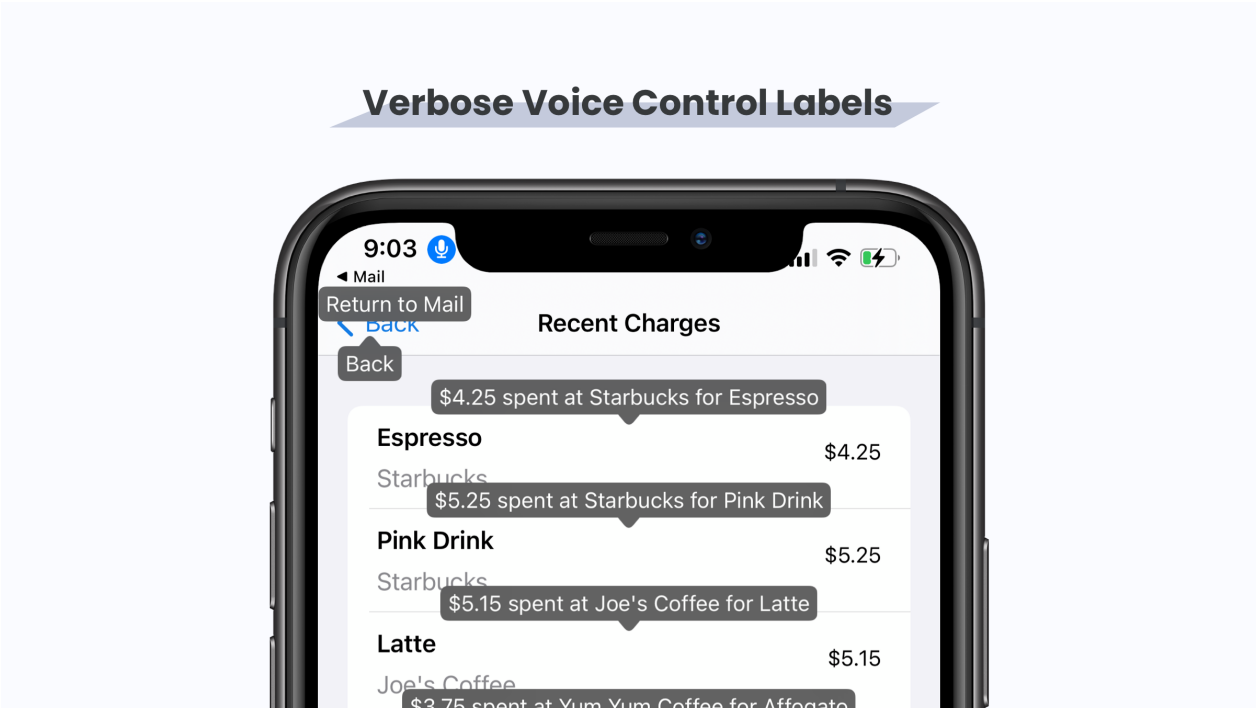
1. Users can refer to one element by many different names.
2. We can provide more succinct values if the default one Voice Control uses is too verbose.

Tips

Evaluate Interfaces for Verbose Accessibility Labels

We've discussed this notion of "flattening" your view hierarchy for accessibility purposes. If you'd like to revisit the thinking behind this methodology, look at the "Voice-Control" chapter and then check out the "Flattening Hierarchy" tip.

Assuming we've done that, though, it could inadvertently lead to another issue. Check out what the labels would produce for Voice Control users if we had used the flattening technique on this user interface:



These labels are too long to user easily with Voice Control.

That's not ideal, and no Voice Control user wants to say any of those out loud to navigate around. True, they could switch to either the grid or numbers overlay, but why force them to make the choice? Accessibility, in many respects, is all about offering choice to the user. As such, forcing them down one predefined accessibility path should typically be a non-starter. Here's how we can clean this up using the accessibility input labels to allow for easier item name navigation.

In particular, look at where we assign to `accessibilityUserInputLabels` in the UIKit sample, and `accessibilityInputLabels` for the SwiftUI code:

```
// UIKit, Xcode -> VoiceControlFig1ViewController.swift
let datasource = LineItemTableDataSource(tableView: tableView,
cellProvider: {
(tableView, indexPath, model) -> UITableViewCell? in
    let cell = tableView.dequeueReusableCell(withIdentifier:
BasicTableViewCell.cellID, for: indexPath) as! BasicTableViewCell
```

```
    cell.setData(withTopText: model.name, bottomText: model.vendor,
trailingTex
model.amount, customAccessibilityLabel: model.amount + " spent at "
+
model.vendor + " for " + model.name)
    cell.accessibilityUserInputLabels = [model.name + " " +
model.vendor]
    return cell
})

// SwiftUI, Xcode -> VoiceControlFig2View.swift
HStack {
    VStack(alignment: .leading) {
        Text(item.name)
            .padding(.bottom, 4)
        Text(item.vendor)
            .font(.caption)
            .fontWeight(.semibold)
            .foregroundColor(Color(UIColor.tertiaryLabel))
    }
    Spacer()
    Text(item.amount)
        .fontWeight(.medium)
}
.padding(.all, 8)
.accessibilityElement(children: .combine)
.accessibilityAddTraits([.isButton])
.accessibilityInputLabels([Text(item.name + " " + item.vendor)])
```

Now, we've traded out the long, wordy labels for a short and simpler way to refer to the elements:



These labels lend themselves well to Voice Control navigation without sacrificing any meaning behind what they represent.

Even better, we haven't lost anything accessibility wise. Recall that VoiceOver and Voice Control are mutually exclusive technologies - so that means we can *still* flatten our hierarchies and leverage the accessibility input labels without either one taking away from the other.

One more point here regarding SwiftUI. Due to its open ended nature when it comes to its lists and what kind of View you throw at it, you'll likely need to spend some time getting the accessibility traits, container strategy and labels right. In our example, if you comment out this part:

```
.accessibilityElement(children: .combine)
```

You'll find that the accessibility engine puts an input label over *all* of the elements in the "cell", which is not what you want. So, take care to run your SwiftUI views over a few

times when it comes to Voice Control testing to get the results you'd expect - which is sound advice regardless.

Localizing Descriptions

You can add localized strings to accessibility input labels, regardless of whether or not you're using SwiftUI or UIKit. Many times, these labels are derived from user input, so this may not be necessary - but the option is there if you need it:

```
// UIKit
let localizedHeader = NSLocalizedString("headerInput", comment:
"Header input for accessibility input labels")

// Later on, in your UIView
view.accessibilityUserInputLabels = [localizedHeader]

// SwiftUI
let localizedHeader: LocalizedStringKey = "headerInput"
let inputLabels = [Text(localizedHeader)]

// Later on, in your View
struct HeaderView: some View {
   Text("Check out our deals")
     .accessibilityInputLabels([inputLabels])
}
```

You might run into this when you have controls whose accessibility label don't quite lend itself to intuitive Voice Control navigation, as we just saw above, and its value isn't derived from any sort of input. In these cases, be sure to localize the text.

Faster Accessibility Testing

One beautiful side effect of testing your app with Voice Control is that you can get a nice feel for where your app stands overall accessibility wise.

Why?

A few reasons:

1. You can trivially see if your elements have an accessibility label set, since that's the default value for Voice Control elements in the "item name" screen overlay.
2. Using the "numbers" screen overlay, you can see if you've got elements ordered and grouped sensibly.
3. Anything you can activate with Voice Control also means you can activate it with The Switch Control (and vice-versa).

In a way, Voice Control has afforded developers with a visual representation of where your app stands with many accessibility technologies. This isn't a substitute to go through flows and test things out yourself, but it is a useful method to leverage in the beginning stages of accessibility development to quickly discover what might be left to consider working on.

Three Key Takeaways

1. Voice Control allows users who might have difficulties touching a device navigate and interact with their iPhone.
2. Like many of iOS accessibility technologies, a lot of the work is done for you if you've setup sensible accessibility elements.
3. However, you can help users navigate more efficiently by leveraging the `accessibilityUserInputLabels` array.

Customized Accessibility Content

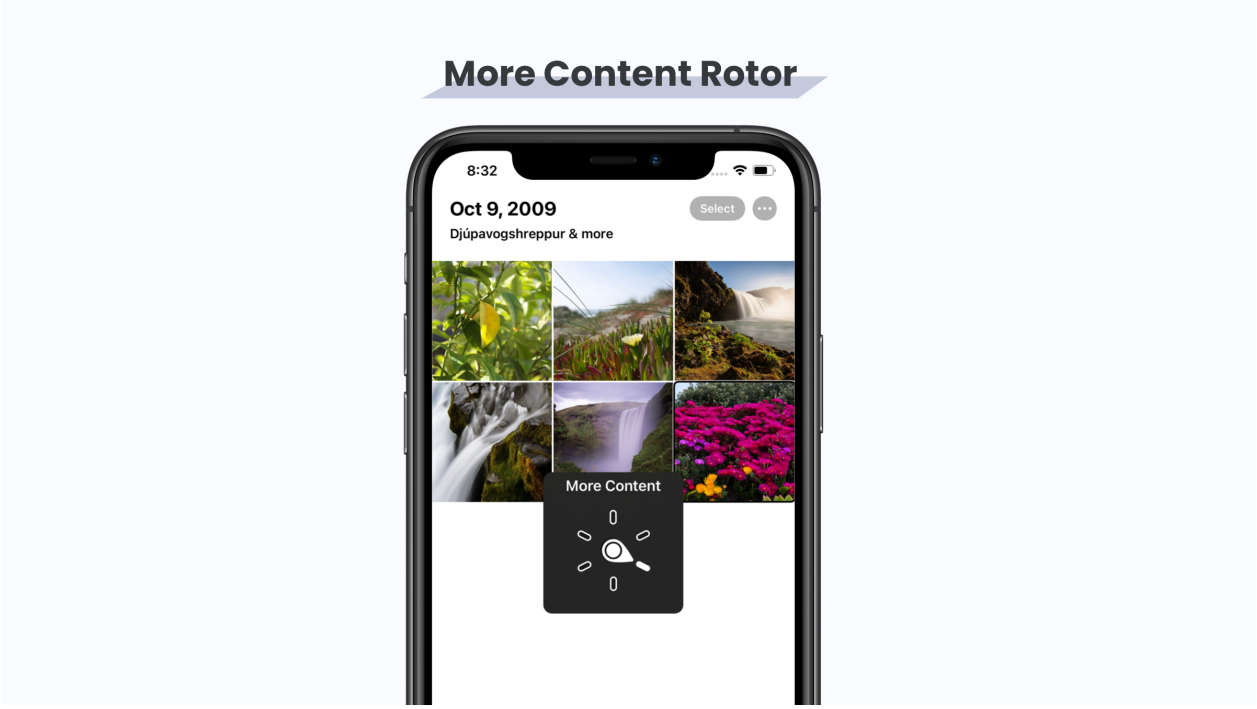
Throughout several chapters so far, we've often optimized our apps and their accessibility experience to make sure we strike an important balance:

1. Giving enough context to make choices.
2. Avoiding *too* much context resulting in cumbersome navigation.

It takes a combination of testing, gut calls and feedback to find the right balance. There are times, however, where elements within an interface intrinsically contain dense information. What do you do here?

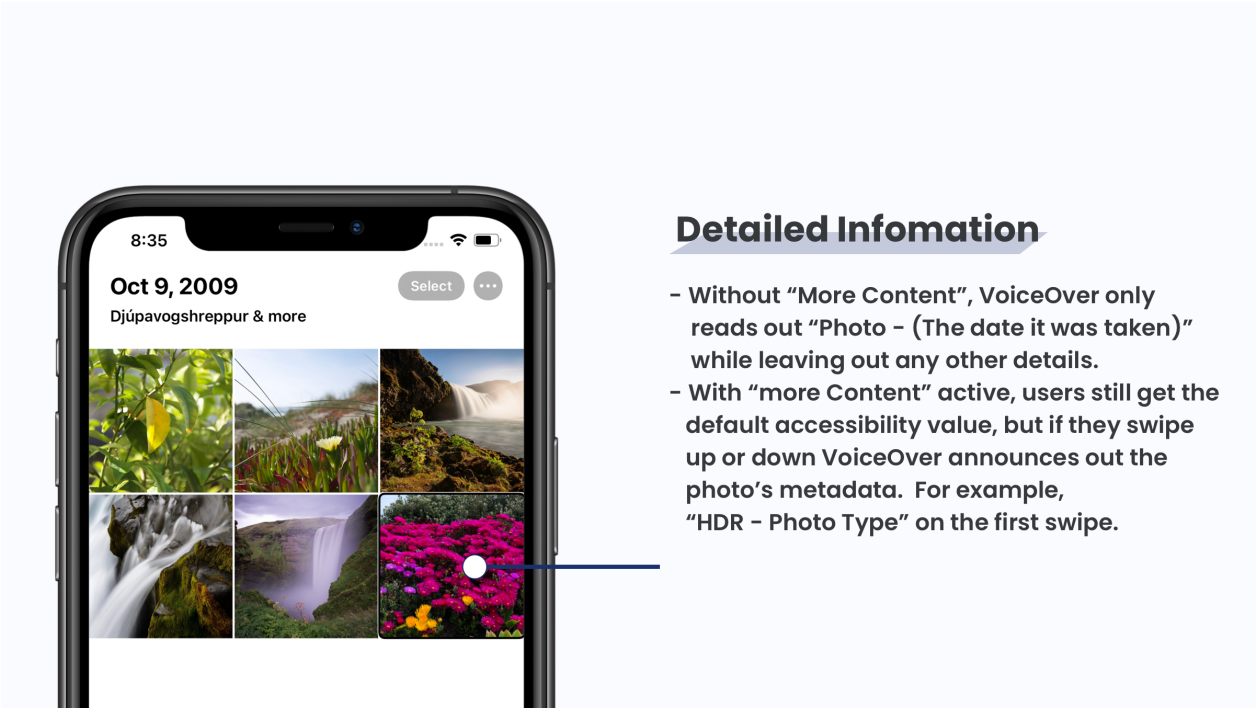
We've seen a number of ways to address this. You could flatten the hierarchy for easier navigation, provide input labels to make Voice Control work or tweak which subviews, or Views within another SwiftUI View, register as an accessibility element.

Starting with iOS 14, Apple has added another tool for this exact situation, and it's called customized accessibility content. It's an API that allows you to customize your interface to vend more detailed accessibility information about an element, but *only* when a user specifically requests it. For VoiceOver users, it's surfaced via the "More Content" rotor:



Customized accessibility content is activated via the "More Content" rotor.

In you go to Photos on your iOS device right now, you can try it yourself. Photos is a great example for this API when you consider how much information each photo itself might have. Each photo could have metadata that's full of details; camera make, location, the time and things like weather conditions could all be in there. Of course, if that was all exposed to VoiceOver from the beginning – navigating through Photos would be impossible. Yet, that's also meaningful information to surface about a photo. As such, Apple has done just that via the "More Content" rotor:



Information density is thoughtfully handled by leveraging custom accessibility content.

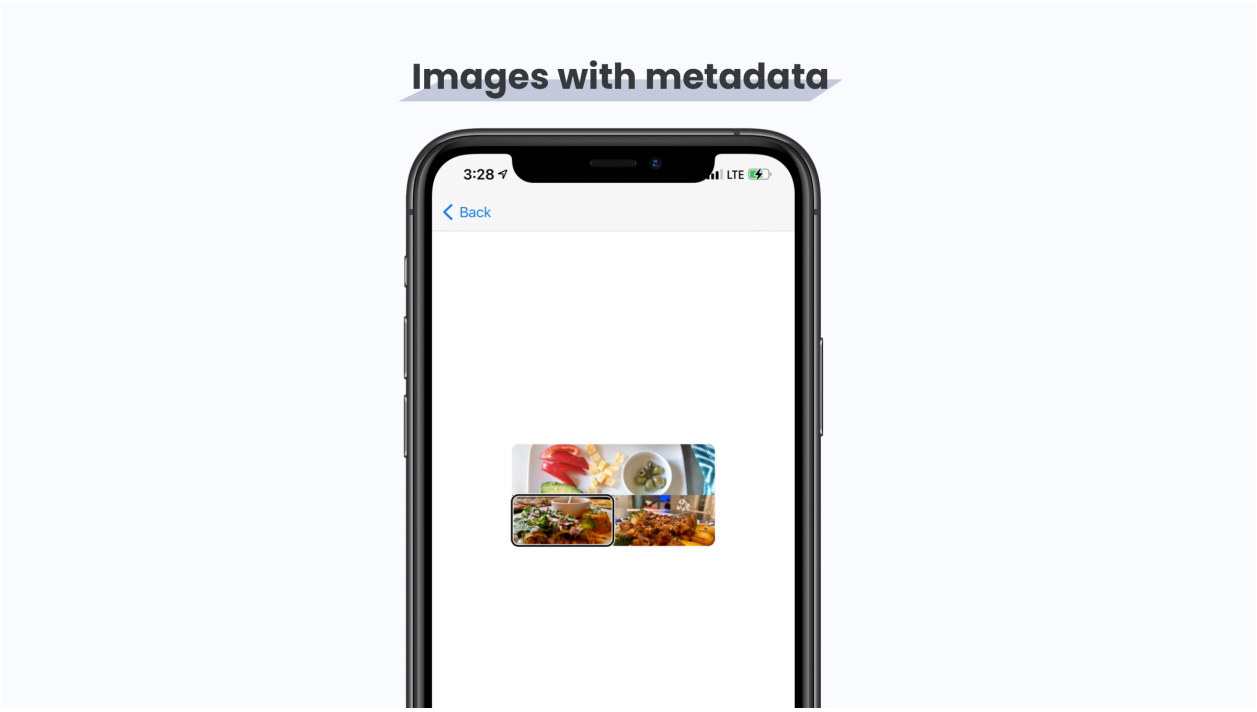
How it Works

Customized accessibility content is housed within one protocol for UIKit based interfaces, `AXCustomContentProvider`, which itself only has one requirement:

```
var accessibilityCustomContent: [AXCustomContent]! { get set }
```

The `AXCustomContent` object is simple, it works essentially like a dictionary. You give it a label and a value, which signifies *what* you’re describing and then what that thing’s *value* is. For images, that could be a label of “Orientation” which has a value of “Landscape”. In SwiftUI, the same functionality is exposed via a modifier, `accessibilityCustomContent`.

Consider this image cluster interface below, with each image having metadata stored for each of them:



By default, the accessibility label does an adequate job of describing each image. In the sample project, the UIKit control has each image's name describing what it is anyways - so we can recycle that for accessibility purposes:

```
// Xcode -> UIKit ->
CustomizedAccessibilityContentFig1ViewController.swift ->
CornerRoundedImage.swift

// Within the CornerRoundedImage initializer...
isAccessibilityElement = true
accessibilityLabel = "Photo of \(name)"
```

When an image is selected, VoiceOver will read out something like "Healthy food." That solves navigation and supplies VoiceOver with enough context about the photo, but we don't want to hide all of the metadata from accessibility users. We want to optionally expose it and let users discover it at their own pace when they select the "More Content" rotor.

To do so in UIKit, we'd use `AXCustomContent` objects to describe each piece of metadata found below in the `mockMetadata` dictionary, where each key-value pair becomes an instance of custom content:

```
// Xcode -> UIKit ->
CustomizedAccessibilityContentFig1ViewController.swift ->
CornerRoundedImage.swift

// Within the CornerRoundedImage...
private let mockMetadata: [String:String] = ["Created":"July 8th,
2021", "Camera Model:":"Leica Deluxe 7 Digital
Camera", "Location":"Ozark Missouri"]
```

Putting it all together for our image cluster interface above, the implementation would look something like this:

```
// Xcode -> UIKit ->
CustomizedAccessibilityContentFig1ViewController.swift ->
CornerRoundedImage.swift

class CornerRoundedImage: UIImageView, AXCustomContentProvider {
    var accessibilityCustomContent: [AXCustomContent]! {
        get {
            return mockMetadata.map {
                return AXCustomContent(label: $0.key,
                                       value: $0.value)
            }
        }
        set (customContent) { }
    }

    private let mockMetadata: [String:String] = ["Created":"July
8th, 2021", "Camera Model:":"Leica Deluxe 7 Digital Camera",
"Location":"Ozark Missouri"]

    // Rest of image related implementation...
```

```
}
```

With that set, we can activate "More Content" via the Rotor Control. Then, VoiceOver will cycle through each instance of `AXCustomContent` for the focused image. When the user swipes up or down, it'll announce its label and value.

In SwiftUI, the *idea* is the same, but the implementation is completely different. Instead of adopting a protocol (which is restricted to a class instance anyways, so SwiftUI and its struct-based View type is out) we leverage modifiers. Consider this view that shows an image of food which exposes its extra metadata using the custom content modifier:

```
// Xcode -> SwiftUI -> CustomizedAccessibilityContentFig2View.swift

struct AXCustomContentImageView: View {
    let foodName: String = "Healty Food"

    var body: some View {
        Image(foodName)
            .resizable()
            .aspectRatio(nil, contentMode: .fit)
            .frame(width: 300, height: 300)
            .accessibilityLabel(foodName)
            .accessibilityCustomContent("Description", "Healthy food
platter")
            .accessibilityCustomContent("Created", "July 8th, 2021")
            .accessibilityCustomContent("Camera Model", "Leica
Deluxe 7 Digital Camera")
            .accessibilityCustomContent("Location", "Ozark
Missouri", importance: .high)
    }
}
```

By stacking them up together, you can build up content for the "More Content" rotor much the same way we leveraged a dictionary in UIKit. The`accessibilityCustom-`

`Content` modifier comes with a few different signatures, all of which boil down to what we've done with the UIKit method:

1. A key.
2. It's value.
3. And optionally, its importance.

However, it's not always as simple as supplying the modifiers. If you do so on any sort of complex view, you likely need to composite them together and opt out any other views from the accessibility engine. If you don't, VoiceOver will have, quite literally, a lot to say:

```
var body: some View {
    VStack {
        Text("Pizza Receipe")
        Image("Pizza")
        HStack {
            /* Text() with recipe components below */
        }
        .accessibilityHidden(true)
    }
    .accessibilityCustomContent("Sauce", "Marinara")
    .accessibilityCustomContent("Toppings", "Pepperoni")
    .accessibilityCustomContent("Cheese", "Cheddar")
    .accessibilityCustomContent("Crust", "Thin")
}
```

In the above example, the same ingredients already exist in the view tree within the HStack. But, we've also supplied them via custom content – so there is no reason to keep them eligible to the accessibility engine. If we did, users would get all of that information upfront which defeats the purpose of the API, repeats content and makes navigation a chore.

Tips

Choosing the Correct Importance

The `importance` property of `AXCustomContent`, or the parameter with a similar name for SwiftUI's modifier, allows you to elevate certain information that you think should be necessary for VoiceOver. If you mark content as important, it'll be included along with whatever the element's `accessibilityLabel` property or modifier currently reflects.

If your accessibility label is already descriptive enough, there isn't much of a reason to mark any custom content as important. If you mark too many things, or even just one element, as important - you can accidentally undo some of the benefits you've worked so hard for by implementing custom content.

Why? Because now, VoiceOver could be speaking quite a lot of text, making element descriptions far too long.

However, this property exists for a reason - so if you need it, by all means, use it. In our UIKit example, we might consider the photo's location as something we'd like to elevate to VoiceOver regardless if "More Content" is active or not:

```
// Xcode -> UIKit ->
CustomizedAccessibilityContentFig1ViewController.swift ->
CornerRoundedImage.swift

return mockMetadata.map {
    let content = AXCustomContent(label: $0.key, value: $0.value)
    if $0.key == "Location" {
        content.importance = .high
    }
    return content
}
```

```
// Xcode -> SwiftUI -> CustomizedAccessibilityContentFig2View.swift

/*
See View code above as AXCustomContentImageView()
This is here to move attention specifically to the
Modifier using high importance.
*/
.accessibilityCustomContent("Location", "Ozark Missouri",
importance: .high)
```

Now, when a photo is focused, VoiceOver will read out "(photo name). Location, Ozark Missouri." And in SwiftUI, we'll get the same effect.

When using SwiftUI, there are a few different ways you can think about using the importance parameter when supplying custom content:

1. Omit an `.accessibilityLabel` modifier, and use the custom content marked with a `.high` importance in its place.
2. Use both together to enrich any existing `.accessibilityLabel modifier` since they will both be spoken by default.

If you go with route number two, again – be sure to be cognizant of information overload. After all, the whole point of this API is to avoid exactly that.

Auditing for Dense Elements

Finding elements who might have dense information is what this API is essentially made for. So, be sure to audit your interface and ask yourself a few questions:

1. Did this element have more information that's useful to the user, but might have been too much to pack into an accessibility label?
2. If it does, is any of that critical data that should *always* be announced?

3. Or, is there any data I previously planned on omitting that I can now expose with customized accessibility content?

Any of those questions, if answered with a "Yes", should lead you to implement the `AXCustomContentProvider` protocol or SwiftUI's equivalent modifier.

SwiftUI and Custom Content Keys

If you've got a piece of custom content that you'd use in more than one view, consider extending `AccessibilityCustomContentKey` so you don't end up repeating yourself. In our examples, we've used some mock metadata of photos – each of which had a location.

Now, consider we've got another view in an app that also displays these photos a bit differently but it exposes the location metadata within it as well. Instead of writing out the location key again, we can create a strongly typed key the lends itself better to future-proofing and ease of use:

```
// Xcode -> SwiftUI -> CustomizedAccessibilityContentFig3View.swift

extension AccessibilityCustomContentKey {
    static var location: AccessibilityCustomContentKey {
        AccessibilityCustomContentKey("Location")
    }
}
```

With that in place, we can use another overload of the custom content modifier to leverage the strongly typed key instead of the string based ones we've been using:

```
// Xcode -> SwiftUI -> CustomizedAccessibilityContentFig3View.swift

struct CustomizedAccessibilityContentFig3View: View {
```

```
    var body: some View {
        Image("Healthy Food")
            .resizable()
            .aspectRatio(nil, contentMode: .fill)
            .frame(width: 80, height: 80)
            .clipShape(Circle())
            .accessibilityLabel("Healthy Food")
            // Note the key we made is used here
            .accessibilityCustomContent(.location, "Ozark,
Missouri")
    }
}
```

This is the classic D.R.Y. principle at play. If you've got more than one or two places in your code where you need the same key, better to extend `AccessibilityCustomContentKey` than to rewrite it over and over. Even more so here, as the cost of abstracting it out is simply one property that we've got to create.

SwiftUI and Pre iOS 15

Before iOS 15, this API wasn't exposed to SwiftUI as a modifier. Further, as we mentioned above, you cannot simply adopt the protocol on a `View` struct because it requires a class instance. But, you can still leverage it using the `UIViewRepresentable` protocol. If you've got a view, such as our UIKit image example above, you can wrap it up in a representable to use in SwiftUI:

```
// Xcode -> CustomizedAccessibilityContentFig1View.swift

struct CustomizedAccessibilityContentFig1View: View {
    var body: some View {
        WrappedCornerImageView()
            .frame(width: 240, height: 120)
            .clipped()
    }
```

```
}

struct WrappedCornerImageView: View, UIViewRepresentable {
    typealias UIViewType = CornerRoundedImage

    func makeUIView(context: Context) -> CornerRoundedImage {
        return CornerRoundedImage(withImageName: "Healthy Food",
roundingCorners: .all)
    }

    func updateUIView(_ uiView: CornerRoundedImage, context:
Context) {

    }
}
```

Or, going the other direction and with a bit more work, you can wrap an existing SwiftUI view into a containing `UIView` instance solely to access the custom accessibility content API. Then, you can double back and use the representable approach as we've done here.

Three Key Takeaways

1. Customized accessibility content allows users to be in control of when they want to receive more detailed information about elements in your app.
2. Look for elements that are information dense to opt-in for customized accessibility content.
3. Carefully consider when to leverage the `.high` importance attribute.

Behaviors Overview

Accessibility in UIKit and SwiftUI have a lot of surface area to cover. When you think about all of the ways we interact with our device, the sheer amount of options we have is staggering.

We could swipe through pages in a modal view, navigate in a "power user" fashion around an app using only keyboard shortcuts or even leverage the precision of the cursor on iPadOS. Maybe it's a mix of all three of those things! With all of the options we have as users to influence how the system should behave in response to our interactions, how do you even keep up from an accessibility standpoint?

The answer is little by little, and then all at once. As I've said previously, you won't score perfect marks on your first accessibility attempt. Instead, it's about making things incrementally better while *always* keeping it in mind. I've found that this is a realistic way to get your app where you want it to be from an accessibility standpoint.

With that in mind, we now arrive at a discussion around some accessibility "behaviors" we can implement. They'll help us focus elements, read out content, understand text or adjust the size of things like images to ensure they behave as they should for accessibility users. Here's what we'll focus on the following four chapters:

- `UIAccessibilityFocus`: An informal protocol[12] which helps technologies like VoiceOver know if an element is, can be or should be focused.
- `UIAccessibilityReadingContent`: A protocol that represents readable content. Useful for things like a long form article or a book within your app's experience.
- `UIAccessibilityContentSizeCategoryImageAdjusting`: A protocol that helps determine if an image should adjust its size for different content size categories and if so, by how much.
- `UIAccessibilityTextualContext`: A struct with several static properties that help define the *context* of some text. For example, are assistive technologies looking at text that represents source code, or is this a message inside a messaging app? Those two scenario have different needs for things like VoiceOver.

Together, these protocols (along with the textual content struct) work together to influence the *behavior* of your device, always ensuring that it works as intended for accessibility users. Let's look at each concept one by one.

---

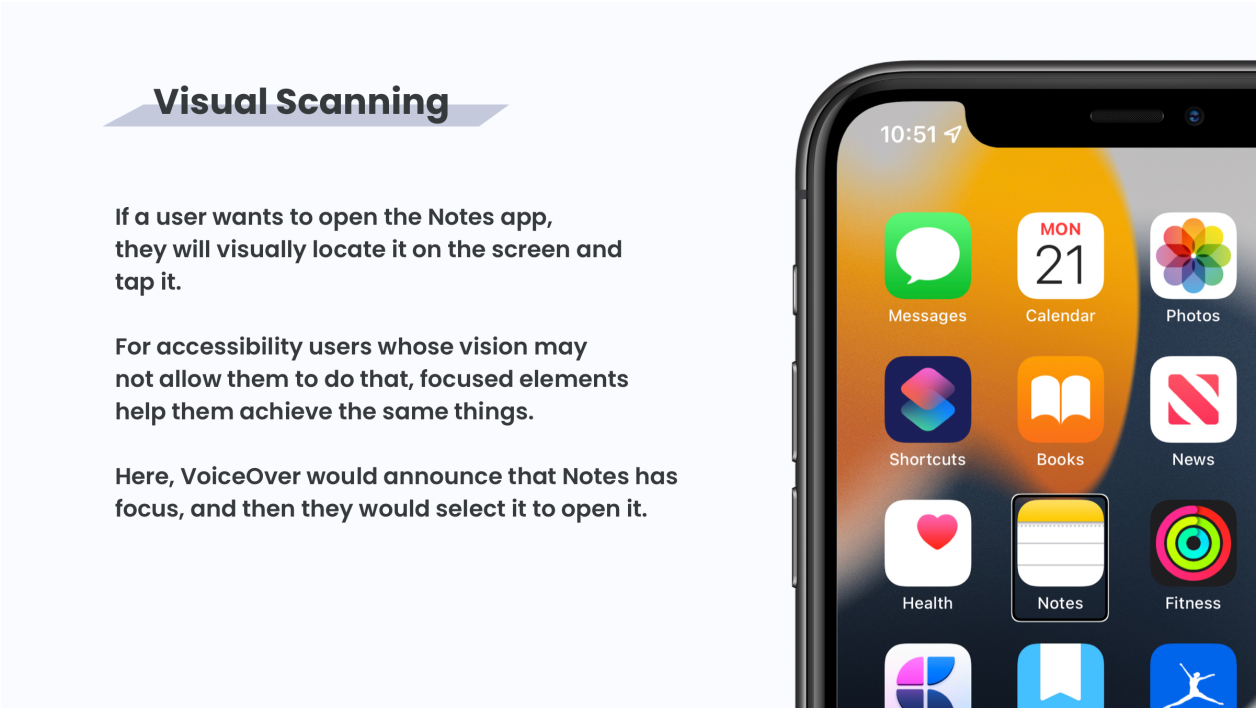
12 Like UIAccessibility itself is, as well.

Focus

When it comes to VoiceOver, Voice Control or any of iOS' accessibility technologies that help with navigation, the concept of focus helps answer the question "Where am I at?" along with "What is selected?" For many, this is performed visually, where the need for an explicit focus isn't as important. We may scan the screen with our eyes and simply decide what to do next.

For others with visual disabilities, VoiceOver can relay what's currently focused to help them make decisions for navigational purposes. Note that being focused *isn't* the same as being selected or activated. Instead, when an element is focused, it *can* be activated or selected.

In short, `UIAccessibilityFocus` lets accessibility users inspect an element without activating it. This is how they "scan" interfaces.



How it Works

As with the other behaviors we'll be discussing, you respond to focus by adopting a protocol in UIKit. As we'll see later, in SwiftUI it's achieved with a combination of a modifier and a property wrapper.

For UIKit, the `UIAccessibilityFocus` protocol gives you three opportunities to respond to focus changes within your interface. Each function corresponds to the focus state, and they are:

1. `accessibilityElementDidBecomeFocused`
2. `accessibilityElementDidLoseFocus`
3. `accessibilityElementIsFocused() -> Bool`

You can override any of the three functions above to perform logic when the focus state changes.

In SwiftUI, you can achieve much of the same via the `AccessibilityFocusState` struct, which is exposed as a property wrapper. Using it, SwiftUI can update a value that changes with the focus of any active accessibility element being used by accessible APIs such as VoiceOver, Switch Control and more. Instead of explicit overrides for each, you can use the `accessibilityFocused(_:equals:)` modifier to achieve the same things:

```
// Xcode -> SwiftUI -> AccessibilityFocusFig1View.swift

struct AccessibilityFocusFig1View: View {
    var body: some View {
        VStack {
            FocusSquareView()
            Spacer()
            FocusSquareView()
            Spacer()
            FocusSquareView()
        }
    }
}

struct FocusSquareView: View {
    @State private var focusText: String = "Focus State:\nUnknown"
    @AccessibilityFocusState private var isFocused: Bool

    var body: some View {
        ZStack {
            Rectangle()
                .fill(Color(UIColor.systemTeal))
            Text(focusText)
                .multilineTextAlignment(.center)
                .font(.callout)
                .lineLimit(nil)
        }
        .frame(width: 100, height: 100)
        .accessibilityElement(children: .combine)
```

```
            .accessibilityValue(focusText)
            .accessibilityFocused($isFocused)
            .onChange(of: isFocused) { val in
                if val {
                    focusText = "Focus State:\nFocused"
                } else {
                    focusText = "Focus State:\nNot Focused"
                }
            }
    }
}
```

When focus enters the accessibility element, SwiftUI mutates the property declared with the `AccessibilityFocusState` property wrapper to reflect the focus state. For example, if it were declared with a custom type, when accessibility focus left the element it would be set to `nil`, or `false` for a boolean type. By querying it, you can determine the focus state of elements.

On the other hand, if you want to programmatically change focus (much like we did in UIKit in the Switch Control chapter) you can mutate the value yourself. Pay attention to the `onAppear` implementation towards the bottom:

```
// Xcode -> SwiftUI -> AccessibilityFocusFig2View.swift

enum AXPurchaseFocusableElement: Hashable {
    case textfield
    case button
}

struct AccessibilityFocusFig2View: View {
    @State private var additionField:String = ""
    @AccessibilityFocusState private var axFocus:
AXPurchaseFocusableElement?

    var body: some View {
        Form {
```

```
            Text("Kids, Grab a Parent.\n\nComplete the problem below
to view purchases.")
                .lineLimit(nil)
            TextField("", text: $additionField, prompt: Text("What's
3+4?"))
                .accessibilityFocused($axFocus, equals: .textfield)
            Button("Continue"){ }
            .buttonStyle(.bordered)
            .tint(.blue)
            .font(.system(size: 18, weight: .bold,
design: .default))
            .foregroundColor(.blue)
            .accessibilityFocused($axFocus, equals: .button)
        }
        .onAppear {
            DispatchQueue.main.asyncAfter(deadline: .now() + 0.35) {
                self.axFocus = .textfield
            }
        }
    }
}
```

The key to getting the most out of this API[13] is to appreciate the context for what makes it so useful for accessibility users, VoiceOver in particular. After all, iOS will focus accessible elements automatically on our behalf and for free anyway - so what responsibilities fall on our end? To answer that, again - consider the context of users who are leveraging accessibility technologies.

Many times, these folks need to take extra steps to activate elements. If we can speed up their interactions for a low-cost action, we should. If you've read the chapter "The Switch Control", this information may be redundant to you. If you haven't, please check out the focus section in that chapter now.

---

13 And several other accessibility APIs, for that matter.

Regardless, if you need a refresher, go through your interface and ask yourself the following:

1. Is the action I'm considering to perform automatically "low cost" (i.e. it doesn't delete something, or go to a new screen, etc).
2. If so, if it happened every time - would that be helpful or obnoxious?
3. If the answer is, it's helpful - then it's a good choice to do that action automatically based on the focus state.

To get a feel for how the accessibility engine fires these off in UIKit, run the sample project on your device and navigate to `AccessibilityFocusFig1ViewController`. It works the same as the previous SwiftUI example. In it, you'll see a subclass of `UIView` which updates a text label according to the focus state it has:

```
// Xcode -> UIKit -> AccessibilityFig1ViewController.swift

// MARK: Accessibility Overrides
override func accessibilityElementDidLoseFocus() {
    focusLabel.text = "I lost\nfocus."
}

override func accessibilityElementDidBecomeFocused() {
    focusLabel.text = "I've become\nfocused."
}

override func accessibilityElementIsFocused() -> Bool {
    focusLabel.text = "I am\nfocused."
    return true
}
```

Tips

Leveraging Identifiers

The code you write to support switches and for VoiceOver users is usually the same. However, if you need to distinguish between the two during focus changes, UIKit vends identifiers, via `AssistiveTechnologyIdentifier`, representing the current method being used:

1. `.notificationSwitchControl`
2. `.notificationVoiceOver`

Since the function to retrieve the current identifier is housed within the `UIAccessibilityFocus` informal protocol, you can simply call it from any accessibility element, most commonly from the currently presented view controller:

```
// Xcode -> AccessibilityFocusFig2ViewController.swift
class AccessibilityFocusFig2ViewController: BaseSampleViewController
{

    override func viewDidLoad() {
        super.viewDidLoad()

        // Returns nil if not assisstive technology is active
        let identifier: String =
accessibilityAssistiveTechnologyFocusedIdentifiers()?.first.debugDes
cription ?? "Identifier Unavailable"
    }
}
```

Getting the Currently Focused Element

Should you need to query which element is currently focused, `UIAccessibility` has a static function which will return the currently focused, or most recently focused, element. Further, if you need to distinguish between something that was focused using a switch control or VoiceOver, you can pass in a `AssistiveTechnologyIdentifier` value that we just mentioned above.

For example, here's how you would get the focused element from VoiceOver:

```
// Xcode -> AccessibilityFocusFig2ViewController.swift
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)
    let currentVoiceOverFocus =
UIAccessibility.focusedElement(using:
.notificationVoiceOver)
    print("Current focus is on: \(String(describing:
currentVoiceOverFocus))")
}

/* That would yield something like the following:
Current focus is on: Optional(<UILabel: 0x106e0dae0; frame =
(134.667 442.667; 159 40.6667); text = 'Focus Identifier:
Identif...'; userInteractionEnabled = NO; layer = <_UILabelLayer:
0x280215cc0>>)
*/
```

The identifier isn't required, however. If you pass nil, you'll just get the most recently focused element regardless of whether it was focused from a switch control or VoiceOver.

Managing Multiple Focus Controls in SwiftUI

You might be wondering where the utility lies within the `equals:` parameter of the focus modifier. In short, it allows you to swap focus between controls in a trivial manner. As we saw in a previous SwiftUI focus code sample:

```
enum AXPurchaseFocusableElement: Hashable {
    case textfield
    case button
}
```

Any type conforming to `Hashable` can be used with the parameter, which means we can easily switch focus between controls using it:

```
// Set up property wrapper
@AccessibilityFocusState private var axFocus:
AXPurchaseFocusableElement?

// Use in the modifier
viewOne
    .accessibilityFocused($axFocus, equals: .textfield)
viewTwo
    .accessibilityFocused($axFocus, equals: .button)

// Programmatically change focus to button
axFocus = .button
```

Be sure to make your property an optional type, as SwiftUI will set it to `nil` once focus has left the element.

Specifying a Certain Accessibility Technology in SwiftUI

If you need to handle focus, but only for a specific accessibility technology – you can. The `AccessibilityFocusState` has an initializer specifically for this purpose:

```
public init(for technologies: AccessibilityTechnologies) where Value
== Bool
```

For example, if you're only concerned with toggling or querying focus for VoiceOver you can pass in the relevant `AccessibilityTechnologies` value for that technology:

```
@AccessibilityFocusState(for: .voiceOver) private var axFocus:
AXPurchaseFocusableElement?
```

Three Key Takeaways

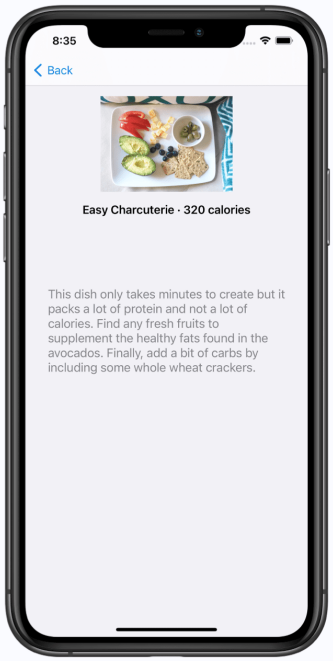
1. The concept of focus from an accessibility standpoint helps users navigate while setting clear expectations about what element will be acted upon if it were to be activated.
2. There are three different states you can react to from focus changes in your interface.
3. Look for ways to help navigation be more efficient by leveraging any of the focus states and performing actions automatically if it makes sense.

Reading Content

There may be times when you've created specialized views for the purpose of reading long-form text or similar content. In those cases, accessibility may not come as naturally as it might've with other interfaces in your app. If you find yourself dipping down into TextKit, CoreText or even vanilla UIKit for these kinds of experiences, you'll likely discover that VoiceOver reads out your content either all at once, not at all or out of order.

In these cases, it's time to adopt `UIAccessibilityReadingContent`. This protocol houses functions we can leverage to help the accessibility engine understand things like what line number is currently being read, what's at that line and more.

To demonstrate how this works, consider the interface below which shows a mock recipe journal:



It's got an image, a label with the name and caloric content and finally a longer form text view. This is all housed within a `UIView` subclass:

```
// Xcode -> MenuItemView.swift
class MenuItemView: UIView {
    let itemImageView: UIImageView = {
        let imgView = UIImageView(frame: .zero)
        imgView.contentMode = .scaleAspectFit
        imgView.translatesAutoresizingMaskIntoConstraints = false
        imgView.isAccessibilityElement = true
        imgView.accessibilityLabel = "Photo of delicious food."
        return imgView
    }()
    let subtitleLabel: UILabel = {
        let lbl = UILabel(frame: .zero)
        lbl.translatesAutoresizingMaskIntoConstraints = false
        lbl.font = UIFont.preferredFont(forTextStyle: .headline)
        return lbl
    }()
    let descriptionTextView: UITextView = {
```

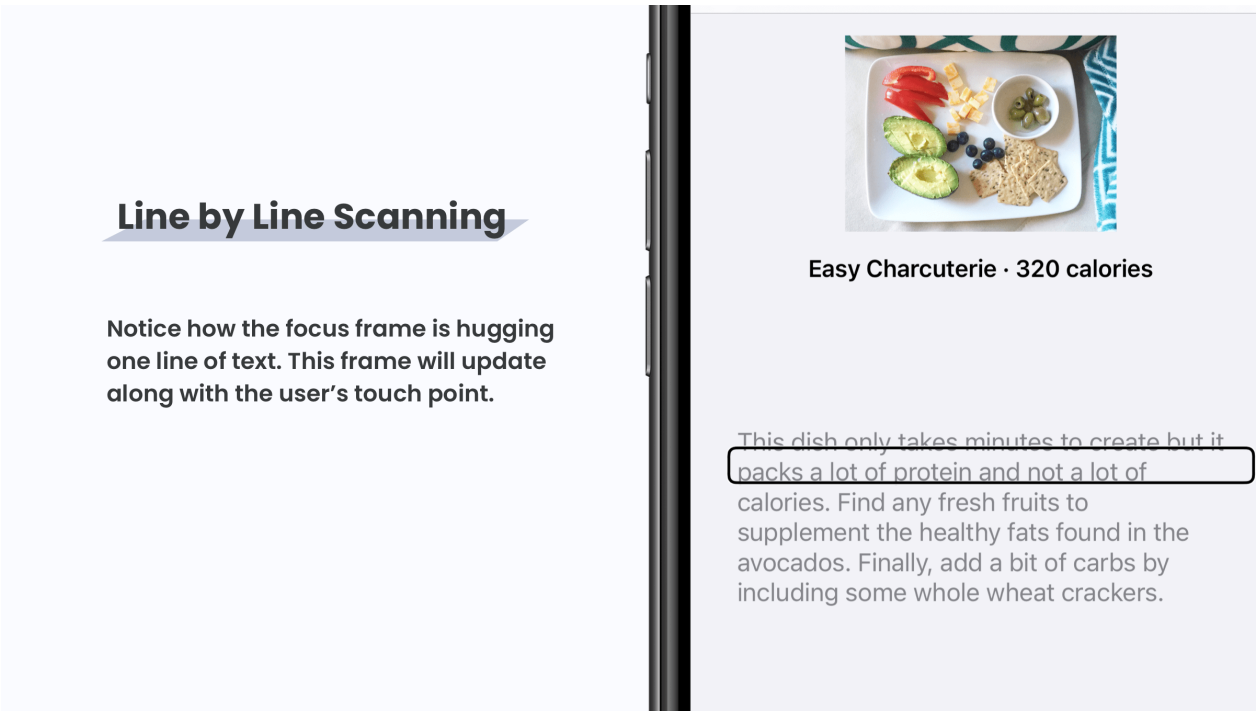
```
        let textView = UITextView(frame: .zero)
        textView.translatesAutoresizingMaskIntoConstraints = false
        textView.textColor = .secondaryLabel
        textView.backgroundColor = .clear
        textView.font = UIFont.systemFont(ofSize: 18)
        textView.isEditable = false
        return textView
    }()

    enum DisplayType: Int {
        case headerImage = 0
        case nameAndCaloriesLabel = 1
        case longFormDescription = 2
    }

    required init(withItem item:MenuItem) {
        itemImageView.image = item.image
        subtitleLabel.text = item.headerInfo
        descriptionTextView.text = item.longformInfo

        super.init(frame: .zero)
        translatesAutoresizingMaskIntoConstraints = false
        addSubview(itemImageView)
        addSubview(subtitleLabel)
        addSubview(descriptionTextView)

        // Layout constraints and more...
    }

}
```

If we run this with VoiceOver as is, we get…..nothing. VoiceOver will play its "ding" noise as the user attempts to navigate to signal there's nothing left in the view hierarchy that is visible to the accessibility engine. Ideally, here's what *should* happen:

1. The user could get all of the content read out at once.

2. Or, if the user wanted to scan the interface by scrubbing their finger up and down, we'd read out and focus *only* the text that's found on that line. This includes showing the black border around each individual line which VoiceOver uses to denote focus.

We can make all of that happen using the readable content protocol. With it implemented, we'd get an experience like this as the user scrubs their finger up and down (in addition to supporting all of the content being read out all at once, if that's what they want):



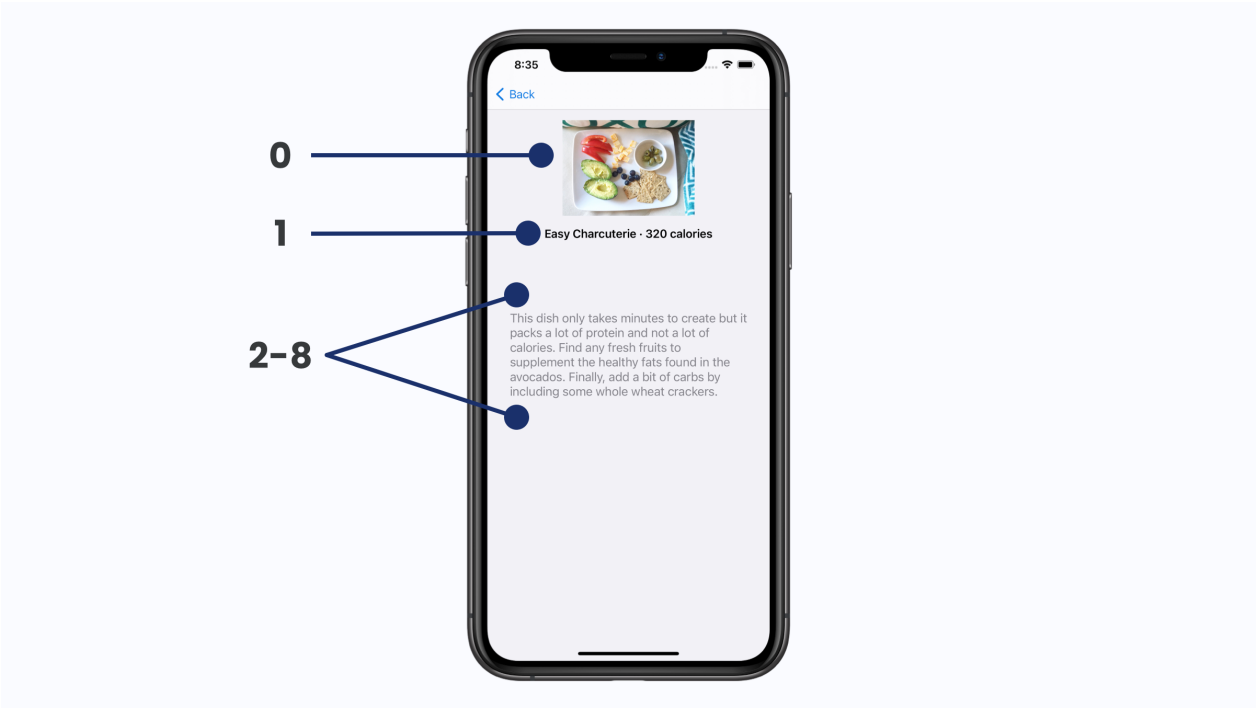
In the image above, the user's finger would be at that particular line in the text - and it would focus and follow each individual line in the text view as they moved or focus on the image or label above themselves. The readable content protocol gives us quite a lot of flexibility to tell the accessibility engine where things are and how they are laid out, but it does require some (sometimes) involved implementations.

How it Works

The protocol has four required functions :

1. `func accessibilityLineNumber(for point: CGPoint) -> Int`: This signifies what the line number should be at the touch point passed in.
2. `func accessibilityContent(forLineNumber lineNumber: Int) -> String?`: This represents the actual text that should represent the passed in line number.
3. `func accessibilityFrame(forLineNumber lineNumber: Int) -> CGRect`: This is the accessibility frame for the passed in line number, which will be used to draw the black border around the element or text.
4. `func accessibilityPageContent() -> String?`: This represents an aggregate of all of the line numbers content. You can think of this as the text for the entire "page" of content within the interface.

Before we go into the implementation, let's map out what the line numbers should look like within the view:



From top to bottom, each view represents a line to be read. The part that gets a little tricky is the text view, which itself has six lines. In most cases, I imagine, you may not have anything *but* a text view for these kinds of interfaces. However, having more views makes the implementation slightly more involved, so I've included a few more here to give you the tools to handle either case. Once you've implemented `UIAccessibilityReadingContent` in a view like this, going the other direction is trivial.

Let's check things out function by function, starting with defining a line number for a given point.

Determining Line Numbers

```
// Xcode -> MenuItemView.swift
func accessibilityLineNumber(for point: CGPoint) -> Int {
    guard let subviewAtPoint = subviews.filter({ subview -> Bool in
        return subview.frame.contains(point)
    }).first else {
        return NSNotFound
    }
```

```
    switch subviewAtPoint {
    case itemImageView:
        return DisplayType.headerImage.rawValue
    case subtitleLabel:
        return DisplayType.nameAndCaloriesLabel.rawValue
    case descriptionTextView:
        let lineNumberAtPoint = lineNumberInTextView(atTouchPoint:
point)
        guard lineNumberAtPoint != NSNotFound else {
            return DisplayType.longFormDescription.rawValue
        }
        return lineNumberAtPoint +
DisplayType.longFormDescription.rawValue
    default:
        return NSNotFound
    }
}
```

Here, we map a line number to a view. When we hit the text view, though, there's a little dancing around we've got to do. First, we need to convert the point to the text view's coordinates. Then, we determine the line number for that point. While the focus here is on how to implement the reading content protocol, if you're curious how this is done - look at `func textAndRectForLinesInTextView() -> [LineContent]` in `MenuItemView.swift`.

If we can get a line number for the text, we return it while being sure to add the text view's `DisplayType` value to it as well, since the first line in the text view *isn't* the first line in terms of the overall controller's line number. Again, this is why I've included multiple views in this example. It's a bit more to track and requires some bookkeeping, but you'll be equipped to handle these interfaces if you come across one.

One note here - why use subview diving here and not `hitTest` in the beginning? In short, it all depends on your view hierarchy. Hit testing returns the view which is the

furthest descendant of the receiver, and often times that ends up being the `MenuItemView` itself in our setup. We want the subviews of that view, hence this approach.

Vending Content at a Line Number

Next, we've got to say what the text is at a given line number:

```
// Xcode -> MenuItemView.swift
func accessibilityContent(forLineNumber lineNumber: Int) -> String?
{
    switch DisplayType(rawValue: lineNumber) {
    case .headerImage:
        return itemImageView.accessibilityLabel
    case .nameAndCaloriesLabel:
        return subtitleLabel.accessibilityLabel
    default:
        let translatedIndex =
translateLineNumberToTextViewLineNumber(lineNumber)
        let allTextDataAtLine = textAndRectForLinesInTextView()

        guard translatedIndex >= 0 && translatedIndex <=
allTextDataAtLine.count
else {
            return nil
        }

        let textAtLine = textForLineNumber(translatedIndex)
        return textAtLine
    }
}
```

For views which don't have multiple line numbers within them, this part is easy. In most cases, you simply return the view's `accessibilityLabel` and you're done. That's exactly how we handle it for the image and label views. Again, it's the text view where we've got to get our hands a bit dirty. The reason it's a little complicated is that the line

number we're given is what I call the *global line* number. It's a line number that's holistically represents the entire view's interface.

As such, we need to offset it to the local number of line numbers in the text view. To simplify this, every text view in this example will have six lines. If the global line number we get is 3, then what we need to return is the *first* line number in the text view. For situations like this (which can occur when implementing datasources of collection and table views, more commonly), I find it important to write a specific function to call out *why* this is happening and document the logic:

```
// Xcode -> MenuItemView.swift
private func translateLineNumberToTextViewLineNumber(_
globalLineNumber: Int) -> Int
{
    // The accessibility engine sees the line numbers starting with
the image and then the name
    // And calorie label, those being line 0 and 1. Each line in the
text view then represents a
    // Specific line number. So for those to map to the array of
lines in the text view (there are six)
    // We need to minus the text view's line number (2) and then
offset it from the global line number.
    // Then, we're at the correct spot in the line number array.
    let translatedIndex = globalLineNumber -
DisplayType.longFormDescription.rawValue
    return translatedIndex
```

After we've done this, we have the correct string to return at a global line number translated to the text view's line number.

Determining Line Number Accessibility Frames

Next, to get the focus frames to show around each individual line of text, we need to vend accessibility frames:

```
// Xcode -> MenuItemView.swift
```

```
func accessibilityFrame(forLineNumber lineNumber: Int) -> CGRect {
    switch DisplayType(rawValue: lineNumber) {
    case .headerImage:
        return itemImageView.accessibilityFrame
    case .nameAndCaloriesLabel:
        return subtitleLabel.accessibilityFrame
    default:
        let allTextDataAtLine = textAndRectForLinesInTextView()
        let translatedIndex =
translateLineNumberToTextViewLineNumber(lineNumber)

        guard translatedIndex >= 0 && translatedIndex <=
allTextDataAtLine.count
else {
            return .zero
        }

        let textDataAtLine = allTextDataAtLine[translatedIndex]
        let rectToScreen =
descriptionTextView.convert(textDataAtLine.textRect, to:
nil)
        return rectToScreen
    }
}
```

Much like we simply returned the `accessibilityLabel` for views without multiple lines of text above, we do the same here only we use their `accessibilityFrame` properties.

Stop me if you've heard this before in this chapter, but yes - the text view is the tricky part once more. We don't want to return a frame for the entire text view, because scrubbing through the content line by line wouldn't match the frame that shows as focused. We need the focus frame to show line by line. Again, it's likely that you'll need to convert the point from the text view coordinates to the window's coordinates,

and an easy way to do that is to simply pass `nil` in any point or rect based functions which allow for an optional view parameter:

```
descriptionTextView.convert(textDataAtLine.textRect, to:
nil)
```

Now, we can draw the focus frame around each individual line of text within the text view, which is critical for VoiceOver users to keep their context centered as they scrub through your view.

Vending Page Content

To round things out, how about an easy one? The last function we need to handle is one which simply reads out *all* of the content if the user simply wants everything spoken entirely in VoiceOver, which is often the case:

```
// Xcode -> MenuItemView.swift
func accessibilityPageContent() -> String? {
    return itemImageView.accessibilityLabel! + subtitleLabel.text! +
descriptionTextView.text
}
```

In this function, it's simply a matter of concatenating a string with all of the text that should be spoken aloud - just be sure to return it in the right order. If we switched things up here ordering wise, then the experience from reading it all at once wouldn't match the experience they would get from scrubbing through the views manually, which is likely not what you want.

Tips

Automatic Page Turning

In interfaces such as this, it might make sense to automatically go to the next page or progress to the next controller of content once all of that content has been read out. For that, you'd turn to an accessibility function that handles doing so automatically, `accessibilityScroll(_ direction: UIAccessibilityScrollDirection) -> Bool`.

When this is invoked, you either page forward or backward depending on the direction that's passed in. This works similar to how we created custom rotors in the chapter "The Rotor Control" and its `searchDirection` property on the predicates used to create custom rotors.

This does work in tandem with a few more things, however. Specifically, your accessibility element which overrides this needs to have an accessibility trait that tells the accessibility engine it's responsible for doing so, `.causesPageTurn`. When a scroll occurs a special notification, `UIAccessibilityPageScrolledNotification`, is posted by us to the accessibility engine which kicks off the "page turn".

We're going to look at this in more detail in the chapter "Navigation and Interactions".

Attributed Strings

While `UIAccessibilityReadingContent` does have four required functions, it does have two *optional* functions that can come in handy. Those are:

1. `accessibilityAttributedPageContent() -> NSAttributedString?`: Use an attributed string for the entire page's content.

2. `accessibilityAttributedContent(forLineNumber lineNumber: Int) -> NSAttributedString?`: Use an attributed string for the particular line number.

If these are used, they'll supersede any implementations you've got for the non-attributed string versions. So, why would you want to leverage an attributed string in these cases?

Simply put, to add more specificity to the way VoiceOver announces text. Consider a book that's being represented on screen, a novel from a Spanish author who will, naturally, have Spanish words in the text. Instead of just reading them as is, we could use an attributed string to have them announced as they were intended.

For example, consider a string which contains the namer of a Spanish actor:

```
NSAttributedString(string: "Pedro Almodóvar", attributes:
[.accessibilitySpeechLanguage:"es-MX"])
```

Or, perhaps if your text had a code sample where punctuation should be read literally:

```
NSAttributedString(string: "let var = Baz()", attributes:
[.accessibilitySpeechPunctuation:NSNumber(integerLiteral: 1)])
```

There are several predefined keys under `UIAccessibilityConstants` which you can use for attributed strings to make VoiceOver speak out the content with more tact and accuracy. For reference, here are a few other useful values to consider:

1. `accessibilitySpeechPitch`: Given an `NSNumber` value from 0 to 2 which corresponds to the pitch you want. The default is 1.0. This could be useful to emphasize an action with a higher pitch, for example.
2. `accessibilitySpeechQueueAnnouncement`: Given an `NSNumber` value representing a boolean, any announcement you're posting will be queued

up to play when VoiceOver is done with its current speech, if any. However, if you wish to interrupt the current speech and announce anyways, you can pass a true value in your `NSNumber` when using this, `NSNumber(booleanLiteral: true)`.

3. `accessibilitySpeechIPANotation`: Useful for pronouncing a word or phrase containing International Phonetic Alphabet symbols. The key is the string which contains the symbols.
4. `accessibilitySpeechSpellOut`: Given as `NSNumber` value representing a boolean that's true, VoiceOver will read out each individual letter in the string.

These keys, combined with ranges of a string, give you a lot of flexibility to control how text is spoken.

Use an Enum to Represent View Ordering

A tip I caught from a WWDC session from 2019 covering this protocol had a handy tip I've used when implementing this. It's simply using an enumeration to represent line numbers mapped to views in your hierarchy. You might've noticed it above in the `MenuItemView`:

```
// Xcode -> MenuItemView.swift
enum DisplayType: Int {
    case headerImage = 0
    case nameAndCaloriesLabel = 1
    case longFormDescription = 2
}
```

This helps avoid magic numbers when implementing many of the functions that require some sort of line number logic in many of the required functions in the `UIAc-`

`cessibilityReadingContent` protocol. For example, `accessibilityContent(forLineNumber lineNumber: Int) -> String?` has you querying the data at a particular line. Instead of doing something like this:

```
switch lineNumber {
case 0:
case 1:
// etc
}
```

Opt for readability by having the line number represent the view that's actually there:

```
switch lineNumber {
case ThisView:
case ThatView:
// etc
}
```

Opt for a Lean View Hierarchy Where Possible

On the other hand, many times you can get away with having one view to represent your text, especially if your long form content doesn't require rich media or any imagery. If you can fit all of the content into something like a text view that's customized using TextKit, these implementations get a lot easier.

This won't always be possible, but you'd be surprised with how much is customizable when dipping down into TextKit. Creating an experience similar to what you'd find in Books for iOS is not only possible, but makes managing `UIAccessibilityReadingContent` both necessary and a bit easier since it essentially becomes a matter of tracking the text and frames at a particular line number - but only in one view.

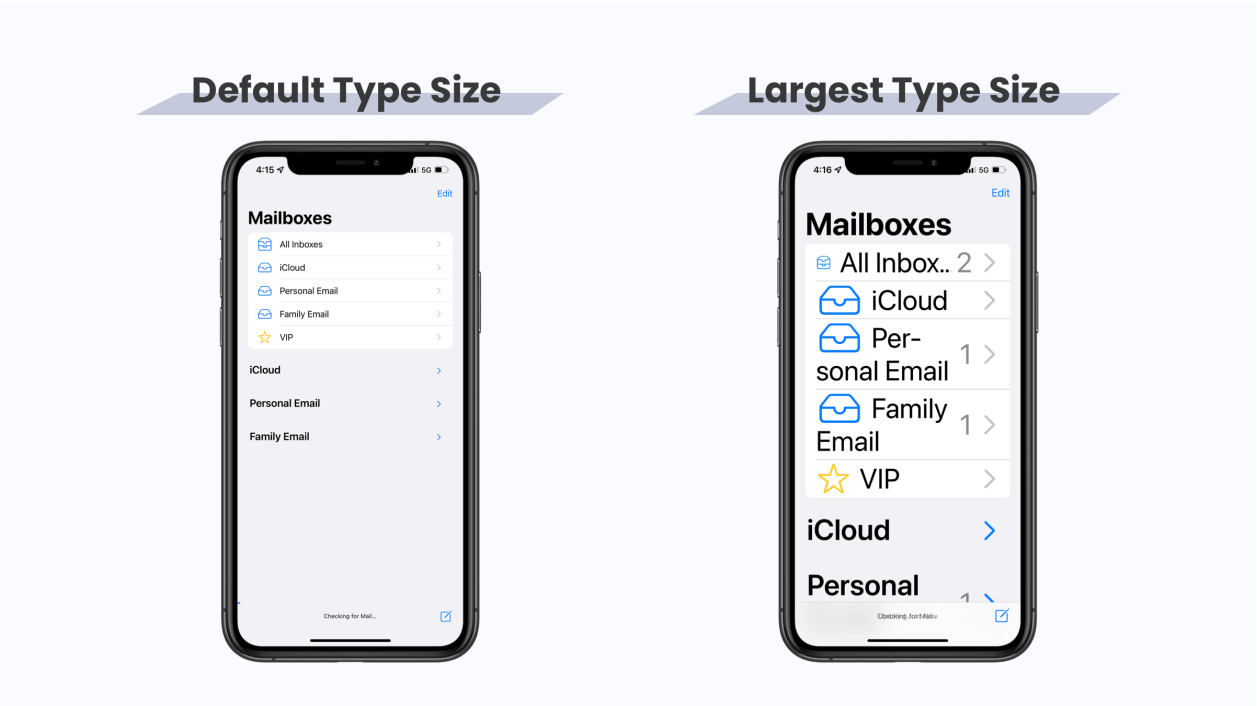
Three Key Takeaways

1. Customized reading experiences or long form content often don't work "out of the box" very well with the accessibility engine.
2. In these cases, we need to define specific line numbers, and what content is at each one, to help out VoiceOver.
3. If you have long form text content which could have words from several different languages or other specialized content, leverage attributed strings to have them spoken aloud accurately.

Image Adjustments

I'd bet that many iOS developers are familiar with the Dynamic Type APIs. For the uninitiated, it's how we provide support in our apps for a scaling font size. The idea is that as the user adjusts their font size settings, developers are given new content size categories (found within `UITraitCollection`) and labels, text views or text fields map a `UITextStyle` to them.

Here, you can see a label with the the default text size and the largest text size you can use:



Dynamic Text scales font sizes along with user preferences for their selected content size categories.

We'll chat a lot more about Dynamic Type in its own chapter, but the concept of adjusting content within interfaces extends beyond just text and font sizes. The `UIAccessibilityContentSizeCategoryImageAdjusting` protocol affords us the same opportunity to do the same thing with imagery. It's incredibly simple in terms of adoption, in fact - it only has one property:

```
var adjustsImageSizeForAccessibilityContentSizeCategory: Bool
```

But, like several other protocols within the iOS ecosystem, it's not necessarily meant for developers such as us to adopt directly. Instead, the system has done it for you in a trio of controls. If you view the header file, you'll see that `NSTextAttachment`[14],

[14] This is a macOS control, by the way.

`UIButton` and `UIImageView` all have been extended to adopt it (warning - Objective-C code on the way!):

```
@interface UIImageView
(UIAccessibilityContentSizeCategoryImageAdjusting)
<UIAccessibilityContentSizeCategoryImageAdjusting>

@end

@interface UIButton
(UIAccessibilityContentSizeCategoryImageAdjusting)
<UIAccessibilityContentSizeCategoryImageAdjusting>

@end

@interface NSTextAttachment
(UIAccessibilityContentSizeCategoryImageAdjusting)
<UIAccessibilityContentSizeCategoryImageAdjusting>

@end
```

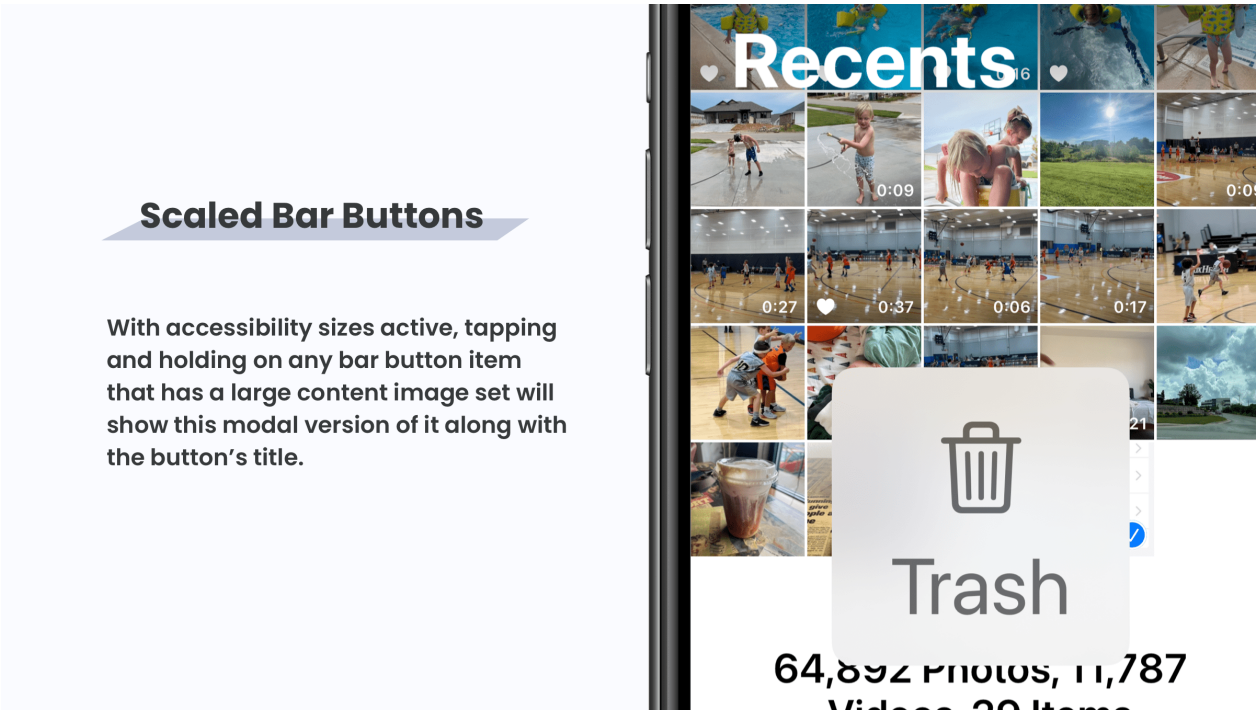
When the content size is changed, these controls will update their image to resize accordingly. Though, I should say that this protocol's name is slightly misleading. The controls will only update imagery when the content size category is within the *accessibility* size ranges or it falls outside of them, and that can be checked with the trait collection's `isAccessibilityCategory` boolean property, which itself hangs off the `preferredContentSizeCategory` instance:

```
 if
traitCollection.preferredContentSizeCategory.isAccessibilityCategory
{ // Image will be resized }
```

So, why do this?

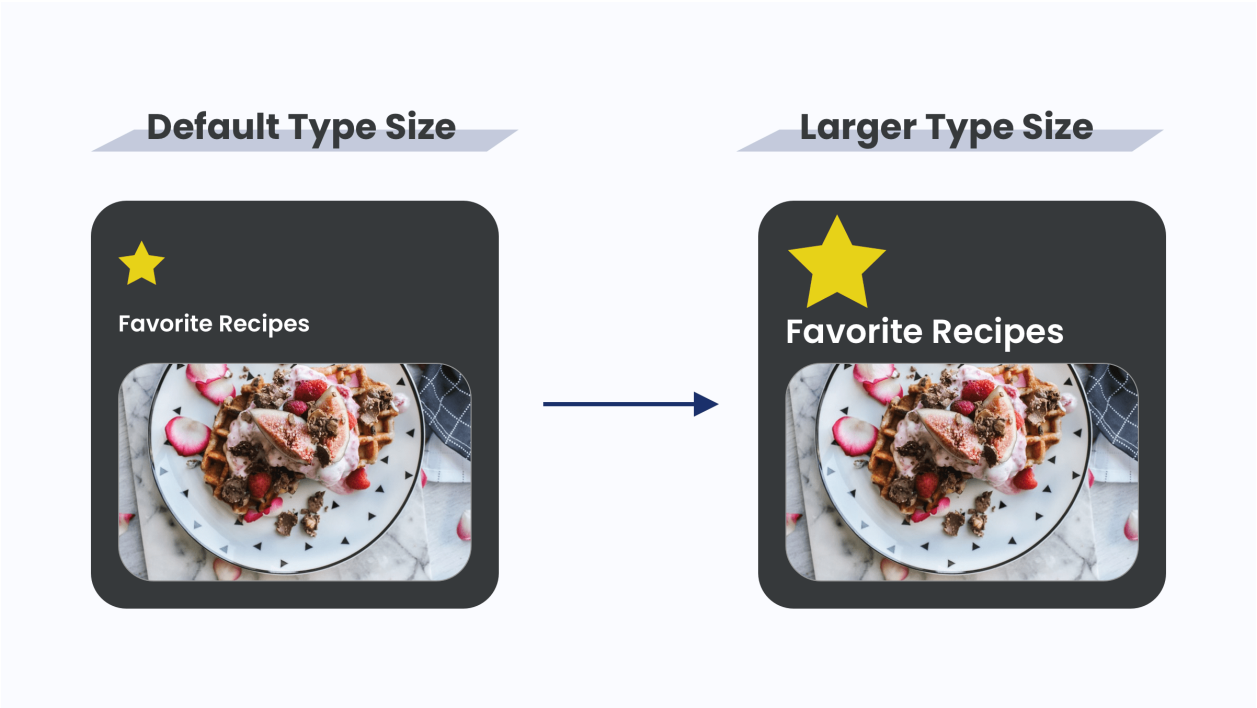
Simply put, it benefits those who need a bit of visual aid in their apps. For the same reasons adjustable font sizes exist to help with visual impairments by vending larger text, several parts of imagery should scale with it too. Think symbols in your interface, or glyphs that denote some sort of functionality or promote direction in your user experience.

You can see this idea at work in bar button items and its `largeContentSizeImage` property. When a large accessibility content size is active and a user taps and hold on a bar button item, they'll get this nice modal which shows the glyph at a larger size:



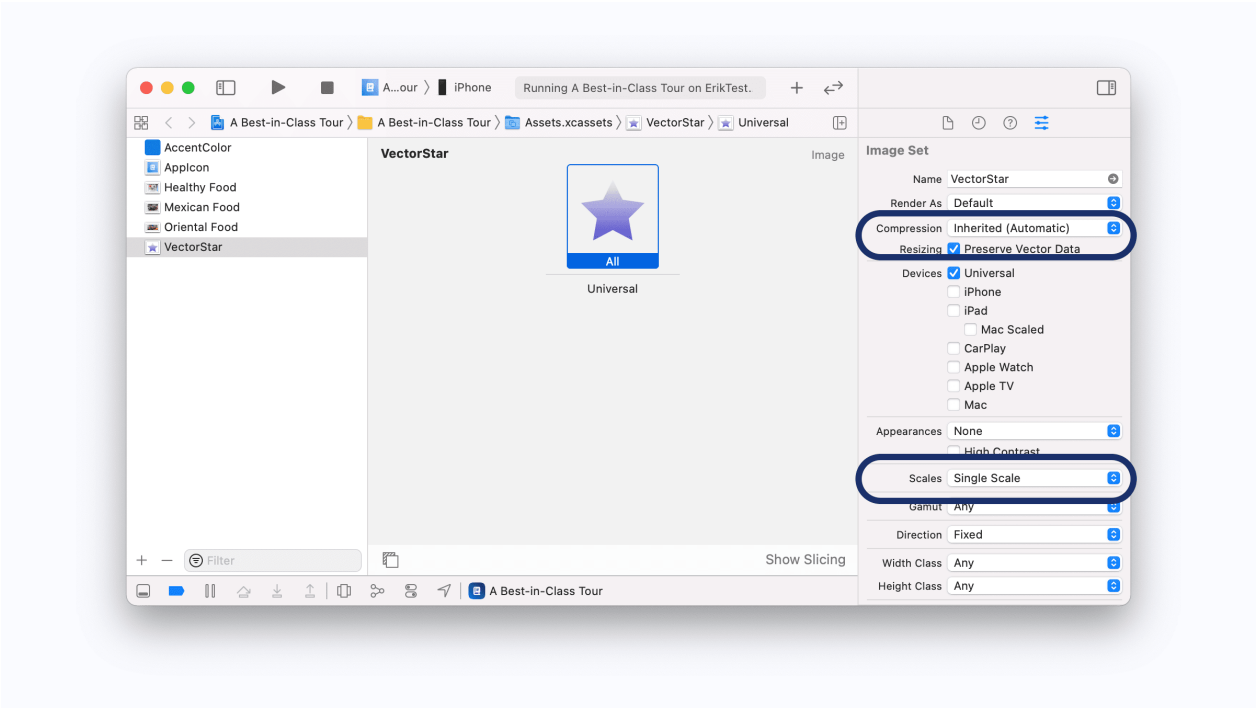
UIBarButtonItem has a property to allow for large image scaling, activated alongside larger content size categories.

Now, consider this interface below which leverages adjusting image sizes to get a similar effect:



Notice how not only the label, but the star icon, has scaled up.

Because of its resizing nature, you'll want to supply the image asset as a vector asset. This ensures smooth image scaling from the system. There are many workflows out there to get a vector asset in your app, but here's how it typically looks. First, you need an asset exported in a format that'll have vector data such as a .pdf asset. Then, it gets added to the target's asset catalog and "Preserve vector data" gets checked and with a single size set:



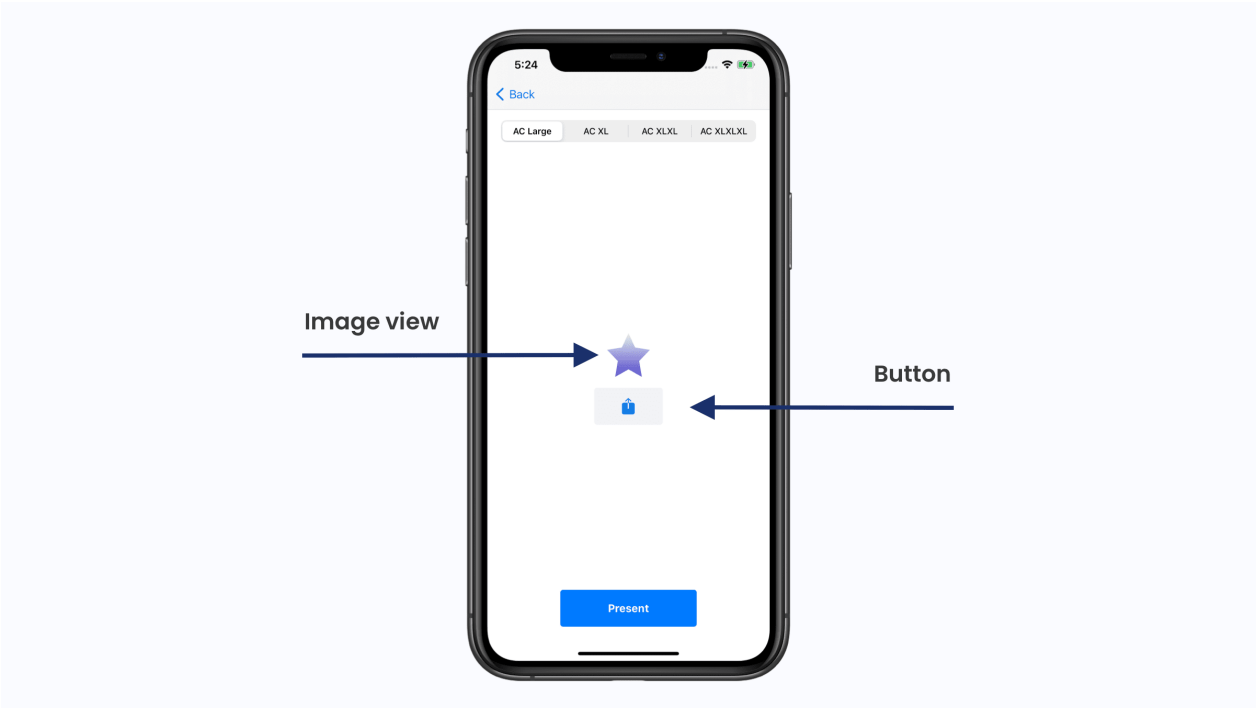
Using these kinds of assets will work with image adjustments.

How it Works

So long as you are using scalable artwork for your images and iconography and you've got `adjustsImageSizeForAccessibilityContentSizeCategory` set to true, you're all set with this API. That said, there are a few things that are good to know.

For Image Views

Image views will increase their intrinsic content size when adjusting their image's size. To see this in practice, open up `ImageAdjustingFig1ViewController.swift` and play around with increasing the content size:



The button and image view will scale their respective images up along with the selected content size category.

You can see how, the larger the accessibility content size that's chosen, the larger the image in the image view (and the image contained within the button) will be. This also assumes the content mode is set to something that allows for scaling, though its default setting does. If you aren't getting results, this is a good first place to check.

For Buttons

For buttons, it's the same story. The only differences are which image gets scaled, and the insets that are applied to it. For `UIButton` that means any image set on it via `func setImage(_ image: UIImage?, for state: UIControl.State)` will be scaled. If you set a background image, however, that's left unchanged:

```
// This would scale
demoButton.setImage(UIImage(systemName:"square.and.arrow.up.fill"),
for: .normal)

// This wouldn't
```

```
demoButton.setBackgroundImage(UIImage(imageNamed:"btnBackground"),
for:
.normal)
```

Further, if you set insets on the image - those will remain in place:

```
demoButton.imageEdgeInsets = UIEdgeInsets(top: 8, left: 0, bottom:
-8, right: 0)
```

If the image scales up, you'll still see those insets applied.

Tips

Custom Implementations?

You may look at this protocol and think, "I'm gonna adopt this on my custom control!" While it may be a fun exercise in reacting to a trait collection's content preferences, rescaling elements or scaling them back - it helps to go back to the question you should nearly always ask yourself before coding something, "Why?"

And our "why" here is largely answered elsewhere in the form of other available accessibility APIs. The reason this protocol isn't really meant for public adoption, and is instead intended for us to just toggle off and on with baked in controls, is that we already have the means to scale and adapt things with other APIs. This protocol is all in the name, *image adjusting*, so don't try and piggyback off of it for scaling other elements. An API exists for that purpose, `UIFontMetrics`, and we'll look at it in the "Dynamic Type" chapter.

Constraints

Your Auto Layout constraints play a role here, and if you've got them set to a specific height and width then this protocol won't interfere with them. For example, in our sample code above, if we had set the constraints like so:

```
imgView.widthAnchor.constraint(equalToConstant: 64).isActive = true
imgView.heightAnchor.constraint(equalToConstant: 64).isActive = true
```

…then your image view is going to stay at 64 points by 64 points, even when the content size category is changed to an accessibility size setting.

Instead, opt for inequalities for your width and height constraints. These allow the layout system to grow as needed so the image can scale along with the content size category the user has chosen. For a refresher, here's how it looks in our example from above:

```
// Xcode -> ImageAdjustingFig1ViewController.swift
imgView.widthAnchor.constraint(greaterThanOrEqualToConstant:
64).isActive = true
imgView.heightAnchor.constraint(greaterThanOrEqualToConstant:
64).isActive = true
```

Should You Scale?

Keep in mind that this protocol is meant for glyphs and icons primarily. If the imagery in question helps a user navigate or orientate themselves within your interface, then it it's a candidate to use with this technique.

Even then, though, the fit may not always be right. There aren't always cut and dry rules of when to use some accessibility technologies, and this is one of those times. Here's how I think about it:

1. Does Apple scale similar glyphs or icons in their own apps? If so, I should probably follow their example if I have a similar parallel in my own apps.
2. If a user couldn't see this icon very well if it didn't scale, *and* it signifies how something is done or where a tap might lead, then I should probably adopt this.

Three Key Takeaways

1. Adjusting images to scale with content size preferences aids those with visual impairments.
2. The majority of the work is done for you by the system on buttons and image views.
3. However, be sure to supply vector assets to ensure smooth scaling occurs.

Textual Context

Text, and the way it's meant to be read and understood, comes in so many varieties it's a wonder the human brain can keep up. Many of us likely take the vast world of grammar for granted when reading, not paying attention to all of the context switching we do automatically as we process the words we see.

If you saw a code snippet in a book like this, then you know it's not meant to be read out loud literally, like a novel might:

```
// You process source code differently than you would...
let example: String = "Hello, World!"

// ...words and text like this
Hello World! Welcome to W.W.D.C. - our best one yet!
```

In these cases, the only way those two things are alike are the building blocks which make them up. Characters, letters, punctuation – all of them work together to form a

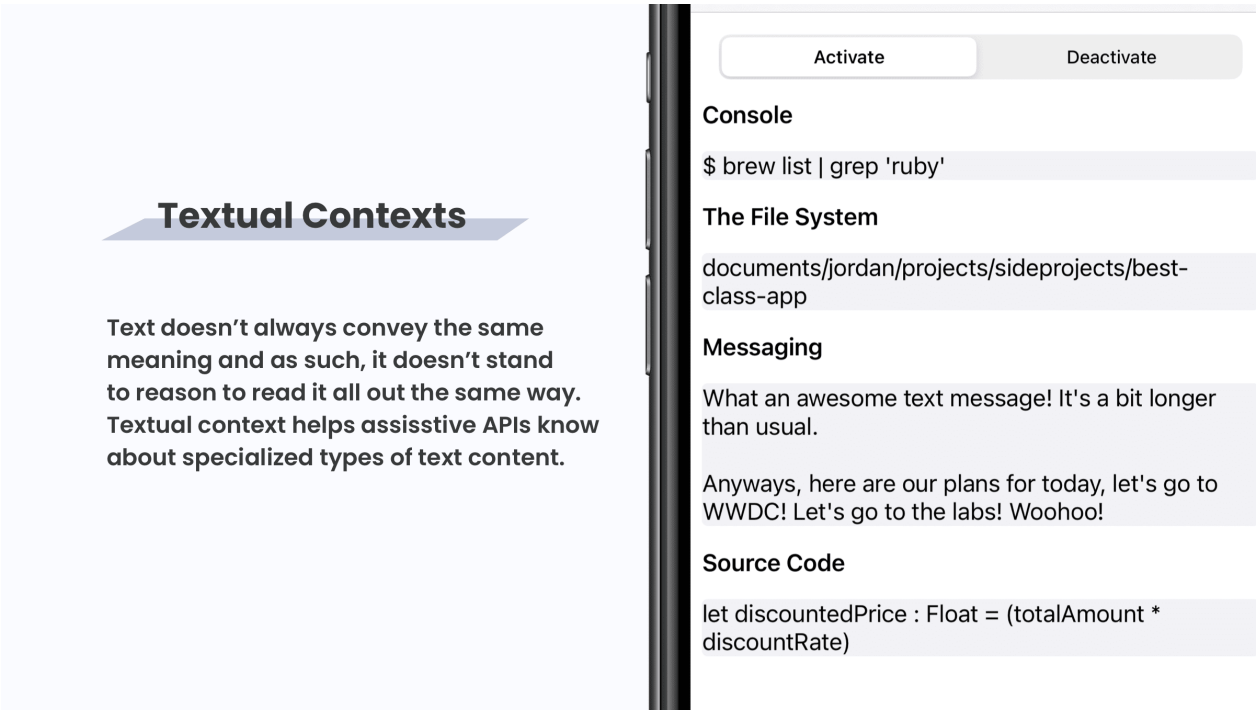
specific context. In our apps, we might have the same variety of strings. In those cases, we can give VoiceOver accurate context about those strings too, just like we do ourselves when we read.

This is what the `accessibilityTextualContext` property is for, found within `UIAccessibility`. In SwiftUI, the same idea exists in the `AccessibilityTextContentType` struct. Their purpose is to give text that doesn't quite work being read verbatim the correct context to VoiceOver so that it will be read correctly.

Following our code sample above, with the correct textual context VoiceOver may choose to speak all punctuation in the text – which is what users likely would want for source code.

In UIKit, it's an extension off of `NSObject`, so any standard UIKit control (or any custom one, which will likely ultimately inherit from `NSObject` anyways) has the ability for this to be set, so it's meant to work outside of just text controls as well. In SwiftUI, you'll be leveraging a modifier with all of the same benefits as UIKit, since its `View` construct is more abstract and flexible.

In the Xcode project (find it in `TextualContextFig1ViewController.swift` or `TextualContextFig1View.swift`), there is a view which contains four different types of text controls. Each one would be read by you differently either out loud or in your head, so they are each given the current textual context so VoiceOver can do the same:



To compare the differences, use the toggle at the top while running the demo project to see how VoiceOver would read them aloud differently without a textual context set.

There are seven ways to flag this kind of text to the accessibility engine in UIKit:

- `static let console: UIAccessibilityTextualContext`: Text that would be typed, or generated, from a console setting such as Terminal.
- `static let fileSystem: UIAccessibilityTextualContext`: Text that represents things on a file system, like a file path for an application on macOS for example.
- `static let messaging: UIAccessibilityTextualContext`: Text that belongs to some sort of messaging app or scenario, like Messages on an iPhone.
- `static let narrative: UIAccessibilityTextualContext`: Narrative text, such as a book or a play.

- `static let sourceCode: UIAccessibilityTextualContext`: Text that's raw source code, like `.swift` files you might open in Xcode.
- `static let spreadsheet: UIAccessibilityTextualContext`: Text that belongs to a spreadsheet paradigm, like text or header cells in the Numbers app.
- `static let wordProcessing: UIAccessibilityTextualContext`: Text that represents a word-processing document, like something you might type in a new document in the Pages app.

Whereas in SwiftUI, those same seven values exist — but it also adds in the `.plain` option as well:

- `public static let plain: AccessibilityTextContentType`: A type that represents generic text that has no specific type.
- `public static let console: AccessibilityTextContentType`: A type that represents text used for input, like in the Terminal app.
- `public static let fileSystem: AccessibilityTextContentType`: A type that represents text used by a file browser, like in the Finder app in macOS.
- `public static let messaging: AccessibilityTextContentType`: A type that represents text used in a message, like in the Messages app.
- `public static let narrative: AccessibilityTextContentType`: A type that represents text used in a story or poem, like in the Books app.

- `public static let sourceCode: AccessibilityTextContentType`: A type that represents text used in source code, like in Swift Playgrounds.
- `public static let spreadsheet: AccessibilityTextContentType`: A type that represents text used in a grid of data, like in the Numbers app.
- `public static let wordProcessing: AccessibilityTextContentType`: A type that represents text used in a document, like in the Pages app.

How it Works

There are two ways to mark textual context. First, simply assign to the property itself in UIKit and in SwiftUI use the corresponding modifier:

```
// UIKit
mySourceCodeContainer.accessibilityTextualContext = .sourceCode

// SwiftUI
var body: some View {
    Text("Hi")
        .accessibilityTextContentType(.plain)
}
```

Again, since `NSObject` has an extension adding the `accessibilityTextualContext` property and SwiftUI uses a modifier, you can set a textual context on virtually any interface element you can think of across the two interface frameworks:

```
// UIKit
let messagingLabel = UILabel(frame: .zero)
```

```
messagingLabel.accessibilityTextualContext = .messaging

let pathsTextView = UITextView()
pathsTextView.accessibilityTextualContext = .fileSystem

let terminalOutput = CustomTerminalView()
terminalOutput.accessibilityTextualContext = .console

// SwiftUI
var body: some View {
    MessagingTextView(message: messageModel)
        .accessibilityTextContentType(.messaging)
}

var body: some View {
    LocalPathView()
        .accessibilityTextContentType(.fileSystem)
}

var body: some View {
    TerminalEditorView("Hi")
        .accessibilityTextContentType(.console)
}
```

The second way is to leverage keys for attributed strings via `UIAccessibilityTextAttributeContext` when using UIKit or `AttributedString`'s `accessibilityTextualContext` property in SwiftUI. So long as you use that for the key in the attributes, you can choose any of the textual contexts mentioned above:

```
// UIKit
let context =
[.UIAccessibilityTextAttributeContext:UIAccessibilityTextualContext.
sourceCode])

label.attributedText = NSAttributedString(string: demoText,
attributes: context]
```

```
// SwiftUI
var attString = AttributedString(demoText)
attString.accessibilityTextualContext = .sourceCode
```

Of course, this method deals exclusively with controls who have some sort of attributed text to set out of the box, such as `UILabel`, `UITextView` and `UITextField`. The SwiftUI variant follows much of the same rules, since you will end up using an attributed string within a text view of some sort as well.

Tips

Substrings

Occasionally the full string may not represent the same types of context. For example, imagine if I were to package up this book's content and ship it in my own custom "Best-in-Class iOS" app. To create a custom reading experience, some of the text in a string might be regular words and some might be source code.

In cases like this, common in long form text controls such as `UITextView` or TextKit custom controls, you can mark certain ranges to have particular textual context types:

```
// UIKit -> Xcode -> TextualContextFig2ViewController.swift

let sourceCode: String = "let foo: Bar = Bar()"
let text = """
    Here is the first code sample to try out:
    // Swift
    \(sourceCode)
    As you can see, Bar() gets initialized.
"""
```

```
let sourceRange = (text as NSString).range(of: sourceCode)
let attributedText = NSMutableAttributedString(string: text,
attributes: [.font:UIFont.preferredFont(forTextStyle: .subheadline)

// Now, just mark the source code substring with the .sourceCode
// Textual content attribute.
attributedText.addAttributes([.UIAccessibilityTextAttributeContext:U
IAccessibilityTextualContext.sourceCode], range: sourceRange

textView.attributedText = attributedText
```

In SwiftUI and it's nascent `AttributedString` type, it's even easier:

```
// SwiftUI -> Xcode -> TextualContextFig2View.swift
struct TextualContextFig2View: View {
    private let sourceCode: String = "let foo: Bar = Bar()"
    private var text: String

    // MARK: Initializer

    init() {
        self.text = """
        Here is the first code sample to try out:

        // Swift
        \(sourceCode)

        As you can see, Bar() gets initialized.
        """
    }

    var body: some View {
        Text(generateAttributedString())
    }

    private func generateAttributedString() -> AttributedString {
        var attString = AttributedString(text)
        guard let sourceCodeRange = attString.range(of: sourceCode)
else {
```

```
            return attString
        }
        attString[sourceCodeRange].accessibilityTextualContext
= .sourceCode

        return attString
    }
}
```

This flexibility means it's not necessary to use multiple controls for the same body of text just to gain differing textual context types.

Nullability in UIKit

The `accessibilityTextualContext` property is nullable, so should your text content change and the current context isn't applicable, simply nil it out:

```
myTextView.accessibilityTextualContext = nil
```

By default, this property's value is nil. While this may not come up often, it does come in handy in situations where text could change, page or update from a server and is refreshed within the same control. In these cases, be sure to set a sensible value for the textual content - but *also* remember that if the text doesn't fall within the seven types listed here, then the right call is to set it to nil.

If you find yourself in a similar situation using SwiftUI, simply use the `.plain` value to accomplish the same thing.

Containers

Remember, controls outside of text elements can use this property. Assistive APIs can all benefit from it, whether it effects navigation or VoiceOver. So, if you've got a con-

tainer or custom control which *houses* text that falls into one of these contexts, be sure to set it on the container itself, too:

```
// UIKit
mySourceCodeContainer.accessibilityTextualContext = .sourceCode

// SwiftUI
MySourceCodeContainer()
    .accessibilityTextContentType(.sourceCode)
```

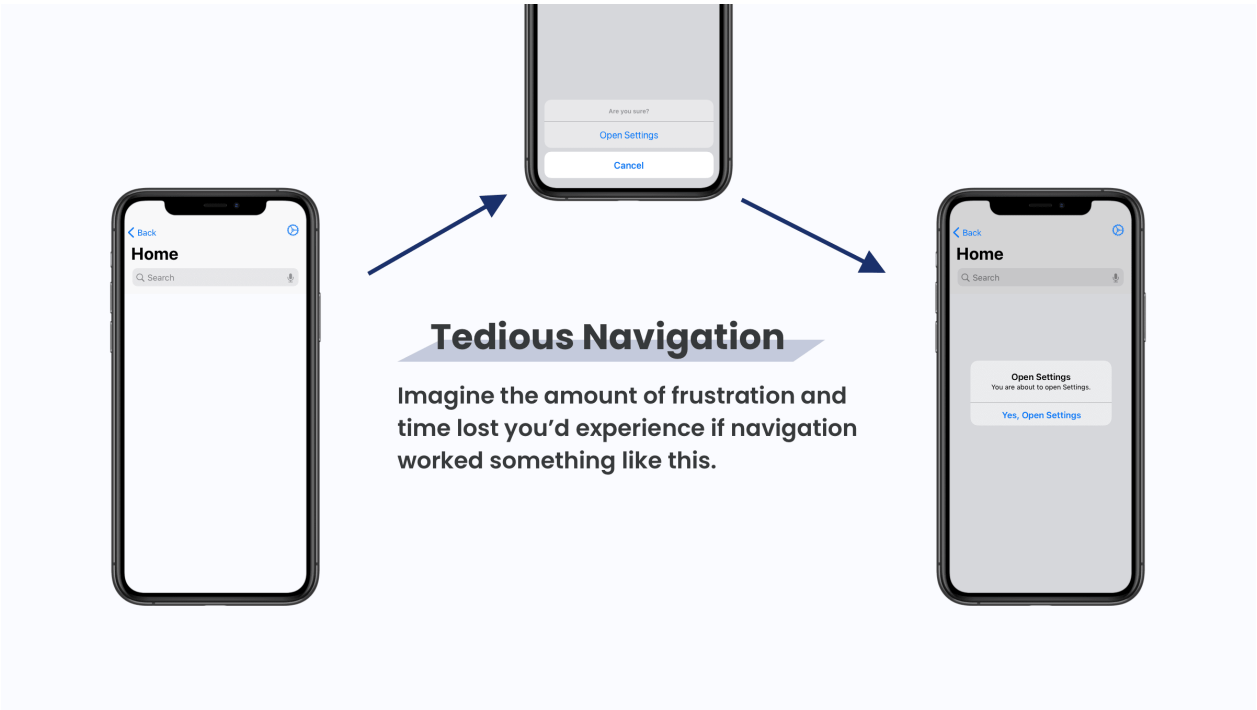
Three Key Takeaways

1. Textual contexts help describe the type of text that's displayed.
2. Certain types of text don't read out very well by default, so supplying the right context will help VoiceOver say it correctly.
3. You can mark substrings within a string to have a certain context as well.

Navigation and Interactions

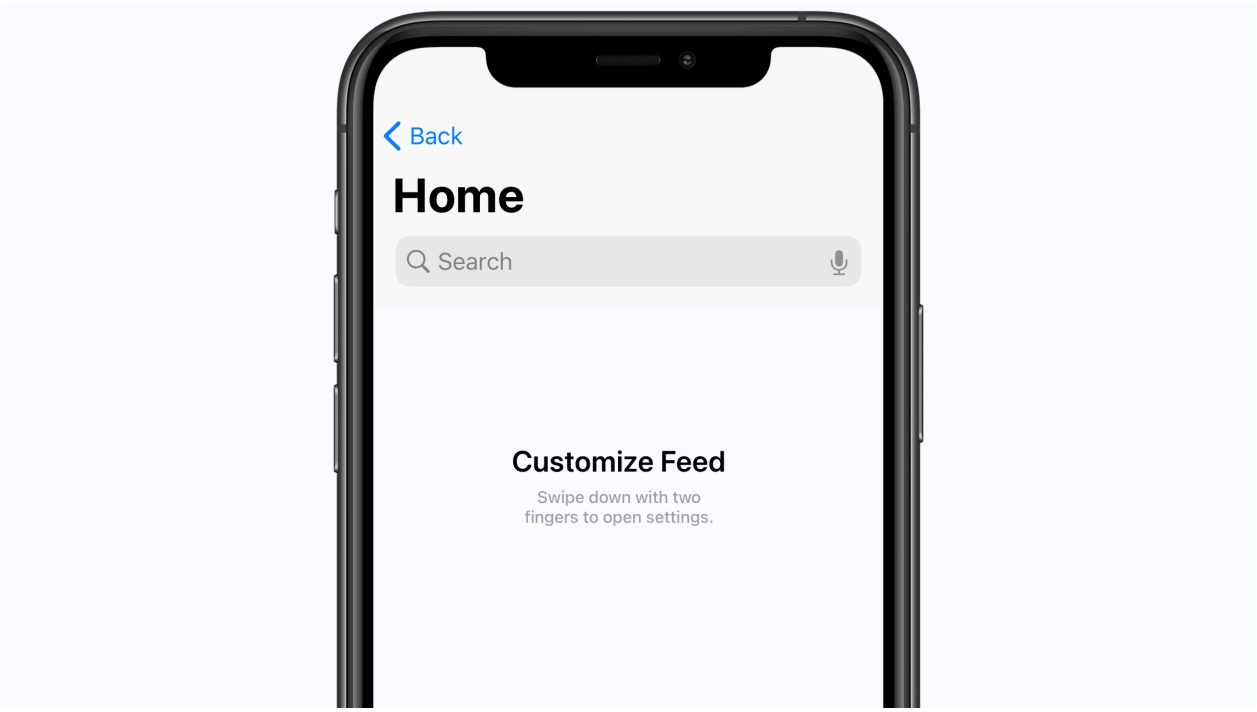
It's easy to take tactful navigation for granted. When it works, it plays the role of an unsung hero — you don't even think about it, even though it's uplifting the app's whole user experience. The other side of that, though, is when it's bad it's *awful*. It stands out. It's nearly all you can see.

Imagine if opening up a settings option worked like this:



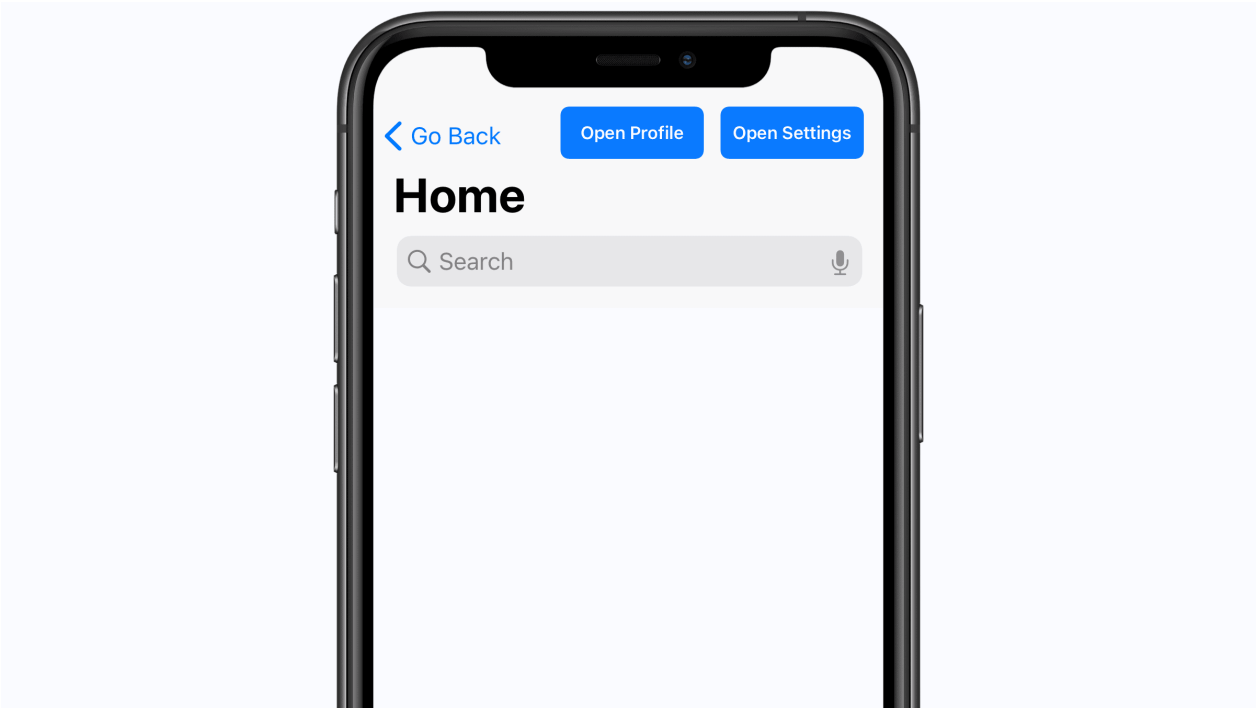
Imagine if navigating from view to view took too many needless steps.

Or, what if it was hidden exclusively behind a gesture:



Obscure gestures are rarely a good choice for navigational purposes.

What if it looked so out of place on the system, it was all you could notice (and not in a good way):



Placing elements in places where they don't normally reside on the platform is a sure way to make your app stand out, but not in a good way.

What navigation should be is clear and concise. Out of the way. And if it isn't, well, for most of us overcoming poor navigation is a matter of patience. It's not something we want to mess with it, but it's also doable and there's usually a way out.

Not everyone is so fortunate. For users who rely on accessibility APIs, poor navigation can leave them with no choice. Force quit the app and start again. Lose their orientation and take several steps to reorient themselves. The list goes on.

This chapter is all about what navigation should be in an accessible world. What does it look like, and what APIs exist to help us achieve it? What kinds of things should we work to avoid?

To answer those questions, we'll look at three specific APIs:

- VoiceOver
- The Escape Gesture

- Scrolling Custom Content

If you're reading this book linearly, you'll notice that some of these topics have been covered in previous chapters in some form. However, even if that's the case, there's still more to each of these APIs that we can uncover to help us maintain consistent, obvious navigation for accessibility purposes.

One last note here, this chapter deals with the technical aspects of navigation and accessibility APIs. Design and user experience definitely play a large role here too, and we'll cover navigation from those perspectives later on in their own chapters.

How it Works

VoiceOver

VoiceOver relies on the cursor to help users navigate around an interface. Elements that report themselves to the accessibility engine are eligible. Thankfully, good VoiceOver navigation is mostly about what *not* to do. If you set up the correct values for "The Big Three" as discussed "The 10,000 Foot View: The Basics" you'll be off and running.

To get a general framework of how to make navigation effortless with VoiceOver, consider these steps.

1. Make sure there aren't any elements that don't report to VoiceOver — unless they don't aid in the navigation experience. For example, if there is iconography that's purely decorative, then it likely makes sense to opt those out of VoiceOver:

```
// UIKit
myIcon.isAccessibilityElement = false
```

```
// SwiftUI
var body: some View {
    Image("Arrow")
       .accessibility(hidden: true)
}
```

2. Group elements logically. VoiceOver reads left-to-right (locale depending) and then top-to-bottom. For many VoiceOver users, visual and contextual cues are not available to them. Grouping and the spacing of your interface used to convey its semantics may not be as obvious to users who rely on visual aid and assistive technologies.
3. Use Rotors. By creating custom rotors and using the correct accessibility traits, VoiceOver users can jump around to certain parts of your interface much faster than it would take to navigate between them all manually.

VoiceOver is probably Apple's most important accessibility API. Millions of users rely on it all the time to navigate their iPhones, so it's imperative you are able to access each part of your app using it. This doesn't mean you should skip any customized interfaces elements, or eliminate gestures from your app or even navigation – but it *does* mean you need to ensure there's still an open door to access all of the same things for VoiceOver users.

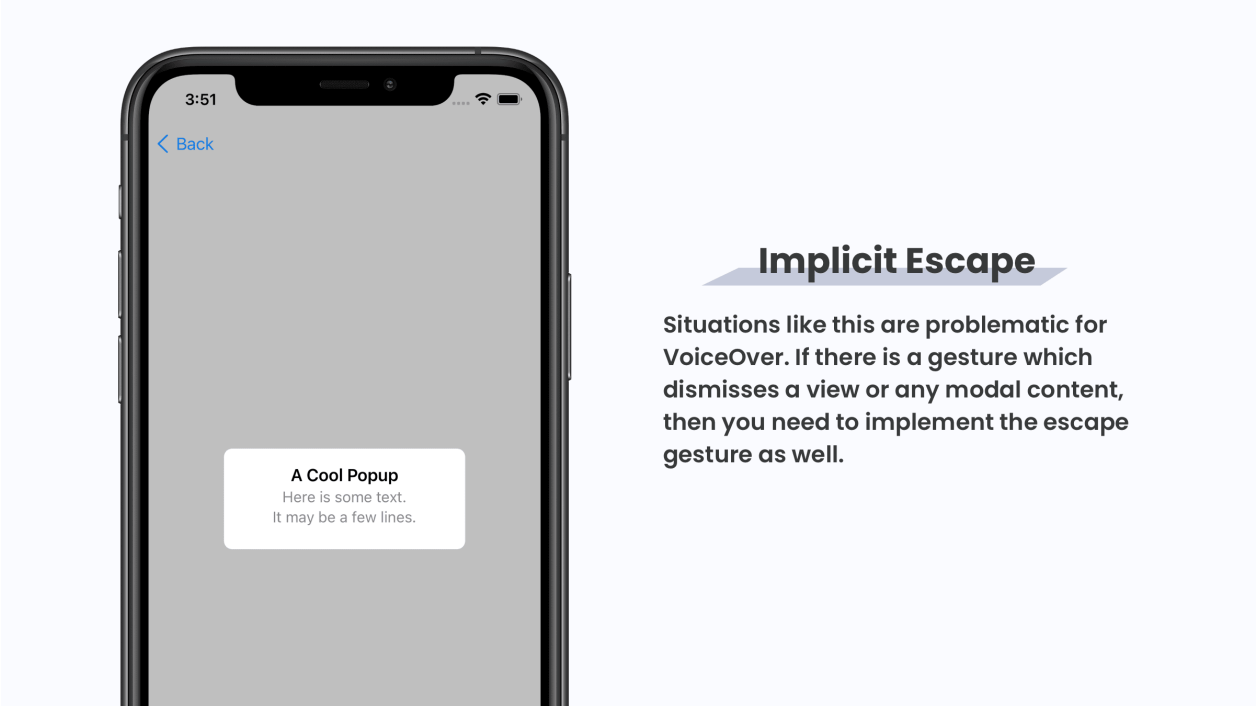
The Escape Gesture

The escape gesture, sometimes referred to as the two finger Z Gesture, is a way to dismiss views that present themselves modally or within a hierarchy. Try it out now to see why it's so paramount for navigation:

1. Activate VoiceOver.

2. Open the Settings app and choose any option which opens another view (in technical terms, push another controller onto the navigation stack).
3. Now, perform the escape gesture to pop back to the root view controller (or the the previous controller if you were a few levels deeper in the navigation stack).

This becomes essential if the way to get back to where you were requires a tap on a "dimming view", or some other visual affordance like the interface below:



VoiceOver users are accustomed to using this gesture to "go back" or "dismiss" things. If you've created custom interfaces, especially any pop ups or custom alerts like the one above, you'll want to implement this. Thankfully, it's typically a matter of one function implementation and two lines:

```
// UIKit -> Xcode ->
NavigationConsiderationsFig1ViewController.swift
```

```
@objc func dismiss() {
    UIView.animate(withDuration: 0.25, delay: 0.0,
options: .curveEaseOut) {
        self.transform = CGAffineTransform(scaleX: 0.25, y: 0.25)
        self.parentDimmerView?.alpha = 0.0
    } completion: { done in
        if done {
            self.parentDimmerView?.removeFromSuperview()
            self.removeFromSuperview()
        }
    }
}

// The necessary override
override func accessibilityPerformEscape() -> Bool {
    // Your custom dismissal code
    dismiss()
    return true
}
```

If the dismissal failed, be sure to return false here. The accessibility engine relies on this return value to relay other important context to the system, but in most cases – you're likely to simply return `true`, as dismissing something is rarely error prone.

Bottom sheets, or pop up cards that come from the bottom of the view and rise up, typically rely on tap gestures to dismiss. This is another case where the escape gesture would be the preferred means to exit such views. Be sure to poke around your app and look for places like these where VoiceOver users could potentially get stuck by not having a way to back out.

You may be wondering how to handle the same scenario in SwiftUI. There, this override isn't exposed as a view modifier. The reason why likely has to do with the way the framework handles any presentations – they are explicitly tied to `@State` variables. That said, you can still technically find yourself in such a state – but if you do,

you're likely fighting the framework. If something is being presented, it should be controlled from the appropriate SwiftUI binding.

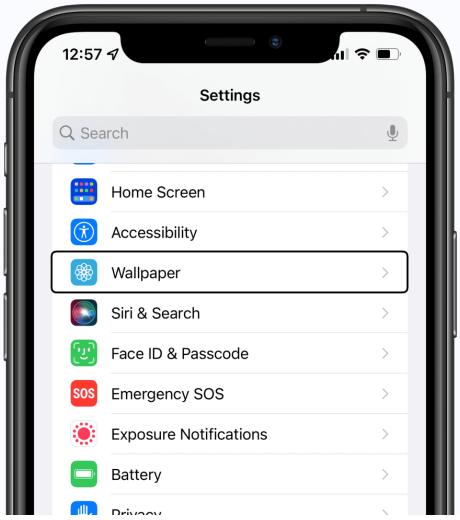
Scrolling Custom Content

Scrolling and paging content is an important part of navigation. We tend to put navigation strictly in a "go from this view to this other one" bucket, but navigation within views is equally important. Scrolling falls into that category. While it doesn't usually take you to another place in your app, it does help consume all the content in the active interface.

You'd be hard pressed to find any *any* app that doesn't rely on it. Table or collections views are likely the most common controls found on the platform, and long form text experiences such as reading or consuming content usually requires significant scrolling as well. For accessibility purposes, most users can scroll or page content using a three finger swipe gesture going up or down.

In the Settings app, iOS will scroll up or down with a three finger swipe and then call out the row that the VoiceOver cursor is on, and how many rows there in total ("Row 34 of 122"):

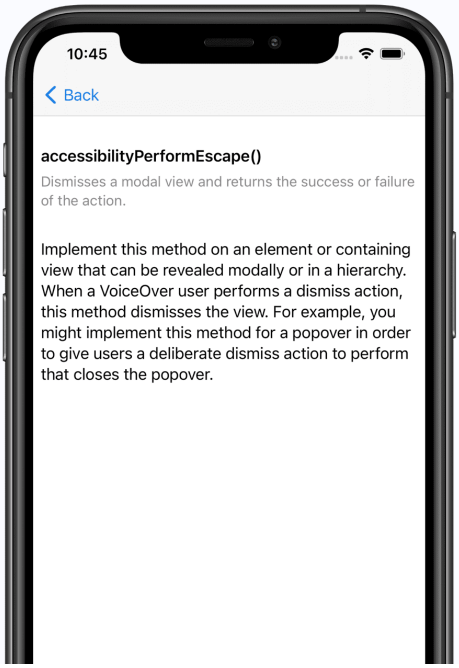
Scrolling content using VoiceOver.



Several scrolling scenarios are handled internally for you on behalf of table and collection views. However, if you've got a custom control you may need to implement it yourself with a three step process:

1. Implement `func accessibilityScroll(_ direction: UIAccessibilityScrollDirection) -> Bool`
2. Set your view's accessibility traits to include `.causesPageTurn`.
3. If it was successful, post the `.pageScrolled` accessibility notification. If it wasn't, the same function will be called on the parent view in the hierarchy.

Consider a custom view which showed some function documentation, and it could toggle or scroll between different functions to display:



You could leverage accessibility scrolling to accomplish this for VoiceOver users if it didn't already come for free by way of being in a scroll view or similar control:

```
// UIKit -> Xcode ->
NavigationConsiderationsFig2ViewController.swift -> DocView.swift

override func accessibilityScroll(_ direction:
UIAccessibilityScrollDirection) -> Bool {
    guard let currentDocIndex = docs.firstIndex(where: { $0 ==
activeDoc }) else {
        return false
    }

    var nextDoc: Doc?

    switch direction {
    case .previous, .left:
        if currentDocIndex == 0 { return false }
        nextDoc = docs[currentDocIndex - 1]
    case .next, .right:
```

```
            if currentDocIndex == (docs.count - 1) { return false }
            nextDoc = docs[currentDocIndex + 1]
        default:
            return false
        }

        setActiveDoc(doc: nextDoc)
        UIAccessibility.post(notification: .pageScrolled, argument: nil)
        return true
}
```

Here, when the control gets the accessibility swipe gesture, our override is called and we perform the scroll logic and indicate whether or not it was successful. Present `NavigationConsiderationsFig2ViewController` on your own device to try it out.

There two other important aspects to address here, and those are handled in the control's initializer:

```
isAccessibilityElement = true
accessibilityTraits = [.causesPageTurn]
```

If we didn't take care to add the accessibility trait, we'd be missing out on an important feature: auto scrolling. When a VoiceOver user performs any gesture that reads out the view's contents from beginning to end, we can have our control automatically "page" to the next piece of content.

In our case, that's the next function to show in `DocView`'s private array of docs:

```
// When the current function is read out by VoiceOver
// Then the next one in this array will be selected and
// Paged to.
static let samples: [Doc] = [/*All the docs*/]
```

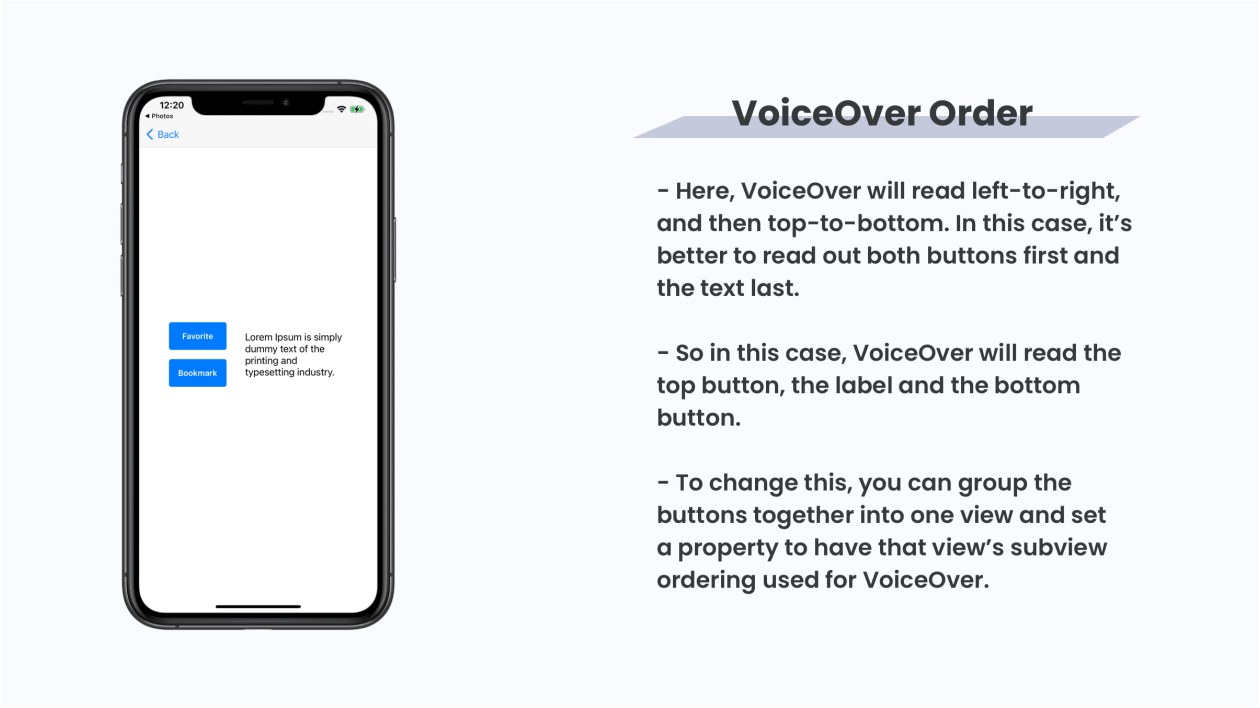
With this view presented, swipe down with two fingers to have VoiceOver read out the current function's data. When it completes reading out all of the content, our accessi-

bility scroll implementation will automatically be invoked by the system to continue. This is a case where the boolean you return is of more importance as opposed to other accessibility functions such as `accessibilityPerformEscape()`, where you nearly return true each time. If we return false, we're indicating to the system that we've reached the beginning or end of the content.

Tips

Grouping Accessibility Elements

There are times when VoiceOver should follow the grouping of an element's subviews instead of its default left-to-right, and top-to-bottom order. For example, consider the interface below:



Without any modifications, "Favorite, (text of label), Bookmark" is the order VoiceOver will follow. In this case, the visual flow is conveying a close relationship between the two buttons. As such, we'd likely prefer those read back-to-back with VoiceOver, and *then* the text label's text.

To remedy this, you can add the relevant views (in this case, the two buttons) into a container view and leverage `shouldGroupAccessibilityChildren` to indicate to VoiceOver that it should follow the element's subview ordering instead:

```
// UIKit -> Xcode ->
NavigationConsiderationsFig3ViewController.swift

// NOTE: To toggle the correct ordering on or off,
// Change the boolean property at the top of this file.
let containerView = UIView(frame: .zero)
view.addSubview(containerView)

// Now add both buttons in the container view
containerView.addSubview(favoriteBtn)
containerView.addSubview(bookmarkBtn)
containerView.translatesAutoresizingMaskIntoConstraints = false

// And specify we want them grouped for VoiceOver to follow
// This view's subview ordering.
containerView.shouldGroupAccessibilityChildren = true
```

Now, VoiceOver correctly follows the order we're after. The two buttons will follow each other, and then the text label is focused last.

With SwiftUI, you can have a similar interface but a very different result with VoiceOver. For example, here is the same interface built in SwiftUI:

```
// SwiftUI -> Xcode ->
var body: some View {
    HStack(alignment: .top) {
        VStack {
```

```
            Button("Favorite", action: {})
                .padding()
                .frame(width: 120)
                .background(Color.blue)
                .cornerRadius(4.0)
                .foregroundColor(.white)

            Button("Bookmark", action: {})
                .padding()
                .frame(width: 120)
                .background(Color.blue)
                .cornerRadius(4.0)
                .foregroundColor(.white)

        }
        Spacer()
        Text("Lorem Ipsum is simply dummy text of the
printing and typesetting industry. Lorem Ipsum
is simply dummy text of the printing and
typesetting.")
            .lineLimit(nil)
    }
    .padding()
}
```

In this case, VoiceOver reads out the elements in the order we desire already because it's built with a `VStack`, which is essentially accomplishing the same thing our container view did above. However, if you do need the same effect in SwiftUI, look to the `.accessibilityElement(children: .contain)` modifier.

Alert VoiceOver of Significant Content Changes

Whenever your app presents a new view, a context switch is occurring. You were in screen A, now you're in screen B. Visual cues help us navigate these context switches, and typically we gravitate towards the middle of the screen with our eyes when we do.

With VoiceOver, it'll focus the first accessible element in the left-to-right, top-to-bottom order. This may work most of the time, but there are cases where these users would be better served to have their focus directed somewhere else. We can do that with an accessibility notification specifically for letting users know the screen has changed. We can optionally pass in an element for VoiceOver to focus on as well.

Consider our function documentation view. By default, VoiceOver may focus on the back button in the navigation controller depending on how `DocView` is implemented. That's not what is most helpful in this case, VoiceOver users would be better served by ensuring the `DocView` instance gets focused first.

This can be done by leveraging the `.screenChanged` notification:

```
// UIKit -> Xcode ->
NavigationConsiderationsFig2ViewController.swift

override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)
    UIAccessibility.post(notification: .screenChanged, argument:
docView)
}
```

Ask yourself, when my view is presented, is the first element that's focused the most helpful in terms of navigation? If it's not, consider making a similar change as we've done here. Again, most elements should be accessible to VoiceOver, sure, but if the first element doesn't help users orient quickly or get something done - opt to focus on the control that will help them the most.

Even though the argument for the `.screenChanged` notification is of `Any` type, it's not meant for SwiftUI. In fact, if you attempted to use it for SwiftUI elements then your app will likely crash with a similar warning such as this:

This class **'SwiftValue' is not a known serializable element and returning it as an accessibility element may lead to crashes**

To see for yourself, try opening up `NavigationConsiderations-Fig1View.swift`, which mimics the `DocView`, in SwiftUI and uncomment the notification code in `onAppear`:

```
// SwiftUI -> Xcode -> NavigationConsiderationsFig1View.swift

struct NavigationConsiderationsFig1View: View {
    let doc: Doc = Doc.samples.first!
    var docDiscussion: some View {
        Text(doc.abstract)
            .foregroundColor(.secondary)
            .padding([.bottom], 32)
    }
    var body: some View {
        VStack(alignment: .leading) {
            Text(doc.functionName)
                .font(.headline)
                .padding([.bottom], 8)
            docDiscussion
            Text(doc.discussion)
            Spacer()
        }
        .padding()
        .accessibilityElement(children: .contain)
        .onAppear {
            // This will crash
            UIAccessibility.post(notification: .screenChanged,
argument: docDiscussion)
        }
    }
}
```

Attempt Navigation with Screen Curtain

When VoiceOver is activated, triple tap the screen with three fingers to toggle Screen Curtain. When it's on, your phone shuts off its screen and forces you to navigate solely using VoiceOver or other assistive technologies. Look for holes in the experience to fill.

One effective way to test with Screen Curtain is to set a predefined goal to accomplish. For example, if you have a weather app, you might set a goal to:

1. Listen to the forecast.
2. Change the location of that forecast.
3. Then customize notifications about the weather.

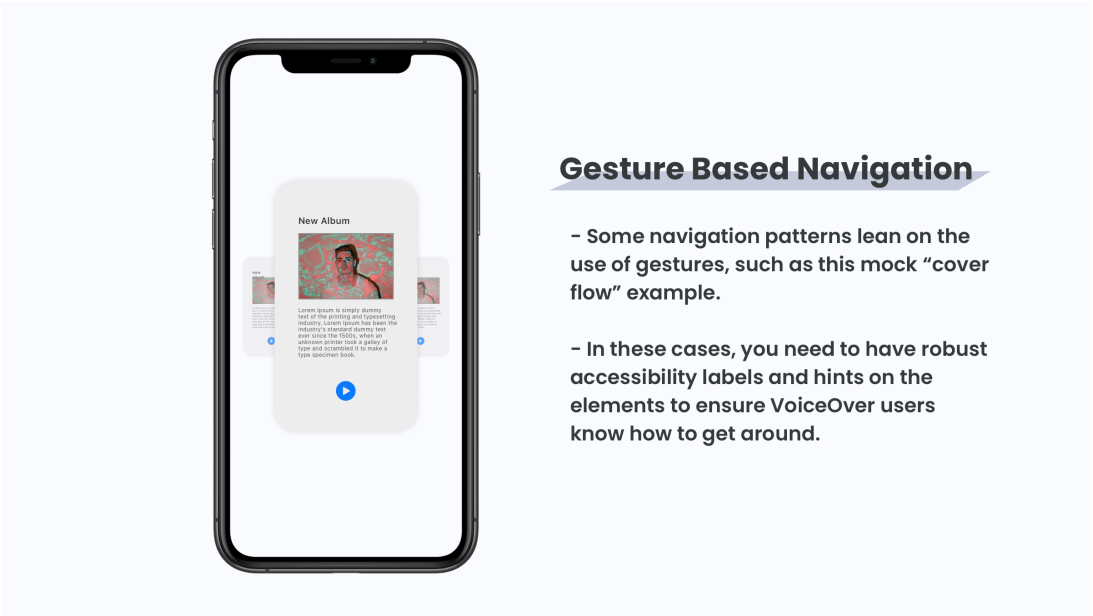
You should aim to do this all from a cold boot of your app. If you can't do some of those things, then it's likely VoiceOver users may not even know those features even exist.

Look for Custom Gesture Based Navigation

Some apps lean into a more customized navigational flow. While the jury is out on whether or not that leads to a more cohesive user experience, these kinds of apps definitely need to do some more work to make sure they remain accessible. The usual tips still apply:

1. Support the escape gesture for customized alerts, modal views or any scenario where there is no obvious "out" interface element.
2. If a gesture is the sole means of navigation, install something else in the interface when VoiceOver is active to allow for navigation, such as a button that either presents or dismisses the current view.

3. Consider more verbose accessibility labels or hints if the context of your app's navigation leans heavily into visual cues or contextual hints that are gleaned by element grouping or visual indicators like this interface below:



Here, the visual grouping hints to users that they should swipe left or right to change selected items. For VoiceOver users who don't get that benefit, ensure accessibility properties tell the full story on how to get around.

In general, gestural navigation can provide some great avenues for power users of your app. However, the general population will either skip right past them or are likely unaware they exist. Don't assume anything. Or, if you do, assume that a VoiceOver user *will* use your app, and if they do – it can't be exclusively gesture based.

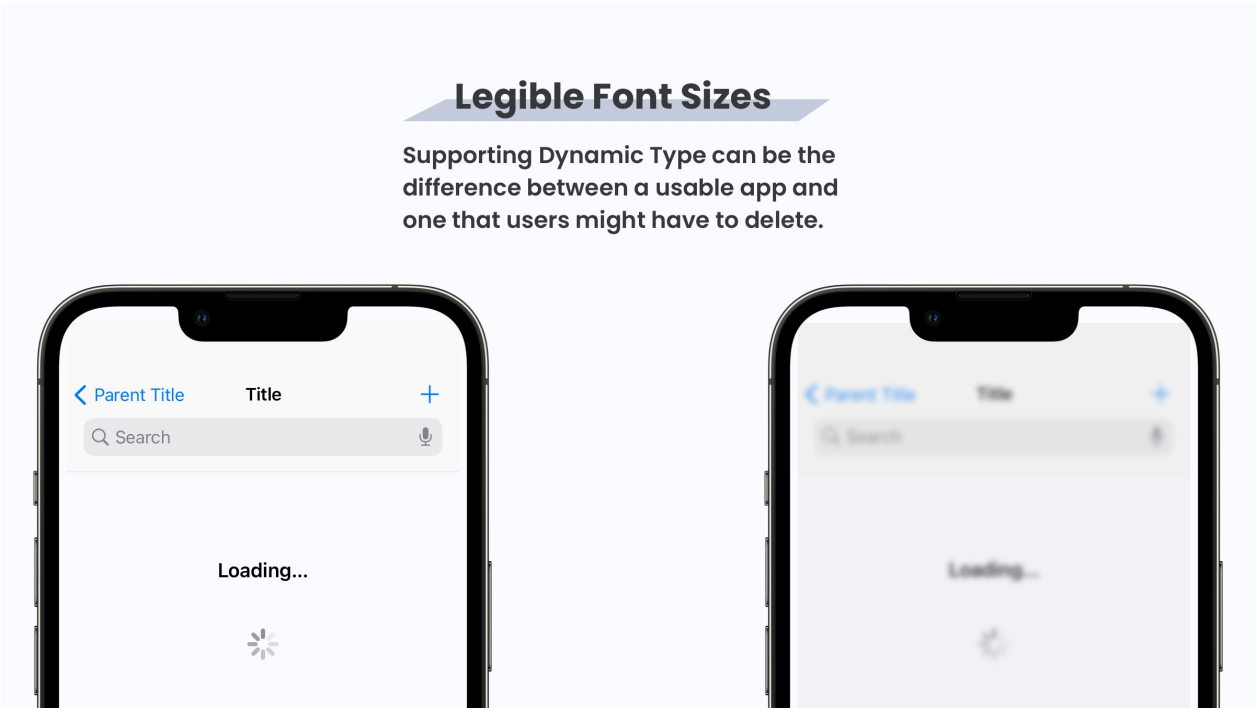
Three Key Takeaways

1. Navigation should be predictable and dependable, and it's *usually* not the best place for experimentation.
2. As such, accessibility users need clear buttons, iconography and the right APIs implemented by us to make navigation feel that way.
3. If we don't, users can easily become frustrated or worse – trapped in particular parts of our apps.

Dynamic Type

Aside from VoiceOver, there may not be a higher impact accessibility API than Dynamic Type. Introduced in iOS 7, it accomplishes one simple goal – make sure *everyone* can read text at a size of their choosing. Supporting this API can be the difference from someone being able to use your app, or deleting it.

That's no exaggeration either. If you don't support Dynamic Type, there's a real possibility that for some users, the text in your app simply isn't legible. And, try using any app with text that you can't read.



If your font size doesn't scale correctly, users may have a hard time reading text.

From an outsider's perspective, the simple act of letting users choose their text size preferences would appear to have limited implications to the apps they use. But as developers and designers, there's a lot of angles we need to consider. Your design will be stretched and flexed every which way and designing in absolutes is no longer a viable option.

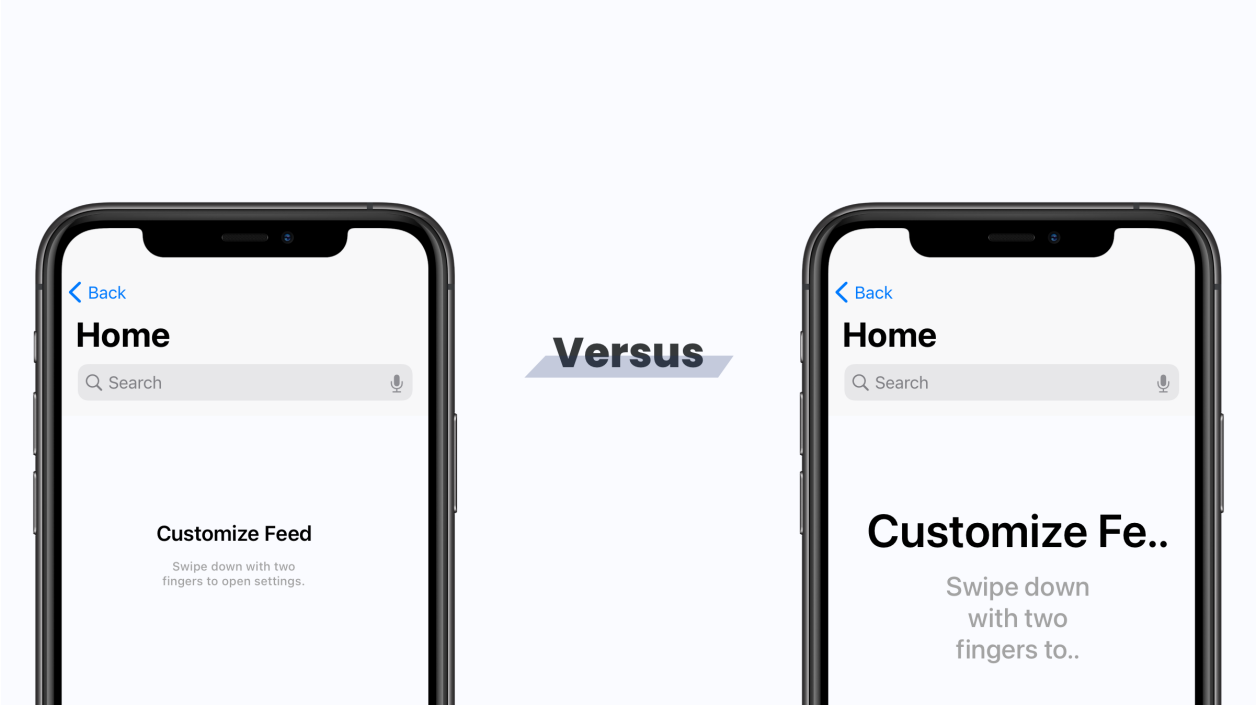
To make the process as simple as possible, I've always followed a simple set of rules when it comes to text in my own apps. These exist to ensure my text stays accessible, usable and readable:

1. All text should respond to Dynamic Type (accessible).
2. Make your layout adaptable (usable).
3. Strive to avoid text truncation (readable).

Truncation and Clipping

You should strive to avoid truncating text since, by its very nature, it removes some content that your user likely wants to see. There are exceptions, of course – there always are, but for the most part text shouldn't truncate or clip. If you are creating an app similar to News on iOS, then perhaps headlines could truncate. But even then, the intent is understood by most users. The headline hints at the rest of the content.

On the other hand, if you've got text that explains a feature which clips at a higher text size – now you've got a problem. This is content that *was* viewable at a lower text size, but now isn't. This also fails the accessibility smell test since content is being taken away from a user solely based off one of their accessibility preferences they rely on.

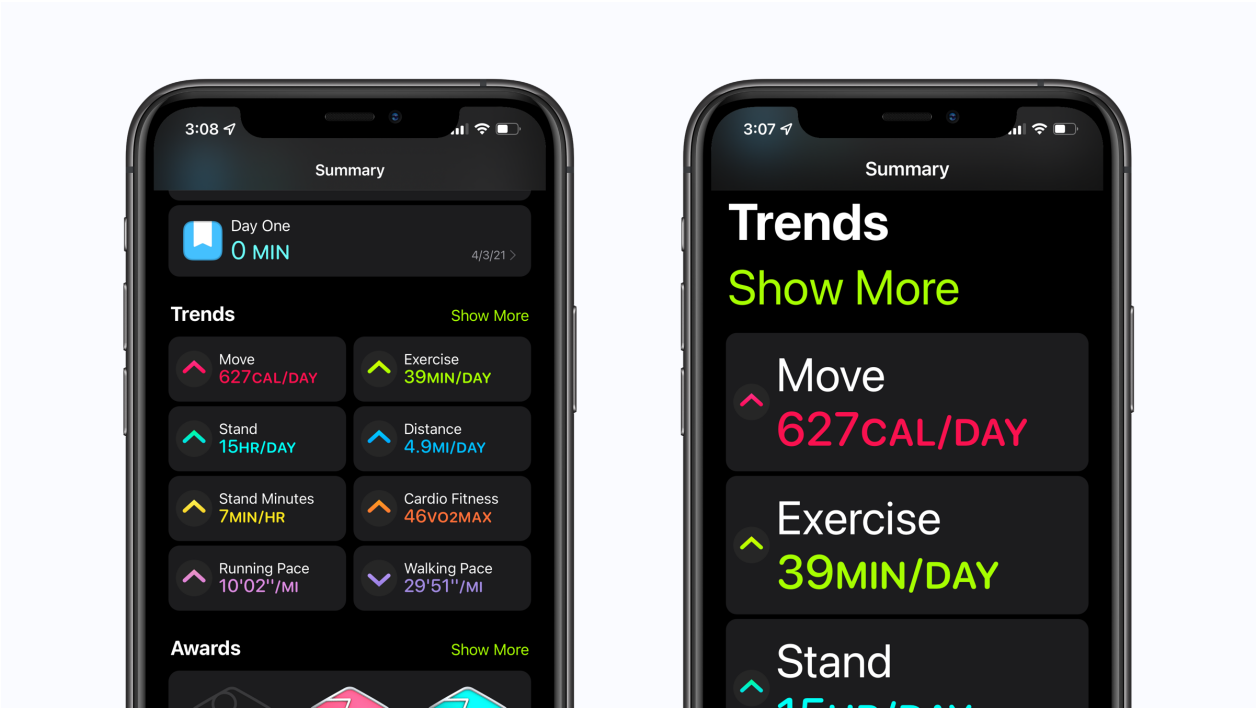


Clipping or truncating text is rarely a good idea.

Adaptability

Your layout should remain adaptable to support text both small and large. In short, this means you might need to flip from a horizontal layout to a vertical one at a

moment's notice. You can see this occurring all throughout iOS. Open up any stock app from Apple at a default text size, and then change it to the largest text style. In most cases, the layout shifts from a horizontal axis to a vertical one:



Notice how the interface flips to a vertical axis once the text size was increased.

Text Resizing

And of course, you should opt in text to support Dynamic Type. Though it may seem challenging to implement at first, due to the unpredictability of the font size your interface may be showing, it becomes second nature after a few iterations of designing, developing and supporting it.

Dynamic Type APIs

Dynamic Type is primarily driven through two things:

1. Text Styles

2. UIFontMetrics

We'll look at the technical side of both of these below. For now, it helps to understand that text styles describe the *purpose* of the text you want to use rather than a specific point size. If you're familiar with the semantic color APIs to support Dark Mode which rolled out with iOS 13, the idea is the same. It's less about saying "I want 24 point text" and instead thinking about the text's purpose. For example, "This text should represent a headline."

This helps from an implementation standpoint, too, because you're less inclined to find a perfect font size and instead focus on the purpose of the text. Thankfully, iOS has text styles to meet virtually every need:

```
// UIKit
extension UIFont.TextStyle {

    @available(iOS 11.0, *)
    public static let largeTitle: UIFont.TextStyle

    @available(iOS 9.0, *)
    public static let title1: UIFont.TextStyle

    @available(iOS 9.0, *)
    public static let title2: UIFont.TextStyle

    @available(iOS 9.0, *)
    public static let title3: UIFont.TextStyle

    @available(iOS 7.0, *)
    public static let headline: UIFont.TextStyle

    @available(iOS 7.0, *)
    public static let subheadline: UIFont.TextStyle

    @available(iOS 7.0, *)
```

```
    public static let body: UIFont.TextStyle

    @available(iOS 9.0, *)
    public static let callout: UIFont.TextStyle

    @available(iOS 7.0, *)
    public static let footnote: UIFont.TextStyle

    @available(iOS 7.0, *)
    public static let caption1: UIFont.TextStyle

    @available(iOS 7.0, *)
    public static let caption2: UIFont.TextStyle
}

// SwiftUI
public enum TextStyle : CaseIterable {
    /// The font style for large titles.
    case largeTitle
    /// The font used for first level hierarchical headings.
    case title
    /// The font used for second level hierarchical headings.
    @available(iOS 14.0, macOS 11.0, tvOS 14.0, watchOS 7.0, *)
    case title2
    /// The font used for third level hierarchical headings.
    @available(iOS 14.0, macOS 11.0, tvOS 14.0, watchOS 7.0, *)
    case title3
    /// The font used for headings.
    case headline
    /// The font used for subheadings.
    case subheadline
    /// The font used for body text.
    case body
    /// The font used for callouts.
    case callout
    /// The font used in footnotes.
    case footnote
    /// The font used for standard captions.
    case caption
```

```
    /// The font used for alternate captions.
    @available(iOS 14.0, macOS 11.0, tvOS 14.0, watchOS 7.0, *)
    case caption2
    /// A collection of all values of this type.
    public static var allCases: [Font.TextStyle]
}
```

However, you may find the need to scale glyphs or other interface elements in your app alongside text. Or, perhaps you're using a custom font. For these cases, `UIFontMetrics` can vend a meaningful and adaptable size to scale effectively anything.

Now, let's get into implementation details.

How it Works

UIKit Support

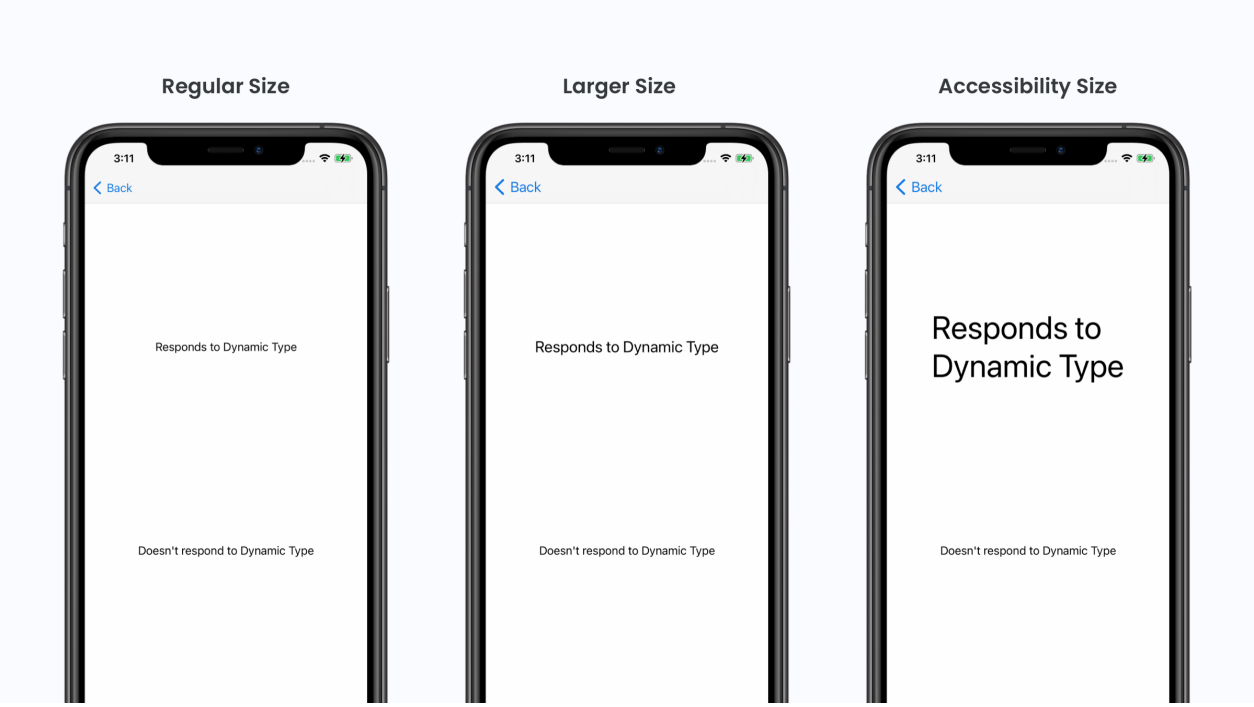
To enable dynamic type in UIKit's three main text controls (UILabel, UITextView and UITextField) you need to do two things:

1. Set `adjustsFontForContentSizeCategory` to true. This property signals to the control that it should adjust its font according to the user's content size preferences.
2. Assign a font vended from a text style using `UIFont.preferredFont(forTextStyle:)`.

Note that if you only did step two, you would *kind of* have dynamic type support. The system would vend the right font size at the time it was invoked, but it would not adjust thereafter should the user change their font size on the fly.

Here is an example with two labels. The first one fully supports Dynamic Type, while the second does not:

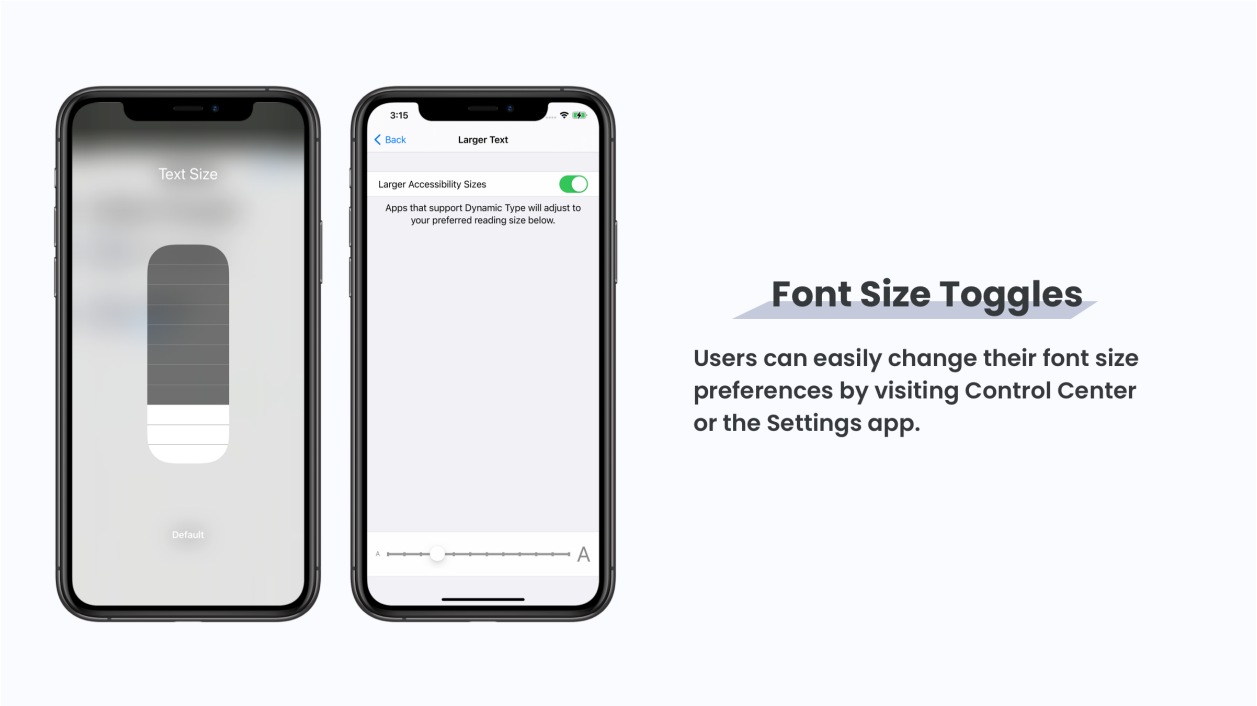
```
// UIKit -> Xcode -> DynamicTypeFig1ViewController.swift
let dynamicLabel = UILabel(frame: .zero)
dynamicLabel.numberOfLines = 0
dynamicLabel.text = "Responds to Dynamic Type"

// Dynamic Type support is here
dynamicLabel.adjustsFontForContentSizeCategory = true
dynamicLabel.font = UIFont.preferredFont(forTextStyle: .body)

let nonDynamicLabel = UILabel(frame: .zero)
nonDynamicLabel.numberOfLines = 0
nonDynamicLabel.text = "Doesn't respond to Dynamic Type"
```



Fonts on iOS are immutable. When the content size category changes, a new instance of the correct font is assigned to the text control as opposed to mutating the instance already present. This might lead you to ask, "When does the change actually occur?"

From a user's perspective, the change happens when they edit their font size choice. From a system level, this is represented as a *content size preference*. This

could happen within iOS' Settings app, or via Control Center and the Dynamic Type shortcut found there:



Users can change their content size preferences easily in iOS, so your apps needs to be ready to adjust on the fly.

In code, that means you've got two avenues to go down to intercept these changes, or otherwise handle them:

1. By registering for the content size category trait change using `registerForTraitChanges`.
2. By observing `UIContentSizeCategory.didChangeNotification`

By registering for trait collection changes at the view controller level, you get a few benefits. One is that you are passed the previous trait collection, which can be beneficial should you need to apply any application logic based off of what was showing be-

fore the change occurred. You can see what the content size is now, and what it was prior to the update trivially:

```
// UIKit -> Xcode -> DynamicTypeFig1ViewController.swift

registerForTraitChanges([UITraitPreferredContentSizeCategory.self])
{ (self: Self,

previousTraitCollection) in
    print("Changed from \(String(describing:
previousTraitCollection.preferredContentSizeCategory)) to
\(self.traitCollection.preferredContentSizeCategory)")
}
```

In addition, the `registerForTraitChanges` function lets you specify *which* traits you are interested in, ensuring that you focus your logic. However, this method won't be much use in SwiftUI unless your `View` is housed in a hosting controller via `UIHostingController`.

Using the `didChangeNotification`, you can leverage Combine to be alerted to changes (or the classic way of observing notifications if you prefer):

```
// UIKit -> Xcode -> DynamicTypeFig1ViewController.swift

NotificationCenter.default
    .publisher(for: UIContentSizeCategory.didChangeNotification)
    .compactMap{ $0.userInfo?
[UIContentSizeCategory.newValueUserInfoKey] }
    .sink { newValue in
        print("Changed content size to: \(newValue)")
    }
    .store(in: &subs)
```

This method is perfectly fine to use in SwiftUI, but note that you no longer are getting the previous trait collection value. If you still needed that due to your application log-

ic, adding a local variable to track it is the way to go. However, if you only need to know the Dynamic Type size, there is an environment variable just for that:

```
@Environment(\.dynamicTypeSize) var dynamicTypeSize
```

Both are viable options, but you'll likely make your choice on which to use by assessing the needs of your app's current architecture. Some lend themselves well to a Combine pipeline, others would be better served by simply responding to things in a trait collection override. Remember, both of these methods are places to apply logic to things other than your text size, which should already be handled via Dynamic Type support.

UIFontMetrics

I recommend most apps stick to the system font. It scales great, looks phenomenal and has Dynamic Type support right out of the box. But, there are times when you have to use a custom font. Or, what about icons or glyphs that should scale along with text size as well? It's not just fonts we have to consider with Dynamic Type support, it's our interface holistically that should respond and adapt to content size preferences.

For these cases, we've got `UIFontMetrics`. By giving it a baseline font from which to scale by, it can then vend a variant of a custom font scaled according to the font passed in. For example, UIKit supplies several static sizes of a label that you could scale a custom font from:

```
extension UIFont {

    open class var labelFontSize: CGFloat { get }

    open class var buttonFontSize: CGFloat { get }

    open class var smallSystemFontSize: CGFloat { get }
```

```
    open class var systemFontSize: CGFloat { get }
}
```

So, if you had a basic label, you could dynamically scale its size by `labelFontSize`:

```
// UIKit -> Xcode -> DynamicTypeFig2ViewController.swift

override func viewDidLoad() {
    super.viewDidLoad()

    let customFontLabel = UILabel(frame: .zero)

    // The font we want to scale using
    // UIFont.labelFontSize
    guard let customFont = UIFont(name: "alarmclock", size:
UIFont.labelFontSize) else {
        fatalError("Couldn't find custom font alarmclock")
    }

    // UIFontMetrics.default then scales the font
    customFontLabel.font = UIFontMetrics.default.scaledFont(for:
customFont)
    customFontLabel.adjustsFontForContentSizeCategory = true
    customFontLabel.text = "Custom font that scales."
}
```

If you run this sample and the content size preferences change, it now scales along with it.

More commonly, though, is to use text styles. The process then looks like this:

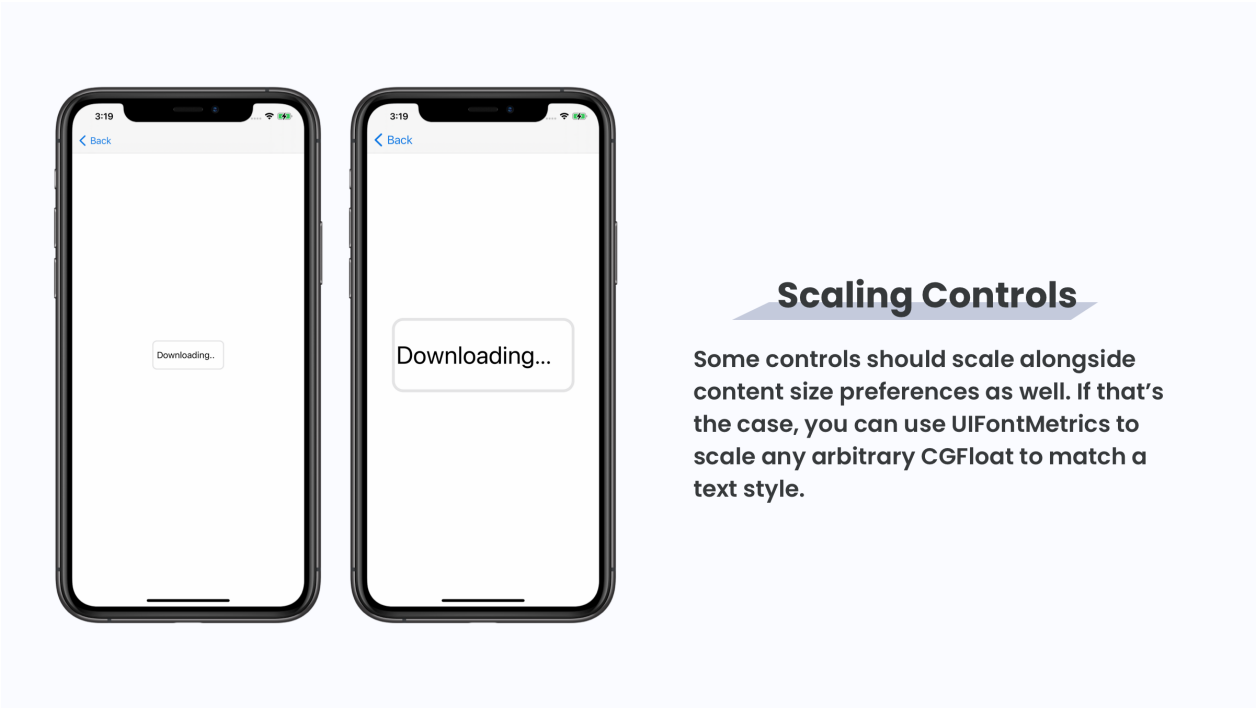
1. Figure out which text style to scale the text control with.
2. Initialize a `UIFontMetrics` object with it.
3. Then assign the font vended from it via `scaledFont(for font: UI-Font) -> UIFont`

```
// UIKit -> Xcode -> DynamicTypeFig2ViewController.swift

let customFontTextStyleLabel = UILabel(frame: .zero)
let metrics = UIFontMetrics(forTextStyle: .body)
customFontTextStyleLabel.font = metrics.scaledFont(for: customFont)
customFontTextStyleLabel.adjustsFontForContentSizeCategory = true
customFontTextStyleLabel.text = "Custom font from text style."
```

This code sample and the one before are functionally equivalent. The `default` property from `UIFontMetrics` returns a text style of `body` by default. However, if you need something larger, such as `headline`, you'd pass it in when initializing the metrics object.

However, `UIFontMetrics` can used outside of fonts, too. For most images that use vector data, `adjustsImageSizeForAccessibilityContentSizeCategory` (discussed in depth in the "Image Adjustments" chapter) will take care of things. But what about arbitrary controls we make ourselves? For these, we can leverage the metric's ability to return a scaled font from any `CGFloat` value.

By doing so, you can get an effect like this when the content size is changed:



Fonts are not the only way to use these APIs, any controls can scale as well.

In my own apps, where controls need to dynamically resize like this, I typically create an `enum` to represent how the pieces of it should be calculated according to content size preferences:

```
// UIKit -> Xcode -> DynamicTypeFig3ViewController.swift

enum ScaleElement {
    case borderWidth, cornerRadius, controlWidth, controlHeight

    func scaledElementSize() -> CGFloat {
        switch self {
        case .borderWidth:
            return UIFontMetrics.default.scaledValue(for: 2)
        case .cornerRadius:
            return UIFontMetrics.default.scaledValue(for: 8)
        case .controlWidth:
            return UIFontMetrics.default.scaledValue(for: 120)
        case .controlHeight:
            return UIFontMetrics.default.scaledValue(for: 52)
```

```
        }
    }
}
```

And then, you can set them accordingly in their initializer:

```
// UIKit -> Xcode -> DynamicTypeFig3ViewController.swift ->
ScaledControl.swift

override init(frame: CGRect) {
    super.init(frame: frame)
    clipsToBounds = true

    // Dynamic corner radius and border width
    layer.cornerRadius =
ScaleElement.cornerRadius.scaledElementSize()
    layer.borderWidth = ScaleElement.borderWidth.scaledElementSize()

    layer.borderColor = UIColor.systemFill.cgColor
    backgroundColor = .systemBackground
    addSubview(textLabel)
    textLabel.centerXAnchor.constraint(equalTo:
centerXAnchor).isActive = true
    textLabel.centerYAnchor.constraint(equalTo:
centerYAnchor).isActive = true

    // Dynamic width and height
    widthConstraint = widthAnchor.constraint(equalToConstant:
ScaleElement.controlWidth.scaledElementSize())
    heightConstraint = heightAnchor.constraint(equalToConstant:
ScaleElement.controlHeight.scaledElementSize())
    widthConstraint.isActive = true
    heightConstraint.isActive = true
}
```

Whenever the control is presented, you get a size based off of the Dynamic Type settings. However, the code above is similar to attempting to support Dynamic Type in a text control and doing this:

```
myLabel.font = UIFont.preferredFont(forTextStyle: .body)
```

Can you spot the error? We didn't set `adjustsFontForContentSizeCategory` in the label. This means when the control is loaded, it will initially be correctly sized. But, if the Dynamic Type size is changed, it wouldn't reflect the correct size until it was re-allocated and presented once more. That's not what we're going for, especially since the API itself is named *Dynamic* Type, our views and controls should be dynamic too.

Solving it requires taking one of the approaches mentioned above to intercept these changes. In our case, we'll use Combine:

```
// UIKit -> Xcode -> DynamicTypeFig3ViewController.swift ->
ScaledControl.swift

// Continued in the initializer...
NotificationCenter.default
    .publisher(for: UIContentSizeCategory.didChangeNotification)
    .compactMap{ $0.userInfo?
[UIContentSizeCategory.newValueUserInfoKey] }
    .sink { [weak self] newValue in
        guard let self = self else { return }
        // Apply the new dynamic sizing
        self.widthConstraint.constant =
ScaleElement.controlWidth.scaledElementSize()
        self.heightConstraint.constant =
ScaleElement.controlHeight.scaledElementSize()
        self.layer.borderWidth =
ScaleElement.borderWidth.scaledElementSize()
        self.layer.cornerRadius =
ScaleElement.cornerRadius.scaledElementSize()
    }
```

```
    .store(in: &subs)
```

Now, the control resizes alongside any content size preference changes. If you run this sample code, you'll notice how each aspect of the control resizes along with the content size preference changes. The border, size and font all adapt.

SwiftUI Support

SwiftUI supports text styles the same way that UIKit does. Though, it is a bit easier from an implementation standpoint. In fact, SwiftUI will support Dynamic Type out of the box given any text control when a font with a specific size isn't used:

```
// SwiftUI -> Xcode -> DynamicTypeFig1View.swift

// Responds to content size changes automatically
Text("Dynamic Type")
```

There is no boolean property or font style to specify to opt-in. However, if you were to provide a specific font size, the font is guaranteed to stay at that size and will not respect Dynamic Type settings:

```
// Stays at 10 points no matter the content size change
Text("System Font: Size 10")
    .font(.system(size: 10))
```

Text styles can be applied to the font modifier as well. This is likely the method you're after for the majority of your text controls. Staying close to Apple's text styles ensures your designs flows with the rest of the system, and many other apps, as well:

```
// SwiftUI -> Xcode -> DynamicTypeFig1View.swift
Text("Text Style: Headline")
    .font(.headline)
```

If you require custom fonts, they are supported in SwiftUI too. In fact, the API is even simpler than UIKit. To get the same effect as `UIFontMetrics` would bring, simply use the modifier which takes in a custom font name, base size and a text style to scale it against:

```
public static func custom(_ name: String, size: CGFloat, relativeTo
textStyle: Font.TextStyle) -> Font
```

To match our example from UIKit from earlier, it would look like this:

```
// SwiftUI -> Xcode -> DynamicTypeFig1View.swift
Text("Custom Scaled")
    .font(.custom("alarmclock",
           size: UIFont.labelFontSize,
           relativeTo: .body))
```

Tips

Use Extensions to Simplify Dynamic Type Support in UIKit

While supporting Dynamic Type does not require much setup, I find small utility extensions make it a bit easier:

```
// UIKit -> Xcode -> DynamicTypeFig3ViewController.swift
func supportingDynamicType(withTextStyle textStyle:UIFont.TextStyle
= .body) {
    self.numberOfLines = 0
    self.adjustsFontForContentSizeCategory = true
    self.font = UIFont.preferredFont(forTextStyle: textStyle)
}

// At the call site
let testLabel = UILabel(frame: .zero)
```

```
testLabel.supportingDynamicType()
```

Of, if you prefer inline chaining as a coding style, you can return the instance itself:

```
func supportingDynamicType(withTextStyle textStyle:UIFont.TextStyle
= .body) -> UILabel {
    self.numberOfLines = 0
    self.adjustsFontForContentSizeCategory = true
    self.font = UIFont.preferredFont(forTextStyle: textStyle)
    return self
}

let testLabel = UILabel(frame: .zero)
                    .supportingDynamicType()
                    .someOtherFunction()
```

Think about Dynamic Type Early

Dynamic Type support is going to painful and problematic for apps that are mature and are just *now* thinking about it. Even so, the effort is worth it. However, if you have the luxury of designing a new feature or perhaps starting on a new app altogether, thinking about Dynamic Type now will help you going forward.

Dynamic Type is unique in that it fills many roles in modern software engineering:

1. **It's a design consideration:** Your font size is not guaranteed, and you have to design accordingly. Accounting for an interaction with 12 point text versus 24 point text is different, and so too are the designs to accommodate each. By thinking about Dynamic Type at this stage, you start to grow a new design muscle which is more resistant to the pitfalls of variable text sizes.
2. **It's an accessibility necessity:** Much like VoiceOver users simply cannot continue in some places of your app if you don't implement an escape

gesture, many people cannot read text in your app if their content size preferences are ignored. It's another door shut for a potential user. A best-in-class iOS app is designed for everyone, and Dynamic Type support covers a large base of that "everyone" cohort.

3. **It's an engineering consideration:** No matter how your app is architected, there will be some thought required to make sure Dynamic Type works well. For example, views that were never meant to scroll will now need to when the Dynamic Type size is cranked up all the way. Constraints that were static have to be dynamic. A simple container view should probably be a stack view. A view that didn't scroll now might require it.

As you can see, Dynamic Type is a team effort and will require input from several folks across all disciplines.

Custom Font Weights and Dynamic Type in UIKit

The Dynamic Type API does a great job of vending fonts at variable sizes to meet user's needs, but it's also opinionated when it comes to font weights. Since the text styles convey a semantic meaning to each font, it stands to reason that a particular font weight is applied to each one as well.

For example, a headline font should be heavier than a caption style. And, so it is when using them in your code and interface:

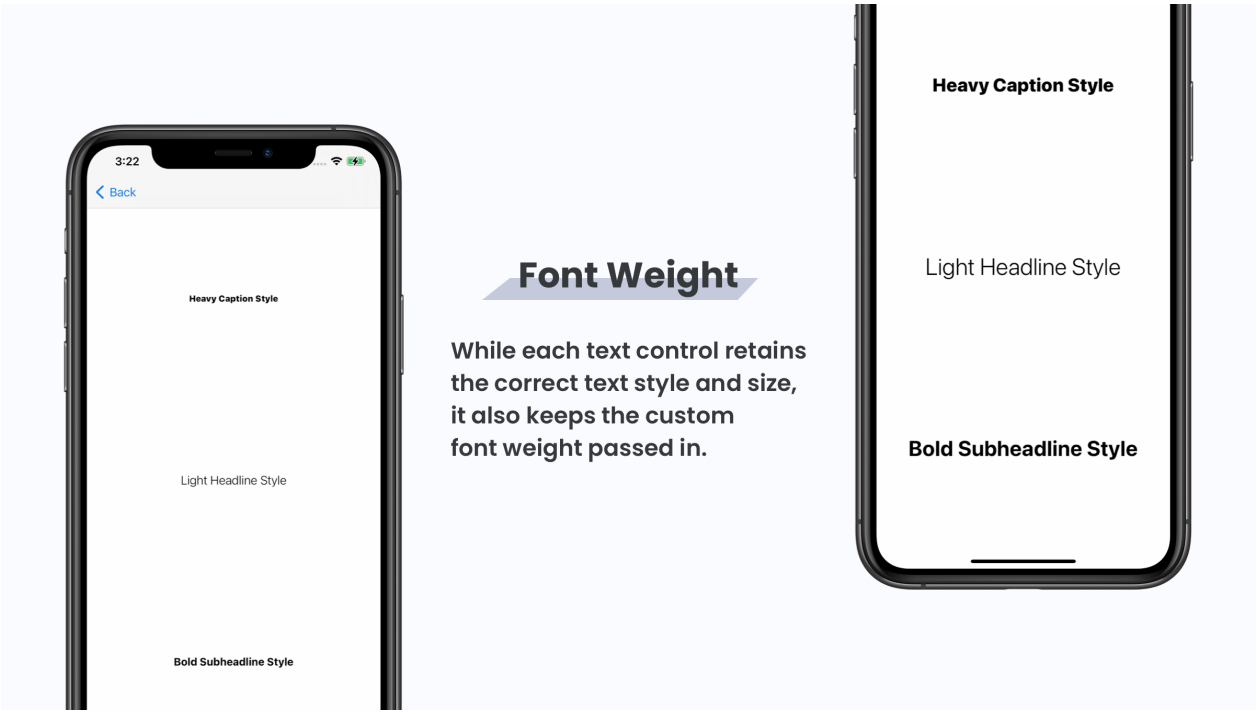
```
// This is heavier...
UIFont.preferredFont(forTextStyle: .headline)

// Than this
UIFont.preferredFont(forTextStyle: .caption1)
```

However, there are times when you want to leverage the ease of Dynamic Type but apply a different font weight. For these cases, a simple subclass can get the job done:

```
// UIKit -> Xcode -> DynamicTypeFig4ViewController.swift
class CustomWeightLabel: UILabel {
    private let fontWeight: UIFont.Weight
    private let textStyle: UIFont.TextStyle
    private var contentSizeSub: [AnyCancellable] = []

    init(withWeight weight:UIFont.Weight, textStyle:
UIFont.TextStyle) {
        self.fontWeight = weight
        self.textStyle = textStyle
        super.init(frame: .zero)

        NotificationCenter.default
            .publisher(for:
UIContentSizeCategory.didChangeNotification)
            .compactMap{ $0.userInfo?
[UIContentSizeCategory.newValueUserInfoKey] }
            .sink { [weak self] newValue in
                guard let self = self else { return }
                self.configureFont()
            }
            .store(in: &contentSizeSub)

        configureFont()
    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    private func configureFont() {
        let scaledFontSize = UIFont.preferredFont(forTextStyle:
textStyle).pointSize
        font = UIFont.systemFont(ofSize: scaledFontSize, weight:
fontWeight)
    }
}
```

The trick is to get the point size of the text style using the Dynamic Type API, and then use `UIFont. systemFont(ofSize: weight:)` to get a new font with the font size that *you* want while retaining the size of the text that the *user* needs.

Using Combine, listening for content size changes mean you get the "dynamic" part too, which is important. If the user changes their text size on the fly, your text control will adapt as well. In the sample code directory, if you run `DynamicType-Fig4ViewController.swift` you'll see that not only can you use different text styles with whatever font weight you wish, but they'll change on the fly when you adjust font sizes:



I've also included some print statements so you can see what's occurring to support this behavior. For example, you'll notice the delta between the font sizes stays consistent across differing content size preference changes:

```
// Default size
```

```
Applying 12.0 to text style UIFontTextStyle(_rawValue:
UICTFontTextStyleCaption1)
Applying 17.0 to text style UIFontTextStyle(_rawValue:
UICTFontTextStyleHeadline)
Applying 15.0 to text style UIFontTextStyle(_rawValue:
UICTFontTextStyleSubhead)

// Accessibility Large
Applying 32.0 to text style UIFontTextStyle(_rawValue:
UICTFontTextStyleCaption1)
Applying 40.0 to text style UIFontTextStyle(_rawValue:
UICTFontTextStyleHeadline)
Applying 36.0 to text style UIFontTextStyle(_rawValue:
UICTFontTextStyleSubhead)

// Extra Large
Applying 43.0 to text style UIFontTextStyle(_rawValue:
UICTFontTextStyleCaption1)
Applying 53.0 to text style UIFontTextStyle(_rawValue:
UICTFontTextStyleHeadline)
Applying 49.0 to text style UIFontTextStyle(_rawValue:
UICTFontTextStyleSubhead)

// Down to the smallest size
Applying 12.0 to text style UIFontTextStyle(_rawValue:
UICTFontTextStyleCaption1)
Applying 17.0 to text style UIFontTextStyle(_rawValue:
UICTFontTextStyleHeadline)
Applying 15.0 to text style UIFontTextStyle(_rawValue:
UICTFontTextStyleSubhead)
```

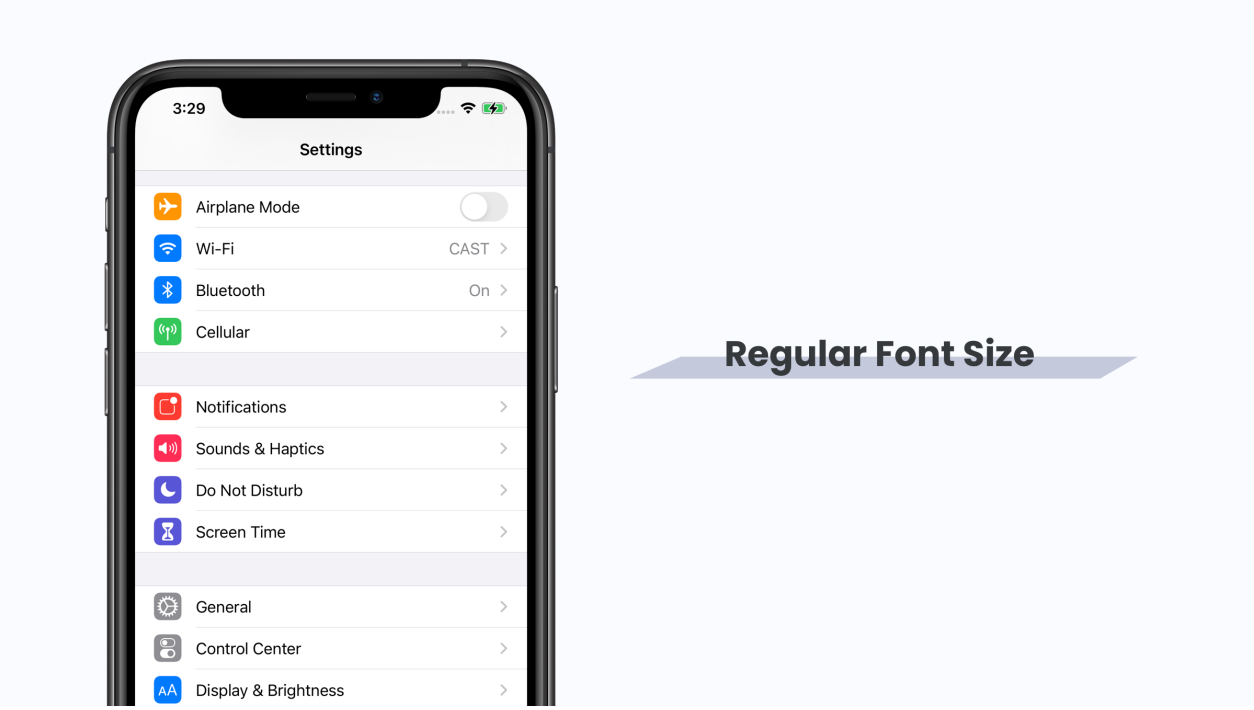
In SwiftUI, you can get this same behavior if you *don't* apply a specific font size but do request a particular weight using the `fontWeight(_:)` modifier. When it comes to Dynamic Type flexibility, SwiftUI makes it much easier:

```
// Still adapts to dynamic type
// You can stil request a text style *and* a font weight
// And it'll respond to content size changes
```

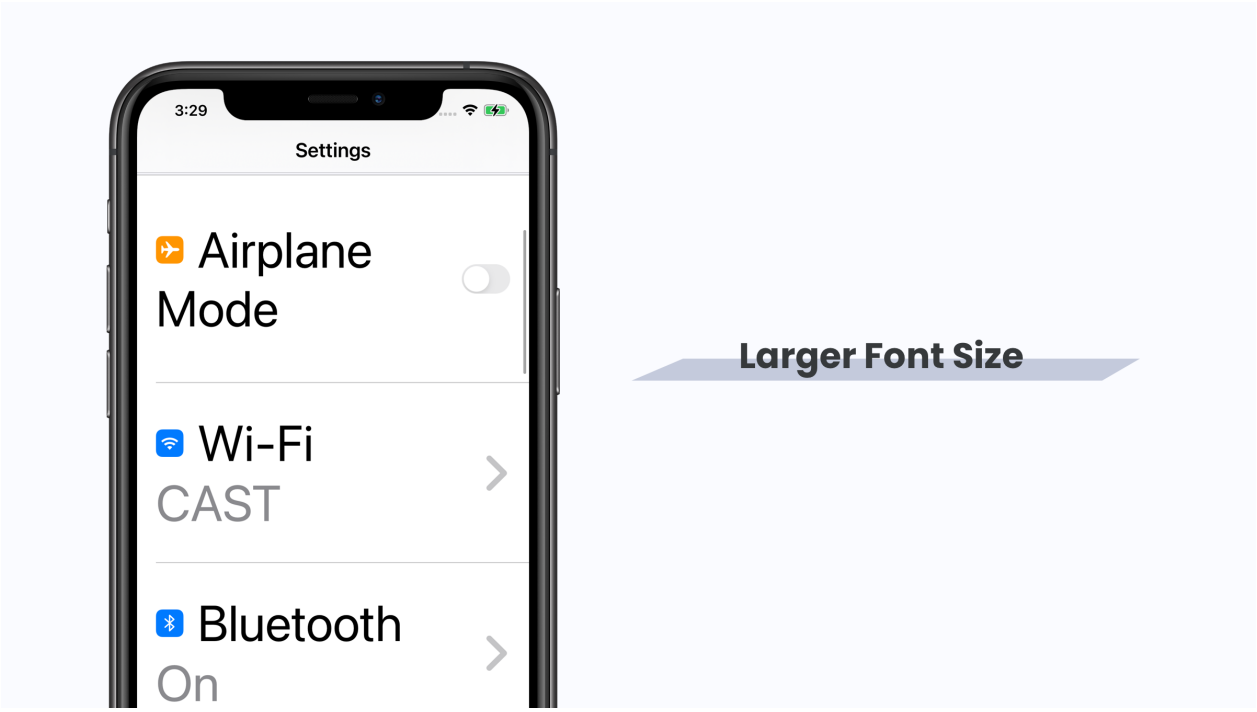
```
Text("Dynamic Type")
    .font(.body)
    .fontWeight(.thin)
```

Flipping Interface Axis

With Dynamic Type, you're all but guaranteed to face an interface that cannot fit the text within a horizontal axis. For example, consider the interface in the Settings app which features horizontally placed elements in each row:



Now, let's see how it fares under the largest accessibility text size:



To follow the same interface pattern, there are two common approaches you can use. Commonly, you'll use them together:

1. Use a `UIStackView` in UIKit that flips its axis under large content sizes, or switch between a `HStack` and `VStack` in SwiftUI.
2. House interfaces in a `UIScrollView` in UIKit or a `ScrollView` in SwiftUI to discourage any text, or other interface elements, from clipping.

In UIKit, we can take the same approach as we did above to build a stack view which changes its axis. Instead of looking at text styles, it's a matter of seeing our application's content size preferences have crossed over into an "accessibility size":

```
// UIKit -> Xcode -> DynamicTypeFig5ViewController.swift

class AdaptableStackView: UIStackView {
    private var contentSizeSub: [AnyCancellable] = []
```

```
    override init(frame: CGRect) {
        super.init(frame: frame)
        NotificationCenter.default
            .publisher(for:
UIContentSizeCategory.didChangeNotification)
            .compactMap{ $0.userInfo?
[UIContentSizeCategory.newValueUserInfoKey] }
            .sink { [weak self] newValue in
                guard let self = self else { return }
                self.adjustAxisIfNeeded()
            }
            .store(in: &contentSizeSub)
    }

    required init(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    private func adjustAxisIfNeeded() {
        let isAccessibilitySize =
UIApplication.shared.preferredContentSizeCategory.isAccessibilityCat
egory

        if isAccessibilitySize && axis != .vertical {
            axis = .vertical
            alignment = .fill
        } else if axis != .horizontal {
            axis = .horizontal
            alignment = .firstBaseline
        }
    }
}
```

Using this approach, our stack view remains on the horizontal axis when the content size preference isn't an accessibility size, but then it flips to vertical automatically when it is. To check when to perform a change, `UIContentSizeCategory` has a handy boolean property which will tell us exactly when an accessibility size is active:

`UIApplication.shared.preferredContentSizeCategory.isAccessibilityCategory`

Note that if you're doing this at a view controller level, you can query the trait collection's content size category changes to do the same check:

```
// UIKit -> Xcode -> DynamicTypeFig5ViewController.swift

// In viewDidLoad...
registerForTraitChanges([UITraitPreferredContentSizeCategory.self])
{ (self: Self,

previousTraitCollection) in
    print("Is AX Size: \
(self.traitCollection.preferredContentSizeCategory.isAccessibilityCa
tegory)")
}
```

Though, this could report a different preference than the one `UIApplication` will have since trait collections can be overridden.

In SwiftUI, we can support a dynamically changing stack using its `@Environment` property wrapper to listen for changes:

```
// SwiftUI -> Xcode -> DynamicTypeFig2View.swift
struct AdaptableStack<StackBody: View>: View {
    @Environment(\.sizeCategory) var contentSizeCategory
    let stackBody: StackBody

    init(@ViewBuilder stackBody: @escaping () -> StackBody) {
        self.stackBody = stackBody()
    }

    var body: some View {
        if contentSizeCategory.isAccessibilityCategory {
            VStack { self.stackBody }
        } else {
```

```
            HStack { self.stackBody }
        }
    }
}

// Used in a view...
struct DynamicTypeFig2View: View {
    var body: some View {
        AdaptableStack {
            Text("Here is some content!")
            Text("And here is some more!")
        }
    }
}
```

Embedding content in a scrollview can be more challenging, and the best approach will vary according to the state of your architecture. In the past, I've used several approaches from inheritance in UIKit at the controller level, to a custom control itself which manages the scroll view and other similar approaches.

The main takeaway with scrolling is this: a lot of content that you didn't expect to fill a whole entire screen can and will. One common area that I see a lot of app's miss is the ubiquitous "About" view. It usually consists of the app's icon, some metadata and a blurb about the developer. Views like this *might* support Dynamic Type, but when they do - they clip. While it seems to fit with plenty of room at first, a large font size changes things real quick. In these situations, be sure to embed controls in a scroll view to allow for the growth of the content and the ability to see it all.

Specifying a Maximum Point Size

While supporting Dynamic Type in all cases is important, there are designs or experiences where you need to define a ceiling. For example, the font size here should be a `.headline` style, but it can't be larger than 40 points. The numbers of such cases

shouldn't outnumber the amount of times you support Dynamic Type without "limits", but they can happen despite our best efforts.

In UIKit or SwiftUI, you can mirror the same approach and use some code like this for either framework:

```
func adjustFontSize(withMax max:CGFloat, forTextStyle textStyle:
UIFont.TextStyle) -> UIFont {
    let adjustedFont = UIFont.preferredFont(forTextStyle: textStyle)
    let realFontSize = adjustedFont.pointSize

    if realFontSize > max {
        return UIFont.systemFont(ofSize: max)
    } else {
        return adjustedFont
    }
}
```

Using this, you can vend a font a few different ways:

1. In a view controller, you can respond to the trait collection changing and then update any text control fonts as needed.
2. Or, in a custom text control, use Combine to listen for the content size preference changes and use the same approach to change font sizes.
3. In SwiftUI, you can use Combine and the `.onReceive(_ : perform:)` modifier to get content size changes and assign to `@State` variables to update font sizes.

Along a similar train of thought, you can also declare an upper and lower bound for a text or font style to use for Dynamic Type. This is useful to keep your text as dynamic as possible and lean away from designs that say "Keep this at 40 Points" and instead it

changes the dynamic to "Keep this no bigger than .accessibility1 but no smaller than .large" – and that's much more ideal for accessibility purposes.

In SwiftUI, the font is derived from the range you specify, allowing you to skip the font definition when you use these ranges. They are applied via the `dynamicTypeSize` modifier:

```
// SwiftUI -> Xcode -> DynamicTypeFig3View.swift

struct DynamicTypeFig3View: View {
    var body: some View {
        VStack {
            Text("No smaller than accessibility1")
                .dynamicTypeSize(.accessibility1...)
            Text("No bigger than Large.")
                .dynamicTypeSize(...DynamicTypeSize.large)
            Text("No smaller than large but no bigger than
xxLarge.")
                .dynamicTypeSize(.large...)
                .dynamicTypeSize(...DynamicTypeSize.xxLarge)
        }
    }
}
```

In UIKit, it works a bit differently. You still need to apply a font and opt a text control into Dynamic Type – but then the ranges will kick in from there:

```
// UIKit -> Xcode -> DynamicTypeFig6ViewController.swift

class DynamicTypeFig6ViewController: BaseSampleViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        let stack = UIStackView()
        stack.axis = .vertical
```

```
        let label1 = UILabel(frame: .zero)
        label1.font = UIFont.preferredFont(forTextStyle: .callout)
        label1.adjustsFontForContentSizeCategory = true
        label1.text = "No smaller than accessibilityLarge"
        label1.minimumContentSizeCategory = .accessibilityLarge
        label1.numberOfLines = 0

        let label2 = UILabel(frame: .zero)
        label2.font =
UIFont.preferredFont(forTextStyle: .largeTitle)
        label2.adjustsFontForContentSizeCategory = true
        label2.text = "No bigger than Large."
        label2.maximumContentSizeCategory = .large
        label2.numberOfLines = 0

        let label3 = UILabel(frame: .zero)
        label2.font = UIFont.preferredFont(forTextStyle: .headline)
        label3.adjustsFontForContentSizeCategory = true
        label3.text = "No smaller than large but no bigger than
xxLarge."
        label3.minimumContentSizeCategory = .large
        label3.maximumContentSizeCategory = .extraExtraLarge
        label3.numberOfLines = 0

        stack.addArrangedSubview(label1)
        stack.addArrangedSubview(label2)
        stack.addArrangedSubview(label3)

        view.addSubview(stack)
        stack.frame =
view.safeAreaLayoutGuide.layoutFrame.insetBy(dx: 2.0, dy: 2.0)
    }
}
```

These properties are available at the `UIView` level, which means you could (in this example) set the ranges on the stack view instead of each label assuming that the requirements were all the same:

```
stack.minimumContentSizeCategory = .large
```

Even so, you could still have different ranges used in each label and those would still take precedence over the range you applied to the parent view, making it a flexible API.

If you get stuck deep in a view hierarchy with multiple ranges set, you can also leverage the string property `appliedContentSizeCategoryLimitsDescription` to get a nice readout of all of the ranges currently set. It would look, in our example, like this if we used it for `label3`:

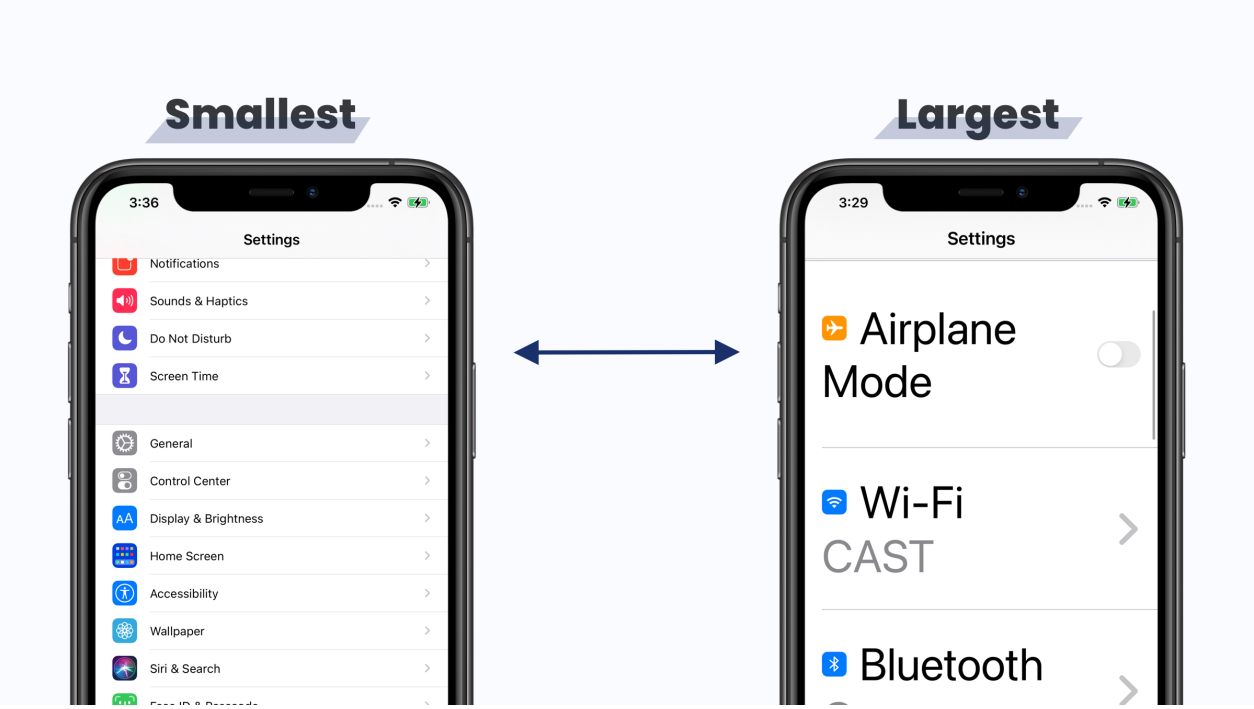
```
 3. <UIView:0x13b112930>: L <= (none->)L <= XXL
 2. <UIStackView:0x138f20310>: L <= (L->)L <= XXL
 1. <UILabel:0x138f23540>: L <= (L->)L <= XXL
--> L
```

No matter the method, be sure to allow the largest font size possible.

Efficient Testing

Testing Dynamic Type is paramount to ensuring a great experience. Thankfully, Xcode and iOS itself make the process straightforward and there are a number of ways to do it whether you are running on device or using the simulator.

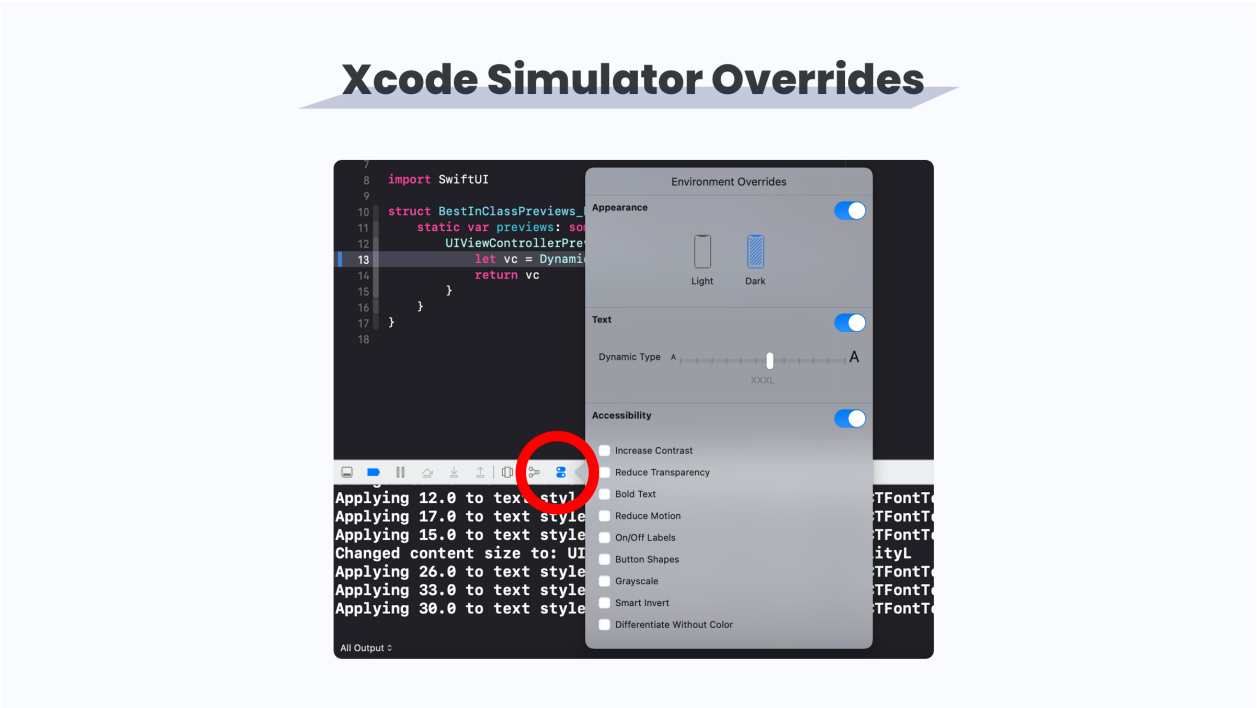
No matter which of the following methods you use, it helps to follow a "Lower Bound Upper Bound" rule. That is, test your app with a very small Dynamic Type setting, and then test with the largest option. If your app handles both gracefully, anything in the middle will do just fine as well.



When your app can handle the smallest and largest sizes, anything within that spectrum will likely display just fine.

Use Simulator Overrides

Starting with iOS 13 and the introduction of Dark Mode, Xcode included a way to tweak certain trait environment variables on the fly:



When your app is running, you can use this to change the Dynamic Type settings as you see fit.

Add Text Preferences to Control Center

You can also add a Dynamic Type setting to Control Center. Navigate to the Settings app -> Control Center -> Text Size. Now you can swipe down from the top right at any point to change Dynamic Type settings. This is a common way for iOS users to change font size, and it's a big reason why you should focus on the "dynamic" part of the API as we've discussed in this chapter. If a user changes their font size and it's not reflected immediately in your app until the view is reloaded, that's a problem.

This method is also great to get some inspiration from other apps. Survey how they handle similar interfaces such as your own with Dynamic Type. How do they look with it turned all the way down or up? Apple leads the way here, and if you find your-

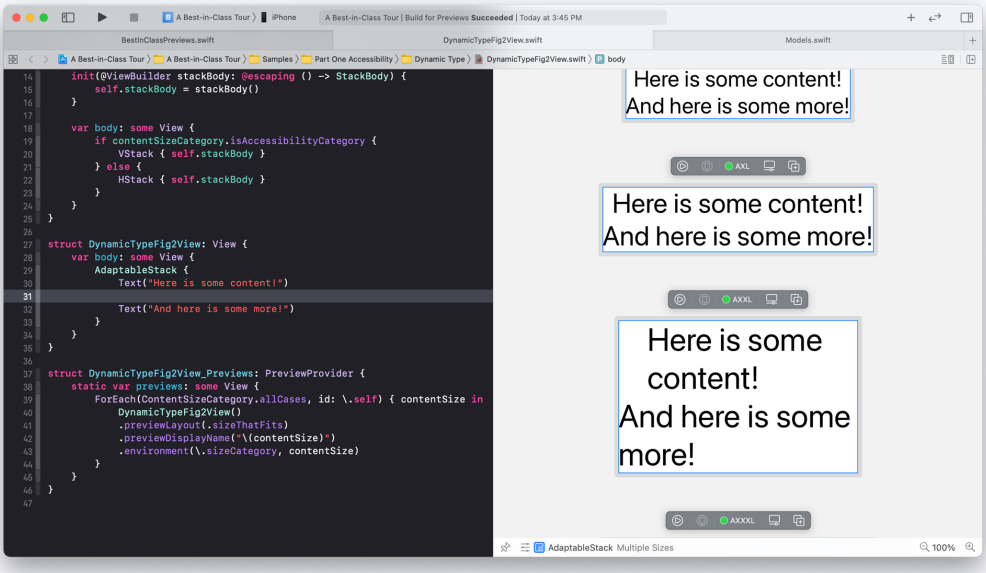
self "stuck" design wise with Dynamic Type, visiting one of their stock apps is usually a good place to start.

Xcode Previews

Using Xcode Previews and its `PreviewProvider` protocol, we can have a SwiftUI view display at all Dynamic Type sizes as expressed by its corresponding `ContentSizeCategory`:

```
// UIKit -> Xcode -> DynamicTypeFig2.view
struct DynamicTypeFig2View_Previews: PreviewProvider {
    static var previews: some View {
        ForEach(ContentSizeCategory.allCases, id: \.self)
{ contentSize in
            DynamicTypeFig2View()
            .previewLayout(.sizeThatFits)
            .previewDisplayName("\(contentSize)")
            .environment(\.sizeCategory, contentSize)
        }
    }
}
```

Using our `AdaptableStack` we created earlier, we can now see what all of the Dynamic Type sizes will display as *and* see that our stack turns to using a horizontal axis when an accessibility size is used:


A Best-in-Class Tour
iPhone
A Best-in-Class Tour | Build for Previews Succeeded | Today at 3:45 PM
BestInClassPreviews.swift
DynamicTypeFig2View.swift
Models.swift
A Best-in-Class Tour
A Best-in-Class Tour
Samples
Part One Accessibility
Dynamic Type
DynamicTypeFig2View.swift
body
init(@ViewBuilder stackBody: @escaping () -> StackBody) {
self.stackBody = stackBody()
}
var body: some View {
if contentSizeCategory.isAccessibilityCategory {
VStack { self.stackBody }
} else {
HStack { self.stackBody }
}
}
}
struct DynamicTypeFig2View: View {
var body: some View {
AdaptableStack {
Text("Here is some content!")
Text("And here is some more!")
}
}
}
struct DynamicTypeFig2View_Previews: PreviewProvider {
static var previews: some View {
ForEach(ContentSizeCategory.allCases, id: \.self) { contentSize in
DynamicTypeFig2View()
.previewLayout(.sizeThatFits)
.previewDisplayName("\(contentSize)")
.environment(\.sizeCategory, contentSize)
}
}
}
Here is some content!
And here is some more!
AXL
Here is some content!
And here is some more!
AXXL
Here is some
content!
And here is some
more!
AXXL
AdaptableStack Multiple Sizes
100%

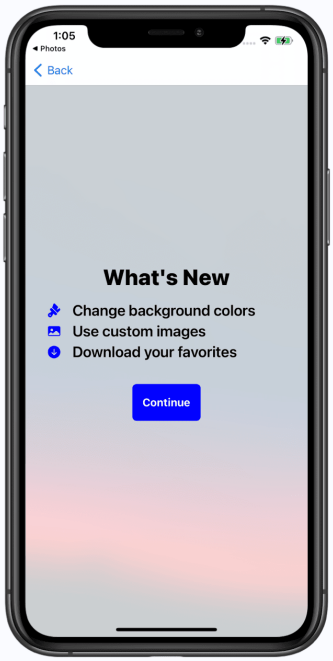
Three Key Takeaways

1. Dynamic Type is one of the most important accessibility APIs in the iOS ecosystem, it's critical to support.
2. Doing so requires a team wide effort, but thinking about it upfront can make the process easier.
3. Fortunately, there are several APIs to help you support Dynamic Type and just as many ways to test it.

Adaptable Interfaces

When I first began creating and designing iOS apps, I'd iterate and iterate over a particular design until I was happy with it. Dynamic Type didn't exist yet, so choosing a particular font size and sticking with it was more common. Getting the spacing just right could be challenging, but the advent of Auto Layout made things easier.

I'd spend time laboring over the tiny details, as I still do today. But when it was finally done, it was always rewarding. I would get it looking *just* right, exactly how I wanted it – and then move on. Take, for example, a "What's New" screen. Perhaps I might create one which looks something like this:



A simple view to show what's new in a fictional app's update.

Early on in my career, I'd works towards a functional point with that design and carry on. But years later, I've since developed for an eye for accessibility considerations — mostly by learning the hard way and unfortunately at the expense of some users who couldn't use my apps in the way they needed to.

So today, and in the case of our screen above, we need to train ourselves to look at each screen and consider several accessibility implications. Instead of moving on once the design is functional, we'll make sure the interface adapts to accessibility needs, and isn't simply optimized for the design spec.

Developing an Eye for Accessibility

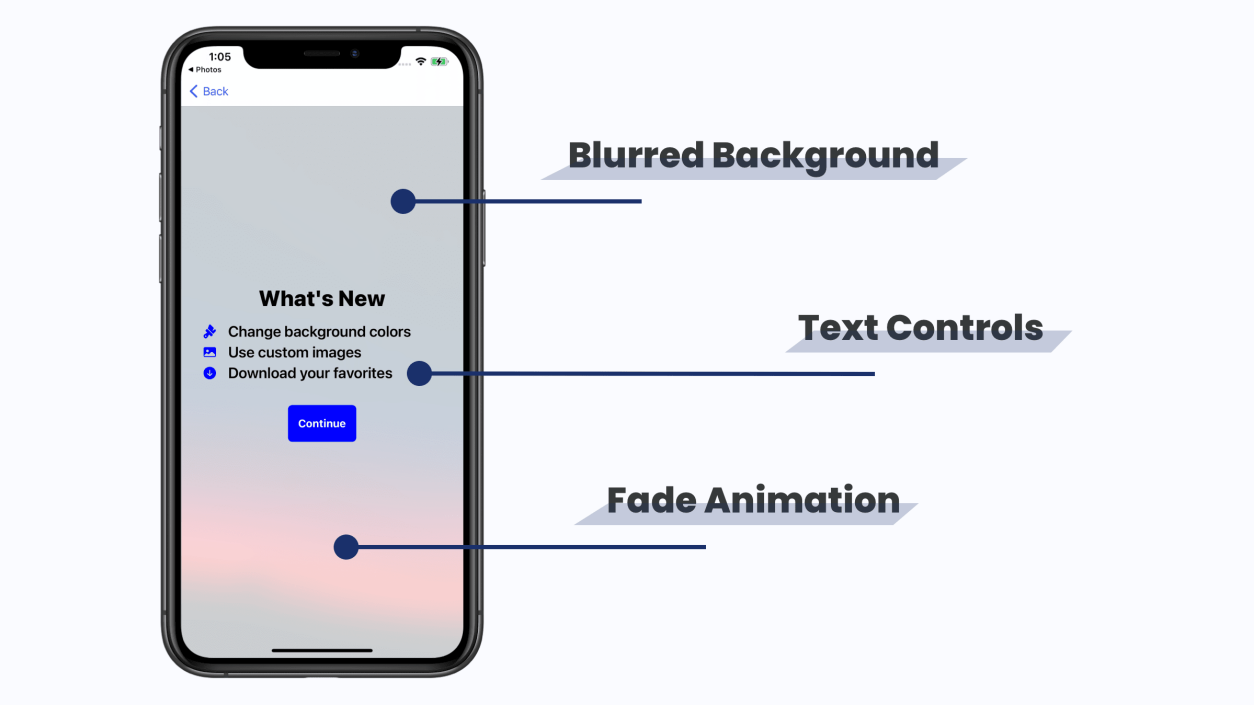
To make sure our screen stays accessible, readable and reliable for all users, we need to address a few parts of the interface. Zooming out a little bit, think over the interface

and see if you have any initial thoughts on what we might need to retool or pay a little bit more attention to in our "What's New" view.

What can you spot?

Looking at this, even if I hadn't yet viewed the code, here's what I can think of. We've got:

1. A blurred background.
2. Animations (run the sample project to see this one occurring).
3. Text.

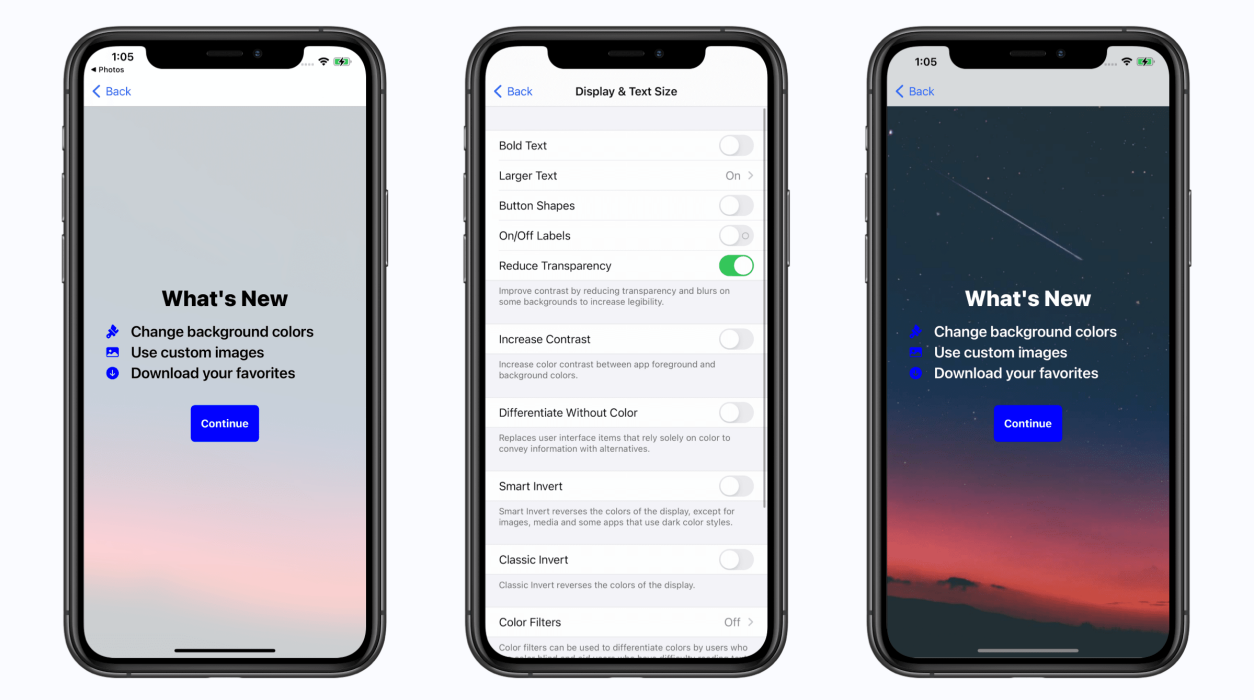


Audit views for accessibility needs by thinking about each individual element.

All of these could have accessibility concerns that we need to address. In the case of a blurred background, we'd need to consider the accessibility setting `isReduceTransparencyEnabled`. If enabled, we'd want to make the blurred background

opaque or hidden. Similarly, in the case of the fade in animation, some users may prefer less motion in their apps. There, we could look at `isReduceMotionEnabled`. Finally, as discussed in the previous chapter – where there is text, there should be Dynamic Type support.

Beyond simply responding to those, we need to be *reactive* to those settings, too. Much like the user shouldn't have to reload a view or force quit your app for Dynamic Type settings to be reflected, they should also see any of these settings be applied in the same way. If the user goes into settings after this view was loaded and enabled reduced transparency, then the view should be updated when they get back. They shouldn't need to restart your app for that.



When accessibility settings are toggled, those preferences should be reflected immediately within your app.

If they did, that's only meeting them half way. And not in a good sense, either. Fortunately, bridging all of those gaps is not as difficult as it may seem at first. With a few

sound patterns implemented early on, you can setup a framework to make sure these areas are covered by paying the upfront cost once engineering wise.

Adapting to Accessibility Needs

Let's tackle each of these areas of need one by one. If you open up `AdaptableInterfacesFig1ViewController.swift` for the UIKit version or `AdaptableInterfacesFig1View.swift` for the SwiftUI version, you'll see that in their current state neither one respects any of these settings. That's to say, if you had reduce transparency enabled – the blur views would still show, etc.

Let's fix each issue one by one. Our approach in many of these will be the same:

1. Query the relevant accessibility settings for initial setup.
2. Respond to its notification that it was toggled in real time.

To house logic for things like this, we'll create a simple object that vends accessibility settings and notifies us of changes. We'll start with support for handling accessibility users who prefer reduced transparency[15]:

```
// Xcode -> AccessibilitySettings.swift
class AccessibilitySettings {
    var prefersReducedTransparency: Bool {
        get {
            return UIAccessibility.isReduceTransparencyEnabled
        }
    }
}
```

[15] We'll cover why this in its own class a little later on.

While that tackles initial setup, perhaps you might be inclined write some code like this:

```
if UIAccessibility.isReduceTransparencyEnabled {
   // Dont use a blurred view
} else {
   // Use a blurred view
}
```

While it may work on the initial load, it doesn't pass our test of being reactive and adaptable. We want this type of thing to work on the fly.

Combine can help us out:

```
class AccessibilitySettings {
    var prefersReducedTransparency: Bool {
        get {
            return UIAccessibility.isReduceTransparencyEnabled
        }
    }

    let reduceTransparencyPublisher =
NotificationCenter.default.publisher(for:
UIAccessibility.reduceTransparencyStatusDidChangeNotification)
}
```

With that, there's a publisher in place that we can respond to when the setting is changed by the user. Now let's put it together for the first fix, the blurred background.

The Blurred Background

When a user prefers opaque backgrounds, `isReduceTransparencyEnabled` will be enabled. Under iOS' settings, it's listed under Settings -> Display & Text Size -> Reduce Transparency. In fact, you'll find several accessibility options there, almost all of which have a corresponding boolean property in `UIAccessibility.h`.

The UIKit Fix

If we look at the current implementation, the issue is obvious. A `UIVisualEffectView` does not respond to this accessibility setting out of the box, so we've got to build that functionality in ourselves. Currently, we just display a visual effect view and that's it:

```
// UIKit -> Xcode -> AdaptableInterfacesFig2ViewController.swift

// Blur
let blurView = UIVisualEffectView(effect:
UIBlurEffect(style: .extraLight))
blurView.autoresizingMask = [.flexibleWidth, .flexibleHeight]
blurView.frame = view.bounds
```

We'll create a custom subclass and leverage our accessibility object we created above to show the blur when `UIAccessibility.isReduceTransparencyEnabled` is false, and hide it when it's true. And again, I likely sound like a broken record here, but we need to ensure that it responds to this setting in real time:

```
// Xcode -> UIKit -> AdaptableInterfacesFig2ViewController.swift

class AdaptableBlurView: UIView {
    private let axSettings = AccessibilitySettings()
    private var sub: AnyCancellable? = nil
    private var fxView: UIVisualEffectView!
    var effect: UIVisualEffect? = nil

    init(withEffect effect:UIVisualEffect) {
        super.init(frame: .zero)
        self.effect = effect
        self.fxView = UIVisualEffectView(effect: self.effect)
        self.sub = axSettings.reduceTransparencyPublisher.sink
{ [weak self] _ in
            guard let self = self else { return }
            self.updateBlurViewAppearance()
```

```
        }
        addSubview(fxView)
        fxView.frame = bounds
        fxView.autoresizingMask = [.flexibleWidth, .flexibleHeight]
        updateBlurViewAppearance()
    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    private func updateBlurViewAppearance() {
        if self.axSettings.prefersReducedTransparency {
            self.fxView.effect = nil
        } else {
            self.fxView.effect = self.effect
        }
    }
}
```

This class simply is a `UIView` with a visual effect as its sole subview. But, it queries the reduce transparency setting upfront *and*, using Combine, changes it as the user toggles the setting.

If you run this on your device, open up this view and go to settings to toggle the reduced transparency preference, you'll see it update instantly when the app returns to the foreground. Exactly the experience we're after.

As an aside, you may find when your interface changes you likely need to update other parts as well. With the blur view gone, there isn't enough contrast in the text, so here we should also change the font color to address that.

We can reuse our accessibility class that we used in the visual effect subview at the controller level to respond to the setting there as well:

```
// Xcode -> UIKit -> AdaptableInterfacesFig2ViewController.swift
```

```
axSettings.reduceTransparencyPublisher
    .sink { [weak self] _ in
        guard let self = self else { return }
        let textColor: UIColor =
self.axSettings.prefersReducedTransparency ? .white : .black
        whatsNewLabel.textColor = textColor
        benefitStackView.arrangedSubviews.forEach{
            if let view = $0 as? BenefitsView {
                view.textLabel.textColor = textColor
            }
        }
    }
    .store(in: &subs)
```

The SwiftUI Fix

SwiftUI provides the same value via an environment value. These are propagated throughout a view hierarchy and make fixes like this easy to implement in a SwiftUI codebase. Using it, we don't need to use Combine or listen to changes ourselves at all – it's as simple as using control flow.

First, we'll query the environment for the reduced transparency setting:

```
// Xcode -> SwiftUI -> AdaptableInterfacesFig2View.swift ->
WhatsNewViewV2.swift

@Environment(\.accessibilityReduceTransparency) var
prefersReduceTransparency
```

This value will automatically update when the user changes the relevant setting in iOS' Settings app, which means SwiftUI will update its `body` and corresponding interface elements affected by it. So we'll add a simple `if` statement around the blur view:

```
// Xcode -> SwiftUI -> AdaptableInterfacesFig2View.swift ->
WhatsNewViewV2.swift

if !prefersReduceTransparency {
```

```
    VisualEffectView(visualFX: UIBlurEffect(style: .extraLight))
}
```

Now, the view will hide when the user prefers reduced transparency and show when they allow it. As with our UIKit example, this also occurs immediately so the user never has to reload our view or restart our app.

The Animations

Next up, there is a fade animation that occurs when the view loads along with a slight delay before it begins (which accounts for about the same time UIKit takes to push a controller onto the navigation stack, which is how this is presented). If users prefer less motion in their apps, we'll forgo this altogether.

The UIKit Fix

Here, we'll extend our accessibility object further to query for reduced motion:

```
// Xcode -> AccessibilitySettings.swift

var prefersReducedMotion: Bool {
    get {
        return UIAccessibility.isReduceMotionEnabled
    }
}
```

Keep in mind, if your coding style or architecture is better suited by checking these values directly — by all means, do so. In my experience, my accessibility needs and logic tend to be similar and are needed all throughout the app.

For that reason, I've found a lot of value in keeping similar functionality within an object such as this. For example, we used the same publisher twice in the reduced transparency fix (one in the subview for the blur view, and once more in the view controller housing it).

Either way you go, in the cases of animations like this we need to ensure two things if this preference is enabled:

1. The animation does not occur.
2. The state of the interface is such that the animation has *ended* to ensure it looks right if the user prefers reduced animations.

Due to the second point, I always setup my interface in the "end" state of the animation. Then, if the user allows motion, I then mutate the interface elements to reflect the "beginning" state of the animation. I've had too many times where I did the opposite, and shipped bugs with invisible elements if the reduced motion preference was active.

Following our previous pattern, in our "What's New" view we only fire off the animations, and change the elements to hide and fade back in, if motion is allowed:

```
// Xcode -> UIKit -> AdaptableInterfacesFig2ViewController.swift

if !axSettings.prefersReducedMotion {
    blurView.effect = nil
    let dimmedViews = [whatsNewLabel, benefitStackView, button]
    dimmedViews.forEach{ $0.alpha = 0.0 }

    UIView.animate(withDuration: 1.0, delay: 0.5,
options: .curveEaseOut) {
        blurView.effect = UIBlurEffect(style: .extraLight)
        dimmedViews.forEach{ $0.alpha = 1.0 }
    } completion: { _ in }
}
```

Now, we're all set. The animation plays if the user has the accessibility preference disabled and its skipped (and the interface is already at the "end" state) if it's on.

Also, some developers prefer to hang another boolean off of objects like these to promote readability by checking for a value that's true, as opposed to here where we've looked for a value to be false. If that's you or your organization, I encourage you to follow those patterns and add a property to allow for that coding style:

```
var prefersMotion: Bool {
    get {
        return !UIAccessibility.isReduceMotionEnabled
    }
}
```

The SwiftUI Fix

Much like we fixed the blur view above in SwiftUI, we'll turn to the environment once more to query the accessibility preference:

```
// Xcode -> SwiftUI -> AdaptableInterfacesFig2View.swift

@Environment(\.accessibilityReduceMotion) var prefersReducedMotion
```

Since our SwiftUI view, `WhatsNewView`, leverages modifiers for the animations and doesn't have the logic housed within the view itself, we can apply the fix inside `AdaptableInterfacesFig2View.swift`. There are a number of ways to go about this, and the path of the least resistance would be to simply check the preference and return two different views:

```
if (prefersReducedMotion) {
    WhatsNewView()
} else {
    WhatsNewView()
        .opacity(hidden ? 0.0 : 1.0)
        .animation(.easeInOut(duration: 1.0))
        .onAppear {
            DispatchQueue.main.asyncAfter(deadline: .now() + 0.5) {
                self.hidden = false
```

```
            }
        }
}
```

However, I would not recommend this approach as it means you've got two view instances to manage, read and maintain. While this chapter isn't over how to write sensible SwiftUI code, I do think it pays to think about how to manage conditional modifiers in this case because you'll run into this quite a lot with accessibility programming in SwiftUI.

For cases like this, leverage inert modifiers and inline boolean checks to maintain the same set of modifiers:

```
// Xcode -> SwiftUI -> AdaptableInterfacesFig2View.swift
WhatsNewView2()
    .opacity(isHidden && !prefersReducedMotion ? 0.0 : 1.0)
    .animation(.easeInOut(duration: 1.0))
    .onAppear {
        DispatchQueue.main.asyncAfter(deadline: .now() + 0.5) {
            self.isHidden = false
    }
```

Now, the animation only occurs if the view is hidden *and* the user is okay with having motion and animations occurring. If they don't, the view is already at 1.0 opacity, so the animation has no effect.

For cases like these, you may be tempted to write an `if` extension which evaluates a condition and applies modifiers if it's true:

```
// Xcode -> UtilityExtensions.swift

extension View {
    @ViewBuilder func `if`<Content: View>(_ shouldModify: Bool,
modified: (Self) -> Content) -> some View {
        if shouldModify {
```

```
            modified(self)
        } else {
            self
        }
    }
}
```

With that in place, it would result in code like this:

```
WhatsNewView()
    .if(prefersReducedMotion == false) { view in
        view
        .opacity(hidden ? 0.0 : 1.0)
        .animation(.easeInOut(duration: 1.0))
        .onAppear {
            DispatchQueue.main.asyncAfter(deadline: .now() + 0.5) {
                self.hidden = false
            }
        }
    }
```

For most cases, I would *not* recommend this approach.

Using a conditional modifier like this is typically not a good idea due to the way identity works in SwiftUI. This means we have two different view types, which can lead to a host of problems that could be difficult to debug if you weren't aware of this.

For example, if you wanted something like an animation to occur, code like this wouldn't result in what you might expect:

```
@State private var showRed: Bool = false

var body: some View {
    Text("Test)
        .if(showRed) { view in
            view.foregroundColor(.red)
        } else {
            view.foregroundColor(.black)
```

```
        }
}
```

Using a conditional modifier would break an animation and default to a fade in and out. As we said above, the branching modifier means there are two different view values here, and SwiftUI cannot interpolate between those two types smoothly - because they are, well, two *different* types.

So, for the most part, you'll want to opt for approaches like this:

```
@State private var showRed: Bool = false

var body: some View {
    Text("Test)
        .foregroundColor(showRed ? .red : .black)
}
```

Or, use inert modifiers where you pass nil for the false condition:

```
@State private var shouldShowWider: Bool = false

var body: some View {
    Text("Test)
        .frame(width: shouldShowWider ? 300 : nil)
}
```

Either of those approaches is the path you should opt to take in SwiftUI to maintain identity. However, if you understand the implications of returning two different views in full – then you know there are cases where this is exactly what you want and is, sometimes, unavoidable.

For those new to SwiftUI, it can result in hard to fix bugs. Sticking with the same set of modifiers is something that I recommend developers strive for. And, that's what we've done with our accessibility fix.

As with UIKit, now the animation only runs when the accessibility preference is not active. That takes care of the blurring and animations, let's turn to the text next.

The Font Size

Currently, the text is not respecting Dynamic Type settings (something we learned about in-depth in the previous chapter). Here, we simply need to adjust a few lines of code.

The UIKit Fix

If we visit `BenefitView2`, the issue is obvious:

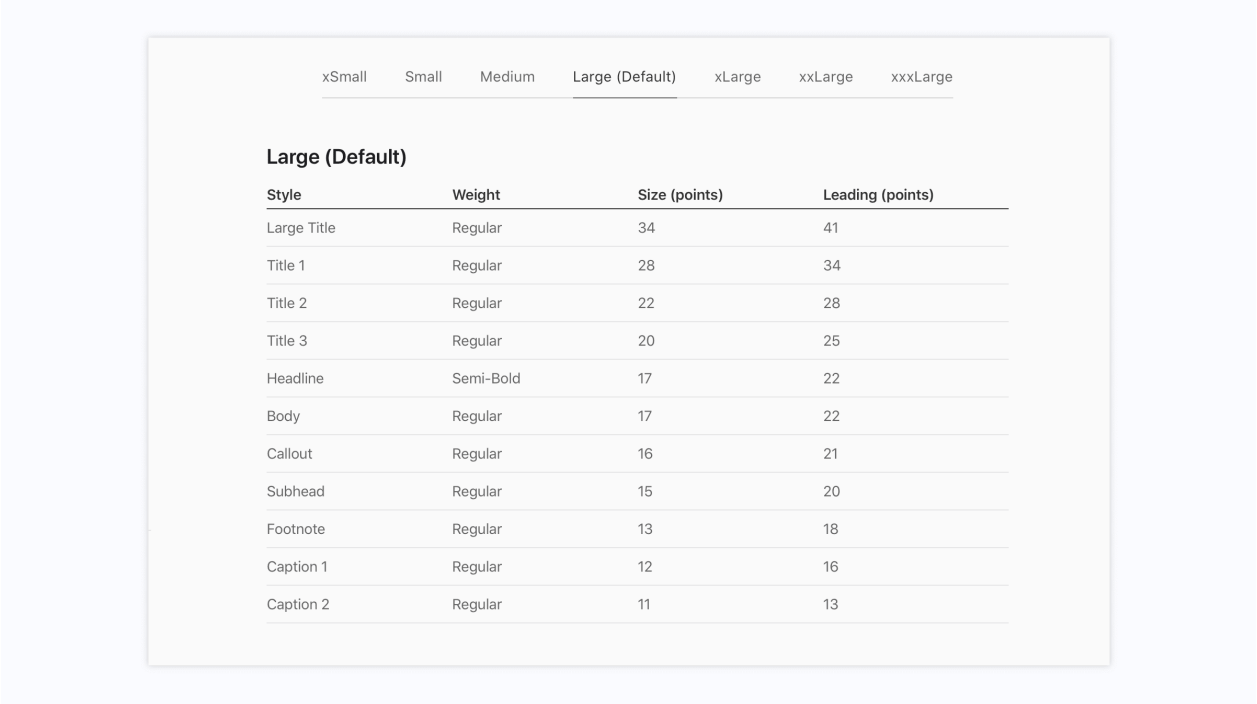
```
// Xcode -> UIKit -> AdaptableInterfacesFig2View.swift ->
BenefitView2

let whatsNewLabel = UILabel(frame: .zero)
whatsNewLabel.translatesAutoresizingMaskIntoConstraints = false
whatsNewLabel.text = "What's New"
whatsNewLabel.font = UIFont.systemFont(ofSize: 32, weight: .heavy)
whatsNewLabel.textAlignment = .center
```

A hardcoded font size will not respond to any content size preference changes. We'll switch that out for a similar font which respects Dynamic Type:

```
// Xcode -> UIKit -> AdaptableInterfacesFig2View.swift ->
BenefitView2

textLabel.adjustsFontForContentSizeCategory = true
textLabel.font = UIFont.preferredFont(forTextStyle: .title2)
```

Also note that in this case, the font weight goes from semibold to a regular weight. Persisting Dynamic Type settings and a custom font weight is a more involved endeavor in UIKit than SwiftUI, but you can use the strategy we outlined in the "Dynamic Type" chapter under "Custom Font Weights and Dynamic Type in UIKit" to keep things

consistent. Apple also has a reference guide which is helpful to match a hardcoded font size to a similar font family:



Style	Weight	Size (points)	Leading (points)
Large Title	Regular	34	41
Title 1	Regular	28	34
Title 2	Regular	22	28
Title 3	Regular	20	25
Headline	Semi-Bold	17	22
Body	Regular	17	22
Callout	Regular	16	21
Subhead	Regular	15	20
Footnote	Regular	13	18
Caption 1	Regular	12	16
Caption 2	Regular	11	13

This table is useful to match font sizes and weights to a Dynamic Type representation.

With that, our font will now adjust to Dynamic Type settings. Depending on your design preferences, you could also intercept the content size changes and change the stack views to display on a vertical axis too. See "Flipping Interface Axis" in the "Dynamic Type" chapter for more details on how to accomplish this.

The SwiftUI Fix

It's the same story in SwiftUI, as we've got a hardcoded font size[16]:

```
// Xcode -> SwiftUI -> AdaptableInterfacesFig2View.swift ->
BenefitsViewV2.swift
```

16 Note that I've created copes of these views, such as BenefitsView and BenefitsView2, to have an example where it "works" and doesn't "work" for reference.

```
Text(benefitText)
    .font(.system(size: 22, weight: .semibold))
```

We'll make a simple switch:

```
// Xcode -> SwiftUI -> AdaptableInterfacesFig2View.swift ->
BenefitsViewV2.swift
Text(benefitText)
    .font(.title2)
    .fontWeight(.semibold)
```

And we're done!

While these text examples may seem strikingly simple (which, in my opinion, is a good thing – software engineering is complicated enough), you'd be surprised at how easy they are to sneak into production. Just testing a quick design change here, patching up a messy merge there – and before you know it, it's shipped. Always take care to double back and check things like this, Dynamic Type support is an essential ingredient to a best-in-class iOS app.

It Takes Time

If you've reached the end of this chapter and are slightly discouraged, thinking something like "I would've never had caught these issues!", don't be discouraged. These things take time. You don't know what you don't know. It's only by learning about these APIs, by reading a book such as this one or some other means, that you begin to develop the muscle memory necessary to tackle scenarios like this.

In short, it takes time. And that's perfectly fine. In fact, there's even more that we could've done here such as adding a scroll view if the text was too large or responding to low power mode. As I've mentioned in other parts of the book, accessibility

programming is a world of discovery and opportunity. Nobody knows it all, but keeping a healthy empathy for the end users will help you learn what you need to do and where you might be falling short.

Three Key Takeaways

1. Adaptable interfaces mean you are constantly aware of the needs of *every* user, and that one interface likely isn't ideal for all of them.
2. Because of that, paying the upfront "cost" of design and development is the best way to create your apps.
3. By leveraging certain patterns and techniques, creating adaptable interfaces becomes much easier and realistic.

Posting and Consuming Accessibility Notifications

Accessibility notifications can be used to accomplish a large numbers of tasks to make life easier for users of your app. From focusing a certain element, announcing interface changes to being notified when text is bold – there are virtually endless avenues available to us which we can use to adapt our interfaces every which way.

In this chapter, we're going to look at every accessibility notification that you can post. Many of these are built around UIKit, but most all of them have SwiftUI implications as well. However, some of these are exposed to SwiftUI by way of environment variables making the need to respond to a notification less useful. We'll learn how to respond to these types of preferences in the next chapter, User Preferences in SwiftUI.

The notifications are grouped into seven different categories:

1. **Interface changes:** These notifications help us adapt our interface to preferences or control interactions.
2. **VoiceOver:** All of these help you create an excellent VoiceOver experience.
3. **Text**: These text based notifications help you respond to accessibility preferences that deal specifically with text and reading.
4. **Colors:** Colors have massive accessibility implications, so all of these help us respond to various needs around color preferences.
5. **Assistive Technologies:** This one is kind of a grab bag of accessibility notifications, but they'll mainly help us know when certain technologies are in use.
6. **Audio and Speech:** These vend audio events that occur, such as when a user connects a hearing aid.
7. **Motion:** Finally, these notifications announce preference changes around motion.

There are *a lot* of notifications here. To get value out of each of them, I'll list out what the notification is meant for and how it can be used. There are two ways you'll be handling them. The first way is *responding* to any accessibility notification that comes in. Secondly, you can also manually post these notifications to give the system a hand for accessibility purposes.

A Note on Handling Notifications

Each of these are vended via `NotificationCenter`, and its job is to dispatch information to registered observers. Found all throughout iOS, handling notifications is an

important skill to have for developers of all skill levels. Consuming notifications comes down to a matter of preference. How do you want to observe them?

Consider this notification name:

```
let didBecomeActive = UIApplication.didBecomeActiveNotification
```

To observe it, you could go the selector route – which is more of a carryover pattern from when Objective-C and its message sending architecture were more prominent:

```
let didBecomeActive = UIApplication.didBecomeActiveNotification
let defaultCenter = NotificationCenter.default
let selector = #selector(handleDidBecomeActive(withNotification:))

defaultCenter.addObserver(self,
                          selector: selector,
                          name: didBecomeActive,
                          object: nil)

// In the same class, file or somewhere else...
@objc func handleDidBecomeActive(withNotification note:Notification)
{
    // Respond accordingly
}
```

Or, you could use a block based approach. While convenient, it's also prone to retain cycles:

```
let didBecomeActive = UIApplication.didBecomeActiveNotification
let defaultCenter = NotificationCenter.default

defaultCenter.addObserver(forName: didBecomeActive,
                          object: nil,
                          queue: .main) { note in
    // Respond accordingly
}
```

Finally, you can use Combine – which is the method I tend to favor and personally consider the most modern, maintainable and team-friendly approach:

```
private var subs: [AnyCancellable] = []

let didBecomeActive = UIApplication.didBecomeActiveNotification
let defaultCenter = NotificationCenter.default

defaultCenter.publisher(for: didBecomeActive, object: nil)
    .sink { notification in
        // Respond accordingly
    }
    .store(in: &subs)
```

For this chapter, I'll be using the Combine method to demonstrate these notifications. I find it advantageous to the other approaches due to its modern API, rich operators to transform notification data and its overall flexibility. In particular, I have an `AXPublisher` struct whose only job is to vend these notifications. In the examples below which involve *consuming* notifications posted on behalf of the system, it'll be used.

Feel free to open it up in the sample project to view it, but here is an idea of how it works:

```
// Xcode -> AXPublisher.swift

struct AXPublisher {
    let switchControlStatusChanged =
NotificationCenter.default.publisher(for:
UIAccessibility.switchControlStatusDidChangeNotification, object:
nil)

    //... and other accessibility notifications
}
```

```
// Later on, in a view controller or similar...
private let axPublisher = AXPublisher()

let switchChangedSub = axPublisher.switchControlStatusChanged.sink {
_ in
    if UIAccessibility.isSwitchControlRunning {
        // Perform any switch control considerations
    } else {
        // Or, undo them
    }
}
```

Any approach works, so it's up to you to consider which makes the most sense for your codebase and promotes maintainability.

Posting Accessibility Notifications

On the other hand, you'll also find yourself in situations where you need to post an accessibility notification on your behalf. While most notifications are just shipped up and posted via `NotificationCenter`:

```
NotificationCenter
.default.post(name: .NSUndoManagerWillUndoChange, object: nil)
```

…accessibility notifications work slightly differently in that they have a dedicated function found in `UIAccessibility` to do the job:

```
public static func post(notification: UIAccessibility.Notification,
argument: Any?)
```

You'll notice the first parameter is a specialized kind of `Notification` – an accessibility notification. This helps strongly type notifications that are specifically meant to aid assistive processes, and all of them are listed at `UIAccessibilityConstants.h`.

In fact, you can't even post them by any other means as the default Notification Center API uses notifications typed to `NSNotification`.

Some of these notifications might take a parameter as well, such as `.screenChanged`, to help guide any accessibility processes along.

Now, let's take a look at all of these notifications. Again, I'll share what the notification is and a short comment on how you can use it. Try not to become overwhelmed with the large number of notifications. The key here is to have a passing awareness of them, so when you hit a spot where you need one of them you'll remember that notification X or Y exists, even if you don't remember the finer details of its implementation.

Let's get started with the interface changes category.

Interface Changes

Screen Changes

```
static var screenChanged: UIAccessibility.Notification
```

Type

Manually posted.

Purpose

Post these when a significant change occurs in the interface, like changing a tab in a web browser, or a new view takes over a good portion of the screen. You can pass the interface element VoiceOver should focus on, optionally.

Example

```
// Consider a custom tabbing UX, you might post this right when the
user changes to a new one.
```

```
UIAccessibility.post(notification: .screenChanged, argument:
newBrowerTab)
```

Layout Changes

```
static var layoutChanged: UIAccessibility.Notification
```

Type

Manually posted.

Purpose

Unlike the previous notification, which is meant for major context changes in the layout, this one is useful for when the layout of the currently presented view changes. For example, an element is hidden or moved in some way.

Example

```
// Consider a form that had a birthday entry component that was
hidden, but some other action like selecting a toggle that your 18
years or older, made it appear.
UIAccessibility.post(notification: .layoutChanged, argument:
birthdaySelectionView)
```

Page Scrolled

```
static var pageScrolled: UIAccessibility.Notification
```

Type

Manually posted.

Purpose

After you implement and invoke `accessibilityScroll:`, then your scrolling action has (or at least, should be) completed. At that point, you'd also post this notification. Supply a string to inform the system of the new scroll position such as "Page 1 of 4" or similar.

Note that if you post the same string, then the accessibility engine will let the user know that the “page” or scroll bounds have been reached.

Example

```
// After accessibilityScroll has invoked...
UIAccessibility.post(notification: .pageScrolled, argument: "Page 1
of 2")
```

Switch Control Status Changes

```
static let switchControlStatusDidChangeNotification:
NSNotification.Name
```

Type

Consumed.

Purpose

Posted via UIKit to inform you of when a switch control is activated or deactivated. This is meant to be used in tandem with `UIAccessibility.isSwitchControl-Running` .

Example

```
let switchChangedSub = axPublisher.switchControlStatusChanged.sink {
_ in
    if UIAccessibility.isSwitchControlRunning {
        // Perform any switch control considerations
    } else {
        // Or, undo them
    }
}
```

Element Focus

```
static let elementFocusedNotification: NSNotification.Name
```

Type

Consumed.

Purpose

When assistive technologies focus an element, this notification gets posted along with the element that received focus.

Example

```
let focusedChanged = axPublisher.elementFocused
    .compactMap { $0.userInfo?
[UIAccessibility.focusedElementUserInfoKey] as? UIView}
    .sink { focusedView in
    // Handle focused view
}
```

Reduce Transparency Preference

```
static let reduceTransparencyStatusDidChangeNotification:
NSNotification.Name
```

Type

Consumed.

Purpose

When the user toggles the “Reduce Transparency” in Settings, this notification is posted. Meant to be used with `UIAccessibility.isReduceTransparencyEnabled` to determine what the user has the preference set to.

Example

```
let transparencyStatusChanged =
axPublisher.transparencyStatusChanged
    .sink { _ in
        if UIAccessibility.isReduceTransparencyEnabled {
            // Don't blur views
        } else {
```

```
            // Blur away
        }
}
```

VoiceOver

VoiceOver Status

```
static let voiceOverStatusDidChangeNotification: NSNotification.Name
```

Type

Consumed.

Purpose

This notification is posted anytime VoiceOver is stopped or started. Meant to be used with `UIAccessibility.isVoiceOverRunning` to determine if VoiceOver is on or off.

Example

```
let voiceOverStatusChanged = axPublisher.voiceOverStatusChanged
    .sink { _ in
        if UIAccessibility.isVoiceOverRunning {
            // Perhaps make a bottom custom modal and default UIKit
presentation
        } else {
            // Use custom navigation
        }
}
```

Announcement

```
static var announcement: UIAccessibility.Notification
```

Type

Manually posted.

Purpose

Post this notification, along with a string as the parameter which is the announcement itself, to let the accessibility engine know about things that update interface elements or effect the user experience in some way.

Example

```
// When you have a modal loading indicator or something similar
during a sign up network request...
UIAccessibility.post(notification: .announcement, argument: "Signing
up")

// When it's done, you can either use the same notification or focus
an element for VoiceOver elementFocusedNotification mentioned above.
```

Announcements Completed

```
static let announcementDidFinishNotification: NSNotification.Name
```

Type

Consumed.

Purpose

UIKit posts this notification when an announcement (posted on our behalf using the notification listed above, `announcement`) completes regardless if it was successful or not. It'll contain two keys in the user info dictionary which have the spoken announcement contents and a boolean indicating success or failure.

Example

```
let announcementFinished = axPublisher.announcementFinished
    .compactMap{ $0.userInfo }
    .sink { userInfo in
```

```
        guard let spokenText =
userInfo[UIAccessibility.announcementStringValueUserInfoKey] as?
String,
              let wasSuccessful =
userInfo[UIAccessibility.announcementWasSuccessfulUserInfoKey] as?
Bool else
            return
        }

        // Use spoken text, or more commonly perform
        // Some UI considerations when the announcement
        // Is finished and was successful
}
```

Text

Bold Text Preference

```
static let boldTextStatusDidChangeNotification: NSNotification.Name
```

Type

Consumed.

Purpose

When users change the Bold Text preference in Settings, UIKit posts this notification. Meant to be used with `UIAccessibility.isBoldTextEnabled` to determine if bold text is on or off.

Example

```
let boldTextChanged = axPublisher.boldTextStatusChanged
    .sink { _ in
        if UIAccessibility.isBoldTextEnabled {
            // Consider bolder or more constrasted interface
elements
```

```
        } else {
            // Regular display
        }
}
```

Closed Captioning Preference

```
static let closedCaptioningStatusDidChangeNotification:
NSNotification.Name
```

Type

Consumed.

Purpose

When users enable or set up the Closed Captioning preference in Settings, UIKit posts this notification. Meant to be used along `UIAccessibility.isClosedCaptioningEnabled` to determine if this feature is on or off.

Example

```
let closedCaptioningChanged =
axPublisher.closedCaptioningStatusChanged
    .sink { _ in
        if UIAccessibility.isClosedCaptioningEnabled {
            // If you've got any places where you have custom
            // Captioning capabilities, turn them on here
        } else {
            // Or, disable them
        }
}
```

Blinking Cursor Preference

```
static let
prefersNonBlinkingTextInsertionIndicatorDidChangeNotification:
NSNotification.Name
```

Type

Consumed.

Purpose

When users can toggle a setting to enable or disable blinking cursors. If you've created custom text rendering or entry, you should respect this setting. Meant to be used along `AccessibilitySettings.prefersNonBlinkingTextInsertion-Indicator` to determine if this feature is on or off.

Example

```
let closedCaptioningChanged =
axPublisher.prefersNonBlinkingCursorChanged
    .sink { _ in
        if
AccessibilitySettings.prefersNonBlinkingTextInsertionIndicator {
            // Disable blinking cursor for text entry
        } else {
            // Or, enable it
        }
}
```

Colors

Darker System Colors Preference

```
static let darkerSystemColorsStatusDidChangeNotification:
NSNotification.Name
```

Type

Consumed.

Purpose

When users enable or set up the Increase Contrast preference in Settings, UIKit posts this notification. Meant to be used along `UIAccessibility.isDarkerSystemColorsEnabled` to determine if this feature is on or off.

Example

```
let darkerSystemColorsChanged =
axPublisher.darkerSystemColorsStatusChanged
    .sink { _ in
        if UIAccessibility.isDarkerSystemColorsEnabled {
            // Increase the contrast of interface elements
        } else {
            // Apply regular constrast to interface elements
        }
}
```

GrayScale Preference

```
static let grayscaleStatusDidChangeNotification: NSNotification.Name
```

Type

Consumed.

Purpose

When users enable or set up the Grayscale preference in Settings, UIKit posts this notification. Meant to be used along `UIAccessibility.isGrayscaleEnabled` to determine if this feature is on or off.

Example

```
let darkerSystemColorsChanged = axPublisher.grayScaleStatusChanged
    .sink { _ in
        if UIAccessibility.isGrayscaleEnabled {
            // Perform any considerations for gray scale
        } else {
            // Apply normal interface elements
        }
```

```
}
```

Inverted Color Preference

```
static let invertColorsStatusDidChangeNotification:
NSNotification.Name
```

Type

Consumed.

Purpose

When users enable or set up the Inverted Color preference options in Settings, UIKit posts this notification. Meant to be used along `UIAccessibility.isInvertColorsEnabled` to determine if this feature is on or off.

Example

```
let invertedColorsChanged = axPublisher.invertedColorStatusChanged
    .sink { _ in
        if UIAccessibility.isInvertColorsEnabled {
            // Perform anything that might be effected by this,
            // For example images typically should ignore this
            // By using UIView.accessibilityIgnoresInvertColors
        } else {
            // Apply normal interface elements
        }
}
```

Assistive Technologies

Pausing Assistive Technologies

```
static var pauseAssistiveTechnology: UIAccessibility.Notification
```

Type

Manually posted.

Purpose

Halts assistive technologies temporarily once posted, such as the Switch Control. Currently, this notification only applies to Switch Control. You need to balance this notification with its sibling, `resumeAssistiveTechnology`. That is, both must be used one after the other.

Example

```
// Do some animation within the interface where Switch Control
// Scanning might not make sense to occur

let pauseID =
UIAccessibility.AssistiveTechnologyIdentifier.notificationSwitchCont
rol

UIAccessibility.post(notification: .pauseAssistiveTechnology,
argument: pauseID)

// Then post the resume notification. More details directly
// Below.
```

Resuming Assistive Technologies

```
static var resumeAssistiveTechnology: UIAccessibility.Notification
```

Type

Manually posted.

Purpose

Resumes assistive technologies if they were halted once posted, such as the Switch Control. Currently, this notification only applies to Switch Control. This is used to balance calls from its sibling, `pauseAssistiveTechnology`.

Example

```
// After some animation within the interface finishes
// And the pause notification was used before this.

let pauseID =
UIAccessibility.AssistiveTechnologyIdentifier.notificationSwitchCont
rol

UIAccessibility.post(notification: .resumeAssistiveTechnology,
argument: pauseID)
```

Assistive Touch Status

```
static let assistiveTouchStatusDidChangeNotification:
NSNotification.Name
```

Type

Consumed.

Purpose

Signifies when AssistiveTouch starts or stops. If Guided Access isn't enabled on the device, this notification won't be posted by the system. The two technologies work together, but Assistive Touch can be on *without* Guided Access enabled. Meant to be used `UIAccessibility.isAssistiveTouchRunning` to determine whether or not AssistiveTouch is enabled.

Example

```
let assistiveTouchChanged = axPublisher.assistiveTouchStatusChanged
    .sink { _ in
        if UIAccessibility.isAssistiveTouchRunning {
            // Hide some chrome that Guided Access
            // May not need showing
        } else {
            // Apply normal interface elements
        }
}
```

Guided Access Status

```
static let guidedAccessStatusDidChangeNotification:
NSNotification.Name
```

Type

Consumed.

Purpose

Signifies when Guided Access mode starts or stops. Meant to be used with `UIAccessibility.isGuidedAccessEnabled` to check whether or not Guided Access mode is on.

Example

```
let guidedAccessStatusChanged =
axPublisher.guidedAccessStatusChanged
    .sink { _ in
        if UIAccessibility.isGuidedAccessEnabled {
            // Hide some chrome that Guided Access mode
            // May not need showing
        } else {
            // Apply normal interface elements
        }
}
```

Audio and Speech

Mono Audio Status

```
static let monoAudioStatusDidChangeNotification: NSNotification.Name
```

Type

Consumed.

Purpose

Signifies when system audio is changed from stereo to mono. Meant to be used along with `UIAccessibility.isMonoAudioEnabled` to check audio status.

Example

```
let monoAudioStatusChanged = axPublisher.monoAudioStatusChanged
    .sink { _ in
        if UIAccessibility.isMonoAudioEnabled {
            // Perform any audio considerations
        } else {
            // Mono audio isn't active
        }
}
```

Speak Screen Status

```
static let speakScreenStatusDidChangeNotification:
NSNotification.Name
```

Type

Consumed.

Purpose

Signifies that Screen Speak has been toggled. Screen Speak, along with Speak Selection, are different from VoiceOver. Screen Speak has a control panel to control speed, play and pause and more. It's more honed towards users with low vision, but who may be able to still navigate their device through visual means. Meant to be used along with `UIAccessibility.isSpeakScreenEnabled` to check its status.

Example

```
let speakScreenStatusChanged = axPublisher.speakScreenStatusChanged
    .sink { _ in
        if UIAccessibility.isSpeakScreenEnabled {
            // Unlike VoiceOver, typically you don't need
```

```
            // To hide or show any interface elements with
            // Speak Screen.
        } else {
            // See above.
        }
}
```

Speak Selection Status

```
static let speakSelectionStatusDidChangeNotification:
NSNotification.Name
```

Type

Consumed.

Purpose

Signifies that Speak Selection has been toggled. Screen Selection, along with Screen Speak, are different from VoiceOver. Speak Selection has a control panel to control speed, play and pause and more and only considers any highlighted text. It's more honed towards users with low vision, but who may be able to still navigate their device through visual means. Meant to be used along with `UIAccessibility.isSpeakSelectionEnabled` to check its status.

Example

```
let speakSelectionStatusChanged =
axPublisher.speakSelectionStatusChanged
    .sink { _ in
        if UIAccessibility.isSpeakSelectionEnabled {
            // Unlike VoiceOver, typically you don't need
            // To hide or show any interface elements with
            // Speak Selection.
        } else {
            // See above.
        }
```

```
}
```

Hearing Device Status

```
static let hearingDevicePairedEarDidChangeNotification:
NSNotification.Name
```

Type

Consumed.

Purpose

When a user connects or disconnects a hearing device, such as a hearing aid, this notification is posted. Meant to be used along with `UIAccessibility.hearingDevicePairedEar` to check which ear a device is in, or if one is connected at all.

Example

```
let hearingDeviceStatusChanged =
axPublisher.hearingDeviceConnectionStatusChanged
    .sink { _ in
        if UIAccessibility.hearingDevicePairedEar ==
UIAccessibility.HearingDeviceEar.both  {
            // User has a hearing device connected to both ears
        } else {
            // Could continue more checks or back out here
        }
}
```

Motion

Reduced Motion Status

```
static let reduceMotionStatusDidChangeNotification:
NSNotification.Name
```

Type

Consumed.

Purpose

When a user toggles the Reduce Motion preference in Settings, this notification gets posted. Meant to be used with `UIAccessibility.isReduceMotionEnabled` to determine its status.

Example

```
let reducedMotionStatusChanged =
axPublisher.reducedMotionStatusStatusChanged
    .sink { _ in
        if UIAccessibility.isReduceMotionEnabled {
            // Pare back animations
        } else {
            // Perform any animations
        }
}
```

Shake to Undo Status

```
static let shakeToUndoDidChangeNotification: NSNotification.Name
```

Type

Consumed.

Purpose

When a user toggles the Shake to Undo preference in Settings, this notification gets posted. Meant to be used with `UIAccessibility.isShakeToUndoEnabled` to determine its status.

Example

```
let shakeToUndoStatusChanged = axPublisher.shakeToUndoStatusChanged
    .sink { _ in
```

```
        if UIAccessibility.isShakeToUndoEnabled {
            // Expose other means for undo and redo
        } else {
            // You can still rely on shake motion for undo+redo
        }
}
```

Three Key Takeaways

1. Accessibility notifications cover every facet of the Apple's accessibility suite.
2. As such, knowing when to listen for them (or post them yourself) is a critical skill for accessibility programming on iOS.
3. You won't need to memorize them all, but a broad understanding of them will help you know when to reach for one.

User Preferences in SwiftUI

As we saw in the last chapter, "Posting and Consuming Accessibility Notifications", UIKit provides several hooks to respond to virtually every accessibility preference found in iOS. By way of notifications, developers can adapt and change their apps to make sure it's accessible to everyone. However, that's primarily a UIKit-centric approach to addressing those types of scenarios.

In SwiftUI, the same opportunities exist to adapt to user's needs, but they are exposed via environment variables – which feel much more at home with SwiftUI and its declarative paradigm. These are vended via the `EnvironmentValues` struct, and using Swift's property wrappers feature, we can query their values by simply specifying the relevant value's key path inside of `@Environment(\. EnvironmentValues.theKeyPath)`.

When we looked at responding to accessibility needs in UIKit, we typically updated our interface in response to an incoming notification. In SwiftUI, we define ahead of time what should happen whether or not these preferences are active. This is a classic example of the primary difference between the two interface frameworks. With UIKit we are responsible for making the moves to our interface manually by responding to external events, in SwiftUI we specify what *should* happen upfront and let the system decide how to best carry out the task:



SwiftUI allows us to describe what should happen, but abstracts away the "how". The opposite is true of UIKit.

SwiftUI has a bevy of environment variables, but for our purposes we'll focus on the accessibility ones:

- `accessibilityEnabled`
- `accessibilityVoiceOverEnabled`

- accessibilitySwitchControlEnabled
- accessibilityReduceTransparency
- accessibilityReduceMotion
- accessibilityDifferentiateWithoutColor
- accessibilityInvertColors
- accessibilityShowButtonShapes
- legibilityWeight

The naming conventions are similar to those found in `UIAccessibility.h` and for good reason – they express nearly all of the same ideas and represent similar purposes. For that reason, most of these are read-only in nature. As we did in the last chapter, we'll go through each of these with an example, starting with `.accessibilityEnabled`.

Accessibility Enabled

This signifies whether or not any accessibility technology is activated. Think of things like the Rotor Control or VoiceOver. Note, however, that it doesn't delineate between *what* is active, just that something *is* active.

To offset that limitation, you can rely on `UIAccessibility.h` as we did in the previous chapter. For example, you can check if VoiceOver is on in tandem with this environment variable:

```
// SwiftUI -> Xcode -> UserPrefsFig1View.swift

@Environment(\.accessibilityEnabled) var axEnabled

VStack(alignment: .leading, spacing: 8.0) {
    Text("AX Technology is \(axEnabled ?
"Active" : "Inactive")")
        .fontWeight(.medium)
    if UIAccessibility.isVoiceOverRunning {
```

```
            Text("Voice Over is active")
        } else {
            Text("Voice Over is not active")
        }
}
```

In of itself, this environment value is not very useful. If you couple it alongside other checks to determine which accessibility technology is active, you can go further with it. However, the need to know *which* technology is being used is typically far more useful than simply knowing that one is activated at all.

Accessibility VoiceOver Enabled

Reports whether or not VoiceOver is activated. Use this to reconsider interface elements that may serve as clutter or hinder navigation for VoiceOver users.

For example, consider a "Favorite" button in a list of video games. While you could report the element to VoiceOver, maybe you have several other actions you've implemented as custom accessibility actions. You could follow the same pattern with favoriting if VoiceOver is active:

```
// SwiftUI -> Xcode -> UserPrefsFig2View.swift

struct UserPrefsViewFig2: View {
    @Environment(\.accessibilityVoiceOverEnabled) var axVoiceOverOn
    let videogames = VideoGame.data

    var body: some View {
        List(videogames) { game in
            HStack {
                Text(game.name)
                Spacer()
                if !axVoiceOverOn {
                    Image(systemName: game.isFavorite ?
"star.fill" : "star")
                        .foregroundColor(.indigo)
                }
```

```
            }
            .accessibilityAction {
                // Toggle favorites
            }
        }
    }
}
```

Accessibility Switch Control Enabled

Reports whether or not the Switch Control is activated. When the user is relying on the Switch Control for navigating their device, it's a good idea to strip away any interface or user experience flows that could get in the way. One common thing that can "get in the way"? Animations. While done right and found all throughout iOS, for Switch Control users they can sometimes be an annoyance keeping them from getting to the elements they want.

Consider this "Getting Started" splash view, which animates its elements in. If the Switch Control is active, you can leverage an inert animation modifier to forgo animating things at all. Specifically, look at the `animation` modifier below:

```
// SwiftUI -> Xcode -> UserPrefsFig2View.swift

struct UserPrefsViewFig3: View {
    @Environment(\.accessibilitySwitchControlEnabled) var
axSwitchControlOn
    @State private var scale = 0.5

    var body: some View {
        VStack(alignment: .leading, spacing: 8.0) {
            Group {
                RoundedRectangle(cornerRadius: 6.0)
                    .fill(LinearGradient(colors:
[.teal, .blue, .blue], startPoint: .bottomLeading,
endPoint: .topTrailing))
```

```
                    .frame(width: 80, height: 80)
                    .overlay {
                        Image(systemName: "highlighter")
                            .font(.system(size: 28, weight: .bold))
                            .foregroundColor(.white)
                    }
                Text("Welcome to Highlighter Pro!")
                    .font(.system(size: 44, weight: .bold,
design: .rounded))
            }
            .scaleEffect(scale)
            .animation(axSwitchControlOn ? nil : .easeOut, value:
scale)
        }
        .onAppear {
            scale = 1.0
        }
    }
}
```

Accessibility Reduce Transparency Enabled

Reports whether or not the Reduce Transparency preference is enabled within Settings. If this is on, consider making blurred views, commonly created using background materials, more opaque or using a solid color in place of them.

In this example, if the user prefers reduced transparency, the background material flips from `.ultraThinMaterial`, which is very transparent and blurs quite heavily, to `.ultraThickMaterial`, which acts more like a solid color.

```
struct UserPrefsViewFig4: View {
    @Environment(\.accessibilityReduceTransparency) var
axReduceTransparencyOn

    var body: some View {
        ZStack {
            Image("Wallpaper")
```

```
                .resizable()
                .aspectRatio(contentMode: .fill)
            VStack {
                Spacer()
                HStack {
                    // Ommitted for readability. Refer to source
code.
                    }
                    .foregroundColor(.primary)
                    .blendMode(axReduceTransparencyOn ? .plusDarker
: .overlay)
                    Spacer()
                }
                .padding()
                .background(axReduceTransparencyOn ? .ultraThickMate
rial : .ultraThinMaterial)
                .frame(height: 84)
                .clipShape(RoundedRectangle(cornerRadius: 44, style:
.continuous))
                .padding(EdgeInsets(top: 0, leading: 10, bottom: 10,
trailing: 10))
            }
        }
        .edgesIgnoringSafeArea(.all)
    }
}
```



Accessibility Reduce Motion Enabled

Reports whether or not the Reduce Motion preference is enabled within Settings. If it is, dial back animations or "micro animations" that could occur in your app. For example, navigation controllers change their classic push and pop animations to simple fade in and fade out interactions.

Here, this mock splash view skips its introduction animation if the option is on:

```
// SwiftUI -> Xcode -> UserPrefsViewFig5.swift

struct UserPrefsViewFig5: View {
    @Environment(\.accessibilityReduceMotion) var axReduceMotionOn
    @State private var scale = 0.5

    var body: some View {
        VStack(alignment: .leading, spacing: 8.0) {
            Group {
                RoundedRectangle(cornerRadius: 6.0)
```

```
                    .fill(LinearGradient(colors:
[.teal, .blue, .blue], startPoint: .bottomLeading,
endPoint: .topTrailing))
                    .frame(width: 80, height: 80)
                    .overlay {
                        Image(systemName: "highlighter")
                            .font(.system(size: 28, weight: .bold))
                            .foregroundColor(.white)
                    }
                Text("Welcome to Highlighter Pro!")
                    .font(.system(size: 44, weight: .bold,
design: .rounded))
            }
            .scaleEffect(scale)
            .animation(axReduceMotionOn ? nil : .easeOut, value:
scale)
        }
        .onAppear {
            scale = 1.0
        }
    }
}
```

Accessibility Differentiate without Color Enabled

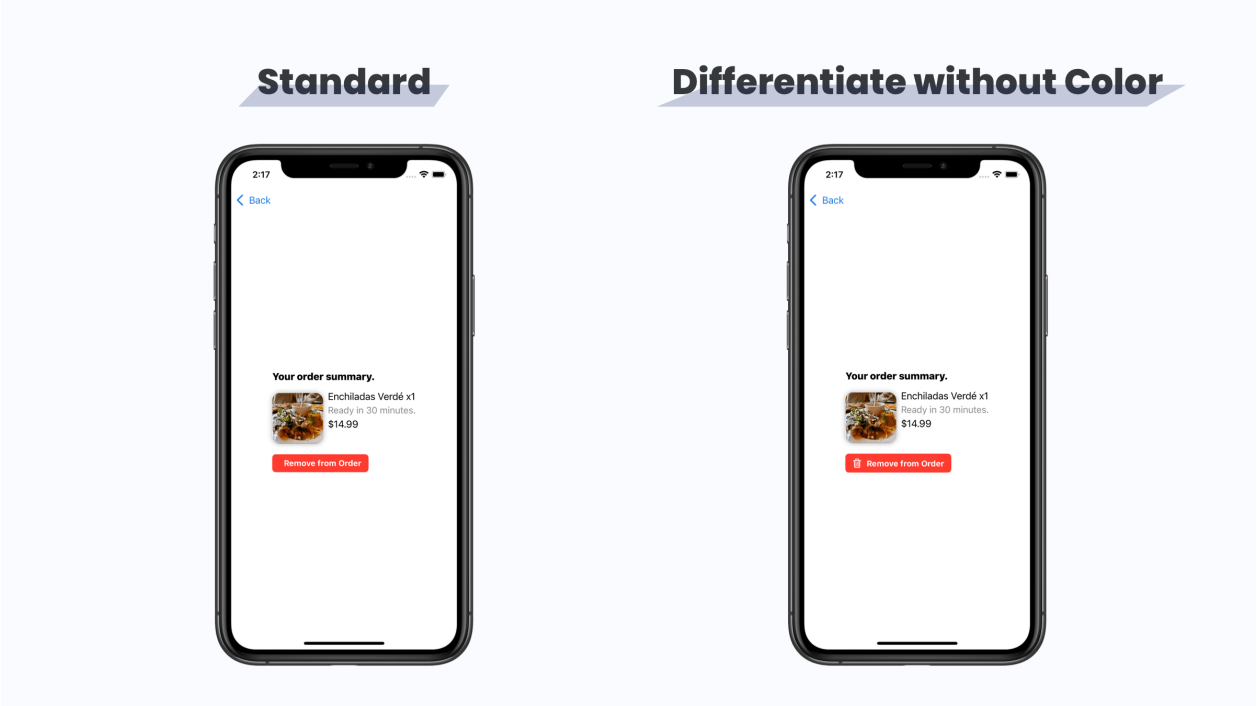
Reports whether to not the Differentiate without Color preference is enabled within Settings. This setting is useful for users who might be using apps that could rely on color to convey meaning. Relying on color for interaction purposes usually doesn't internationalize very well. What red means in western countries (stop, error, typically a negative connotation) doesn't hold true in other territories such as China, where red carries a more positive meaning.

When this is on, make sure you've included affordances to convey what things do outside of just a color. In this mock food ordering view, if the option is on we add a

trashcan to the red delete button to better convey the meaning of the control outside of simply relying on it being red to express its purpose:

```
// SwiftUI -> Xcode -> UserPrefsViewFig6.swift

struct UserPrefsViewFig6: View {
    @Environment(\.accessibilityDifferentiateWithoutColor) var
axDifferentiateWithoutColorOn

    var body: some View {
        VStack(alignment: .leading, spacing: 18.0) {
            Text("Your order summary.")
                .font(.headline)
                .fontWeight(.heavy)
            HStack {
                Image("Mexican Food")
                    .resizable()
                    .frame(width: 88, height: 88)
                    .aspectRatio(nil, contentMode: .fit)
                    .cornerRadius(14.0)
                    .shadow(radius: 4.0)
                VStack(alignment: .leading, spacing: 4.0) {
                    Text("Enchiladas Verdé x1")
                    Text("Ready in 30 minutes.")
                        .font(.callout)
                        .foregroundColor(.secondary)
                    Text("$14.99")
                }
                .offset(x: 0, y: -14)
            }
            Button(action: {}) {
                Label("Remove from Order",
systemImage:axDifferentiateWithoutColorOn ? "trash" : "")
                    .font(.system(size: 14, weight: .bold))
            }
            .tint(.red)
            .buttonStyle(.borderedProminent)
        }
```

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



Notice how we don't rely solely on the red color to denote a destructive action when we differentiate with more than just color.

Accessibility Invert Colors Enabled

Reports whether to not the Invert Colors preference is enabled within Settings. For users with visual disabilities, this option helps them view and use their device with a greater comfort level. However, it does exactly as it says - all of the colors in your interface become inverted. Note that this reflects the "Smart Invert" color option which works the same as classic invert, save for affecting media and videos.

What does that mean for you? Look for places where you've used a custom color or drawing that may need to be updated. Here, our text view that shows a gif card promotion is using an `.indigo` color which doesn't adapt very well with inverted col-

ors. In this case, we've opted to use a more dynamic-capable color, `.label`, when this setting is on:

```
// SwiftUI -> Xcode -> UserPrefsViewFig7.swift

struct UserPrefsViewFig7: View {
    @Environment(\.accessibilityInvertColors) var axInvertColorsOn
    private let labelColor: Color = Color(UIColor.label)

    var body: some View {
        VStack {
            HStack {
                Image("Mexican Food")
                    .resizable()
                    .frame(width: 44, height: 44)
                    .aspectRatio(nil, contentMode: .fit)
                Text("Enchiladas")
                    .foregroundColor(labelColor)
                    .fontWeight(.medium)
                Spacer()
                Button("Order") {

                }
                .buttonStyle(.bordered)
            }
            HStack {
                Image("Healthy Food")
                    .resizable()
                    .frame(width: 44, height: 44)
                    .aspectRatio(nil, contentMode: .fit)
                Text("\(Image(systemName: "giftcard.fill")) Fruits
and Veggies")
                    .foregroundColor(axInvertColorsOn ? labelColor :
.indigo)
                    .fontWeight(.medium)
                Spacer()
                Button("Redeem") {
```

```
                }
                .buttonStyle(.bordered)
            }
        }
        .padding()
    }
}
```

Accessibility Show Button Shapes Enabled

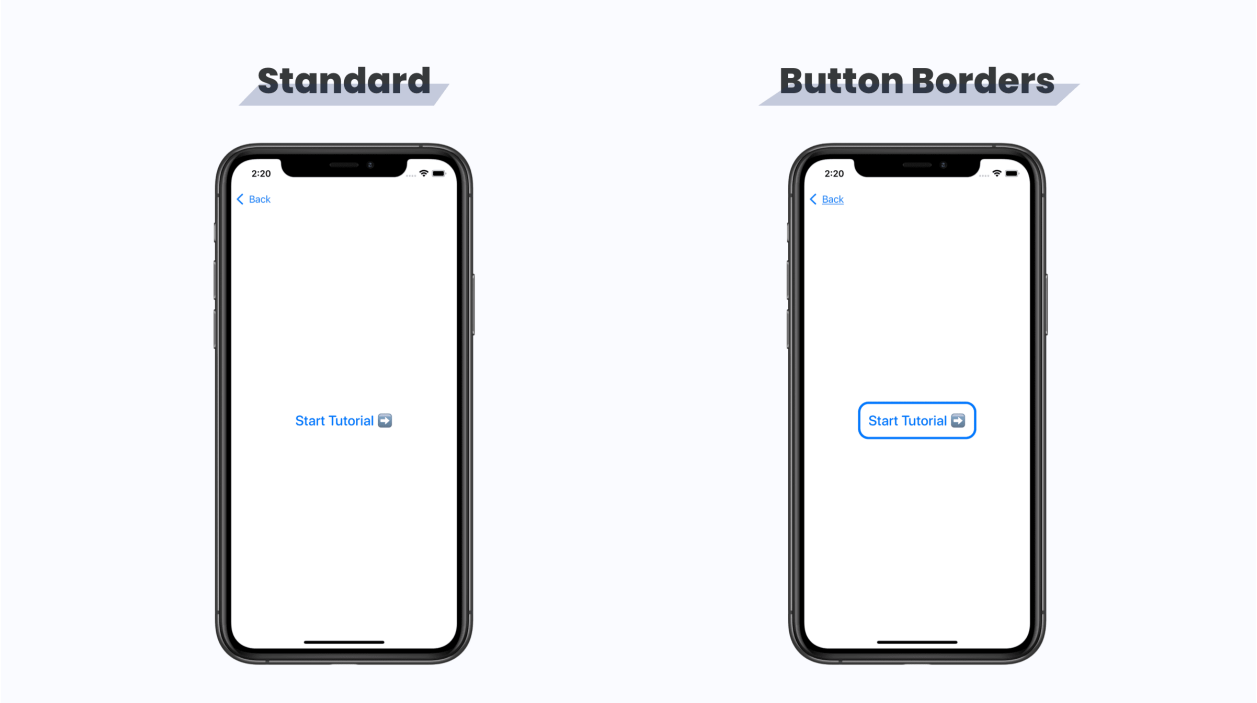
Reports whether to not the Show Button Shapes preference is enabled within Settings. If this is true, all system vended buttons will show borders and shapes around them to better convey their interactivity. For example, buttons that are typically text based, like the "Back" button in a navigation controller, will be underlined.

When this is on, be sure any of your custom views you've made respect the setting. Below, I've created a custom button that is primarily text based. When Show Button Shapes is enabled, I've added a more classic-style border around the text to better denote its button-based role[17]:

```
// SwiftUI -> Xcode -> UserPrefsViewFig8.swift

struct UserPrefsViewFig8: View {
    var body: some View {
        CustomButton()
    }
}

struct CustomButton: View {
    @Environment(\.accessibilityShowButtonShapes) var
axShowButtonShapesOn
```

[17] In a production app, you'd also want to add the necessary accessibility traits to controls like this as well.

```
    var body: some View {
        Text("Start Tutorial ➡")
            .foregroundColor(.blue)
            .font(.system(size: 24, weight: .medium,
design: .rounded))
            .padding()
            .overlay {
                RoundedRectangle(cornerRadius: 16)
                    .stroke(Color.blue, lineWidth:
axShowButtonShapesOn ? 4 :0)
            }
    }
}
```

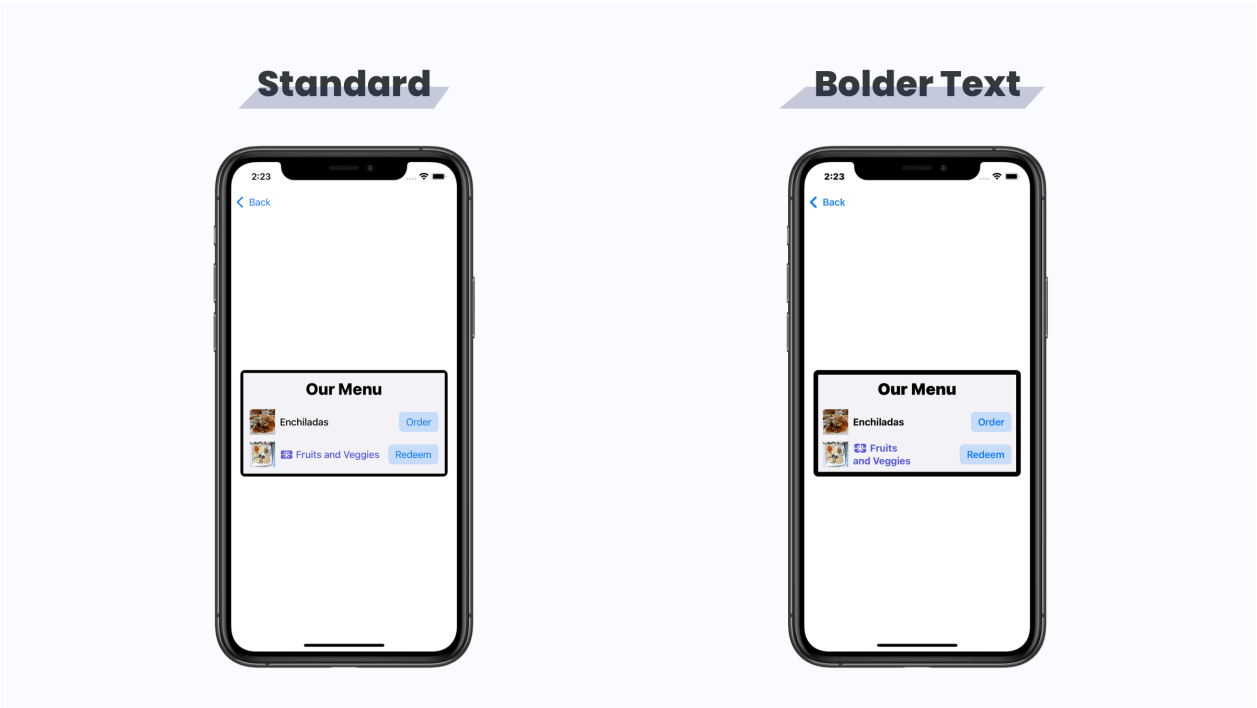


Legibility Weight

Reports whether to not the Bold Text preference is enabled within Settings. While you should certainly make font weights a bit heavier when this is on, in reality most of your SwiftUI views will likely do this by default if you're relying on Text Styles.

Aside from that, you should also consider interface elements such as borders. Make things like this a bit thicker when this setting is on:

```
// SwiftUI -> Xcode -> UserPrefsViewFig9.swift

struct UserPrefsViewFig9: View {
    @Environment(\.legibilityWeight) var axLegibilityWeight

    /// Description
    var body: some View {
        VStack {
            // Ommitted for brevity. Reflects the mock
            // Food ordering view.
        }
        .padding()
        .background {
            Color(UIColor.tertiarySystemGroupedBackground)
                .border(Color.primary, width: axLegibilityWeight
== .bold ? 8 : 5)
        }
        .cornerRadius(8.0)
        .padding()
    }
}
```



Tips

Creating Custom Environment Keys

You might've noticed that throughout these examples, there were similar checks we've done. For example, the mock splash view does an animation involving scale, and it also fades out a blur view. We can map both of these interactions to accessibility preferences:

1. Scale animation -> Reduced Motion
2. Blur view -> Reduced Transparency

While it's certainly fine to combine a boolean value that checks for both of these things…

```
@Environment(\.accessibilityReduceMotion) var axReduceMotionOn
@Environment(\.accessibilityReduceTransparency) var
axReduceTransparencyOn

let shouldAnimate: Bool = !axReduceMotionOn && !
axReduceTransparencyOn
```

...if you find yourself coming back to this or other similar compounded checks, consider creating a custom environment key. We've been leaning on environment variables in all of these examples, and it can be helpful to extend them to create your own accessibility flows.

Continuing with the splash view example, we could create an environment variable that reports true when *both* allow transparency and motion are off. The process involves adopting the `EnvironmentKey` protocol and then extending the `EnvironmentValues` struct we mentioned at the top of the chapter:

```
// SwiftUI -> Xcode -> UserPrefsViewFig10.swift

// MARK: Custom Environment Key
private struct AXBlurredAnimationsEnabled: EnvironmentKey {
    static let defaultValue: Bool = (!
UIAccessibility.isReduceMotionEnabled && !
UIAccessibility.isReduceTransparencyEnabled)
    typealias Value = Bool
}

extension EnvironmentValues {
    var axAllowsBlurredAnimations: Bool {
        get { self[AXBlurredAnimationsEnabled.self] }
    }
}
```

The `EnvironmentKey` protocol requires us to supply the type that the extended key will represent, along with a default value. Then, once we extend `EnvironmentVal-`

ues to house the new key, we can make the property either read only (as we've done) or you can optionally supply a setter to allow the environment value to be written to.

Putting that together, we now have a tidy check in place which follows the same pattern that other accessibility values follows to be used in SwiftUI (i.e., an environment variable):

```
// SwiftUI -> Xcode -> UserPrefsViewFig10.swift

struct UserPrefsViewFig10: View {
    @Environment(\.axAllowsBlurredAnimations) var
allowsBlurredAnimations: Bool
    @State private var scale = 0.5

    var body: some View {
        VStack(alignment: .leading, spacing: 8.0) {
            Group {
                RoundedRectangle(cornerRadius: 6.0)
                    .fill(LinearGradient(colors:
[.teal, .blue, .blue], startPoint: .bottomLeading,
endPoint: .topTrailing))
                    .frame(width: 80, height: 80)
                    .overlay {
                        Image(systemName: "highlighter")
                            .font(.system(size: 28, weight: .bold))
                            .foregroundColor(.white)
                    }
                Text("Welcome to Highlighter Pro!")
                    .font(.system(size: 44, weight: .bold,
design: .rounded))
            }
            .scaleEffect(scale)
            .animation(allowsBlurredAnimations ? nil : .easeOut,
value: scale)
        }
        .onAppear {
            scale = 1.0
```

```
            }
        }
}
```

I find myself creating a few of these in SwiftUI, as accessibility needs vary and sometimes require combining some permutation of them together.

Three Key Takeaways

1. Adapting your interface to a user's accessibility needs is imperative.
2. Whether you are using UIKit or SwiftUI, iOS offers you methods to handle each of those potential needs.
3. Fortunately, in SwiftUI, our job becomes a bit easier due to its declarative nature which excels at defining interface elements quickly and easily.

Convenience Functions

If you're reading this book in chronological order, you've read about several scenarios that accessibility APIs cover. Hearing, motor disabilities, visual aids – iOS has a large surface area to address to ensure that the operating system remains usable for virtually every human being on the planet. And, that's a wonderful thing.

However.

The challenge, as with anything that has a "large surface area", is that inevitably there are things which are lying around in the tool chest that don't fit into a nice, tidy box which translates easily into a certain category. They are nice to know about, and certainly useful – but possibly obtuse in definition.

This chapter is all about those things. The hidden gems. The "nice to know" about APIs. They can save time and headaches.

Most of these functions typically enhance an existing API we've already covered. You'll notice how they might fit in nicely with VoiceOver, Switch Controls or a combina-

tion of all of them. Like the previous two chapters, we'll start from the top and work our way down.

AXNameFromColor

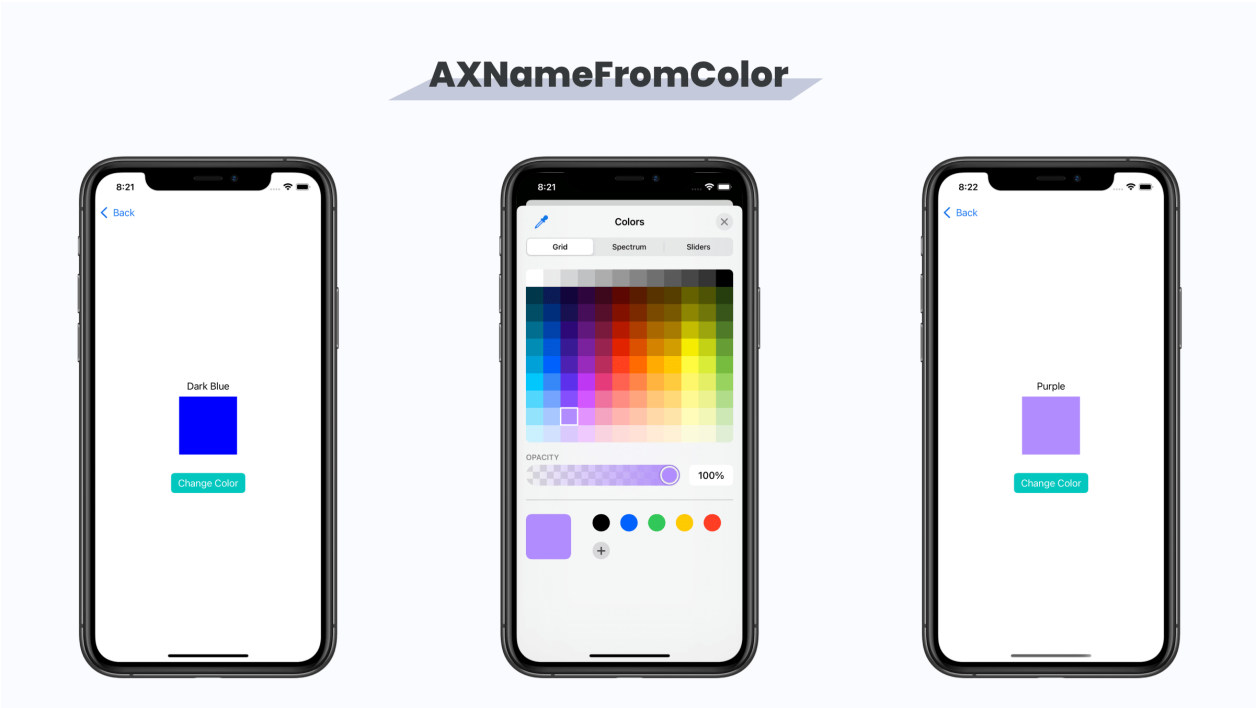
This returns a human-readable string of the color passed in. This is useful for accessibility attributes, such as reading out interface elements to VoiceOver, or for displaying the value within text controls. Outside of accessibility, this is perfect for any media based creations apps, such as those that may offer drawing, sketching or color palettes to easily label colors.

```
public func AXNameFromColor(_ color: CGColor) -> String
```

While it requires a `CGColorRef`, both SwiftUI and UIKit offer this type up from their respective color types – `Color` and `UIColor`:

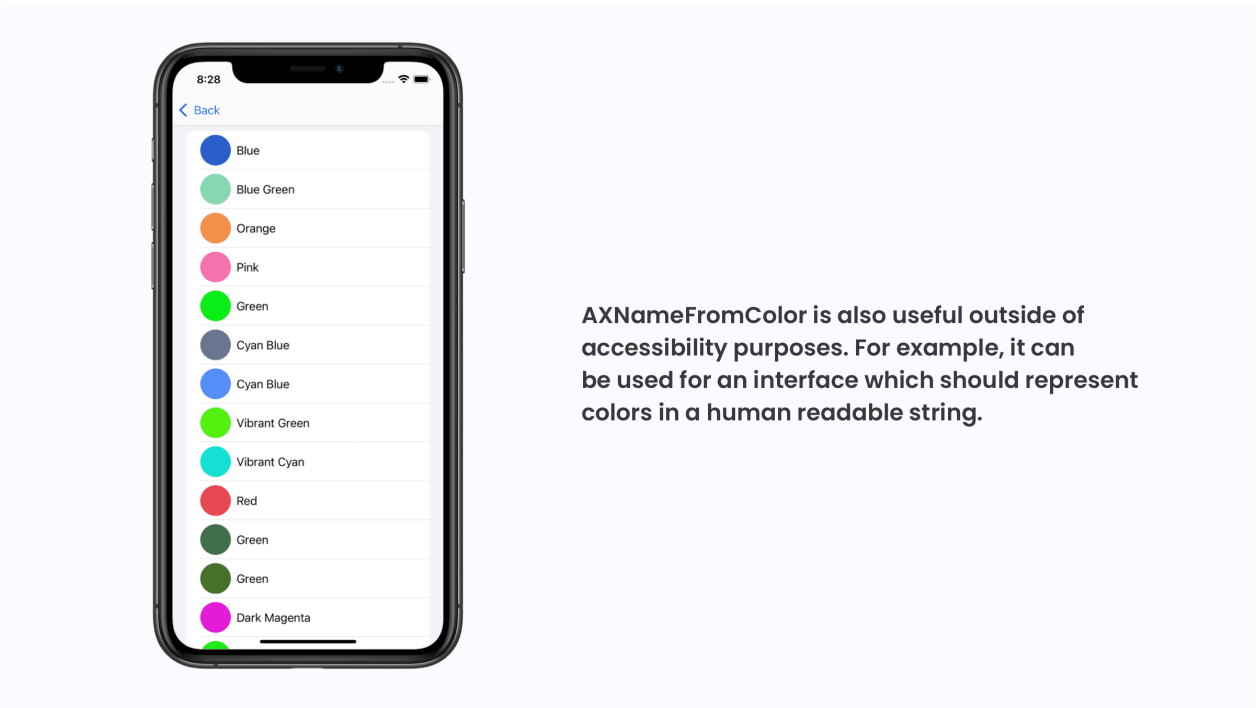
```
// UIKit
let blue: UIColor = .blue

// SwiftUI
let blue: Color = .blue

// CGColorRef access looks identical for both types
let blueColorRef = blue.cgColor

// Finally, pass it off to the function
let colorName: String = AXNameFromColor(blueColorRef)
```

To get a feel for how powerful this function is, run either `ConvenienceFunctions-Fig1ViewController.swift` for a UIKit example or `ConvenienceFunctions-Fig1View.swift` for a SwiftUI version. Both approaches allow you to pick any color using iOS' color picker control and then, using this function, it updates a label to reflect the new color's name:



Notice how the color label is updated each time a new color is selected.

This function is ideal for any scenario where a VoiceOver user needs to know the name of a color, but beyond that perhaps your interface simply requires listing colors out:



This function has utility outside of just accessibility programming.

One thing to be aware of using this method is that sometimes, the `.cgColor` property can return nil. This is often the case with using dynamic colors, which can change at runtime. Be sure to account for this case and provide a sensible value if you are getting `nil` returned.

AXHearingDeviceUtilities

We touched on the concept of discovering paired hearing aids previously in the "Posting and Consuming Accessibility Notifications" chapter, but there is also a nascent set of APIs to help you be aware of whether or not these devices are active, which ones are in use and more found within `AXMFiHearingDevice`.

While this is likely a *very* niche use case, when you need it – you need it. Certain apps made specifically for hearing devices will provide domain-specific functionalities such as reporting the device battery or generalized "health" checks or possibly a re-

mote fitting feature much like how Apple's AirPods Pro work. All of those, though, rely on streaming audio to the hearing devices.

It's possible for users to disable audio streaming. When that occurs, those features wouldn't work. To account for this, these APIs allow you to audit the state of streaming preferences so that you may react accordingly. You can also get unique identifiers for the streaming devices or determine if device-specific features are available, such as bidirectional streaming.

For example, you can check which ears are streaming audio:

```
let streamingEar: String = {
    switch AXMFiHearingDevice.streamingEar() {
    case .both:
        return "Both Ears"
    case .left:
        return "Exclusively Left"
    case .right:
        return "Exclusively Right"
    default: return ""
    }
}()
```

Along those lines, when the value of `streamingEar()` changes, the accessibility engine will post a notification:

```
let nc = NotificationCenter.default
let steamEarChangedNote =
AXMFiHearingDevice.streamingEarDidChangeNotification
let earStreamSub = nc.publisher(for: steamEarChangedNote).sink { _
in
    // Update UI or invoke AXMFiHearingDevice.streamingEar() once
more
}
```

Perhaps the most critical functionality is the ability to find identifiers for the hearing aids and match them against your app's entitlements. In this case, these apps must have the `hearing.aid.app` entitlement, otherwise these functions won't work.

By using `AXMFiHearingDevice.pairedDeviceIdentifiers()`, you can query for devices which match your app's entitlements:

```
// Get all unique device identifiers
let deviceIDs = AXMFiHearingDevice.pairedDeviceIdentifiers()

// Create a manager to connect them
let manager = CBCentralManager(delegate: nil, queue: .main)

// Find matching devices against your's entitlements
// Which are connected
let connectedDevices = manager.retrievePeripherals(withIdentifiers:
deviceIDs).filter{
$0.state == .connected }

// Connect to your own manager
connectedDevices.forEach {
    manager.connect($0)
}
```

And, just like with the notification pushing out any updates regarding the activated streaming ears, a notification exists to notify you if any device identifiers changed (i.e. new devices were connected):

```
let nc = NotificationCenter.default
let idChangedNote =
AXMFiHearingDevice.pairedUUIDsDidChangeNotification
let idChangedPub = nc.publisher(for: idChangedNote)
```

Finally, `AXMFiHearingDevice` also reports if the iOS device has support for bidirectional streaming. It's simply a static function:

```
let hasBidirectionalStreaming =
AXMFiHearingDevice.supportsBidirectionalStreaming()
```

registerGestureConflictWithZoom

An import accessibility technology, aside from VoiceOver, that helps users who require visual aid is the Zoom feature. When activated, a window appears over the operating system to magnify the contents below it. Below is an example, but please note that screenshots using Zoom don't show the actual contents zoomed in the view as it will when running on device:



The Zoom feature allows users to zero in on specific parts of the screen to view its contents easier

This has a reserved `UITapGesture` that the accessibility engine manages and installs, which is a two tap, three finger gesture. However, this can conflict with existing gestures in your own app. If that's the case, register the conflict with the system. That way,

when the gesture occurs — the user can choose which action they want to happen. The system zoom, or your app's action. It only takes one function call to register conflicts:

```
UIAccessibility.registerGestureConflictWithZoom()
```

Calling this will invoke an alert controller which lets the user decide which gesture they'd prefer to activate:



The system will ask users which gesture they prefer to use when there are conflicting ones marked by the developers.

Consider an interface wherein a triple tap, two finger gesture shows or hides editing chrome for an image:



Some gestures are used by "Power Users" to speed up interactions.

When the view loads, simply call this function and the accessibility engine will do the rest. You also don't need to balance it with another call (i.e. `unregisterConflict` or something similar), the accessibility engine handles it all for you:

```
// Xcode -> UIKit -> ConvenienceFunctionsFig3ViewController.swift

override func viewDidLoad() {
    super.viewDidLoad()

    let tapForChrome = UITapGestureRecognizer(target: self, action:
#selector(toggleChrome))
    tapForChrome.numberOfTapsRequired = 2
    tapForChrome.numberOfTouchesRequired = 3
    imageView.addGestureRecognizer(tapForChrome)
    imageView.isUserInteractionEnabled = true

    UIAccessibility.registerGestureConflictWithZoom()
}
```

While this helps mitigate the problem, I'd suggest avoiding it in the first place. Unless that specific gesture has become well-worn muscle memory for your users, it's best to shift to something else to sidestep any conflicts with the zoom gesture.

In terms of SwiftUI, initiating a gesture such as the Zoom one isn't possible with its `.onTap` or other gesture based modifiers – so as of now there's not really a use for this API in such apps.

zoomFocusChanged(zoomType:, toFrame: in:)

While users are utilizing the Zoom feature mentioned above, there may be times when something important happens within the interface and it'd make sense to direct attention to it. If that occurs, this method will focus the Zoom window over to the `CGRect` that you provide – scoped to the screen's coordinates:

```
// Xcode -> UIKit -> ConvenienceFunctionsFig4ViewController.swift

// Zoom to error view to let Zoom users know
// There has been an issue they should tend to
UIAccessibility.zoomFocusChanged(zoomType: .insertionPoint, toFrame:
self.errorView.frame, in: self.view)
```

By default, Zoom does a great job of following focus. So, if you had a form which swapped out which controls had first responder status (which, if you're unfamiliar with the responder chain, means that control could utilize the keyboard) as the user edited data – it should follow right along.

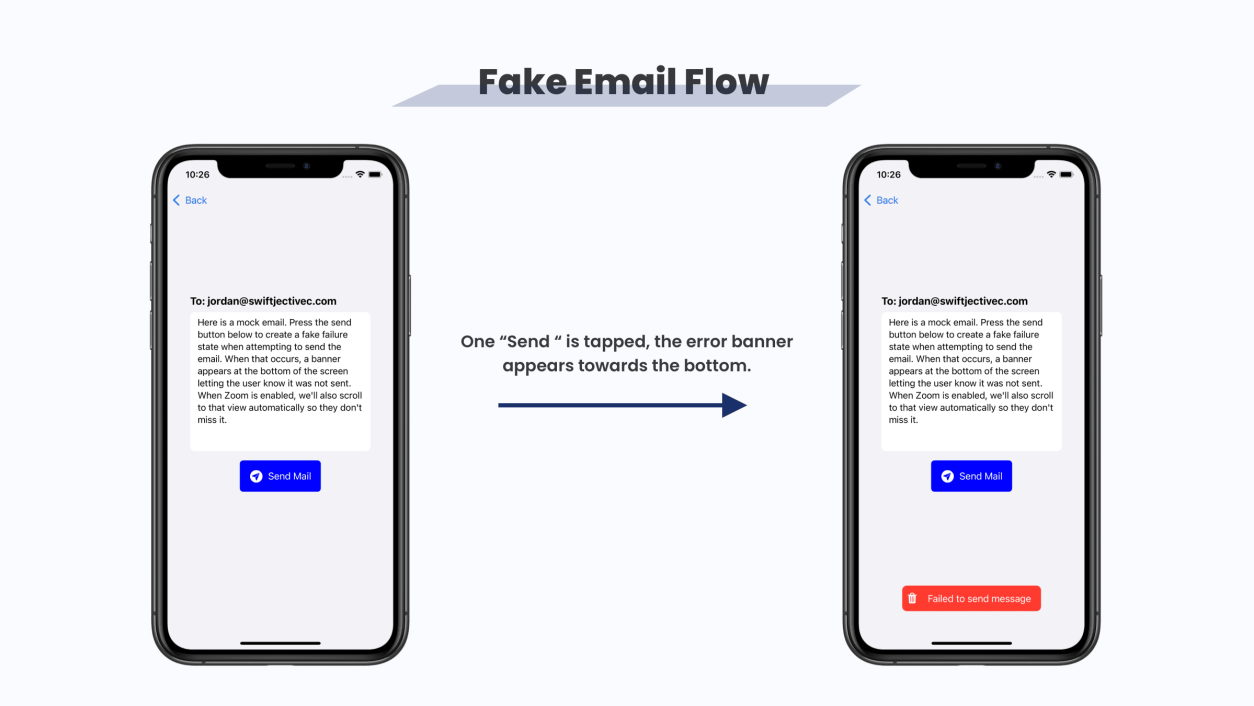
Sometimes, though, you've got custom controls that should gain focus. If that's the case, we'd use this function. At first, that may seem a bit counterintuitive given that the first argument, an enumeration, only has one possible option:

```
@available(iOS 5.0, *)
public enum ZoomType : Int {
    // The system zoom type is the text insertion point.
```

```
    case insertionPoint = 0
}
```

Given that, it seems this is only meant for text controls. But, I've seen it used (and requested to be used from users who rely on Zoom) to also call out things that don't require user input. Taking a step back, then, this is useful for times when a customized text control isn't getting focus when it should *or* you need an interface element to become the focus of the user's attention.

Consider an interface where the user is sending some mail something which requires a network call. If the server was down, or it failed along the way, maybe you would display some sort of banner indicating that it was unsuccessful:



In situations like this, the Zoom window may not let banners like this be "obvious" interactions.

In such cases, if the user is focused on a different area of the screen while Zoom is active, they could miss this entirely. Given situations like this - it makes sense to direct at-

tention to the part of the interface that is most important. The flow would look something like this:



By changing the focus, we can let users be aware of important interface events.

Following the image above, your approach would be as follows:

```
// Xcode -> UIKit -> ConvenienceFunctionsFig4ViewController.swift

// Abbreviated sample. View source code for more.

let sendButton = UIButton(configuration: config, primaryAction:
UIAction() { [weak self] _ in
    guard let self = self else { return }
    let showError = self.errorView.alpha == 0.0

    UIView.animate(withDuration: 1.0) {
        self.errorView.alpha = showError ? 1.0 : 0.0
    } completion: { _ in

    }
```

```
    if showError {
        // Zoom to error view to let Zoom users know
        // There has been an issue they should tend to
     UIAccessibility.zoomFocusChanged(zoomType:.insertionPoint,
                                      toFrame: self.errorView.frame,
                                      in: self.view)
    }
})
```

However, use this sparingly and really consider whether or not changing the Zoom window's frame is worth it for the user. You're making a decision for them here, and when you using APIs that do this (doubly so in accessibility settings) – you better be sure the pay off is merited.

This is a UIKit specific API. However, unlike most assistive technologies, there currently isn't an interface to check whether or not Zoom is activated. If you need to provide affordances for such a situation in a SwiftUI app, currently your best bet would be to fall back to a UIKit representable view.

Three Key Takeaways

1. For every one-off accessibility need on iOS, there is typically a one-off tool available to solve it.
2. Try to *familiarize* yourself with these niche tools so you know they are there when you need them instead of trying to *memorize* the finer details of each one.
3. Finally, remember that these are the tools you may not always need, but it pays to have passing knowledge of them.

Contextual Scenarios

Interfaces on iOS come in many flavors. Those that deal with navigation are sure to feature `MKMapView` heavily. Or, consider a interaction heavy app that uses several gestures. Nearly all of those types of apps (or really *any* app) benefit from drag and drop interactions.

All of these are contextual interactions, or specialized interactions. As such, they requires contextualized accessibility considerations that go along with them. They require a bit of specialized knowledge on how to best expose their unique forms on interactions to users with accessibility needs.

As such, we'll cover each individually:

1. Using `UIAccessibilityPath`.

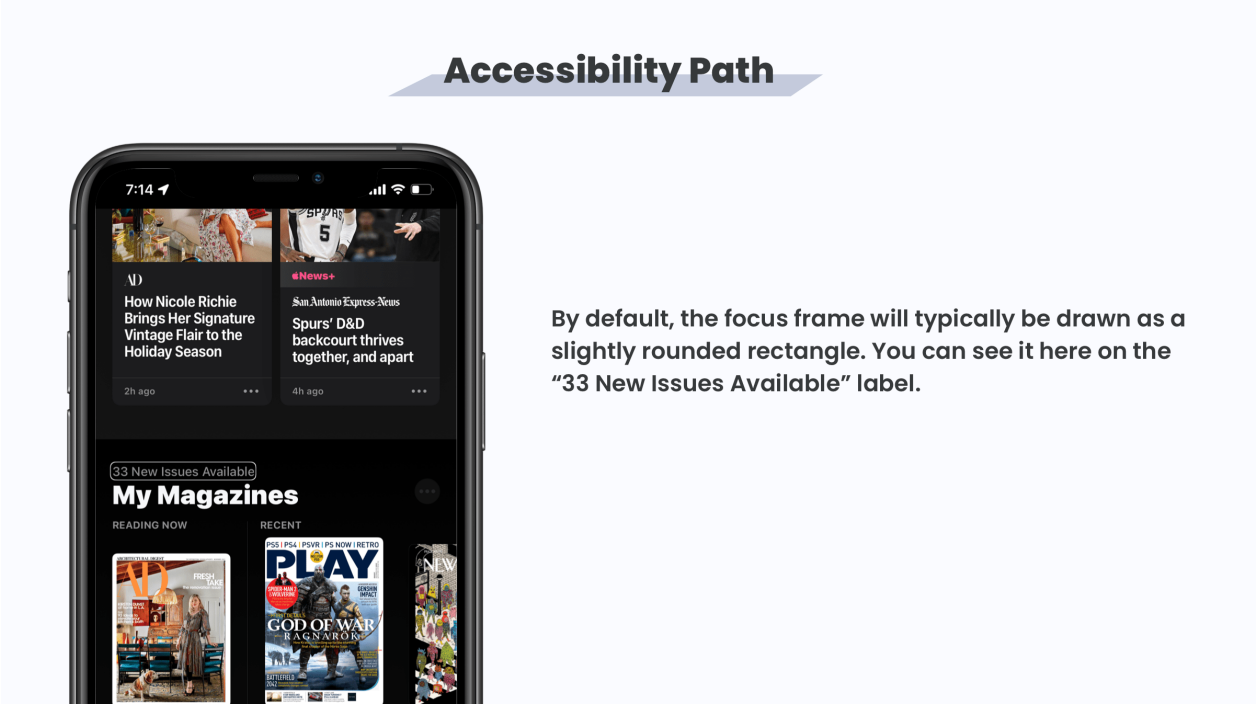
2. Drag and Drop.

Before we begin, these APIs are not exposed to SwiftUI as of iOS 15. This is all about UIKit.

How it Works

Accessibility Path

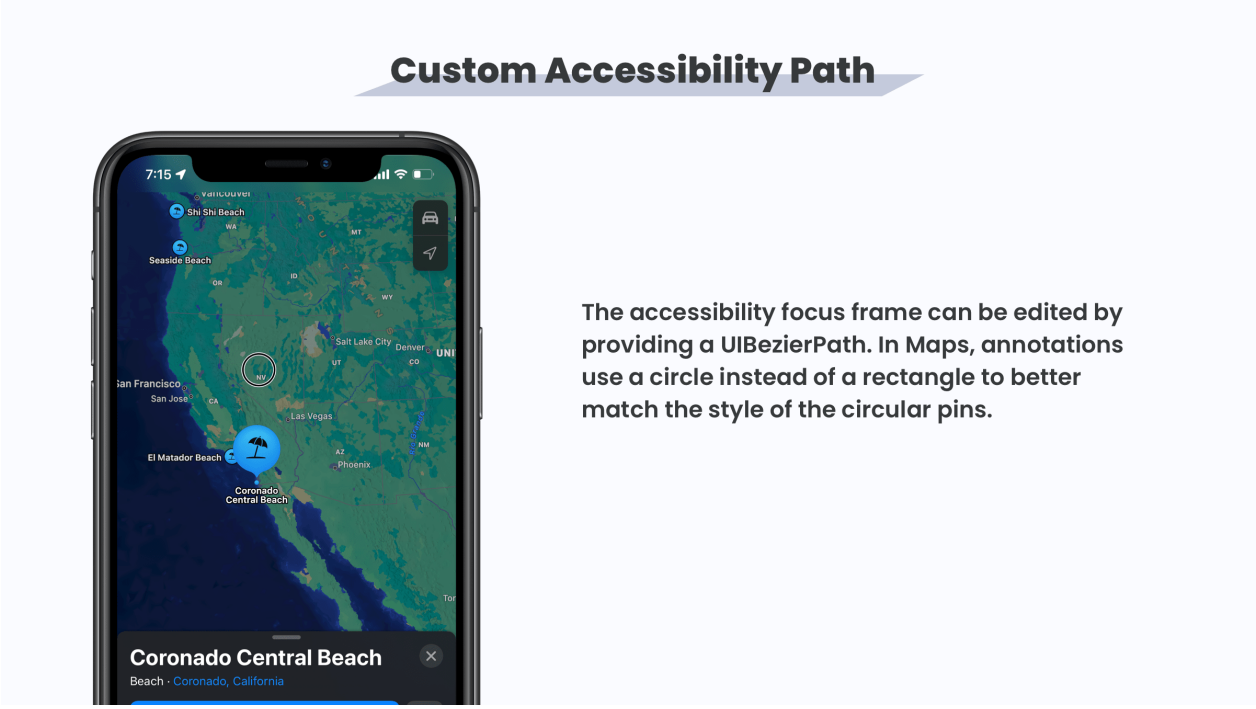
Long since a member of iOS' accessibility suite, `accessibilityPath` lets you customize the accessibility focus frame of elements:



While this doesn't directly make things like VoiceOver experiences *strides* better than if you had not bothered, it does make it *incrementally* better. And, if you're reading

this, making things incrementally better is probably vital to you. It can be easy to forget this, but a VoiceOver user doesn't imply blindness by default – so visual aids can still be of great benefit to them.

Take, for example, Maps. Instead of using the default black rectangle accessibility focus frame – it creates a circle instead:

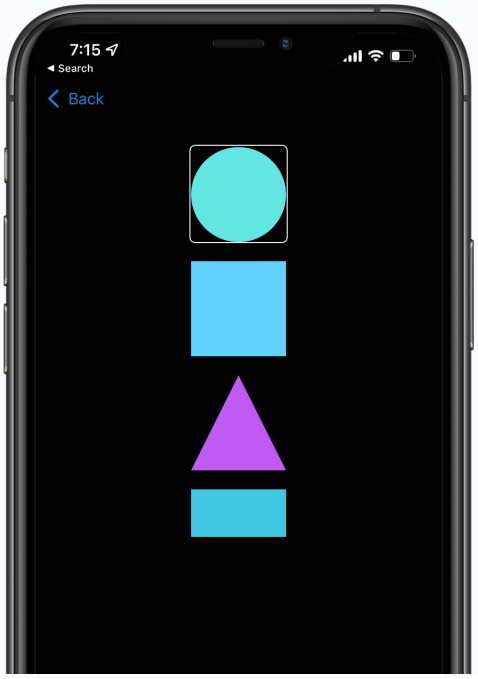


While it may seem superfluous, I think it creates a better flow in the interface and leads to a superior user experience. It simply *feels* better. We can replicate the same types of experiences with nothing more than a bezier path:

```
@NSCopying open var accessibilityPath: UIBezierPath?
```

Consider a game targeted at children learning shapes. It might look something like this:

In a scenario like this, it would likely feel more fluid for VoiceOver users to have the focus frame match the geometry of the shapes. If no changes were made here, each shape would be focused by the rounded rectangle.



Instead of using the same black rectangle, we could match the shape's path to make the frame feel more at home. For example, for the circle shape it'd look something like this:

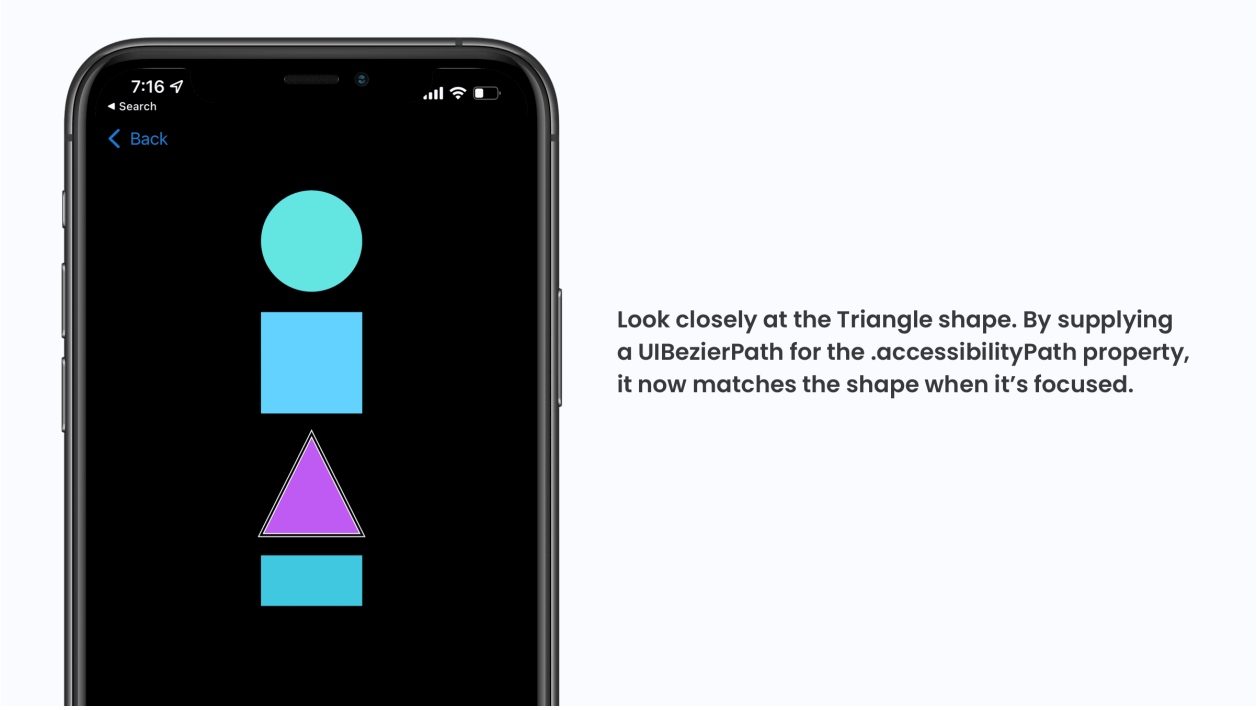
```
// UIKit -> Xcode -> ContextualScenarioFig1ViewController.swift
open override var accessibilityPath: UIBezierPath? {
    get {
        var boundsToScreen = self.convert(self.bounds, to: nil)
        return UIBezierPath(ovalIn: boundsToScreen)
    }
    set { }
}
```

Note that for shapes that don't fit neatly into `UIBezierPath`'s initializers, you'll have to do a bit more work, drawing the path itself when a initializer isn't available for a particular shape you're drawing:

```
// UIKit -> Xcode -> ContextualScenarioFig1ViewController.swift
let path = UIBezierPath()
```

```
drawTriangle(with: path, convertToScreenCoordinates: true)
return path
```

Now, shifting to each one feels much more fluid:



An accessibility path, or the lack thereof, in your app isn't likely to make or break an accessibility audit. But when it feels right to match the geometry of a shape, or possibly a circular annotation on a map view or some other similar interface element, the price of entry is so low for supporting it that it really doesn't seem to make sense *not* too.

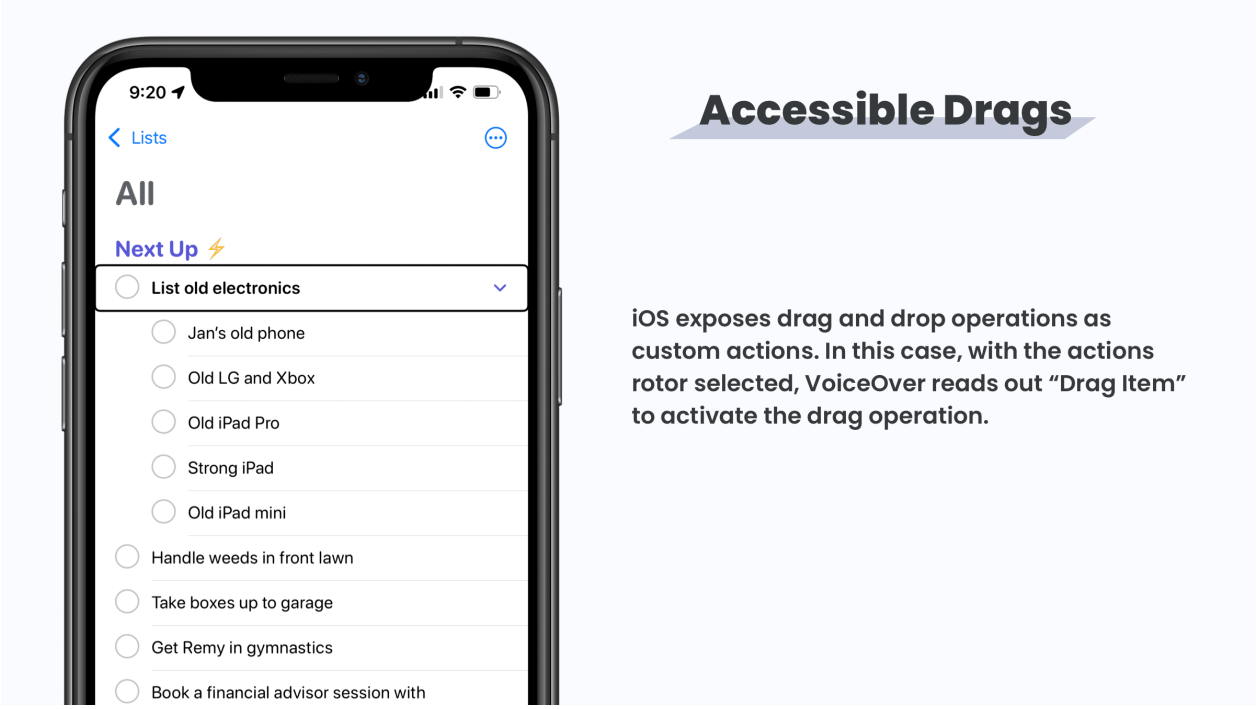
Finally, if you do set a custom path and business logic dictates that it's no longer relevant – simply assign nil back to it. This will result in the interface element using the default focus frame:

```
myCustomShape.accessibilityPath = nil
```

Drag and Drop

The best way to get started with comprehending how accessibility plays into drag and drop, understand what custom actions are and how they work. If you aren't familiar with them, read the "Custom and System Actions" chapter first and then come back.

The reason why that's so helpful is because accessibility considerations are vended on behalf of drag and drop much the same way custom actions are. VoiceOver surfaces drags via the "Actions" rotor, and when it's selected VoiceOver will read a custom name you supply (or a default one if you didn't) for any drag interaction.



Accessibility exposes drag and drops through custom actions.

When you think about breaking down what a drag and drop operations really is, you can pick apart the main themes and discover two primary components:

1. Knowing where, and at what point, a drag can occur.

2. Knowing where, and at what point, a drop can occur.

Basically — we're talking about locations. Where can I drag this? Where can I drop it?

And those are exactly the pieces of information what we need to hand off to the accessibility engine to create accessible drag and drops. It all starts with the `UIAccessibilityDragging` informal protocol.

`UIAccessibilityDragging`

This is exposed via an extension off of `NSObject`, which means pretty much anything in Cocoa Touch can use it. That means `UIView` instances or even custom accessibility elements such as `UIAccessibilityElement`. It only has two properties:

```
extension NSObject {

    @available(iOS 11.0, *)
    open var accessibilityDragSourceDescriptors:
[UIAccessibilityLocationDescriptor]?

    @available(iOS 11.0, *)
    open var accessibilityDropPointDescriptors:
[UIAccessibilityLocationDescriptor]?
}
```

Both are arrays of `UIAccessibilityLocationDescriptor`. As the names suggest, one is used to describe where drags can begin in the object, and the other describes where a drop can occur. That information is housed in the location descriptor.

`UIAccessibilityLocationDescriptor`

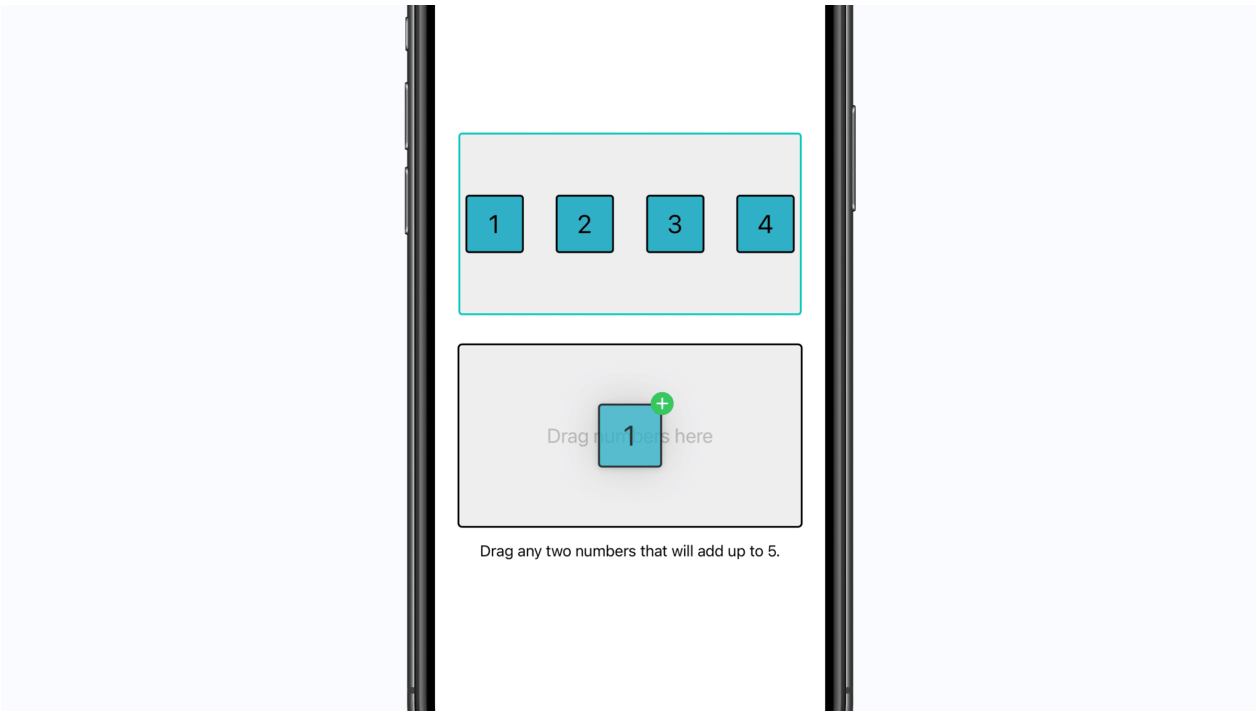
The location descriptor is a very simple object. It's going to answer the "where and what" question of the drag and drop process using three things:

1. A name to describe the drag or drop.
2. The point where it occurs.

3. And in what view.

Number three might seem like an odd choice for this API, but in reality – it's also what makes it so powerful. You can decouple the view hierarchy considerations from the actual drag and drops occurring in them using it. No matter if the drag or drop is a descendant of a view or a parent housing other ones within in, you can express those practically anywhere from your code that is most convenient.

Consider a math game for children, where they might drag and drop numbers to solve addition problems:



In this mock game, the goal is to drag any two numbers to the bottom which would add up to 5.

Each number can be create a drag (via `UIDragInteraction`) and the bottom "well" view accepts a drop (via `UIDropInteraction`). We can use the accessibility location descriptors to ensure VoiceOver users can participate.

For the drag objects (the digit views), it would look like this:

```
// UIKit -> Xcode -> ContextualScenarioFig2ViewController.swift ->
DigitView.swift

// In the initializer...
let dragDescriptor: UIAccessibilityLocationDescriptor =
UIAccessibilityLocationDescriptor(name: "Drag " +
accessibilityString, view: self)
accessibilityDragSourceDescriptors = [dragDescriptor]
```

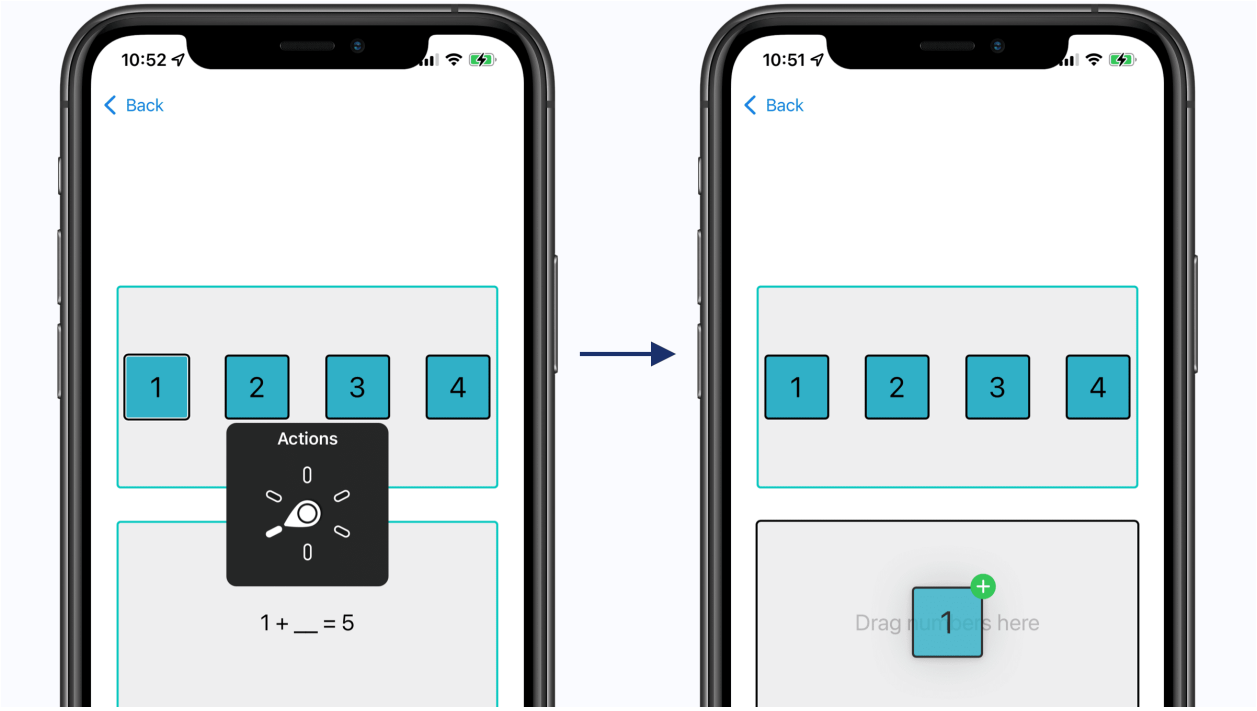
We've supplied two of the three bits of information here. The "what", using the `name` parameter. VoiceOver will announce this when the user is about to kick off a drag. Next, the "where" – in this view. The other part of the "where" question is answered by default for us (i.e. the `CGPoint` where the drag can begin). If you don't pass one, the accessibility engine uses the center of the view which is correct a majority of the time unless you've got multiple drag points. Finally, we assigned to the view's drag source descriptors to expose it to the accessibility engine.

Only two lines of code, and your drag operation is now accessible.

The good news is that exposing our drop locations is achieved much the same way. Here's the code for the "well" view which accepts the drops of digits to solve the addition problems:

```
// UIKit -> Xcode -> ContextualScenarioFig2ViewController.swift ->
DigitWellView.swift

// In the initializer
let dropDescriptor = UIAccessibilityLocationDescriptor(name: "Drop
Numbers", view: self)
accessibilityDropPointDescriptors = [dropDescriptor]
```

Now, if you run the code sample and use the "Actions" rotor, you'll see you can perform the whole drag and drop process using VoiceOver:



Before you can access a drag, make sure the Actions rotor is selected.

Finally, you can also supply these values "on demand", much like we did with the accessibility path above, by overriding these properties instead of doing direct assignments to them (as we've done here).

Tips

UIAccessibilityPath: Translate Custom Paths to Screen Coordinates

Pay close attention to the comments about `accessibilityPath`:

```
/*
 Returns the path of the element in screen coordinates.
```

```
 default == nil
 Setting the property, or overriding the method, will cause the
assistive technology to prefer the path over the accessibility.
 frame when highlighting the element.
 */
```

The path has to be expressed in *screen coordinates*. A lot of times, this is not an issue – because expressing geometry in screen coordinates is trivial on iOS, you simply pass nil in one of these functions:

```
// Nil will put the result in screen coordinates
myView.convert(rect, to: nil)
```

The challenge is that you're using a `UIBezierPath`. And true to its name, a lot of the times you're drawing a path every which way. If you're simply using a rectangle or a circle, then your job is easy (as you can see in the sample code, `ContextualScenarioFig1ViewController.swift`):

```
open override var accessibilityPath: UIBezierPath? {
    get {
        var boundsToScreen = self.convert(self.bounds, to: nil)

        switch desiredShape {
        case .circle:
            return UIBezierPath(ovalIn: boundsToScreen)
        case .square:
            return UIBezierPath(rect: boundsToScreen)
        case .triangle:
            let path = UIBezierPath()
            drawTriangle(with: path, convertToScreenCoordinates:
true)
            return path
        case .rectangle:
            boundsToScreen.size = CGSize(width: 100, height: 50)
            return UIBezierPath(rect: boundsToScreen)
        }
```

```
    }
    set { }
}
```

However, the triangle requires you to draw itself at each point using the screen coordinates:

```
private func drawTriangle(with path:UIBezierPath,
convertToScreenCoordinates:Bool = false) {
    path.move(to: self.convert(CGPoint(x: 0, y: 100), to:
convertToScreenCoordinates ? nil : self))
    path.addLine(to: self.convert(CGPoint(x: 50, y: 0), to:
convertToScreenCoordinates ? nil : self))
    path.addLine(to:self.convert(CGPoint(x: 100, y: 100), to:
convertToScreenCoordinates ? nil : self))
    path.close()
}
```

It's a bit of an eyesore, but once you realize how to go about it (i.e. draw each path according to the screen) – you can abstract out the code however you like.

UIAccessibilityPath: Override the Path, Don't Assign to It

There's one more important thing to take away from the code sample above:

```
open override var accessibilityPath: UIBezierPath? {
    get {
        // Path Logic
    }
    set { // do.nothing() }
}
```

We're overriding the path, not assigning to it like this:

```
customShape.accessibilityPath = thePathCode()
```

Why?

Because this likely needs to be updated by the accessibility engine quite often. Consider if these shapes were in a scroll view, where the content offset is changing every frame – and as such, where the shape is at in terms of the screen's coordinates. Overriding it and calculating it on the fly ensures we are giving iOS the right frame, at the right time, right when it asks for it.

Drag and Drop: Flattening View Hierarchy

Take a look back at the `DigitView.swift` example and you'll notice the view is composed of two, very simple, parts:

1. The view subclass itself.
2. And the `UILabel` it adds on top of it, matching its height and width.

When exposing our drag via `accessibilityDragPointDescriptors`, if we *didn't* opt the `UILabel` out of the accessibility engine, it wouldn't work. The label would swallow it up. Just as we've talked about flattening your view hierarchy before, you may need to do the same thing here to make your drag and drops accessible:

```
// UIKit -> Xcode -> ContextualScenarioFig2ViewController.swift ->
DigitView.swift

let accessibilityString: String = "Number \(digit.description)"
accessibilityLabel = accessibilityString

// Mark the view as the accessibility element
// Opt the label out
numLabel.isAccessibilityElement = false
isAccessibilityElement = true

accessibilityTraits = [.button]

let dragDescriptor: UIAccessibilityLocationDescriptor =
```

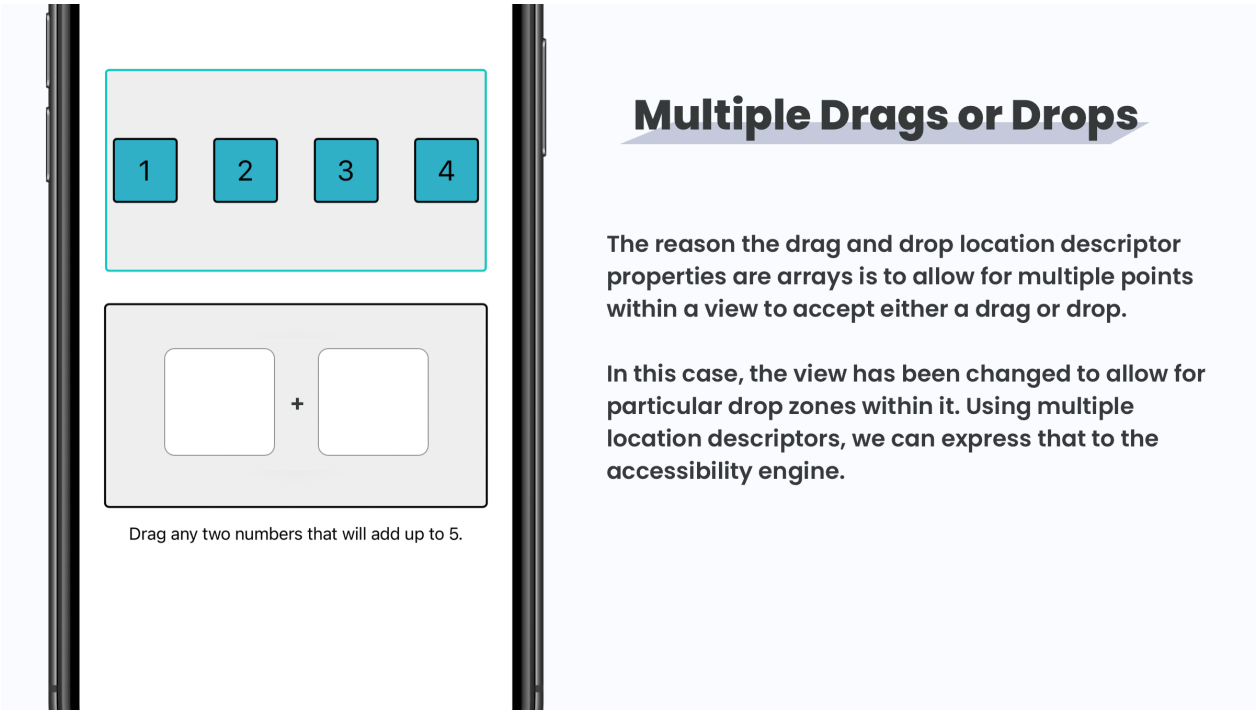
```
UIAccessibilityLocationDescriptor(name: "Drag " +
accessibilityString, view: self)
accessibilityDragSourceDescriptors = [dragDescriptor]
```

However, if your implementation details prevent that – recall that you can pass any view you want to when creating a `UIAccessibilityLocationDescriptor` instance. So, you *could* use the label in this case, if you needed too.

Drag and Drop: Multiple Drop Points in One Element

If you've finished this chapter and thought to yourself, "Why are these `UIAccessibilityDragging` properties arrays?", that's a great question. The reason is because one view, or object, could be exposing multiple drags or multiple drops within it. That's also why the `point` argument exists in the initializers.

Consider if we changed the "well" view to have two distinct zones where the numbers were dropped:



Expose multiple drag points or drop zones by assigning them to the array of location descriptors.

In that case, we would change the assignment to the drop descriptors to look like this:

```
let leftCoefficientDropDescriptor =
UIAccessibilityLocationDescriptor(name: "Drop Left Coefficient",
point: leftCoefficientPoint, in: self)
let rightCoefficientDropDescriptor =
UIAccessibilityLocationDescriptor(name: "Drop Right Coefficient",
point: rightCoefficientPoint view: self)

accessibilityDropPointDescriptors = [leftCoefficientDropDescriptor,
rightCoefficientDropDescriptor]
```

Drag and Drop: What about Table and Collection Views?

If you're using a table or collection view, UIKit is taking care of the details behind `UIAccessibilityDragging` for you assuming you're using their built-in drag and drop APIs. For example, a table row has drag descriptors that communicate the

"where" part of the accessibility location descriptor. In this case, it relays where in the table view you can activate a drag of the row itself.

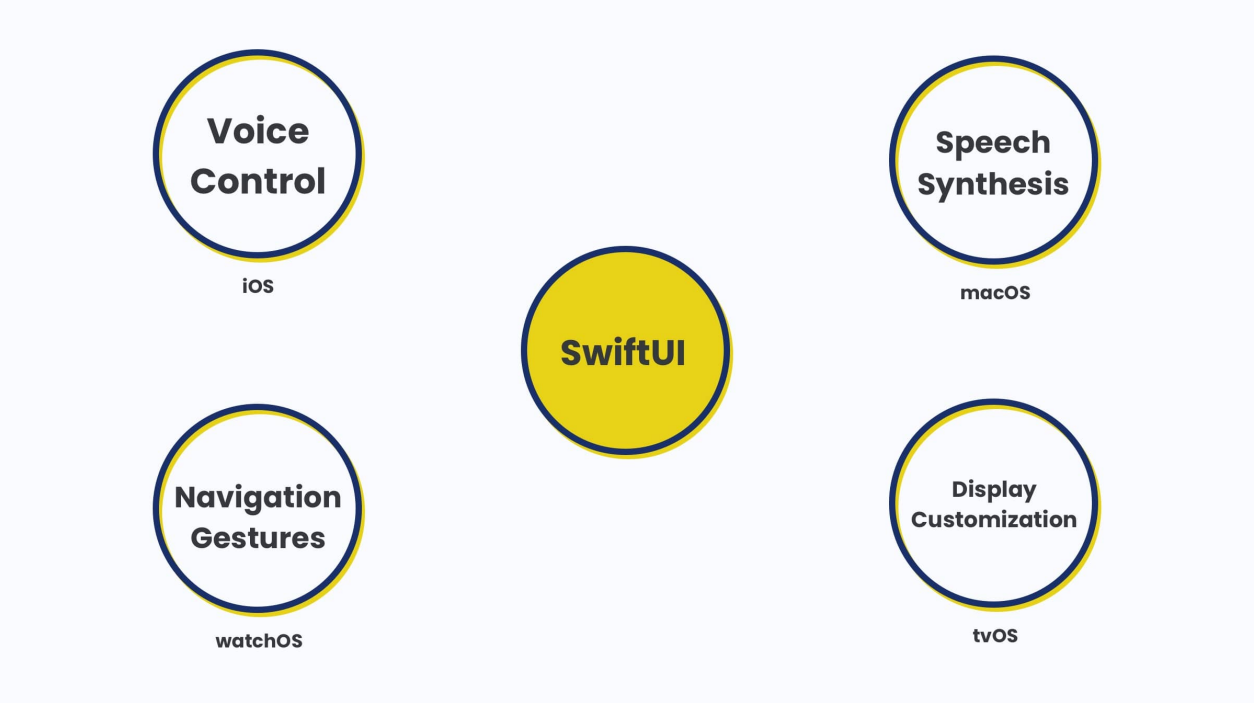
That isn't to say you can't implement these APIs yourself. If you find the drags aren't descriptive enough, for example, you could supply a better suited name using your own accessibility location descriptors.

Three Key Takeaways

1. Use customized accessibility paths to promote a more fluid user experience for accessibility focus frames.
2. Drag and drop APIs are accessible in many controls by default, but you can make them even more expressive via `UIAccessibilityLocationDescriptor` objects.
3. Make sure any drag or drop interaction is exposed to the rotor control, and test them out in any area that leverages `UIDragInteraction` or `UIDropInteraction`.

SwiftUI Techniques

SwiftUI, since its introduction, has faced a tall task — measure up to four mature interface frameworks. Two of which (UIKit and AppKit) that have been around, combined, for decades. As if that weren't challenging enough, the engineers at Apple also had to consider accessibility. How do you bridge these important frameworks from tvOS, watchOS, iOS and macOS and bring them together under one declarative umbrella?



Accessibility technology specialize across all of Apple's platforms, but many of the underpinnings are the same such as VoiceOver.

Now, several years into its lifecycle, SwiftUI has become more capable with each release and throughout the years it has exposed nearly all of the accessibility APIs UIKit has used since its introduction in 2008. This feat of engineering has been fascinating to watch from the outside. It's not only close to reaching framework parity in terms of accessibility, in many instances it makes it much easier to program against. For example, look at the Rotor Control APIs in UIKit versus SwiftUI. Or customized content. What takes a hundred or more lines of code is achieved with a few modifiers in SwiftUI:

```
// A few modifiers is enough to express customized
// Accessibility content in SwiftUI.

var body: some View {
    Image(foodName)
        .resizable()
        .aspectRatio(nil, contentMode: .fit)
        .frame(width: 300, height: 300)
```

```
        .accessibilityLabel(foodName)
        .accessibilityCustomContent("Description", "Healthy food
platter")
        .accessibilityCustomContent("Created", "July 8th, 2021")
        .accessibilityCustomContent("Camera Model", "Leica Deluxe 7
Digital Camera")
        .accessibilityCustomContent("Location", "Ozark Missouri",
importance: .high)
}
```

This chapter is all about celebrating these accessibility APIs in SwiftUI. How do they work? Can we draw some similarities to how they function in UIKit to better understand them? More importantly, which APIs are *specific* to SwiftUI?

First, let's review what we've already covered in terms of accessibility in SwiftUI:

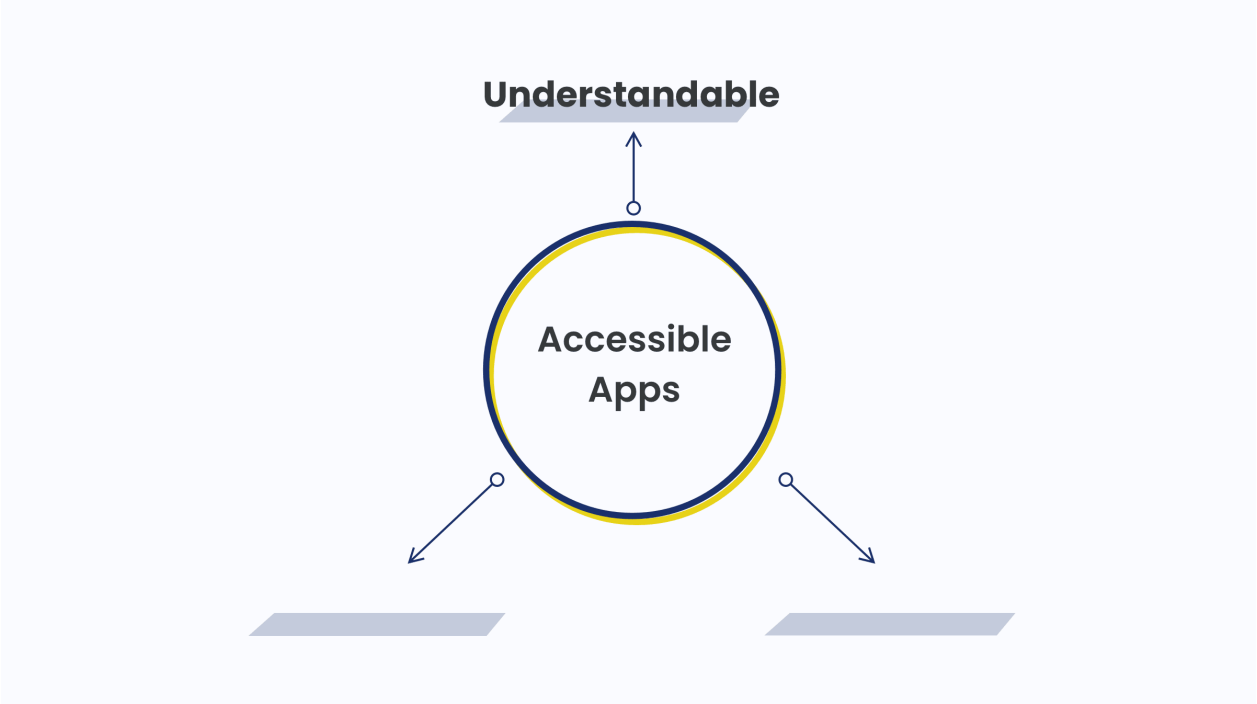
- VoiceOver
- The Rotor Control
- Custom Actions
- Voice Control
- Customized Accessibility Content (i.e. `AXCustomContent`)
- Textual Context
- Dynamic Type
- Adaptable Interfaces
- Focus
- User Preferences (i.e. preferring less motion)

As such, we won't retread any ground here. If you need more information on how any of those APIs function in SwiftUI, please review those chapters. Now, let's take a high level look at how accessibility APIs function in SwiftUI.

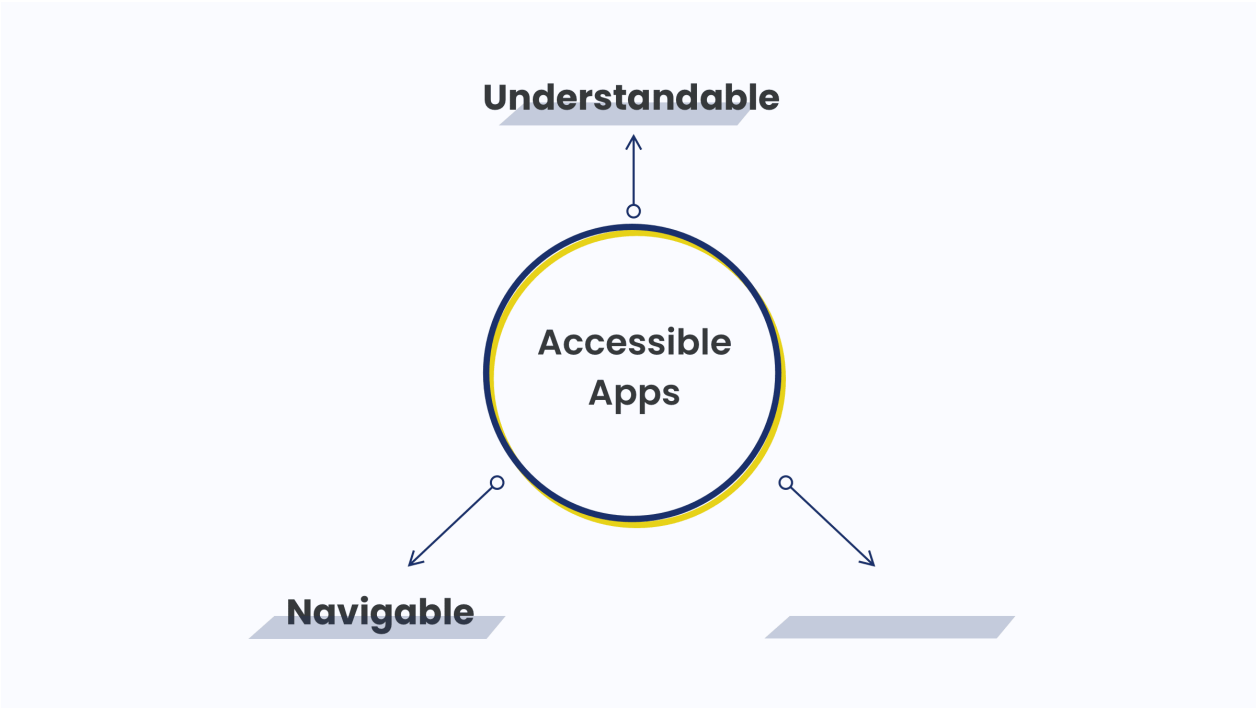
How it Works

SwiftUI's accessibility system aims to enforce three key aspects of using iOS software. Together, these ideas form a system that make software available to anyone, no matter the accessibility need.

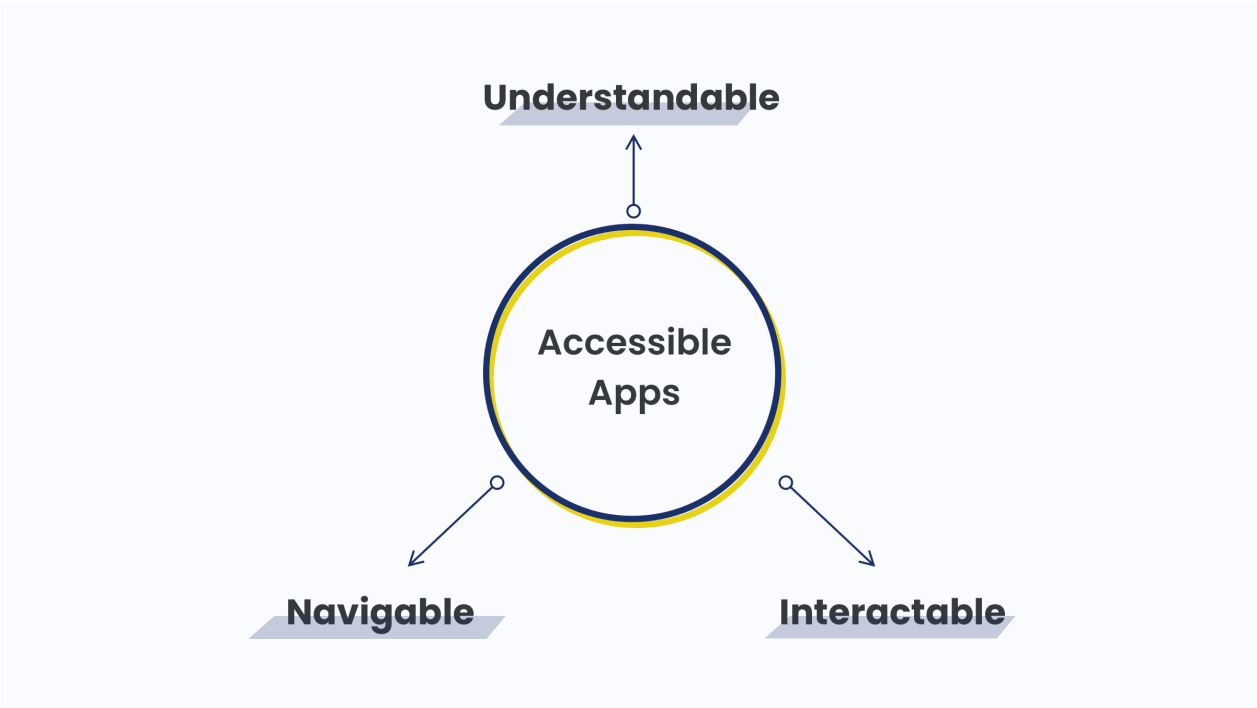
First, it should make things *understandable*. If users don't know the context or meaning of things, then your app's experience stops there. By providing information to the accessibility engine, we can make sure all SwiftUI interfaces are understood and no context is lost or (perhaps even worse) presented in confusing ways.



Once things are understandable – we can move to making them *navigable*. Understandability only goes so far, users need to move around your interface at a bare minimum but they should also be able to do it with a high degree of efficiency.



Finally, once context is achieved and users can move around – the final piece is to make sure interface elements are *interactable*. Ensuring users can touch, tap and move things is what makes software tangible and usable.



Like with UIKit, SwiftUI aims to make much of your app "accessible by default" which rings true for the most part in my experience. Its declarative nature coupled with state tracking means that many things can be done for us, such as posting accessibility notifications. When our interface, and what it can become, is defined from the beginning – things like accessibility labels, values and traits are often packaged up for us.

Under the hood, SwiftUI assembles a view graph which is translated into native components depending on the platform on which it's executed. A `List` on iOS could end up as a table or collection view, or something entirely different on watchOS, tvOS or macOS. Through this generic system, the view tree builds up "accessibility elements" from its nodes that make all of these accessibility APIs work under the hood.

Once the view graph is passed to a host view, the accessibility engine starts to vend information to VoiceOver, Voice Control or any other marquee accessibility API iOS offers. Though, because we now know that SwiftUI builds up a view tree – it also

stands to reason that the fewer views within that tree, the simpler accessibility becomes. And that's *true*, but also something I personally don't worry too much about.

We've talked about "flattening" hierarchy before, and all of those same ideas apply here. There is no replacement for trying things out on your device, too. So, going forward, it pays to be mindful of *how* accessibility works in SwiftUI — it's great to know! However, you'll sink or swim based on your implementation and willingness to test out accessibility scenarios.

With that said, let's look at SwiftUI specific accessibility APIs and ideas.

Accessibility Elements

Let's start with the idea of how the view tree eventually influences the resulting accessibility elements that SwiftUI produces. As with UIKit, we can essentially create custom accessibility elements by using the `.accessibilityElement(children:)` modifier. With it, we can let SwiftUI know how to treat the elements it might contain.

This is all driven by the `AccessibilityChildBehavior`. This struct lets us express to SwiftUI how the children should be viewed by the accessibility engine, and under the hood it does any work necessary to make those change be reflected. For example, using the `.contain` value actually wraps the resulting `View` instance into a `UIViewRepresentable` type. This is all abstracted away from you, true to SwiftUI's declarative nature — but it's helpful to understand.

Creating custom accessibility elements is the basis for most of your SwiftUI accessibility work. It's simple and powerful, but if you don't know how it works upfront — any accessibility work to make things accessible can be frustrating. Let's take a look at each of the three values to see how they work.

Consider a simple `VStack` with a couple text controls within it. This is how each example will be constructed:

```
VStack {
    Text("First Text Control")
    Text("Second Text Control")
}
```

As we'll discover, by using either `.combine`, `.contain` or `.ignore` we can get very different results:

Combine

When you use the `.combine` value, you're telling SwiftUI to "flatten" the view hierarchy and treat the element as one. This is great for navigational purposes, and if you have a depth-heavy view tree, it's typically the correct choice to consider.

```
// Xcode -> SwiftUITechniquesFig1View.swift

struct SwiftUITechniquesFig1View: View {
    var body: some View {
        VStack {
            Text("Combine Children")
                .font(.headline)
            Text("Both Text view values are reported.")
        }
        .accessibilityElement(children: .combine)
    }
}
```

For VoiceOver, this would have a value of *both* of the `Text` view's text which results in "Combine Children, Both Text view values are reported." Its trait would also be `.isStaticText`. For the purposes of accessibility, these three Views are smashed together (the `VStack`, and two `Text` views).

However, using this approach also assumes that you semantically *can* treat all of these elements as one. That's not always the case. For example, if this view was heavy with interactivity you may not be able to represent all of those varied navigational interactions to the accessibility engine easily when combining the views together.

There is a silver lining, though. SwiftUI can take things like a `Button` and translate those interactions into custom accessibility actions for you. That's an incredible amount of work you can get for free. So, as long as you are using "stock" components – it's likely that your work will be expressed to the accessibility engine:

```
var body: some View {
    VStack {
        Text("Combine Children")
            .font(.headline)
        Text("Both Text view values are reported.")
        Button("Tap Me") {
            // This button and its action will be pulled
            // Out as a custom accessibility action for us
        }
    }
    .accessibilityElement(children: .combine)
}
```

More customized controls may not get that work for free, though. This is a common theme throughout the book. The more custom you go, the more work that may lie ahead to make things accessible. The inverse is true, too. The most effective way to find out is to run your app on a device and try it yourself.

Contain

When we use contain, we tell SwiftUI that the view which is using the `.accessibilityElement(children: .contain)` is an accessibility container. This means all of its children will be navigated in order and the next accessibility container won't be focused until those views have been cycled through first.

Why do this? It's great for things like a list of data. It's likely you want the user to cycle through that data before reaching things below it, such as a tool bar or another view. That's how users who navigate via visual means would think about the interface.

```
// Xcode -> SwiftUITechniquesFig2View.swift

struct SwiftUITechniquesFig2View: View {
    var body: some View {
        VStack {
            Text("Contain Children")
                .font(.headline)
            Text("Text views report value, and are navigated in
order since this VStack is now an accessibility container.")
        }
        .accessibilityElement(children: .contain)
    }
}
```

Now, the `VStack` doesn't report a value for VoiceOver to read directly - instead it indicates that it has two children that VoiceOver should know about. The user can choose each one to focus individually – unlike our example above where everything was treated as one.

You may think that this approach could make navigation slow – but you shouldn't. Remember, the Rotor Control API lets VoiceOver users skip around to other containers if they need to - which makes this much less of an issue. Even better, this value will opt the interface element into the `Container` rotor for you. It still *can* be an issue, though, if you've supplied the wrong accessibility traits to other parts of your interface.

Ignore

True to its name, this approach opts out the view and its children from the accessibility engine. You may be thinking this is ideal to hide decorative interface elements

such as icons which might be inconsequential to VoiceOver users — and that's true, but it also is a great way to express complicated or verbose views a little easier.

For example, in our code we've simply got two basic views with text. When we tack on the `.ignore` modifier, this whole element doesn't interact with VoiceOver. A new accessibility element is created which reports as much. So, if your intention is to still make the view accessible but also simpler, then it's on you to add the correct traits and any of the "Big Three" accessibility values.

Here, the view remains accessible because after we told SwiftUI to opt-out the view to the accessibility engine, we turn around and added the necessary information to let VoiceOver users know the context and intention behind the interface element:

```
// Xcode -> SwiftUITechniquesFig3View.swift

struct SwiftUITechniquesFig3View: View {
    var body: some View {
        VStack {
            Text("Ignore")
                .font(.headline)
            Text("Both Text view values are reported.")
        }
        .accessibilityElement(children: .ignore)
        .accessibilityValue("Ignore behavior. Children aren't
reported.")
        .accessibilityAddTraits(.isStaticText)
    }
}
```

Default Behavior

So what happens if we don't do anything? In this case, we get the implicit values SwiftUI vends. In the sample code we've been working with, a few things happen:

- The `VStack` is treated as a container.

- The two `Text` views report individually to the accessibility engine.
- There are two accessibility elements.

```
// Xcode -> SwiftUITechniquesFig4View.swift

struct SwiftUITechniquesFig4View: View {
    var body: some View {
        VStack {
            Text("Default Behavior")
                .font(.headline)
            Text("Both Text view values are reported.")
        }
    }
}
```

After all of that information, you might still be thinking "When do I know which one to use?" Deciding that is an exercise in recognizing data density, the complexity of the interface and the purpose of the control:

- **Contain** is useful for depth-heavy controls like lists, and it's navigated according to the sort order of its children.
- **Combine** essentially is the "flattening" technique we've discussed often in other chapters. It helps ease navigation.
- **Ignore** can help you make complex controls easier to surface to the accessibility engine, or opt out controls that don't really service VoiceOver users.

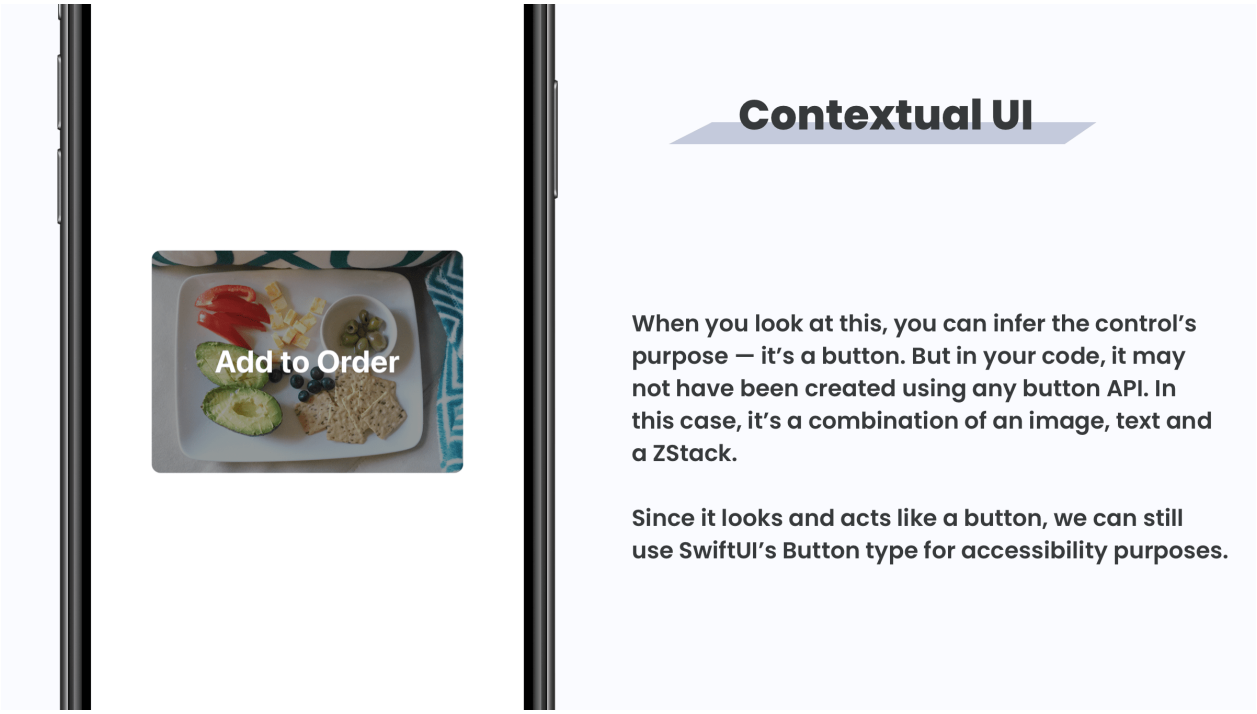
Accessibility Representation

The accessibility representation modifier provides a convenient way to avoid the following quandary:

I've made this amazing custom control, but now I've missed out on Apple's accessibility work that comes out of the box with standard controls.

How? It lets you reflect any arbitrary view or existing control Apple has already made to the accessibility engine *in place* of the one that's actually created on device.

Consider a food ordering app, where the interface might have the picture of a meal with text overlaid that says "Add to Order".



Such a view could be composed like this:

```
// Xcode -> SwiftUITechniquesFig5View.swift

struct SwiftUITechniquesFig5View: View {
    var body: some View {
        ZStack {
            Image("Healthy Food")
                .resizable()
            Color.black
```

```
                .opacity(0.4)
            Text("Add to Order")
                .font(.title)
                .fontWeight(.bold)
                .foregroundColor(.white)
        }
        .cornerRadius(8.0)
        .frame(width: 280, height: 200, alignment: .center)
        .onTapGesture {
            onOrder()
        }
    }
}
```

What we have here is a component that is acting like a button, but isn't represented to the accessibility engine that way. Using the accessibility representation modifier, we can tell SwiftUI that this is simply a `Button` to the accessibility engine – and the view remains exactly the same when it's run in your app:

```
// Xcode ->

struct SwiftUITechniquesFig5View: View {
    var body: some View {
        ZStack {
            Image("Healthy Food")
                .resizable()
            Color.black
                .opacity(0.4)
            Text("Add to Order")
                .font(.title)
                .fontWeight(.bold)
                .foregroundColor(.white)
        }
        .cornerRadius(8.0)
        .frame(width: 280, height: 200, alignment: .center)
        .accessibilityRepresentation {
            Button("Add to Order") { }
```

```
            }
            .onTapGesture {
                onOrder()
            }
        }

        private func onOrder() { /* do.nothing() */ }
}
```

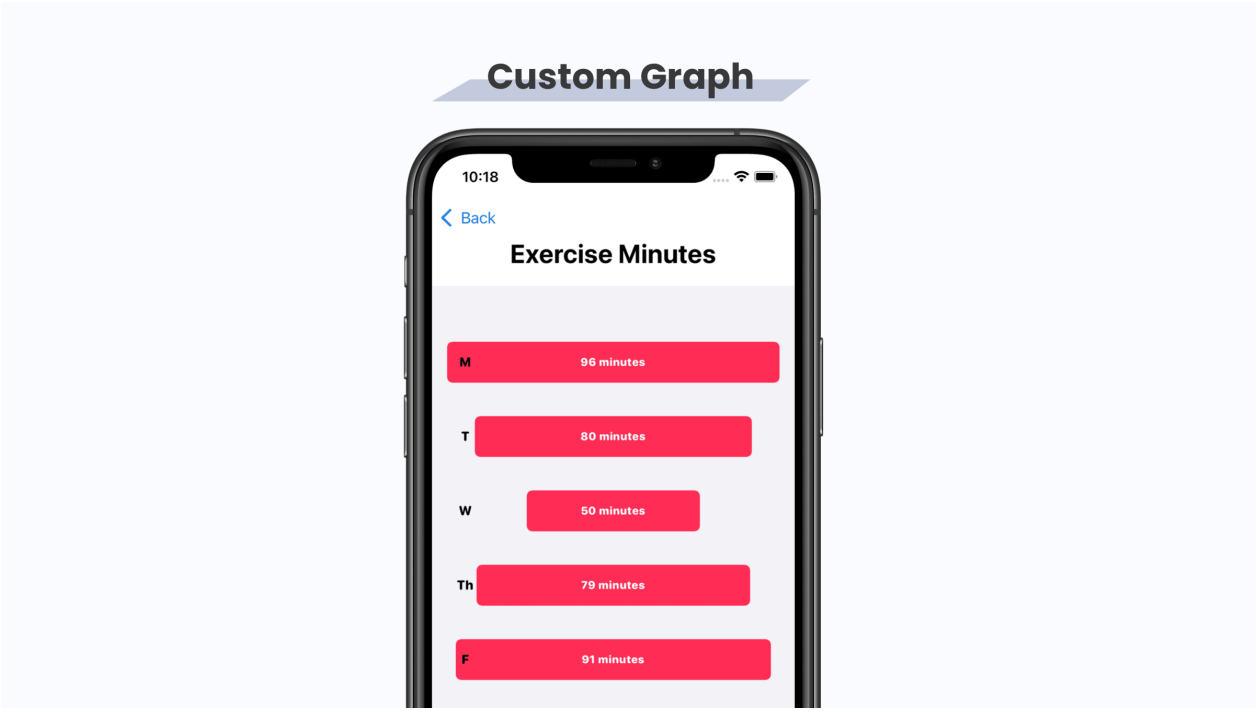
Now, the accessibility engine sees this entire `VStack` as a button with a label of "Add to Order", and that's exactly what we want. Whenever Apple improves or changes the meaning of how buttons work with the accessibility engine, we'll gain those improvements for free.

One thing to note above, SwiftUI is hiding the view we vended in the `accessibilityRepresentation` modifier and is making it non-interactive. A non-interactive button won't get us anywhere, so that's why you still see the `onTapGesture` modifier. In terms of the accessibility representation that you provide, the accessibility engine is only using it to create accessibility elements on our behalf.

In a one-sentence pitch, accessibility representation does this: It allows you to use any control or view for accessibility purposes in place of the real view that is created.

Even better, you can use the `ViewBuilder` parameter to create something like a `ForEach`. This is ideal for controls like a custom drawn graph, perhaps created from the `Canvas` APIs. In those cases, you could use the same data point used in the graph in a `ForEach` — making every bar item accessible, focusable and more.

If you were to build a health app which recorded exercise minutes, perhaps you'd add a bar graph of some sort to visualize that data. Maybe you wanted to spruce things up a bit and implement a vertical based data flow instead of the traditional horizontally based graphs. It might look something like the one here:



Custom drawn controls like these used to be challenging to make accessible, but the accessibility representation modifiers make the job much easier.

This is drawn completely with SwiftUI's `Canvas` view, and as such - there is no accessibility we gain out of the box. And, how could we? The framework can't very well ascertain what our intentions and meanings are behind the things that we've drawn.

Here, we could return a `ForEach` nested in a `VStack` to reflect the data as it appears. Using the same exact data model that powers the custom drawn graph, we can return a `ForEach` to make each individual chart line its own accessibility element. That's the approach we take to make this custom graph accessible, and you can see it implemented as the last modifier in this code sample:

```
// Xcode -> SwiftUITechniquesFig6View.swift

var body: some View {
    VStack {
        Text("Exercise Minutes")
            .font(.title)
```

```
            .fontWeight(.bold)
        Canvas { ctx, size in
            caloricIntake.forEach {
                let caloryIDX = Double(caloricIntake.firstIndex(of:
$0) ?? 0)
                let barItemHeight = 44.0
                let barItemWidth = CGFloat(CGFloat($0)/
caloricCeiling) * size.width
                let vOffset = (barItemHeight/2) + (caloryIDX * 80.0)
                let cornerRadius = 6.0
                let barItemRect = CGRect(x: (size.width/2) -
(barItemWidth/2), y: vOffset +
cornerRadius, width: barItemWidth, height: barItemHeight)

                let barItemPath = Path(roundedRect: barItemRect,
cornerRadius: cornerRadius,
style: .continuous)
                ctx.fill(barItemPath, with:.color(.pink))

                let textItemPoint = CGPoint(x: barItemRect.midX, y:
barItemRect.midY)
                ctx.draw(Text("\($0) minutes")
                        .font(.system(size: 12, weight: .heavy,
design: .rounded))
                        .foregroundColor(.white),
                         at: textItemPoint)

                let textLegendItemPoint = CGPoint(x: 20, y:
barItemRect.midY)

ctx.draw(Text(dayOfWeekFor(index:Int(caloryIDX)).short)
                        .font(.system(size: 14, weight: .heavy,
design: .rounded))
                        .foregroundColor(.black),
                         at: textLegendItemPoint)
            }
        }
        .padding([.leading, .trailing], 16)
        .padding(.top, 32)
```

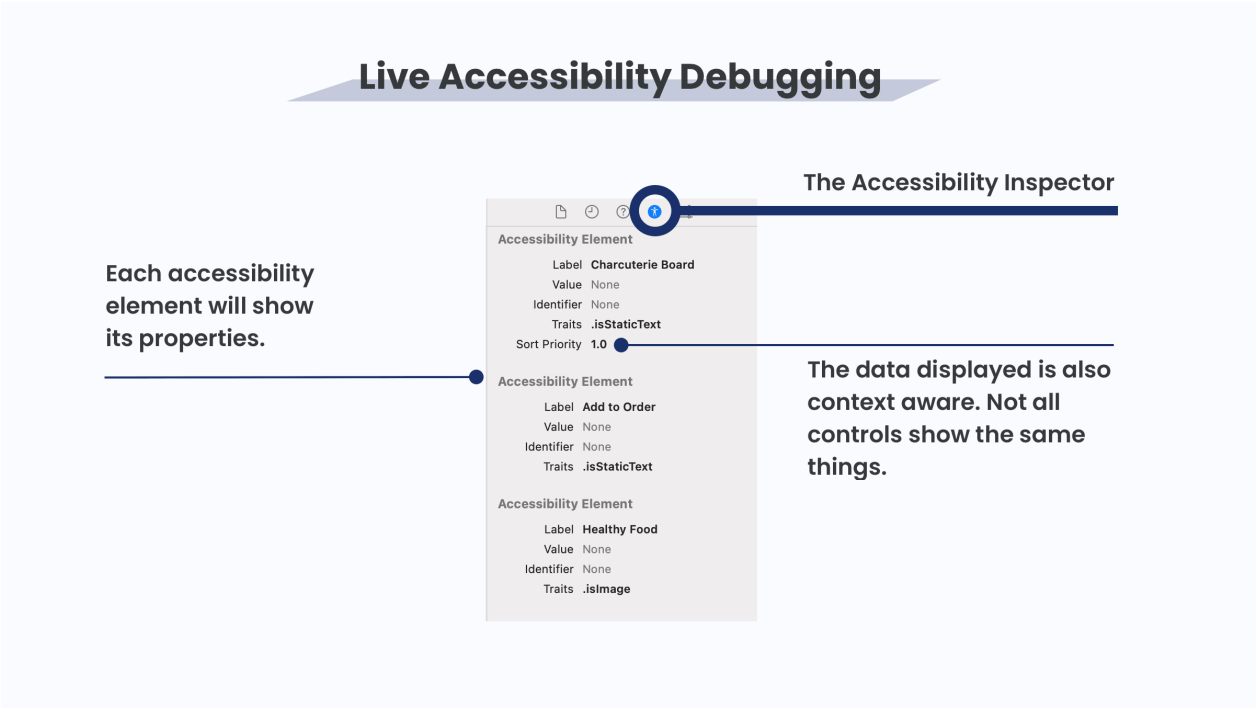
```
        .background { Color(uiColor: .systemGroupedBackground) }
        .accessibilityRepresentation {
            VStack {
                ForEach(caloricIntake.indices, id: \.self) { index
in
                    RoundedRectangle(cornerSize: .zero)
                        .accessibilityValue("\(caloricIntake[index])
minutes of exercise.")
                        .accessibilityLabel("Exercise for \
(dayOfWeekFor(index:index).long)")
                }
            }
            .accessibilityElement(children: .contain)
        }
    }
}
```

No matter the interface, SwiftUI has so many high level abstractions available in its accessibility APIs that you can make literally *anything* accessible, so long as you know which tool to reach for.

GUI Inspector

Xcode Previews and the live canvas now allow you to see exactly what the accessibility engine is doing with your interface. True to SwiftUI's lofty goals of preventing developers from having to constantly compile and run their apps via the simulator or on device, so to can you skip that process for accessibility debugging purposes.

If you've got a preview running, Xcode's inspector also has an Accessibility Inspector specifically for SwiftUI:



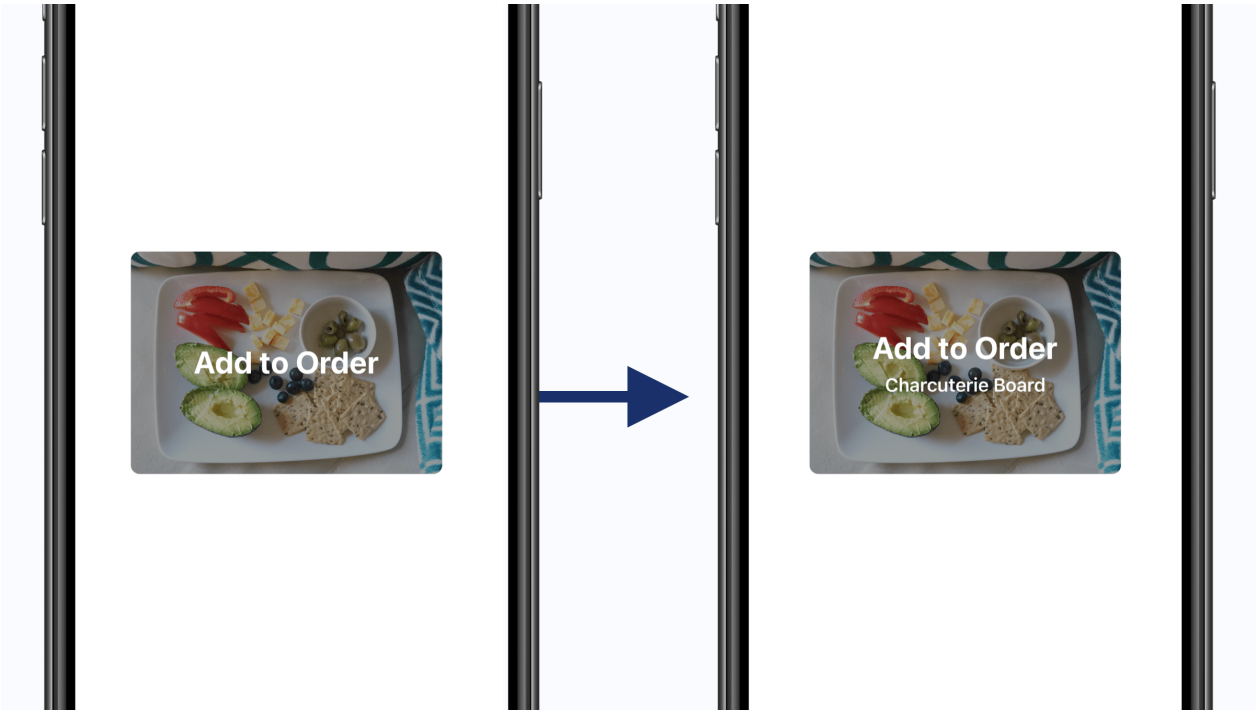
This is the fastest way to check high impact accessibility values such as its label, value, identifier, traits, number of children, sort order and more. Even better, those changes are reflected in real time alongside any code changes you make.

If you were to revisit the accessibility element examples that we started with, you could see all of this information in the inspector when you highlight the code.

Sort Priority

SwiftUI assigns a sort priority of 0 to all accessibility elements within a container. Even though SwiftUI will navigate to elements by their sort priority first and foremost, if they all start with the same value – that means it'll navigate to them in the order you inserted them into the view tree.

Higher sort priority values get precedence. This becomes important when you have a container of elements. Let's revisit our example of a food ordering app, only this time we've added the name of the item to the interface as well:



And here's what the code looks like:

```
// Xcode -> SwiftUITechniquesFig7View.swift

struct SwiftUITechniquesFig7View: View {
    var body: some View {
        ZStack {
            Image("Healthy Food")
                .resizable()
            Color.black
                .opacity(0.4)
            VStack(spacing: 6.0) {
                Text("Add to Order")
                    .font(.title)
                    .fontWeight(.bold)
                Text("Charcuterie Board")
                    .fontWeight(.semibold)
            }
            .foregroundColor(.white)
        }
        .cornerRadius(8.0)
```

```
            .frame(width: 280, height: 200, alignment: .center)
    }

    private func onOrder() { /* do.nothing() */ }
}
```

Currently, three elements will report to the accessibility engine and in this order:

1. The “Add to Order” text.
2. The “Charcuterie Board” text.
3. The image.

This is an issue, because a VoiceOver user would want to know *what* they are ordering before they actually do it. We can patch this up with a sort priority modifier:

```
// Xcode -> SwiftUITechniquesFig7View.swift

var body: some View {
    ZStack {
        Image("Healthy Food")
            .resizable()
        Color.black
            .opacity(0.4)
        VStack(spacing: 6.0) {
            Text("Add to Order")
                .font(.title)
                .fontWeight(.bold)
            Text("Charcuterie Board")
                .fontWeight(.semibold)
                .accessibilitySortPriority(1)
        }
        .foregroundColor(.white)
    }
    .cornerRadius(8.0)
    .frame(width: 280, height: 200, alignment: .center)
}
```

With that change, the order would now look like this:

1. The “Charcuterie Board” text.
2. The “Add to Order” text.
3. The image.

There is no need to sacrifice or bend on interface design or accessibility impact in these situations – you can have the best of both.

Always audit your container accessibility views to ensure they have sensible ordering. Remember, it’s all about context – and a hallmark of delivering on sensible context is that you also vend sensible ordering to the elements, no matter what they look like in the interface.

Accessibility Zoom Action

People with physical or motor disabilities can have a hard time performing pinch gestures. To that end, if you have something like an image which can be zoomed in or out of, use the accessibility zoom action to make this easier for these users.

```
// Xcode -> SwiftUI -> SwiftUITechniquesFig8View.swift

struct SwiftUITechniquesFig8View: View {
    @State private var zoomValue = 1.0
        @State var imageName: String?
        var body: some View {
            Image("Sky")
                .scaleEffect(zoomValue)
                .accessibilityZoomAction { action in
                    let zoomQuantity = "\(Int(zoomValue)) x zoom"
                    switch action.direction {
                    case .zoomIn:
                        zoomValue += 1.0
```

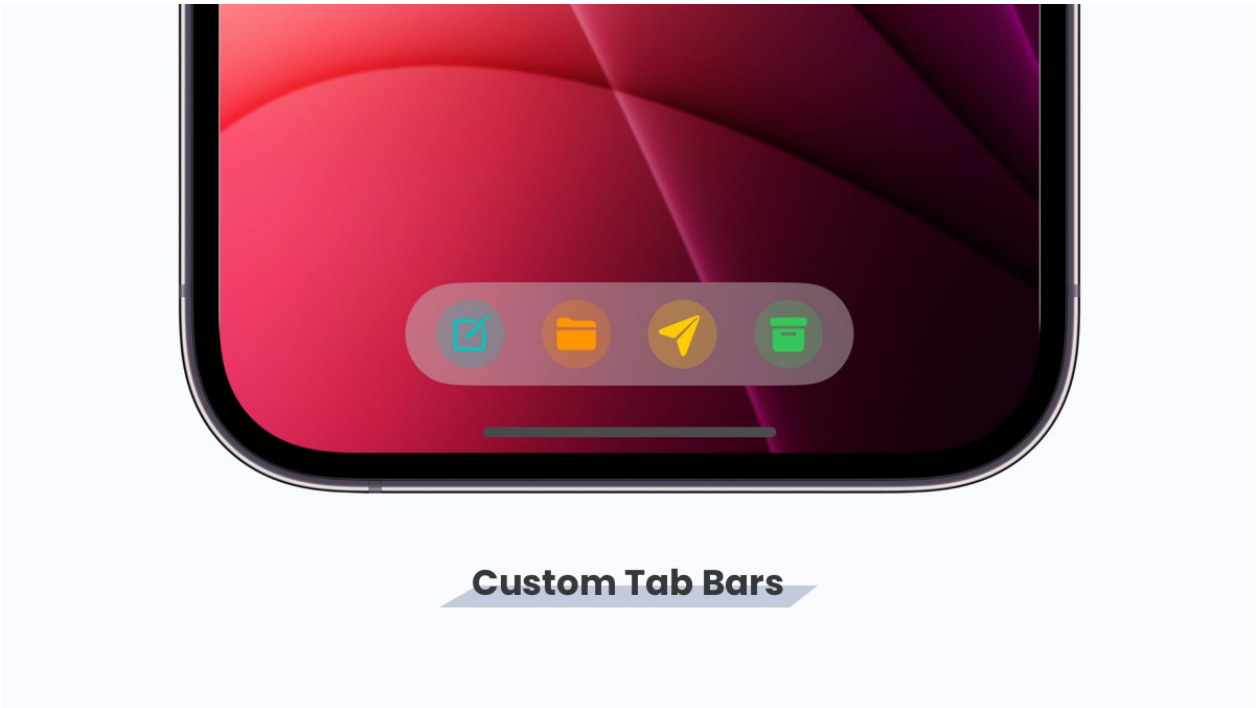
```
                        AccessibilityNotificatio
.Announcement(zoomQuantity).post()
                    case .zoomOut:
                        zoomValue -= 1.0
                        AccessibilityNotificatio
.Announcement(zoomQuantity).post()
                    }
                }
        }
}
```

Using the `.accessibilityZoonAction` modifier on the `Image` view struct, we can enable zooming using assistive technologies. However, it's still best practice to post an `AccessibilityNotification` of the zoom quantity.

With these changes in place, once the user selects the `zoom` option from the Rotor Control, they can perform zooming in and out of the image.

Custom Tab Bars

With SwiftUI's ability to make custom interfaces easier than ever, the need to accurately represent such interfaces comes up often. One such element that shows up in a custom fashion? A customized tab bar, much like this one:



This tab bar is custom made, and doesn't use any of SwiftUI's built-in tab bar APIs.

In cases like this, we need to tell iOS that this acts as a tab bar:

```
// Xcode -> SwiftUI -> SwiftUITechniquesFig9View.swift

struct CustomizedTabBar: View {
    private let tabs: [(String, Color)] = [
        ("square.and.pencil.circle.fill", .mint),
        ("folder.circle.fill", .orange),
        ("paperplane.circle.fill", .yellow),
        ("archivebox.circle.fill", .green),
    ]

    var body: some View {
        HStack(spacing: 12) {
            ForEach(tabs, id: \.self.0) { tabData in
                Button {

                } label: {
                    Image(systemName: tabData.0)
                        .symbolRenderingMode(.hierarchical)
```

```
                        .foregroundStyle(tabData.1)
                        .font(.largeTitle)
                }
            }
        }
        .padding(EdgeInsets(top: 6, leading: 12, bottom: 6,
trailing: 12))
        .background {
            Capsule()
                .fill(.ultraThinMaterial)
        }
        .padding(.bottom, 64)
        // Mark to the AX Engine as tab bar
        .accessibilityAddTraits(.isTabBar)
    }
}
```

Now, technologies like Voice Over will recognize the component as a tab bar. Another alternative? Use the accessibility representation APIs we discussed earlier.

Conditional Accessibility Labels

There are several times where accessibility labels don't apply. For example, perhaps an element is disabled, hidden or otherwise shouldn't be recognized. While you can always hide and show elements like this:

```
if componentIsActive {
    MyCustomComponent()
        .accessibilityLabel(Text("Component AX Label"))
}
```

What about when you have user experience which necessitates that components shouldn't be removed? In that case, you can also pass a boolean to tell the accessibility engine to respect or ignore the accessibility label:

```
// Xcode -> SwiftUI -> SwiftUITechniquesFig10View.swift

VStack {
    DemoPickerView()
    Button {
        playSelectedDemo()
    } label: {
        Image(systemName: "play.fill")
    }
    .accessibilityLabel(Text("Play demo."),
                        isEnabled: isDemoEnabled)
}
```

Here, the play button only reports to the accessibility engine is `isDemoEnabled` is true – making it a little easier to work with situations like this one when implementing accessibility labels.

Three Key Takeaways

1. SwiftUI and UIKit have many similarities when it comes to accessibility APIs.
2. Both can make your apps accessible by default, but there is nearly always more work to be done to get you the rest of the way there.
3. Be aware of SwiftUI-specific techniques and ways of thinking in terms of accessibility programming.

Accessibility Settings

Apple's accessibility APIs have typically done a standup job of providing developers with *context*. What a screen reader is doing, how a user interacts with custom components and whether specific accessibility features are active.

However, it wasn't until recently that there's been a unified place to check for global accessibility preferences in your app. That's (sort of) what the nascent `AccessibilitySettings` API delivers. Though it's not a replacement for other accessibility APIs, Apple does seem to be bolstering it more and more with each release.

Introduced in iOS 17 (but also backported), `AccessibilitySettings` gives you a simple, declarative way to read important system-level accessibility preferences. Among them:

- Whether users have disabled animated images.
- Prefers horizontal text layout, or wants non-blinking cursors.

- If Assistive Access is currently running.
- And, a single entry point to deep link to these types of settings.

While there are only a few checks represented here, we can respect those user choices with a single property check. In this chapter, we'll look at each of the settings currently available and what kind of improvements we can offer by responding to them.

Animated GIF Playback

Apple added a new API in iOS 18 called `AccessibilitySettings`. It offers read-only values that reflect how users have customized their device's accessibility behavior. These preferences live in the Settings app under **Accessibility → Motion → Animated Images**, and when turned off – we should have our own apps reflect that preference.

In our example, we use the `animatedImagesEnabled` property to check whether animated image playback is allowed. If it is, we animate the `.gif` as usual using `CGAnimateImageAtURLWithBlock`. If it isn't, we grab just the first frame and display that instead:

```
if AccessibilitySettings.animatedImagesEnabled {
    CGAnimateImageAtURLWithBlock(...)
} else {
    image = firstFrame(from: gifURL)
}
```

This approach respects the user's system-level preference, ensures a better experience for motion-sensitive users. Check out the full example here:

```
// Xcode -> AccessibilitySettingsFig1View.swift

import SwiftUI
```

```
import UIKit
import ImageIO

struct AccessibilitySettingsFig1View: View {
    var body: some View {
        AnimateGifView()
    }
}

struct AnimateGifView: View {
    @State private var image: UIImage? = nil

    var body: some View {
        ZStack {
            if let img = image {
                Image(uiImage: img)
            } else {
                Image(systemName: "photo.circle.fill")
            }
        }
        .onAppear {
            loadGifAsset()
        }
    }

    // MARK: Private Functions

    private func loadGifAsset() {
        guard let filePath =
Bundle.main.path(forResource:"exampleGifAsset", ofType: "gif") else
{
            print("Couldn't locate local .gif asset.")
            return
        }

        let gifURL = URL(filePath: filePath)

        if AccessibilitySettings.animatedImagesEnabled {
```

```
            CGAnimateImageAtURLWithBlock(gifURL as CFURL, nil)
{ index, image, stop in
                self.image = UIImage(cgImage: image)
            }
        } else {
            self.image = firstFrame(from: gifURL)
        }
    }

    private func firstFrame(from url: URL) -> UIImage? {
        guard let source = CGImageSourceCreateWithURL(url as CFURL,
nil),
              let frame = CGImageSourceCreateImageAtIndex(source, 0,
nil) else {
            print("Unable to extract first frame of GIF.")
            return nil
        }
        return UIImage(cgImage: frame)
    }
}
```

Deep Linking to Accessibility Settings

There's a recurring pain point in accessibility testing for me: I'm experimenting with a feature, and then I need to flip a system-level toggle buried deep in the Settings app. On top of that, we all know reliable Setting's search is…not. Thankfully, `AccessibilitySettings` finally provides some relief here, eschewing the need to find a toggle tucked four menus deep.

That's the good news. The bad news? It only works for *two* settings currently. If you look at `AccessibilitySettings.Feature`, you'll find a couple values in the Enumeration:

```
personalVoiceAllowAppsToRequestToUse
```

```
allowAppsToAddAudioToCalls
```

And of those, the latter is only available on iOS 18.2 and up. Regardless, it's a start. Opening these settings is done via an asynchronous function (which throws) and it uses an enum to represent its feature to open, but it isn't hardcoded to a single destination. To me, that all signals future expansion:

```
try await AccessibilitySettings.openSettings(for: feature)
```

For now, this API is useful in apps that leverage synthesized speech with a user's custom voice or uses the audio calling APIs. But, the broader value is obvious: fewer steps for users, quicker iterations for developers, and a more seamless accessibility experience overall:

```
// Xcode -> AccessibilitySettingsFig2View.swift

import SwiftUI

extension AccessibilitySettings.Feature: @retroactive CaseIterable,
                                         @retroactive
CustomStringConvertible,
                                         @retroactive Identifiable {

    public var id: Self { self }

    public static var allCases: [AccessibilitySettings.Feature] {
        if #available(iOS 18.2, *) {

[.personalVoiceAllowAppsToRequestToUse, .allowAppsToAddAudioToCalls]
        } else {
            [.personalVoiceAllowAppsToRequestToUse]
        }
    }
```

```
    public var description: String {
        if #available(iOS 18.2, *) {
            switch self {
            case .personalVoiceAllowAppsToRequestToUse:
                return "Personal Voice Settings"
            case .allowAppsToAddAudioToCalls:
                return "Allow Apps to use Audio to Call"
            @unknown default:
                fatalError()
            }
        } else {
            switch self {
            case .personalVoiceAllowAppsToRequestToUse:
                return "Personal Voice Settings"
            default:
                fatalError()
            }
        }
    }
}

struct AccessibilitySettingsFig2View: View {

    var body: some View {
        Form {
            Section("Quick Links") {
                ForEach(AccessibilitySettings.Feature.allCases)
{ axCase in
                    Button("Open \(axCase.description)") {
                        Task {
                            await openAXSetting(for: axCase)
                        }
                    }
                }
            }
        }
    }
```

```
    private func openAXSetting(for feature:
AccessibilitySettings.Feature) async {
        do {
            try await AccessibilitySettings.openSettings(for:
feature)
        } catch {
            print("Unable to open setting: \
(error.localizedDescription)")
        }
    }
}

#Preview {
    AccessibilitySettingsFig2View()
}
```

Text Preferences

There are custom text preferences which affect accessibility that are easily queried with `AccessibilitySettings`, let's check those out:

1. `prefersHorizontalTextLayout`: A Boolean which represents a user's preference for horizontal text in languages that support *both* vertical and horizontal text layouts.
2. `prefersNonBlinkingTextInsertionIndicator`: In editable text fields, there's a blinking cursor where the current text insertion point is. This lets users pop then on or off.
3. `prefersHeadAnchorAlternative`: This allow content to follow user's head position – but it's for visionOS.

Using these works like every other accessibility setting we've covered. Simply check the property directly, or subscribe for notification changes as well:

```
// Xcode -> AccessibilitySettingsFig3View.swift

struct AccessibilitySettingsFig3View: View {
    var body: some View {
        Form {
            HStack {
                Text("Text Layout Preference:")
                    .fontWeight(.semibold)

Text(AccessibilitySettings.prefersHorizontalTextLayout ?
"Horizontal" : "Vertical")
                    .foregroundColor(.blue)
            }
            HStack {
                Text("Non-Blinking Cursor:")
                    .fontWeight(.semibold)

Text(AccessibilitySettings.prefersNonBlinkingTextInsertionIndicator
? "On" : "Off")
                    .foregroundColor(.purple)
            }
        }
    }
}
```

Assistive Access

Assistive Access is a relatively new accessibility mode, introduced in iOS 17, but it's quietly one of the most thoughtful features Apple has shipped in recent years. Its goal? Reduce cognitive load by streamlining iOS down to the *bare* essentials.

So, that means instead of the personalized design of the apps we know and love today, we get bigger text, slower transitions, clear call-to-action buttons, and fewer distractions . It completely changes iOS wholesale, all so that users who might otherwise find iOS overwhelming can use it without any distractions. Whether that's some-

one with a cognitive disability, an elderly parent, or even your own child, Assistive Access makes iOS more approachable.

And, that's why you should realize as a developer, it's not necessarily meant for you to reintroduce your app in an entirely different way. In fact, it wasn't until a year after the mode was formally introduced that developers could even query for Assistive Access being enabled at all.

The best way to get a feel for this to turn in on yourself. Visiting the Accessibility settings in iOS to do so, there's just a few configuration steps you've got to setup first.

When it's running, the layout changes are immediately noticeable. Apps in Assistive Access get less space to draw their UI (due to persistent navigation controls – they never are more than a touch away), so it's worth testing your layouts for adaptability. If needed, you can opt out of that constraint by setting the `UISupportsFullScreenInAssisstiveAccess` key in your Info.plist. Apple's own apps do this, but it's not the default – and for good reason. You'll need to ensure your layouts don't hide navigation or interactive elements if you use that key.

As I mentioned, for over a year, there was no way to detect if Assistive Access was active – unlike other accessibility features like VoiceOver or Bold Text. That changed in iOS 18. Now, there are two ways to check:

```
// UIKit
AccessibilitySettings.isAssistiveAccessEnabled

// SwiftUI:
@Environment(\.accessibilityAssistiveAccessEnabled)
```

This makes a huge difference. It opens the door to custom Assistive Access experiences from third party developers. Now, we can offer our own streamlined interfaces or simplified modes of ours apps when this Assistive Access is on. Apple's own apps

(like Messages or Camera) already do this, it's worth checking them out to see how drastically they are simplified. I think we should try to follow suit.

Live Checks

Back in the accessibility notifications chapter, we covered an idiomatic way to react to preference changes on the fly. Here's an abbreviated version:

```
struct AXPublisher {

    private typealias NC = NotificationCenter

    let switchControlStatusChanged = NC.default.publisher(for:
UIAccessibility.switchControlStatusDidChangeNotification)
}
```

All of these properties covered in this chapter have notifications you can query for, too. So, that means you can follow the same pattern. In SwiftUI, it's even easier:

```
// Xcode -> AccessibilitySettingsFig4View.swift

struct AccessibilitySettingsFig4View: View {
    @State private var prefersGifs: Bool = false

    var body: some View {
        Form {
            Text("Prefers GIF: \(prefersGifs)")
                .onChange(of:
AccessibilitySettings.animatedImagesEnabledDidChangeNotification) {
                    self.prefersGifs =
AccessibilitySettings.animatedImagesEnabled
                }
        }
```

```
        }
    }
```

Three Key Takeaways

1. The `AccessibilitySettings` struct gives you an entry point to new, and forthcoming, accessibility settings.
2. Query its properties to know when to enable or disable certain experiences, such as .gif playback.
3. Leverage the struct to deep link straight into accessibility settings, as more features will surely be supported in the future.

Assistive Access

Three Key Takeaways

1. SwiftUI and UIKit have many similarities when it comes to accessibility APIs.
2. Both can make your apps accessible by default, but there is nearly always more work to be done to get you the rest of the way there.
3. Be aware of SwiftUI-specific techniques and ways of thinking in terms of accessibility programming.

Accessibility Inspector

Aside from Xcode, Apple ships several important programs to help iOS developers do their jobs and do them well. Chief among them is the Accessibility Inspector, a stand-alone application that helps developers identify and debug common accessibility issues in both UIKit and SwiftUI apps.

Using it, you can test several mission-critical accessibility inspection points, such as VoiceOver. It also has features to test visual components for things like color contrast thresholds. You can open it several different ways:

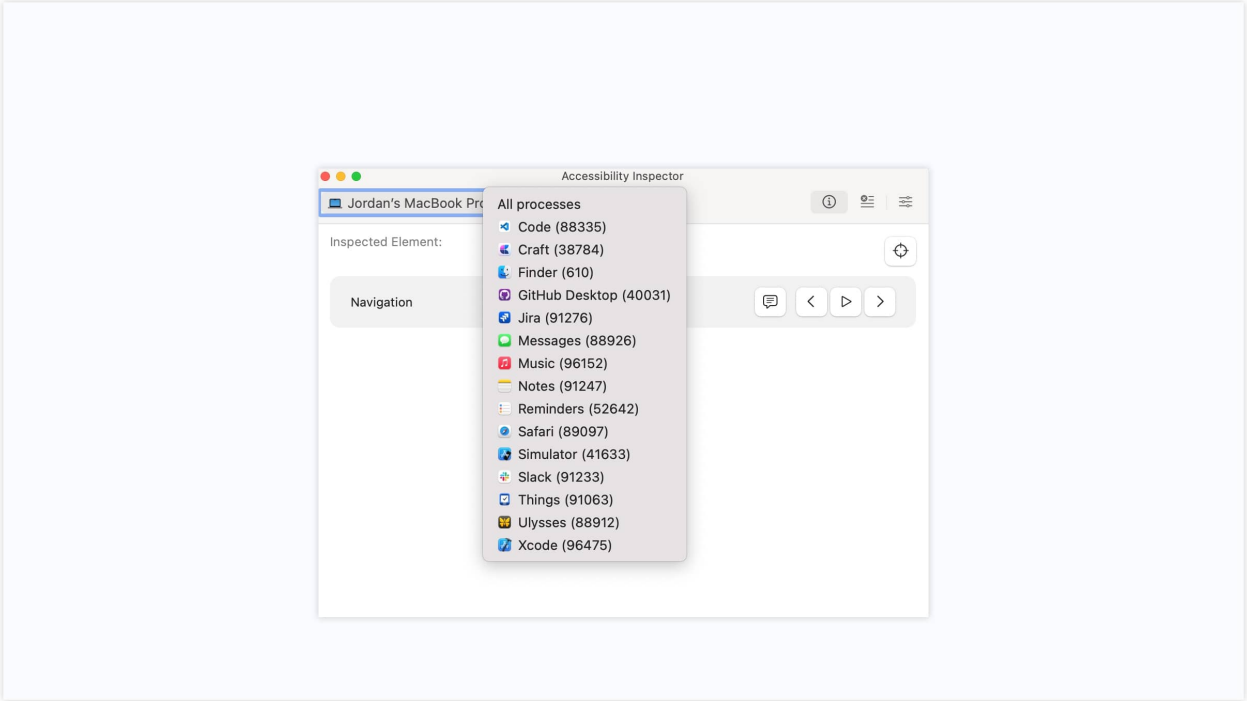
1. If Xcode is open, in the menu bar go to Xcode -> Open Developer Tool -> Accessibility Inspector.
2. From Finder, you can search for "Accessibility Inspector" to find the app icon quickly. Otherwise, you'd need to go to the Xcode app and open its package contents to find it.

3. Or, from Terminal, simply execute the following:

```
$ open -a 'Accessibility Inspector'
```

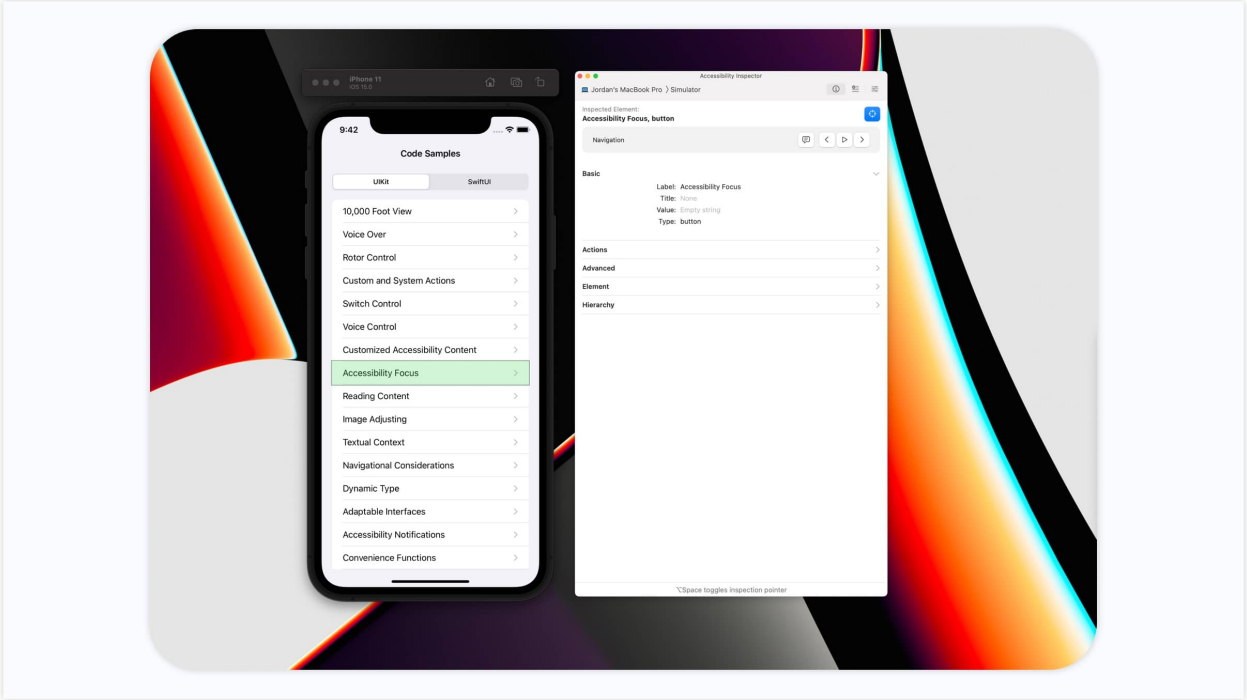
If you're coming to this chapter after reading the last one, you might be wondering what the difference is between this application and the Accessibility Inspector in Xcode's Inspector panel. First, that inspector was strictly built for SwiftUI – so it has no capabilities to diagnose accessibility issues in UIKit. Secondly, while wildly useful, the Accessibility Inspector app boasts a bit more in terms of capabilities. You'll likely find yourself using both of them often, even if you're primarily working with SwiftUI based interfaces.

When the Accessibility Inspector first runs, you'll find two things rather quickly. One, you need to select a process to debug or audit:



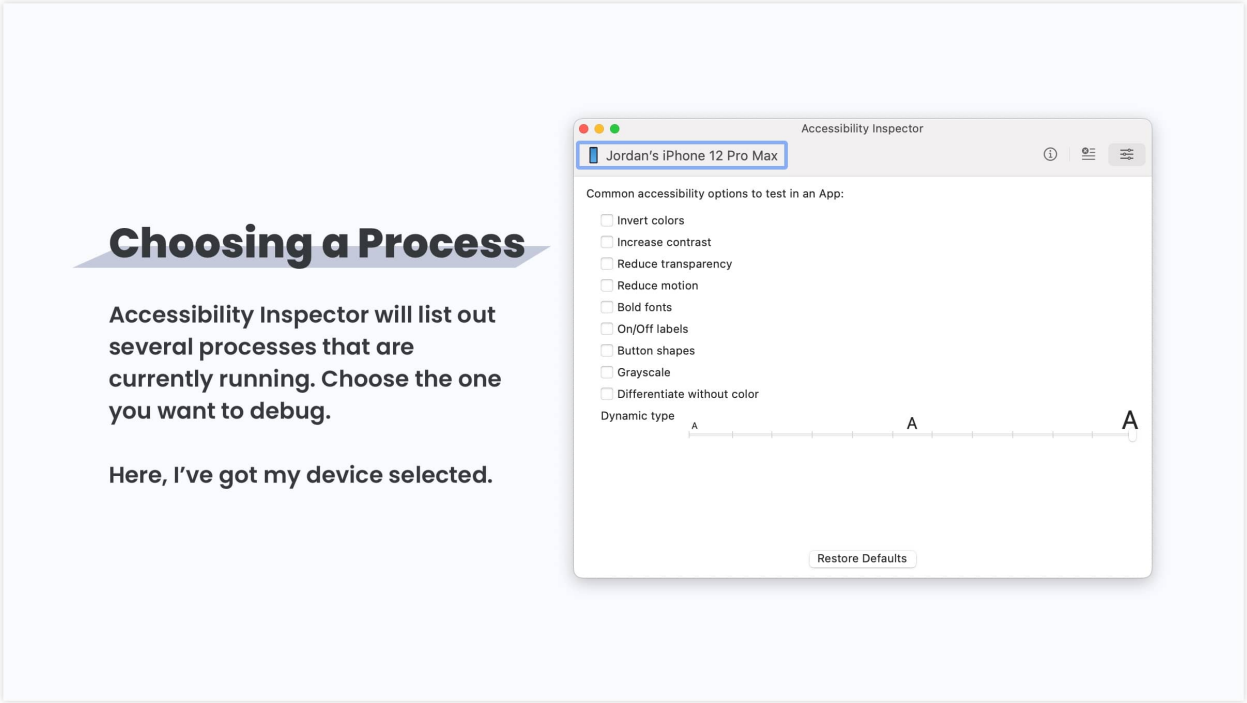
Secondly, its window will always be on top of anything else in macOS. This may take a bit of getting used to at first, but I think you'll find it makes things a little easier to navi-

gate. When the Accessibility Inspector is open, you're likely wanting it front and center alongside the simulator:



How it Works

Let's tour the demo project to get a feel for how to use Accessibility Inspector. First, choose the simulator process at the top with demo project running (or your device if you're not using the simulator):



Now, let's cover its three main modes of use:

1. **Inspection:** The primary view. Here, you can control navigation, activate elements and preview what VoiceOver would read aloud.
2. **Audit:** This lets you "scan" the current process being debugged to automatically uncover potential accessibility issues, such as elements missing values for some of the "Big Three" accessibility properties.
3. **Settings:** This lets you activate several helpful accessibility settings, such as Reduce Motion or Invert Colors.

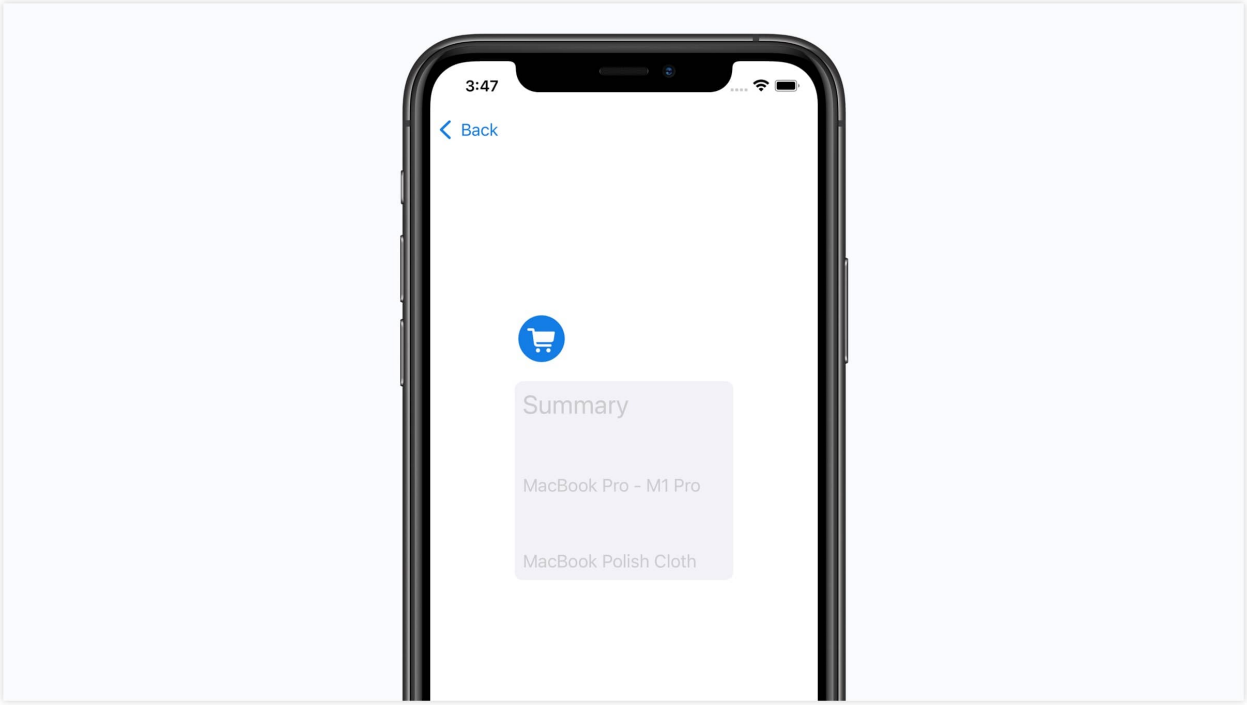
You can switch between these modes using the buttons in the top right of Accessibility Inspector:



Unlike previous chapters in this book, this one follows more of a tutorial-style format. I feel like the best way to get a feel for all Accessibility Inspector can do, and how it will fit into your workflow, is to use it hands on.

If you open up the sample code for this chapter, you can follow along with me as I use Accessibility Inspector to fix up a bunch of issues. Either the SwiftUI (`AXInspectorFig1View.swift`) or the UIKit (`AXInspectorFig1ViewController.swift`) variant will work. Functionally, they are the almost exactly same.

Here's what they look like:



This mock ordering interface has a few labels and an icon that acts as a button. When tapped, it places the order. There are several accessibility issues throughout this interface, and we'll use the Accessibility Inspector to uncover each of them.

To do so we'll go through every primary aspect of Accessibility Inspector, starting with the Inspection view. One note before we kick things off – it's best to follow along here using an actual device if you can. Since the Accessibility Inspector will audit an *entire process*, it'll pick up things that we may not care about such as elements in the iOS simulator application itself.

Inspection

The Inspection view is where you will likely spend the majority of your time in Accessibility Inspector. With our example open and running (Figure 1 of either the SwiftUI or UIKit view), we could get started in one of two ways:

1. Click the target icon in the top right to cherry-pick which element we'd like to investigate first.
2. Click the play icon to start auto navigating the interface to audit the VoiceOver experience.

Let's start by clicking the auto navigate button and see what happens. Accessibility Inspector will read out everything VoiceOver would pick up, and navigate the items in the order they are reported to the accessibility engine.

After auto navigation begins, the first issue becomes obvious. Our add to cart button isn't seen by the accessibility engine at all – it's skipped right over it. Looking at the code, we can see why:

```
// Xcode -> UIKit -> AXInspectorFig1ViewController.swift

// A mock button, that's icon based.
let buyIcon = UIImage(systemName: "cart.circle.fill")
let buyIconImageView = UIImageView(image: buyIcon)
buyIconImageView.translatesAutoresizingMaskIntoConstraints = false

// Simulate a buy action
let tapGesture = UITapGestureRecognizer(target: self, action:
#selector(onBuyButtonTapped))
buyIconImageView.isUserInteractionEnabled = true
buyIconImageView.addGestureRecognizer(tapGesture)

// Xcode -> SwiftUI -> AXInspectorFig2View.swift
Image(systemName:"cart.circle.fill")
    .resizable()
    .frame(width: 54, height: 54)
    .foregroundColor(.blue)
    .onTapGesture {
        showConfirmation.toggle()
    }
    .alert(String("Added to Cart"),
```

```
isPresented: $showConfirmation) {
        Button("Got It", role: .cancel) { }
    } message: {
        Text("We've added way too many
Apple products to your cart.
Please enjoy them, because
they were finely crafted and
designed in California at
least.")
    }
```

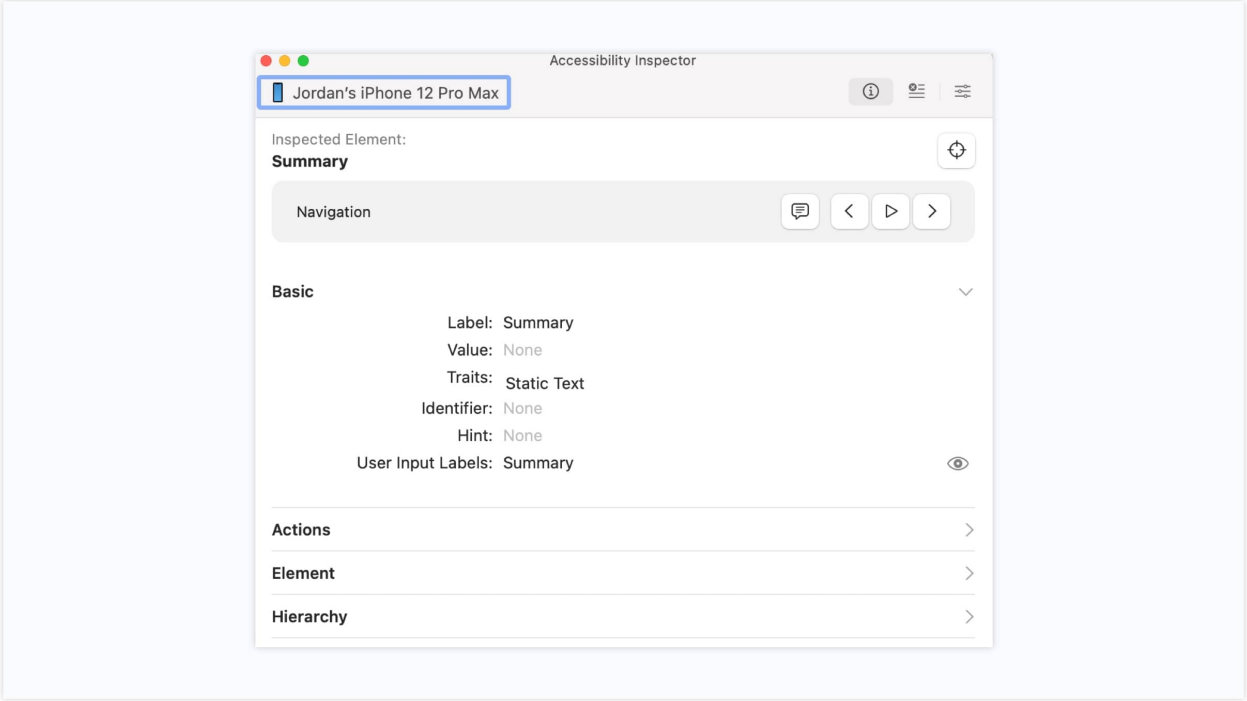
To fix this, we'll add the appropriate values so that VoiceOver picks up the "button" correctly. Going forward, all of the fixes will be found in the Figure 2 versions of the views. Here's the first one:

```
// Xcode -> UIKit -> AXInspectorFig2ViewController.swift

// Fix 1
buyIconImageView.accessibilityTraits = .button
buyIconImageView.isAccessibilityElement = true
buyIconImageView.accessibilityLabel = "Add to Cart"

// Xcode -> SwiftUI -> AXInspectorFig2View.swift
Image(systemName:"cart.circle.fill")
    .resizable()
    .frame(width: 54, height: 54)
    .foregroundColor(.blue)
    // Fix 1
    .accessibilityAddTraits(.isButton)
    .accessibilityLabel("Add to Cart")
    .onTapGesture {
        showConfirmation.toggle()
    }
    .alert(String("Added to Cart"),
isPresented: $showConfirmation) {
        Button("Got It", role: .cancel) { }
    } message: {
        Text("We've added way too many
```

```
Apple products to your cart.
Please enjoy them, because
they were finely crafted and
designed in California at
least.")
    }
```

Now, if we run it again – Accessibility Inspector correctly reads out the "Add to Cart" accessibility label just as we intended.

Next, run auto navigate again – but this time pause it when it reaches the "Summary" label. Here's what we see:



In this case, imagine we could have several of these "Summary" views if the user had multiple orders. In those cases, it would make more sense to add the `.header` trait for easier navigation. Not only does this help with semantic considerations, it also opens the element up for the header rotor too. We'll do that now:

```
// Xcode -> UIKit -> AXInspectorFig2ViewController.swift

// Fix 2
summaryLabel.accessibilityTraits = .header

// Xcode -> SwiftUI -> AXInspectorFig2View.swift
Text("Summary")
    .font(.system(size: 28))
    // Fix 2
    .accessibilityAddTraits(.isHeader)
```

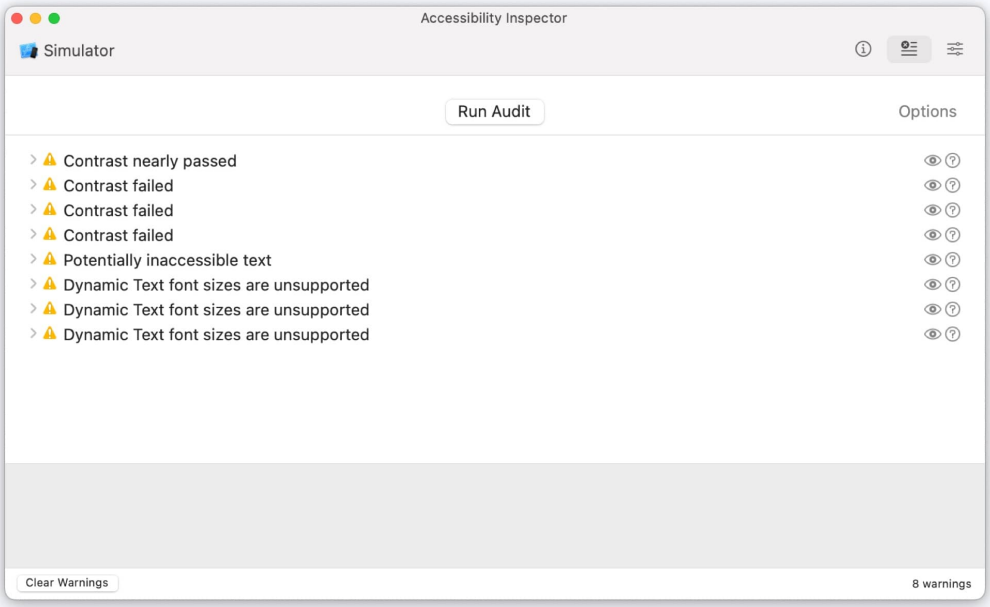
Running it back, we can confirm that the header is there under the "Basic" area listing out all of the common accessibility values. The inspection tab is indispensable when it comes to quickly seeing the "Big Three" accessibility values like this, testing VoiceOver at a glance and more.

Audit

Let's turn to the audit tab next. The audit tab will scan the interface and give us actionable insight on potential issues. With the Figure 1 view open, click "Run Audit" to see what we get.

> *Please note that with SwiftUI, the audit feature can produce some unexpected results. I'm not entirely sure why this is, but I've filed feedback with Apple about it. Remember that SwiftUI also has its own, dedicated Accessibility Inspector in Xcode's inspector pane. If you're seeing something you don't expect, be sure to check there and as always — run it on device too.*

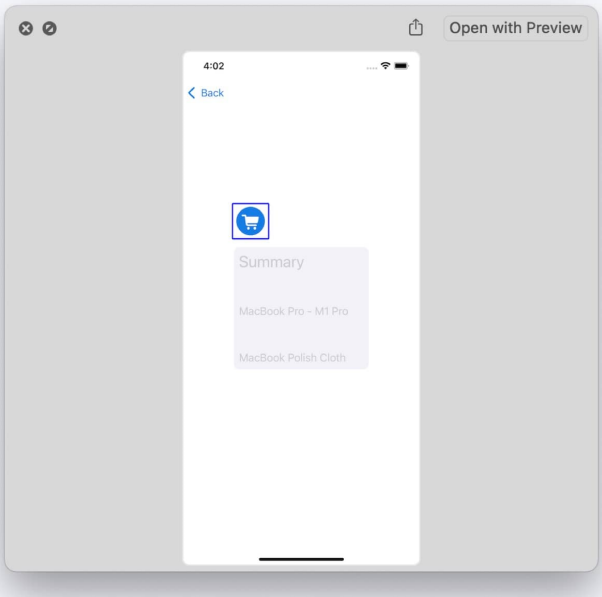
With one click, Accessibility Inspector picked up several issues:



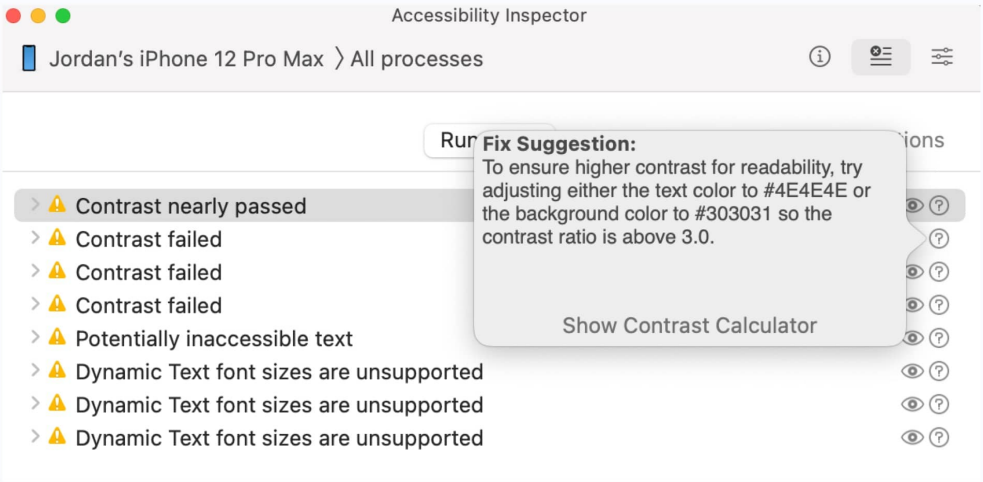
What's great about audit isn't *just* that it picks up issues, it tells you two more important things:

1. Where the issues are.
2. And suggestions to fix those issues.

You can get this information by clicking either the eyeball icon to show *where* the issue is within the interface (pay attention to the blue box around the shopping cart button):



While the question mark icon will give you hints about how you might be able to fix it:



Some of the warnings can be false positives, but those are usually easy to spot since they typically begin with the word "Potentially (issue)". Still, it pays to read each warning to make a game plan on what to fix.

Here, the issues are glaringly obvious – we've got some terrible contrast issues with the text. Plus, our text doesn't respect Dynamic Type options, which is another point of failure in terms of accessibility programming. The accessibility inspector has let us know about all of these.

Let's apply some fixes and run the audit again:

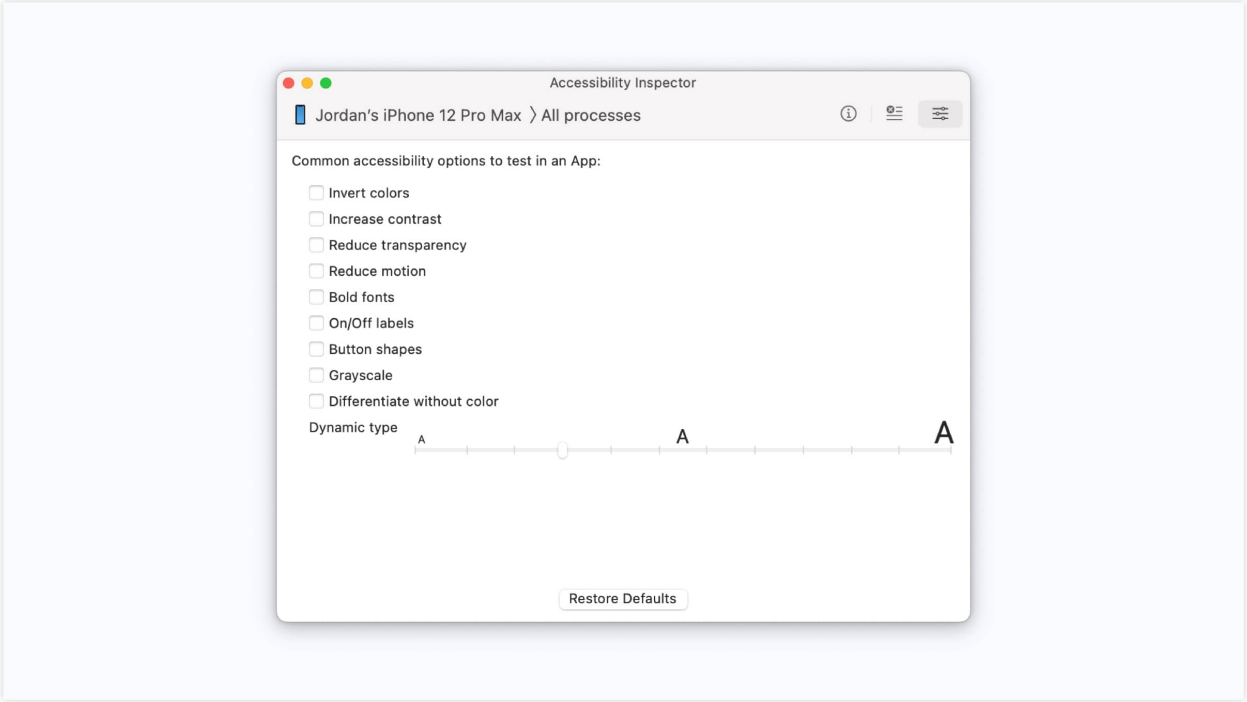
```
// Xcode -> UIKit -> AXInspectorFig3ViewController.swift

// Fix 3
detailsContainerStackView.arrangedSubviews.forEach {
    if let label = $0 as? UILabel {
        label.textColor = .label
        label.adjustsFontForContentSizeCategory = true
    }
}

// Xcode -> SwiftUI -> AXInspectorFig4ViewController.swift
// Fix 3
private let textColor: Color = Color(uiColor:
.label)

// Later, in body property...
Text("Summary")
    // Fix 3
    .font(.title)
    // Fix 2
    .accessibilityAddTraits(.isHeader)
Text("MacBook Pro - M1 Pro")
    // Fix 3
    .font(.title3)
Text("MacBook Polish Cloth")
    // Fix 3
```

```
    .font(.title3)
```

And just like that, we now have acceptable color contrast ranges and text controls that respect Dynamic Type – which is a baseline you should always strive for. You can also run the audit again to confirm the fixes.

Settings

Finally, let's visit the settings view. While the name may sound like it controls actual settings within Accessibility Inspector itself – it doesn't. Instead, you can easily toggle several accessibility APIs with a click:



The implications here are obvious. If you've got any blur views, animations, custom type faces or your own text stack – you'll want to run through those views using these

options. We've discussed how to address many of these concerns in several chapters. Here are a few to visit:

1. Adaptable Interfaces
2. Posting and Consuming Accessibility Notifications
3. User Preferences in SwiftUI

To see what we can do with our view, though, let's test out Dynamic Type. Even though we fixed the issue of our text controls not responding to Dynamic Type in the last section, we still need to ensure that the interface adapts accordingly.

Open up the UIKit Figure 1 view[18] with Accessibility Inspector running, and crank up the Dynamic Type setting using the slider at the bottom of the settings view. Once we do, we can see an issue – our text clips and isn't readable:

[18] By nature of SwiftUI's layout system, the following issue won't be an issue there.



Remember, Dynamic Type is critical to many users and they shouldn't be taken out of your app's experience because you forgot to support large text sizes. In our case, we can fix it up by providing a greater width for the container and not limiting the number of lines in the text controls themselves:

```
// Xcode -> UIKit ->

// Fix 4 (abbreviated)
detailsContainerStackView.widthAnchor.constraint(equalTo:
view.readableContentGuide.widthAnchor),
detailsContainerStackView.heightAnchor.constraint(greaterThanOrEqual
ToConstant: 100)

// Fix 4
label.numberOfLines = 0
```

And with that, our interface passes the test:



Accessibility Inspector is a vital tool to fix common issues that shouldn't ship. But we are human, and bugs will inevitably make it out the door – but I truly believe the more you integrate Accessibility Inspector into your workflow, the more accessible your apps will become.

Even if you're in an established codebase or brand new one, run through your critical paths in your app and see how they fare in Accessibility Inspector. Does your sign up flow look good? Any icons that act as buttons but aren't marked as such? How about Dynamic Type?

Those questions are not tedious to ask, address and uncover using Accessibility Inspector – so use it to your advantage.

Tips

Test Other Processes

When Accessibility Inspector is running on your device, you can inspect *any* app that you've got installed. This is extremely useful to gain reference points, or see how Apple handles accessibility situations that you might be programing for in your own app.

For example, if you wanted to see how Mail handles Dynamic Type at its lowest and highest settings, you could do that using the settings panel. Simply open the Mail app and then set the Dynamic Type setting however you like using Accessibility Inspector.

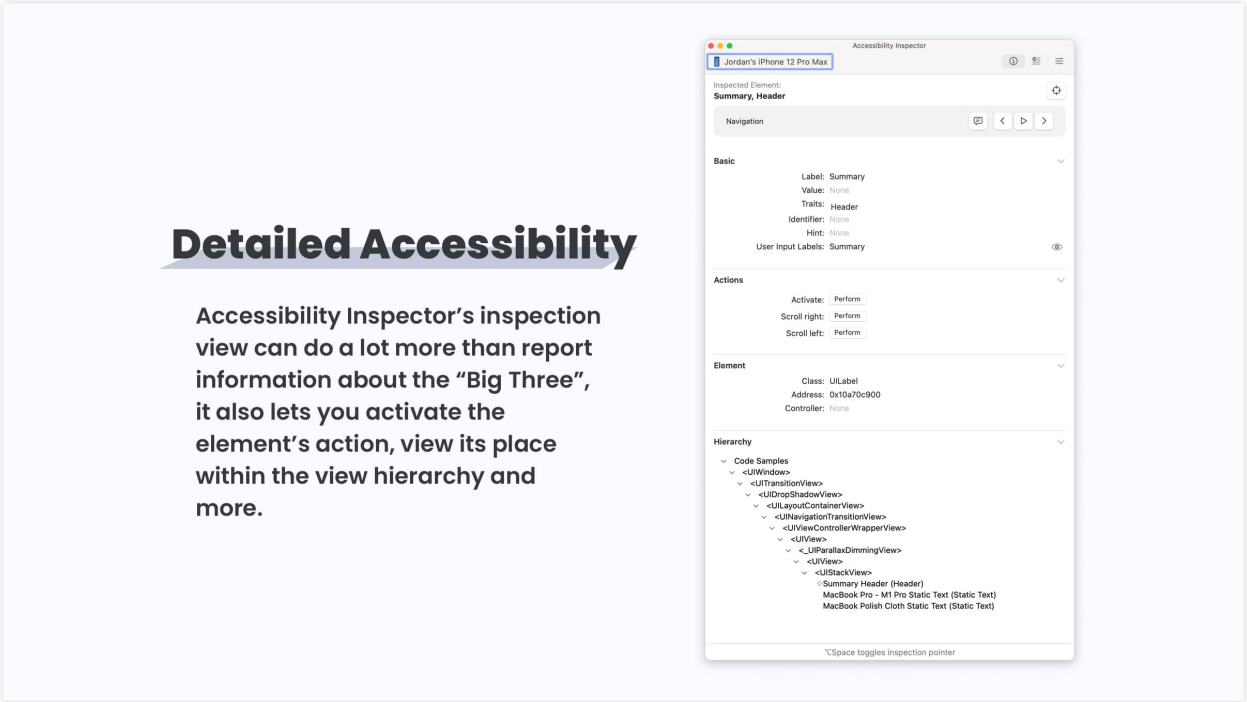
So if you find yourself needing inspiration, remember that Accessibility Inspector, and all of its tooling, works beyond just your own software. When I find myself stuck or wondering how other developers handle similar situations I'm facing, my first stop is Accessibility Inspector. I poke around other apps that handle accessibility well to get some ideas.

Activate Actions and More using the Inspection Tab

You can performs actions, such as tapping a button or some accessibility action, using the inspection view. Beyond just showing what the basic accessibility values are, it also lists the following:

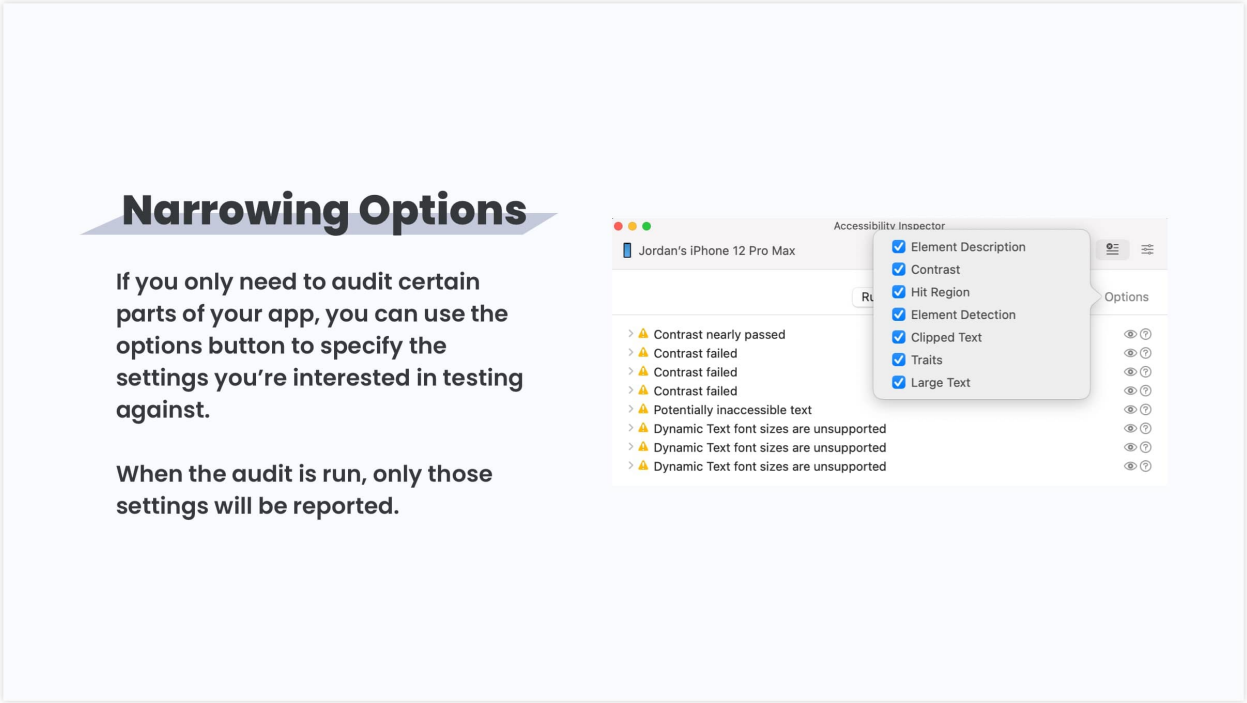
1. **Actions:** Things the element can perform or do when interacted with.
2. **Elements:** Its address in memory, class name and more.
3. **Hierarchy:** This lists out the *entire* view tree for the element.

In many cases, this information is vital in debugging issues with accessibility programming.



Narrow Options with Audit

The audit view has a helpful options tab:

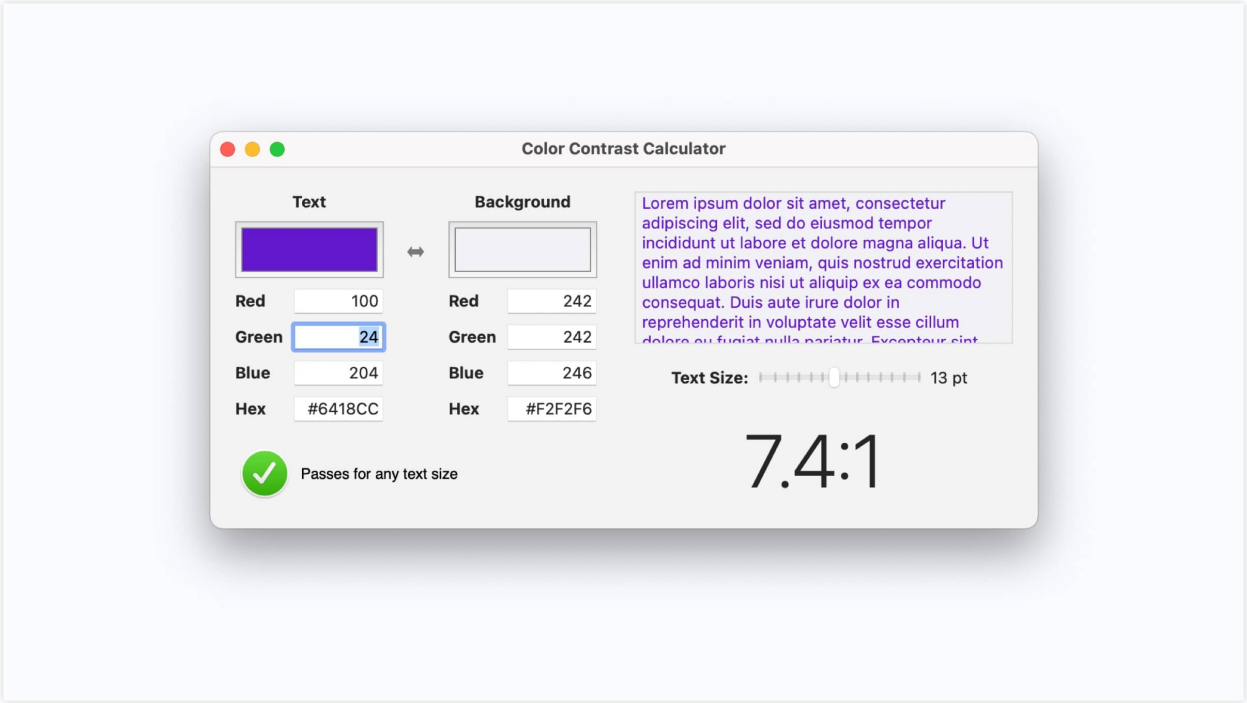


This lets you narrow down what it audits for, so if you're getting a lot of noise and only want to test, say, Dynamic Type — you can just select that and it will be the only thing that's audited.

Tackle Contrast with Contrast Calculator

While I recommend you should stick to the system color palette as much as possible, nearly every app has some permutation of custom colors. As well they should, too, as they express creativity, branding and more. In those cases, though, be sure you've got colors that pass accessibility contrast ranges.

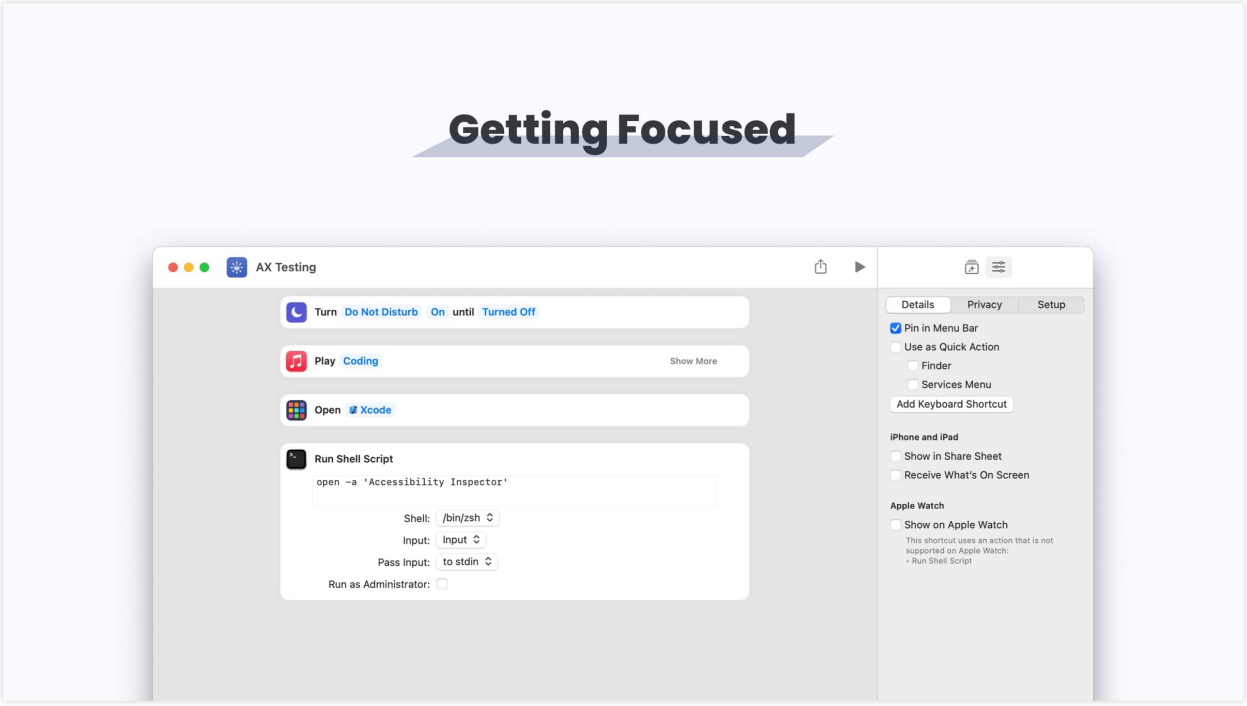
If you don't pass the recommended color contrast threshold, Accessibility Inspector has a Contrast Calculator you can use to get something that will work:



Even better, you can change color values and get immediate feedback on whether or not the new color would be within the acceptable color contrast range.

Make It Routine

When I specifically sit down to work on accessibility programming, I find a simple Siri Shortcut helps me get in the zone. If you're running Monterey or later, you can create something similar to the Shortcut in the following picture for the same purpose:



What this really does, though, is create routine for me. When I run this, I *know* what I'm specifically setting down to do — accessibility work. It's a mindset, and while it sounds silly, a little pageantry around your routines has been proven time and time again that it can help you get things done more efficiently.

So, whether that's a pomodoro timer, a particular place you go to only for accessibility work — whatever it is, think about doing it as much as you can to kick off a productive programming session to make your app accessible to everyone.

Three Key Takeaways

1. Accessibility Inspector plays a critical role in accessibility work.
2. It can help you uncover nearly every accessibility issue on the spot, but it's on you to audit your apps and integrate it into your workflow.
3. If you get stuck on a challenging accessibility issue, use Accessibility Inspector against other apps to see how they handle the particular scenario you're facing.

Accessibility Shortcuts

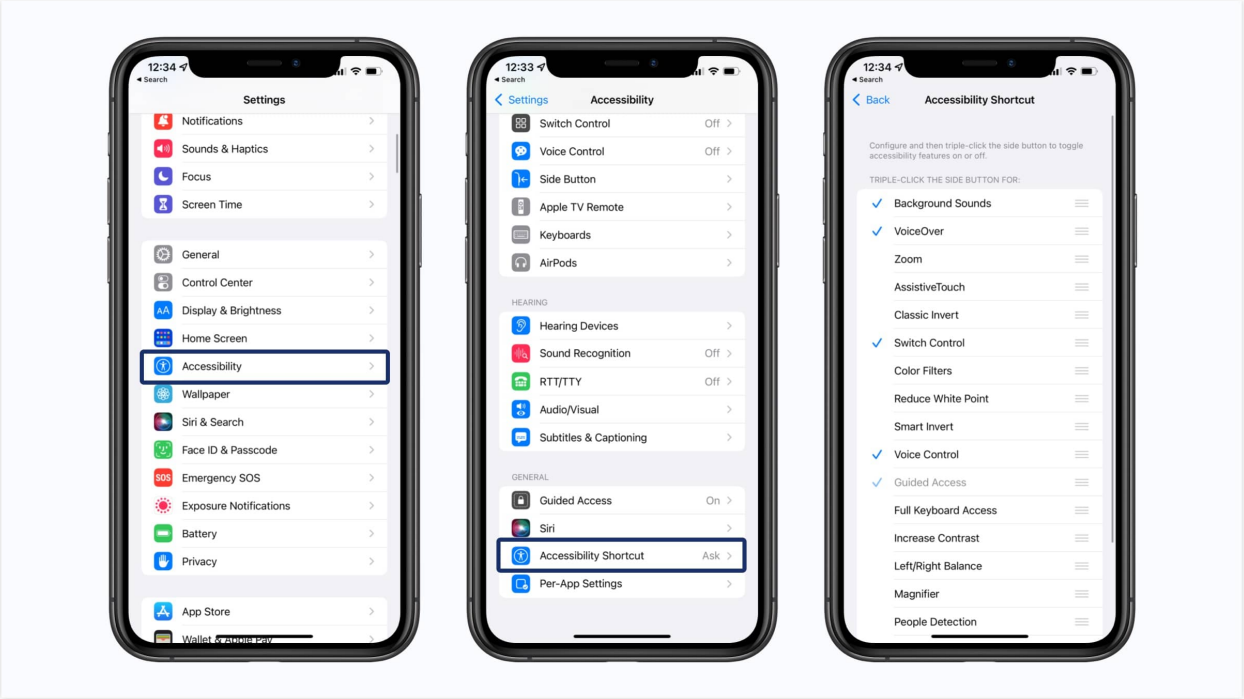
On your journey towards becoming passingly familiar with accessibility APIs and needs, to becoming right at home with them — there will be one constant along the way. And that is the need to be testing your app for accessibility purposes, and doing it a lot.

Luckily, this doesn't need to involve feature flips, hard coding certain paths or frequent trips to the Settings app to toggle them on. Instead, iOS has several ways to activate accessibility technologies built right into the operating system. This not only makes it convenient for users who rely on these technologies to quickly access them, but it also is useful for development purposes too.

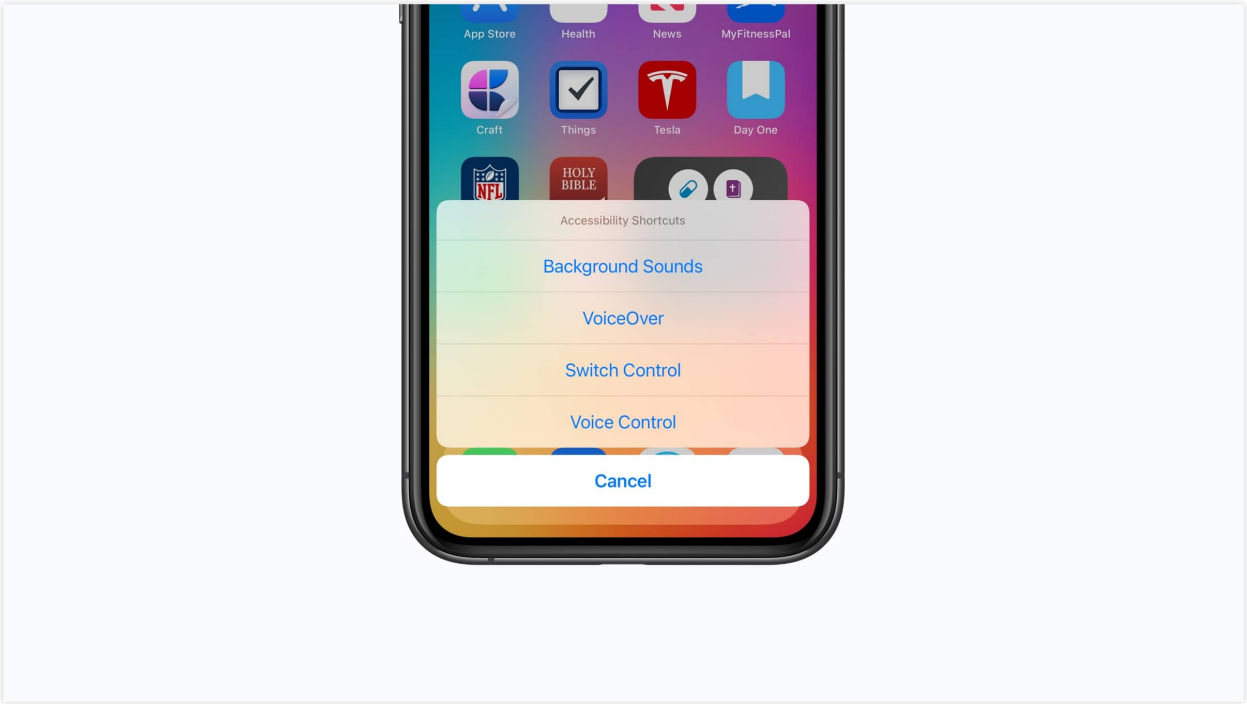
Let's take a look at a few ways you can activate accessibility technologies on the fly.

Custom Accessibility Shortcuts

You can have an array of accessibility options disabled or enabled by activating them in Settings. Specifically, to set them up go to the Settings app -> Accessibility -> Accessibility:



If you've got a device with a home button, you can triple press it present them. Otherwise, it's a triple press of the sleep wake button. Here's what it looks like when you bring them up:



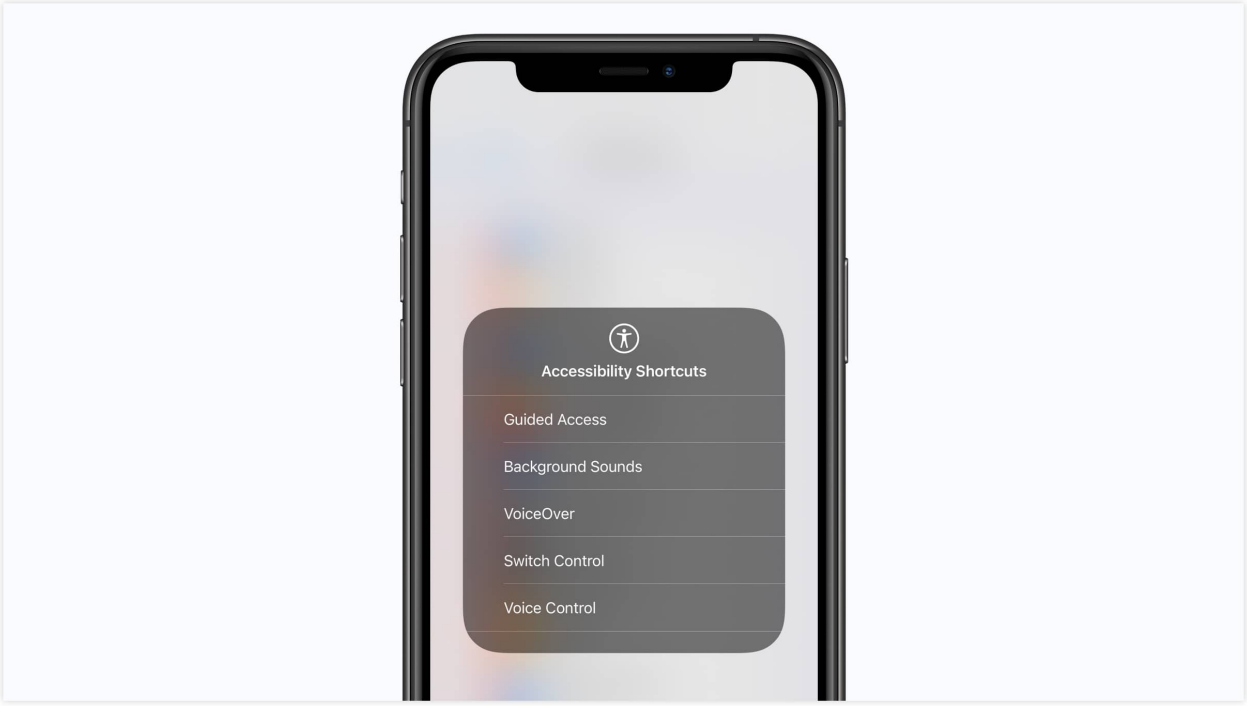
Use these to quickly change Dynamic Type settings, enabled VoiceOver and more. This is my most used accessibility technique for testing – you simply can't go without it.

Control Center Buttons

You can also access the accessibility shortcuts you've setup from the previous section using Control Center. To turn this one on, once again you'll go to the Settings app -> Control Center and then enable them there:



You can also use a button specifically for Dynamic Type. You'll find that under "Text Size". Remember, to use these you'll long press on the icon in Control Center to view the options:



Screen Curtain

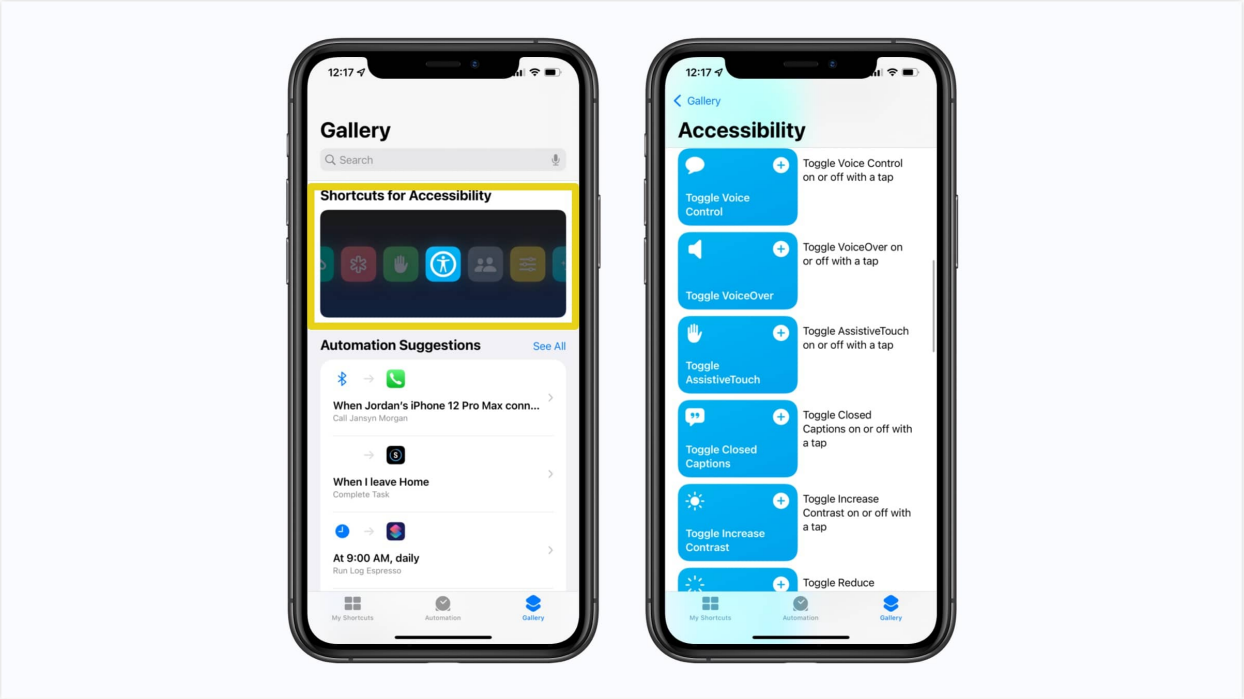
Throughout the book we've covered a lot of ways to test VoiceOver. But there is a "best way", I'd argue. And that is Screen Curtain. This completely blacks out your screen and puts you on a level playing field with other blind users.

When VoiceOver is running, three finger triple tap anywhere on your screen to toggle it on and off. You're likely to uncover issues fast in your own apps when you can no longer fallback to visual cues to get things done.

There is simply no better way to make an argument for better accessibility support in an app than to show someone else on your team this feature, and see how much (or little) they can get done without their eyesight to help them.

Siri Shortcuts

Siri Shortcuts has several useful accessibility shortcuts available in the Gallery view:



If you'd rather activate any of the accessibility technologies with Siri over any of the methods above, this is for you. For example, you could simply say "Hey Siri - toggle increase contrast", saving you a trip to Control Center, the accessibility shortcuts menu or any other means to activate them.

I also find that making some Shortcuts that aggregate several accessibility APIs together helps me test things out a little faster. For example, I have one called "Test Type" which turns on a large Dynamic Type setting with bold text enabled. If you've got flows that you often test with a combination of Apple's accessibility technologies – a simple Shortcut can definitely save you some time.

Three Key Takeaways

1. There is simply no better way to test your app for accessibility needs than to run it and use the technologies yourself.
2. Making them quick and easy to toggle makes it trivial to do so, which will encourage you to test things more.
3. That means you've got an open and constantly running challenge with yourself — there are simply no excuses to skip testing accessibility APIs when it's this easy to do it.

Epilogue: Are You Done?

When I first began in this industry, I knew I had a lot to learn. What I didn't anticipate was how much software engineering is less about the code you write and more about the people you create things with, and the people you create things for.

Whether you're a seasoned veteran in the industry, no longer wooed by a new framework or big release that's been in the works for a long time — or a bright eyed newcomer who feels the world is ahead of them in the tech industry, you're likely to discover something quickly:

Writing code is the easiest part of the whole gig.

It's the people, however, that make things interesting. They make or break jobs. They bring your development to new heights or hinder it altogether. They make you excited to go to work or dread clocking in.

People, and our relationships with them, will always be one of the most critically important parts of not only your career, but your life. And so it is, over the years I learned that accessibility programming is about *people*. People that are far too often understated in our industry.

As we talked about at the very beginning of this book – you won't be too hard pressed to find an app with millions of users that *doesn't* handle accessibility APIs well, if at all. The passing notion that "we can put it on the next sprint" soon becomes "it's technically challenging" and ends anticlimactically with "we'll keep an eye on it!" – and we know what that means.

Taking stock of the current state of affairs with hyper growth goals and the aggressive nature of the App Store and its competition found within it, that's less a revelation and more of a foregone conclusion. But we can do better, and we're naive if we think that our "we can fix it next release" promises won't inevitably turn into a job abandoned altogether.

That stance is easy to take until you are personally affected by the things people with accessibility needs deal with every day. Read the next few sentences a few times until you take them to heart, until you really *feel* it:

What if you were blind? What if you couldn't use your hands? What if you couldn't talk? Then, what if you couldn't use the apps that might bring you some joy or do your job because they simply didn't support accessibility APIs?

The work you do around accessibility matters to people, and it is never finished.

The wrong question to ask is "Are you done?" *When* are you done? Is it when the accessibility inspector reports zero errors? When you pass its audit? When you invest in accessibility testing at your job? Or is it when you reach out to folks with real disabilities to see how your app could be improved for them?

The answer, is of course, never.

Accessibility programming is predicated upon empathizing with your user base, making sure the entire world has access to the things you work so hard on making. Caring about quality and craftsmanship. And those are things you should never stop doing so as long you're putting code to compiler.

This all may make you think that accessibility work is like a task that you can't possibly finish. In some ways, yes. It's not a mountain to be moved, or task to be conquered. Instead, it's something that you get to do daily. Even though it's never "done" – you can make things better than they were yesterday, and for someone out there using your app – maybe that will make all the difference?

So, as I see it, never being "done" is a blessing. A privilege. It's an incredible responsibility as a software engineer.

And I think that you are just the right one to handle such a tall order.

Book Two: Design

Can You Learn Design?

When you think of Apple, you probably think of design.

Throughout the years, Apple has been the cause of several industry trends, both in hardware and software design. In fact, I would argue they even have a competitive edge when it comes to design outside of those two things.

Consider the design of their advertising and packaging. You could likely spot an Apple commercial within the first few seconds of its runtime. Maybe you could identify the iconic white packaging of their products when placed among others.

Outside of the effect that their designs have on consumers, people like you and I are no different. When Apple designs something, it becomes a trend. Remember seeing this design crop up at their keynotes?



The feature cluster design Apple has recently made popular among the design community.

Guess who else started using that same design for product announcements? Just about everyone else who had something to say in tech. Cultural osmosis occurs at a feverishly quick pace when Apple designs something.

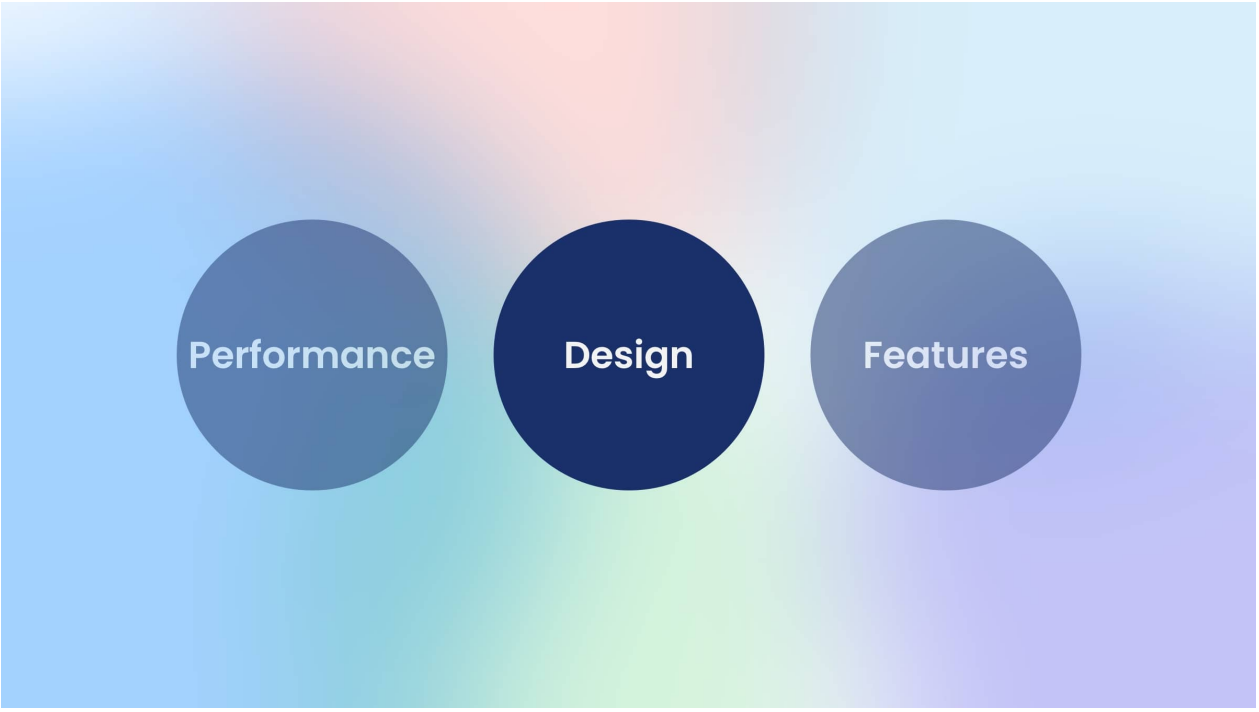
When apple speaks to design, people listen and follow. They set trends, shift existing ones and pave the way forward in several design-based conversations — like it or not.

Apple excels at design, and it has led them to places a lot of companies will probably never get to. If it weren't for their design pedigree, the iPhones wouldn't sell, the apps wouldn't be developed and the world would move on to something else.

But year over year, that doesn't happen.

I think it's because truly great design is *sticky*, and when people feel drawn to it — they have a hard time leaving it behind for other alternatives. The good news about all of this is that you and I can make our "dent in the universe" by leveraging design in

our own products. You can argue correlation versus causation all day, but I believe that if you can grow your design muscle, you can truly grow your product. Design is paramount to any app's success, no question.



Like it or not, design will make your app sink or swim.

The sticking point, then, is this: Design, and all that comes with it, is a heavy topic. Perhaps a bit loaded. Even, paradoxically enough, a bit messy. But it's an important topic, and I think it's the one holding most developers back.

Born Into or Grown Into?

Like any skill, design comes easy to some and quite unnaturally to others. The good thing is, though, it doesn't *really* matter which of those buckets you fall into. The only difference between the two is the time it takes to start creating objectively "good" designs.

Some do it quick. Some take years to figure out how. Me? I was the latter.

But we can flip this the other way, too. Take programming. It works the same way. From the very first class I had in college, it came naturally to me. Looking at code clicked with the way I think about solving problems.

There's an order to it. A universal uniformity, no matter the language you're using. You take puzzle pieces, consider the bigger picture and figure out how to make it all add up to something that compiles, is stable and runs. I also think programming has come to me naturally because an important question is typically already answered when I start: What, exactly, am I making?

By the time I get involved, a product has been scoped out, wireframes are made and a high-fidelity design arrives on my plate — ready for me to bring it to life. The parts that are most difficult for me are already done.

When *you* have to answer those questions, though, *you* have to design. There's no getting around it. What happens when the user taps here? Should this button be on the bottom or the top? What about color, typography, hierarchy, navigation — the list simply never ends.

Programmers have to think about those things too, sure — just as designers must take into account technical considerations. But the right-brain, left-brain implications of software engineering and design are a pertinent example of the differences the two disciplines have. This is painting with broad strokes here, I know, but programming tends to be more prescriptive and rule-bound, while design is more free-form and abstract.

So, are you the problem-solving savant who can code anything? Or the expressionist who can paint a beautiful canvas of interfaces in Sketch, Figma or anything else? A bit of both?

In the end, it doesn't *really* matter.

Both sides of the equation can do things to meet in the middle. If you've got the chops to learn to program, you can learn some good design. The opposite is true, too. I get some pushback on that line of thinking, but I stand by it.

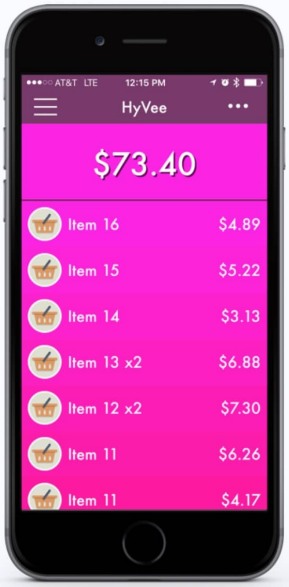
After all, design is opinionated.

The Impact of Thoughtful Design

So we know that design is important, and I've made the case that we can learn it. Next, let's look first-hand at what a dash of design has done for an app.

Allow me to get a bit personal, if you will. Here's a quick story about how design has impacted my life, and my always-evolving quest to learn more about it. Back in 2011, I had an idea. I wanted an app that calculated how much my shopping trips would cost, but to the penny. Discounts, quantities, local tax rates — all of it.

So I got to work, and I made it. It was called Spend Stack. Take a look:



Before I learned about design, this is what my apps would look like.

Yup, I designed that. All me. Before you ask for a discount on a book over design from someone who made that, stick with me.

Spend Stack did not do well. It was full of gesture-based operations, which could only be carried out using them. The colors were….striking. Not in the good way. It didn't cater to accessibility needs.

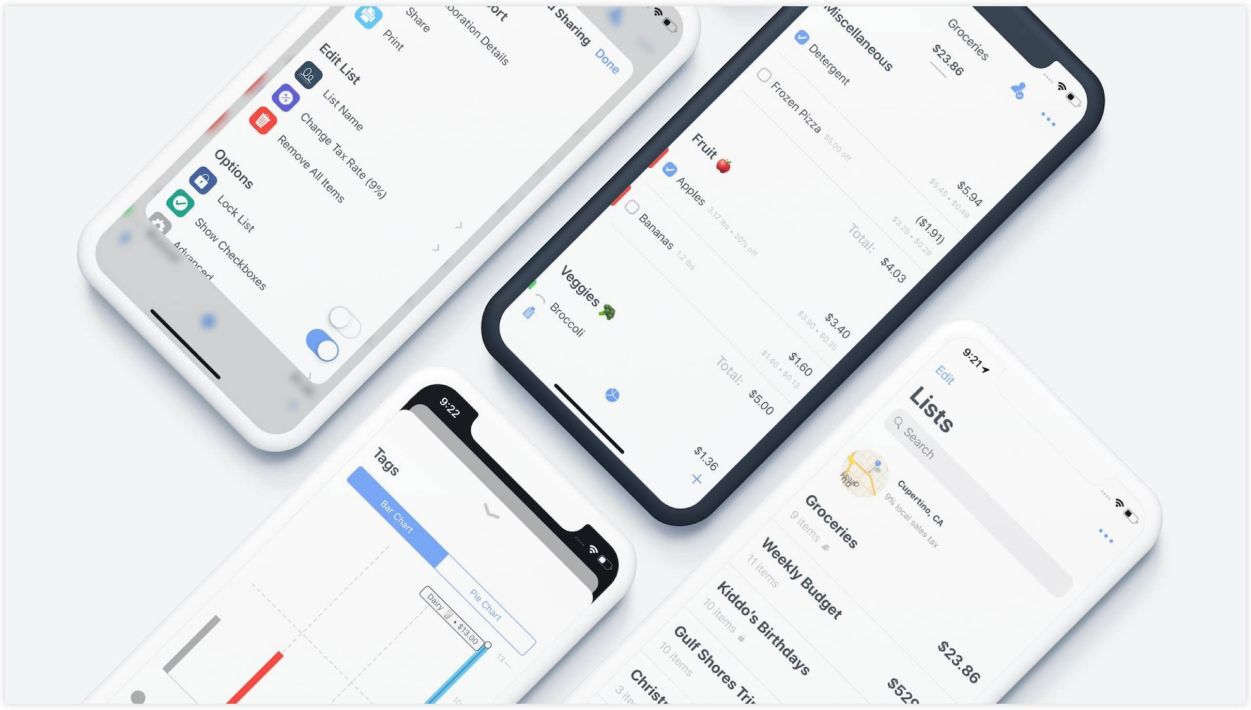
I spent a year or so on it. I shipped it, all starry eyed. Then, my family bought it on launch day, and they were about the only ones. In short, it failed.

I'm glad it did, though, because I'll never forget what one user who reached out to me said: "The idea is rock solid! It's just not pretty to look at or use." They were exposing the reality of where the app was.

Right idea, wrong execution. It wasn't designed well. So I started to take design seriously. I read Apple's Human Interface Guidelines. I started writing down in my journal why I liked some apps, and why they made me feel the way they did. I kept my fa-

vorite apps on my home screen, even if I didn't use them much – just to look at them. Stay inspired.

I did a few versions of this process for a long time (I still do). And in 2019, I released Spend Stack again. Only this time, it looked a little different:



After getting serious about design, my apps started to look and feel much better.

This time, it was downloaded, used, won awards – people liked it. It was featured in the App Store, used as App of the Day and critically received well. Apple even used it as a retail demo in their stores, loading it onto to thousands of iPhones and iPads to help showcase the best apps that those devices could offer to people. A year after I launched it, it was acquired.

So this time, things worked out.

Writing stories like this is always tricky – because I don't want to come off as thinking I'm some titan of industry. A design genius. I'm not. And honestly, I'm not even

close. In the end, my app was a tiny blip in a universe full of plenty of them. Regardless of the fact that it was big in my world.

But I can't call my own app beautiful, or bestow a title of good design on it. I can like its design, and feel good about it. And I (and you) should. But those are things that users will decide for me. Given this, let's look at the facts of my story, and how they played out. Because the end result of what happened is the only reason why I'm telling you any of this.

One app had no users, brought in no money and was everyone's bête noire who bought it. The other was critically well-received, made money and was acquired. The mission was the same, the features were close to identical - yet the one *major* delta between them made all the difference.

The design.

If you can start to learn how to design an app, care about that process, respect the discipline and learn more about its systems — you will go further than where you are today. No question.

So, how about it then? Can you really learn design though? Can you go from making something that's objectively bad to putting something together that makes you and other people happy?

Of course you can.

It's just not easy. It's hard, and often times frustrating. The progress you make towards being more skilled at design can feel slow. And, when things feel slow, you run the risk of conflating steady progress with not having the ability to learn something at all.

But you can learn it. Seriously, you can. And I made this book specifically to help you do it.

Defining Design

Before we start in earnest, I want to lay out a few things about design and the way I think about it. I don't want to claim that my definitions here are the canonical right ones, or that this process is foolproof. Neither of those are true.

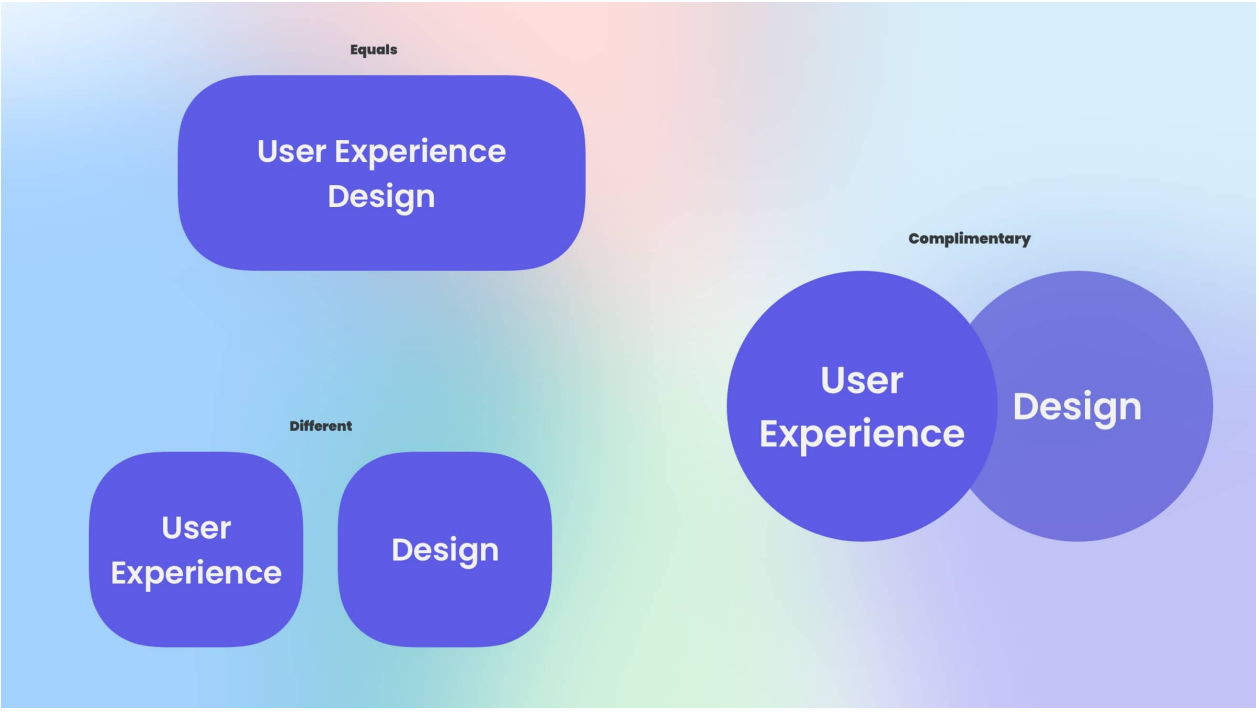
Nonetheless, I do want to let you know upfront a few things:

1. How I'm defining design in terms of this book, and
2. How that relates to user experience.

To me, design is how something looks *and* how those looks back up the app's goal. If it's a finance app, does its design really help me work with my money? If it's a sports app, does the design help me keep up with the scores? And so on.

It also ties into how something *feels*. And how something feels is predicated, largely, by how it works. But when we start going there, in my mind, we start getting into user experience. And user experience is incredibly important.

This is where different people have different viewpoints. Some think you can't possibly touch on design without including user experience. Some think they are different things entirely. Others think they simply work closely alongside each other, but are still unique topics in of themselves.



Everyone seems to have a different view on the relationship between design and user experience, and that's okay.

For our purposes, design and user experience are separate topics that are closely related. You need them both. But both also deserve their own spotlight. So in this book, while some of what I cover may veer into user experience, the focus is mostly on the design aspects.

This book series has an entire book over user experience specifically on iOS. To get the high-level thinking between how the two are different, consider this: A splash screen.

The design book (this one) would cover the thinking behind how you lay out the screen, how to consider fonts, colors - what it should do when the device is rotated. The user experience book would speak more to how that screen should be snappy – informative, but not in the way. If it relies on a network request, it would talk about how to handle that so that it doesn't impede progress.

It's a bit reductive, and I apologize for that — but one is more about how it looks. One is more about how it feels. And that's how I think about design and user experience.

Also, you should know this book is also highly opinionated. What you are reading is a mix of advice from Apple designers or evangelists, guidance from the human interface guidelines and a whole lot of what I personally think about design, what makes it great in the context of iOS and why you think about X or Y. You're likely to find a lot of things you agree with, and a lot of things that, well, you don't.

That's okay! Put your own spin on things where you see fit. Design doesn't happen in a closed off box with written rules, it happens with a good foundation of what works mixed with a healthy splash of your own creativity and ideas.

What this Book Covers

Okay, last thing. I want to lay out what you can expect to read from here on out. You can largely think of it as three separate schools of thought.

The first one is how to start the design process. How do we know what we're designing for, and where to start so we can know that at all? Then, we'll move over to how to shape that design a little further. We've got an idea of what to make, now we have to fight through the huge ideas and sand them down into something tangible. Finally — how does the next phase of putting together a design work *specifically for iOS*? How do we think about navigation on the platform, what rules does iOS have and how can we know when to follow them or break from the mold?

It's a fun journey. Let's get started.

The TL;DR

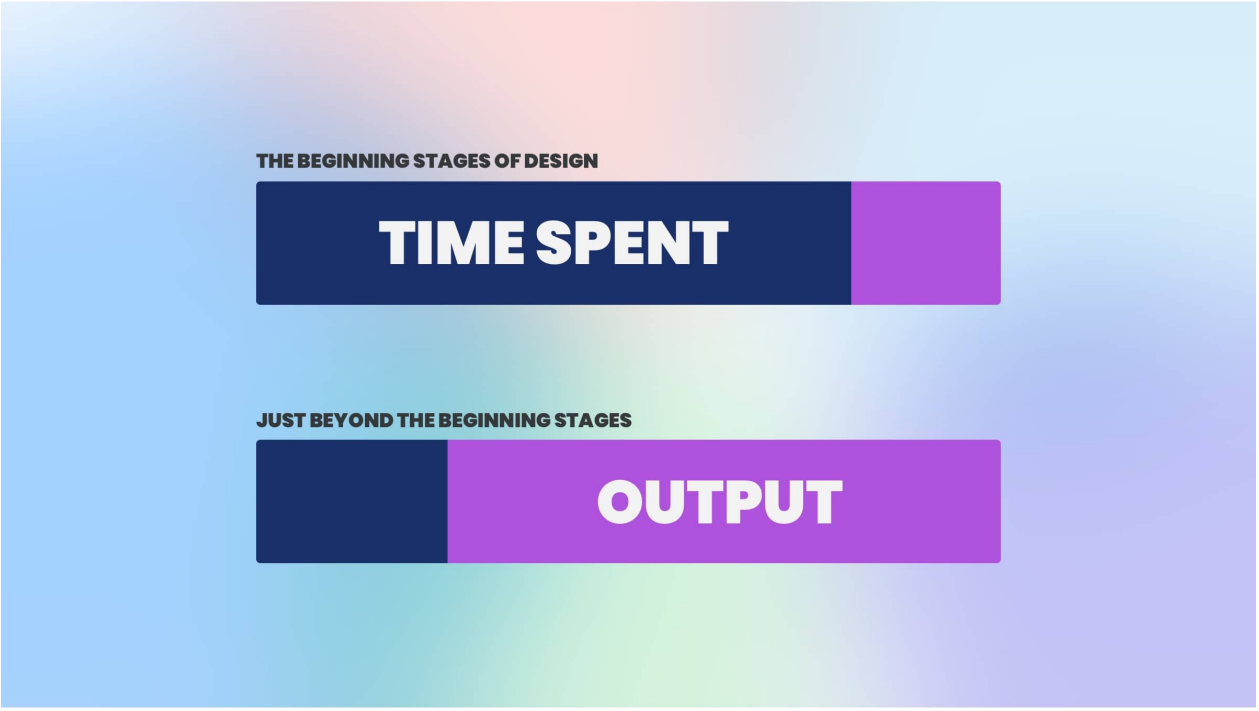
Anyone can learn design, and it pays off to invest in doing so because it can take bad apps to good, and good apps to great.

What, Exactly, Is Your App?

Let's start with some good news:

The hardest part of design is the beginning.

By far, too. Once you get started, you can really get moving. Progress starts to come a little easier, ideas to start to flow and you are putting the big pieces together. Getting to that point, though, now that can take a bit.



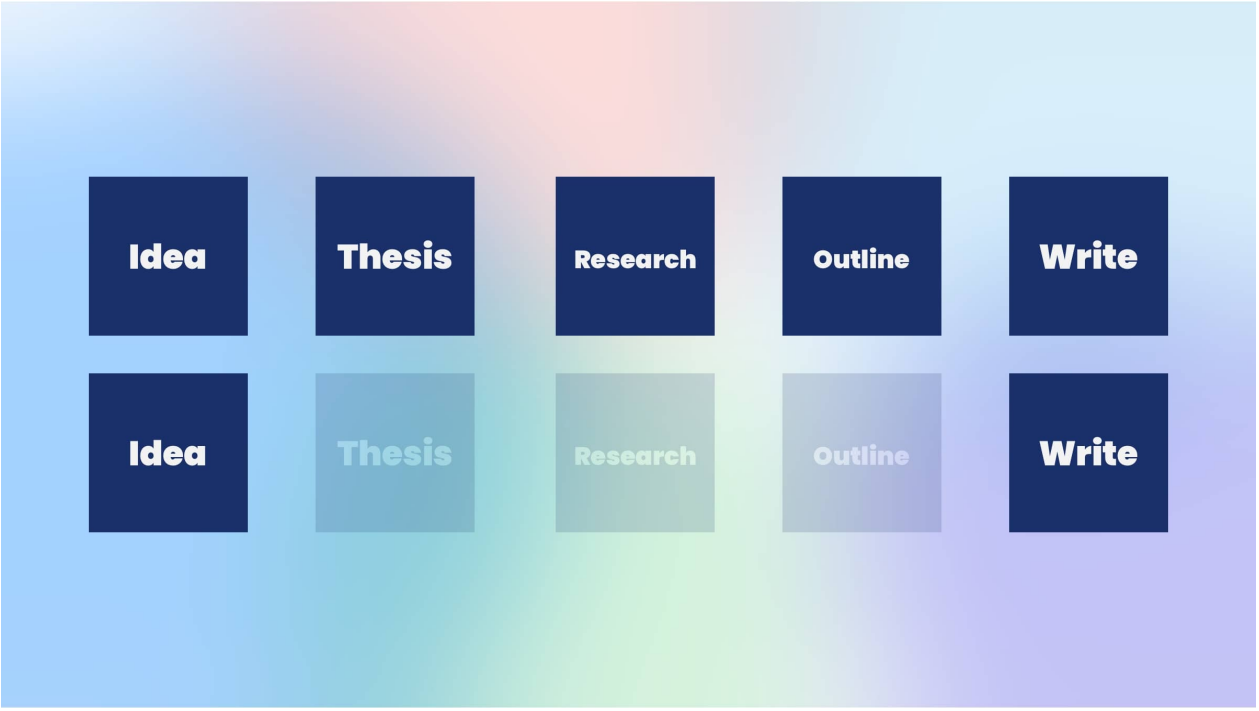
The hardest part is the start.

So, where do you start? How do you build up to that momentum?

One good way to start breaking the problem down is by taking cues from other disciplines to observe how they solve similar situations. Consider authors, or anyone who has a lot of things they've got to write. They have obstacles standing in their way of getting started.

Consider this — as a writer, nothing is more intimidating than a blank page. That sight of a blinking cursor staring back at you, almost sarcastically provoking you to write something, anything, can stall progress.

But, writers don't typically start there. If they did, most of them wouldn't get far. Instead, they have a system in place to combat "blank canvas syndrome". By the time they've started to write, they probably formed the thesis for the book or piece, have done a healthy amount of research and ended up with a robust, but mutable, outline.



Writers just don't jump into writing, they lay a foundation first.

Designing for iOS is no different. We've got to lay a foundation before building up our designs on it. Let's figure out how to do that.

Don't Start With Designing It

By now, you can probably guess that the beginning of design doesn't involve firing up Sketch, Figma or even busting out pen and paper. As the saying goes, design is far more than the perfectly placed pixels you hopefully end up with. The beginning stage is less ceremonial than that. That's because the starting phases are mostly centered around answering a lot of questions.

If you haven't asked the right questions, you won't come up with the right designs. So, what are the right questions to ask yourself? Here's where I start:

1. What problem is this app trying to solve?

2. Who would it be for — who is most likely to use it?
3. What does the smallest version of this app look like?

None of these questions are written in stone or have proven to be qualifiers of great iOS design, but they help get to the heart of the issue which is figuring out the app's purpose and who it might serve. And there's nuance to these, too.

Once you have these questions answered, you've got more than just a fleeting thought about an app that would be exciting to develop. It takes these racing ideas and endless potential you've got floating around in your head and channels them into something actionable. Something you can make a bit later, but start to design right now.

What this *doesn't* mean is that you just have an idea. Everybody has ideas. And everybody, and I do mean everybody, has an idea for an app.

Very few of them, though, actually take time to flesh those fleeting thoughts out much more more beyond a "this sounds neat" phase. I'm not saying you should give yourself a truckload of stop energy and write a dissertation about why an app should exist, but what I *am* saying is that by asking yourself some important things upfront — you can save a mountain of heartache and wasted time.

In short, answering these questions really shouldn't take longer than thirty minutes to an hour. I think being nimble here is an advantage, actually. Otherwise, you start to wade into analysis paralysis.

Another reason to not get bogged down here?

Design inherently leads to a lot of organic discovery. There's no reason to get caught up in details or edge cases, at least *not yet*. Once you start to place things in your interface, the tough questions will come up anyways. You'll get to them, but stop-

ping your enthusiasm and momentum too early can be a death sentence to would-be projects.

So, now we've got these questions I think are vital as a precursor to iOS design. What does it look to think about each one? How high or low do you go?

Here's how I tackle all three of them. To recap, we're answering (at a minimum):

1. What problem is this app trying to solve?
2. Who would it be for – who is most likely to use it?
3. What does the smallest version of this app look like?

Because when we've answered those questions, we have a better idea of what we're trying to design.

On a Mission

A study found that about 10% of every dollar a business spends goes to waste as a direct result of poor management efforts or unclear objectives[1]. While the dollars and cents may not be as dire for you or me, I'd argue the cost of fiddling around with an app you haven't thought through yet is even higher – sunk time.

I'm not talking about the time you spend to learn a new API, or having an app you hack around on "just because" – those are all great things! I encourage them. It's how you learn a lot of things that you can take forward to other projects. But, once an app starts to "get serious", as it were, you have to shift your thinking along with it. That's true no matter if the app is a side project or something for your day job.

And if you can't decide what the app is, you'll waste a lot of time.

---

[1] https://www.pmi.org/learning/thought-leadership/pulse/pulse-of-the-profession-2018

Trust me! I've got the scars to prove it. Recall my app Spend Stack from the previous chapter? The biggest challenge I faced with it was that I didn't define exactly what it was upfront. Was it a pure finance app? A grocery, or general shopping list, companion? A running expenses calculator?

The problem was, it was all of those things. And while you can make the argument that being the jack-of-all-trades and master of none is a good thing, that's not what I set out to do. Eventually, the users of the app greatly influenced "what", exactly, it was and should be.

If your users are telling you what your app is *after you've already shipped it*, and it's not necessarily close to what you had in mind – that's a problem.

Learn from someone who has been there, define the "what" first. Most refer to this as the app's mission statement, parlance I'm particularly fond of, but you can call it whatever you want. The app's purpose, its main value add – whatever. What it does, though, is define one thing the app has set out to do.

It can look like this…

(App Name) is a (category of the app) that does (problem it solves).

…or possibly closer to something along these lines:

(App Name) (problem it solves) for (target market or audience).

Feel free to strip it down even further if you like, this is the one I personally use most often:

(App Name) (Value add)

This shouldn't be longer than a sentence, and if it is – challenge yourself to narrow things down. A common misconception is that an app's mission statement has to be grand and ambitious – and they really don't.

Let's try it out with a hypothetical app, we'll call it MiniStocks. MiniStocks will do a few things well, but its main focus above all else is showing a few particular stock symbols quickly. We'll start with a mission statement that is a little too big and narrow it down from there:

> *MiniStocks is a way to view three stocks at once with the ability to view their historical data for traders who only care about certain companies.*

This is way too long, first of all. While it does focus on core *ideas*, it's too verbose. A mission statement is something you will be glancing at a lot, and if it's long – then it's going to be too hard to grasp quickly.

Let's take out some of the excess wording:

> *MiniStocks shows three stocks quickly while allowing users to view more historical data.*

Better, and we're getting there – but the former part is where the "meat" is, and the latter part is more of a feature than a core responsibility. Think of it in parent-child relationships. You can't dig into the stock's historical data without being able to quickly view it. So, let's drop that and see where we are:

> *MiniStocks is a stocks ticker built for quick, short-burst viewing of three stocks.*

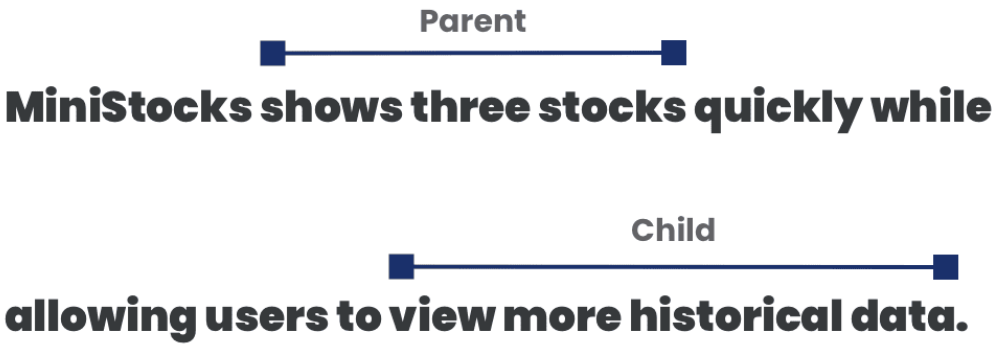
There we go. I like this. It's super short, easy to digest and focuses on one, core responsibility. I encourage you to strive for this clarity.

Here's why that's important:

- You should use this to benchmark features. Does X help us do (mission statement)?
- As I mentioned, when you need to reset your frame of mind to get focused on critical thinking about your app, reading the mission statement is where you should start. So, the shorter the better.
- It helps you know what to cut, and what to keep.
- It helps others who may later help you with the design or development of the app quickly get up to speed of what you're trying to do.
- It helps with promoting and advertising your app, the "elevator pitch" aspects.

The list goes on. Plus, we aren't losing anything here by being short and concise, but we gain all the advantages listed above. For example, you may be getting a bit worried we cut out the "dig into historical stocks data" out of the mission statement. That's okay – it doesn't mean it's gone, it just means it's not the core idea of the app.

If you get stuck on your app's mission statement, where it may have one or two main ideas, use the "parent-child" method to try and figure out which is more important.



What you can't end up with is an app with too many competing ideas. If you're at this spot, figure out what truly gets to the heart of what you're trying to do – then cut the rest or see if they make sense as a complimentary feature to the core mission statement. Basically, two is a crowd when it comes to defining what your app should be.



Too many ideas fighting for mind share is a sign that you need to narrow your focus.

At this stage, don't get stressed out about having a lot of answers. You're after just one. And it's okay if things take a turn later on. Can your app's mission statement change? Could it evolve? Be something entirely different?

Absolutely!

But that doesn't mean you didn't at least start with a North Star goal in mind. You're not likely to just fall into one, you need to be intentional about it and make one yourself.

Who Is It For?

This question may be thought of in tandem with the above since they often work together to form a strong identity for your app. Who is likely to use this app? What ten-

dencies do they have, how do they like to be communicated with — what's important to them and what isn't?

At this point, you might end up in one of two buckets. First is the niche, more focused bucket. The second is the general, catch-all bucket.

If you're making a finance app, well, everyone has to think about finance, right? All of us use money, spend it, save it — whatever. There's no avoiding it. That doesn't mean by default our app is a generalized app with a generalized audience, though.

Let's go back to the MiniStocks example. Do we want the "everyman", for lack of a better term, to be using it? Or, are we targeting established traders or novice investors? Perhaps we only care about highly experienced ones.

All of those have a lot of differences, and all of those need different design affordances. So, if you think you have a generalized app for everyone — maybe you don't. And, if you don't, I think that's actually better.

It is okay, and valid, for some of those audiences to cross over, though.

I don't think you need to get hyper specific when figuring out who is going to use your app — but I absolutely do advocate that you at least spend some time thinking about it. For MiniStocks, things could look different depending on which type of person we focus on even if the app's mission statement remained the same:

If we targeted a general audience…

…then maybe we'd include some educational features or content to help them learn about how stocks work, what they are and how the market functions.

If we focused on established traders or novice investors…

…then maybe we'd up the information density in some places with the faith that the users of the app might already understand stocks, how to trade them or the nuances of the market.

And finally, if we focus on highly experienced traders…

… perhaps it would make sense to include more "power user" features to gauge whether or not they should make some trades on that day.

The app's mission statement doesn't change whoever we are making the app for (though it can happen), but the way we might execute on that vision might. Plus, when we initially thought we had a very "general" audience, what we really had was an audience who might be beginners in the finance world.

I personally believe you should try to narrow in on who you want to be building for as best as you can while leaving a healthy margin of wiggle room in your thought process for folks who are just above or below that fold. When you do this, you can really narrow in on pain points faster. If you've got a focused idea on the group of people you're making something for, you can get a feel for what *not* to pursue just as easily as you can figure out what you should be pursuing.

If you feel like you're in the second bucket (a very general audience), where you think that your app might be for any or everyone — I'd encourage you to revaluate things. It probably isn't for everyone, and that's absolutely fine. Even if the audience ends up being extremely large, there are likely overlaps and similarities within that cohort you can find and optimize for.

Now, Narrow It

Once we get here, we know what we're building and a have a pretty good understanding of who we're likely building it for. Now, we determine what features support that goal and those people. Then, we chisel all of those features down to something that's shippable.

We could fill an entire book on the concept of an M.V.P. (minimally viable product), the "lean startup" methods or whatever you want to call it.

I won't be doing that.

Instead, I keep it extremely simple. We need to know the answer to this question, "What does the smallest version of this app look like?"

At this stage, figure out what the most "shippable" version of your app would be *feature* wise. To do that, we'll lean on the app's mission statement to determine three things:

1. What we should do now.
2. What we might do later.
3. What we won't do at all.

Things you answer for the first question should be a mix of a major feature, a few supporting ones and a "this is nice" type of value add. Here's how I would think about it for MiniStocks:

> *To ship version one, we'll have a main view showing the three main stocks a user can pick to view right away each time the app is opened. We'll also let them tap into it and see (A, B and C). We'll add viewing (D, E and F) later. To add a little polish, I'll make a custom transition from opening a stock to viewing its details.*

I think that feels like the right amount. You get to the core purpose of the app, add a little bit of functionality to support that while letting yourself indulge a little bit with that one "nice to have" part.

Main Feature

Show three stocks quickly.

Supporting

Show some detailed data points about a stock.

Just for fun

Snazzy transition from stock overview to data.

What you want to ship is something that gets out the door in a timely manner, supporting your app's mission statement is the quickest route. You've got a focused release. If you did it right, people who have used the app can probably tell you what the app's mission statement is and not be far off from what you thought it was, too.

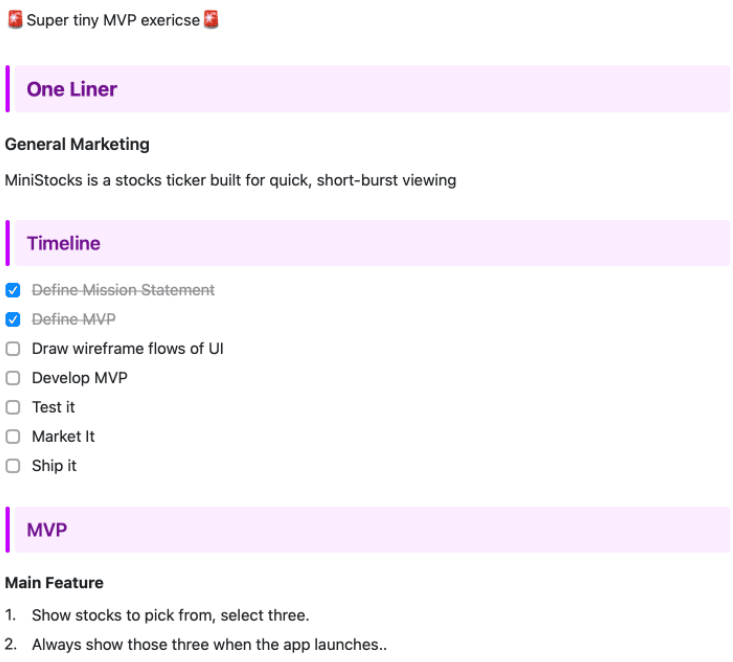
I also want to focus in on a common misunderstanding about shipping a 1.0 version here — because it directly relates to design. Shipping your version one doesn't mean you left a lot of design considerations on the chopping block, it means you left a lot of features out that could come later after your first release.

This is important for a myriad of reasons. People delete apps they don't enjoy from a design front quickly, but they will tolerate and stick around for one that might not have feature parity with something else yet. The inverse of that is not true, from my experience.

If you've got a ton of features, but they are slow, obviously embedded in a lot of web views or they don't look right – your app is gone unless it has a strong monopolistic hold on the app's genre.

Instead, opt to do a few things well instead of a lot of things okay. This is your chance to narrow things down and get focused to deliver a razor sharp version one. When you have a only a few tasks to work on for a release, you leave yourself more time to make sure they're designed appropriately.

To keep me to task and not stay in "development hell", I usually make a document like this:



Again, I want to stress that this first version should be absolutely rock solid in terms of accessibility, design sensibility and user experience. Building things like that, contrary to popular wisdom, really isn't what takes time or is the cause of an app lingering to the version one finish line. Shipping a lot of features upfront is what does that.

So, narrow it down and make it manageable, and then you can allow yourself to deliver on those things with a design that supports it. Sacrifice on features first, not on quality.

Ambiguity is your number one enemy to great iOS app design, this is your last chance to stop it before it creeps in.

Bonus Questions

Here are some other questions I like to ask myself, but they are supplementary to the ones above. In my opinion, I don't ever skip the aforementioned questions – but I like to think about these things throughout the process too.

How can it be set apart?

I think if your motivation to create something is along the lines of "an app like this already exists, but I can do it better" – that's a perfectly good reason to make an app. Several of my close colleagues disagree with me on this. A common refrain might be "this app exists a million times already", and yeah, they are probably right.

In that case, the world doesn't need another one of those apps. My friends are correct in that regard, no question. But what people always want is an app like that which simply does things better, feels better, looks better and works better.

Don't let market saturation kill a project. Be aware of it, respect it and be realistic with your chances. Success with these projects is probably the hardest to pull off, because what you lack in originality you have to make up for in execution. It's possible, but it's a steep hill to climb.

But timeless design can overcome a lot of things. Market saturation is one of them.

Is It On iPad?

One of the first questions you might ask if, "Should I ship it on iPad?" I think that answer should nearly always be yes. We'll look at how to do this in a lean way without skipping out on quality design in the "Device Considerations" chapter.

But, to give you a shorter response – with the advent of technologies such as AutoLayout, SwiftUI and the power of UIKit in general, it's easier than ever to support iPadOS. I think you should, and if it's not part of your first shipping version, it should follow shortly thereafter.

Part of designing a great iOS app is adapting to form factors that the device throws at us – and there are a ton of them already *just on the iPhone*. For example, this chart from Apple's developer documentation just goes to the iPhone X (a model that's several years old) and already the delta between the smallest and largest is huge:

iOS Device Display Summary

Table 2-1 summarizes the physical dimensions of iOS displays and how those pixels are mapped to the logical coordinate system in UIKit.

Table 2-1 Screen Geometry

Device	Native Resolution (Pixels)	UIKit Size (Points)	Native Scale factor	UIKit Scale factor
iPhone X	1125 x 2436	375 x 812	3.0	3.0
iPhone 8 Plus	1080 x 1920	414 x 736	2.608	3.0
iPhone 8	750 x 1334	375 x 667	2.0	2.0
iPhone 7 Plus	1080 x 1920	414 x 736	2.608	3.0
iPhone 6s Plus	1080 x 1920	375 x 667	2.608	3.0
iPhone 6 Plus	1080 x 1920	375 x 667	2.608	3.0
iPhone 7	750 x 1334	375 x 667	2.0	2.0
iPhone 6s	750 x 1334	375 x 667	2.0	2.0
iPhone 6	750 x 1334	375 x 667	2.0	2.0
iPhone SE	640 x 1136	320 x 568	2.0	2.0
iPad Pro 12.9-inch (2nd generation)	2048 x 2732	1024 x 1366	2.0	2.0
iPad Pro 10.5-inch	2224 x 1668	1112 x 834	2.0	2.0
iPad Pro (12.9-inch)	2048 x 2732	1024 x 1366	2.0	2.0
iPad Pro (9.7-inch)	1536 x 2048	768 x 1024	2.0	2.0
iPad Air 2	1536 x 2048	768 x 1024	2.0	2.0

With a little practice, hopping up to the iPad will come naturally in your design process.

Who Do I Look Up To?

Throughout any design process, even to this point where we haven't truly started laying things out, it pays to find some inspiration. I refer to these as "North Star Apps", the ones you have really look up to.

Have a few these on your home screen, and come back to them early and often when you get stuck on a problem. A few readily available reference points will help you out as you start to design in earnest.

The TL;DR

By figuring out what your app is, and who might generally be using it, you're ready to start coming up with a quality design that supports it.

Embracing Scattering Thoughts

And now comes the fun part. Up to this point, we've figured out three important things to help us start designing an app:

1. What problem the app solves.
2. Who it might be for.
3. What the smallest version of it looks like.

Beyond that, we've got a tightened up list of features we want to support out of the gate. We've identified things to do now (which are paramount to the app's mission statement), some to do shortly after launch and some we might vet down the road in the future.

So the challenge now becomes — what does it look like?

To answer that, we'll be as quick and nimble as possible. At this stage, a lot will change because a lot of questions will get answered. This stage is a little messy, and by the end of it, nothing is likely to look pretty.

It's important, though, because this phase helps squash any ambiguity you might have about *how* the app will work from a design standpoint. More questions will remain, sure, but we've got to lay a foundation to build up to the good stuff.

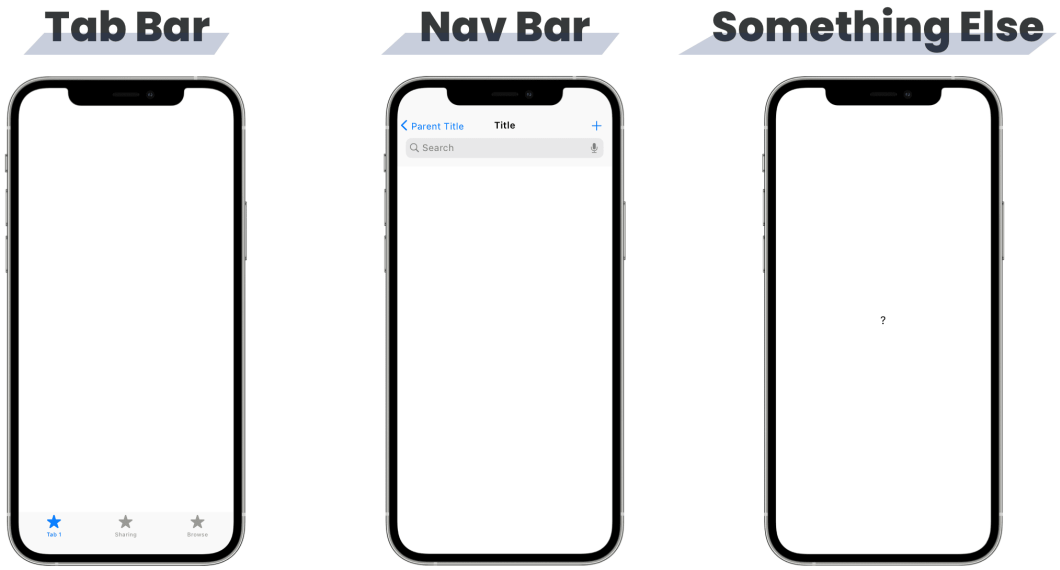
Here, nothing will be high fidelity. This is all about sketching flows on paper, laying it all out on the table and figuring out what makes sense in context of the app. You've probably got a million different ideas of how the app should function, so now it's time to prune them down and find the ones that work.

Let's get started.

Early Designs

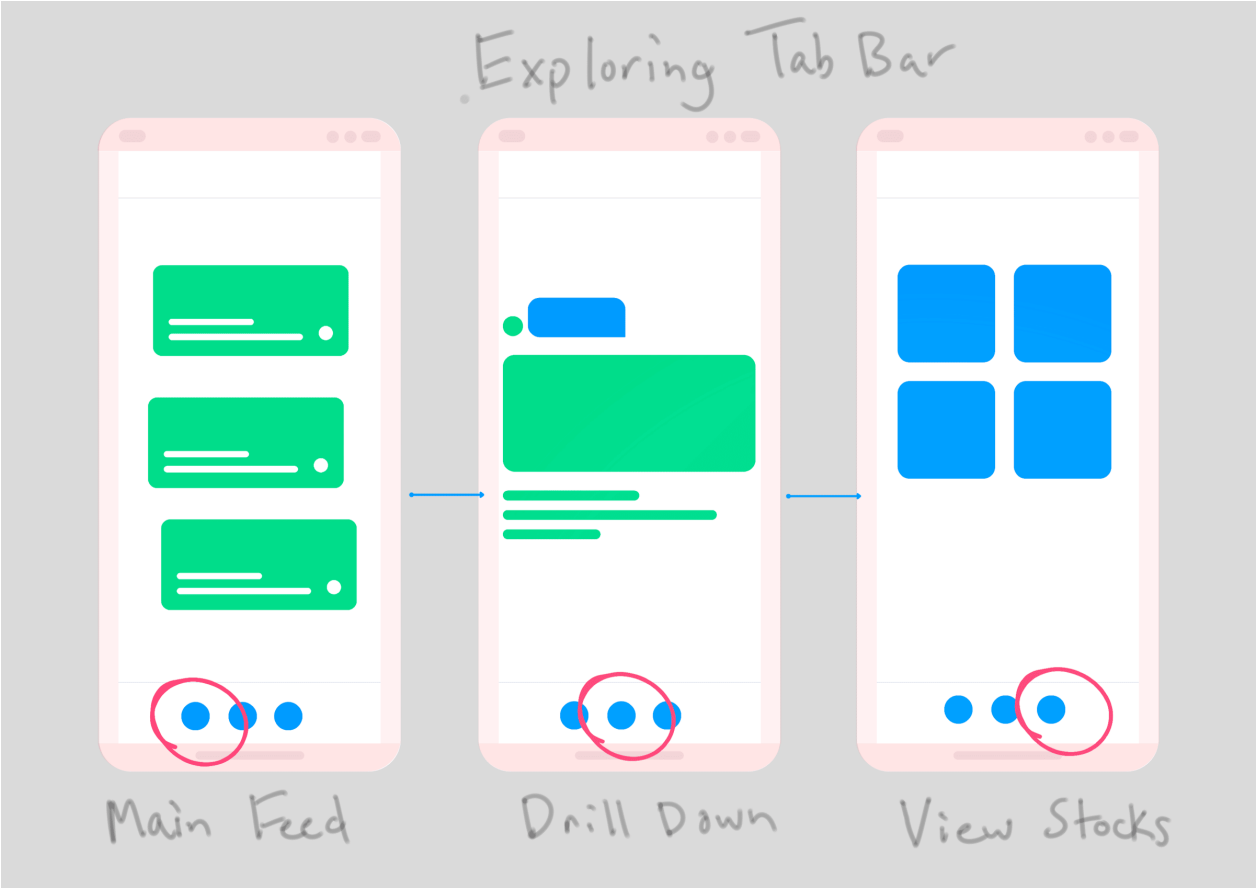
As I alluded to above, I start this phase of design with nothing more than a sheet of paper or any app on the iPad which lets me draw. This phase is mostly about *flow* – how does the app guide the user through all of its functionality? The goal here is to figure out that flow, and along the way we'll likely discover all of the screens that we'll need to develop.

The hard part about all of this is that you've likely thought of several ways people could use the app. Does it use a tab bar for navigation, perhaps a navigation stack? Something else?



iOS offers several ways to promote sensible navigation, but they each have different strengthes and contexts for which they work.

The truth is, any one of those might be viable. And, I can't really figure out which one actually is until I think through them critically by sketching them out. Then, I have a feel for where they all might take me. These sketches aren't pretty (I'm far from an artist, nor do I have a steady hand) but they let me explore thoughts and ideas easily:



A lightweight sketch helps you figure out if one idea works as good in practice as it did in your mind's eye.

This is a process which helps me cut through the noise at an early stage, let's take a look at it in a little more detail now.

Sketching Out Flows

Mentally, I like to start right at the beginning – the splash page. From there, I can already figure out a few things:

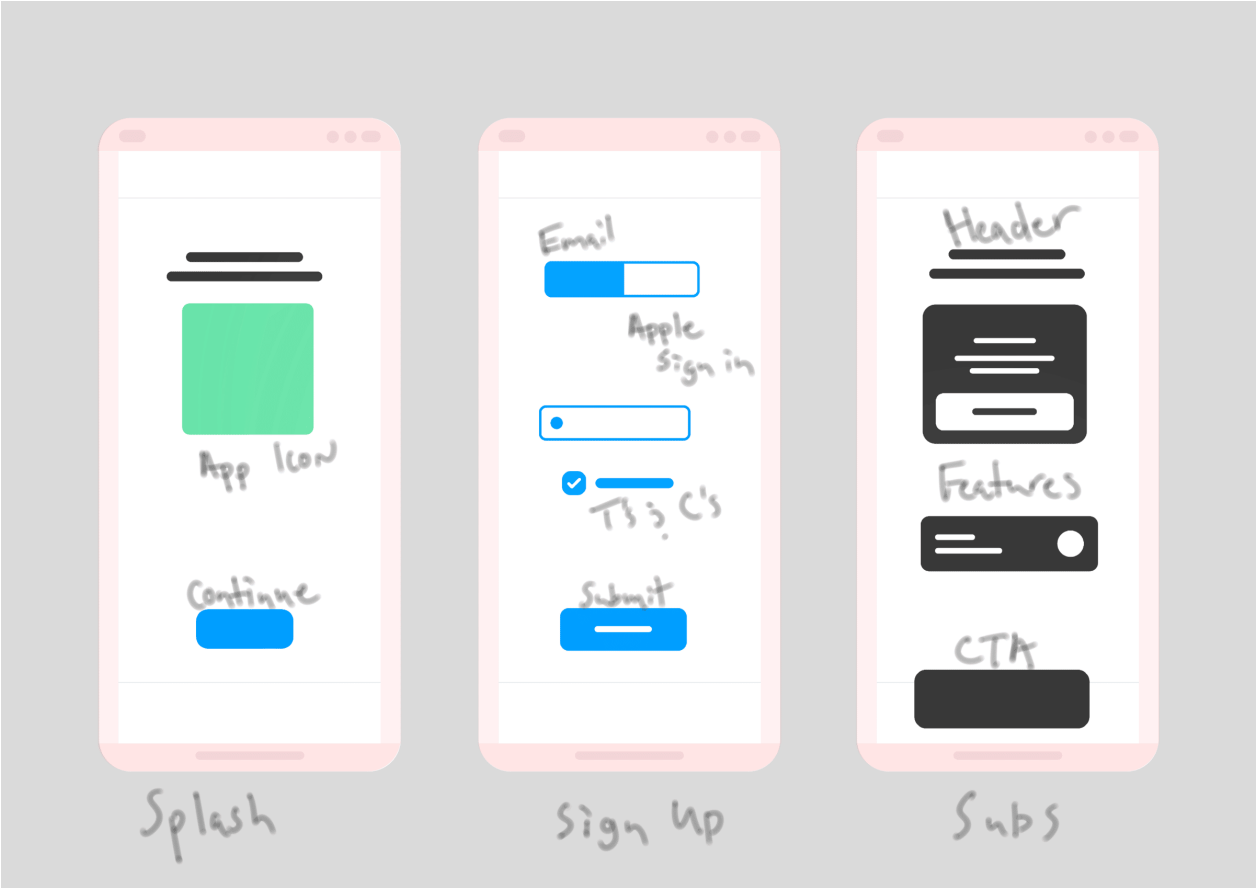
- How much do I mention upfront to get them started in the app?
- Is there some sort of sign up flow?

- Do I present any subscription pricing?

All three of those work entirely differently. Which, again, is why you must be nimble here. By having the low effort medium of pen and paper or a drawing app, you can quickly jot them all down to see how they feel.

Be sure that when you are asking these types of questions to not go too far into detail. For example, even if I knew that I wanted to do a subscription flow I wouldn't write out all of the copy or designs for it here. I would just know that's what I'll likely end up doing, and that it's part of the app's flow.

Let's revisit our canonical example, MiniStocks. Here's what I would sketch out from those scenarios above:



Explore competing ideas by quickly sketching them out side by side.

At this point, once you've drawn those out – it really pays to visualize them in a high fidelity manner in your mind's eye. If you were to open up the app you're working on with any of those flows, how would they feel to you? Does one work better than the other?

For MiniStocks, our mission statement was the following:

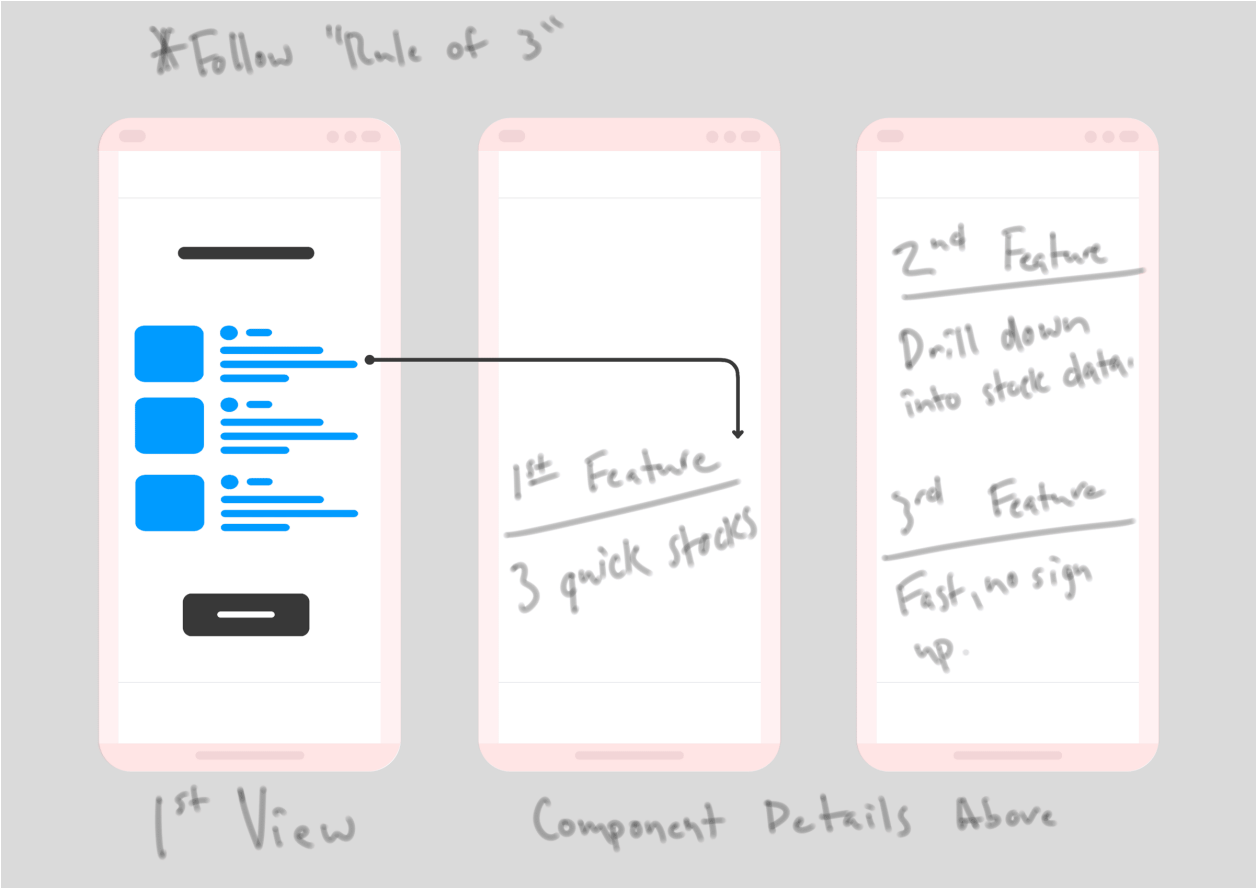
MiniStocks is a stocks ticker built for quick, short-burst viewing of three stocks.

Since I want to promote the main value proposition as quickly as possible, I can make a couple more choices:

1. I want to defer any sort of sign up. People can use the app without it. Generally, I advise following that path if you do have any sort of sign up component.
2. I want to show as little as possible on the splash page, with the goal being to get right into the action.
3. Any monetization it has will come after the fact, once we've proven the value of MiniStocks to the user.

The leverage you might have to make those choices will vary depending on if this is a personal project or you are in a team, but the sentiment doesn't change. You are looking for the right flow that fits the app best. All of those choices work together to create that flow.

With that in mind, here's where I would go next – by thinking about that splash screen a little more:



I tend to write just as many notes in my sketches as I do actual components or drawings.

At this point, it's tempting to really dig into to what makes a great onboarding page.

Don't.

We'll get to that in a later stage. In fact, I don't even worry about copy. I just scribble a few lines to let myself know that I'm thinking some text controls will end up there. As you've likely noticed, I don't even try to line up wire frame components perfectly - it's just enough to give me a good idea of where I'm headed next.

Feeling out the App

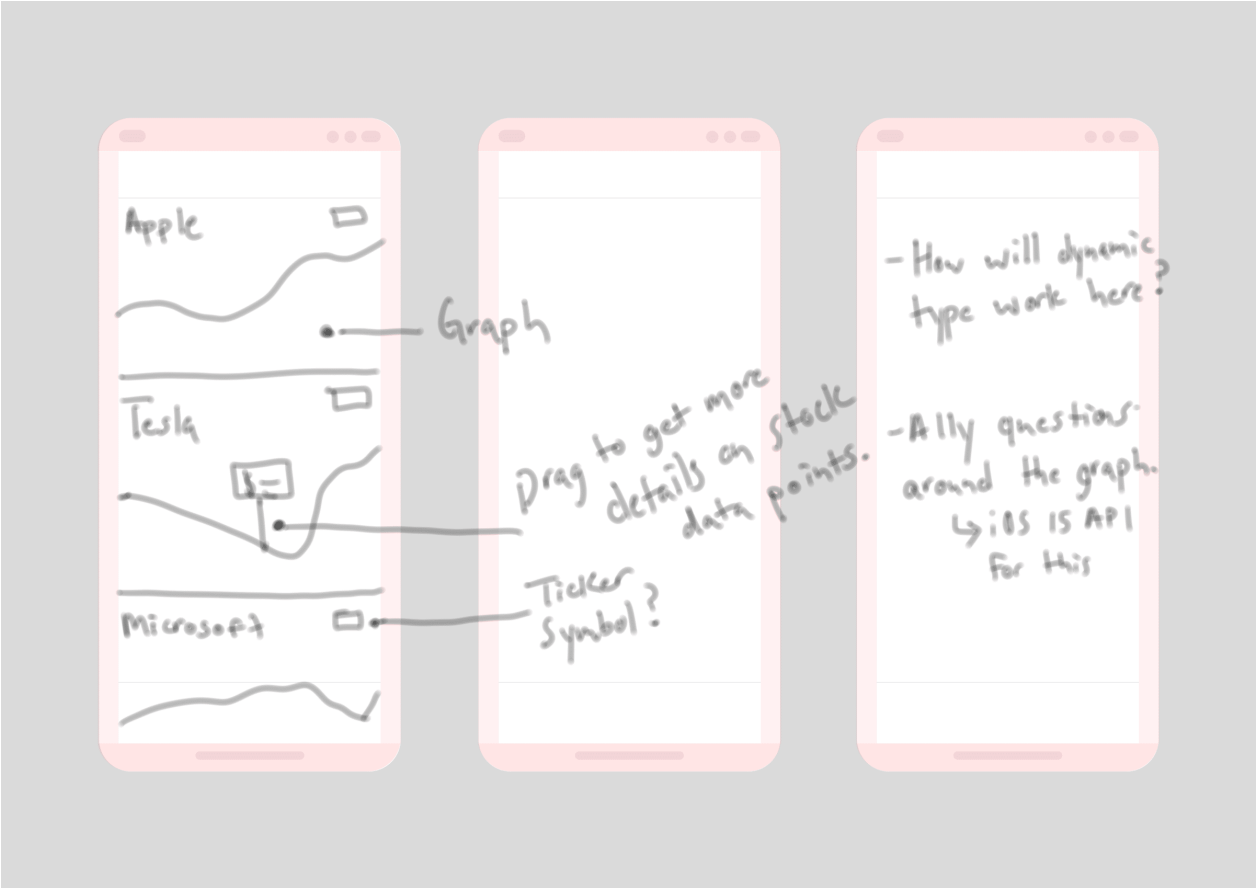
At this point, we've started to build up some momentum. For me, the hard part is getting started. And, well – we've started!

Now, it's about taking yourself through the rest of the app. Think through all of the places folks could go. For a quick refresher on MiniStocks, this is where we landed feature wise for the first version:

> *To ship version one, we'll have a main view showing the three main stocks a user can pick to view right away each time the app is opened. We'll also let them tap into it and see (A, B and C). We'll add viewing (D, E and F) later. To add a little polish, I'll make a custom transition from opening a stock to viewing its details.*

That answers the what, and for "the who" part – we didn't settle on one audience or another for the sake of providing examples in the last chapter. So, let's just pick one type of audience now to inform our design going forward – and that's the novice audience. We'll assume, going forward, we want to target those who might be new to stocks or trading.

Now that we've "opened the door" to the app by getting the splash view sketched out, let's feel out way through the rest of the flows. We know we've got a stocks app where we want to show three stocks right away:



Now, we can make think about some decisions:

- Should the user pick the stocks as part of the onboarding flow?
- Should they just show up right away, as in the sketch above?
- Perhaps something else?

This is why I love using a low friction medium during this stage. I find myself drawing out several different scenarios to see which would be best. These kinds of choices can happen at different parts of the app's lifecycle. At larger companies, these choices have likely already been made – yet can change during design or implementation.

For Indies, they can happen all over. I've found that momentum is one of the most important parts of app development, and if you can find it, you need to keep nurtur-

ing it. As such, since there are so many things being discovered at this stage, simply sketching them out and then sticking with the one you feel good and leaving the rest of them behind is no time lost.

For MiniStocks, since we've settled on targeting a novice audience, I think we'll opt to pick some high-impact stocks for them. This supports the app's mission statement as well, as it gets them right into the action and we can make what could be a slightly stressful decision (which stocks to look at first) on their behalf to avoid any stop energy it might create, which could keep them from using the app.

From here, the process is rinse and repeat until you've crawled through all of the main screens of the app. For MiniStocks, we decided that you can drill down into a stock to see some information. Let's say that information is:

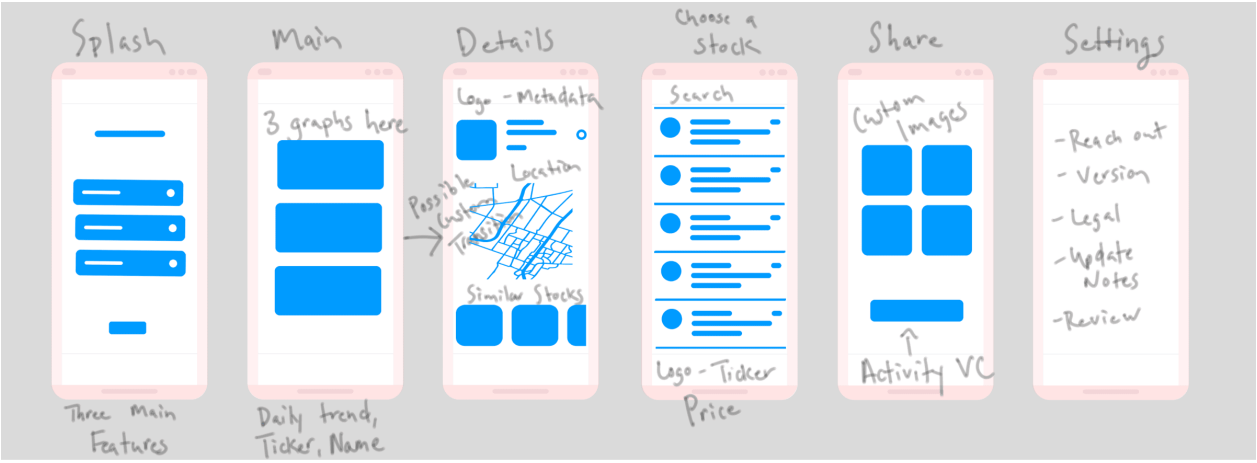
1. Performance over a time range.
2. Market Cap data.
3. Some information about the stock.

> *One thing I want to stress here — you may or may not know at this stage how much data you have access to, or if you can even get it. Instead of spending days comparing API prices and data access providers, try to stay nimble. Time-box your decision if you can, finding a good compromise between value and utility between API providers and then move forward.*

The inclination here is to go *further*, and it's easy to have happen here for two reasons:

1. The medium that we're designing with is low friction.
2. Since things are moving fast now, you don't feel the "weight" of adding more screens or features at this stage.

Stick to what you've already identified as your version one, and remind yourself that anything else comes after. After cycling through the screens, here is what my sketch of what MiniStocks might end up like:



It may not be pretty, but it does bring clarity. We know what our app is, who might use it, what it does and now – how it'll feel to execute on all of those thoughts. How many screens it probably has. How it navigates. A rough idea of how it looks.

Also just as important is what it *doesn't* have – added features. We stuck to our initial goals.

If this chapter leaves you with anything – I hope it's this: Ambiguity and indecision kills product momentum. Don't let it. Wireframing or sketching out an app shouldn't be a process about pretty straight lines and figuring out every color or element, it's about getting the big picture of what the app is.

When you find an app's flow, finding out the rest is much easier.

The TL;DR

Now that you are actually designing an app, avoid getting frustrated with all the early choices you've got to make by doing things fast and iteratively.

The Fidelity Problem

At this stage of designing an iOS app, you're at a crossroads. Do you invest in making a pixel accurate high-fidelity design in Sketch, Figma or something similar? Or, do you continue with what you have up to this point and move forward?

I call this the fidelity problem, and I've lived on both sides of it. How do you figure out which way to go? As we spoke about in the last chapter, gaining momentum and capturing it is critical. As such, do you invest the time into a high fidelity design, or do you find it as you go?

Whichever path you choose, it's mostly a matter of figuring out *when* you want to pay that time investment. Upfront, or later on? Going high fidelity takes a bit longer to move forward, but once you have it – it's incredibly useful. You pay the cost upfront. Going low fidelity means you fire up Xcode quicker, but you have to answer questions along the way which can be time consuming.

For many, you likely already know which you prefer. For example, I tend to fall on the "find it in code" method – wherein I take the wireframe sketch we created from the last chapter and start tweaking things from there. If I do use Sketch or Figma, I make a "low fidelity" version of the wireframe. It sounds odd, but it's basically a low fidelity version of a high fidelity design technique – but it works for me because it answers just enough (colors, font size, etc.) for me to keep moving.

Others I know create a fully realized document of the design, complete with color palettes, buttons for just about any scenario and more. They might even have a design system in place. They've thought through the entirety of the app, top to bottom.

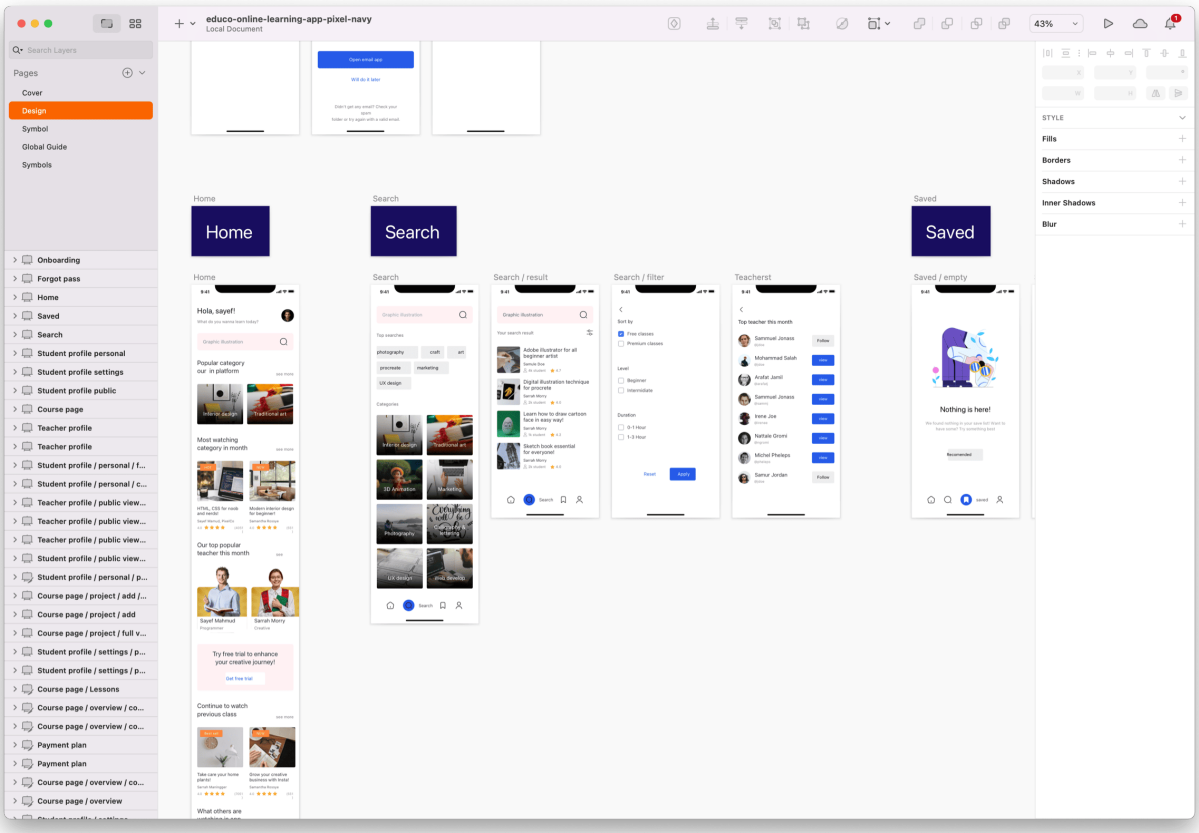
Of course, neither approach is wrong. It's about opening your eyes to both of them, though, and figuring out a way forward. A lot of times, we lean on our strengths at this point, too. Developers tend to dive right into code, whereas a designer at heart relishes the chance to create a high fidelity vision of the app before ever opening Xcode.

To help you make a decision, let's take a look at both approaches and uncover some pros and cons they carry with them. Keep an open mind here, as these are pros and cons as *I* have seen them. Depending on your personality or skills, you may have this list reversed. The point, of course, is that you think critically about them. Then, you can be realistic about either approach being a good one for you and your team.

Going High Fidelity

The allure of going high fidelity is obvious. A shiny, pixel perfect canvas laying out what to do, and where. I've worked with these in the past, and they are amazing to have around.

For the uninitiated, they look something like this:



A high fidelity design file in Sketch, showing all of the app's functionality.

For our purposes, I'm defining high fidelity designs as a document that has every flow, screen, button, color, copy and pretty much anything else defined and designed.

Pros

Less Guessing

This is probably the biggest benefit I've seen from high fidelity designs. A designer, or you, has already thought through some of the difficult questions that have come up during the design phase.

Flows are figured out, where a button goes (and *when* it goes there, if the design is a functional prototype) or how the text fits into this area or that area has been ad-

dressed. That's great, because those things are faster to figure out in a program built for design than it typically is to figure out in code (regardless of if you're using SwiftUI or UIKit).



When you have less guessing to do, it frees you up a bit more to get going on the code.

Faster Implementation

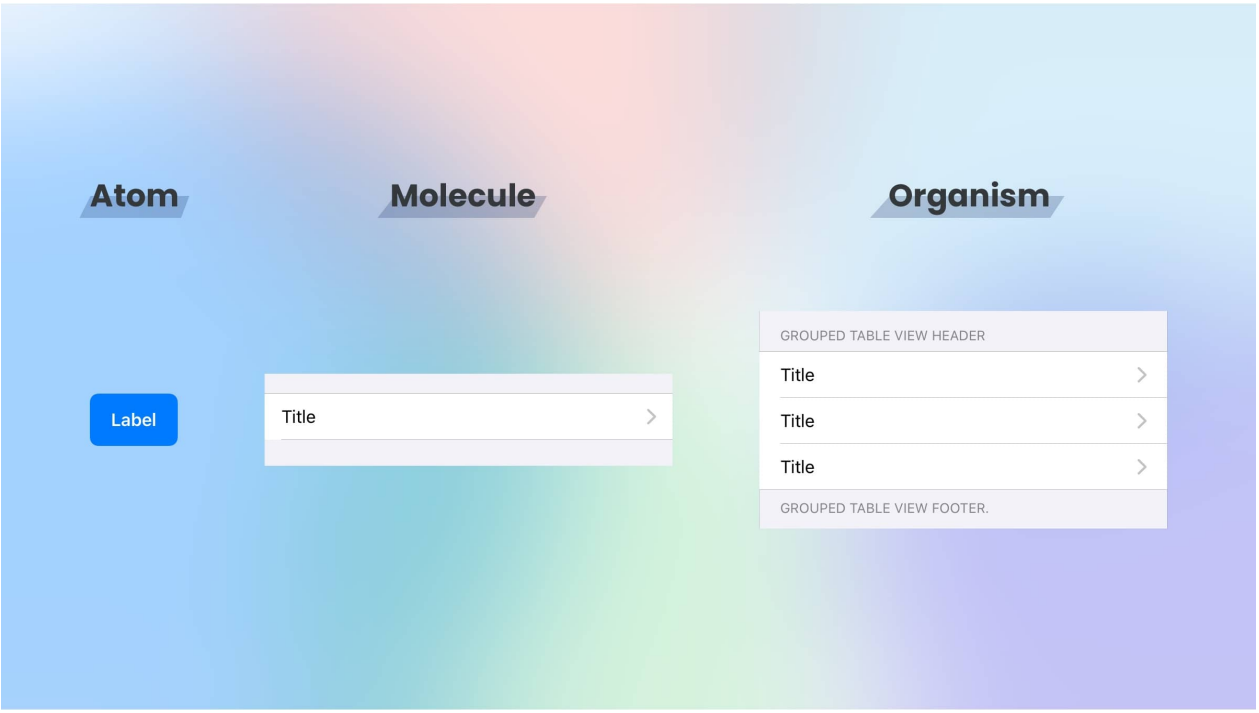
A high fidelity design is, in many ways, like a blueprint for developers. It tells them what to make. As such, it's a lot easier to reason about technical questions, too.

Programming, no matter how you architect anything, is never simple. It just isn't. Programming is hard, and it requires a lot of patience and thought – so any upper hand you can give yourself when you do it is a huge plus. That's what these designs can give you. When you have to work your left brain and right brain at the same time, it can be draining.

By virtue of the simple fact of knowing *what* you're building, how it looks, where a user goes when they tap X or Y – you can program for it easier.

Components are Predefined

If you utilize any sort of design system, a lot of components become modular. You have designs ready to go for buttons, sub types of buttons, colors and more. Using them, you can build up screens using all of these smaller components to add up to something bigger. In some communities, this is also called atomic design:



An example of atomic design elements, which progressively build up from one another to create screens in your designs.

This is great, because when a new screen arises (and they will), putting them together is a little easier. If you already have buttons, their font weights and colors, predefined – it means designing new screens is more about putting a bunch of individual pieces together than it is creating something from scratch.

Think of it like abstraction for design. Individual pieces and components with a specialized use case and purpose. When you have that, things like a settings screen became a fairly trivial affair to pull together.

Cons

Time Investment

High fidelity designs inherently take more time. They are well thought out, typically have a lot of production value and shoot to be a complete realization of the app.

That's perfectly fine — but be realistic that if you go this route yourself, you likely to need to put in a healthy chunk of time to do it right. All good things take time, that's no different in software development and design than it is in anything else in life, but if the time is a worthy investment to you — please go for it!

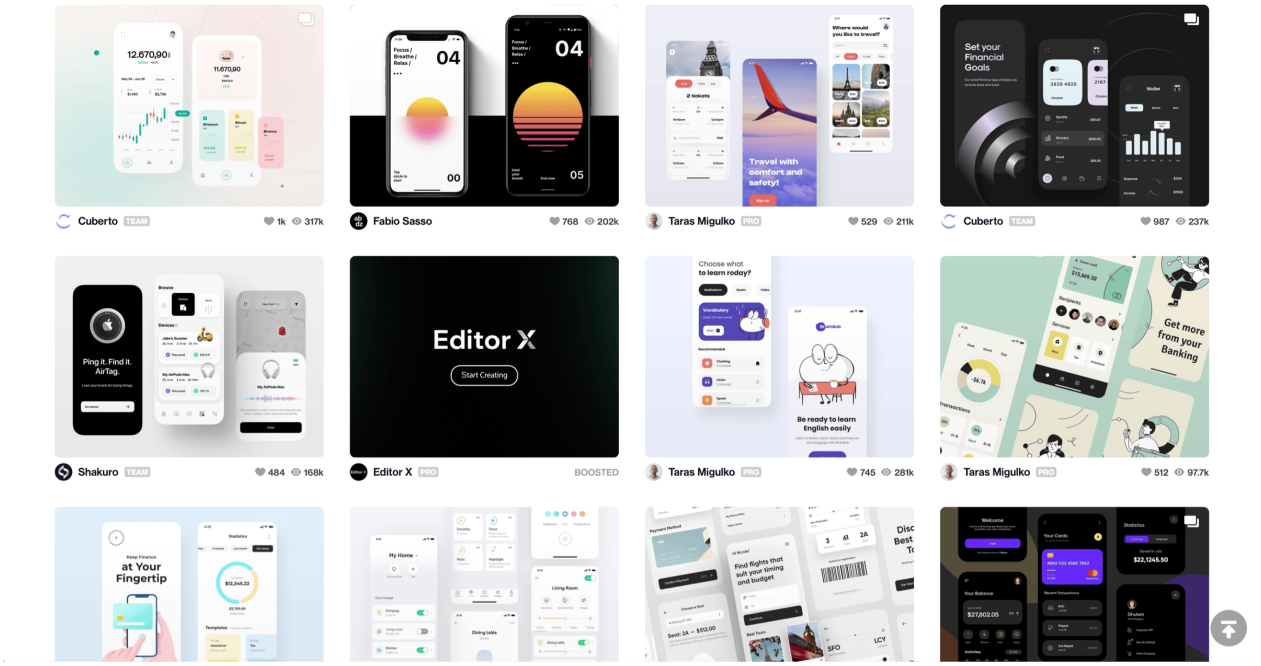
In fact, I'd argue that if you do well with high fidelity designs, then they *should* take quite a bit of time to make. It means you've thought about colors, active versus inactive states, flows and more.

Golden Path

One issue I frequently see with high fidelity designs is that they all tend to take a "golden path" approach. The data input is perfect for the interface, the text fits *just* right and the font size is matched up conveniently.

In reality, we know this hardly ever happens.

When making high fidelity designs, account for the real world. And, in the real world, folks rely on Dynamic Type, may have longer (or shorter!) names, their text may go right to left and the list goes on. For example, a quick search on the popular design website Dribbble for iOS designs yields several pretty, but potentially problematic designs for things like varying font sizes:



So if you see a high fidelity design like this, be sure to account for these things yourself or if you didn't make it, challenge the designer to consider these things too. Images will clip, things have to stack, interface axis often change, dark mode comes into play, accessibility has to be considered — the list is huge. High fidelity designs with data are great, but they also can embody the timeless saying of it's "too good to be true" because they don't often reflect the messy world of actual data entry from your users.

Going Low Fidelity

Pros

Instant Gratification

One of my favorite things about going low fidelity design is the instant gratification you get. You feel like things are moving and progress is being made. I think this is the same reason why many developers tend to start their side projects by either buying a domain name, creating a Github repository for it or making a new Xcode project.

It feels like you're going somewhere. And going somewhere is always good for a quick endorphin hit.

If you are the type that is motivated by moving and getting a project installed on your device right away, this method of design can work great. I personally love it for that same reason. Though, if you are wired primarily as a designer, it's also true that making a high fidelity design would give you much of the same feeling, too.

Design and Implementation are Inherently Married

When you do a lot of designing as you create your app, the two disciplines sort of become intertwined together. This can be a powerful thing because if you do it right – you can knock out two Herculean tasks at once: Design, and implementation.

This is one of the biggest draws of SwiftUI, as it aims to eliminate discussions such as:

- Should I use a storyboard?
- Should I make all of my interface in code?

SwiftUI gives you the best of both worlds. Visual feedback as you code your interface. Putting code and design close together can be powerful for those who are primarily programmers. I've been creating a lot of interfaces using UIKit programmatically for many years, and it's always been my favorite way to design things, too. It clicks with me more than design programs ever have – but that's an obvious bias I carry as someone who learned to program long before I ever thought about design.

Cons

Time Investment Can Be a Gamble

When you create a high fidelity design, you are pretty much signing up for the sizable time commitment required to produce a polished canvas full of the app's many states.

When you go low fidelity, you aren't immune from that time sink, either. In some cases, it can take *more* time. No matter your development setup, or user interface framework of choice, you are still bound to a simulator or device to fully realize any changes you make. With SwiftUI, you feel this burden less since you can sort of "design while you code" — but no matter, it still means you are putting code to compiler to figure out how things should look.

This can really take a lot of time, especially if a flow you're looking to design for is deep within the user experience of the app. If you're constantly tapping three buttons to bring up a screen and are bound to some technical constraints preventing any obvious shortcuts to get there quicker from being put into place, consider mixing things up with a high fidelity design for such a case.

Lack of References

The thought of having a design system upfront is comforting. All of your questions are answered! What color this text should have, what the background color of a button should be – it's all there. When you do things primarily in code, this isn't as true.

I've found myself referencing code files to figure out what my design system is. Depending on who you ask, this is either a very lean thing or a very bad idea. I've obviously listed this in the "cons" section, so it's no surprise where I fall on the issue.

If you're going to do a lot of design in code, make your life a bit easier by setting up a very loose design system somewhere. You don't want to have to boot up a simulator or open your app to know what color a section header should be.

Going Forward

I find myself doing a mix of both of these methods. For example, when I created Spend Stack I made a half finished version of it in Sketch. The final product was quite a bit different than what Sketch had. For me, this was great – the Sketch file served as a guideline of where I was going and I found the sweet spot during development.

This works for some people. For others, it's a disaster. You've also got to consider the team aspect, as well. High fidelity approaches are far more common in teams, but if you aren't working with others – you've got the freedom to figure out which works best for you.

Either way, I'm better off for having tried both approaches in earnest.

With the advent of SwiftUI, you can make the argument that it's nearly as fast, or faster, to create the designs in code. Depending on your skill level, this is a newer, but viable, approach to take.

As with any advice, do what's best for you – but stay open minded to all of the approaches. Perhaps try the one you *didn't* think would work for you. You will learn some things along the way, even if you don't stick with it going forward, that you can take to your design flow.

The TL;DR

At this point, you may have a fully polished design or you might have a "rough draft" of it – and either one is fine, depending on how you develop apps.

Finding Your App's Voice

Tone plays a critical role in your app. It gives the user an expectation of how your app will work, how it will feel and perhaps who it's for. Get it wrong, and users might be off-put or confused. Find the right voice, though, and users will feel connected to your app and share a sense of agency with it.

When we talk about an app's voice, we are talking about how the app primarily chooses to represent itself to the user. Based on that, several user experience and design choices will follow. An app's voice represents things like how copy is written, how vibrant or muted colors may be or how interactive the app feels.

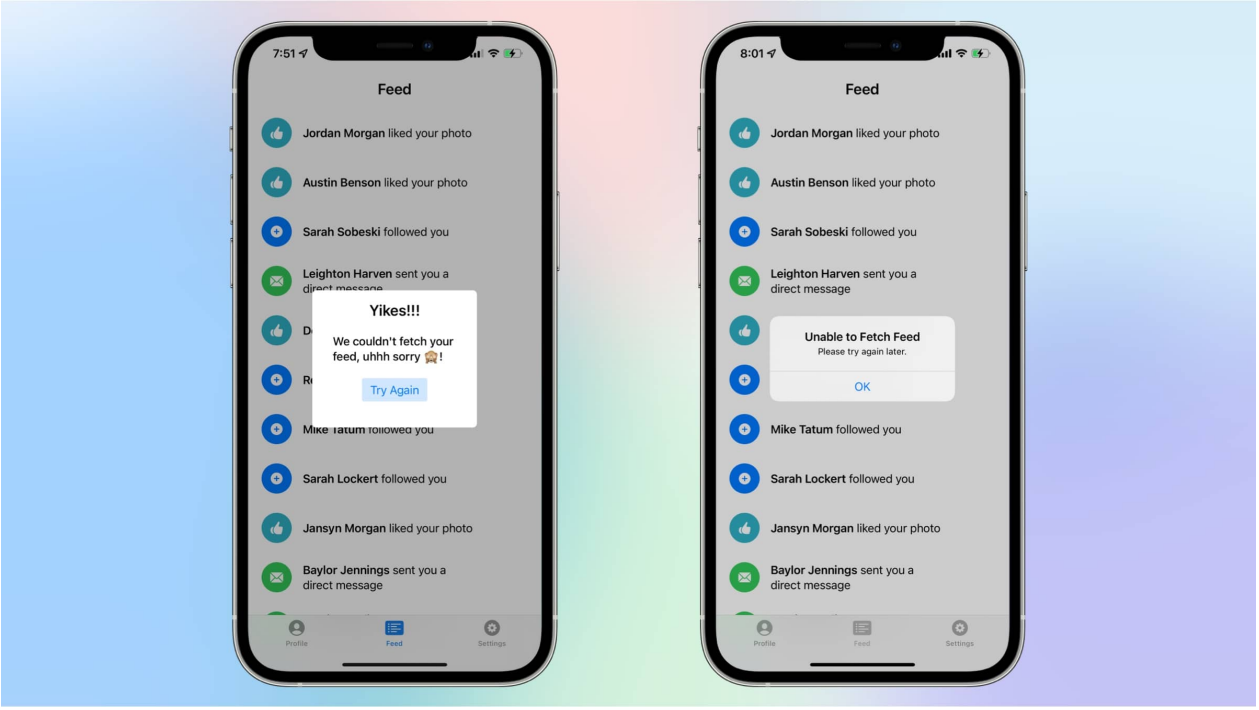
Holistically, it sets a tone for the app. Tone equals voice, and vice-versa.

Design offers several opportunities to break from convention and define new territory, but when it comes to your app's voice – it pays to display consistency. Settle on what your app's voice is early on, and go from there.

How Your App's Voice Shows Up

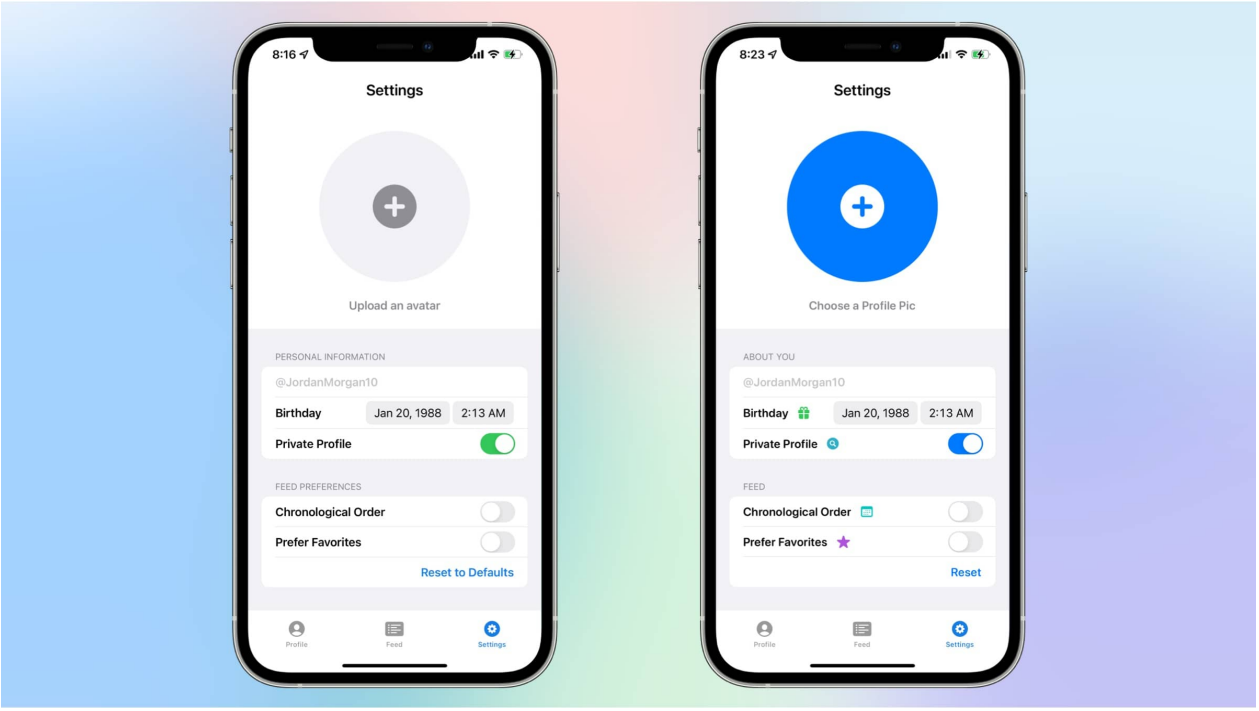
Be cognizant of the places your app displays any sort of tonality. You may be surprised at how often you have an opportunity to convey tone. It happens more than you think.

For example, look at these two error messages and ask yourself what's fundamentally different about them:



Functionally, there is no difference. They both display a network error. But yet they feel miles apart. Why? Their error copy is drastically different. One is silly, one is serious. They have different voices.

Going away from just copy, consider this next example:

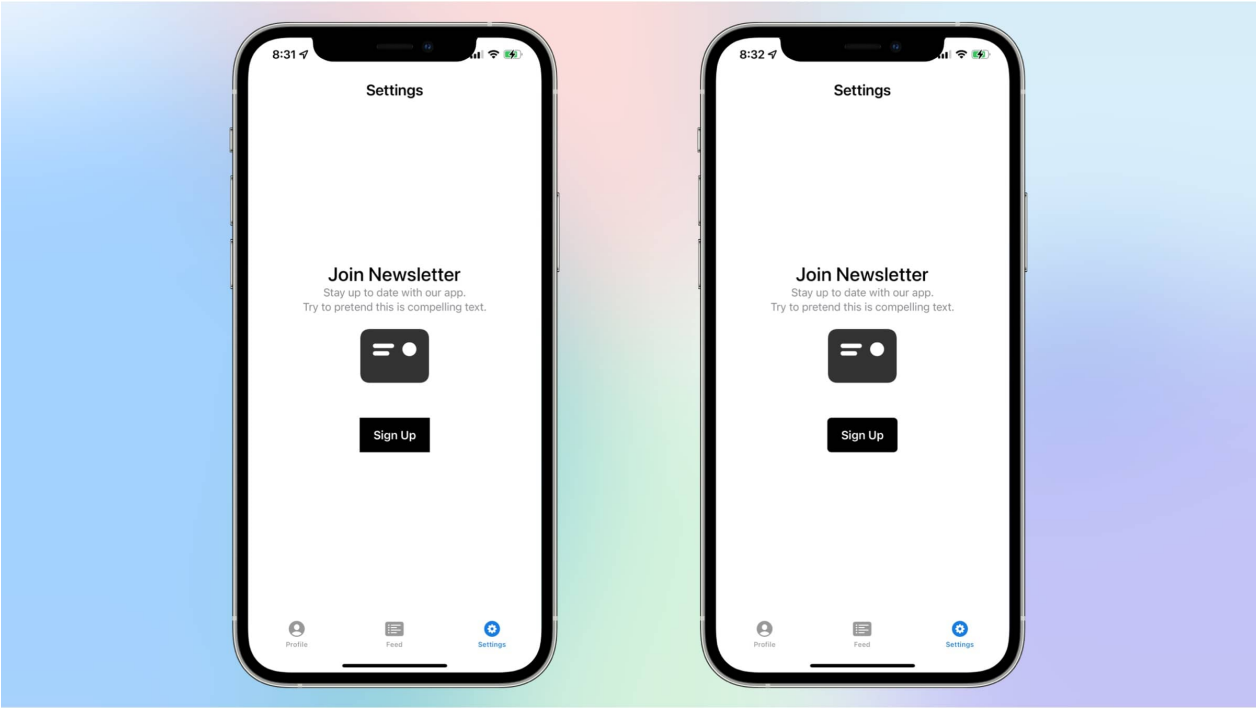


You get the same vibes as before, don't you? One feels like all business, while the other one feels playful and less serious. All that from simply changing colors, font and some text around.

Neither one is wrong. But they are different. And that difference is important, because deciding what your app's voice is drives decisions like these.

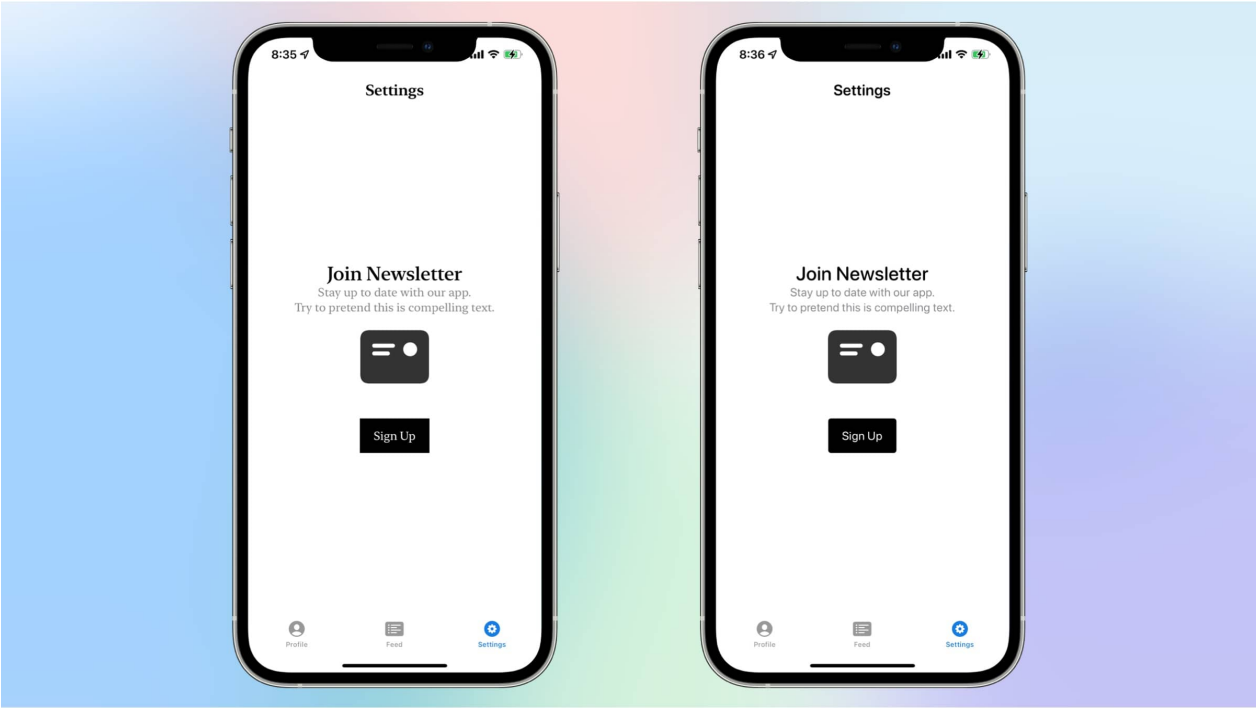
Colors. Typography. Copy. Navigation. We like to say these are functions of design, but I believe they are functions of an app's voice first. Because once you figure that out, the design comes next.

Let's look at some other examples. Buttons are a good one, look closely at the "Sign Up" buttons below:



The squared off edges of the first button convey a sense of urgency or seriousness. The rounded corner radius feels a little less dramatic in the other example. That's because tone is commonly bound to interface elements.

The same is true if we switch up the font and keep the colors, copy and other interface parts exactly the same. Here, the left uses a serif font while the right uses the one:



It all comes down to your goals, audience and app's mission statement. That's where you find your app's voice — and that's also why I recommend you do the things I've said in the order that I've said them. Skipping around, for me at least, leads to an unfocused process when it comes to design. But when you know what your app does, and who it's for and what that first version looks like — well then, you can settle on its voice.

And that's the best part, you get to choose it. So, what kinds of voices are out there? Let's take a look at some screens within apps that use a certain type of tone.

Types of Voices

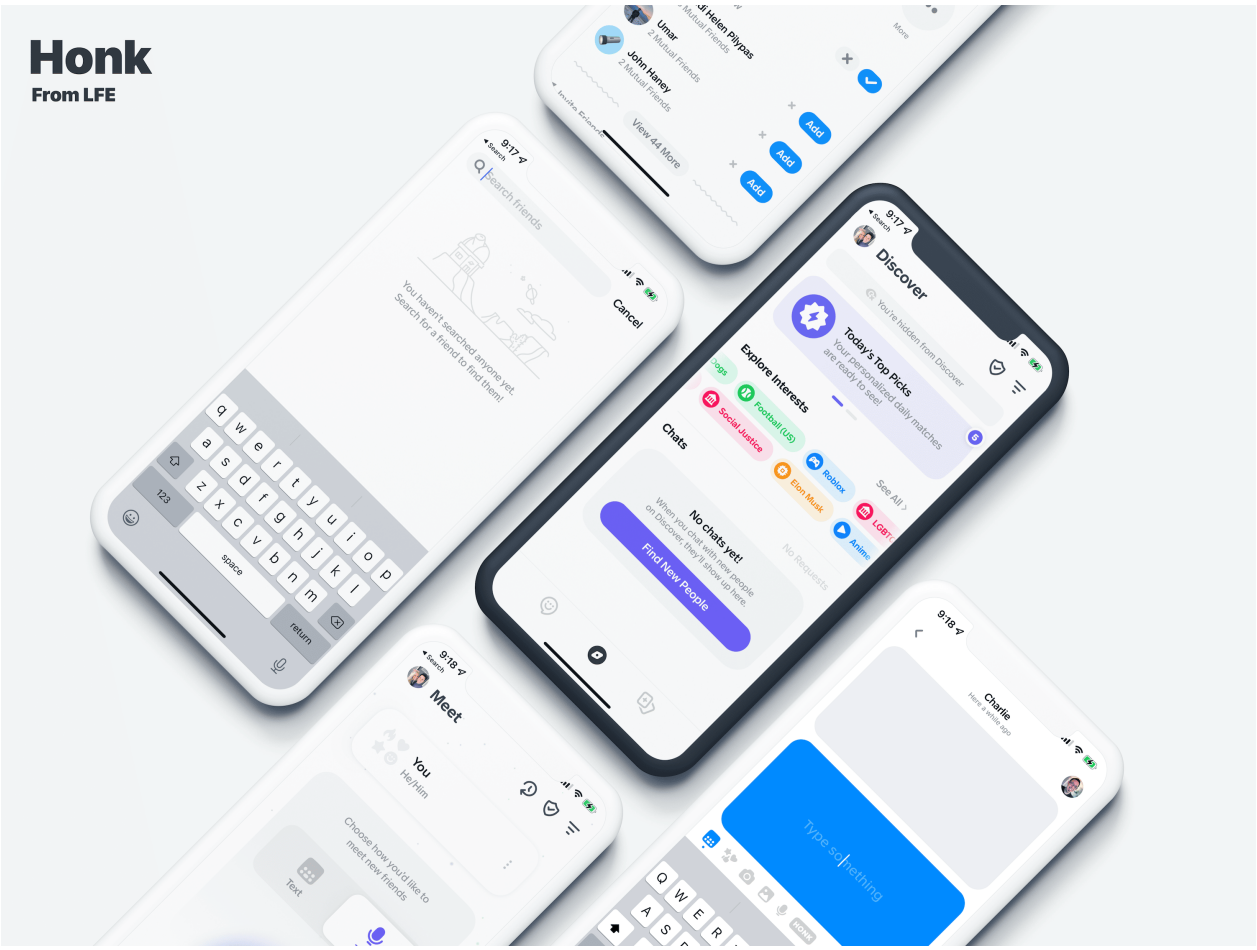
Casual

Calendar, by Apple, is a casual app used by, quite literally, billions across the globe. It strikes a casual tone that doesn't make too many opinions, being approachable to just about anyone:



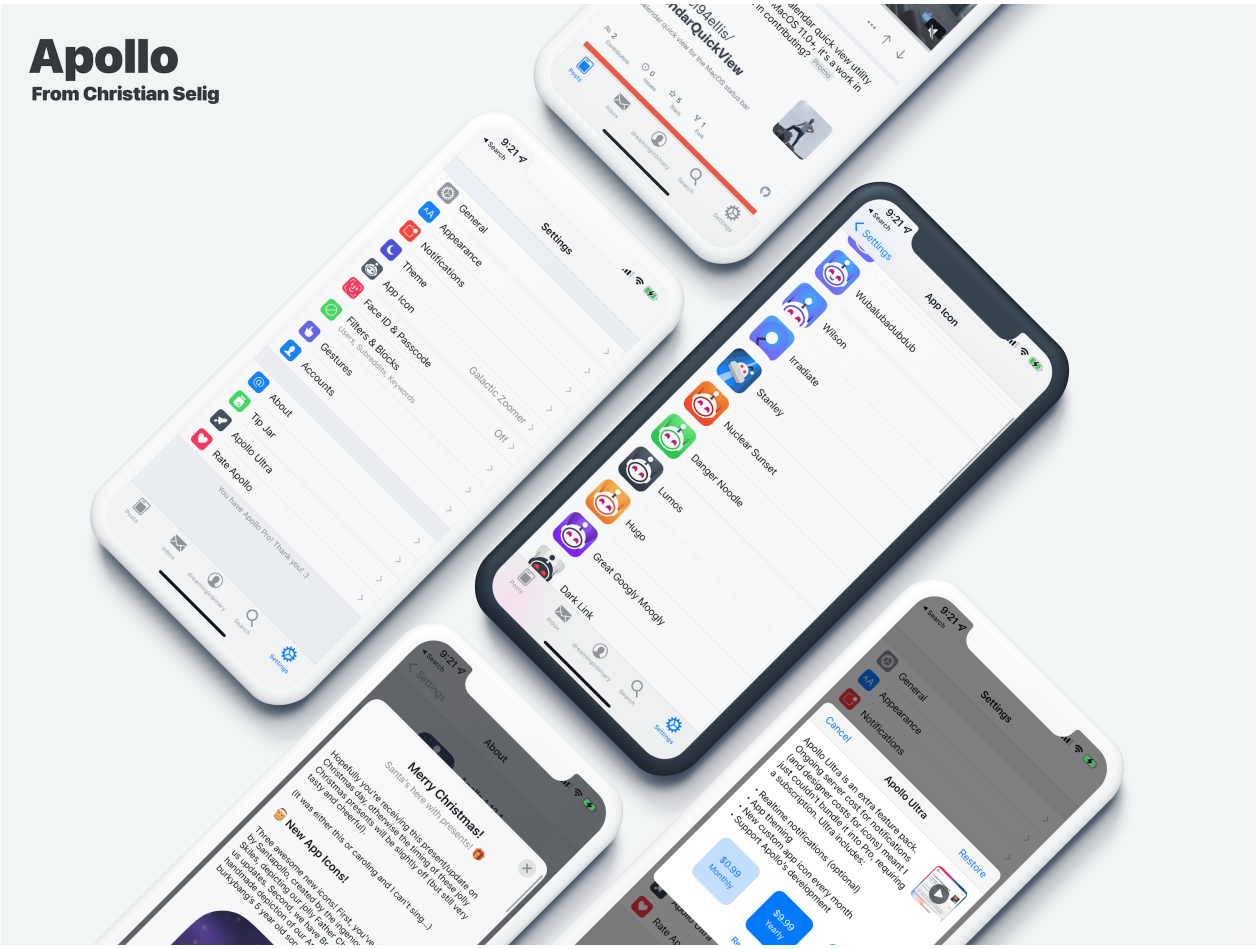
Playful

Honk, by LFE, is an incredibly playful app that invites fun and interactivity. It doesn't take itself too serious, but instead tries to enforce its goal of communication and meeting new friends:



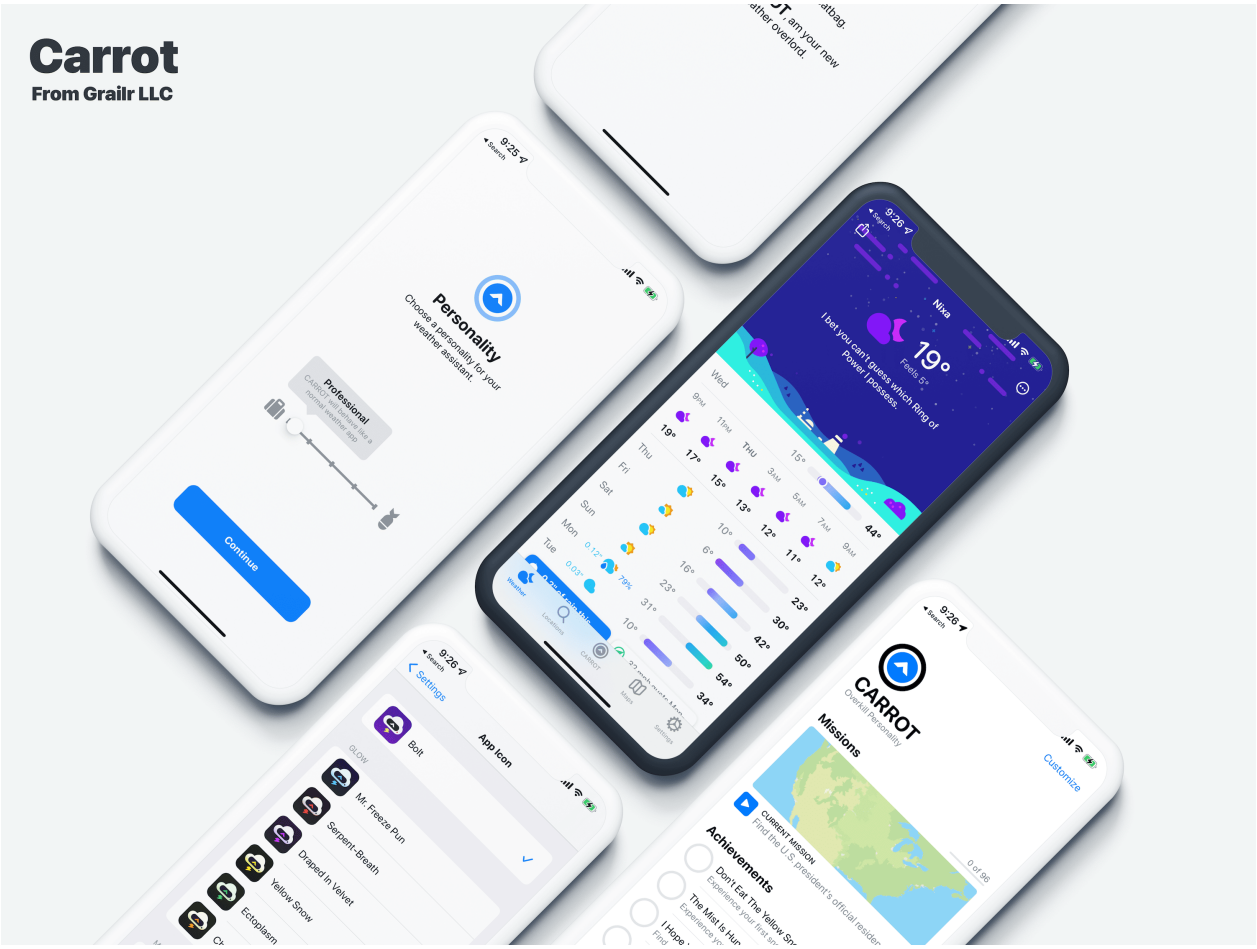
Friendly

Apollo , by Christian Selig, strikes an informal and friendly tone. It often refers to its developer in first person, and makes you feel like the both of you know each other.



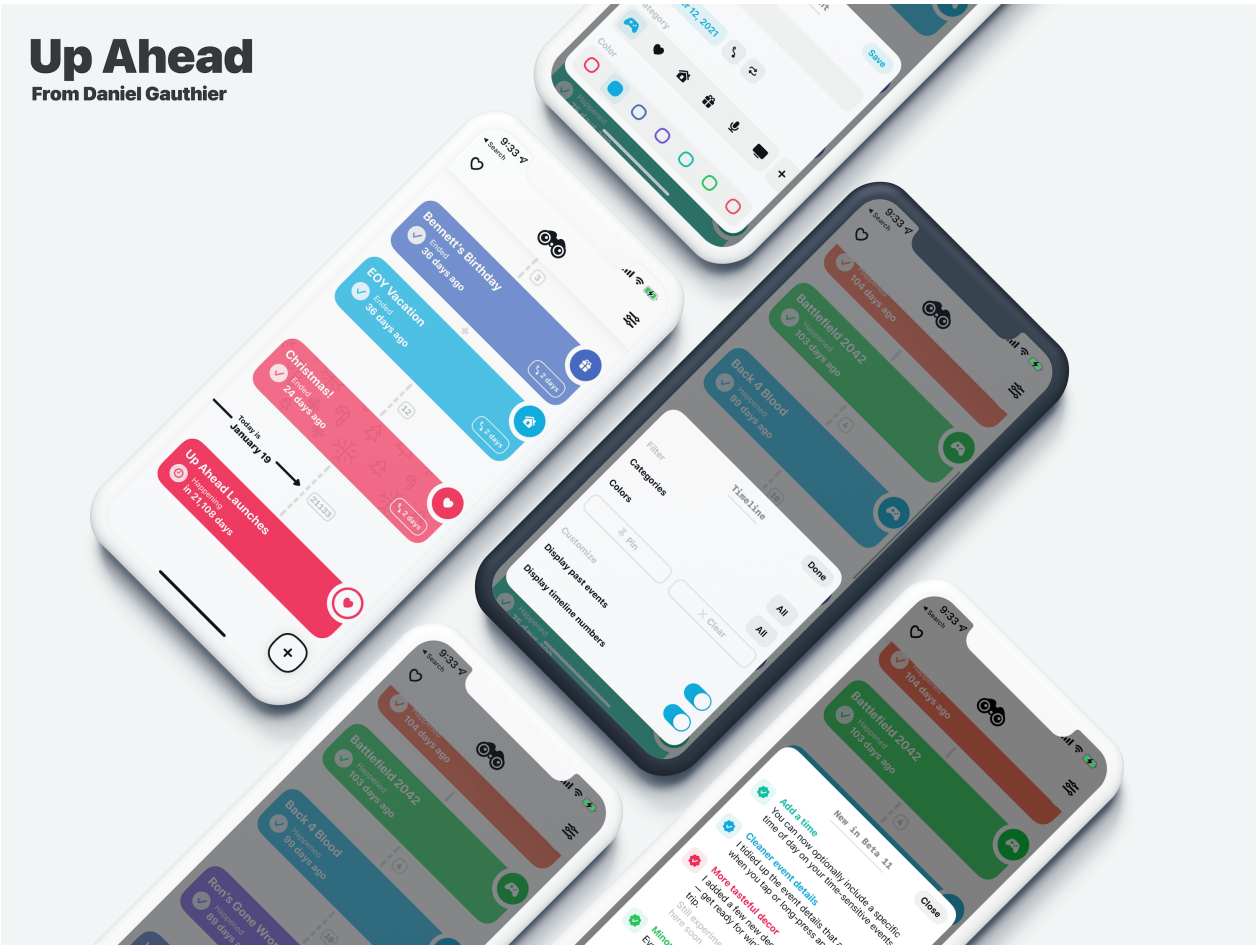
Sarcastic

Carrot, by Grailr LLC, has a unique tone in that it's configurable by the user. For example, users can opt for a sarcastic, insulting tone throughout the app that its titular character, Carrot, will use when speaking to you:



Lively

Up ahead, by Daniel Gauithier, uses a blend of bright colors and mono fonts to give off an energetic, personal feel. Its rounded corners and playful copy promote a loose, low-stress tone:



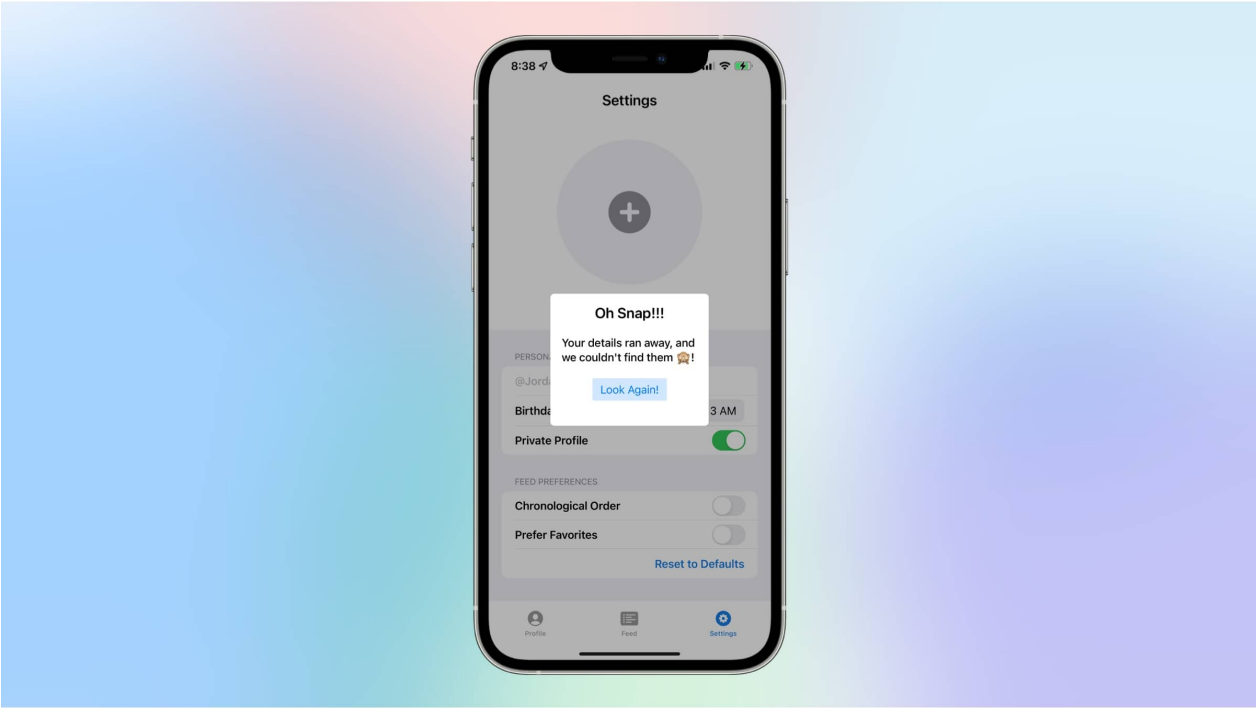
Something Else!

This list isn't exhaustive, of course, but it is demonstrative. There are lot of different ways to create a voice for yourself. But it's important that you decide what yours will be, and lean into it.

Choosing the Right Voice

Just as different covers of a song have a different feel, so can your app – regardless of its stated function and purpose. So, how do you know which one is right for you?

Making that decision is usually personal if it's your own app, or maybe it's been decided for you in a team setting. Whichever way it's formed, try your best to stick to it. You don't want to have something like this happen:



That has two tones competing for attention. The silly error message doesn't really fit the formal tone below it. Try to avoid using two different tones.

If you're in a position to decide, try to derive it from the information you've already discovered about your app. Who is it for? What does it do? And from there, ask yourself:

- How should it make those people *feel* when they use my app?
- How do I want them to talk to others about it?

And it's really as simple as that.

Find your app's voice, and use it right — it can end up being a competitive advantage for you. Maybe there's a cohort of "all business" apps that you can put a not-as-serious spin on. Or, maybe the opposite is true. Who knows?

But find a voice you're excited about, and breath it into all of the other design decisions you make.

The TL;DR

Every app has a voice, and the sooner you find yours – the easier it becomes to make decisions going forward and enforce consistency.

Navigation

Navigation, in terms of iOS apps, is a topic that doesn't get much attention. You're not likely to find many books, blog posts or conference talks covering navigation. In fact, the topic itself is sort of….boring?

It's fair to say. Navigation in iOS really hasn't changed much since iOS 1, which was actually not even called iOS at that point (it was "iPhoneOS"). For example, the home screen's navigational structure is still intact from iPhone OS 1:



So, when *do* you hear about it? Well, you hear about it when one thing is true:

When it's bad. Confusing. When it simply doesn't work.

Navigation, inherently, should be a background role. It epitomizes a part of your app that shouldn't garner much attention from users. If people are talking about it or thinking about it – you've taken a wrong turn (terrible pun intended).

Let's talk about how to get it right. To begin, let's discover the types of navigation that iOS primarily relies on.

Types of Navigation

When implementing navigation in your own apps, there are three primary routes that Apple recommends:

1. Hierarchical.
2. Flat.

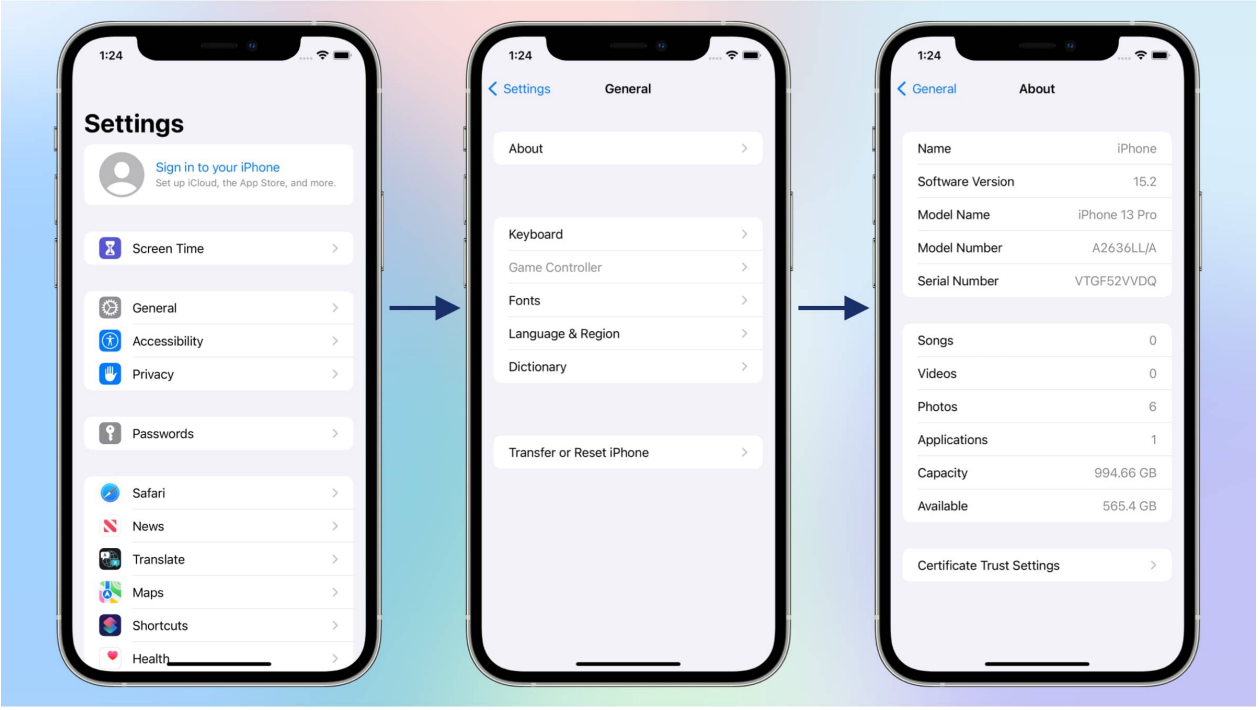
3. Content or Experience Driven.

I would also add to that a fourth option, which is some sort of combination of the first two.

Hierarchical

This type of navigation is likely the most common. It's composed of a single route per destination. Think of it like a path of breadcrumbs. To get to where you were, you have to retrace your steps.

Several stock apps use this method. Look at Settings:

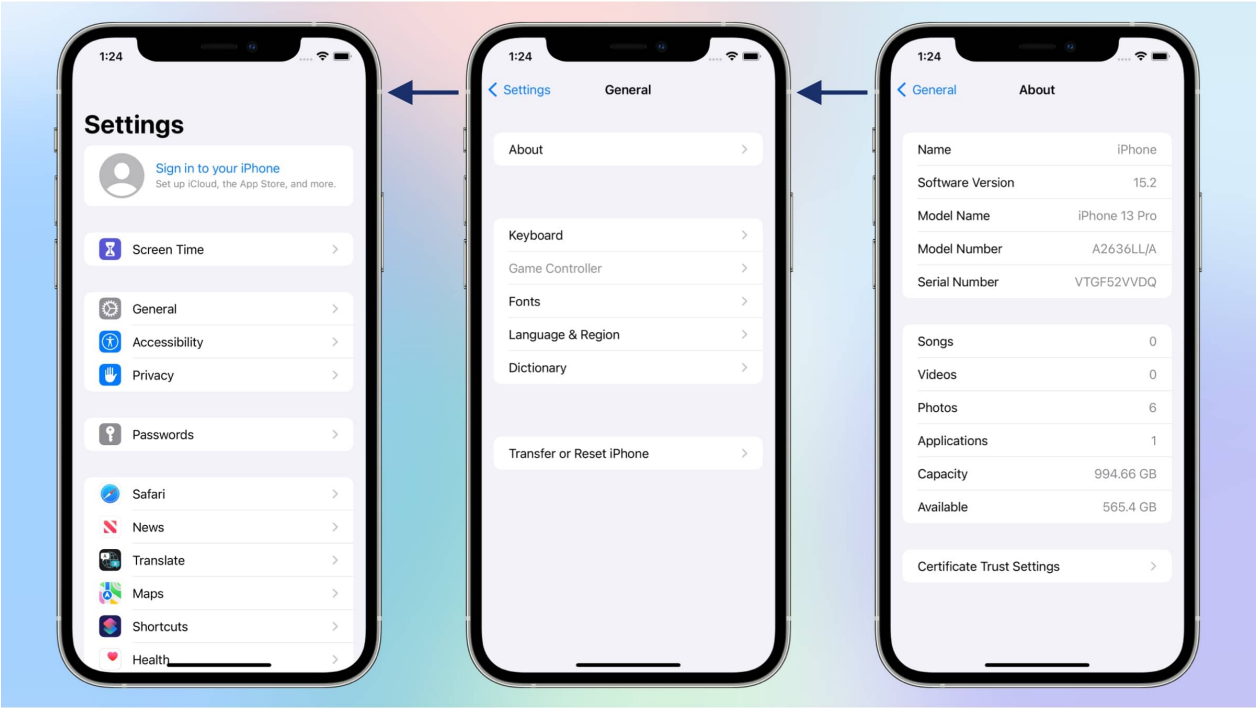


Settings allows you to "drill down" to other sections of content.

Navigation controllers are ubiquitous with this navigational flow – you drill down further until you get to where you want to be. It's orderly and understandable, though it's also the most time consuming. There aren't many affordances you can make for short-

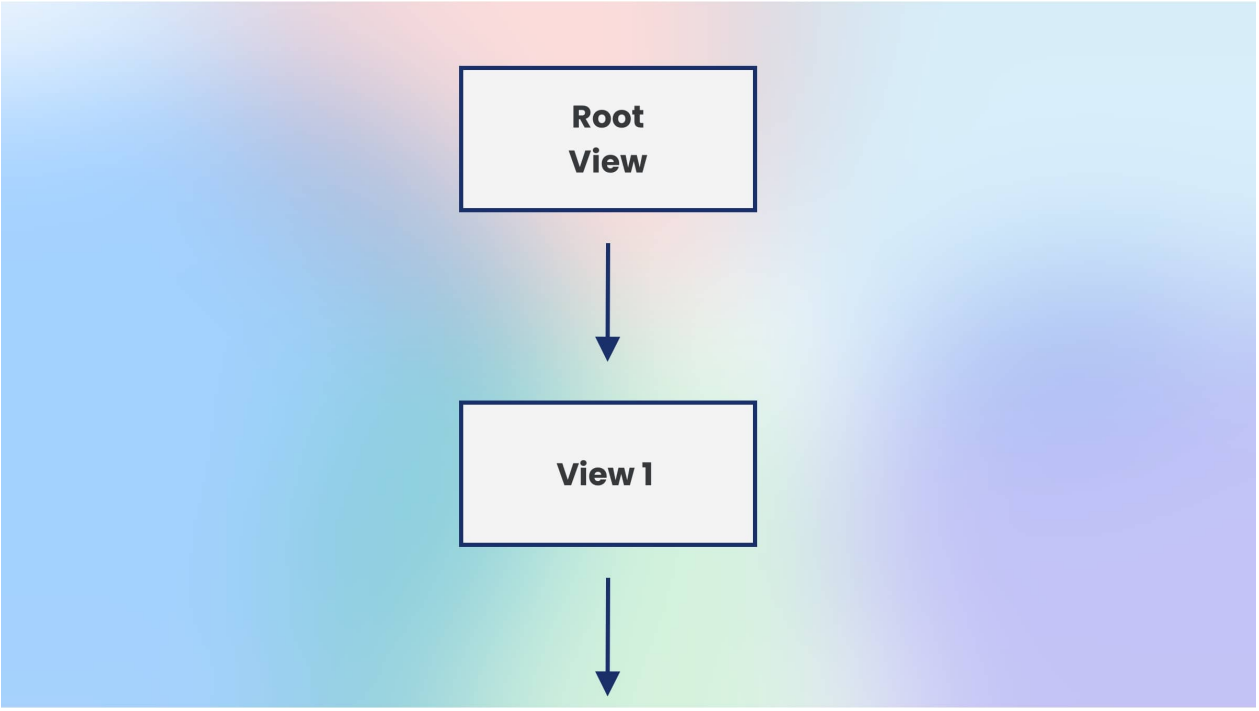
cuts, but that's by design. Each step a user takes is purposeful and, more importantly, repeatable.

They can back out, and start over. There's no ambiguity on how to do it, either.



The back buttons in Settings let you know how to go back to exactly where you were.

Visualizing this is fairly intuitive, too. Have you seen an "Org Chart" before? Those charts which show a company's C.E.O., who reports to them and so on. Your app work can be shown in much the same way, replacing titles and names with screens users can go to:



This type of navigation is the most prominent on iOS. It's a great choice for many apps whose primary content is of one central type. Settings is just that – a collection of settings. Mail, the stock iOS app, works the same way. No matter what you do or where you go, you're dealing with mail. Nothing else.

Flat

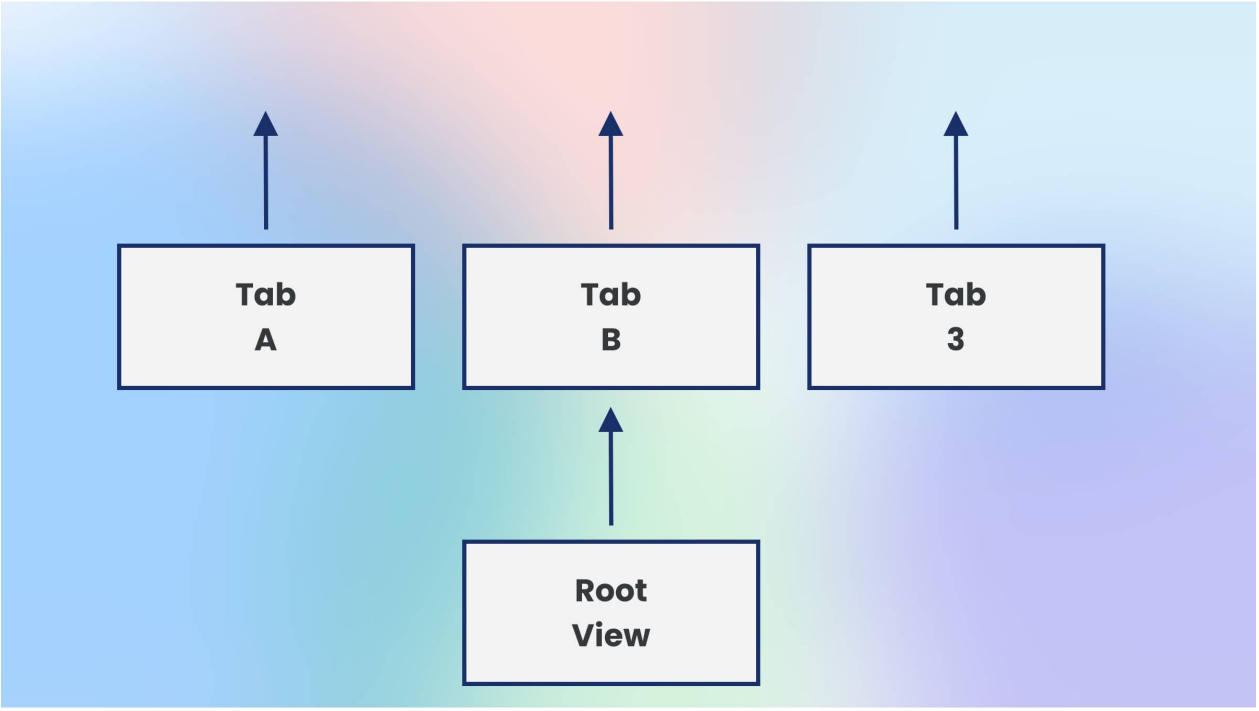
Flat navigation is another marquee option for navigation on iOS. Opting for flat navigation means you've got purposeful, unique content categories your app contains. News, for example, is divided up into five distinct "experiences" within the app on iOS 15.3:

1. Today
2. News+
3. Audio
4. Following

5. Search

It's easy to see why flat navigation shines when you've got specific content to choose between. They are different paths within your app that wouldn't make much much sense to bunch together. And even if it did, navigating between them all would be cumbersome. In fact, their only connecting string might be "they are in the same app", and that's it.

It wouldn't make much sense to use hierarchical navigation here, much the same way it wouldn't make sense to use flat navigation if you only had one central content category within an app. Going back to our "org chart" example, flat navigation might look more like this:

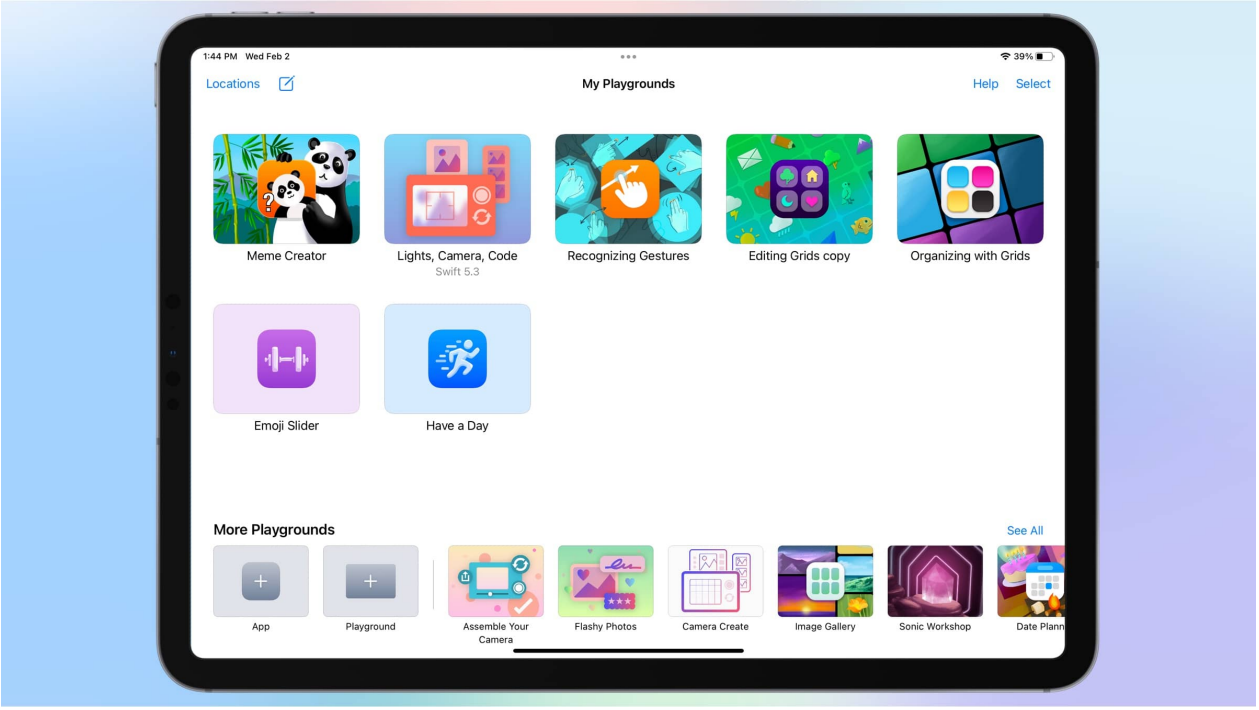


You'll see flat navigation typically implemented using tab bars, where each tab represents a path to take within your app.

Context or Experience Driven

This last one is a bit more abstract, and you won't find it in many stock apps from Apple. The idea here is that the experience itself dictates the navigation. If you're making a game, an immersive magazine or a learning app for kids – you might find the two prior means of navigation limiting.

Swift Playground uses this type of navigation:

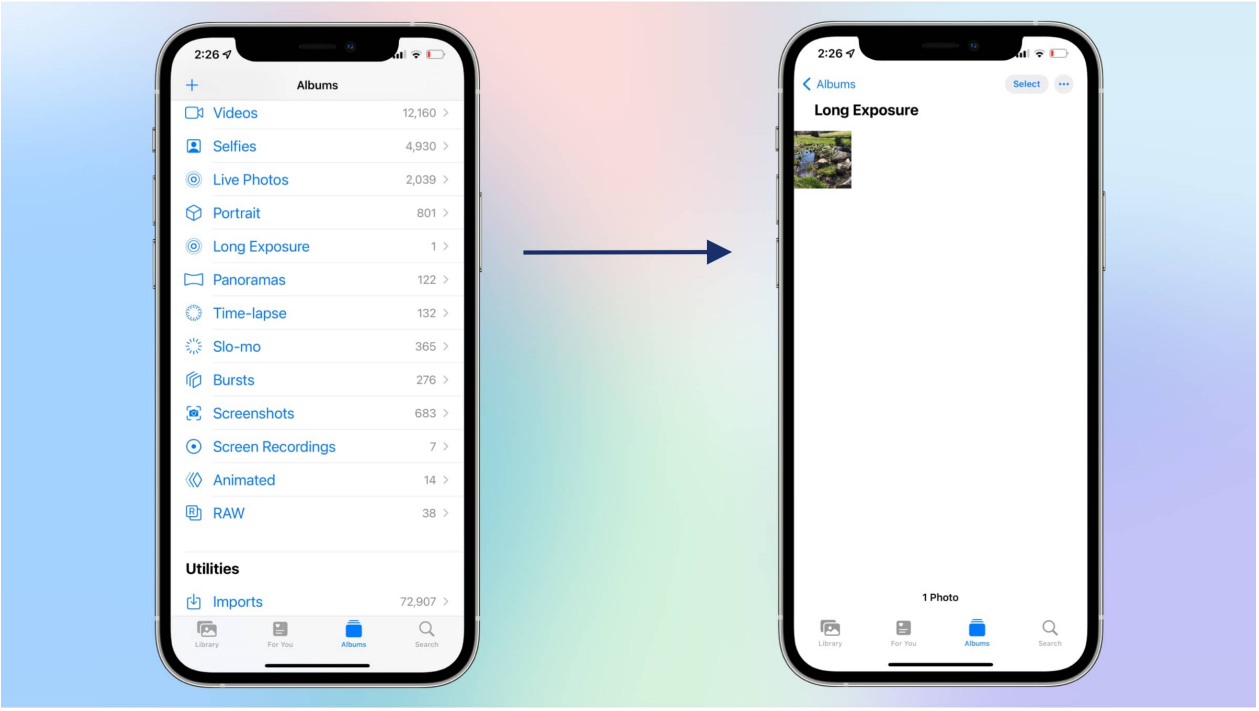


So here, it's less about "here's what's in my app, and here is how we get there" and more of "you discover parts of the app and its content along the way, so I'm not going to reveal it all to you at once."

The org chart example doesn't really hold here, but if there were a chart for it, it might look a bunch of circles with arrows taking several detours back and forth. That doesn't mean it's confusing, it just means it's more dynamic and contextual.

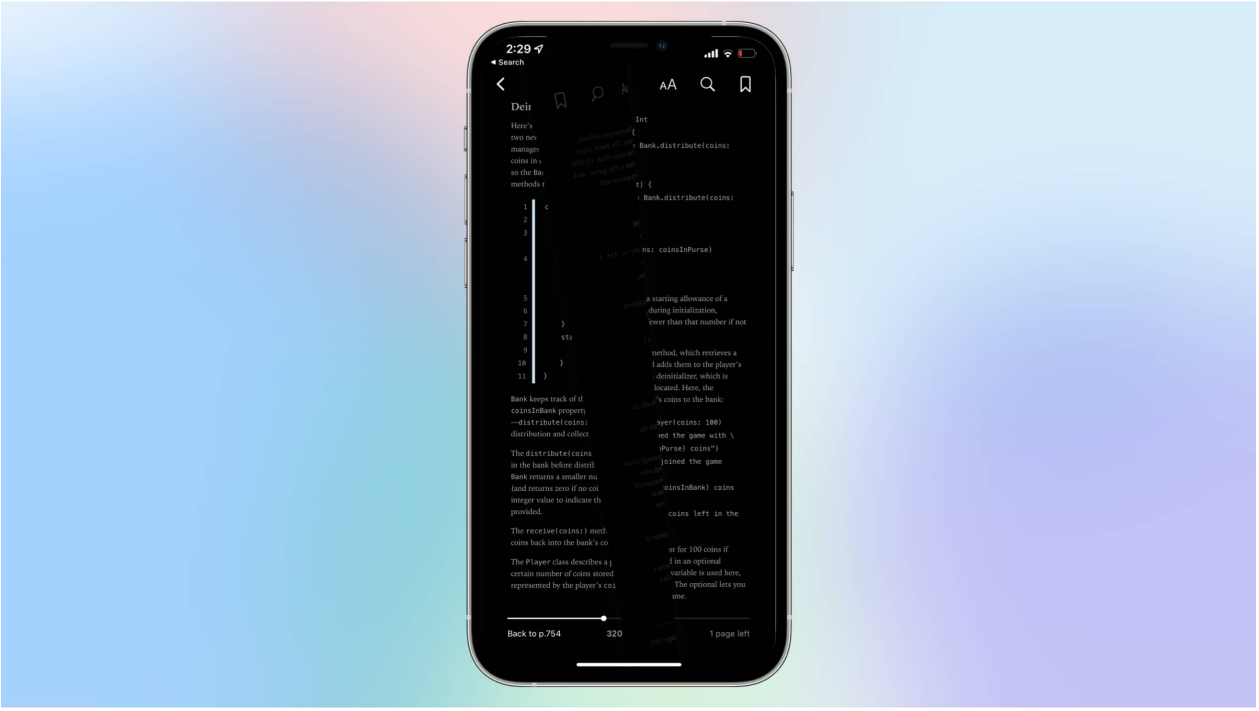
A Combination

In theory, these are the types of navigation you'll shoot for. In practice, you're likely to combine some of them together — and that's expected. For example, if you've looked at your app and decided that it's a fit for flat navigation, you can still implement hierarchical navigation within one of the tabs:



In Photos, you can see both flat and hierarchical navigation working together.

Or, perhaps you might work on an app that shows magazines, books or some other readable content. Those might work well with a page control:



In Books, you can see a page flip animation presents the new view.

Whichever way you go, make these goals your North Star targets for navigation in your app:

1. No guesswork. If users get lost, things have to be made simpler — full stop.
2. Clarity over everything. This means useful titles in navigation bars, and copy that reinforces where users "are" in your app.
3. Getting back is easy. To do this, there should always be a clear way out or "exit" at any point.

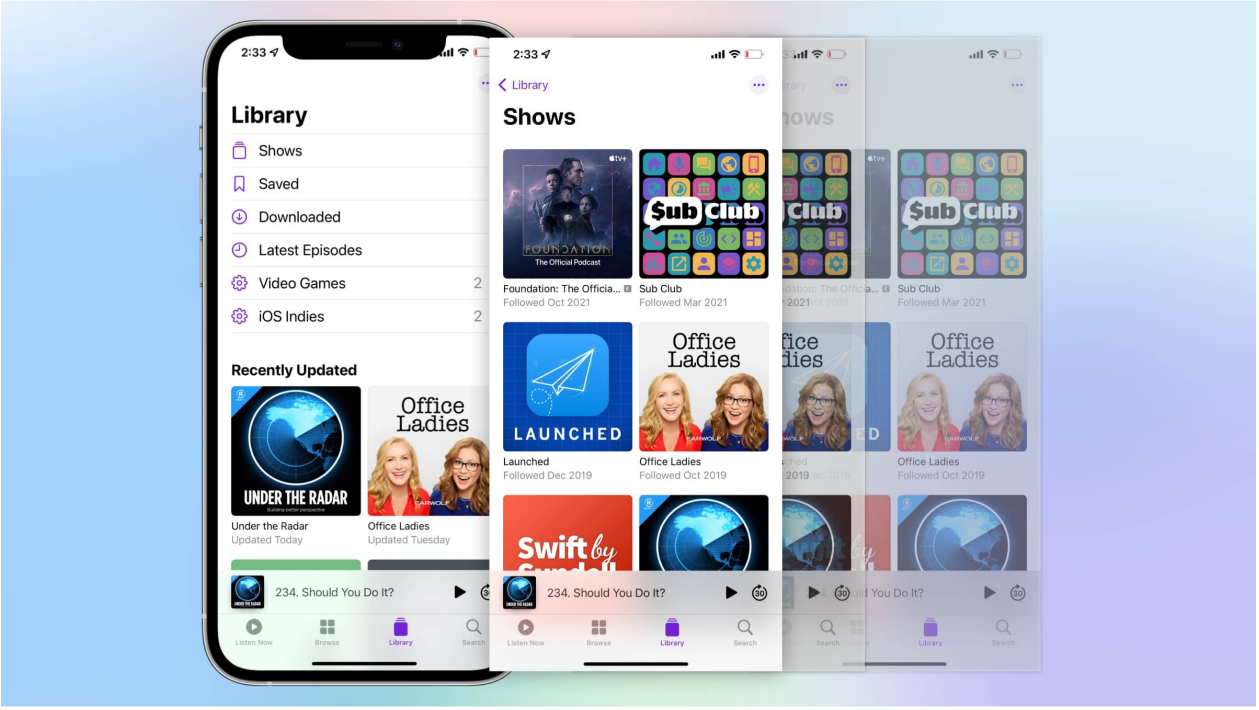
Things to Avoid

When navigation doesn't work, you feel it. We've established that. So, how can we avoid doing things that make it feel that way? Answering that is, thankfully, easy — sim-

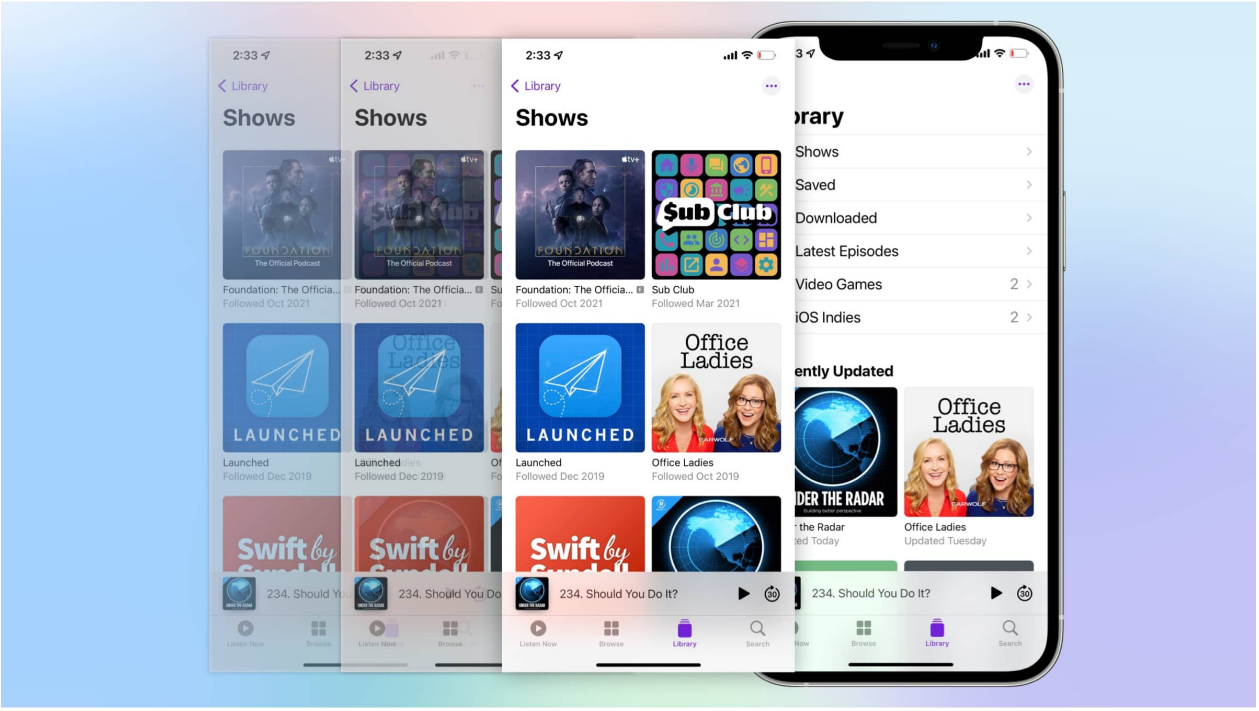
ply stick to the rules and design of the controls that drive your navigational method. Use navigation bars as they should be used, use tab bars as they were designed. That's how easy it is to *not* screw up navigation.

To drive this home a bit more, when using hierarchical navigation, you'd want to make sure you're pushing and popping content the right way. So that means in left-to-right locales – content is pushed onto the navigation stack by sliding in from the right edge of the screen.

For example, in Podcasts – if I tap on "Show", the navigation controller pushes the "Shows" view onto the stack by sliding it in from the right hand side:



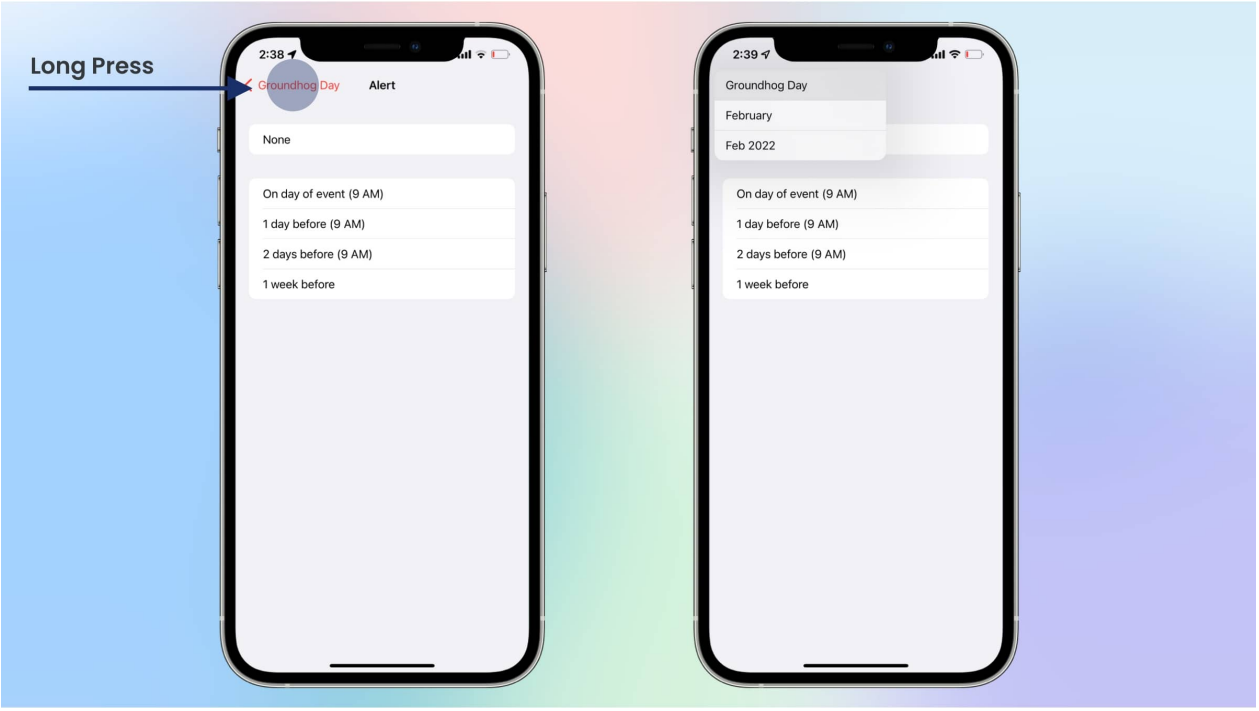
Where things would feel entirely off, though, were if you tapped on the "Shows" row and the content slid in the from the *left*:



If you use stock iOS controls, it's nearly impossible for this to happen. Which is why using stock controls from UIKit and SwiftUI is the most fundamental and important tip for navigation on iOS. By doing so, Apple is saving you from yourself.

Don't reinvent the wheel when it comes to navigation. It doesn't have to be rethought — in fact, users don't want it to be. They just want to get to your app's content, not learn how to walk (digitally speaking) again.

Plus, if you didn't use those controls, you're likely to be found out quickly. When iOS introduced the ability to long press on the back button in a navigation bar to choose a view to pop too - everyone who used that got it "for free":



The same goes for our locale discussions. In regions where users read left-to-right, navigation bars push the opposite way, too. Don't skip these APIs, navigation should be boring and "just work" – and the best way to make that happen is to use the tools as is.

Tips

Consider Revisiting Navigation on iPadOS

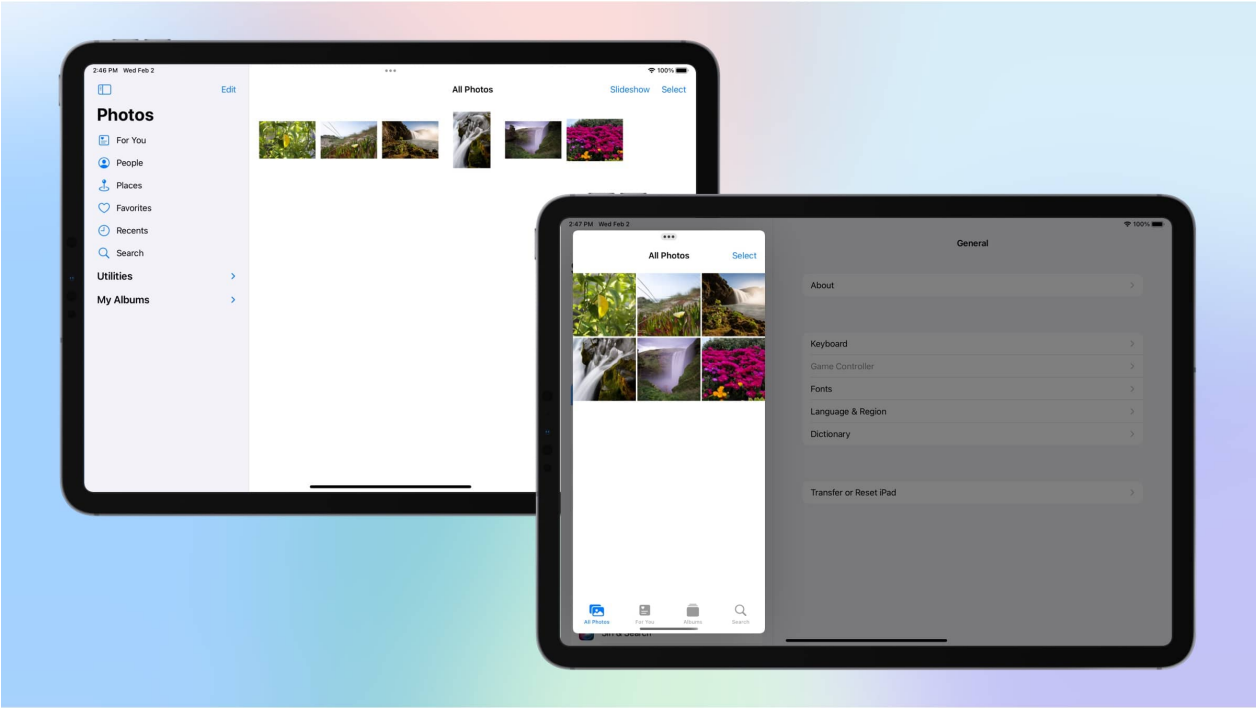
The same navigational structure that works on iOS may not work quite as well on iPadOS. This makes sense and is obvious when you think about it – on average, the screen release estate is much larger even when you consider the smallest possible delta between the two platforms (i.e. the largest iPhone versus the smallest iPad).

If that's the case – why do so many apps keep the same structure? Architecturally, this used to be hard to manage, but recent APIs have made things more realistic. Size classes, auto layout, split view controller enhancements and SwiftUI itself make the task easier to manage.

For example, a split view controller in UIKit allows you to assign a controller to show for certain size classes:

```
let splitVc = UISplitViewController()

let masterVC = UIViewController()
let suppVC = UIViewController()
let detailVC = UIViewController()

// For regular size
splitVC.setViewController(masterVC, for: .primary)
splitVC.setViewController(suppVC, for: . supplementary)
splitVC.setViewController(detailVC, for: .secondary)

// For compact size
let tabVC = UITabBarController()
splitVC.setViewController(tabVC, for: .compact)
```

For a more practical example, consider a flat navigation structure on iPhone or compact size classes versus iPad. With all of that extra space, a split view makes more sense on iPad *and* it still produces the same flat navigation, whereas in compact size classes the tab bar works better:



One last aside here – I've been saying "maybe rethink things on iPad" but what that *doesn't* mean is to do things like this:

```
if UIDevice.current.userInterfaceIdiom == .pad {
    // Do different nav
}
```

That's almost never what you want. Instead, you most likely want to lean on size classes and think less about the device itself for logic:

```
if traitCollection.horizontalSizeClass == .regular {
    // Do different nav
}
```

In SwiftUI, an environment variable will report the size class for you:

```
struct RootView: View {
    @Environment(\.horizontalSizeClass) var hSizeClass

    var body: some View {
```

```
            if hSizeClass == .compact {
                TabView()
            } else {
                SplitView()
            }
        }
}
```

The reason being is that, on iPad, if your app is being used in multitasking as the small slide over app, you could still fire off your logic for navigation meant for much larger screens. The API isn't perfect, and sometimes you'll need to do some fine-tuning – but don't assume an "isOniPad" check is enough.

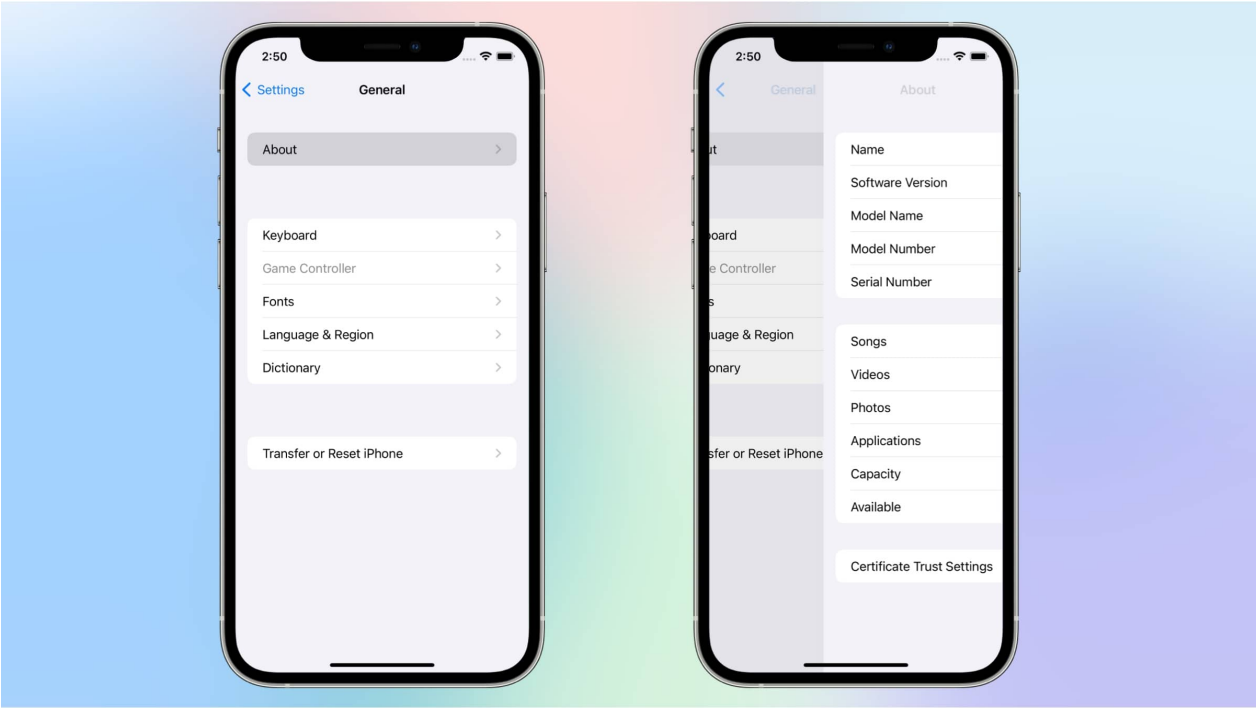
Much the same way websites adhere to "responsive" design, it pays to think the same way about iPadOS. Your app "stretched" to fill the space is a baseline effort for an iPadOS app.

Persist Selections in Hierarchical Navigation

Part of effective navigation is persisting that "breadcrumb" we talked about earlier. It's the small things that add up here. Apply a "leave things how you left them" type of thinking to your interface.

For example, if you push a view onto a navigation stack from a table view, collection view or List selection – you likely show a highlight on that row, cell or whichever control is applicable when the user taps on it:.

Here, I took a screenshot right when I tapped on "About", and another when I was going back to the "General" screen, and you can see the highlight state persisted and animates away as I come back:



As you saw above, you don't want to outright deselect that row when they push another view on the navigation stack. Instead, have it deselect when the user pops the view off of the navigation stack. Otherwise, you're effectively erasing a breadcrumb if you remove the highlight right after they tap on it. This is hard to show in text, but if you open up the Settings app – tap any option. Then, slide backwards using the gesture to go to the last view where you just came from, and you'll notice the row deselect its highlighted state as you do.

If you're using a table or collection view controller or a List in SwiftUI, this should happen for you. If you added it to your hierarchy yourself, you've got to manage it. For example, using a table view, it would look something like this (this is taken from the demo sample project included with the book):

```
extension UIViewController {
    func animateDeselectionOfSelectedRow(in tableView:UITableView) {
        guard let selectedIndexPath =
tableView.indexPathForSelectedRow,
```

```
            let coordinator = self.transitionCoordinator else
{ return }

        coordinator.animate { context in
            tableView.deselectRow(at: selectedIndexPath, animated:
true)
        } completion: { context in
            if context.isCancelled {
                tableView.selectRow(at: selectedIndexPath, animated:
true, scrollPosition: .none)
            }
        }
    }
}
```

Be on the lookout for similar situations like this - and make sure the user has the context of *where* they came from just as much as they do where they're going.

Choosing your Navigational Means

Throughout this chapter, you might've noticed that you derive your navigational structure from your content. I think that feels right, and it's how I end up making my decisions for it, too.

So, when you're considering which way to go – ask yourself how different or unique each chunk of content that you'll show is or isn't. If you end up recognizing several unique experiences, you're likely looking at flat navigation.

The opposite is also true. If things are centralized and all of the ideas work to build off of one another, that typically calls for hierarchical navigation. And remember, you may end up using a combination of both.

The last route, context or experience driven, should be fairly obvious to choose. They fit for apps which are games, an I.D.E. or something else along those lines – not your "typical" app.

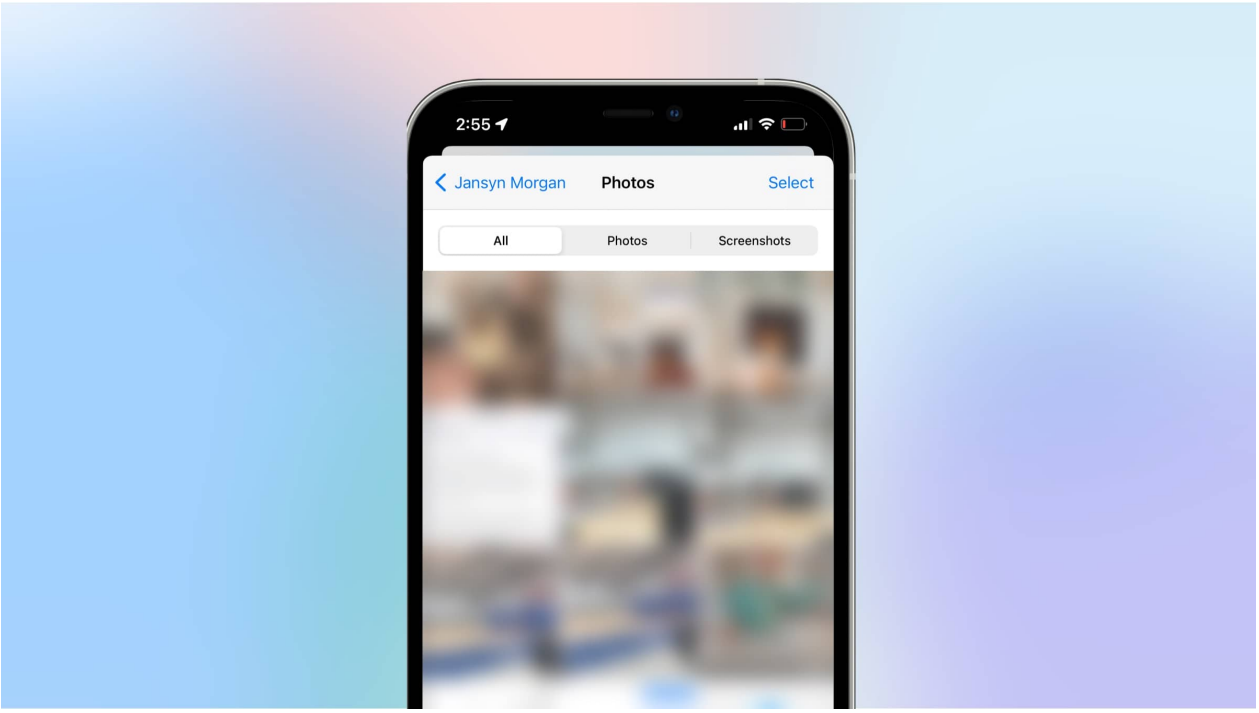
Don't use Right Chevrons for Modal Presentations

If the result of tapping a row or button in a list *isn't* to push another view onto a navigation stack, don't use the leading arrow chevron icon. The expectation from this glyph is a push, not a modal presentation.

A Note on Segment and Toolbars Controls

While segment and toolbar controls may look like they enable navigation, they aren't meant for that. Instead, lean on them for organizing, filtering and interacting data.

For segmented controls, you're saying to the user "You can switch some data about the *current context* in and out" — whether that means filtering the same view, or showing something different, is up to you:



For toolbars, they are used to extend interactivity with the current context of controls. They should help perform actions that only make sense for the view that's showing.

This advice can walk a thin line, I admit, because if you want to get technical — I guess they *could* produce some form of navigation via modally presented views.

But, again, this is a result of an action of the current view. As long as you don't rely on a toolbar for navigation **and** you're using them to pronounce the most common actions in the view that's showing, you're likely on the right path.

The TL;DR

An app's navigation should always take a back seat and just work, and if doesn't – you're likely to hear about it quickly.

Bars

Bars play a front and center role within iOS design. They are used to segment information, provide structure to interface elements and enforce a visual hierarchy. Their purposes are defined and rigid – but that's okay. Bars offer many affordances within iOS design, and they play several roles within the ecosystem.

The point is, no matter the need you have for *some* sort of bar, iOS probably has it. Need to navigate somewhere and control multiple views at once? Navigation bar. Need to segment data as we did in the last chapter? Tab bar.

And those are just two. There are sidebars, toolbars, status bars and more. We'll discover when to use each of them, and then see how to make them really shine with a little extra know-how. The truth is, we could fill a whole book over how these bars work and how you could design interfaces using them. They represent a gigantic surface area within iOS design.

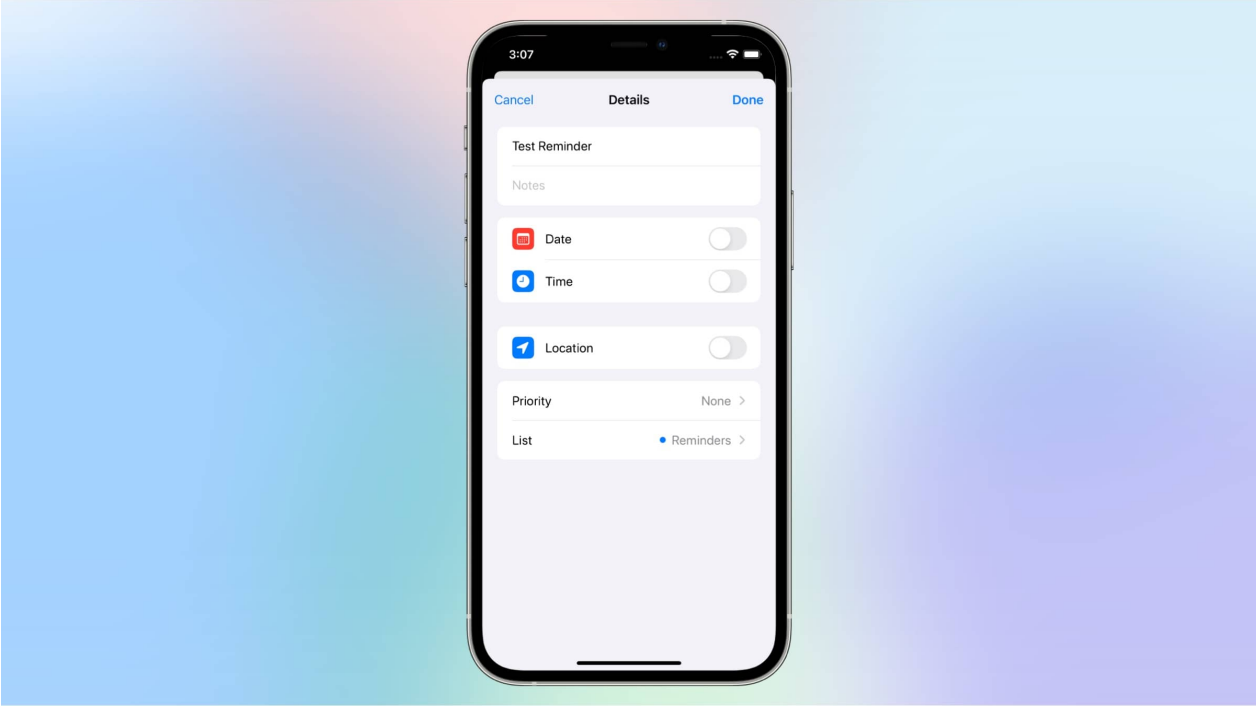
But what I want you to have is *just enough*.

Just enough knowledge about each of them so you know their role, along with a good idea of what they can do and why they do it. That way, you'll have a foundation in place to further exploration of them should you need to.

With that said, let's start with the most common bar in iOS, the navigation bar.

Navigation Bars

Navigation bars are a way for inform navigation and provide context of where the user is at. They are always displayed at the top of the screen, right below the status bar. Navigation bars are most commonly used alongside navigation controllers – which allows for hierarchical navigation styles. But, they can be used in a standalone context, too.

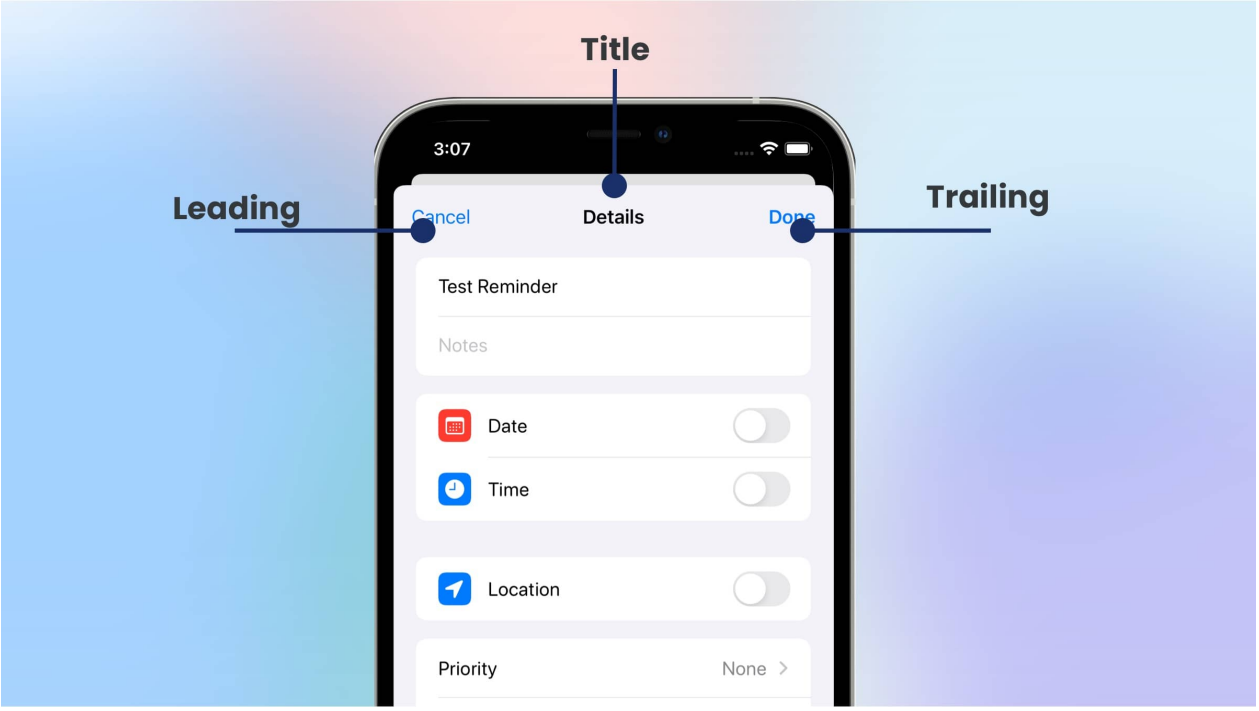


A modally presented navigation bar.

Navigation bars are typically composed of three components, *most* of the time. The content you're showing will inform whether or not you use all, some or most of a navigation bar's components. Those components are:

- The leading bar button items (via `UIBarButtonItem` using UIKit or `.toolbar` in SwiftUI).
- The title view.

- The trailing bar button items (via `UIBarButtonItem` using UIKit or `.toolbar` in SwiftUI).



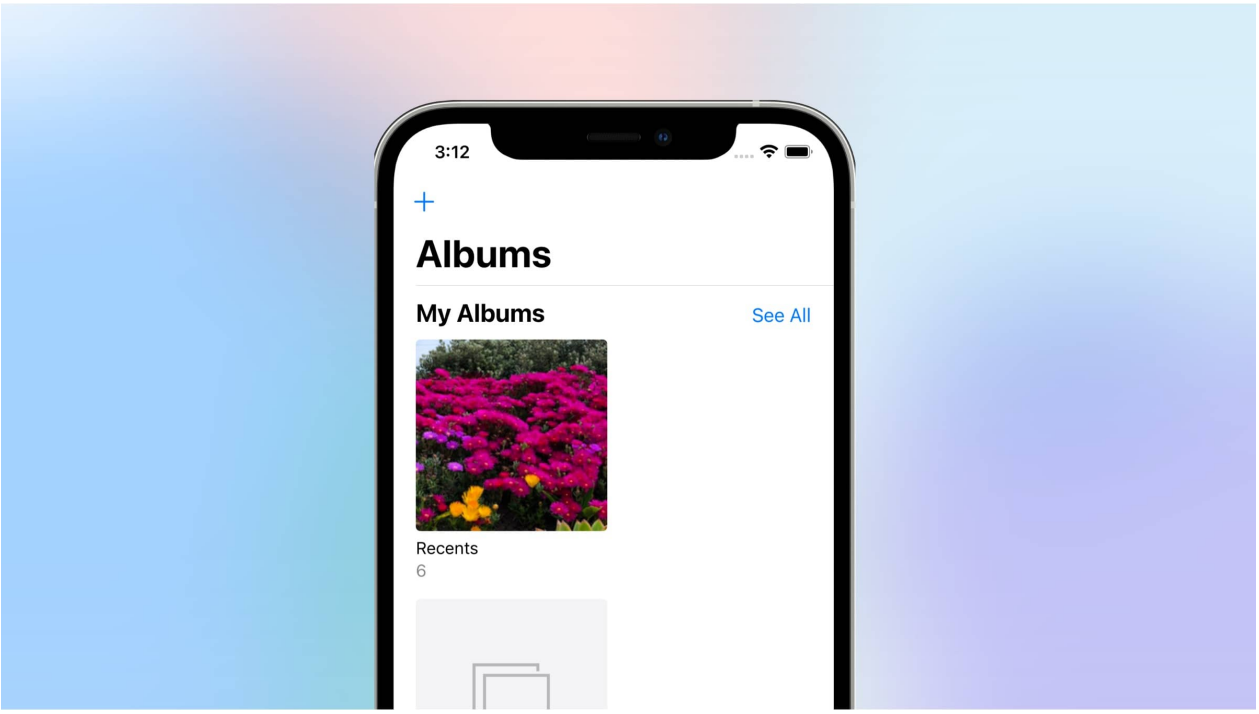
There is also an item prompt you can use, which is displayed right about the title view.

The first leading item is nearly always a back button, unless you're showing a standalone navigation bar in a modally presented context. If you aren't, you can add in some context specific actions such as editing a data set. But, if you're more than one level deep in the navigation hierarchy – this should be a back button.

The title view is something you have quite a lot of freedom to explore ideas with from a design sense due to its generic nature. Typically, it'll be a label with a title of the view. But, if the context you're showing already provides enough information about where the user is – using the title view for that is redundant.

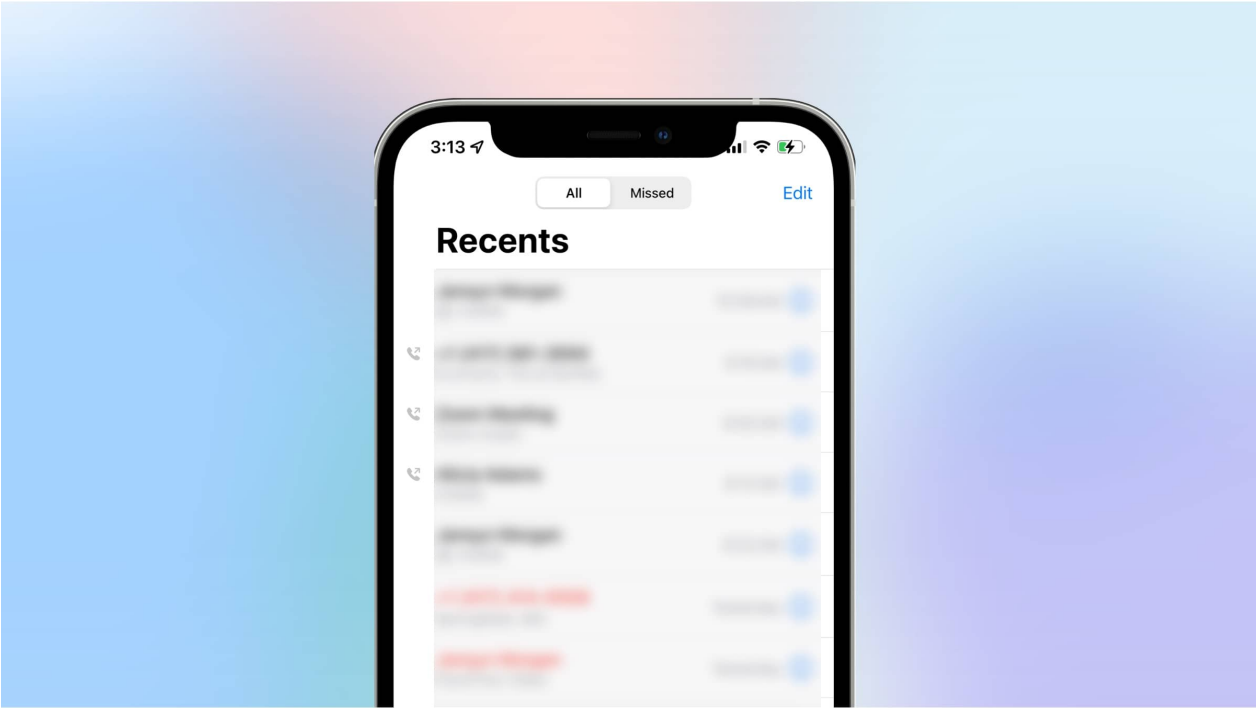
To help decide what the title view's title should be, think of your view that's showing like a book. If you were to name that book, what would it be? That's your title bar.

If you do go the route of using a title, pay close attention to how you style it. There is a large title format available if it helps drive the proper emphasis on the content. The key wording there is *drive emphasis on the content*, because if it's distracting users away from the content – don't use it.

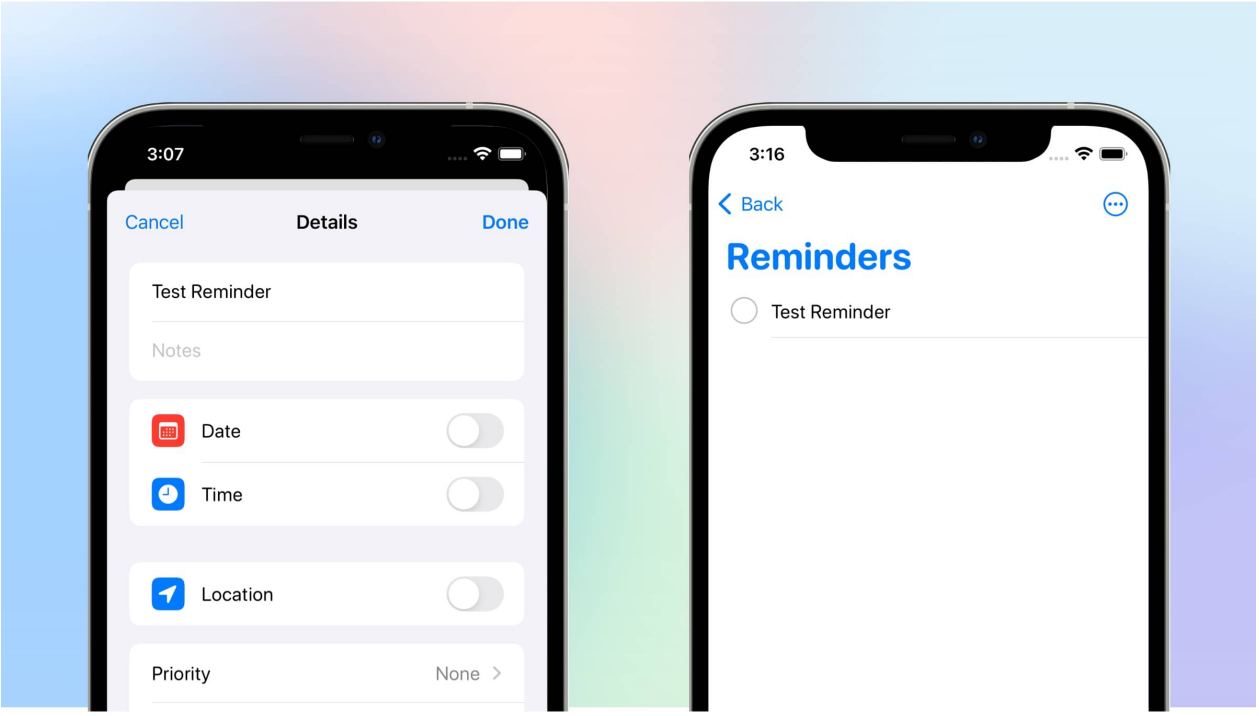


A large display title style.

Aside from simply using a title – you can supply just about anything you'd like here. In fact, it can even be used to help flatten navigation by using something like a segment control at the top, much like the Phone app does:



Finally, the trailing items are probably the most context dependent of all when it comes to a navigation bar's components. If you're wanting to reinforce critical actions for the current context, they should contain actionable items (i.e. "Add", "Edit", or something similar). Or, if you've presented a view modally – it should probably be a "Done" button.



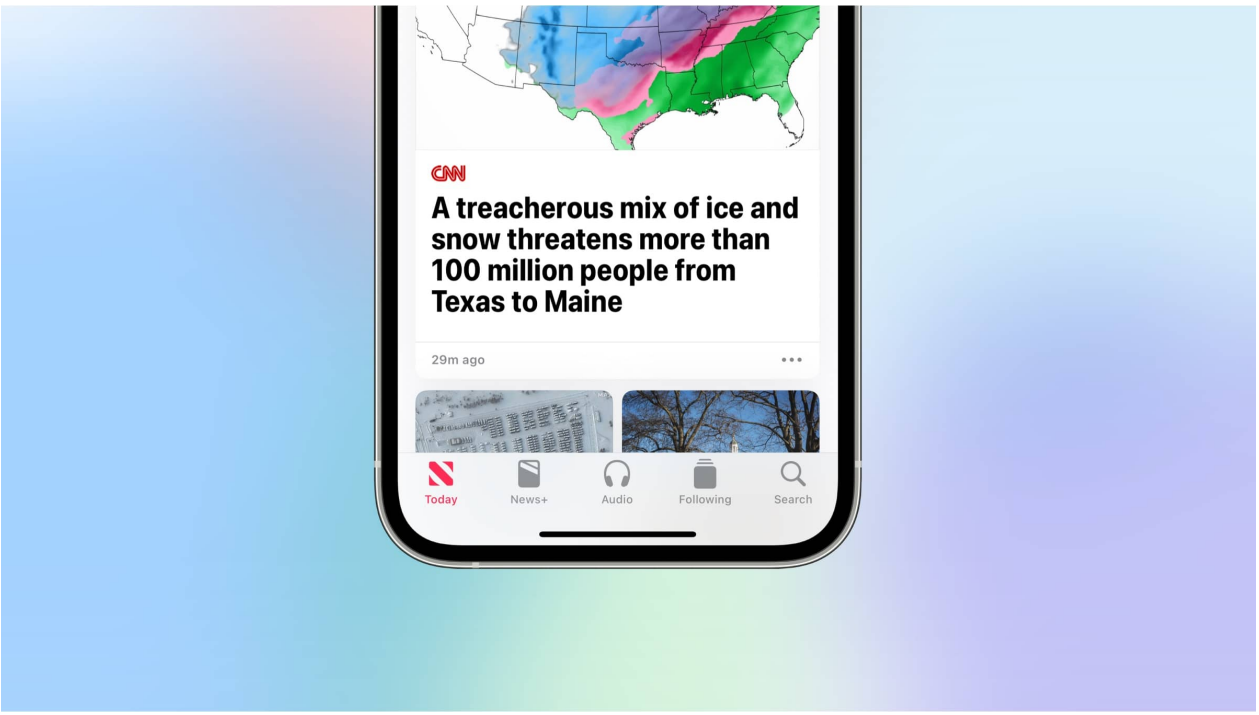
On the left side, a Done button shows in the top right. On the right side example, it's an actionable glyph.

Where Done buttons should go is a war that's constantly being waged. Me? I just follow Apple's lead here, especially when there is no de-facto advice in the Human Interface Guidelines (which, as of this writing, I'm not certain there is). So that's the top right in a lot of views.

The navigation bar has been around a long time and certainly has seen a lot of change. An API well over a decade old – there's a lot to discover about it. It's versatile and much more flexible than developers would have you believe. From tweaking the bar's font, spacing, background material and more – there's probably API available to make it behave how you wish.

Tab Bars

Tab bar is another major component in iOS' bar ecosystem. It, like the navigation bar and side bar, help drive navigation. Each tab represents a mode within your app, and tapping it displays that mode's contents. Tab bars also show in one primary position, and that's the bottom of the screen.

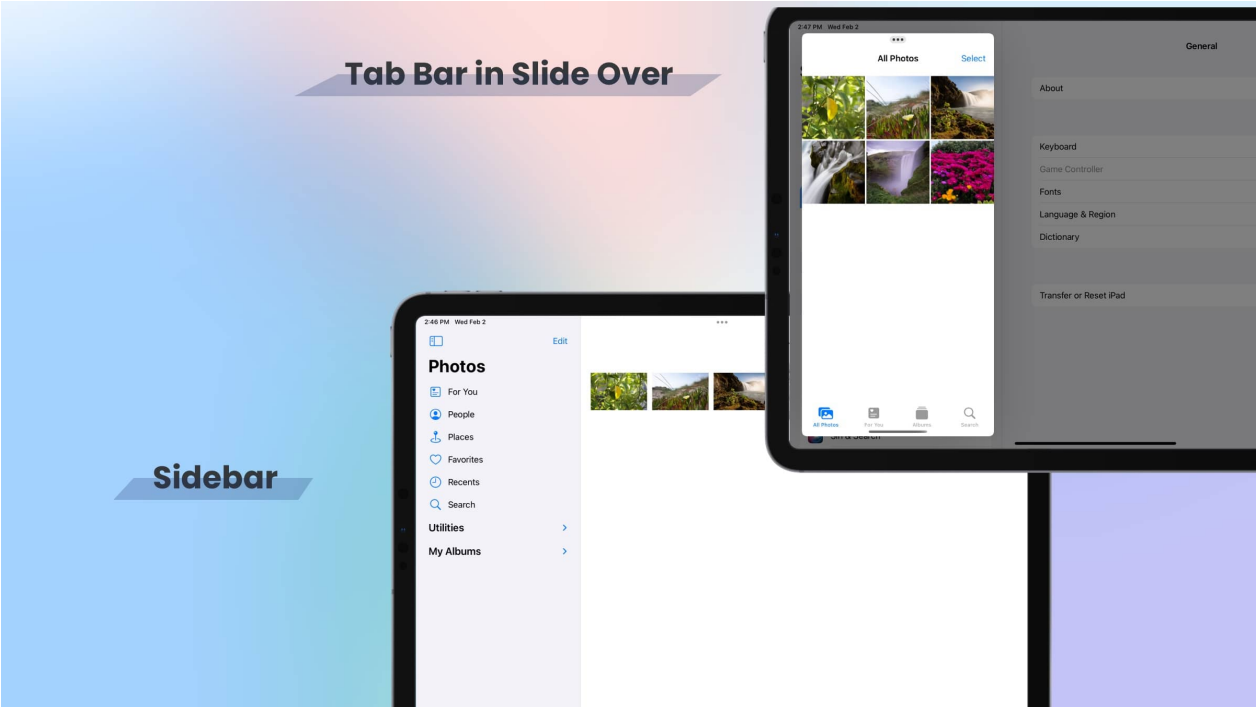


Tab bars are composed of tab bar items, and those typically contain both an image and a title. In fact, I'd propose to you that you should always associate a tab bar item with an image and a title — tab bar items containing *only* titles feel like they aren't using the available space sensibly. SF Symbols are a perfect tool to complement tab bar titles, and we'll talk more about those later.

A tab bar can show several number of items, but try to aim for less than five. Going over that limit shows a More button, which typically means you need to rethink

what's the most important top level information to show or refactor your navigational flows.

Tab bars can play a varying role for navigation depending on size classes, too. For example, on iPadOS it likely makes more sense to use a side bar (which we'll look at more below). Both of these bars have a similar job – drive navigation based on content modes, and each one can show dynamically:



You can switch these views out dynamically as the size classes change. Recall from the last chapter that even though you're on iPadOS – that doesn't mean you'll have a regular size class available to you due to multitasking.

When you commit to a tab bar – ensure your content modes won't change or be unavailable. For example, if a user logs into a social media app one day and sees one set of these bar items, but comes back the next and some have been removed or changed position – that's disorientating.

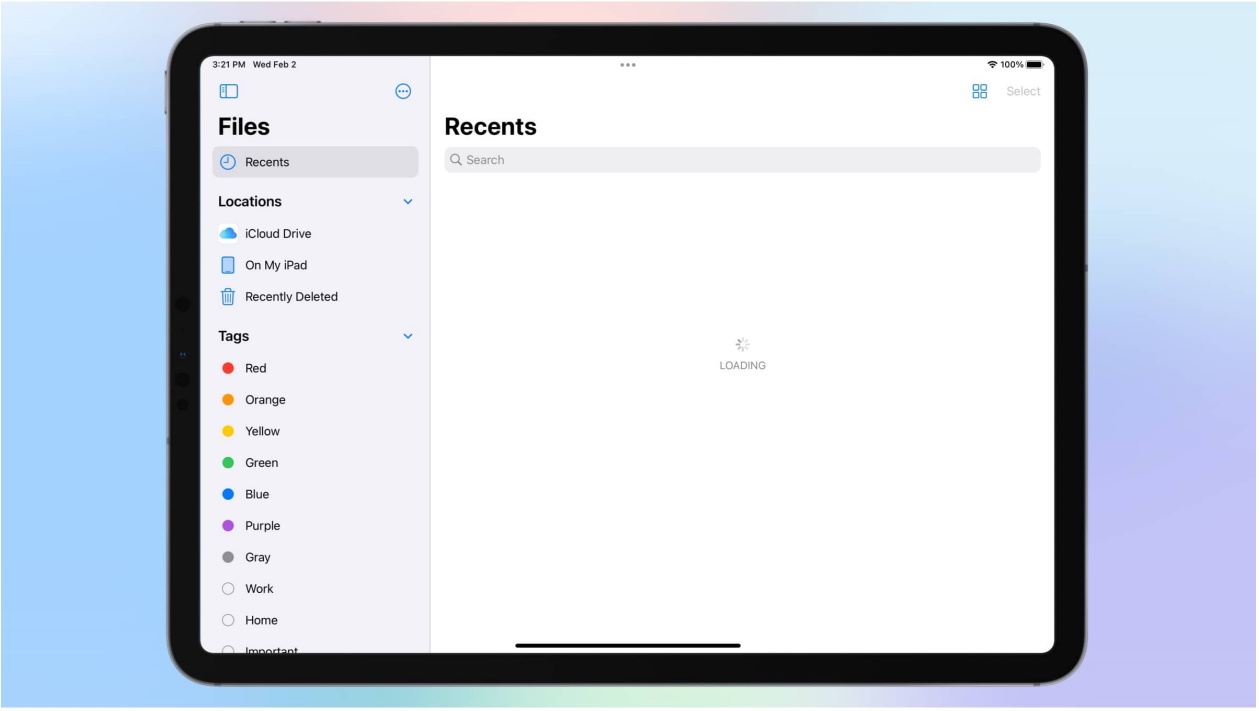
By the same token, don't disable them either. If something isn't available or not reachable, use the content view to reflect that with a text control or something similar, or perhaps an alert view mentioning the issue.

Tab bars also have the ability to participate in badging, which means you can show a badge with a number above them. Use this to drive interaction to that tab when there is something important to show the user. But don't overuse them — badging is a lot like notifications, except the user can't turn them off at the tab bar level. So abusing that is, quite frankly, very annoying.

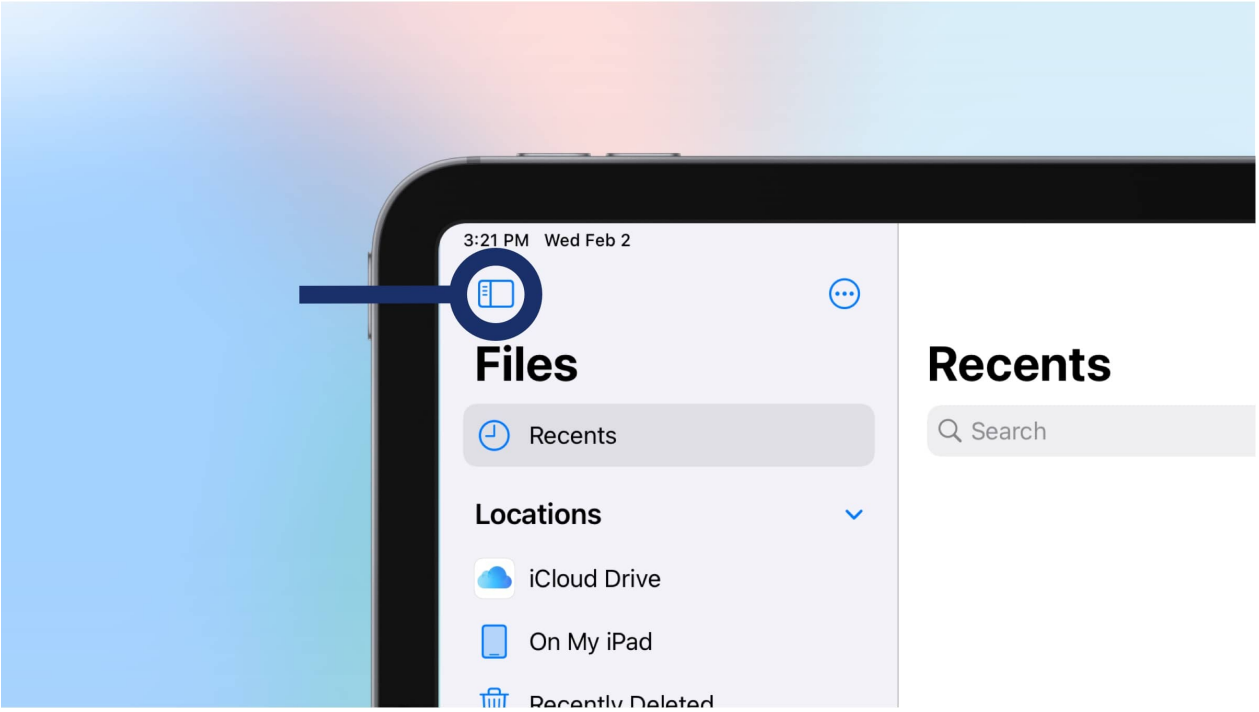
Sidebars

As mentioned above, sidebars play nearly the same role as tab bar — but what they do differently is take advantage of the available space they are used in a bit better than a tab bar could. Sidebars are found within split views — which is a type of view in iOS that allows for top-level hierarchical navigation like navigation controllers but are well suited for larger screen real estate.

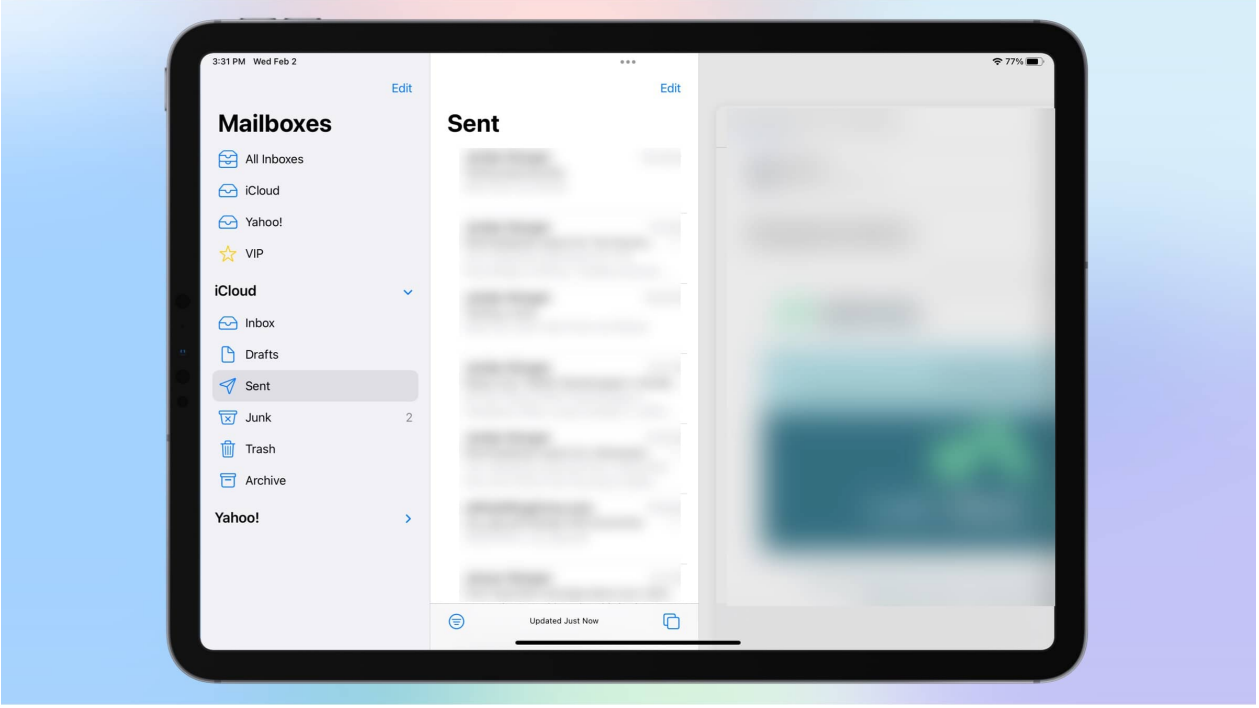
One reason for that is because these are positioned vertically, whereas tab bars are laid out horizontally:



They also allow for a higher degree of flexibility in terms of toggling visibility. While tab bars and navigation bars can hide and show based off of scrolling, that's a situational decision. For sidebars, this is a choice you should nearly always offer. In fact, there's even a system level component for this:



Another benefit of sidebars is that they can go more than one level deep – up to two (or three if you're including the detail view):



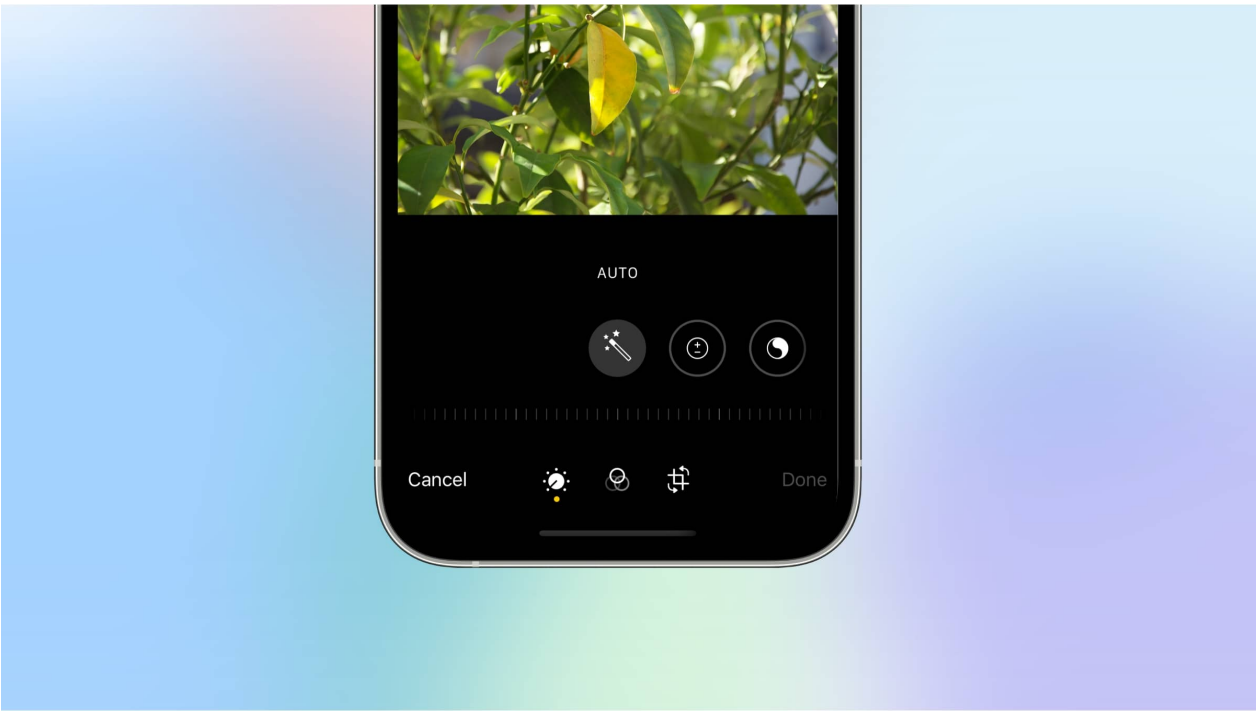
Consider taking advantage of this, because it can allow for navigational flows that utilize an efficient use of hierarchical *and* flat navigation structures together.

Sidebars allow for a bit more flexibility, too. For example, since these aren't composed of tab bar items but rather generic views you supply it – you can open up more "power user" flows such as customizing which shortcuts or actions show there.

Toolbars

Tool bars look much like a tab bar, but their role is to bring attention to common actions within the current context. They don't typically promote navigation, but rather – interaction.

Let's again revisit Photos, and look at its tool bar for editing media:



You can see it provides all of the high impact actions I'd want to do there — and really, that's the whole point of tool bars. When you take a step back, knowing when to use them is quite easy. Just ask yourself, am I trying to provide actions or navigation?

If it's actions, then a toolbar is the correct component for the job. If it's navigation, you're after either a tab bar or sidebar.

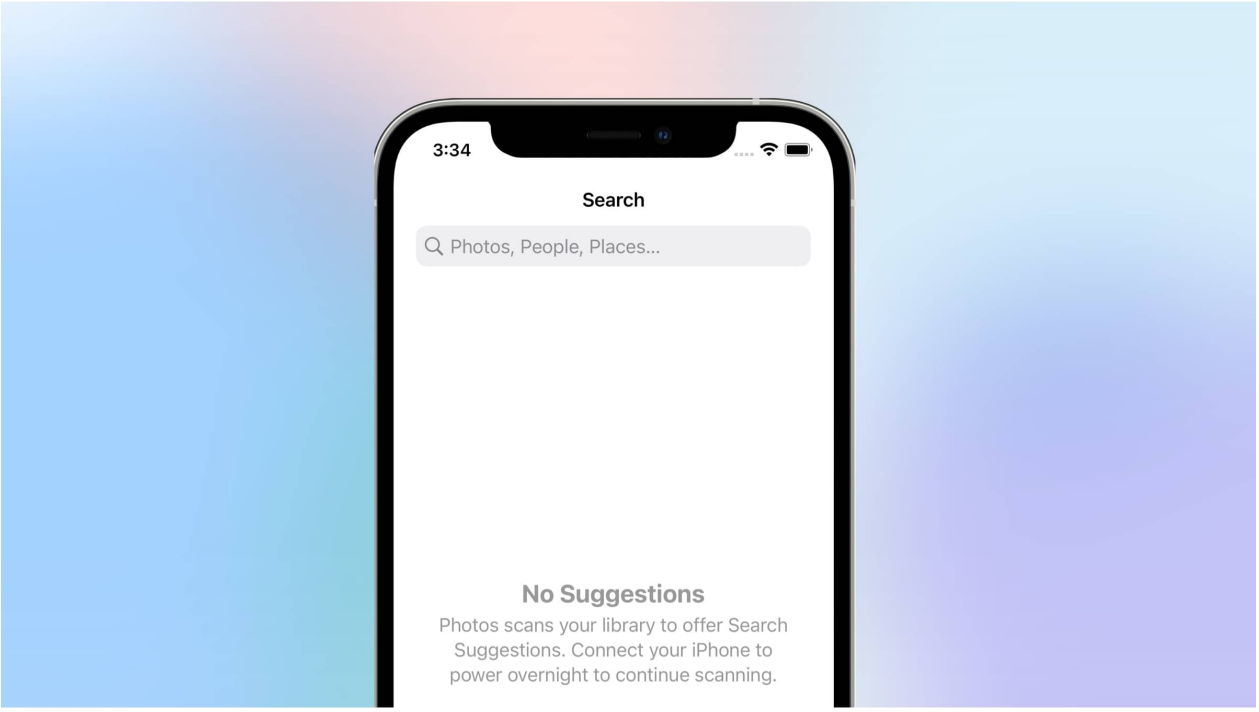
Another note on adaptability here — on regular size classes, opt for navigation bars to elevate actions. They'll show at the top, are generally easier to use in those contexts (i.e. large iPads) and they have more visual prominence. On the flip side, within compact size classes, tool bars are great. People don't have to reach high for them, and they are visually attention grabbing enough too. Users aren't likely to skim over them as they might on larger size classes.

As we've seen with the other bars and how they are composed, tool bars leverage bar items as well. Stick to the rules that apply to those bars with items, too. Don't mix and match titles with glyphs, or put too many items in them and make sure they are only reinforcing high impact actions in the case of tool bars.

However, a toolbar can add more than just bar button items. Given this, you may think it's easier to include things like a segment control in them. While you can do this, I'd avoid it. Only add in controls like this if they are filtering data within the context. Remember, tool bars are all about elevating actions for the stuff that's showing *right now*.

Search Bars

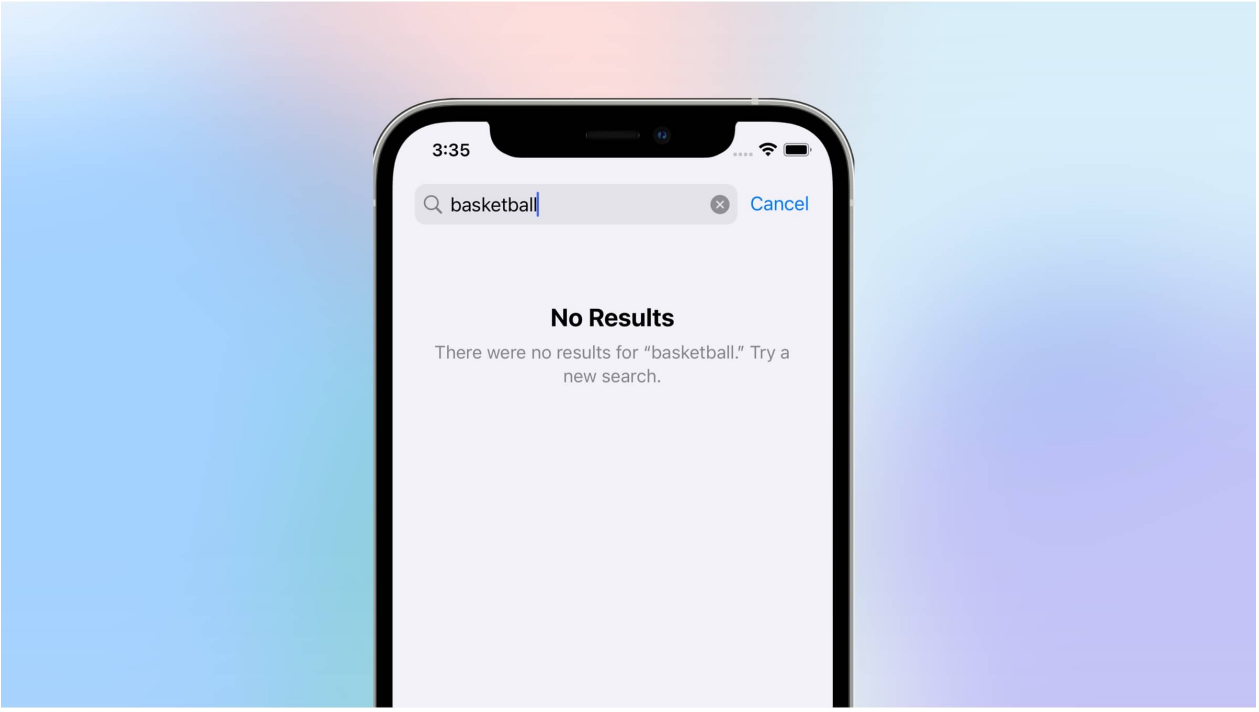
Search bars are all about sifting through the data in the current context. Most of the time, you'll find these right below a navigation bar:



They are flexible, and can be pinned or collapsed at the API level depending on what makes most sense for your view. Search bars are familiar in iOS, so it pays to lean on them. You could just throw a text control in your design for searching, but it would feel out of place. Search bars give you a lot of functionality out of the box, so using anything else means that you have to reimplement that functionality, or worse – you miss it when would users expect it.

If you need to search anything in your app – you want a search bar.

Search bars also allow for adding in buttons in the trailing position, which is great to quickly clear out the text or do things like cancel the search altogether.



Notice the clear out button on the trailing side.

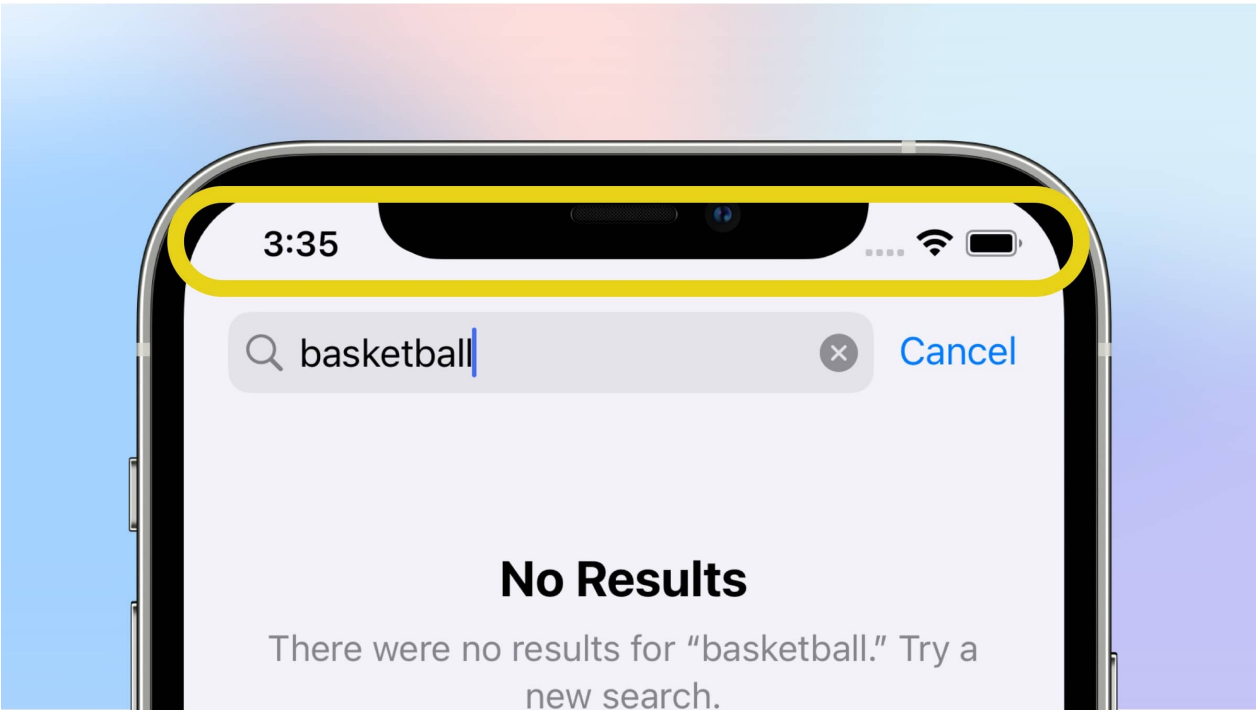
Like most text controls, you can also add in placeholder strings to let the user know what they might want to search for. These types of affordances hint at what search bar is really all about – providing the lowest friction route to the data they want to see.

For example, a search bar doesn't have to operate explicitly on what's showing on screen. When they become focused – they can show an entirely unique view. This is useful to easily display results without necessarily having to worry about accommodating that in the view where the search bar is housed.

Status Bars

The last of the major bars in iOS, the status bar is something you should largely just leave alone. It always shows at the top of the screen, and provides critical information that users are used to seeing.

Let's zoom in at the top of the last picture we used to show it clearly:



The status bar is highlighted in yellow.

Thankfully, for you and I, there isn't much to say here. Designers and developers can't really control much about the status bar, and that's very much on purpose. As far as design, there are really only two things to consider when it comes to the status bar:

- When to hide and show it.
- Ensuring your design doesn't obstruct it.

For the first point, there are situations where you'd want to hide it temporarily. If you're watching a video or showing some media, for example, it makes sense to toggle its visibility.

As for the second point, remember that content will often scroll under the status bar. This could be because you're ignoring the safe area insets by design, using a

transparent or clear element or maybe you're not using a navigation bar (which takes care of that job for you).

In these cases, be sure to account for that by putting some view or element underneath it to ensure its legibility. A blur view or an opaque color should do the job – but far better to avoid this situation altogether.

Tips

Don't Overcrowd Bars

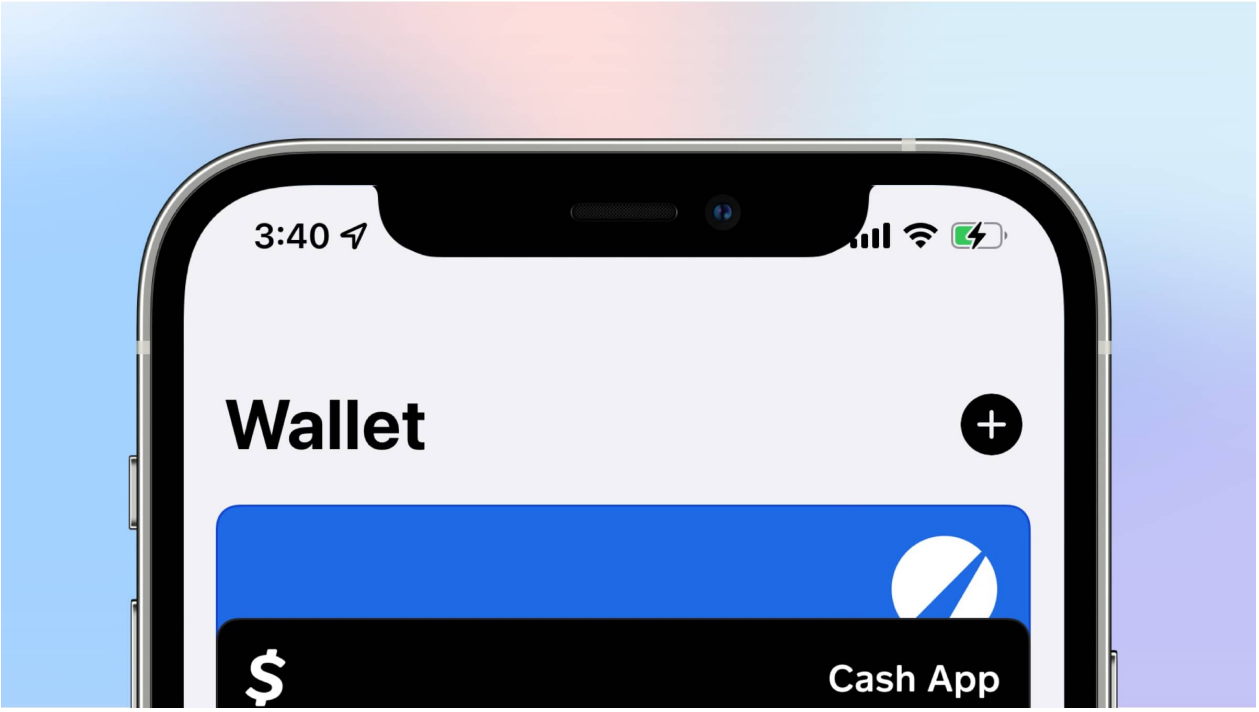
Navigation and tool bars will become unwieldy and distracting should you provide too many items within them. For example, in a navigation bar – If you've got a back button, don't include more leading items. They are too easy to miss when trying to tap one or the other.

If you're in a situation where you need that bar item, consider supplementing your interface using a tool bar instead. As far as trailing items go, I typically don't go over three items in compact size horizontal classes, or four in a regular one.

Strive for Bar Item Consistency

It typically doesn't work from a visual standpoint to mix bar items that are glyphs versus text. Strive to go with one or the other.

Glyphs typically command more attention, so use them with a bit more caution. For example, the Wallet app gets straight to the point in its navigation bar – only showing the a title string and a bold add bar button item in the trailing position:



Navigation Bars Don't Have to Push and Pop Exclusively

iOS includes robust APIs to customize the transition of push and pop operations inside navigation controllers. Even though the navigation structure doesn't change (i.e. you're still using hierarchical navigation) don't be afraid to get creative here if it serves the app well. Check out the Wallet app for example of this in iOS 15.

Springloading Navigation Bars

Drag and drop is an important part of iOS, and by extension iPadOS. You can leverage spring loading in your bar items to drive power user flows. When a user drags something on top of a spring loaded bar item and keeps it there for a bit – it'll blink to activate that action.

If the user would benefit from going back or activating an action from something drag

and dropped onto it – ensure this behavior is supported via the `UISpringLoaded-InteractionSupporting` protocol (which `UIBarButtonItem` adopts).

Changing Bar Button Positions

Bar button items and their positions do not have to be a static affair. In fact, when you're adapting to different size classes I recommend that they should have specific placement for each.

Let's take a look at Photos on iOS 15. Depending on the size class, Photos adapts its approach of bar items by changing their position and even using a tool bar in some instances.

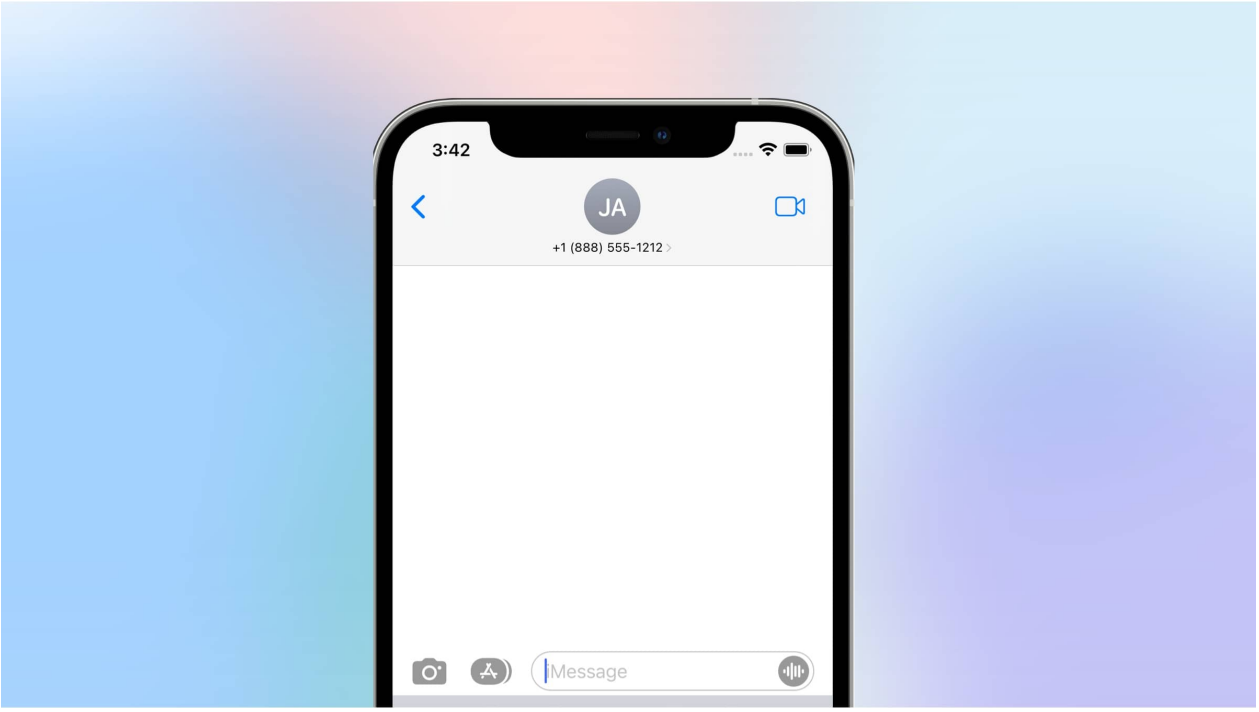
Masking Custom Mask Images for Navigation Bars

It typically doesn't make a lot of sense to make a custom back button image inside a navigation bar. The chevron that indicates "tap to go back" is all over iOS, and users are familiar with it. But, you can do it – so if you *do* make sure you provide an appropriate custom mask image too. If you don't, you'll get an undesirable clipping effect when popping views off of the navigation stack.

Leveraging Tool Bars for Accessory Views

When users are entering in data in text controls, the keyboard will show up. A common pattern to help dismiss the keyboard or otherwise manipulate the data being focused is to attach a tool bar with custom items right above it.

Doing this is a common pattern in iOS, check out Messages:



The tool bar here can be created as an accessory input view.

Or, they can be used to simply perform the next action or dismiss focus of a text control with a simple "Done" button.

Tool bars and Tab Bars are Mutually Exclusive

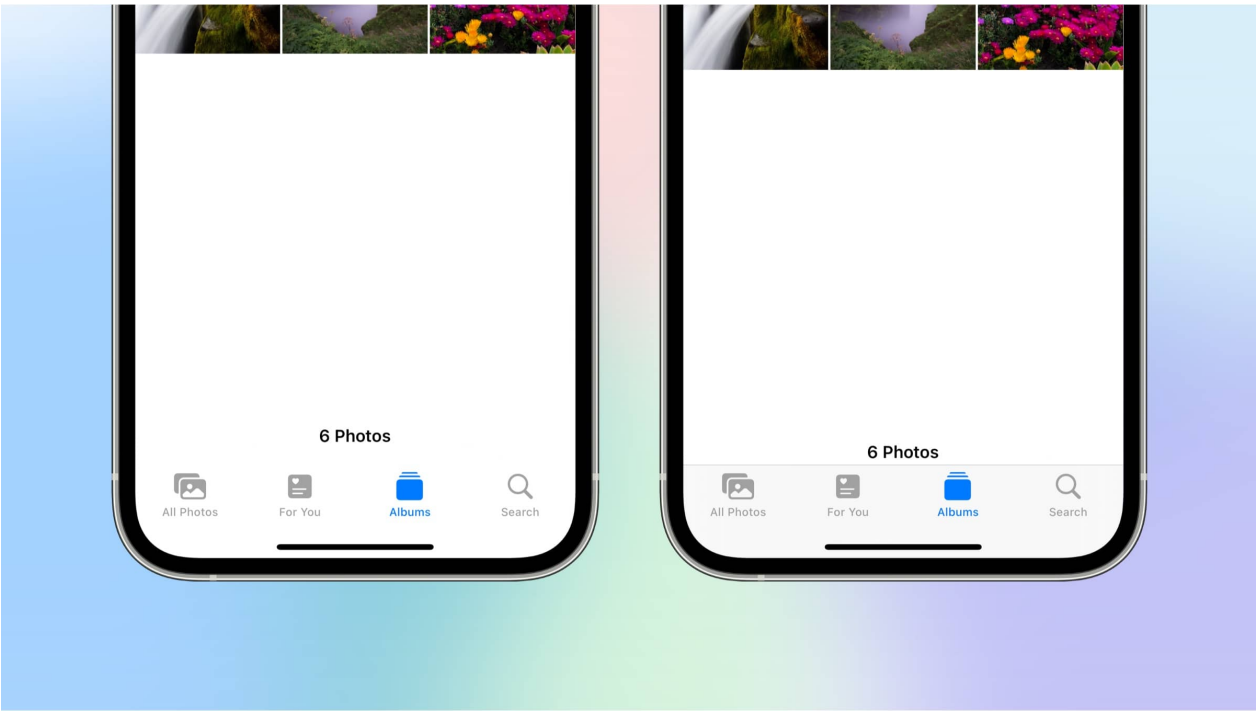
Both of these types of bars appear at the bottom of the screen. If you've read this whole chapter, then you already know they have different roles to play. One drives navigation, the other helps perform actions within the current context.

Because of this, don't use both at once. Unless your tool bar is playing an input accessory view role, both of these bars in the same view could be confusing.

Retooling Navigation and Tab Bar Material Appearances

iOS, by default, hides the background material of tab and navigation bars when there is no content beneath them. When content is under them, then they appear.

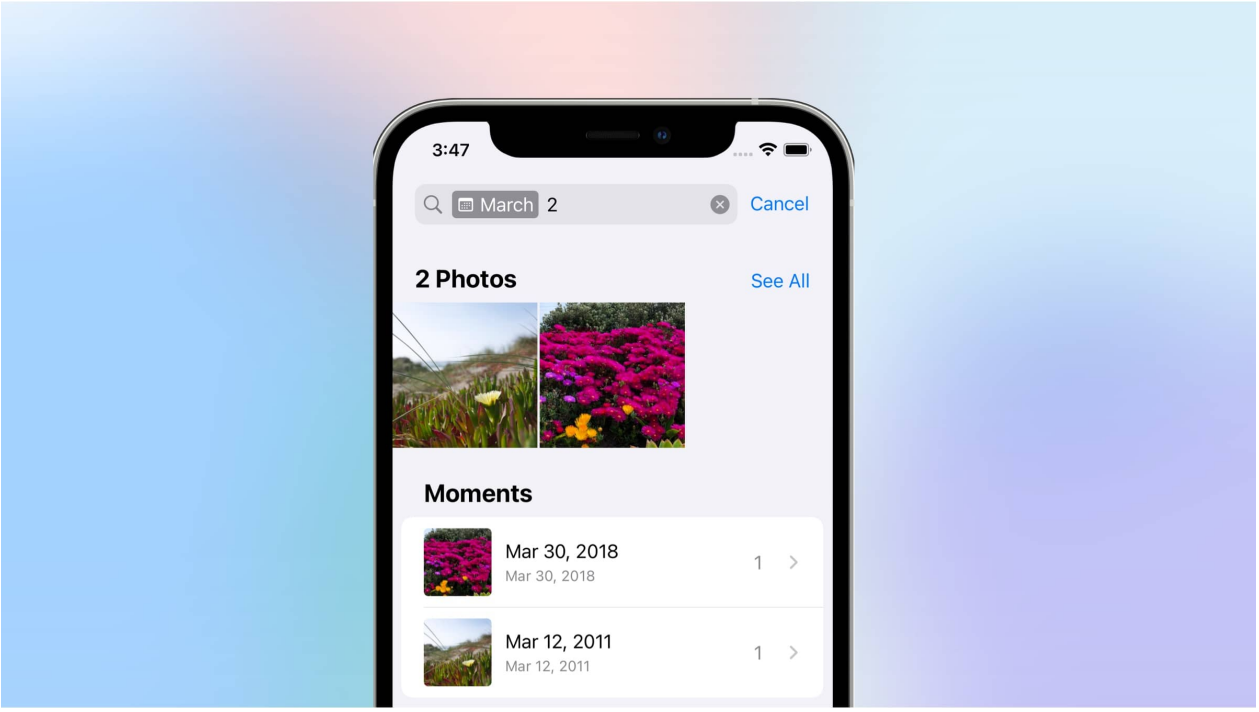
Here in Photos, on the left side the tab bar has no material opacity, but when I scroll down the “6 Photos” just underneath it, it’ll animate in:



If you need, or want, to tweak this for design reasons you certainly can. For example, perhaps you don’t want blurring to occur or prefer an opaque color. All of those are achievable using `UITabBarAppearance` APIs, and even better – you can control *when* those styles should kick in and provide unique looks for when content is scrolled beneath it or not. Fonts and everything.

Leverage Tokens in Search Bars

Search bars support system level tokenization, and that can really help users drive data sets down to what they are after. Check it out in Photos, with “March” being an active search token:



Use them to match against common terms users might search for, and allow them to combine them together to make filtering or finding data much easier. Plus, they allow for users to "undo" search terms easily with one tap. I don't see a lot of apps leverage search tokenization, but they should – and I believe they don't because they were an under the radar enhancement from iOS 13.

Scope Bars for Searching

Use scoping with a search bar to help drive categorical navigation. For example, in Mail – when searching, you can filter against the current mailbox or all of them using the segment control that shows up.

The search bar API provides flexibility to add in controls for this purpose, though I wouldn't recommend using much more than a segment control. A crowded search bar space can be overwhelming.

The TL;DR

Unlike navigation, bars in iOS are "front and center" in design. Using the right ones, for the right situations, is paramount.

Modality

Modality deserves more attention on iOS. Why?

Simply put, it pays to think about when you'd actually use it instead of defaulting to it.

Modally presented views, at their core — should mean one thing to the user: Here's an item you need to take action on, and here is the exit. How we use them, though, is more like this: "Here's a modally presented view. It's going to stay up here for quite a while and there are several paths that you can take."

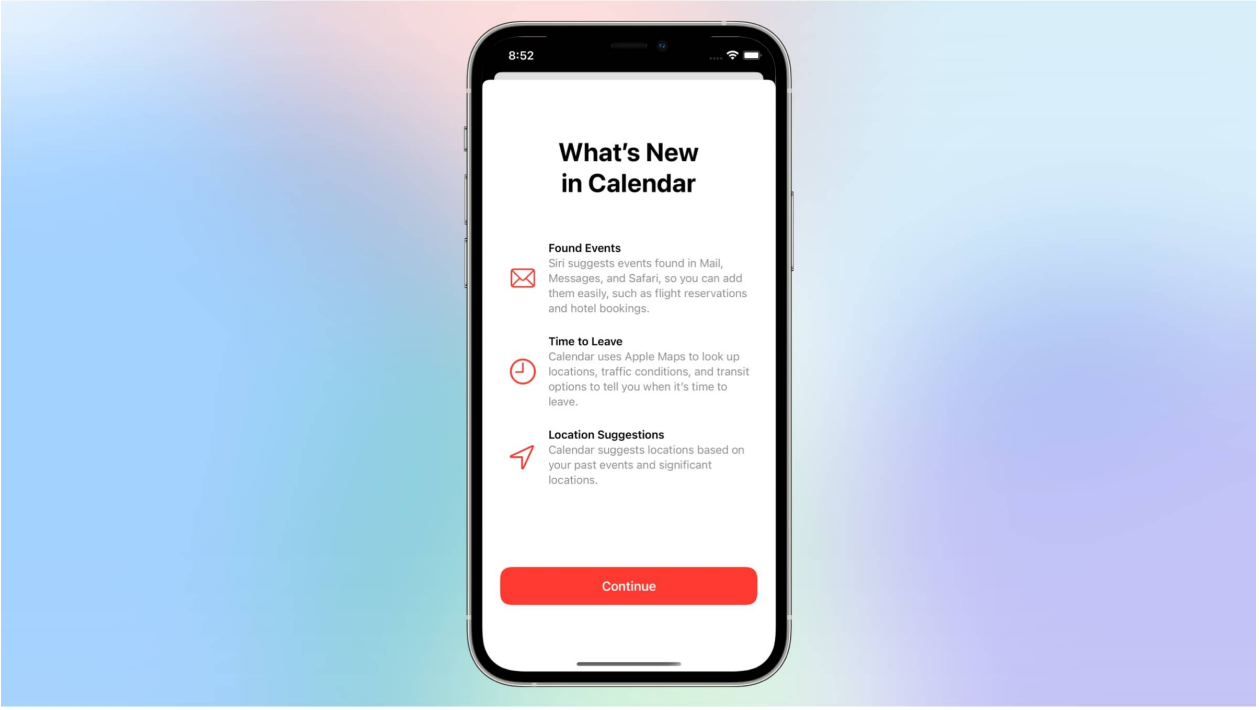
I don't believe you should draw too many lines in the sand with design, but if you'll allow me to for just a second — that second example is what I might consider "wrong". That's not how I think we should use modality on iOS.

However, modality isn't a bad thing at all. It exists for a reason, and when used correctly, it's the best way for a user to complete a focused task. Let's peel back its layers and figure out how to leverage it tactfully.

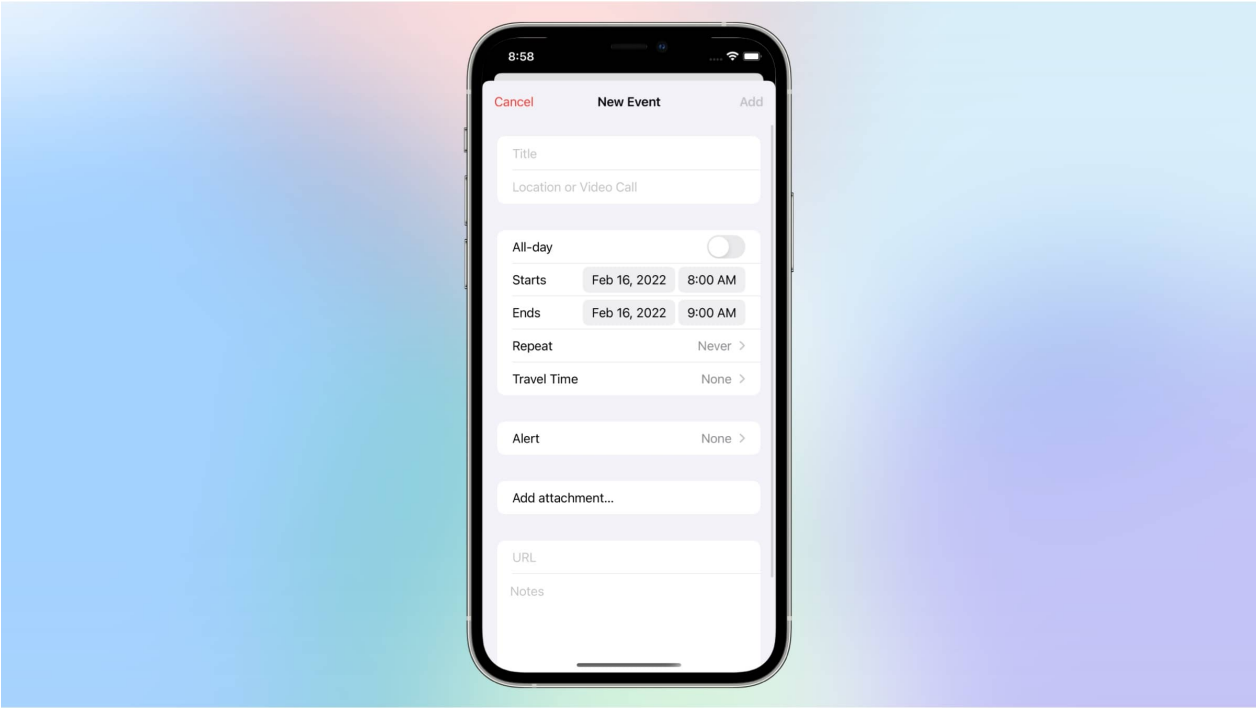
We'll do this by introducing what exactly modality looks like on iOS. Then, we'll look at what its true purpose is. We'll finish out with some tips and topics on how to use it well.

Defining Modality on iOS

Presenting things in a modal context means that you are essentially presenting content over other content. The intent is to focus the user's attention, and minimize distractions. You see it all over iOS, for example – when there is a "What's New" screen shown:



By design, this causes a few side effects. One is that a modally presented view has to be "self sufficient", and it should contain all of the information that the user needs to complete a task. For example, if you're creating a new event in the Calendar app, it's presented modally:



But what's the drawback here?

The previous events are no longer showing, and those might be kind of critical for the context of creating another event. Did you already have something planned for the previous day? What about the next? It's not that Apple's event creation modal is a bad idea, it's just that modality has a cost – and you can't avoid paying it.

You could solve those types of problems a few ways – perhaps you show the events in the modal view somehow, or you don't use modality altogether.

The route Apple could take here isn't the point, though – it's to show how modality is a "take over" of the interface. So if you do it, make sure you have all that you need.

But sometimes, that "take over" is exactly what you want. There are a lot of built in modal controls in iOS to help users take care of important flows, highlighting errors and more.

System Modals

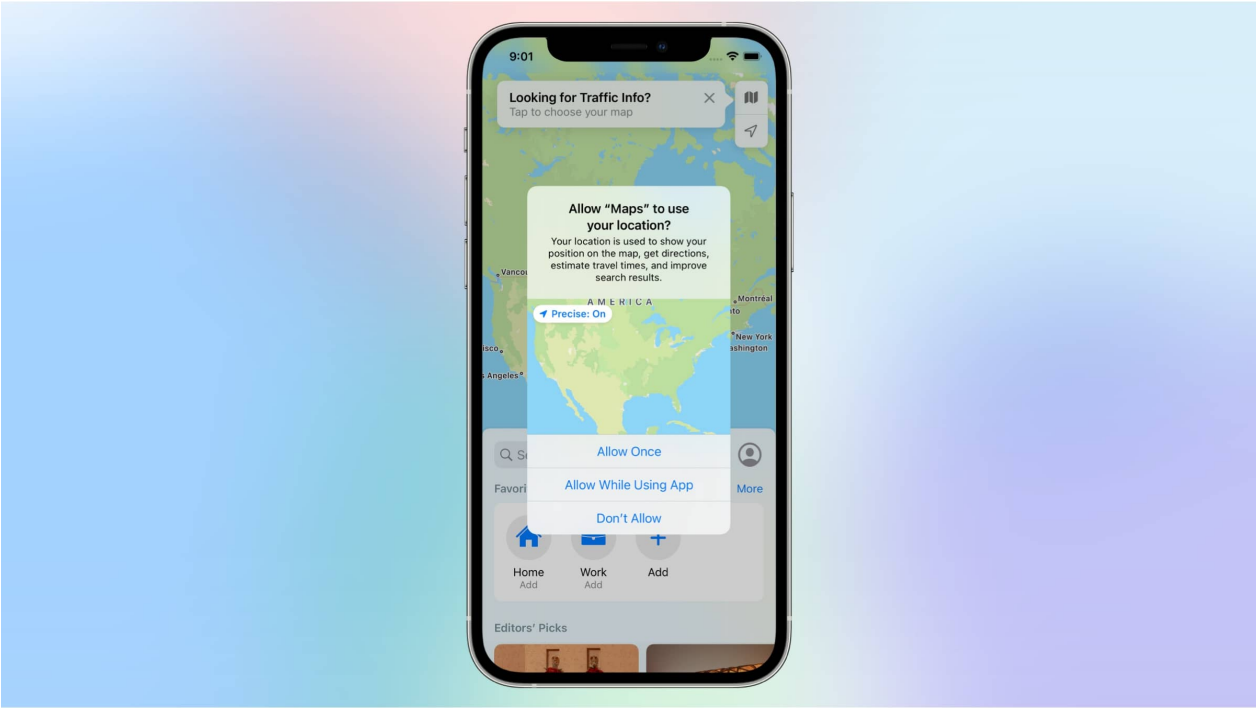
Each of iOS' modal controls default that way because they are prompting the user to take care of a task here and now. Let's take a peek at some of them.

Alerts

Alerts are probably the most commonly used modal view in iOS. They are shown when a user needs to know about something right away, and more than likely take action on it.

For example:

1. The user has no data connection.
2. A critical error occurred.
3. A confirmation of a high value action completed.
4. Permissions need to be granted.

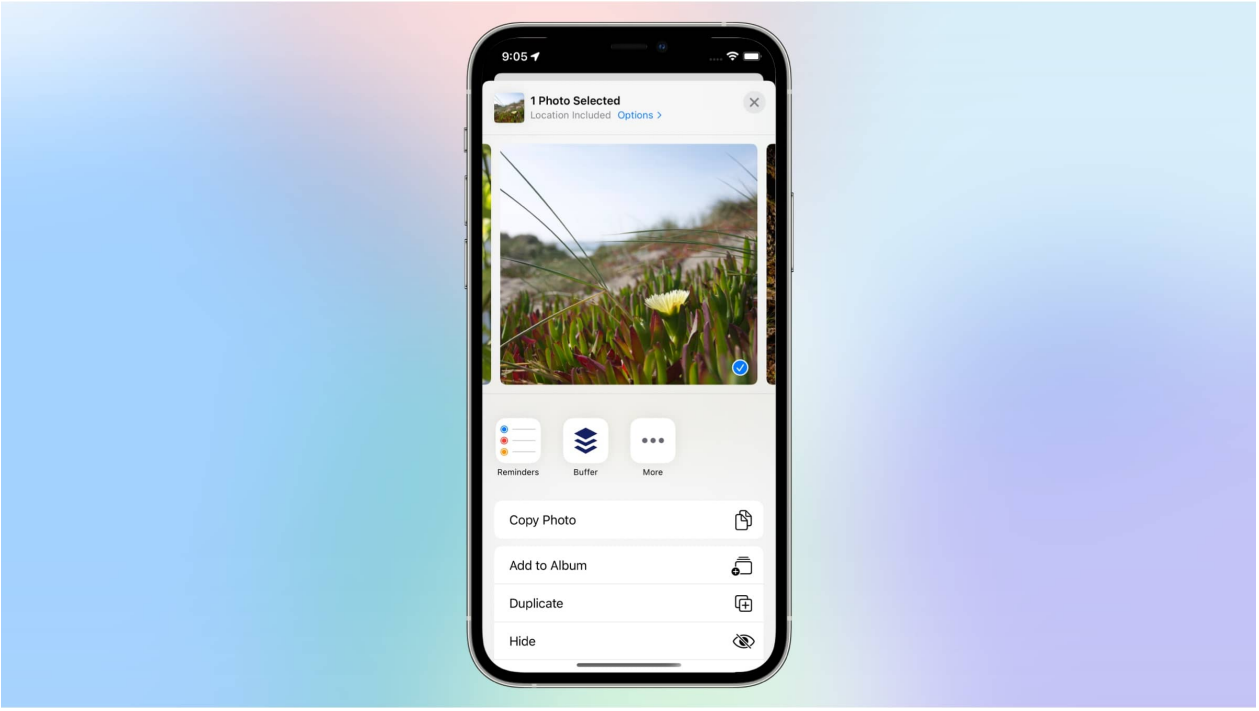


Maps uses a modal alert to ask the user for location tracking permissions.

All of those examples are things the user should be alerted to right away. Because of that, the alert shows modally over everything currently onscreen.

Share Sheet

The share sheet is one you might use every day. Ever since extensions expanded the capability, and more importantly – shareability, of iOS, the share sheet has become a widely used control:



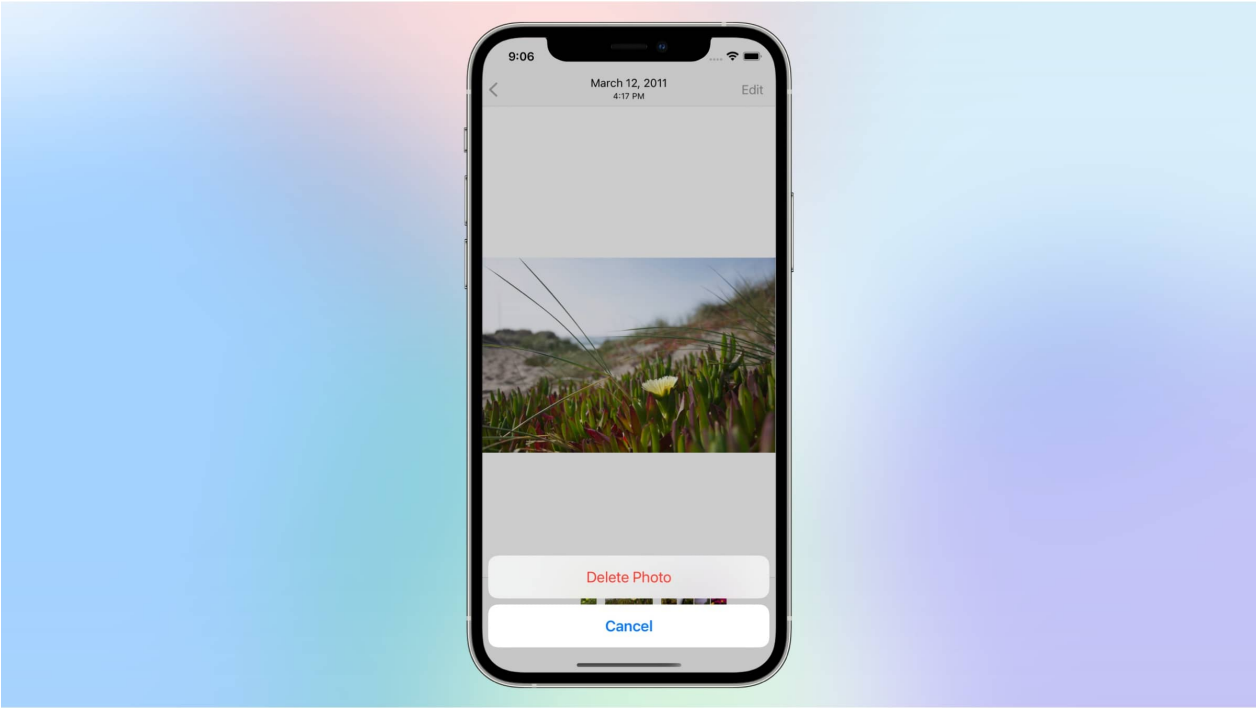
You can see why its modal, too. You need to share some content, and that's it. It's quick and efficient, as most modally presented views should be.

Action Sheet

Action sheets are similar to alerts, but they are different in two primary ways:

1. They present at the bottom of the view.
2. They confirm the user wants to carry out an action related to a task they just started.

Because they are finishing up a task that probably has a high impact, the modal presentation is warranted.



And More

iOS has a lot of these modal views. From activity views, App Store overlays and more. So it begs the question, when should we use modality in general?

When Modality Makes Sense

Fortunately, when it comes to modality, the process of deciding whether or not to use it can just about be simplified into a simple flow chart or set of rules. Here's how I think about it:

1. Does the user need to accomplish a focused task?
2. If they do, then is it different *enough* from the current context?

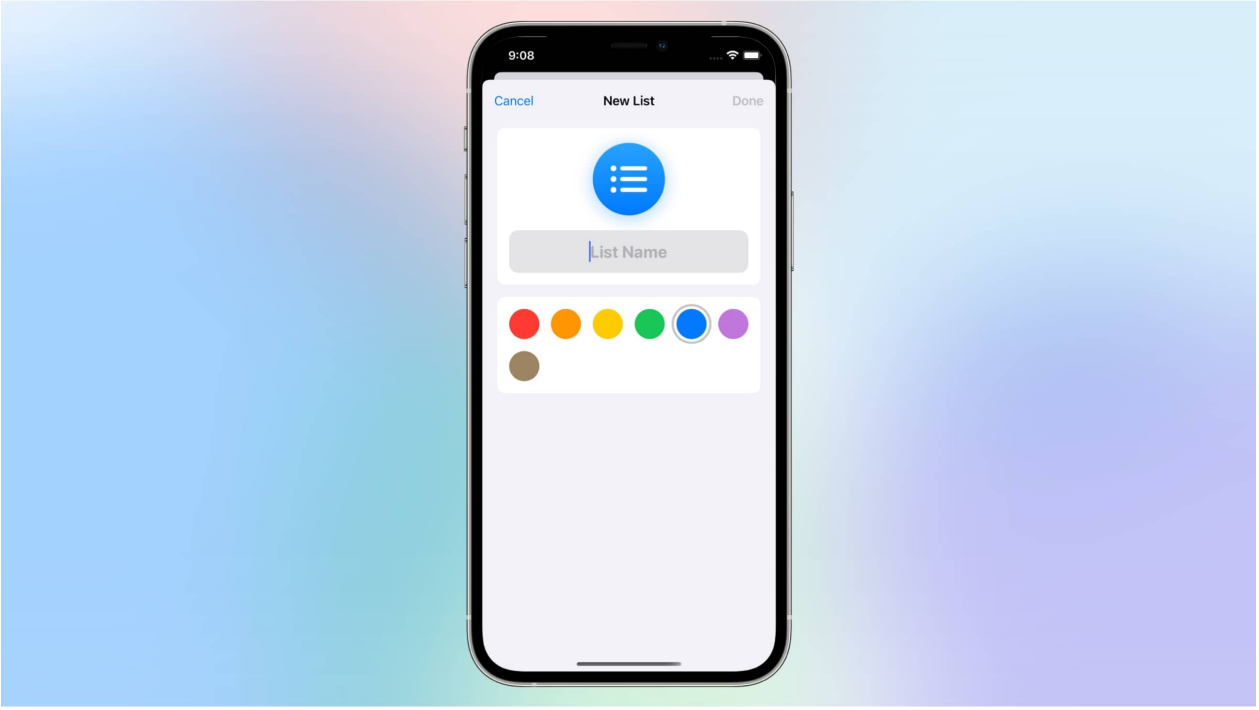
If both of those answers are yes, you're likely in the right spot to leverage modality. To go a bit further, if you're creating a universal app that's shipping on iPadOS and iOS, I'd also ask this:

3. Do I have enough screen real estate to inline this information, or otherwise flatten it?

If that answer is yes, then I *wouldn't* use modality. We'll chat a bit more on that below under the tips section. Overall, we can learn a lot about modality's role on iOS by surveying Apple's example:

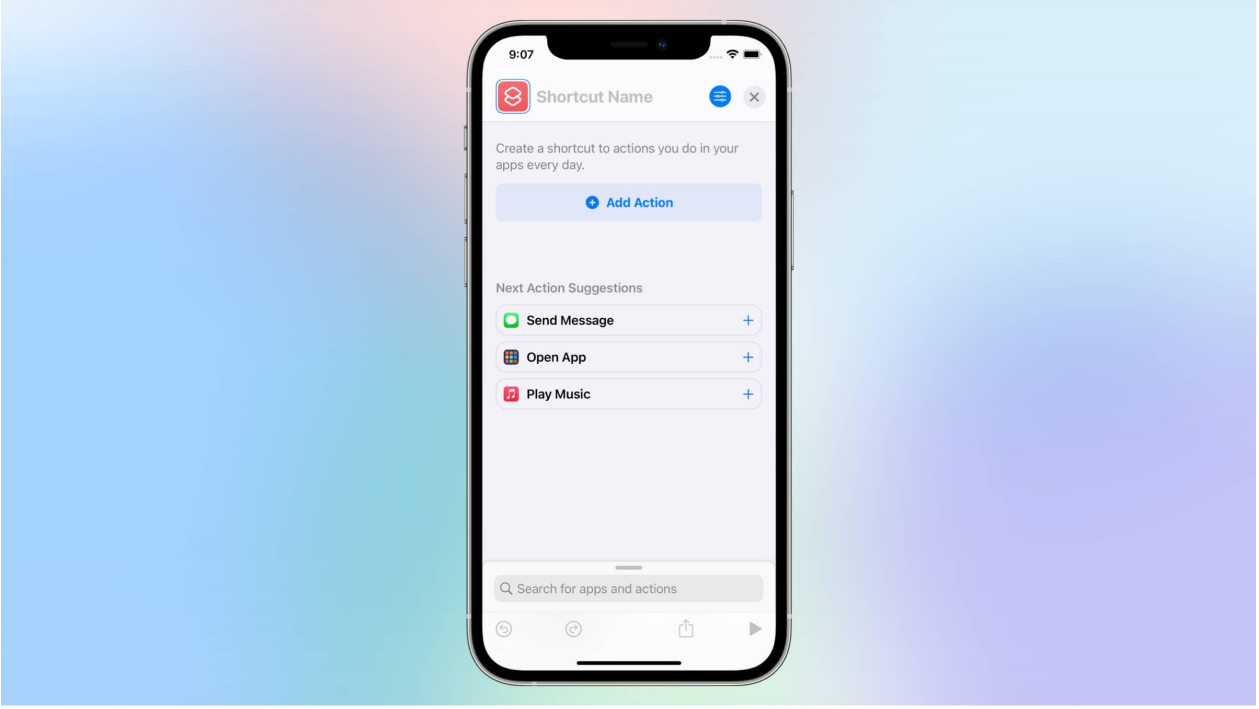
Reminders

When creating a new list, we get a modal view. Why? Because it's a new, focused task with a clear beginning and end.



Shortcuts

When you create a new Shortcut, it's presented in a full screen modal:



You can start to pick out some similarities — namely, these modals show the user a contained "thing" to begin and end. And, when it ends, the modal is dismissed.

Focus versus Context

Modality is tradeoff though, as is everything in design. But it pays to know the trade-offs you're weighing. And for modality, in my mind it's this constant pull between focus and context.

Focus, because when you present things modally — focus is the benefit. There is nothing else to do! You've hidden everything else. All the user can focus on is the task at hand, and that's it.

Context, because due to that focus, there is no way for the user to really gain any contextual benefits that they might've had from the screen you just covered up.

So, weigh those two when making design decisions on how to present modal views, where you do it and why.

Tips

Always Include an Exit Control

Some modal presentations won't include an exit route by default. If that's the case, you'll need to include an explicit out for the user. Remember, modally presented views are all about doing something, and then leaving.

This may seem painfully obvious if you're a designer primarily, but when programming for iOS you'd be surprised how easy it is to find yourself in this situation. We'll look at built-in modal presentation types below, but to jump ahead for a second – some of those types take up the full screen. So if you missed the exit button in those cases, the user is now trapped in a modally presented jail from which there is no escape.

iPadOS versus iOS

On iPadOS, you might have more room to work with. That means you shouldn't try to rely on modals as much. Again, modals inherently take away a lot of context because it covers up so much information – so if you can keep that then, then you should.

This is especially true since iOS 14 and the introduction of the sidebar on iPadOS. So many of Apple's own apps removed what were modally presented views and inlined them into the sidebar.

Calendar is a textbook example of this. When creating events before iOS 14, a popover was presented with all of the calendars you could associate it with on iPad. Now, the calendars are persistently showing in a sidebar from a toggle button – and in compact sizes, it falls back to the modal.

Another place to look is when editing data, especially text-based data. If you've got a list of stuff with names or text information, renaming those things is usually a modal affair. But why? If you can just edit the name inline, right there – do it.

Be on the lookout for situations like this. Editing data, showing data – it's so easy to just present it modally because we often begin on the iPhone and work our way up to the iPad. But when we do, we bring some modal baggage with us that we probably don't need.

Modal Presentation Techniques

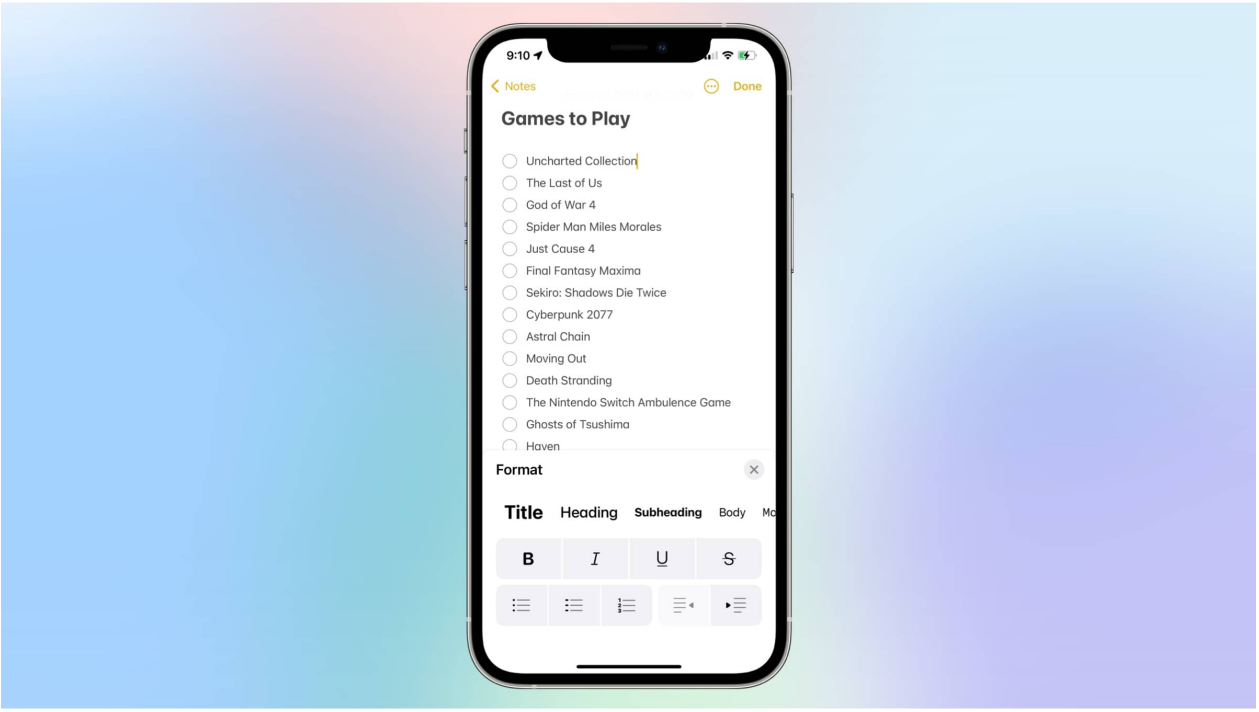
When you need a modal, be aware that there are a lot of ways to show it:

1. Automatic, Page and Form Sheets: Most of the time, these use a sheet with rounded edges. Recent versions of iOS add a little padding to the top of the sheet, which also means you can slide to dismiss it. Forms and pages might adapt to present a bit differently, but they won't cover the whole view either.
2. Fullscreen. This takes up the whole view, which means you'll need a close button with these.
3. Popover: This one *is* modal, but it won't take up a huge portion of the screen most of the time. It's the "tool tip" of iOS in many ways, found primarily on iPadOS.

There are a few more, but most of those are implementation details when programming such as displaying content over a certain context or customizing the modal transition altogether.

Meeting Halfway

iOS also has a way to present some sheets half way, literally. This is great if you want to provide some editing controls for the current context. Notes does this and it works great:



Using the sheet presentation APIs, Notes on iOS shows a view that's technically modal in presentation, but doesn't obscure the whole view.

This way, you can keep information front and center but display some controls to help manipulate any data. Another way this might be useful is when it comes to the inlining conversation we had above. Sometimes, it's just not viable to include editable controls

within a list of data, so this is a good compromise in that situation. It'll still avoid full-on modality.

The TL;DR

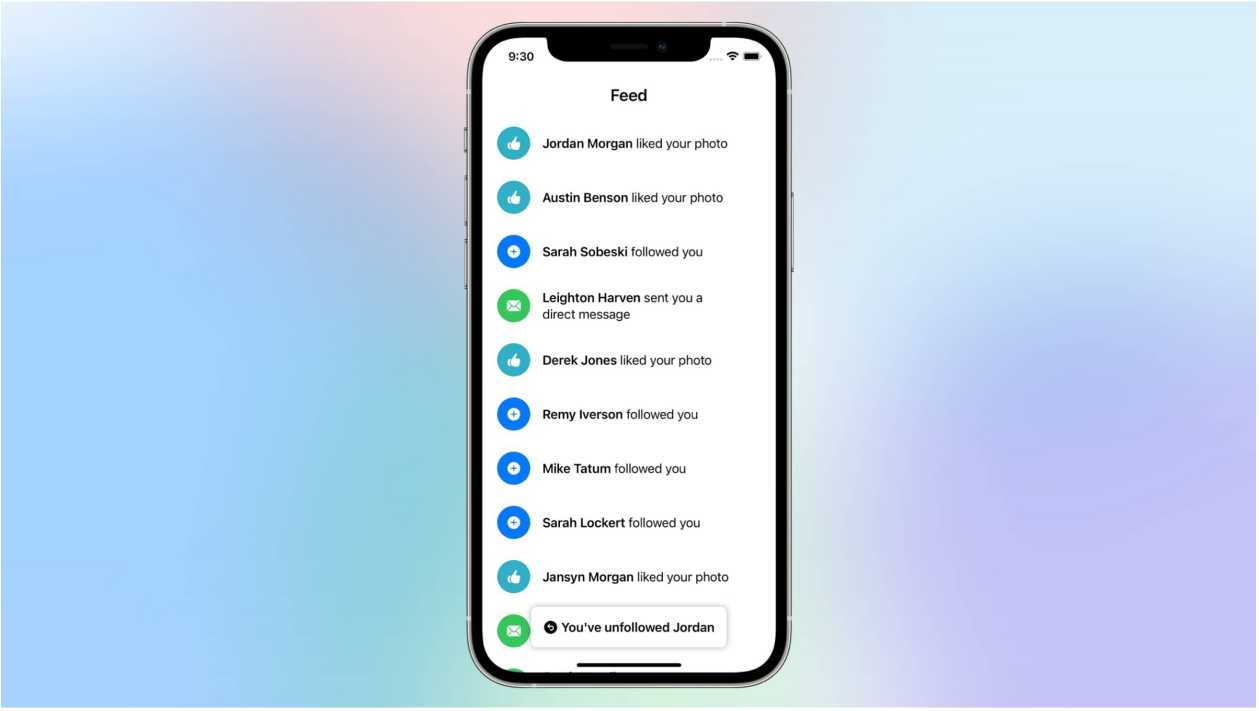
Modality on iOS has a focused purpose, and that's to present singular contexts that require attention. Try not to use it outside of that.

Feedback and Reinforcement

Apps are full of interactivity and the consequences of actions, both good and bad. Take data, for example – especially the kind users enter in themselves. This is the stuff you don't want to mess up. What's worse than drafting up a long response and then losing it via the many ways things can go wrong in iOS (i.e. a crash, booted from the foreground due to low memory, a freeze – you get the idea).

So, what do we do? The safe thing. We try to save the user from themselves. When they try to delete something, we ask them if they really intended to go through with it. At the same time, we offer a way out just in case they accidentally tapped a delete button or changed their minds. We *reinforce* their actions with intentionality using our designs.

Continuing that thread, let's assume a user did delete some of that data they entered in. They deleted a draft, removed a post or trashed a ticket they entered. Maybe you might display something like this:



Some apps may show some interface element to let users undo critical actions.

Now, we've given them a form of *feedback*. Plus, since deleting things is a high impact action – they can undo it right away. A user experience like that may or may not make sense for your app, but that's not the point.

So what is?

Creating Confident Users

The point is this – users are going to do all sorts of stuff in your apps. Most of it will be on purpose, and some of it won't be. No matter, when we talk about the notion of feedback and reinforcement we're driving home the idea of giving the user confidence.

When users are confident in your app, they will, undoubtedly, be more engaged with it. Confidence breeds all sorts of amazing benefits, think about it for a moment.

Confidence outside of just apps, but, well — in anything, really. When you are confident in yourself, you go about your business differently than if you weren't. When you are confident in your personal relationships that are important to you, you enjoy them and the companionship that they bring.

Confidence is key. We want to give people that same sort of feeling in the things that we make.

There is a line that exists within software, whether you realize it or not, that represents the degree to which someone will either lean into your app and be its advocate, or abandon it altogether and forget it ever existed on their home screen. And, where they fall on that line is usually tied to their confidence level in your app.

The bad news is that it's definitely a *line*, it's not a spectrum. They love it or they hate it. They tolerate it because they have to, or they use it because they want to.

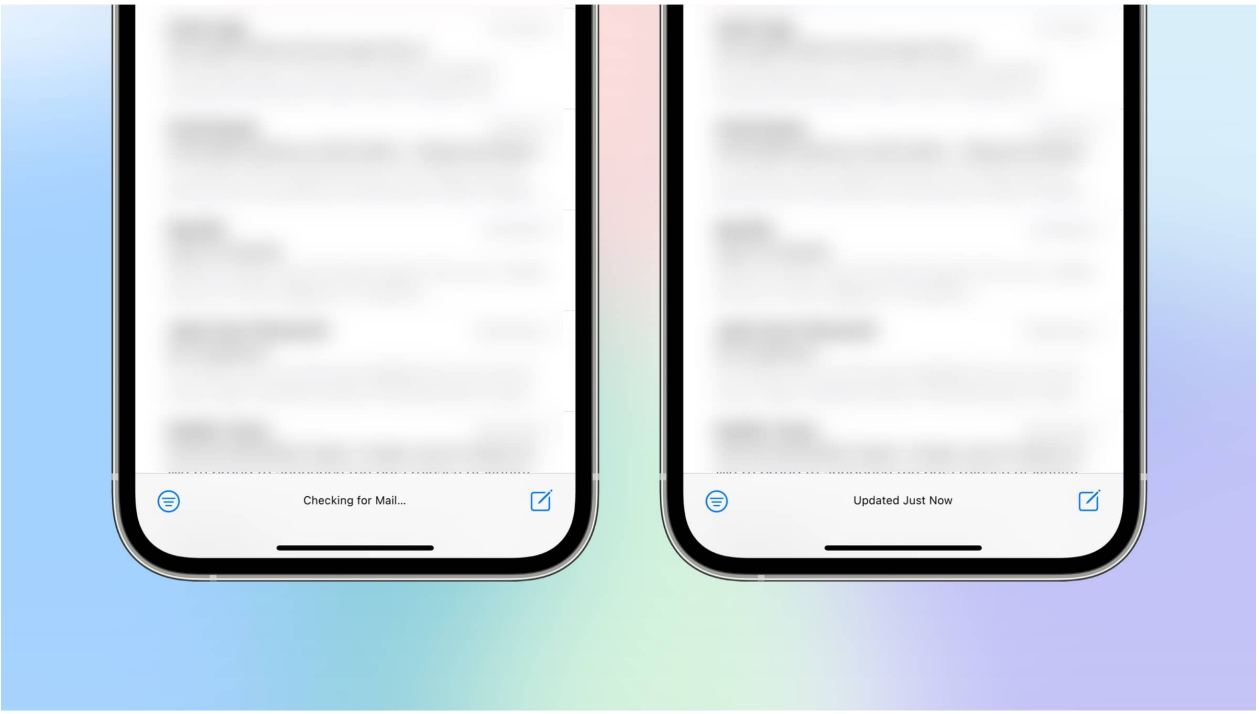
The good news is that swaying someone while you still have the chance from a fresh install or reactivation isn't that hard. Just don't do things that lower confidence, and lean into the things that strengthen it. Make them confident!

Confident that what they expected to happen, did happen:



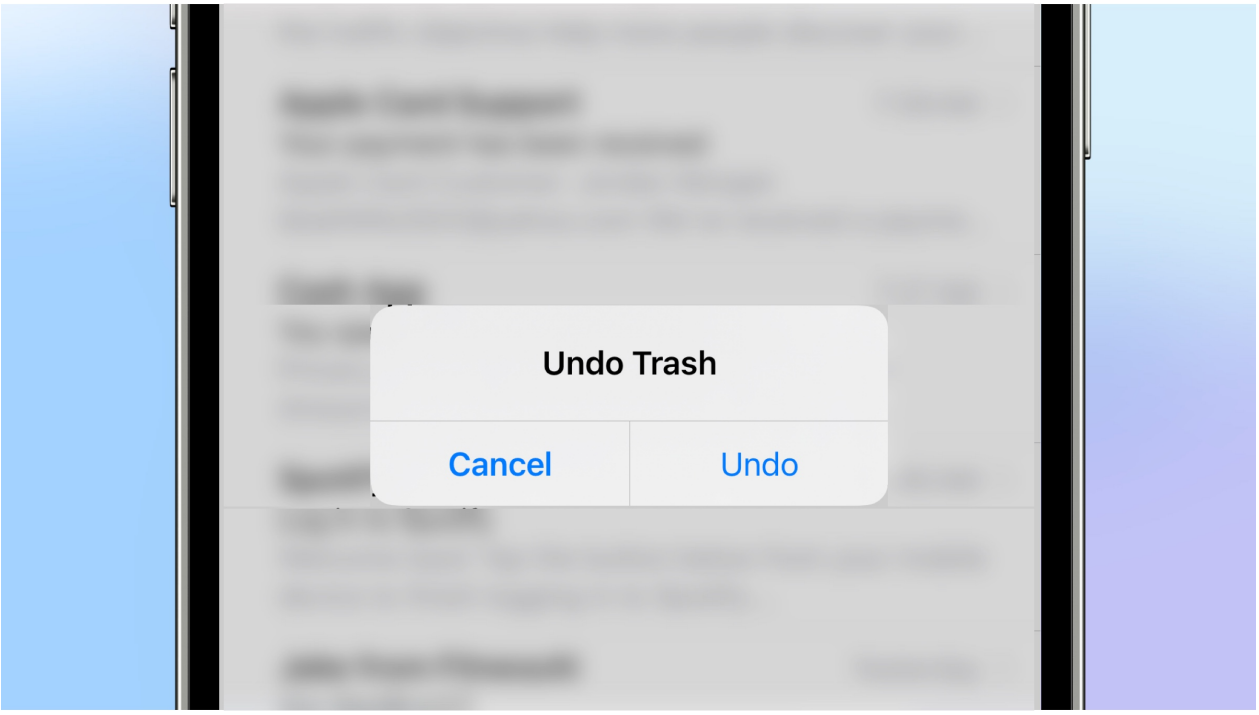
This banner indicates that the iPhone was successfully muted or unmuted right after the user flips the switch.

Confident that something they started, finished:



Mail shows a status text view at the bottom in its toolbar to let users know when mail was last fetched.

Confident that a mistake they made, can be undone:



After you delete some mail, you can shake your device to potentially undo it.

If you view things that you add to your app as a way to increase confidence, you'll find that several of those things probably promote the idea of feedback and reinforcement.

Next, let's take a look at a few ways I think about those two ideas and the buckets they typically fall into.

Types of Feedback and Reinforcement

For the types of feedback and reinforcement I design, I've found that they mostly fall into one of four categories:

1. Confirmational: These reinforce that an action a user intended to take was carried out.

2. Informational: These convey context and status to the user, regardless of whether or not it was the result of an action they took.
3. Reversible: This type of feedback lets the user know they can undo an action they recently executed.
4. Tactile: This represents the idea of providing physical or digital feedback to the user as they do something, such as using haptics.

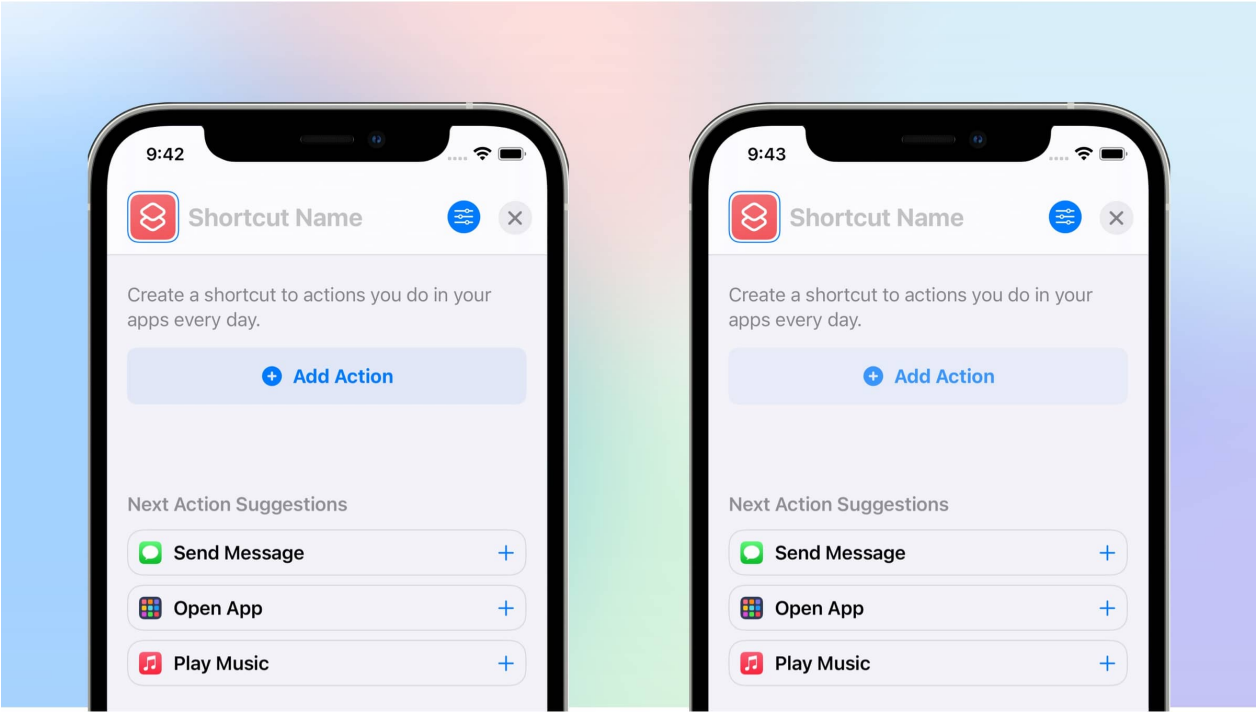
Some types of feedback and reinforcement could easily fall into more than one category, but for the most part – I tend to think about them distinctly to help me more easily decide when to utilize them.

Confirmational

Confirming actions can be a great way to promote predictability in your app. We often paint predictability in a pejorative sense, but in app design, I'd argue it's a wonderful concept to promote.

When you tap a button, something happens. We expect that. If we tap a button, and…..nothing happens, well – that's unpredictable behavior. It's stressful. What just happened? Who knows.

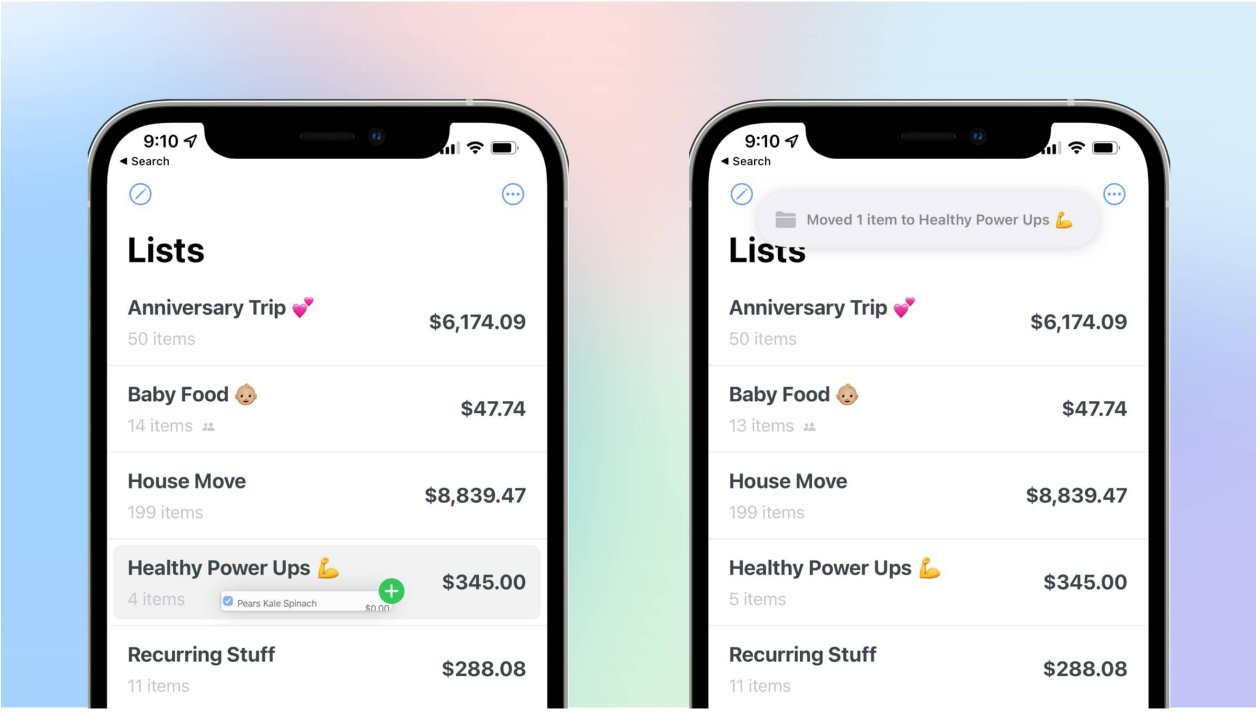
You can do this several ways. Buttons, for example, have a highlighted state when they are tapped, which typically reverts when you lift your finger up:



Notice the subtle difference in color in the "Add Action" button.

That little bit of color difference between the two states confirms to the user that they tapped it. But, going further – sometimes you can add even more to those interactions if the situation warrants it.

In a previous app of mine, I had users confused on if items that were dragged from one list to another completed successfully. So, I added a little confirmation banner to reinforce that the action occurred as they intended:



Now, this was before that same style of banner was more prominent throughout iOS, so I wouldn't necessarily recommend that today – but the goal here is to get the gears turning in your head. Is there something you can confirm happened, or *didn't* happen, in your app that has a high-impact consequence?

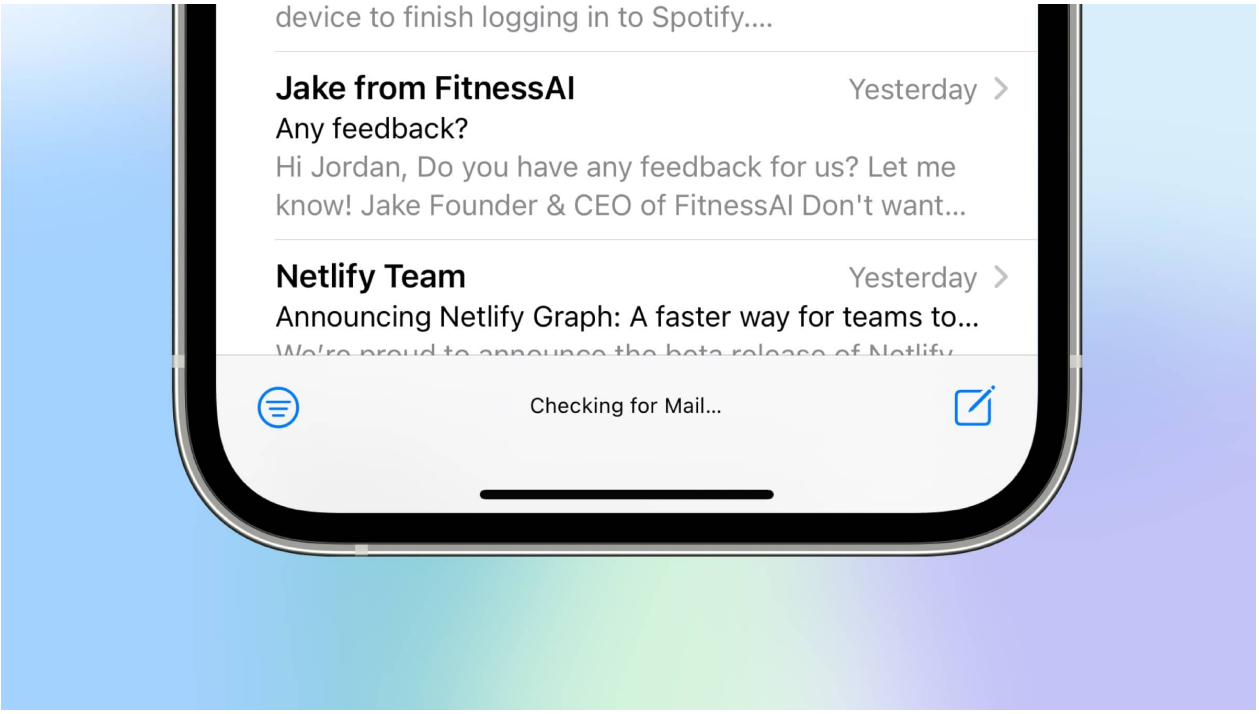
Informational

Speaking of that banner, it's a wonderful example of informational feedback. Much like when you toggle the mute switch, the same banner appears when you connect AirPods to your iPhone.

That's a low-friction way of iOS informing us of something we likely want to know about – did the headphones I just pick up connect to my phone or not? We've all read, or have, stories of people becoming frustrated with connecting something to bluetooth. Frequent trips to settings, turning things on and off and doing many other ceremonial rituals all in the name of getting two devices to pair.

It's not ever described as fun, is it? So, this little piece of information is great because we're confident that something that historically was a giant pain is now seamlessly done – and we *know* that it's done, thanks to the handy feedback interaction.

Another example that Apple themselves talk about in their Human Interface Guidelines is Mail. That status at the bottom within the toolbar is a wonderful example of providing timely information to us:



When did we last fetch mail? What's new? Is there an error?

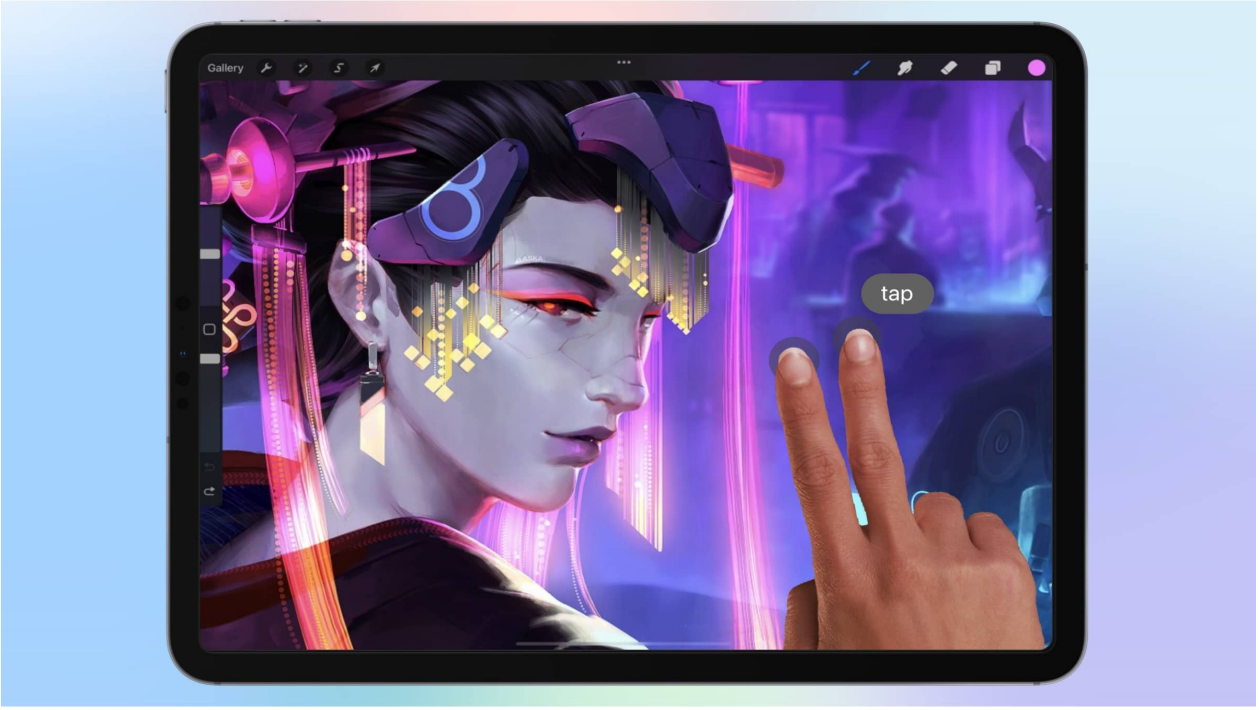
That text label provides all of that feedback with virtually none of the drawbacks informational feedback can sometimes fall into. There's no extra steps to get to it, it's out of the way and it doesn't block any interface operations.

Informational feedback is all about giving users important information while keeping it out the way.

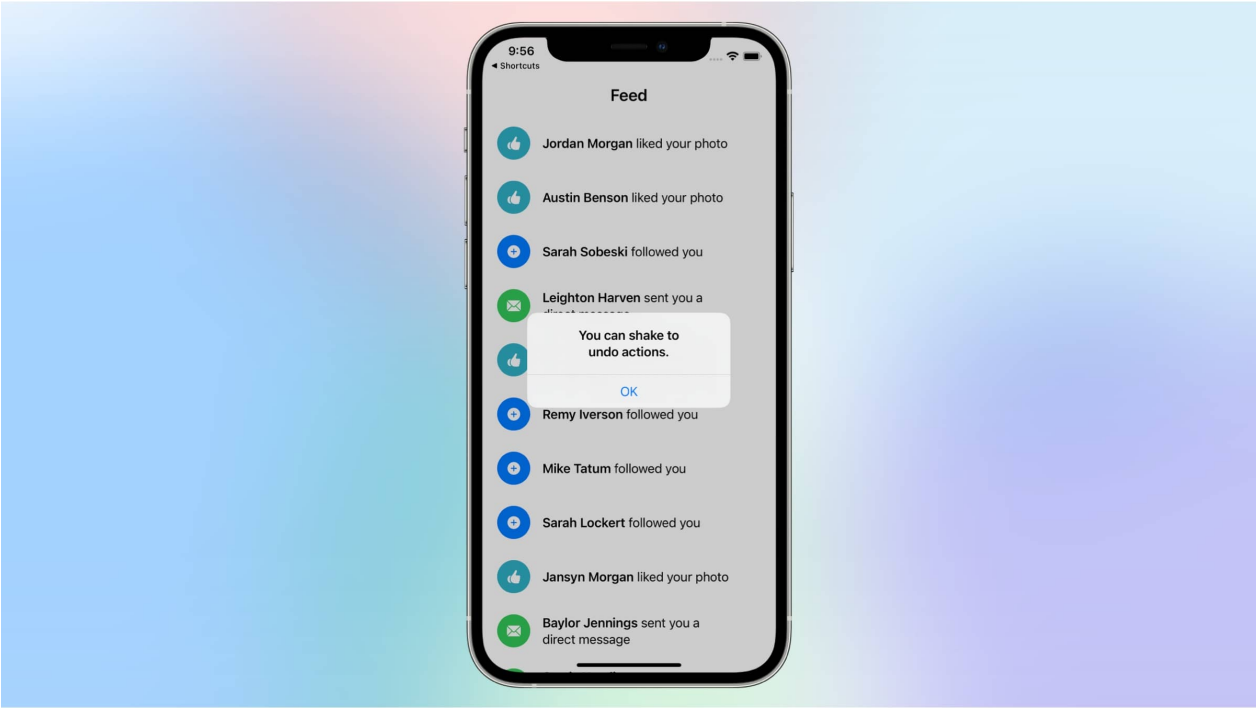
Reversible

We've all made mistakes in an app, and reversible methods are critical to provide. In fact, iOS itself contains undo and redo APIs that programmers can leverage by either shaking their device or using gestures. Both are important to support, but unfortunately – hardly anyone outside of the technology industry seem to be aware that they exist.

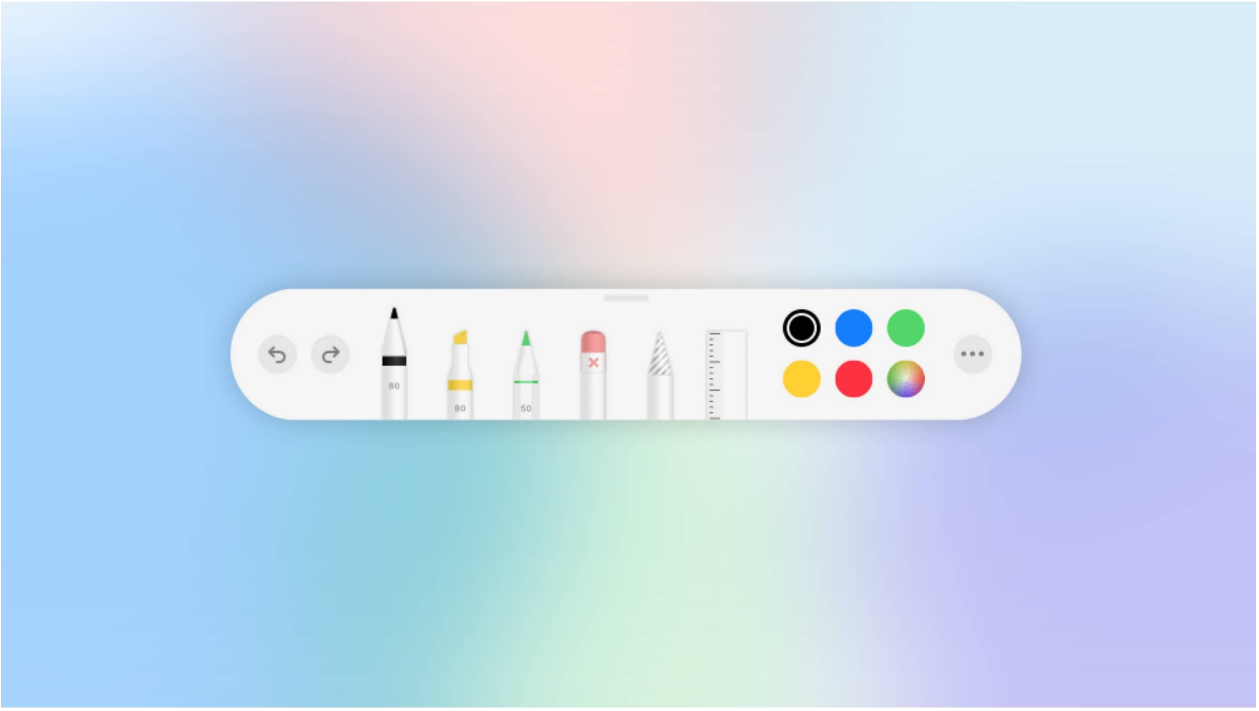
In such cases where a user has done something that they may want to reverse or undo, let them know that they can. For example, the popular Pro Create app coaches users that there are gestures available to perform undo and redo actions:



Where you place things like this can vary, and I've even seen some apps simply promote the system-level undo and redo technologies after they've done something that could warrant an undo:



Or, in the case of using the Apple Pencil – it's a dedicated button that shows persistently:



The system tool picker for Apple Pencil has undo and redo buttons that are always displayed.

No matter how you go about it, this type of reinforcement is critical. And it's critical for much of the same reasons you see a lot of apps offer things like an archive to find recently deleted or completed items, a grace period before permanent deletion such as Photos in iOS and other similar flows.

Let people undo their mistakes or reverse critical actions. The "how" isn't as important as you actually offering it at all.

Tactile

This type of feedback is much more subtle than the others, but it's also one of my favorite. You can use physical or digital techniques to bind a user's actions with your app and the device it's running on. The best example I can think of for this is iOS' haptics.

Think about when you use Apple Pay, and what occurs after the transaction goes through successfully. You get that delightful "ding" sound and a nice haptics pattern plays. It's a great technique that does both of the things we're talking about – it reinforces an action and provides valuable feedback.

You'll feel haptics all throughout iOS. The system controls use it out of the box, such as pickers when they change their value. As designers, we can leverage the same techniques in our app. Look for spots where a little bump, nudge or tap might help an interaction feel more lively, or activate it for reinforcement purposes.

A Caution on Overabundance

After reading all of this, don't jump into your app and start pumping in feedback and reinforcement actions everywhere. Like animations, a little goes a long way here. And, a lot of things – actually, probably *most* things, don't need it.

For example, if the user entered their name in a naming field, we don't need to throw a party and toss up banners saying "Name Entered Successfully!"

This is exaggerated, but you get the idea. Don't spam the user with annoying alerts that have no value, don't get in their way or block interactions while trying to reinforce things – and don't throw haptics in on every control you make.

Instead, apply tact and wisdom. Ask yourself, as a user – what kinds of things in your app would make sense to reinforce? What actions would they want to undo? Is there a way to provide important information about the state of the app that's out of the way, but also useful?

All of this is a skill, but it's also one you can learn and get better at along the way. The first step is to simply be aware of it, and start thinking about it.

The TL;DR

Feedback can be a beneficial way to reinforce interactions within your app, but remember that a little can go a long way.

Color

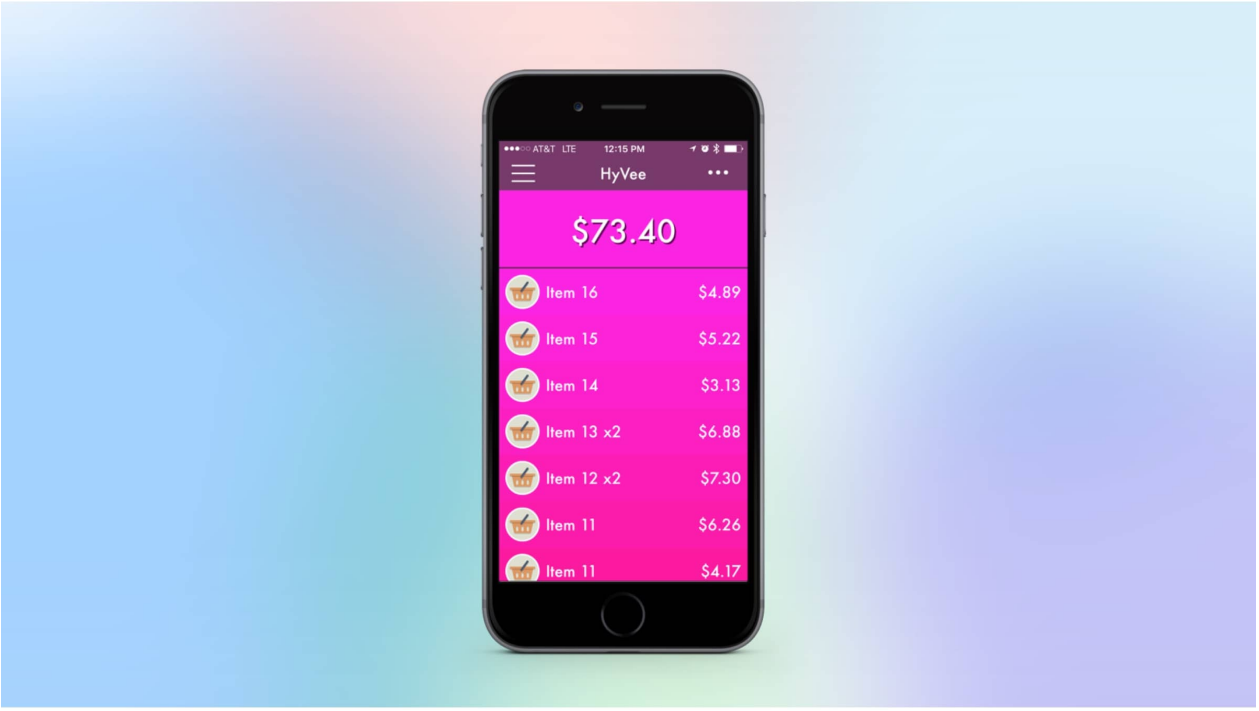
Colors are everywhere in app design. You can't hide from them. The good news? A little know-how on the matter makes it a lot easier to get them right. In fact, I could sum up all you need to know to become competent with colors in a few sentences:

- Get your typography colors right.
- Create pleasing hierarchy from colors.
- Don't over do them.

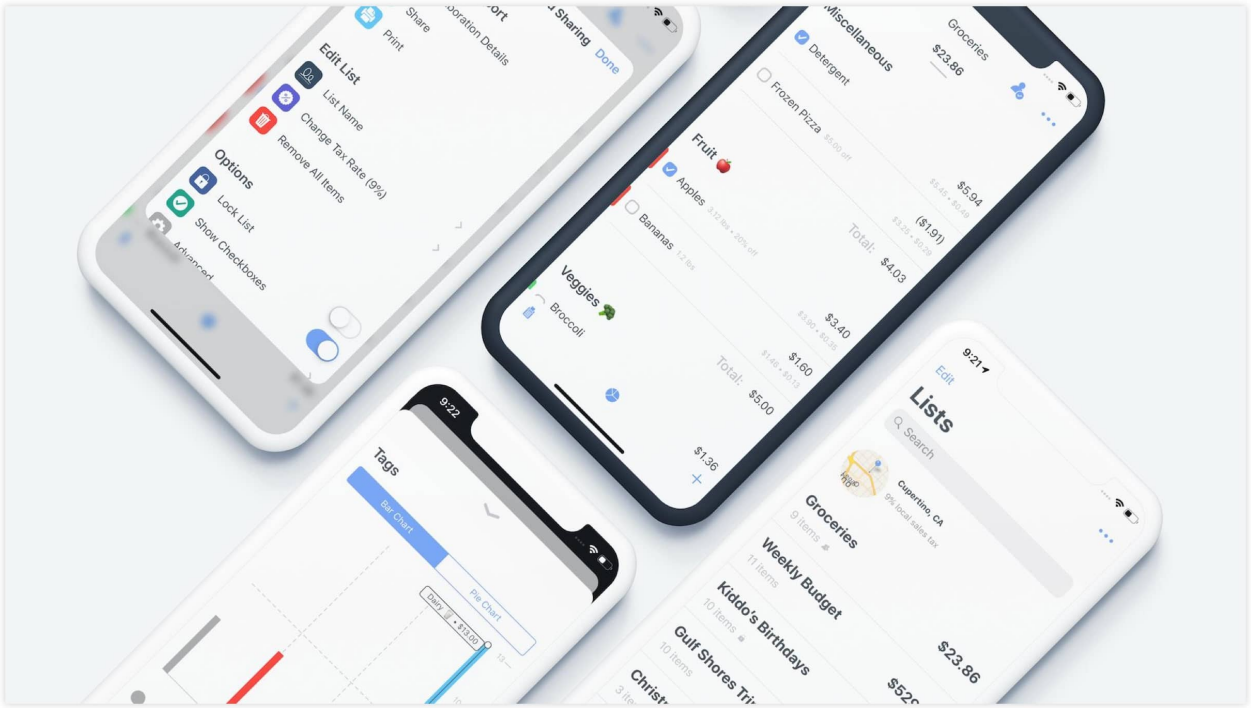
That's about it, if you follow those three rules, you'll do just fine when it comes to colors on iOS. The rest of this chapter more or less reinforces those notions.

If you're new to design in general, colors are a magnificent place to hone your skills. Anytime I talk about getting better about design with other developers or nascent designers, it's the first place I recommend they start. Why? Because it's the shortest route to drastically improving your app's design.

As a refresher, my first app Spend Stack had a lot of issues with colors and competing contrasts. Here's what it looked like:



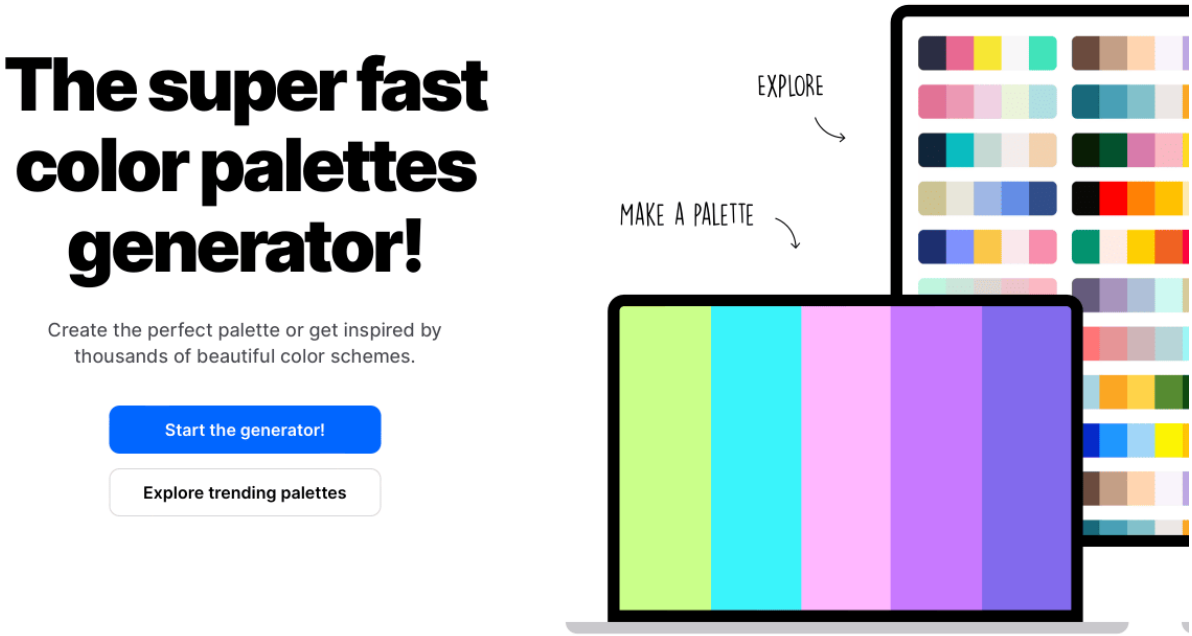
And here is where it ended up several years later:



As you can, simply tweaking typography and color can make an objectively bad looking app look nice. Let's dive into the particulars.

iOS is Unique

One lesson I think you should apply to colors on iOS fairly quickly is that rules you've learned about colors on other software platforms may not apply as much. For example, on web – you'll come across designs with expansive color palettes or tools to generate a swath of them:



Web design usually calls for a large density of colors.

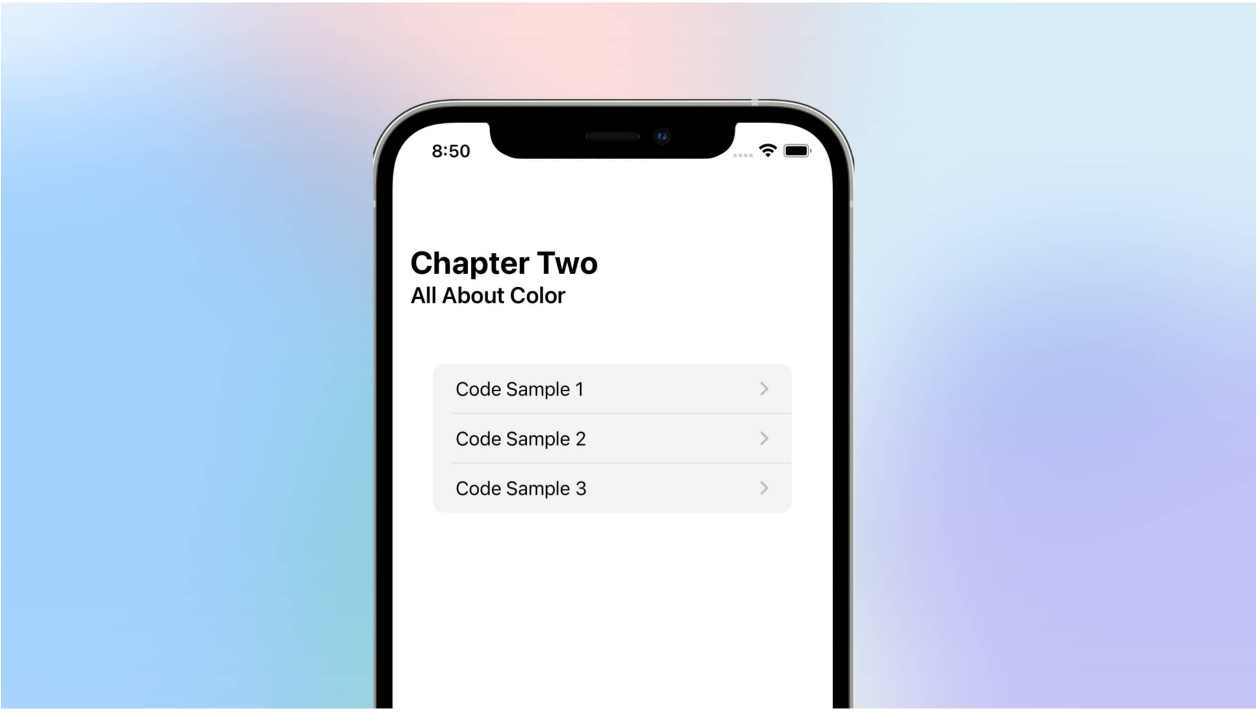
I think that amount of depth on web can make sense, but designing for iOS is a bit different. Namely, you'll have less colors to deal with typically. It's not just the web, either. Software on adjacent Apple platforms have different lessons and degrees of variance in their colors, such as macOS or tvOS.

That's certainly not to say you don't want to take the lessons you've learned from other platforms over to iOS. Things like color theory and balance all still apply. I think the primary thing here, though, comes down to one word: less.

You'll probably have less colors to deal with, and that's completely fine. There are upper and lower bounds to that rule, of course — but overall, if you're finding yourself managing fewer color options on iOS, that's a sign that you're probably on the right path.

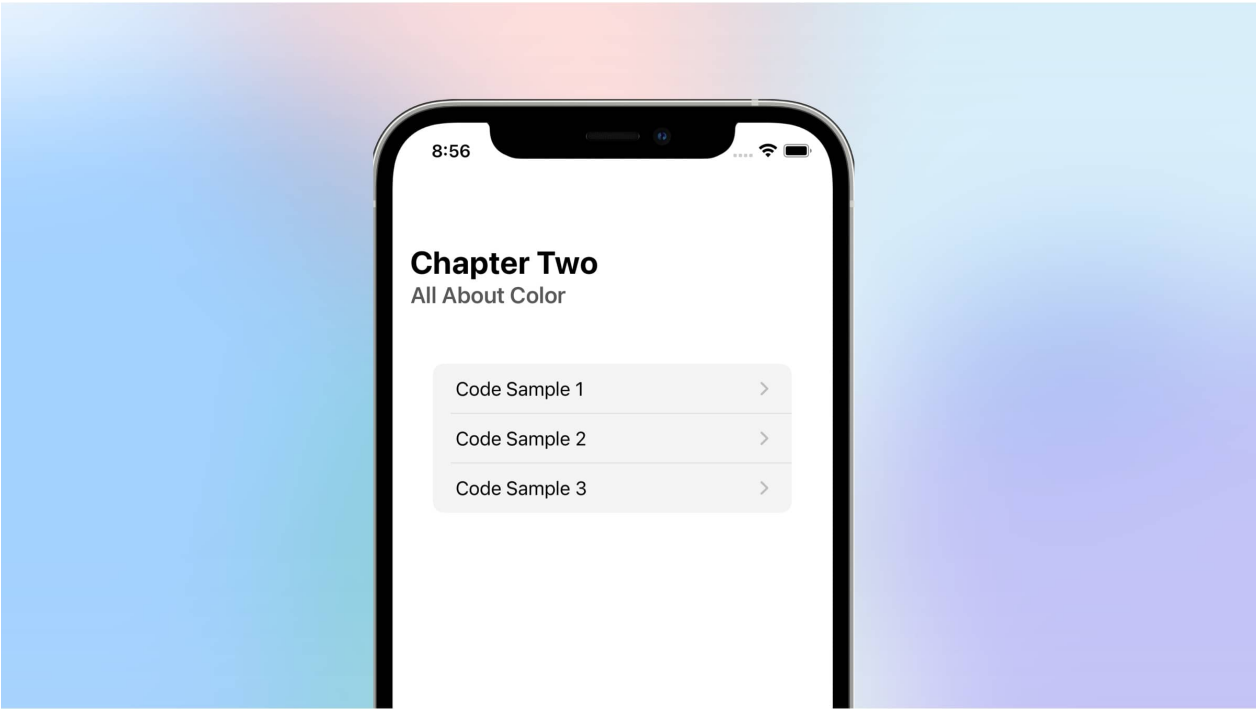
Starting with Typography

As I alluded to above, the place to start with colors (beyond your branding) is typography. Let's start with a humble example that has default colors by iOS:

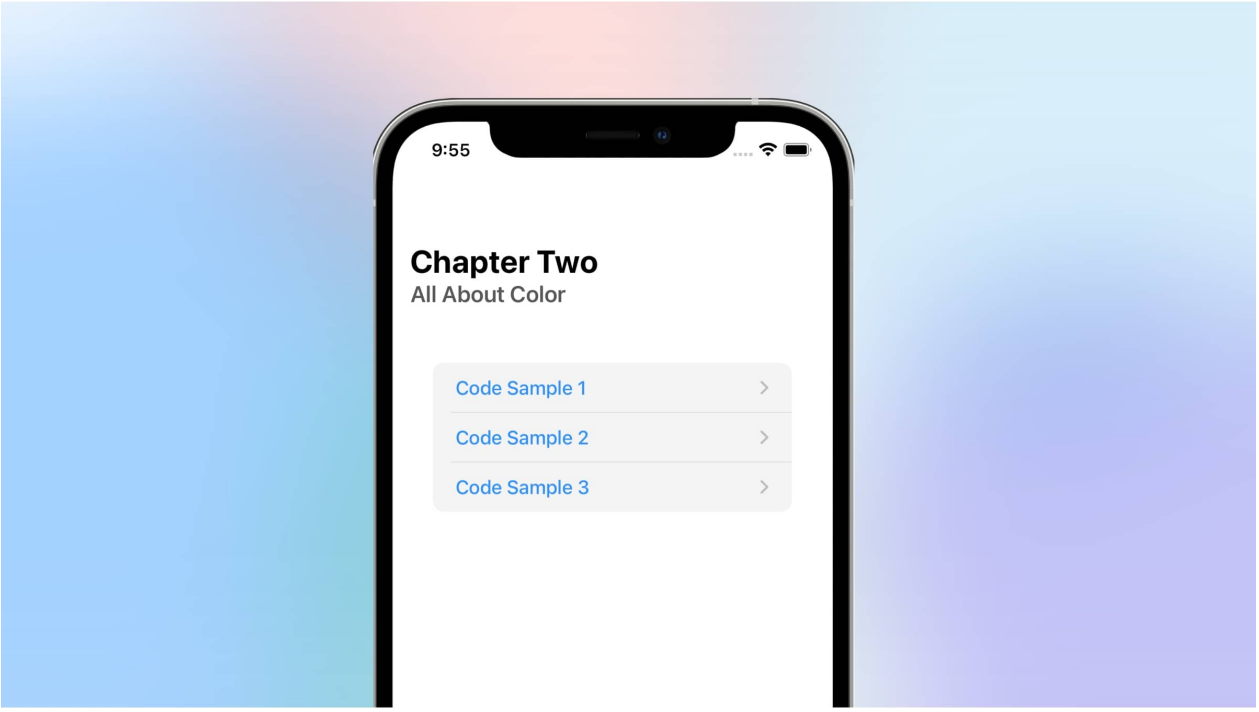


The color you choose for typography determines how users will visually navigate the hierarchy. For example, for our most bold, eye-catching text (i.e. the header here), a black works well.

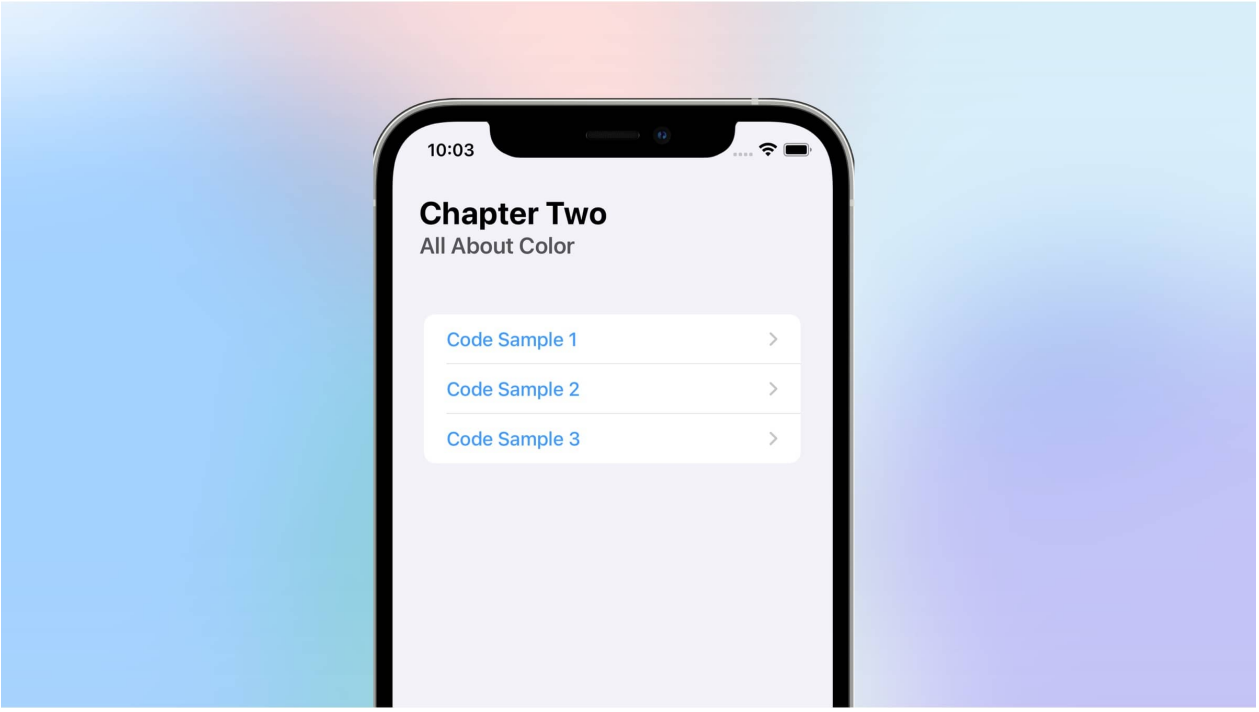
So, we'll leave that as is. But, for the subheader we could do a few things to signify its role. We could give it a smaller font weight or size, but since we're dealing with color – I like to make it a few shades lighter than its header counterpart:



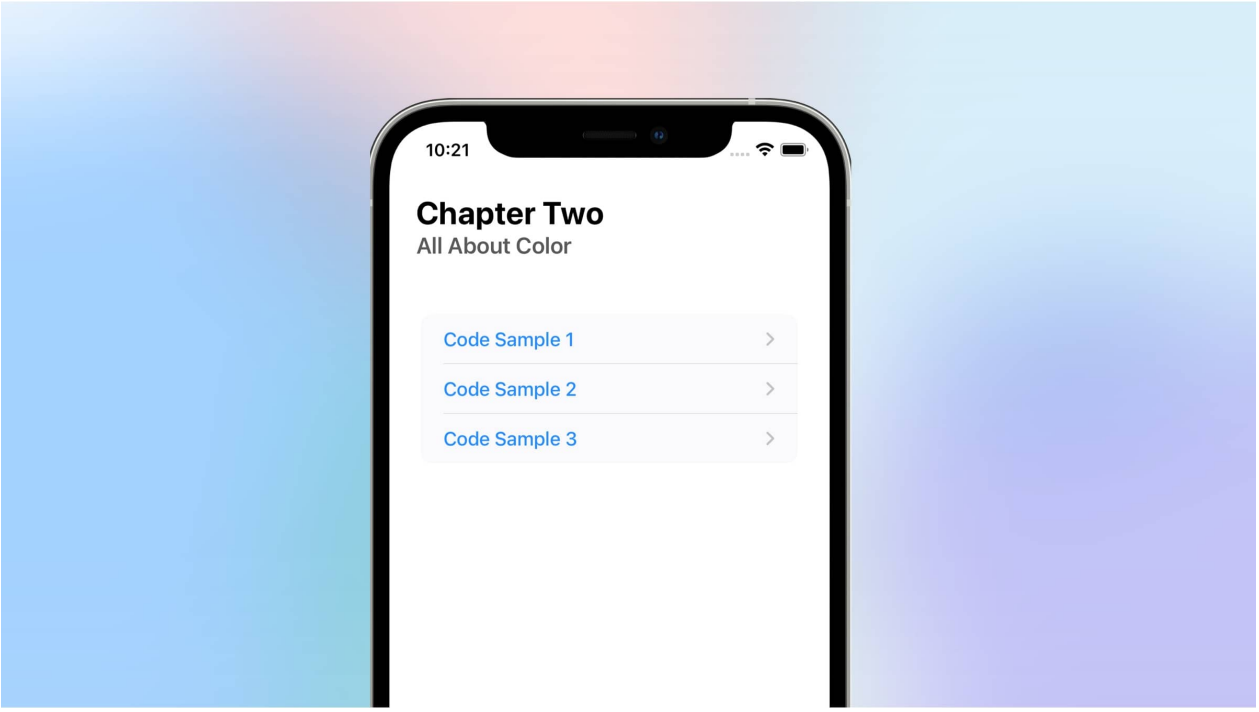
Next, we could apply our brand's color to the app's tint. Since the rows are not purely informational, but actionable, we could leverage our app's primary color as the tint to better show they are tappable:



And lastly, though this isn't explicitly typography – we could make the buttons a little friendlier by using color to promote the grouping of the content. If we use a gray for a primary background color, that'll open up the interface to use a soft white for the row backgrounds. Depending on your primary background color, this can make the grouping of content more obvious:

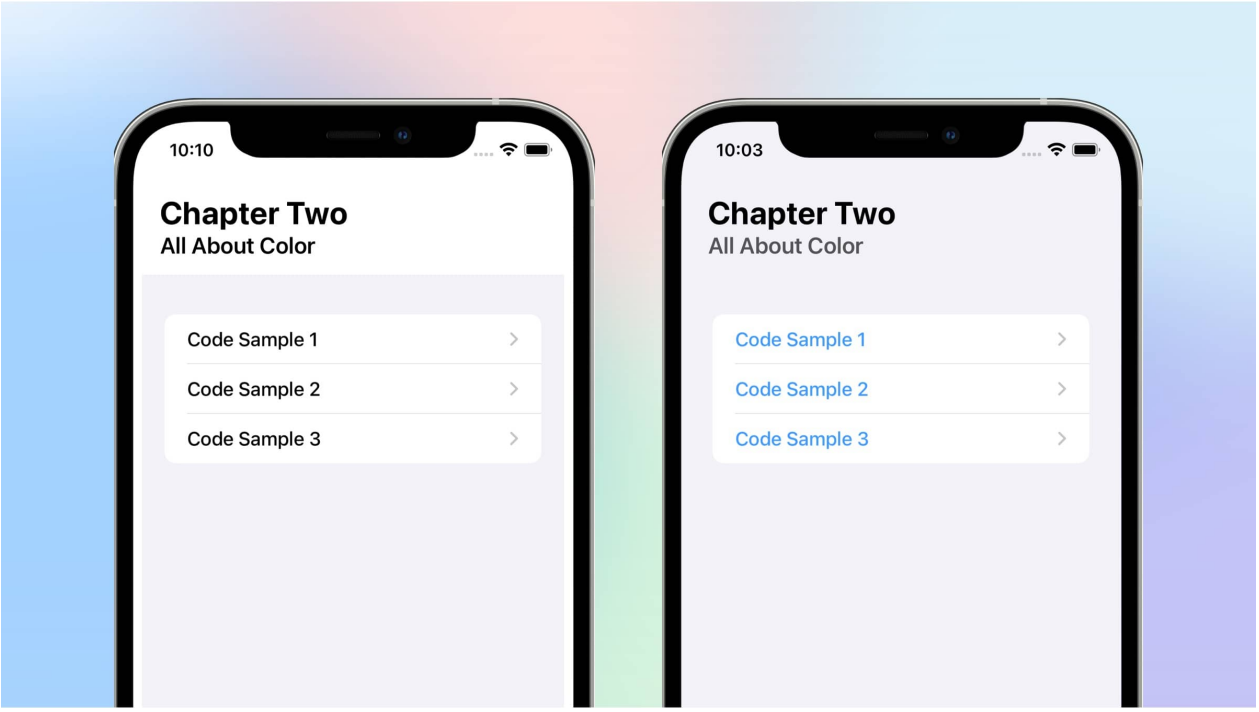


However, there are a lot of times where you may want white background, which puts you in somewhat of a bind for your selectable rows. In those cases, I still typically lean towards a soft gray variant. This is similar to where we started, but the rows aren't as harsh as they were with the default gray:



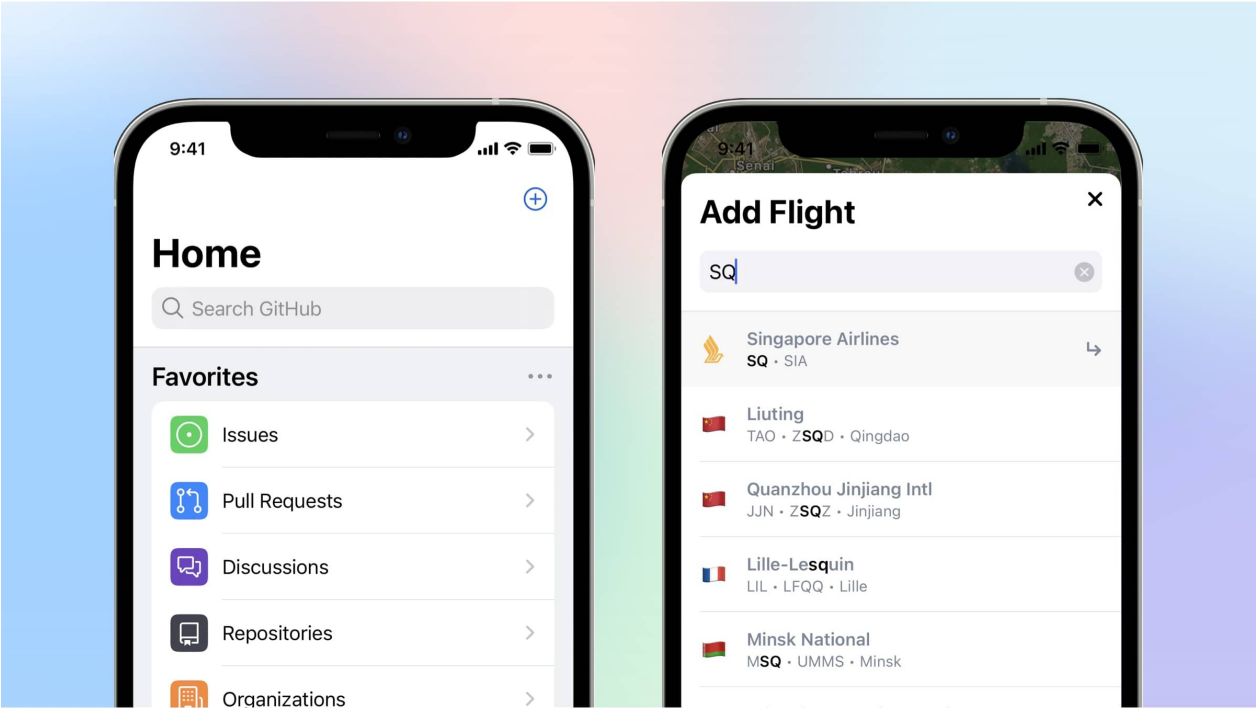
Depending on where that screen would take you, and all that this view would need to contain – there is still more work to be done. But, you can see that by simply tweaking the colors of an interface, and otherwise keeping everything else the same, colors have a dramatic impact on how things feel.

Here's what it looks like by default, without changing any colors, versus where we went with it:



The same design – just with different colors.

Getting your typography colors takes you a long way. Don't be discouraged if things don't like great at first. To get over the hump, try to decide what your background color is, because that'll directly inform how the chunks of content should look within it. For example, Github for iOS uses a grouped background color and a white background for its rows, whereas Flighty does the opposite:



If you can figure out what color you want to use for these four things:

1. Heading labels.
2. Subheading labels.
3. The main body of text.
4. And, description or callout text.

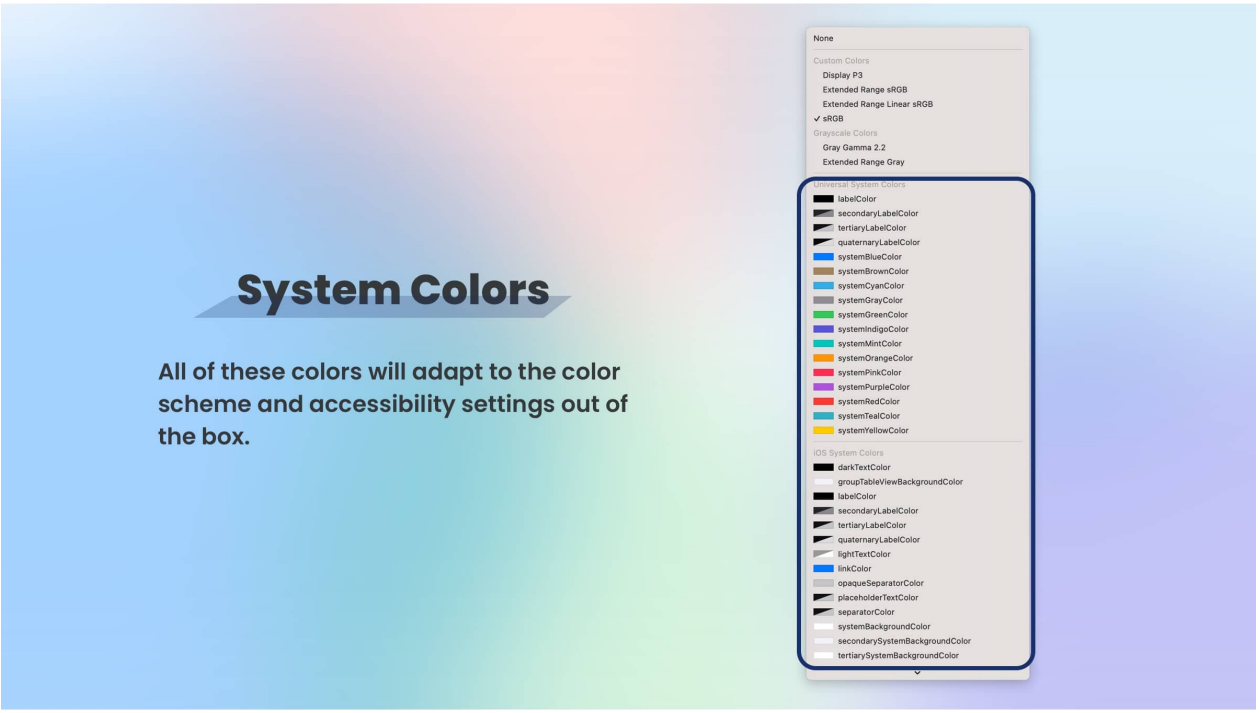
…you're going to solve a lot of problems and inform the rest of your design right out of the gate. Start with your text. Maybe the only other thing to skip ahead on might be the containers that hold the text (insofar as their background color).

What I've seen, time and time again, is that if I can get to a place where I love the color of my typography – everything else just seems to fall in place *much* easier.

Semantic Colors

Since iOS 13, iOS has leaned on the idea of semantic colors. The goal of these colors is to describe what they do and what situations they might be used for instead of just simply being “red”, “#FFFFFF”, or an HSL (hue, saturation and lightness) value.

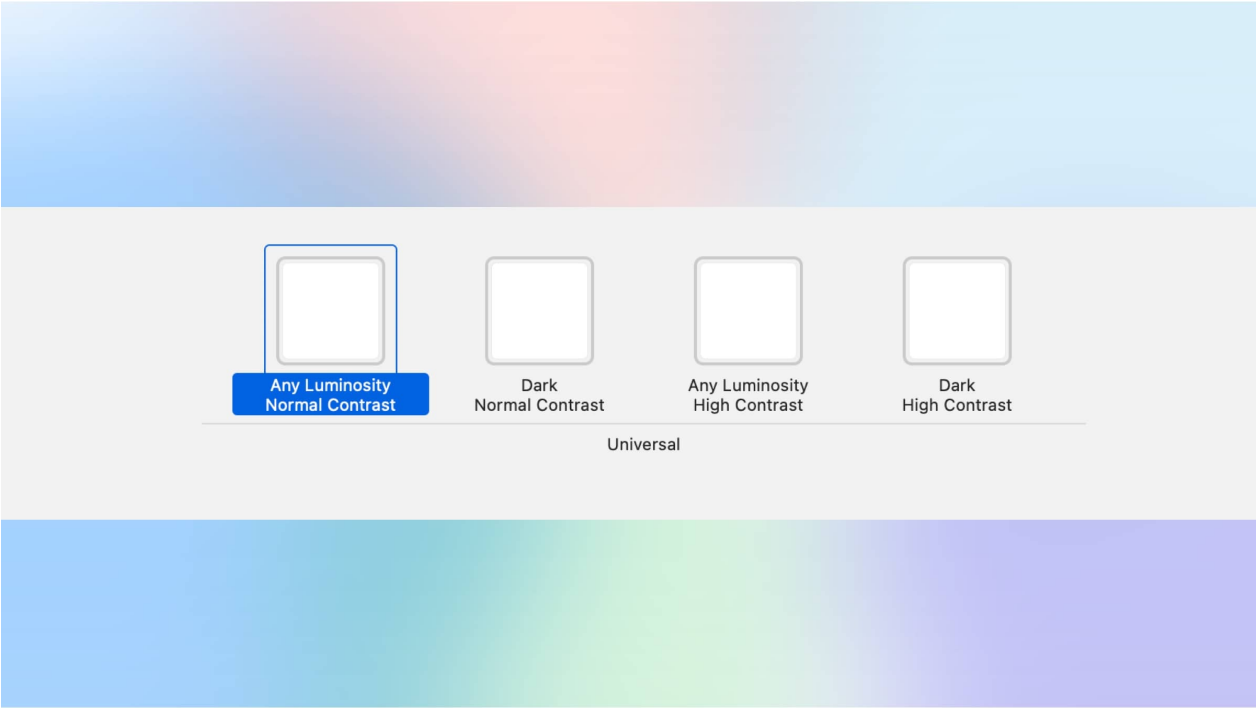
Instead, you refer to them as “label”, or “secondaryLabel”. iOS comes bundled with a multitude of these, and they are a fantastic starting point for any app:



You can create some of your own using Xcode’s asset catalog. In fact, that’s the preferred way to handle things for a few reasons:

1. You can specify the semantic meaning of the color.
2. You can supply values for light and dark modes.
3. You can specify different colors for accessibility settings.

Given those constraints, your one color is now four different colors:

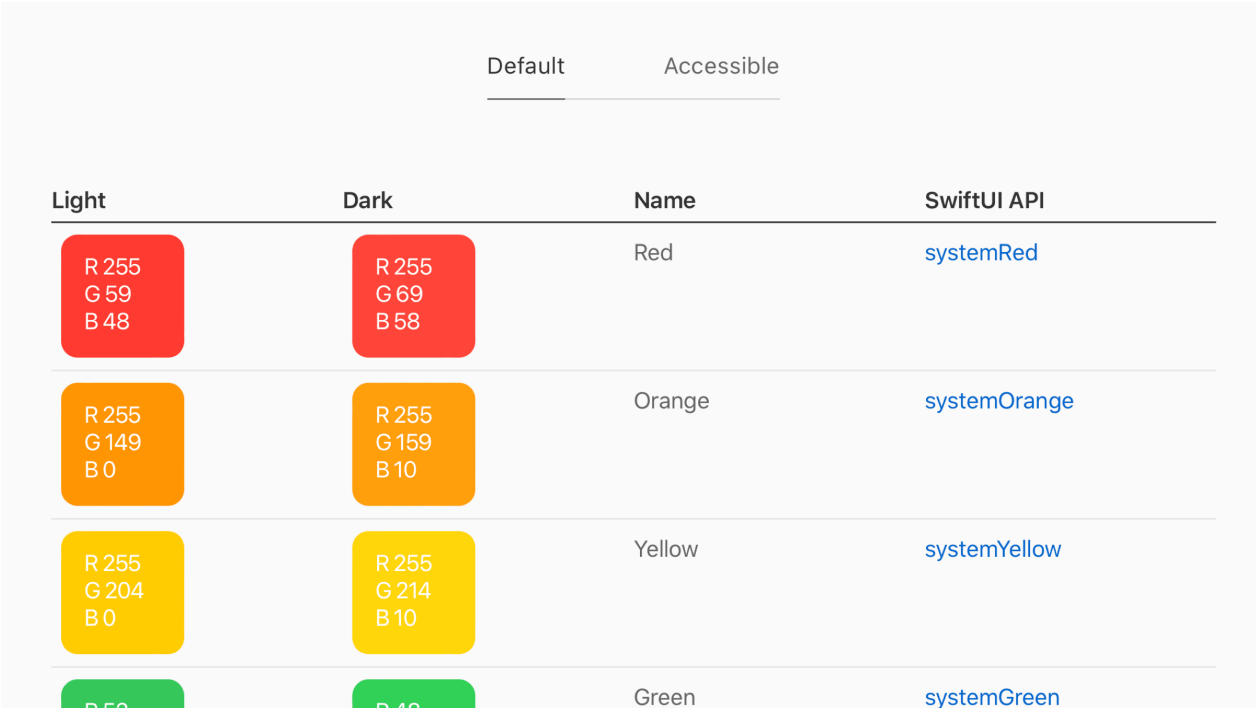


That's a lot easier for you and your developers to use than the alternative such as labelLightMode, labelDarkMode, labelHighContrastLightMode and labelHighContrastDarkMode. That's not scalable and the mental model isn't easy to follow.

So, when designing your interface – opt to keep things in terms of semantic colors. Even if you are privy to a certain gray (or grey, depending on where you're reading this), you can still package it up as a semantic color.

Apple does this. With all of this discussion of semantic colors, you might think Apple did away with including any system level reds, oranges or any other color. That's not so, in fact – the opposite is true. They've given the "semantic power ups" to each of those.

Case in point – for the system level red, Apple has actually given us four different reds (the two in default, and two in accessibility):



Dark Mode

Continuing with the idea of "dynamic" colors on iOS, we arrive at dark mode. Dark mode isn't nearly as challenging to design for thanks to the way semantic colors work, but there are a few things to remember about it when designing apps. The first, and most important, is to *actually do it*.

Seems obvious, doesn't it?

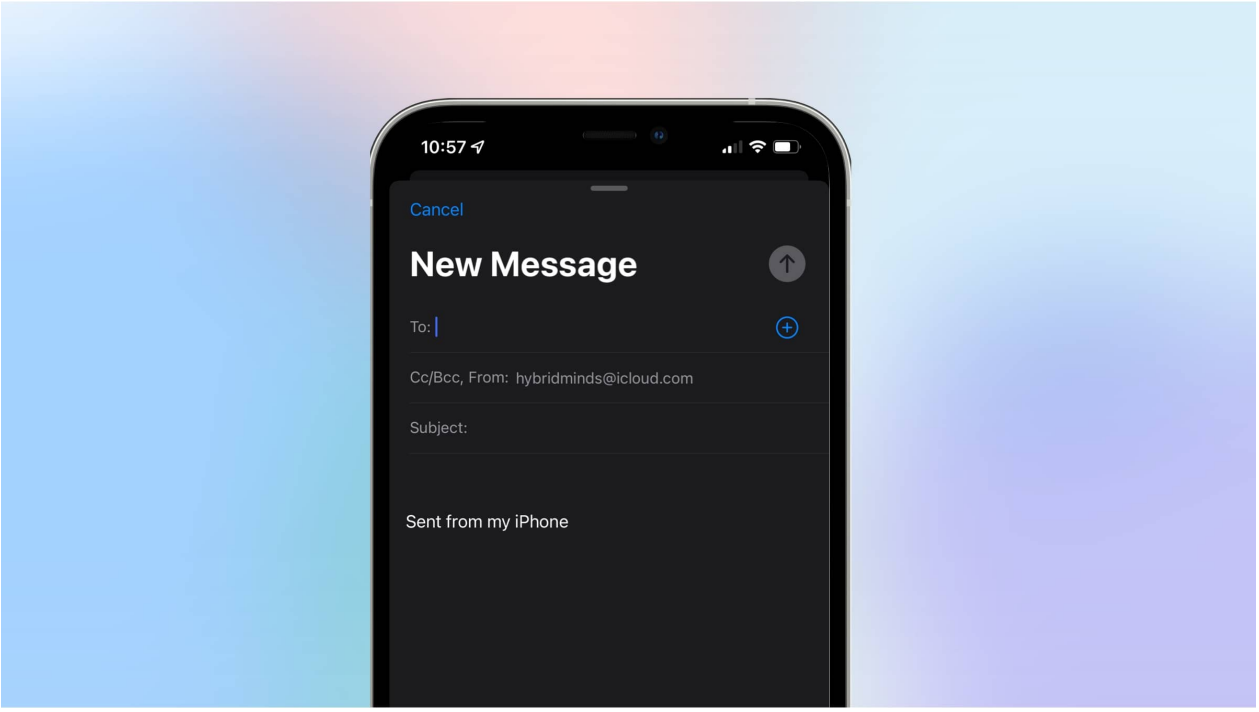
But, it's usually is treated as an afterthought. I've worked with designers who create high fidelity mock ups in the default setting of "light" mode, and then don't do anything for dark mode. It's not out of spite or laziness either, it's just not often part of the process – and it absolutely should be.

Dark mode isn't a second class feature on iOS, it's prominent all throughout. People **will** use it. The choice to keep it on is even part of the process of setting up a new iOS device. Further, it's also trivial to toggle on from iOS' Notification Center.

And, well, people just like using it for aesthetics or they rely on it for accessibility purposes. So design for it, and start designing for it from the beginning while testing it on devices. You'll find some contrast issues fairly quickly, which seem to be the easiest mistake to make with colors in dark mode.

Beyond that, keep in mind iOS can tweak color values on the fly based off this idea Apple refers to as elevation. If you're working with developers, you can refer them to `UIUserInterfaceLevel`. But, as a designer, the impact for you is that depending on a control's elevation, its color may change.

For example, in Mail, both the compose sheet and the view behind it use the primary background color. So, with the compose view being "elevated", it changes ever so slightly so that depth is maintained in the design and the two colors don't clash:



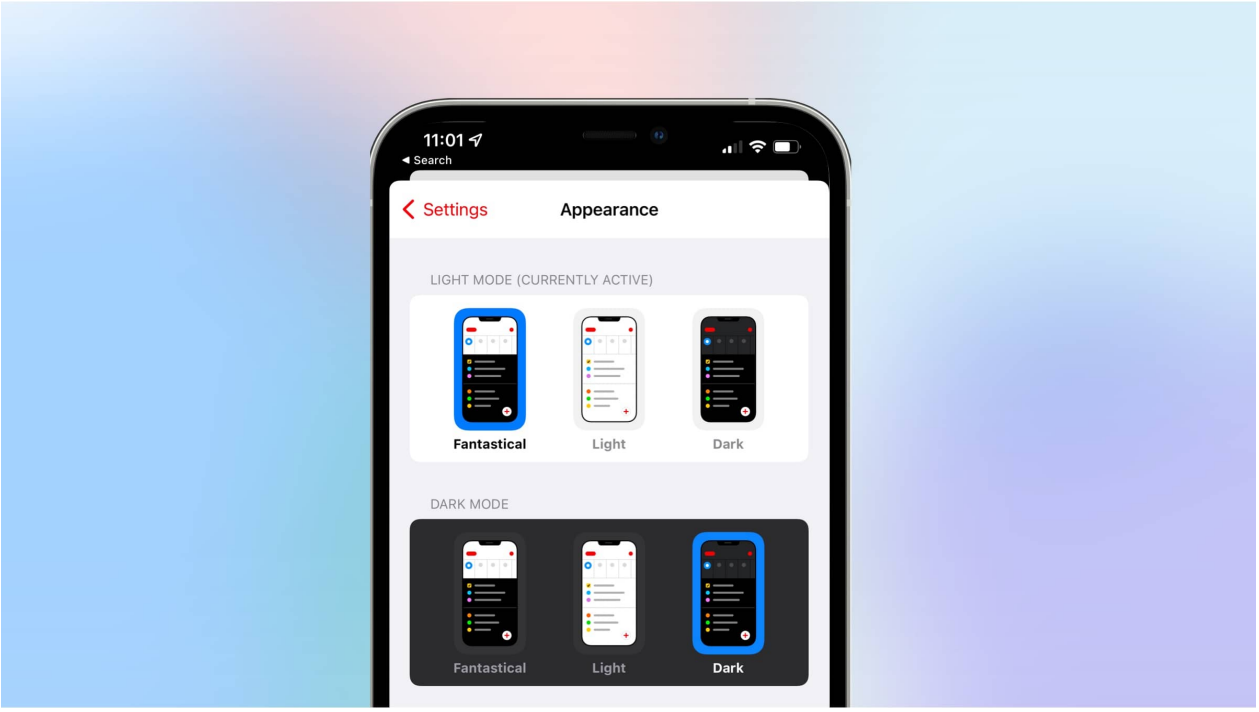
It's a bit hard to see on a screenshot, but if you've got an iPhone nearby – compare the black from the main view with the black of the compose sheet with dark mode on to see it a bit easier.

Themes

There are some apps that provide an app wide theme option. As designers – these are fun! You get to go a bit crazy and pursue a lot of ideas.

From an implementation perspective for developers – it's very complex and technical to get right. It's an exercise of patience and fighting a lot of edge cases. So, if you feel that having a theme toggle is critical to your app – make sure you developers know early. And then, buy them lunch.

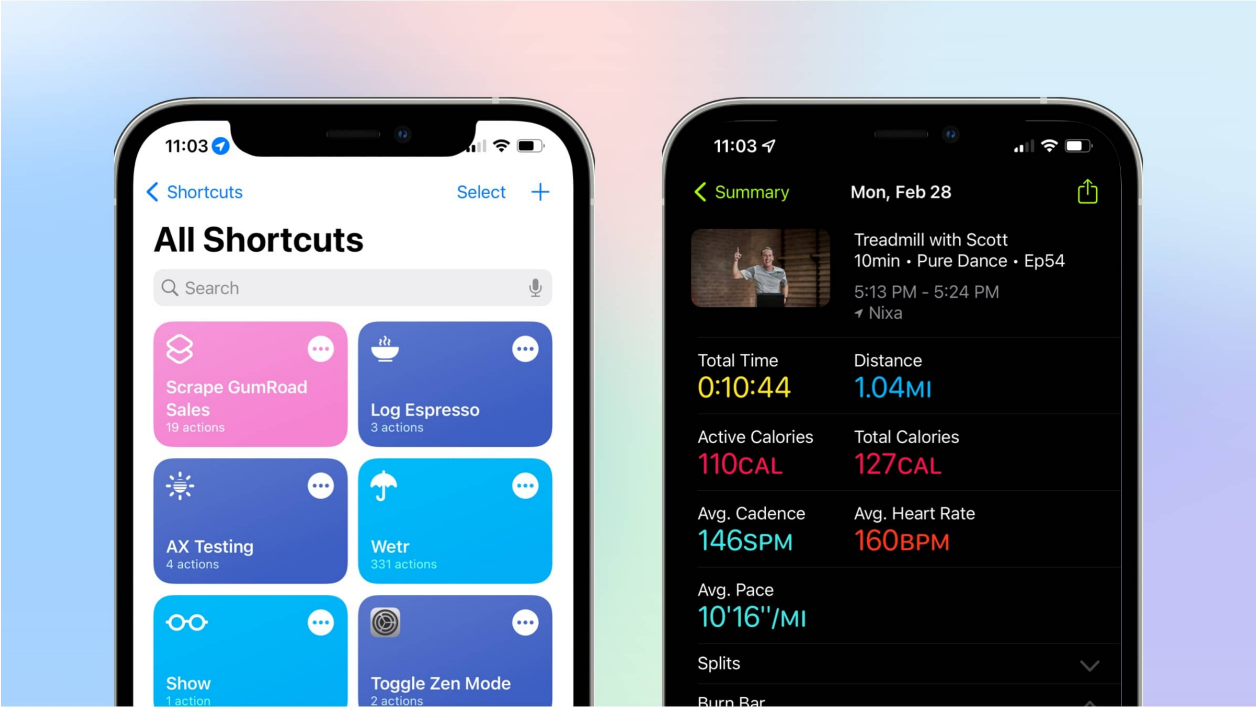
Now, you may have seen some screens like this and might be wondering why I'm suggesting that it's so hard to pull off. For example, here is one from the popular calendar app, Fantastical:



This is *not* what I mean by themes. Following this pattern is fairly trivial on iOS for developers because the system is built to explicitly work this way. It's a way to always have dark mode or light mode on, or follow the system. While I typically advocate for following the system, if you want to design a screen like this – it shouldn't give your developers any headaches.

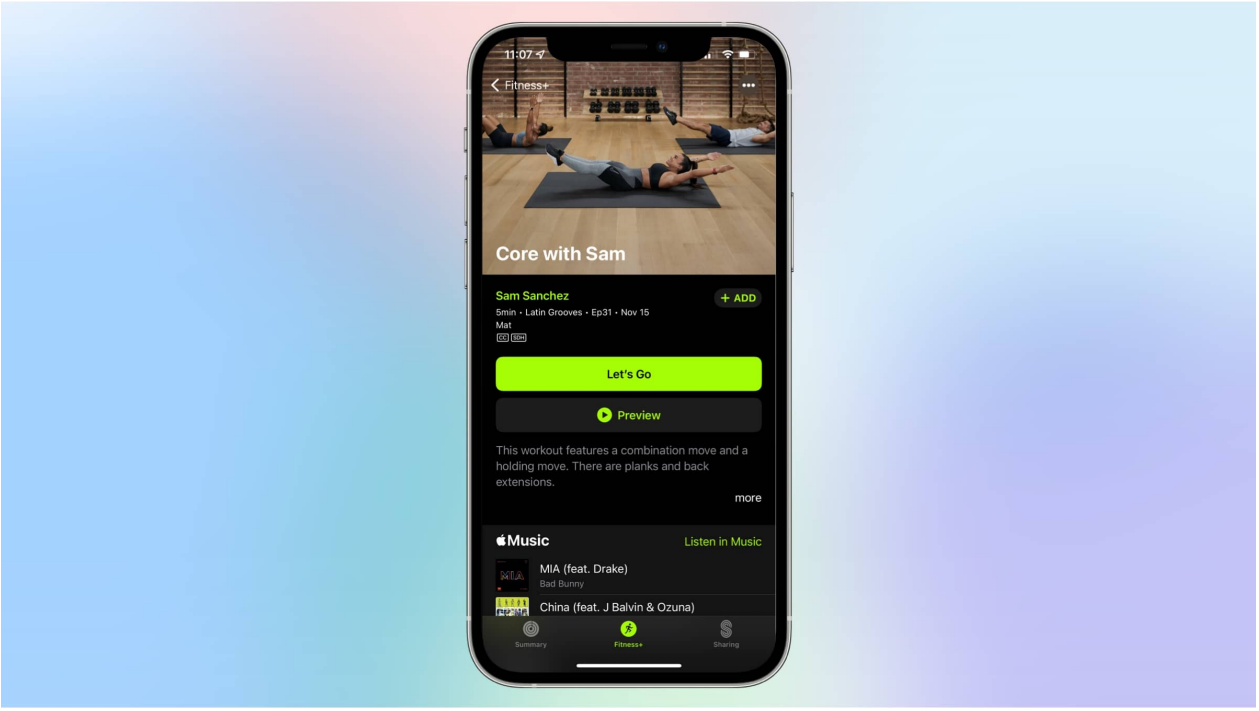
Tinting and Accent Colors

iOS apps have a notion of a tinting colors, and it's used in several places throughout the system. For example, the navigation bar back bar is a place that iOS applies a tint. In Shortcuts, it uses the default blue. In the Fitness app, it uses a green:



This is a great place to bring in your brand. Whatever its primary color, it's likely a good fit for your app's tint color. Keep in mind that you can also provide two different versions of this color for light and dark modes, but they shouldn't drift apart too wildly. Recall our discussion above over semantic colors, and how you might need to use separate values for accessibility settings or dark mode.

Beyond that, I also like to use the tinting color as a primary button fill or text color if you've opted to use your brand's color. Let's go back to the Fitness app, they use the that neon green color that's used for tinting as a primary button fill too:



Accessibility and Internationalization

When choosing colors, bear in mind the many accessibility and territorial implications. Here is a quick list to consider:

1. Contrast: On iOS, the gold standard is a 7:1 contrast ratio. If you stick to the system's semantic colors, you'll achieve that quite easily. But even then, there are certainly combinations of colors that won't meet that criteria. Be sure to check your contrast values wherever you're designing, or use the Accessibility Inspector's color contrast calculator if you're testing things about as a developer.
2. Color Blind Users: Remember to consider color blind users. Not every color will be perceived the same by everyone – or your intended colors could

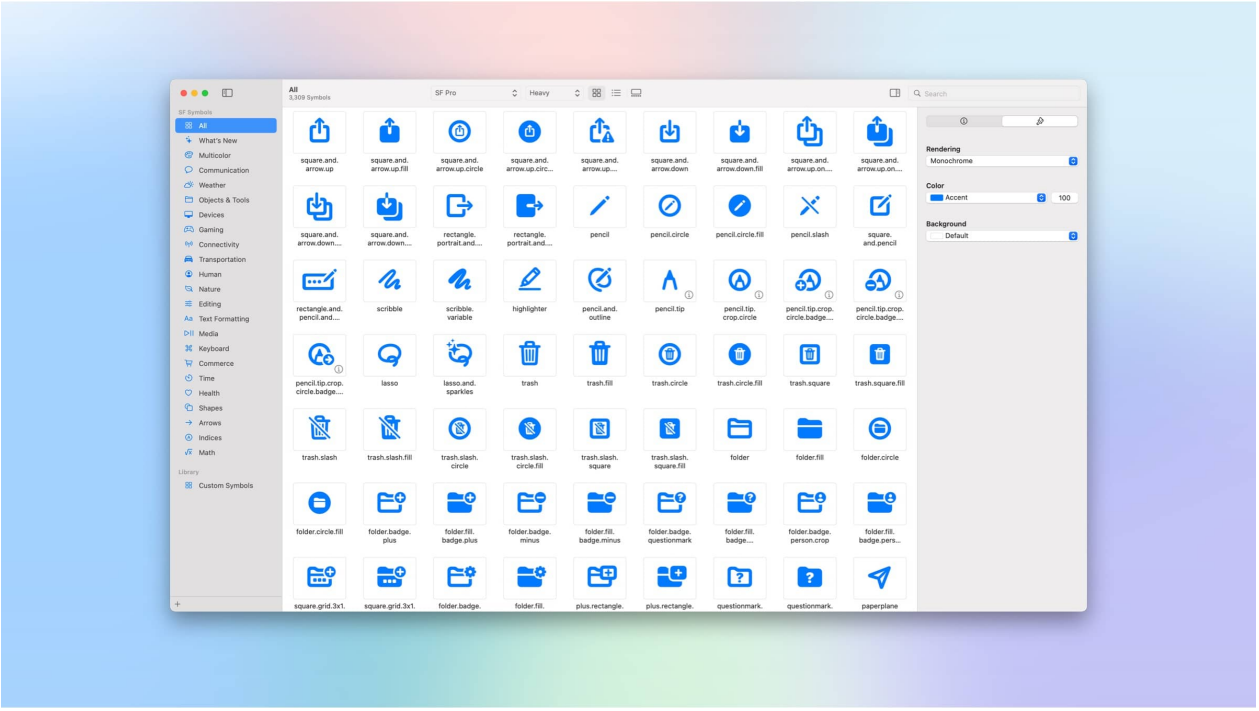
be changed themselves. For example, users can enable the "Color Filter" option on iOS to apply different color tints

3. Localizations: In different territories, color meaning can vary drastically. For example, in Western cultures – red usually means danger. In other territories it carries a calming vibe or even represents positive outcomes. So, for that reason – don't rely on color to convey meaning on its own. Instead, opt to pair situations like these with iconography too (i.e. a trashcan icon along with a deletion action).
4. Other accessibility settings: At any point, someone could enable increased contrast or they may want to reduce transparency. Consider how those settings will change your design but most importantly – ensure they will work with the colors you've chosen and if they don't, supply different ones for those situations.

SF Symbols and Rendering Styles

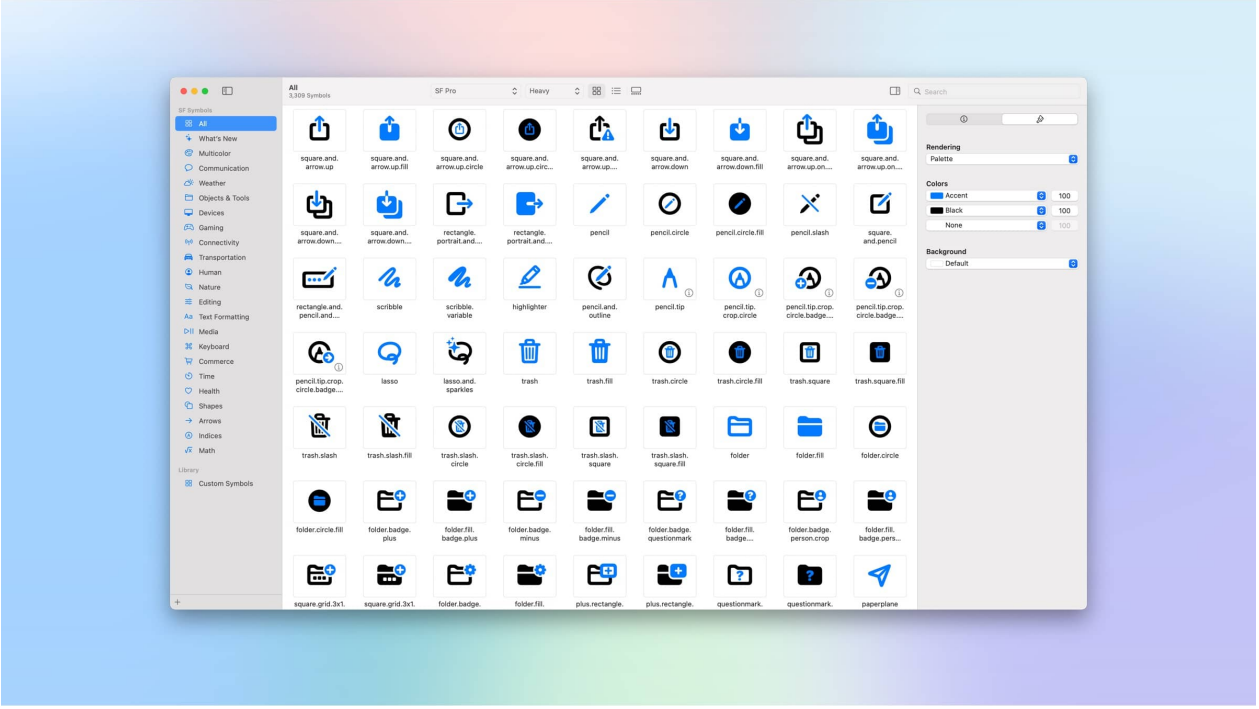
Designing for iOS in terms of iconography is great, because Apple themselves give us a huge shortcut: SF Symbols. These icons adapt to the system interface style (i.e. light or dark mode), align with fonts and can even change their weight according to various settings.

Another advantage? You can apply colors to them as well:

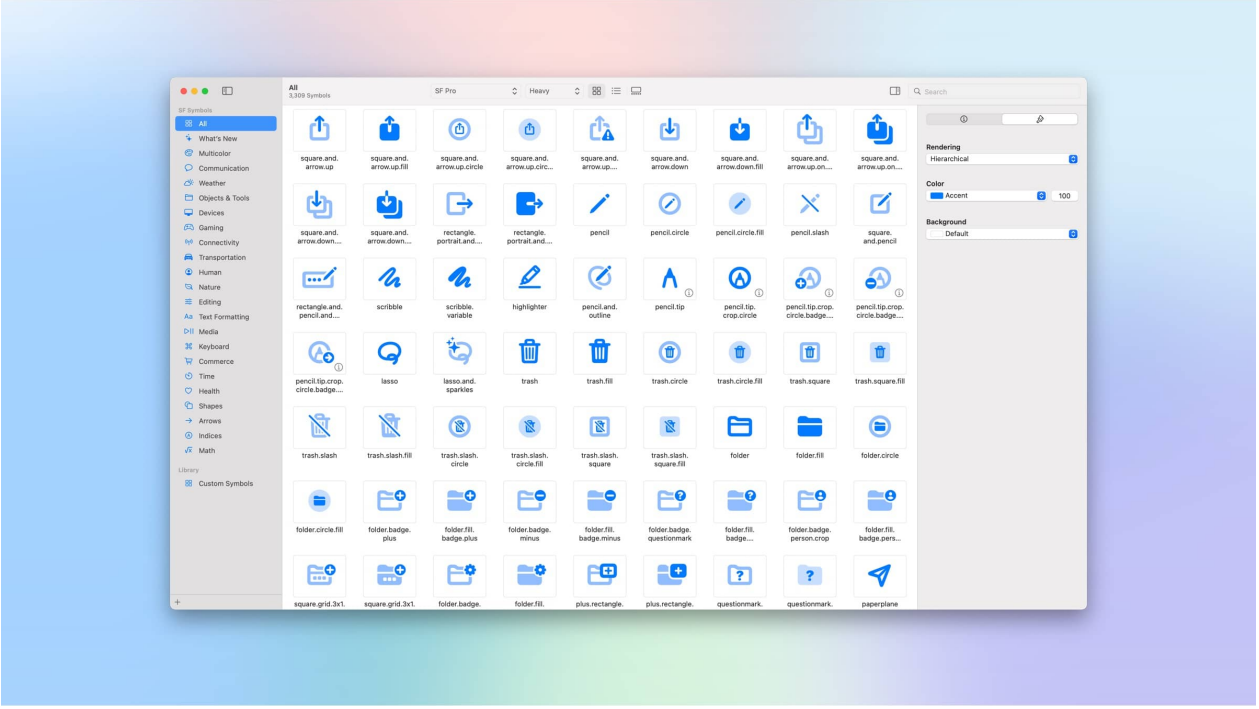


SF Symbols can be customized several different ways.

Further, some of them even support multiple color options:



You can also have them rendered a few different ways to get different looks:



To play around with different variations, be sure to download Apple's free macOS app, SF Symbols, to get a feel for them or just to prototype some icons for your design.

The TL;DR

Colors play a huge role in iOS design, they don't take a backseat. Take some time to get them right, and try starting with typography if you're stuck.

Copy

Copy, or the overall representation of text, in apps is part of their design. We don't generally group the things we write with other disciplines as a design component, but I think we should. Typography gets a lot of attention, colors do too — or which control to use along with the W's (when, where and why).

All important, no doubt. But, there's not a lot of meetings happening over what text to put in an alert letting someone know a network request failed. Or, how to communicate with people about a new feature.

But it is a design discipline, and getting it right will make your app stronger. Getting it wrong is one of those "stand out" design mistakes, people will notice.

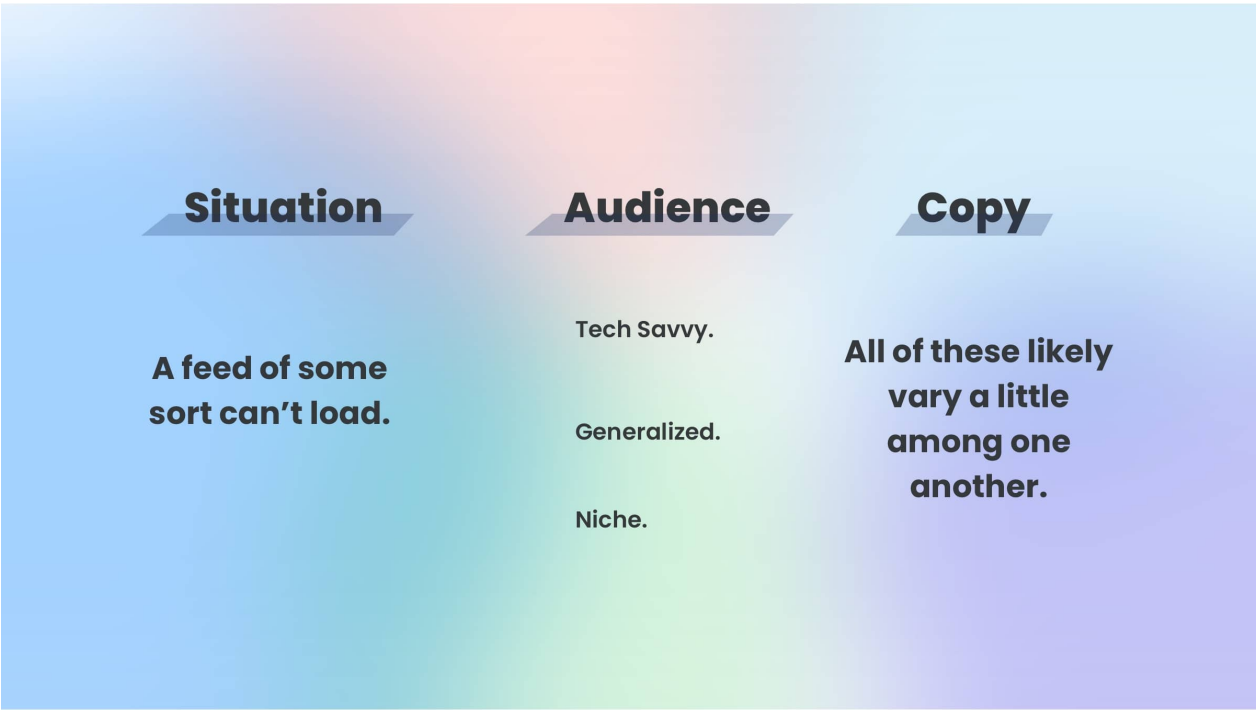
This will likely be the shortest chapter in the book, but I truly believe it's an important component to install into your design repertoire. Even though there's not *a lot* to say about it, what is there — is important. So to that end — let's go over some basic rules you can take with you to make sure your copy works.

Remember Who You Build For

The first part of designing effective copy in your app is to remember who you're building for. This goes hand in hand with getting your app's voice right, something we've

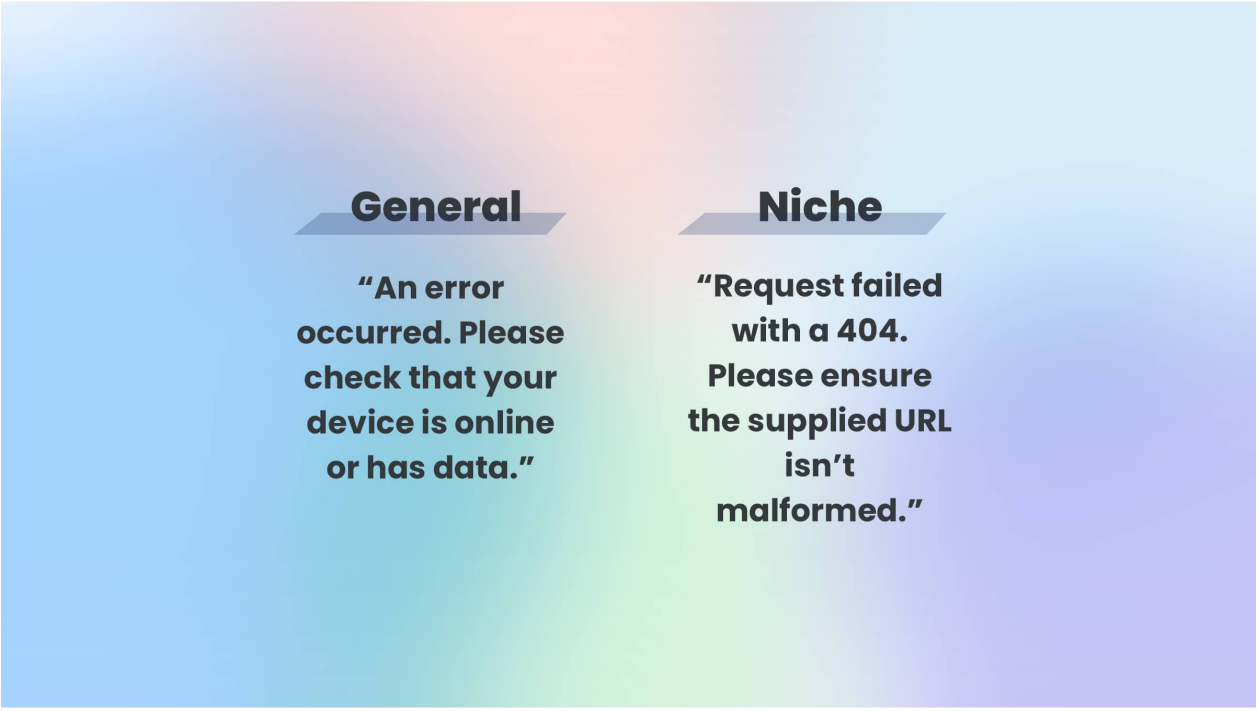
talked about previously. When you know who you're building for, you have an idea of how to talk to them.

For example, something that happens in every app – bugs, should be handled with certain copy depending on who you're talking to:

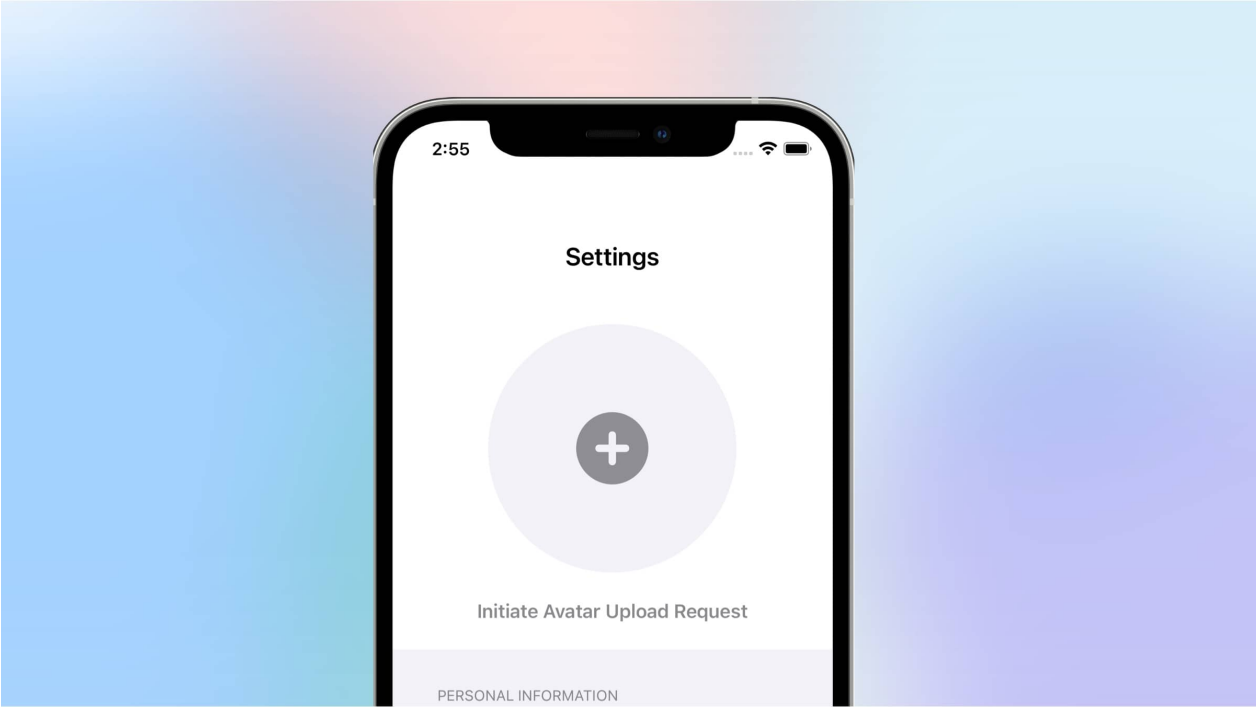


The way you talk to people should reflect the audience you're building for.

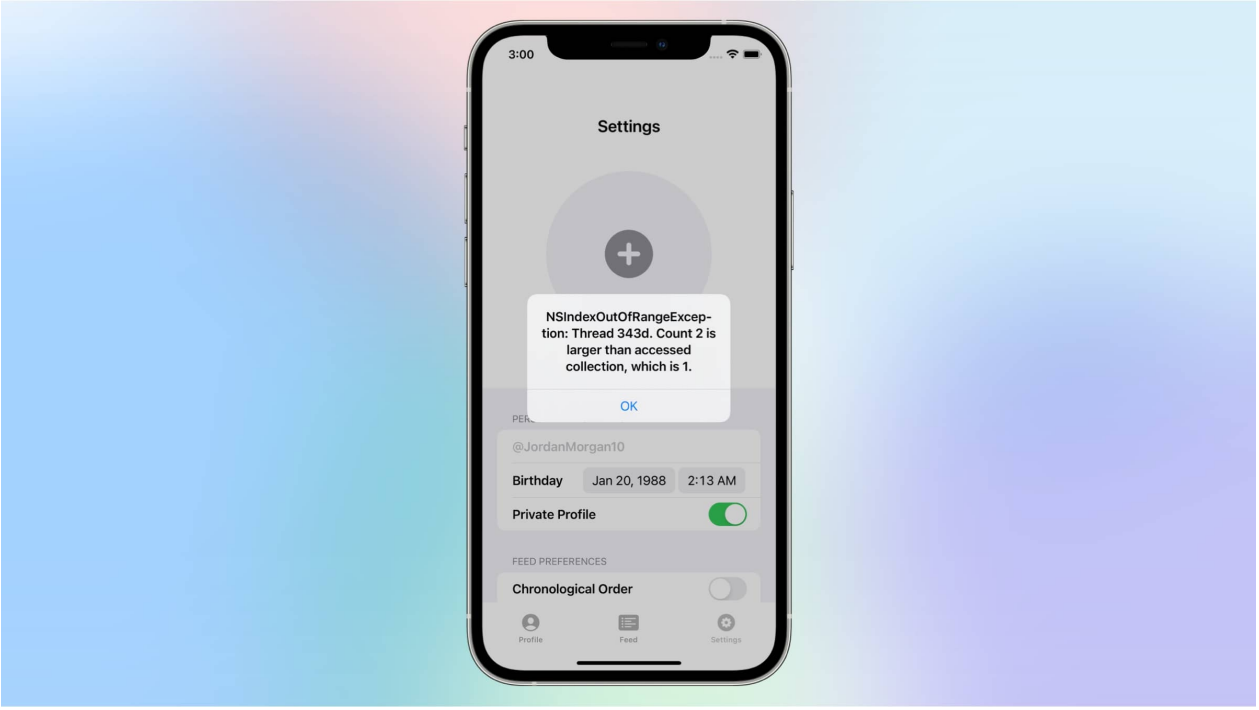
If you're not sure of who your particular audience is, always assume it's a generalized demographic. Then, stray towards being incredibly direct, polite and simple. You speak differently to people if you've got a grocery list app versus one that, say, mimics APIs calls for developers:



If you're primarily a developer by trade, the biggest thing you'll need to fight against is…talking like a programmer:



Uploading an avatar sounds with that wording is verbose, confusing and honestly – just plain weird to most people. Or worse, letting the *errors themselves do the talking*:



Look, I'm a developer by trade – I know how easy it is to do this. We write comments in code, or leave notes to come back and fix X or Y, and then we ship with copy similar to the last two examples. Not ideal.

Jargon is almost always a bad idea, unless your target market is actually other developers. Even then, tread lightly.

It's just not developers either, we all have our own inclinations when it comes to how we communicate. But we'd do well to remember who we're talking to, as opposed to having your users realize who's talking to them (i.e. a programmer or designer).

Actionable Text

When you have interactive elements, pair them with verbs. Or, if the context isn't crystal clear, use a verb-noun pattern. For example, opt for "Create a Message", or "Delete Message" instead of simply "Create" and "Delete".

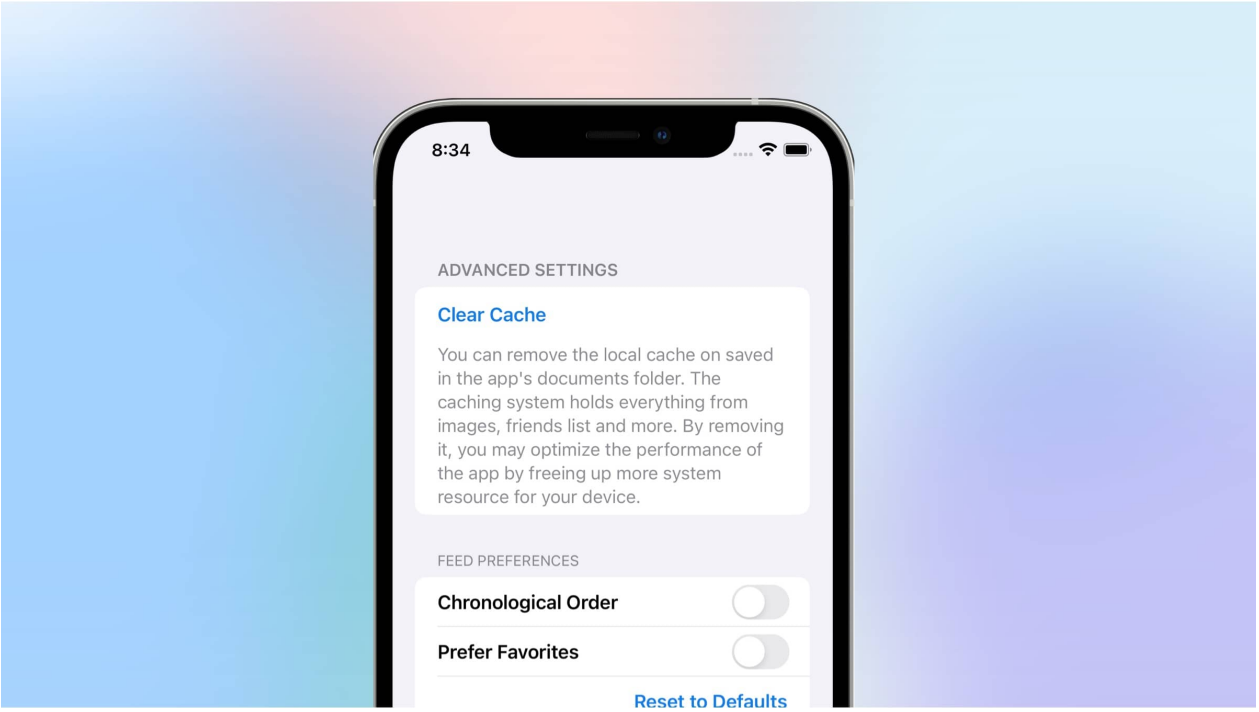
Short and informative should be your goal. There's a balance you're aiming for where you provide enough text to explain what something does, but short enough to skim over. That way, people can get a feel for the interface quickly by scanning over it.

Explain things simply by using verbs, and pair those with nouns where needed.

Consider Length

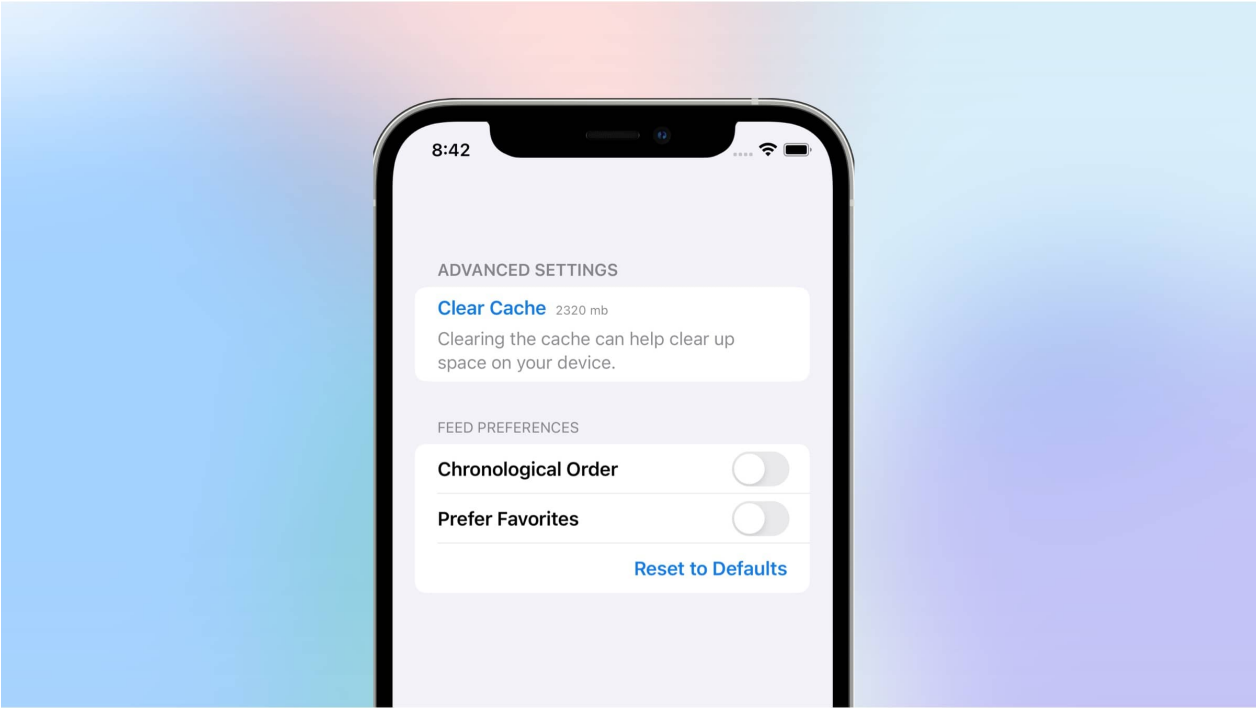
In fact, when explaining anything – remember that your "first draft" of that text is probably too long. This is doubly true when you are trying to convey the consequences of something technical. Say your app would benefit if the user cleared the cache, for whatever reason.

The concept of caching, what it is or why it would help to clear it is understandable to a very small group of people. So even if you apply the tips listed above about considering who you're talking to in your app – it's still easy to end up with something like this:



That won't work, and people won't read it. That's a bad thing, because if they just tap a button that could have important consequences, they need to understand that. So, we can bridge the gap between needing to explain something complex and users not wanting to read about it by simply be succinct.

Limit yourself to a sentence, or two at max, that simply explain why they would use such an option:



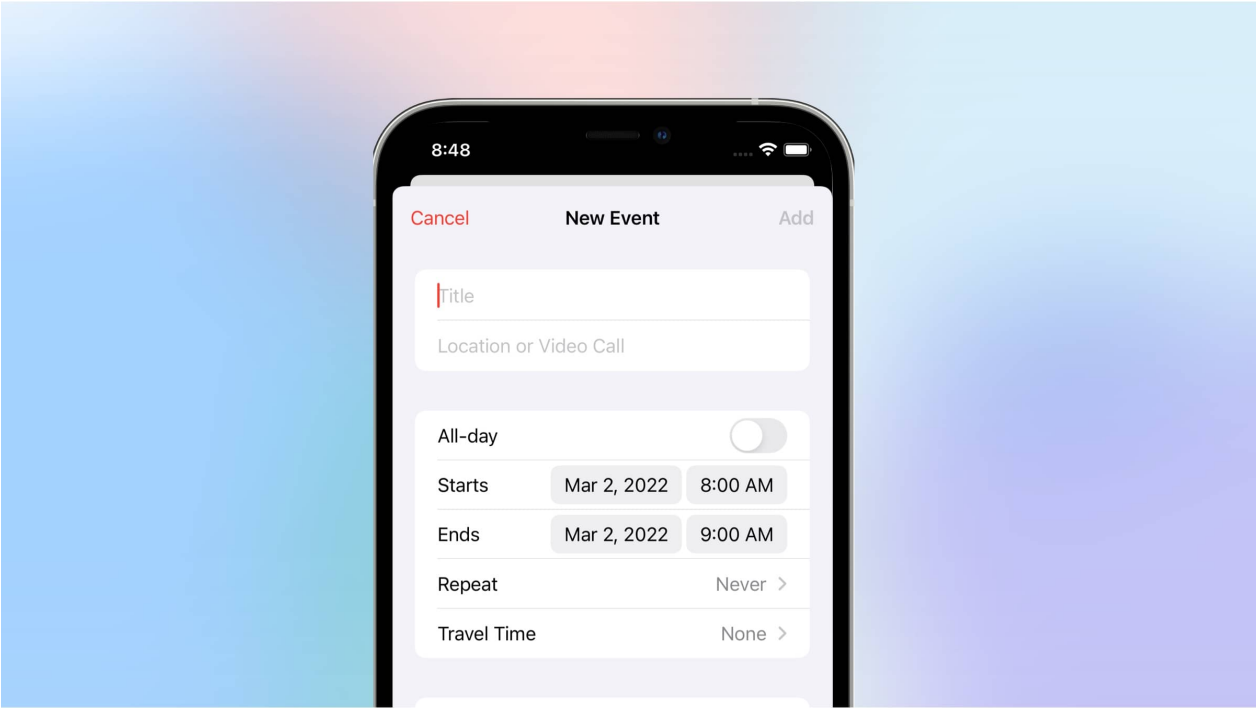
Getting Data Right

Your app is full of data. Formatting it correctly is paramount. Here's some common examples to look out for.

Dates

When using dates, be aware of the context they are displayed in. For many scenarios, using a friendly, understandable word instead of an actual date may be more useful. If you were showing a view where you needed to select a date, try using things like "Today", "Tomorrow" or "Yesterday" instead of the actual dates.

Also consider which way you should format a date. Should you include the time? If it's in the context of creating an event tied to a date, then yes – I think so. For example, the Calendar app:



But, if you're looking across a large span of time – then the time itself really isn't an issue. The day of the week and the actual date the day occurs within a month is probably more important.

Dates are a tricky one to get right, especially when you consider time zones. But it is one place where you can look in the mirror, so to speak. How would *you* feel about the interface you're designing with the dates formatted the way they are? Is it understandable? Is there enough data, not enough?

Currency

Put simply, there are a lot of ways to format and display currency. In iOS, there are even prebuilt styles developers can use, and as such – designers can rely on. For example, take the number 12.21 – here's all of the ways iOS will format it out of the box:

1. **No Style**: 12
2. **Decimal**: 12.21

3. **Currency**: $12.21 (this is localized)
4. **Percent**: 1,221%
5. **Scientific**: 1.221E1
6. **Spell Out**: twelve point two one
7. **Ordinal**: 12th
8. **ISO Code**: USD 12.21
9. **Plural**: 12.21 US dollars
10. **Accounting**: $12.21

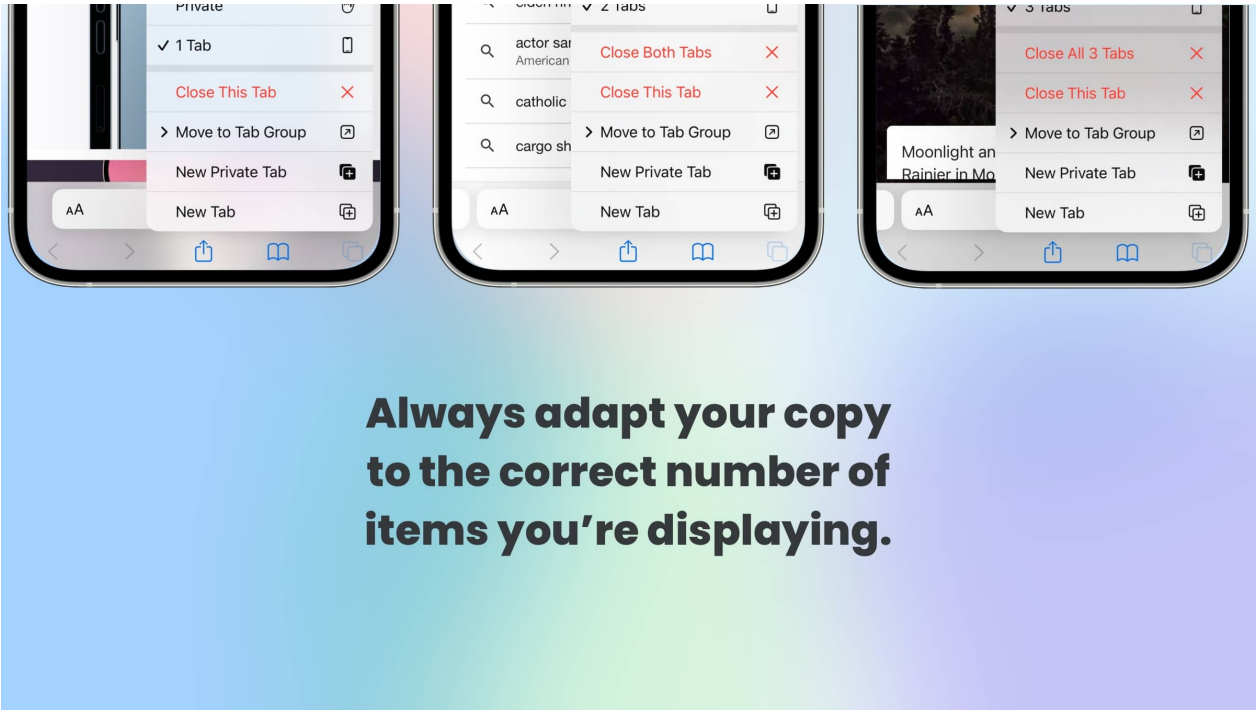
Deciding when to round up or down is always a context-aware decision. Should you include the currency symbol or ditch it? Be sure to ask yourself how much granularity you need to show when displaying money or currency.

There are countless questions you can ask yourself when designing copy for money and numbers. You're in the best position to decide how to format it.

You probably are aware of this already, but just in case you aren't — money is certainly a situation where the wrong decision can have a vast negative impact to users (i.e. the difference between, say, $1.2 and $1.2k) if it's shown incorrectly. So, stop to think about how it should be displayed.

Plural

This is a small one, but it hurts to see it in the wild. When you are dealing with a plural set of items, be sure to include a pluralized version of your copy. From left to right in the following picture, I had one tab open, two tabs and then three. Look at how the "Close Tab" message is tailored perfectly for each scenario:



We've all seen that countdown or timer interface that starts like "Two Minutes Remaining" and then hits one minutes and displays "One Minutes Remaining". Don't do that - make singular items singular and plural items plural. It's an extra bit of polish that too many apps skip.

The TL;DR

Start thinking of your copy as a first class citizen of your designs. Considering who you're making an app for will help, and it likely ties directly into your app's voice.

Playfulness

I think that no matter the app you're making, or the people that you're making it for — a little dash of playfulness can go a long way. A little micro animation here, or an unexpected interaction there, helps people feel connected, and in a way, "seen", by your app in fun little ways.

Remember the Lock Screen from iOS 7 when it debuted? You could slide up from the bottom to quickly open your iPhone's camera — but if you stopped half way up and "dropped" it, the Lock Screen image would bounce.

That's just *fun*. It's something you might show your friends or the the person sitting next to you. It's not a feature, per se, or critical to the app. But, it makes you feel a little better than you did a second ago.

Why not take a little time to invest in some interactions like that? A small dose of playfulness?

Playfulness can be used in a lot of places and interactions. Looking for the right places to use them usually doesn't reap obvious wins. Instead, playfulness tends to be more of an intuitive type of addition that you just feel is right for the context you're working on. You sort of know when it works, and don't force it in places where it doesn't make sense.

So, where are some places that it has worked for other apps?

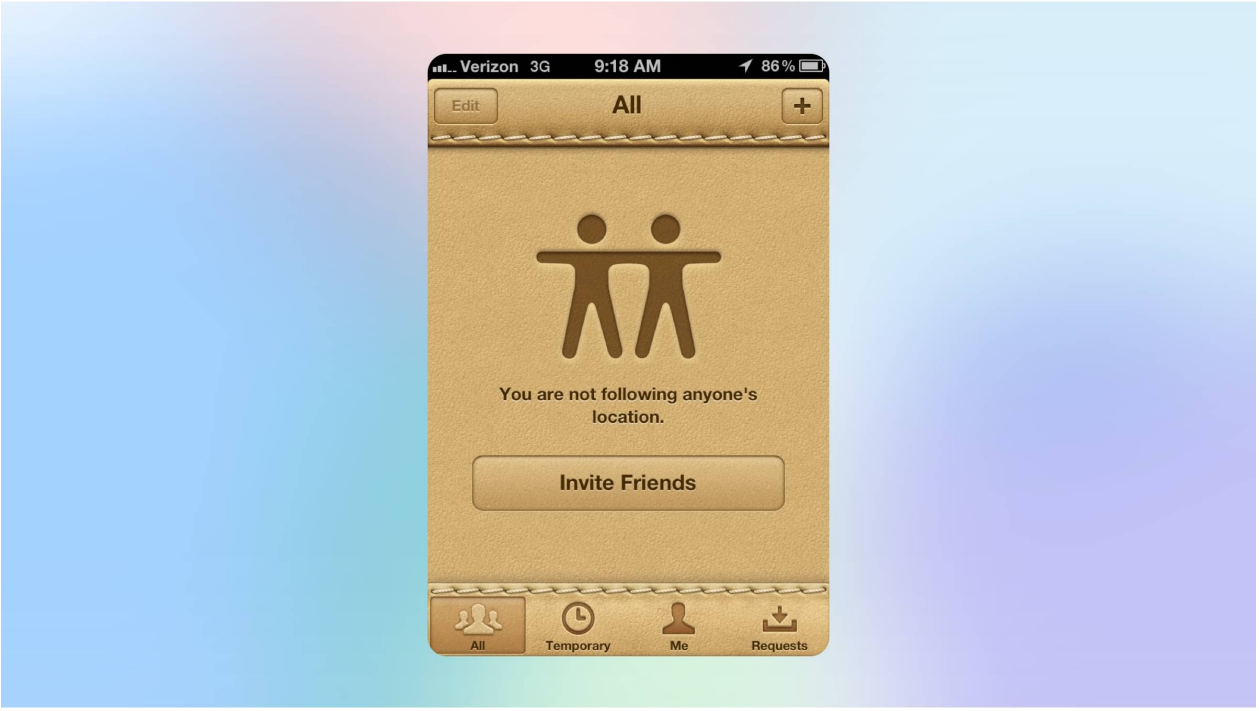
Physical Metaphors

The example of the Lock Screen above demonstrates a physical metaphor. Early in the iPhone's lifecycle, a full screen display without buttons was not the forgone conclusion that it is today.

It was new, different and most importantly – intimidating to some people.

So, one of challenges for Apple was to figure out a way to bring the entire globe over to a flat, glass screen for all of their interactions when they were used to tactile, clicky and plastic keys or buttons. A powerful tool to help through these transitions were metaphors, because people can take something they understand well and apply it to something they aren't as familiar with.

For that reason, we saw a lot of skeuomorphic design elements all throughout the operating system. Felt table tops for Game Center to leather bound contacts:



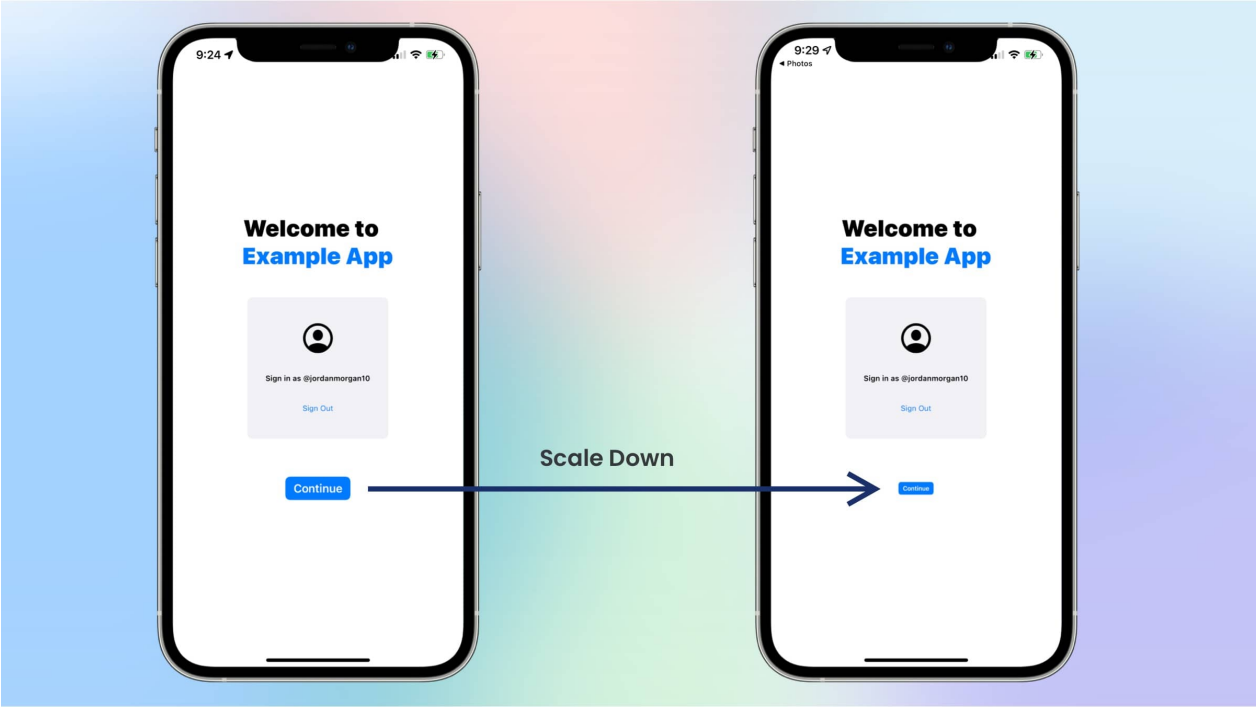
Earlier versions of iOS reflected their physical representations in their digital format too.

…the list goes on. You're probably familiar with these. But the *why* behind why they were chosen and used is far more interesting. So, following that thread, we also saw several physical metaphors in digital form for physical laws. In fact, the simple act of scrolling a list and having parts hide as you swipe on your screen is probably the greatest example of this.

While many of us understand how to use iPhones now, and the look of physical objects in digital forms has largely become obsolete, the act of including fun little physical responses to interact with interface elements still remains.

When we push something with force, we expect it to slide. When something drops from a tall building, we expect it gain velocity and fall down. We can play off these expectations in our app's design. And, they are typically ripe for a little dash of playfulness.

Whether that interaction is simply small, such as a button's scale changing when you press it down and release…



This example is exaggerated, but tactile feedback like this is often beneficial for usability as it is for fun.

…or swiping your avatar the screen and having it stick to a corner when using Picture-in-Picture mode, they add a bit of fun to the experience:

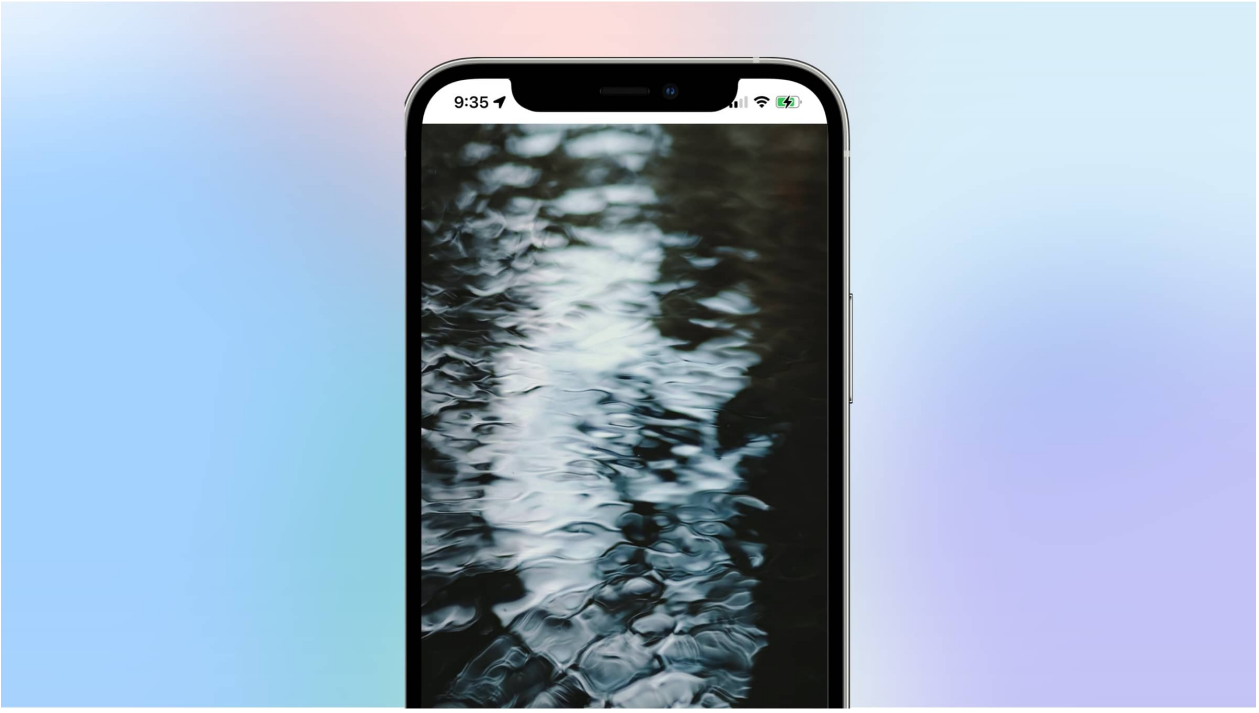


In fact, iOS itself even has frameworks in place to help drive these types of interactions. While this book isn't meant to show programming examples, if you're working with a developer, you can refer them to UIKitDynamics.

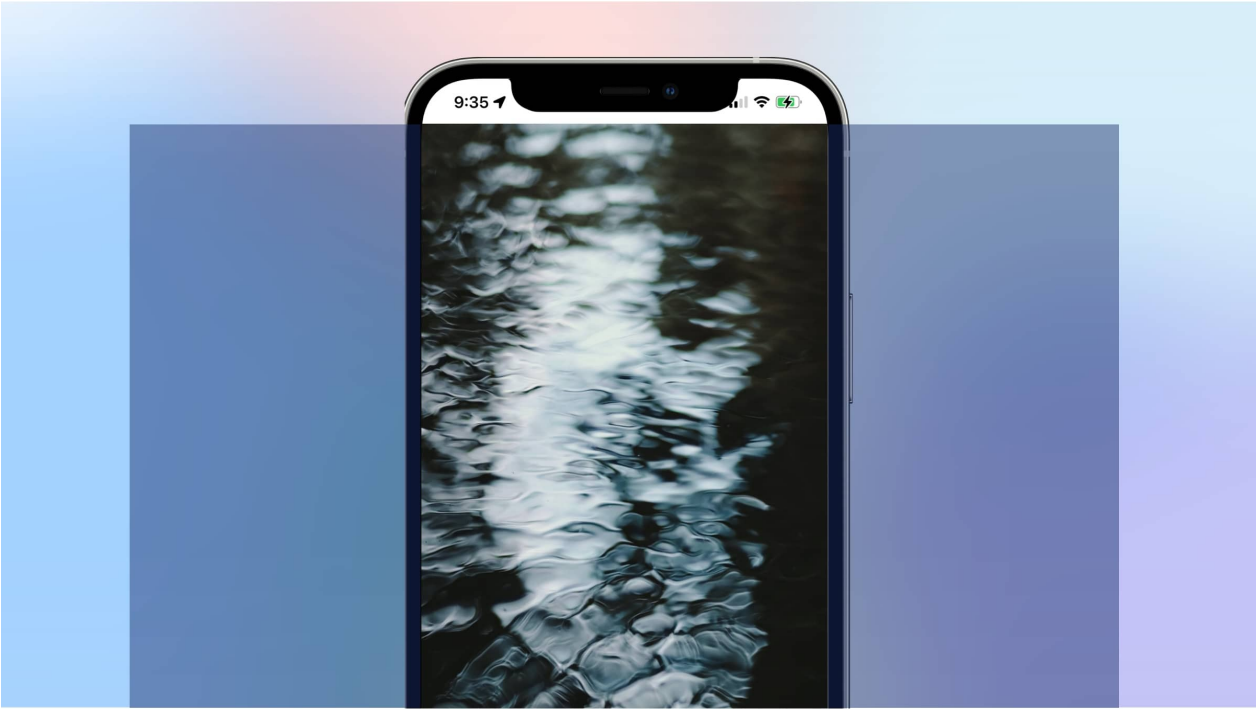
Surprise and Delight

One good way to incorporate playfulness is through surprise and delight. Give people a little punch of fun when they aren't looking for it. And, even better if they just happen to stumble upon it.

Let's consider a concrete example. Say you've got an image or media you're showing in your app. It's full screen and modal. The entire screen is taken over (rightfully so, media is an ideal time to employ modality) but depending on the aspect ratio of the image – some of it may be clipped:



Now, consider the device's accelerometer. As developers and designers, we can be informed of when the device is being tilted. So, just for fun – what if we revealed more of the image as the user tilted their device? For example, consider the blue areas below the clipped part of the image, which would offset and reveal themselves as the user tilted their device:



That would be fun, delightful and playful all at once. And, it checks all of the boxes I look for when implementing playfulness:

- It's not a critical feature that would be a net negative if it were never discovered.
- It doesn't get in the way, and it's not annoying.
- And, there is probably another way to achieve it's effect.

Let's talk about that last one real quick. In our example, suppose some people do want to see the other regions of the clipped photo. Even if they never discover that they could gently tilt their device one way or the other, it pays to include more traditional means to achieve the same thing.

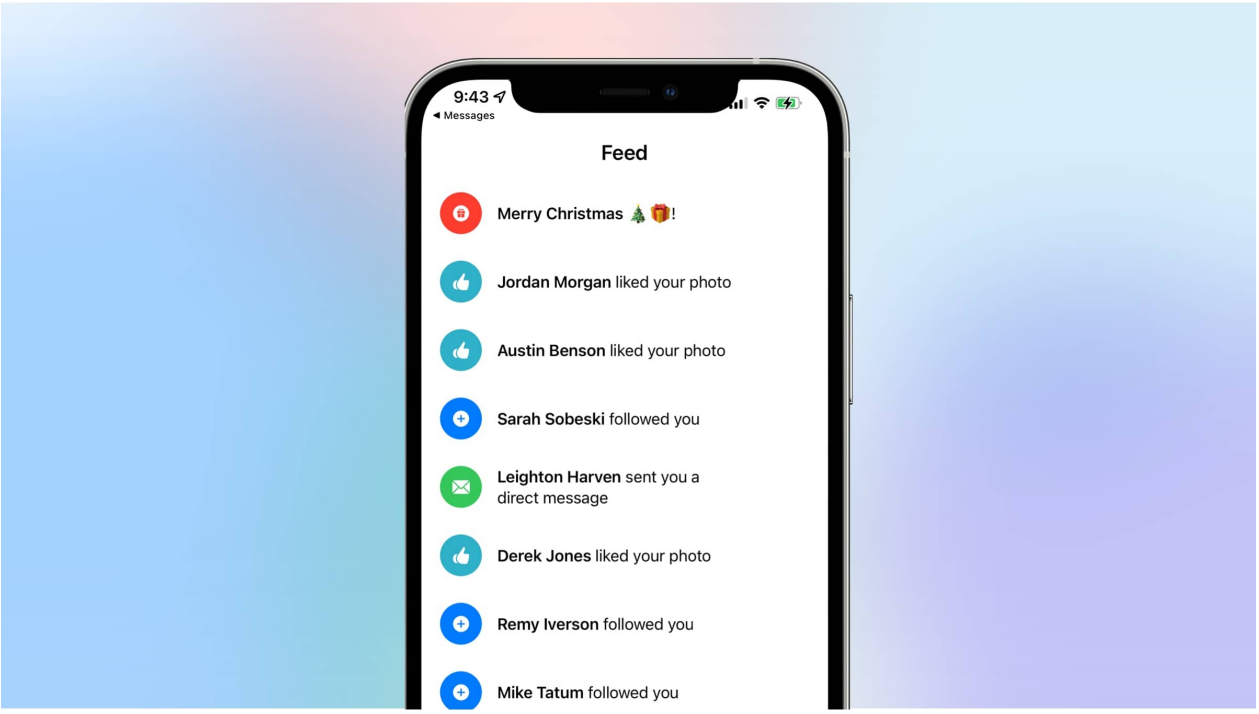
Here, I'd also expose seeing more of the image by simply swiping left or right on the image. That's standard and understood. It would be a mistake to *only* let people see more of the image from panning their physical device.

However, the people who do experience the panning interaction will likely enjoy it. It's something unexpected and discovered. It also shows a degree of craftsmanship and care that many people appreciate.

Novelty

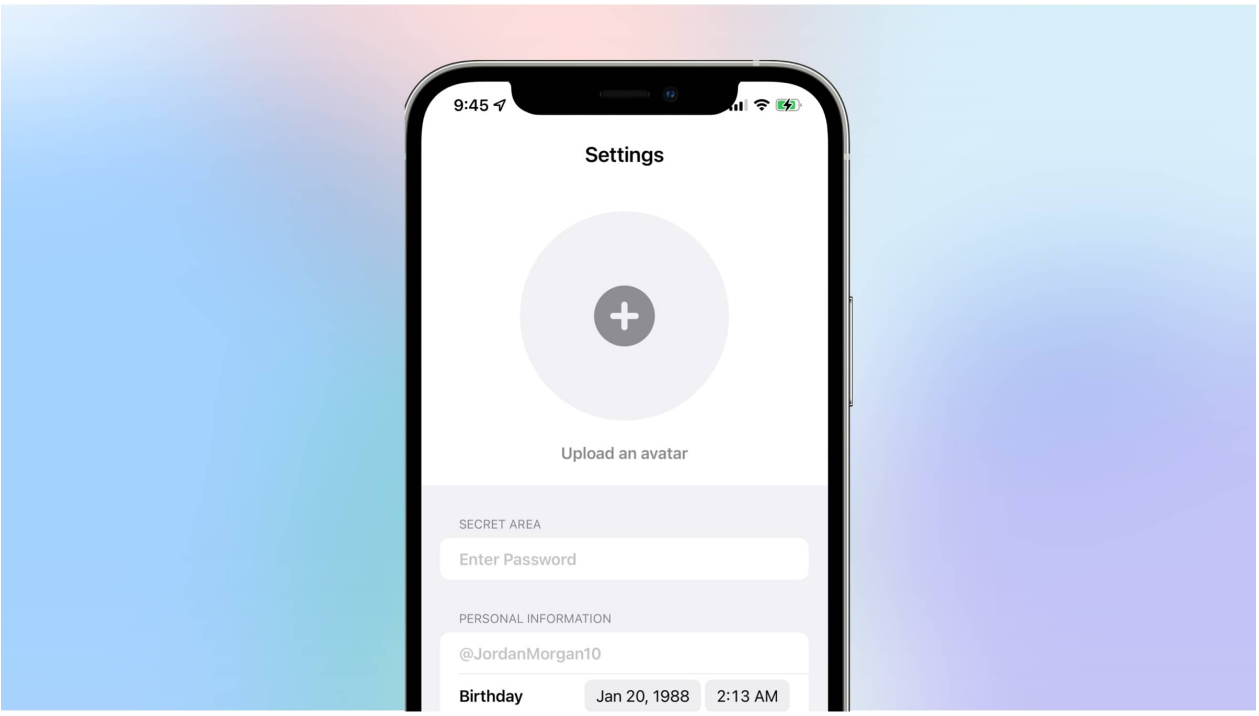
Some ways to incorporate playfulness is through the act including of novel additions. These are little dashes of copy or features that don't serve critical to the app's mission statement, but they are fun.

What does this look like? If you've got an established user experience where you show notifications of some sort, then maybe you'd play off of that for fun little experiences. For example, maybe it's Christmas Day when they open the app:



Or, in Twitter's iOS app – if it's your birthday, than the profile page is filled with balloons when you visit it. These aren't expected, but they are novel. And, again, they aren't in the way and be ignored altogether. That's key because a lot of people open your app to get something done, and if you litter it with too many novel little touches like this, it becomes annoying and even worse – a roadblock in getting whatever they wanted done, done.

Another thing I've seen be fun are Easter eggs. Maybe you've got an ominous "password" field in your settings screen:



What does it do? Does anyone else know?

Those little interactions can create community in your app, if done right. Perhaps you gameify things as reward users for doing something by unlocking an extra icon or something fun along those lines. People enjoy sharing those types of things, too.

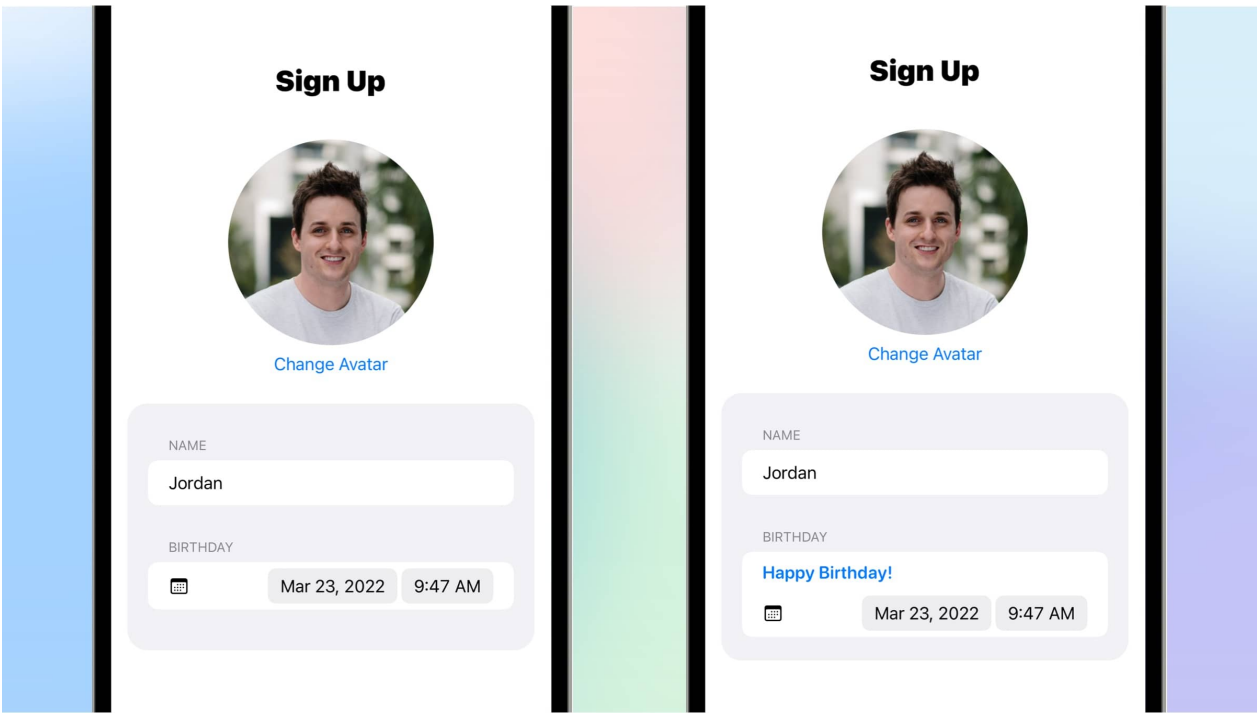
Being novel can add a nice degree of playfulness to your app. Knowing when to be novel, or what people will perceive as novel, is hard though. Getting novelty designed features into your app takes him and experience.

Copy

Another area where you can get playful is through your app's copy. We've covered this idea twice so far, once when we talked about finding your app's voice and another in the previous chapter dedicated about copy.

As such, I won't rehash much here. But what I will say is, be careful with this one. Copy is hard to get right in terms of playfulness but incredibly easy to get wrong. Don't come off as patronizing, for example, if an error occurred. That'll just upset people who are already upset that something went wrong.

Instead, try for lighter tones and low impact situations to use playful copy. For example, what if we had a sign up form for a service where a date of birth was required. If it happens to be their birthday on the same day they are signing up, maybe we could play off of that:



A little playful copy goes a long way. I think the example above works, but I've also seen many more situations where designers tried to get it work, and it just didn't land. Think twice and test it more with people before committing to it.

Designing Fun

No matter which way you go, remember this rule:

Use as much as needed, but as little as possible.

Playfulness in design is just that – little dashes of delight and glee. It really is as simple as that. But, most apps are made for a focused purpose. People want your app to do its "thing" for them much more than they want your app to be playful. So, there is an art to designing playfulness.

Personally? I love playfulness in apps, but I love it because I don't see it that much. And, I don't *want* to see it that much. If you see it too much, it's saturated and no

longer a net positive to the experience. But a little every now and then? That's the sweet spot.

Life is serious enough. So is the software we make, most of the time. How can we design ours to bring out a few smiles from people throughout their day?

The TL;DR

Playfulness can really endear people, but like anything else in iOS — it must be designed and deployed at the right time.

Gestures

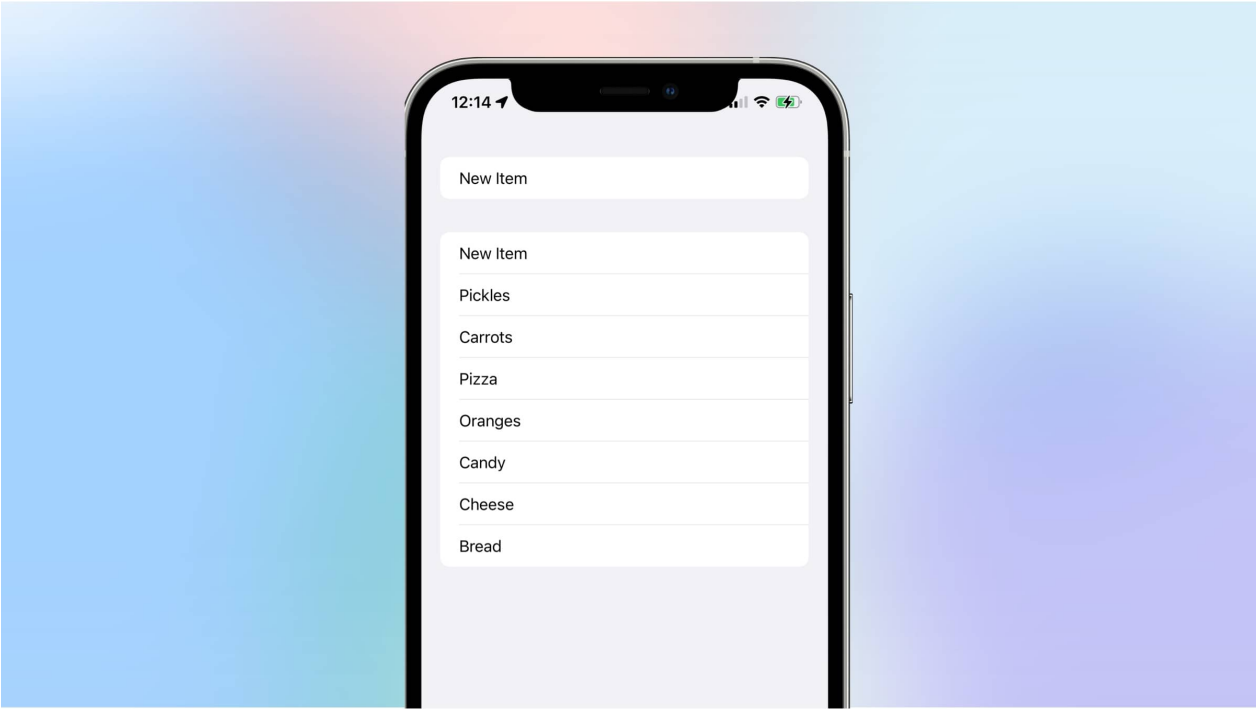
Gestures are inherently bound to iOS. They have to be — because no iPhone to date has any buttons on the front of it. They need to be — because they facilitate every interaction the device employs. And they should be — because time has proven Apple's intuition that a gesture driven operating system not only works, but it is a superior means to drive software in the form factor of a phone.

Given that context, I find it ironic that one area where designers can really frustrate people is through gestures. Including obscure ones, or making people relearn traditional gestures, isn't a design direction that often works on iOS.

The great thing about gestures and design on iOS is that to get them *right*, you just have to avoid getting them overtly *wrong*. If you avoid doing what's wrong with gestures, by default — you're incorporating them correctly. Let's take a look at the rules of gestures in iOS design.

Gesture Design Rules

The cardinal sign of gestures on iOS is making core interactions solely available via gestures. For example, if you had an app that was a shopping list — and the only way to add a new item was from a swipe on some cell that looked like a button:



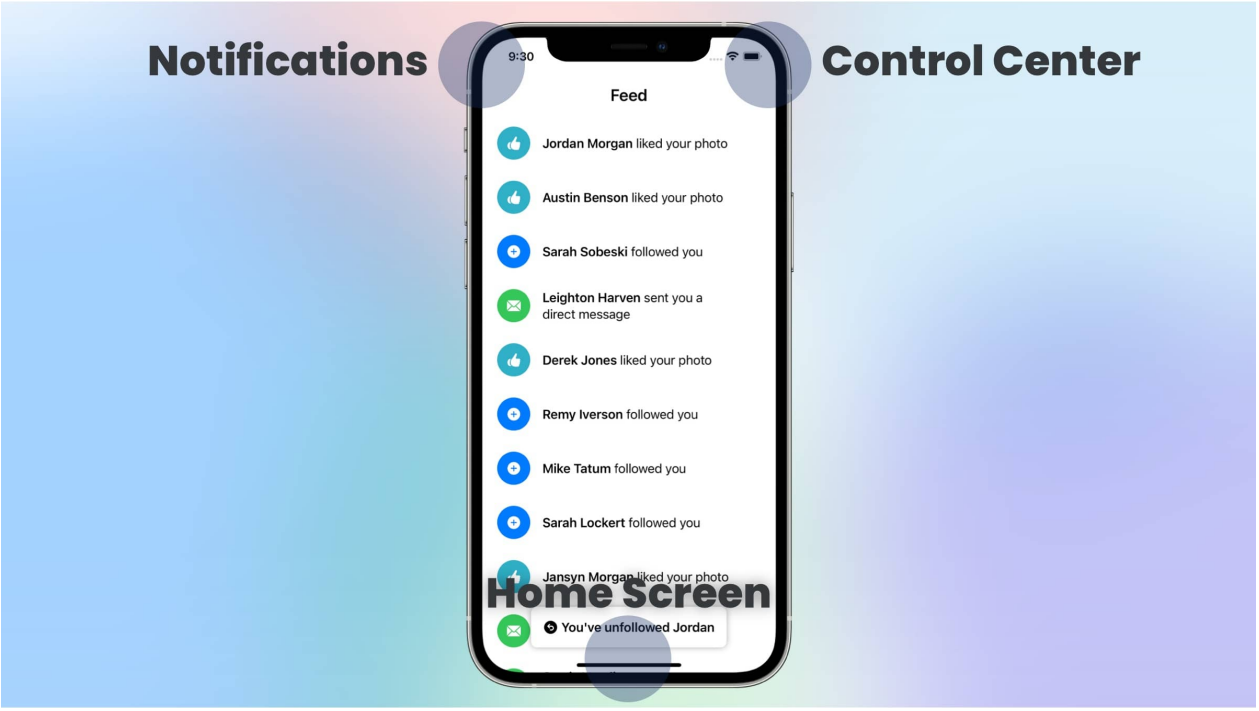
If the only way to add an item in this app was to swipe on the "New Item" cell, that would present several issues.

…then you've got a lot of problems. Chief among them are that:

1. That isn't discoverable.
2. That isn't traditional.
3. That isn't accessible.

If you had one of those three problems, that would be an issue. When you're committing all three of them, that's a huge problem. But it's one easily avoided. Don't put things behind a gesture and only a gesture.

Another rule to follow is to avoid edge-based gestures as much as you can, unless you're designing a game. If you override a gesture near the edge, you're all but certain to take away system capabilities:



Avoid conflicting gestures by leaving the system gestures alone.

People want to use control center, swipe up to go home and other similar interactions the same way they do in other apps. If you get in front of those, it can be frustrating. In fact, most people who aren't designers or developers aren't even aware that another app could "hijack" those gestures, so to speak, so it makes things all the more confusing.

With that said, as a designer, you'll want to be familiar with the types of gestures people expect in iOS apps. You probably know every single one of these, but having them in a list to easily reference has helped me several times. The list I'm including below is something I look at when I'm trying to think about how people might use some little interaction I'm working on that might be outside of the box, or not as common.

Then, if I can't match up that interaction with obvious gesture actions that are already established, it's a clear sign to probably go another route. The following list

probably isn't all of iOS' gesture suite, but it is the most commonly used gestures to be aware of and design for.

Types of Gestures

Multiple or Single Finger Swipes

What it is:

One or more fingers swiping.

Where you can see it:

In Mail, to select multiple items at once you can swipe three fingers up or down.

Multiple Finger Pinches

What it is:

One or more fingers making a pinch motion in, or out.

Where you can see it:

The marquee example of this is pinching in or out of images within the Photos app, or virtually any app that shows media.

Multiple or Single Finger Taps

What it is:

One or more fingers making one or more discrete taps.

Where you can see it:

If you've enabled the Accessibility Zoom shortcut on iOS, you can three finger double-tap to bring up the magnifying glass.

Shake

What it is:

A light physical shake motion of your iPhone.

Where you can see it:

This gesture is exclusively reserved for Undo and Redo purposes.

Rotations

What it is:

Moving your fingers in a circular motion.

Where you can see it:

If you edit a photo, you can often rotate it clockwise, or counter clockwise, by making a rotating gesture with two fingers.

Tap and Hold

What it is:

A tap followed by the finger (or fingers) staying held down for a specified amount of time.

Where you can see it:

If you want to edit text on iOS, you can tap and hold on the relevant words to bring up an editing menu.

Scrolls and Flicks

What it is:

The basic swipe in any direction, most commonly to create an offset of content.

Where you can see it:

Almost everywhere on iOS where there is content larger than the device's screen, such as the Settings app and its long list of options that one must swipe up or down to reveal.

Drag

What it is:

A persistent glide with your finger from one area of the screen to another.

Where you can see it:

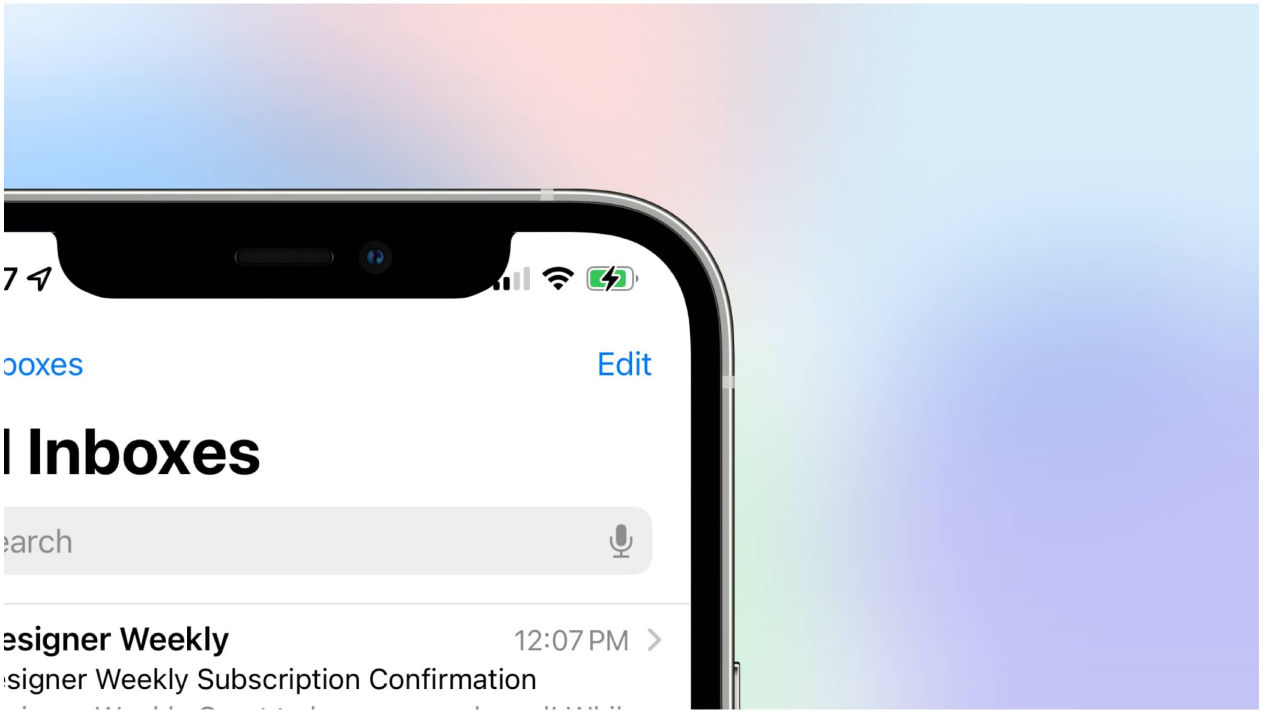
If you use drag and drop on iOS, added in iOS 11, the drag gesture is used to create this interaction. For example, dragging an image outside of the Photos app and dropping it into a note.

Complementary Gestures

Even though it's not a good idea to marry exclusivity with gestures, what *can* work is to supplement existing flows with gestures. For example, in iOS 13 Apple added the ability to bulk select items, and I couldn't live without it. In Mail, if you swipe with three fingers up or down – you select all the items that you scroll over:



And it's the fastest way to quickly select a lot of items in your inbox. But, it's not the way most people will do it. Most use the Edit button in the top right, and select things from there:



This gesture is another way to do the same thing, but in my opinion – much better.

While some will posit "that gesture isn't discoverable", in my opinion that argument often misses the point. If something is easily and obviously accessible via an interface button (in this case, editing), providing a gesture to do the same thing is complementary and often much faster.

This gives you the best of both worlds:

- The speed and intuitiveness of a gesture action versus other interface interactions.
- And, you avoid the con of it not being very discoverable, because it's not the only means to execute the task at hand.

If people find it, awesome! If you provide a tool tip to coach them, great! But so long as it's not exclusive but complementary, go for the gesture interaction. Once people find them, they typically become a core part of their workflow for many tasks.

The TL;DR

Gestures are inherent to iOS design, but getting them right is mostly a matter of avoiding the wrong things as it is accidentally doing the right ones.

Device Considerations

By and large, we have it pretty good when designing for iOS. Since Apple controls the ecosystem, we never have to worry out OEMs creating hardware for the operating system — which would invariably lead to thousands of form factors to worry about. Just look at Android. That's not a criticism either, it's simply the result of two different methodologies when it comes to creating products.

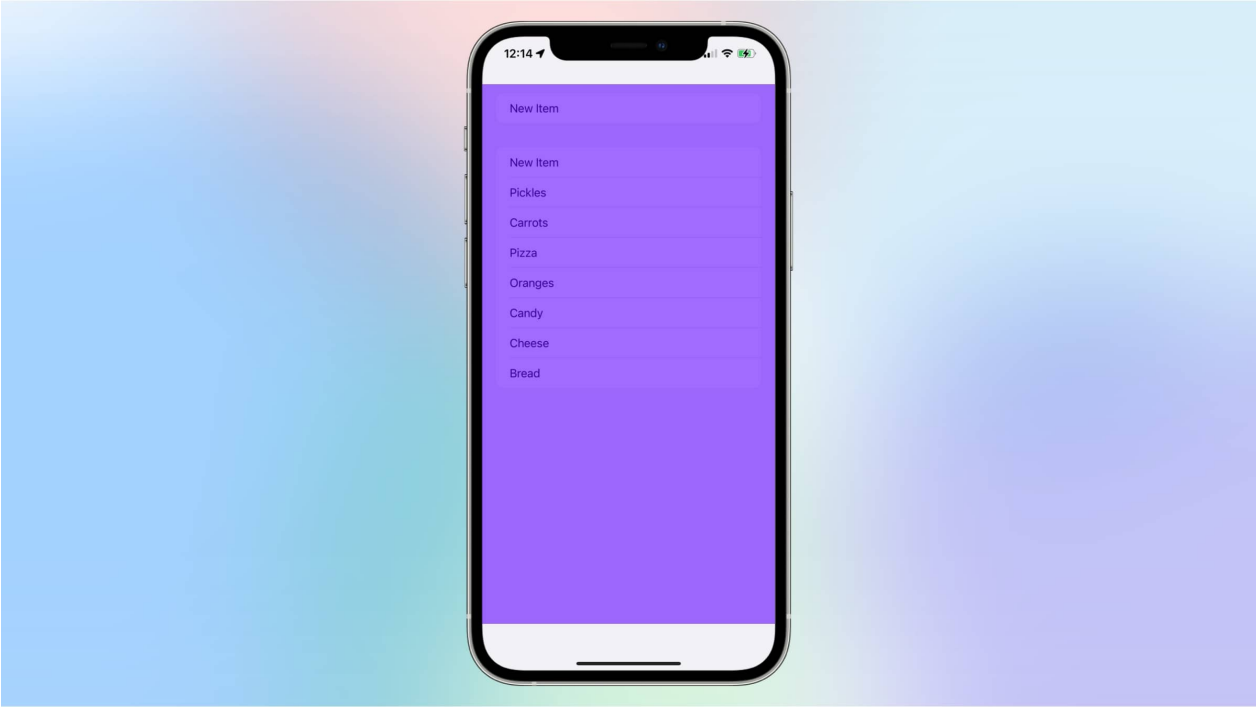
Even so, there are still many device considerations to think about during the design process. This chapter is all about laying those out and evaluating what they mean for our designs. Much like the last chapter over gestures, the primary concern is to not interfere with parts of the device or operating system that are known quantities to the billions of iOS users around the globe.

Further, we'll cover a bit about the differences between iPhone and iPad. The two devices require two different designs, and simply blowing up your existing experience for the iPad isn't a great option. We'll look at ways to take advantage of the extra space iPads typically afford us.

Safe Area Layout

Since the advent of the "notch", Apple had to come up with a way to ensure developers and designers alike wouldn't put content underneath it. The solution? The safe area layout guide and margins.

Here's what it looks like:

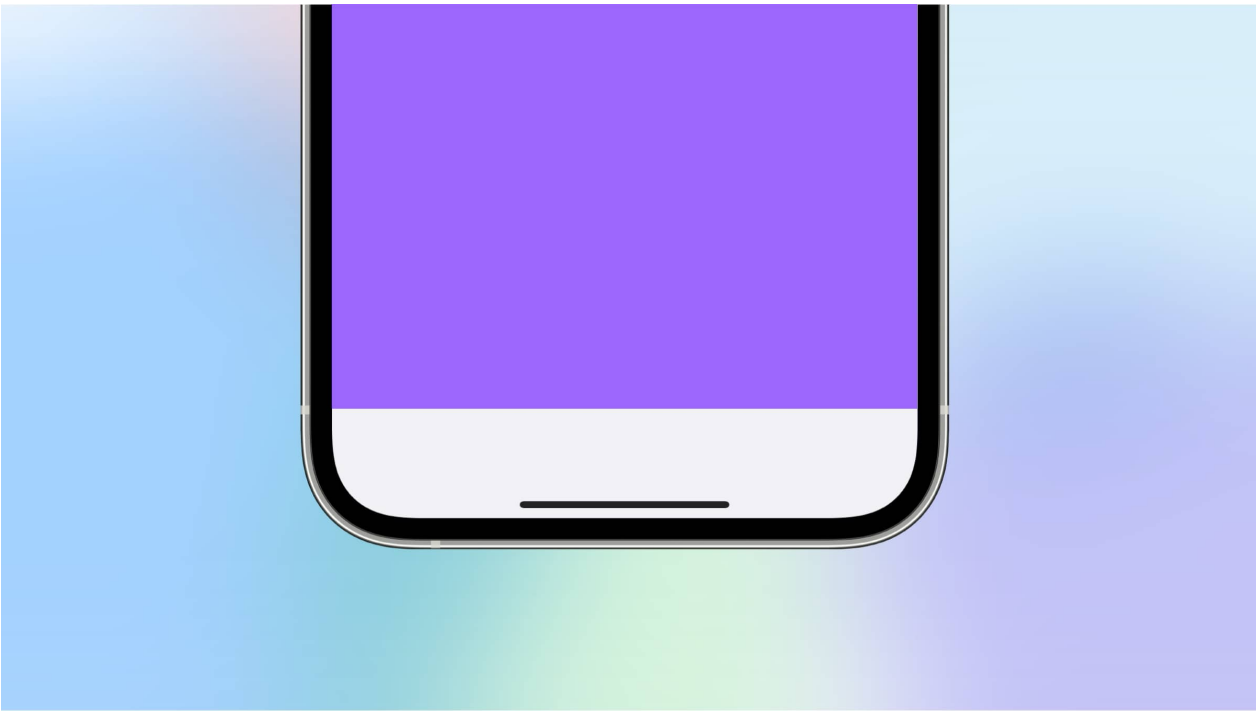


The purple area represents the safe area, where you are free to design as you wish without clipping content.

On "notched" devices, like in the example above, the safe area will ensure you don't dip into the home indicator or clip underneath the notch enclosure at the top. Developers have APIs to work around these concerns, but what do we need to do as designers?

The main thing we can do is ensure that we come up with designs, we're aware of the fact that some iPhones will have different safe area margins. If those are followed,

you mostly don't even need to worry about the fact that some devices have the home indicator, like this:



Or some don't, like the SE series of devices:



Safe Area

No matter the device, each one has a safe area. For iPhones that have the classic home button, it's effectively the whole screen.

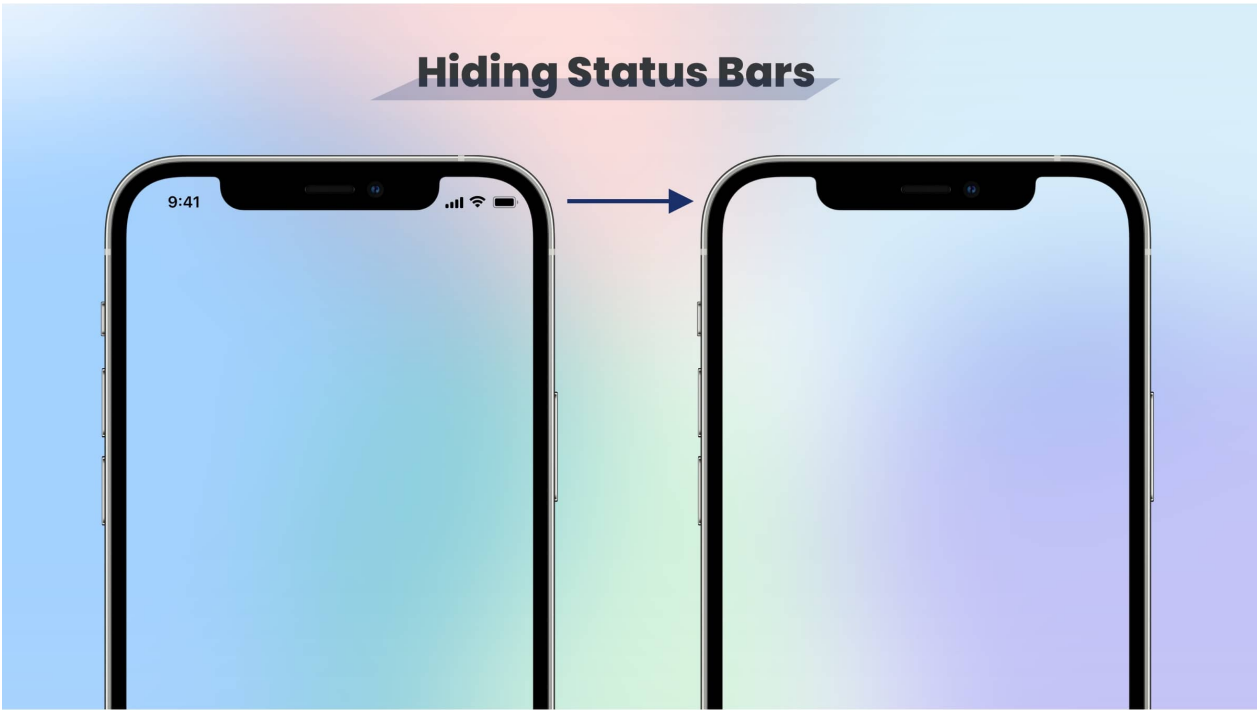
However, if there were controls such as a navigation bar, then the safe area would ensure it's also underneath it.

As we've previously discussed, there are a few times when you account for more immersion in your designs and put content above or below the safe area. However, that's almost entirely exclusive to showing media, or designing games.

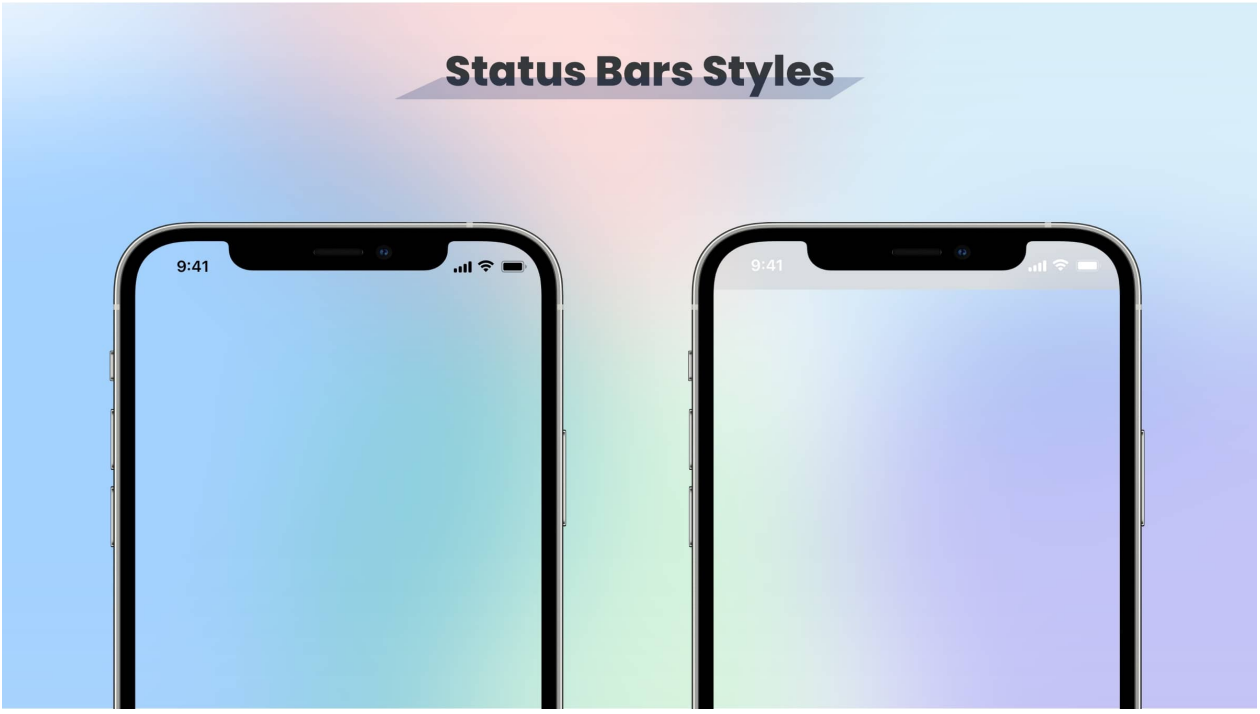
Status Bar Handling

The status bar is nearly always present on iOS, and for good reason – it houses mission critical information that people probably want visible the majority of the time. You can, however, tweak it a little bit.

For example, you can animate it away:



Or even change its color to light or dark:



I'll repeat a common theme here, but don't mess with the status bar from a design perspective. That is, don't intentionally hide it and try and replace it with something else. That leads to a fractured experience and gives your app that uncanny valley feeling, where it looks and almost feels like an app, but something is off.

One design consideration you might encounter is how to handle content scrolling underneath the status bar. There are times where content can be a bit jarring when scrolling under it. In such cases, you can typically take advantage of iOS' material system to mitigate this by putting it right under the status bar:



On the left side, the words scroll under the white status bar making it hard to read. On the right, a small blur view is added under the status bar to help with this.

iPhone versus iPad Design

Designing for iPad is often overlooked. While designing a universal app (one that runs on both iOS and iPadOS) is typically the baseline with releasing an app, I'd encourage you to hold off on a release for iPadOS unless you've taken the time to be intentional about its design.

While we covered several iPad considerations throughout this book, let's look at some principles to help. We've mentioned several of these ideas, but I find it helps to have them all in one quick list for reference.

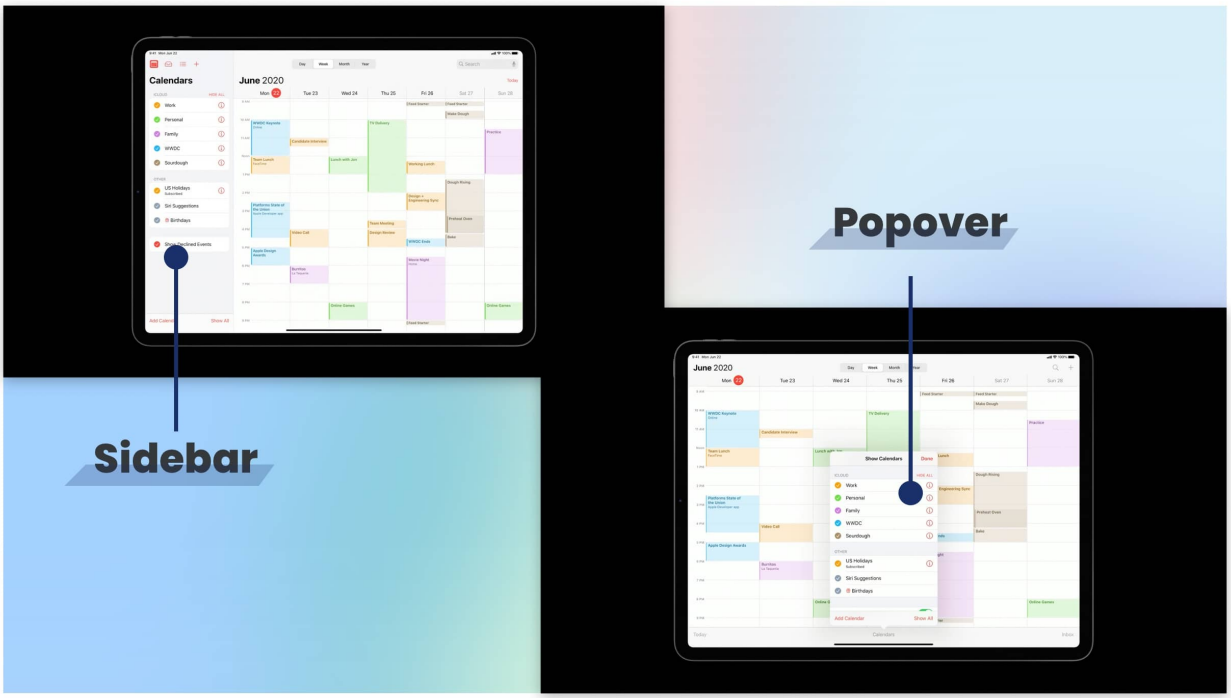
Reduce Modality and Taking Advantage of Space

Reducing modality in general is a good idea, as we discussed in the chapter covering modality exclusively. The reasons why are:

- It covers up existing context.
- It might create confusing navigational flows.
- It's meant for focused tasks, but it often isn't used that way.

To combat this, especially on iPad, consider inline editing for data. Instead of showing a modal view to edit a name, change a quantity or what have you — consider letting people do those operations without presenting another view. This rule applies to iPhone as well.

In general, consider moving information somewhere else that's persistent to take advantage of extra space you've got on iPad. If you can tactfully show more information — I'd recommend doing it. For example, Apple used to show data about calendars with a popover on iPad, but has since moved it to a more permanent place:



You probably have several cases where you could make similar design choices. It speeds up workflows and provides more clarity – both of those are things you should aim to do on iPad. The form factor calls for it, and one way to create a North Star for your iPad designs is to think in these terms:

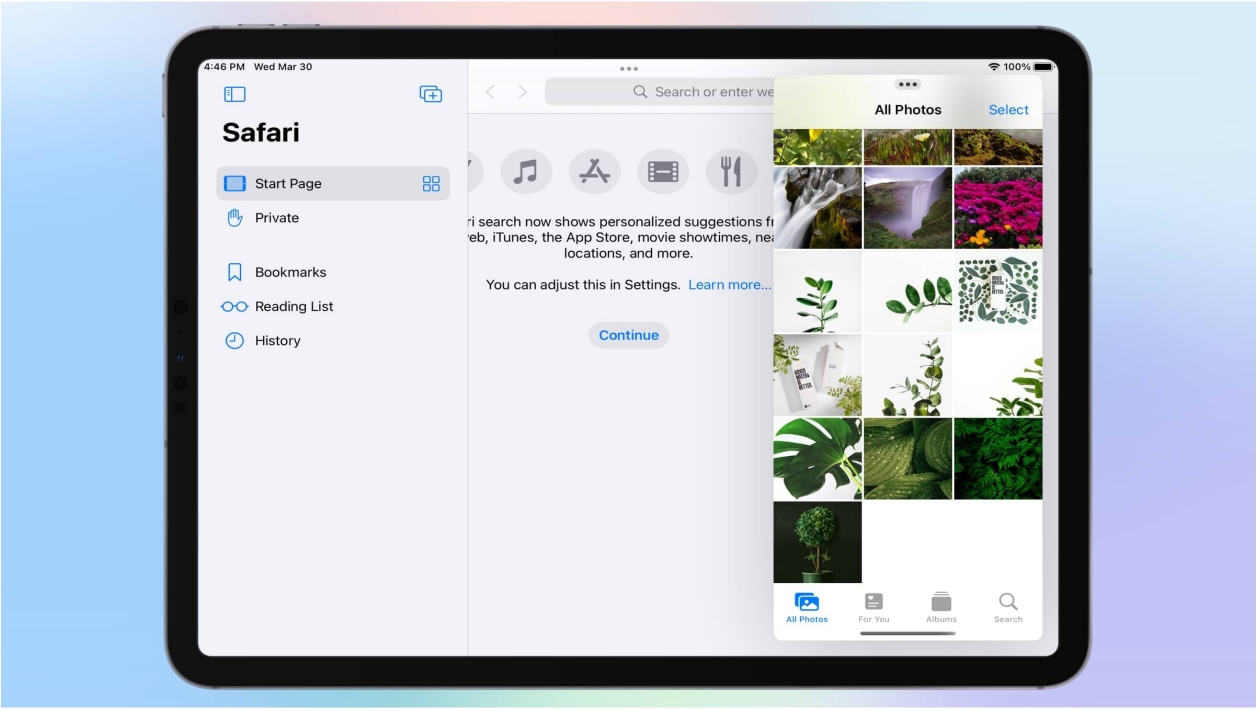
- On iPhone, people are trying to see or do something relatively quickly.
- On iPad, people want to see and do more and likely have more time to do it.

It's also common for people to use a keyboard or mouse for input or perhaps the Apple Pencil. Both of these provide them flexibility in getting things done. Most people want to do, simply put, more in your app on iPad than they might on iPhone. Optimize for that.

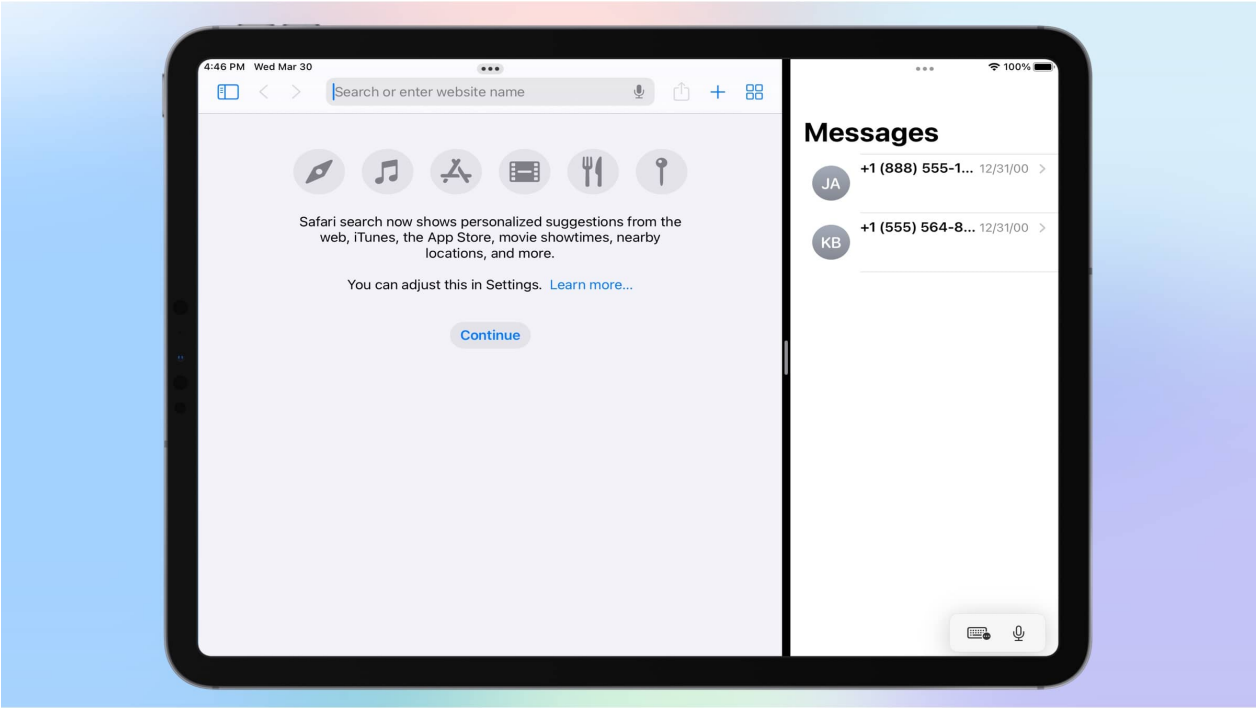
Remember Multitasking

Multitasking is paramount on iPadOS. It's manifested in a few different ways that it pays to be aware of when designing.

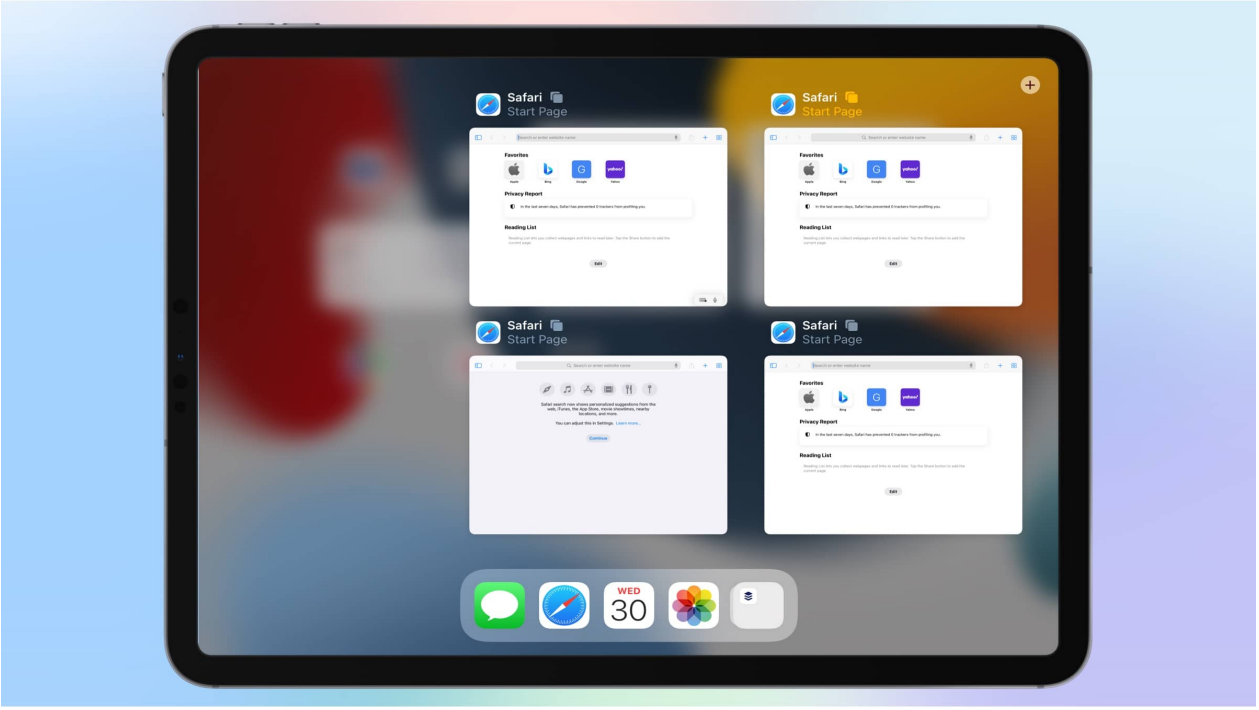
There is slide over:



A side-by-side, resizable approach:



Or, one app could have several windows running:



So, as a designer — remember that even though you are *thinking* about iPad designs, your "iPhone" design will surely come up on the device too. The inverse isn't as true to that degree it is on iPad. So, it's better to abstract your design in terms of "I have a lot of space" and "I have limited space" instead of "This is for iPhone" and "This is for iPad."

Developers will appreciate you thinking like this as well, since at the API level, this is how Apple exposes things too via the concept of size classes.

Flatten Navigation

Sidebars are a great way to flatten your hierarchy. This is a great idea on iPad too, since making getting around easier in your app is one of the best ways to invest in extra space. Saving taps is one of the highest impact tweaks you can make in design.

It's not about throwing out a mountain of information on the screen, it's about making sure the highest impact flows are available without extra work. That's why sidebars work so great on iPad. Apple uses them extensively on iPadOS:



Whether it's a sidebar or something else, consider making navigation easier on iPad. And, I define "easier" almost exclusively by "less taps to get to X", because that means less work and mindshare will be spent on navigation from the people trying to get things done in your app.

Please refer to the Navigation chapter for a deep-dive on more tips.

The TL;DR

Each device has different design constraints that require attention whether it's the size of the screen or the lack, or presence, of a home button. As such, it pays to design for each individually.

Feeling at Home

As we near the end of this book, you've now read most of the tips and techniques that guide my design process. We've covered how to define what is we're designing, how to determine what should be a part of that definition and then how to ultimately sculpt that into an app.

The whole process is exhilarating, rewarding and exhausting. And, often at the same time. The challenge now is this; How do we take all of this knowledge and use it to form a foundation where all of our design processes will be derived from going forward? If we've read all of this design information, and it goes away tomorrow, we haven't done much to further our design skills.

To that end, let's try to bring it all together. There is this idea I have about iOS design that helps me form my own "design foundation", and I remember it by this phrase:

Feeling at home.

When an app feels at home on iOS, it was probably designed carefully before anything else happened. It feels nice, premium and sure of itself. It looks that way, it feels that way, it works that way. Feeling at home means that you take advantage of iOS de-

sign paradigms, that it's easy to navigate, it uses system features and overall it simply becomes something that you look forward to using. You have confidence in it.

None of these ideas are new, to wit – we've talked about each one of them at length. But, these are the things we need to keep front and center in our mind. If you read this entire book, and for whatever reason – you don't work on any app designs for a month, two months or even a year, this is the chapter you should come back and read to get in a good state of mind.

Let's form a quick list of some key aspects of what feeling at home on iOS looks like.

Following iOS Design Themes

Apple has long followed foundational themes for its design philosophies, but since the advent of iOS 7's dramatic redesign they've been rethought for the iOS platform. We need to be aware of the following three themes so that we can better understand how to think about iOS design from a holistic standpoint. These three ideas act as our design compass, as it were.

Clarity

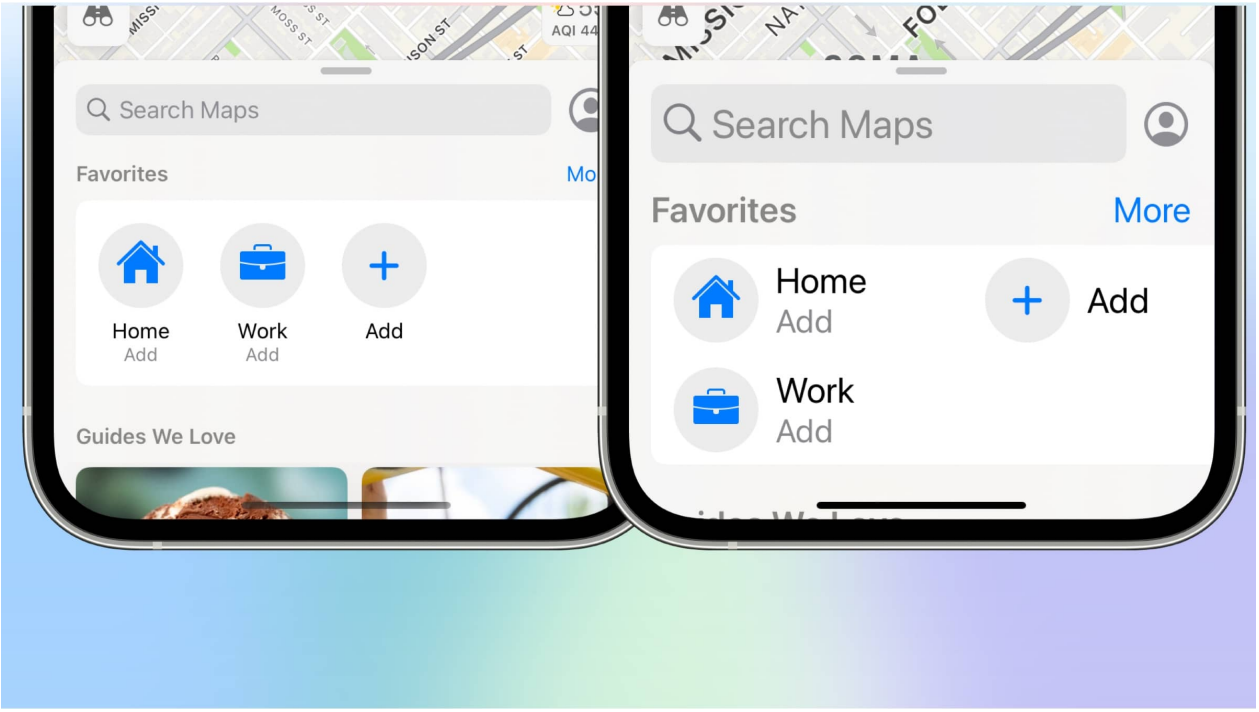
Let's take a look at Apple's definition of *clarity*:

> *Throughout the system, text is legible at every size, icons are precise and lucid, adornments are subtle and appropriate, and a sharpened focus on functionality motivates the design. Negative space, color, fonts, graphics, and interface elements subtly highlight important content and convey interactivity.*

You could make the argument that this is the majority of design on iOS. All over, we see apps that win awards, are critically acclaimed and loved by users following this principle more than any other. It's a defining bedrock of not only iOS, but Apple's approach to software, hardware, marketing and just about any other part of Apple's DNA.

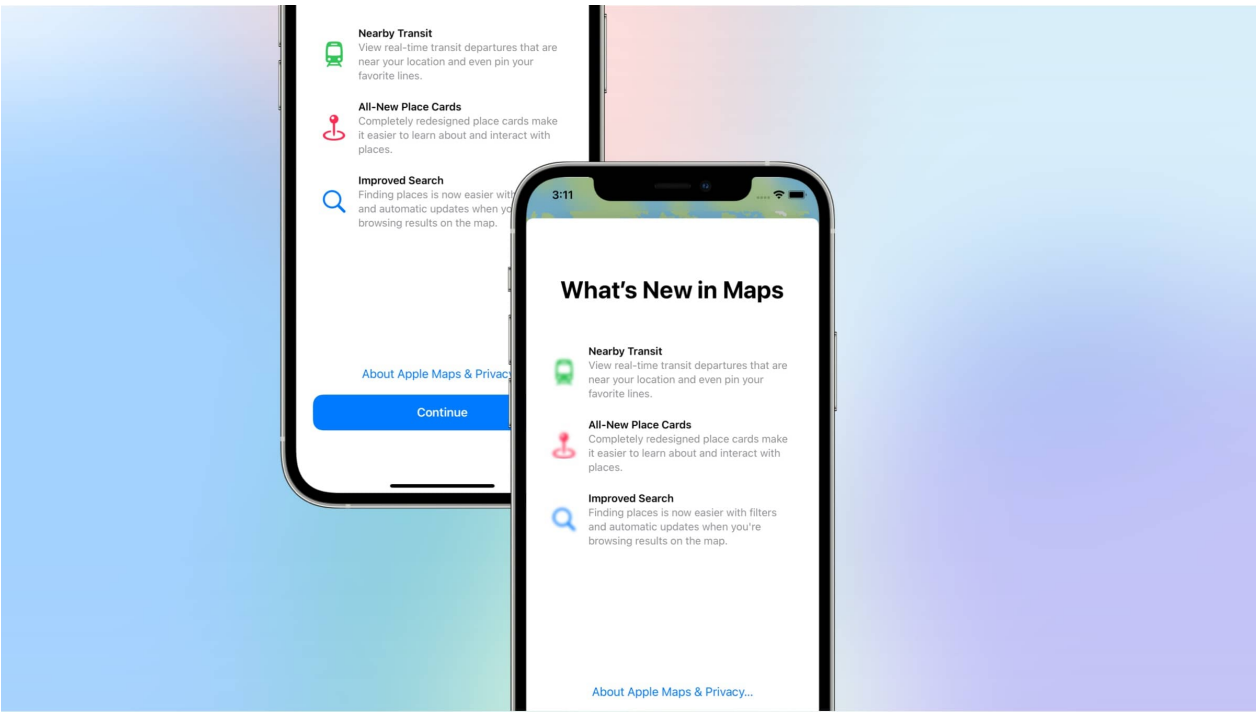
Clarity is how Apple does things, full stop.

So, how do these ideas manifest themselves on the platform? Let's take one of the clarity principles – text is legible at every size:



No matter the size, typography on iOS scales well when it's done right.

Or, icons are precise and lucid. Most importantly, they *aren't* blurry or sized incorrectly. You've probably used apps that have crisp iconography like the left photo, but also have been caught off-guard when you come across another like the right:



Blurred icons are a staple of design being an afterthought.

Even things like negative space make a screen that might've felt off, feel right. Or worse, one that was already working, into one that feels cramped:



Negative space can make something feel just right when there's an acceptable amount applied, but too much ends up making designs feel cramped.

Keep clarity at the forefront of your designs, and while I don't believe Apple would ever come out and say one principle is more important than any of the others — *I* certainly will: Clarity is the most important theme to follow.

If your app doesn't have clarity, it won't really matter if it follows the next two principles. Meanwhile, in my opinion, the opposite isn't true.

Deference

Next, we arrive at the concept of *deference*, which may seem a little abstract at first. But in practice, it's fairly easy to comprehend and ultimately to design for. Again, let's see what Apple has to say on the matter:

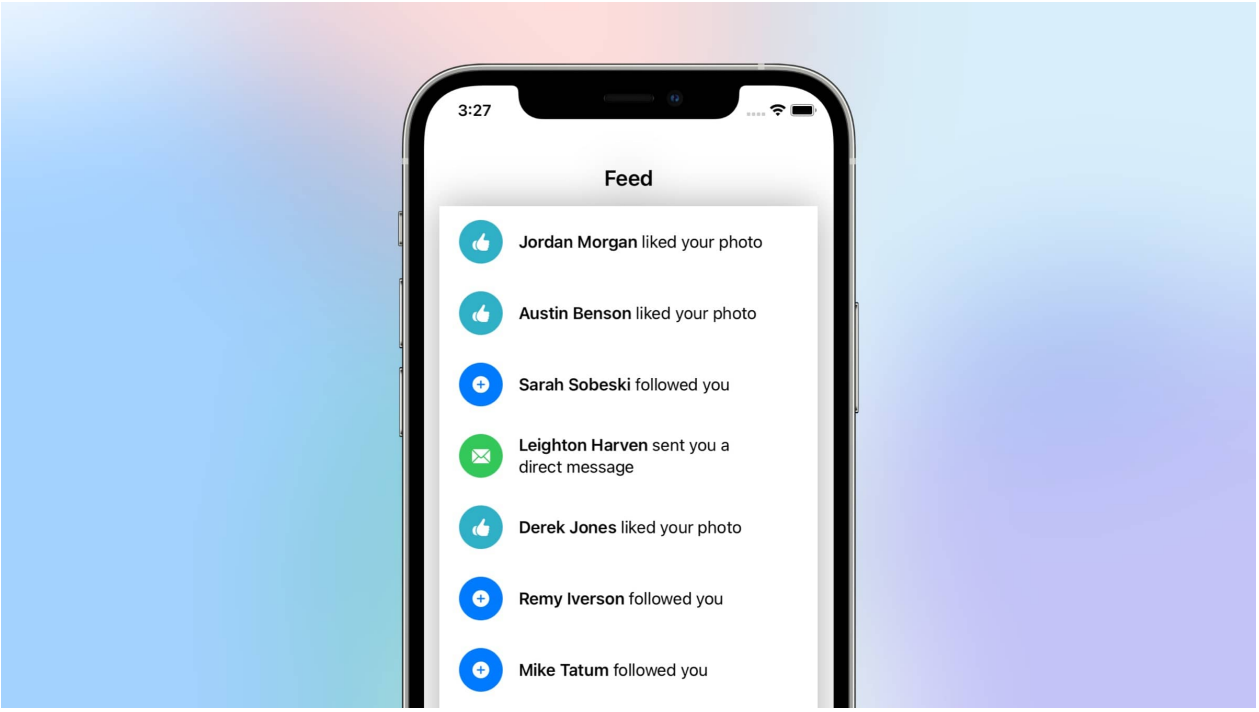
> *Fluid motion and a crisp, beautiful interface help people understand and interact with content while never competing with it. Content typically fills the entire*

screen, while translucency and blurring often hint at more. Minimal use of bezels, gradients, and drop shadows keep the interface light and airy, while ensuring that content is paramount.

So once we have an app that promotes clarity, it now is setup to support deference. Just think of the concept of deferring things – if you defer something, you're essentially saying that thing X is not as important as thing Y. It doesn't mean that thing X doesn't have its place, but it *does* mean it's complimentary to the overall picture.

Gestures, bezels and shadows are a core component of most design paradigms I've seen, regardless of the platform. But they can't be the hero on iOS, and they shouldn't be either. Your content should. That's what we're trying to remember when we incorporate deference into our designs.

Consider the difference. If we reused the exact same "Feed View" example we've worked with throughout the book and toss in a lot of shadow – now it's annoying and distracting. We aren't *deferring* to the content, we're actually aware of adornments instead:



Depth

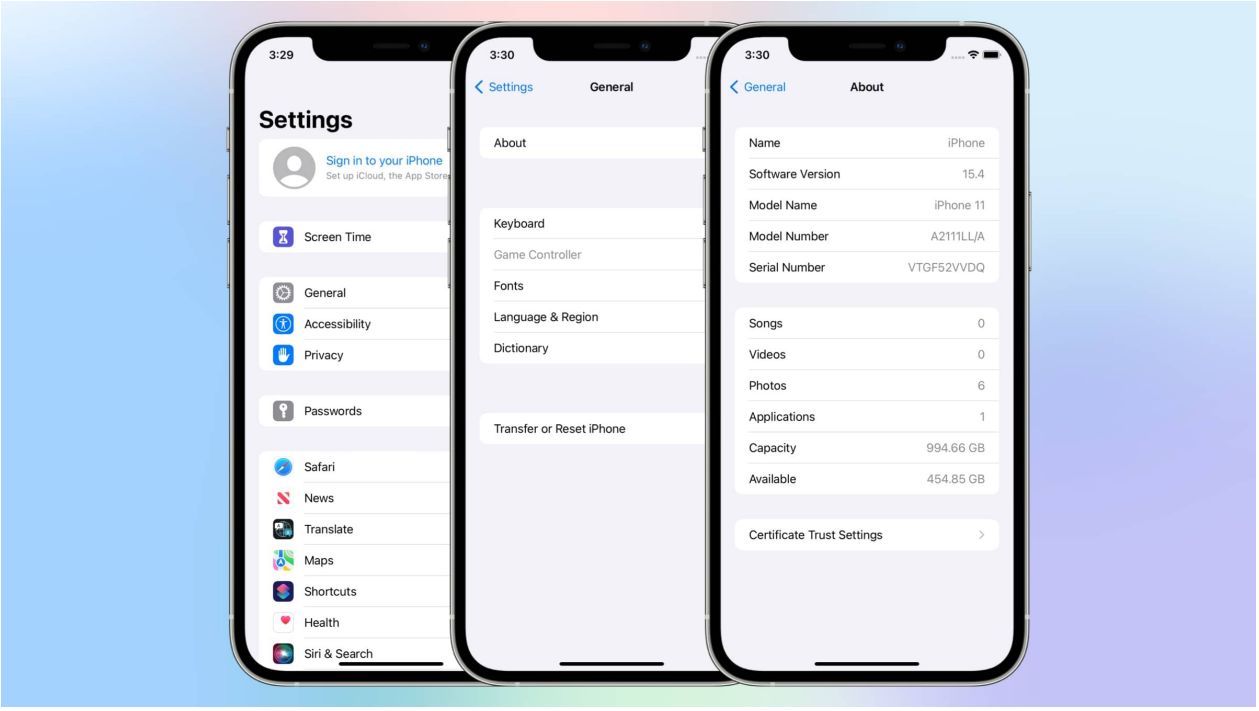
Let's round out the core iOS design themes with *depth*.

> *Distinct visual layers and realistic motion convey hierarchy, impart vitality, and facilitate understanding. Touch and discoverability heighten delight and enable access to functionality and additional content without losing context. Transitions provide a sense of depth as you navigate through content.*

Depth is a fairly broad topic in these terms, but its implications are paramount to effective iOS design. You can see depth incorporated properly on any iOS device – just open up Settings.

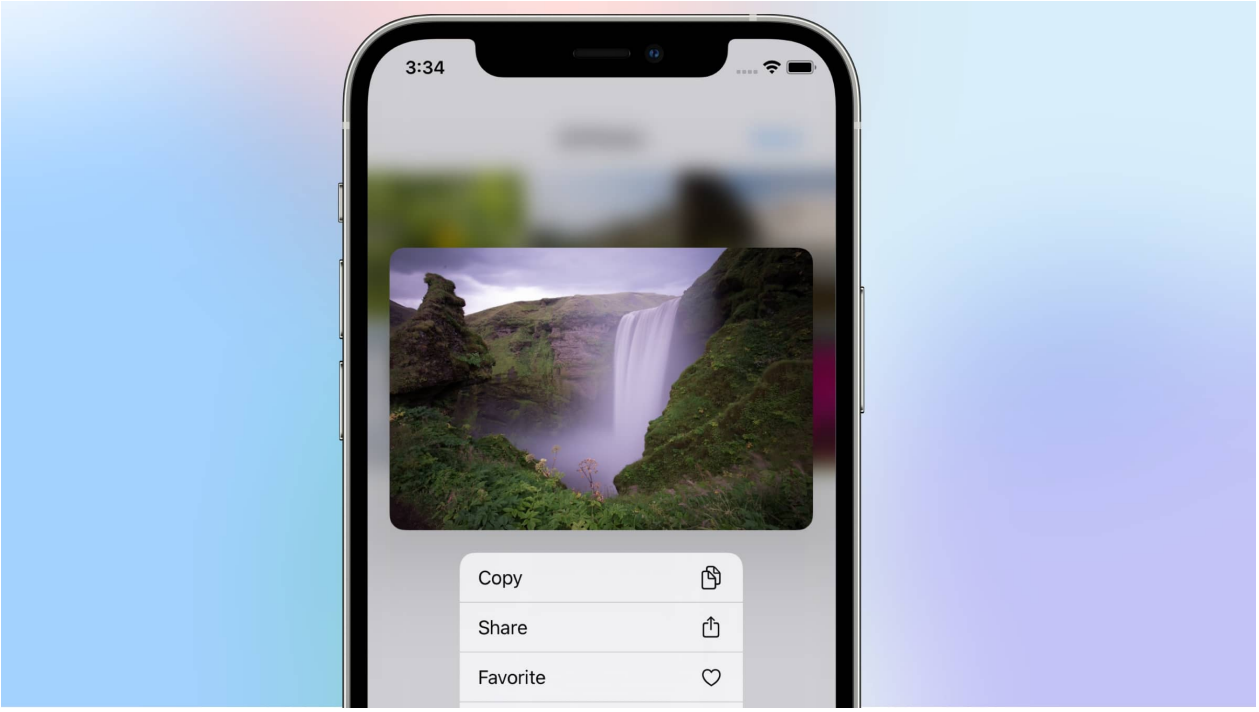
The concept of using a navigation stack and drilling down into other screens demonstrates depth. You can sense it, subconsciously or otherwise, that you going

further down a given path with each choice you make. As a user, you feel it and are cognizant of it, but it also has meaning:



As you use a navigation structure for drilling down into content, you can feel the depth increase but you're never unaware of how you got there. So in this case, depth aids navigation and understanding.

You start at a "shallow" depth, and go deeper. Further, depth is used in the traditional sense of the word:



Depth can also be used in literal terms to help people understand what should have your attention within your interface.

We get a feel for what's on top, and what's below it. And, that depth helps us determine what's important to act on now and what isn't. In fact, in the case of an alert, the depth signifies to someone that there is no other choice, it's the alert that needs your attention.

Even the idea of using "realistic motion to convey hierarchy, impart vitality, and facilitate understanding" can be seen in places where a sheet is presented like in the Maps photos above. You know that you can drag it up and down to see more options, that's depth working in your designs.

All of these ideas come together to create that iOS look and feel, but I think it helps to remember them in the order I've written here. And, coincidentally or not, that's the way Apple presents them too. Clarity, deference and depth.

Following, and Breaking, Conventions Tactfully

Further, I don't want to imply that you can't break iOS design conventions. There are certainly times for it. But there is a line between feeling at home and feeling different just to feel different.

You can still feel at home and *be* different, but there is a correct way to do it. But, you don't get that design muscle working without a foundation in place already. The start of that foundation comes from the three themes above. Then, with practice, you'll know when it feels right to go off the beaten path.

For example, one of my favorite pieces of software is Things 3 from Cultured Code. They have a custom drag and drop implementation that doesn't necessarily follow all of the standard iOS drag interactions, but it's understandable, feels great and makes sense for the context of the app.

The ability to drag entire sections of data around like you can in Things 3 isn't standard on iOS, but it does feel at home. Now, you can extrapolate that idea out further to other parts of your design too. Being different isn't bad, if it was — every single iOS app would use the standard controls and designs and that would be it.

But you can be different from a design perspective correctly and incorrectly, so try to develop a design sense so that you can justifiably tell the difference.

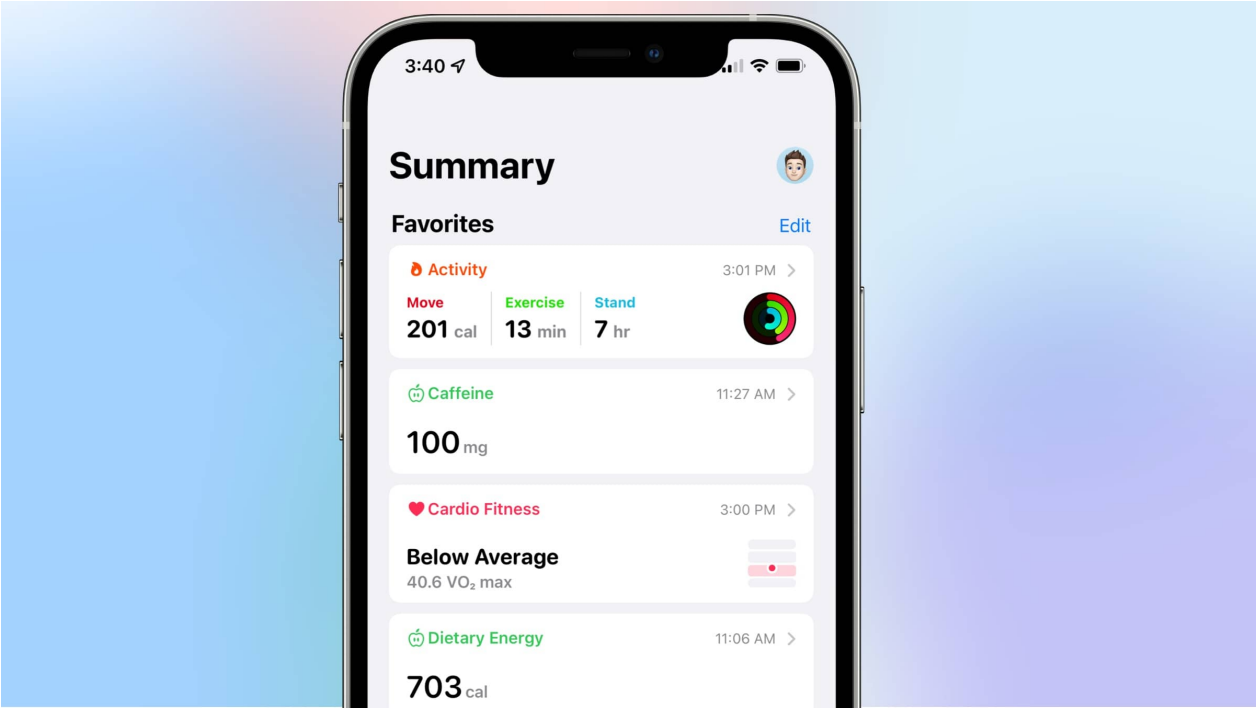
One more quick example. In most cases, to show more data about a certain item, it's typical to follow this flow:

1. Show a list of stuff that has more details about it.
2. Tap on one of them, and present a view which shows more of that data.

On iOS (not iPadOS, where the following idea is more common) it doesn't *typically* go like this:

1. Show a list of stuff that has more details about it.
2. Inline that data on the row.

But, that's exactly what we see in Apple Health:



And I love that. The tiny chart in Cardio Fitness, the details of sleep patterns – it all works. But that's because it was well thought out, and Apple's designers likely realized that users who skim this screen probably want quick shots of data, right away. So, that's one reason to think "differently" about this. But, you have to be able to execute on that "different" idea, and that's what they've done by expanding out the list rows a little higher, and providing some valuable data within them.

If every app worked this way, it would probably be annoying. It's not conventional. But when different ideas strike that "feeling at home" balance, they are wonderful demonstrations of a designer thinking about common problems in a new light.

The TL;DR

Feeling at home means caring about what it truly means to design an iOS app. When at app feels at home, you've won when it comes to designing for iOS.

Keep or Cut

One of the most difficult feats to accomplish in our industry is simply to...ship. To get the first alpha out. The first beta. The first public release. We begin with a set of rigid design specs or supported features, but we know that rarely lasts. Things get added, dates get pushed – this is all doubly true if you are an indie or work on a small team where constraints and more flexible. Wait, maybe I've got that wrong. Large teams covered in top-down decision makers and red tape can suffer development delays for years.

I guess, no matter the situation, it's always hard to ship on time.

So, how do you prevent that from happening? And, what does it have to do with design?

As it turns out, quite a lot. In this chapter, we're going to look at the concept of "keep or cut", that is – how do we advocate for keeping the things we think we should keep, push the things that we think need pushing or removing things that don't quite fit entirely.

Reasons to Keep

It Fits the Mission Statement

Defining your next version of shippable software is something we often think we'll do but rarely keep the promise. One way to stack the odds in your favor is to figure out what really needs to go out, and what can come later. The best way that I've found to figure out what's important is to start at the beginning:

What's the app's mission statement?

All of your choices should ideally flow through this question. When you're deciding what to design, don't just jump in. Ask yourself if what you're designing *actually* helps drive forward the app. Design takes a lot of time, developing that design takes a lot of time — and then doing revisions for both takes a ton of time.

So, before you start in on something, ask if it makes the app do the core that it should better. Is your mission statement "App X helps beginner photographers edit photos quickly"? Then adding in an advanced slider for some obscure element of photo editing that most beginners wouldn't be familiar with is certainly "cut" material for an early version. If fact, it may be "cut" material altogether, actually.

Define that next release and benchmark the things that you think are important against your mission statement, full stop.

Security Implications

This may not be something designers have to deal with a lot, but every now and then there are things you just have to get done due to security implications. Maybe sensitive data is leaking, something shows for user X when its user Y's — you'll know these when they come up.

In such cases, get those out the door and design them with tact as quickly as possible without degrading quality. A lot of folks assume these types of things fall solely into a programmer's domain, but I don't think that's true. Sometimes these type of situations require a lot of copy or explanations, some sort of modal alert or ways to communicate complex things to millions of users. You can help from a design perspective.

It's A Major Differentiator

If you've got some sort of "hook" or novel feature that really sets you apart, make room for it. But only make room for one or two of these things, per release. As a designer, these types of items usually take a fair amount of time to get right. As such, I advocate for still *doing* these things, but don't do them in bulk.

A few of these per release, if that, is a good spot to be in. But be careful, it's easy to justify to yourself and others that almost anything falls into this category. And if we're being honest, a lot of these items don't really reinforce an app's mission statement. They're just fun or new, and those things can wait if they have to.

Reasons to Cut

Chat with Developers Early, Often and Openly

When you've rolled out a design, show it to a developer early. They will know which parts of those need more discussions. Things that you may think are easy, like tweaking a corner radius, could end up being insanely difficult for a thousand different reasons. Or, on the brighter side of things, stuff you may have deemed too difficult may turn out to be easy implementations.

So, bring them into your process. Help them inform your design, and your designs will inform their architectural approach in many cases. But, if push comes to shove, and developer says that something is going to be out of scope or too difficult for the current release, take them at their word. It's fine to cut these things for now, it just means they need a little more time to get around to later.

It Doesn't Truly Make The App Better

I won't spend much time here, because this point is basically the inverse of the first point under reasons to keep designing something. If it doesn't support the app's mission statement, than as a designer – those things should be your first proposed cuts.

It doesn't mean they never happen, they just don't have to happen now. But the design isn't support the app's mission statement quite clearly, then you should take a deeper look at it to see why that is, or if you can change it to better be in tune with the app's purpose.

The Design Isn't Working

There are frustrating moments of design where, even though what you're going for is true to the app's purpose, the design isn't coming together. In those cases, *don't ship it if you can avoid it.* I know there are things outside of your control at your job or within a team context, but if this is just you – if the design isn't there yet, then turn your attention to something else until it does get there.

Shipping subpar designs doesn't feel good, so try not to do it. And I get it, people like to say perfect is the enemy of good and other turn of phrases that sound wise on paper but in reality, they don't always apply to design. The idea isn't to be perfect, it's to put your best foot forward. And sometimes, your best work doesn't come naturally. And in a field such as design, you can't just brute force that work out of you.

A Quick Note on Navigating Disagreements

Just like any other field, there are times where people aren't going to agree on what the best course of action is. What you think should be kept gets cut by someone else, and vice-versa. This is less a possibility and more of a foregone conclusion. In short, it'll come up.

Be ready for them, and learn how to handle them gracefully. It's human-relationship skill that'll serve you well outside of just the field of design, but in any interaction, really. And the biggest advice I can give you that's served me well is to simply try and be agreeable. Look in the mirror first, and consider your viewpoint not entirely as valid as you might've initially thought.

It's easy to put up walls and defend your design choices, or why this feature has to ship or why this feature *shouldn't* ship, and honestly – you'll find that you don't get far because the other side of the argument already thinks your wrong.

Instead, try being agreeable. Seriously. You'd be surprised at the impact it can make in these types of discussions! Try to reframe thoughts from "I don't think that toggle works here" to "I wonder if we've considered moving the toggle to the right to better flow with the rest of the interface?".

And hey, if you get vetoed – then you get vetoed. It happens. Or, perhaps you see things from a new perspective and you change your mind on the matter. But if you want to change someone else's mind of the matter, the best way to start is by being agreeable.

So that's keep or cut. While it sounds like a chapter out of a product management book than it does one for design, remember that you have a large sphere of influence on this. Use it wisely.

The TL;DR

Shipping software is a complex process that is made more difficult by handling competitions opinions and relationships among your team. As such, being diligent, and sometimes flexible where warranted, about what to keep or cut can help you get to the finish line.

Epilogue: An Endless Cycle of Tiny Improvements

I remember starting at the screen of my laptop on a late December night in 2011. In the early parts of the year, I was enjoying a seemingly endless amount of enthusiasm stemming from learning how to code. I wanted to crack away on a project for hours in my spare time outside of school, morning or night – it didn't matter. I could make anything now!

But tonight was different. My mood was a bit somber, and as I eyed my finished project (it was some permutation of the classic To-Do list app) I couldn't help but shake the feeling that it just didn't feel right. It looked amateur, and I was trying to diagnose why that was. I couldn't entirely place it on any particular reason, but I did know that the package as a whole wasn't something I was confident in.

After that app shipped and eventually several others did too, I realized that even though I could write some code and ship a project, I couldn't make any of them look good. Not only that, but I didn't seem to get even remotely close.

Today, I like to think that was the moment which marked the beginning of my design journey. It came much later after I had learned to program, and I didn't even real-

ize the design seeds were being planted in my brain. But, I started to realize how important design was, and that a solid product always has it. Even though I couldn't do it yet, I knew I had to learn.

It was years and years, and then some more years, until I became comfortable with design and all of its intricacies. Whether you are developer starting to learn design, or a well-weathered designer who survived the transition to iOS 7 and beyond — I hope that you can look back after reading this book and believe that you've at least become better at the craft. No matter how big or small that improvement was.

Maybe you aren't ready to lead a redesign at work, or you still struggle a little bit with your own apps — take heart that you are on the right path. Design is not a linear process, even though my personality type wishes that it were. When I code, I see a path towards success fairly early on. I know where to go and how to get there, and even though things may change along the way, the primary choices I made early on usually stick and work out for the most part.

Design is not like that. I'm not sure if it is for *anyone*, honestly. You go through revision after revision, in search of that one moment where you step back and look at things and think, "This might work." Sometimes the approach changes completely. The finished product isn't recognizable from what you started with. That also makes it fun, if not frustrating, for those of us who lean towards having a prescriptive creative process.

The art of learning design is a constant cycle of getting a little bit better over and over again. It looks like micro improvements to your work or thought process, not one monumental revelation. And then, stereotypical as it may sound, one day you realize that you can kind of design something that makes you happy to share with the world.

Maybe that day is right around the corner for you. And if it is (or isn't!) I can't wait to see what you put out there.

Happy designing, and enjoy the process.

Book Three: User Experience

Easy Flying

Je ne sais quoi.

Have you heard of this term before? In literal terms, it's French for "I don't know", but more recently it's been used in English-speaking territories to represent an expression of quality that makes something work exceptionally well. You can't quite peg what it is, but it's this x-factor that makes you drawn to something.

In short, it's just got that special something – even if you can't put it into words.

In iOS apps, and software at large, that's the feeling you get from a superior user experience.

This book is all about how to create a good one for iOS apps.

Defining User Experience

Before we start in earnest, I want to lay out a few things about user experience and the way I think about it. I would rather not claim that my definitions here are the canonical "right" ones, or that this process is foolproof. Neither of those assertions would be true.

Nonetheless, I do want to lay out a few things from the start:

1. How I'm defining user experience in terms of this book, and
2. How that relates to design.

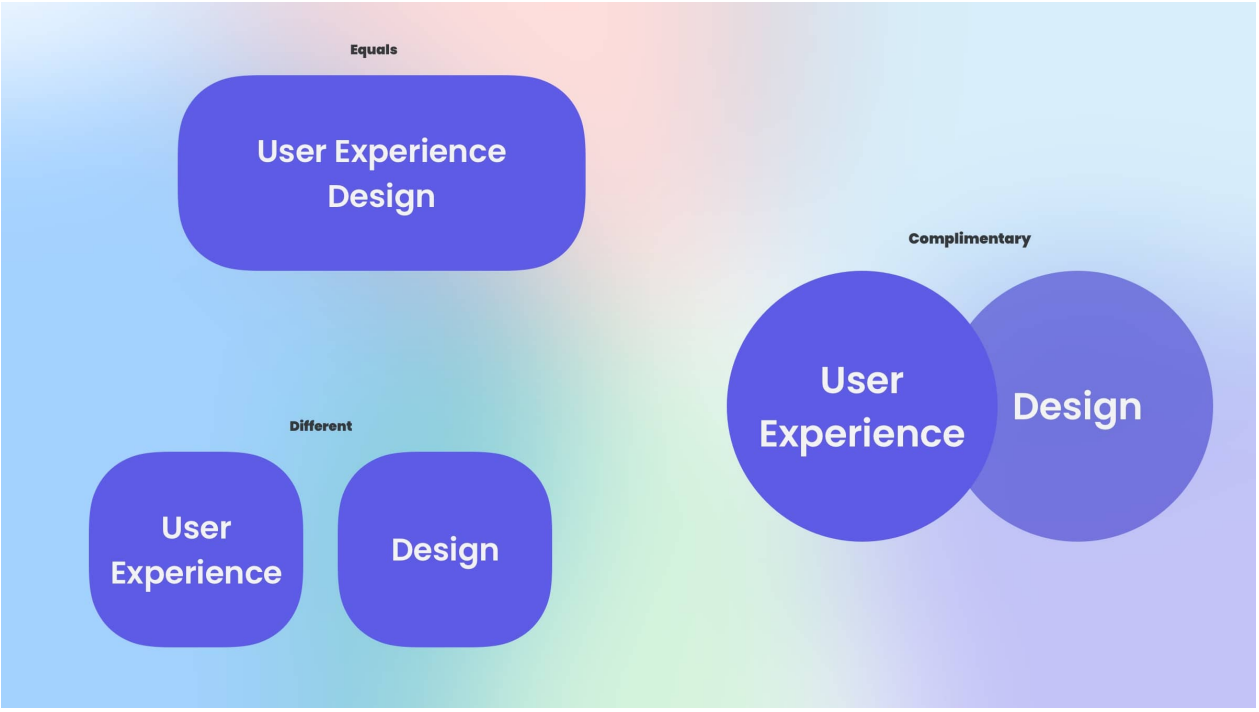
To me, user experience is less about how something looks, but much more about how it *feels*.

Is it snappy?

Do things work without you having to think about them?

How do people feel when they open your app, are they confident it'll do what they need it to? What they expect it to?

Holistically, as the title suggests, it's about the overall experience folks will have with the software that you create for them. And you can affect that experience in several ways, for better or worse. As such, it's no surprise that we often read about "Design and UX" put together, as it's often written, because those two concepts do work hand in hand:



Everyone seems to have a different view on the relationship between design and user experience, and that's okay.

For our purposes, design and user experience are separate topics that are closely related. You need them both. But, both also deserve their own spotlight. So in this book, while some of what I cover may veer into design at some point, the focus is on user experience.

This book series has an entire book over designing specifically for iOS. To get the high-level thinking between how the two are different, consider this: A splash screen.

The design book would cover the thinking behind how you lay out the screen, how to consider fonts, colors – what it should do when the device is rotated. The user experience (this one) book would speak more to how that screen should be snappy and informative, but not in the way. If it relies on a network request, it would talk about how to handle that so that it doesn't impede progress.

It's a bit reductive, and I apologize for that — but one is more about how it looks. One is more about how it feels. And that's how I think about design and user experience.

Unlike design, I think that user experience (and what makes a good one) is a little less abstract than design is. To me, that's exciting. I have a lot of opinions on design, and you might find yourself agreeing with some of them and vehemently detesting others if you've read my book on the subject. But, I don't think user experience really operates like that.

A good user experience, and the practices that come along with how to foster one, are more accepted across our industry than design practices might be. You can rest assured that the tips or methods I'll talk about here are battle tested and work. While not every tip will fit every app, the idea remains the same — make the user experience better for doing X or Y.

Design Meets Technical Expertise

If you've read my book over design, you should know that this book is more technical. Whereas anybody can read my book over design whether your expertise is programming or as a designer, the topics in this book are a beautiful marriage between design-leaning topics *and* technical know-how of Apple's APIs.

To demonstrate, recall my little blurb above over a splash screen. The differences between what the design book might cover about it, and what this one would. The topics this one covers do fall into programming territory quickly:

1. How do we make it snappy? That can certainly become a technical discussion fast.

2. How do we defer network requests? See above, technical.

That doesn't mean this is solely a book for developers or engineers — I simply want to set expectations. This book has design sensibilities that are unlocked by using the right APIs, in the right way and at the right times.

That's also why I found this book to be an absolute joy to write. I love design. I love programming. I love making things feel better. This book is about how to do all of those things smashed together, which — again, means user experience to me.

What This Book Covers

How do we make sure annoying errors don't show up while someone is in the middle of something? How do we save data so people can pick up where they left off? What about making sure folks get into your app quickly, with just enough information to get them started?

There is an art to all of that. We're going to cover topics like them within the context of iOS apps. For example, we all can generally agree that blocking the user while you download a video to play when a certain view opens up is not the best user experience. How can we avoid it? Well, one way is to use Apple's on-demand resources API. So, we'd dive into that. Those are the things we want to unpack in this book.

How will we do that?

This book will carry a similar format throughout its entire contents. We'll begin by presenting what the scenario is *and* what APIs are available to solve it. Then, we cover the big picture of that API followed by more "nice-to-know" tips. Finally, we'll reassert what we've learned by three bullet points (which you'll find here in a bit about this chapter, too).

If you have read any of the previous books in my Best-in-Class iOS App series, you have a great idea of what to expect. If this is your first, the thinking behind this format is threefold:

1. Build intrigue behind an idea.
2. Set the foundation for it.
3. Then, close things out by seeing how we can take it to the next level.

By doing so, we'll hopefully pave the way for apps that not only look great, but *feel great*. That's easy flying in the software world. We don't want turbulence found throughout each flow we create, and for any that do exist we'll smooth them out along the way with the ideas presented here.

Let's do it!

Three Key Takeaways

1. User experience and design are closely related, but they aren't the same.
2. When you create a user experience that is good, users will know it even if they don't particularly realize it. On the other hand, if your user experience is bad or generally poor – it will stick out.
3. We'll look at several scenarios in this book to raise our chances of landing on a good user experience, but all within the context of iOS apps in particular.

Onboarding

Onboarding is a natural place to start with a book on user experience for iOS apps. It's included in nearly every app available today, and when done right – it can be quite useful.

It is, however, easy to get wrong.

Onboarding should be insanely quick, informative and cheerful. In too many apps, though, it's annoying, intrusive and frustrating. Let's take a look at how we can create an experience that leans more towards the former and not the latter.

Keys to Effective Onboarding

When you think of onboarding and how you should build it, you need to contextualize your app and its goals to its own onboarding experience. What I mean by that is the approach you take will change depending on what you're building. You might need a little more or a little less.

With onboarding, you want to accomplish a few goals:

1. **Show "Just Enough":** This is not an ideal time to display a modal view of terms and conditions, badger folks for a sign up or show a huge carousel of features. Show the value of your app or perhaps some critical information, and then get out of the way.
2. **Be Quick:** Onboarding, no matter the form it takes, should aim to be very snappy and quick. You're not looking to display some loading indicators which might be sitting there for any longer than a few seconds (at most) or really *anything* besides showing information from the previous point. In addition, folks should have a clear out and be able to exit it quickly if they want to. Remember, several iOS users have multiple devices — they don't want to experience the same onboarding over and over again when they reinstall your app.
3. **Be Happy:** No matter the app, try to aim for a smile at this phase. Why? Because you're all about first impressions here, and if you can make something as arduous as onboarding a good experience — you're on the right path. I don't mean you need to show jokes or be overly "cute" with the onboarding, but consider your app's voice and how it can be made (for lack

of a better word) delightful within that context. Even if it's a view they'll only spend a few seconds on.

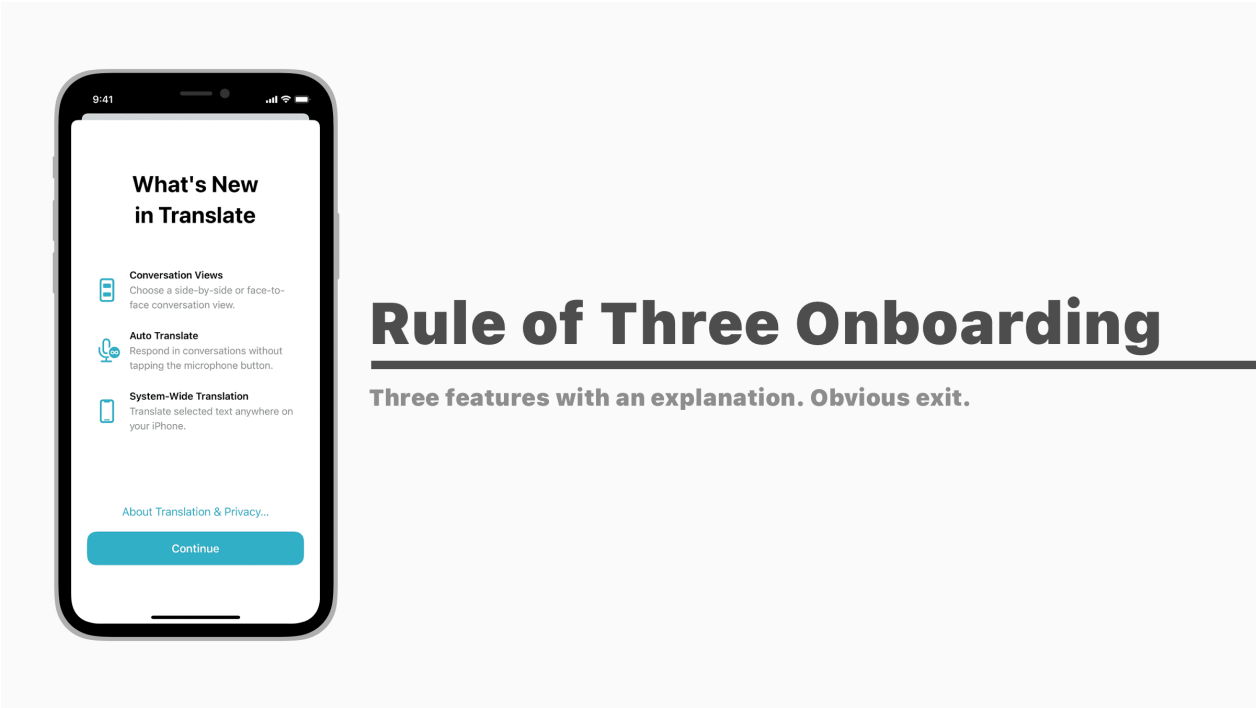
Let's see how some apps tackle onboarding using those three principles.

Examples

Apple's stock applications are used by virtually everyone across the world. They need to convey generalized, friendly experiences. Most of the apps they include are not built for power users[1], but for everyone who might need a phone (i.e. just about *anyone*) with wildly varying degrees of know-how when it comes to using iOS. So, they need to get their point across quickly and easily.

For the most part, Apple follows the rule of three. That is, three main value adds shown with one or two sentences for each of them. That's followed by a button towards the bottom with an obvious call to action, typically to dismiss the screen.

[1] Though some certainly are, such as Shortcuts.



Apple has made this type of onboarding quite common and for good reason. It fits several use cases and apps well.

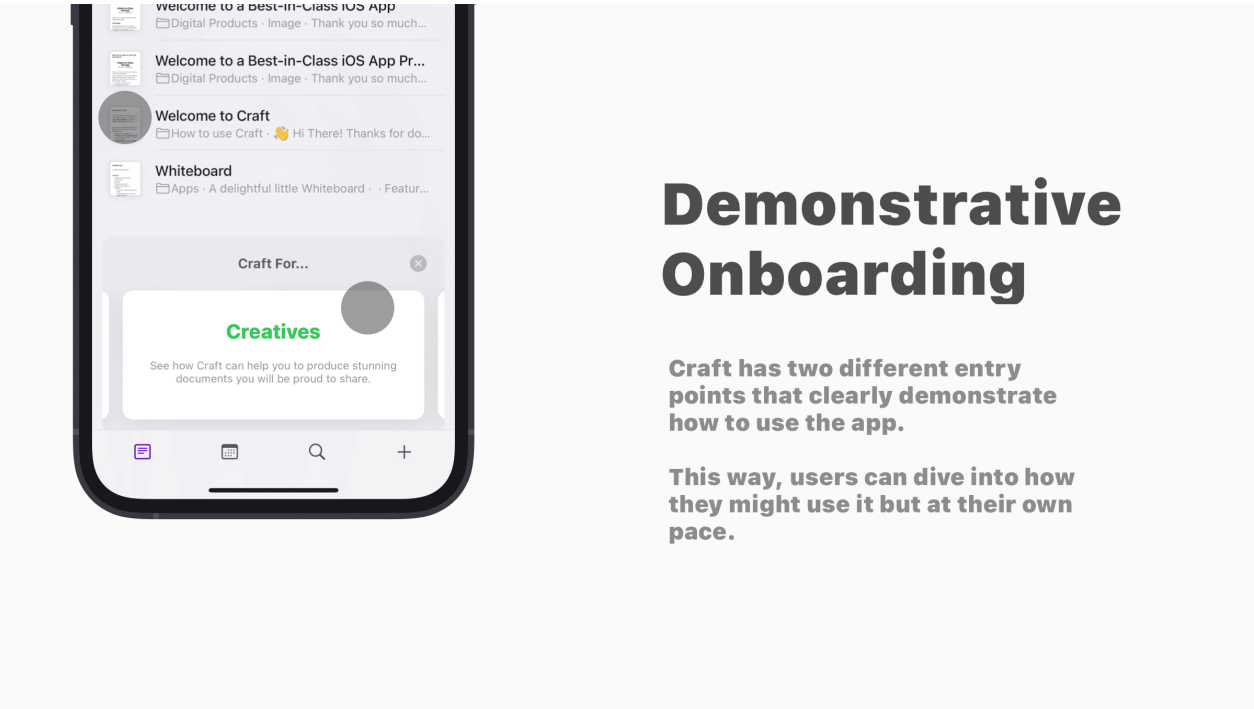
For iOS' translate app, we can see them pick up on three main things:

1. Conversation Views.
2. Auto Translate.
3. System-Wide Translation.

If you've used the app yourself, you won't be surprised to learn that those are three core features. Using this type of onboarding is extremely popular, in no small part because Apple uses it almost exclusively for all of their apps.

It's also comfortable to parse. Humans read off list of threes very easily, and better yet – they are relatively easy to remember. If this was four or more things, that "stickiness" is quickly lost. Worse, the features you call out begin to diminish one another – because how are fourteen features more distinct and memorable than just a couple?

Next, let's take a look at an app that has a lot of utility and features behind it, Craft Docs.



Craft Docs allows you to go back and try out demo documents while also providing a prompt which shows you how to use it depending on your use case.

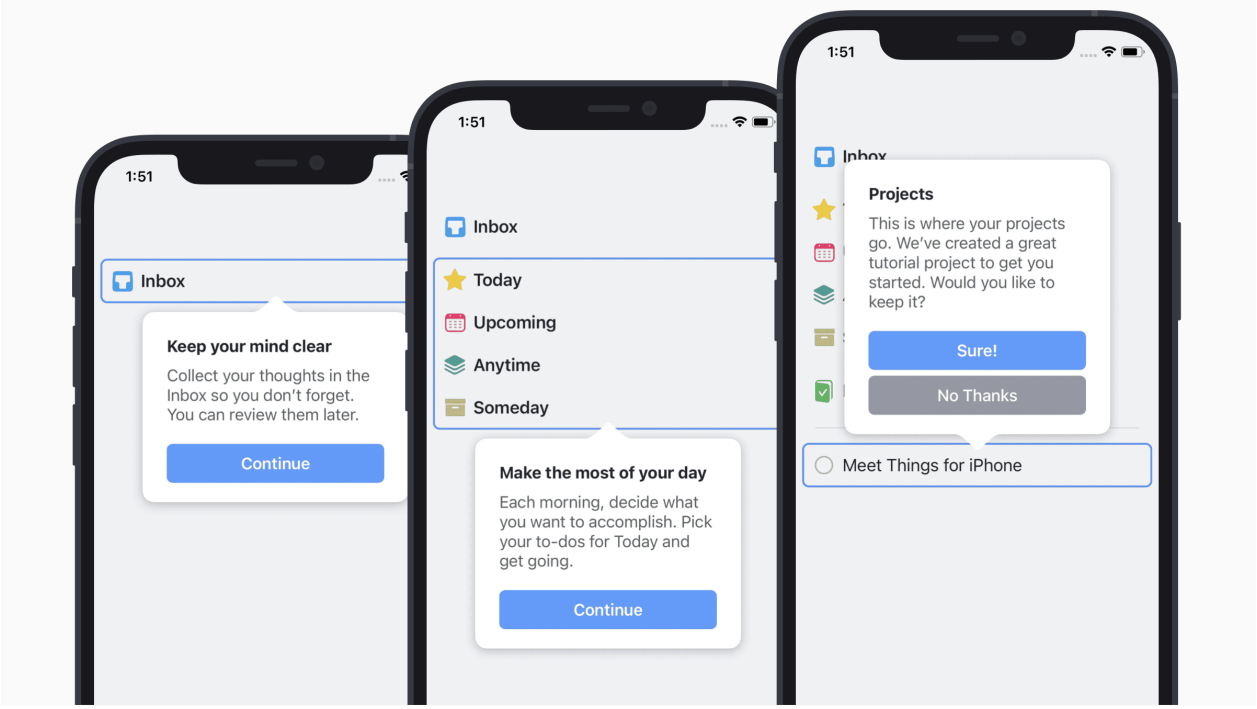
This approach takes a "show, don't tell" route. People who download Craft can learn along the way, and that can be a very powerful method to use. When you make an app that has several features or use cases, this could be a great option. A note taking app (not to sound reductive to Craft, but to generalize it) can be used in a thousand different ways. Further, its target market could be quite varied.

Teachers could use it to plan assignments, authors may rely on it to outline books (which is exactly how I use it) or anything else in between. Because of that, you might lean on two assumptions:

1. The people who download this app are likely familiar with how it may work already.
2. They also might need a quick guide to get them up and running, but they may not want that information right away. The want it only *when they want to see it*.

This type of onboarding achieves those goals quickly by letting them into the app and showing them obvious guides that are available at their own pace.

On the other hand, there is also the opposite approach to that which can also be executed well. Instead of "show, don't tell" – there is a "show *and* tell" approach used by Things 3:



Things 3 takes a more interactive approach to its onboarding.

Here, when folks open up the app, they go through an abbreviated experience which shows them how to perform critical actions. This one can be irreplaceable when it

comes to showing users who to get things done, but be cautious here — you run the risk of annoying people right away, too.

I think, in this case, Things 3 does it well and it works. There is a way to skip the process if you wish, but the tips are so informative and quick that I don't mind to run through them all. Also, they also include a "how to" project that's available at any time so people can get a refresher from that information if they need it.

To recap:

1. Apple uses a "rule of three" approach.
2. Craft shows sample documents teaching people how to get the most out of it.
3. Things 3 walks new users through a quick project, having them perform actions to teach them how to use it.

Those are three different approaches to onboarding, but each of them fit the context of the app for which they were built. More importantly, they all reinforce the original three rules we started off with. They show just enough, are quick and delightful.

Harder Than It Looks

However, these types of experiences aren't as easy to build as you might think. Let's take arguably the simplest of the previous examples, Apple's "rule of three" approach. Consider the following questions:

1. What happens when they rotate the phone?
2. What happens if they *opened* this in landscape mode?
3. Or, what if they've got the maximum Dynamic Type setting enabled?

How you tackle of all of those things determines how well your user experience will be. This idea embodies what I started off mentioning in this book. User experience is a mash-up of design and technical skills. Here, design aspects come into play with considering if the "rule of three" is right for your app. If it is, the typography, colors and more determine how well it will be received.

Beyond that, though, the code comes into play. When creating this view, you've got to make sure it scrolls, responds to accessibility settings and different view sizes. When you think about those types of scenarios, you're on the right path.

Tips

Provide a Way Out

If you don't provide an escape hatch for your onboarding, you're in for a bad time. Folks may already be familiar with the app or simply don't care enough to read what you have to say.

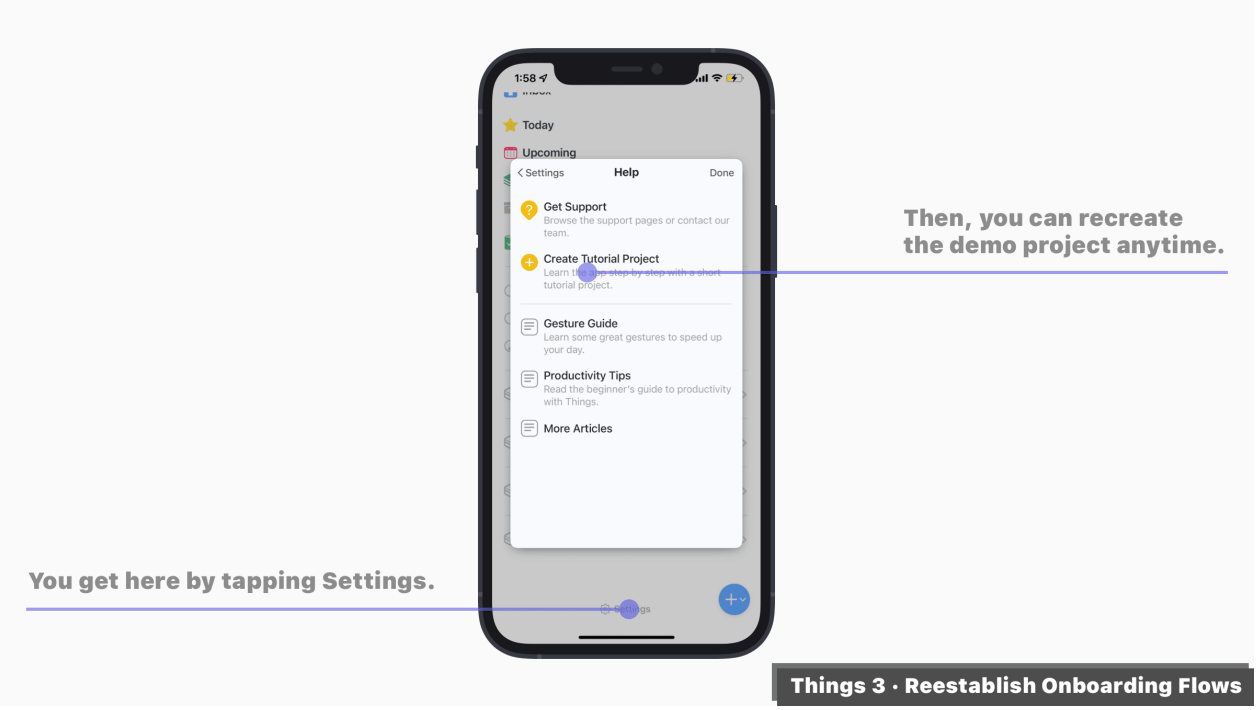
No matter the approach you take, be sure to provide a button to let them skip past it. Much like how showing a ratings prompt far too early in the user flow of an app is a quick way to get a bad review, so too is forcing people to do something they simply don't want to. People want to feel like they are in control of the apps they use, not the other way around.

Make it Repeatable

Given the above point, make sure your onboarding is easy to get back to. A lot of folks, like it or not, simply download an app to make their decision of whether or not

they *actually* want it. They want to hop around, see what it has to offer or confirm their assumptions about it before committing it to their Home Screen.

Because of that, they may decide they do want to see what your onboarding had to offer, after all. So, make it easily accessible for them. An obvious place for it is in some type of settings or configuration view. To revisit Things 3 approach, they've got an option to restore the demo project within settings should you want to experience that flow again:



Show It Correctly

You want to show onboarding experiences to new users with fresh installs, that should be your timing. You can cache a value in `UserDefaults` to track whether or not it has shown:

```
// Xcode -> OnboardingFig1.swift
```

```
struct Store {
    static let OnboardingDefaultsKey: String = "hasViewedOnboarding"

    var didViewOnboarding: Bool {
        get {
            let defaults = UserDefaults()
            return defaults.bool(forKey:
Store.OnboardingDefaultsKey)
        }
    }

    func markOnboardingViewed() {
        let defaults = UserDefaults()
        defaults.set(true, forKey: Store.OnboardingDefaultsKey)
    }
}
```

From there, you can make decisions on when to present it. For UIKit, a great place is in the scene delegate:

```
func scene(_ scene: UIScene, willConnectTo session: UISceneSession,
options connectionOptions:
UIScene.ConnectionOptions) {
    guard let windowScene = (scene as? UIWindowScene) else
{ return }

    window = UIWindow(windowScene: windowScene)
    let store = Store()

    if store.didViewOnboarding {
        window?.rootViewController = MainViewController()
    } else {
        window?.rootViewController = OnboardingViewController()
    }

    window?.makeKeyAndVisible()
```

```
}
```

For SwiftUI, you'll want to look at your `App` struct and leverage the `@AppStorage` property wrapper, which itself is backed by user defaults:

```
// Xcode -> OnboardingFig1.swift

@main

struct Store {
    @AppStorage("hasViewedOnboarding") var hasViewedOnboarding: Bool
= false
}

struct MyAmazingApp: App {
    @StateObject private var store: Store = Store()

    var body: some Scene {
        WindowGroup {
            // Backed by user defaults
            if store.hasViewedOnboarding {
                ContentView()
                    .environmentObject(store)
            } else {
                OnboardingView()
                    .environmentObject(store)
            }
        }
    }
}
```

Load Critical Assets in the Background

There are some onboarding experiences which display some video or media asset. Fine, but if you do – don't make people wait for things to download. If your app opens

to a loading indicator before anything else, that is going to be an early exit for many potential customers.

Even better, maybe rethink your onboarding and show that type of thing in a spot where they can access it at their own pace, like a settings or configuration screen. When you have material that you depend on to teach users, make sure they are fetched in the background where users will never notice them.

Apple has a framework built specifically for this type of thing too, called on-demand resources. We'll look at them later on in the book. But, you're free to use any sort of networking approach – a lot of companies host this data themselves.

For Reinstalls – Skip Onboarding If Possible

As previously mentioned, it's quite possible that users will download your app on another device. For example, someone could have an iPhone and an iPad. Or, maybe they simply bought a new iPhone. You get the idea, your app could be on all of those devices.

In such cases, you can still be smart about onboarding. It would be better to pop them right back into the action than it would be to show onboarding again. If your app uses CloudKit, you can accomplish this with `UserDefault`'s syncing variant, `NSUbiquitousKeyValueStore`:

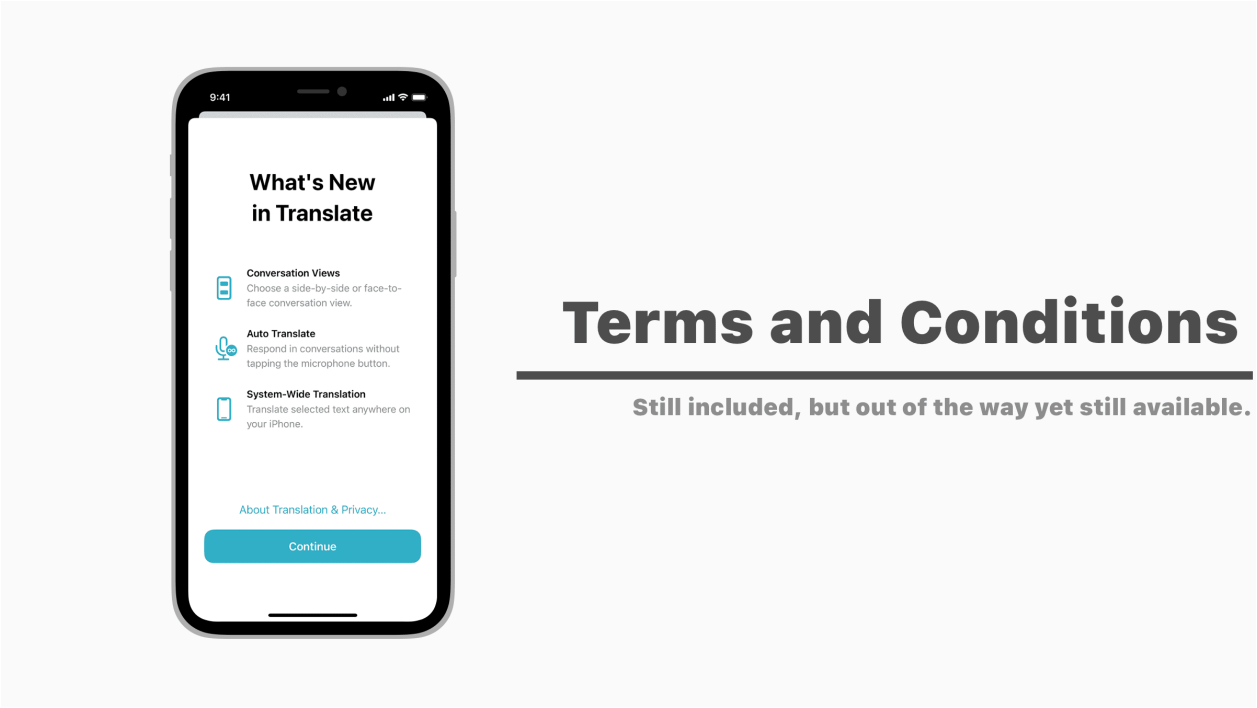
```
NSUbiquitousKeyValueStore.default.set(true, forKey:
"HasViewedOnboarding")
```

This isn't foolproof – the store may not have synced by the time your app has launched and it's ready to go, but it can work in several conditions. You can expand this idea out to other parts of your onboarding too, such as if you've got a more hands on experience, by tracking which parts of it they've already done.

Since this isn't foolproof, though, it also drives home the importance of making your onboarding skippable.

Handling Legalities

Sometimes there is no getting around having to show the "T's & C's", as we like to call them. In that case, show them unobtrusively. In our leading example with Apple's Translation app, I think they handled it gracefully by showing it in secondary colors towards the bottom:

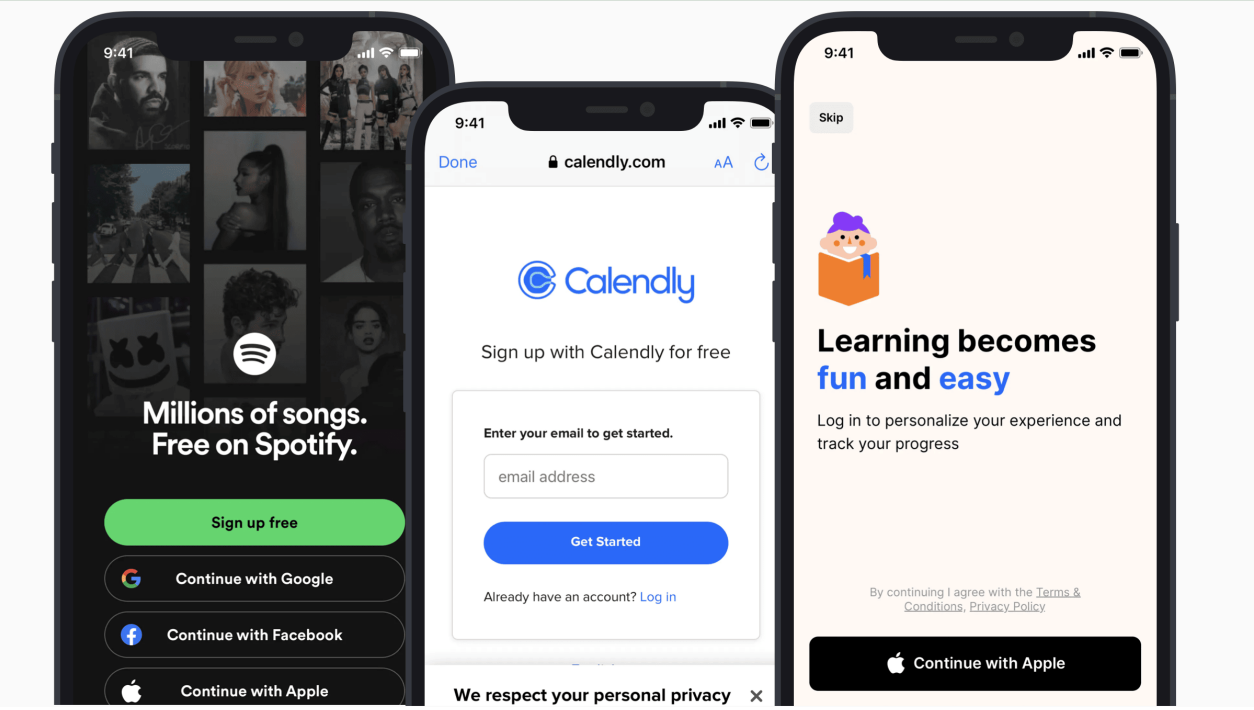


I get it, nobody reads those. However, if they have to be there then they have to be there – and I think putting them as out of the way as possible while still being accessible is a great way to handle it.

Defer Logging In If You Can

A lot of apps have a log in or user account. I've downloaded several apps that I was interested in taking a spin, but I was brick-walled by a required sign up. So many apps require a sign up of some sort. However, some will enforce it before you enter and some don't.

Look at the photo below and ask yourself if you're more likely to try and potentially subscribe to one of these services if you could try it out first, without going through sign up flows:



If you can do it, try to avoid making someone sign in before they get to see all of the great things your app can do. Let the users in your app, or give them an option to experience parts of it, without having to make the commitment to yet another log in to create, remember and maintain.

Three Key Takeaways

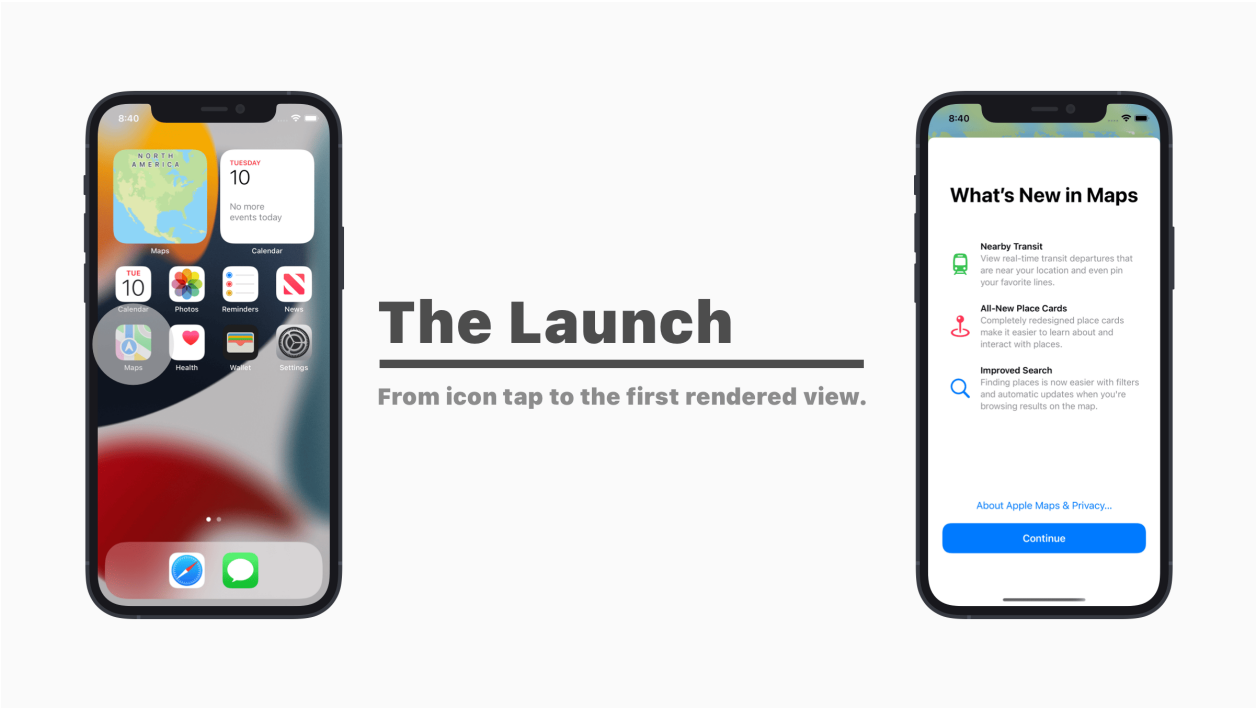
1. Onboarding is often the first experience of using an app, so you want it to be done tactfully.
2. Remember to contextualize your onboarding against the three goals of what effective onboarding does.
3. More than anything, ensure that your onboarding gets out of the way if users don't want it, *but* it's also available for them later on, too.

Launch Time

A sluggish app launch time is a great way to make a poor first impression. If you combine one with a poor onboarding experience, you may not have many chances (if at all) to right the ship with prospective users. It's often said that performance is a feature, and I agree with that.

But, performance is also inexorably tied to user experience. If tapping a button doesn't do something immediately, scrolling halts and hitches or the app takes five seconds to launch – you've got some major user experience issues that can be traced back to performance bottlenecks.

So, what goals should we have for launching our app? There really is only one you should care a lot about, and that's simply speed. Launching should be quick. In this case, we're defining "launch" as the time between when someone first tapped your app's icon to getting to the first interactive parts of it.



Defining a launch.

So, what's fast? **Apple has set the standard at 400 milliseconds**. Or, in plainer terms – about half a second. Poke around your home screen on your iPad or iPhone right and launch a few apps, try to get a feel for how many of them get close to that mark. After a little digging, you can probably find ones that feel fast and snappy and some that don't.

Here, we'll look at how launching works on iOS and possible reasons why your launch may not be as quick as you'd prefer. Most importantly, we'll explore some potential avenues you can take to make it better.

Understanding the Launch Process

If you know how something works, you can better understand how to fix it. As such, it can help to have a good overview of what exactly occurs once someone taps on your app's icon. When it's launched, it can happen in one of three ways:

1. **A Cold Launch:** This is what we typically think of as launching. Your app is read from disk into memory. This is essentially a "cold boot" or a complete reboot of your app.
2. **A Warm Launch:** Your app is already spawned and in memory here, so it'll launch much faster.
3. **Resume:** This *isn't* a launch, but it's often thought of that way. When you have an app you recently used in the app switcher and go back to it, this is most likely considered a "resume" and not an actual launch. I'm calling this out so you're aware of it, namely – that you don't use it to measure launch times because it wouldn't be an accurate representation of it.

What we're trying to improve here is the first one, a cold launch. There are several ways you can improve launch times, but they all occur at specific points in the launch lifecycle. Speaking of, here is what typically happens during a cold launch:

1. **System Interface Initialization:** This happens in two phases, and in those two phases iOS does all it needs to make sure the system is ready to launch your app. The first part deals with `dyld` – otherwise known as the dynamic linker. This process loads up all of your shared libraries and frameworks. Secondly, your app's low level components are created by `libSystemInit`. This is system-side work and it typically has a fixed cost.

2. **Static Runtime Initialization:** Here, the Objective-C and Swift run times are initialized.
3. **Interface Initialization:** Now, interface elements are prepared for us in the following steps, plus `UIApplication` and `UIApplicationDelegate` are created.
4. **Application Initialization:** Delegate callbacks occur at this phase, and generally the code you write begins to come into play. For scene-based apps, scene delegate methods are called here as well.
5. **Initial Frame Rendering:** Your first views are created, their layouts are calculated and finally they are drawn.
6. **Extended Phase:** This may or may not occur – it's up to you. But, it *will* occur if you have specific code or logic to perform in between the app's first frame commit and the final frame rendering.

Of those six phases, you can affect nearly all of them. Some, such as `libSystemInit`, have fixed costs that you don't have any control over. But, even for things that you can't directly reach into (such as `dyld`) you can still do things that make it or work better or worse. No matter the process, the rules typically remain the same:

Simply put – the more that you do, the slower you might be.

And, technically, *where* you do it. For example, loading data synchronously (instead of asynchronously) in your app delegate is asking for trouble. Let's look at each phase and see if there's anything we can do to make it perform optimally.

System Interface Initialization

When the dynamic linker process switched to dyld 3 around iOS 13, developers gained performance wins without having to lift a finger. Why? Mainly due to caching.

Your runtime dependencies are now cached on the system, which should help speed things along without any work.

However, there are still things you can do at this first phase to promote faster launches. The first is to avoid linking unused frameworks. This is a good practice no matter if you're optimizing a launch or not, as you don't want to bring in code that you aren't using anyways. The thing is, nobody does this intentionally. But over a few years, perhaps you've added a few libraries and forgot to remove them once they were no longer relevant. So, check for that.

Next, there are few things you actively want to avoid *if you can*. Using `DLOpen` or `NSBundleLoad` will result in dynamic library loading which will probably undo the performance gains `dyld` gives you through its caching methods. Basically, to be a good steward in this phase of launching – opt to hard link all of your dependancies.

As we've mentioned, when it comes to `libSystemInit` (the second phase of the first part of a launch) – there's not much we can do here. It's mostly up to iOS and we don't have any influence over how it performs.

Static Runtime Initialization

For the most part, this falls under the same category as `libSystemInit` – there isn't anything to do here.

So when do you need to consider it? For those of us who use static initialization. You want to look for any code that follows a static initialization pattern:w

```
+[MyAmazingClass load]
```

This will run every single time a launch occurs, and you can easily sidestep it in most cases by using lazy initialization:

```
+[Class initialize]
```

Even if you don't write the code above, it could be present in some third-party dependancies you've included in your app – so be sure to look for them.

Interface Initialization

This is another phase where there isn't much you need to do. The only time you will need to dig into it is if you've subclassed `UIApplication` or do anything significant in a `UIApplicationDelegate` *initializer*. If that's you, first – try to figure out a way to *not* do that!

But, if you must, consider all of the typical performance and data loading tips. Don't perform heavy network requests, block the main thread or bring in huge assets into memory. If you do, opt to perform those operations later on and on a background queue.

Application Initialization

This is the area where you can most likely have the biggest impact when it comes to improving your launch time. If you're using a non-scene based app with UIKit, your code in these places comes into play:

- `UIApplicationWillFinishLaunching`.
- `UIApplicationDidFinishLaunching`.

And, if you're using scenes (which is preferred):

- `willConnectToSession`.
- `sceneWillEnterForeground`.
- `sceneDidBecomeActive`.

All of the things you do there should be written with one rule in mind – do the bare minimum. Setup critical objects and interface and then return as quickly as you can.

For scene based app, ensure you're sharing resources between scenes as well – which comes into play when multiple windows of your app are created.

Initial Frame Rendering

At this point, your interface is coming to the screen. All of the typical view lifecycle callbacks will occur. You can make things better here by trying to flatten hierarchy and avoiding complex layout calculations. If you can reduce the number of views that you require to get going, all the better.

Extended Phase

This is your app specific logic within the window of the time between your first frame commit and it showing to the user. The takeaway here is to be responsive and interactive. And, you stay that way by ensuring your app loads data off of the main queue if it has a high cost, network loads aren't blocking the user and generally being aware of events that could block user interaction.

There is a lot you can learn about the launch lifecycle. From Grand Central Dispatch policy tuning, sandboxing, runtime sys call elimination and more – it can be intimidating. But, if you can grasp a basic knowledge of how launching works, you can make quick wins. With that, let's finish up with some general tips for fast launching.

Tips

Storing Launch Data Correctly

If you need to install files or otherwise any arbitrary data during some part of the launch phase, ensure you're putting things in the right places. And what is the right place? That would be in your app's `~/Library` subdirectories.

For any mission-critical information that you want backed up, look to the application support directory:

```
// Returns the app support directory
FileManager.default.urls(for:
      .applicationSupportDirectory, in: .userDomainMask)
```

Plus, you can create subdirectories there as well if you need. There are several places to store data in iOS, from caching data to storing things temporarily in a temporary directory. As long as you grab them from the file manager, you should be in good shape so long as you consider the search path directory you request.

Prewarming

Starting with iOS 15, the system can potentially "prewarm" your app. You don't have direct control over this, but I'm including it here because you should be aware of any advancements that happen from iOS-to-iOS versions.

In short, prewarming your app means that iOS executed the app launch sequence up to, but *not* including, the `main()` function of `UIApplicationMain(_:_:_:_:)`. In most modern Xcode projects, this step isn't even included within the project anymore and the Swift compiler takes care of that for you. But, what you should know is that iOS will build and cache what it can to anticipate a full launch.

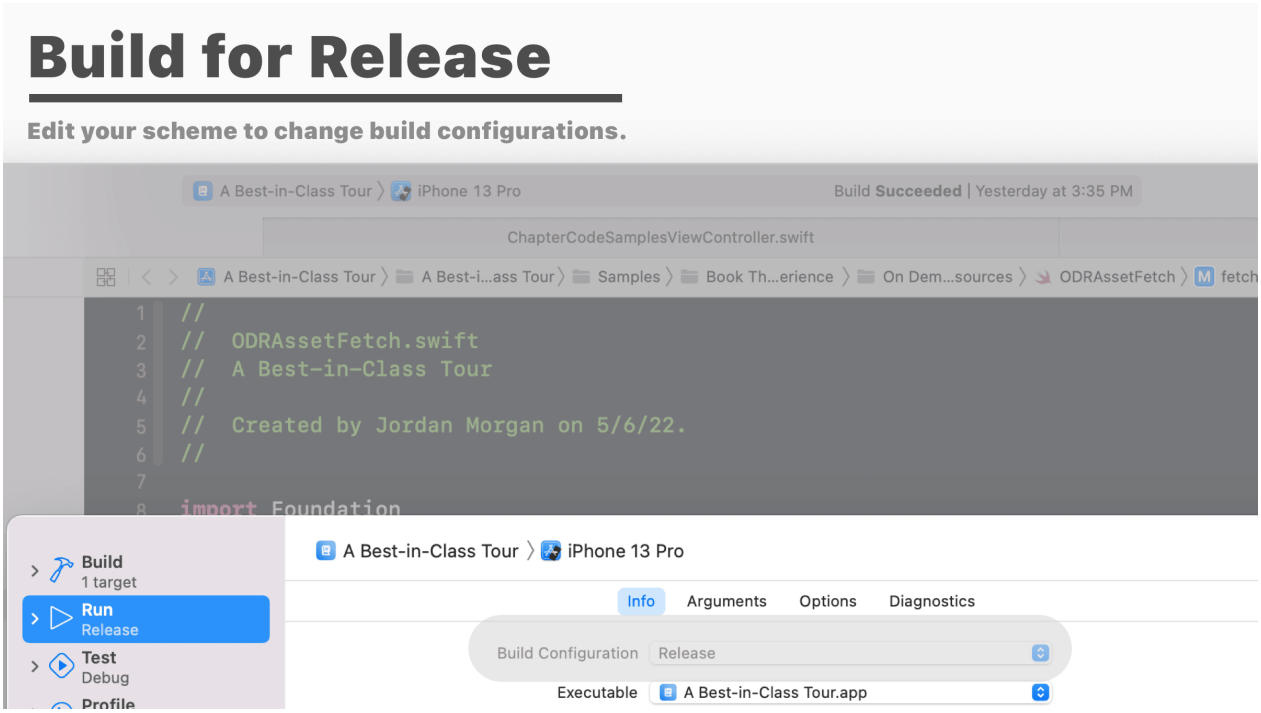
So, what does this mean for you in terms of benchmarking? Namely, if you are measuring app launches and some are faster by a little without any code changes, it's possible that your app was prewarmed.

Measure Accurately

Launches vary under a lot of different conditions. Because of that, you want to try measuring it similarly as possible on a wide array of devices. Try launch tests on three devices if you can, preferably using a min-mid-max approach:

- Test it on the "slowest", or oldest device, that your app can run on.
- Test it on the most common device.
- And finally, test it on the high-end "Pro" models too, if you can.

Also make sure your app's scheme is set to "Release" to get an accurate read:



You'll need to build for release to get an accurate assessment of your launch time.

Finally, networking times and conditions are out of your control (aside from what queue you call them on) so it's best to test on Airplane Mode if possible. This way, you know you're measuring the things that you have direct influence on.

XCTest

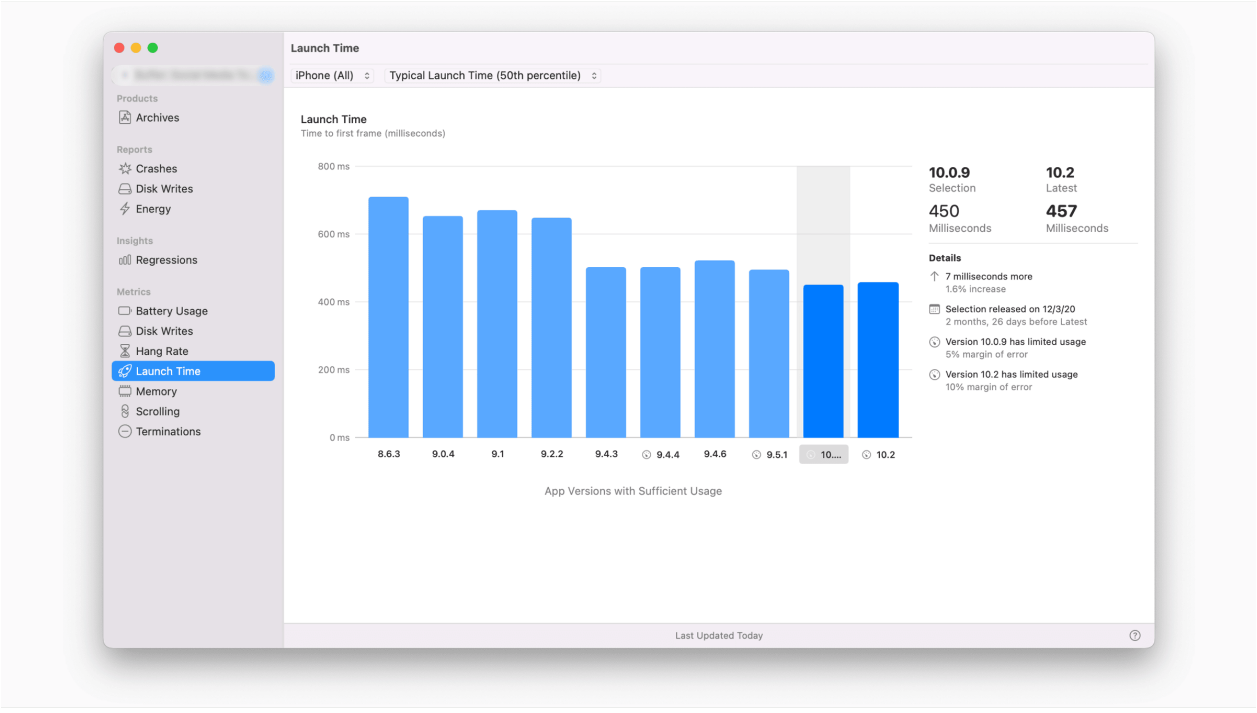
There is a dedicated sign post value that you can use with Xcode's testing suite to measure launch times:

```
measure(metrics: [XCTOSSignpostMetric.applicationLaunch]) {
    XCUIApplication(bundleIdentifier:"com.your.identifier").launch()
}
```

Using it, you can get readouts of how your launch is performing via timestamps. Auditing those is a great way to ensure launches are always considered as your app matures, whether it's tied to your continuous delivery pipeline or just ran manually.

Leverage Organizer

If you open the Organizer in Xcode, you can gain valuable insights about how your app is launching without having to dig for any of the data yourself:



This is a great way to spot regressions between versions and get a high level understanding of how your app launch sequence is performing. I tend to check it between versions once enough data has rolled in to get an idea of where things are at.

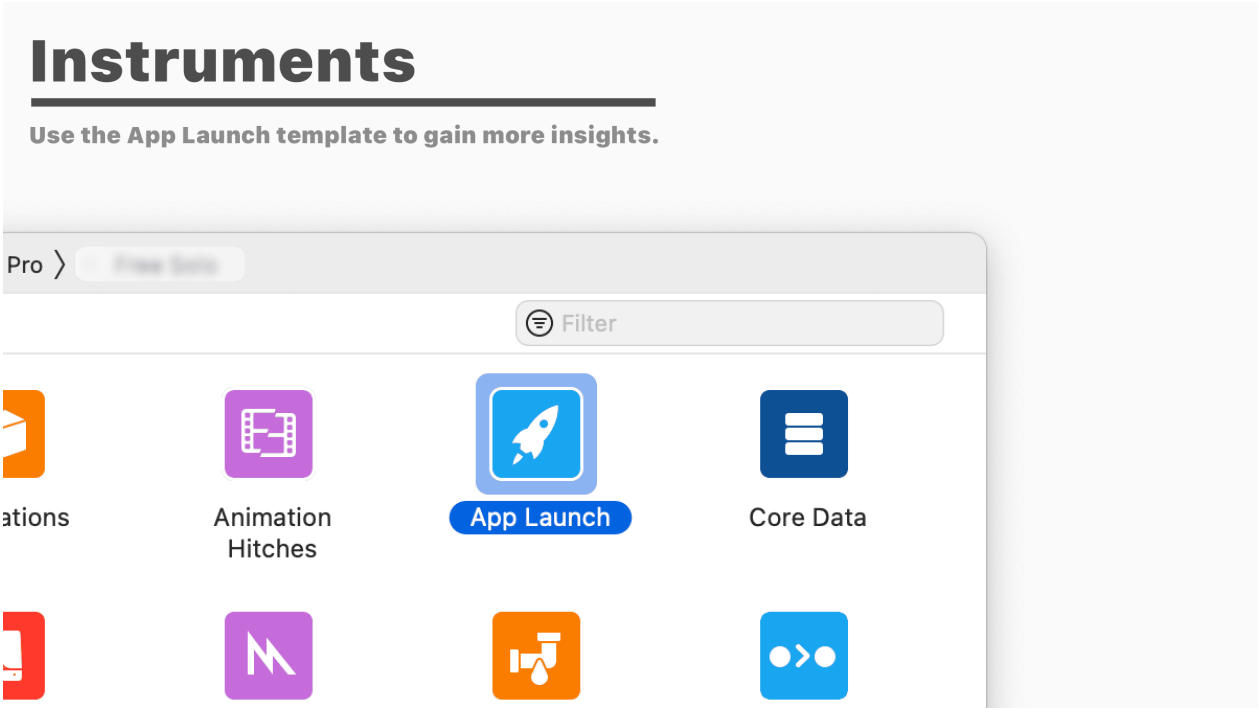
Keep in mind that this data is only going to come in from folks who opt into Apple's prompt during iOS setup that asks if they want to share this information with developers. I don't have any hard data on how many folks end up opting into this, but from what I've heard from others, it's safe to assume it's around twenty-five percent.

State Preservation

Preserving your app's state is a great practice in general, but it can also help with ensuring a good launch time. We've got an entire chapter to cover on this, so please visit that for more in-depth looks at what we can do here.

Use Instruments to Benchmark Launches

There is an entire template in Instruments to measure app launches, and this should be your number one tool in optimizing it. Before you use it, be sure to put your app's scheme in Release mode (from debug – which is generally what's set in most cases).



A few things to remember when using it are to find anything happening on the main thread that may be taking too much time. Since "blocking main", as many typically call it, incurs heavy cost on iOS – it is usually the first spot to check. UIKit and SwiftUI are interface frameworks that run on the main thread, so if you block it for long – users will notice.

Your goal with this template is threefold:

- Minimize work: Find ways to defer anything unrelated to pushing the first frame of your app.
- Prioritize work: If you are doing everything on main in terms of loading in critical data or network calls, consider using a background queue or auditing your quality of service on any existing ones.
- Optimize work: Now, make sure you've covered your codebase for any other wins you can manage. This might look like limiting the amount of data

that you need *only to launch*, considering the right data structures or lazily computing objects where possible.

Consider MetrickKit

If you need more fine-tuned data handling, consider adopting MetricKit. Over a 24 hour period, MetricKit aggregates data and delivers it to your app — and then you can handle it any way you see fit. If you conform to `MXMetricManagerSubscriber` you can get information about launches there, but in theory — this should be the same data you see in Xcode's organizer.

What About SwiftUI?

There hasn't been much (that I'm aware of) official documentation from Apple as it pertains to how the launch lifecycle could differ, if at all, with SwiftUI apps and UIKit ones. That said, even if your entire app is built with SwiftUI's `App` paradigm, you can still benchmark it as you would any other UIKit app using Instruments, XCTest and all of the other tools I've mentioned here.

Take Advantage of DYLD Diagnostics

The first phase of the launch deals with `dyld`. If you want to get more diagnostic information, you can do so by the various flags it supports. If you open Terminal, you can type `man dyld` to see all of the environment variables you can use within Xcode to get more information about your launch.

As of the time I'm writing this book, here's what we've got available to us:

```
DYLD_FRAMEWORK_PATH
DYLD_FALLBACK_FRAMEWORK_PATH
DYLD_VERSIONED_FRAMEWORK_PATH
```

```
DYLD_LIBRARY_PATH
DYLD_FALLBACK_LIBRARY_PATH
DYLD_VERSIONED_LIBRARY_PATH
DYLD_IMAGE_SUFFIX
DYLD_INSERT_LIBRARIES
DYLD_PRINT_TO_FILE
DYLD_PRINT_LIBRARIES
DYLD_PRINT_LOADERS
DYLD_PRINT_SEARCHING
DYLD_PRINT_APIS
DYLD_PRINT_BINDINGS
DYLD_PRINT_INITIALIZERS
DYLD_PRINT_SEGMENTS
DYLD_PRINT_ENV
DYLD_SHARED_REGION
DYLD_SHARED_CACHE_DIR
```

To use them, simply edit your scheme within Xcode and add them as an environment variable. This can be a great way to quick insights into how the dynamic linking processing is working during launch. Check Xcode's console window to get the resulting data.

Three Key Takeaways

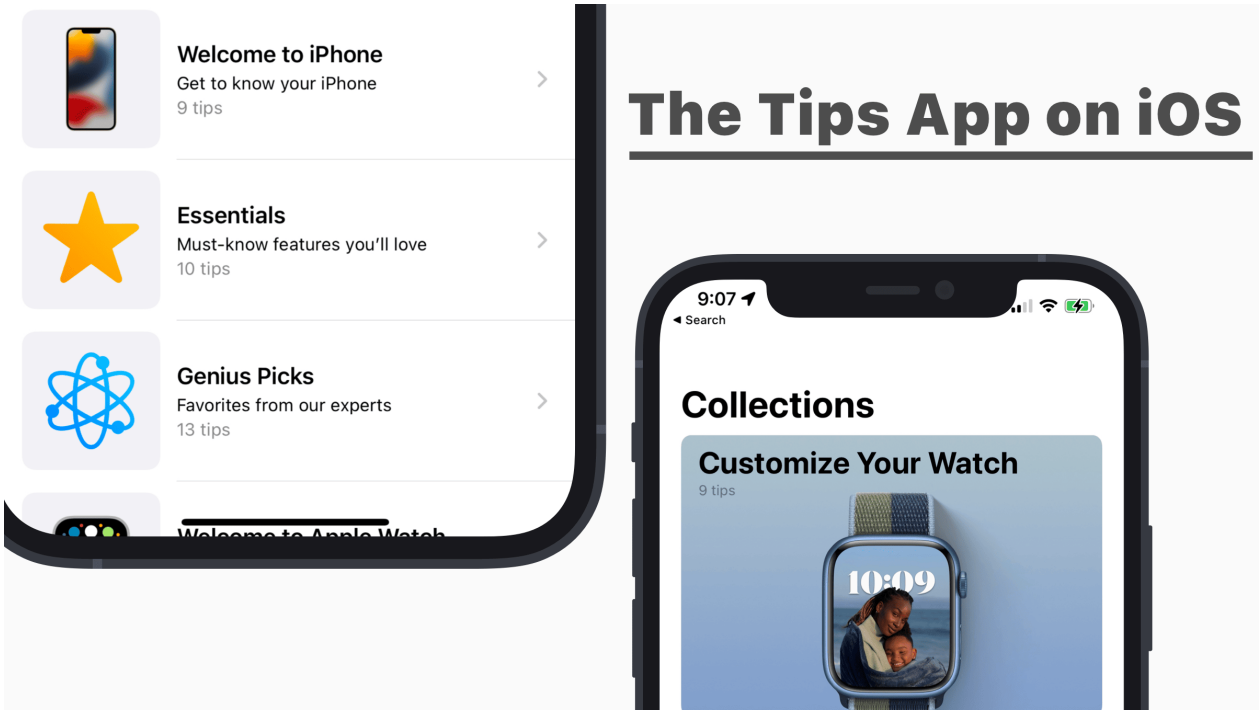
1. Launching is the very first experience anyone has with your app, so invest time into getting it right.
2. While there are several processes that you can't control, there are several that you *can*. Taking time to understand how it works can help you address issues.
3. Be sure to continually check into your launches and ensure you aren't regressing in terms of launch speed as your app matures.

On-Demand Resources

In many cases, there are assets that would be valuable to have in your app, but you might be hesitant to include them since they make your app's install size larger. It's a trade off, not unlike anything else in software development, that you have to make. In the last chapter, I made the argument that your app's launch is the first experience people have with your app.

I guess that's wrong – your download is. So this chapter is all about a useful API that helps you keep it slim by fetching assets post-download.

For example, consider a brief "How-To" video, or a section of tips and tricks you might include for your app. In fact, let's say you wanted to mimic the look and feel of the Tips app itself, found on recent versions of iOS:



The Tips app on iOS.

While those videos are certainly well done and helpful, weighing down your app's binary size with them may not be the best idea if you were trying to implement something similar. To help solve situations like this, Apple has *on demand resources* that you can leverage. On demand resources, or going forward – ODR, allow you to tag assets that you can fetch from Apple's servers (or your own) at a later time. This means they don't have to be included with your application's binary, thus reducing its download cost.

So, keep the assets, drop the binary bloat. Nice, right?

Consider if you had a game where you could fetch assets for later levels once the player was close to them, or other content like cooking videos for a cooking app – these are all things you can download as ODR assets. In fact, you can download entire asset catalogs as opposed to a singular file. If it isn't any executable Swift, Objective-C, C or C++ file, it can be used as an ODR asset.

Once you download ODR assets, they are available until the system purges them. As such, you may need to fetch them again. There is API available to affect how important, or long, assets are downloaded and subsequently stick around. Keep in mind, though, these assets are transient. If you need them permanently once they are retrieved, you'll need to save them yourself.

Regardless of how you use them, everything will run primarily through one class, and that's `NSBundleResourceRequest`. When you determine the time is right, you initialize an instance to fetch the ODR assets and use them. The process loosely looks like this, end-to-end:

1. You create "Tags" which represent ODR assets under Xcode's "Resource Tags" section within the project navigator.
2. Within Xcode's Attributes Inspector panel, you assign a tag (which is just a plain string) to an asset.
3. In code, you use `NSBundleResourceRequest` to fetch assets by their tag.

Tags

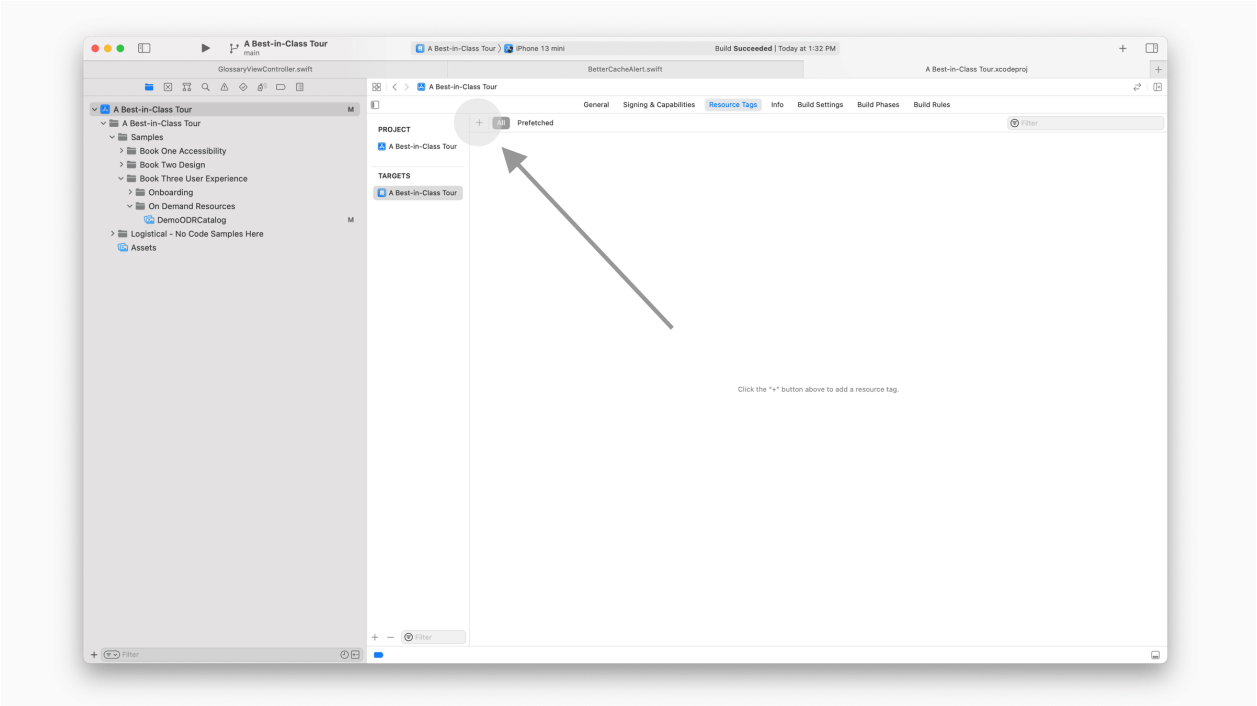
As mentioned above, tags are nothing more than simple strings you assign to assets you want to use with the ODR APIs. However, there are different tag types you can use choose for each of them which control how urgently they become available. Those tag types are:

1. **Initial Install:** These are resources that will be downloaded the same time as your app — they incur a cost in terms of download size. Why would use them? Check out the "Tips" section below.

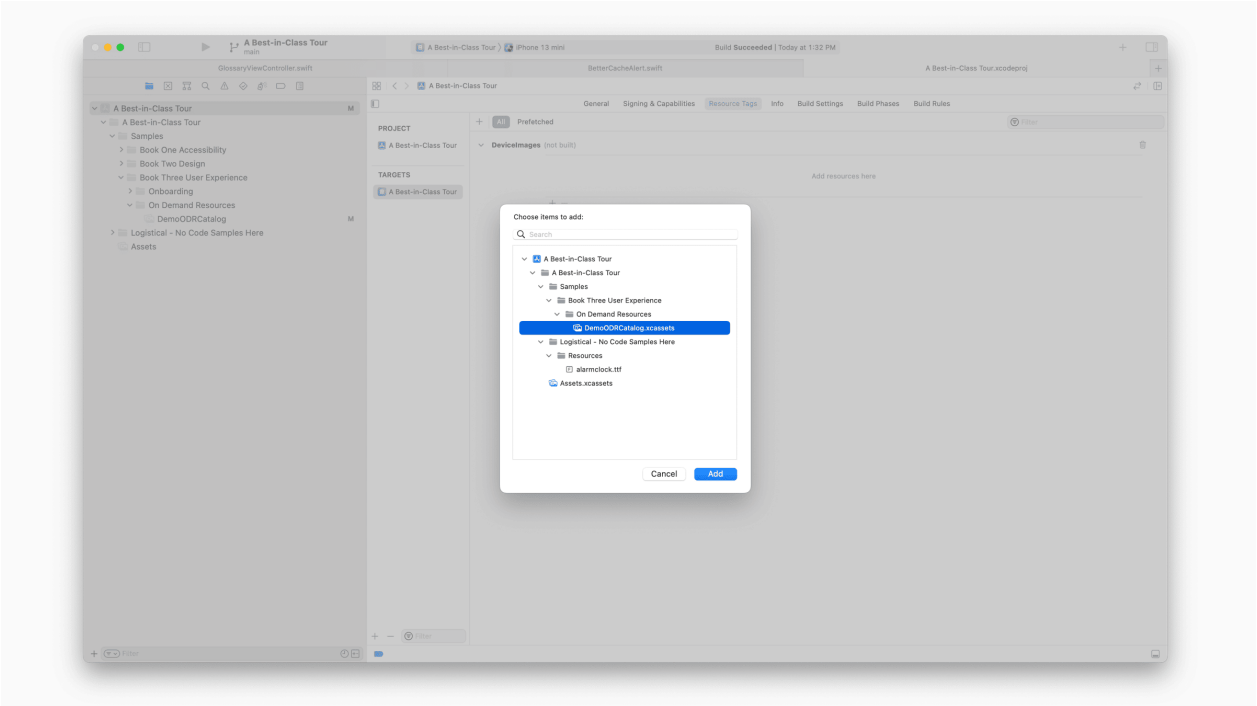
2. **Prefetch Tag Order:** These automatically begin to download once the app is installed, and in the order which you specify.
3. **Only On Demand:** These are what you probably think of when you consider ODR assets, meaning – these are available only when you fetch them manually.

Creating tags only takes a few steps in Xcode, here's what it looks like.

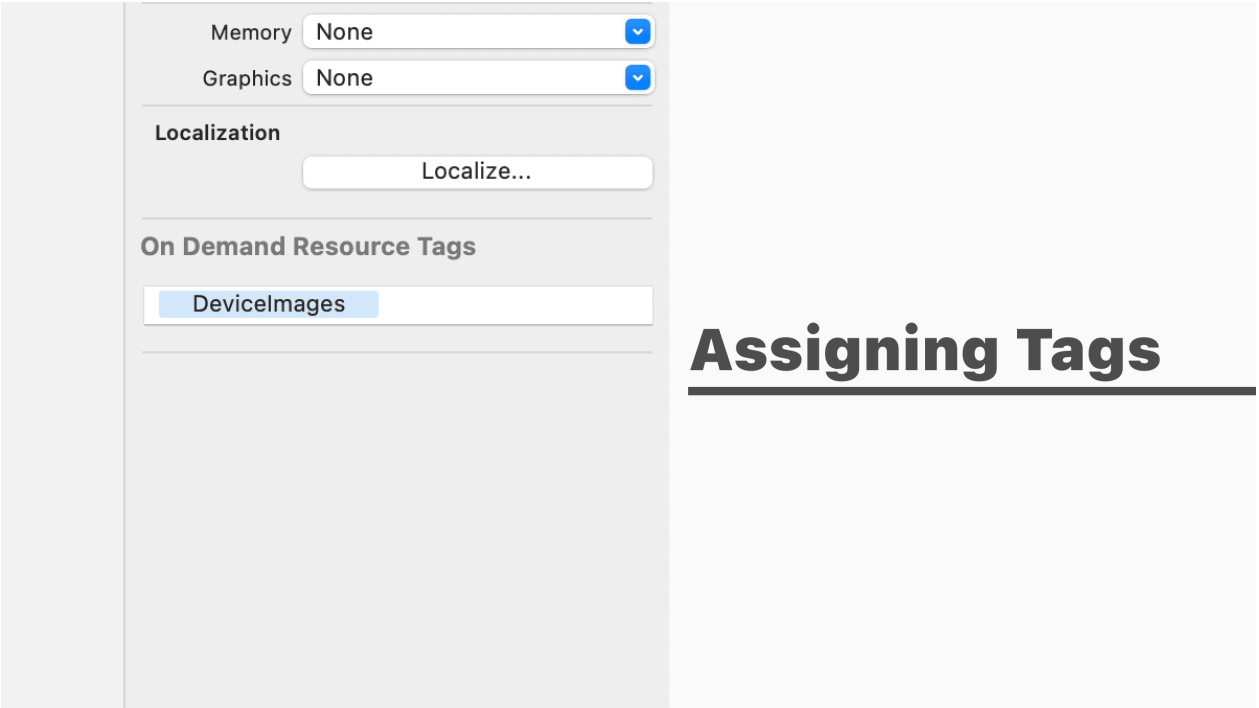
First, you visit the "Resource Tags" section under your project settings and click the "+" icon:



Next, you can assign assets directly from this view to associate with the tag:



And that's it – to confirm the tag was added, you can check out the asset's inspector pane and double check that the tag is present under the "On Demand Resource Tags" section. You can also set the tag this way too if you prefer:



Fetching Assets

Once you've got tags set in place, the implementation is nothing more complicated that invoking a few functions that `NSBundleResourceRequest` offers us. Remember, this object does it all when it comes to ODR assets. Using it, we can:

- Track download progress via `Progress` objects.
- Mutate download priorities.
- Check for low disk warnings.
- Let the system know if we no longer need the asset.
- And of course, there's API to request the asset itself.

In the demo project, I've included an asset catalog with a tag already assigned to it. You can find it under the "On Demand Resources" folder. Here's how we would access it using `NSBundleResourceRequest`:

```
// Xcode -> ODRAssetFetch.swift

struct ORDAssetFetch {
    enum AssetTag: String {
        case images = "DeviceImages"
    }

    func fetchAssetWith(tag assetTag:AssetTag) async throws ->
UIImage {
        let bundleResource = NSBundleResourceRequest(tags:
[assetTag.rawValue])
        let hasResourceLocally: Bool = await
bundleResource.conditionallyBeginAccessingResources()

        if hasResourceLocally {
            // Catalog available, access the image
            let image = imageFromCatalog()
            return image
        }

        do {
            bundleResource.loadingPriority =
NSBundleResourceRequestLoadingPriorityUrgent
            try await bundleResource.beginAccessingResources()
            bundleResource.endAccessingResources()
            return imageFromCatalog()
        }
    }

    private func imageFromCatalog() -> UIImage {
        return UIImage(named: "DemoImage") ?? UIImage()
    }
}
```

The whole flow will look like this:

1. Initialize the bundle resource object with the tags you're after.
2. Then, check if they are already present via `conditionallyBeginAccessingResources() -> Bool`.
3. If they are, you're done – access the asset just as if it existed locally.
4. If it's not there, request it.
5. Finally, end the request with `endAccessingResources()`, *if you needed to retrieve the asset from the network*.

If you're new to the API and you are designing a wrapper around it, it helps to think of `NSBundleResourceRequest` as nothing more than an informer object. It doesn't actually return the asset to you (via its API, anyways), it just tells you when it's there. Then, you access it as if the asset were included in your app all along. In practical terms – yes, it does retrieve it, but that part is more or less abstracted away from you.

To see things in a practical use, here's how you might use something like that in SwiftUI to fetch an image (there's also a UIKit sample in the sample project):

```
// Xcode -> SwiftUI -> ODRAssetFetchFig1View.swift

struct ODRAssetFetchFig1View: View {
    private let odrFetch: ORDAssetFetch = ORDAssetFetch()
    @State var image: UIImage? = nil

    var body: some View {
        VStack {
            if let img = image {
                Image(uiImage: img)
                    .scaledToFit()
                    .transition(.scale.combined(with: .opacity))
            } else {
                Text("Loading Asset...")
```

```
                    .transition(.scale.combined(with: .opacity))
            }
        }
        .onAppear {
            Task {
                self.image = try await
odrFetch.fetchAssetWith(tag: .images)
            }
        }
    }
}
```

In this case, though, this code isn't resilient to error handling. So if you're copy and pasting this into your own codebase, be sure to account for that.

Tips

Use Asset Catalogs

Asset Catalogs benefit from several optimizations, one of which is App Slicing. When your app is processed for the App Store, only the necessary assets for the particular device will be included. So if there's an image at @3x and the device doesn't support that resolution, it wouldn't be a part of the resulting download of your app.

So, if you have a set of images, media or whatever else that you're wanting to use with ODR – you don't have to setup multiple tags for them. Instead, put them all in one asset catalog, give that a tag – and you can request them all at once.

Automatically Prefetch Critical Assets

If you need a resource alongside your app's download, use the "Initial Install" tag prefetch type. This means the asset will be installed in lockstep with your app.

So, why would you do this?

Well, remember that ODR assets are temporary. If you have some assets that you want to use for something like onboarding, maybe this tag type would be great for that. Once onboarding is done, those assets could be purged safely – thus no longer counting against the space your app is taking up.

Be Wary of Limits

There are some hard limits iOS enforces for ODR resources. For the most part, you probably won't come to close any of these, but if you rely heavily on the API it's certainly possible. Those sizes are:

- Each tag can't be more than 512 MB after app slicing.
- And, the total size for stored ODR assets can't exceed 20 GB.

Making the Most of Bundle Requests

`NSBundleResourceRequest` has a few moving parts to it that can help customize how they function. One of the most important ones is the `loadingPriority` property. This is a double value from 0.0 up to 1.0. The default is set right in the middle – so 0.5.

If you feel an asset should have more precedence from the system than others, you can use this property to do that. In the sample code for this book, that's what we've done. For a refresher:

```
bundleResource.loadingPriority =
NSBundleResourceRequestLoadingPriorityUrgent // Maps to 1.0
```

Outside of that, the object also has a `progress` object you can use. This is great if you want to include a download indicator or progress bar. For example, in our sample code we could update the interface using a Combine subscriber:

```
let sub = bundleResource.progress.publisher(for:
\.fractionCompleted).sink { progress in
    // Udpate UI
}
```

Preservation Tags

When requesting tags, iOS uses preservation priorities to determine which tag assets to purge first, if it needs to. If everything has the same tag, this flag doesn't do anything. If there's something you need to stick around longer, you can give it special treatment apart from the others:

```
// 0 to 1
Bundle.main.setPreservationPriority(1.0, forTags: "MyTag")
```

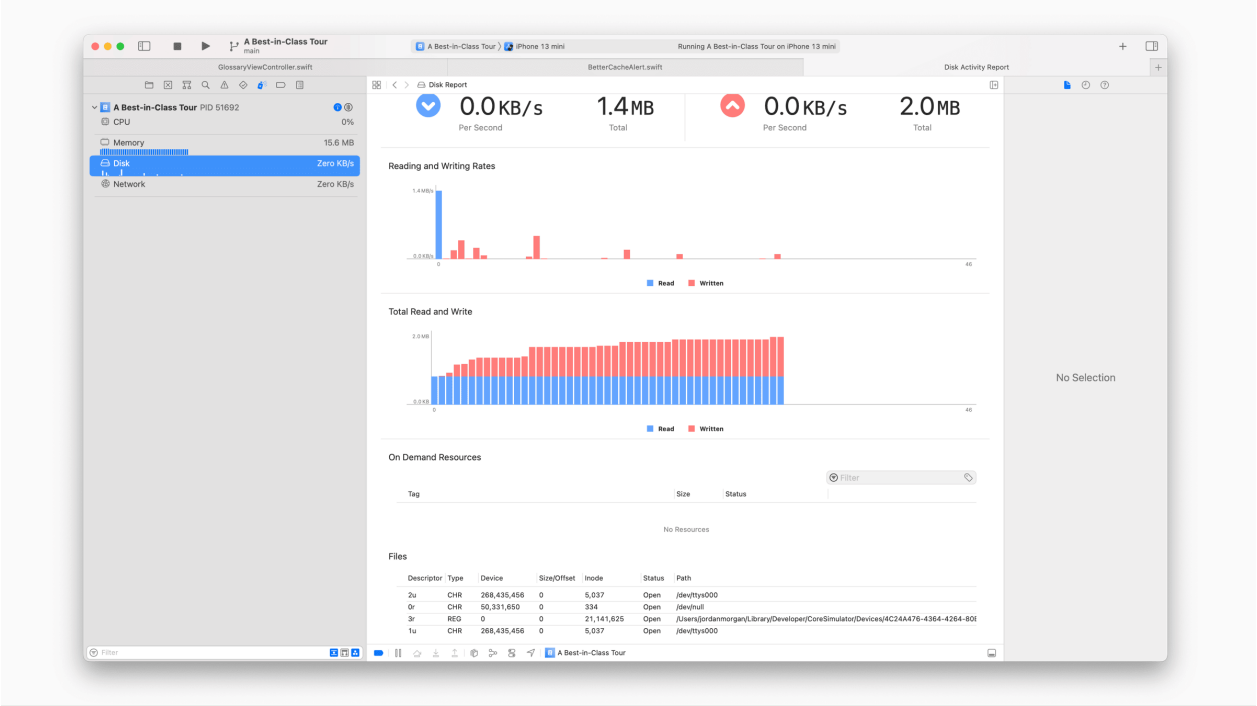
Not much to say here, the API works exactly as you'd expect – tags with a higher priority stick around longer than the ones with a lower priority.

Debugging On Demand Resource States

There are a few common errors that can occur when accessing ODR assets. Obviously, if you don't have a network connection, they aren't accessible. You'll get an error back from the bundle resource request letting you know as much. Aside from that, here are three other common ones to look out for:

1. `NSBundleOnDemandResourceExceededMaximumSizeError`: Your app is at or over the size limit allowed for ODR assets. Those limits were covered above, but if you hit them – this is the error you'll encounter.
2. `NSBundleOnDemandResourceInvalidTagError`: The tag doesn't exist, more than likely – this is your "typo" error. You probably misspelled the tag when constructing the request.
3. `NSBundleOnDemandResourceOutOfSpaceError`: This one simply means the user's device just doesn't enough space available to request the ODR asset.

If you need to debug the state of your ODR requests, Xcode also has a built in tool to help you do just that:



Three Key Takeaways

1. Evaluate if there are assets you can fetch at a later time in your app's usage timeline that aren't required to get started so keep your download size as small as possible.
2. Identify which of those you should download, but more importantly — when.
3. Be aware of on demand resource asset lifecycles to avoid relying on data that may not be there after getting purged by the system.

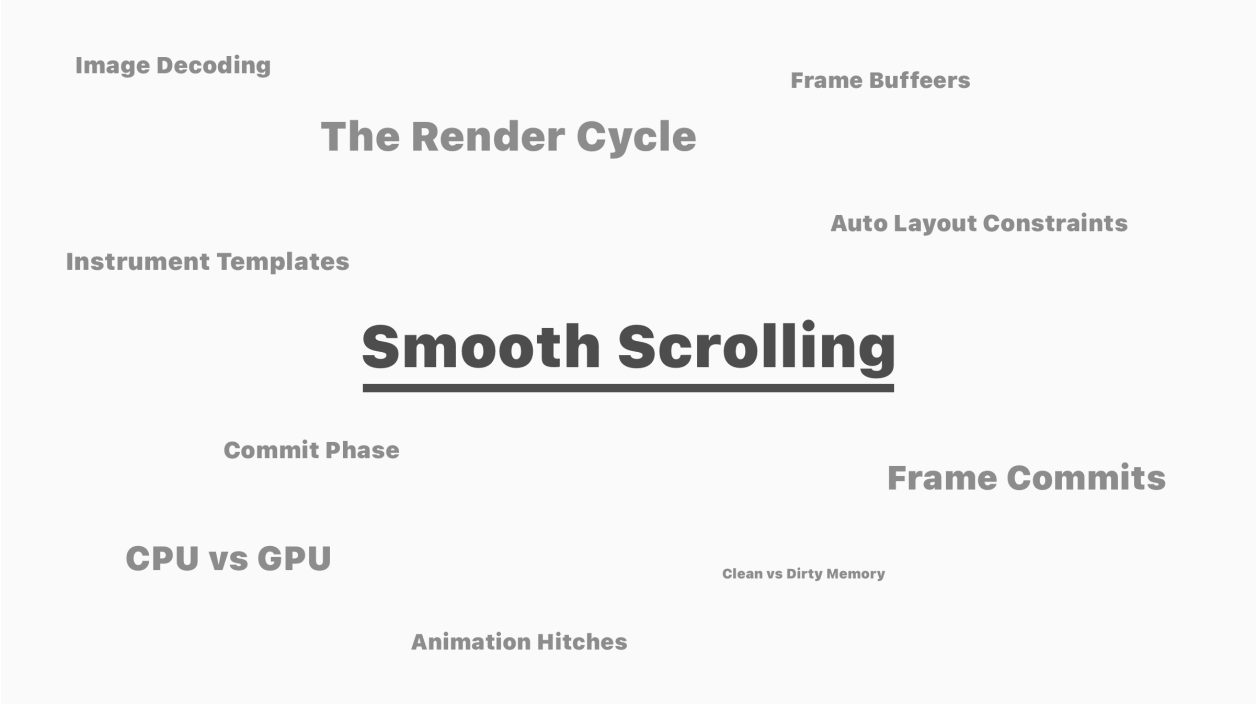
Scrolling Performance

Smooth scrolling is paramount to user experience on iOS. If your app doesn't scroll smoothly, or it jumps around and experiences hitches — even the most tech-illiterate person will notice. And, how could they not? It's jarring.

The problem for developers is that achieving a smooth scrolling experience really isn't that easy. If it was, every app would be able to do it. Is it achievable? Absolutely — but it can take a bit of know-how to get there. So, this chapter is all about getting there.

Here, we'll break down how to achieve a smooth scrolling experience at a high level, unpack why hitches occur and go over a simple, no-nonsense look into the render loop. We want to know why scrolling isn't always smooth, and these topics can help us form a vocabulary around that.

To be honest, though, and here's the frustrating part about all of this — choppy scrolling can happen for *all kinds* of reasons. When I sat down to brainstorm this chapter, I realized I could write an entire book on rendering, the intricacies of collection and table views, the quirks they have or how to tame misbehaving scroll views. The list goes on.



Smooth scrolling is the cause of several technologies on iOS.

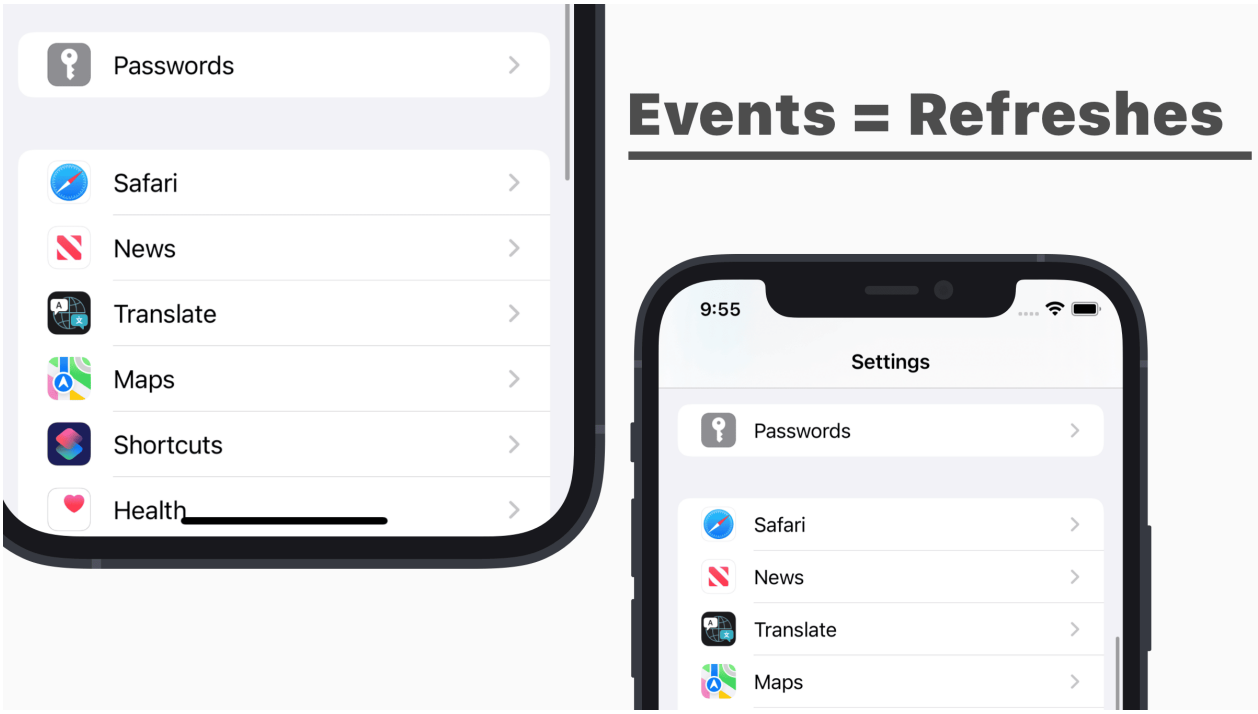
Oh, and that's not even considering SwiftUI — which can be an entirely different animal when it comes to smooth scrolling! It's just a gigantic, technical topic.

To best serve you, though, I thought about it like this: If I didn't know a whole lot about how rendering works on iOS and I wanted to be mindful about how to create a smooth scrolling experience, what kind of information would I want to know? What would help? I've written this chapter with those questions at the forefront of my mind.

With that, let's check out how the render loop works and discover where hitches typically come from. We'll then move on to some common pitfalls that can be a source of choppy scrolling on iOS.

How Rendering Works

At its core, your iPhone right now is performing a continuous loop. Over and over again, the system is reacting to all sorts of events that are happening and then it updates its interface accordingly. If you open the Settings app, and scroll down a bit – the screen repaints the display incredibly quick to create the illusion that we actually scrolled some content off screen:



Events, either by the user, external events or internal system events - will cause the display to need to refresh.

Tucked away in Core Foundation is the object powering it all, `CFRunLoop`. Its job is to monitor sources of input and in turn, help those resulting tasks get processed. In short, this is called the "render loop" – and smooth scrolling occurs when it can happen without delay and, in turn, it *doesn't* happen when it gets behind schedule. If you've done any game development, it's analogous to the game loop.

Take, for example, scrolling. The render loop is redrawing the pixels on the display to reflect the gestures that you're making. And, it wouldn't be convincing if it didn't do that fast. For most iPhones, it updates the display an impressive sixty times per *second* – which is when you often see the term "FPS" come up.

However, with modern iPhones and iPads – that number is doubled! Pro Motion displays push frames twice as fast. That means that you've got 1/120th of a second to push a frame so the render loop can keep things scrolling buttery smooth. That's es-

sentially 8.33 milliseconds to meet a deadline that the operating system will impose on you.

This all happens in three distinct phases. There are more phases that happen in-between these, but at a high level it looks like this:

1. **App Events**: You perform some action on your phone which results in the interface changing, whether that be scrolling, or a video is playing, etc. Or, some external event occurs (like a notification coming in) that would require the display to refresh in response.
2. **Render Server**: The render server does the work of creating, or rendering, what the user interface should look like.
3. **Display**: Everything is ready to go, and the frame is rendered and shown on device.

So, then, what we want to avoid are creating scenarios where a frame appears *later* than it was expected to – that's a common cause for a hitch. At that point, smooth scrolling can't occur because the render server may not have a frame ready to commit and we'll then miss the imposed deadline. We lost a frame, or more precisely – we didn't prepare the next one in time.

On top of all of this, Apple's user interface frameworks have another requirement – which is to run on the main thread. You can't safely create interface elements from SwiftUI or UIKit on a background thread, and if you do – you're certain to get a warning in the console.

So, here's where we are at: We know that we've got a deadline to meet to push frames to the render server, a particular thread we have to do it on and some devices will require that we do it twice as fast. It's a testament to the durability of Apple's APIs

that this is even possible to do, and if you stick to the tips below you should be well on your way.

Now, let's book up on things we can do to promote smooth scrolling.

Tips

Blocking the Main Queue

One of the primary ways smooth scrolling is hindered is by "blocking main", which means you were doing a lot of work on the main thread and it didn't have enough time or resources to update the interface because of it.

For things like loading in data over the network, file I/O or other "hands off" operations that don't require user input – consider offloading that work to a background thread:

```
// Instead of...
dataloader.load { data in
    // Operate on data
}

// Consider something like this...
DispatchQueue.global(qos: .default).async {
   dataloader.load { data in
        // Operate on data and then pop back
        // To the main queue
        DispatchQueue.global.main.async {

        }
    }
}
```

Remember, all of Apple's user interface frameworks are bound to the main queue – so if we offload work, that's great! But, we've got to be sure to update the interface on the main queue. I should also note, don't default to do doing this. Concurrency, threading and queues are technical in nature, and they can be easy to get wrong. So, I recommend using these techniques in *response* to performance bottlenecks – not preemptively starting this way.

When deciding how to offload that work, you can also take advantage of Grand Central Dispatch's Quality of Service APIs. Using it, you can clue in the system about the nature of your work so that it can intelligently schedule it. Here is what you can choose from:

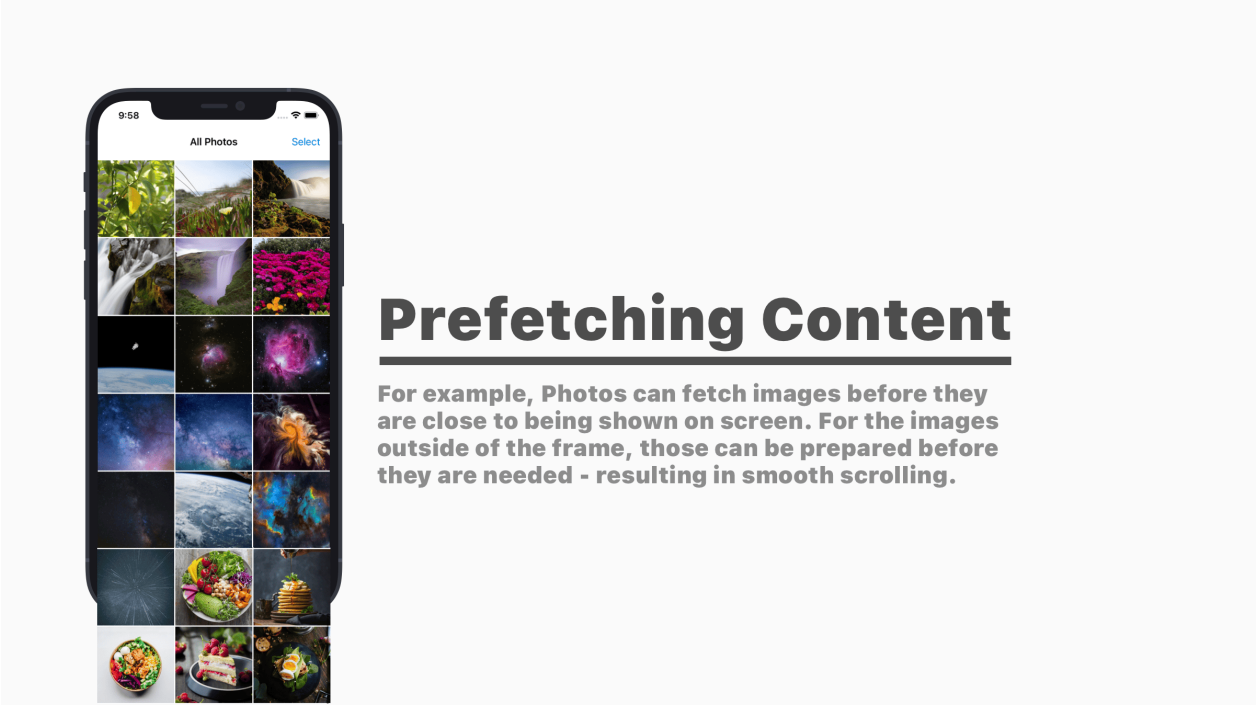
1. `.userInteractive`: These have the highest priority by the system, it deals with work that is actively working towards updating the interface.
2. `.userInitiated`: Much like above, but the work may not be dealing directly with the interface being updated – yet it still is mission critical work that your app needs to complete.
3. `.default`: This is the baseline value – it represents generalized work.
4. `.utility`: This work may take some time to finish up, but the user isn't necessarily aware of it occurring and thus, doesn't really track when it's started or finished.
5. `.background`: Anything that deals with clean up tasks or similar work which could be run just fine in the background.

In the example above, we chose `.default`, but you can leverage each one as you see fit.

Leverage Collection and Table Views APIs

Collection views and table views offer several APIs to help you make an incredibly fast scrolling experience.

First, there's a prefetch API, which allow you to fetch assets, or models, to inject into your views. Basically, this enables you to fetch content before it's displayed on screen:



Even better, when you can do this off the main queue – you can ensure you're not blocking main as we chatted about directly above, for anything that UIKit might need. For collection and table views, the API is strikingly similar, as they both are powered though protocols. For example, to use `UICollectionViewDataSourcePrefetching`, you'd set an object confirming to that to a collection view's `prefetchDataSource` property and implement the prefetch method:

```
func collectionView(_ collectionView: UICollectionView,
prefetchItemsAt indexPaths:
[IndexPath]) {
    // Async data fetch
    indexPaths.forEach {
        let imageURL = data[indexPath.item]
        someAPIObject.downloadImage(with: imageURL)
        // Persist in a cache to later use for the cell
    }
}
```

Further, you can also cancel these calls too:

```
func collectionView(_ collectionView: UICollectionView,
cancelPrefetchingForItemsAt indexPaths: [IndexPath])
```

Next, be aware of the cell lifecycle. These controls are so efficient at scrolling a large set of data because of the way they can reuse their views. Cells are not allocated and deallocated over and over again – think about how much resources that would swallow up – instead, they go into the reuse pool (*most* of the time, prefetching APIs can tweak this timeline a little due to the user suddenly changing scroll directions, for example).

When a cell goes offscreen, it sets itself up to be used again – just with the relevant data that should be showing on screen for its given index. You can tweak what happens and where, namely – either during the preparation phase or the display phase.

In the preparation phase, the cell is going to be handed your data model (regardless of how you perform this, either with delegates or configuration handlers) and the cell will attempt to size itself.

In the display phase, the content becomes visible and is now onscreen. Your work needs to be done at this point, which is what makes prefetching APIs so valuable. You

can make sure things are tidy and ready to go *far* before the cell is ever displayed. That's is a big difference in how both of the controls work now – they didn't used to prepare themselves so far in advance in earlier versions of iOS. The fact they they do now makes achieving smooth scrolling with them much easier.

Collection and table views give you fine grain control over showing a list or grid of data. If you find yourself unable to hit the smooth scrolling you are after in SwiftUI, you should consider switching to one of those controls and wrapping it.

The reason why is, with their imperative nature, you can fully tweak every phase of the process. From cell reuse, to having a say in how the interface is measured and created (i.e. manual layout, Auto Layout, etc) – there's simply no matching the amount of tricks you can pull to produce smooth scrolling.

Lazy VStack and HStack

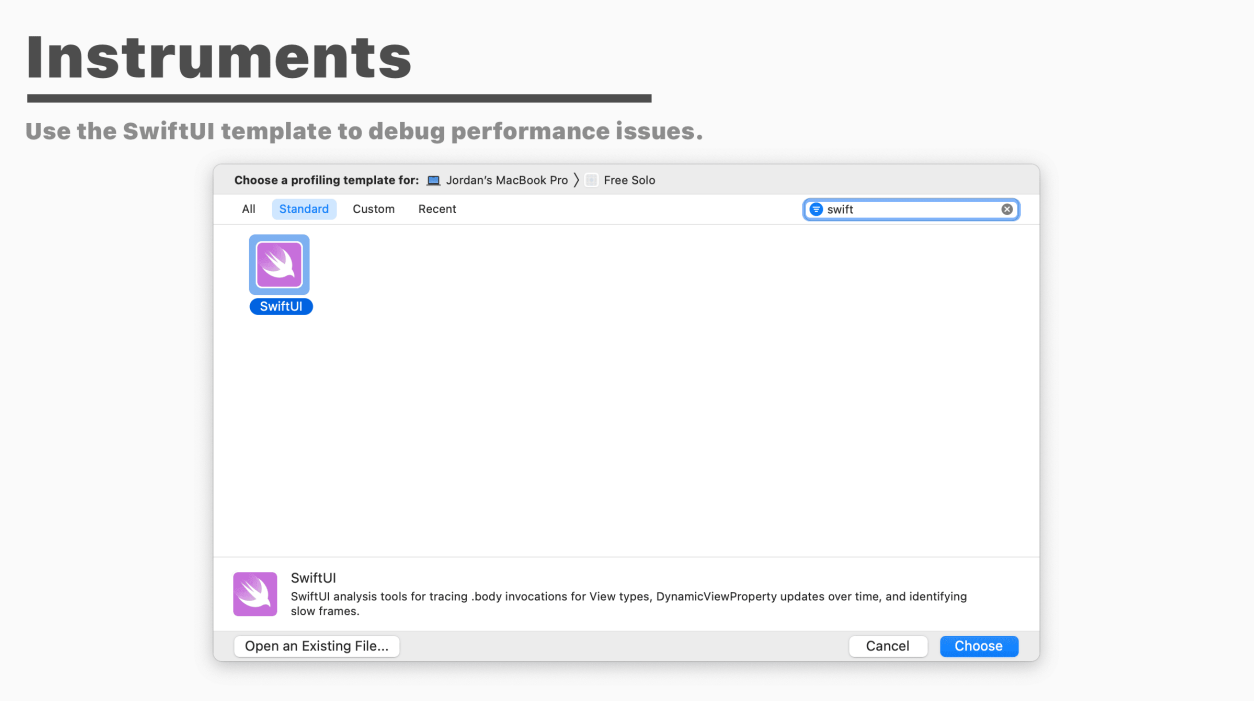
Speaking of SwiftUI, it has become more powerful in terms of smooth scrolling with the introduction of its lazy stacks. Beneath the surface, lazy stacks only create views when it needs them.

The beauty of it all? It's trivial to use and replace existing stacks in your code base:

```
// From this
ScrollView {
    VStack {
        ForEach(model, id: \.identifier) { model in
            SomeView(data: model)
        }
    }
}

// To this
ScrollView {
    LazyVStack {
```

```
            ForEach(model, id: \.identifier) { model in
                SomeView(data: model)
            }
        }
}
```

If you find that your scrolling isn't where you want it to be with SwiftUI – this should be your first stop. Then, optimize from there by attempting to reduce view updates and digging in further with the SwiftUI templates in Instruments:



That template can help you nail own where `body` invocations are coming from, highlight slow frames and tracks updates over time. Along with lazy stacks, it's a great way to dial in performance.

Avoid Expensive Computed Properties in SwiftUI

Expanding a bit on SwiftUI – avoid computed properties in your models that could result in your `body` being unnecessarily reevaluated or performing heavy logic that isn't needed within the property itself. For example:

```
class BlogPost: Identifiable {
    let id: UUID
    let url: URL
    var hasImages: Bool {
        return images.isEmpty == false
    }
    var images: [Data] {
        // Download and decode images
    }
}
```

If we conditionally set some image view to show based on if the images were present or not from the model above, we could fire off a ton of logic in the computed variable here, `images`:

```
@StateObject private var post = BlogPost(id: UUID(),
                                         url: url)

var body: some View {
    VStack {
        // This is sure to slow things down
        if post.hasImages {
/* Show images */
        } else {
            Text("This post has no images.")
        }
    }
}
```

As we'll see later, image decoding can be a costly process. Doing it hundreds of times via a computed property for a lot of data you have scrolling in a list is a surefire way to crater performance. In these situations, rethink the problem and see if you can refactor your architecture to avoid this.

I should note, however, the idea of avoiding costly computed properties certainly extends beyond just SwiftUI — I typically avoid them altogether because somewhere down the line, I or another coworker is bitten by them.

Text Measurement

Measuring text isn't as straightforward as it may seem. Think about the difference between this simple greeting:

"Hello, my name is Jordan!"

The Latin character set is, broadly speaking, fairly straight forward and not all too dense. Compare that to writing down the same phrase in Korean:

"안녕하세요, 제 이름은 조던입니다!"

Perhaps Tajik:

"Салом, номи ман Ҷордан аст!"

Or even traditional Chinese:

"你好，我的名字是喬丹！"

Now, expand that out to an entire paragraph, or book! Text measures itself in a hierarchical manner, so if you include a lot of it tacked right into a scroll view, you could experience dropped frames and hitches quickly. If you're building an eBook reader for example, you can't throw the entire thing into a text view and call it a day. The memory allocations and layout hit would be massive. You'll want to sensibly separate text as much as your interface will allow.

This advice only applies to UIKit – but when you're creating lists where performance becomes an issue, you may want to consider your choices when it comes to laying out your content. Those are Auto Layout and manual layout. Both have pros and cons.

But! And this is a big "but" to understand, Auto Layout is generally slower than manual layout. On the other hand – it's also faster than it's ever been, with iOS 12 greatly improving its algorithm speeds. They now achieve O(n) instead of $O(n^2)$ for multiple common layout scenarios, and it's sure to improve every single year. So, I typically advise you *start* with Auto Layout and only consider manual layout if you can use Instruments and prove that Auto Layout's solver is taking up a noticeable amount of time.

With that caveat out of the way, specifying your frames manually is a wonderful way to improve scrolling:

```
let lable = UILabel(frame. zero)
label.frame = /* your rect */
```

Manual layout is simply faster than setting up constraints for Auto Layout to sort out. However, you've got to give something to get something, and what you're giving with manual layout is the flexibility that Auto Layout can give you when responding to all of the various sizes that your app is sure to encounter. You need to plan on overriding `viewWillLayoutSubviews` or similar methods to calculate positions and sizes in response.

Some of those things can be mitigated with auto resize masks. For example, this would simply fill up the screen no matter what happens:

```
let fillView = UIView(frame: .zero)
view.addSubview(fillView)
fillView.frame = view.bounds
```

```
// Expand its width and height
fillView.autoresizingMask = [.flexibleWidth, .flexibleHeight]
```

Further, in scrolling scenarios common to table and collection views, you can manually calculate the height of cells and cache them to use later. There are many APIs to measure these sort of things, and in fact – I used this technique in my previous app to achieve fast scrolling when the size of each cell was dynamic due to user input – but the width was constant.

For example, if you know the width available to you (which is the case with some collection view layouts and any table view one) you can calculate the height of, say, a label based off its given font and string:

```
func boundingRect(with size: CGSize, options: NSStringDrawingOptions
= [], attributes: [NSAttributedString.Key : Any]? = nil, context:
NSStringDrawingContext?) -> CGRect
```

So if you've got the text, you can get really close to its resulting size:

```
let lbl = UILabel(frame: .zero)
lbl.text = "I need the height of this text. No matter if Dynamic
Type is small, large or anything else."

let textAtts: [NSAttributedString.Key : Any] = [.font: lbl.font]
let size = lbl.text!.boundingRect(with: CGSize(width:
collectionView.bounds.size.width, height:
CGFloat.greatestFiniteMagnitude), options: .usesLineFragmentOrigin,
attributes: textAtts, context: nil)

// size.height gives you the label's resulting height.
```

Using techniques like this, you can coalesce all of the information you've got about the controls together to use for delegate methods to return a specific size for a row

and cache it. This is great, because calculating a cell's height can often be a source of choppiness in scrolling:

```
// For a tableview
class TableViewDelegate: NSObject, UITableViewDelegate {
    func tableView(_ tableView: UITableView, heightForRowAt
indexPath: IndexPath) -> CGFloat {
        guard let cachedHeight = cachedHeight[indexPath.row] {
            return cachedHeight
        } else {
            let height = self.calculatedHeightsFor(indexPath.row)
            cachedHeight[indexPath.row] = height
            return height
        }
    }
}
```

You can apply the same techniques to any of `UICollectionView`'s layout APIs too, such as creating a flow layout or using `UICollectionViewCompositionalLayout`. The idea is the same – if you can provide the height, you're likely to make things smoother.

I want to stress once more – don't *start* here, but don't be afraid to use these types of methods if you find that Auto Layout isn't getting you the performance that you're after.

Background Choices

If you can make a view opaque, that is – a view without transparency, then do it. An immutable axiom in software development is to not speak in absolutes – but saying that an opaque view is *always* more efficient than one that *isn't* rings true in my experience.

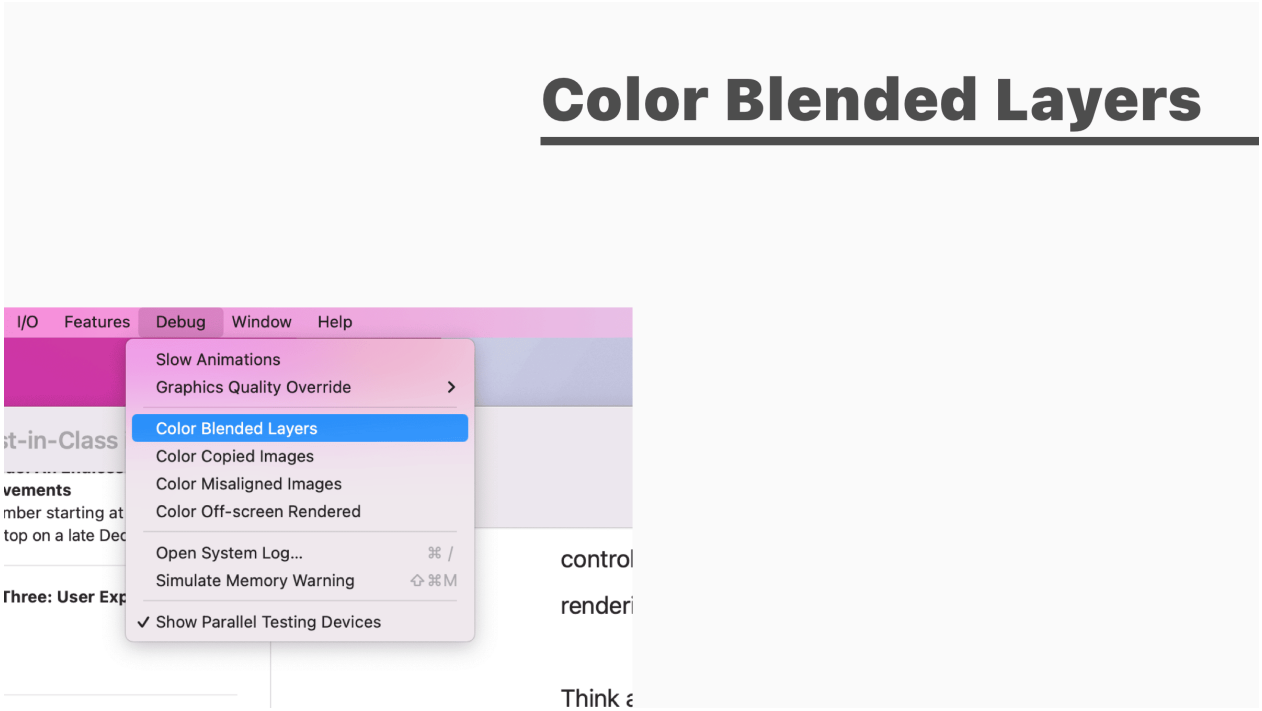
If you've got cells in a list, it's tempting to make their background color clear to match its containing control's color too:

```
cell.contentView.backgroundColor = .clear
```

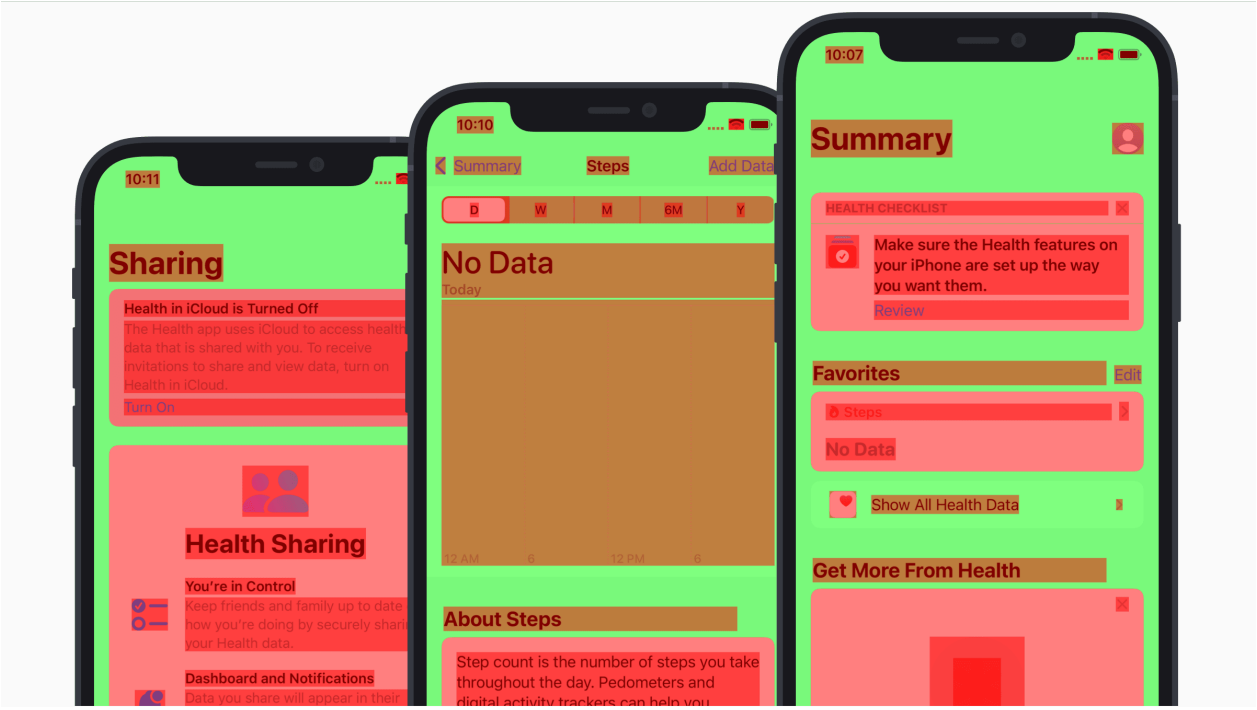
But this is tasking performance wise. Instead, set the color that the containing control's background is and that will optimize drawing performance during the rendering phase.

Think about it this way: if a view is transparent, then iOS has to figure out what the resulting transparency should look like underneath the content every time a frame is rendered. If it instead knows that the view is just white, black or whatever the color is – it can forgo that process. It doesn't have to blend the multiple layers behind it.

The iOS simulator has a great feature to help with this process. Go to Debug -> Color Blended Layers:



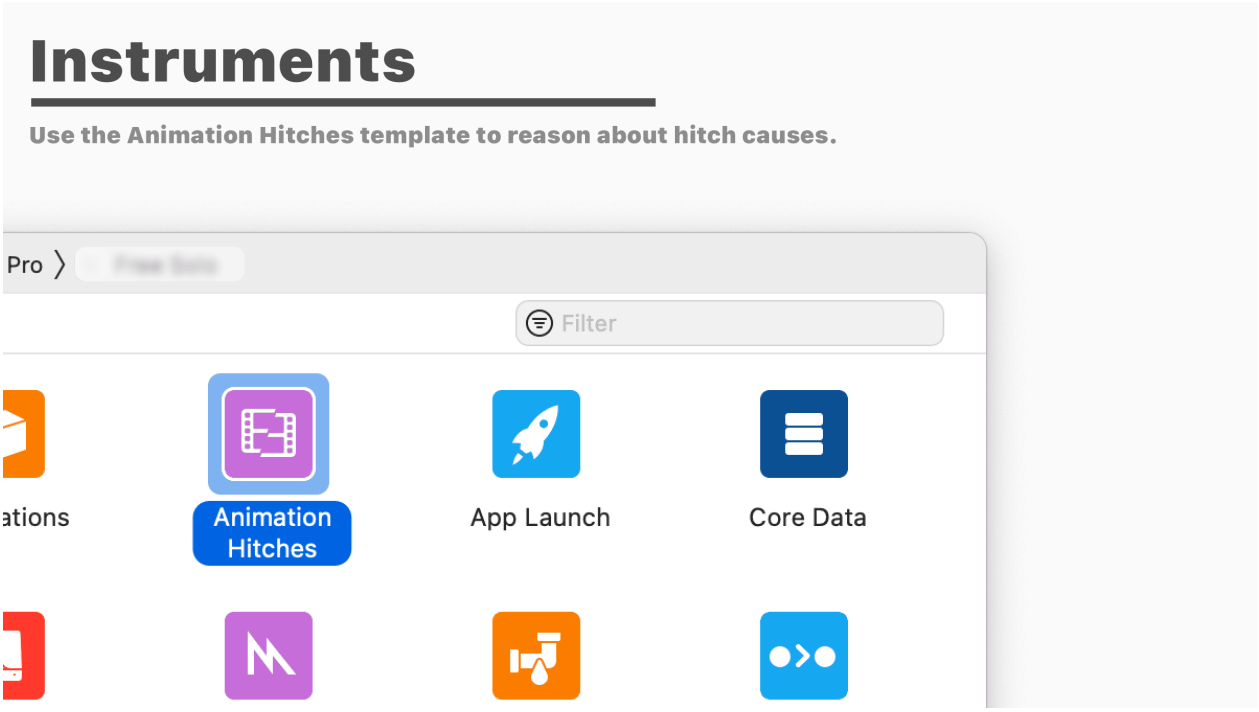
Then, you can see which views are blended which will be marked with a red box, and which aren't – those are marked as green. Here, it's marking those layers in Apple's Health app:



You want to go for, obviously, as much green as you can here.

Animation Hitch Instrument

In Instruments, there is an “Animation Hitches” template:



Using it, you can get an in-depth look at the render loop phases which, as we learned above, compose the resulting frames. There is a track specifically for hitches, and it'll show their duration, the code that was running that led up to it and also the type of hitch that happened (i.e. was it in the commit phase? GPU?).

Like many Instrument templates, it helps to select a particular part of the graph and dig into the call tree to further investigate where the issue might be.

Image Decoding

If you have images that is part of the data that you're scrolling, you can run into hitches fast. The problems start innocent enough, you have an image and you put them into an image view:

```
let image = UIImage(named: "MyImage")
imageView.image = image
```

Here's the thing though - let's say that your image view is only 200 points in width and height. The image you assign to it? Perhaps it might be 1718 x 2048 pixels. How much memory do you think that would require to use?

It's a small image view. The image file itself might be something like 266 kilobytes, so this should be fine — right?

In many cases - it isn't. The system, through a process known as image decoding, might allocate about *14 megabytes* for such an image. Why? Because the size doesn't come from the image view you're putting it in, or the size of the file itself. It's derived from the dimensions of the image asset.

So, assuming each pixel will cost us four bytes:

```
1718 * 2048 * 4 / 1024 / 1024 = 13.42 megabytes give or take
```

We arrive at about 13.42 megabytes. Now, scale that across hundreds of images that you scroll in a list, and you can see where problems might start to add up, and quickly! So, it helps to know about the rendering pipeline.

For images, there are three phases you should know about:

1. **Load**: iOS takes the compressed image and loads (in our example) the 266 kilobyte into memory. Really, no worries yet.
2. **Decode**: Now, iOS takes the image and converts into a way the GPU can read and understand. It's now uncompressed, and it's here we're at the 14MB size listed above.
3. **Render**: Just like it sounds, the image data is ready and willing to be rendered any which way. Even if it's just by a 200 by 200 point image view.

So, the decoding phase is where things can go south. That's because iOS has an in-memory representation of the image, which is why it stands to reason that the size is

intrinsically tied to the proportions of the image itself and has nothing to do with its actual file size on disk. Dimensions *really* matter when it comes to decoding.

Further – since the process is so intensive, iOS will hang on to that image buffer as well. And why not? Things would be much worse if iOS performed decoding over and over again. This is another reason memory can start to balloon, and quickly.

So, what can we do about all of this?

Basically, what we want to do is create a smaller image buffer – and we can do that with `UIImage` or Core Image APIs. Ideally, the image buffer should just match the size of the image view we are placing it in, that way we can avoid a lot of overhead with image decoding. You can do this by providing a thumbnail using a function off of `UIImage`:

```
let image = UIImage(named: "MyImage")?.preparingThumbnail(of:
CGSize(width: 200, height: 200)
```

Or, if you want to get more granular with options, you can use Core Image APIs directly:

```
func downsampleImage(at URL:NSURL, maxSize:Float) -> UIImage {
    let sourceOptions = [kCGImageSourceShouldCache:false] as
CFDictionary
    let source = CGImageSourceCreateWithURL(URL as CFURL,
sourceOptions)!
    let downsampleOptions =
[kCGImageSourceCreateThumbnailFromImageAlways:true,

kCGImageSourceThumbnailMaxPixelSize:maxSize

kCGImageSourceShouldCacheImmediately:true,

kCGImageSourceCreateThumbnailWithTransform:true,
                            ] as CFDictionary
```

```
    let downsampledImage =
CGImageSourceCreateThumbnailAtIndex(source, 0, downsampleOptions)!

    return UIImage(cgImage: downsampledImage)
}
```

Image decoding can take a toll on memory and scrolling. For images that you have included from a bundle or download over the network, you may need to apply extra caution on how those are displayed during the decode phase. Images in asset catalogs are generally optimized for you, so those become less of an issue. However, for all of these reasons and more, it pays to cache images much the same way we cached heights earlier on in our examples.

Three Key Takeaways

1. Smooth scrolling on iOS might be the most critical piece of providing a great user experience on iOS.
2. Getting it right can be tough, though, due to all of the ways it can be negatively impacted.
3. Try to remember that if you have a good understanding of what to *avoid*, Apple's APIs are extremely capable in helping you achieve smooth scrolling.

Quick Look Preview

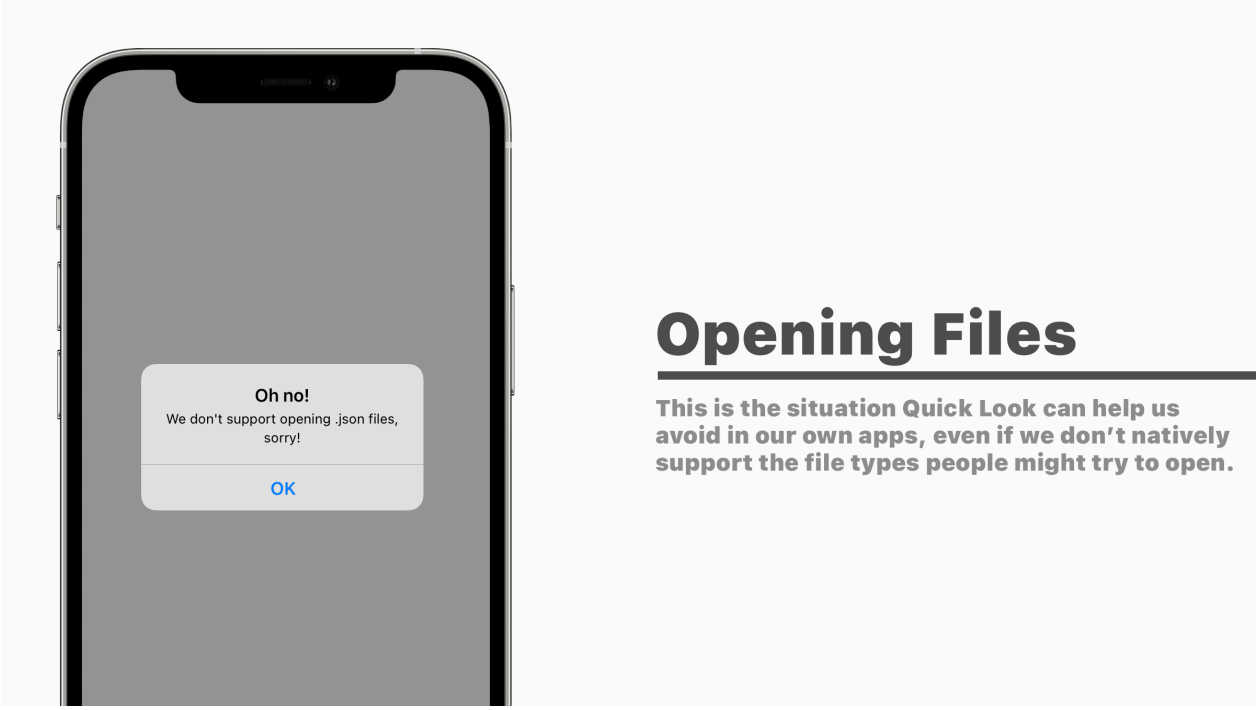
Software running on iOS started its life rather simply. Apps were created exclusively in-house by Apple, engineers knew what to expect since they were the only stewards of every single app running on iOS (then iPhoneOS) and the gatekeeping (in a good way) of sandboxing made sure external processes were kept in check.

In short, there were a lot of known quantities. And that's largely due to the fact that there was no App Store. Obviously, times have changed — drastically. To the tune of several million apps.

The question then becomes, how do millions of apps built for a million different purposes which operate on a million different file types interact with each other? Part of the answer to that question was presented with iOS 4 and the Quick Look framework.

At its core, Quick Look is there to present, open and potentially edit files of all types. And, that's a great experience, isn't it? If your app presents a file of a certain type, and a person taps on it and….

…nothing happens, or arguably worse – an error shows, that's not great, right?



Quick Look can expand your app's capabilities to view just about any file iOS supports, even custom ones from third party apps.

That core idea is why I've opted to chat about Quick Look APIs in a book about user experience. When used correctly, Quick Look provides a phenomenal "it just works" aspect to your app that Apple is known for.

Quick Look can help you in three main ways

1. Primarily, it can open several file types that your app may not primarily support, like a Key Note file or something in a custom file format.

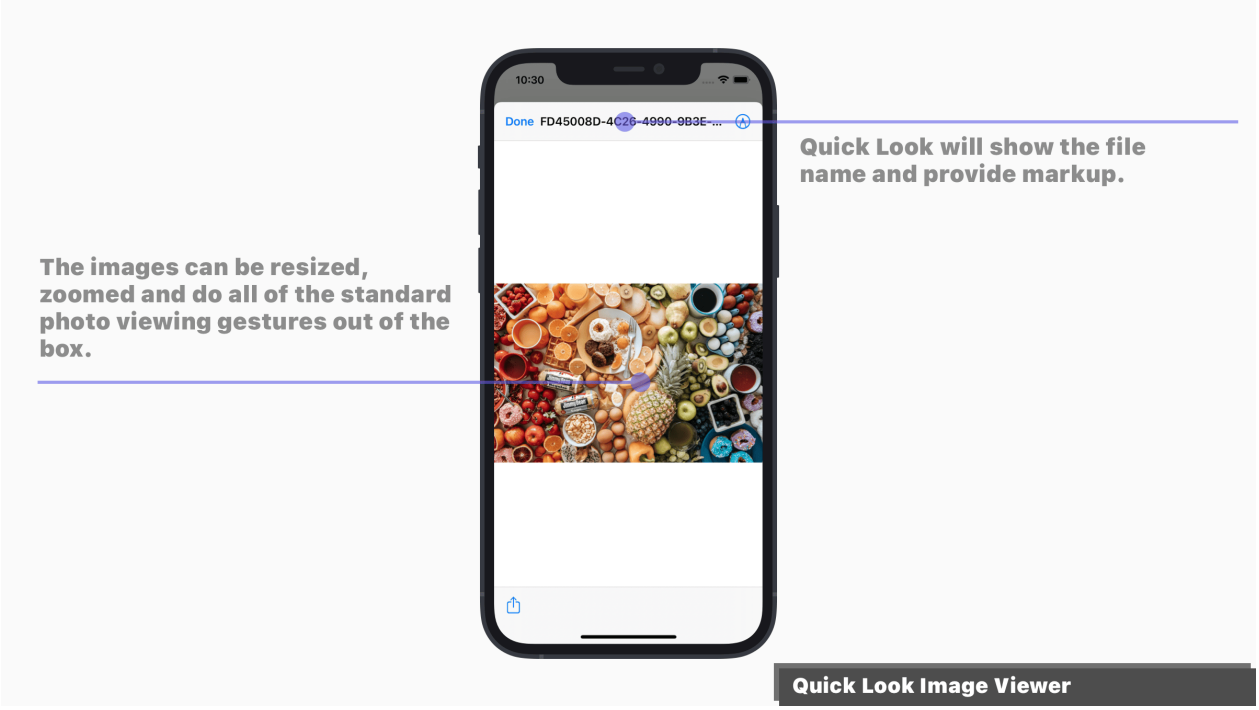
2. Secondly, you can use it to make *your own* custom file format work with Quick Look previewing.
3. Finally, you can also use it to create a custom thumbnail view for your custom files.

In this chapter, we're going to focus on the primary use case of Quick Look – which is that first point above, previewing data. Let's dig in.

Previewing Data

Your primary use of Quick Look is going to be simply this: You have a file, and you don't support opening it natively in your own app. So, you'll invoke the Quick Look framework APIs to do that for you.

Those files can range from basic file types, like images:



To something more custom to a specific app, like viewing a Numbers file or even an augmented reality asset using the USDZ format. To do this, there is one primary class to consider and one protocol to implement (at a minimum).

The Preview Controller

To present this data, we'll turn to `QLPreviewController`. It works like any `UIViewController` in UIKit, in that you instantiate it and present it:

```
let quickLookController = QLPreviewController()
present(quickLookController, animated: true)

// Or, for navigation controllers...
navigationController?.push(quickLookController, animated: true)
```

With SwiftUI, it's implemented as a presentation modifier:

```
struct QuickLookView: View {
    @State private var url: URL? = nil

    var body: some View {
        Button("Street Corner") {
            url = createURL()
        }
        .buttonStyle(.borderedProminent)
        .quickLookPreview($url)
    }
}
```

However, you need to supply it with data (as we did above for the SwiftUI example), and we can do that by performing two tasks:

1. Adopting a protocol to supply data.
2. And, wrapping that data up as a `URL` in some form.

No matter if you're using UIKit or SwiftUI, Quick Look works the same way under the hood – it'll evaluate each file URL's `UTType` to figure out what it is displaying, and ultimately – *how* to display it. For us, that means we need to know about `QLPreviewControllerDataSource` if we're using UIKit, and `QLPreviewItem` for both SwiftUI and UIKit.

Consider a view where we've got a few files that represent photos. If we wanted to display the images so people could pinch to zoom, edit them and more – we could use the preview controller and provide the data for that like this:

```
// Xcode -> UIKit -> QuickLookPreviewFig1ViewController.swift

extension QuickLookPreviewFig1ViewController:
QLPreviewControllerDataSource {
    func numberOfPreviewItems(in controller: QLPreviewController) ->
Int {
        return datasource.snapshot().numberOfItems
    }

    func previewController(_ controller: QLPreviewController,
previewItemAt index: Int) -> QLPreviewItem {
        let snap = datasource.snapshot()
        let data = snap.itemIdentifiers[index].generateURLData()
        return data.url
    }
}
```

In UIKit, you can see that there are two primary delegate functions to consider. The first lets the preview controller know how many items its displaying, while the second provides that data via `QLPreviewItem` instances.

You might've noticed that for `QLPreviewItem`, we just supplied plain `NSURL` instances. That's because `NSURL` adopts the `QLPreviewItem` protocol for us out of the

box, making them ideal to use with the framework. In SwiftUI, the modifier asks for a `URL` type, and it'll handle `QLPreviewItem` responsibilities for you.

I should note that the preview controller also has a delegate protocol you can use, and I highly recommend that you do use it. With it, you can perform a custom view controller transition that animates the view your previewing (if available) by only using one function.

For example, if we wanted to animate the image views from the sample above, you could simply specify that the image view itself is the source of the preview, and the framework will take care of the rest. This results in the beautiful transition you see in stock apps like Photos, or tapping an image to view it in Messages:

```
// Xcode -> UIKit -> QuickLookPreviewFig1ViewController.swift

extension QuickLookPreviewFig1ViewController:
QLPreviewControllerDelegate {
    func previewController(_ controller: QLPreviewController,
transitionViewFor item: QLPreviewItem) -> UIView? {
        guard let cell = collectionView.cellForItem(at:
IndexPath(item: tappedIndex, section: 0)) as? CustomListCell else
{ return nil }
        return cell.imageView
    }
}
```

Like many of Apple's delegate protocols, you can also optionally get called back for things like when the controller dismisses, presents and even editing the preview items in-place that you display.

SwiftUI and Quick Look Delegates

Of course, we can wire this up in SwiftUI too via using representable APIs. While the delegate methods are not exposed via its own `View` type or a modifier, setting it up is simple enough:

```
// Xcode -> SwiftUI -> WrappedPreviewView.swift

struct WrappedPreviewView: UIViewControllerRepresentable {
    typealias UIViewControllerType = QLPreviewController
    let fileLocation: URL

    // MARK: Initializer

    init(withFile file: URL) {
        self.fileLocation = file
    }

    // MARK: Representable

    func makeUIViewController(context: Context) ->
QLPreviewController {
        let controller = QLPreviewController()
        controller.dataSource = context.coordinator
        controller.delegate = context.coordinator
        return controller
    }

    func makeCoordinator() -> Coordinator {
        Coordinator(parent: self)
    }

    // MARK: Cooridnator

    class Coordinator: NSObject, QLPreviewControllerDataSource,
QLPreviewControllerDelegate {
        var parent: WrappedPreviewView
```

```
        init(parent: WrappedPreviewView) {
            self.parent = parent
        }

        func numberOfPreviewItems(in controller:
QLPreviewController) -> Int {
            return 1
        }

        func previewController(_ controller: QLPreviewController,
previewItemAt index: Int) -> QLPreviewItem {
            return parent.fileLocation as NSURL
        }

        // Implement any desired delegate methods here
    }
}
```

And from there, so long as we can get at the file we're after, we can simply present it like any other `View`:

```
// Xcode -> SwiftUI -> QuickLookPreviewFig1View.swift

struct TapImage: View {
    let item: LocalImage
    @State private var presentQuickLook: Bool = false

    var body: some View {
        Image(uiImage: UIImage(contentsOfFile:
item.generateURLData().filePath) ?? UIImage())
            .resizable()
            .scaledToFit()
            .opacity(presentQuickLook ? 0 : 1)
            .onTapGesture {
                presentQuickLook.toggle()
            }
            .sheet(isPresented: $presentQuickLook) {
                let url = item.generateURLData().url
```

```
                WrappedPreviewView(withFile: url as URL)
            }
        }
    }
```

Tips

Consider Quick Look's Image Viewer

I alluded to this above, but it bears repeating. A lot of times, developers (myself included) tend to build their own image viewer of sorts. It can be a lot of work to support pinching gestures for zooming, double tapping the image to change scales and more.

These interactions are all expected, though. Plus, if you add in the zooming view controller transition that nearly all photo viewers support – you're looking at quite a bit of work.

Unless you have specific requirements, just use Quick Look for this. Its image viewing experience supports every single one of these things, plus a lot more that you get for free.

Remember Quick Look's Purpose (And Limits)

Though Quick Look is extremely capable, consider its limits and what it was built for. Namely, lightweight viewing and editing experiences. If you need to do something a little more advanced, like add new views on top of the preview or perhaps you want to provide playback controls for videos – you'll want to instead opt for those system frameworks and APIs built for such purposes.

For example, in the case of videos – that'd be AVPlayer instead of using Quick Look.

Return Titles for Assets

By default, the preview controller will use the `QLPreviewItem`'s URL as the title. If you want to change that, you'll need to conform to the protocol yourself and then you can set the title to something a little more readable:

```
class PreviewingImage: NSObject, QLPreviewItem {
    var previewItemURL: URL?
    var previewItemTitle: String?
}

// Then, return this in your datasource for the preview controller
let preview = PreviewingImage()
preview.previewItemURL = myURL
preview.previewItemTitle = "WWDC Sign"
```

Leverage NSURL

As I briefly touched on above, you'll want to try and use `NSURL` instances when vending data to the preview controller. Unless, that is, you need to also control the title – in which case, you'll want to create a lightweight class conforming to `QLPreviewItem` as above.

It's likely that most of you are writing your apps in Swift, as such, you can leverage toll free bridging to `NSURL` from Swift's `URL` type easily (and vice-versa):

```
let url = myURL as NSURL
```

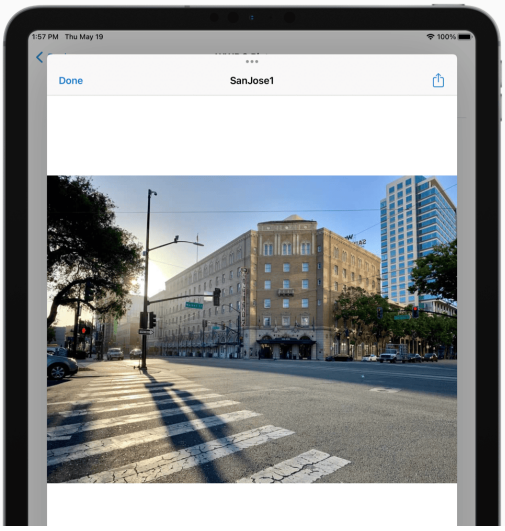
Present Quick Looks via Interactions

Quick Look offers a few ways out of the box to quickly preview files via system-supported gestures. Think of pinch interactions or showing a context menu. You can do this with `QLPreviewSceneActivationConfiguration`, where you simply give it the files and the framework takes care of the rest.

```
// Xcode -> UIKit -> QuickLookPreviewFig2ViewController.swift

func tableView(_ tableView: UITableView,
contextMenuConfigurationForRowAt indexPath: IndexPath, point:
CGPoint) -> UIContextMenuConfiguration? {
    guard let item = datasource.itemIdentifier(for: indexPath) else
{ return nil }

    let url = item.generateURLData().url
    let openFileAction = UIWindowScene.ActivationAction.init
({ provider in
        let options =
QLPreviewSceneActivationConfiguration.Options()
        options.initialPreviewIndex = indexPath.item

        let quickLookActivation =
QLPreviewSceneActivationConfiguration(itemsAt: [url as URL],
options: options)
        return quickLookActivation
    })

    return UIContextMenuConfiguration(identifier: nil,
previewProvider: nil) { _ in
        return UIMenu(title: "", image: nil, identifier: nil,
options: [], children: [openFileAction])
    }
}
```

So if you tapped on the context menu item for that, we'd get the preview controller all setup for us in a prominent window:

Prominent Window Presentation



Or, if you wanted to let any `UIView` support this, you could via `UIInteraction`. So following the example above, you could tweak it to support that interaction in any view you wanted like this:

```
let options = QLPreviewSceneActivationConfiguration.Options()
options.initialPreviewIndex = datasource.indexPath(for:
item)?.item ?? 0

let url = item.generateURLData().url
let quickLookActivation =
QLPreviewSceneActivationConfiguration(itemsAt: [url as URL],
options: options)

let interaction = UIWindowScene.ActivationInteraction { interaction,
point in
            return quickLookActivation
        } errorHandler: { error in

        }
```

```
view.addInteraction(interaction)
```

Also, take note that we were able to set the selected index, much like we did with `QLPreviewController` directly at the start of the chapter, by using the `QLPreviewSceneActivationConfiguration.Options()` object:

```
let options = QLPreviewSceneActivationConfiguration.Options()
options.initialPreviewIndex = datasource.indexPath(for:
item)?.item ?? 0
```

You can achieve the same effect in SwiftUI, too:

```
struct QuickLookPreviewFig2View: View {
    private let allURLs: [URL] = Demo.allURLs()
    @State private var url: URL? = nil

    var body: some View {
        Button("Street Corner") {
            // Chose the first image, but whichever
            // You use for `url` here will be the selected
            // Asset, so it needs to be present in allURLs.
            url = allURLs.first!
        }
        .buttonStyle(.borderedProminent)
        .quickLookPreview($url, in: allURLs)
    }
}
```

Three Key Takeaways

1. Quick Look can help you support virtually any file type outside of your app's primary supported file types.
2. If you have your own file types, extend them with `QLPreviewingController` to ensure they can be opened with Quick Look.
3. Finally, you can also create custom thumbnails to represent your file types.

In-App Purchases and Subscriptions

If you're reading this book, you develop or design iOS apps. And you probably don't it for free, either. Which means that by selling your expertise through consulting, working at a company or as your own employer – the software you work on has to do one thing at the end of the day: make money or generate revenue in some form or fashion.

While there are many indie developers out there who simply ship delightful software for the love of the game, eventually – reality tends to set in. You want to keep doing what you love, or perhaps you wish to stop doing it for another company and do it on your own terms. And, the way to do that is to generate paying customers.

In-app purchases and subscriptions are the main drivers for revenue for most iOS apps. This chapter is all about how to handle them from a user experience angle, something that doesn't typically get lumped in with monetization. But it should, be-

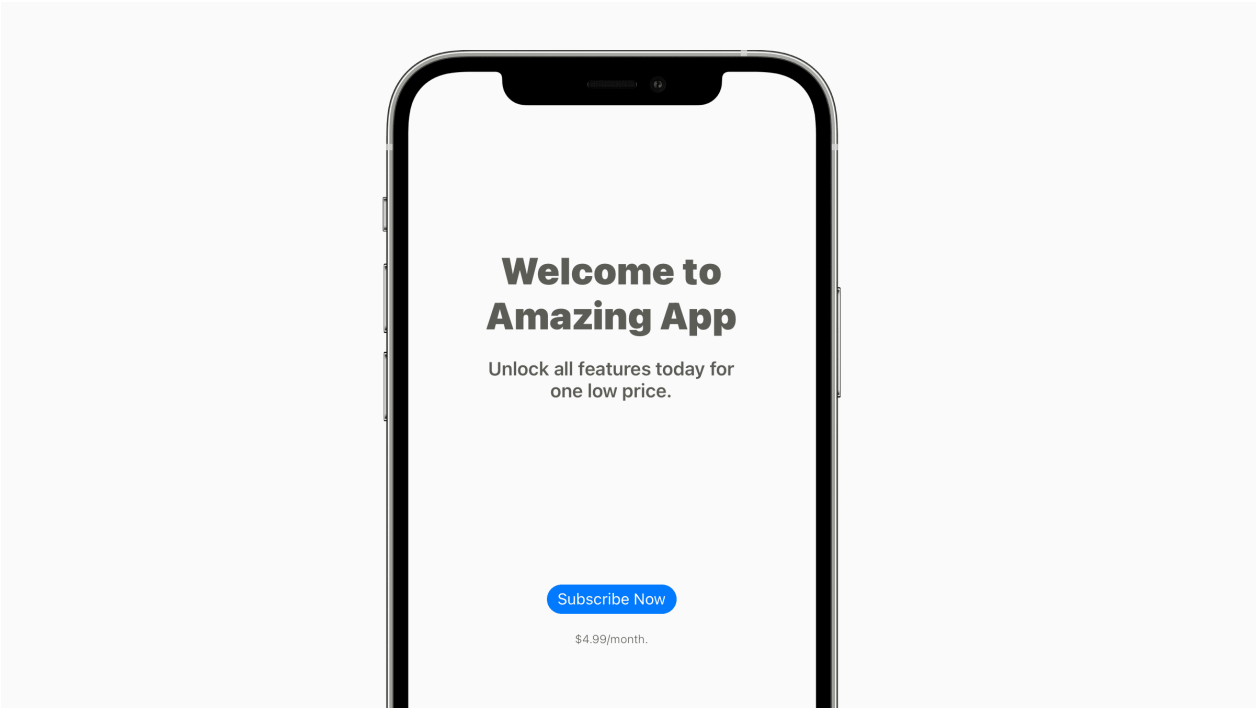
cause while great user experience brings forward several advantages for your app, it also does another thing particularly well – it *sells*.

What's more, the bar here isn't really that high, unfortunately. With the App Store constantly fighting back against scam apps, gamed ratings and fake reviews – customers have become somewhat used to putting up with (to be blunt) crap.

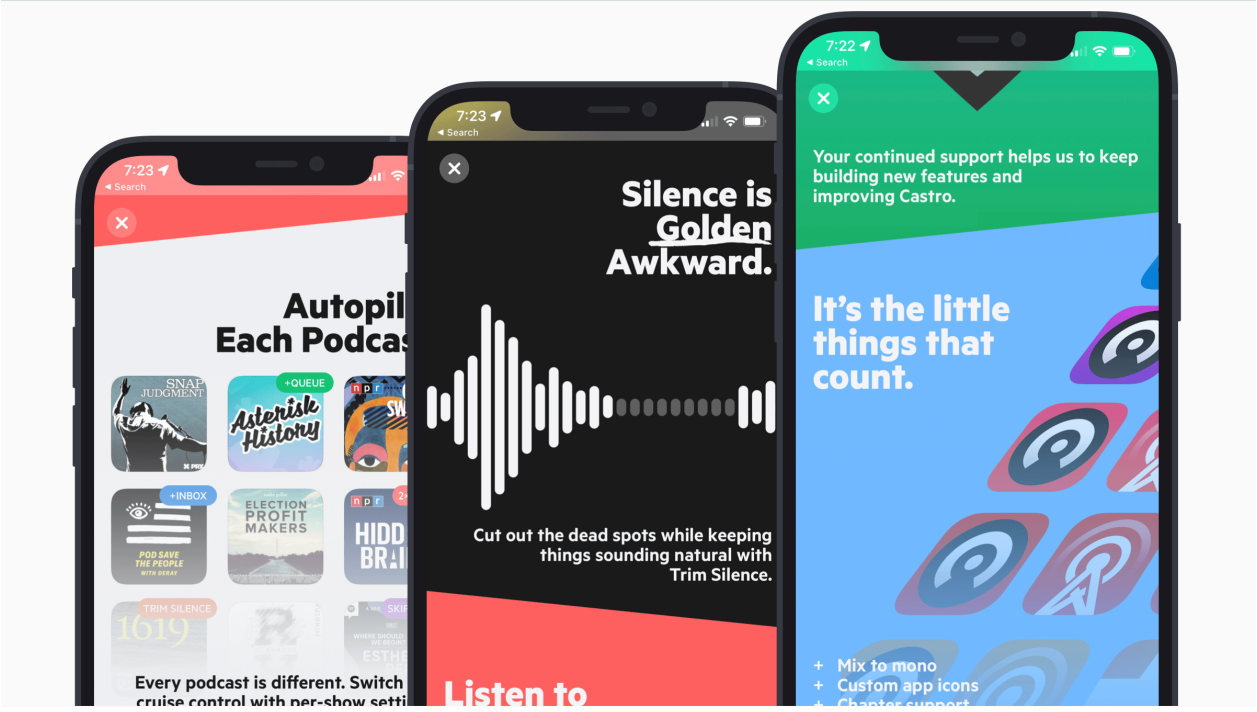
But it shouldn't be that way, and there are many apps that do it right – and this chapter is here to make sure yours is one of those. To be clear, this isn't about how to convert well, tweak copy for more activations or how to decide on pricing strategies. Those topics all have their place, and they are important to think about. But they aren't a fit here – because this chapter is all about tips on making it feel *right*, whatever route you take. And if you feel right, many times those things will become a natural byproduct.

When it comes to selling something, a lot of developers tend to forget the human connection. We often tend to think of marketing as something that's a bit self-serving or disingenuous and that people just tend to put up with it. I'd suggest that really isn't true, and people actually love to be sold something. The real issue seems to be we have forgotten the art behind how to actually do it. When I look at a lot of the apps out there, it's become clear to me the problem is that we don't *sell* people things, we just tell them what they can buy – and there is a difference.

This is shoving a price point into someone's face:



And this, from Castro, is *selling* me something:



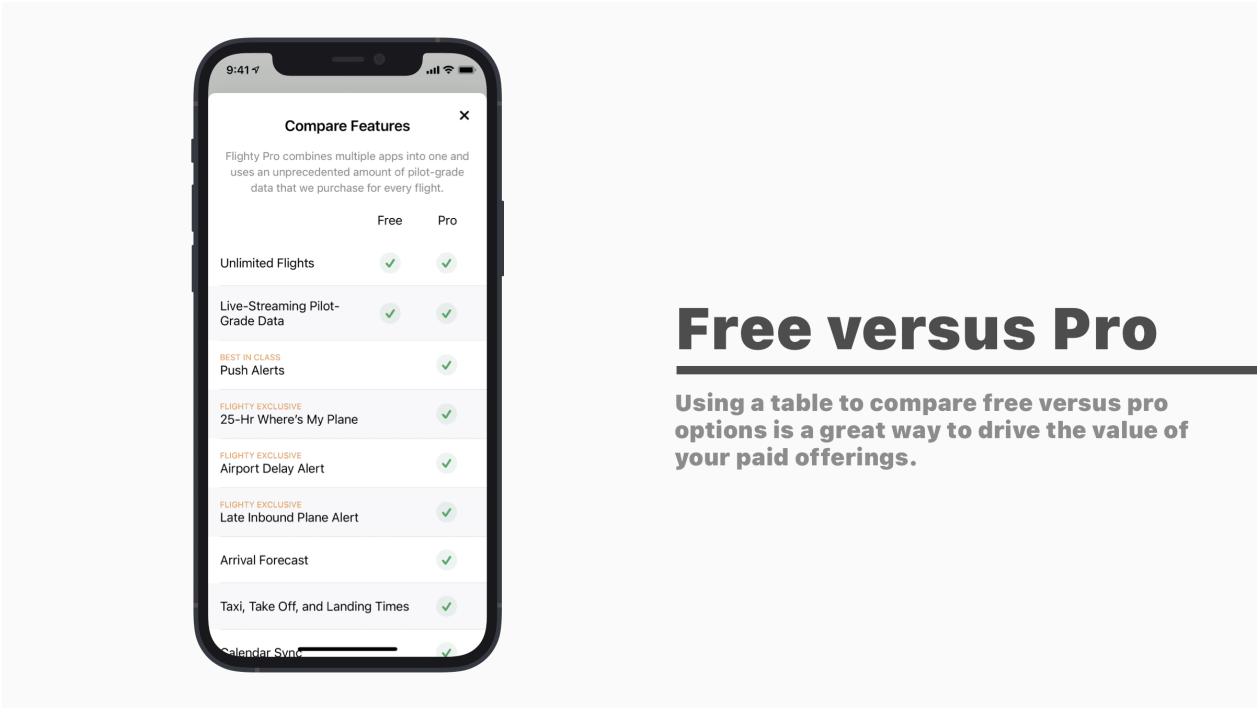
One pretty much just said "Here is a price" while the other took me on a journey of why I'd love the product, should be happy to pay for it and should become a customer. You want to opt to be the latter, not the former.

The rest of the chapter will give you some tips on how to get to that point.

Clearly Show Selling Points

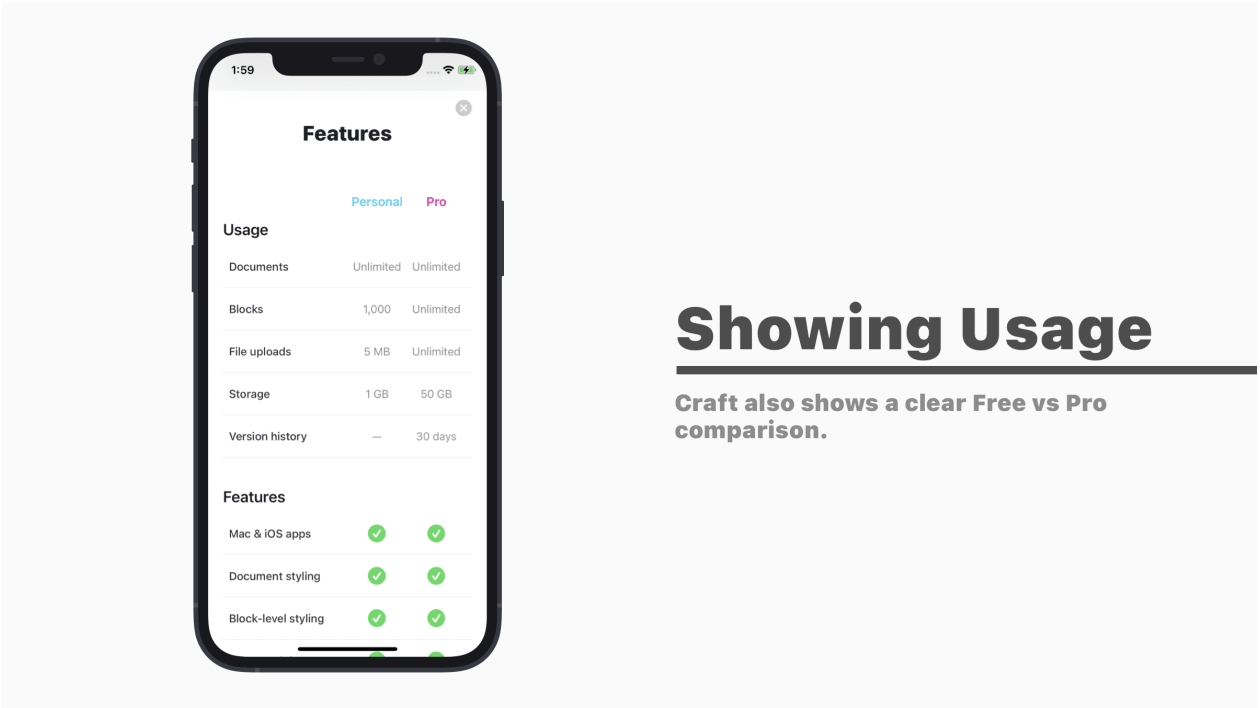
When it comes to selling on iOS, you want to have a great experience around showing the value of your key product features. To that end, having a clear delineation between what they *have* and what they will *get* is key. You want to subtly remind people that, while the free plan or features they might be currently using are great – the ones that are "paywalled" are even better.

To do that, though, it's best to keep that information incredibly easy to understand. Here, the app Flighty clearly demonstrates the value it brings to the table with a subscription tier versus the free one:



Note that it doesn't diminish the value of its free offerings, but rather — it shifts focus over to how great the paid ones are. Adornments like "Flighty Exclusive" and "Best-in-Class" help drive value adds further.

Craft Docs takes a similar approach:



It sounds simple on the surface, but if you can list out what is free and what is paid – people can make an informed and trusted decision about whether or not to convert, and that's a great user experience.

Time to Paywall

Speaking of showing free versus paid features, you need to put a lot of thought behind how you present those things. For example, even though the examples above were really well done – they wouldn't be as well received if they were shown right after launching the app.

You just feel icky when you run into this, and you should try to avoid it as a developer or designer. I realize that for some apps, there is no free trial, freemium version or any other option *but* to pay – but as we said at the top, there is still a way to sell.

People don't feel great about your app when you hit them with a sales pitch from the start.

So, that naturally begs the question – when should you? The moment from when they launch your app to the moment you show any sort of sales prompt is what I call "time to paywall", and there isn't an easy answer on what that should be because it'll depend on the experience your app offers.

Regardless, here are some techniques to consider to get your thought process started:

1. **Discoverability**: People who use your app see paid features shown in the interface, and interacting with them shows your paywall.
2. **Onboarding**: Yes, you can still do it as part of the app's onboarding – and many apps do. But, there is a good way to do it.
3. **Action-Based**: Here, you track any relevant metrics locally in your app and when people hit a certain threshold, you present the paywall. This is a similar method to how several apps go about presenting their ratings prompt.

There are, of course, other avenues to take – but these are some common ways I've seen that work and respect the user at the same time.

Discoverability

When using this approach, your app has either a free tier or freemium aspects that let people naturally use your app without committing to any sort of payment. To educate them on things they could do with a paid in-app purchase or subscription, you might show options that are typically "paywalled" in your interface.

For example, if you had a photo editing app that has premium filters – perhaps you might display those filters alongside the free ones. When they interact with it, then

you present your pay wall. Note that if you go this route, I think it's a good idea to clearly show that whatever feature is being presenting is, in fact, a paid one. Perhaps you can use a lock icon, or a "pro" label so it's still discoverable but obvious that it's a paid feature.

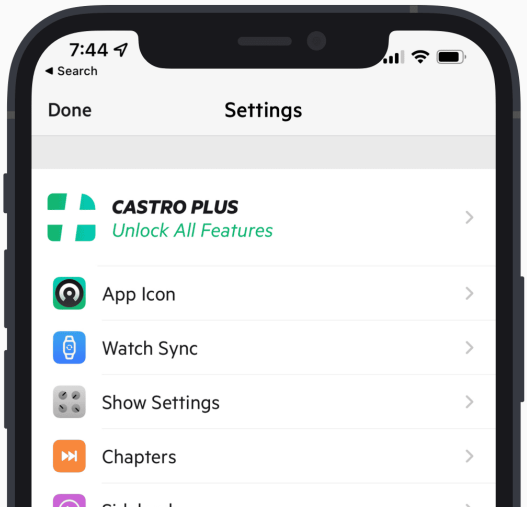
This way, they don't feel like they were "gotcha'd", which might annoy them. And, well – annoying people isn't a great way to get them to pay for things. If you really want to step up your game here, you can even provide an option to hide these altogether.

While this might be a bit of work, it lets people feel like they are in control – and again, they should feel like they are in control (and should be) when it comes to buying experiences.

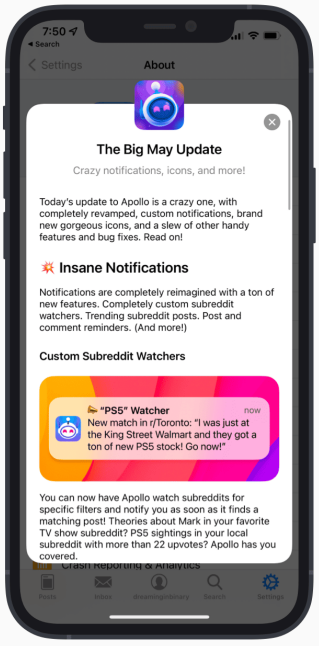
Another thing to consider here is to pin down natural touch points in your interface where it would make sense to show paid offerings. One of those is any sort of "Settings" view:

Showing Subscriptions

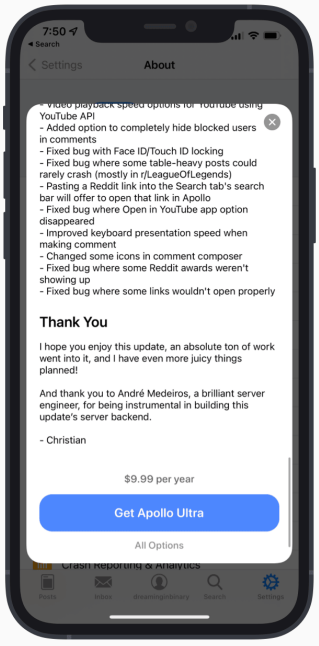
Make it easy for people to discover premium options.



And another might be after a large update ships, like the Reddit client Apollo has done here:



Apollo offers its paid tier at the end of its release notes for a large update.

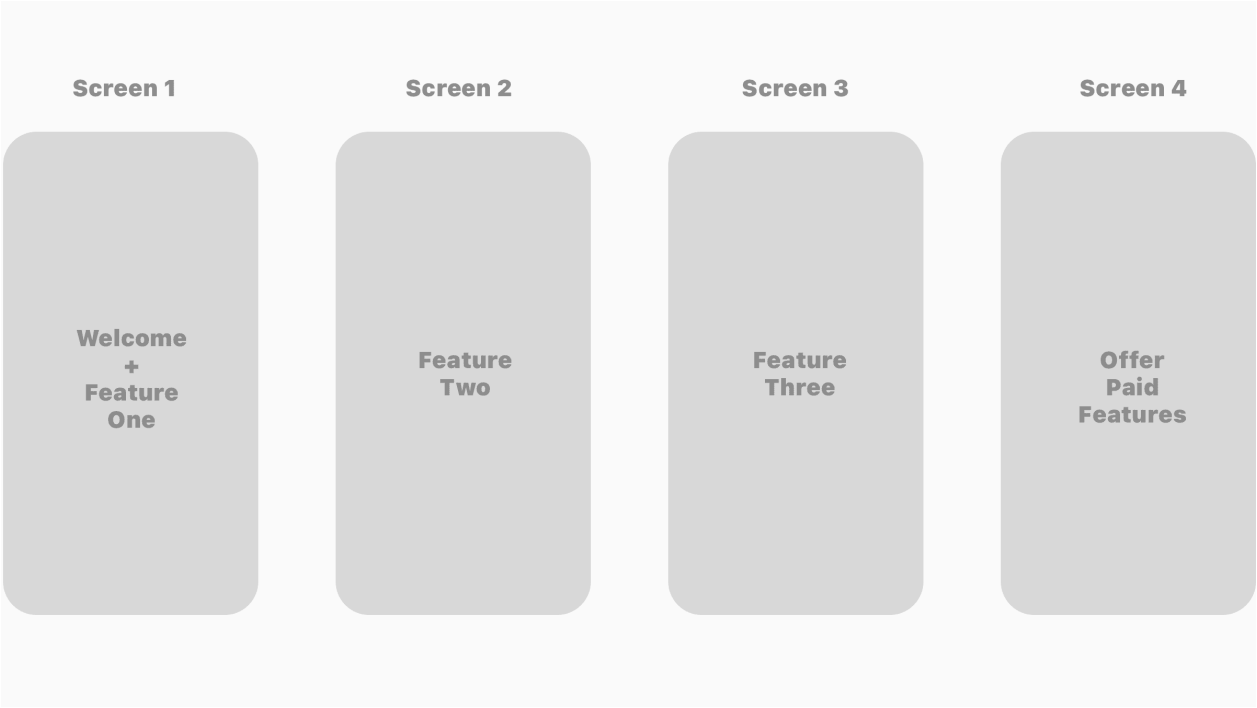


It's a balance with this – you're basically trying to make it front and center without really making it obviously front and center. I know that sounds cheesy, but it's true. When people are in your app a lot, they like it. Remind them it can be even better, but tastefully.

Onboarding

Several apps require a subscription to work, and that's when you see the paywall as part of the onboarding. It's simply part of the deal – perhaps the app is part of some sort of S.a.a.S. product (software as a service) where there are no free options. If that's the case, you want to deliver value upfront and fast.

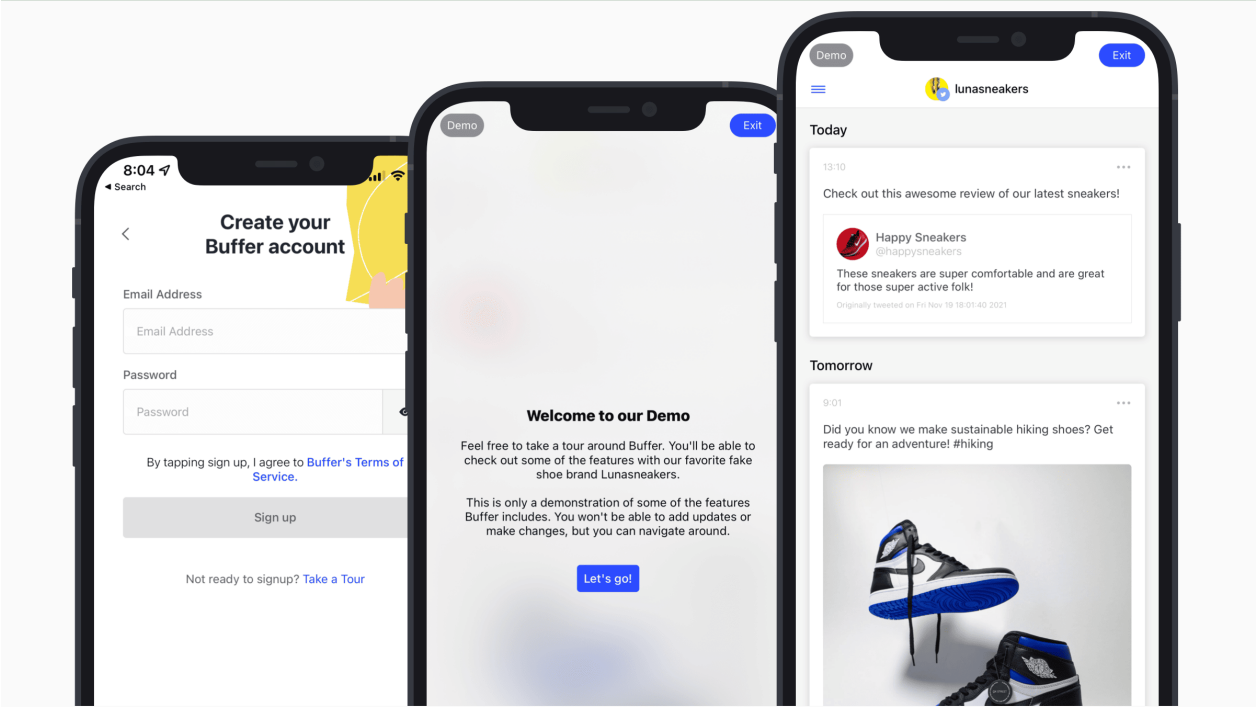
Instead of immediately showing the paywall, try for for something like this:



Instead of simply getting right to the payment, tell me why I would want the service at all. Further, if you can swing it – you can really help this scenario by including some sort of "demo" mode.

When you use the paywall during onboarding, several people are going to drop off right then and there. And if they aren't familiar with your product, why wouldn't they? Even if you used the "good" example above, they had no way to really feel out the offerings before deciding to pay for it or trial it out.

Offering some sort of "read only" demo mode helps to mintage that issue. The app Buffer was originally started a SaaS product, with the its iOS app coming along later. No matter, it offers a fully featured "tour" that acts as a sandbox where people can play around with its offerings to get a feel for if it:



This is great, because now they can either confirm it's what they were expecting and will be more likely to pay for it, or they realize it's not and they leave. Either one of

those is fine, because as much as you want paying customers — you don't paying customers paying for anything other than what they thought they were expecting to get.

On one hand, I get it. Most of these apps that offer the paywall during onboarding are operating under the assumption that a majority of their users came from a web app, first. The app is simply an extension of it, and they already have a paid account. But on the other hand, that certainly won't always be true because along the way you'll have organic discovery happen. Why not optimize for those people, too?

Action-Based

With this route, you meter a person's actions in your app to determine when a paywall should show. This could look like:

1. Counting "key actions" taken. For a social media app, perhaps it might be creating X posts before showing the paywall.
2. Ensuring they've used the app for X amount of time or days.
3. A combination of the two above.

Whenever those things occur, then can you can pop up the paywall because you have a degree of confidence that they already enjoy what your app brings to the table.

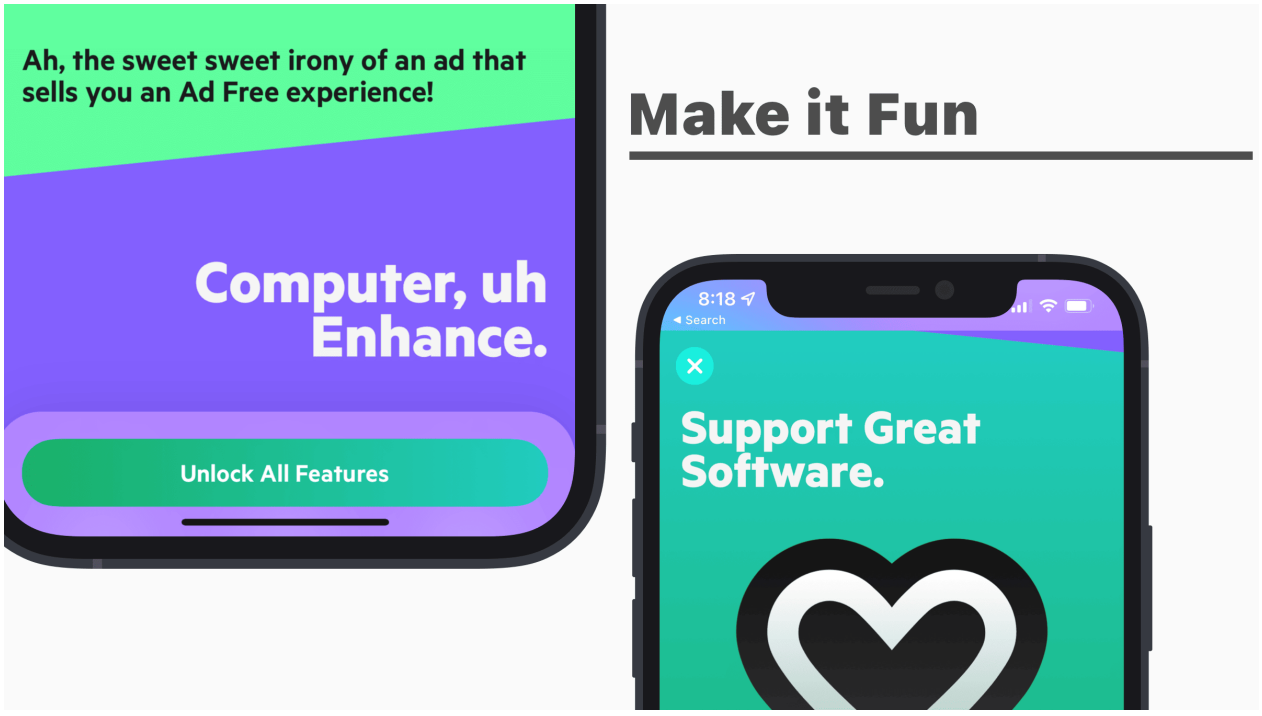
Another way you can handle this is by subtly showing a banner or some other "call to action" prompt, because when you pop up a modal paywall without explicit action from their end — that can cause some frustration. So, in one of the more prominent views in your app, you could show a banner that says "Explore Pro" or something similar. That way, people can interact with it on their own terms instead of it being forced on them.

Remember what we said earlier, people want to feel like they are in control of the end-to-end buying experience. They don't want to feel like they were forced into buying something.

Don't Be Boring

This may seem obvious, but if you can spice up the buying experience for your app – do it. Depending on what your app's voice is (if you have the design book from this series, check out the "Finding Your App's Voice" chapter for a refresher), you can tailor the buying experience along with it.

For example, let's revisit the podcast app Castro. It could've simply listed out what subscribing will get you, but instead if shows beautiful graphics along with what each feature does to really drive home the value:



That's *fun*, and I admit — when I first saw this, I had no intention of subscribing to Castro, but I rethought that choice after seeing this.

Handling Refunds

Refunds are simply a part of selling anything. You will absolutely, positively run into a refund situation sooner or later. And that's ok — but what we want to avoid here is making those refunds hard to perform.

Thankfully, you can have more control over the whole process now in iOS. Even though Apple handles transactions (and still do), you can point them in the right direction far easier than you used to be able to:

```
try? Transaction.beginRefundRequest(aTransaction)
```

Using the above function, you can present the system provided interface to request refunds. Plus, it'll provide the status of the refund as well:

1. `Transaction.RefundRequestStatus.userCancelled`: The user decided to cancel the request.
2. `Transaction.RefundRequestStatus.success`: The refund was processed.

This flow is a lot smoother than answering a support ticket about a request, or even pointing them to the refund destination with a url (as of today, that's https://support.apple.com/en-us/HT204084 for products in the United States). Since you also know the result of the refund request, you can also tailor copy in your app to reflect that — or possibly even offer them the opportunity to reach out for more help.

Refunds are inevitable. Handling them gracefully counts.

Show Timely Information

In addition to simply offering a nice in-app purchase flow or subscription opportunity, it also helps to make important information about those things available within your app. Much like our discussion about refunds above, you also want to keep them within your app if you can for things like this. It just feels easier and doesn't introduce any unnecessary friction.

So, what should you show? Here's my list:

1. **Offer codes:** If you've got any to use, based off a number of factors perhaps driven by reactivation scenarios, list them here.
2. **Subscription Management:** This is a big one, as most of the revenue these days are driven via subscriptions. So, make it easy for people to see the status of theirs.

   2a: Renewal Dates: Make it easy for people to know when their subscription will renew.

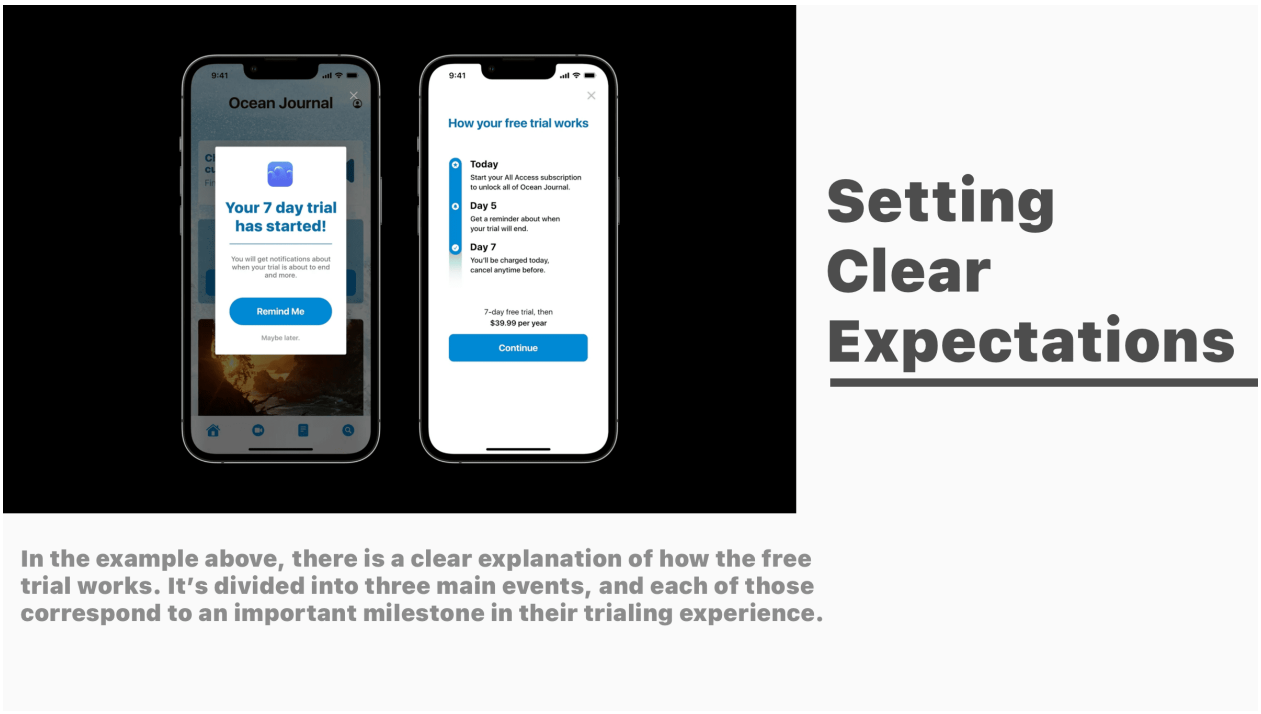
   2b: Current Subscription Tier: Do you have multiple subscription tiers? Likely. Show it to them.

For the renewal date, subscription status and more, you can leverage StoreKit to query all of that information:

```
let info = myProduct.subscriptionInfo
```

Set Expectations Upfront

When someone is about to sign up or trial your subscription, consider a commonly used design pattern to set timeframes, pricing and expectations right away. In fact, even Apple themselves are recommending this flow now:



In the example above, there is a clear explanation of how the free trial works. It's divided into three main events, and each of those correspond to an important milestone in their trialing experience.

When you do this, there are no surprises. This is especially critical if your time to paywall occurs during onboarding. Whatever your trial duration is, try to break it down into three critical junctures:

1. **Today:** Let them know they are on the trial, and that all of the relevant features are free to try.
2. **Payment Reminder Date:** Here, you'll send them a push notification to remind them that they've got a few days left in their trial. If you've got a back-

end set up, you might consider doing this via email too. It's possible the user didn't grant your app push notification permissions.

3. **Charge Date:**Here, their trial has ended – and if they didn't cancel, they are now a paying customer with all of the feature sets unlocked that they were trialing.

Personally, I enjoy this flow. I think it's *honest*, and when so many apps aren't – being honest is an easy win.

Don't Be Trashy

To be upfront, I rewrote this section a few times. I felt it was a little harsh, or maybe too pointed. Look, we all would like to make money on the App Store. Apple wants that, we want that and customers who enjoy what you make will give you that.

So don't be trashy, scammy or otherwise do shady things in your app. If you're reading this book, I already know that you care. So, you know what this looks like. Hiding the close button on paywalls. Being dishonest with copy. On and on.

Taking shortcuts with apps and payments is the quickest way to a bad reputation. Don't do it.

Tailor Promotional Copy

Your app is going to be handling a lot of state during in-app purchase or subscription flows. They are in a trial, not in a trial, up for reactivation or a thousand other combinations of those things and others.

It's a lot to track.

State has always been difficult in software, but take extra care to see how your copy holds up in several flows. StoreKit 2 allows for you to test these things much easier than it previously was, too.

Here's an example: If a user is eligible for a free trial, then prompt them to use it. Show something like "Try it Free for 7 Days" somewhere in your interface. However, if they later come back to your apps and they've used that trial – don't show it again if it's not longer valid.

Using StoreKit APIs and its notifications, you can keep tabs on a lot state like this. To be honest, if you mess this up – it doesn't instill a lot of confidence to potential customers that you can get more important things right. I mean, *I* get it, these things aren't easy, but regular, everyday customers aren't as forgiving with these types of things.

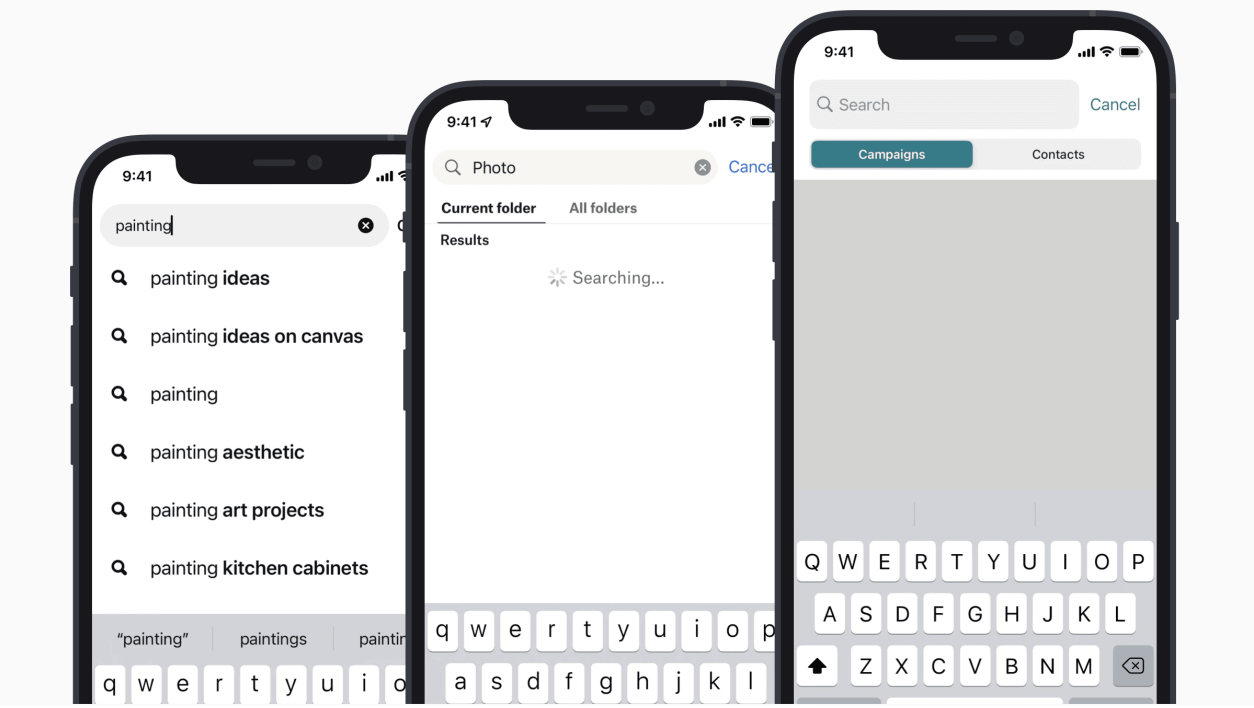
Three Key Takeaways

1. As your revenue driver, making in-app purchases or subscription flows pleasant *literally* pays.
2. Find the balance between showing your app's true value and asking for conversions, and try to win them over before you even present any sort of monetization screen.
3. Realize getting this whole thing right takes a lot of misses along the way – so be especially perceptive to how your purchasing flows are working and be ready to change things quickly.

Search

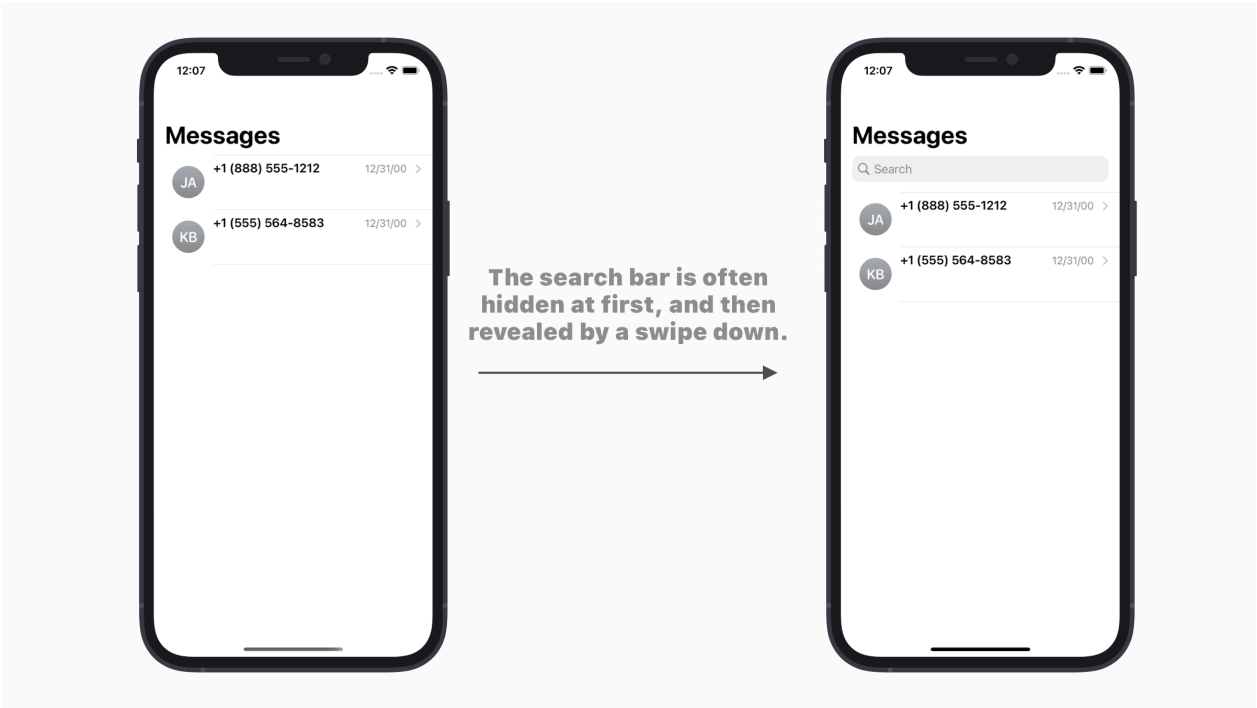
Apps are full of all sorts of data, no matter if they have mountains of it such as a social media app might, or just a little bit of it – more common in apps focused towards a singular purpose. No matter the data density your own app has, one thing is certain – people will want to find a particular part of it at some point, and they'll want to do it quickly.

A good search user experience helps people get around quicker, filter data and drill down into things they frequently visit. On iOS, the searching experience has been well defined for several releases – and I don't expect it to change much anytime soon. The most common setup is a search bar that's found within a navigation bar:



If you need to form a stronger vocabulary around bars in iOS, visit the "Bars" chapter found in the design book in this series.

By and large, it pays to follow the conventions iOS has created around searching. It's not really a time to get fancy, however – there can be a time and a place to create a customized search experience as we'll see later on. Search bars are typically hidden when people visit a screen that has them, and it's exposed when someone scrolls down:



While that's typical, you can also opt to have the search bar showing by default:

```
navigationItem.hidesSearchBarWhenScrolling = false
```

Also, there can be an accessibility impact here as Voice Over may not catch it if it's not showing – so be sure to test for this and if need be, show the search bar:

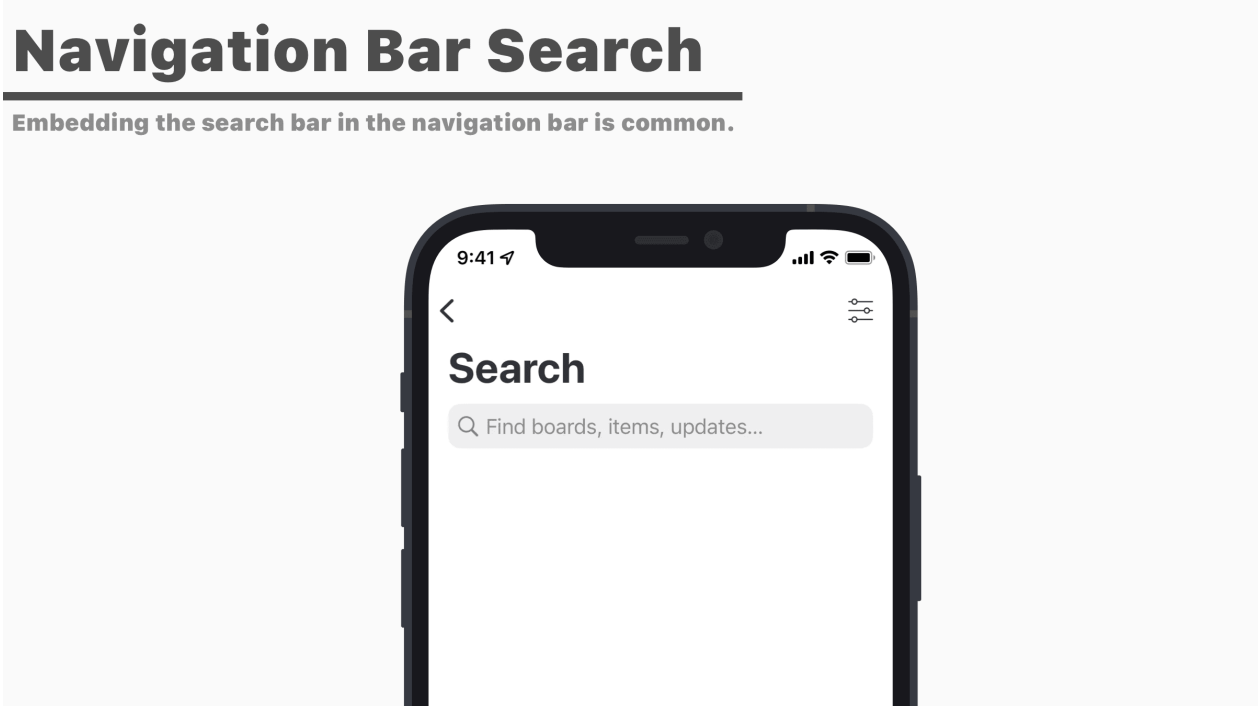
```
if UIAccessibility.isVoiceOverRunning {
    // Show search bar by default
}
```

Where to Include Searching

Searching can occur in a few common spots in iOS such as within navigation bars, tab bars or perhaps activated by a dedicated button or shown in a bottom sheet.

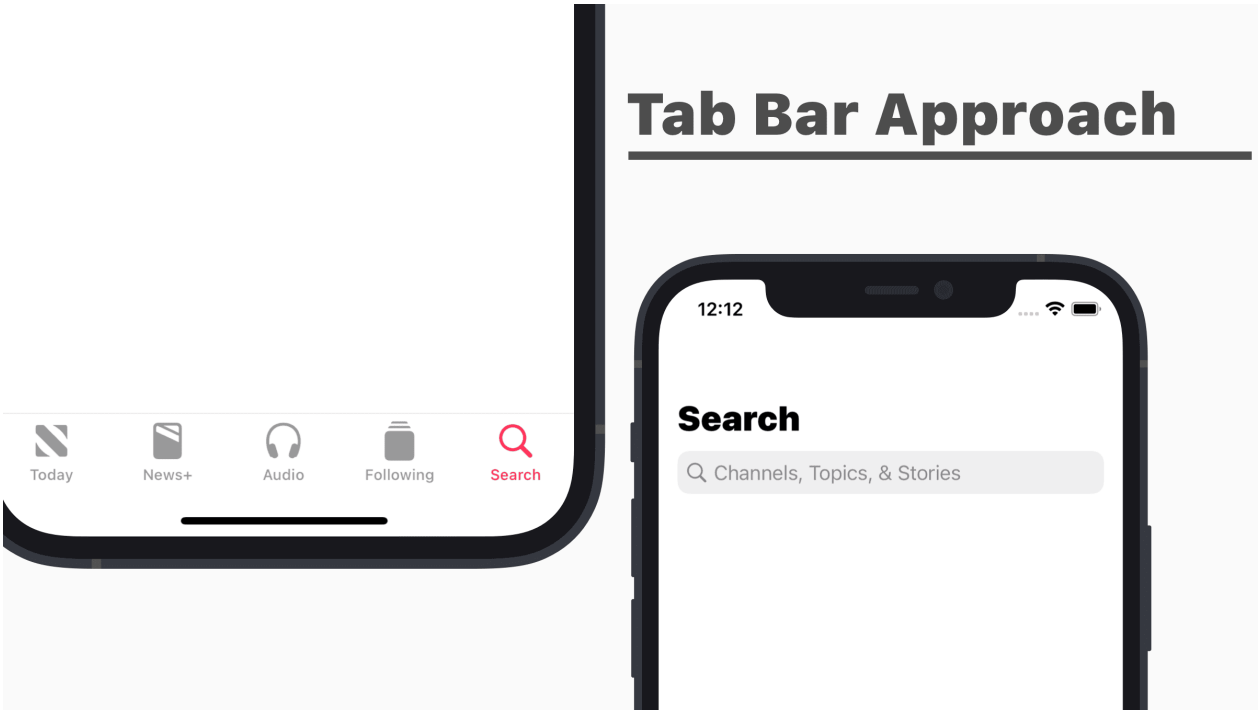
Navigation bars

By far the most common approach on iOS, including search bars in navigation bars has a lot of system support, is well understood and is probably the option you should use a majority of the time:



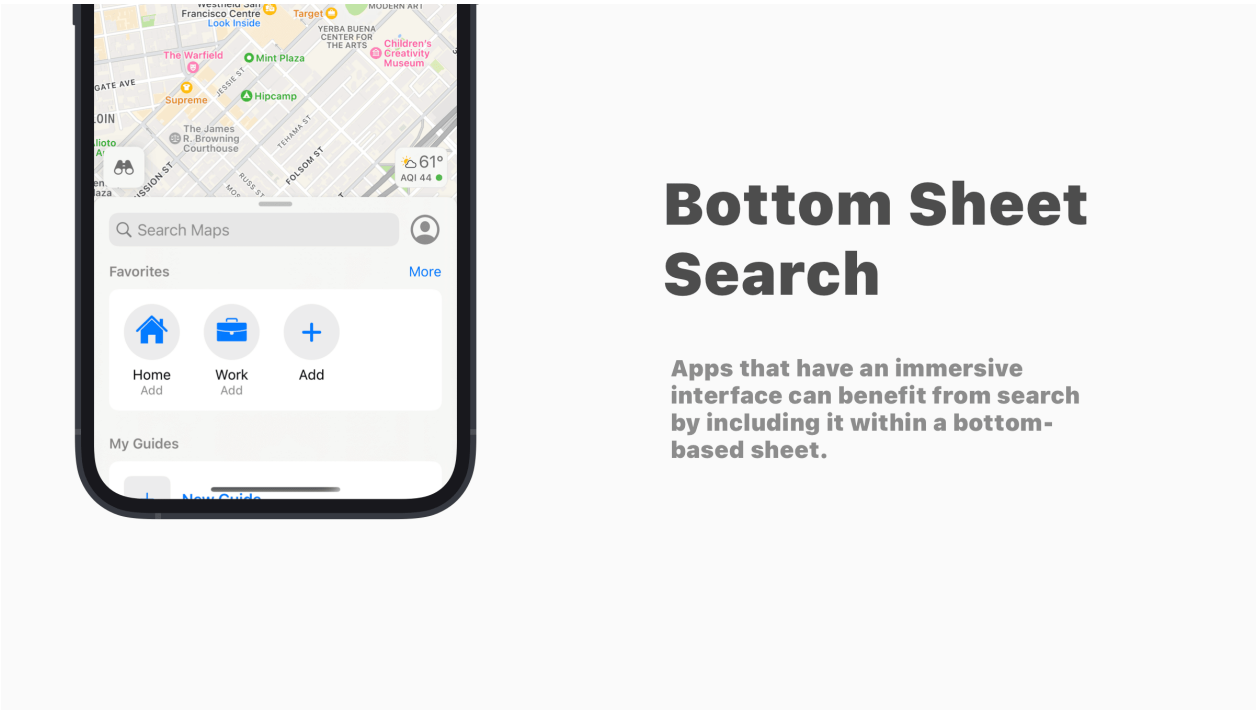
Tab Bars

If you're using a navigational structure that lends itself to tab bars, sometimes it can be helpful to provide a dedicated search tab bar item. This allows near-immediate access to searching. For example, the App Store follows this pattern. Once the search button is tapped, however, the common patterns still apply:



Bottom Sheet

Here, the search interface is shown towards the bottom. You can see this approach in Maps:



This pattern is useful when you have a more immersive interface, much like Maps does, where you don't have a top navigation or tab bar as part of your interface.

No matter where you activate search from, remember these three rules to make sure it remains easy to find and use:

1. **Make it obvious:** There should be no surprises when it comes to searching interactions. Stick to common patterns, copy and iconography that are well known to indicate interacting with a certain element will begin search.
2. **Make it quick:** Even if you need to hit a server for search results or otherwise load a large data set in, make your search as snappy as you can. Present the search interface right away, even if you don't have all of the data you need yet.
3. **Make it accessible:** People shouldn't ever have to dig around for your search functionality, so always keep it visible if that's possible. It shouldn't

be hiding out in a menu or a settings view, unless it's searching exclusively for the data found within those places.

Components of Searching

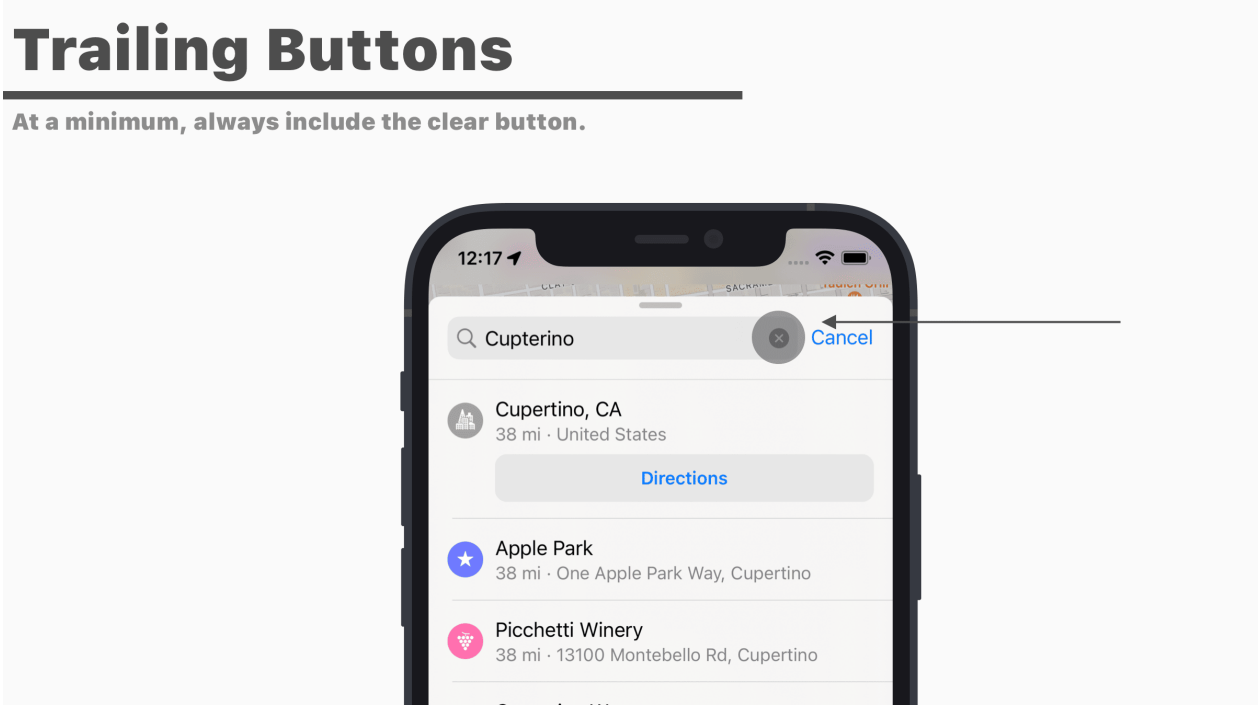
To create a good search experience, it helps to form a vocabulary around the common components of iOS' search controls. Here are the most common ones you're likely to use, provided you stick to standard controls.

Search Input

The search field will typically be embedded in another bar, or pinned to the top of the screen. This is the primary search component, in UIKit – that's `UISearchBar`. In works in conjunction with `UISearchController` to coordinate search results and update the interface. In SwiftUI, it's handled via a modifier, `.searchable(text)`. Its placement in SwiftUI will be determined by where in your view's `body` you use the modifier in conjunction with the environment. Using that information, it'll pick the most sensible spot.

Search Input Buttons

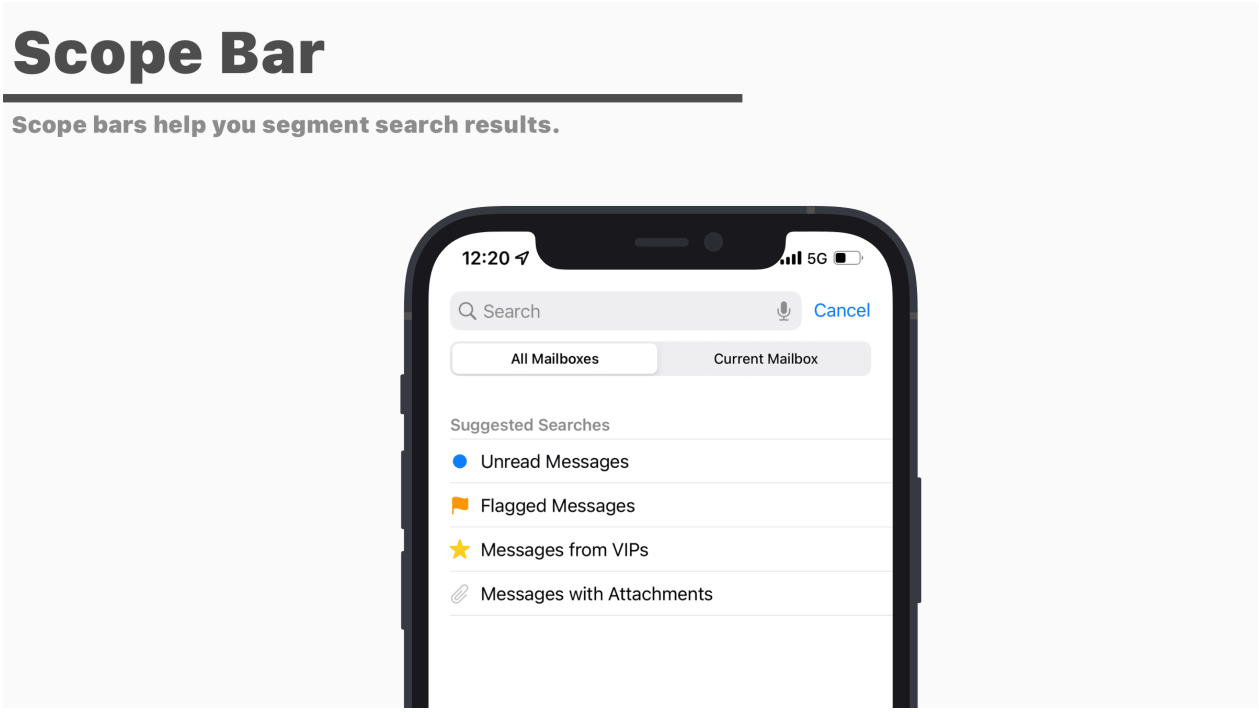
To help quickly clear results, filter them or otherwise operate on the search terms, many search bars contain buttons in the trailing position:



You can add these yourself, or use some system provided ones such as a clear button or a bookmark button. I highly recommend, at a minimum, always including the clear button. It's common for someone to want to quickly clear out their search term and start again, and selecting the text and deleting it is far more cumbersome than simply tapping a button.

Scope Bar

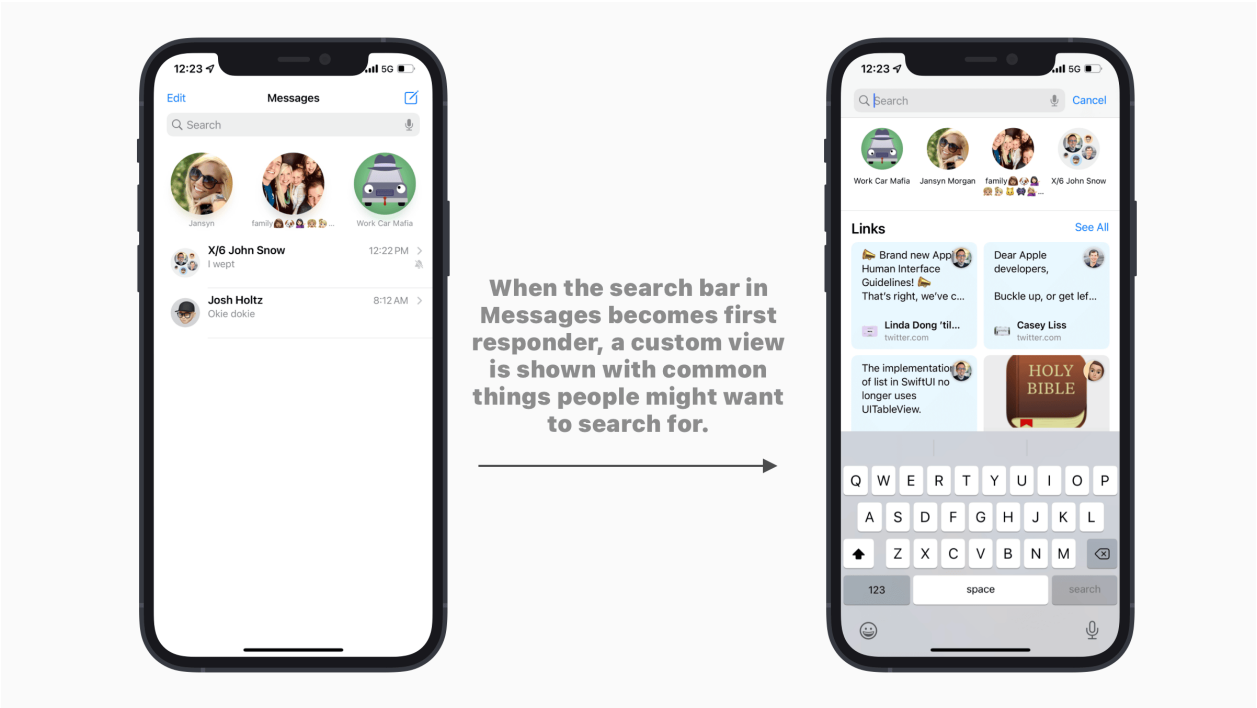
The scope bar sits beneath the search bar, and it's typically a `UISegmentedControl`. This aids in defining search results categorically. For example, in Mail, the scope bar offers two choices to filter results from:



Scope bars are helpful when you are aware of common search "buckets" in your app. If you're searching for documents in a note taking app – perhaps you'd offer a scope bar for a note type if you offered only a handful of them (such as .pdf or .txt). However, if the scope bar has more than three options, that's probably an indicator the filtering options have either outgrown the scope bar pattern or you're simply offering too much that may not be relevant.

The Search Controller

The last part is a little more involved. When you think of searching for things, it can commonly be a two step process. When search terms are added, either the existing data that's already showing is updated, *or* an entirely different view is shown. When the search bar becomes the first responder (i.e. it's been tapped) there could be a customized view which presents common search terms:



This is driven via a search controller in UIKit. It's up to you whether or not you want a new view to show when you activate search. If you want your own search controller to show when searching starts, you can indicate that preference to UIKit:

```
searchController.showsSearchResultsController = true
```

The search controller is responsible for showing the data you search, or you can forgo using it altogether. If all you want to use is a search bar and then filter down your existing data that's showing, you're free to do that as well.

In SwiftUI, this is all abstracted away from you. The "search controller" is mostly determined by how you setup the view hierarchy, and things like results and suggestions are driven by you explicitly:

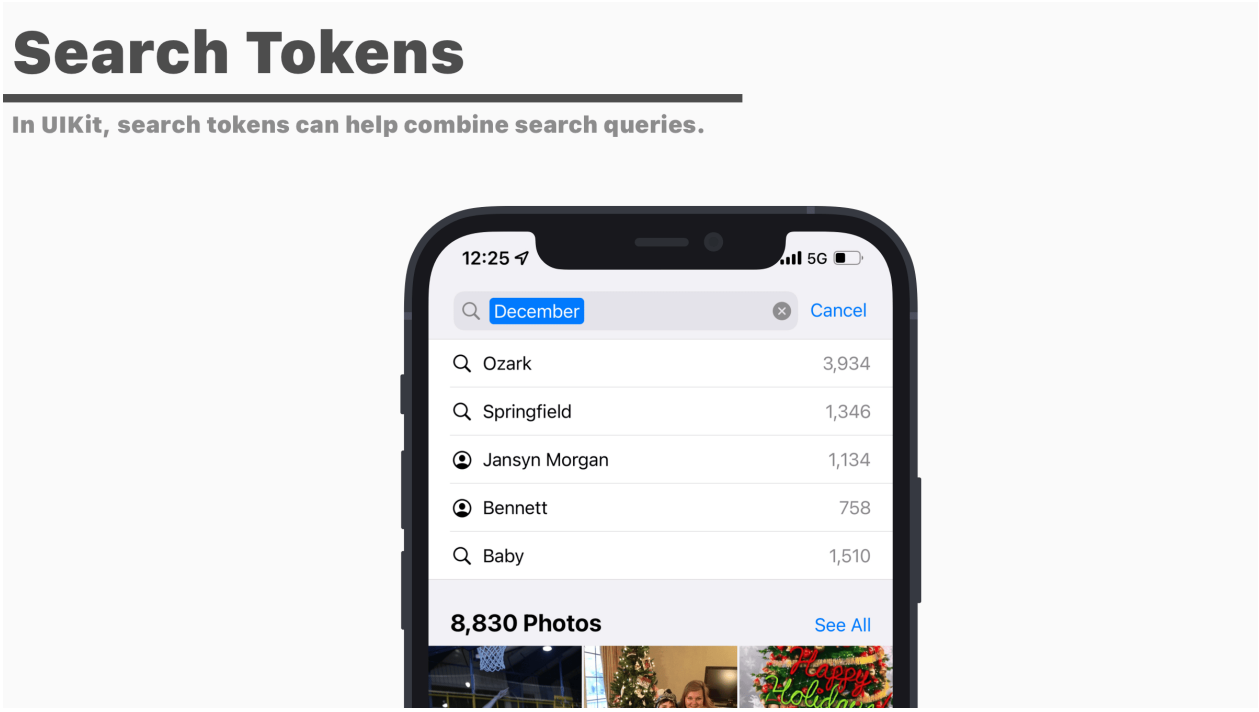
```
// Xcode -> SwiftUI -> SearchFig1View.swift

List(places) { place in
    Text(place.name)
```

```
}
.searchable(text: $searchTerm) {
    ForEach(suggestions) { suggestion in
        HStack {
            Image(systemName:"globe.americas.fill")
                .resizable()
                .frame(width: 12, height: 12)
                .font(.caption)
            Text(suggestion.name)
        }
        .searchCompletion(suggestion.name)
    }
}
```

Tips

Leverage Search Tokens in UIKit

UIKit has a nice feature called *search tokens* which help combine multiple search functions together. For example, in Photos, I can combine a year and a location in my search. Here's what they look like:



Search tokens are a powerful, yet understandable, way to create complex search queries. Ask yourself, what's easier? Searching "All photos from 2019 that were in New York City, New York" or simply combining a year and a location token?

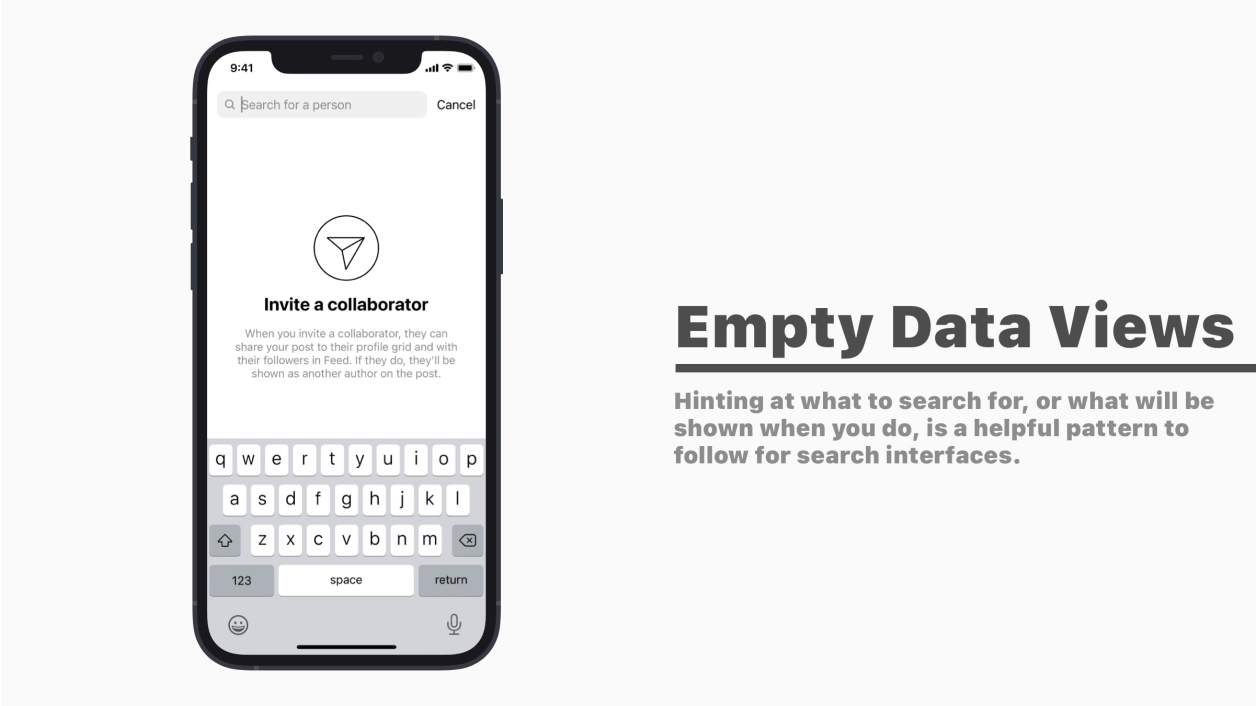
Hint at Common Data

When a user begins searching in your app, it can help to present a robust interface that shows the most commonly searched for things. As we saw in Messages, when I tap on the search field I'm immediately shown relevant things like recently shared links, contacts and more.

If you can do this right, you might be able to let the user skip their search altogether! Plus, it really can boost your "surprise and delight" factor. Many apps don't go this route, and I wish they would – so you can stand out by showing smart suggestions up front.

Empty Data

A common pattern to follow with searches is to show the "empty data view" when there aren't any results or nothing has been searched for yet:



The idea is simple enough, if the user hasn't searched for anything or there were no results — you can leverage the middle of the view to indicate as much. Unfortunately, there is no system level support for this. Implementation wise, it can a little tricky. But I think it's worth it, and it has utility outside of searches, too.

If someone first opens up your app and you present them with a list of stuff, this is a great place to hint to them what to do next. For example, if you were working on a social media app you might include copy that says something like "Create Your First Post."

For more on how to use stock UIKit or SwiftUI views in this case, see the "Reloading Data" chapter.

Feedback

When people search for things, you essentially have two different ways to give them the results. You can update data as they type, or wait for them to hit a submit (or similar) button to show results.

Whichever you do, be sure to provide feedback on where things stand. For example, most data isn't local to the app – you probably have to use some sort of A.P.I. to fetch it. Be sure to show a progress view when this is the case, whether that's the typical "spinner" or anything else.

It sounds so simple, and it is – but you'd be surprised how easy it is to find an app that has this kind of search but gives no indication to the user of where the search process is at. They'll just show a static screen, and that's confusing. Did I do something wrong, is it searching, did it freeze? So, update them.

Also, if your search takes more than about four or five seconds, it can be a nice touch to update the feedback to let them know you're still fetching data.

Also along the lines of "obvious, but sometimes missed" – always provide helpful placeholder search terms. Just slapping "Search…" as the placeholder text isn't as helpful as "Find bookings, flights and more…" for a travel app, for example.

Filtering Patterns

If the scope bar is too limiting for filtering down your data, provide another button to provide filtering options. If you provide a filtering icon, most people will know that's where to go to further refine their search. There are no hard and fast rules over what filtering should look like, but this is how I think about it.

If you've got, say – four to five filtering options, it probably makes sense to stick them in a context menu. Those are better suited to handling several options, and can even offer support to show another menu within a menu. For example, if you had an

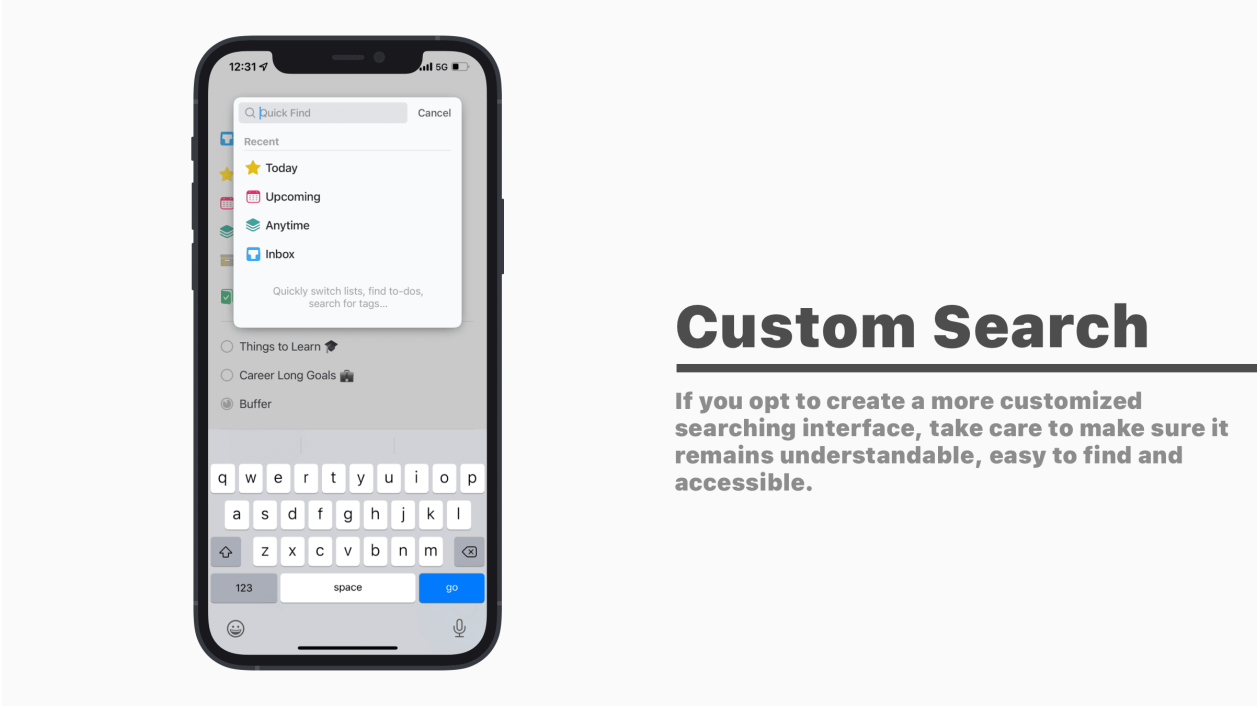
app centered around video games it could have filtering options for simply the video games themselves, *but* also filtering them by platform.

However, if it's more involved than that – consider a sheet or a modal view with all of the array of filtering options.

Customized Search

There are times when apps break from the established conventions of iOS' search patterns and do something new. This can either go great, or terrible – and it's rare if the end result doesn't fall into one of those two buckets.

If you feel the urge to go this route, try to design them in ways that reinforce all of the standard conventions. For example, a search button brings up search. A text field helps narrow things down. You get the idea. For example, Things 3 on iOS has a gesture to activate a modal, card-based search:



But, it's still functional because it's easy to understand – and beyond the card interface, it mostly works the same way as any other search does.

Also, if you're using UIKit, the search bar's text label is a public property. So, you can customize it however you want . If you're wanting to change fonts, colors and padding – this might offer all of the customizations you're after.

Three Key Takeaways

1. Virtually every iOS app can benefit from search capabilities.
2. Unless you have a compelling reason, stick to the defined patterns and conventions of performing searches on the platform.
3. You can make search more efficient by suggesting common data upfront when search is activated.

Loading States

Loading is bound to happen in today's server driven world. All sorts of apps need to pull in data from the network, load in large datasets or generally fetch something from somewhere else. While it would be nice if Moore's Law took over, and everything was so fast and efficient that none of this mattered – that's not the world we live in.

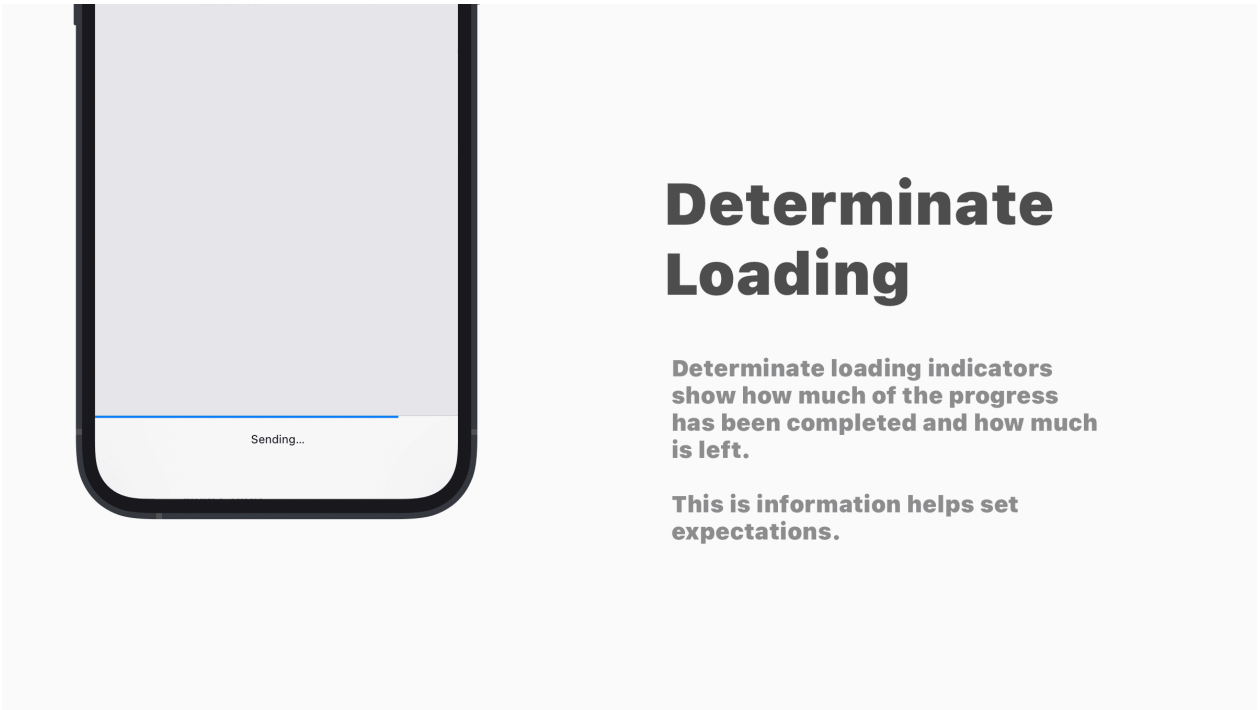
So it goes, while loading things may not be a glamorous task – we can certainly make it tolerable and informational for people, and I think doing those two things leads to a much better loading experience. Further, when loading *fails* (and it **will** fail) – we need to communicate this gracefully.

When presented with a situation where you have to load in data, try to keep these goals in mind:

1. It should be **informational**. If you can tell someone how far along the loading process is, that's usually a better experience.

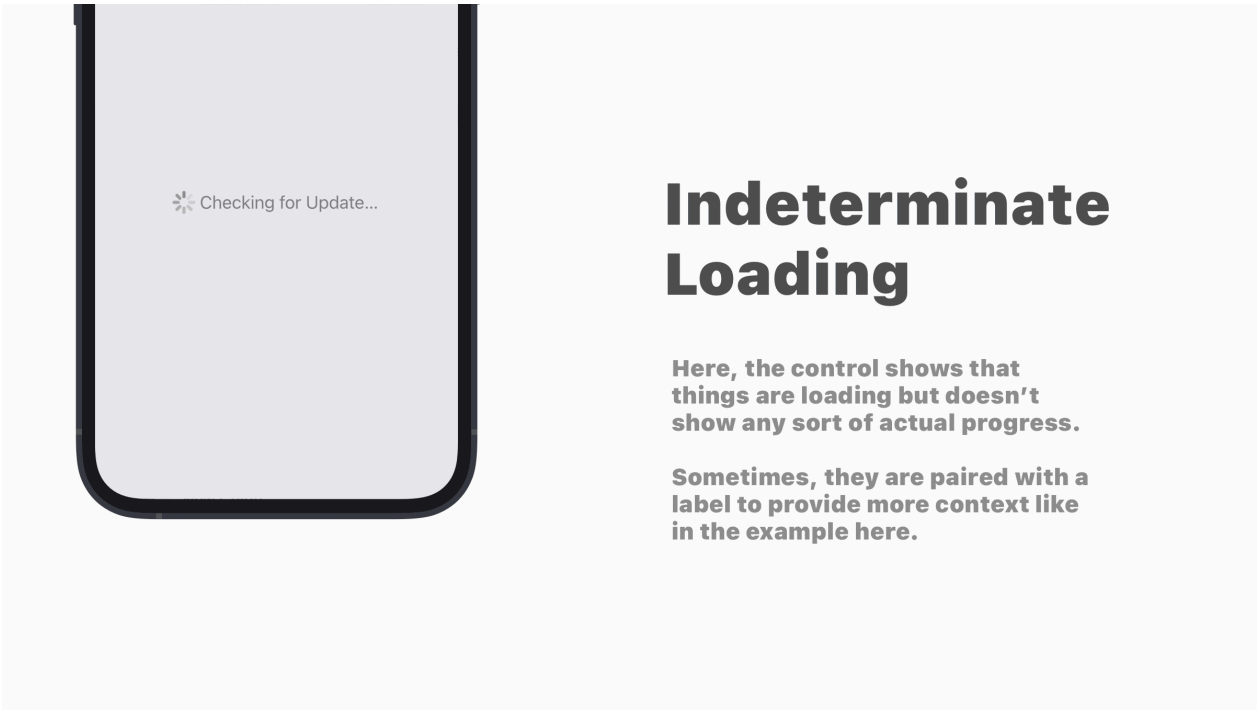
2. It should be **sensible**. What I mean by this is that it's placed according to the task at hand. If you don't need a loading indicator front and center, then you might place it in some place like a tool bar. Conversely, if the data being loaded is critical to the app's experience, then the loading interface should likely be more prominent.
3. They are **truthful**. This means you don't show a loading indicator filled up 80% of the way if the task is only 10% done. This is frustrating to people, and gives the wrong impression. While it's not always possible to provide an airtight representation of how much of the loading process is done, and how much time is left – you should try to get as close as possible.

When loading in data, there are two primary ways to indicate the state of things. You can use, preferably, **determinant** loading indicators. These types of loading indicators show how much loading has been completed, and how much is left:



This route is preferred because it gives people more information. They know how much is done and how much there is to go – there's really no blackbox. If the loading session is going to take longer than 10 seconds, then this method is really the only way you should go. Otherwise, the app feels broken or frozen. Moreover, it's frustrating to be waiting a long time, and on top of that – having no idea how much time you've got left to wait is even worse.

Second, there are **indeterminant** loading indicators. These simply convey that something is loading, but it doesn't provide any notion of how much of that loading is actually done, or how much is left to do. While these types of loading indicators aren't preferred for longer loading use cases, they still have many uses and are the most common types of loading controls you'll see throughout iOS:

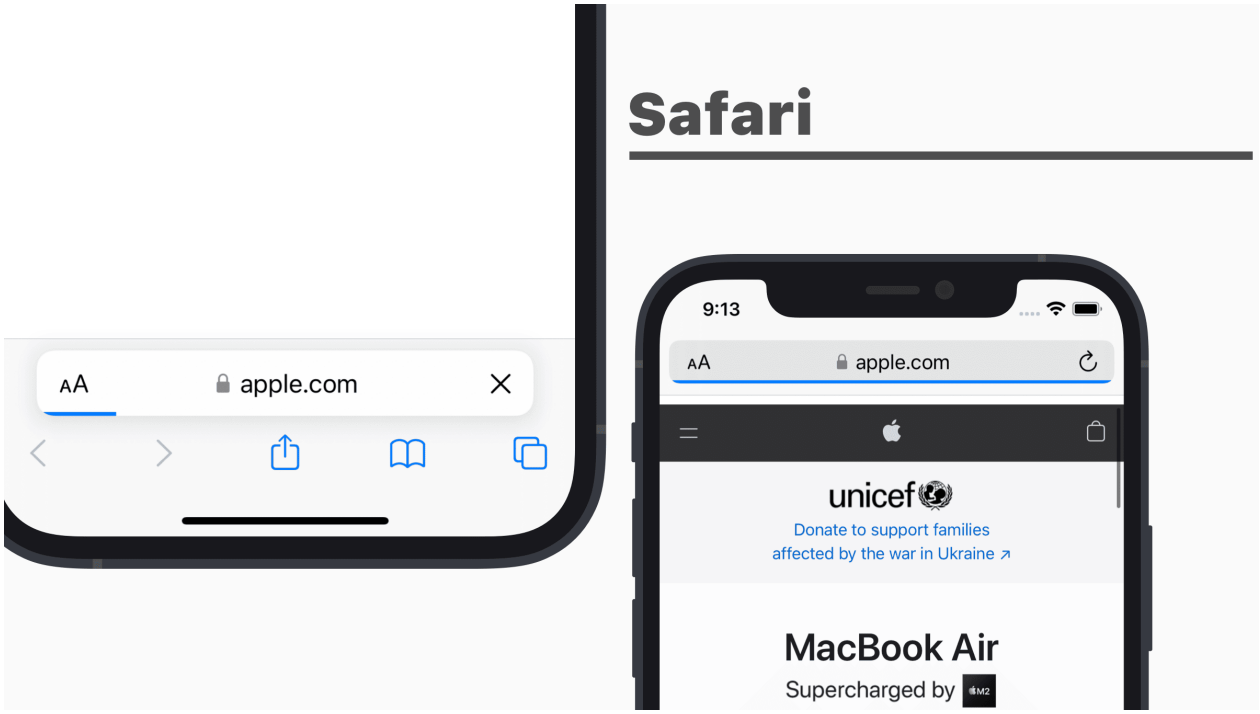


Loading Controls

No matter which route you take, there are controls that both UIKit and SwiftUI have to get the job done.

Progress Bars

The progress bar is used for determinate loading situations. You can see it in places like Safari's address bar:



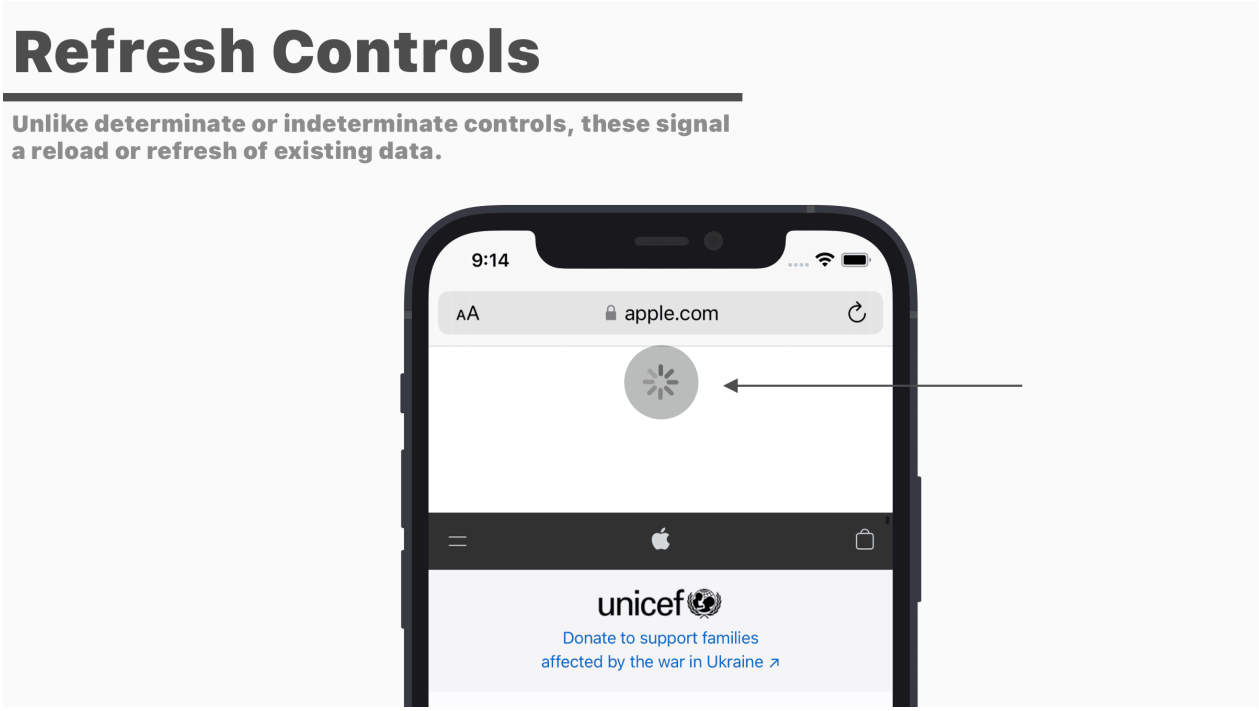
You'll want to use a progress bar when you can accurately measure loading. If you know how much work you've done and how much is left, it's hard to argue against a progress bar.

However, they can feel heavy and somewhat out of place if they are used for quick loading events. When people see a progress bar, they reasonably expect a loading

process of at least a few seconds, or — it could scale up to something much longer, like an hour for a large backup of photos.

Refresh Control

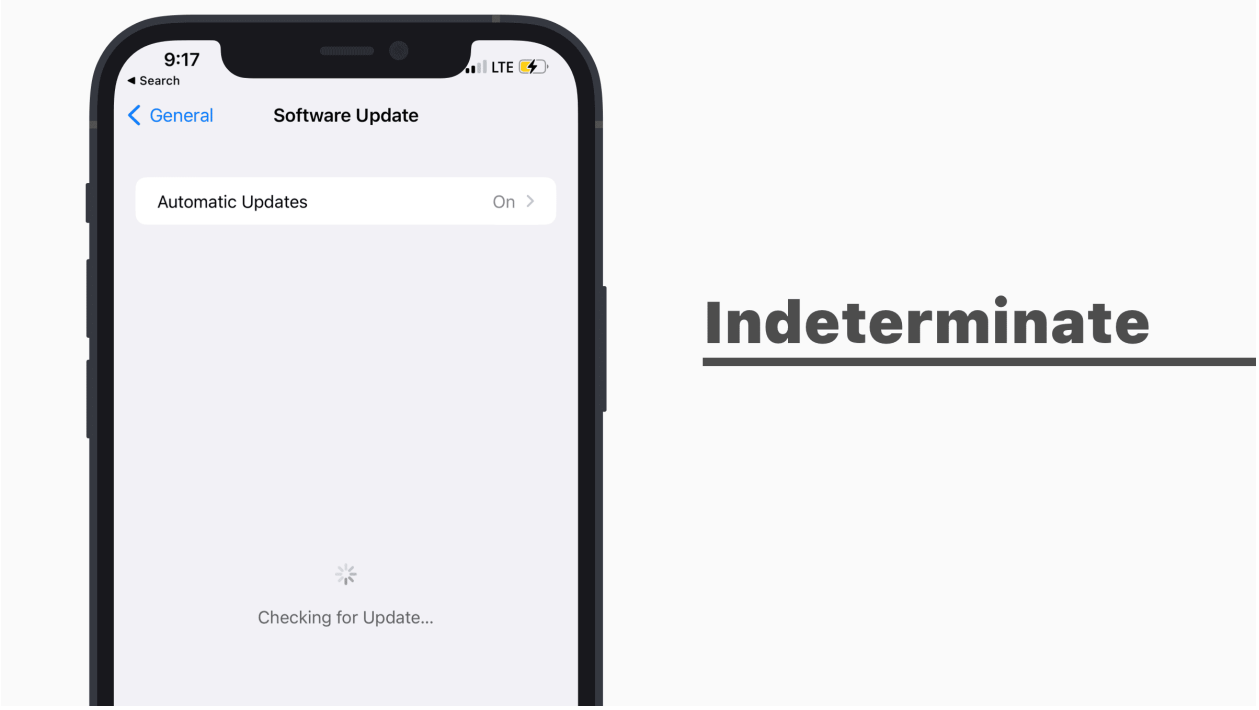
Refresh controls are a form of specific loading — namely, refreshing content that is already there. Refresh controls are used by "pulling down" some sort of interface in your app, most commonly — something like a table view. Again, if we turn to Safari, we can see this form of loading when a person refreshes a page:



Refresh controls are a known quantity on iOS, making them easy to slot into many designs. If you have existing data that can change, lean on the refresh control. But, to be clear — this isn't meant for loading in *initial* data, this is something that requires a user's action to perform. This is exclusively for updating what's already there.

Activity Indicators

Activity indicators are used for indeterminate loading situations. For example, when iOS checks for an update, you'll see an activity indicator used:



Often times, activity indicators are accompanied by text to remove any ambiguity around what is being loaded. If the context is crystal clear, you can forgo including text. Indeterminate loading is the most common form of loading, but if things start to run long and you can't use determinate loading patterns – consider including text on the fly to let the user know things are progressing as they should:



Custom Loading Controls or Experiences

For most of you reading this book, these controls are the way to go. You can always jazz up the loading experience if it's going to take a long time to load everything – but those types of situations are typically better suited to other app genres, such as games. For example, there may be a screen that cycles through games tips while a level loads – or a fun little mini game that's playable while a content update is downloaded.

Handling Failures and Empty States

In some cases, whenever you load data – there simply won't be any data that's returned. In such cases,

Tips

Determinate versus Indeterminate Loading

I briefly touched on this from the top, but when it comes to loading states – how do you decide which method to use? Determinate or indeterminate? While it's easy to say that determinate loading is better in many ways, namely – it's informative, that doesn't mean that it's always the correct choice.

Here's how I think about it: If the loading to be performed is only a few seconds long, indeterminate loading is likely the way to go. In fact, the informative nature of determinate loading can be distracting in these situations since they are often accompanied by text too. So, if your app opens and feeds to fetch some sort of feed or list – the "loading spinner" is understood and it's a fine choice.

Now, if the loading session is going to be longer – you should lean on determinate loading. Even if you can't get the accurate measurements for determinate loading, try to "chunk" your work so that you can create some form of aggregation.

For example, if you've got four sets of data to load but can't measure the bytes coming over the network, you can still use determinate loading because you at least know you've got four operations to complete.

In these cases, provide a label with your loading controls that updates as each chunk of work finishes, e.g. "2 of 4 workouts loaded" and so on while updating the progress bar the same way. One loaded workout loads up one quarter of its total progress, and you can animate this too to make it seem smooth and not a sudden, jerky update.

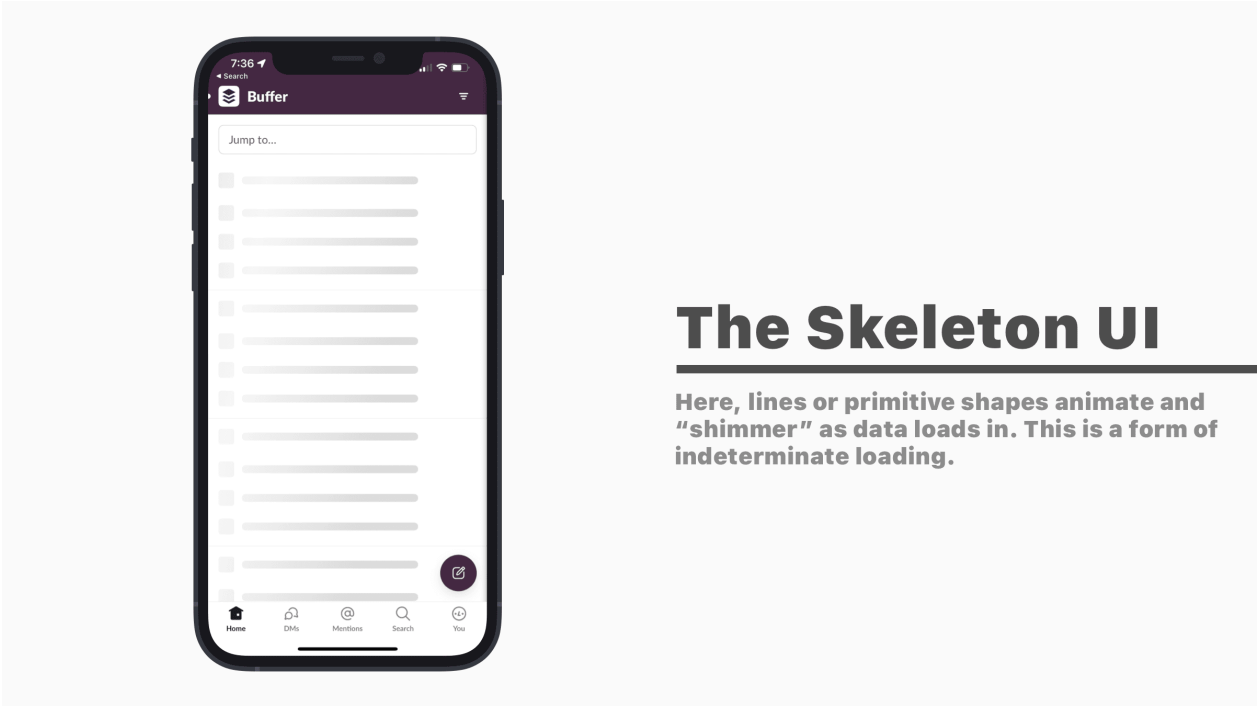
And remember, these are suggestions – they aren't rules. You'll see indeterminate loading in places where you think determinate should be, and vice-versa. For exam-

ple, Safari uses determinate loading in the address bar for web pages even though most web pages should load within a second or two. But, that's not always the case – sometimes the network is slow or congested, so this could take five to ten seconds (or more!), making determinate loading a great choice.

As with all things user experience related, it's about toying around with a few approaches while keeping conventions in mind. When you do that, you'll know when it's the right time to break them.

The "Skeleton UI" Approach

Several apps use the "skeleton UI" approach to loading data. Major apps such as Slack and Facebook rely on it today to express that loading is occurring, and when it's done those "loading skeletons" are replaced with the actual interface:



To be honest, I'd be careful with this one.

While it's easy to say that his pattern is understood at this point, it also comes with a few drawbacks. The biggest issue I have with this approach is that the skeleton interface doesn't represent the data that's actually loaded. For example, if you see four or five "blocks" with that shimmering effect but when the data is fetched there are only two things to show — well, the skeleton UI gives off a false impression. The heights of the actual interface are different, the numbers aren't the same and more.

You might read that and think, "Well, widgets on iOS do this. Apple loves this pattern!" Not so fast. Widgets on iOS have a similar design, but their placement promises the resulting interface, too. They always match one-to-one. Plus, they have more utility in that they mask sensitive data. So, while they may look similar, they aren't the same.



WidgetKit Loading Placeholders

While they may seem close to the "skeleton UI", they are not functionally the same. In addition to masking private data that may not be viewable, their shapes also promise what the interface will end up looking like . They match nearly 1:1.

Further, I've seen a few folks who thought that this type of interface was interactive. As someone reading this book, you likely know that's not true — but we don't just make apps for us, most of us make apps for everyone. The traditional approach of loading

controls on iOS don't have these drawbacks because they don't claim to represent the resulting interface at all. A spinning circle doesn't promise any sort of interface, only that things are being fetched. I like the delineation.

In short, it's an opinionated approach. And after considering it in full and you still feel it's right for your apps – by all means, go for it.

Indicate if Loading is Long Winded

If your loading operation is lasting more than about five seconds, let people know things are moving along. If you don't, it might seem like your app is stalled. Even if you don't expect that too happen, plan for it if it does:

```
private func kickoffLoad() {
loader.begin()

DispatchQueue.main.asyncAfter(deadline: .now() + 5.0) { [weak self]
in
        guard loader.isLoading else { return }
        self?.loadingText.isHidden = false
    }
}
```

Long loading operations that give no indication of their progress can be frustrating, but even if you can't use determinate loading here – a simple "2 of 6 Workouts" for a workout app downloading new content goes a long way.

Use Human Language

At several points throughout this book series, I've reiterated the idea that you should write copy for humans, and not programmers. A determinate loading label paired with confusing terminology hinders as much as it helps.

Reading out "860 kilobytes fetched of 23,434 kilobytes" isn't understandable to a lot of people. Aside from using industry lingo, ensure you format things correctly too. I've seen a lot of apps say something like "3 items left", and when they get to the last one – it still reads "1 items left" and that doesn't exactly ooze confidence. Use iOS APIs and Foundation's many formatters to combat this so you correctly read out "1 item left" or "3 mb", which are more widely understood.

Leverage Background Updates

The best loading experience is honestly the one that doesn't exist.

Loading doesn't have to be forgone conclusion. In fact, iOS even encourages this with a robust set of APIs to load data in the background. While your app isn't in the foreground, be smart about grabbing updates for your app so people can come back to just that – your app, and not a loading spinner.

Modeling Progress in Code

If you need someway to inform your loading indicators how much of the "work" behind loading is done, look no further than `Progress`. You can use one or more of these to aggregate loading operations together. In conjunction with iOS' loading indicators, you'll have all you need for effective determinate loading interfaces.

In fact, SwiftUI has a built in control that takes in a `Progress` object and automatically updates according to its progress and even adds text label feedback:

```
// Xcode -> SwiftUI -> LoadingStatesFig1View.swift

class Loader: ObservableObject {
    private var subs: [AnyCancellable] = []
    var work: Progress = Progress(totalUnitCount: 100)

    func start() {
```

```
        subs.forEach { $0.cancel() }
        subs.removeAll()
        Timer.publish(every: 0.10, on: .main, in: .default)
            .autoconnect()
            .sink { [weak self] tick in
                let current = self?.work.completedUnitCount ?? 0
                self?.work.completedUnitCount = current + 1
        }
        .store(in: &subs)
    }
}

struct LoadingStatesFig1View: View {
    @StateObject private var loader = Loader()

    var body: some View {
        VStack(spacing: 8.0) {
            Text("Downloading Workouts")
            ProgressView(loader.work)
        }
        .padding()
        .onAppear {
            loader.start()
        }
    }
}
```

In UIKit, it's a little more work – but not much:

```
// Xcode -> UIKit -> LoadingStatesFig1ViewController.swift

class LoadingStatesFig1ViewController: BaseSampleViewController {
    private let progressBar =
UIProgressView(progressViewStyle: .bar)
    private let loader = Loader()

    override func viewDidLoad() {
        super.viewDidLoad()
        view.addSubview(progressBar)
```

```
        progressBar.translatesAutoresizingMaskIntoConstraints =
false
        progressBar.observedProgress = loader.work

        NSLayoutConstraint.activate([
            progressBar.centerYAnchor.constraint(equalTo:
view.centerYAnchor),
            progressBar.centerXAnchor.constraint(equalTo:
view.centerXAnchor),
            progressBar.widthAnchor.constraint(equalToConstant: 260)
        ])

        loader.start()
    }

}
```

Keep in mind, though, you'll have to manage any text feedback yourself, unlike the SwiftUI example. So, if you wanted that, you could use Combine to track things more finitely:

```
private var subs: [AnyCancellable] = []

// In initializer or viewDidLoad:
loader.work.publisher(for: \.completedUnitCount).sink
{ completedWork in
    // Update UI with completedWork
}
.store(in: &subs)
```

Further, you can cancel and pause them. They also have support to observe several lifecycle events such as cancellation and completion in addition to the current progress it's completed.

Three Key Takeaways

1. Know the difference between loading user experiences available to you, namely – determinate and indeterminate loading.
2. Whichever one you use, try to follow the three goals of effective loading experiences so that it's informational, sensible and truthful.
3. Try to leverage iOS' suite of loading controls instead of building your own – which is more suited to situations like games.

State Restoration

Since the birth of the iPhone, Apple has had to handle an especially tricky scenario – how do they allow people to use several apps all at once? People close one, open another and switch to something new all within seconds. Memory was tight, and the available resources that apps could potentially take up could crash the operating system itself (which led to the creation of jetsam, which will eject your app if it needs to).

Then, the iPad came along – and the needs only became more complex not only due to the larger screen real estate and the new ways people could use apps, but also the release of multiple windows on iPadOS 13. To help solve all of these problems, iOS has the concept of state restoration – and you can think of it basically like this: As a developer, you can supply the operating system with a bit of information to load things up the same way they were when someone last left off.

If you had a workout app, then when the app comes back into the foreground – their last selected workout is still selected. Or, more generally, maybe the last view they opened is still represented. I like to think high level here, you don't have to go so deep that you restore, for example, the same scroll view offset for a scroll view when it's restored – instead think in terms of the overall experience. If users can continue the last task they were in the middle of doing, then you're in a good spot. To me, that's effective state restoration.

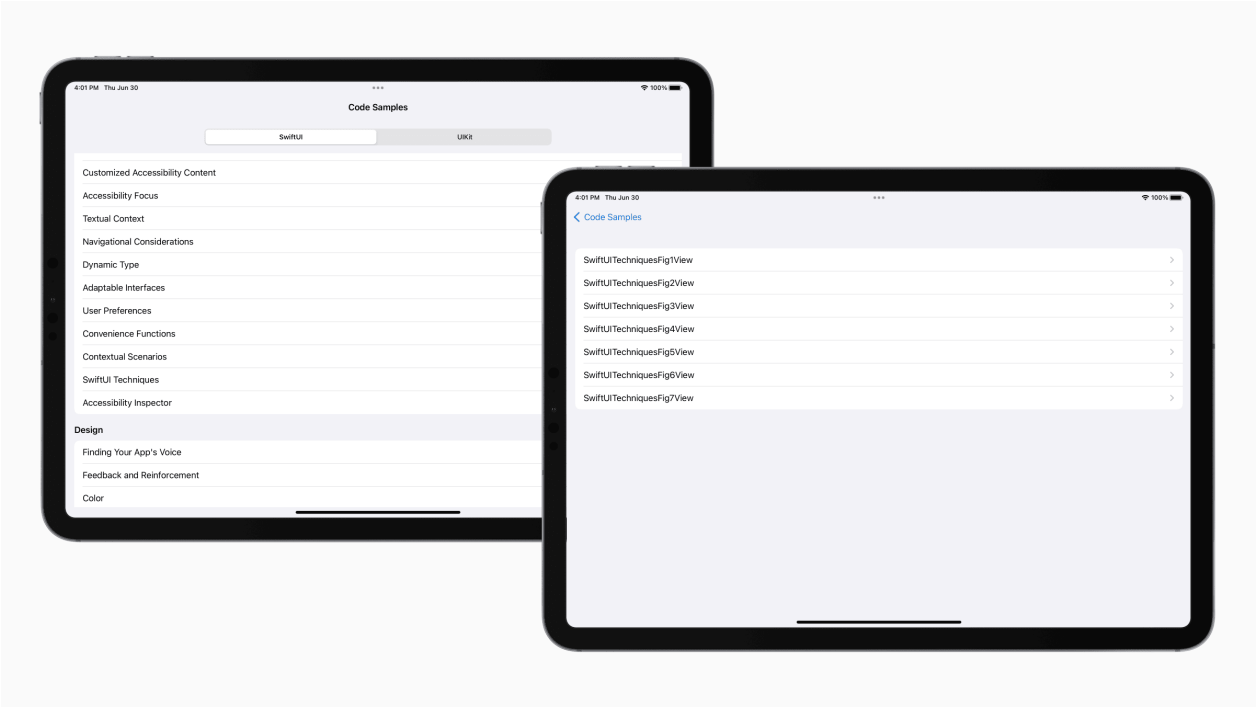
From a technical standpoint, getting started with state restoration used to be a little tricky because there was (until very recently) a lot of things you had to do as a developer to restore state. Today, the approaches vary whether you're using SwiftUI or UIKit but both options have been streamlined. As such, we will not be covering the classic view controller based approach to state restoration that used pre-iOS 12. Apple has formally deprecated this approach, so if you're using it – now is a good time to move to the techniques we'll discuss here.

No matter the interface framework you choose, both of them will deal with `NSUserActivity` to store information about how to restore state. If you haven't used this object before, you *will* eventually no matter the type of app you make. It does a lot in iOS, such as supporting Spotlight search, handoff, universal links, quick notes and more.

The idea centers around storing *just enough* information (such as an identifier) within a user activity to later restore your app's state. As the user moves around the app, you create activities to reflect the state the app is currently in. For example, in a movie list app – you'd track the most recently viewed movie.

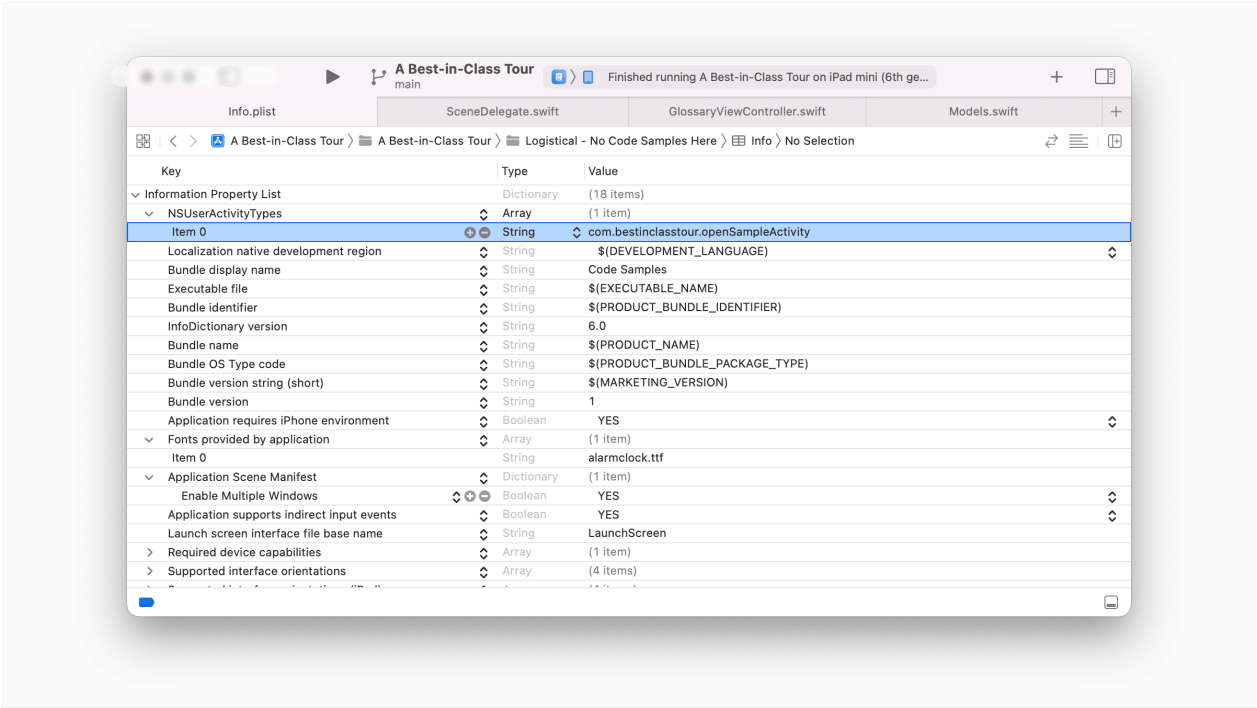
To get started, let's look at how we could implement lightweight state restoration within the sample code app included with the book series. Our goal is to restore the most recently viewed chapter, and the interface choice. For example, if we had SwiftUI

selected and chose the “SwiftUI Technique” chapter, we’d want the app to open up to the screen on the right, *not* the primary one on the left:



UIKit

First, we’ll need to add an activity in our .plist file for every one we’d like to support. This is a simple string that follows the reverse domain pattern you likely use for things like bundle identifiers:



Here, I've called ours `com.bestinclasstour.openSampleActivity` – this also will correlate to our user activity later on via its `activityType` property.

Next, we'll setup an extension for a view controller to create an activity that will hold two pieces of information:

1. The currently selected code sample.
2. The interface choice, i.e. SwiftUI or UIKit.

```
// Xcode -> UIKit -> ActivityVendor.swift

let BookChapterActivityPListString : String =
"com.bestinclasstour.openSampleActivity"

enum MainActivityKeys : String {
    case SelectedChapter = "SelectedChapter"
    case SelectedInterface = "SelectedInterface"
}
```

```
extension UIViewController {
    func setSceneUserActivityFor(bookChapter: BookChapter,
interface: InterfaceFramework) {
        var sceneActivity = view.window?.windowScene?.userActivity
        if sceneActivity == nil {
            sceneActivity = NSUserActivity(activityType:
BookChapterActivityPListString)
        }
        sceneActivity?.title = bookChapter.rawValue

        // Track the selected book chapter
        sceneActivity?.addUserInfoEntries(from:
[MainActivityKeys.SelectedChapter.rawValue: bookChapter.rawValue])

        // Track the interface chosen
        sceneActivity?.addUserInfoEntries(from:
[MainActivityKeys.SelectedInterface.rawValue: interface.rawValue])
        view.window?.windowScene?.userActivity = sceneActivity
    }
}
```

Now, we're at the point where we would ask ourselves "When should we restore state as people use our app?"

For us, a good spot would be when someone taps on a book chapter to view code samples for. So, we'll create the activity when someone taps on a row to view one:

```
// Xcode -> UIKit -> GlossaryViewController.swift

func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
    guard let chapter = datasource.itemIdentifier(for: indexPath)
else {
        tableView.deselectRow(at: indexPath, animated: true)
        print("Could not find a chapter for the selected row.")
        return
    }
```

```
    // Update our user activity
    setSceneUserActivityFor(bookChapter: chapter, interface:
interfaceChoice)

    // Present it
    pushBookChapter(chapter: chapter, with: interfaceChoice)
}
```

Now, we've got an activity registered and we're creating it. To continue, we need to opt-in to state restoration at the scene level:

```
// Xcode -> UIKit -> SceneDelegate.swift

// Opt-in to state restoration
func stateRestorationActivity(for scene: UIScene) -> NSUserActivity?
{
    return scene.userActivity
}

func sceneDidBecomeActive(_ scene: UIScene) {
    if let userActivity = window?.windowScene?.userActivity {
        userActivity.becomeCurrent()
    }
}
```

And that brings us to the very last step, which is to use that information we stashed away in the `userInfo` dictionary to restore state. This happens when a new scene connects:

```
// Xcode -> UIKit -> SceneDelegate.swift

func scene(_ scene: UIScene, willConnectTo session: UISceneSession,
options connectionOptions: UIScene.ConnectionOptions) {
    guard let windowScene = (scene as? UIWindowScene) else
{ return }
```

```
    let rootController = GlossaryViewController()
    let rootNavController =
UINavigationController(rootViewController: rootController)
    window = UIWindow(windowScene: windowScene)
    window?.rootViewController = rootNavController
    window?.makeKeyAndVisible()

    // Scene restoration
    if let previousActivity = session.stateRestorationActivity,
        let userInfo: [AnyHashable:Any] = previousActivity.userInfo,
        previousActivity.activityType ==
BookChapterActivityPListString,
        let chapter = BookChapter(rawValue:
userInfo[MainActivityKeys.SelectedChapter.rawValue] as? String ??
""),
        let interfaceChoice = InterfaceFramework(rawValue:
userInfo[MainActivityKeys.SelectedInterface.rawValue] as? Int ?? 0)
{
        // Open up back to the last viewed chapter
        rootController.pushBookChapter(chapter: chapter, with:
interfaceChoice)
    }
}
```

And we're done. Now, when the app is restored – if the user had selected a book chapter, we'll automatically open that back up.

Let's recap what we just did to get it all hooked up. Doing it isn't hard, but it is one of those things where there are a few steps that are easy to miss.

End-to-end, the entire process looked like this:

1. In your app's property list, add in array type named "NSUserActivityTypes", and then add in a string for each activity type you want to support. To simplify your thinking here, usually you correlate an activity to any scenes you can create in their own window.

2. Next, create the activity as people use your app to keep track of the last thing or place they were at or viewing.
3. Opt-in to state restoration in your scene delegate by returning a scene user activity via `func stateRestorationActivity(for scene: UIScene) -> NSUserActivity?`.
4. Then, when your app is restored, you can pull this activity out in `scene(_:willConnectTo:options:)` and set things up according to the data you previously stored in it. You'll find it in `session.stateRestorationActivity`.

If that sounds a little simplistic for something as "advanced" sounding like state restoration, you're right – it is simple. In fact, it's a lot easier from a technical perspective than the classic approach Apple used to encourage with its view controller based APIs.

However, one possible reason this was rethought was the advent of multiple windows. Following our movie app example – if the user had three different windows showing three different movies, than each one should restore the correct movie. The user activity approach is well suited for things like that.

SwiftUI

For SwiftUI using scene-based navigation (which includes virtually all SwiftUI apps created within a few years) – you'll follow much of the same pattern as we did in UIKit. iOS still requires you register an activity in the .plist and then you vend it via a modifier. Lastly, you check for it and navigate to a view accordingly.

In SwiftUI, the process of creating the activity and using it is a two-step process. I should note, the sample project is created using a UIKit, scene-based approach. As such, it can't demonstrate how to perform state restoration using SwiftUI in the simulator or a device, but I've written it all out as if it were. Luckily, the code is easy to follow.

If the sample project *were* created in SwiftUI, first we'd vend the user activity for the selected book chapter:

```
// Xcode -> SwiftUI -> StateRestorationFig1View.swift

struct StateRestorationFig1View: View {
    @State private var book: BookChapter = .StateRestoration

    var body: some View {
        Text("All about \(book.rawValue)")
            .userActivity(BookChapterActivityPListString)
{ userActivity in
                userActivity.addUserInfoEntries(from:
[MainActivityKeys.SelectedChapter.rawValue: book.rawValue])
                userActivity.addUserInfoEntries(from:
[MainActivityKeys.SelectedInterface.rawValue:
InterfaceFramework.SwiftUISample.rawValue])
            }
    }
}
```

Here, using the `.userActivity` modifier, we are doing essentially what we did in UIKit. We add in the metadata we need in the user activity, in this case the book and, well – SwiftUI for the interface framework, since it'll always be SwiftUI in our example.

Next, when the app goes to restore its state, we'd intercept that user activity via `func onContinueUserActivity(_ activityType: String, perform action: @escaping (NSUserActivity) -> ()) -> some View`, and that will happen wherever you define a `WindowGroup`:

```
// Xcode -> SwiftUI -> StateRestorationFig1View.swift

struct MockSwiftUIApp: App {
    var body: some Scene {
        WindowGroup {
            StateRestorationFig1View()
                .onContinueUserActivity(BookChapterActivityPListStri
ng) { userActivity in
                    guard let userInfo: [AnyHashable:Any] =
userActivity.userInfo,
                          let chapter = BookChapter(rawValue:
userInfo[MainActivityKeys.SelectedChapter.rawValue] as? String ??
""),
                          let interfaceChoice =
InterfaceFramework(rawValue:
userInfo[MainActivityKeys.SelectedInterface.rawValue] as? Int ?? 0)
else {
                        return
                    }

                    Swift.debugPrint("Restore \(chapter) using \
(interfaceChoice)")
                }
        }
    }
}
```

Again, this is very similar to UIKit. We get an activity and we pull out the data we need, and then navigate to it.

SwiftUI also has some other tricks up its sleeve outside of `NSUserActivity` by way of a special property wrapper, `@SceneStorage`. You can store anything you can put into a .plist here, and it's a very SwiftUI-centric way to restore things on a per-scene basis. In fact, it's much easier than dealing with user activities.

You can use it by and large the same way that you would use `@State`, only now when the scene restores, so will the value that was set:

```
// Xcode -> SwiftUI -> StateRestorationFig2View.swift

enum Difficulty: String, CaseIterable {
    case Easy
    case Medium
    case Hard

    func color() -> Color {
        switch self {
        case .Easy: return .blue
        case .Medium: return .yellow
        case .Hard: return .red
        }
    }
}

struct StateRestorationFig2View: View {
    private let levels: [Difficulty] = Difficulty.allCases
    @SceneStorage("GameView.difficulty") private var
selectedDifficultyLevel: Difficulty =
.Easy

    var body: some View {
        VStack {
            Text("Select Difficulty")
                .font(.title)
                .bold()
                .foregroundColor(selectedDifficultyLevel.color())
            Picker("Choose Difficulty Level", selection:
$selectedDifficultyLevel) {
                ForEach(levels, id: \.self) { level in
                    Text(level.rawValue)
                        .font(.body.bold())
                }
            }
            .pickerStyle(.segmented)

        }
        .padding()
```

```
    }
}
```

One thing to note here is how we named our scene storage. While each key does need to be unique, scene storage is always local to the app. Because of that, you don't need to add in the app's bundle identifier or anything else to disambiguate it.

Tips

Determine Activity Types

While feeling out scene restoration, you probably will wonder how many activity types you'll need. To answer that, ask yourself two questions:

1. Do I have multiple scene delegates? Then, I'll want one activity for each.
2. Do I have drastically different flows throughout your app in terms of content someone might view? Then, it *might* make sense to make an activity for each of those, too.

Most of the time, sticking to one user activity per scene is more than good enough. You only should consider point number two above if you find yourself sticking way too much into the activity's `userInfo` to where it becomes unwieldy to use.

Keep in mind, though, if you do have multiple different scenes that can be used for their own window – how you get the user activity out is different. Consider:

```
guard let userActivity = connectionOptions.userActivities.first ??
session.stateRestorationActivity else { return }
```

This is the code you'd use during scene session connection if you had multiple scene delegates being used. Why? Because when you open a new scene type (i.e. a profile view scene configuration in a social media app), you'll find that activity in `connectionOptions.userActivities`, whereas if you don't – it'll be found in `session.stateRestorationActivity`.

Simplify State Restoration

When thinking about state restoration, remember to think small. You don't want to put entire model objects into user activities – and, well, if you do, your app will crash. Remember that the dictionary for user activities should only contain primitive types such as numbers, strings, boolean, arrays, data, dictionaries, sets or a URL.

That's why, for most use cases, including an identifier to something is more than enough. In the UIKit example, if we would've gone this route:

```
userActivity.addUserInfoEntries(from:
[MainActivityKeys.SelectedChapter.rawValue: book])`
```

The app would've crashed, full stop. That's because the `Book` type is an enum I created for the sample project, meaning we need a way to tie some data back to one. The raw value of `Book`, which is just a String type, is perfect for this. That's why we store it this way:

```
userActivity.addUserInfoEntries(from:
[MainActivityKeys.SelectedChapter.rawValue: book.rawValue])`
```

Then later, we're able to create the book using its initializer accepting a String.

How to Test State Restoration

You can test state restoration in the simulator, but it may not work how you might expect. Force quitting the app seems like a practical solution, but in this case – it won't work. The system deletes preserved state when an app is force quit, which is actually a very good thing. On device, the same thing occurs if your app crashes at launch. These are both good things, because if you had a crashing bug tied to state restoration and the system didn't clear it out, well – then your app would be stuck in a perpetual crash loop. Not good.

So, to test things – simply make sure your user activity code fires and then return to the Home Screen. Then, if you're on the simulator, stop debugging and re-run the project. Your code in the scene delegate should fire now, and the same is true of SwiftUI's modifier approach.

Request Restoration Extension

If you know that scene restoration might take more than a second or two, you can request an extension:

```
func scene(_ scene: UIScene, restoreInteractionStateWith
stateRestorationActivity: NSUserActivity) {
    scene.extendStateRestoration()

    // Do work that might take a bit

    scene.completeStateRestoration()
}
```

Tie into `func scene(_ scene: UIScene, restoreInteractionStateWith stateRestorationActivity: NSUserActivity)` to use this technique. While you're doing the work, your launch image will stay showing and the main run loop will

also still execute. Just be sure to signal when you're done via `scene.completeStateRestoration()`.

Typed Activities in SwiftUI

While I mentioned at the top of this chapter that you *shouldn't* store models in user activities, allow me to introduce an exception to the rule. In SwiftUI, you can use user activities "typed payload" functions to do just that. If your model is already adopting the `Codable` protocol, you can greatly simplify things and improve semantics around the whole process by using your model instead of diving through a dictionary:

```
// When saving your activity
try? userActivity.setTypedPayload(book)

// When pulling it back out
try? userActivity.typedPayload(Book.self)
```

When you use this technique, the system automatically keys the `userInfo` dictionary to match the coding keys from your model. Then, it's able to reconstruct it from this information. If you're using SwiftUI and user activity, this is a great approach to take and the one I'd recommend.

Scene Storage Considerations

This may be an outlier tip, but you need to keep scene storage light. If you use this property wrapper and load it up with large models or data, you could hit performance issues. Further, the system manages this storage and you don't have much say in how it handles the data you give it. Given that, storing sensitive information in it is also a bad idea. Keep it simple with scene storage – restore selected tabs, picker values and other similar things.

Three Key Takeaways

1. Support state restoration such that users can continue what they were doing, but it keep it simple.
2. State restoration is especially important on iPadOS, where multiple windows are a key part of people's workflow.
3. If you're using the `UIStateRestoration` protocol, now is the time to switch to the modern approach.

Custom View Controller Transitions

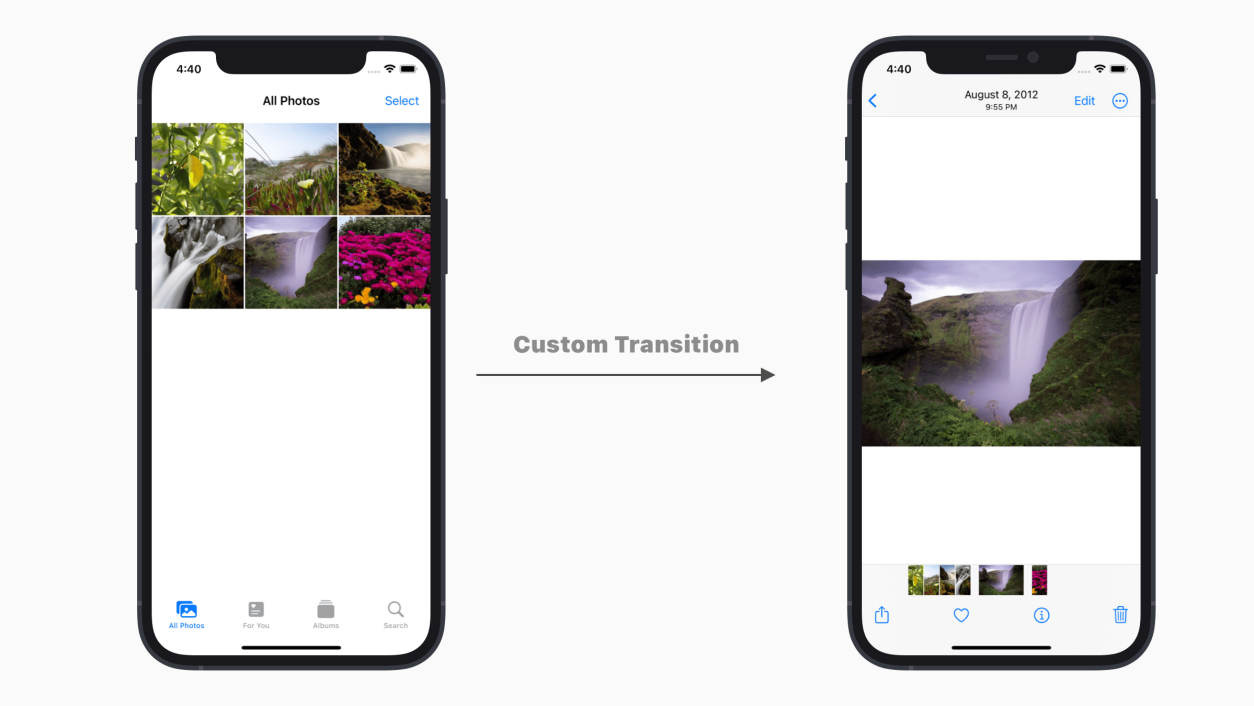
Custom view controllers add that *jena se qua* to any app. They just do. When done correctly, they delight and surprise people. When they are gratuitous or implemented unnecessarily, they get in the way. This chapter goes into what makes a good transition, when you might look to use them and how to implement one. It also dives into the *how* quite a bit, because in this case – the "how" is a bit of sticker.

> *Please note, this chapter focuses solely on UIKit – this A.P.I. is specific to view controller paradigms.*

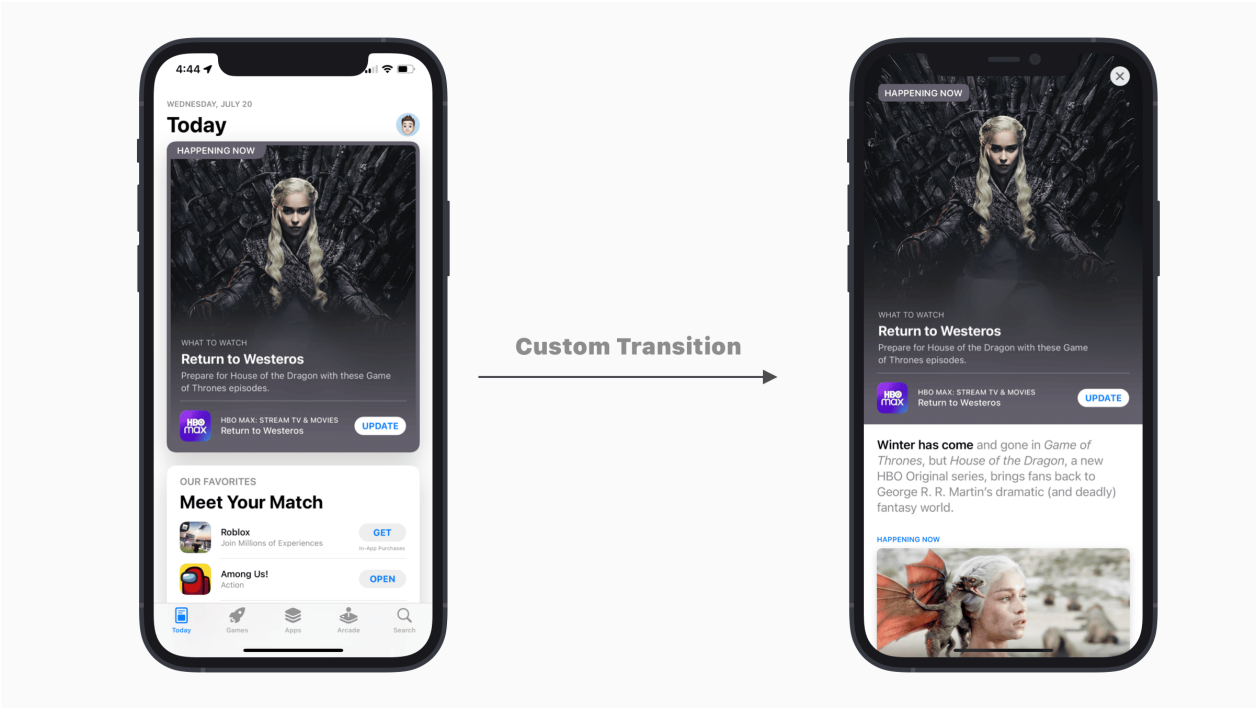
To kick off our discussion, what do I mean by a custom view controller transition? Within UIKit, anytime you present a view controller from another view controller – a view controller transition is occurring. For the most part, these are modal in nature – though that's not always the case when you're using a navigation controller. In that

scenario, you *push* a view controller onto the navigation stack, and when you go back, you *pop* it off.

For our case, we're focused on those modal transitions. To see a custom view controller transition, you don't have to look far within iOS. Perhaps one of the best examples of them can be found in the Photos app, which has several custom transitions it uses. For example, anytime you tap on a photo to view it, and it expands from where it was to where it needs to go – that's a custom transition Apple built:



If you visit the App Store app and tap on anything in the "Today" view, again – that's a custom transition:



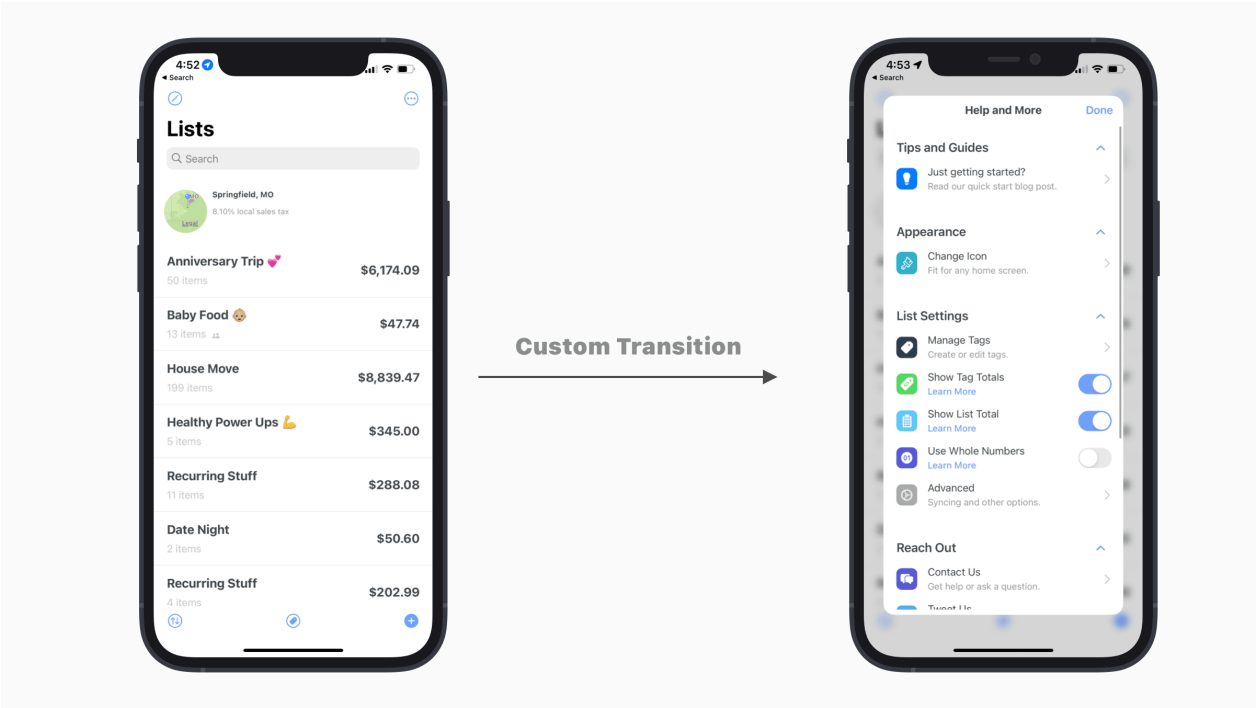
Further, a lot of these transitions can be interactive, meaning you can control their progress with a gesture, pause them while they are occurring or cancel them altogether. Many transitions are start and stop in nature, meaning that a user intends to view something, the system kicks off the transitions and then it finishes. But others, like the firstPhotos example, are interruptible and interactive.

While this does add complexity in terms of implementation, there's no denying that it simply *feels* better as a user to know that you're in control of the process. I'm sure that, like me, you've tapped on a photo within the Photos app that you wanted to view but accidentally tapped on the wrong one. Maybe you hit the one next to it, above it — whatever. With their interactive transition, you can simply swipe it back down before it even finishes to "cancel" the transition. You don't have to wait for it to finish — and if you did, those moments that you would spend waiting for something to complete that was an accident anyways would really add up.

Custom transitions work best when people don't notice them, but at the same they are also delighted by them. That statement sounds like a contradiction, so allow me to dig into it a little more because I really believe it encompasses the heart of custom view controller transitions. In my last app, Spend Stack, I have the ubiquitous "Settings" view. You could open it by tapping this button in the top right:



What would you expect to happen? A sheet transition, probably. That's what occurs in almost any other app. It's expected to be modal, after all. Instead, I opted to show a centered modal that expanded from the middle of the presenting view. Below, there was a blur view obscuring the presenting controller's content:



Why do I think it worked?

Because it was a fun, little transition that didn't get in the way. It still served the purpose of a modal transition. It was the same duration as UIKit transitions, so it didn't feel any longer because it simply *wasn't* any longer. People probably expected the stock transition, but they got a different one that was just as utilitarian as it was playful. That's the sweet spot I think you can hit with custom transitions.

How Transitions Work

UIKit offers several ways to present your view controllers out of the box, and most of them are achieved by simply assigning a style to their `modalPresentationStyle` property:

```
myViewController.modalPresentationStyle == .fullScreen
```

There are several styles you can choose from, which require no customization at all:

1. `automatic`: The system will choose the most appropriate presentation style for you.
2. `currentContext`: The view controller will be presented over another view controller's content whose `definesPresentationContext` property is set to `true`.
3. `none`: A style indicating that no other adaptions should occur.
4. `fullScreen`: This style will cover the entire screen.
5. `pageSheet`: Likely the style you are most used to, this is the classic "card" presentation, where it covers *most* of the presented view controller but not all of it. You'll also get the rounded top edges on the presented view.
6. `formSheet`: A style centering the contents in the middle of the screen. This style is the same as a page sheet presentation unless you're in a regular horizontal and vertical size class (i.e. iPads who aren't sharing the window with another app via multitasking). You can also set the size of the presented controller by overriding `preferredContentSize`.
7. `overFullScreen`: Like the full screen presentation, except this style doesn't remove the views it's being presented over. This is a good choice when you want content underneath to still be visible for stylistic or user experience purposes.
8. `.custom`: This indicates you've got custom objects in place to create a custom view controller transition. We'll be looking more at this in a bit.

If you open the sample project and visit `CVCTFig1View.swift`, you can demo all of these yourself to see how they work.

In addition to the modal presentation style, there's also a *transition* style view controllers can use. These let you perform different effects during the transition, such as a cross dissolve or the classic "page turn" animation which is perfect for reading apps. If the modal presentation style is telling the system where to begin and where it should end up, the transition style tells the system if it should animate any certain way during the process of getting there. One places a view controller somewhere, one makes some certain effects or animations during the process.

As such, many of these values can work in tandem with a view controller's modal presentation style too, like the partial page curl – which itself requires a full screen modal presentation style to use at all. Again, try them all out to get a feel for them. I've got them all setup in `CVCTFig2View.swift` to use. For example, if you want a page sheet transition, but with a cross dissolve transition style – you can certainly do so.

But in the end – if you want to have full control over all of these things, we need to create a custom view controller transition. So, let's dive in there.

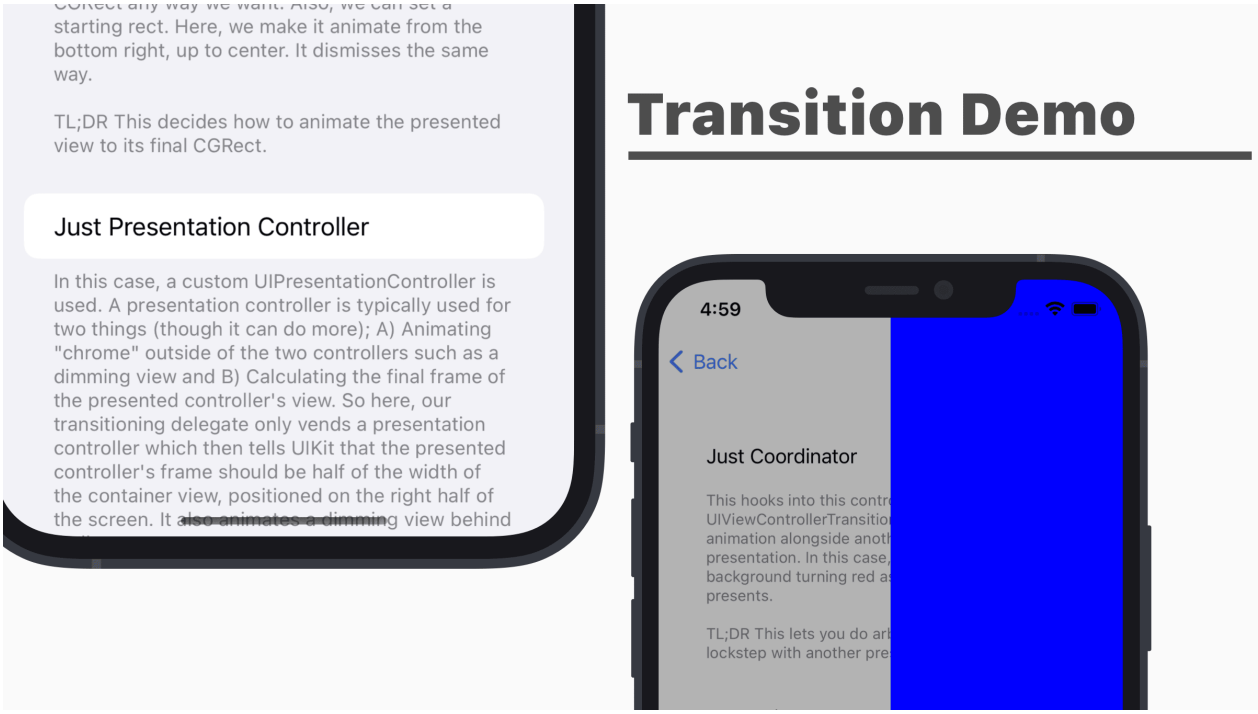
How Custom Transitions Work

To understand custom view controller transitions sometimes feels akin to learning black magic. It's a little hard and certainly confusing at first but *awesome* once it all clicks. So, that said, please give yourself a little grace here and realize that if this is new to you – you might need to read this section a few times until it all becomes clear.

To wit, I've read many documents explaining how transitions work. While many of them are full of incredible information, there's no denying that there's simply *a lot* of information, classes, protocols and functions to grasp. My goal is to take a different approach – I'll lay out just what you need to know. In our case, that's a few key objects

that drive these transitions. Once you've got a handle on those, you'll know where to hook into your code to achieve the effects that you're after.

For context, in the demo project I've included *a lot* of sample code with some in-depth documentation built right into the user interface. After you've read this chapter, you can open it up and play around with it to strengthen your knowledge of how all of this works together:



I've adapted code from an open source project I run that shows these concepts, and it's great for going through each part piece by piece. So, please use it! Custom view controller transitions are definitely a piece of iOS that you've got to get hands on with. Reading about it will help you get a feel over how things are working, but looking at this code (along with its explanations) is just as much required to learn the API.

The transition that we built (in most cases) in the demo code is simply showing a view controller that takes up half the screen, and presents from the bottom right of the

view. This isn't a very practical transition, but it gives an opportunity to mess around with the frame of the transition, dimming views you might want to customize and basically gives us a reason to touch each piece of the custom transition API.

So with that, let's look at the objects we may want to make to create a custom transition.

The Transitioning Delegate (Required)

The transitioning delegate is the "brains" behind custom view controller transitions. It's important because it vends a lot of objects to UIKit which drive the whole process. As such, you typically need a strong reference to this. There are a lot of ways to engineer this whole process, but for me – I typically put this in the presenting view controller.

```
private let myCustomTransitioningDelegate = CustomTransition-
Delegate()
```

This object will conform to the `UIViewControllerTransitioningDelegate` protocol, and that'll provide UIKit an opportunity to use custom transitioning objects you make. With it, you can vend:

1. Custom animation controllers, which we'll look at below. Those conform to `UIViewControllerAnimatedTransitioning`. These drive any custom animations you want to make to the presented controller.
2. Interactive animation controllers. These can work with the animation controllers, but here the user is driving the animations via a gesture or some other means. As such, you provide the progress of the animation.
3. Presentation controllers. Most commonly, these can be used to control things on the side of the *presenting* controller. For example, if you all you

wanted to do was provide a "dimming" view during a transition, a presentation controller would be a great place to do that.

Most importantly, you set a custom transitioning delegate object to a view controller's `transitioningDelegate` property to let UIKit know you're providing a custom transition:

```
// Xcode -> UIKit -> CVCTFig3ViewController.swift

let vc = createDemoController()
vc.transitioningDelegate = customTransitionDelegate
vc.modalPresentationStyle = .overFullScreen
present(vc, animated: true, completion: nil)
```

An Animator, Interactive or Not (Optional)

An animator provided by a custom transitioning delegate vends any custom animations you want the presented view controller to perform. In our case, it's responsible for making the presented controller's view come up from the bottom right of the view. It adopts `UIViewControllerAnimatedTransitioning`. It also decides the duration of the custom transition.

What it does *not* do is decide where its final frame should end up, which is a common misconception with animation controllers. That's handled by a presentation controller, and eventually that information comes down to the animation controllers via `UIViewControllerContextTransitioning`.

If your head is starting to spin right now, just a take a beat and think of it like this: Animation controllers will make a snazzy animations during the presentation and tell UIKit how long it should take. That's it.

This all occurs in `func animateTransition(using transitionContext: UIViewControllerContextTransitioning)`, whereas we provide the length of the transition in via `func transitionDuration(using transitionContext: UIViewControllerContextTransitioning?) -> TimeInterval`.

The flow for implementing them usually looks like this:

1. Get the views and container view from the transitioning context.
2. Put them in their "starting" spot.
3. Use UIKit animation APIs to animate those frames however you want.

You can see all of these occurring in the sample code. Here's step one:

```
// Xcode -> UIKit -> TransitionAnimator.swift

// Views and controllers
let containerView = transitionContext.containerView
let fromVC = transitionContext.viewController(forKey: .from)
let toVC = transitionContext.viewController(forKey: .to)
let fromView = transitionContext.view(forKey: .from)
let toView = transitionContext.view(forKey: .to)
```

Then, we put them in our desired starting position:

```
// Xcode -> UIKit -> TransitionAnimator.swift

if isPresenting {
    toViewBeginningFrame.origin = CGPoint(x:
containerFrame.size.width,
y: containerFrame.size.width)
    toViewBeginningFrame.size = toViewEndingFrame.size
} else {
    fromViewEndingFrame = CGRect(x:containerFrame.size.width,
                                 y:containerFrame.size.height,
                                 width:toView?.frame.size.width ??
0,
```

```
                                    height:toView?.frame.size.height ??
0)
}
toView?.frame = toViewBeginningFrame
```

And finally, we animate them in:

```
UIView.animate(withDuration: transitionDuration(using:
transitionContext), delay: 0.0, usingSpringWithDamping: 0.8,
initialSpringVelocity: 0.9, options: .curveEaseOut) {
    if self.isPresenting {
        toView?.frame = toViewEndingFrame
        fromVC?.view.frame = containerFrame
    } else {
        fromView?.frame = fromViewEndingFrame
    }
} completion: { done in
    let succeeded = !transitionContext.transitionWasCancelled
    let failedPresenting = (self.isPresenting && !succeeded)
    let didDismiss = (!self.isPresenting && succeeded)

    if (failedPresenting || didDismiss) {
        toView?.removeFromSuperview()
    }

    transitionContext.completeTransition(succeeded)
}
```

As previously mentioned – these can be interactive. The easiest route to get that done is by using UIKit's object that handles most of that for you, `UIPercentDrivenInteractiveTransition`. Your transitioning delegate is a great spot to put these, and then later vend during the transition setup:

```
// Xcode --> UIKit --> TransitionDelegate.swift

func interactionControllerForDismissal(using animator:
UIViewControllerAnimatedTransitioning) ->
```

```
UIViewControllerInteractiveTransitioning? {
    guard demoedUseCase
== .interactiveAnimatorAndPresentationController
else { return nil }
    return interactiveAnimator
}
```

With that in place, you can use the animations from the animation controller while telling UIKit how much of it is completed via a gesture, such as a pan or a drag. We do this in the presentation controller, which we'll look at now.

A Presentation Controller (Optional)

The presentation controller can handle, unsurprisingly, anything related to the transition from the *presenting*controller's side. It also can provide the final frame that the presented controller's view should have. Finally, it's a great place to respond to changes in the app's environment. For example, if you need to react to something like a size class or trait collection change, the presentation controller can query those values and adjust frames accordingly.

Going back to our example, if we wanted to provide a dimming view for the transitions – the presentation controller is where it should happen. To begin with, though, our transitioning delegate has to provide it – which happens here in the demo project:

```
// Xcode -> UIKit -> TransitionDelegate.swift

// MARK: Presentation Controllers
func presentationController(forPresented presented:
UIViewController,
presenting: UIViewController?, source: UIViewController) ->
UIPresentationController? {
    // Truncated for clarity
    TransitionPresentationController(presentedViewController:
```

```
presented, presenting: presenting)
}
```

Creating our own, in contrast to previous objects, doesn't require a protocol but inheritance from `UIPresentationController`. There, we can hook into the presentation lifecycle to add in our "chrome" to the transition. This is where we add in the custom dimming view in the demo example:

```
// Xcode -> UIKit -> TransitionPresentationController.swift

override func presentationTransitionWillBegin() {
    guard let container = containerView else { return }

    dimmingView.frame = container.bounds
    dimmingView.alpha = 0.0
    container.insertSubview(dimmingView, at: 0)

    presentedViewController.transitionCoordinator
.animate(alongsideTransition: { [weak self] context in
        self?.dimmingView.alpha = 1.0
    }, completion: { [weak self] context in
        if context.isCancelled {
            self?.dimmingView.alpha = 0.0
        }
    })
}
```

The presentation controller also provides the frame we want the presented controller to end up at. This is where the animation controller will animate it to once the transition has completed, and it'll find it in the transitioning context that UIKit will provide.

The code shows how we make the presented controller take up only half of the screen:

```
// MARK: Sizing
override var frameOfPresentedViewInContainerView: CGRect {
```

```
    var presentedViewFrame = CGRect.zero
    guard let containerBounds = containerView?.bounds else
{ return .zero }

    presentedViewFrame.size = CGSize(width:
(containerBounds.size.width/2),
height: containerBounds.size.height)
    presentedViewFrame.origin.x = containerBounds.size.width -
presentedViewFrame.size.width;
    return presentedViewFrame;
}
```

Remember – these objects are a great spot to handle interactive transitions too. Since these are nearly always driven via a gesture of some sort – and the presentation controller handles the chrome, it's a perfect spot to add that gesture to the view hierarchy and respond to it. That's exactly what we do once the transition is complete, and we intend for the user to be able to "swipe to dismiss" the view:

```
// Xcode -> UIKit -> TransitionPresentationController

override func presentationTransitionDidEnd(_ completed: Bool) {
    guard let container = containerView,
    transitioningDelegateWantsInteractiveDismissal == true,
    let _ = presentedViewController.transitioningDelegate as?
TransitionDelegate else { return }

    // Attach a gesture recognizer to the dimming view to allow for
a swipe
to dismiss
    // Once we've confirmed that our transition delegate wants one
    panGesture = UIPanGestureRecognizer(target: self, action:
#selector(handleSwipeToDismiss(pan:)))
    panGesture?.maximumNumberOfTouches = 1
    container.addGestureRecognizer(panGesture!)
}
```

If you open that file, you'll also see that at the bottom – we handle the gesture and update the interactive transition's progress according to how far the user has drug up or down with their gesture.

To recap, presentation controllers hint at all they should be doing within their name – they handle the presentation. They don't control the animations. They can *animate* chrome outside of the custom transition, but most of what they are responsible for is providing a final frame for the transition, and optionally handling any chrome considerations too.

UIKit Created Objects

There are couple of important objects that UIKit will make on our behalf that we can use in a few different places. Namely, the transitioning coordinator and the transitioning context. Again, we don't typically have to subclass or otherwise make any of these ourselves, UIKit is going to hand them off to us when we need them.

UIViewControllerTransitionCoordinator

The transition coordinator will be present on a view controller during any presentation or dismissal. You can find it on a view controller's`transitionCoordinator` property to use. This object is very handy to customize the transition further, *but* for tasks that fall outside of the animator object's responsibilities.

The reason it's key to know about is that it works in lockstep with the animator object to perform animations within the same animation group as the transition is animating. That means you don't have to mess with any of the timing parts yourself, the system can take care of it.

```
transitionCoordinator?.animate(alongsideTransition: { context in
    // These animations will happen alongside the transition
```

```
}, completion: { context in
    // Perform any clean up
})
```

Transitioning Contexts

You'll notice that UIKit also passes a context, which is an instance of `UIViewControllerTransitionCoordinatorContext`, so you can get critical information about the transition that is occurring to transition coordinators. This is similar to the object we use in our animation controllers, which is slightly different in naming – that one is typed to `UIViewControllerContextTransitioning` but they do very similar things. You can see if the animation is done, cancelled, interruptible and more.

You'll use these transitioning contexts to get information – that's it. They give you the container view for animations, the view controllers taking part in it and let you know what the current state of the transition is.

Kicking off the Transition

This may seem silly, but after all of that information – it's actually very easy to miss how exactly we kick off these transitions. That's done in a two step process:

1. Tell UIKit that your view controller is supplying a custom transition, `myViewController.modalPresentationStyle = .custom`.
2. Then, supply your transitioning delegate to vend all of the necessary objects taking part in it: `myViewController.transitioningDelegate = myCustomTransitioningDelegate`.

Then, UIKit will take over and hit all of the objects we've been talking about. I'll cover this a bit more in the tips section below, but remember – you don't always need all of

these objects. Now that you know what each of them do, start by asking yourself what exactly you're trying to achieve with the animation of the transition and work your way backwards from there to know what you need to use.

The High Level View

I think developers of would-be custom transitions typically struggle with the API because there are quite a few objects involved. You've objects that you yourself make, ones that UIKit creates and on top of all of that – you only need *some* of them depending on what you're doing.

So, again – browse the sample code here, I think that's critical when it comes to bringing it all together. To recap, things that we make ourselves:

- Our transitioning delegate object vends other objects to help drive the transition.
- Our animation objects animate the presented controller's view however we wish, and it specifies how long the transition should be.
- Our presentation controller optionally vends custom chrome during the transition from the *presenting* controller's point of view – and it also tells UIKit where the custom animation's frame (occurring from the animation controller) should end up by supplying a `CGRect`.

And, things UIKit makes for us:

- A transition coordinator can help you animate something *alongside* the transition that is about to occur, whether it's a customized transition or just a stock one.

- The transition context is sent to the animator to give you information about the views taking place in the transition, and also (by way of the presentation controller) where they should end up.

Tips

Use Only What You Need

Remember that you don't need to use all of this API to get a lot out of it. For example, if all you wanted to do was make your presenting controller's view have a red background color during a transition — you don't have to make a custom transition or presentation controller at all, you can just reach into the transition coordinator:

```
// Xcode -> UIKit -> CVCTFig3ViewController.swift

transitionCoordinator?.animate(alongsideTransition: { context in
    self.tv.backgroundColor = .red
}, completion: { context in
    if context.isCancelled {
        self.tv.backgroundColor = .systemGroupedBackground
    } else {
        UIView.animate(withDuration: 0.25, delay: 0.0, options:
.curveEaseOut) {
            self.tv.backgroundColor = .systemGroupedBackground
        } completion: { finished in

        }
    }
})
```

Similarly, if you want to make the presented controller's frame do something specific, then all you really need is the custom transitioning delegate to vend an animation

controller that you create. In that case, there's really no need for the custom presentation controller here.

All of this demonstrates why you *need* to know how all of these pieces fit together, otherwise you'll end up giving yourself too much work for little payoff by including pieces which end up being superfluous.

A Little Onboarding Punch

I like to include custom view controller transitions especially in two places:

- Onboarding.
- Or, very early on in the app.

As in my Spend Stack example at the top of the chapter, you've got a quick chance to really "wow" people and show off the care and craftsmanship you've put into your app. But be smart here, when you shoot for these early, you're in a high payoff and high risk scenario. If your transition gets in the way or simply feels weighty and "too much", then the opposite becomes true – you can lose people early and they may not come back.

The Bottom Sheet and Detents

Some may reach for a custom view controller transition to present a "bottom sheet", that is – a modal presentation that only shows about half way up the screen. UIKit has support for this out of the box, and you can do it by reaching into its `UISheetPresentationController` which every view controller has:

```
// Xcode -> UIKit -> CVCTFig4ViewController.swift

let vc = UIViewController()
vc.view.backgroundColor = .purple
```

```
if let sheet = vc.sheetPresentationController {
    sheet.detents = detents
}
self.present(vc, animated: true, completion: nil)
```

Knowing what you know now – it also makes a lot of sense that configuring how high or low the sheet should go is controlled by this presentation controller, right? Recall that they can specify the ending frame of the presented controller's view – and at a high level that's all that these detents are doing.

Understanding Full Screen Transition Gotchas

If you opt to use an out of the box UIKit presentation style, you may be confused to find that the views underneath the presented view controller are removed. This can lead to odd results, especially if you're attempting to show some sort of blur view or one which incorporates transparency. These types of experiences are meant to hint at the content underneath it.

In these cases, you don't *have* to go to the trouble to create a custom view controller transition. Instead, specify `UIModalPresentationOverFullScreen` as your `modalPresentationStyle` – this will leave the view content underneath intact.

What About SwiftUI?

Custom view controller transitions are, naturally, tied to UIKit and its view controller paradigm. However, what we're really talking about with these transitions is "Animate this view I'm looking at now away, and animate this other view in."

In SwiftUI, you can achieve similar effects by using `.matchedGeometryEffect` and hooking two `View` structs up using a namespace. Remember that SwiftUI is completely different from UIKit in so many ways, so it helps to remember that presenta-

tions are just a different animal altogether. Still, `.matchedGeometryEffect` is probably the closet analog to UIKit's custom view controller transitions.

Three Key Takeaways

1. UIKit's stock presentations are well done and cover several scenarios, so take time to understand all of the options they provide to you.
2. Figuring out how the A.P.I. fits together is critical, because it can be difficult to understand at first.
3. Custom view controller transitions can add level of unmatched interactivity and delight, so look for places where they can add value.

Localization and Internationalizations

There's simply no getting around the fact that iOS devices are a global phenomenon. While they were originally created by a Western-based company, their reach literally expands across the entire planet.

For this reason, you should try not to think of your app in a silo – where only your demographic and its tendencies apply. If you live in the United States, it's easy to make assumptions like this:

- The color red means something bad, or dangerous.
- The imperial system of measurements will suffice.
- Things like formal greetings, such as "Hi, I'm Jordan!" will not require a lot of horizontal space.

All of these things fail an internationalization test. For example, in China, red often symbolizes luck or joy. That's a far cry from Western culture, where a red icon is synonymous with deletion, or actions that carry destructive outcomes.

Further, as you probably know, the United States is one of the last countries to still use the imperial measurement system (nearly every other territory relies on the metric system)…

```
// Wrong, except if you're in the 🇺🇸
let milesLeft = getMilesRemaining()
let milesRemaining = "\(milesLeft) miles remain."
```

…and simple greetings in English can run incredibly long in other languages. From our example above, here's what "Hi, I'm Jordan!" would look like in other languages:

Language	**Result**
English	Hi, I'm Jordan!
Simplified Chinese	大家好，我是乔丹！
Indonesian	Hai, saya Yordania!
Japanese	こんにちは、ジョーダンです！
Russian	Привет, я Джордан!

Make no mistake, you need to have a global mindset – even if you're not shipping worldwide today. With the App Store being available in over 175 countries supporting 40 languages – it's quite literally the definition of participating in a global market. In this chapter, I'll introduce some core concepts to think about and list out some tips to help you get started with gradually making your app ready for use no matter the territory it's released in.

The process of *internationalizing* your app essentially means you're thinking about a global audience. When you think about internationalizing your app – that naturally leads to the most common component of that process, which is *localizing* it. That deals solely with the process of providing a translation of your app's user interface in the places it's available in.

The two terms get mixed and mashed together often, but for our purposes that's how I'll be using them. In the same way that several developers assume that VoiceOver support comprises all of your accessibility work, so to do many assume that localizing your app is "enough". I don't think there's any ill intent in that statement, but the fact remains – there is a lot we can do beyond localizing our apps, even though it may be the highest impact portion of the internationalization process.

With that said, let's dig in on the higher level concepts. Afterwards, we'll use the majority of this chapter to look at several coding techniques you can apply which reinforce these concepts.

Internationalization Touchpoints

Text

Written language is a complicated beast. The object in which we represent them in iOS apps, the Swift `String`, or the Foundation based `NSString`, are typically some of the first things nascent programmers encounter when learning the ubiquitous "Hello, World" learning exercise.

While simple on the outset, they are intricate structures with enormous capabilities. From representing all sorts of encoding standards, glyphs, representing charac-

ters and more – they abstract many of the complicated bits away from us (thankfully!). And so it is, representing text in your app is similar.

Slotting in text into your user interface seems easy at first, and you might even move on after your initial implementation. But, there are two key things to remember when constructing them for a global audience. That is their alignment and length, because those two things can vary by territory. In the case of the latter, the delta from one country to the next can be staggering (i.e. a string in English versus Russian).

In Arabic languages, text is read from right-to-left. So, if you're not careful you can exclude a large swatch of users by not considering text alignment. For example, if you always used something like this…

```
myLabel.textAlignment = .left
```

…you'd have some odd looking text in most cursive script languages. There's almost *never* a reason to use that. Things like this, your tonality in how you speak and how long you think a string might end up being are all pivotal in effective internationalization.

Again, we'll look at things you can do to get these types of things done correctly in the "Tips" section.

Before we move on – of course I should that a big part of text efforts around internationalization is actually localizing it. That can take a lot of work, but if you get your strings localized – you're taking the biggest step necessary in making your app gain potential users in other territories.

Color

As we mentioned above, colors have different meanings across different territories. There isn't much to add here other than to say – do some research. Find out *why* you're using a color and if it's supposed to represent a common result (i.e. success is usually green, or deletion is typically red, etc.). Then, check into your supported territories to make sure you go with the right one.

Written Copy and Inconography

Tonality, like color, is another important point to consider. Slang and other territory specific language customs are typically discouraged. I disagree, because I think they can add a lot of personality to an app. But, they must be done tactfully.

Where I think developers and designers get into trouble is when they come up with a clever slogan that goes over well in their main demographic's territory, but is completely backwards somewhere else. Be wary of that, since being "edgy" may fit your app's voice – there could be a perfect motive to do so. But if it doesn't get localized, you can find yourself in some awkward situations where the translation might become nonsensical.

For a lighter example of this, you can even be as detailed as getting words that technically are understandable but spelled differently correct per territory. A canonical example of this would be "color" versus "colour." If you can get the small details right, people will typically trust your app to handle the bigger ones – which is a common theme I try to reiterate throughout this book series.

Further, iconography at times will need some changes too. An obvious place? Text. Revisiting our Arabic text versus English text, one is a cursive script while the oth-

er is Latin script. Differences between the two, like the absence of any notion of upper or lower case letters, make aligning icons tricky.

The good news is that, by using the correct alignment APIs and leaning into SF Symbols wherever possible – most of these issues (aside from context meaning) are taken care of for you.

Tips

Testing for Different Territories

Throughout the process of internationalization, you'll want to test early and often. Apple has rich support to do that, but the approach differs depending on your technology stack.

For SwiftUI, things are fairly trivial by injecting whatever layout direction you need into the environment:

```
.environment(\.layoutDirection, LayoutDirection.leftToRight)
```

In practice, you'll use this in your previews. In this example, the direction of the faux-back button arrow switches depending on the layout direction:

```
// Xcode -> SwiftUI -> ILFig1View.swift

struct ILFig1View: View {
    @Environment(\.layoutDirection) var direction

    var body: some View {
        VStack {
            HStack {
                Text("Go Back")
                Image(systemName: symbolForGlyph())
```

```
                Spacer()
            }
            .font(.system(.caption,
design: .monospaced).weight(.semibold))
            .frame(minWidth: 0, maxWidth: .infinity)
            RoundedRectangle(cornerRadius: 4.0, style: .continuous)
                .fill(Color.blue.opacity(0.5))
            Spacer()
        }
        .padding()
    }

    private func symbolForGlyph() -> String {
        let direction = direction == .leftToRight ? "backward" :
"forward"
        return "arrow.uturn.\(direction)"
    }
}

struct ILFig1View_Previews: PreviewProvider {
    static var previews: some View {
        Group {
            ILFig1View()
                .environment(\.layoutDirection,
LayoutDirection.leftToRight)
            ILFig1View()
                .environment(\.layoutDirection,
LayoutDirection.rightToLeft)
        }
    }
}
```

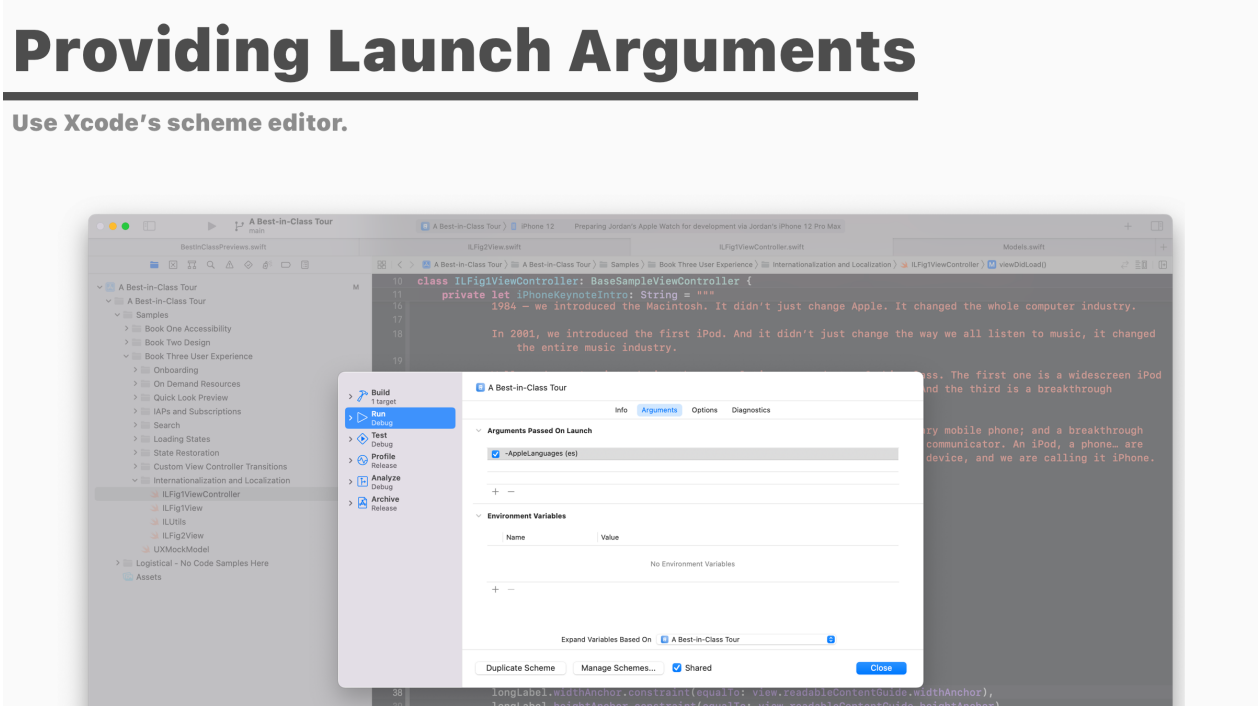
Additionally, whether or not you're using UIKit or SwiftUI, you can edit your scheme to put it into a specific territory. This is great for testing the localization parts of your app. Even better, there are a few ways to do it.

Use a Launch Argument

You can edit your scheme to use any locale you'd like as a launch argument:

```
-AppleLocale (en-AU) // Austrailia
```

So long as you include the language code, it'll be keyed according to its ISO 639 code. For this reason, it's better to provide the language *code* instead of the language.



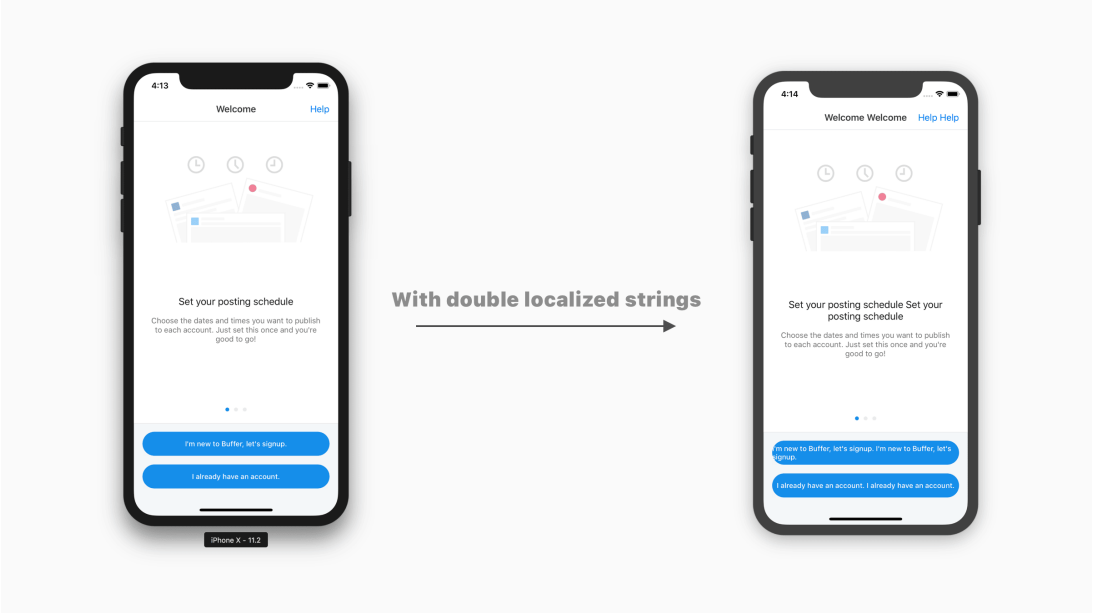
Edit your scheme to provide launch arguments.

There are other useful launch arguments Xcode supports to aid with testing, but in my experience — they aren't well known among developers. Whether Apple doesn't push them as much or we aren't aware, regardless, I find them valuable for locale testing:

1. `-AppleLanguages (language code)`: Much like our example above, this forces the *language* to use whatever language code you provide. For

example, you may use the locale argument to test right-to-left languages whereas this one is great if you're looking to test only localized strings.

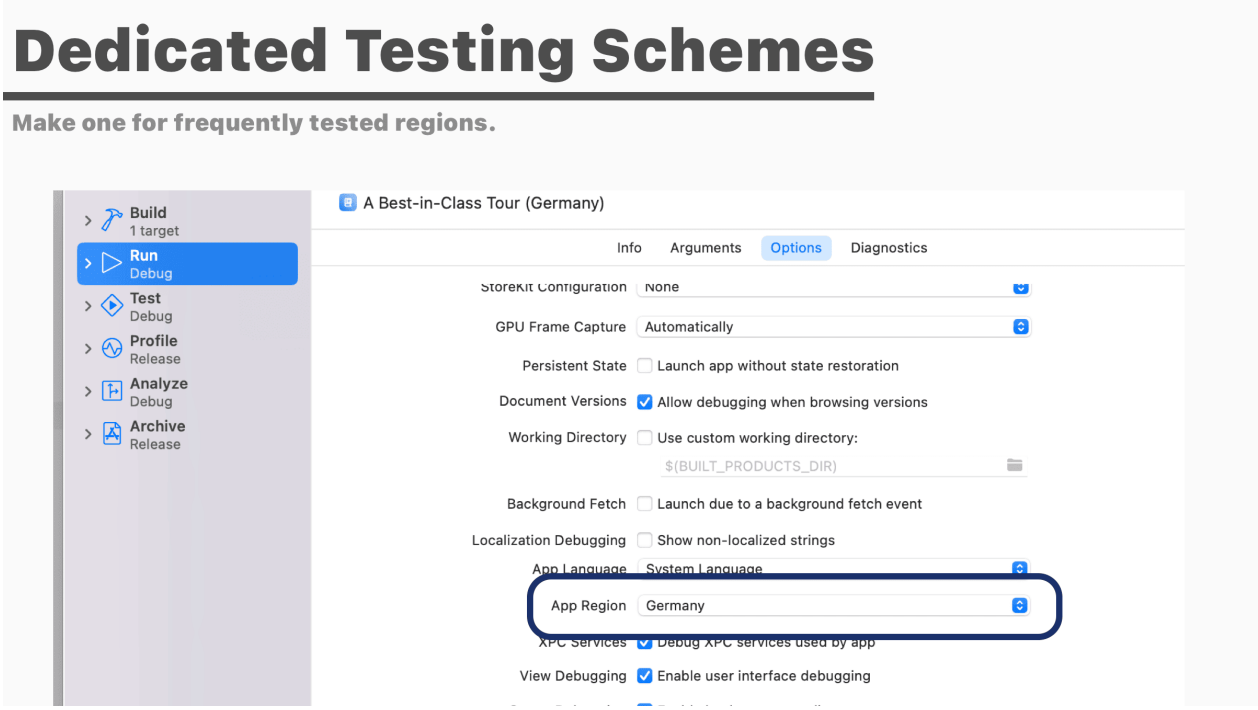
2. `-NSForceRightToLeftWritingDirection`: Just like it says on the tin – this will force text to read right to left, no matter the locale you're using.
3. `-NSShowNonLocalizedStrings`: This is key for identifying any strings which *haven't* been localized.
4. `-NSDoubleLocalizedString`: Another tool to help you combat longer text scenarios. This simply doubles each and every localized string in your app, and displays it according to your current locale. Here's an example from when I used this technique on the app Buffer:



In my opinion, launch arguments are one of the best tools you have to test the internationalization process. By editing a scheme to provide these values, you can fast track fixes by identifying issues quickly. In fact, Xcode's schemes provide several avenues to shore up testing for any locale. If you find yourself needing to test for one particular

territory quite often, you could even create a dedicated scheme for it instead of messing around with launch arguments. Here's how it works:

1. Duplicate your existing scheme.
2. Name it according to the region you're testing for (i.e. My App (German)).
3. Then, under the scheme's Run (debug) –> Options, edit "Application Region" to be wherever you need to test for.



These techniques free you from having to go into the actual Settings app in the simulator and change the device's region, as you would on an actual device. Plus, that route also requires a reboot.

Handling Measurements

As some point, you're likely to run into a situation that requires measurement. Let's say, as an American, I have an app which calculates some sort of distance measurement. Perhaps how many miles it is from point A to point B.

You might be tempted to write something like this:

```
// Xcode -> ILUtils.swift

private let distanceInMiles = 100.0

func incorrectDistanceLeftExample() -> String {
    return "\(distanceInMiles) kilometers to go."
}
```

Obviously, 100 miles is not equivalent to 100 kilometers. Taking care of these types of situations would be a lot on your plate, and thankfully – Foundation has taken care of it for you via `Measurement`:

```
// Xcode -> ILUtils.swift

func distanceLeftExample() -> String {
    let miles = Measurement(value: distanceInMiles, unit:
UnitLength.miles)

    let formatter = MeasurementFormatter()
    formatter.unitStyle = .medium
    formatter.locale = Locale(identifier: "fr_FR")

    // Returns 160,934 km to go
    return "\(formatter.string(from: miles)) to go."
}
```

As you can see, `Measurement` combined with `MeasurementFormatter` is an incredibly powerful combination. I should note that in the example above – you're unlikely to need to set the `locale`:

```
formatter.locale = Locale(identifier: "fr_FR")
```

The formatter, like its other formatter classes, will use the device's locale by default. I've set it here to force the result to show in the metric system. Though, on the other hand – if I hadn't set and considering I live in the United States, it would correctly format it for me (i.e. "100 miles to go."), but that's not the result I was after in the sample code.

This is great, too, because if it *didn't* work like that – could you imagine the amount of work you'd have to do?

```
if (isCanada)  {
    // kilometers
} else if (isChinese) {
    // Translate the unit
}  else if (isArabic) {
    // Translate the unit, number representation AND make it right
to left
}
```

The measurements you can track are virtually endless, as most any unit you can think of is provided and every locale is supported:

```
let distance = Measurement(value:10, unit: UnitLength.miles)

let frenchDistance = MeasurementFormatter()
frenchDistance.locale = Locale(identifier: "fr")

let chineseDistance = MeasurementFormatter()
chineseDistance.locale = Locale(identifier: "zh")
```

```
let arabicDistance = MeasurementFormatter()
arabicDistance.locale = Locale(identifier: "ar")

// 🇫🇷** **-> 16,093 km
print("🇫🇷 -> \(frenchDistance.string(from: distance))")

// 🇨🇳** **-> 16.093公里
print("🇨🇳 -> \(chineseDistance.string(from: distance))")

// 🇯🇴** **-> كم **١٦,٠٩٣**
print("🇯🇴 -> \(arabicDistance.string(from: distance))")
```

Further, if you simply need to convert between units, `Measurement` has you covered by way of `NSUnitConverter`. If all I was after above was the actual number value of what 100 miles is in kilometers, I could get that easily by using the `converted(to:)` function:

```
let kilos = miles.converted(to: .kilometers)`
```

Of course, since a measurement is typically tied to a number, you can combine a number formatter right along with a measurement formatter:

```
let distance = Measurement(value:0.2, unit: UnitLength.miles)

let frenchDistance = MeasurementFormatter()
frenchDistance.locale = Locale(identifier: "fr")

// 🇫🇷** **-> 0,322 km
print("🇫🇷 -> \(frenchDistance.string(from: distance))")

let digitFormat = NumberFormatter()
digitFormat.minimumSignificantDigits = 4

frenchDistance.numberFormatter = digitFormat

// 🇫🇷** **-> 0.321868 km
```

```
print("🇫🇷 -> \(frenchDistance.string(from: distance))")
```

Having Siri Speak Individual Languages

By using the `.accessibilitySpeechLanguage` key with an attributed string, you can guarantee that Siri will speak out the phrase in the correct tongue no matter what localization a device is in.

Consider an eBook app you've made, where part of the story has a French character speaking. For an English territory, you don't want Siri to attempt a French phrase in an English tone. So, you can match a String to a language to get the correct result:

```
// Xcode -> ILUtils.swift
func haveSiriSpeakFrench() -> NSAttributedString {
    return NSAttributedString(string: "joie de vivre", attributes:
[.accessibilitySpeechLanguage:"fr-FR"])
}
```

Handling Text Across Locales

To tackle that right-to-left text (or any localized and internationalized text), it really comes down to a few core principles:

1. Use the correct alignment.
2. Avoid truncating text.
3. Allow for ample reading width.

Alignment, thankfully, is very easy to get right. In SwiftUI, well – it's just done for you out of the box. In UIKit, we'll need to set our `.textAlignment` accordingly:

```
let longLabel = UILabel(frame: .zero)
longLabel.text = iPhoneKeynoteIntro
longLabel.numberOfLines = 0
```

```
longLabel.textAlignment = .natural // Natural will adapt
longLabel.translatesAutoresizingMaskIntoConstraints = false
view.addSubview(longLabel)
```

When you use `.natural`, under the hood UIKit assigns `.left` for left-to-right languages, and `.right` for right-to-left languages.

For the second point, how you handle it will depend on your scenario. But, for starters, consider that text should grow so it doesn't truncate. For example, if the string was "Tap here to complete your purchase", and you didn't provide a means for that string to grow – now you're in trouble. What if the last word was cut off for Indonesian locales?

Text should grow anyways, really. Dynamic Type all but ensures the perfect spec you see in a design is perfect *only* there. Once the user opts for the largest accessibility type setting, the string's point size (not font size – the actual point size of a label's frame, for example) could easily quadruple. So, leverage scroll views, AutoLayout, the trait collection, `ViewThatFits` or adapting `HStack` to `VStack` methods in SwiftUI or any of the many tools Apple has provided us API for to solve that problem.

Though my third point isn't distinctly about internationalization, I think I'd be remiss if I didn't mention how important it is to provide a sensible width for your text. Nobody wants to read small text that spans the entire width of an iPhone, let alone if it was in landscape.

Apple has API to give you a "just right" space, and you can find it in UIKit via a `UILayoutGuide`, more precisely – `readableContentGuide`:

```
// Xcode -> UIKit -> ILFig1ViewController.swift

override func viewDidLoad() {
    super.viewDidLoad()
    let longLabel = UILabel(frame: .zero)
    longLabel.text = iPhoneKeynoteIntro
```

```
    longLabel.numberOfLines = 0
    longLabel.textAlignment = .natural
    longLabel.translatesAutoresizingMaskIntoConstraints = false
    view.addSubview(longLabel)

    // Readable Content Guide usage
    NSLayoutConstraint.activate([
        longLabel.centerXAnchor.constraint(equalTo:
view.centerXAnchor),
        longLabel.centerYAnchor.constraint(equalTo:
view.centerYAnchor),
        longLabel.widthAnchor.constraint(equalTo:
view.readableContentGuide.widthAnchor),
        longLabel.heightAnchor.constraint(equalTo:
view.readableContentGuide.heightAnchor)
    ])
}
```

Using it, the text's width (by way of the `UILabel` frame being set from the constraints) has enough breathing room. This is especially useful on iPad, where the available width is typically more abundant. Unfortunately SwiftUI doesn't have a similar API, but you can get *close* by using about 70% of the container's width for your "readable content guide" in these cases.

Semantic Colors for Territories

Apple has already coined the phrase "Semantic Colors", and its part of its API suite. But, I can't think of a better term for deciding what a color should make a user *feel* like. For example, going back to "red usually means something happy in Chinese" and "red means something dangerous in the United States" – we could give that feeling a meaning in our code to present the right color.

By using a computed property, you could check the device's locale and return one color or the other respectively:

```
// Xcode -> ILUtils.swift

extension UIColor {
    var success: UIColor {
        get {
            if Locale.current.identifier == "zh_CN" {
                return .red
            } else {
                return .green
            }
        }
    }
}
```

Localize Piecemeal

It can be incredibly intimidating to look at a mature project and consider the work that might go into internationalization. If that's you – take a deep breath and consider taking an incremental approach.

Try find a consolidated experience or spot in your code (either one will do) and get a foundation for how that work will look for you by implementing it there first. Then, once you've got an architecture that works, you can begin scaling it across the other parts of your app.

Three Key Takeaways

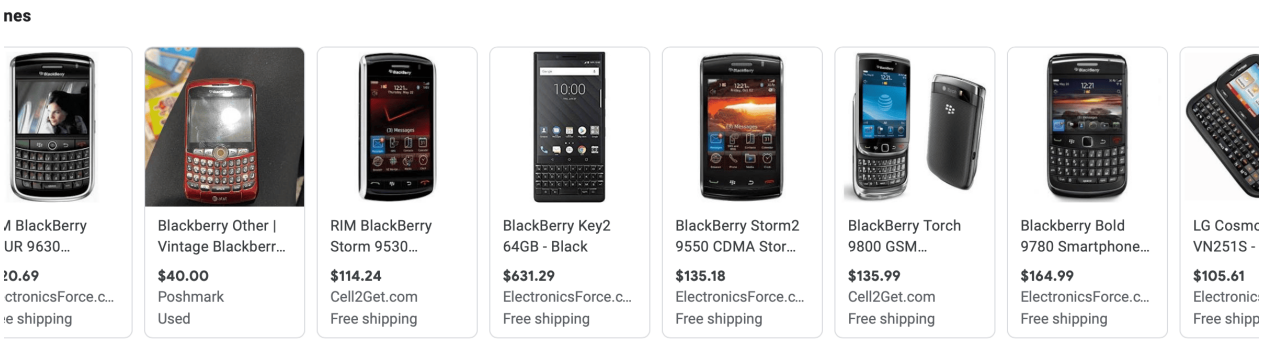
1. Our apps could be used all over the world, so try not to box your architecture or designs into one demographic.
2. Take the process seriously, and don't simply ship it everywhere because you can.
3. If you think about localization and internationalization from the start (much like accessibility design), the process can be fairly seamless.

Micro Animations

The feeling of tactile feedback adds a lot to apps. Before the first iPhone changed everything about phones, Blackberry was the leader in the arena by several measures. One such area? It's signature, plastic, clackity keyboard:

Physical Keyboards

A quick Google search shows the popular keyboard present on all Blackberry phones.



In the early 2000's, this is how you got things done on the go. Replying to emails and composing them was a novelty, and a lot of it was done on classic QWERTY keyboards that looked a lot like the ones Blackberry popularized (and my sympathies if you can even remember texting with T9 standards).

Of course, you're reading this book – so you know how the story ends. The iPhone's iconic keyboard changed everything we knew about user experience on a touch-only device. No stylus, no hardware keyboard components – and yet it still worked.

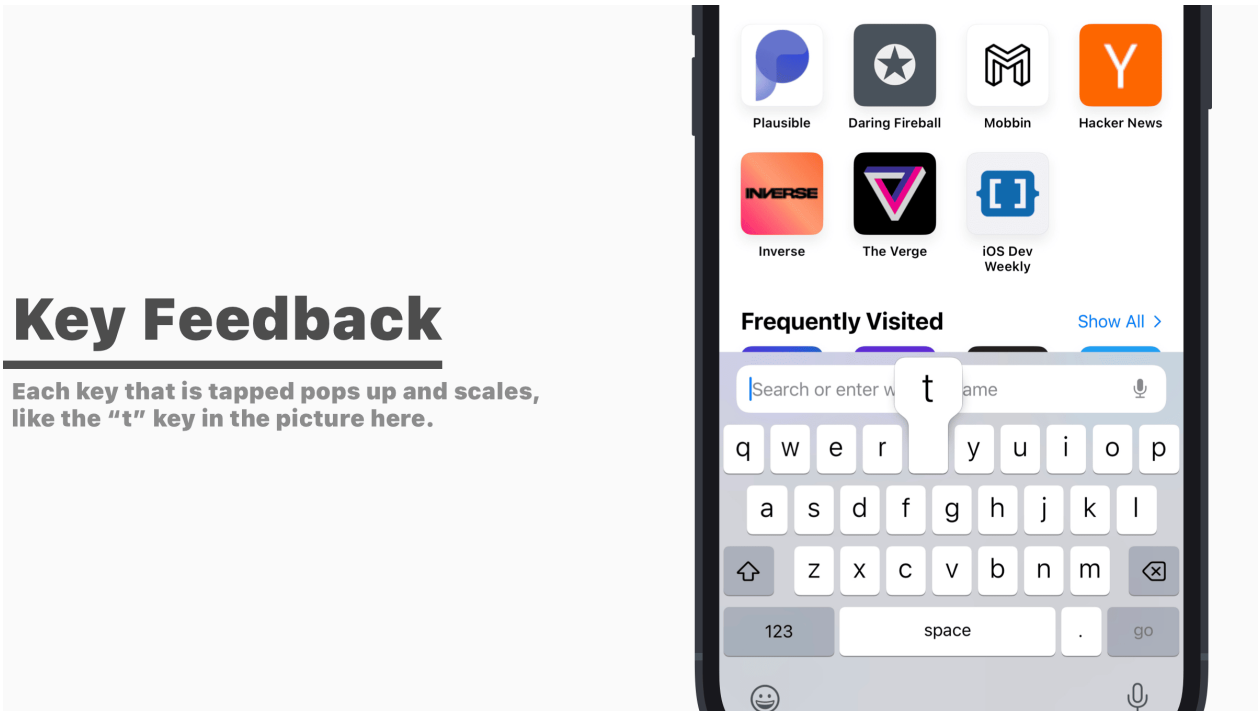
Part of the lesson here? To help replace hardware with software, we need to provide a fine degree of feedback to interactions.

Haptics help a lot with this. They introduce a level of confidence that what you thought would happen, did in-fact happen. In iOS 16, Apple even introduced an option to use haptics for the system keyboard, and I personally can't imagine typing without it once I started using it. It just *feels* right.

There is a gap we need to fill somewhere between the physical feeling of hardware, and interactions we've replaced then with via software.

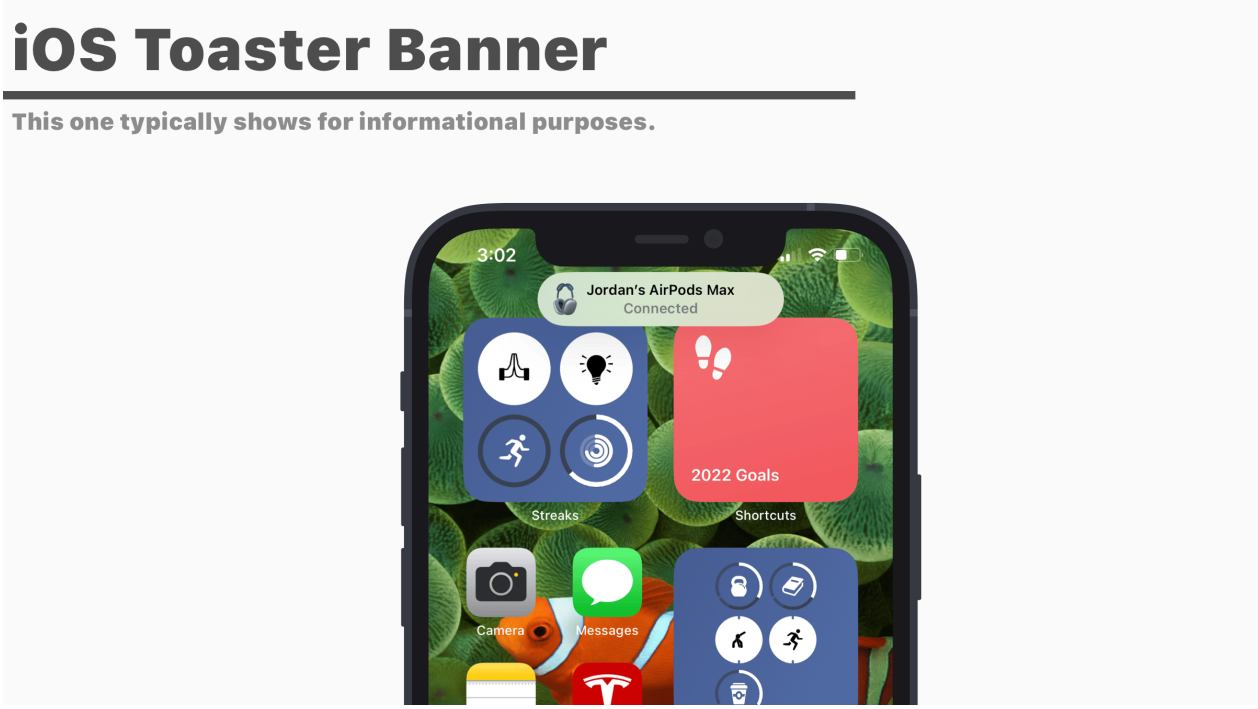
Aside from the physical feedback of technologies like haptics, what else can we do? The use of "micro animations" can help us out here. I don't think this is an industry wide term, but I think of them as little animations, usually less than a second, that help drive delight and confidence in interactions.

In the case of the iPhone's keyboard, they **reinforce** an action — showing you the key that you tapped:

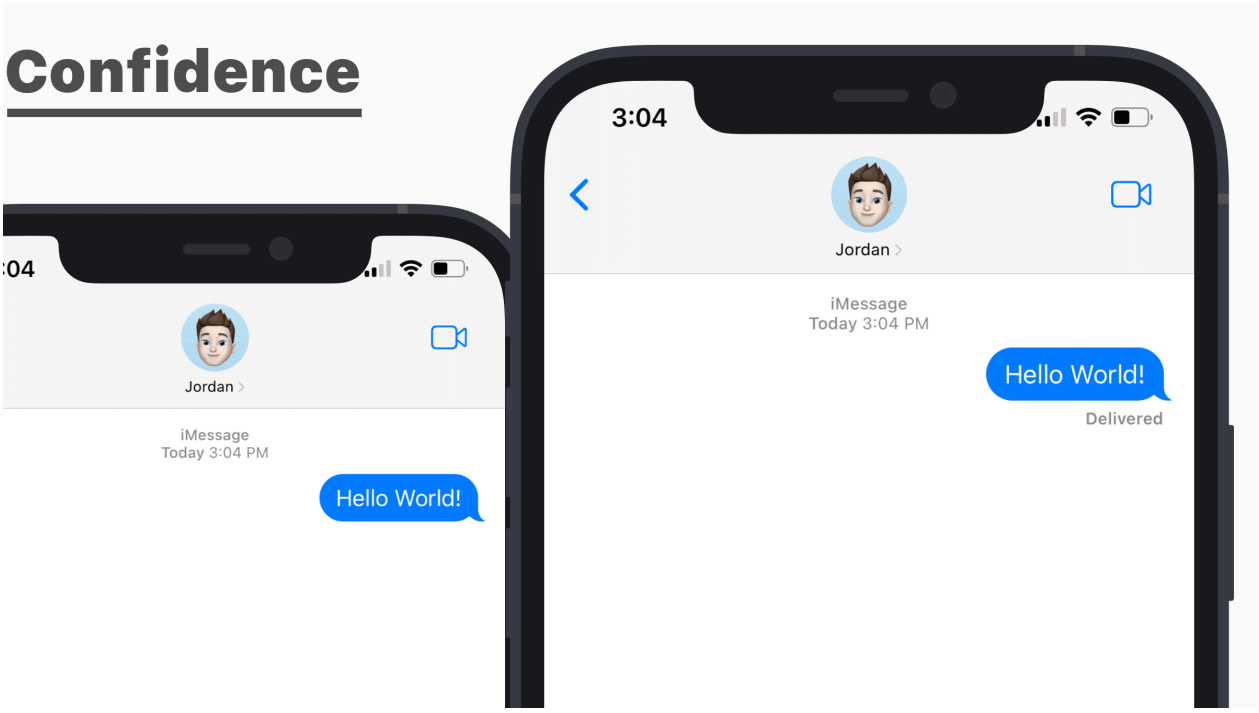


In social media apps, where the ubiquitous "like" or "favorite" actions are found, they manifest themselves as little pops and scales interpolations that evoke a sense of joy when you perform them. Really, it's only purpose is to **delight** you.

Or, how about the "toaster" banner iOS has been using for a few years now? It slides down and scales its contents — it's purely **informational**, but it's also fun:



Here's another one I love — when you send a message in Messages, it has two states. It's sending state, where it shows in the list — but also a delivered state, once the actual network requests has finished. If it was sent successfully, a tiny "Delivered" label scales and fades in — making me confident the message has gone through:



It's transient too, so after a few seconds it fades away. What a nice little touch.

Take an element that's disabled. Maybe if you try to tap on it, you'd get a little jiggle animation along with a haptic response – letting you know that particular element isn't available for use. The list of examples goes on and on, but with that frame of mind you can likely recall a few micro animations you've seen yourself.

So taking a step back, we can use micro animations to do a few primary things. We can:

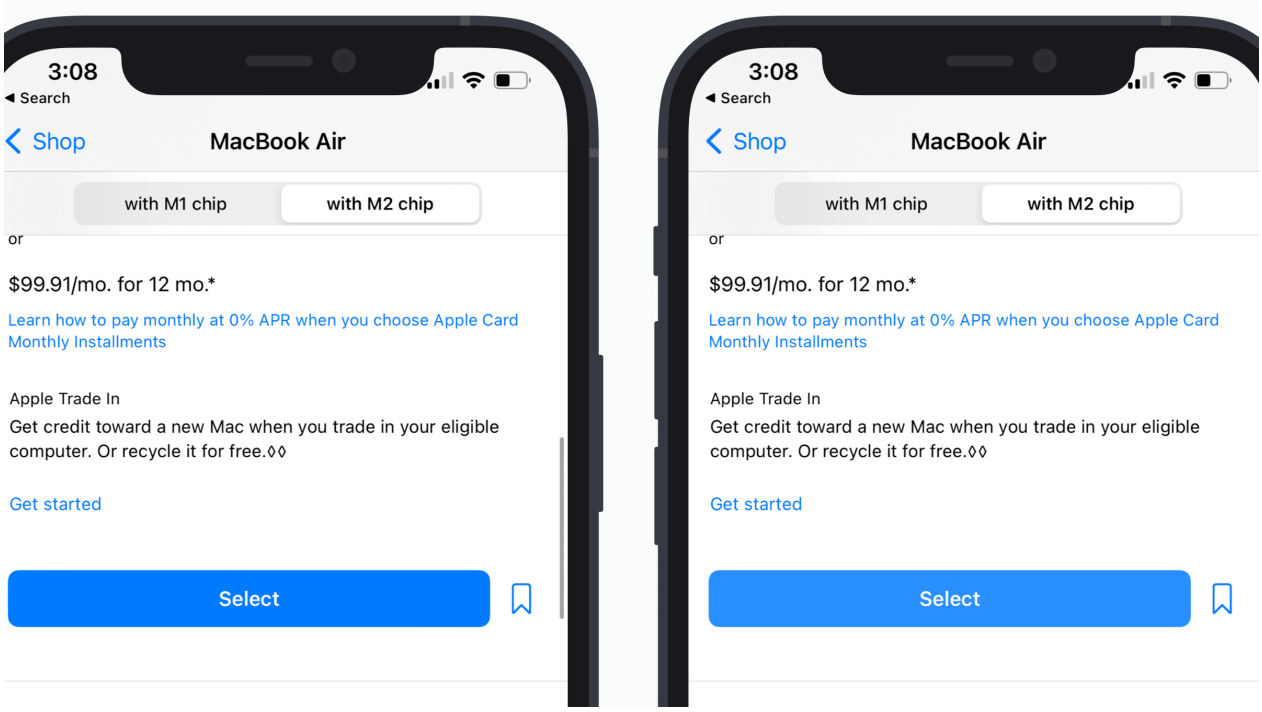
- Use them for informational purposes.
- Reinforce actions.
- Direct attention to certain elements.
- Make things simply more fun and delightful.

Their utility expands beyond that list, but typically I might consider using them when a little animation fits into one of those buckets. Especially that last one – if a little anima-

tion makes something more fun, heck – do it! Software that's fun usually works out well. But, it can't be distracting *and* it has to be earned. If every piece of your app pops, scales and fades – you might have accidentally developed a toy, not an app.

iOS Examples

To further the impact these animations can have, consider the humble button. On iOS, stock `UIButton` instances, or its SwiftUI variant, `Button`, will fade when you tap them:



How off would things feel if that *didn't* happen? As a user, you might wonder if the tap took at all. Did it perform any action? Is the app frozen, is it a bug – is it broken altogether?

Such a simple, tiny animation that eliminates nearly all of those questions. When you tap a button that gives you no feedback, it feels wrong.

Now consider navigation. If you were to open the Settings app on iOS, and pick a row – it becomes selected. Now, if you use the swipe gesture to go back – notice how the row slowly fades out of the selected state. This is kind of hard to demonstrate with an image, so I definitely encourage you to do it now.

But what does that give us? It's informational – it's showing where you just came from, and what you did to get there. It's a tiny, tiny detail that many might miss but it bolsters the overall experience.

Apple incorporates these little touches all over the operating system, and iOS is better for it. These are the kinds of opportunities you can look for, too. And notice – none of these micro animations are long or bombastic, in fact, they are the opposite. They have a single purpose, and they aren't in the way. Endeavor to do the same.

Tips

Animation Timing is Everything

Micro animations are all in the name. They are tiny, microscopic animations. They aren't annoying and are purely additive in nature. As such, try to keep them within less than a second in terms of animation duration.

Think of the iPadOS cursor. If those hover animations, which occur when you move the cursor over an element that supports cursor APIs, took long – it wouldn't work at all. Imagine hovering over an element and then waiting two seconds for the animation to finish. I wouldn't work.

In fact, this is a good talk about animations in general. Yours are probably too long! I struggle a lot with this myself, because it's easy to sprinkle in animations, but over time they can become too much. You're always going to be battling an attention

budget the user has, and nothing makes withdraws quicker than animations. Keep them snappy, keep them quick.

Combining Micro Animation Purposes

The "toaster" banner example from before is typically used for the reinforcement of an action, but it also displays information about something that just happened. Many banner-like interactions work like this. Here's something you did, and here's what happened because of it.

For example, in Twitter, once a tweet is posted – you get a little banner letting you know it's live:



It reinforces the action I took (compose a tweet) and informs me that it's completed (it's posted). I like that.

In fact, if you peel back the reasons why you might use a micro animation, combining those reasons together for one singular purpose can lead to incredibly useful. It can for something that's delightful *and* informative, or attention-directing *and* informational.

If you think about tit, tool tips work a lot like this. Sometimes, they simply direct attention. Other times, they direct attention to a new interface element and let you know what happens when you interact with it. But the point is, micro animations can usually exist for a few reasons — it never has to be just one.

Lean on Your App's Voice

It's easy to think about micro animations strictly in terms of playfulness, but that's not explicitly true. When you think about your app's voice, what it's trying to convey and how it should make people feel — that will inform *how* you deploy micro animations. But, it's certainly nothing that should inform whether or not they should be present or not present entirely.

If you've got a serious, "all business" app — micro animations can still be useful. They just look and feel different. The conversation around them with your team shouldn't look all that different from how you'd design components. For example, buttons. All apps have buttons, and they serve the same purpose. But the styling of buttons is near infinite. So, apply that same thinking here.

When Do You Use Micro Animations?

Micro animations serve a purpose. More than anything, they are *additive*. You can like a post on social media without the resulting animations. You can tap a button without the fade, or scale. You can still do the interactions, but in these cases — they are made better by a micro animation.

Though, not all micro animations are a result of an action someone took. Sometimes, they reveal a tool tip, a banner or something else. Whatever the case may be, frame your thinking around them according to the reasons that they exist listed above. Can something be more informational, ease user anxiety (i.e. did I really delete this post?), inform navigation or just be delightful if you added a micro animation?

A lot of times, the answer is yes. Animations are awesome! They are a bedrock of mobile apps. But, a little goes a long, long, *long* ways. That's no different with micro animations. So, make sure your micro animations are typically a result of something that occurred. They are intentional. And, if they aren't, make sure they are helping your users along in some form or fashion.

Get Some Inspiration

To get going with micro animations, try to take stock of some of the ones you came across that you liked. Create a note or something with a little screenshot mentioning what the micro animations was for, and why it worked for you. This is such a simple, obvious "well of course" piece of advice, and I hear you.

But!

So often, we skip doing things like this when learning a new craft. And, I think micro animations are their own little craft. Especially when a lot of apps are coming out from one person teams, it pays to have a repository of examples when it comes time to design or implement something. This approach has helped me with micro animations a few different ways:

- I can see the "why" behind micro animations in other apps. Am I also using them in places where people might expect them to be?

- On the other hand, I can discover places where I found them to be a novelty. They didn't *need* to be there, but I was happy that they were.
- From a design sense, I can see how other designers tackled similar micro animations I'm looking at.

I try to stay away from linking to external websites in a book series that is meant to be evergreen, but I can't help myself here. If you visit https://www.reddit.com/r/ you'll find a lot of great examples to get your creative juices flowing.

Three Key Takeaways

1. Micro animations are small interactions, typically less than a second, that add value to the overall user experience.
2. They can be used to reinforce actions, be informative or simply add a little bit of joy.
3. A little goes a long way, so make sure each on you add is "earned" and fits the overall objective of the interaction.

Haptics

Previously, I touched on how iOS 16 added haptic feedback to the system keyboard. It feels *just* right, and in just about every respect – it's changed an act I perform every single day in my life (i.e. typing on a digital keyboard) into a more enjoyable act.

And, that's fairly monumental if you think about. I've had an iPhone since they were first released in 2007. I've been typing on these things for a long, long time. And yet, here is one simple tweak that has made it feel brand new while at the same time making the "old" way feel broken altogether.

Haptics personify a je ne sais quoi factor on iOS. It's hard to pinpoint why they work so well, but when deployed correctly – they can make a user experience that's been around to the tune of 15 years feel completely revitalized. When you use haptics, you are bringing in a physical response to your app and its interactions. It helps

bring a sense of physicality to what might otherwise be a rigid, digital and stale experience by comparison.

Consider Apple Pay:

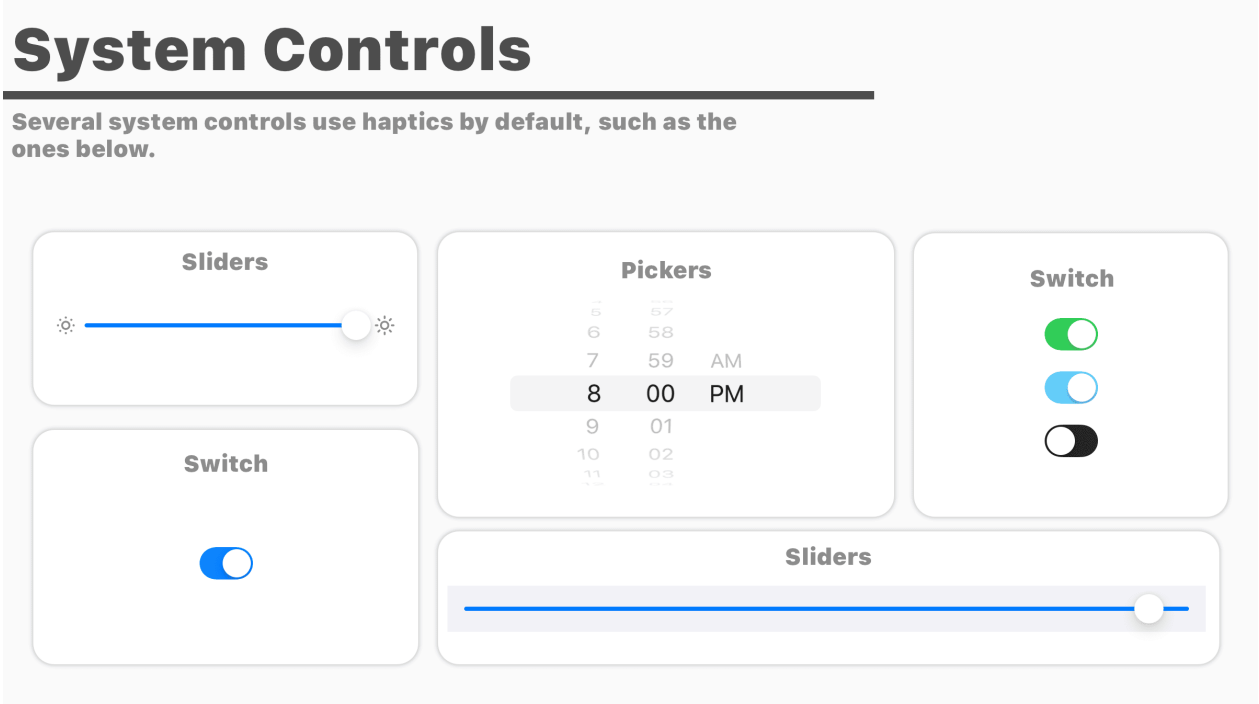


Credit: Apple Inc

When you successfully pay for an item, you get that delightful little "ding" audio along with a haptic cue. If that *didn't* happen, it would feel broken, wouldn't it? Or, if you have a MacBook which uses the force trackpad (any recent model within a few years does) you would think your clicks weren't taking if it didn't provide haptic feedback.

So, what we have are digital interactions that have existed for a long time (such as typing on a digital keyboard) that are augmented with haptic feedback to strengthen the experience. Or, conversely, in the case of the trackpad Apple has taken a traditionally motor interaction of clicking on a mouse and replaced it altogether with haptics. Now, the trick is figuring out how all of this fits with your own app.

Of course, when using any new API or designing for Apple platforms, it helps to look and see what they've done themselves. And if you take stock of Apple's apps, you'll notice that several of them use haptics. That's because their interface frameworks use them for you in several cases. Use any picker, switch or slider and you'll instantly feel haptic feedback:



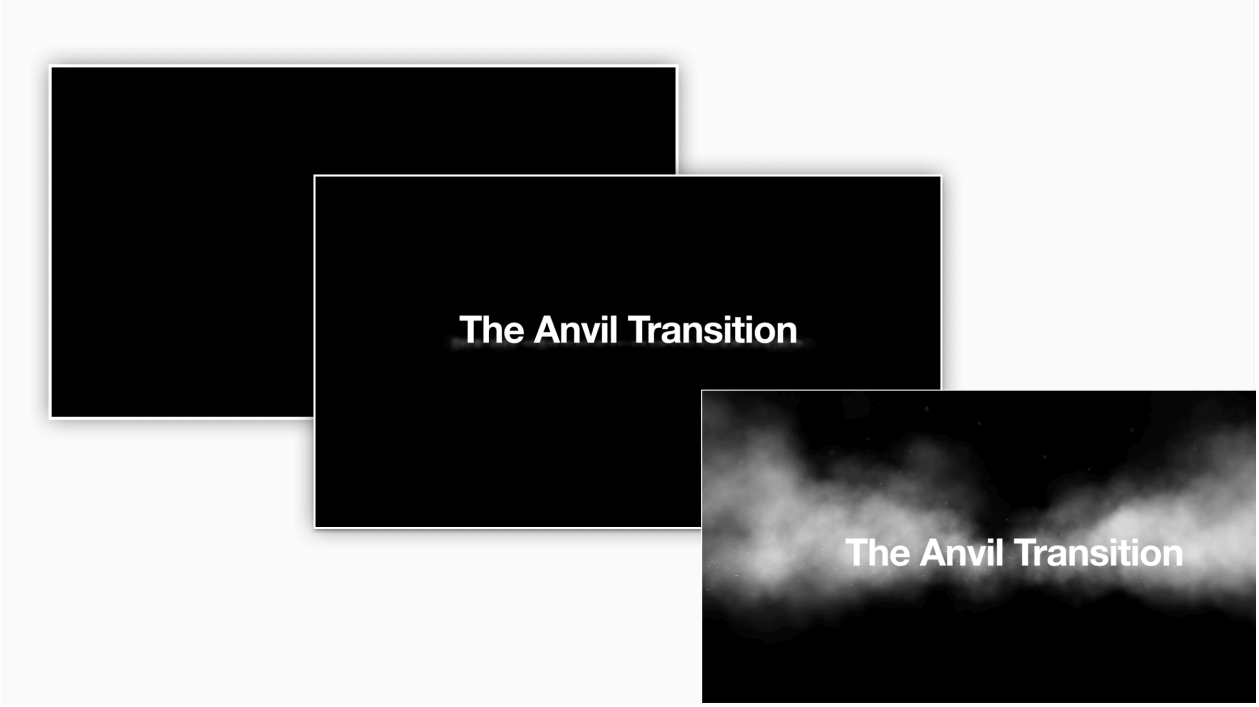
The thing is, you don't really design for haptics in a Figma, Sketch or an Adobe XD file. It's more a feel thing, and by playing around with your app during development – there might be some spots that stick out to you as good candidates. If you're having trouble finding them, though, look at how Apple has categorized their own haptics so you can have a better idea of where to deploy them. Let's look at those now.

Types of Feedback

Apple provides several haptic patterns right out of the box. Using them in your app is, thankfully, trivial. They are offered in three different categories, and each one of them has varying amounts of patterns:

Impact

This family of haptics is all about relaying physical metaphors into your user experience. For example, do you know about the classic anvil transition in Keynote? It's been used in several of Apple's own product keynotes:



That's perfect for an impact pattern. Things snapping into place, colliding with other interface elements or some arbitrary bounds or generally any real-world physical interaction that you've made digital in your app could be a candidate for impact haptics.

The patterns themselves are sensibly named, in fact — they just correspond with the amount of "force" they'll produce:

- Light
- Medium
- Heavy
- Rigid
- Soft

Whichever physical metaphor you're trying to convey will determine which one is right to use.

Notification

These are useful for letting people know the outcome of something. Did they just try to send a post on a social media app? Scan something via OCR? Whether those things worked or not, the notification family of haptics is a perfect compliment to reinforce outcomes. Once the task is completed and you know what happened, you'd play one of its three patterns:

Success

The task went was a success, nothing went wrong.

Warning

The task or interaction may have worked, or it did only to a degree. For our example listed above with OCR — maybe you'd consider this tone when *some* of the text was scanned, but you couldn't get all of it.

Error

And of course, there is the error. This pattern is a little more aggressive, but it works *so* well when you need to signal that something went wrong.

Do you have an asynchronous task, a high-impact flow or something else that users would generally benefit from having the outcome reinforced to them? If so, then the notification haptics are something you should consider.

Selection

Selection is the easiest of the three to understand and use, as it's all in the name. Any-time an interface element changes its selection (i.e. a picker for choosing a time as it scrolls through its options) – this pattern would play. There is only one to use here (the "Selection" pattern), too, unlike the other haptic families.

To get a feel for all of these, I've included a haptic playground file to demo them all out. You can find it in the demo project under `HapticsFig1View`.

Core Haptics

Where to begin with Core Haptics? It's a big topic, albeit an impressive one. Using the framework, you can create you own haptics patterns and do all sorts of amazing things with them such as reflect physical collisions, start and stop them at will and more. While the haptics we look at here are housed in UIKit, Core Haptics extends what's possible with Apple's Taptic Engine. When the stock haptics don't cut it for your use case, this is your next stop.

The reason I don't really dive into them here is most of these situations are ideal for very hands-on, interactive apps such as games. However, it's good to be aware that it exists as it is part of the haptics family on iOS. For now, though, just know that this is the place to go if you need any custom haptics pattern. Further, it's possible to do things like sync haptics with audio, provide custom haptic .json files in the form of an AHAP file (Apple Haptic Audio Pattern) and more.

For the other 99% of us, though, the stock haptics more than suffice. Beyond that, they are recognizable – which is a big plus in user experience. People already know what they represent, and as long as you use them correctly – you'll get that "muscle memory" benefit, too.

Tips

Don't Skip iPad

You may think writing your code with a check for iPad is a proactive approach to working with haptics:

```
guard UIDevice.current.userInterfaceIdiom != .pad else {
    return
}

// Setup haptics
```

But this is actually a mistake. Some game controllers support haptics, and the Core Haptics framework will thread right down to them in additions to Apple's own hardware. So if a user has a recent Xbox or Playstation controller paired via bluetooth, for example, your haptics will still play on those devices via `CHHapticEngine` if you're leveraging Core Haptics.

The Button Question

It's tempting to use haptics when people tap a button in your app. However, try to remember that Apple's `Button` view type or `UIButton` in UIKit don't do this by default

– and they don't for a reason. If every button plays haptics in your app, that's probably a bit too much.

But, there is a time and a place for it, though. Remember, it's great to think of haptics and where to use them in terms of how Apple has named their APIs. So if a button products an order, or posts a tweet (you get the idea), then a haptic pattern may be perfect to reflect the outcome.

Haptics Wrapper

Using haptics isn't painful from an implementation standpoint, but it is *just* enough to make it worth abstracting out a little bit. Since all of the haptic engines are extending a base class, you won't be initializing a common object each time you use them. The type you end up with corresponds to the haptic family you play (i.e. selection, impact or notification).

As such, I generally use something like this in each of my projects:

```
struct Haptics {
    static func hapticsSelectionFeedback() {
        let selection = UISelectionFeedbackGenerator()
        selection.prepare()
        selection.selectionChanged()
    }

    static func hapticImpactFeedback(style:
UIImpactFeedbackGenerator.FeedbackStyle, intensity: CGFloat? = nil)
{
        let impact = UIImpactFeedbackGenerator(style: style)
        impact.prepare()

        if let customIntensity = intensity {
            impact.impactOccurred(intensity: customIntensity)
        } else {
            impact.impactOccurred()
```

```
        }
    }

    static func hapticNotificationFeedback(style:
UINotificationFeedbackGenerator.FeedbackType) {
        let notification = UINotificationFeedbackGenerator()
        notification.prepare()
        notification.notificationOccurred(style)
    }
}
```

That way, any stock haptic is always one line of code away – no matter the type of haptic you need:

```
func performOCR() {
    let taskResult = OCREngine.parse()

    switch taskResult {
    case .success:
        Haptics.hapticNotificationFeedback(style: .success)
    case .failure:
        Haptics.hapticNotificationFeedback(style: .error)
    }
}
```

Make Haptics Optional

Some people may not like haptics playing all the time, and they may have them disabled on their device altogether. Even so, offering a simple option to forgo them is good taste. However your data model looks, simply stashing a boolean value away in user defaults is probably more than enough.

So, extending our wrapper above, it may look like this now:

```
static func hapticsSelectionFeedback() {
    guard AppPrefs.hapticsEnabled() else { return }
```

```
    let selection = UISelectionFeedbackGenerator()
    selection.prepare()
    selection.selectionChanged()
}
```

Three Key Takeaways

1. Haptics are an excellent tool to use when you need to bridge the gap between a digital interaction to its physical, real-world example.
2. You can use Core Audio to create your own, custom haptic patterns — but typically, the system provided haptics should be suitable for most needs.
3. Make sure your haptic use compliments interactions in your app, but it doesn't overpower them.

Undo and Redo

At the heart of this chapter lies the inconvenient truth that every user of your app will inevitably do one of two things:

1. Change their mind, or
2. Make a mistake

Somewhere throughout the process of using your app, an import piece of data gets accidentally deleted, a button receives an errant tap or they realize they didn't *actually* want to edit thing Y, but rather it should've been thing Z.

I get it, I've done the same thing in almost every app I've ever used. And, honestly, about half of those apps made a bad situation worse by not properly supporting undo and redo support. This chapter is all about making sure you're not one of those apps.

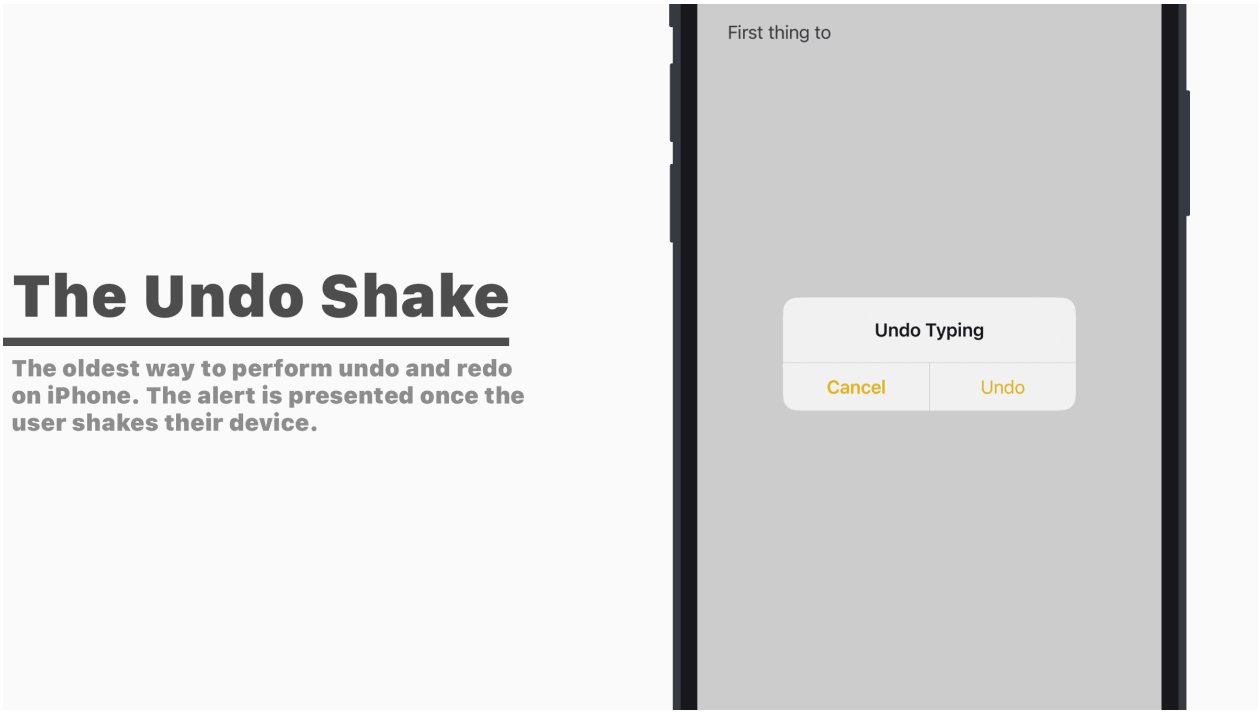
Beyond just undoing mistakes or afterthoughts, another added benefit from undo and redo support is the element of safety it brings to your app. If someone just downloaded it and is feeling their way around things, the safety guarantee of undo and redo makes things a lot less stressful. It's more than just about being a good platform citizen (which you should be) but the very existence of it promotes a sound user experience.

System Undo and Redo

Fortunately, there are system level practices for undo and redo on iOS, and they've been there since iOS 3. For many of these, if you support undo and redo with the correct APIs (which we'll touch on in just a minute) you should get most of these for free. Let's take a look at them next.

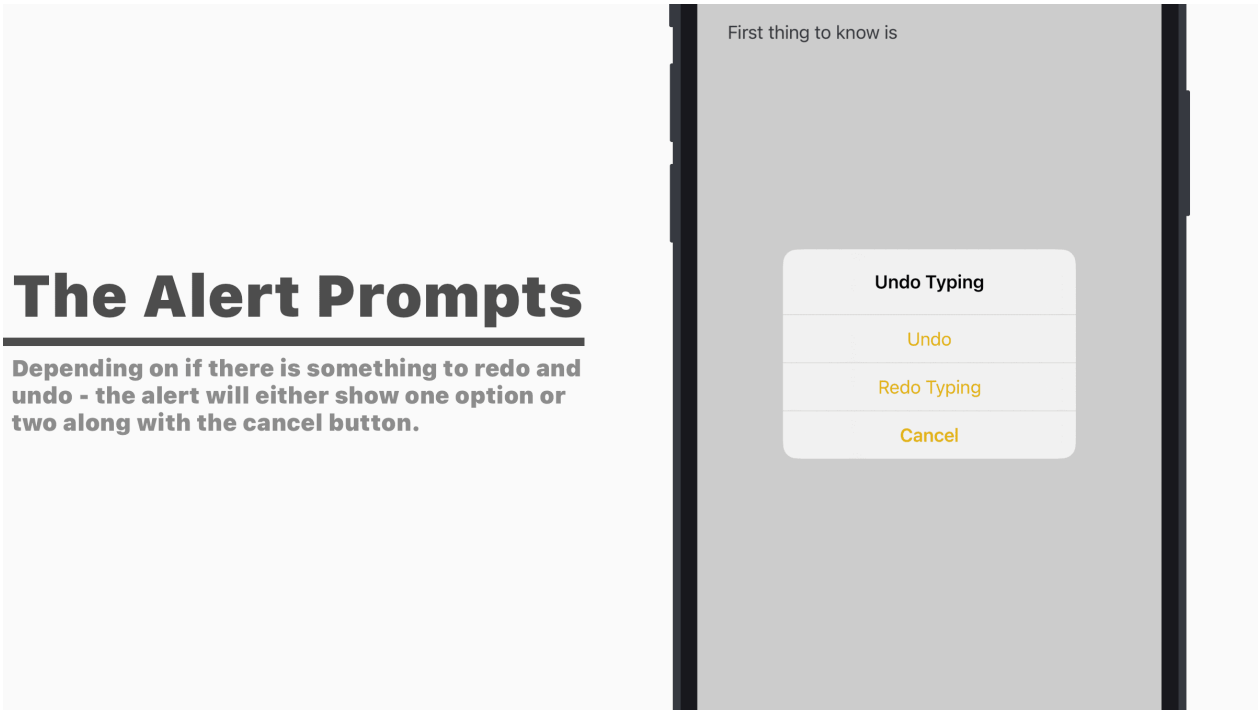
The Shake

By far the most common method of performing an undo, shaking your phone will bring up an undo prompt:



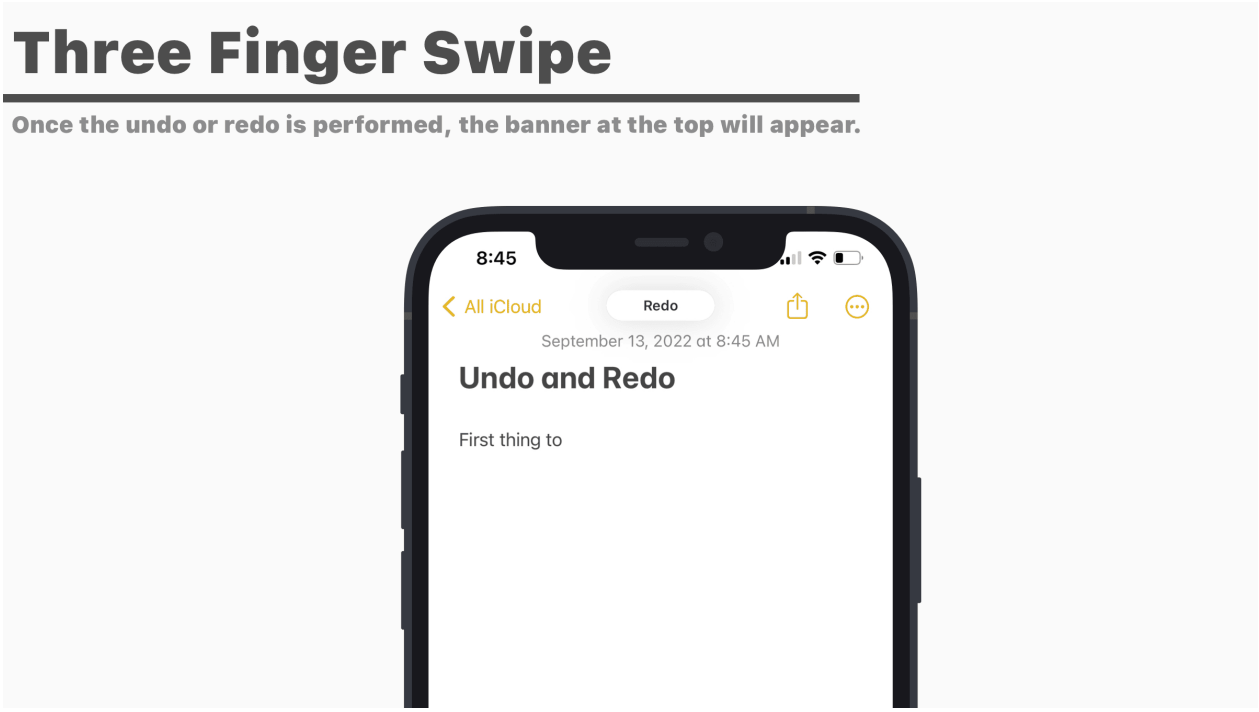
Of course, this originated from an earlier version of iOS, where the user experience wasn't quite established to perform undo and redos. It's a little wild if you think about – to undo something, the best way to get it done is to shake your $1,000 iPhone and hope for the best! Regardless, it's probably the most recognized way to perform undo and redo.

Depending on the state of your undo and redo tasks, you'll get either an alert that displays all of the options you can perform or a simple one option version:



Three Finger Swipe

Aside from shaking your iPhone, a more modern (and less dangerous) approach is to use a three-finger swipe. If you three finger swipe from the leading edge to the trailing edge, you'll perform an undo. Vice-versa, a redo:



Additionally, you'll get a little toaster banner that iOS has leaned on more and more over the years that informs you of which action occurred.

Keyboard

If your app is using any keyboard APIs, you'll get undo and redo action support along with it. The common keyboard shortcuts apply here, so ⌘+Z to undo and ⌘+R to redo. This ties into the undo and redo APIs, so you don't need to manually add keyboard shortcuts to get this to work.

Edit Menu

While text based controls show the stand edit menu on our behalf, this API is configureable for third party developers as well. By default, those undo and redo items should be included.

Bonus: Text Controls

Finally, as briefly mentioned above, text-based controls have undo and redo support built into them for you:

- `TextField`
- `TextEditor`
- `UITextField`
- `UITextView`

So, unless you're working somewhere deeper into the stack with something built off of a framework like `CoreText`, there isn't anything else you'll need to do.

Meet `UndoManager`

The `UndoManager` is essentially all you've got to get familiar with to support undo and redo on iOS. This is the API that all of the above methods use to perform undo and redo operations. The easiest way to understand how it works is to think of it as a stack, that is — you push things onto it, and pop things off. So, if you're working on a photo editing app and took some steps like this:

1. Applied new filter.
2. Cropped photo.
3. Added watermark.

Then, your undo manager would also reflect those same steps. That is, if you performed an undo — step 3 ("Added watermark") is what would be undone. Similarly, redoing actions works the same way. If were to *redo* our *undo* (I know, I know) then the watermark would be added back.

To make things simple, just remember it like this:

- Anything I register to undo gets added to the stack.
- When I undo anything, it gets popped off the undo stack, and then that step is pushed onto the redo stack.

To make all of this work, you're either working with the target-action pattern or providing a closure. I prefer the latter part, as the former is relic of the Objective-C selector days. Regardless, here's how both of them look:

Using Target-Action

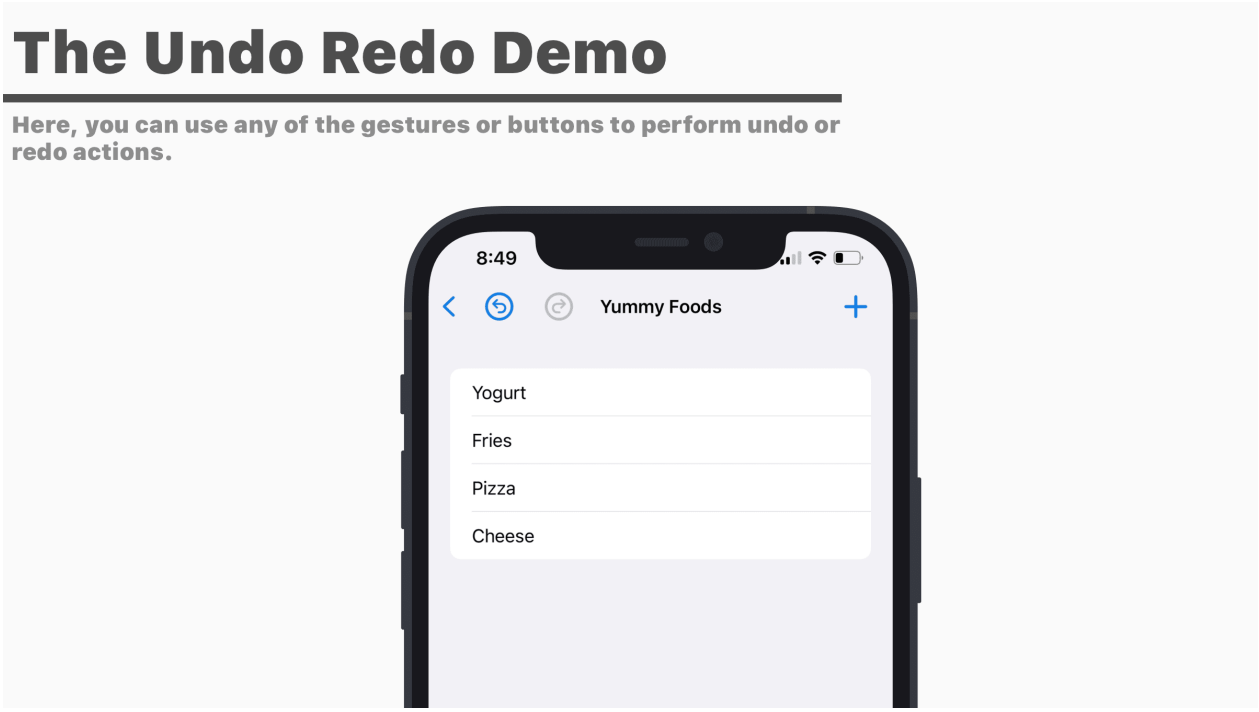
```
undoManager.registerUndo(withTarget: self,
                         selector: #selector(remove()),
                         object: self)
```

Using Closures

```
undoManager.registerUndo(withTarget: self) { [weak self] target in
    self?.remove(food: food)
}
```

Whichever route you choose, the flow is essentially the same. Identify something you'd want to undo or redo – and when that particular bit of the code is invoked, register the manager to undo or redo that particular action.

If you visit `UndoRedoFig1View.swift` (or its UIKit equivalent `UndoRedoFig1ViewController.swift`, you'll see a simple list that adds a food to it when the plus button is tapped:



For this type of thing, the undo and redo implications are obvious. If we added a food, the undo action should remove it. And, if we perform a redo – a food should be added back. That's exactly how it works if you play around with it. Try adding some foods, and undoing that and then redoing it.

The code to get it up and running is simple. For SwiftUI, I've included a reference to `UndoManager` on the data store itself:

```
// Xcode -> SwiftUI -> UndoRedoFig1View.swift
func add(food:Food) {
    undoManager.registerUndo(withTarget: self) { [weak self] target
in
        self?.remove(food: food)
    }

    foods.append(food)
}

func remove(food:Food) {
```

```
    undoManager.registerUndo(withTarget: self) { [weak self] target
in
        self?.add(food: food)
    }

    guard let foodIndex = foods.firstIndex(of: food) else {
        return
    }

    foods.remove(at: foodIndex)
}
```

As you can see, we just register undo and redo actions – and by doing so, the undo manager creates a stack of actions the user takes so it can later unwind (or rewind) them accordingly.

The story for UIKit is exactly the same except that each `UIViewController` instance already has an undo manager property, so we can reach into it directly. For example, this code looks the same but it's housed right in the view controller itself, not in a data store class as we saw with SwiftUI:

```
// Xcode -> SwiftUI -> UndoRedoFig1ViewController.swift

private func add(food:Food) {
    undoManager?.registerUndo(withTarget: self) { [weak self] target
in
        self?.remove(food: food)
    }

    var snap = self.datasource.snapshot()
    snap.appendItems([food])
    self.datasource.apply(snap)

    updateBarItemState()
}
```

In addition to performing undo and redo operations, you can also be informed when any operation is occurring, about to occur and more via notifications:

1. `NSUndoManagerCheckpoint`: Posts when your undo manager opens or closes an undo group or when querying the redo stack. It doesn't contain a user info dictionary.
2. `NSUndoManagerDidOpenUndoGroup`: Posts when your undo manager opens an undo group, which happens when `beginUndoGrouping()` is invoked. It doesn't contain a user info dictionary.
3. `NSUndoManagerDidRedoChange`: Posts when you perform a redo, it also doesn't contain a user info dictionary.
4. `NSUndoManagerWillRedoChange`: The same as above, only it fires *before* the redo occurs.
5. `NSUndoManagerDidUndoChange`: Posts when you perform an undo, it also doesn't contain a user info dictionary.
6. `NSUndoManagerWillUndoChange`: The same as above, only it fires *before* the undo occurs.
7. `NSUndoManagerWillCloseUndoGroup`: Posts when your undo manager closes an undo group, which happens when `endUndoGrouping()` is invoked. The user info dictionary might contain the `NSUndoManagerWillCloseUndoGroupNotification` key with a `NSNumber` boolean if the undo group is discardable.
8. `NSUndoManagerDidCloseUndoGroup`: The same as above, only it fires *after* the group is closed instead of before.

Note that for all of these, the object that the notification returns is your undo manager responsible for the invocation. As with some classic Foundation-based notifications, getting these up and running requires fully qualifying them at the call site in Swift:

```
let checkpoint: Notification.Name =
Notification.Name.NSUndoManagerCheckpoint

let nc = NotificationCenter.default
nc.publisher(for: checkpoint).sink { value in
    // use notification
}.store(in: &mySubs)
```

Tips

Grouping Operations for "Undo All"

You can leverage a concept called "grouping" to undo and redo entire sets of actions. To begin implementing this, you'll need to know about `groupsByEvent`, which is true by default. This is typically how you'll want undo and redo to work, as the manager will handle grouping for you which results in the standard undo and redo operations.

However, if you wanted to undo or redo quite a lot of operations, then you can start opting into grouping actions by calling `beginUndoGrouping()`. Once the group has been undone, though, you need to balance that call with `endUndoGrouping`. If you don't, you'll have an exception thrown by the system (`NSInternalInconsistencyException`).

To see this in action, check out `UndoRedoFig2View` or its UIKit counterpart. Here, there is an "Undo All" bar button item. If you add several foods to the list and tap it, you'll see that *all* foods are removed. This is accomplished by beginning grouping:

```
// Xcode -> SwiftUI -> UndoRedoFig2View.swift

.task {
    beginGroupingUndoActions()
}
```

And then, once the "Undo All" is tapped – everything in the undo group is undone:

```
// Xcode -> SwiftUI -> UndoRedoFig2View.swift

withAnimation {
    store.undoManager.endUndoGrouping()
    store.undoManager.groupsByEvent = true
    store.undoManager.undo()

    beginGroupingUndoActions()
}
```

Making Undo and Redo Predictable and Understandable

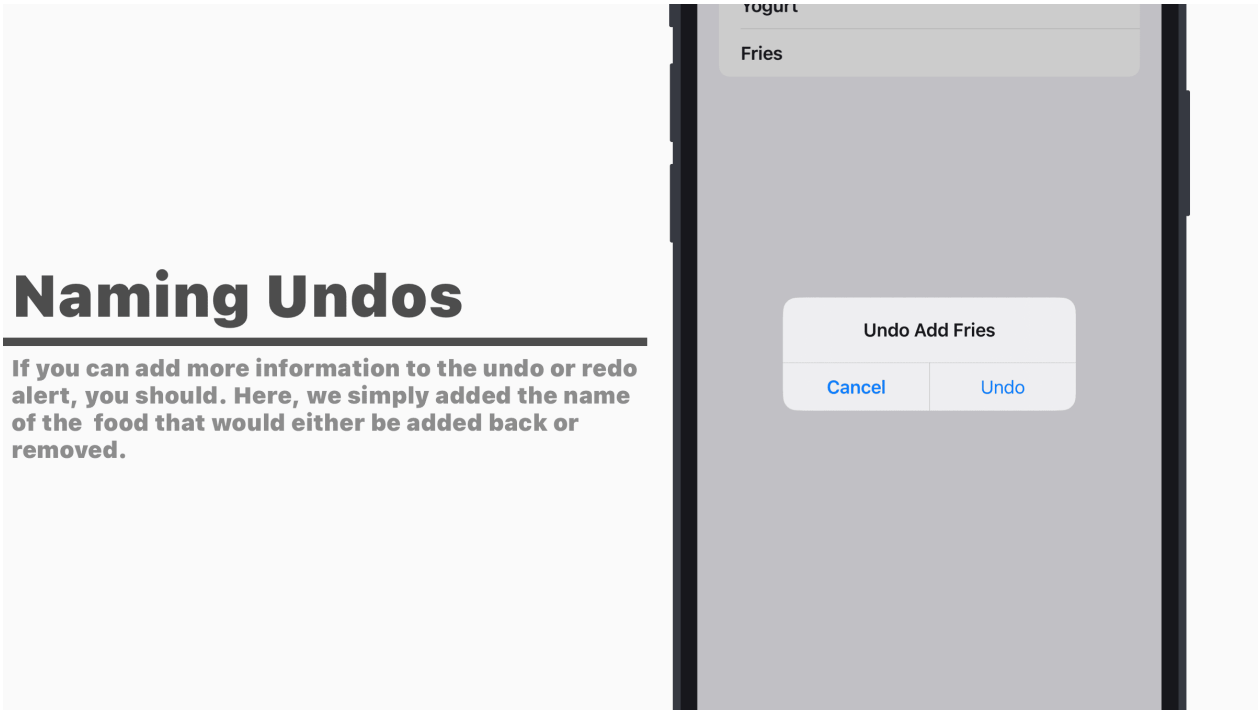
Undo and redo should be both *predictable* and *understandable*. You don't want guessing games taking place when people are using undo and redo mechanics. There are few ways you can help achieve those goals.

One way is to name things accordingly. For example, for our food scenario – we should include the name of the food and whether we are removing it or adding it. Look at a simple change to our functions we've been using (the first line in the function was added):

```
// Xcode -> UIKit -> UndoRedoFig3ViewController.swift

private func add(food:Food) {
    // Set a name for the action
    undoManager?.setActionName("Add \(food.name)")
```

```
    undoManager?.registerUndo(withTarget: self) { [weak self] target
in
        self?.remove(food: food)
    }

    var snap = self.datasource.snapshot()
    snap.appendItems([food])
    self.datasource.apply(snap)

    updateBarItemState()
}
```

Now, if the shake is performed to undo – instead of just a "Undo" string, we'll get something like "Undo Add Pizza", or whatever the name was:



The same is true of the edit menu too, as there are functions to change those titles as well:

```
undoManager?.redoMenuTitle(forUndoActionName: "Remove \(food.name)")
```

Further, a nice touch is to call out the part of the interface that changed as a result of an undo or redo. Highlight a row, scroll to the text operated on, scale or pop the photo – the list goes on. Whatever was added or removed should be ideally called out in your interface. This makes it incredibly clear what the result of the undo or redo was.

Take `UndoRedoFig3View.swift`. When you redo an action (i.e. add a food back to the list), now that row is briefly highlighted to make it clear that the row was added as a result of performing the redo:

```
// Xcode -> SwiftUI -> UndoRedoFig3.view.swift

// Abbreviated for clarity. See the file for the full
implementation.

// Track the last food added by a redo
@State private var readdedFood: Food? = nil

// Conditionally show a highlighted row if it was
// The food added from a redo
List(store.foods) { food in
    Text(food.name)
        .listRowBackground(food == readdedFood ?
Color.gray.opacity(0.25) : nil)
}

// Track the last food added as a result of a redo
Button {
    withAnimation {
        store.undoManager.redo()
    }

    let animationDuration = 0.25
    withAnimation(.easeOut(duration: animationDuration)) {
        readdedFood = store.foods.last!
    }
```

```
    DispatchQueue.main.asyncAfter(deadline: .now() +
animationDuration) {
        readdedFood = nil
    }
} label: { /* View code }
```

Responding to the System Undo Redo Shake

Earlier I mentioned how integrating with the undo manager opens up a lot of system support for undo and redo. That's still true, but in the case of the classic "undo shake", you've got *one* more step you might have to take.

Since the undo and redo alert is the result of a shake, you need ensure your view controller can become first responder. When the shake happens, the first one in the responder chain that can become first responder will handle it. Doing so is trivial, and we did so in `UndoRedoFig3ViewController.swift` already:

```
// Xcode -> UIKit -> UndoRedoFig3ViewController.swift

override var canBecomeFirstResponder: Bool {
    true
}
```

With that in place, you can get the undo and redo alerts showing. Note that for SwiftUI, unfortunately you'll need to wrap up your view in a view controller representable and override the first responder property above. You can't manually trigger these undo and redo alerts, so simply using a notification to see when your phone shakes isn't enough (though you *could* manually show an undo and redo alert that you provide in such cases).

How Do You Decide What To Undo and Redo?

More of a philosophical question, but an important one: what exactly should you redo and undo? I think it boils down to the context of the screen. If you've designed them correctly, they mostly do one primary task. You're editing a photo, creating a recipe or constructing some sort of post. Any operations related to those things should participate in undo and redo in my opinion.

Plus, since undo and redo is based on stack based operations, you can mix and match things quite easily. It doesn't matter if you add a thing, rename another or change some sort of data. You can leverage the same undo manager for each of those.

In short, don't overthink it. I think undo and redo is a case where you should have it and not need it, rather than a user possibly needing it and not having it.

From a technical standpoint, though — be sure to consider the state of your data. For example, something could still be in the undo stack that might not exist anymore. If you deleted something, but a sync service added it back or the data was changed by some other means you need to be sure to check that whatever it is you're operating on still exists and makes sense to actually undo or redo.

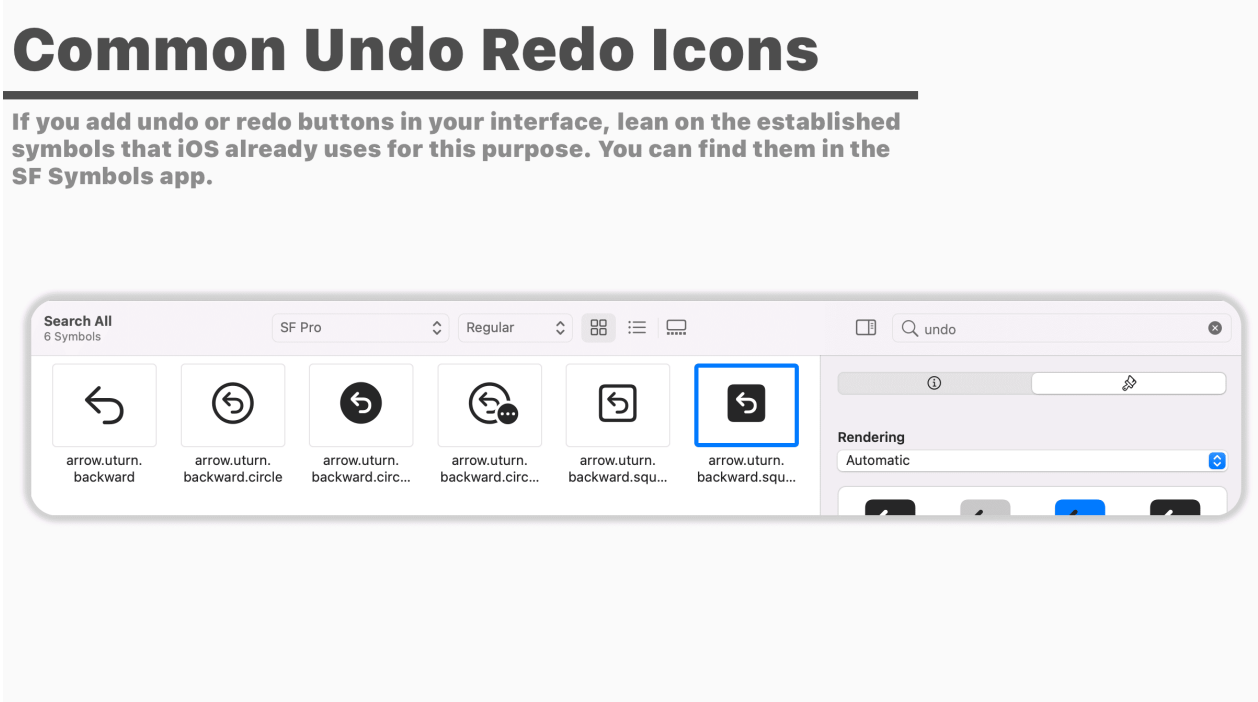
Using Interface Adornments

I've seen compelling arguments to include or exclude interface buttons for performing undo and redo operations. Apple says to include them "only when necessary", while other apps (such as photo editing ones) include them front and center. Apple's PencilKit interface has them showing by default, too.

So it is with anything software related — it's all about context. Sure, it would be grand if everyone knew how to perform system gestures or shakes for undoing and redoing stuff — but the fact is, many of them don't. So, if you can add an affordance for

undo and redoing actions (especially if what you can undo and redo is a high impact action) then, why not?

If you do add them, though – it's not a time to get fancy. Simple rely on the SF Symbols for doing so. Just search for "Undo" or "Redo" in the SF Symbols app to get an idea of what to use:



Plus, you'll want to stick them in familiar spots too – ideally a navigation or tool bar.

Binding State to UndoManager

Continuing with the point above, if you have buttons that fire off undo and redo operations, you likely want to enable or disable them depending on whether or not there are things to operate on. You never want a button to be enabled when tapping it doesn't do anything – that's confusing.

To that end, you may be tempted to bind the state of the button to some sort of data model you have. For example, for our food list scenario, in SwiftUI you might do something like this for the undo or redo button:

```
.disabled(!store.foods.isEmpty)
```

The thinking might be that if there are no foods in the list, then certainly that's nothing to *undo* – but a better way is to peek into the undo manager itself to see if there's anything on the undo or redo stack:

```
.disabled(!store.undoManager.canRedo)
```

If it can undo or redo anything, the relevant properties will report it via `canUndo` or `canRedo` making them ideal to use for controlling the state of interface buttons. While undo and redo almost always deals with your own data model, the actual source of truth of what can be undone or redone isn't your store – it's the undo manager.

Disabling System Undo and Redo Gestures

I don't think it's ever a good idea to override system gestures such as the undo and redo provided ones, but if you have to for interactive reasons (such as a game or some other out of the box interaction) you certainly can at the `UIResponder` level. By returning `.none` for `UIEditingInteractionConfiguration` – iOS will disable all of system provided three finger editing gestures:

```
override var editingInteractionConfiguration:
UIEditingInteractionConfiguration {
    return .none
}
```

However, note that the implications of this could change over iOS releases. If Apple expands upon these editing gestures which are dictated by this property, you'll be opting users out of those, too.

Three Key Takeaways

1. Supporting undo and redo shouldn't be optional, it should be table stakes to any version you ship of your app.
2. Figuring out what should be undone and redone is critical, as the result of each operation should be expected before they actually happen.
3. Even though the APIs for undo and redo support are showing their age, it's still important to implement them alongside Apple's modern technologies.

Context Menus

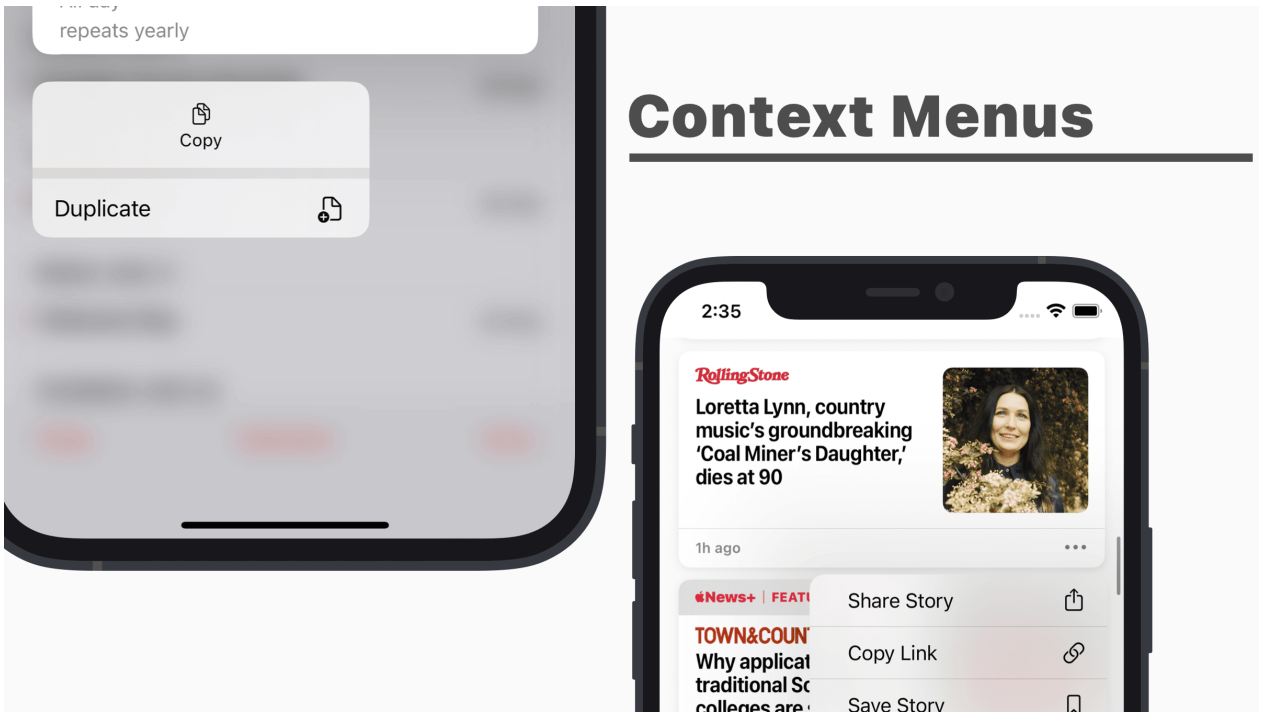
Starting with iPhone 6S, Apple introduced 3D Touch. It was a phenomenal blend of software and hardware working together. Using pressure sensitivity on the iPhone model which supported the feature – iOS could track how hard someone was pressing on the display. Simply put, it knew the amount of force being applied.

Depending on that force, certain interactions would activate. Links would preview, photos would pop up – and if the force was increased, those destinations would fully "open" and they would be directed to that item. Apple called that interaction peek and pop.

I absolutely loved it, and I think it was some of Apple's best work. But, as you probably know, Apple killed 3D Touch some time ago.

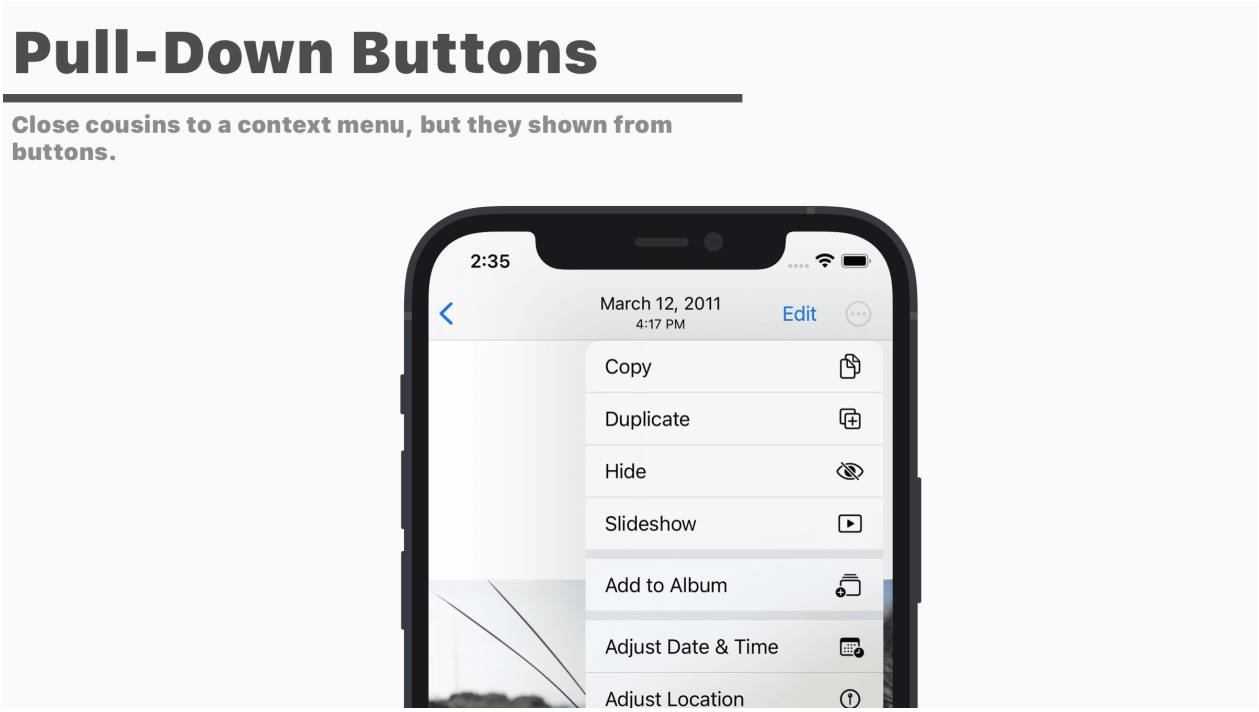
The reasons why vary, and everyone has an opinion on it – but there was something to that interaction, and parts of it have prevailed on iOS. Debuting on the iPhone

XR, tapping on things and holding now, for the most part, is analogous to showing "more" within your interface. Apple has dubbed this as "Haptic Touch" – and to show those "more" parts, we've got context menus.



It's easy to draw parallels between context menus and right clicking on macOS, their interaction and purpose are virtually the same. In fact, if a keyboard and mouse are attached to iPadOS – that's exactly what happens. A right click activates a context menu.

However, their utility doesn't just lie within right click interactions or showing more options for things. While they do work wonderfully for that purpose, you can also leverage them in *pull-down buttons*. In this interaction, a button shows a context menu, and in many cases - they can replace what were once entire view controllers or views. Apple has gone this direction in several places within iOS:



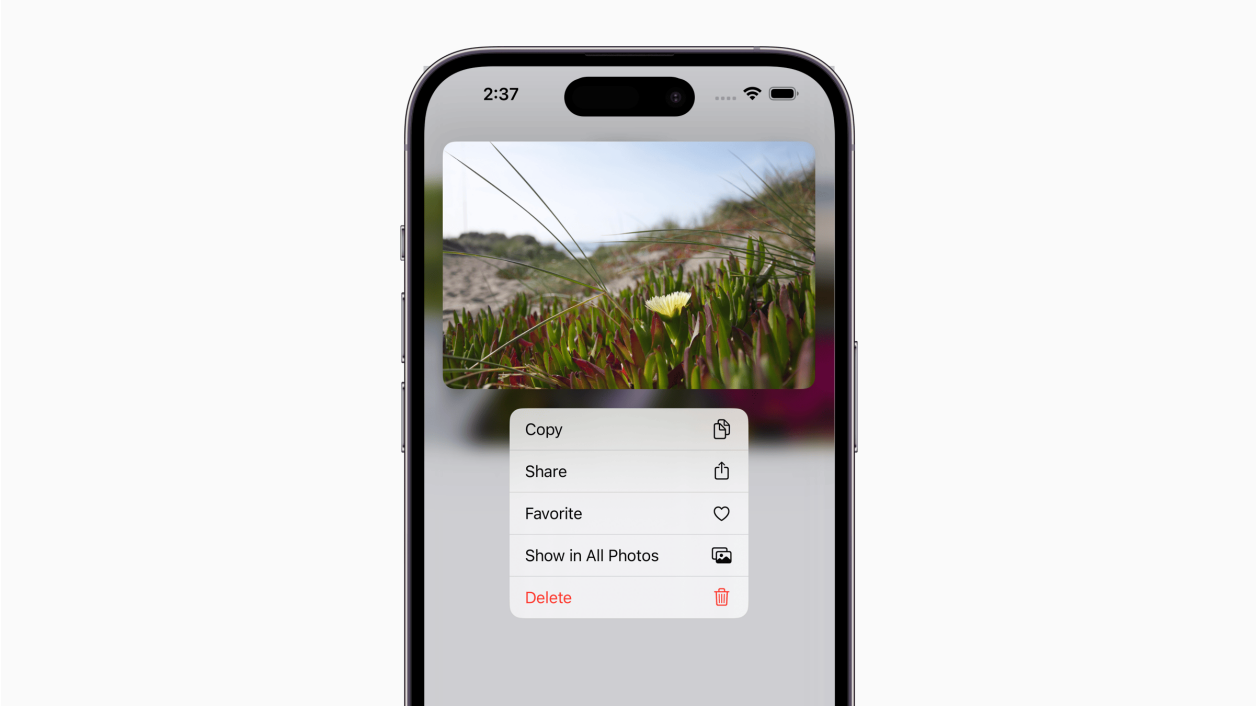
I should stop here and note the differences about these two things, because this is where it can get a little confusing. Under the hood, Apple uses the same API to achieve these interactions. But, they aren't technically the same. In fact, calling this chapter "Context Menus" is itself even a bit of misnomer, since we're talking about context menus *and* pull-down buttons. Two very similar interactions using the same APIs, but are distinct in their purpose.

I think it makes sense to cover them both here, though, as the concept of "menus" is relatively new to iOS, but it is used heavily throughout. So, for clarity – here's how you can think about them:

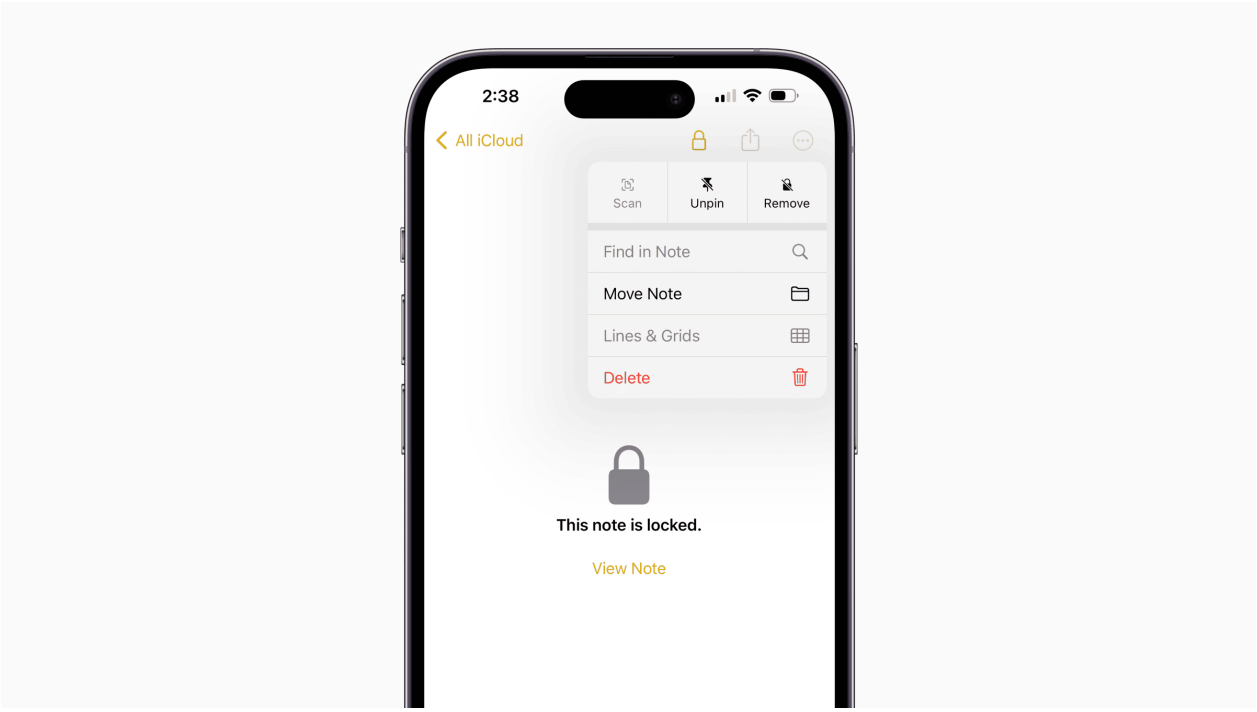
- A context menu extends functionality of a related interface element. Again, think of it as your "right-click" interaction. It can also preview a destination to navigate to.

- A pull-down button displays a menu of actions that directly relate to either the button that presented it, or the screen at large where it's contained.

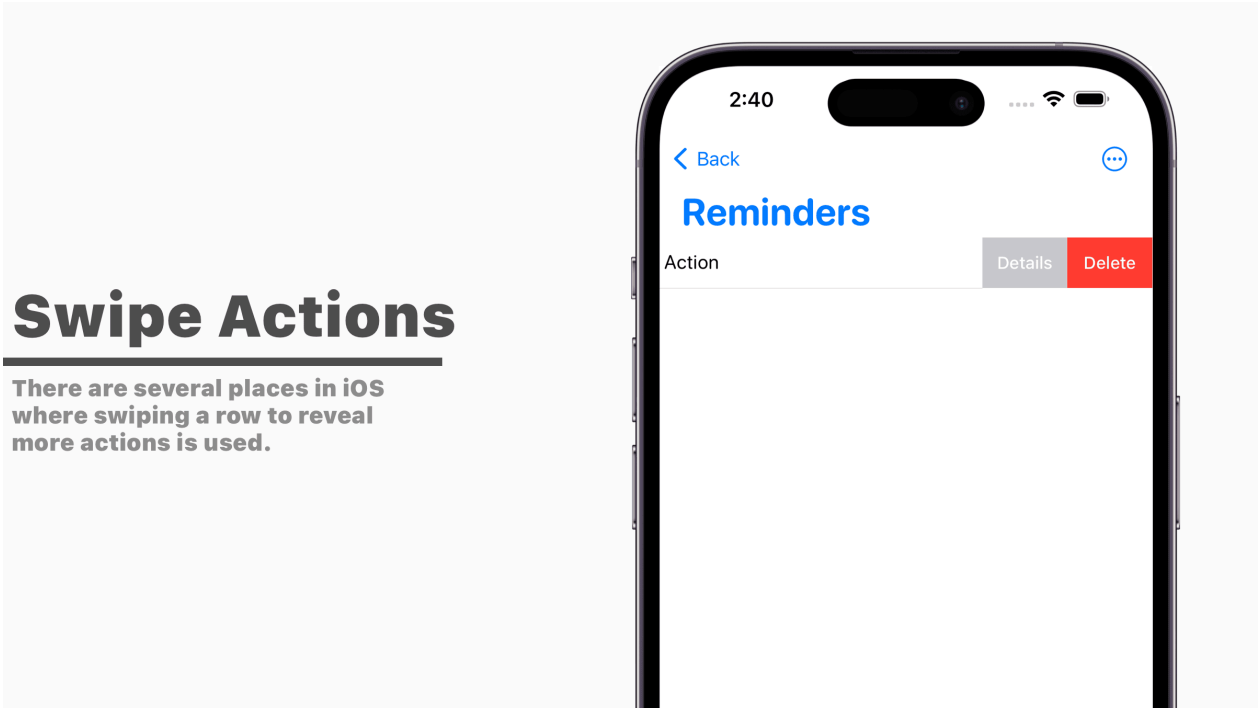
Here, a context menu helps someone do more with the image in the Photos app they've interacted with:



While in Notes, the common three dot ellipses button provides more options for the entire note itself from a pull-down button:



In short, these two interactions can be provide a lot of functionality that was hard to come by in the earlier days of iOS. Where would you put all of the options that a context menu affords? The answer used to be, at least in part, trailing and leading actions of swippable rows in table views:



But, even though that interactions is still supported, those actions are nearly always available in a context menu anyways. To me, that's a hint that menus are the direction iOS is trying to head.

Navigation Previews

Something specific to context menus, however, is the ability to preview where you can go as a result of the context menu presentation. It may seem a bit odd, at first, to do something like this. After all, a context menu extends the abilities of the item that presented it. But, often these items lead to a navigation destination anyways – so the paradigm starts to fit together a little more when you think of it that way.

Again, let's travel back to Photos. When you long press on an item and get the context menu presentation, the photo itself also shows up at the top. We saw this in one of the images above, but for a refresher:



Apple typically refers to this as a *previewing destination*. If you tapped on that photo, the context menu preview would go there and open up the view. Going back to our peek and pop discussions – this is basically *that*.

In UIKit, it's exposed via the `UIContextMenuConfiguration` . Where you use it, though, depends on a few things. If you're using it from a custom location, then you'll construct it yourself via the `UIContextMenuInteractionDelegate` protocol. Using it, you can control the entire stack and even animate the control presenting the context menu alongside its presentation any way you see fit.

More commonly, though, you'll be using it from a table or collection view. Both of these have extensive support for presenting context menus, saving you from having to deal with `UIContextMenuInteractionDelegate` altogher. For example:

```
// Xcode -> UIKit -> ContextMenusFig1ViewController.swift
```

```
func collectionView(_ collectionView: UICollectionView,
contextMenuConfigurationForItemAt indexPath: IndexPath, point:
CGPoint) -> UIContextMenuConfiguration? {
    if let place = datasource.itemIdentifier(for: indexPath) {
        return UIContextMenuConfiguration(identifier: nil,
previewProvider: {
            return PlaceViewController(place)
        }, actionProvider: { suggestedActions in
            let mockAction = UIAction(title: "Edit", image:
UIImage(systemName: "pencil")) { _ in

            }
            return UIMenu(title: "", children: [mockAction])
        })
    } else {
        return nil
    }
```

In the `previewProvider` closure, we're able to produce a view controller that shows where the interaction will take you. It previews the destination. You can do the same thing in SwiftUI, too:

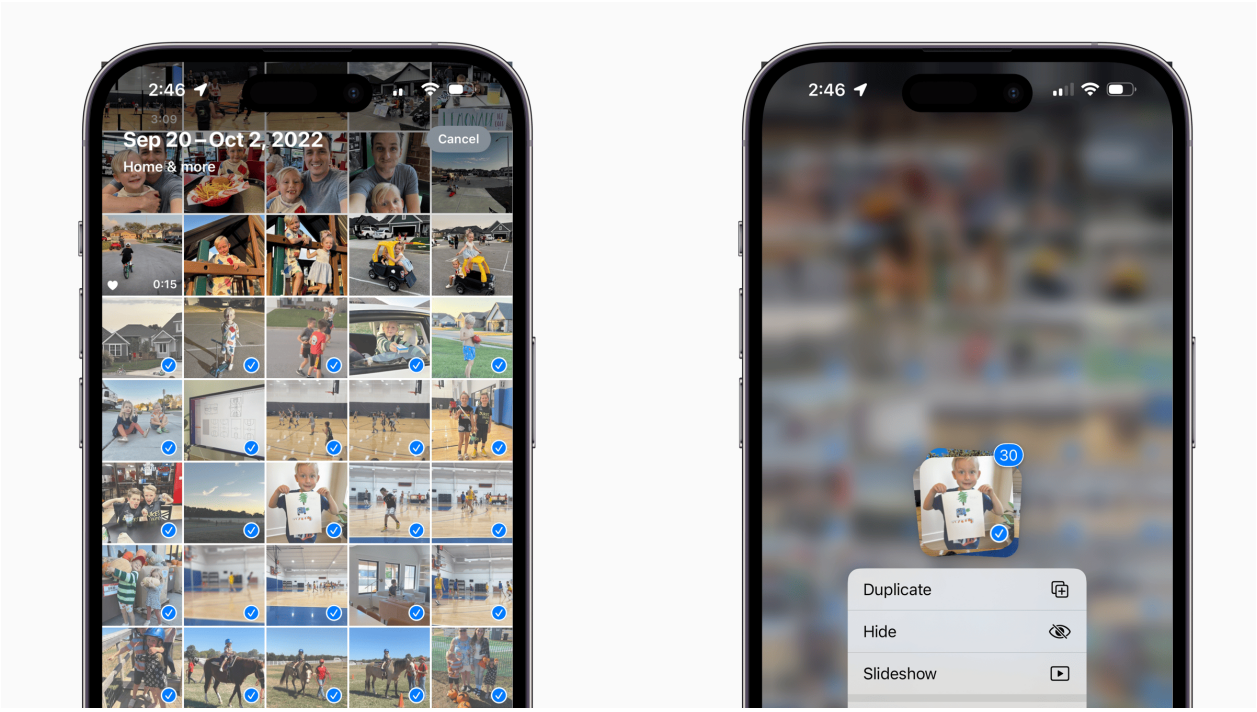
```
// Xcode -> SwiftUI -> ContextMenusFig1View.swift

List(Place.places()) { place in
    NavigationLink(destination: PlaceView(place:
place)) {
        Text(place.name)
        .contextMenu {
            Button {

            } label: {
                Label("Edit", systemImage:
"pencil")
            }
        } preview: {
            PlaceView(place: place)
        }
```

```
    }
}
```

Here, the `.contextMenu` modifier has a variant that accepts a `preview` closure, which, like UIKit, expects you to return a preview of the destination. This technique is a great way to offer many options for the item someone is interested in, while offering navigation capabilities at the same time.

UIKit offers several variations of context menu configurations depending on the control you're using. For example, exclusive to collection view in iOS 16, you can operate on several items at the same time while getting a system provided animation and preview destinations as well by implementing `collectionView(_ collectionView: UICollectionView, contextMenuConfigurationForItemsAt indexPaths: [IndexPath], point: CGPoint) -> UIContextMenuConfiguration?`.

You can see this in action in Photos, for example, by selecting several photos and presenting a context menu from any of the selections. The end result will look something like this:



While there is a lot of API for context menus and pull-down buttons, it also means there is an avenue to present them from nearly any context in both SwiftUI or UIKit.

Tips

Consistency is Key

Simply put, don't make context menus available in some places, but not others. Defining where a context menu should be available isn't too hard, but you'd be surprised at how many apps have a list of data in one view with rich context menu support – and then in the very next screen, it's the same story with *no* context menu support.

That's frustrating. If you have any options that an item could benefit from – give it a context menu, too. If the item itself has no data that can be operated on, then that's another story. For example, if you had a view with version information inside a table

view — it's probably clear that tapping on those rows wouldn't do anything, and I don't think anyone would expect them too. So, there's nothing to use for a context menu here.

But a lot of data isn't like that, even single views or components probably have something that can be changed, edited or otherwise manipulated about them. Those are situations where you should consistently support context menus.

Consider Hinting at Context Menu Options

Context menus are most likely table stakes for someone reading this book. We know they exist, and we know they are a common interaction. However, many people have no idea that they exist. For that reason, I recommend showing an alert with some copy along the lines of "Tap and hold on any (item/note/tag/etc) to (copy/paste/etc) and more."

Again, it's up to you to know your audience and who primarily uses your app. If it's a developer tool, for example, it's likely most of your user base already knows about context menus. If it's more generalized, though, you'd be surprised at how many times they can be glossed over.

I also want to reiterate that showing this alert once is probably enough. Once people know that you can do it one place, it stands to reason that they should expect it to work in other places in your app, too. That's yet another reason to be consistent with your context menu implementation.

Delay Presentations from Menus

There is a little quirk with context menus that bugs me, and it has to do with presentations. If, for whatever the reason, you decide *not* to use preview providers, but do have

some option that *results* in navigation – then I recommend considering a delay before you kick it off.

It takes the system, give or take, about a third of a second to dismiss a context menu. So if the result of the context menu is that another view is pushed onto a navigation stack or presented modally, I think it looks better to let the system fully dismiss the context menu *and then* do your presentation. If you don't, then you get this odd mix of the context menu dismissing itself while your navigation presentation happens at the same time, and to me – that looks weird.

This is something you've got to try yourself to decide if it's something you want to do. I've got examples showing it in both SwiftUI and UIKit, just visit either of the Figure 2 files. They both open another screen via a context menu, and there are two options:

1. One to do the presentation with the delay, which I prefer.
2. And, one that does it with no delay.

If you fall on my side of the argument, doing this is quite easy – just use Grand Central Dispatch to delay your presentation and you're good to go. I've included the relevant parts from both interface samples to show you the basic idea:

```
// Xcode -> UIKit -> ContextMenusFig2ViewController.swift

let open = UIAction(title: "Open Delay", image: UIImage(systemName:
"arrow.up.forward")) { _ in
    DispatchQueue.main.asyncAfter(deadline: .now() + 0.33) { [weak
self] in
        self?.present(FoodViewController(food), animated: true)
    }
}
let openNoDelay = UIAction(title: "Open No Delay", image:
UIImage(systemName: "arrow.up.forward")) { _ in
    self.present(FoodViewController(food), animated: true)
```

```
}
return UIMenu(title: "", children: [open, openNoDelay])

// Xcode -> SwiftUI -> ContextMenusFig2View.swift

.contextMenu {
    Button {
        DispatchQueue.main.asyncAfter(deadline: .now() + 0.33) {
            self.selectedFood = food
        }
    } label: {
        Text("Open Delay")
    }
    Button {
        self.selectedFood = food
    } label: {
        Text("Open No Delay")
    }
}
```

Again, I'd open up the demo files in the included project to see the difference.

So, What Should Go in a Context Menu?

Do you have a junk drawer in your house? Most of us do.

Mine? It's right by the refrigerator in the kitchen. If you were to open it up, there is literally no telling what you can find. Tape, pens, my kid's homework assignments, drawings – and that's just a preview. Simply put, it's got it all and it's got a little *too much* going on.

This is the antithesis of what we want context menus to become. A junk drawer that has *everything*, simply because it can hold just about *anything*.

It's for the same reason that I personally think that the classic "hamburger menu" leads to bloat in iOS apps – it's usually a wave of the white flag that there is simply too

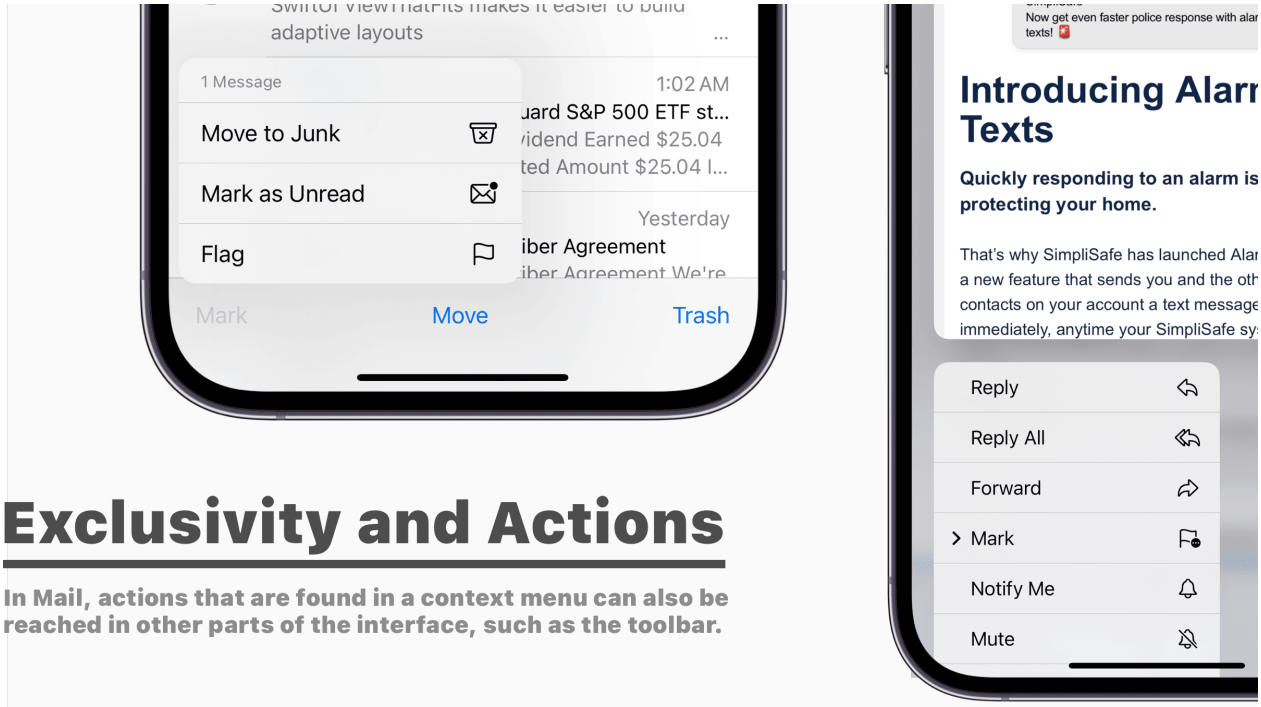
much going on, and that the developers have run out of room to put things. It's a junk drawer. That's not *always* the case, I've seen a few done well, but I've also seen a lot more done wrong.

When it comes to context menu, deliver high value. What actually helps people get meaningful stuff done? You don't want a laundry list of actions, instead, opt for mission-critical features. I think about context menus as less of an advanced power user thing (even though I did just concede that many power users know about them, and the average iOS user may not) but more of a helpful expectation people know my apps will support.

Context Menus and Exclusivity

It's easy to have exclusive actions in context menus, but you shouldn't default to thinking that way. These should be shortcuts to actions that could be achieved by other means, too. If you look at the desktop, this holds true. Open up almost any macOS app and right click on something. I'd bet that you can do every single one of those things by other interactions.

For example, check out Mail. If you pop up a context menu for an email, you might get several actions you can use. But, if you were selecting multiple emails (or even just one for that matter) – the toolbar has most of those actions available, too.



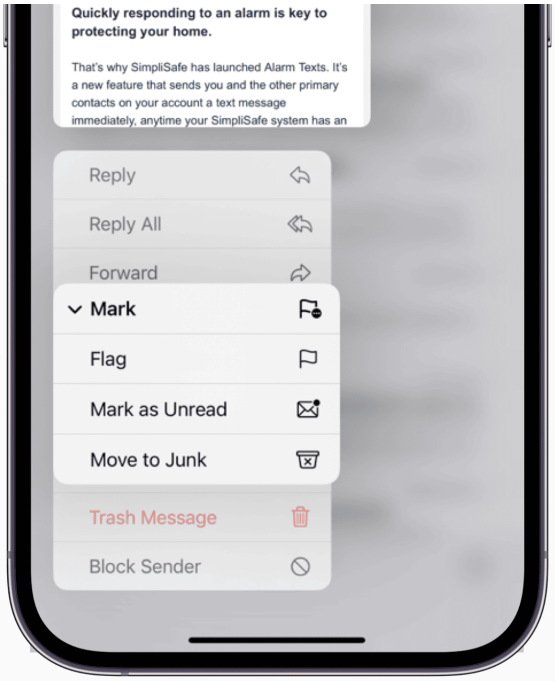
So, try not to hide things away in a context menu. Context menus are wonderful, but at the end of the day they are a means to quicker actions or navigation, not the *sole* means of those things.

Limit Submenus

You can nest a context menu within another context menu. You can see that here in Mail with its “Mark” action:

Submenus

If there is a logical set of actions that can expand on an original pattern, such as "Mark" in Mail, you can expose those in a submenu of another parent menu.



These are super helpful, but they do create a navigation hierarchy and get a little messy if you don't limit them. For this reason, I wouldn't go beyond one nested sub menu.

If you find yourself with more actions that logically make sense to expose in a sub-menu beyond one level, then it's likely a sign that the context menu is pulling too much weight. Beyond being a little confusing, and quite honestly not too pretty on the eyes — at that point, it's unlikely that the action is even exclusively related to the item which presented the original context menu — so the typical pattern expected of them is broken anyways.

Limit Dividers to 3

A separator is great for providing a visual hierarchy of the things a context menu can do. It really provides a "quick scan" flavor to them, as we logically switch contexts

when we see things divided out into separate groups. To figure out what should go where and how you might divide them, put actions into similar "buckets" of patterns.

If you open Figure 3 for either SwiftUI or UIKit, you'll see the list of foods we've been working with. If your actions available are:

- Mark as Favorite
- Edit
- Duplicate
- Delete

Then you've got three distinct patterns by my estimation. I would separate them out like this:

Action	**Pattern**
Mark as Favorite	Data Specific Flow
Edit	Editing Flow
Duplicate	Editing Flow
Delete	Destructive

So, your context menu might end up with two dividers. In SwiftUI, it's trivial to include those separators, you simply use a `Divider()` when constructing the context menu:

```
// Xcode -> SwiftUI -> ContextMenusFig3View.swift

.contextMenu {
    Button { } label: { Text("Make as Favorite") }
    Divider()
    Button { } label: { Text("Edit") }
    Button { } label: { Text("Duplicate") }
    Divider()
    Button(role: .destructive) { } label: { Text("Delete") }
```

```
}
```

In UIKit, you've got a little more to do – but it's still fairly easy, just not as obvious. There, you'll want to use a `UIMenu` with the `.displayInline` option while passing the items above it within the `children` argument. Here's the same result above, written in UIKit:

```
// Xcode -> UIKit -> ContextMenusFig3ViewController.swift

let topActions = [
    UIAction(title: "Mark as Favorite", image: nil, handler: { (_)
in
    })
]
let divider = UIMenu(title: "", options: .displayInline, children:
topActions)
let middleActions = [
    UIAction(title: "Edit", image: nil, handler: { (_) in
    }),
    UIAction(title: "Duplicate", image: nil, handler: { (_) in
    })
]
let secondDivider = UIMenu(title: "", options: .displayInline,
children: middleActions)
let bottomAction = UIAction(title: "Delete",
attributes: .destructive) { (_) in

}
let items = [divider, secondDivider, bottomAction]
let menu = UIMenu(title: "", children: items)
return menu
```

However, much like with our discussion over nesting submenus, you'll want to be conservative here. Generally speaking, I don't think going beyond about three sections is very useful anymore – that gets a little overwhelming. Remember, context menus are

about quick, snappy shortcuts to high-impact actions. You want people doing important things quick, not taking three to five seconds to read over a table of contents of actions.

Don't Default to Titles

While it's easy to add titles to context menus or pull-down buttons since most of their initializers have a title parameter as the first argument – don't default to this. Context menus and pull-down buttons should really have all of the context they need by virtue of the element they were presented from.

However, there are two situations where a title can help. The first is when you've got multiple things selected, and then you present a context menu. In this case, a title can help drive home the fact that the user is operating on more than one item at a time. You can see this in Mail – select multiple emails, and then tap "Mark" in the toolbar. You'll see it presents the context menu with a title now, where it otherwise wouldn't, "3 messages selected."

The other stems from our discussion above over submenus. In those cases, a title can help as well, since it's technically a form of navigation. You tapped a title that presented another menu, but now that button with the title that you tapped is covered up. A title within the presented submenu can help with way finding, and remind the user of what the options presented really correspond to.

Using Context Menus to Add or Create App Specific Data

Context menus can present in blank spaces, and as such – you can use a paradigm (especially on iPadOS) that macOS uses heavily when a keyboard is attached. That is, presenting a context menu with the ability to add app-specific data when presented from a context menu presented from white space.

If you right click in most macOS apps, one of the options typically ends up being "Add New (Note/Tag/etc)" and you can do the same for iPadOS. Whether it's a long touch or right click from the mouse, it's an expected pattern at this point – especially since iPadOS and macOS seem to converge more and more every year. As such, their patterns are becoming similar.

For inspiration, try this out in Files. If you right click on a blank space, you have the option to create a new folder from the context menu.

Clip Context Menu Previews Correctly

When you present context menus, iOS animates the presenting interface element before it "pops" open. In some cases, you might have something that doesn't look quite right with the stock presentation. Thankfully, you can tweak this and customize it however you want in UIKit.

Again – the API for this can change whether you're using a table view, collection view or regular view, *but* they all work through the same object. Look for any delegate method that returns `UITargetedPreview`, and you're good to go. A targeted preview can be constructed with `UIPreviewParameters`, and using those together you can fix common "paper cut" issues like if your presenting view has rounded corners, and the default preview shows a square:

```
// Make the preview rounded
let params = UIPreviewParameters()
let presentingRect = cell.contentView.bounds.insetBy(dx: -8, dy: -8)
let clippedPath = UIBezierPath(roundedRect: presentingRect,
                               cornerRadius: 10.0)

params = clippedPath

return UITargetedPreview(view: cell.contentView,
```

```
parameters: params)
```

Menu Sizes

In iOS 16, you can leverage a size parameter to tweak the display on a context menu. This looks really nice, especially in pull-down buttons where you might be trying to convey a different pattern than simply a list of actions, or perhaps you've got logically grouped ones. It's driven by `UIMenu.ElementSize` – and there are three choices:

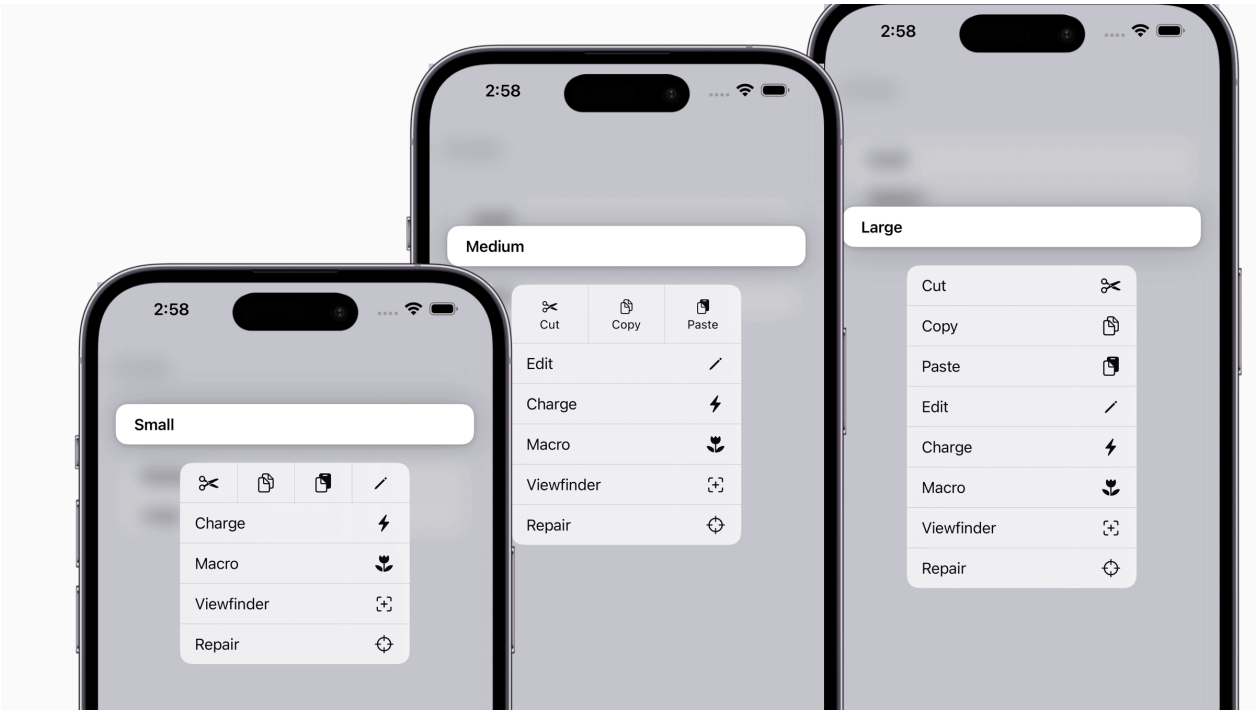
1. `small`: This presents a compact, side-by-side appearance. You'll get the `UIMenuElement`'s icon, but not the title. So here, you'll want to use iconography. This is capped to four items, and since the images are the only thing showing – you've got to make sure they are universally understood.
2. `medium`: The same as above, only now you've got the title *and* the image. This is capped to three items as opposed to four.
3. `large`: This is the default option. It works just like context menus always have – a list of rows.

You can see this in action if you open up `ContextMenusFig4ViewController.swift`. All three options can be presented with the exact same menu, constructed like so:

```
// Xcode -> UIKit -> ContextMenusFig4ViewController.swift

let topActions = [
    UIAction(title: "Cut", image: UIImage(systemName: "scissors"),
handler: { (_) in
    }),
    UIAction(title: "Copy", image: UIImage(systemName:
"doc.on.doc"), handler: { (_) in
```

```
    }),
    UIAction(title: "Paste", image: UIImage(systemName:
"doc.on.clipboard"), handler: { (_) in
    }),
    UIAction(title: "Edit", image: UIImage(systemName: "pencil"),
handler: { (_) in
    })
]
let bottomActions: [UIMenuElement] = [
    UIAction(title: "Charge", image: UIImage(systemName:
"bolt.fill"), handler: { (_) in
    }),
    UIAction(title: "Macro", image: UIImage(systemName:
"camera.macro"), handler: { (_) in
    }),
    UIAction(title: "Viewfinder", image: UIImage(systemName:
"plus.viewfinder"), handler: { (_) in
    }),
    UIAction(title: "Repair", image: UIImage(systemName: "scope"),
handler: { (_) in
    })
]
let items: [UIMenuElement] = topActions + bottomActions

return UIMenu(title: "",  preferredElementSize: size, children:
items)
```

In this case, a `size` argument passes in the selected menu size of small, medium or large menu:



Deferred Actions

If you've got some sort of logic that is asynchronous that you want to use in a context menu, it's still possible in UIKit. Look to `UIDeferredMenuElement`, which takes a completion handler that lets you pass off another menu when it completes:

```
UIDeferredMenuElement({ completion in
    data.fetch { info in
        let items = info.map { UIAction(title: info.title) { action
in } }
        completion([UIMenu(title: "", options: .displayInline,
children: items)])
    }
})
```

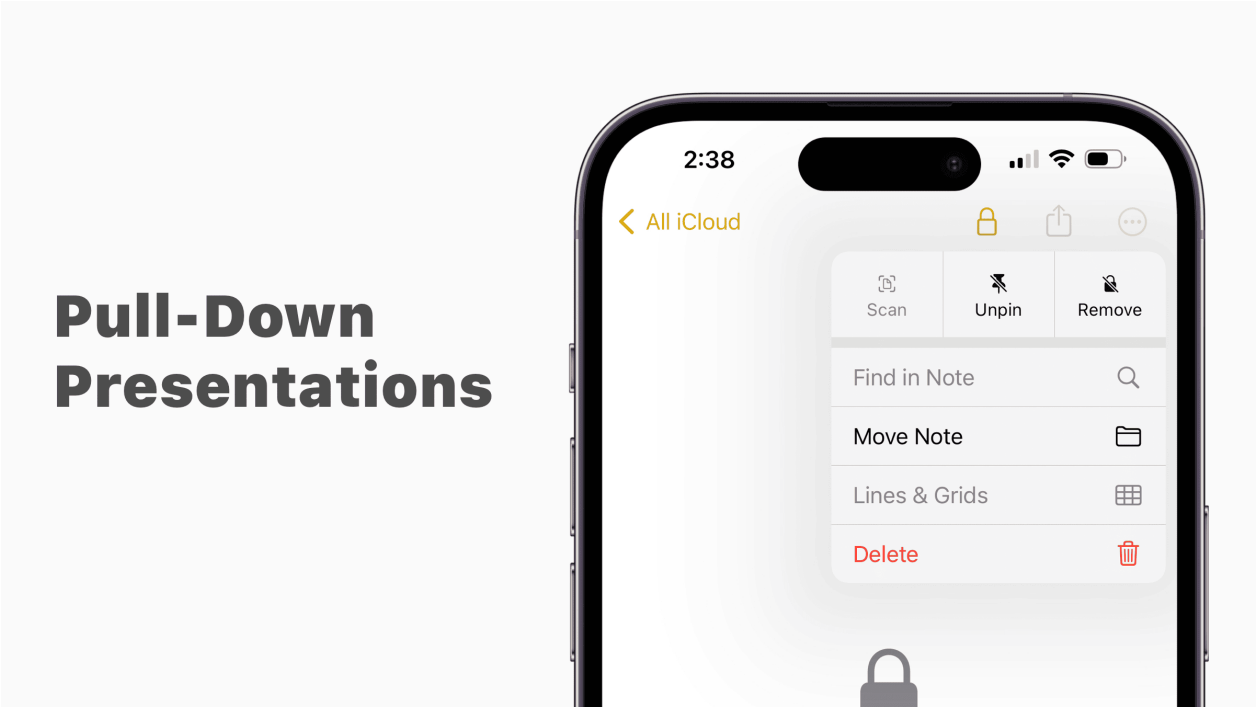
So, if you've got some network call to make or expensive data to fetch – this is how you do it with a context menu. An activity indicator will show while the work is being performed. Though, I will say, I would try to avoid this altogether if you can. Context

menus are all about quickness – and this is the opposite of that. Either way, it's nice to have the affordance for it, but it doesn't mean you should lean on it.

Replacing Entire Views with Pull-Down Buttons

For years, we have sort of been conditioned to present a view or view controller to show a different context. And, for the most part – that's the right call. But, I'd invite you to look at using a pull-down button if you can, especially if those choices that you're trying to show are relatively lightweight *and* directly relate to the view that's already showing.

Check out Notes, which does this very well. Tapping the more button (which, by the way, is a common pattern to present pull-down menus) simply shows a pull-down menu instead of another view controller with those options:



Three Key Takeaways

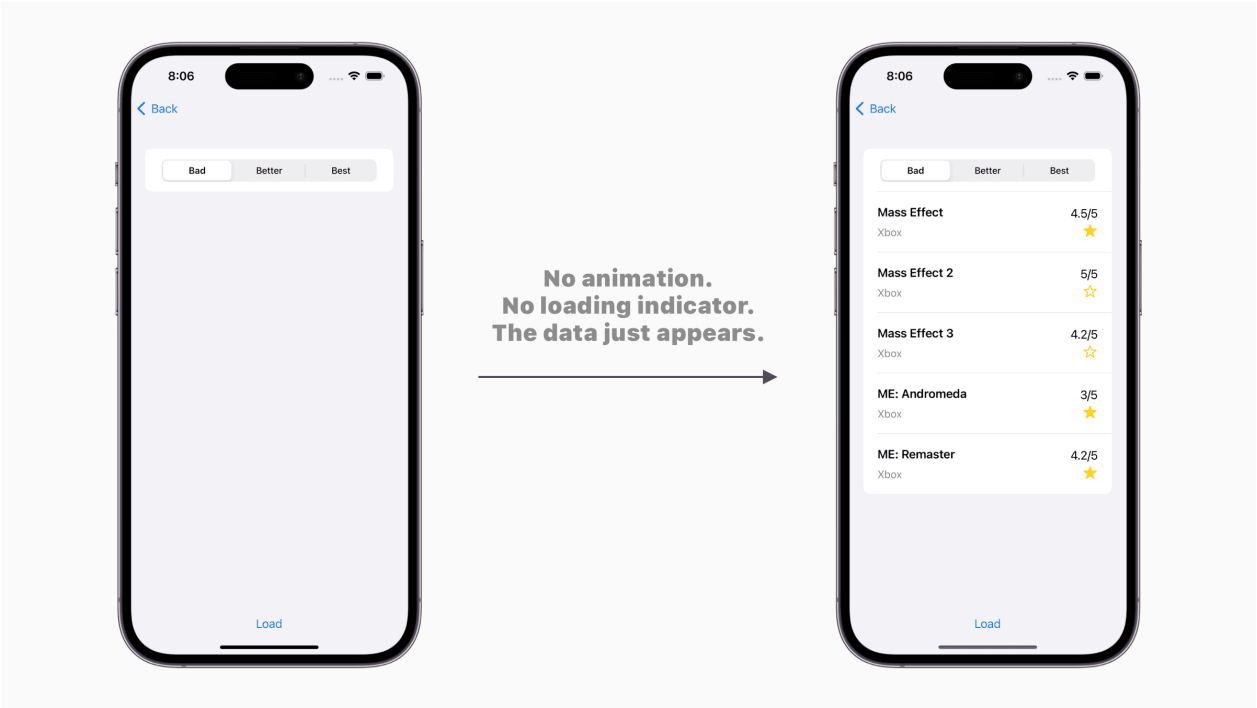
1. Context menus are the system provided interaction to offer options associated with an item that's selected (or many items that are selected).
2. Similarly, pull-down buttons offer a great way to operate on the item it was presented from, or the screen at large.
3. Leverage menus judiciously throughout your app, as they are the system provided way to present flexible options that would otherwise clutter the interface.

Reloading Data

There is truly an art to refreshing and then displaying information on a screen.

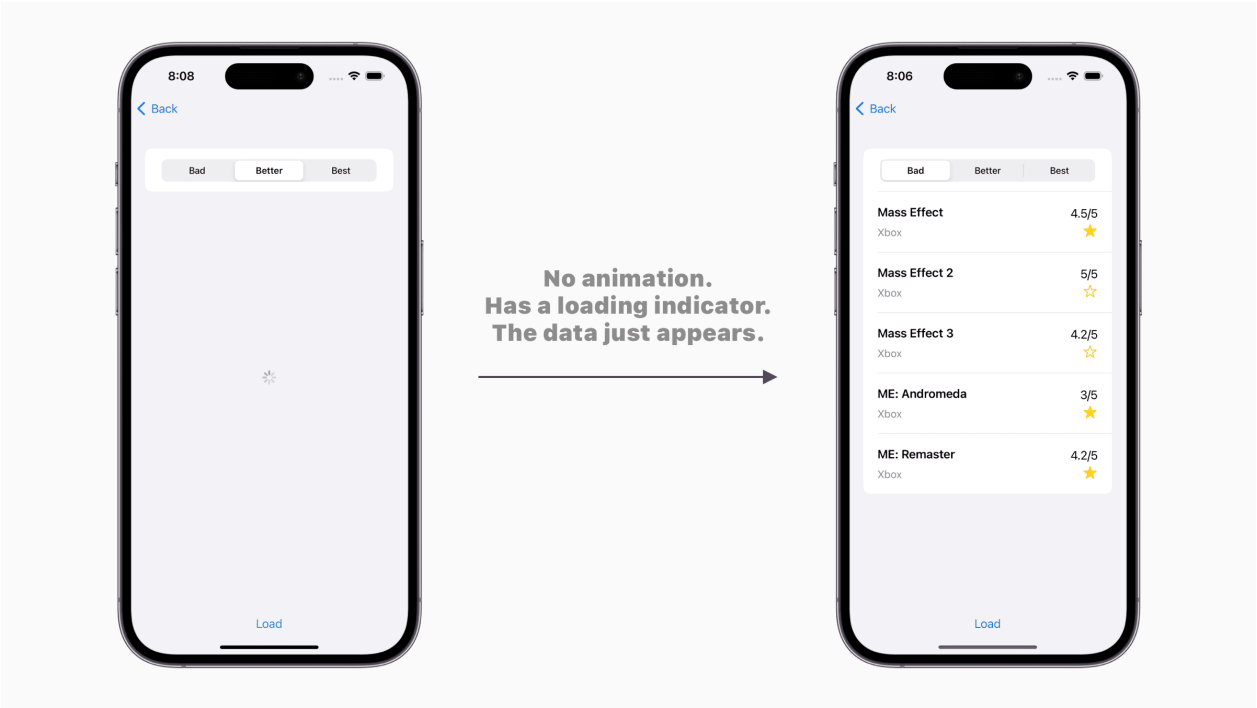
If you think about it, reloading data is one of the most important user experience interactions you can have in just about any app. Every app someone interacts with has data, and how that data is loaded, presented and then subsequently reloaded can either be done well or done wrong. It's similar to loading patterns in many ways, with the subtlety that the information isn't solely gated behind a loading operation, though in many times it can be.

There are layers to this, too. Consider an app that fetches a list of video games. If it just loaded up and then all of the games "blinked" into place – that doesn't feel great, right?



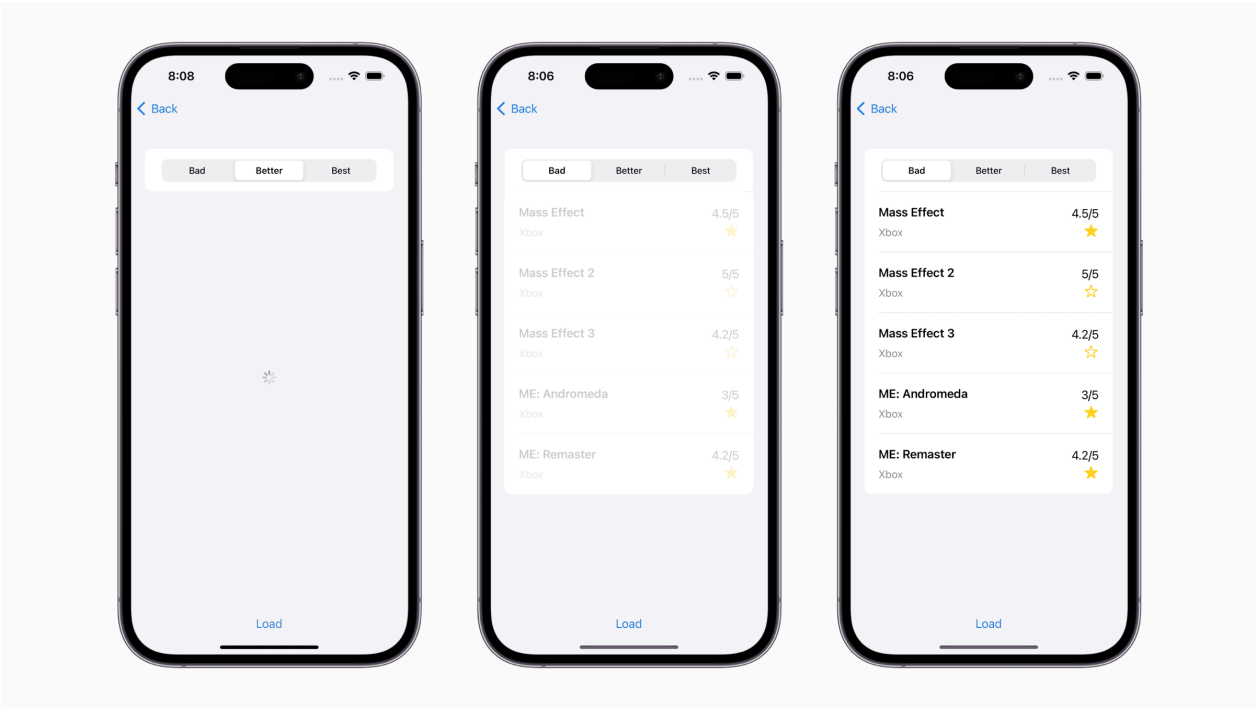
That's a hallmark experience of an app either not done well, rushed or perhaps indicative of a non-native solution where some compromises had to be made.

Either way, maybe it's done a little better if we have a loading indicator of some sort:



That's a step in the right direction, no doubt, because at least the user knows that something is happening. But even then, it still doesn't feel right. Cared for. Or crafted.

I think what *should* happen, at a minimum, is that the loading experience needs to feel fluid and purposeful. If we tweak the last two examples, we might keep the loading indicator and when the data is ready to show, we could quickly fade it in. Something like this:



Pictures can only show so much here, so please visit `ReloadingDataFig1.swift` to see all of these side-by-side. To recap, though…

Loading Experience	Shows Loading Indicator	Animates Changes
Bad	No	No
Better	Yes	No
Best	Yes	Yes

That chart should, in most cases, be your compass for reloading data. But it takes work to get it right, even when you do it the "correct" way, because often times developers or designers get tripped up on that last part – the animation.

In short, the animation isn't a time to show off, it just needs to convey a context switch. And, when you switch contexts, it should be deliberate and concise – which is why in most cases, a brief opacity fade in or out is just enough. It feels smooth, but the

animation itself isn't drawing attention necessarily – it's reinforcing the switch of "There was nothing here" to "Now it's loaded, here's your stuff."

Even if the data is already loaded, and you're simply appending or removing things to it – animations can go a long way. It helps people know what exactly was added or removed, and gives the whole interaction a fluid feel.

As an aside, it's key to remember that these animations are ones that people would likely see several times. So if you've gone a little overboard with an animation, the first time it might be novel – but after that, it's probably just annoying.

As it is in our example, the subtle and quick animation of the transition between content reinforces the operation that is happening under the hood, which was a three-step process:

1. Loading begins.
2. Convey loading is happening.
3. Show the loaded data, i.e. – reload it.

Smooth animations and movement that conveys meaning while avoiding stiff, rigid context switches are some things that have made iOS a very nice operating system for billions of people on earth to use. That notion extends to any interactions related to reloading data probably as much as anything else.

As I hinted at earlier, the difference in effective loading and reloading is exacerbated even more so, for better or for worse, when you are removing or adding data to an existing body of data that's already displayed.

To see this in action, visit `ReloadingDataFig2View.swift` and toggle between the two options. This simply adds a new video game to the list in a random spot. It could be the end, beginning or somewhere in-between. When the data isn't animated

– it's hard to tell where the new item was added at altogether. With an animation, it's much easier to understand.

What I call "blink reloads", that is – reloading data without any animation, nearly *always* leads to a poor experience. Nearly. There is still a place for it, as we'll discuss later. In most cases, though, it should be a last resort, or something used to prototype experiences and views.

Core APIs

The good news for designers and developers is this: The entire pipeline to load and reload data seamlessly has gotten *much, much easier*. Avoiding blink reloads in favor of "batch reloads" used to be an error prone affair, with developers having to manage a list of indices and track which were removed, added or simply needed updating.

With the advent of SwiftUI – those pain point no longer exist to the degree in which they used to. Further, even in UIKit, the introduction of diffable data sources has made all of this several orders of magnitude easier to manage. Most importantly, it's safer too – the infamous "index out of bounds" crashes are more or less a pain point of a bygone era in data management.

If you've done any degree of SwiftUI development, reloading data, and specifically *how* it works, is not really something you have to think a whole lot about. And, that's a good thing! SwiftUI simply uses the fact that it's a declarative interface framework which updates based upon state changes to handle it for you.

As such, I'm not going to cover how to set up an array and add data to it, but if you need examples of that, they are already found in numerous places in this book. The last example over reloading data the right and wrong way shows this.

In SwiftUI, we can guarantee animated reloads happen by ensuring that any time you mutate the data that's being shown to load it or reload it – you animate it via either implicit or explicit methods:

```
@State private var foods: [Food] = []

// Implicit
List(foods) { food in
    Text(food.name)
}
.animation(.default, value: foods)

// Explicit
List(foods) { food in
    Text(food.name)
}

// Somewhere else down the line
withAnimation {
    let newFoods = Food.random()
foods.append(newFoods)
}
```

Which way you handle it is up to your engineering preferences or needs, but the fact remains that animating any reloads to data in SwiftUI is the definition of trivial.

Moving over to UIKit, you're going to want to use diffable data source structures. These are tailor made for both table and collection views. If you've followed any of the UIKit samples from this book, you've seen them already. Using them, you can simply change data around and apply those changes to a *snapshot* object – and then let the API figure out the bookkeeping for you.

What's even better is that there is simply an argument you can use to control whether or not those animations are updated:

```
datasource.apply(snap, animatingDifferences: true)
```

So, what used to be a near incomprehensible dance of managing `NSIndexPath` is now as simple as this:

```
// Xcode -> UIKit -> ReloadingDataFig1ViewController.swift

var snap = datasource.snapshot()
snap.appendSections([0])
snap.appendItems(Place.places())
datasource.apply(snap, animatingDifferences: true)
```

Further, and some developers forget this, the diffable datasource is built to…well, *diff* your data. So there's (mostly) no issue with using a similar code sample like the one above every single time you want to update data.

Why?

Because under the hood, the framework isn't throwing everything away and starting from scratch, it's figuring out what's changed between the new snapshot and the old one – and then just animating those differences. Unless you're dealing with especially large data sets, there's no need for you to figure it out. Just create a new snapshot, add the relevant data, and then let UIKit figure out what's changed.

Before moving on to some best practices, I do want to note, this discussion is covering the most common way data is reloaded on iOS – that is, lists of things. Reloading data effectively extends far beyond this use case, but the same principles apply. Whether it's hiding or showing a login button or fetching a giant, technical feed – both cases are a form of loading or reloading data of some variety.

Tips

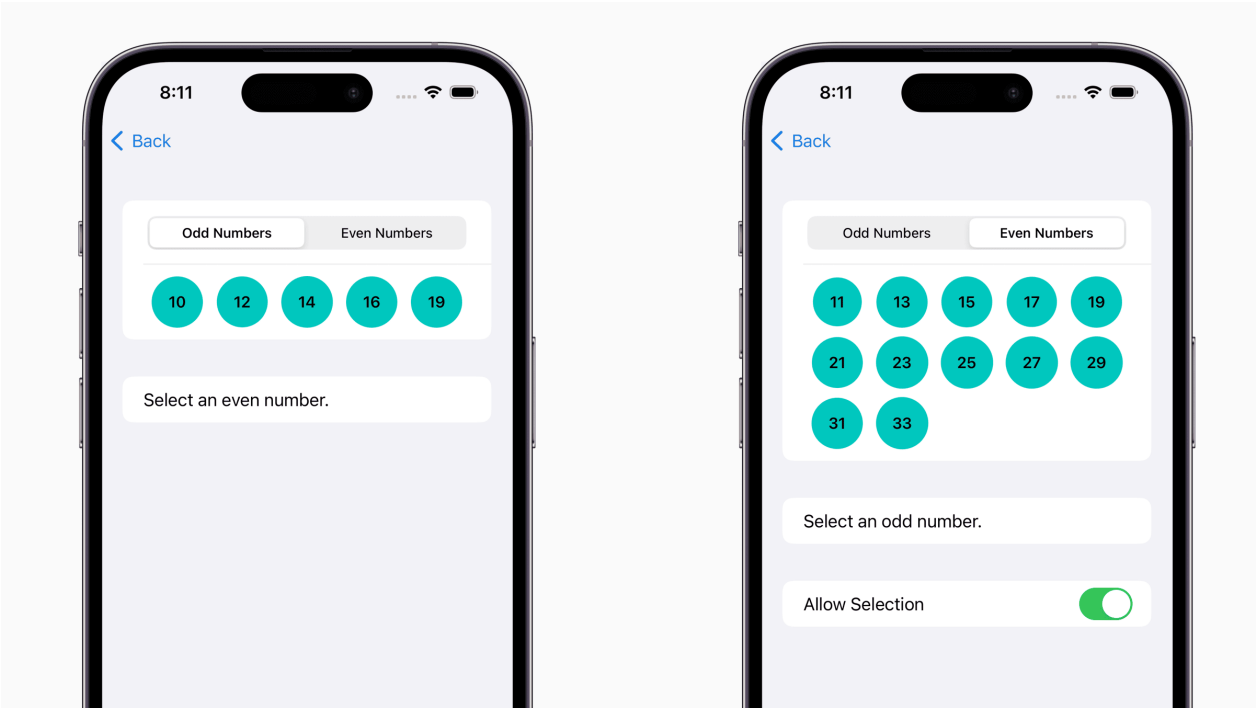
When to *Not* Animate Reloading Data

There are times when animating reloads of data isn't the optimal choice. One spot is when dealing with initial data loads, or data you know you can fetch before the interface is displayed. In those cases, if you can have your data ready to go — opt to just show it.

A common example of this is when you're dealing with locally cached data. In those cases, reading it from disk is lightning quick — so there's no reason to have the interface display and then animate it in. When the app comes into the foreground, having the data right then and there should always trump any animation bringing it in.

Show it right away! *Always* show it right away if you can do it. The faster you can people to content, the better.

Another area where I've found it better to forgo animations is when you're switching up the view hierarchy in a significant way. Consider the two options below. One has a single row of data, while the other has several of them plus another section:



Here, the animation doesn't really look that great, and it makes things feel a little bit slower. This comes down to a "feel" thing, but in my opinion and from using apps that have gone both ways – I find it better to simply change the views out so people can do what they want quicker.

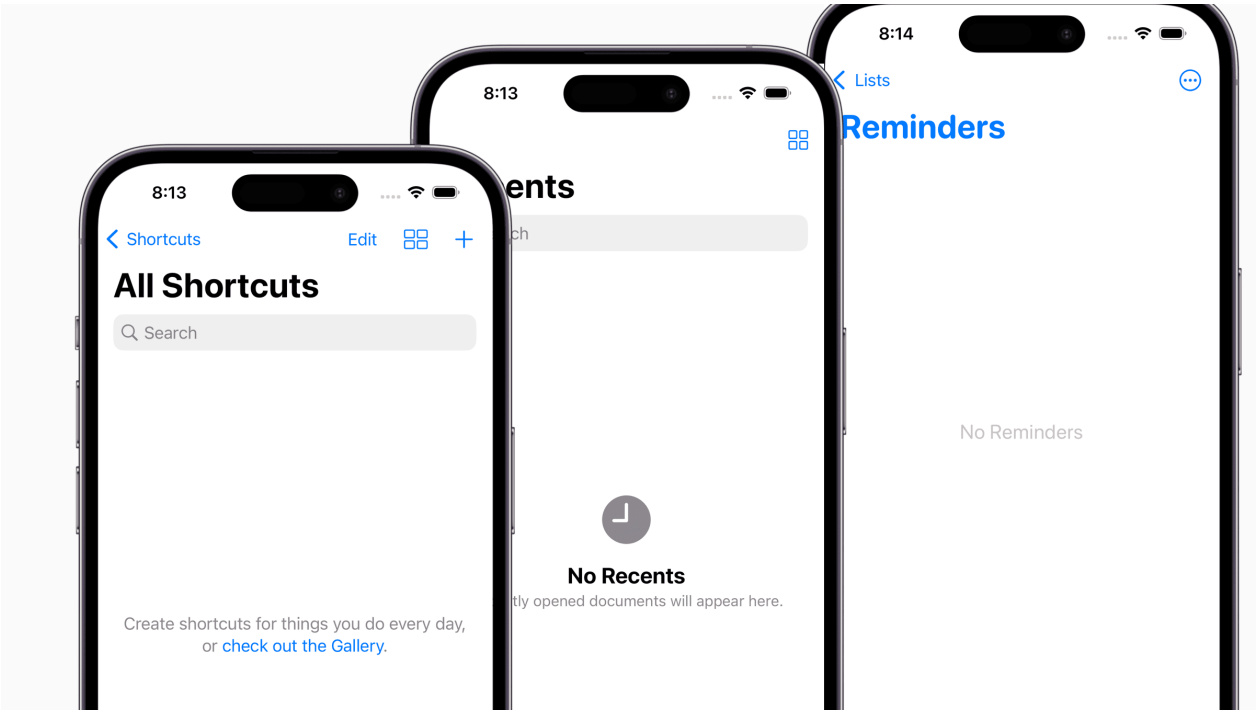
There are times when it feels exciting to animate these types of things, but to me – it really comes down to how drastic the change in hierarchy is. To revisit an earlier discussion, if the only thing we were going to change about the hierarchy was adding or removing some sort of log in button – then yeah, it probably looks nice to animate that back in. But, for wholesale or significant interface changes – you likely want to skip the animation.

Using "Empty Data" Views

Every app will eventually start off with no data, or at least not have any at some point. In those cases, it can help to show the classic "empty data" view. They serve a two-fold purpose:

1. They hint at what *will* show when there is data.
2. They can provide a clear call to action to get that data.

You can see this pattern all over iOS:



There are a few ways to do this. First, we'll cover `ContentUnavailableView` in SwiftUI, and `UI ContentUnavailableView` in UIKit. These controls are built specifically for this scenario. Remember, showing these views when certain events occur, such as:

- There is no data returned.
- A network error occurred.

- A call to action is warranted.

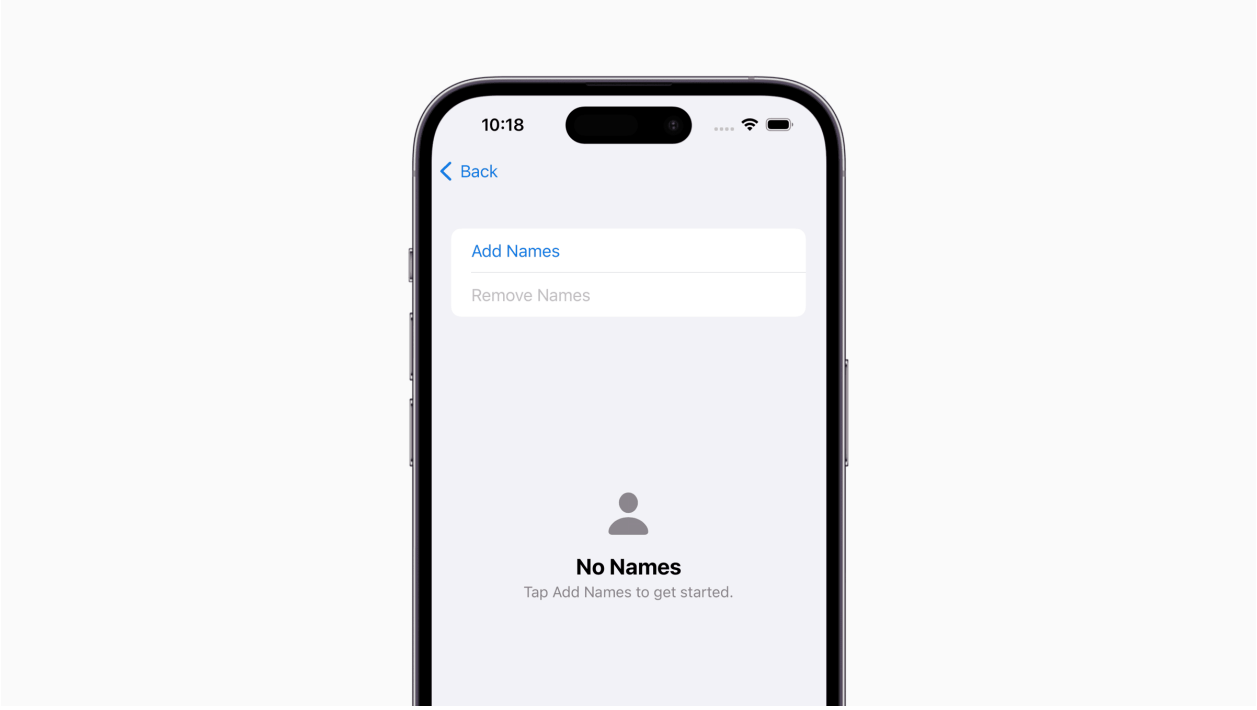
In SwiftUI, these work like any other view instance would — and that is perhaps the key piece to understanding them at first. There is no "magical" presentation about them, you simply hide or show them based on your own conditions.

Here, the "empty view" will show if an array is empty:

```
// Xcode -> SwiftUI -> ReloadingDataFig7View.swift

struct ReloadingDataFig7View: View {
    private let allNames: [String] = ["Jordan", "Jansyn", "Benny"]
    @State private var displayedNames: [String] = []

    var body: some View {
        Form {
            Section {
                Button("Add Names") {
                    withAnimation{ displayedNames = allNames }
                }
                .disabled(!displayedNames.isEmpty)
                Button("Remove Names") {
                    withAnimation{ displayedNames = [] }
                }
                .disabled(displayedNames.isEmpty)
            }
            List(displayedNames, id: \.self) { name in
                Text(name)
            }
        }
        .overlay {
            if displayedNames.isEmpty {
                ContentUnavailableView {
                    Label("No Names", systemImage: "person.fill")
                } description: {
                    Text("Tap Add Names to get started.")
                }
                .transition(.blurReplace())
```

```
                }
            }
        }
}
```

In the `overlay` – notice how the `ContentUnavailableView` is transitioned in and out. The design is familiar to iOS users, since Apple has been using this pattern and design for years:



Further, there are some static variables with pre-made empty views to use too. For example, `ContentUnavailableView.search` will display a localized empty data view for search scenarios (the most common use-case for these empty views). Even better, you can pass it a String that was searched for, and it'll append it to the empty view's contents:

```
// Xcode -> SwiftUI -> ReloadingDataFig8View.swift
```

```
struct ReloadingDataFig8View: View {
    var body: some View {
        ContentUnavailableView.search(text: "Cheap flights!")
    }
}
```

And, the results:



In some cases, you may want to prompt the user of what to do when there is no data. For example, if there the empty view is presenting a “favorites” list of some sort, your call-to-action might be “Add a Favorite”. This is supported with `ContentUnavailableView` by using its `actions` parameter:

```
ContentUnavailableView {
    Label("No Favorites", systemImage: "star")
} description: {
    Text("When you favorite items, they'll show up here.")
} actions: {
    Button("Add a Favorite") {
```

```
            showAddFavorites.toggle()
        }
}
```

Switching gears to UIKit, its `UIContentUnavailableView` is powered by configuration configuration instances. This is a direction that UIKit has been heading for several years, so it makes sense that we let UIKit handle presentation of empty views via configurations.

The flow works like any other content configuration. You create one, and assign it directly to the view controller. Then, when needed, you ask for UIKit to update its content configuration for the for content unavailable view.

For example, here's how to display a loading empty data view, like this one:



The code is very simple:

```
// Xcode -> UIKit -> ReloadingDataFig3ViewController.swift
```

```
class ReloadingDataFig3ViewController: BaseSampleViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        let loadingConfig =
UIContentUnavailableConfiguration.loading()
        contentUnavailableConfiguration = loadingConfig
    }
}
```

And, when you need to update one – you simply invoke `updateContentUnavailableConfiguration(using state:)`:

```
// Xcode -> UIKit -> ReloadingDataFig4ViewController.swift

override func updateContentUnavailableConfiguration(using state:
UIContentUnavailableConfigurationState) {
    var emptyView: UIContentUnavailableConfiguration? = nil

    if !createDrawing {
        emptyView = .empty()
        emptyView?.image = .init(systemName: "scribble.variable")
        emptyView?.text = "Create a Drawing"
    }

    contentUnavailableConfiguration = emptyView
}
```

The `UIContentUnavailableConfiguration` Struct will give you all of the customization options you might need, such as title, text attributes, buttons, images and more.

Custom Empty Data Views

If, for whatever reason, UIKit's and SwiftUI's empty data view variants aren't what you want – creating your own isn't difficult either. In SwiftUI, this is, technically speaking, effortless to do:

```
// Xcode -> SwiftUI -> ReloadingDataFig4View.swift

var body: some View {
    List(games) { game in
        VideoGameCell(game: game)
    }
    .overlay {
        if games.isEmpty {
            emptyView
        }
    }
}
```

You can simply check if the relevant data is empty, and if so – show some sort of empty data view. There are a few ways you can handle it, too. Perhaps a `ZStack`:

```
// Shorted for brevity
ZStack {
    List
        .opacity(data.isEmpty? 0 : 1)
    EmptyView
        .opacity(data.isEmpty? 1 : 0)
}
```

Or, an `if/else`:

```
if data.isEmpty {
    List
} else {
    EmptyView
```

```
}
```

But the reason why I prefer an `.overlay` approach is that the modifier can serve an explicit purpose of simply showing the empty data view – it feels like less to manage. Plus, when dealing with greedy views such as a `List` or `Scrollview`, an overlay helps center content easily.

In UIKit, this is a bit harder to pull off. While there are several ways to think about – I've even swizzled several UIKit methods using Objective-C to have empty data views automatically show – I prefer Combine these days if you're data is outside of diffable data sources. Simply listen to data changes, and perform hierarchy changes based off of it.

But, most data is using diffable datasources, so in these cases I try to abstract the reloading function further so I can check if the data is empty after a reload occurs. You can see a demonstration of this in `ReloadingDataFig2ViewController.swift`:

```
// Xcode -> UIKit -> ReloadingDataFig3ViewController.swift

override func viewDidLoad() {
    super.viewDidLoad()
    view.addSubview(collectionView)
    reloadData([]) // Force an empty view
}

private func reloadData(_ places: [Place]) {
    var snap = datasource.snapshot()
    snap.appendSections([0])
    snap.appendItems(places)
    datasource.apply(snap)

    // Empty data? Show the empty data view.
    if datasource.snapshot().itemIdentifiers.isEmpty {
        view.addSubview(emptyView)
```

```
            NSLayoutConstraint.activate([
                emptyView.centerXAnchor.constraint(equalTo:
view.centerXAnchor),
                emptyView.centerYAnchor.constraint(equalTo:
view.centerYAnchor),
                emptyView.widthAnchor.constraint(equalTo:
emptyView.widthAnchor),
                emptyView.heightAnchor.constraint(equalTo:
emptyView.heightAnchor)
            ])

        } else {
            emptyView.removeFromSuperview()
        }
}
```

Simplify Reloads in SwiftUI with Implicit Modifiers

SwiftUI has the idea of implicit and explicit animations. When you perform explicit animations, you're firing off changes within a function's closure and telling SwiftUI you'd prefer said changes to be animated:

```
withAnimation {
    self.isShowingLogInButton.toggle()
}
```

With implicit animations, you leverage a SwiftUI modifier – this signifies that any change in the value you provide should be executed with an animation. This is wonderful for lists of data, because most of the time – you want those animated. If you look at `ReloadingDataFig3View.swift` you can see an example of it:

```
// Xcode -> SwiftUI -> ReloadingDataFig3View.swift
List {
    ForEach(games){ game in
        VideoGameCell(game: game)
```

```
    }
}
.animation(.default, value: games)
```

Of course, if you want more control when those animations occur – there's nothing wrong with taking the explicit route. I have an example of that in the same file mentioned just above. Keep in mind, though, you'd need to be sure to use that technique wherever the data is mutated, which is one reason I enjoy implicit animations – there's generally less to think about.

Using Transitions in SwiftUI for Interface Elements

Speaking of SwiftUI animations, you can leverage the `.transition` modifier to create some nice animations when things are added or removed. As the name suggests, these effects occur when content is transitioned away, or to, the view graph.

While some view types will override your preferences, such as `List`, if you're using something like `ForEach` on its own – you can make rows slide in when added, and fade away and scale down when removed with one line of code:

```
// Xcode -> SwiftUI -> ReloadingDataFig6.view

ZStack {
    Color(uiColor: .systemGroupedBackground)
        .edgesIgnoringSafeArea(.all)
    VStack {
        ForEach(games){ game in
            VideoGameCell(game: game)
                .padding(8)
                .background(Color(uiColor: .systemBackground))
                .cornerRadius(8.0)
                .padding(.horizontal, 16)
                // Animate the transition
                .transition(.asymmetric(insertion: .slide,
removal: .opacity.combined(with: .scale)))
```

```
            }
        }
}
```

Remember Your App's Voice

While the following advice deals primarily with micro animations, which were previously covered, take care to keep your app's voice in mind when deciding how to animate data.

No matter the voice you're going for, you usually follow the same rules. However, when you are dealing with reloading data in terms of your interface elements – there is typically more room to put your name on things. Think of the classic "like" reactions in Facebook, or the exploding heart animation on Twitter when you like a tweet. Both of those are forms of reloading data in that you increment a count, and reload the interface to show that you've done so.

If they add to the experience, and wouldn't be annoying after the user has performed that action hundreds of times – then go for it! But again, a little goes a long way.

Three Key Takeaways

1. Reloading data the right way makes an app feel complete and polished.
2. Using the correct APIs to animate changes is key to a best-in-class app.
3. However, there are times when animating reloads isn't the right answer, so be cognizant of those scenarios as well.

Leveraging Hardware Keyboards

A great hardware experience for iOS and iPadOS leads to a magical experience. I remember when Apple first made keyboard hardware a first class citizen with iPadOS. In my opinion – this was around iOS 13.4, which debuted the pointer APIs. Once I first tried it, my iPad felt like an entirely new machine.

This chapter is all about how to give that same feeling to folks who use your apps with hardware keyboards. The good news is that Apple does so much for you out of the box – which is a phrase I've used several times in this book series. Given sensible use of Apple's APIs, you get a harmonious experience, mostly for free. But we can, and should, go beyond that baseline.

So, that leads us to ask ourselves – what does marquee support for the keyboard look like? Certainly, you'd want the iPadOS pointer to function beautifully, because that's an important component to the conversation. But, that's a topic which is reserved for the next chapter.

So, going beyond the pointer, there are really two primary goals you need to think about to make the keyboard user experience phenomenal:

1. Standout keyboard shortcut support.
2. Navigational considerations specifically for keyboards by understanding how focus works.

Those are the main components to consider. When people are using keyboards, well – they mean business most of the time. You get a lot of "power users" who love to zip around your interface using only the keyboard, and who rely on shortcuts to get things done quickly. To that end, let's dive into how each of these things work.

Keyboard Shortcuts

Usually, I like to introduce technologies for SwiftUI and UIKit together. But here, there are several differences between the two and how they work – so I'm going to go framework by framework, starting with UIKit.

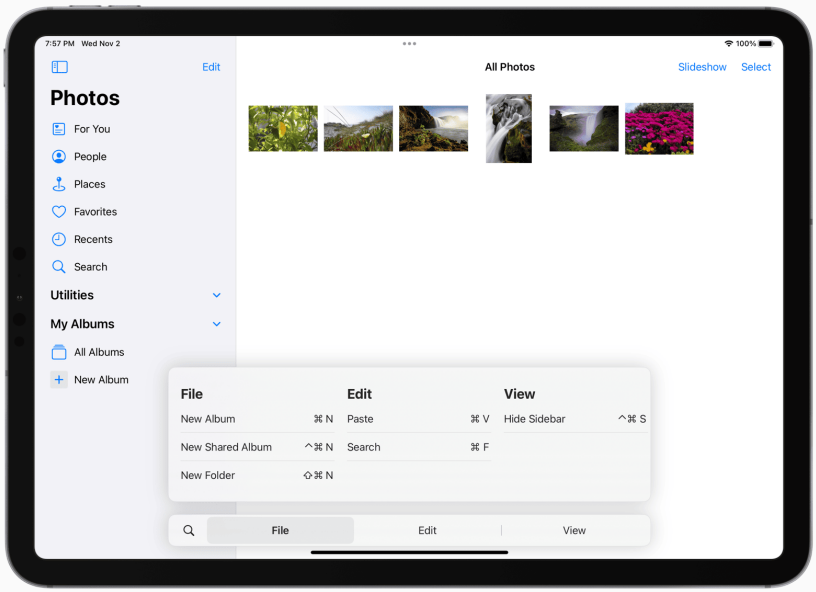
There are basically two routes you can go with keyboard shortcuts in UIKit. In the past, it all ran through `UIKeyCommand`, and you most commonly implemented them at the view controller level:

```
// Xcode -> UIKit -> KeyboardFig1ViewController.swift

class KeyboardFig1ViewController: BaseSampleViewController {
    private var showGreenBackground: Bool = false

    override var canBecomeFirstResponder: Bool {
        return true
    }
```

```
    override func viewDidAppear(_ animated: Bool) {
        becomeFirstResponder()
    }

    override var keyCommands: [UIKeyCommand]? {
        return [
            UIKeyCommand(title: "Change Background",
                         action: #selector(changeBackground),
                         input: " ")
        ]
    }

    @objc func changeBackground() {
        view.backgroundColor =
showGreenBackground ? .green : .systemBackground
    }
}
```

This flow necessitated two things before you even thought about key commands. First, you need to report that you *can* become first responder. Then, you have to report that you currently are. Subsequently, overriding `keyCommands` let you return an array of them. If you do it this way, it still works – but there are a few reasons why I would move away from it if you're doing it now, or skip it altogether if you haven't started.

The first reason is that all of these shortcuts will be uncategorized when you bring up the discoverability interface. If you're unfamiliar with what that is, hold down the ⌘ key on your iPad when the keyboard is connected, and it'll be presented to you:



This interface serves two purposes. One is to simply show you what you can do. The other is to let you perform them quickly. Since this is the primary way people will discover your keyboard shortcuts, you don't want a giant list of them showing uncategorized.

The second reason is that this isn't the the "blessed" API Apple is pushing developers to use when implementing them. You want to skate where the puck is going, so to speak.

So, if that's not the best way to do them – what is? That's where the `UIMenuBuilder` APIs come into the picture. Basically, if you supply several `UIMenu` items, you can build up keyboard shortcuts within your app delegate:

```
// Found within the demo project's app delegate file
override func buildMenu(with builder: UIMenuBuilder) {
    super.buildMenu(with: builder)

    // Ensure the builder is modifying the main menu.
```

```
    guard builder.system == .main else { return }

    let demoMenu = UIMenu(options: .displayInline, children: [
        UIKeyCommand(title: "Change Background",
                     action:
#selector(KeyboardFig2ViewController.changeBackground),
                     input: " ")
    ])

    // Insert in the edit menu
    builder.insertChild(demoMenu, atStartOfMenu: .edit)
}
```

A few things to notice here. One is that if you visit `KeyboardFig2ViewController` you'll see that we don't bother with overriding first responder status or marking ourselves as first responder when the view appears. Why?

This has to do with a change, starting with iOS 15, where the first responder status follows the focus system. So, whatever is focused also has first responder status.

This is great, because things are a little bit easier now – namely, we just have to define the function that we want to be called for the keyboard shortcut. When you visit `KeyboardFig2ViewController` and hold down ⌘, you'll see the discoverability interface once more with our shortcut displayed. And, of course, pressing the spacebar toggles the background color as it did before in the first example.

So, that's UIKit keyboard shortcuts at a glance. It's a powerful system that you should include in any of your apps, and I think you'll find that the menu builder API is much more expressive than the "old" way of creating keyboard shortcuts.

Now, let's check out SwiftUI.

Unsurprisingly, these works with a modifier. Implementation is easy, but you should know that you have to attach them to controls which already perform an action,

namely – `Button` views. You don't pass ad-hoc closures or functions to them (even though I wish it worked this way) but even so, they are lightweight and pragmatic:

```
// Xcode -> SwiftUI -> KeyboardFig1View.swift

struct KeyboardFig1View: View {
    @State private var showGreenBackground: Bool = false
    @State private var bgColor: Color =
Color(uiColor: .systemBackground)

    var body: some View {
        ZStack {
            bgColor
            VStack {
                Text("Bring up keyboard shortcuts to change the
background color.")
                    .padding()
                Button("Toggle Background") {
                    toggleBackgroundColor()
                }
                .keyboardShortcut(.space, modifiers: [])
            }
        }

    }

    private func toggleBackgroundColor() {
        withAnimation {
            showGreenBackground.toggle()
            if showGreenBackground {
                bgColor = .green
            } else {
                bgColor = Color(uiColor: .systemBackground)
            }
        }
    }
}
```

A few things to note here. One is that, if you don't use the `modifiers` parameter, SwiftUI is going to automatically pair your shortcut with the ⌘ key. Here, that's a problem – because our shortcut would map to ⌘ + `space` , and that is already a universal shortcut on iOS and iPadOS for search. So, that's why I've passed in an empty array.

Secondly, as I mentioned above, you need to attach the `.keyboardShortcut` modified to a view which performs an action already. Here, that's our `Button` that changes the background color if it were tapped. Third, the title that's shown in the discoverability interface will match the button's text. With that in mind, that's all it takes to get SwiftUI keyboard shortcuts up and running.

There is another way to handle SwiftUI keyboard shortcuts though, and that is through using common individual keys and the idea of *semantic* keys. This takes regular, common shortcuts and makes them easier to implement. For example, if we simply changed the above example's `.keyboardModifier` to this:

```
 // Xcode -> SwiftUI -> KeyboardFig2.view

// Same as previous implementation, only changes
// This modifier
.keyboardShortcut(.return)
```

Then you'll see that SwiftUI now adds a keyboard shortcut for us mapped to ⌘ + `return`. It's a little quicker, but remember, you're getting that ⌘ key combination with it. Here are the defined single key shortcuts SwiftUI has for us to use:

```
/// Up Arrow (U+F700)
public static let upArrow: KeyEquivalent

/// Down Arrow (U+F701)
public static let downArrow: KeyEquivalent

/// Left Arrow (U+F702)
```

```
public static let leftArrow: KeyEquivalent

/// Right Arrow (U+F703)
public static let rightArrow: KeyEquivalent

/// Escape (U+001B)
public static let escape: KeyEquivalent

/// Delete (U+0008)
public static let delete: KeyEquivalent

/// Delete Forward (U+F728)
public static let deleteForward: KeyEquivalent

/// Home (U+F729)
public static let home: KeyEquivalent

/// End (U+F72B)
public static let end: KeyEquivalent

/// Page Up (U+F72C)
public static let pageUp: KeyEquivalent

/// Page Down (U+F72D)
public static let pageDown: KeyEquivalent

/// Clear (U+F739)
public static let clear: KeyEquivalent

/// Tab (U+0009)
public static let tab: KeyEquivalent

/// Space (U+0020)
public static let space: KeyEquivalent

/// Return (U+000D)
public static let `return`: KeyEquivalent
```

Semantic keys work the same way, except they further abstract the concept of which key you're pressing away from you and focus on the task or purpose for which it's defined. Keeping with our example above, we could change it to this:

```
// Xcode -> SwiftUI -> KeyboardFig3.view

// Same as previous implementations, only changes
// This modifier
.keyboardShortcut(.defaultAction)
```

Now, iOS looks at several different factors such as the type of keyboard hardware and localization considerations to determine what `.defaultAction` should map to. If you bring up the discoverability interface, you'll see that for American localizations it maps to the return key. Unlike our previous example, that means it doesn't require ⌘ to be pressed along with it.

Before we move on to tips, I also want to touch on the idea of focus. We'll look at it more below, but here's what you need to know: giving interface elements the ability to focus means that it can be used with both keyboard shortcuts and for navigational purposes, depending on your implementation.

Thankfully, SwiftUI and UIKit take so much of this work off of your plate. If you open up a view and tab around, it's likely that focus will go through your interface as you expect it to. There are some nuances here, though, and also some handy ways to "debug" focus states too. We'll see those at the end.

Tips

Selection and Following Focus

You'll want to support "expected" behaviors as much as you can, and for the most part folks expect an up and down arrow key press to move around your app. One way to handle this is with the `.selectionFollowsFocus` property along with `allowsFocus` in table and collection views.

For the most part, we're trained to tie a selected cell to an action. So, if you tap a cell – open a new view and so on. For example, when you're running the sample project and are viewing chapters – perhaps you'd implement opening one up like this:

```
func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
    // Open chapter
}
```

But with focus, that kind of flips. Plus, this wouldn't work once we opt-in to focus navigation with a keyboard. You can see this yourself in the code for the sample project in the `GlossaryViewController`:

```
tableView.allowsFocus = true
tableView.selectionFollowsFocus = true
```

So, if we left the code to open up a chapter in the `didSelectRowAt` function, as soon as we navigated to it with a keyboard, the first row focused would open. That's obviously wrong.

So, this is where `performPrimaryActionForRowAt` comes in. Basically, you replace the code you would've put in `didSelectRowAt` here instead:

```
// In the GlossaryViewController

func tableView(_ tableView: UITableView,
performPrimaryActionForRowAt indexPath: IndexPath) {
    guard let chapter = datasource.itemIdentifier(for: indexPath)
else {
        tableView.deselectRow(at: indexPath, animated: true)
        print("Could not find a chapter for the selected row.")
        return
    }

    // Present it
    pushBookChapter(chapter: chapter, with: interfaceChoice)
}
```

By doing this, when someone taps on a row – this logic is fired. But! Now, you can select rows with the keyboard and then hit the enter key to fire this function, allowing for full keyboard navigation.

If you need to control each item individually, you can also use `selectionFollowsFocusForItemAt` to opt certain things in or out:

```
func collectionView(_ collectionView: UICollectionView,
selectionFollowsFocusForItemAt indexPath: IndexPath) -> Bool {
    return someCondition ? true : false
}
```

UIResponderStandardEditActions

Since every `UIView` and `UIViewController`is also a `UIResponder`, there are a bevy of keyboard shortcuts you can implement without ever creating a single `UIKeyCommand` by way of the `UIResponderStandardEditActions` protocol. Here's a list of them you can implement by simply overriding the relevant function:

Common Shortcut	Command	Function
⌘X	Cut	`cut(_ :)`
⌘C	Copy	`copy(_ :)`
⌘V	Paste	`paste(_ :)`
⌘A	Select All	`selectAll(_ :)`
⌘B	Toggle Bold	`toggleBoldface(_ :)`
⌘I	Toggle Italic	`toggleItalics(_ :)`
⌘U	Toggle Underline	`toggleUnderline(_ :)`
⌘+	Increase Size	`increaseSize(_ :)`
⌘-	Decrease Size	`decreaseSize(_ :)`

Here's a quick example showing how you might respond to the cut command:

```
// Xcode -> UIKit -> KeyboardFig3ViewController.swift

class KeyboardFig3ViewController: BaseSampleViewController {
    private let textLabel: UILabel = {
        let lbl = UILabel(frame: .zero)
        lbl.text = "Try the cut command, which is ⌘X."
        lbl.textAlignment = .center
        lbl.numberOfLines = 0
        return lbl
    }()

    override func viewDidLoad() {
        super.viewDidLoad()

        view.addSubview(textLabel)
        textLabel.translatesAutoresizingMaskIntoConstraints = false
        textLabel.widthAnchor.constraint(equalToConstant:
200).isActive = true
        textLabel.centerYAnchor.constraint(equalTo:
view.centerYAnchor).isActive = true
```

```
        textLabel.centerXAnchor.constraint(equalTo:
view.centerXAnchor).isActive = true
    }

    // Edit controls
    override func cut(_ sender: Any?) {
        textLabel.text = "Cut Text"
    }
}
```

If you run that, you'll see that the text changes to "Cut Text" when you use the ⌘X shortcut. Keep in mind that it'll also work with the editing gestures too, not just the keyboard shortcut.

Remember, the responder chain comes into play here. So, if you try these out and they aren't working – it's likely that the control isn't first responder or another control *is* and implements one of these actions.

Raw Key Commands

If you just want to handle individual key taps, you need to dip down into `UIResponder`. By overriding `pressesBegan` and `pressesEnded` you can handle any key code:

```
// Xcode -> UIKit -> KeyboardFig4ViewController.swift

override func pressesBegan(_ presses: Set<UIPress>, with event:
UIPressesEvent?) {
    for press in presses {
        guard let key = press.key else { continue }
        switch key.keyCode {
        case .keyboardUpArrow: textLabel.text = "Key Up"
        case .keyboardDownArrow: textLabel.text = "Key Down"
        default: super.pressesBegan(presses, with: event)
        }
    }
}
```

```
override func pressesEnded(_ presses: Set<UIPress>, with event:
UIPressesEvent?) {
    for press in presses {
        guard let key = press.key else { continue }
        switch key.keyCode {
        case .keyboardUpArrow: textLabel.text = "Key Up - Finished"
        case .keyboardDownArrow: textLabel.text = "Key Down -
Finished"
        default: super.pressesEnded(presses, with: event)
        }
    }
}
```

This is great for many editing experiences, where you might want to nudge an interface element around. Just be super to call `super` if you don't handle the key press so that the system can keep existing behaviors working as they should.

Following Established Patterns

Continuing the point above — if you do handle key presses, make sure you keep platform conventions in place. That is, things should generally work like the table below:

Key Command	Expectation
Tab	Navigates to different groups of the interface.
Arrow	Navigates within a focused area.
Return or Space	Selects something.

Even if you don't handle key press events on your own, some of these you'll just get for free. For example, any sidebar control or text fields and views will work with those commands without any extra code.

If you have to use some of these key events in some other way, you'll need to use the `wantsPriorityOverSystemBehavior` at the `UIKeyCommand` level to override the system behavior. But, I honestly can't think of a great reason to do this – so if you find yourself needing it, I'd try to find another way to offer the functionality.

Creating Custom Categories in the Discoverability Window

Earlier I mentioned how using the "old" way to define keyboard shortcuts in UIKit would lead to uncategorized shortcuts in the discoverability interface. Using the "new" way allows you to create custom categories if the stock ones don't fit all of your app's functionality. Plus, due to how they are made, they are categorized either way because if it's not a custom menu you're adding commands to, then you're using one which UIKit already had defined (like `.view` or `.file`, etc).

To make a custom one, though, it's as easy as creating a new `UIMenu` and inserting it into the `builder`:

```
// Create a "Favorite Samples" menu.
let favoriteSamplesMenu = UIMenu(title:"Favorite Samples,
                            children: [/*Menu Items*/])

// Add it to the root menu
builder.insertSibling(favoriteSamplesMenu, afterMenu: .view)
```

And, if you need to add more to that customized category, you can by appending items to the relevant identifier. Here, that would be the favorite sample's menu instance's identifier:

```
// Insert another menu into the favorite samples menu.
let createNewSampleFolderMenu = UIMenu(/*Code with new menu */)
builder.insertChild(createNewSampleFolderMenu,
                    atEndOfMenu: favoriteSamplesMenu.identifier)
```

Discoverability Title

You may have noticed the titles in the discoverability menu and wondered if you can change them. You certainly can. It's as easy as setting `UIKeyCommand`'s `discoverabilityTitle`.

In our earlier sample at the top of the chapter, we looked at ways to add custom key command shortcuts. If we used the initializer which takes in the discoverability title, that's what would show in the discoverability interface:

```
UIKeyCommand(title: "Change Background",
             action:
#selector(KeyboardFig2ViewController.changeBackground),
             input: " ",
            discoverabilityTitle: "Toggle background")
```

So here, "Toggle Background" would show in place of "Change Background". Further, you can also dynamically make those display as enabled or disabled based off of your app's state, which is better than adding or removing them because then people know that the actions are there even if they can't use them currently.

When the discoverability interface shows, you can inform it of which shortcuts are valid by overriding `canPerformAction`. In keeping with our example above, if we were to override that in the keyboard sample controllers and return false…

```
override func canPerformAction(_ action: Selector, withSender
sender: Any?) -> Bool {
    if action == #selector(changeBackground) {
        return false
    }
    return super.canPerformAction(action, withSender: sender)
}
```

…then it would be dimmed out in the discoverability interface. Going even further, you can change the titles on the fly, too. That's done by overriding `validate(_ command:)` to reflect your app's state:

```
override func validate(_ command: UICommand) {
    if command.action == #selector(changeBackground) {
        if isShowingWhiteBackground {
            command.title = "Change to Green"
        } else {
            command.title = "Change to White"
        }
    } else {
        return super.validate(command)
    }
}
```

Focus Styles

By default, most items have a `UIFocusHaloEffect` for their focus style. This is the adornment shown around the control when it's focused. Others, like table or collection view items, use a highlighted effect. If you find yourself wanting to tweak the halo effect, you'll need to create a `UIFocusHaloEffect` for its `focusEffect` property.

Say you wanted the halo effect, but with more rounding applied to it so it correctly hugs a view that also has a 20 point corner radius. You could accommodate that like so:

```
myView.focusEffect = UIFocusHaloEffect(roundedRect: myView.bounds,
cornerRadius: 20,
curve: .continuous)
```

If you want to completely customize it, though, you'll need to do a bit more. It's not much, however, but it's a two-step process:

```
init(frame: CGRect) {
    super.init(frame: frame)
    // 1
    self.focusEffect = nil
}

func didUpdateFocus(in context: UIFocusUpdateContext,
withAnimationCoordinator coordinator: UIFocusAnimationCoordinator) {

    // 2
    if context.nextFocusedItem == self {
        // Here, it's focused - so customize the view however you
see fit to convey focus
    }
}
```

First, you'll need to assign `nil` to the view's `focusEffect`, and then adopt the protocol function `didUpdateFocus(in context: UIFocusUpdateContext, with coordinator: UIFocusAnimationCoordinator)` found in the `UIFocusEnvironment` protocol.

Views already adopt this, so simply implementing it is enough. But, when you do that, you can tell when the view is focused or becomes unfocused – this way, you can adapt its interface any way that you want to reflect that.

Debugging Focus

If you find yourself in a situation where you can't quite figure out what's focused, what isn't or why something won't become focused, there are a few tricks Xcode has that you can use.

The first is to use LLDB and type in this command at any breakpoint or relevant code path:

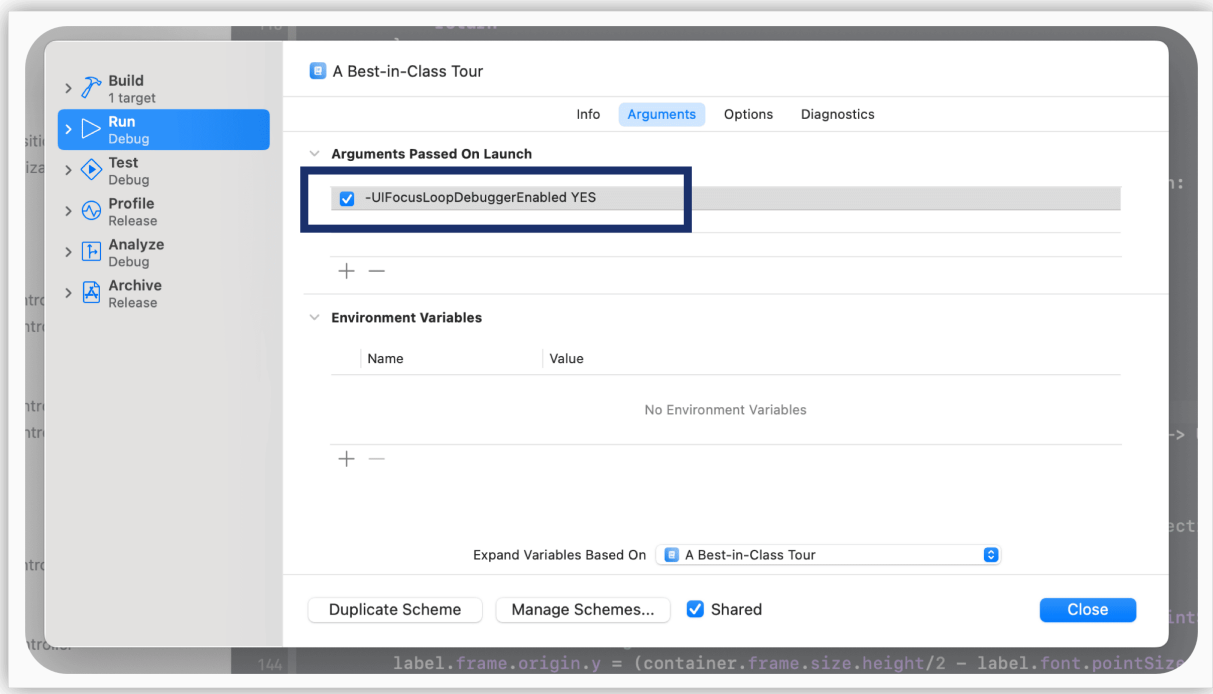
```
po UIFocusDebugger.checkFocusability(for:)
```

This will tell you if something is focusable at all, along with the reasons behind why it is or isn't. The next one…

```
po UIFocusDebugger.checkFocusGroupTree(for:)
```

…will let you see all the focus groups for the provided interface element. This can help you see why some part of the interface is focusing itself over another one you think should be focused.

Finally, if you edit your scheme and add in the `-UIFocusLookDebuggerEnabled YES` launch argument, you'll get an overlay at runtime for the current tab ordering. This helps solve some of the same problems, but in a more visual way. It's also a phenomenal way to visualize tab loop ordering. After you've added it, simply press and hold the option key to activate it. Further, if you hold option and control, you get a visualization of all the focus groups.



NEXT: make images, do the rest.

Three Key Takeaways

1. When you support hardware keyboards efficiently, you're supporting a user's preferred input mode and taking advantage of all the muscle memory that comes with it.
2. Touch is the premiere way people interact with iOS apps, but mouse and keyboard inputs will be increasingly more prevalent in years to come and that's especially true on iPadOS.
3. Understanding the focus system will help you build effective keyboard experiences, since it's intrinsically tried to how navigation works.

The Pointer on iPadOS

The pointer for iPadOS brings to mind one word: delight. It's simply *delightful* to use it, and when it's implemented thoughtfully — it's one of my favorite software experiences in the world.

When you think about the care and thought that went into bringing the pointer to iPadOS, it's no wonder it's a joy to use. Apple could've simply brought the classic arrow pointer experience over from macOS, and most people would've probably been happy with it.

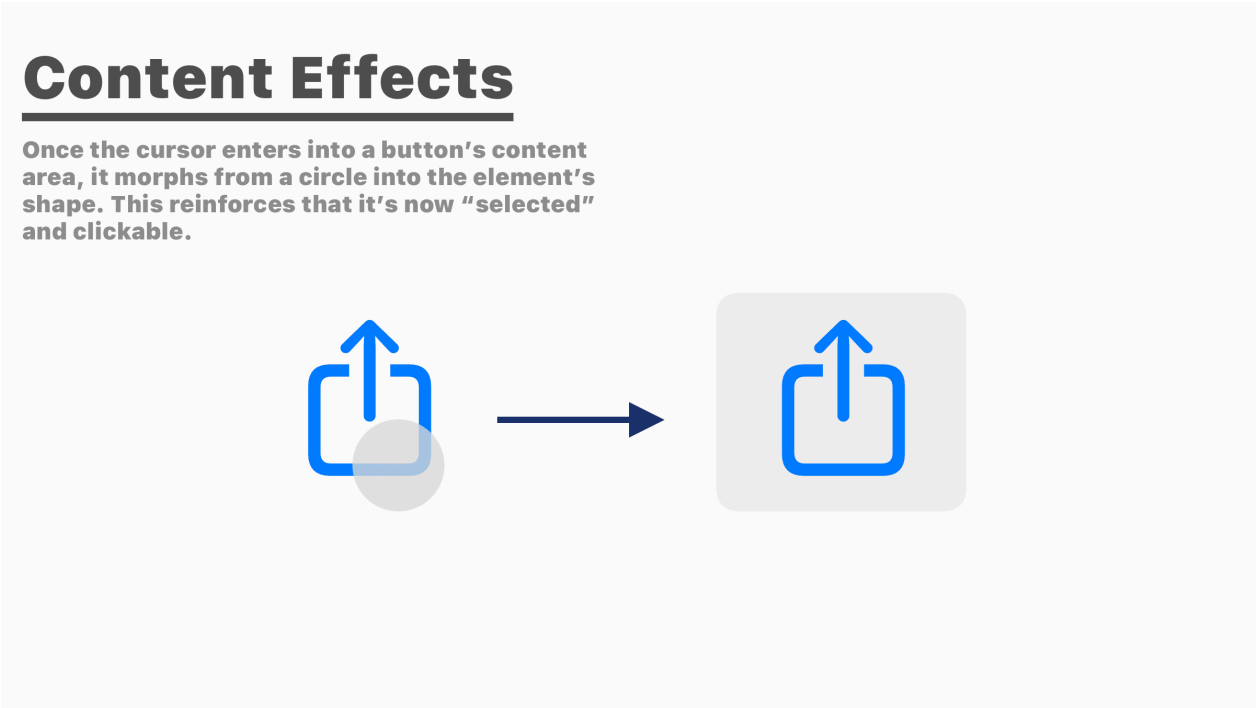
But that's not what they did — instead, they made a pointer experience that's uniquely tailored for iPad. Building upon the concepts of precision and fluidity, the iPadOS pointer works much differently than a traditional mouse experience would. Af-

ter all, as Apple's designers have mentioned – iPadOS is an operating system built for touch, so bringing a high-precision tool like the macOS mouse pointer to a low precision operating system wouldn't fit.

Look at bar buttons – they were created for a finger to tap on them, not for a cursor to point and click on them. This is why it feels so natural to have a floating circle represent the cursor on iPad, it gives more leeway to your selections and fits its interface elements accordingly:

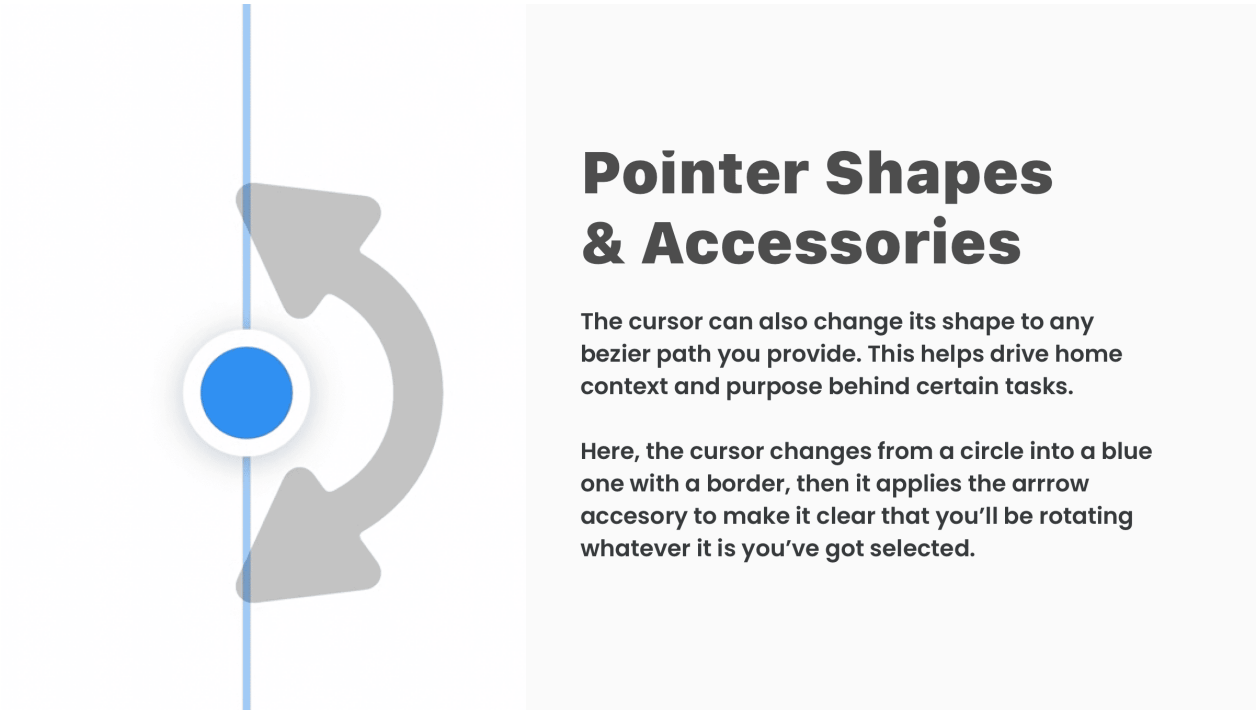


So instead of showing that tiny arrow, the pointer on iPad shifts and adapts to the surrounding interface, and that's one of the things that makes it so magical. In the example above, once the cursor enters the button's content area – it morphs from a circle into the button's shape which reinforces its selection state, giving people the confidence that they're clicking on the things that they expect:



The cursor matches the general size of a fingertip at around 19 points, so it's more of an extension of your tap gestures than it is a classic cursor input mechanism. With these content effects, it makes the interface feel fluid and crisp. Even the magnetism effect that's applied to it in some cases gives it a visceral, tactile feel.

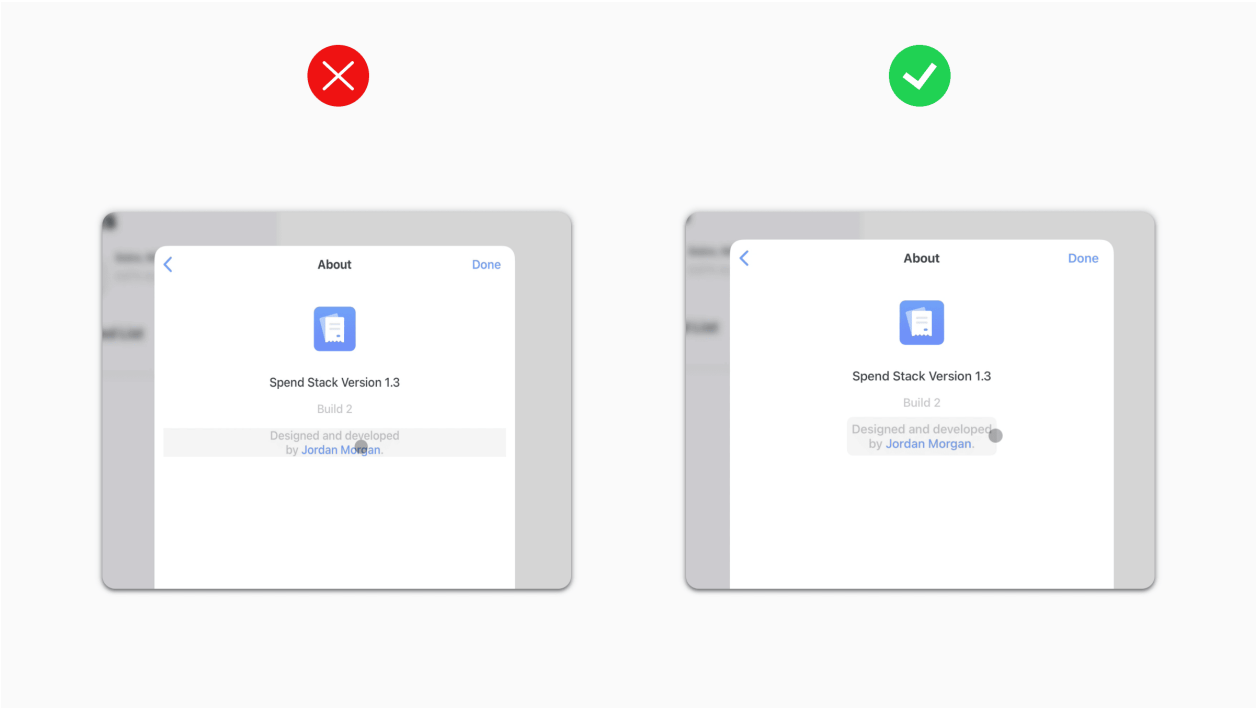
The pointer itself can even provide customized shapes. You're able to change not only the cursor itself, but have it appear as if it is has custom shapes and adornments surrounding it. This can help further contextualize the job at hand. Keynote does this wonderfully, and so does Pages and Numbers. Here's an example of what you could do:



So, there's a lot of happening with the cursor, and you've got free rein to really bend it any which way you see fit. Even so – I'm going to repeat the same phrase I've used over and over again: If you follow Apple's best practices and stock controls, so much of the work is done for you. The correct pointer effects are usually applied, the magnetism "snapping" is used correctly and buttons all "activate" when they are clicked.

So, for us – it's really a matter of fit and finish. We need to audit our interface for the cursor by using our own apps with it. Why? Because most controls already have the right pointer behaviors – but not all will. Further, there may be some padding issues you need to clean up too.

For example, in my last app, I had a text control which acted liked a button. To make centering it easier, the text control itself spanned the whole container's width, but with its text centered. This is where a situation where I had to inform the system of what the cursor's width should be, because taking the entire control's width looked wrong in this case:



Beyond patching up areas like that, we can also aim for some "surprise and delight" additions that you can achieve with custom pointer shapes. But in short, it's an incredible API that works *mostly* without a bunch of effort, but it's nearly certain that you'll have some rough edges to account for.

So, to begin your implementation – check out your bar button items, segmented controls and the menu controllers. These all support pointer effects automatically, but again – you're mainly ensuring they have a nice, breathable padding around them when the pointer hovers over each one.

Beyond pointer effects – there are gestures to think about too. Scrollviews respond to two finger pinch gestures from a connected trackpad, and table views will respond to two finger swipes on rows as well. You'll want to confirm these all work the way that you expect. And, they likely will unless you're doing some heavy customizations. Since several million apps are built from these controls, it makes sense that

Apple would ensure they work great with the iPadOS pointer without much effort from us.

With all that in mind, these are the four things you might end up doing for first-class iPadOS cursor support. You might need all four, or maybe just a few of them:

1. You need to ensure that the pointer looks correct within your interface in controls that already support it.
2. Consider when to enable those pointer effects in places where they aren't automatically used, such as buttons.
3. Ensure custom controls have the right pointer effects, if they should have any at all.
4. And finally, you'll need to search for places where it makes sense to customize things further with lower level APIs, such as custom pointer shapes or responding to hovering events.

Understanding the Pointer's Capabilities

How do you know what looks correct or what to fix? The best way is to build up a vocabulary around the pointer and understand its purpose and the APIs available that help achieve that purpose. To start, the iPadOS pointer has pointer *shapes* and *content effects* that you can control, and we've seen examples of those above. Plus, it has a protocol (`UIPointerInteractionDelegate`) you can adopt to respond to cursor events, vend new shapes dynamically or animate things alongside it.

Let's take a look at the stock pointer effects.

Highlight

This effect has the cursor morphing into the control's background. It also uses a parallax effect as you move the cursor around. When you click, the control has a subtle scale down and up effect.

Lift

This one is ideal for controls that already have a background. It gently scales the element up a bit, which makes it appear to "rise", hence the name lift. You can see this with app icons in the dock. Finally, it also applies a shadow.

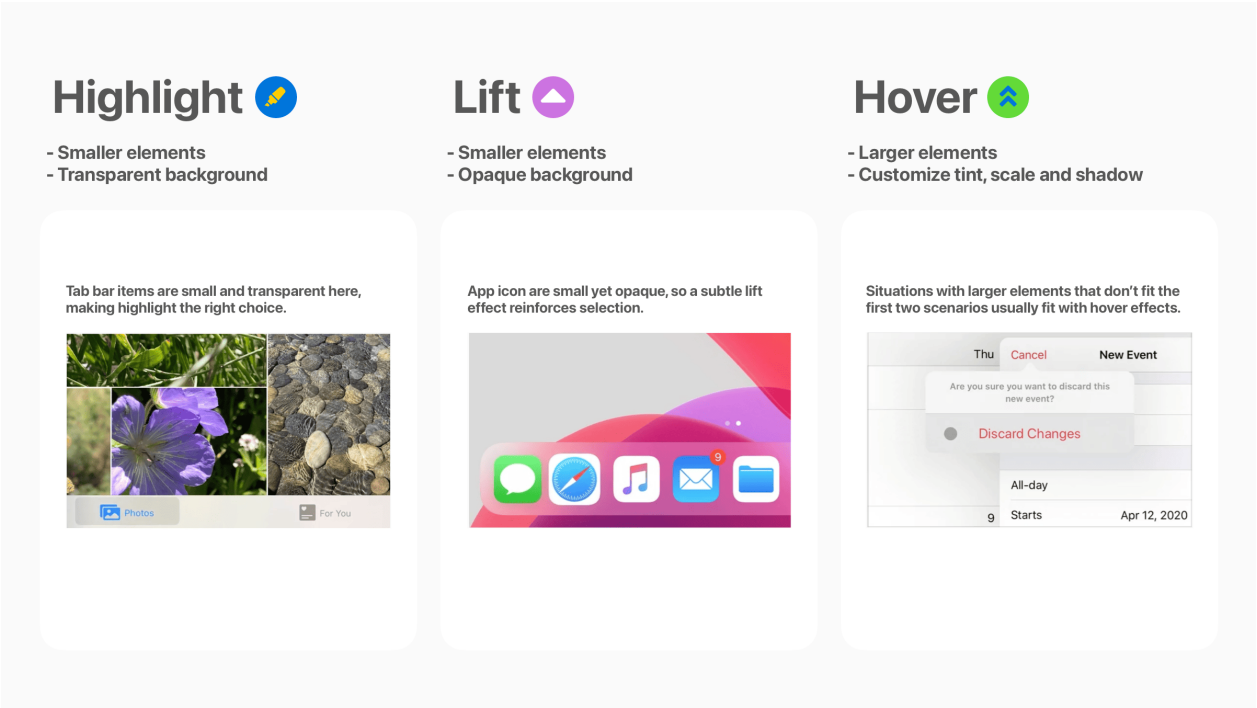
Hover

The last one is meant for larger elements where the pointer would feel jarring if it morphed into its shape. This works almost like a traditional hover effect on the web. You can pair this effect with a custom pointer shape too.

Automatic

Technically, there are four options for your effects — but this one just lets the system figure it out.

As always, it helps to visualize these to understand them:



Implementation of each of these effects in UIKit revolve around `UIPointerStyle`, which is composed of a `UIPointerEffect` along with a shape. These values are vended via an interaction delegate, much like how drag and drop works. By adopting `UIPointerInteractionDelegate`, you can control which pointer content effects to apply, and when to apply them.

Take a regular `UIView`, which by default has no pointer interactions. Here's how you could apply a lift effect to it:

```
// Xcode -> UIKit -> CursorFig2ViewController.swift

override func viewDidLoad() {
    super.viewDidLoad()
    let boxView = UIView(frame: .zero)
    // Add box view to hierarchy, see code sample in
    // Demo project for complete setup...

    // Create a pointer interactions
    let pointerInteraction = UIPointerInteraction(delegate: self)
```

```
    boxView.addInteraction(pointerInteraction)
}

// MARK: Pointer Delegate
 func pointerInteraction(_ interaction: UIPointerInteraction,
styleFor region:
UIPointerRegion) -> UIPointerStyle? {
    guard let interactionView = interaction.view else { return nil }
    // Use a lift effect
    let lift = UIPointerEffect.lift(UITargetedPreview(view:
interactionView))

    // Return a fully formed pointer style
    return UIPointerStyle(effect: lift, shape: nil)
}
```

Beyond stock view instances, there are some cases where you might want these effects for higher lever controls, such as `UIButton`. Note, again, buttons *don't* apply pointer effects by default. If you want to opt them in on a case-by-case basis, it looks a little different. You forgot the protocol adoption, and instead you'll need to set `allowsPointerInteraction` to `true` and then supply a pointer effect in its `pointerStyleProvider` closure.

In `CursorFig1ViewController`, I've got an example of each style being used with a button:

```
// Xcode -> UIKit -> CursorFig1ViewController.swift

liftButton.isPointerInteractionEnabled = true
liftButton.pointerStyleProvider = { button, proposedEffect,
proposedShape -> UIPointerStyle? in
    return UIPointerStyle(effect: .lift(proposedEffect.preview),
shape: proposedShape)
}
```

Beyond those effects, we can change the actual cursor shape itself. Consider the I-beam which shows when editing text. We could recreate that ourselves, like so:

```
// Xcode -> UIKit -> CursorFig3ViewController.swift
func pointerInteraction(_ interaction: UIPointerInteraction,
styleFor
region: UIPointerRegion) -> UIPointerStyle? {
    return UIPointerStyle(shape: .verticalBeam(length: 20),
                          constrainedAxes: .vertical)
}
```

The pointer shape API makes this straightforward. Above, we used the built-in `.verticalBeam` option, but there are several other shapes you can use along with the ability to supply your own custom shape:

```
public enum UIPointerShape {
    case path(UIBezierPath)

    case roundedRect(CGRect, radius: CGFloat =
UIPointerShape.defaultCornerRadius)

    case verticalBeam(length: CGFloat)

    case horizontalBeam(length: CGFloat)

    public static let defaultCornerRadius: CGFloat
}
```

Let's say you wanted to create a crosshair pointer. By creating that shape from a `UIBezierPath` – you could do pull that off effortlessly:

```
func pointerInteraction(_ interaction: UIPointerInteraction,
styleFor
region: UIPointerRegion) -> UIPointerStyle? {
    // Cross shair
    let shape = UIBezierPath()
```

```
    shape.move(to: CGPoint(x: 10, y: 6))
    shape.addLine(to: CGPoint(x: 10, y: 0))
    shape.addLine(to: CGPoint(x: 6, y: 0))
    shape.addLine(to: CGPoint(x: 6, y: 6))
    shape.addLine(to: CGPoint(x: 0, y: 6))
    shape.addLine(to: CGPoint(x: 0, y: 10))
    shape.addLine(to: CGPoint(x: 6, y: 10))
    shape.addLine(to: CGPoint(x: 6, y: 16))
    shape.addLine(to: CGPoint(x: 10, y: 16))
    shape.addLine(to: CGPoint(x: 10, y: 10))
    shape.addLine(to: CGPoint(x: 16, y: 10))
    shape.addLine(to: CGPoint(x: 16, y: 6))
    shape.addLine(to: CGPoint(x: 10, y: 6))
    shape.close()
    return UIPointerStyle(shape: .path(shape),
                          constrainedAxes: .vertical)
}
```

Moving over to SwiftUI, these effects can be used with the `.hoverEffect` modifier:

```
// Xcode -> SwiftUI -> CursorFig1View.swift

Button("Highlight") {

}
.buttonStyle(.plain)
.hoverEffect(.highlight)

Button("Lift") {

}
.buttonStyle(.borderedProminent)
.hoverEffect(.lift)
```

However, it *doesn't* include the hover version that UIKit offers. You can still pull it off, though, because in SwiftUI it's just further abstracted away from you via the `.onHover` modifier. Using it, you can apply the same type of effect:

```
// Xcode -> SwiftUI -> CursorFig1View.swift

Button("Hover") {

}
.buttonStyle(.plain)
.padding()
.background {
    hoverBackground.cornerRadius(8.0)
}
.onHover { isHovering in
    withAnimation {
        hoverBackground = isHovering ?
.gray : .clear
    }
}
```

Tips

Supporting Multi band Selection

Just like on macOS, you can drag and select several items at once on iPadOS. This was added in iOS 15, and expanded upon with iOS 16's release. Now, any collection view that *isn't* using a list configuration *and* also supports the bulk selection gesture via `UICollectionViewDelegate` gets this for free.

Check out `CursorFig4ViewController.swift` to try this out. Since our collection view isn't a list and implements bulk selection with the two finger gesture…

```
// Xcode -> UIKit -> CursorFig5ViewController.swift

func collectionView(_ collectionView: UICollectionView,
shouldBeginMultipleSelectionInteractionAt indexPath:
IndexPath) -> Bool {
```

```
    return true
}
```

…it all just works. If you click and drag the cursor over the items, they all become selected.

If you've got something more custom though, you can still support band selection by adopting `UIBandSelectionInteraction` manually.

```
// Create a UIBandSelectionInteraction object.
let bandInteraction = UIBandSelectionInteraction { [weak self]
interaction in

    switch interaction.state {
    case .began:
        // Change any state you might need to reflect the
        // Bulk selections
    case .selecting:
        // Handle selections, query modifier flags
    case .ended:
        // It's over, so commit the selections
    }
}

// Add the interaction as we did with pointer effect
myCustomGridView.addInteraction(bandInteraction)
```

Unfortunately, as far I know, this isn't available in SwiftUI.

Revealing Elements based on Pointer Movement

A great way to take advantage of the iPadOS cursor is to contextually hide and show interface elements. For example, in Safari – the address bar reveals itself when you hover over it with the pointer. To achieve this, just hook into the animator that `UI-`

`PointerInteraction` provides in the `willEnter` and `willExit` delegate methods:

```
// Xcode -> UIKit -> CursorFig4ViewController.swift

// MARK: Pointer Delegate
func pointerInteraction(_ interaction: UIPointerInteraction,
styleFor
region: UIPointerRegion) -> UIPointerStyle? {
    return UIPointerStyle(shape: .verticalBeam(length: 20),
                          constrainedAxes: .vertical)
}

func pointerInteraction(_ interaction: UIPointerInteraction,
willEnter
region: UIPointerRegion, animator: UIPointerInteractionAnimating) {
    self.dynamicView.frame.origin = view.center
    self.dynamicView.frame.origin.x -=
self.dynamicView.frame.size.width/2
    animator.addAnimations {
        self.dynamicView.alpha = 1
        self.dynamicView.frame.origin.y += 100
    }
}

func pointerInteraction(_ interaction: UIPointerInteraction,
willExit
region: UIPointerRegion, animator: UIPointerInteractionAnimating) {
    animator.addAnimations {
        self.dynamicView.alpha = 0
        self.dynamicView.frame.origin.y = self.view.center.y
    }
}
```

In SwiftUI, there isn't a specific API for this because, again – you can track hover events with its dedicated modifier. So, doing the same thing as we're doing here could be achieved that way.

Use Pointer Effects Correctly

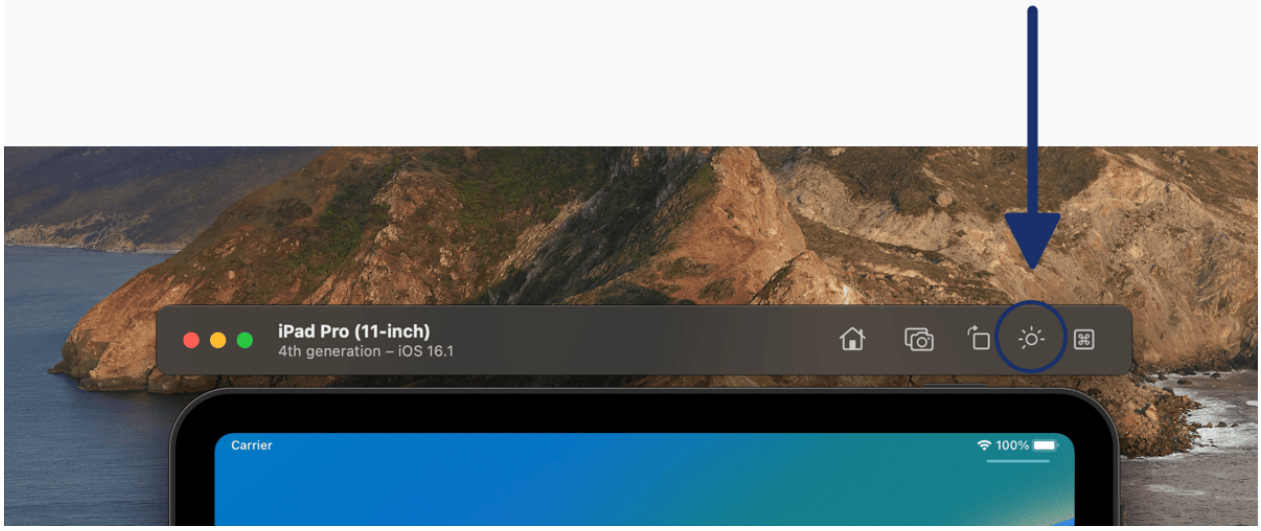
Here's a thought — why don't cells in a list, collection or table view show a highlight by default? Because they shouldn't.

After reading this chapter, don't apply a hover effect to every single table view row or every single button. There is a reason these controls don't show pointer effects by default. Remember, pointer effects drive context and reinforce selections — and large buttons and rows don't really need any more of those things. Their generous tap targets are enough — the pointer effects wouldn't genuinely add any real value in such cases.

Using the Simulator for Testing

I get it, not everyone has an expensive Magic Keyboard sitting around to test these things with. Fortunately, the iOS simulator has us covered. Simply select the circle shape in the top right, and the simulator will use your mac's trackpad or mouse as the cursor. When you're finished, simply hit escape on your keyboard to put things back to normal.

Testing with the iOS Simulator



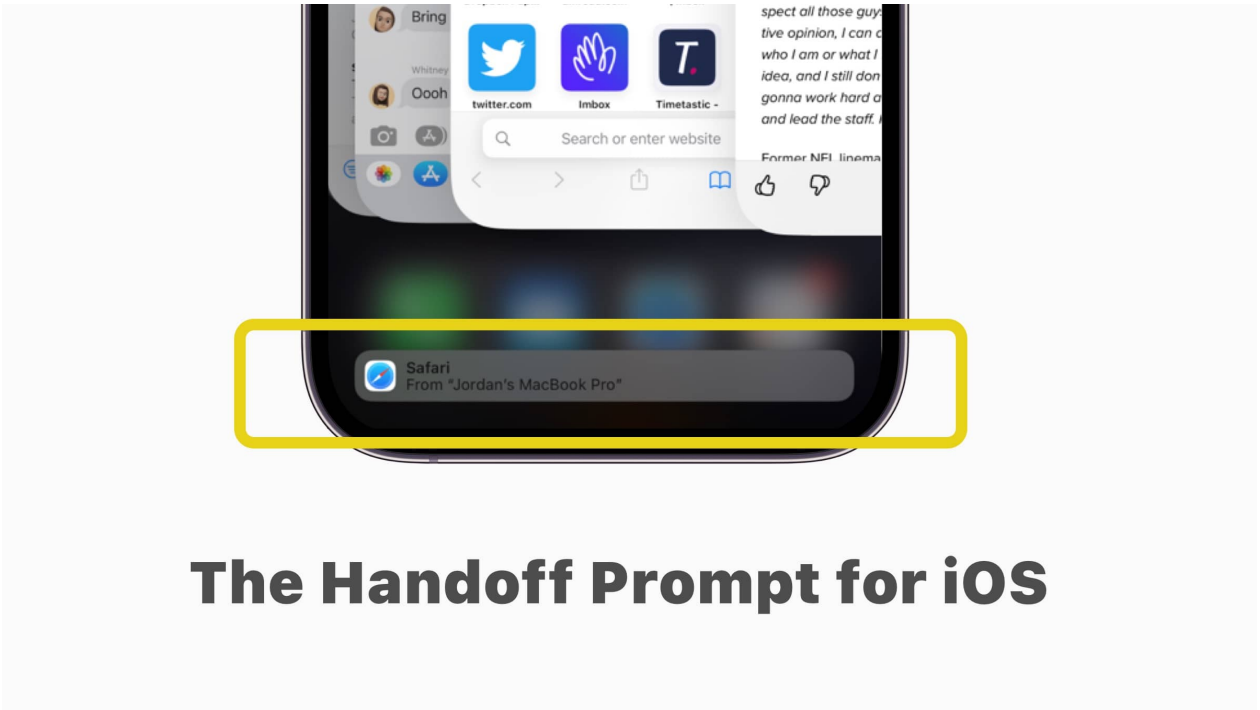
Three Key Takeaways

1. The iPadOS cursor is a unique input mechanism, it's not simply a traditional mouse pointer ported over to the platform.
2. Apply pointer effects correctly, and look for places where they don't quite look right by default.
3. Going further with lower level APIs to customize the pointer's shape can help reinforce what certain interface elements can do.

Handoff

Handoff is one of those things *only* Apple could pull off. Since it controls the entire hardware *and* software stack, it can take advantage of that tight integration to create experiences like it. If you're unaware, Handoff lets you start something on one device, and then seamlessly move that activity over to another one.

For example, open up Messages on your iPhone. If your iPad or Mac is near, you'll get a prompt to continue that activity on either of those devices:



The Handoff Prompt for iOS

This means as a developer or designer, you can tailor experiences that begin on an iPad that can be transitioned over to an iPhone, Apple Watch or a Mac. Further, you can even add support to your app's website. If your app has a web app too, you can hand users off to that as well.

The concept is simple, but the execution is what makes it such a win from a user experience standpoint — and that's why I've included it in this book. There isn't much to design for here, or any liberties you can really take with the API for that matter. In short, it's not an API which requires much expression from you. It's just really just about, "Here's how it works", and then you either support it or you don't.

But, for a book on user experience specifically on iOS — Handoff is such a smooth interaction and timesaver that it would be a shame to not talk about it. Plus, your app is always better off supporting it.

That said, Apple has the entire Handoff interaction all taken care of for us. So, as long as we implement the right APIs, we can offer Handoff support without a lot of ef-

fort. Plus, this is definitely one of those "surprise and delight" experiences which people love when they discover. In short, you'll want to do it.

For starters, this chapter will likely read a little different from the others in this book for the reasons listed above. It's more of a simple guide on how to get Handoff up and running since, again, there aren't many creative choices to make with it. It's just about getting the code in there and letting iOS handle the est.

Let's dive in.

Handoff Prerequisites

To begin, you need to understand how Handoff communicates with other devices. Each app that you want to participate needs to have the same developer Team ID. That's not really an issue because it just means you have to ship your apps through the App Store or sign them with the same credentials. This is the case for almost every app.

Next, from the user's perspective – they'll have to be signed in to the same Apple ID across devices. So, on each of their iPads, Macs – whatever, they'll all need the same Apple ID in use. Otherwise, Handoff won't recognize different versions of your app (i.e., its iPad version or its Mac version) even if everything else is set up correctly. Finally, Handoff requires that both Bluetooth and Wi-Fi are enabled.

I know this reads like a help doc manual, but making sure you've got these things correct upfront will save you the frustration of missing something obvious if Handoff isn't working after you've implemented all the necessary APIs.

How Handoff Works

Handoff is powered by user activities – and that's good, because *a lot* of iOS is. To name a few:

- Spotlight Search
- Siri Shortcuts
- Universal Links
- State Restoration
- Quick Notes
- Scene Targeting

By associating things that people do in your app along with a user activity, you'll have all you need to pick up that activity on some other Apple hardware. So, in Messages, if I'm writing a text to my friend – that's a user activity Apple is tracking behind the scenes. Then, if my iPad is near – the Handoff prompt appears, and tapping it will resume the same conversation over there.

At first glance, this concept appears to intimidate developers, but don't let it. You don't, and actually *should't*, need to add entire data graphs into user activities. Instead, it's mostly about providing identifiers or the minimal amount of information to pick things back up again. After all, you can't shuttle any more than 3 kilobytes of data in your user activity before you have to start implementing other APIs to support it.

So, from start to finish, here's what the entire flow will look like:

1. Someone starts a main flow in your app.
2. You create a `NSUserActivity` to represent that.
3. Ensure that activity is present in your app's Info.plist under `NSUserActivityTypes` (more on that below).

4. When the Handoff prompt is tapped, you recreate that main flow by looking at the user activity passed over to another instance of your app on another device.

Let's take a look at this start to finish.

Implementation

Like many other parts of iOS, such as permission prompts and certain platform capabilities, you've got to add some data to your info.plist to get started. Here, it'll be under `NSUserActivityTypes`, which is an array of `String` instances that are unique names of each activity you offer.

Apple recommends using a reverse-DNS pattern for these. So, something like `com.yourDomain.appName.activity`. For the sample project, I'll reuse the one we've already got in place from our discussion over state restoration. That one was called `com.bestinclasstour.openSampleActivity`, we implemented it like this:

```
// ActivityVendor.swift

let BookChapterActivityPListString : String =
"com.bestinclasstour.openSampleActivity"

enum MainActivityKeys : String {
    case SelectedChapter = "SelectedChapter"
    case SelectedInterface = "SelectedInterface"
}

extension UIViewController {
    func setSceneUserActivityFor(bookChapter: BookChapter,
interface: InterfaceFramework) {
        var sceneActivity = view.window?.windowScene?.userActivity
        if sceneActivity == nil {
```

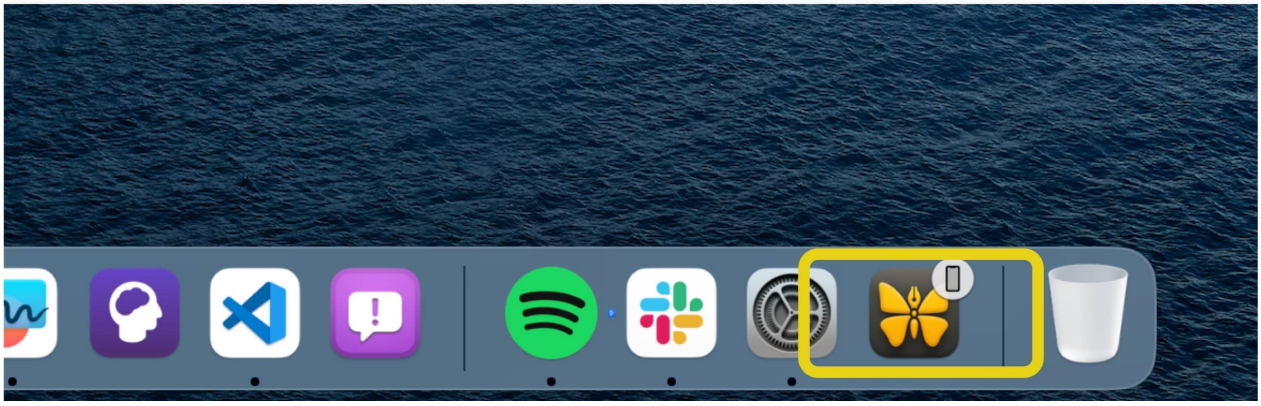
```
            sceneActivity = NSUserActivity(activityType:
BookChapterActivityPListString)
        }
        sceneActivity?.title = bookChapter.rawValue

        // Track the selected book chapter
        sceneActivity?.addUserInfoEntries(from:
[MainActivityKeys.SelectedChapter.rawValue:
bookChapter.rawValue])

        // Track the interface chosen
        sceneActivity?.addUserInfoEntries(from:
[MainActivityKeys.SelectedInterface.rawValue:
interface.rawValue])
        view.window?.windowScene?.userActivity = sceneActivity
    }
}
```

When someone picks a chapter in the book, that activity is saved:

```
func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
    guard let chapter = datasource.itemIdentifier(for: indexPath)
else {
        tableView.deselectRow(at: indexPath, animated: true)
        print("Could not find a chapter for the selected row.")
        return
    }

    // Update our user activity
    setSceneUserActivityFor(bookChapter: chapter, interface:
interfaceChoice)
}
```

This is all that's needed to make the Handoff interaction appear across devices. For example, I'm using the app Ulysses to write this book – and if I open my iOS device with it running in the foreground next to my Mac, the Handoff prompt will appear:

The Handoff Prompt for macOS



Now, it's all about handling it within the app that's receiving the Handoff. That's done with some delegate methods in your app's scene delegate. The first two let you know the Handoff interaction is starting, or if it failed. Both are optional:

```
func scene(_ scene: UIScene, willContinueUserActivityWithType
userActivityType: String) {
    // The app is about to receive the handoff. Update any state.
}

func scene(_ scene: UIScene, didFailToContinueUserActivityWithType
userActivityType: String, error: Error) {
    // If the Handoff failed
}
```

Now, the important one is here – this is where you actually act on the Handoff interaction via the user activity supplied:

```
func scene(_ scene: UIScene, continue userActivity: NSUserActivity)
{
```

```
    openBookChapter(with: userActivity)
}
```

And that's it! Handoff is up and running. To refresh your memory on `openBookChapter(activity: NSUserActivity)` — we also used that for state restoration. Here's what it looks like:

```
private func openBookChapter(with activity:NSUserActivity) {
    if let userInfo: [AnyHashable:Any] = activity.userInfo,
       activity.activityType == BookChapterActivityPListString,
       let chapter = BookChapter(rawValue:
userInfo[MainActivityKeys.SelectedChapter.rawValue] as? String ??
""),
       let interfaceChoice = InterfaceFramework(rawValue:
userInfo[MainActivityKeys.SelectedInterface.rawValue] as? Int ?? 0)
{

        guard let rootController = window?.rootViewController as?
GlossaryViewController else {
            return
        }

        // Open up back to the last viewed chapter
        rootController.pushBookChapter(chapter: chapter, with:
interfaceChoice)
    }
}
```

You might be able to reuse some logic from state restoration and user activities involved in Handoff. That's what we do here in the sample project. That same function is also used when the scene session connects and there is an activity to resume. Check out `SceneDelegate.swift` to see it in action.

SwiftUI

Now – I want to stop right here and acknowledge that the sample project's view lifecycle and navigational structure are built with UIKit. If you are using the SwiftUI scene lifecycle, you'll use the `.userActivity` modifiers to advertise your activity and `.onContinueUserActivity` to handle it:

```
// Xcode -> SwiftUI -> HandoffFig1View.swift

struct HandoffFig1View: View {
    @State private var text: String = "This View advertises a
NSUserActivity for us."

    private let mockActivityID: String = "com.test.example.activity"
    private let payloadID: String = "SomeID"

    var body: some View {
        Text("This View advertises a NSUserActivity for us.")
            .userActivity(mockActivityID) { activity in
                activity.addUserInfoEntries(from: [payloadID:
UUID().uuidString])
            }
            .onContinueUserActivity(mockActivityID) { activity in
                guard let identifier = activity.userInfo?[payloadID]
as? String else {
                    return
                }

                text += "\nFound \(identifier) from user activity."
            }
    }
}
```

Please note, that again – this example won't do anything since we're not using the SwiftUI view lifecycle in the sample app, but this is what the flow would look like.

Tips

Have Sensible Routing Code

Handoff is primarily about *navigation*. That is, I'm doing something over here, and now I want to do it over *there*. As such, you're routing people to different places within your app.

Now, we could write a whole book series over sensible routing and navigation architecture in SwiftUI and UIKit. We aren't going to do that here, but I do want to encourage you to think about navigation and routing, ideally early and often. There are several cases outside of just Handoff where you'll need it, such as responding to push notifications or state restoration.

Mix and Match Activities Across Platforms

Supporting Handoff, by definition, means that you have your app running on more than one platform. Because of this, you don't *have* to support the same types of activities in each one. In fact, in the case of watchOS, you simply *can't*. On the watch, you can only send activities — not respond to them.

So, leverage that. If you've got a mature, fully powered macOS app, I think it's fine if it supports more activities than its iOS or iPadOS counterpart. Maybe their functionality differs. In short, you can mix and match activity support across your app binaries.

Version Payloads

Your app will change, and functionality will eventually be added, changed or perhaps removed altogether. To safely account for this, consider versioning your payloads by adding a version number in your activity's `userInfo` dictionary:

```
let data: [AnyHashable:Any] = ["myID":UUID().uuidString,
                               "myActivityVersion":2]

myActivity.addUserInfoEntries(from: data)
```

That way, no matter if the activity finds itself on a new or old version of your app with differing functionality, you can protect against any crashes or odd behaviors.

Don't Send Entire Data Models

I'll keep this short because I covered it above. Don't send entire data models in your activity dictionaries. Instead, move over "just enough" to have the information you need to route people to the right place.

Here's an example that might help you frame your thinking. In Messages, when I use Handoff it doesn't send the current message I'm composing, it just opens the right thread. That's all it needs to do. Less is more.

Handling File URLs

If you pass URLs in your activity, you'll need to call `startAccessingSecurityScopedResource()` and return `true` before you can operate on it. This may not seem obvious at first, but if you miss it – you'll likely see logs in Xcode or you can look at the `Error` you'll get from the file manager when you try to access it.

Also, when you're done – you'll need to balance the call above with `stopAccessingSecurityScopedResource()`.

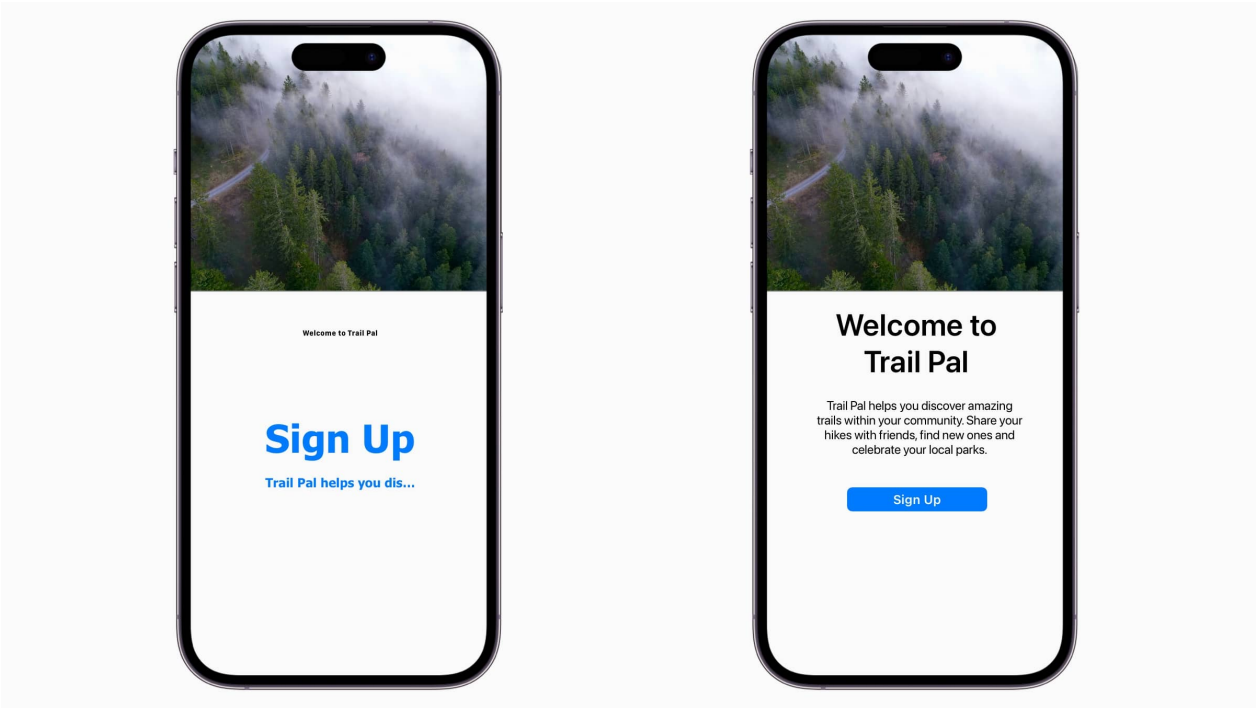
Three Key Takeaways

1. Handoff allows people to seamlessly move from device to device while using your app.
2. Implementation revolves around user activities — and so do a lot of other iOS technologies, so it pays to implement them accordingly.
3. There aren't many technical or design liberties you can take with Handoff, but it does pay to have your app setup to facilitate routing people to certain parts of it easily.

Text that Works

The way you present text can make or break the user experience in your iOS app. Text that is hard to read, doesn't respond to accessibility needs, or that clips off important information is frustrating. Text that responds to accessibility preferences, conveys hierarchy through its size and weight, and actively responds to its available space is text that works.

Those aren't all the things that make text work, but I believe they are the pillars. Consider this example. One of these interfaces looks elementary in its design and execution, and the other is fine, if not unremarkable. The only differences are in how the text is handled:



The left side example and its seemingly random font colors are off-putting. The clipped text means that you can't even read what the benefits of the app are. Further, there is no visual hierarchy at all – the "header" here is smaller than the rest of the text.

It's ugly, hard to read and honestly looks like malware. You'd think this is a contrived example and that there is no way something like this would ship. But, if that were true, this chapter wouldn't need to exist. Yet, here it is.

Conversely, the right side example doesn't do much differently, but it also feels worlds apart. By simply changing text weights, colors and sizes it achieves the mark of text that works.

This chapter is all about how to make text that works for you, and not against the people who use your apps. Throughout the years, there are several little nuggets of information I've learned about handling, displaying and adapting text. In no particular order, I'm going to list out those things in this chapter. Text, while not particularly exciting to some as a whole, is ripe with opportunity in iOS.

First, I want to present some of the foundational beliefs I have about how text should work in iOS apps. Some of these ideas have been met with pushback in my career, and you might even find yourself disagreeing with a few. Whether you agree or not, I hope that these items make you consider how you are treating text in your designs and apps:

1. Text should respond to accessibility needs over design preferences.
2. Text should typically opt to wrap, not clip, its contents.
3. Text should convey hierarchy and meaning through font sizes, weight, and colors.
4. Text should leverage platform features to help people use that text how *they* want to, not how you think they should.

As we go on, I think you'll find that all the tips I list support those positions. To that end, it helps to know about the tools you have at your disposal:

UIKit

Control	**Purpose**	**Example**
UILabel	Short form text display, not meant for editing.	Labels, headers, etc.
UITextField	Single line text display, meant for editing.	Email address entry
UITextView	Multi-line text display, meant for long form editing.	Writing a chapter in a book.
TextKit2	Meant for lower level text rendering.	Custom text controls.

And, over in SwiftUI:

- `Text` correlates to `UILabel`
- `TextField` correlates to `UITextField`

- `TextEntry` correlates to `UITextView`

Note that TextKit 2 isn't directly available in SwiftUI, but it might be used for rendering under the hood — I'm honestly not sure. But, you can use representables to bridge those things over to your SwiftUI projects if you require them. With that, let's dive right into how to make text that works with some tips.

Tips

Dynamic Type and Text Styles

If you only take away one thing from this chapter, this should be it — *use* Dynamic Type. There are many, many compelling reasons to use it. There are *not* very many reasons to skip it.

People should decide how legible, thick, thin, large or small their text should be — and as designers and developers, we should accommodate that. Thankfully, in SwiftUI, you almost have to try to **not** use Dynamic Type from an API level.

Out of the box…

```
Text("Testing!")
```

…SwiftUI will use the `.body` text style. Unless you specify an explicit font size, SwiftUI is using Dynamic Type.

To that end, setting a new font should typically be done using the `func system(_ style: Font.TextStyle, design: Font.Design? = nil, weight: Font.Weight? = nil) -> Font` modifier. When you use a `TextStyle` with SwiftUI over a hardcoded size to represent font sizes, you can rest assured your font will always scale with the person's Dynamic Type settings.

In UIKit, Dynamic Type is more of an opt-in affair:

```
let dynamicLabel = UILabel(frame: .zero)
// Account for text wrapping
dynamicLabel.numberOfLines = 0
dynamicLabel.text = "Responds to Dynamic Type"

// Opt in to Dynamic Type
dynamicLabel.adjustsFontForContentSizeCategory = true

// Use a text style
dynamicLabel.font = UIFont.preferredFont(forTextStyle: .body)
```

When you leverage Dynamic Type, you also get away from thinking about specific font sizes. It's less about "This text should be 14 points" and more about "What role does this text play?" From there, you can match it to the correct text style:

- **Large Title:** For primary or title headings.
- **Title**: For first level hierarchical headings.
- **Title Two**: For second level hierarchical headings.
- **Title Three**: For third level hierarchical headings.
- **Headline**: For body headings.
- **Sub Headline**: For sub headlines that follow headlines.
- **Body**: For primary text, perfect for reading long-form text.
- **Callout**: For text callouts, like tool tips.
- **Footnote**: As the name suggests, for footnotes.
- **Caption**: For standard captions.
- **Caption** Two: For alternative captions.

If you're new to Dynamic Type, you may be thinking at this point that all of these sounds more like *semantic* types, and not *dynamic* ones. If they are meant for a singu-

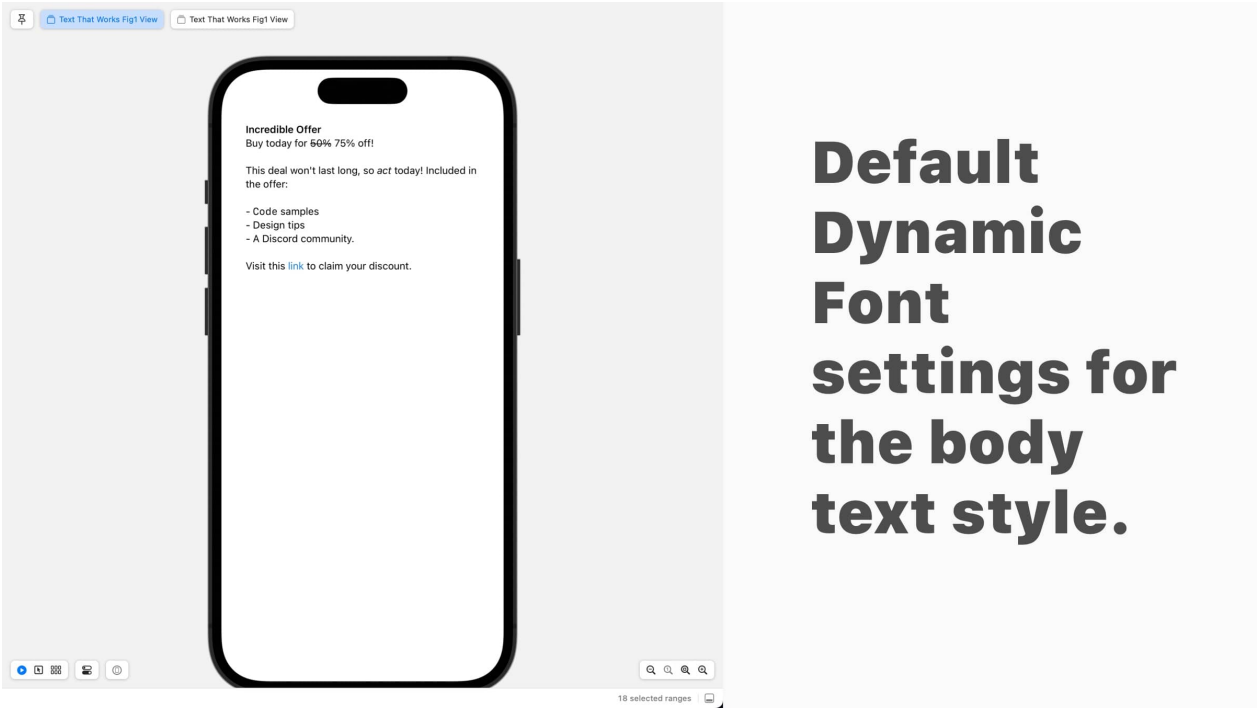
lar use case, what's dynamic about them? Well, the dynamic parts are enabled *by* the semantic meanings you use for fonts.

When you use a font in a text control with a `.body` text style – it's dynamic in that it could be one of *several* different sizes and weights based off the user's accessibility settings. Based off of what they've chosen, the system vends a font that is geared for a "body" type of text scenario. That means it might be 14 points if the user prefers the extra small settings, all the way up to 53 with the largest accessibility size. That's no small difference of 39 points!

For an advanced and in-depth discussion over Dynamic Type, please see the "Dynamic Type" chapter in the first book in this series covering accessibility.

Scrolling Views

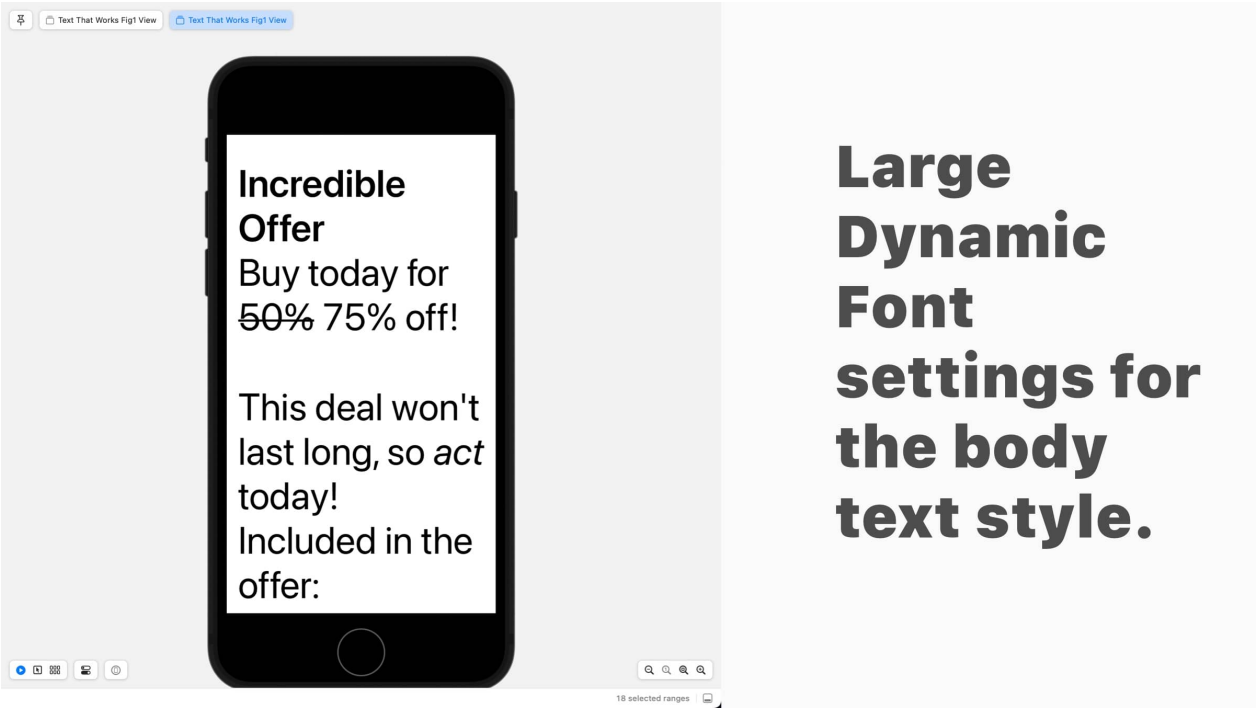
Continuing on with Dynamic Type, we just saw that any text might be fairly reasonably sized, or it could be very large. Consider the delta between these two examples. Here's some text at the regular `.body` size, basically an iPhone's out-of-the-box defaults:



It doesn't appear to ever need to scroll. The code assumes as much, too:

```
// Xcode -> SwiftUI -> TextThatWorksFig1View.swift

struct TextThatWorksFig1View: View {
    let markdown: LocalizedStringKey = """
**Incredible Offer**
Buy today for ~50%~ 75% off!
This deal won't last long, so _act_ today! Included in the offer:
- `Code` samples
- Design tips
- A Discord community.
Visit this [link](https://bestinclassiosapp.com) to claim your
discount.
"""
    var body: some View {
        Text(markdown)
            .padding()
    }
}
```

It fits onto a typical iPhone, and it's easy to think that'd there be no reason for it to ever clip or truncate. But, let's go to one end of the extreme – the smallest available iPhone *and* the largest available accessibility size:

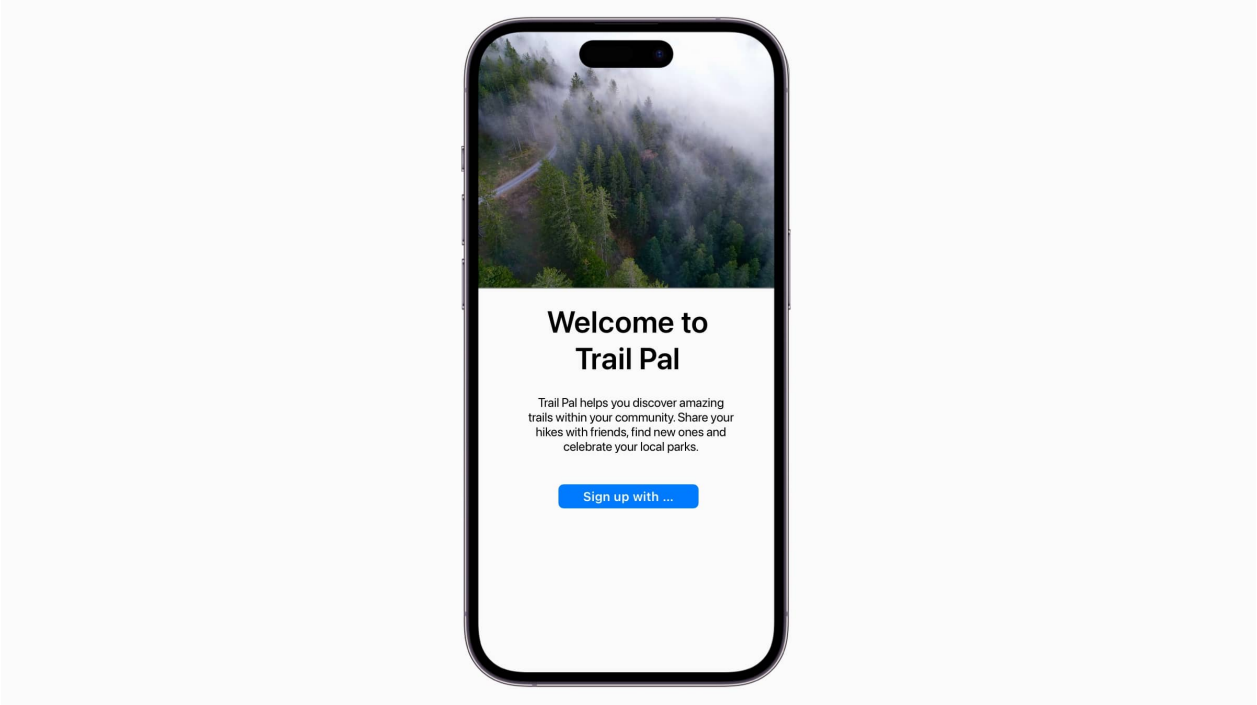


And, now you see why nearly every view should be a scrolling view. In fact, I start implementation with a scrolling view when I begin development, and I recommend you do the same. In SwiftUI or UIKit, our fix for this example is incredibly easy – simply wrap it in a `ScrollView` or `UIScrollView`. For example, in SwiftUI:

```
var body: some View {
    ScrollView {
        Text(markdown)
            .padding()
    }
}
```

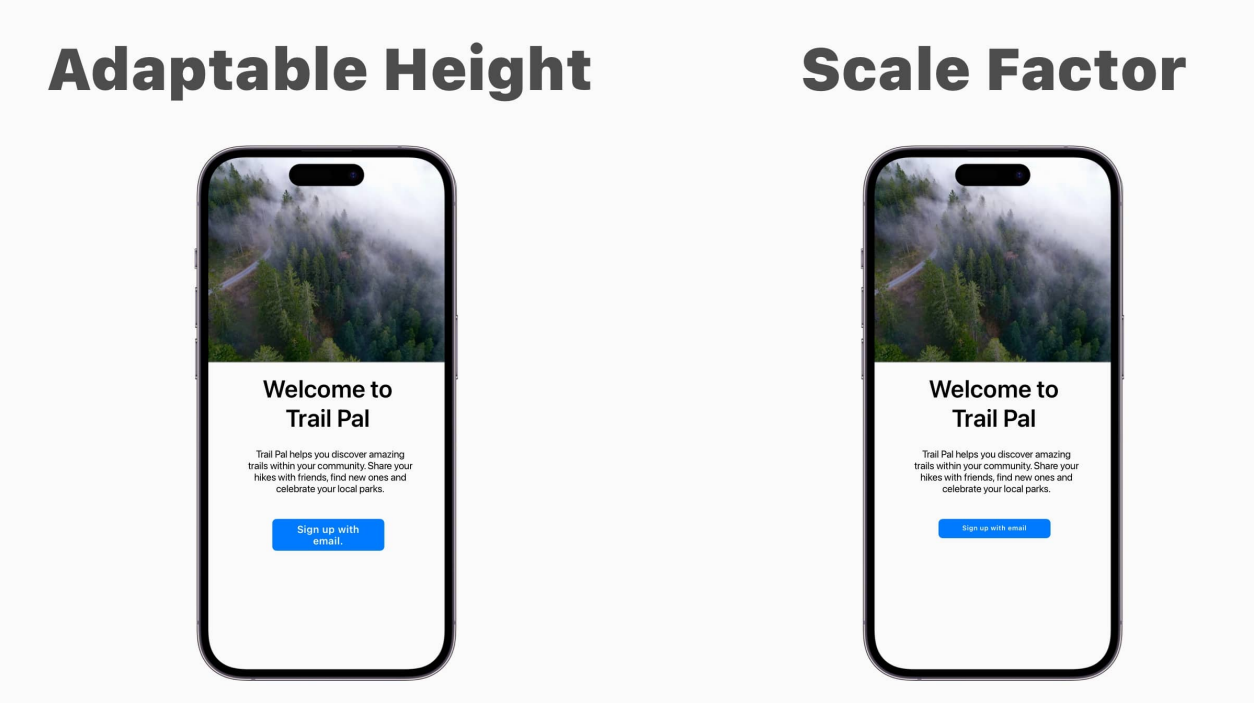
Avoid Clipping or Truncation, Promote Wrapping Text

If your views are primarily scrolling, it also becomes easier to wrap text instead of letting it clip or truncate. There are a few design reasons to clip text, but if the text is part of a larger body of information – typically, you'll want to avoid it. This is doubly true if the text confirms some sort of action. Wouldn't something like this be incredibly frustrating?



Sign up with…what? Email? Social? Something else?

In practice, this means that instead of providing maybe a flexible width with a fixed height – you do the opposite. You'll want flexible height, and optionally flexible width. Or, you can leverage minimum scale factors – which will shrink your text to fit its container down to a certain font size. Here's what those two options might end up looking like:



On the left side option, the button's height expands to fit its contents. On the other hand, using a minimum scale factor – the height is fixed, but the font adapts to the space that it has available to avoid clipping text. This is available in both UIKit and SwiftUI:

```
// SwiftUI
Text(markdown)
    .minimumScaleFactor(0.4)

// UIKit
someLbl.minimumScaleFactor = 0.4
```

Using a minimum scale factor may seem a bit nebulous at first – but it's simple. When you provide a value from 0 to 1, you're saying how small from the text control's current font size that you want it to shrink down to. So, for example, say your font size is 10 points, and you use 0.5 as a minimum scale factor. Here, you're telling the system "You

can shrink this down to, but not any lower than, a 5 point font size if it'll fit the text. If it still doesn't fit – then just clip it."

So, the text could end up being anywhere from 10 to 5 points – or you'll just get truncated if none of the sizes work.

Question Custom Fonts

Simply put, there is just so much that the system fonts give you. There are hundreds of accessibility sizes, varying font weights and more. Plus, they align nicely with Apple's expansive set of built-in iconography, SF Symbols (something we will look more at in the next chapter).

In short, you'll be giving up quite a lot if you opt to use a custom font in your app. Really question if it's necessary, and as we said above – it's better to lean into accessibility needs when it comes to typography over design preferences. And, the system fonts are tailor-made for every single accessibility feature that iOS offers.

If you do consider custom fonts, perhaps save them for limited uses where the benefit of the font's personality is still achieved without all the other baggage that might be incurred if you used it all over. It minimizes risk while letting you inject some branding personality into your designs. For example, perhaps only using a custom font for just headers or titles might work nicely.

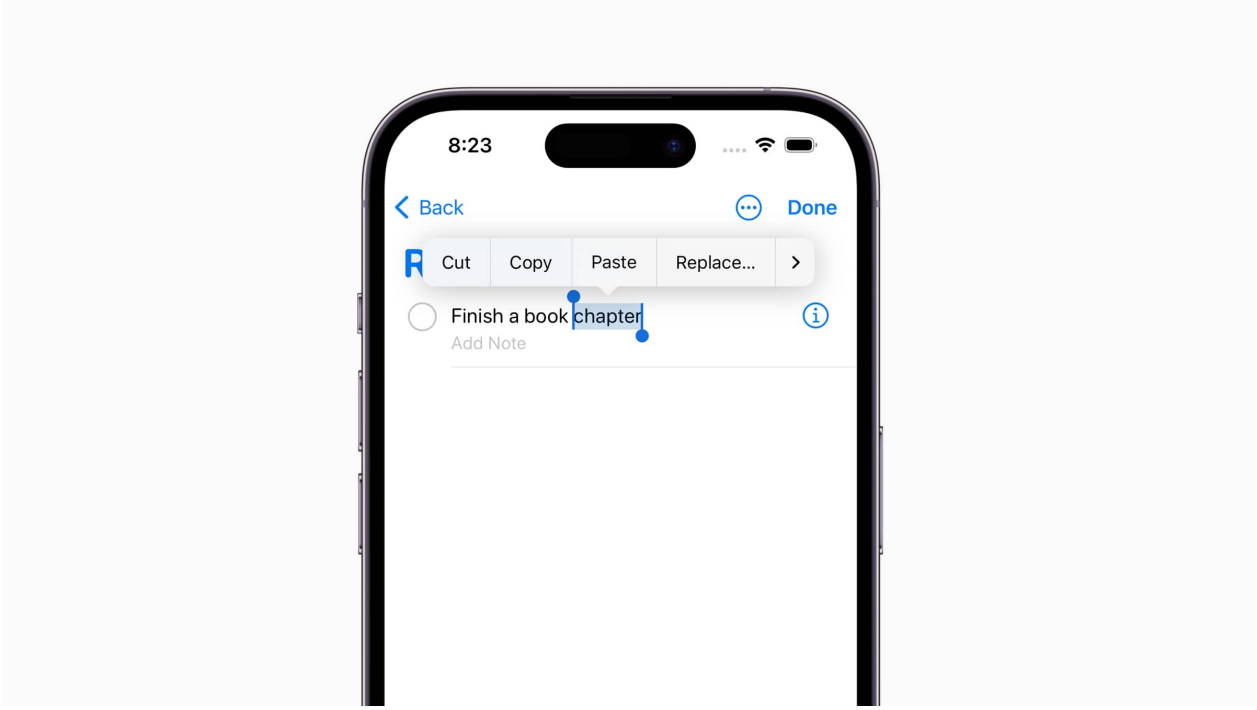
Again, for much more information over supporting Dynamic Type, please visit its relevant chapter. For a small dose of what you might be in for in terms of implementation with custom fonts to support Dynamic Type, it really comes down to `UIFontMetrics`:

```
guard let futuroFont = UIFont(name: "Futuro-Bold", size: 20) else {
    print("Unable to load custom font.")
    return
```

```
}

// Assign scaled font
myLabel.font = UIFontMetrics(forTextStyle: .body)
               .scaledFont(for: futuroFont)

// Opt into Dynamic Type
myLabel.adjustsFontForContentSizeCategory = true
```

Editing Controller

The editing controller is what shows when you select some text and tap and hold on it:



While iOS will vend sensible text for an editing controller, you're also able to put in your own custom actions in here too. Plus, even if you've got a custom text control – so long as you override common editing patterns in UIKit, such as `paste(_ sender:`

`Any?)`, and your control is first responder, these actions will still show up, too. Revisit the chapter over keyboard experiences to see how to implement some of these if you need a refresher.

But, to see how you might use this – consider a markdown editing app you're developing. Perhaps you could include some common actions, like to make some text a heading style. To do so, we can leverage functions on `UITextViewDelegate` to provide custom editing actions. Here, we add a "# H1" option to the menu:

```
// Xcode -> UIKit -> TextThatWorksFig1ViewController.swift

func textView(_ textView: UITextView,
              editMenuForTextIn range: NSRange,
              suggestedActions: [UIMenuElement]) -> UIMenu? {
    var customActions: [UIMenuElement] = []

    // If text is selected, add the custom action
    if range.length > 0 {
        let highlightAction = UIAction(title: "# H1") { _ in
            var attributedString = AttributedString(textView.text)
            if let slicedText = textView.text.substring(with:
range),
               let range = attributedString.range(of: slicedText) {
                attributedString.font =
UIFont.preferredFont(forTextStyle: .subheadline)
                attributedString[range].font =
UIFont.systemFont(ofSize: 24, weight: .bold)
                textView.attributedText =
NSAttributedString(attributedString)
            }
        }
        customActions.append(highlightAction)
    }

    // Return the system actions, plus our own
    return UIMenu(children: suggestedActions + customActions)
```

```
}
```

If you run this sample, you'll now be able to make a heading styled font with any selected text from the edit menu. As long as you append it to the `suggestedActions` iOS gives you – you can vend your own actions alongside the system ones.

Plus, if the action is something the user may want to do several times (such as indenting text) you can have the editing controller persist even after the action been tapped. This lets people perform the same action several times without having to reopen the editing controller. You can do that by leveraging the `attributes:` argument in `UIAction`'s initializer. For our sample above, we'd just change its initializer to achieve this:

```
// From this
let highlightAction = UIAction(title: "# H1") { _ in }

// To this
let highlightAction = UIAction(title: "# H1",
attributes: .keepsMenuPresented) { _ in }
```

The editing controller is a mainstay of iOS textual experiences, so don't be afraid to add in helpful actions to them. Plus, as we did in our example, use the APIs to add or remove relevant actions based on whether text is selected. For example, in a markdown editing app – perhaps you'd want "Add Divider" as an option all the time, no matter if text is selected or not.

Text Content Types

You'd be hard-pressed to find a feature in iOS more widely appreciated than text autofill. We've all had a two-factor authentication code come in, and then iOS simply fills

it in for us with the tap of a button. It's extremely convenient, and it epitomizes user experience. Those types of experiences are driven by assigning a text content type.

These text content types assign a semantic meaning to the text control, and when iOS has that information, it can use contextual cues from the system to suggest text completions. In addition, it will pick the most relevant keyboard type to use too. In short, this supercharges text entry for many scenarios. The system can use information you've given it, such as names, relationships, locations and more – for easier form completion, password resets and much more:

Value	**Purpose**
URL	Defines the content in a text input area as a URL.
namePrefix	Defines the content in a text input area as a prefix or title, such as Dr.
name	Defines the content in a text input area as a name.
nameSuffix	Defines content such as a suffix, like Jr.
givenName	Defines text as a first name.
middleName	Defines text as a middle name.
familyName	Defines text as a family or last name.
nickname	Defines text as a nickname.
organizationName	Defines text as an organization.
jobTitle	Defines text as a formal job title.
location	Defines text as a general location, such as a point of interest, an address, or another identifier for a location.
fullStreetAddress	Defines text as a fully qualified street address.
streetAddressLine1	Defines text as the first line of an address.
streetAddressLine2	Defines text as the second line of an address.

addressCity	Defines text as a city name.
addressCityAndState	The same as above, only a state is included as well.
addressState	Defines text representing a state's name.
postalCode	Suitable for postal code entry.
sublocality	Text that represents a sub locality.
countryName	Defines text that is a country or region's primary name.
username	Represents an account or login name.
password	Suitable for password entry.
newPassword	The same as above, only this is for when someone might be creating a new password.
oneTimeCode	The example we lead with – two-factor authentication codes.
emailAddress	Text entry for emails.
telephoneNumber	Same as above, except for phone numbers.
creditCardNumber	For credit card numbers.
dateTime	Suitable for general dates, times, or a duration.
flightNumber	For entering in flight numbers.
shipmentTrackingNumber	For parcel delivery or general postal tracking.

Using them is as easy as setting a value for `textContentType` in UIKit or the identically named modifier in SwiftUI:

```
// UIKit
label.textContentType = .URL

// SwiftUI
Text("testing")
    .textContentType(.URL)
```

ViewThatFits

In a world where an iOS or iPadOS device could be any number of sizes or form factor – it pays for text to be adaptable. This wasn't always an easy task in SwiftUI, but it's been made much easier with `ViewThatFits`. As the name suggests, it allows you to pass a closure of `Views` – and it'll display the one that fits the available space.

Basically, provide the views from largest to smallest, and it'll go down and pick the one which first fits the given axis you specify (horizontal by default). This is perfect for text. In this example, we provide a longer string if it'll fit, but a shorter one if it doesn't:

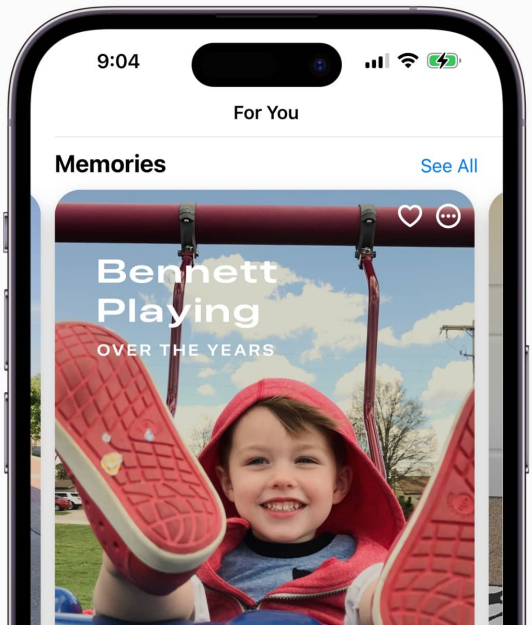
```
// Xcode -> SwiftUI -> TextThatWorksFig2View.swift

struct TextThatWorksFig2View: View {
    @State private var fontSize: CGFloat = 20.0

    var body: some View {
        VStack {
            Text("New Menu Items!")
                .font(.largeTitle.bold())
            Image("Oriental Food")
                .resizable()
                .scaledToFit()
            Text("Abbreviated - see code sample")
                .font(.caption)
                .padding(.bottom, 16)
            ViewThatFits {
                Text("Try it out for free today!")
                Text("Try today!")
            }
            .font(.system(size: fontSize,
                          weight: .black,
                          design: .default))
        }
        .padding()
        .onTapGesture {
            fontSize = (fontSize == 20.0) ? 40.0 : 20.0
```

```
            }
        }
}
```

If you ran the sample on your phone, depending on the screen size available, it'll toggle between the two strings of "Try it out for free today!" and "Try today!" based on how much width is available when you tap on it. This toggles the font size up and down accordingly, this forces `ViewThatFits` to reevaluate if the larger string still fits on the horizontal axis even with the larger text. If it doesn't, we'll get the shorter string.

Variable System Fonts

Another reason that system fonts are so handy is that they offer several stylistic choices you can apply to them via font widths. This allows for more creative freedom in your typography. Photos' memory feature on iOS uses these, and they look great. Here's an example from my device, where "Bennett Playing" is using the expanded font width:



The available font widths we can use are:

- Compressed
- Condensed
- Expanded
- Standard (the default)

It's available in both SwiftUI and UIKit:

```
// SwiftUI
Text("Expanded Font")
    .fontWidth(.expanded)

// UIKit
someLbl.font = .systemFont(ofSize: 20,
                           weight: .bold,
                           width: .expanded)
```

When you combine different font widths with weights, you can achieve some very nice effects. In addition, these animate nicely in SwiftUI as well – so you could interpolate between a compressed font width to an expanded one.

Leverage Safe Areas

Who knows what else Apple will add in terms of hardware. A few years ago, it was the notch. Today, as I write this – it's the Dynamic Island. The point is, you would rather not have to play a guessing game when displaying text which won't be occluded by these hardware features.

The easiest way to do that is to respect safe area layout guides and margins. Beyond just hardware additions, there are several other reasons to leverage them from

handling multitasking, external displays, display zoom, internationalization features, differing resolutions or colors spaces – the list goes on.

In SwiftUI, the layout system will use the safe area layout under the hood for you. In fact, you'd have to manually opt out of this behavior – so in reality, you don't have to think about it too much. But, even in container views – especially in UIKit, it pays to constraint to the safe area layout guide. An example of this can be found in the previous UIKit example:

```
// Xcode -> UIKit -> TextThatWorksFig1ViewController.swift

NSLayoutConstraint.activate([
    textView.widthAnchor.constraint(equalTo:
view.safeAreaLayoutGuide.widthAnchor),
    textView.heightAnchor.constraint(equalTo:
view.safeAreaLayoutGuide.heightAnchor),
    textView.centerXAnchor.constraint(equalTo:
view.safeAreaLayoutGuide.centerXAnchor),
    textView.centerYAnchor.constraint(equalTo:
view.safeAreaLayoutGuide.centerYAnchor)
])
```

Use Xcode Previews for Both UIKit and SwiftUI

There is no better way to quickly test font sizes than Xcode Previews. This is true for both UIKit and SwiftUI. By using the `.sizeCategory` environment key, you can either loop through all the content size categories or specify individual ones.

This is precisely how I demonstrated a previous example in this chapter, where we saw the need for scroll views. I simply used the smallest size and the largest size for the Xcode Preview:

```
// Xcode -> SwiftUI or UIKit -> TextThatWorksFig1View.swift
```

```
struct TextThatWorksFig1View_Previews: PreviewProvider {
    static var previews: some View {
        // Small size
        TextThatWorksFig1View()
            .environment(\.sizeCategory, .extraSmall)

        // The largest
        TextThatWorksFig1View()
            .environment(\.sizeCategory,
.accessibilityExtraExtraExtraLarge)
    }
}
```

Plus, you can use the same trick with UIKit views. In fact, that's how I develop the majority of this book series' UIKit samples – by using Xcode Previews! If you look in the sample project under `Logistical - No Code Samples Here --> Preview Helpers` you'll see code I have to use UIKit along with Xcode Previews. Please don't hesitate to steal it for your own projects to quickly test things like font sizes.

Three Key Takeaways

1. Try to follow the foundational rules of text in your apps to make sure it works well for everyone.
2. Text, above all, is meant to convey information – so respond to all the ways it could be displayed.
3. Leverage platform features to make sure people can get text in and out quickly and use it how they want.

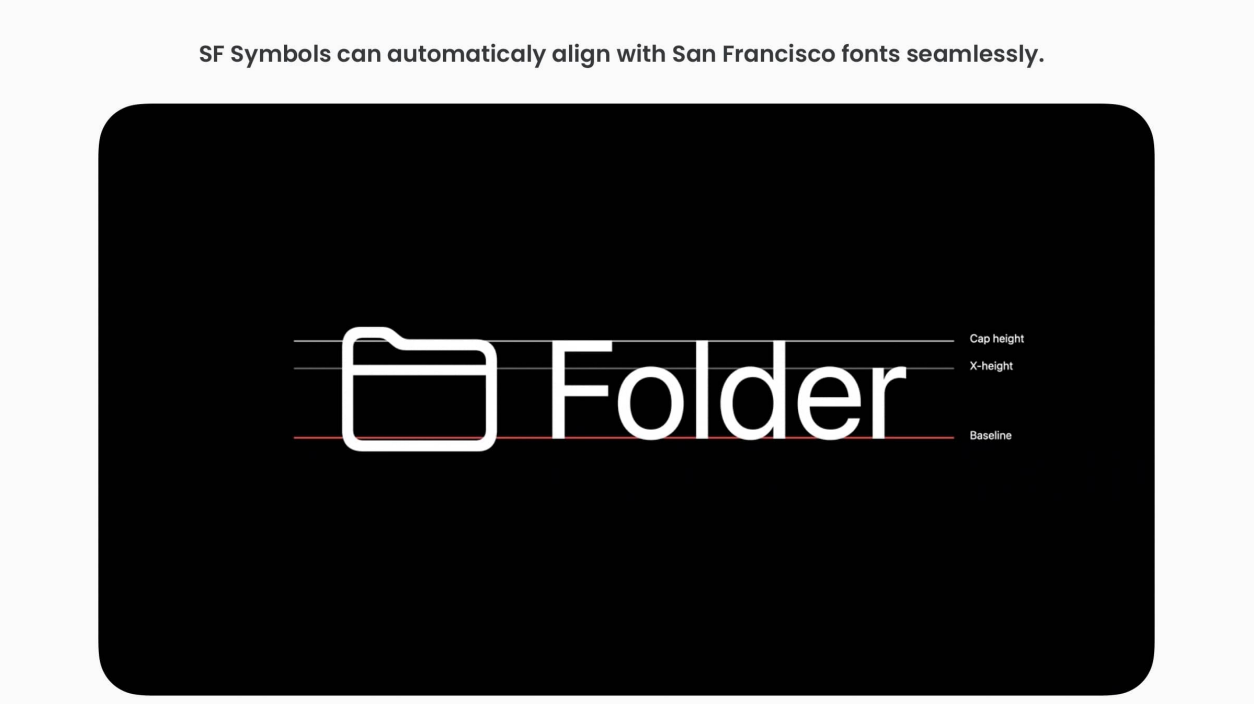
SF Symbols

SF Symbols is a set of over 4,000 and growing vector-based icons. The set spans over several categories, such as gaming, time, health and more. It's bundled along with iOS – which means that using them adds no weight or additional size to your app's binary.

SF Symbols are expressed as *images*, but their API and sizing works closely with *text*. And, while that may seem a little counterintuitive at first, as we'll see – SF Symbols does a lot of work under the hood to ensure that they look spectacular when used with system fonts. In fact, you typically size symbols by a font size in most cases rather than a specific frame size.

SF Symbols have many options to configure, making them flexible for any design. You can tweak their weight, image scale, size, color or how they render. They also support numerous localizations and accessibility features. They'll adapt to settings like Dynamic Type or Bold Text out of the box. Additionally, they also respond to plenty of

tricky alignments situations when they are used with Apple's San Francisco font family (the default font for iOS and iPadOS).



From an implementation and design standpoint, it's good to remember that symbols will *optically* vertically center themselves next to text. Occasionally, this means it might be higher or lower than the text's baseline. Additionally, symbols will also intelligently alter their stroke's thickness to make symbols match up with its surrounding text. In short, there is a lot of work Apple has done to have them look incredibly nice along with text.

Under the hood, a symbol can have up to three different layers – its primary, secondary and tertiary layer. These are important to know about because a symbol's *rendering mode* will use these layers accordingly. Depending on the look you're after, these layers might be rendered the same or differently. For example, perhaps you

want all of them to be the same color, or have a different opacity or maybe varying colors altogether.

Details, Details, Details

Here's an overview of some of the ways SF Symbols can be configured and

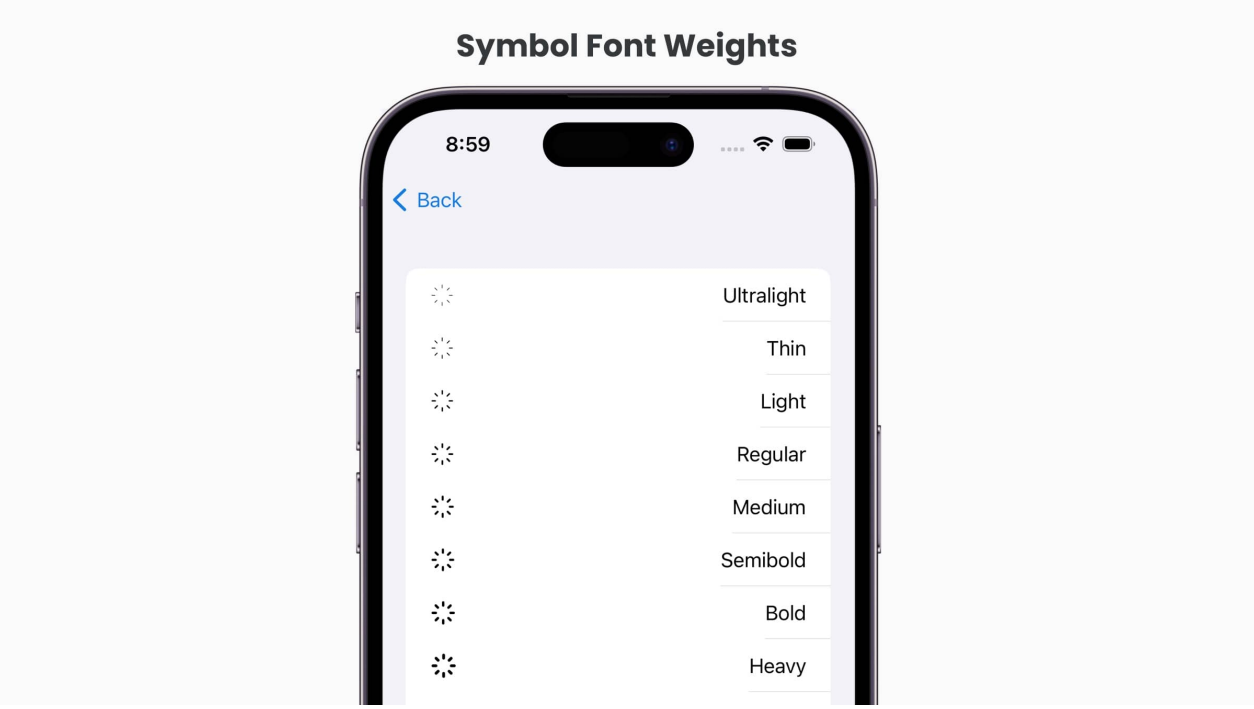
things they can do for you automatically:

Localizations and Accessibility

Many symbols support localization and internationalization scenarios out of the box. For example, if you use `"chevron.backward"`, the direction of the symbol will adapt for right-to-left languages. Plus, there are symbols for Devanagari, Hebrew, and Arabic languages and variants specific to Thai, Chinese, Korean and more locales.

Weights

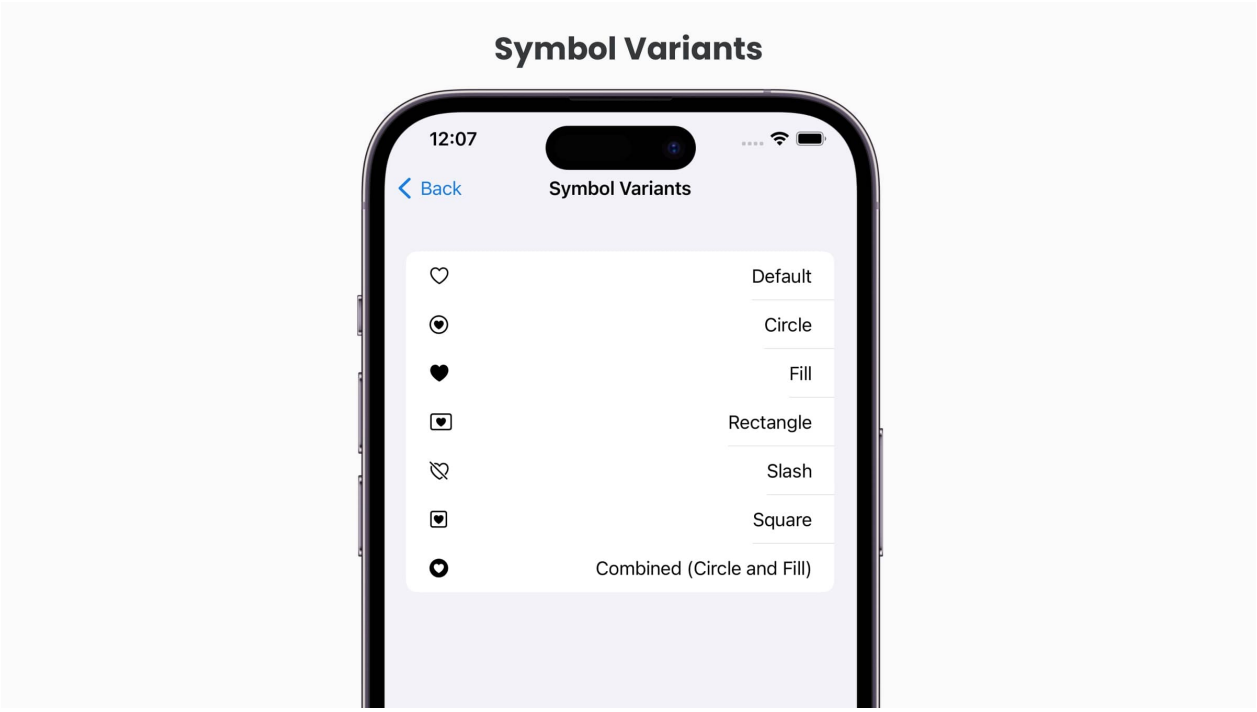
All symbols can be configured with varying weight sizes. Again, this is where thinking of symbol configuration in terms of fonts rather than images can be useful. To tweak their weights (i.e., thickness or stroke border size) you apply a *font weight* to them. As such, you can have a symbol with a tiny weight, such as ultra light, all the way to something very thick by using bold or semi bold font weights as seen here in `SF-SymbolsFig3View.swift`:



Variants

Some symbols support different variants of one another. For example, the heart symbol could have a slash over it, be enclosed in a shape like a circle, filled, not filled and more. Occasionally, the system will prefer one variant over another. In a tab bar, for example, a filled variant is used. The same is true of swipe actions. In other situations, such as navigation bars – an outlined variant is a better option.

Here, in `SFSymbolsFig2View.swift` – you can see all of the different variants for the heart symbol:



Understanding Rendering Modes

The rendering mode options are probably the thing you will tweak most about a symbol. Using them, a symbol can behave a little differently – making them adaptable to nearly every design need. Note that not every symbol supports every rendering style – and if you try to use one that the symbol doesn't support, it'll just fall back to the default option (typically, monochrome).

I'll explain them all here first, and then show you an image to visualize the differences.

Monochrome

This is usually the default rendering mode, which has been around since the beginning of SF Symbols back in iOS 13. It uses one color that's applied consistently to all layers, and it's likely the rendering mode you'll be most familiar with. Basically –

think of this rendering mode as simply using one color, at the same opacity, for the entire glyph.

Hierarchical

This rendering mode relies on one primary color that the system will use at varying opacity levels on the symbol's layers. In a sense, the opacity of the color represents a "hierarchy" of visual importance of the symbol's layer, hence the name.

Palette

This rendering mode uses up to all three layers of a symbol to apply a color and opacity for each. In cases where only two colors are used, the tertiary layer will use the same color as the secondary one.

Multicolor

This is similar to the palette rendering style in that it *can* result in the glyph having multiple colors, but the system will be deciding what the color should be for some components of the glyph. Basically – the "multicolor" component means that you can suggest colors, but the system may apply a different color to a part of the layer which you can't change.

So, for a symbol that represents deleting something – the system might apply a red color (or its localized variant) to part of the symbol no matter which colors you specify.

Automatic or Preferred

The default rendering mode – here, the system will use the context around the symbol to figure out which rendering mode would work best. In many cases, this will end up being monochrome.

To demonstrate how those look, here's how the `list.bullet.clipboard` option appears using each of the rendering modes with varying color options. I've used a heavy font weight for emphasis:



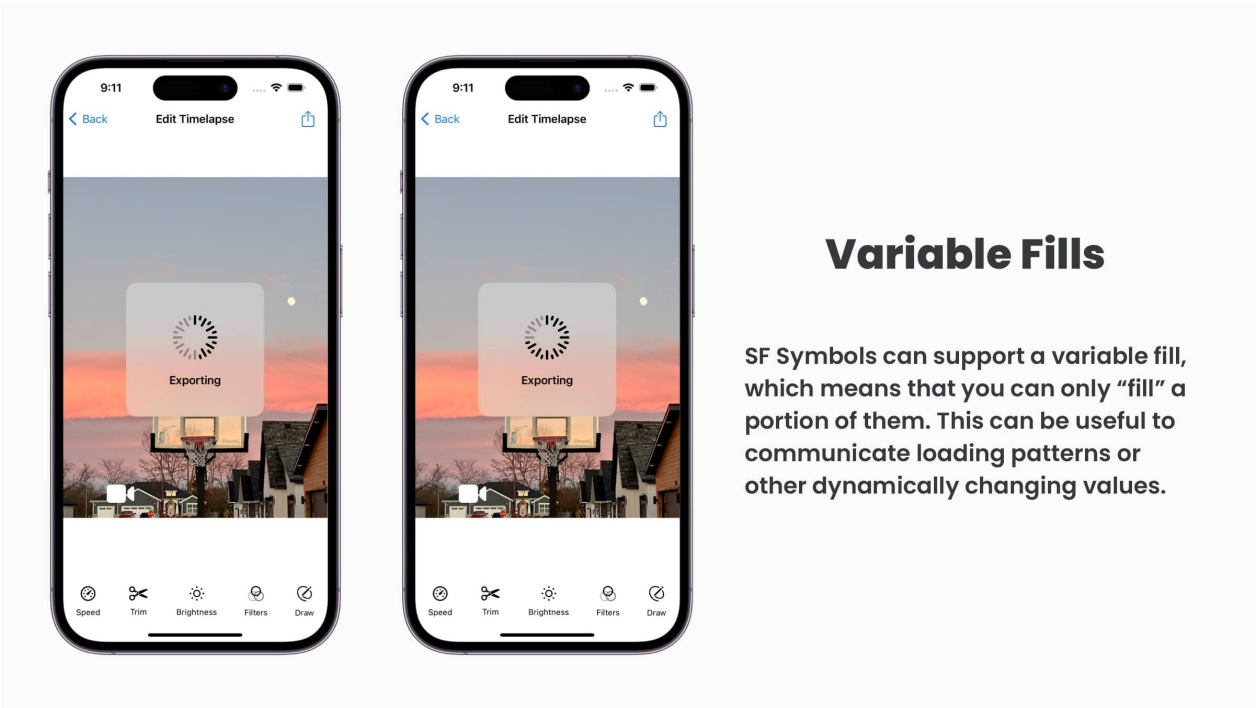
Variable Fills

Some symbols support a variable fill, which means that part of the symbol can be filled while other parts will appear with reduced opacity. This is useful for things like showing the volume, battery health or perhaps Wi-Fi strength.



You can highlight sequences, explain steps or progress using variable fills. The system will apply the fill rate automatically to the correct layers. Keep in mind that not *all* symbols support variable fills. If they don't – you won't be penalized at all for attempting to use the variable fill argument (as we'll see below), it'll just be ignored by the system.

Variable symbol fills can be especially useful for loading patterns, as shown here from the sample project in `SFSymbolsFig1View.swift`:



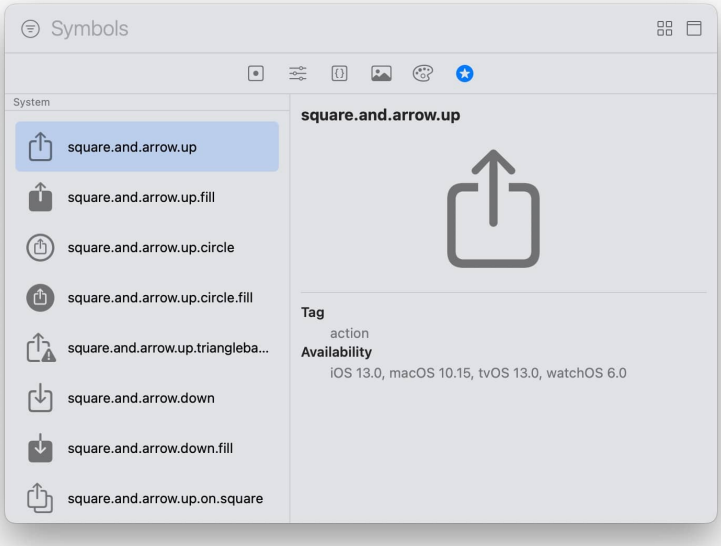
Basic Usage

To use a symbol, initialize an image with the symbol's name:

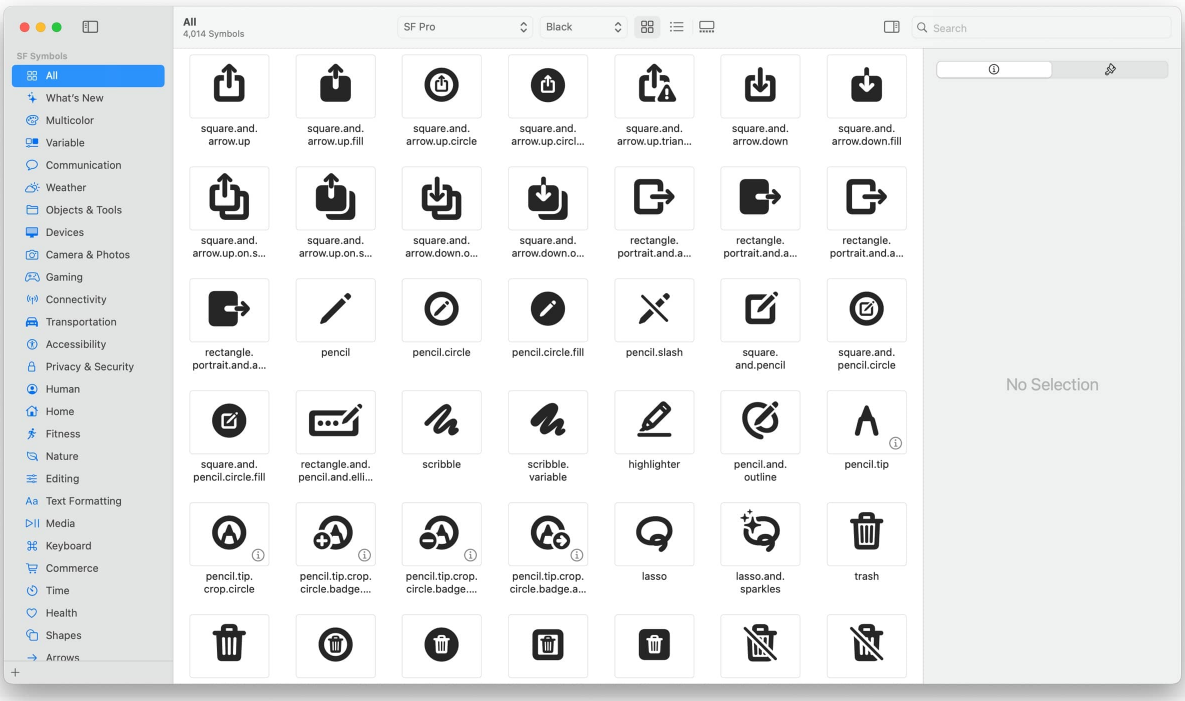
```
// UIKit
UIImage(systemName: "the_symbol_name")

// SwiftUI
Image(systemName: "the_symbol_name")
```

The initializer using `systemName` argument is a forgiving API, fortunately. No crash or exception will occur if the system can't find the symbol from the name you provide, you'll just get a nil image. In most versions of Xcode, you'll get a message in the console letting you know the symbol wasn't found.

You've got a few different ways to find those symbol names, however. In Xcode, you can use `⌘ + ⇧ +. L` to browse them all:



But, the best way to discover and preview SF Symbols is through Apple's SF Symbols app. You can download it directly from Apple's design website, so take note that it's not available through the App Store. Using it, you can see all the rendering modes you can use, what they will look like at runtime, browse the catalog and more:



SF Symbols Recipes

Now, I want to go over some common use cases of SF Symbols and how you would implement each of them in SwiftUI and UIKit. These code snips will take care of most of the ways you will use a symbol.

Using Monochrome Symbols

```
// Xcode -> UIKit -> SFSymbolsFig1ViewController.swift -> // MARK: Monochrome

let timelapseImageView = UIImageView(image: .init(systemName: "timelapse"))

// Monochrome by default. The `tintColor` controls
// The rendering color.
timelapseImageView.tintColor = .purple
```

```
// SwiftUI
Image(systemName: "timelapse")
    .foregroundColor(.purple)
```

Using Hierarchical Symbols

```
// Xcode -> UIKit -> SFSymbolsFig1ViewController.swift -> // MARK:
Hierarchical

let firewallImageView = UIImageView(image: .init(systemName:
"firewall.fill"))

let config = UIImage.SymbolConfiguration(hierarchicalColor: .purple)
firewallImageView.preferredSymbolConfiguration = config

// SwiftUI
 Image(systemName: "firewall.fill")
    .foregroundColor(.purple)
    .symbolRenderingMode(.hierarchical)
```

Using Palette Symbols

```
// Xcode -> UIKit -> SFSymbolsFig1ViewController.swift -> // MARK:
Palette

let starSquareImageView = UIImageView(image: .init(systemName:
"star.square.on.square.fill"))
starSquareImageView.contentMode = .scaleAspectFit

let config = UIImage.SymbolConfiguration(paletteColors:
[.black, .tintColor])
starSquareImageView.preferredSymbolConfiguration = config

// SwiftUI
Image(systemName: "star.square.on.square.fill")
    .symbolRenderingMode(.palette)
    .foregroundStyle(Color.black, .tint)
```

Note that for SwiftUI, you can actually omit the `symbolRenderingMode` modifier. Using a `foregroundStyle` modifier with multiple arguments automatically implies a palette rendering mode.

Using Multicolor Symbols

```
// Xcode -> UIKit -> SFSymbolsFig1ViewController.swift -> // MARK:
Multicolor

let folderBadgeImageView = UIImageView(image: .init(systemName:
"folder.fill.badge.minus"))
folderBadgeImageView.contentMode = .scaleAspectFit

let config = UIImage.SymbolConfiguration.preferringMulticolor()
folderBadgeImageView.preferredSymbolConfiguration = config
folderBadgeImageView.tintColor = .systemTeal

// SwiftUI
Image(systemName: "folder.fill.badge.minus")
    .foregroundColor(.teal)
    .symbolRenderingMode(.multicolor)
```

Combining Configurations

```
// Xcode -> UIKit -> SFSymbolsFig1ViewController.swift -> // MARK:
Combining Configurations

let shareWarningImageView = UIImageView(image: .init(systemName:
"square.and.arrow.up.trianglebadge.exclamationmark"))
shareWarningImageView.contentMode = .scaleAspectFit

// Controls the colors and rendering
let renderingConfig = UIImage.SymbolConfiguration(hierarchicalColor:
.systemTeal)

// Controls the font size
let fontConfig = UIImage.SymbolConfiguration(pointSize: 44,
weight: .heavy)
```

```
// Now, combine them
let combinedConfig = renderingConfig.applying(fontConfig)
shareWarningImageView.preferredSymbolConfiguration = combinedConfig

// SwiftUI
Image(systemName:
"square.and.arrow.up.trianglebadge.exclamationmark")
    .foregroundColor(.teal)
    .font(.system(size: 44,
                  weight: .heavy,
                  design: .default))
    .symbolRenderingMode(.hierarchical)
```

Using an Image Scale

```
// Xcode -> UIKit -> SFSymbolsFig1ViewController.swift -> // MARK:
Image Scale Configurations

let personSmallImageView = UIImageView(image: .init(systemName:
"person.circle"))
personSmallImageView.contentMode = .scaleAspectFit

personSmallImageView.preferredSymbolConfiguration =
UIImage.SymbolConfiguration(textStyle: .body, scale: .small)

// SwiftUI
Image(systemName: "person.circle")
    .imageScale(.large)
```

Variable Fill

```
// Xcode -> UIKit -> SFSymbolsFig1ViewController.swift -> // MARK:
Variable Fill
let badgeViewBaseImageView = UIImageView(image: .init(systemName:
"badge.plus.radiowaves.forward", variableValue: 0.1))

// SwiftUI
```

```
Image(systemName: "badge.plus.radiowaves.forward",
      variableValue: 0.1)
```

Variants

I want to leave a special note about symbol variants. In UIKit, you simply refer to the symbol's variant name to achieve the effect you want. For example, if you want the heart symbol with a circle around it, you'd use this:

```
UIImage(systemName: "heart.circle")
```

In SwiftUI, you can do a few different things. You could either use the name as above, or a modifier:

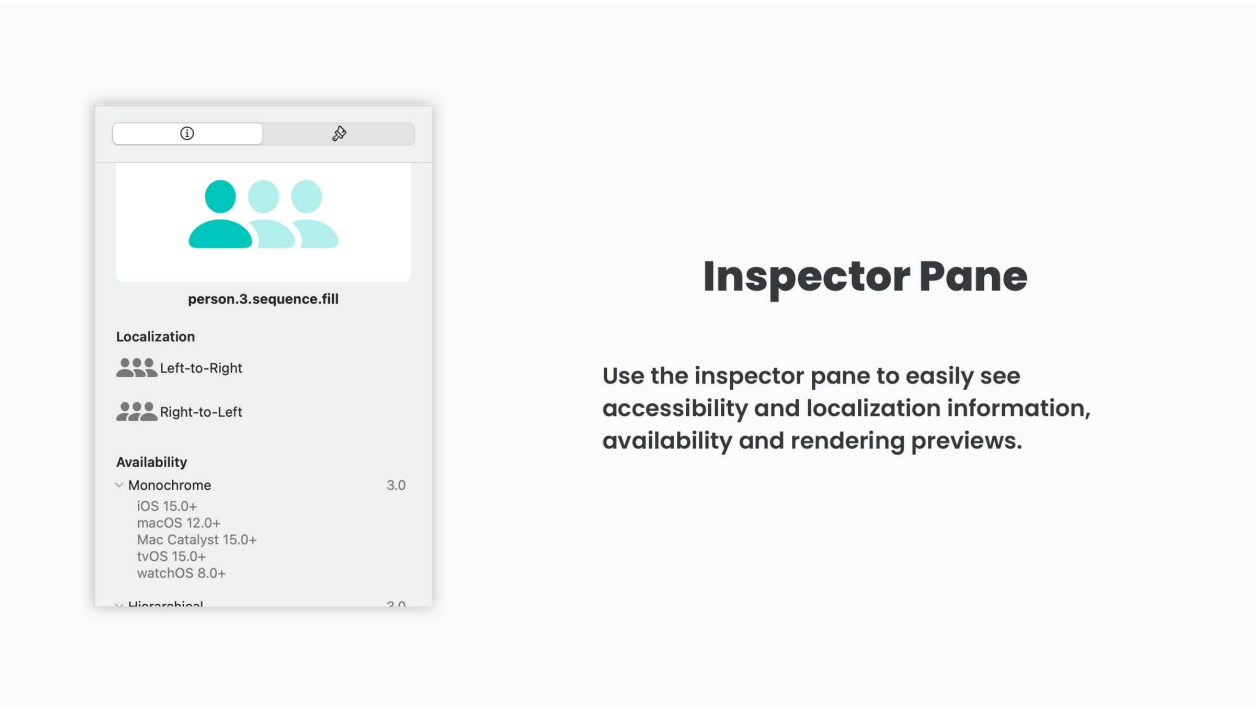
```
// Name
Image(systemName: "heart.circle")

// Modifier
Image(systemName: "heart.circle")
    .symbolVariant(.circle)
```

Tips

Start with the SF Symbols App

The best place to start with SF Symbols is Apple's official app. It's the easiest place to preview styles, see what's available, change colors and more. Plus, there are handy keyboard shortcuts you can use. For example, when you select a symbol and use the ⌘ + ⇧ + v shortcut – you'll copy the symbol's name. Then, it's ready for you to paste right into your code.

On the trailing side of the app, the inspector pane has several bits of useful information as well. You can use it to preview rendering modes, see the symbol's availability in terms of the operating system and more.



Sizing Symbols Correctly

As we've seen in our examples above, you want to size symbols via font sizes. This ensures that they look great next to system text, no matter the layout or scenario. It's a great way to ensure it adheres to accessibility features, too, such as dynamic type.

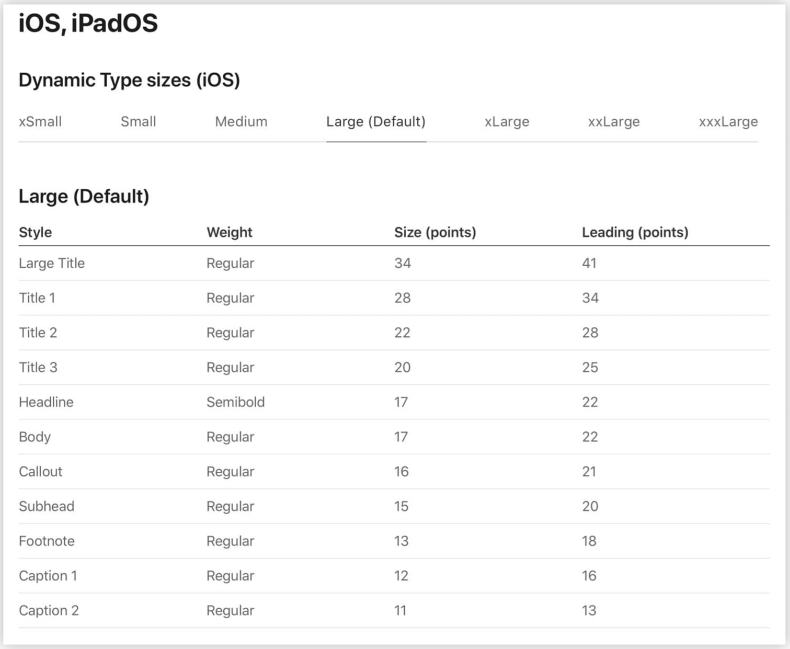
I know, it's weird to conceptually think about it like this at first. You put a symbol in an image, and then size it via…text fonts? But, again, that's how it works. So, try to avoid this:

```
Image(systemName: "rays")
    .frame(height: 40)
```

And instead, opt for something like this:

```
Image(systemName: "rays")
    .font(.headline)
```

If you need to get a good understanding of what font size might match up to a particular point size, Apple's Human Interface Guidelines has a great table under its "Typography" section that can help:



iOS, iPadOS

Dynamic Type sizes (iOS)

xSmall | Small | Medium | **Large (Default)** | xLarge | xxLarge | xxxLarge

Large (Default)

Style	Weight	Size (points)	Leading (points)
Large Title	Regular	34	41
Title 1	Regular	28	34
Title 2	Regular	22	28
Title 3	Regular	20	25
Headline	Semibold	17	22
Body	Regular	17	22
Callout	Regular	16	21
Subhead	Regular	15	20
Footnote	Regular	13	18
Caption 1	Regular	12	16
Caption 2	Regular	11	13

Using Symbols for Loading States

In an earlier example, we saw how we could leverage a symbol's fill state for a loading indicator. I love this technique, and I have an example using the Combine framework to demonstrate how it might work. The same technique applies to either SwiftUI or UIKit.

In short, you'll use a variable to track a loading percentage from 0 to 1. This is very similar to how the `Progress` API in Foundation works already. Then, you update your symbol's `variableValue` argument with the updated progress.

Here's a little recipe for how it might look:

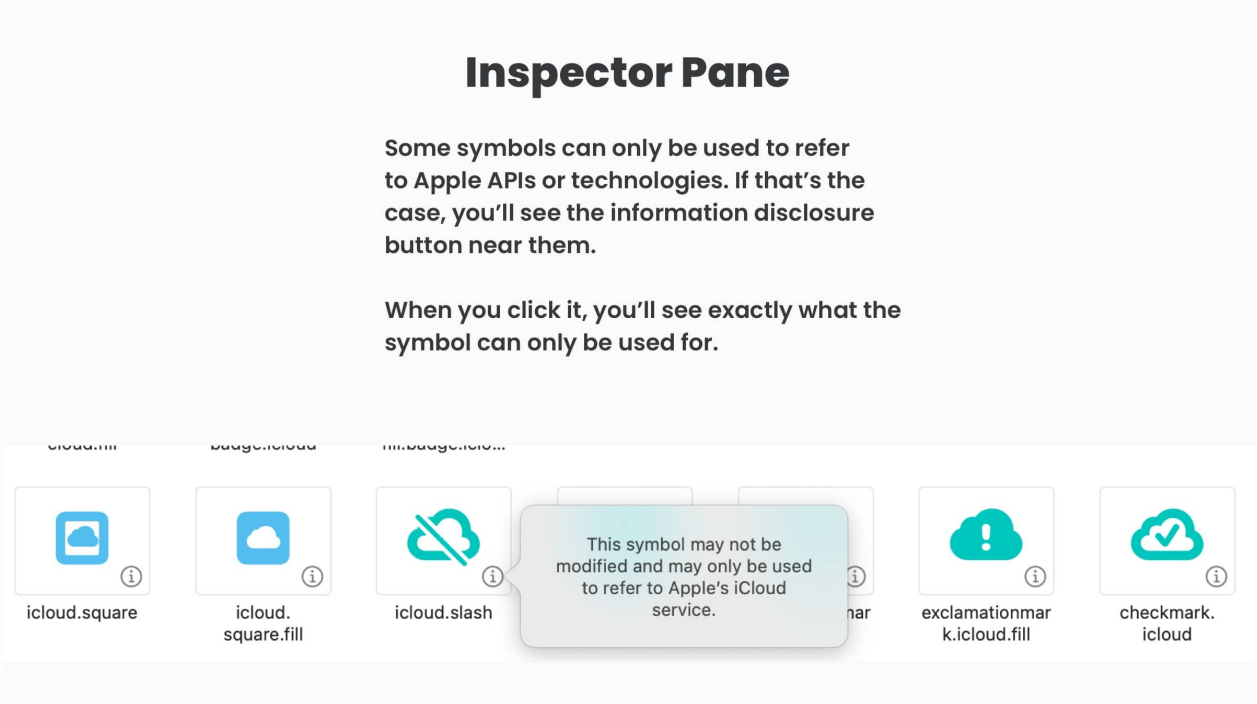
```
// Xcode -> SwiftUI -> SFSymbolsFig1View.swift

// In the `body` property
Image(systemName: exportComplete ?
                  "checkmark.circle.fill" :
                  "timelapse",
      variableValue: loadingProgress)
    .font(.system(size: 60))

// A fake export function
private func exportVideo() {
    Timer.publish(every: 0.05, on: .main, in:
.default)
        .autoconnect()
        .sink { _ in
            guard self.loadingProgress <= 1.0 else {
                withAnimation {
                    exportComplete = true
                    subs.removeAll()
                }
                return
            }

            loadingProgress +=
[0.01..<0.10].randomElement()
.lowerBound
        }.store(in: &subs)
}
```

Here, I'm just using a `Timer` to update progress, but you'd replace that with your app's real loading logic.

Check for Reserved Symbols

Some symbols will have the information disclosure by them. Simply put, this means you can *only* use the symbol for a particular use case as noted by Apple. If you use them in other ways, you'll likely not pass app review.

To see what symbols are reserved, refer to the SF Symbols app.



SwiftUI APIs

SwiftUI has phenomenal support for SF Symbols, and they are a little easier to use than they are in UIKit.

For example, you can easily apply a material style to a symbol:

```
Image(systemName: "badge.plus.radiowaves.forward")
    .font(.largeTitle)
    .foregroundStyle(.thickMaterial)
```

Or, you can use the `Label` view to apply a label and a symbol next to each other – a common use case all throughout iOS, such as headers in a list or any general section:

```
Label("Label View", systemImage: "rays")
```

Plus, creating text with an inline symbol is supported too:

```
Text("Delete Image \(Image(systemName: "trash"))")
```

Even the symbol variant is exposed as an environment value, so you could set several symbol variants all at once:

```
VStack {
    // A bunch of symbols
}
// Apply the circle variant to each of them all at once
.symbolVariant(.circle)
```

This beats having to specify them all manually as you have to do using UIKit, such as "heart.fill", since instead – SwiftUI will apply them up to children views.

Animating SFSymbols

Lately, Apple has powered up SF Symbols to include some nice animations. They can repeat, perform one-shot animations and also customize which layers of the symbol itself participate in the animation. As of now, the animations available are:

Transition & Replacement Effects

- `.appear` – Animates the symbol coming in, either all at once or by layer.
- `.disappear` – Animates the symbol fading out or breaking apart.

- `.replace` — Replaces one symbol with another using a default up/down style.
- `.replace.magic` — Smarter than a basic replace. If the symbols are structurally similar, only the changed parts animate.
- `.automatic` — Lets the system pick what it thinks is best based on context.

Discrete Animations

- `.bounce` — A quick bounce in place. Think scale in and out with a bit of snap.
- `.pulse` — Pulses the symbol's opacity in and out.
- `.rotate` — Rotates the symbol around a chosen anchor.
- `.scale` — Scales the symbol up or down.
- `.wiggle` — Adds a little jitter movement to the symbol.
- `.breathe` — Gently scales the symbol in and out, like a slow inhale and exhale.

Appearance & Style Effects

- `.variableColor` — Animates multicolor symbols over time. You can pick from:
  - `.iterative`
  - `.cumulative`
  - `.reversing`
  - `.dimInactiveLayers`
  - `.hideInactiveLayers`

I have a demo for every single one of these effects, check out `SFSymbols-Fig4View.swift` to see them in action. Keep in mind that some discrete animations are more one-off in nature, so their API is more of a "on/off" situation:

```
Image(systemName: "moon.fill")
    .symbolEffect(.disappear, isActive: disappear)
```

As opposed to others, where a `value` and `options` suffice:

```
Image(systemName: "lungs.fill")
    .symbolEffect(.breathe,
                  options: .repeat(.continuous),
                  value: breathe)
```

Magic Replace

Along the same lines, "magic replace" makes *transitions* from one icon to another much smoother. For example, if a pencil glyph was representing a disabled state, you could use the slash variant. Before, transitioning the two symbols in and out was the best you could do. Now, you can interpolate the slash in and out – resulting in a smooth animation:

```
// Xcode -> SFSymbolsFig4View.swift

Image(systemName: showSmartReplace ? "pencil.slash" : "pencil")
    .contentTransition(.symbolEffect)
```

Using the SF Symbols app on macOS, you can preview this effect, too. Simply toggle on the "Prefer Magic Replace" checkbox when previewing animations.

This won't work with every symbol, however. If two symbols share a similar structure (like slashes or badges), then they can be animate the components of the two that changed. So, a slash could draw itself on or off, or maybe a badge can appear or

disappear. If the symbols *aren't* related, you'll still get the classic down-to-up transition.

Three Key Takeaways

1. SF Symbols are the best way to use iconography in iOS apps.
2. Leverage all the styles, fills and variable colors to achieve distinct, but recognizable, interfaces.
3. SF Symbols respond to accessibility features out of the box, making them ideal for use.

Getting Data In and Out

You want people to own their data. Not only is it a best-practice, but it's also an incredibly user-friendly practice. What if your app shuts down, or they want to export its data somewhere else or anything in between? Maybe they are excited about your app and simply want to share things they've created within it on social media?

Regardless of the reason, a great user experience around data can be crafted. Overall, it's about ownership. When people use your app, they begin to trust it. And you can strengthen that trust by offering a way to manipulate the data they've entered into your app in a way they see fit.

In the age of automation, Siri Shortcuts and similar tooling, data can live on in ways that it couldn't before (at least, not as easily as it could before). We all want to craft successful apps, but sometimes – they come to an end. I've experienced this myself, and when the time came, the first question I had from folks was "How do I get my data out?"

Whether it's future proofing data, or simply offering choice – you can't go wrong with a wonderful user experience around data getting imported or exported. So, let's look at some ways iOS can help us out.

We'll start by looking at a foundational way to think about your data models in code, and then expand on that with APIs that support them. Most importantly, we'll look at some existing experiences other apps have crafted to spur some ideas on how you can design similar functionality.

The end game here is to offer rich data I/O (input and output) avenues for your app, its data and any incoming data.

Making Your Data Exportable

Let's start by looking at some foundational concepts that facilitate data I/O at the API level.

When your app wants to send or receive data, there are essentially two things that your models need to do. First, it has to have a mechanism in place to convert itself to, and from, binary data. Secondly, it'll also need to tell other apps or processes what that *kind* of data it actually is.

To achieve those two things, iOS offers a variety of APIs. To see how they work in practice, we'll be working with a simple model which represents a blog post for some fictional writing app:

```
struct Post {
    let text: String
    let id: String
    let publishedAt: Date
    let lastModified: Date
}
```

With that in mind, let's start with the latter part. That is, describing what kind of data you're sending or receiving.

Content Types

Content types are an Apple-specific technology to help marshal data from apps, ship it over to other ones, to other processes within Apple's ecosystem and more. For a practical example, think about how you might've handled media on macOS. When you drag an image from your desktop to the Photos app – it's looking at the dragged item's content type to determine whether it's something it could import.

So, if the item is a movie, image or a GIF – then Photos can handle it. But, if it's something else, say a text document, then it'll reject it. Content types provide an understandable interface for both programmers, and their applications, to make these kinds of determinations.

From an API standpoint, this is driven via the Uniform Type Identifiers framework, or commonly referred to as "UT Types". A UT Type works in a hierarchal manner, meaning it can represent something very general (like a picture) down to something extremely specific (like a photo, but only a .png type).

The good news is that if you're sharing something "basic", for lack of a better term, such as text, images, or media – then iOS will do most of the work to get that exported or imported effortlessly. But, if you want to support wholesale sharing of something custom – say our `Post` model above, you can create custom types for that, too.

Codable

`Codable` is a protocol which lends itself well to data I/O. This, and the following discussion over `Transferable`, cover the second part of our two requirements for data I/O. That is, packing up your models into data to shuttle back and forth.

In the case of `Codable`, If you look at its definition, you'll see that it's really a type alias that represents something which conforms to both `Encodable` and `Decodable`

protocols. This protocol allows you to perform critical tasks with data. If your type conforms to it, you could…

- Save data to disk.
- Create `.json` to use for network requests, exporting or saving locally.
- Or, decode incoming `json` to map back to our model.
- Use with other data I/O methods, as we'll see below.

When it comes to working with models in Swift, I generally start by making them conform to `Codable` right from the start. And, in the case of our `Post` model — you'll notice that we don't even have to do anything, Swift can do it all for us.

```
struct Post: Codable {
    let text: String
    let id: String
    let publishedAt: Date
    let lastModified: Date
}
```

That's because all the properties themselves already support `Codable`, so the rest of the boilerplate details are handled for us. In fact, most types do. Things like strings, images, URL, and attributed string types all conform to it without any extra work.

If it didn't work out of the box, for example — if we had custom types as properties which couldn't implement `Codable` for free, we'd have to write a bit more to achieve conformance. This still isn't much of an issue, as it entails implementing our `init(from: Decoder)` and `init(to: Encoder)` manually.

To be clear, working with `Codable` can be a large subject. It's true that for simple types, it just works. Other times, you'll need to do things like create an `enum` type conforming to `CodingKeys` to match up network responses with you model data, for example:

```
// JSON Response
{
    "data": {
        "title" : "A Nested Response"
    }
}
```

Here, if you have a model that only cares about `title`, you'd have to either create an intermediary model to decode from `data` and then down to `title` or handle it with custom encoding and decoding. Further, if you had something like this…

```
// JSON Response
{
    "item_title" : "A Nested Response"
}
```

…well, now you're not dealing with nested data – but you've got something keyed to a name that probably doesn't match up with your model's property. Here, you'd need to use a custom `CodingKeys` enum:

```
struct Example: Codable {
    let title: String

    enum CodingKeys: String, CodingKey {
        case title = "item_title"
    }
}
```

Either that, or live with a property named `item_title`, which isn't preferred.

Transferable

`Transferable` is a protocol introduced in iOS 16 that allows you to support serializing and deserializing your data models. In short, it's built specifically for sharing and data transfer. Unsurprisingly, it's also billed as a Swift first framework and it's built as a declarative API.

When you adopt `Transferable` in your models, you'll get several ways to share your data, such as:

- Copy and paste
- Drag and Drop
- The "Share Sheet"
- Saving to disk

How does that look with our `Post` model, then?

When conforming to `Transferable`, you've got one thing to handle — and that's the `transferRepresentation` static property. Within this, you can provide different representations of your data. Namely,

- `CodeableRepresentation`: Under the hood, the system will use your type's `Codable` conformance to shuttle data. By default, it'll create JSON — but you can also implement your encoding and decoding logic if you need.
- `DataRepresentation`: This means you'll be converting the type to `Data`, either going in or out. This is great for simply saving data blobs, or using that data to get it into a more concrete format, such as a .csv file.
- `FileRepresentation`: This shares your type as a file representation written to disk. This can be especially useful when the item being shared is large, for example — a video. This avoids loading a large asset or model into memory, and instead you pass a provided URL for the consumer to work with.
- `ProxyRepresentation`: This is handy to have *parts* of your model be represented by something that the system already supports. For example — text. The system knows a lot about how to perform data I/O with string

types. So, if we wanted to simply make our `Post` type have its `text` property easily used with things like copy and paste, we could use a proxy representation. This essentially says "This `text` property is just a plain `String`, so you can use it around iOS' features that use text for data I/O.", and that would look like this:

```
extension Post: Transferable {
  static var transferRepresentation: some
TransferRepresentation {
      ProxyRepresentation(exporting: \.text)
  }
}
```

No matter the route you take, in each of them you'll need to provide a content type. This will be either a custom content type you've specifically made for the model, or something that fits it best. Since the API is declarative, you can create a hierarchy of preferences when working with `Transferable`, which is how UT Types inherently work anyways.

Consider if you created a custom content type for our `Post` model. What this hierarchy design means is that if you could share that type foremost (a custom `Post` UT type), and if the app accepts – great! They get the full `Post` type.

However, If they *don't*, you can go down the line of hierarchy you provided and supply the "next best thing" you've made available. So, maybe you can share it as a data blob next, or perhaps as a .csv or even simply text as we've done. In short, the app will use the first UT type it supports.

Below, here's what offering a few different UT Types would look like for our `Post` model. We provide our custom UT Type first, but if that isn't recognized, we can fall back to supplying text. This setup is flexible, because you can start with very generalized data types, or very specific depending on your use case:

```
extension Post: Transferable {
    static var transferRepresentation: some TransferRepresentation {
        CodableRepresentation(contentType: .blogPost)
        CodableRepresentation(contentType: .text)
    }
}
```

So, with these core concepts in mind – understanding what uniform type identifiers are, how `Codable` works and a little working knowledge around `Transferable` – you're now opened up to numerous ways to import and export all sorts of data.

Below, we'll spend some time looking more critically at the user experience side of things. Where should you put data I/O features? Should their copy be generic or specific? What do you do if you offer several types of imports or exports? All valid questions.

In the Tips section, we'll also look at some of the ways you open things up for people's data housed in your app from an API level.

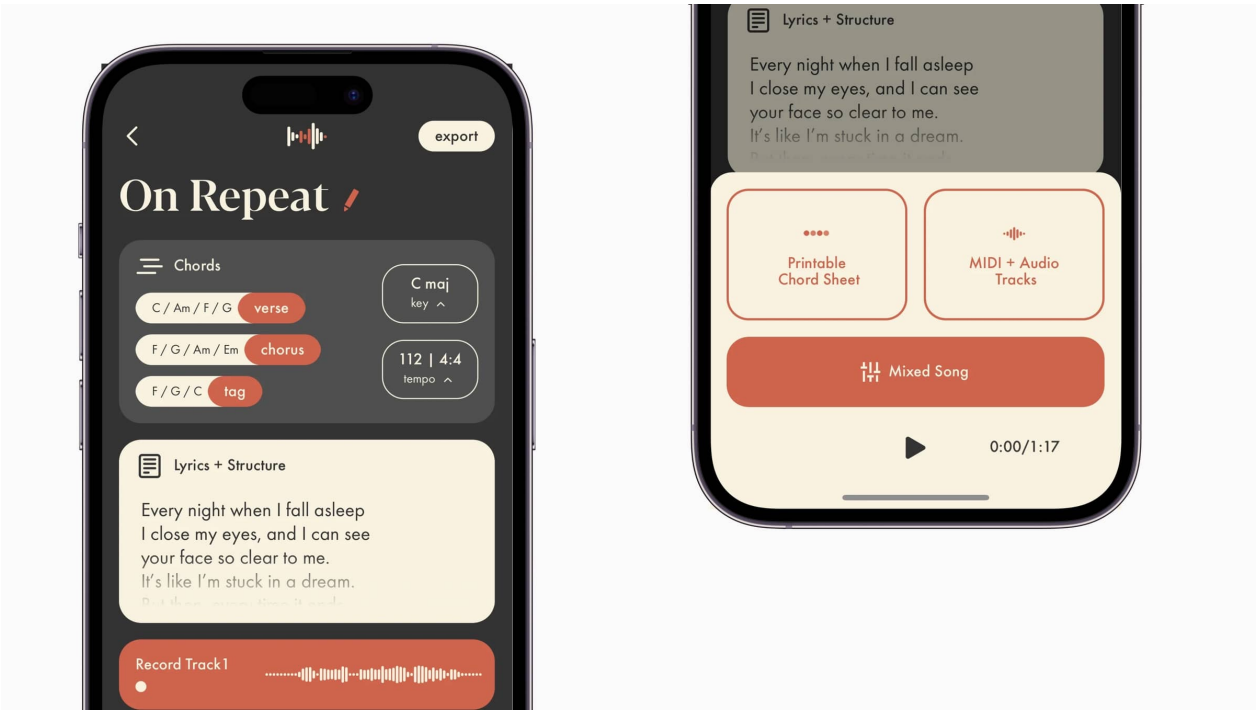
Tips

Make I/O Obvious

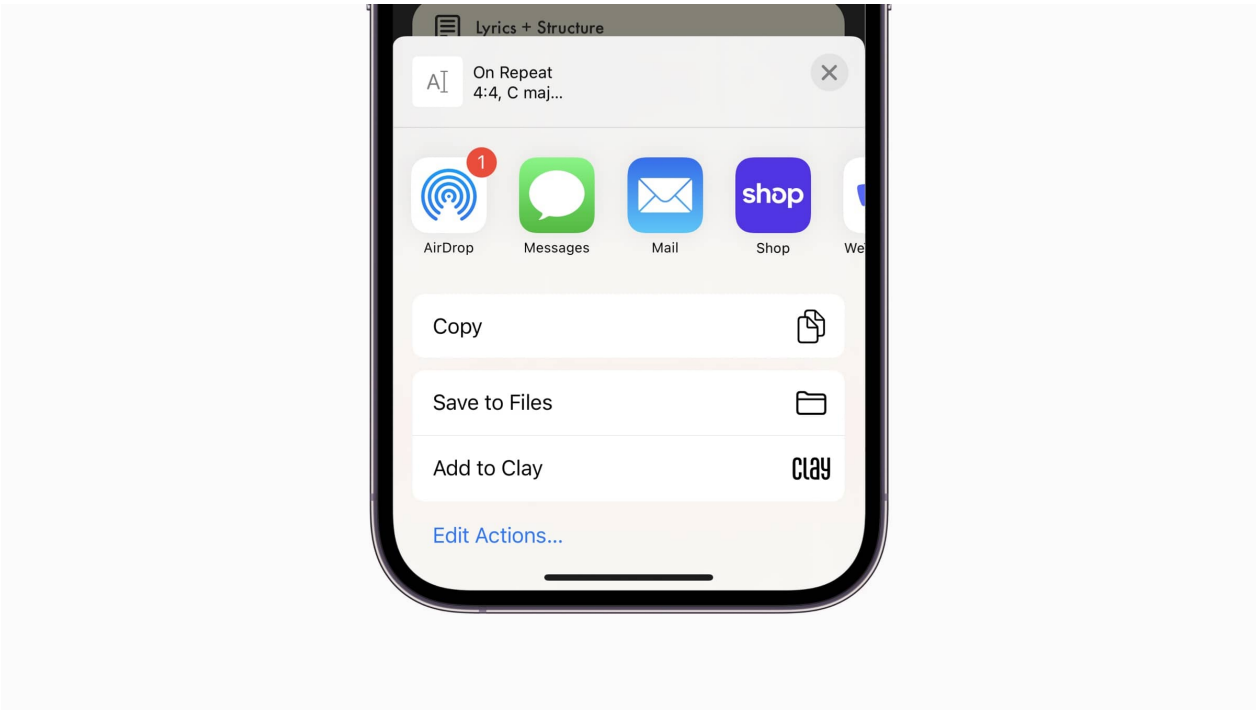
When you offer import and export options, make them available in obvious places. Generally speaking, some sort of settings view is probably where most people will look first. Or, if the user is in the middle of some context where they are actively editing, creating or otherwise changing data – maybe it makes sense to offer it within the same view.

As with anything else in software engineering or design, it depends. Let's take a look at a few examples.

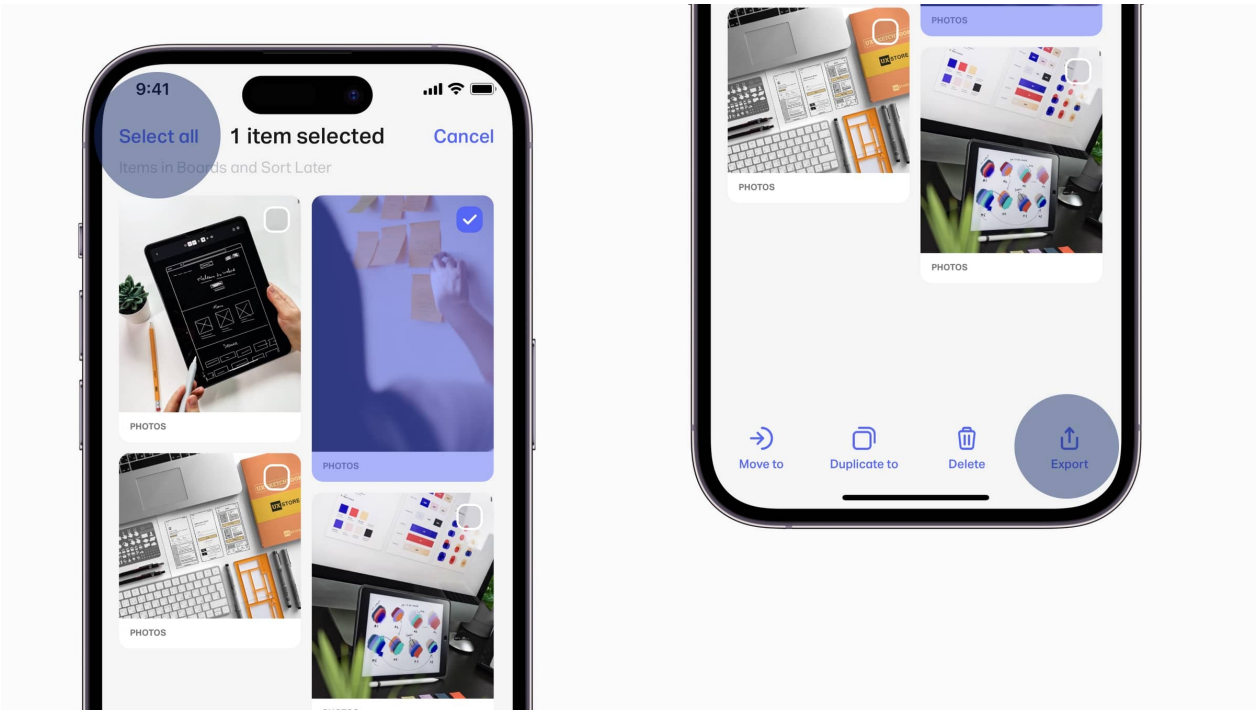
In the app Demo, used for musical chord generation – there is a clear and obvious export button at the top:



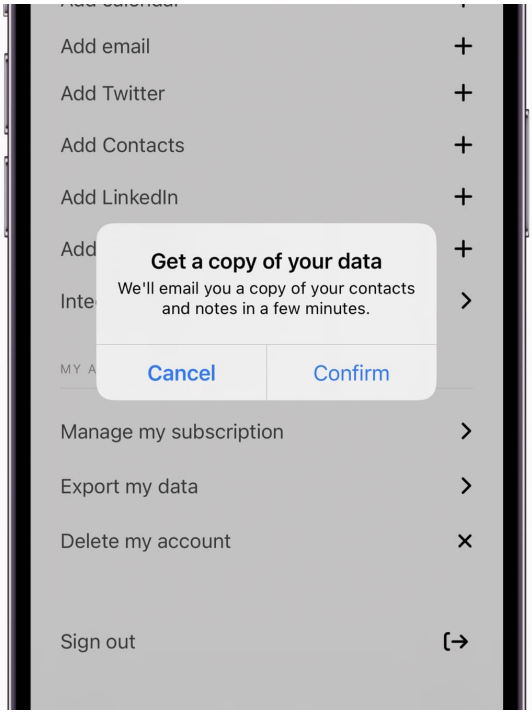
When you tap it, you'll get the familiar share sheet:



In other cases, it may make sense to make exporting options contextual. That is, unless people are selecting things, editing them or anything else within that realm — the export option will hide and show. In the app Collect, this is the pattern they follow. Once someone selects something, the option to export it is revealed:



Some data may take quite a bit of time to gather for exporting flows. If you're developing a social media app, or anything where someone's data may stretch back several years — it probably makes sense to create that server-side. In these cases, the export can't be immediate, so take care to provide some sort of notice that the export will finish later and note how it will be delivered to them. Here's an example from Clay:



Provide Progress Where Possible

Sometimes, client side data I/O can be an expensive operation. If you're importing or exporting data that may take a bit and it's not happening asynchronously – be sure to provide some form of progress or status updates.

Take the popular vector drawing tool ProCreate – when importing media, you get a nice modal that shows how far along it is:

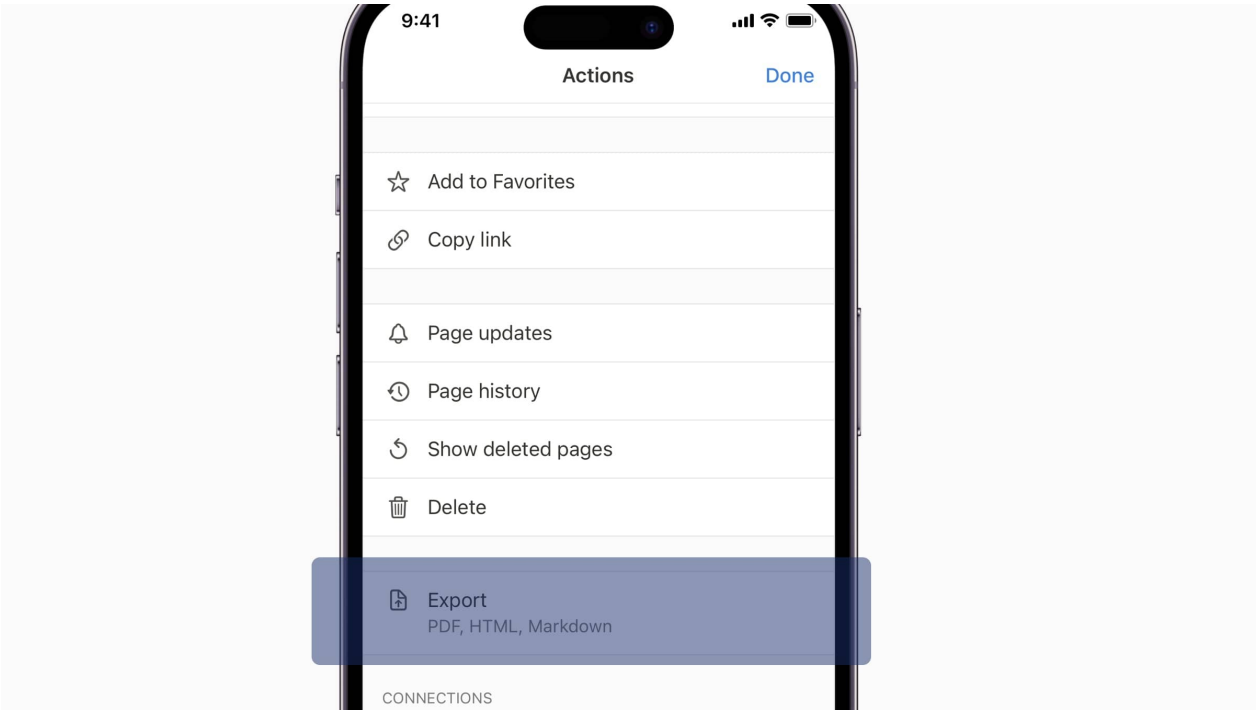


Whether it's a simple progress indicator, copy or something else – informing people of where things are at during long running tasks is critical. For more information on this topic, please visit the "Loading States" chapter.

Provide Context Over Supported Formats

There are times when you might have multiple forms of data formats that you can import or export. For example, apps similar to Notion, which let people create complex charts, diagrams, custom dashboards and more – data exporting could result in multiple types of file formats.

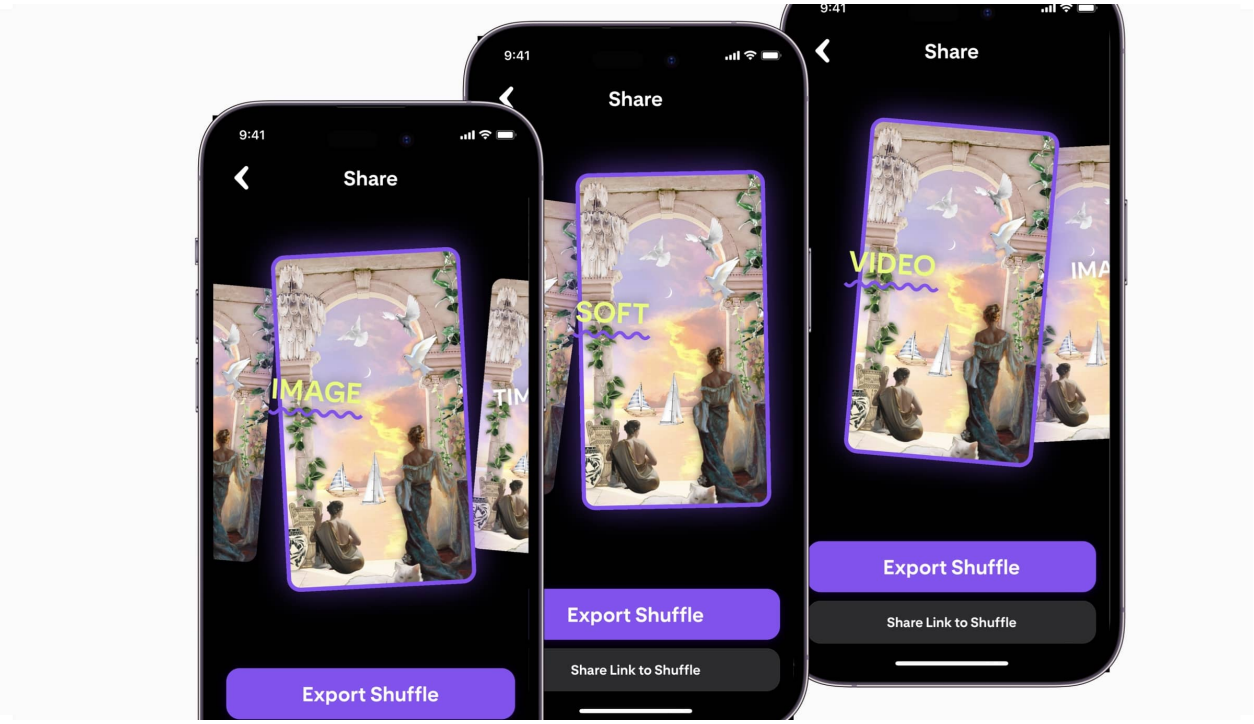
In those cases, I think it helps to remember to do two things well. The first is to make it clear and obvious that multiple export options are available. The second is to make choosing the one which you want to use, well, again – clear and obvious. In Notion, they hint at their file format options upfront, which, I think, is perfect:



Leverage your App's Voice

Even though data I/O will probably run through the system share sheet nine times out of ten (which I think it *should*, by the way – more on that below), it doesn't mean you can't use the opportunity to leverage your app's voice. A wonderful example of this is the app Shuffle, from Pinterest.

It still uses its bold colors and font choices for exporting data, the same as the rest of its design. Even though data I/O can feel very much like a utilitarian task, there is still room to spice it up to match your app's motif:



Rethink Custom Share Sheets

I don't mean to pick on certain apps here, but I have never found a suitable reason to recreate, or otherwise redesign, the system share sheet. They make exporting and sharing introduce more friction, and unless there is an option to specialize exporting media to something like a social network via deep links, I can’t think of a reason to do it.

There are a lot of organizational decisions that are made which may lead to this, but try to stand your ground and give some constructive feedback on why recreating this experience may not be necessary, such as:

- iOS's users expect the system share sheet when they tap the standard share icon.
- It’s “wasted” effort, the system provides rich sharing capabilities.

- It introduces friction, people have to familiarize themselves with your custom sharing experience instead of the one they already know.
- And if the custom share sheet is being pushed for other custom capabilities, then the task is probably misunderstood at a technical level by someone else driving the effort. The system share sheet offers up several APIs routes to do almost anything you need related to sharing data.

APIs for Data I/O

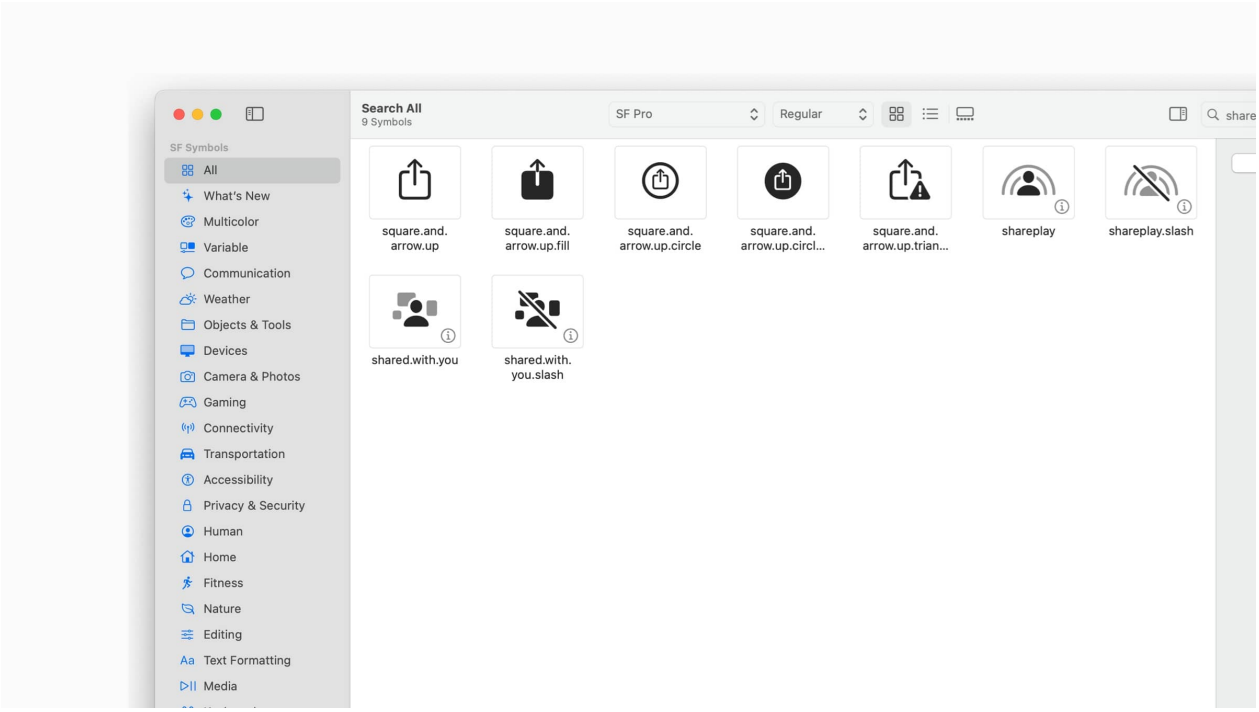
To close things out, here are some APIs you can be thinking about when it comes to data I/O and your app.

Copy and Paste

By leveraging `UIPasteBoard` or the more nascent `UIPasteButton`, you can offer copy and paste functionality. Even further, you can add items to the system's editing menu to copy and paste things too. The latter is a wonderful option to avoid the constant alert that iOS will show if you're querying the system's pasteboard at launch or other similar lifecycle points.

The Share Sheet

The system share sheet is ubiquitous, understood and trusted throughout iOS. It's one of the best ways to export data in iOS. When using it, take care to use understood iconography to present it. The classic "box with an up arrow" is what you're after, and in SF Symbols that glyph is the name "square.and.arrow.up" — though it has several variants you can use:



In UIKit, most of this is run through the `UIActivityItemViewController`. To use it, you supply it with `UIActivity` items which tells the system what you're sharing. If you want all the power and functionality of it outside of UIKit, you can also wrap it inside a representable as I've done here:

```
// Xcode -> SwiftUI -> ShareSheetView.swift

struct ShareSheetView: UIViewControllerRepresentable {
    typealias ShareHandler = (_ activityType:
UIActivity.ActivityType?,
                              _ completed: Bool,
                              _ returnedItems: [Any]?,
                              _ error: Error?) -> Void

    let activityItems: [Any]
    let applicationActivities: [UIActivity]?
    let excludedActivityTypes: [UIActivity.ActivityType]? = nil
    let onShare: ShareHandler?
```

```
    // MARK: Initializer

    init(activityItems: [Any], applicationActivities: [UIActivity]?
= nil, callback: ShareHandler? = nil) {
        self.activityItems = activityItems
        self.applicationActivities = applicationActivities
        self.onShare = callback
    }

    // MARK: Representable

    func makeUIViewController(context: Context) ->
UIActivityViewController {
        let controller = UIActivityViewController(activityItems:
activityItems, applicationActivities: applicationActivities)
        controller.excludedActivityTypes = excludedActivityTypes
        controller.completionWithItemsHandler = onShare

        return controller
    }

    func updateUIViewController(_ uiViewController:
UIActivityViewController, context: Context) {
        // noOp()
    }
}
```

In SwiftUI, there is a more direct path you can use via the `ShareLink` view. Using it makes it easy to trigger the system share sheet. For example, if I wanted to share the website for this book series, I could pop it right into a `ShareLink` and it'll take care of the rest for me without the need for any wrapped UIKit code:

```
// Xcode -> SwiftUI -> DataIOFig1View.swift
struct DataIOFig1View: View {
    let url = URL(string: "https://www.bestinclassiosapp.com")!

    var body: some View {
```

```
        VStack {
            Text("A Best-in-Class iOS App")
                .font(.headline)
                .padding()
            ShareLink("Share Book Series" as String,
                      item: url,
                      subject: Text("View Details"),
                      message: Text("A five book series over iOS."))
        }
    }
}
```

Out of the Box Methods

Remember, data I/O can take many forms throughout the system. Though we've covered the highest impact areas here, remember that there are many other ways to share data. Here are a few that I can think of:

- **AirPrint**: Apple has easy API support for provide printing operations.
- **Specialized Data Formats**: Perhaps you want to create a .pdf, or .csv. Apple has APIs for both of those in the form of `PDFKit` and the `Tabular-Data` framework.
- **AirDrop**: You can even support AirDrop with your data using some APIs presented earlier to serialize, and deserialize, data.
- **Drag and Drop**: People expect to drag and drop items to and from your app, so that's another wonderful API to enrich your data I/O offerings.
- **iCloud**: Though this is a syncing framework, it still acts as a way to protect against data loss and ensuring it's available on all the devices someone may have.

Three Key Takeaways

1. People should own their data, so beyond data export or import being convenient — it's also the right thing to do.
2. The system has several ways for you to export data, so pick the one that makes the most sense in the context of your app.
3. Once you can express your data according to the `Codable` protocol, you're able to express your data however you like.

Controller Considerations

The `UIViewController` is the eternal building block of iOS applications. Around since the dawn of Cocoa Touch, the view controller, representing the "controller" part of the model-view-controller pattern that UIKit is designed for, is responsible for acting as an interface between views and the models they show information about.

In reality, many iOS apps end up with *another* flavor of M.V.C. — that is, the "Massive View Controller", which is symptomatic of shoving everything your app does into one place. Regardless of how you structure your apps, if you're using UIKit — you're absolutely using view controllers. Even if you use SwiftUI, it's likely you may end up wrapping one or two of them.

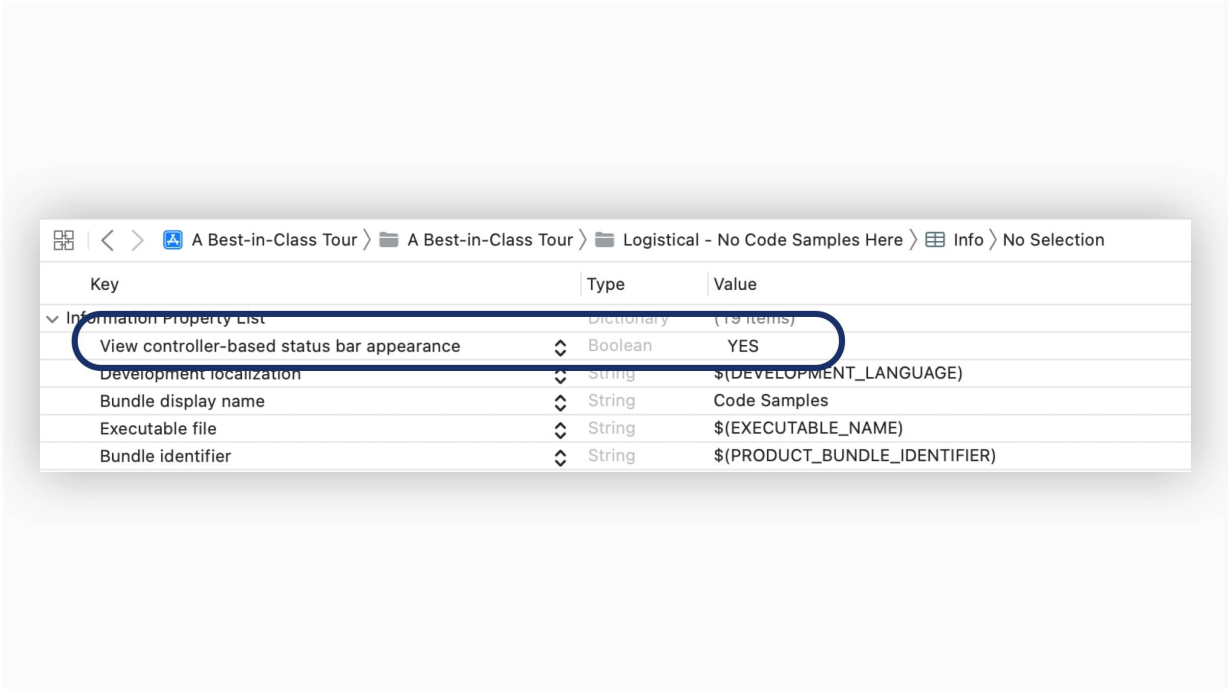
To that end, this chapter is all about the ways you can tweak system behaviors and chrome within your view controllers. There are several options UIKit exposes to you from handling status bars, cursors, the home indicator and more. While this chapter

isn't particularly novel, I'd say it's required reading if you're building any UIKit app. These APIs are trivial to use, but they aren't all that obvious to find if you've not been around the UIKit block for some time.

Let's get started.

The General Flow

To change how a view controller shows system chrome, such as the status bar or home indicator, the setup is generally the same. But, let's get the exception to the rule out of the way. If you're updating the status bar, you'll have to ensure your Xcode project is set up for view controller-based updates. In your project's `info.plist`, add the "View controller-based status bar appearance" key and set its value to `YES`.



Moving on, there are several overrides you can manage to change things like:

1. The appearance on the status bar.
2. Hiding or showing the status bar.
3. The animation of hiding and showing the status bar.
4. Hiding or showing the home indicator.
5. Locking the pointer.
6. Whether the screen edges defers system gestures.

When you want any of them updated – you can call what is mostly named a "prefers(API)update" function. By keeping a local variable around that represents the change you want to see, you can then easily update things on the fly. I know that's a lot of words – but it's not complicated in practice.

Here's what it might look like if we wanted to hide or show the status bar:

```
// Xcode -> UIKit ->
ControllerConsiderationsFig1ViewController.swift

// Abbreviated. See file for more examples.
class ControllerConsiderationsFig1ViewController:
BaseSampleViewController {
    // 1
    private var showStatusBar: Bool = true

    override func viewDidLoad() {
        super.viewDidLoad()

        // 2
        showStatusBar.toggle()
        setNeedsStatusBarAppearanceUpdate()
    }

    // MARK: Overrides

    // 3
    override var prefersStatusBarHidden: Bool {
```

```
        return !showStatusBar
    }
}
```

Going step by step:

1. Our `showStatusBar` variable will track whether we want it showing.
2. By changing that variable, and then invoking `setNeedsStatusBarAppearanceUpdate()` we tell UIKit to update the status bar.
3. We've overridden `prefersStatusBarHidden`, and it'll now be invoked by UIKit and return our preference. The status bar is either hidden or shown at this point.

This is generally what the flow looks like for any system chrome you may want to update.

However, that's not the *full* story.

As with many things in user interface frameworks, it gets a little more complicated. Basically, you'll need to decide *which* controller should be responsible for updating the chrome you're trying to change. This is where most developers can get confused. The API is, more or less, designed to have the root controller be responsible for these things — but many times, that's not what you want. You probably want the presented controller to be in control.

In fact, if you put that code written above in a brand-new file in the sample project, it wouldn't work at all. Why? Because the sample project is built around a root navigation controller that pushes the code samples on and off its navigation stack:

```
func scene(_ scene: UIScene, willConnectTo session: UISceneSession,
options connectionOptions: UIScene.ConnectionOptions) {
    guard let windowScene = (scene as? UIWindowScene) else
{ return }
```

```
    let rootController = GlossaryViewController()
    let rootNavController = UINavController(rootViewController:
rootController)
    window = UIWindow(windowScene: windowScene)

    // Our root navigaiton controller
    window?.rootViewController = rootNavController
    window?.makeKeyAndVisible()
}
```

So, *that* root navigation controller has to tell UIKit that it's okay for *other* presented controllers to manage things.

If you look at our `SceneDelegate.swift` file, that's why you'll see the custom navigation controller at the top:

```
class CustomNavController: UINavigationController {
    override var childForStatusBarStyle: UIViewController? {
        return self.viewControllers.last
    }

    override var childForStatusBarHidden: UIViewController? {
        return self.viewControllers.last
    }
}
```

Much like you create an override at the view controller lever for each system chrome you want to change, often times you must also override things at the root controller level to ensure those preferences are respected. In `ControllerConsiderations-Fig1ViewController.swift` you'll notice we change two things:

1. The status bar appearance (i.e., light or dark)
2. And, whether it's showing.

So, we also override `childForStatusBarStyle` and `childForStatusBarHidden` in this custom navigation controller so that UIKit will use them. Otherwise, they would be ignored.

When we try to change things like the status bar appearance, UIKit will first look at the navigation controller since it's the root controller of the app. In our case, since we've overridden those "childFor(API Name)" functions, UIKit will say "Oh, this controller wants a different controller to be responsible for this – what controller does it want to use?"

Here, we've indicated we want the last controller of its `viewControllers` property to be used – which for a navigation controller will be the one that's on top of the navigation stack. Again, here's that same code from above:

```
class CustomNavController: UINavigationController {
    override var childForStatusBarStyle: UIViewController? {
        return self.viewControllers.last
    }

    override var childForStatusBarHidden: UIViewController? {
        return self.viewControllers.last
    }
}
```

Given this, UIKit then looks at our `prefersStatusBarHidden` implementation in `ControllerConsiderationsFig1ViewController.swift` to get the correct preference when it's presented.

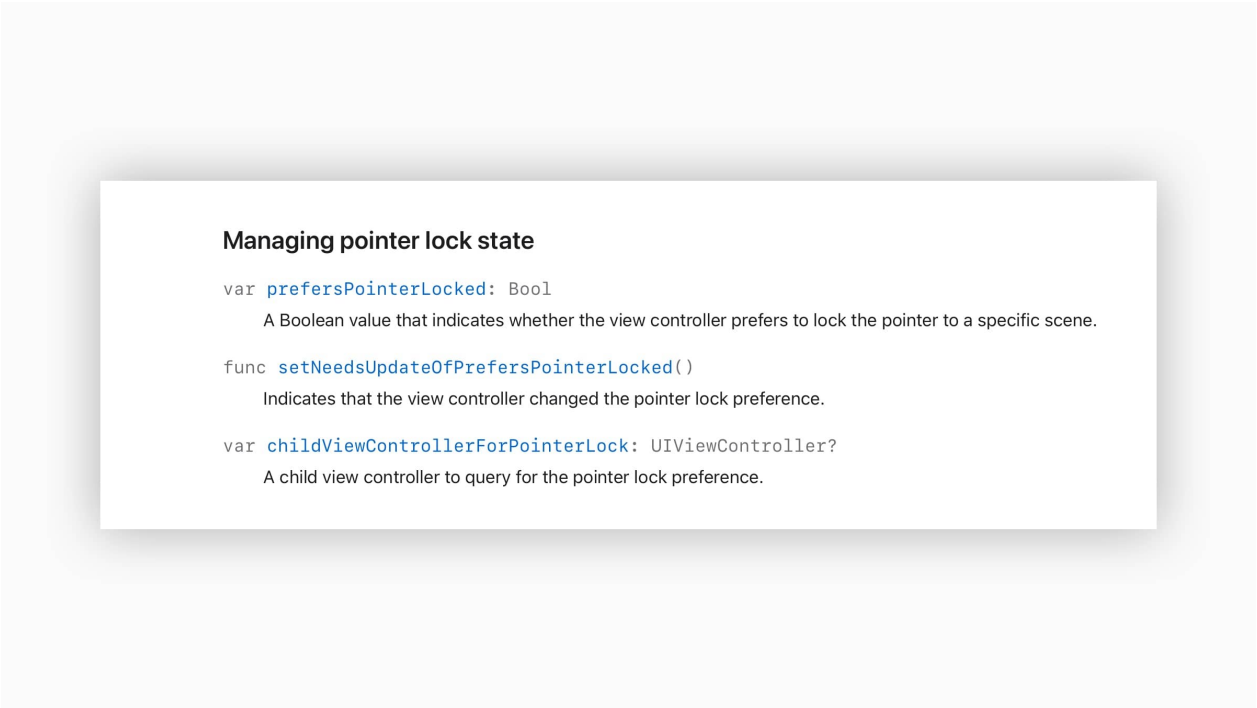
To summarize, most UIKit projects have a navigational root controller that acts as a container for other controllers it manages. Think of tab bars, or navigation controllers. In these cases, you have to let that *root* controller tell UIKit that it's fine with other controls managing system chrome if that's what you want.

To figure out how to implement these APIs, here are some questions you can ask yourself:

1. Is your root controller a navigation controller, tab bar controller or something that will manage and present other controllers — but you want other controls to be able to change system chrome?
2. If your controller being presented modally?
3. Is your controller the root controller?

If the answer to question one is yes, you'll need to override properties to tell the system something else is to manage appearance updates. That's what we've done here. So, that means an override in the root controller to pass off your preferences to some other controller, and an override in the actual controller you want to manage them.

This technique is so common that if you look at the documentation for tweaking these system chrome preferences — you'll see that three-pronged approach. Basically, "here's the thing to change", "here's the function to say you want it changed", and "optionally, here's an override to say another controller can change it":



If question two is where you answered yes, if you're changing things like the status bar, then you'll also need to override `modalPresentationCapturesStatusBarAppearance`.

Finally, the easy one — if you're answering yes to question three, then simply setting things at the view controller level is enough. You won't have to edit any other container controller to reflect your changes.

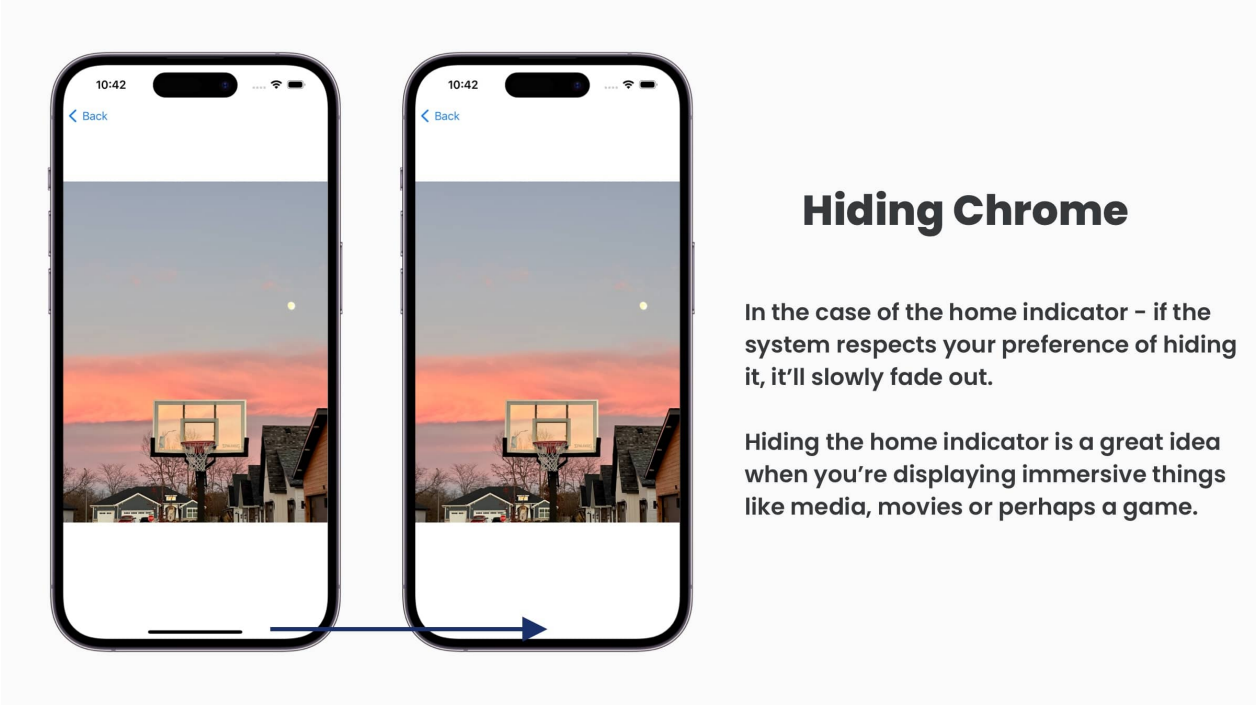
Tips

Context is King

The real question to this whole chapter, from a pure user experience perspective, is probably this: Why would you change any of these things? The answer is context. Take it step by step.

There really is a cheat code to making these choices, though. If you're creating an immersive, full-screen experience – then it's likely fine to hide system chrome. Think media. Are you showing a full-screen photo, playing a movie or perhaps a game? Then you probably want things like the status bar or home indicator out of the way.

For example, in `ControllerConsiderationsFig2ViewController.swift` – we have a view that shows a image full screen.



In that case, I want the home indicator out of the way:

```
// Xcode -> UIkit ->
ControllerConsiderationsFig2ViewController.swift

class ControllerConsiderationsFig2ViewController:
BaseSampleViewController {
    private let imageView = UIImageView(image: .init(named: "Sky"))

    override func viewDidLoad() {
        super.viewDidLoad()
```

```
        imageView.contentMode = .scaleAspectFit
        imageView.frame = view.bounds
        view.addSubview(imageView)

        setNeedsUpdateOfHomeIndicatorAutoHidden()
    }

    override func viewDidLayoutSubviews() {
        super.viewDidLayoutSubviews()
        imageView.frame = view.bounds
    }

    // MARK: Overrides

    override var prefersHomeIndicatorAutoHidden: Bool {
        return true
    }
}
```

Or, say you are developing a game with touch controls. Then, it's also likely you want system gestures deferred at the screen edges; otherwise someone may swipe up to go home when they meant to perform an action in your game.

Outside those things though, to be honest, it *rarely* makes sense to hide or disable any of them. They are there across the system for a reason. People use them often.

Animations and the Status Bar

You can spice up hiding and showing the status bar by doing two things. One is to simply put updates in an animation block. So, if you're changing it from light to dark – you could animate that change:

```
UIView.animate(withDuration: 0.4, animations: {
    self?.setNeedsStatusBarAppearanceUpdate()
```

```
})
```

If you're hiding and showing it – UIKit has two animations you can implement. One is a slide, and the other is a fade. By overriding `preferredStatusBarUpdateAnimation`, you can use either of them:

```
override var preferredStatusBarUpdateAnimation: UIStatusBarAnimation
{
    return .fade // Or .slide, or .none
}
```

Again, so long as you call `setNeedsStatusBarAppearanceUpdate` in an animation block – that specific animation will be used.

SwiftUI

Even though this is a UIKit chapter – there is a loose corollary to SwiftUI. The `.persistentSystemOverlays()` modifier lets you tweak system chrome at the `View` level. Specifically, it can control what the docs refer to as "non-transient" system views, and that includes:

- The Home indicator
- The SharePlay indicator
- The Multi-task indicator and Picture-in-picture on iPad

Check out `ControllerConsiderationsFig1View.swift`, which shows this in action:

```
// Xcode -> SwiftUI -> ControllerConsiderationsFig1View.swift

struct ControllerConsiderationsFig1View: View {
    @State private var showSystemChrome: Bool = true
```

```
    var body: some View {
        ZStack {
            Color.red
            Button("Toggle System Chrome") {
                showSystemChrome.toggle()
            }
        }
        .persistentSystemOverlays(showSystemChrome ? .visible :
.hidden)
    }
}
```

In our case, it'll hide and show the home indicator since the other three things don't apply (SharePlay, PiP and the multi-task indicator). However, if you ran this code sample while using any of those technologies – the system would hide or show them accordingly.

Understanding Preferences

Finally, one important thing to note is that when you change *any* of these things – you're tweaking preferences. That is to say, UIKit is not guaranteeing that it'll respect your changes. The reasons (as far as I know) are not documented clearly, but I've not really seen any of my changes not respected.

I think Apple is simply playing it safe here and letting developers know that if something like the status bar should absolutely be showing – it's still going to show whether you want it to. While it's unlikely that you'll run a situation where your preferences are being ignored, it's still good to be aware of the fact that the system *can* override your intended changes.

Three Key Takeaways

1. The view controller is an essential building block of any iOS app created using UIKit.
2. There are several ways to override common system behaviors and chrome, but ensure you're "earning" the right to do so.
3. An important part to making any of these preferences work is to identify which controller should be in charge of changing the relevant API.

Your App's Website

The typical entry point for an iOS app isn't…well, the app at all. It's usually the App Store, from search or (hopefully!) a feature, a paid ad or some traffic to your app's website. And, that last one is extremely common.

As such, your app's website is important. It goes beyond just ensuring that it's responsive, useable and easy to navigate. Think of it as the front door to your app. People may only walk through it once to get into the "house", but it always leaves a first impression.

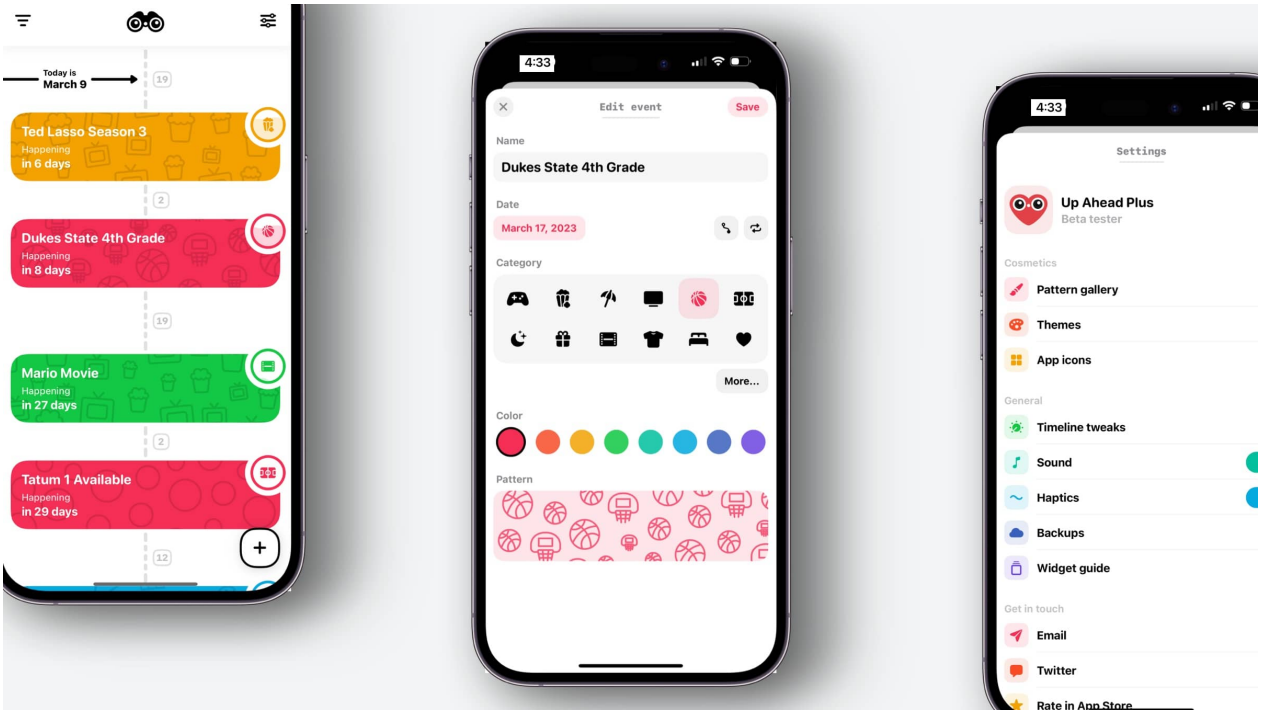
Let's spend some time on figuring out how to make it a good one.

Your App's Voice

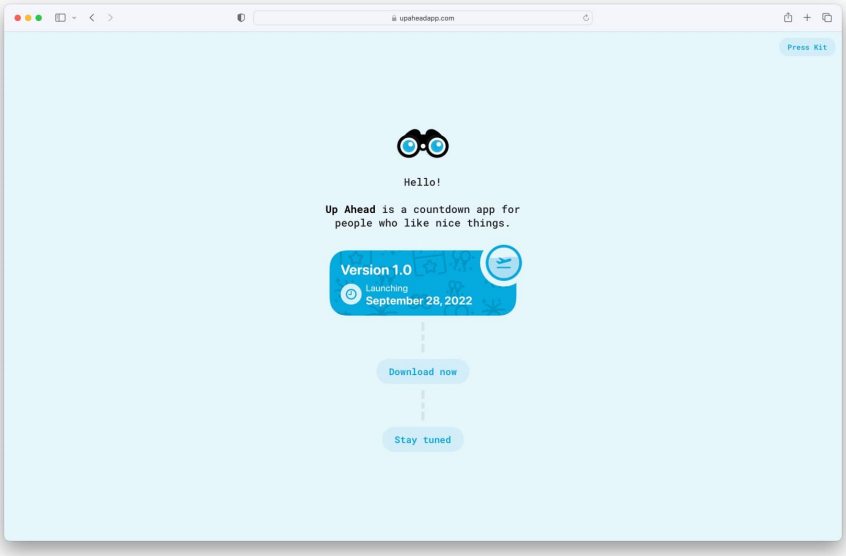
It all begins with your app's voice, something I've mentioned time and time again throughout this series. Just as your app's voice should match throughout its user interface, copy, and user experience – so too should your app's website match that tone and feel.

If we have a sports management app that has a business–like tone, then a playful, colorful website would likely not convey the experience people will have once they download it. That's off-putting and sets expectations you can't match. Instead, line the experience of the website up with the app itself.

Here's an example from Up Ahead, a playful app which helps people countdown to events they are looking forward to. Look at the app's bright colors, rounded edges and fonts:

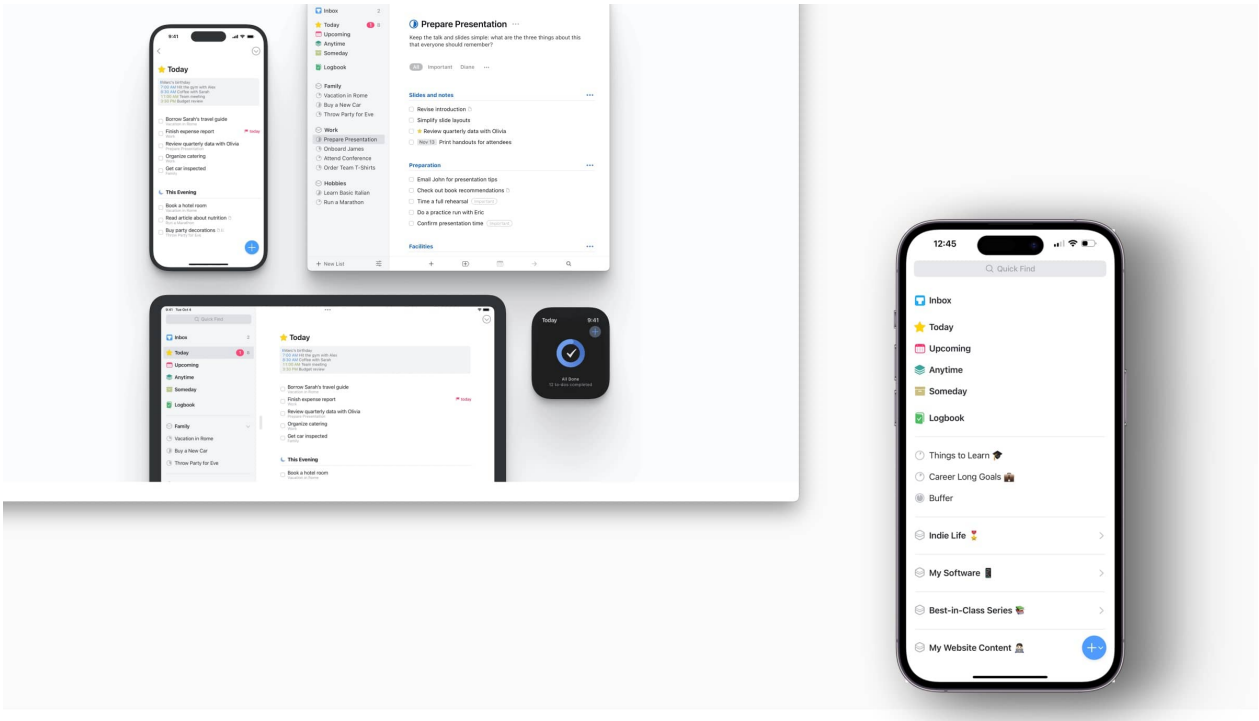


Now, here's the website. Simple, clean and to the point — but it matches the result app's experience perfectly.



In fact, if you click anywhere on the page, it even toggles the background color. This sets people up to expect an app which doesn't take itself too seriously, but puts a premium on fun interactions and playfulness.

For something that swings a bit the other way, check out Things 3. Things 3 helps people get things done, but it also isn't primarily positioning itself as something a business or team should use. It's a direct to consumer product. So, its tone strikes a nice balance between "This will help you be productive" and "It's not Google Docs.":



The Press Kit

Many of you reading this probably fall under the “Indie Developer” banner. If so, it pays to include a press kit on your website. If you’re a large business or company, ensure that there is an easy way to get in touch with your comms team or public relations.

A press kit that is readily available holds several advantages, chief among them:

- It allows for broad press coverage.
- It helps journalists get context quickly, avoiding back and forth exchanges.
- Again, see point one, you *want* press coverage, and you *need* to make that easy for journalists to do!

What does an effective press kit look like?

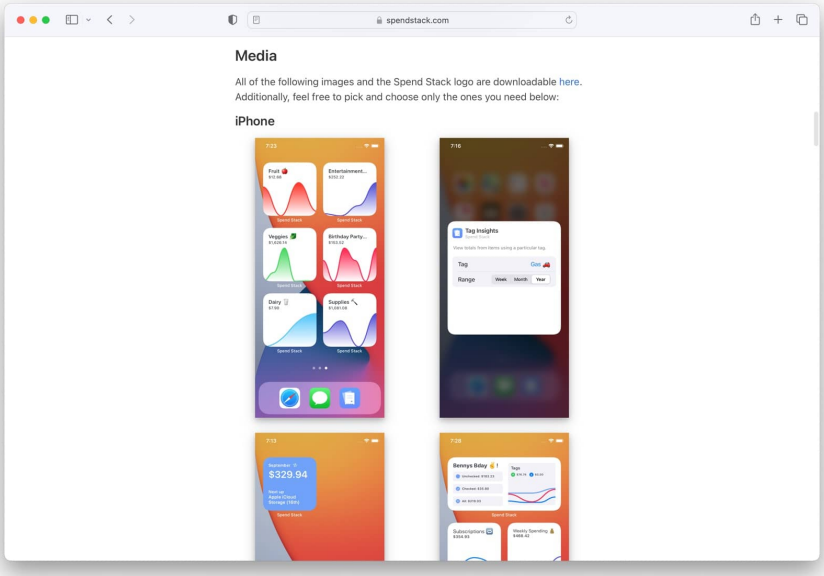
In short, it emphasizes getting to the point. It's not so much a marketing pitch as it is concrete information about what your app does, who it might be for, the problem it primarily aims to solve and then any novel features journalists may want to know about.

My previous app, Spend Stack, had a press kit that several journalists said they found to be a good example of how to do it right. Here's how I broke it down, section by section:

- The "What"
- Core Features
- Media assets
- Video trailers
- Pricing
- Original release date
- Availability
- And quick "About Us" section

I should note that numerous journalists have also told me that they prefer regular screenshots for media assets. That is, not the kind doctored up in iPhone device frames. They typically want to make one of their own like that. That said, it's not a bad idea to offer that "hero" image as well, just make sure they've got vanilla, unedited screenshots available to use (on each platform you support).

To give you some inspiration, here is what my press kit page looked like for Spend Stack:



The App Site Association File

Beyond just making information available, you can add some smarts to your website, too.

You can link up your website directly to your app, and have links to it end up inside your iOS app instead of its website. This makes a lot of sense for apps that have a web-based presence too, like many SaaS companies do (software as a service). There are other benefits as well, and we'll take a look at a few of them now.

The basic idea is to host a simple `.json` file on your server, and depending on what capabilities you want – you simply need to add the relevant key-value pairs within it. For example, maybe something like this:

```
{
    "applinks": {
        "apps": [],
```

```
        "details": [
            {
                "appID":"DJGHSDJGH24.com.bestinclassapp.home",
                "paths": ["*"]
            }
        ]
    }
}
```

Bonus: This can also aid in Handoff support from your iOS app to your website if you want that capability.

So, what would something like that do? If there was a QR code to your app — it would open the app instead of the website itself. Going further, if your app also had a powerful, feature-rich web app — maybe you still want those features (and links to them) to be performed in your iOS app when opened. That's what these links can do for you. In addition to `applinks`, here are some other keys you may be interested in:

- `details`: An array of dictionaries, one for each app that your site ultimately supports. Ordering matters in the array, as iOS looks up links according to it. This allows you to specify an app to handle a certain part of your website, like we just discussed.
- `details[appid]`: The team ID or app ID with a period, and your app's bundle ID tacked on.
- `details[paths]`: An array or strings that determine which parts of your website are supported by the app. Additionally, you can say which parts *aren't* supported.

On the client side, you'll need to use associated domains for your app. To get started, enable them under the "Signing & Capabilities" section in Xcode. This is for security

purposes, as a signed entitlement will be generated which tells iOS which sites that your app wants to be associated with. You can go further with associated domains beyond just deep links, though, by enabling things like easy password autofill and credentials:

```
{
    "applinks": {
    "apps": [],
    "details": [
        {
            "appID":"HSDJGH24.com.bestinclassapp.home",
            "paths": ["/posts/iOS",
            "/websiteapps/20?/*"]
        }
    ]
    },
    "webcredentials": {
        "apps": ["HSDJGH24.com.bestinclassapp.home"]
    }
}
```

Associated domains open up a lot for your website and app, but you might not need all of what it offers. Shared web logins, Handoff, universal links and App Clips all use them in some form or fashion.

Open Graph Tags

Hopefully, people will share your app's website at some point. Whether they tweet it, post it on a social network or send it via a messaging app – chances are that a rich link preview will be displayed. You want these to look nice, with an eye-catching image, headline and more.

To configure these, you'll need to get friendly with the Open Graph protocol. Fortunately, this is trivial to support. In the `<head>` of your site, you'll need to add a few tags:

```
// Just in case, Javascript isn't run when generating rich links -
so the value needs to be in the source. They can't be created
dynamically.
<meta name="og:title" content="The Title" />

// Icons (derived from a favicon or apple touch icon if it's there.
If not, you can roll with something like this):
<link rel="icon" href="path/to/icon" type="image/png" />

// Image (replaces the icon if present, but still provide both
because sometimes Messages prefers the icon over an image in
situations like poor networking conditions)
<meta name="og:image" content="path/to/image.png" />

// Video (of note, you can supply a URL to a YouTube video, which is
the only video player network that'll work if you don't use a file
iOS can natively play)
<meta name="og:video" content="path/to/video.mp4" />
```

To break that down, there's:

1. `og:title`
2. The icon
3. `og:image`
4. `og:video`

There are more, for example – `og:url` for the canonical URL, but that's typically included as part of the sharing action on iOS anyway. For completeness, though, here are additional Open Graph tags you can use as part of the specification:

- `og:audio`: A URL to an audio file to accompany the link.
- `og:description`: One or two sentence description.
- `og:determiner`: The word that appears before the site's title in a sentence, represented as an enum of (a, an, the, "", auto). The default is "" (blank).
- `og:locale`: The locale the site is using. It's formatted like this: `language_TERRITORY`. The default value is `en_US`.
- `og:locale:alternate`: If you've got other supported locales, you can add them as an array here.
- `og:site_name`: If your link is part of a larger website, this is the name which should be displayed for the *overall* site. It's unlikely you'd need this for your app, though.

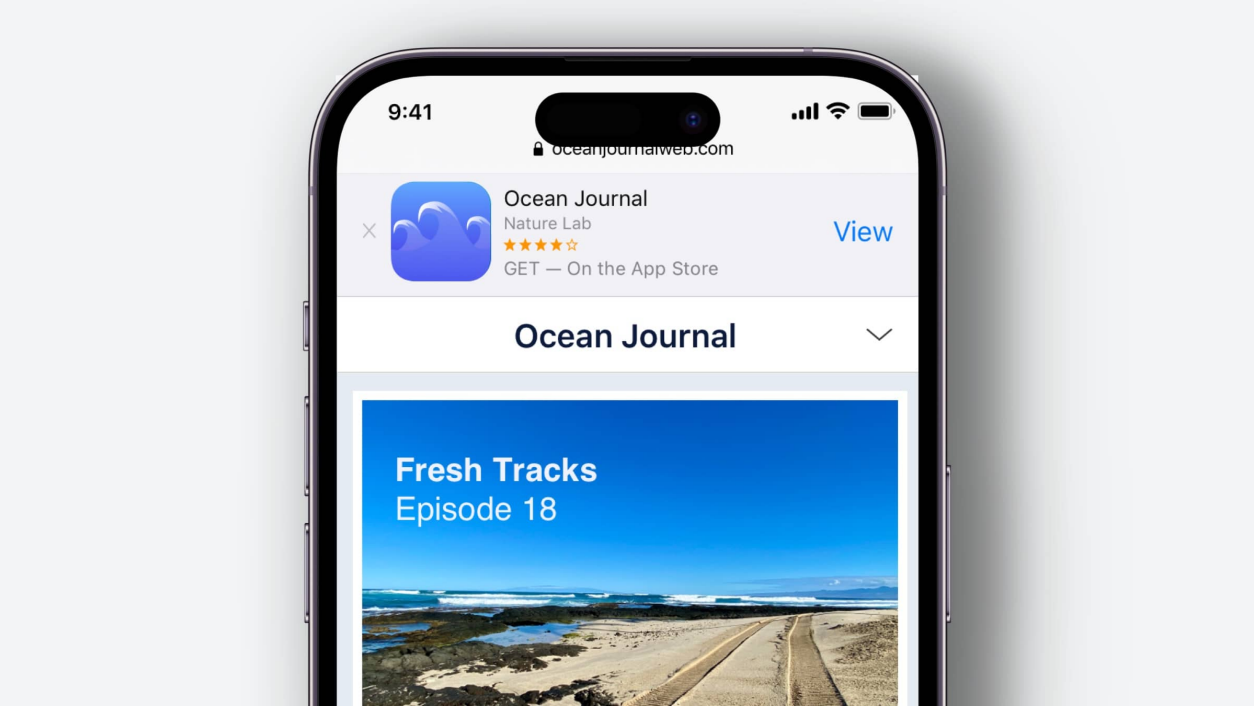
Tips

Apple's Marketing Materials

Apple provides some out-of-the-box marketing tools you can use for free by visiting https://tools.applemediaservices.com/. Using it, you can create QR codes, download media assets of your app icon and more. If you need to quickly share the news about your app in a pinch, this is a great place to go.

Include a Smart Banner

You can prompt people to download your app straight from your website, and you can use Apple's APIs to do it. Using a "smart banner", a nice little toaster banner will show at the top of your website that links directly to your App Store listing:



This is done via a meta tag, much like the Open Graph tags previously mentioned. All you need to hunt down is your app's identifier, and you're good to go:

```
<meta name="apple-itunes-app" content="app-id=123456789">
```

Three Key Takeaways

1. Your app's website is usually the first time people experience your app.
2. Try to explain the problem you solve with it, and make it easy to understand its value.
3. Use Apple's APIs to take them straight to your app.

Epilogue: The Feeling of Feeling Right

User experience. It's a tough nut to crack, isn't it?

In this book, we've dug in a lot, chapter after chapter, all in the name of what? I'd say the pursuit of finding that magical balance. The space that exists somewhere between providing a usable, predictable app — and one that leverages platform technologies mixed with a little touch of your own design tendencies and opinionated choices.

Admittedly, even with a lot of experience and knowledge, it's a hard space to find.

While it's not easy, it *is* a worthwhile pursuit. And, much like design (or, well — *anything* else), it's one that you'll truly never finish. It's a constant pursuit. But, the iOS platform thrives on a good user experience, and my hope is that you've come to form your own thoughts and rules around what a good user experience might look like on iOS.

I'd like to leave you with some core thoughts to wrap this up. Instead of a nice, personal story of how user experience has made or sunk an app — let's try and keep

the main things, the main things. At the beginning of this book, I said that "To me, user experience is less about how something looks, but much more about how it *feels*."

I think that's what I really wanted to help you find – can you identify what makes a feel right?

More importantly, can you now participate in getting an app towards that mark? Perhaps you can recognize those "paper cut" issues that plague some apps now, a few things that are off here and there. Or, maybe you've seen where a simple platform control will feel much better than some customization of one.

Or hey – maybe you can figure out when that's *not* the case, and that a little splash of flair is exactly what some part of your app needs! Simple thoughts, sure, but they have a lasting impact. You feel it when it's done right, and absolutely suffer from an app when it isn't.

All of this reminds me of why this user experience gig is so tough – it's subjective, opinionated and certainly a "feel" thing. So, with that, don't afraid to develop your own personal brand here, but make sure the person behind the screen that you're building for is always the focal point. Let's make their life easier, better and more pleasant!

And finally, never forget the importance of empathy. At the heart of every great user experience is a concentrated, unwavering understanding of the someone's needs – what are they *really* after? Who are they? What are the trying to do? Why did they think your app could do it?

Take the time to "be" the person who might use your app, heck – talk to them if you can! Let their feedback guide your decision-making process.

Finally – thank you, from the bottom of my heart, for letting me play a small part in your user experience journey. By virtue of even reading something like this book, you've got the dedication and drive to do great things.

See you on the App Store.

Book Four: iOS Technologies and Frameworks

The Heart of iOS

Are you familiar with the iOS word tag cloud?

If you've been at or watched any W.W.D.C. event, or more recently – Apple's Tech Talk series, then maybe you know what I'm talking about. It's that glorious slide that shows up in a keynote presentation that lists an avalanche of brand-new APIs or frameworks that are forthcoming.



As such, the tag cloud holds a certain weight among the developer community. The splash of new things to learn is a harbinger of a critical summer that lies ahead after W.W.D.C., and as much as it brings the excitement of nascent platform advances – there is also an equal sense anxiety to "learn the new thing."

When I sat down to plan this book, I realized I was essentially writing about exactly that. This book is really all about helping you learn "the new thing."

Because on iOS – there are simply *a lot* of them. The encouraging news about this whole deal, though, is that these frameworks do a lot of incredible, interesting things! You're probably not surprised to read that in a book over iOS frameworks, but bear with me.

Developing iOS apps is exciting, not just because of its cultural prevalence or the billions of users who use the platform daily, but also because the technology here is always advancing, improving and innovating. While some wax poetic at the fact that

some apps are all the same these days, or that Apple hasn't innovated enough or perhaps the platform has stagnated – history, up to this point, has proven that to be a false narrative.

Whether it's leaps in accessibility programming, with something like Voice Control, or letting an iOS device control your home with Siri automations – year after year, there are apt, useful and exciting technologies to not only use as a consumer, but also to develop for. To push them further.

In short, iOS has done everything *but* stagnate. No small feat, considering the platform has been around since 2007.

The APIs are Key

To that end, this is *the* book, if I'm being honest, when it comes to the best-in-class series. It's all about those iOS frameworks, how to use them and how to take them a bit further. They are the heart and soul of the platform, and without them – iOS would just be another fancy way to use a phone.

But, still – why should you *actually* bother, beyond the notion that these APIs are simply fun to use?

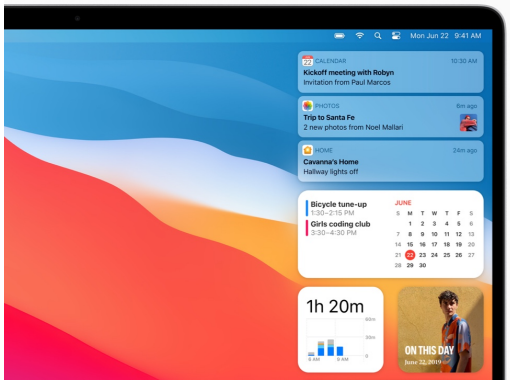
In short, these iOS frameworks are the secret sauce of the platform. They are why people come here, and why they stay here. Let's use a real-world example to demonstrate:

WidgetKit.

This particular framework allows for widgets to be used all over iOS. On the tin, that doesn't seem like such an enormous deal, right? But consider this – widgets, in a similar form as they exist on iOS, have been around for quite some time.

Beyond that, the concept of widgets themselves certainly aren't a new, or even novel, idea. Android has had them for a long time. Even Apple themselves have had a form of them before in macOS with its dashboard:

Modern macOS Widgets
> Powered via WidgetKit.





Early Widgets
> Found in OS X's Dashboard section.

Yet, iOS' release of WidgetKit was decidedly different. It caught the world on fire, and transcended beyond just "the tech sphere" and into the everyday iOS user. It opened several new ways to customize your phone, allowed people to use it in new ways, and generally brought about new possibilities.

For us developers or designers, what more could you want?

Consider the popular app Widgetsmith, from indie developer David Smith. He launched an app centered all around widgets, and how to customize them easily – anyone could use it. It was a massive success, surpassing Facebook, Instagram, and Snapchat in the downloads chart when it shipped. In fact, it cracked the coveted number one spot.

The momentum hasn't stopped, either. David announced that it recently surpassed — are you ready for this? — 100,000,000 downloads. Again, that's *one hundred million*.

A marquee API changed everything for plenty of people, and the chance for that to potentially happen to you occurs every year. Of course, we'll be honest here — for every WidgetKit, there is also an iMessage App framework. By and large, though, what Apple introduces to its platform — tends to take hold.

iOS APIs are the beating heart of this platform. You absolutely want to pay attention to them.

Assumptions I've Made

Before we dig in, I aim to lay out some assumptions I've made about you. I assume you:

- Have programming experience on the iOS platform.
- You could tell me what functions, delegates or view controllers are.
- You probably could reason about Auto Layout versus using frames.
- Or, if you've started with SwiftUI — then you're comfortable using it.

If you don't qualify or fit under these definitions, don't sweat it. Keep reading. Learn more as you go. Worse case scenario — perhaps you don't understand enough to take it all in. That's okay because I'll be here when you do.

What This Book Covers

This book is a celebration of iOS' core APIs and frameworks. That means a lot of the big ones. The ones that are announced at W.W.D.C. each year. Beyond that, I also include other frameworks that potentially aren't as well known, but are helpful to have a working knowledge of regardless – think something like AirPrint.

We will dive into these frameworks, and see how they work and how to use them. If you have read any of the previous books in my Best-in-Class iOS App series, you have a great idea of what to expect. If this is your first one, the thinking behind my format is threefold – and each chapter loosely follows this structure:

1. Build intrigue behind a framework and introduce it.
2. Set the foundation for it by seeing how it works at a high level.
3. Then, close things out by covering more in-depth tips.

That's it. With that, let's start diving into some of iOS' most fun, intriguing and exciting topics.

See you in Xcode!

Three Key Takeaways

1. iOS APIs and frameworks are critical to creating premiere iOS apps.
2. There are a lot of them, and they often change — so do your best to keep up.
3. However, you can't know them all, so try not to agonize at the thought of feeling behind — because you will!

App Clips

When you think about it, iOS apps usually do one thing very well. Calendar for iOS helps you manage your schedule. Stocks, well — helps you keep up with the stock market throughout the day. Messages is for communication.

Even though their features may extend beyond those things, their singular focus is obvious. They have a problem, and they solve it well. Every feature added (or removed) hinges on that purpose.

Which brings us to App Clips.

In many respects, they take this concept further. An App Clip is a slice of your app that can be quickly downloaded on the fly. Its goal? To simply perform one singular task that your app does well.

Let's peek at some existing App Clips that are out there. Panera Bread's App Clip lets you place a quick order. Spin's lets you unlock a scooter to rent. All without downloading their full app:

```
App Clip Examples
> Each one offers a singular, focused task to perform.
```



In short, App Clips are a way to get people right into a slice of your app. They are a lightweight version of it, and they encourage people to want *more*. In my mind, the goal of an App Clip is to delight someone with a quick, focused experience.

If it all went well, then at the end of it, they should want to do one thing – download the full app. To that end, App Clips are also transient in nature. People shouldn't expressly *want* to keep reusing them in most cases, they should be impressed with the experience such that they want to experience more of it.

Hence, downloading the full app.

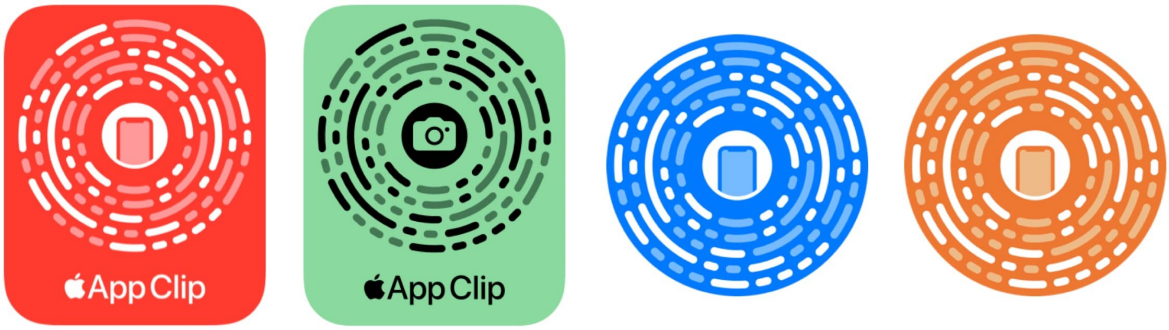
Entry Points

App Clips are flexible. There are several ways someone could find yours. Chief among them are App Clip Codes:



The circular, dashed pattern is reserved uniquely for App Clip Codes. Though, they can be customized to a certain degree as well:

```
App Clip Codes Customizations
> You can change color, shape and more.



> Ex:
```



Using the Camera app, or Code Scanner in Control Center, people can quickly scan one of these to get right into your App Clip. They naturally work well for marketing purposes – and can be printed out individually or used on flyers or posters. They can also leverage NFC tags, which allow people to tap on it to use your App Clip.

App Clip Codes are *perferred* by Apple, but they are far from the only way an App Clip can be discovered:

- **NFC Tags:** An NFC tag can be associated to an App Clip and trigger its download.
- **Safari:** Using a Smart App Banner, you can configure them to use your App Clip instead of going to download your app. This can be a phenomenal way to "preview" your app, since App Clips are inherently low effort for prospective users to try things out. They don't stand to lose anything from trying one.

- **Messages:** You can set up an App Clip to be shared with others and sent via Messages, and they work much like normal links do when they are shared.
- **QR codes:** The old standard that everyone loves to hate, but none-the-less effective and well understood.
- **Maps:** By associating your App Clip to a certain location, Maps can offer it when someone views that particular place within Maps. This is great for brick-and-mortar business' like coffee shops or restaurants.
- **Spotlight Search:** By searching for a similar term in Spotlight, Siri can also suggest your App Clip here.

Anatomy of an App Clip

App Clips have three primary components:

1. **The Containing App:** Of course, there is no App Clip without an app. You can have an app without an App Clip, but the inverse is not true. As such, the app can expose bits and pieces of it to an App Clip.
2. **App Clip Experience:** There are registered URLs in App Store Connect that, when opened, will be handled by an App Clip. Previously, these were consumed by a browser and simply showed as a web page. Now, you can do both.
3. **App Clip Extension:** The extension you provide to show a "vertical slice" of some of your app's core experiences.

As we mentioned above, App Clips are surfaced throughout iOS to quite an impressive degree. Whether it's in Messages, Safari, NFC tag, Maps, business details or Q.R. codes – people will have a chance to jump right into yours often. In that respect, they are *discoverable*.

People also don't need to consciously manage App Clips. They are there when folks need them, but eventually – iOS will delete it if it's not been used in some time. That incudes the App Clip itself, any keychain data and its shared container.

How it Works

The great thing about App Clips is, well – they are *just* your app for the most part. That means there aren't particularly special lifecycle notes to consider, too many memory limits to think about, or other similar affordances you typically have to take into consideration when developing an extension.

While it's true there are some frameworks that simply won't work, such as Health and Fitness frameworks – nearly all the rest do. You can even access CloudKit's public container in a read-only manner. Plus, several core, "OS" level features are available to use – think Sign in with Apple and Apple Pay. In fact, you can even migrate data over to your app from your App Clip or persist sign in status using `ASAuthorizationAppleIDProvider`.

As you can see, App Clips are basically mini apps from both a philosophical standpoint *and* a technical one. After all, the entire point of these are to show off a piece of your app. To that end, Apple has made App Clips immediately understandable, in that you're simply supplying a piece of your app to the target extension.

If there is anything to keep in mind when building an App Clip, it pays to know about `NSUserActivity`. The do-it-all API, which we've covered previously in various books in this series, is what you'll query to know more information about an App Clip.

For example, if we want an App Clip to show information about each book in this series – there wouldn't be a need to create five different App Clip targets. I could just use one, and use a parameter from the provided user activity, typically in the form a URL, to know which one to show:

```
// Xcode -> Best-in-Class App Clip -> Best_in_Class_App_ClipApp

@main
struct Best_in_Class_App_ClipApp: App {
    @StateObject private var bookWrapper: BookWrapper = .init()

    var body: some Scene {
        WindowGroup {
            ContentView()
                .environmentObject(bookWrapper)
                .onContinueUserActivity(NSUserActivityTypeBrowsingWe
b) { activity in
                    guard let appClipURL = activity.webpageURL,
                          let components = URLComponents(string:
appClipURL.absoluteString),
                          let queryItems = components.queryItems
else { return }

                    if let bookQueryItem = queryItems.first?.value,
                       let bookModel: Book = .init(rawValue:
bookQueryItem.capitalized) {
                        self.bookWrapper.book = bookModel
                    }
                }
        }
    }
}
```

No matter the technical stack you've chose for your app, this process should be a familiar one. For SwiftUI, that means using `onContinueUserActivity(_:perform:)` as we've done above. For UIKit, and scene-based apps, that would entail `scene(_:willContinueUserActivityWithType:)`. Further, if you're still on the `UIApplicationDelegate` lifecycle, then it's simply a matter of using `application(_:continue:restorationHandler:)`. In short, if you've already gotten your hands dirty with lifecycle events – then you already know how to wrangle launches of an App Clip.

Of course, if you have a singular App Clip who doesn't need to know anything about parameters – then things are even simpler: Include the code and interface you want to show with the App Clip and…that's it. You're done! Because really, App Clips work just about like any other app.

To be honest, most of the work with App Clips is administrative in nature. Before launching one, you'd configure metadata about it in App Store Connect. Perhaps you'd even create an App Clip card to make its presentation a little nicer when it's presented. Or, you'll update your `apple-app-site-association`file for universal link handling.

Other than that, again – it's simply building an app.

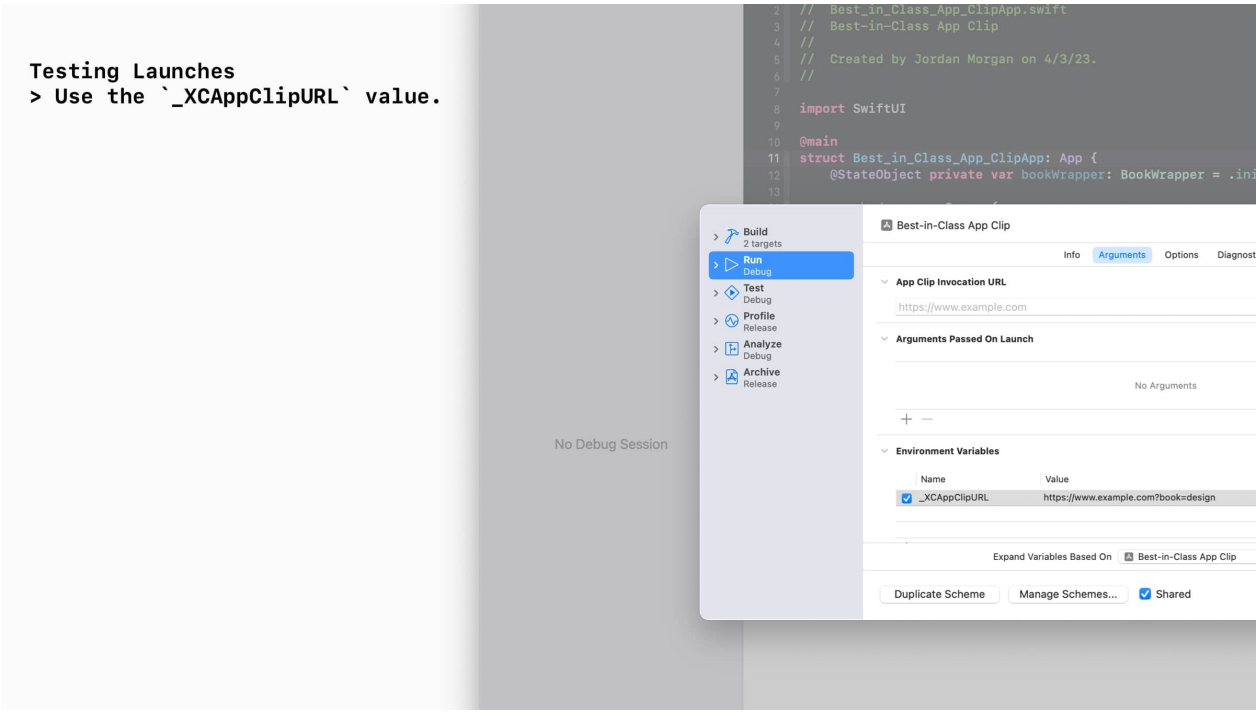
Tips

Testing App Clips

Testing App Clips locally is straightforward, as you can simply run the target directly from Xcode. However, you may want to test different permutations of whichever URL

you've created for the App Clip, and you can easily do that by using an environment variable.

To do so, simply add the `_XCAppClipURL` environment variable with the URL you wish to test to your App Clip's scheme:



This ensures that the user activity's `webpageURL` property will be set to whatever you've added here.

If you want the full blown App Clip experience, though, you may want to enable `Local Experiences` on your device, which itself is found under `Developer Settings` within the Settings app. This route supports nearly all the ways that App Clips can show, including Messages, Safari, NFC, and App Clip code and QR codes.

Confirming Universal Link Information

You can use `SWCutil` to peek at your AASA file (Apple app site association file) to confirm things are deployed as you expect. If they aren't, for example – you've got the wrong, or missing, bundle identifier – you can quickly verify that information.

For example, for `checkcoverage.apple.com` you can see its App Clip bundle identifier and its app links:

```
$ SWCutil dl -d checkcoverage.apple.com
{
    appclips =     {
        apps =         (
            "47Z27PHCDE.com.apple.supportapp.Clip"
        );
    };
    applinks =     {
        apps =         (
        );
        details =           (
                        {
                appID = "47Z27PHCDE.com.apple.supportapp";
                paths =                 (
                    "*"
                );
            }
        );
    };
}
```

Test App Clips Sizes

You can inspect the size of your App Clip ahead of time by exporting your IPA files. The process has a few steps, but it's not complicated. I'll list them out in order:

1. Choose your *app's* primary scheme, *not* your App Clip's scheme.

2. Go to `Product -> Archive` in Xcode.
3. Once that's done, in Organizer, select `Distribute App` and choose `Development`.
4. Next, choose your App Clip.
5. You should get an App Thinning dropdown at this point, so simply choose all the device variants you'd like to test for.
6. Now, prepare the archive and export it.
7. Within the resulting file, you should see `App Thinning Size Report.txt` which will contain all the information, per device, that you're after.

Reducing App Clip Sizes

App Clips take part in app thinning, but even so – sometimes things may balloon a little larger than you'd like. If you find that your App Clip size is bigger than you'd prefer, here are some things you can try to get the size knocked down:

- Rely on SVGs for iconography. These assets can be easily scaled to whichever size you need.
- If you're using audio, ensure it's compressed. As far as my research can tell, Xcode's asset catalogs do not perform any optimizations for audio – so try to use MP3 or AAC codecs. Further, you can try to use a reduced bit rate, just ensure that the tradeoff with your size versus audio quality remains acceptable.
- Double-check your dependencies. Though it's easy to reuse code from your primary app target, that also means it's easy to unexpectedly drag in third-party libraries that you may not need.

- Lean on Apple's frameworks. This is evergreen advice, but remember — these are bundled with iOS itself. So, you won't incur a size penalty when you use them. For example, do you *really* need that .json decoding library? Or, could you simply roll with your own `Codable` implementation and remove it?
- In fact, take that thought even further — lean on SF Symbols for as much iconography as possible. Again, those are bundled with the operating system itself. Image and icon packs are a common source of size inflation, and this is a great way to avoid that trap.
- This may seem obvious, but use asset catalogs to take advantage of all of their benefits — many of which relate to intelligent handling of images and videos.
- Make sure you only have any necessary files included in it. It may seem silly, but perhaps you might've accidentally brought in some README file, or similar, during development.

Leverage Compiler Flags

Even though code reuse is a core part of building your App Clip, sometimes you simply need to test if your code is executing within one or not. Compiler flags are a great choice for this.

To add one, go to your App Clip's target and then:

1. Under `build settings -> Swift Compiler`
2. Click `Define New Condition` and add `APPCLIP` for both schemes to conditionally compile things out.

With that, now you could do something like this:

```
#if !APPCLIP
// Access HealthKit, or similar.
#end
```

Encourage an App Download

Once someone has performed the primary task in your App Clip, consider prompting them to download the app itself. Using `StoreKit`, you can offer up a one-shot download link.

Let's say we've got a coffee ordering App Clip. If the user has ordered a few coffees, or even one – you could show a download banner for the whole app itself, so they could get more out of the experience:

```
struct ContentView : View {
    @StateObject private var purchaseData = PurchaseData()

    var body: some View {
        NavigationStack {
            CoffeeListView()
                .environmentObject(purchaseData)
        }
        .appStoreOverlay(isPresented:$purchaseData.hasBoughtCoffee)
{
            SKOverlay
                .AppClipConfiguration(position: .bottom)
        }
    }
}
```

Three Key Takeaways

1. App Clips offer a focused "vertical slice" of your app.
2. Keep them light weight and focused.
3. If it makes sense to migrate any data or sign in states from your App Clip, take care to do so.

AirPrint

AirPrint removes friction.

Seriously, think about it – since you're reading this book, I'm going to assume you are the "tech person" in your family circle. And what does *everyone* in your family circle struggle with?

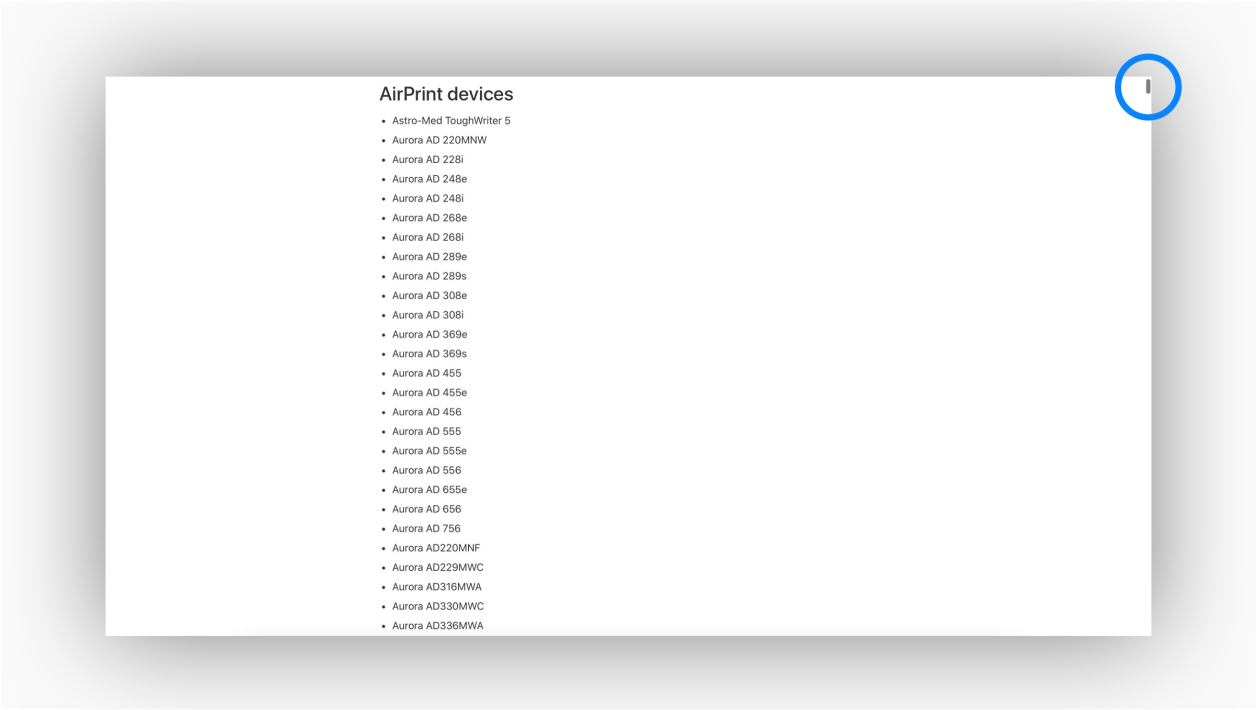
Printers.

Let's be honest, printing software *sucks*. Messing with drivers, connecting printers with their barely usable touch screen interfaces to the internet or messing around with obscure settings – they just aren't pleasant to use. Apple must've noticed this, and in an effort to streamline the whole printing pipeline, they've given everyone in the Apple ecosystem AirPrint.

AirPrint essentially abstracts the entire process behind a nice iOS (or macOS) interface, making the complete task of printing stuff trivial. AirPrint, which is the marketing name of the technology, exposes a set of APIs to select a printer, customize the content being printed and more. Since the technology is already built right into most

printer models, much like CarPlay is available on most modern cars, it's the primary way to handle printing in our apps.

In fact, it's safe to assume that if you are dealing with an iOS app that needs to print in any capacity, this is *the* way to do it. To wit, here's a snippet of the models which support AirPrint. Take note of the scrolling indicator on the right side, basically – the list is *long*:



Apple has made sure AirPrint is installed on most modern printers.

How it Works

First, let's rip off the bandaid. This is a UIKit only framework. While you can certainly wrap a SwiftUI view within UIKit, there is no proper API to utilize AirPrint in SwiftUI. With that said, it pays to learn about the inner workings of AirPrint. Though it's "UIKit

only", once you know how its pieces fit together you can make informed architecture choices to support printing – regardless of whether you're using UIKit, or SwiftUI.

To get going with AirPrint, you need to get familiar with `UIPrintInteractionController` – it runs the process from start to finish. This is a view controller that you'll present, and from there the framework will guide the user along the printing process:

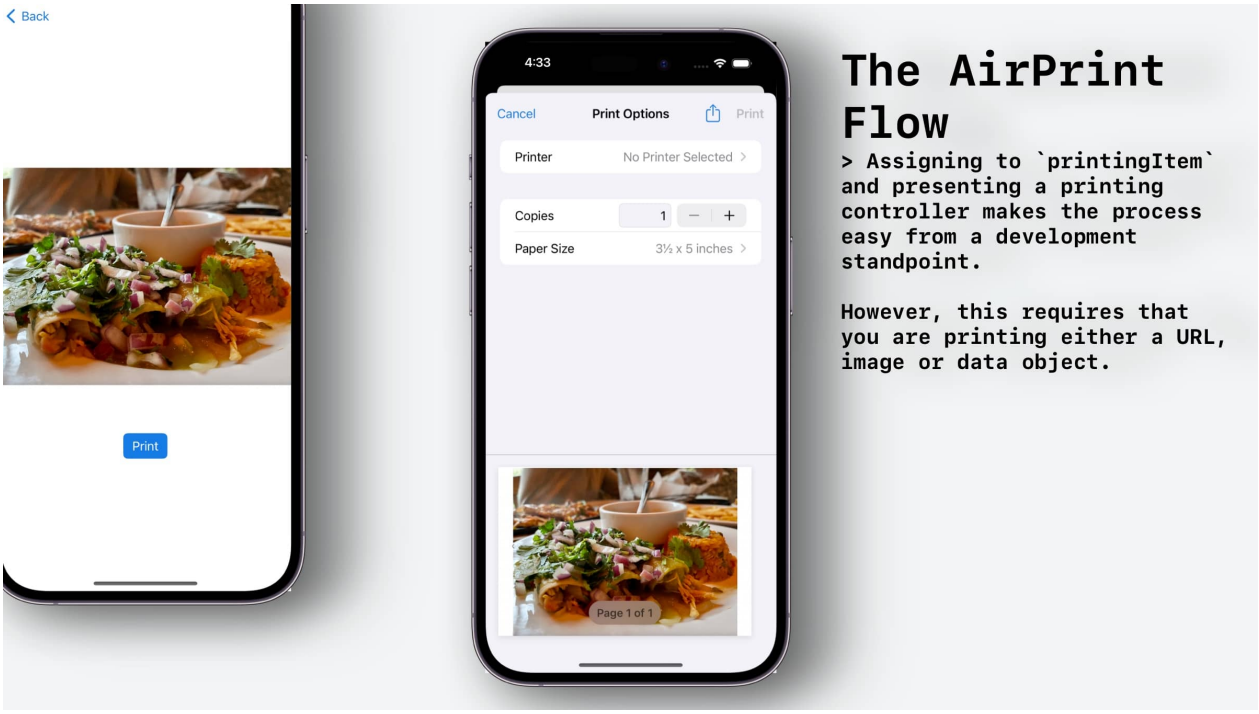
```
let printerController = UIPrintInteractionController.shared

printerController.present(animated: true) {controller, completed,
error in
    // Handle errors or success
}
```

However, that doesn't cover arguably the most important part of printing, which is identifying the content that you actually want to print. To that end, there are a few different ways to pass off items to print to a print controller. You could simply assign a view, photo or URL, or you can render your content to print too. The former part is fairly easy, the latter can be more involved.

Let's check out how to assign views you've already got first, and we can do that with the `printingItem` property. This is a "ready-to-print" view that you can assign to, and so long as it's a URL, Data or `UIImage` instance – you're all set:

```
// Xcode -> UIKit -> AirPrintFig1ViewController.swift

let printerController = UIPrintInteractionController.shared

// View full code sample to see how the UIImageView is setup
printerController.printingItem = imageView.image
```

Here, we've already got an image in the hierarchy, and that's all we want to print. So, the process of setting things up is as simple as assigning the image to the printer controller's `printingItem` property:



Even better, if you've got multiple items of these types that you need to print – you can do that in a one-shot flow, too. There is also a `printingItems` property that allows you to assign an array of printable items to it. The same rules apply, though – it should be images, URLs or data.

If you use either of these properties, the printing controller sets its `printPageRenderer` and `printFormatter` properties to nil. Those properties deal with the more advanced form of printing that we'll look at here in a bit, but it's still something to be aware of. It's either/or, you use printing items, or you use a renderer or formatter that you provide.

Before we do that, though, let's take a quick aside and chat about printing metadata.

Printing Info

In the code sample above, there was a little snippet that I omitted:

```
let info = UIPrintInfo.printInfo()
info.outputType = .photo
info.jobName = "Best-in-Class Photo Example"
info.orientation = .portrait

printerController.printInfo = info
```

The `UIPrintInfo` object says it all in its name – this is how you provide any metadata you might need to associate with the printing job or its contents. UIKit will do its best to provide sensible defaults, so if you don't assign to the printing controller's `printInfo` property – that's what you'll get. However, by using `printInfo` – you can provide:

1. **The job's name:** Using `jobName`, you can name the actual printing job.
2. **Print orientation:** You can choose, unsurprisingly, either portrait or landscape.
3. **Duplex Options:** If the printer supports duplex printing (i.e., printing on both sides of the paper), you can set either short or long edge printing.
4. **Output Options:** Choose between grayscale, general (probably what you typically want), photo or photo grayscale.
5. **Printer Identifier:** This is read-only, and the user will select it in the printing interface.

So, if you need to change any of these types of fields, create a printing info object and provide it to the printing controller. If you don't, you'll be just fine with the defaults that UIKit gives you.

Print Formatters

There are a variety of formatters you can set up for printing, and these are what will give you more flexibility if you need it. Using them, you can supply text, attributed strings, markup and more.

For example, consider a shopping list such as the one found in `AirPrint-Fig2ViewController.swift` in the sample project. It's generated from this function:

```
// Xcode -> UIKit -> AirPrintFig2ViewController.swift

private func generateListText() -> NSAttributedString {
    let makeup = ["Onions", "Cheese", "Dough", "Italian
Seasoning"].map { val in
        return "- \(val)"
    }
    let makeupString = makeup.joined(separator: "\n")


    let pizzaShoppingList = NSMutableAttributedString(string: "")
    pizzaShoppingList.append(.init(string: "Pizza Shopping List\n",
attributes: [.font: UIFont.preferredFont(forTextStyle: .title1)]))
    pizzaShoppingList.append(.init(string: makeupString, attributes:
[.font: UIFont.preferredFont(forTextStyle: .body)]))

    return pizzaShoppingList
}
```

We can turn around and use that with one of UIKit's formatting classes, which are:

- **View formatter:** Used for printing drawn content of a view.
- **Simple text formatter:** Used for plain text printing.
- **Markup text formatter:** Used for HTML-based printing.

For our purposes, we could use the simple text formatter to print out the list, even with its attributed string attributes:

```
// Xcode -> UIKit -> AirPrintFig2ViewController.swift

private func printList() {
    let printerController = UIPrintInteractionController.shared
    let info = UIPrintInfo.printInfo()
    info.outputType = .photo
    info.jobName = "Best-in-Class Text Example"
    info.orientation = .portrait

    let textRenderer = UISimpleTextPrintFormatter(attributedText:
generateListText())
    printerController.printFormatter = textRenderer
    printerController.printInfo = info

    // Present the picker, more on that below in "Tips"
}
```

By passing the attributed string to the `UISimpleTextPrintFormatter`'s initializer, we've got all we need to print out the entire list by assigning it to the print controller's `printFormatter` property. These formatters are incredibly flexible and cover much of the ground you'd need to for most printing jobs.

Further, some of UIKit objects have a printing formatter built in. A web view, text view or map view each can supply their own print formatters. Consider `AirPrint-Fig3ViewController.swift`:

```
// Xcode -> UIKit -> AirPrintFig3ViewController.swift
```

```
printerController.printFormatter = mapView.viewPrintFormatter()
```

The printing code is virtually the same as the previous example. The only change is using the `mapView` to supply the formatter via its `viewPrintFormatter()` function. With that, the entire map shown in the view is ready to print.

Tips

Custom Rendering

Recall that `printItem` can handle .pdf data assets. So, if you've got a complex .pdf document, or hierarchy, you can leverage `UIGraphicsPDFRendererFormat` to draw the entire document and then print it:

```
let renderer = UIGraphicsPDFRenderer()
printerController.printingItem = renderer.pdfData { ctx in
     // Drawing code
}
```

This works with a regular graphics renderer too, outside the .pdf format. If you want to go even further, you can completely customize the whole printing content pipeline with the use of `UIPrintPageRenderer`, which can be used with or without the assistance of printing formatters. By inheriting from this class, you'll have the most control possible to tweak things like customizing the header, content, footers and more.

Print Command

The plist key, `UIApplicationSupportsPrintCommand`, can help with printing. If you support it and add it to your app's `info.plist` file with a corresponding value of

`YES`, you'll get the `⌘+P` printing shortcut included in the relevant places depending on the platform. So, think keyboard shortcuts on iPadOS, or the menu bar in macOS.

When the command is invoked, UIKit will travel the responder chain and call this function on the first responder it finds that implements it:

```
optional func printContent(_ sender: Any?)
```

So, while it's a UIKit-based API (as is AirPrint as a whole, currently), it's a nice little touch to include throughout your app by implementing the above printing method. That way, you get the shortcut added for you out-of-the-box.

Presentation Quirks

There is a bit of a dance you might have to do for AirPrint presentation to work on iOS. The question really comes down to iPhone versus iPad presentation — which isn't something you typically want to think about. Developing apps should usually avoid the "Is this on iPad or iPhone" question, but occasionally you might have to.

Here's what it might look like:

```
if UIDevice.current.userInterfaceIdiom == .pad {
    printerController.present(from: .zero, in: view, animated: true)
{controller, completed, error in
        if error != nil {
            Swift.debugPrint("Error while printing: \
(String(describing: error))")
        }
    }
} else {
    printerController.present(animated: true) {controller,
completed, error in
        if error != nil {
            Swift.debugPrint("Error while printing: \
(String(describing: error))")
        }
```

```
    }
}
```

Here, the second printing method is custom-built for iPhone. You'll get the "slide up from the bottom" modal presentation. But, that won't work on iPad. There, you'll use the one above – which can also handle popover presentations on iPad common from things like bar button items.

Since the "iPad" version works for iPhone, too, however – I typically use it for both platforms.

SwiftUI Printing

Even though it's a UIKit-based API, you may find that printing in SwiftUI is easier than you think. One reason? The printing controller is disconnected from UIKit's presentation from a specific view controller, for the most part. That means you can do things like use an existing printing formatter with SwiftUI without much trouble.

Consider this example in `AirPrintFig1View.swift`, which recreates the shopping list printing from earlier in UIKit:

```
// Xcode -> SwiftUI -> AirPrintFig1View.swift

var body: some View {
    VStack {
        Text(generateShoppingList())
    }
    .toolbar {
        ToolbarItem(placement: .navigationBarTrailing) {
            Button {
                printList()
            } label: {
                Image(systemName: "printer.fill")
            }
```

```
            }
        }
}

// The printList function
private func printList() {
    let printController = UIPrintInteractionController.shared
    let printInfo = UIPrintInfo.printInfo()
    printInfo.outputType = .general
    printController.printInfo = printInfo
    printController.showsNumberOfCopies = false

    let attributedString =
NSAttributedString(generateShoppingList())
    let formatter = UISimpleTextPrintFormatter(attributedText:
attributedString)
    printController.printFormatter = formatter

    printController.present(animated: true, completionHandler: nil)
}
```

Basically, if you can get to any underlying data model in SwiftUI – then you can probably use one of the existing formatters. At that point, it's a matter of passing off the data you have to the AirPrint APIs, it has nothing to do with SwiftUI at all. And, again, since the presentation doesn't have to be tied to SwiftUI, it makes the process a bit easier.

Three Key Takeaways

1. AirPrint is the technology available is most printers to handle any print job in our iOS apps.
2. While it's only available in UIKit, it's not too hard to bridge over to SwiftUI.
3. Be sure to leverage its more powerful features to customize printing content in your app.

AirDrop

AirDrop was one of those magical Apple feature releases. Before it shipped, marshaling data to and from other devices was possible, but it wasn't particularly seamless. You could email yourself a copy of something, shuttle off an image via Messages – the list goes on.

With AirDrop? It's a simple as popping open the share sheet and tapping one button. Sent. Done.

As developers, you might think this is just a core iOS experience that you can't really build around – but that's not *entirely* true. While it's accurate there is no AirDrop framework to import, we are actually better for it.

Why?

Because integrating with AirDrop doesn't require much more than leveraging the APIs you are already familiar with—the share sheet. In fact, some developers may be

surprised to find that their custom models or general data support AirDrop without them having done anything extra. Let's dig in.

How it Works

To break it down, there are two routes to sharing data from your app via AirDrop. The first is to implicitly use the built-in activity type of `.airDrop` with types that already support it, either though SwiftUI APIs such as `ShareLink` or the activity item controller in `UIKit`. The second is to register your Uniform Type Identifier (going forward, I'll simply refer to this as a "UTI") for a custom data model, and then share it via the share sheet as well.

Sharing Simple Types

The good news about going this route, sharing either images, text, or URLs — is that there isn't really anything you need to do to indicate to iOS that you want AirDrop to be available when sharing them. It just *is*, which is a good thing — because if we had to signal we wanted to support it every time we shared something, then AirDrop certainly wouldn't be used as much.

The things we get for free on iOS as developers are some of the best reasons to make apps for this platform.

Here's an example to demonstrate in UIKit:

```
// Xcode -> UIKit -> AirDropFig1ViewController.swift

// See full code sample to see how a text label is setup
private func shareTextViaAirDrop() {
```

```
    let activityViewController =
UIActivityViewController(activityItems: [textLabel.text!],
applicationActivities: nil)
    activityViewController.popoverPresentationController?.sourceItem
= navigationItem.rightBarButtonItem

    // Anything you *don't* want to be included
    activityViewController.excludedActivityTypes =
[UIActivity.ActivityType.postToFacebook,
UIActivity.ActivityType.postToTwitter]

    self.present(activityViewController, animated: true, completion:
nil)
}
```

Since we're simply sharing a text label's `text` property, AirDrop will show as an option within the share sheet. There's no need to explicitly state that you want it there by having to do something like assigning the `UIActivity.ActivityType.airDrop` type to some obscure property on the share sheet.

Now, on the *other* hand, if you didn't want it there – you could certainly do that by changing the code sample above to include the `airDrop` activity type in the `excludedActivityTypes` array:

```
private func shareTextViaAirDrop() {
    let activityViewController =
UIActivityViewController(activityItems: [textLabel.text!],
applicationActivities: nil)
    activityViewController.popoverPresentationController?.sourceItem
= navigationItem.rightBarButtonItem

    // Exclude AirDrop
    activityViewController.excludedActivityTypes =
[UIActivity.ActivityType.postToFacebook,
UIActivity.ActivityType.postToTwitter,
    UIActivity.ActivityType.airDrop]
```

```
    self.present(activityViewController, animated: true, completion:
nil)
}
```

But, with AirDrop being a core platform feature – I struggle to find any reason as to why you might do that. In SwiftUI, the same rules apply for AirDropping simple types. If you were to invoke the share sheet via the `ShareLink` view and use it with either images, text or URLs, then AirDrop would automatically show as an option.

That's all you need to know for sharing primitive types with AirDrop. But, the real fun of AirDrop comes from adding that extra level of convenience to your app by allowing your own proprietary models to be shared with it, too.

Sharing Custom Data Models via AirDrop

Sharing your custom models from device to device isn't *hard*, but it does have a few steps that aren't immediately obvious if you were to go looking for them. As such, the end-to-end process requires a bit of know-how from a few different pieces of API, but once you've got the general flow down – it's not hard to share multiple different model types going forward.

To kick things off, here is what all the steps look like for the most part, no matter if you are using UIKit or SwiftUI:

1. Make a custom `UTType`.
2. Let Xcode know you can import such a type.
3. Associate your app's model to that custom type.
4. Present the share sheet to AirDrop it.
5. Handle the incoming URL from the receiving app.

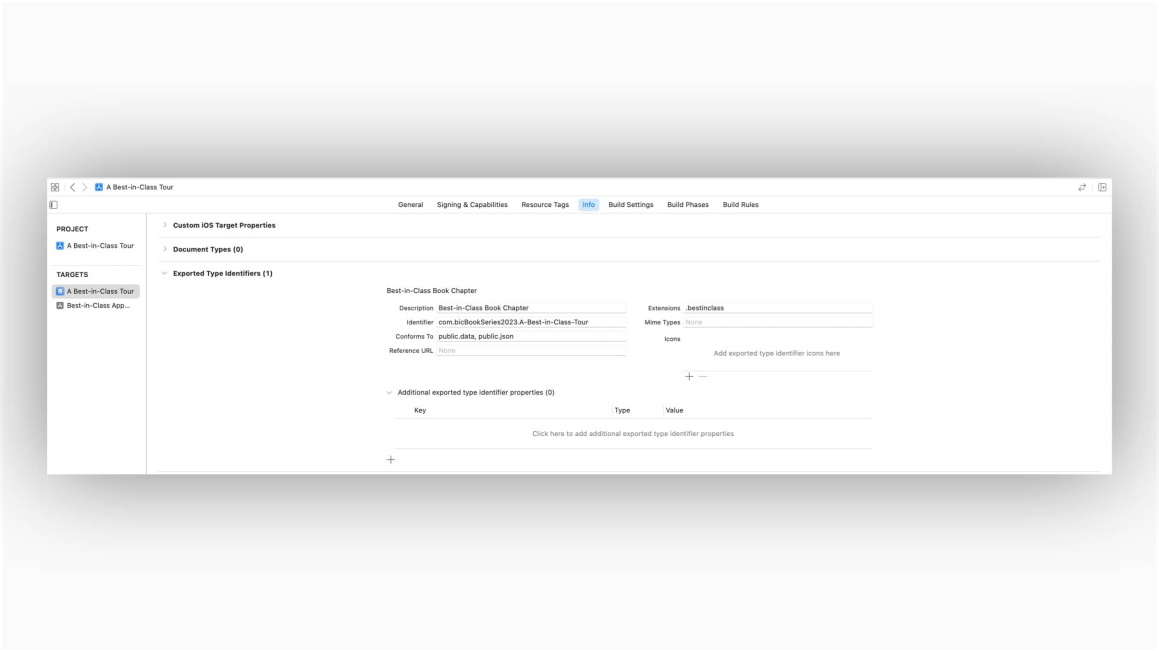
I want to quickly note that you *can* make your own custom model types show up in the share sheet even quicker than that. For example, by conforming your type to `Transferable` and adopting `Codable` in SwiftUI – AirDrop would immediately be an option.

However, in the name of being practical – I want to go through the flow of declaring your type, showing the share sheet and importing it. So, let's start with SwiftUI and then we'll move to UIKit.

First, select the target you'll want to handle the AirDrop operation from and open its Info tab. Here, we'll go to the "Exported Type Identifiers" section and fill out a few fields:

1. **Description**: This is what will show in the interface when performing any file operations, so keep it short and sweet.
2. **Identifier**: You've likely written several of these—a reverse DNS style identifier which the system will use for any data input/output operations. Typically, you can use the same bundle identifier of the target you're adding the custom `UTType` too.
3. **Conforms To**: This is an interesting one – here, you'll identify which *existing* `UTType` identifiers that *your* new type would be compatible with. In our case, we're going to serialize a model to .json using `Codable`, so we'll use `public.data` along with `public.json` here.
4. **Extensions**: Simply put, this is the extension that'll be used with any files created with your new custom `UTType`. You can be descriptive here, as the three letter extension limit hasn't been enforced by any major operating system vendor since 1995.

Here's what it all looks like in our case:

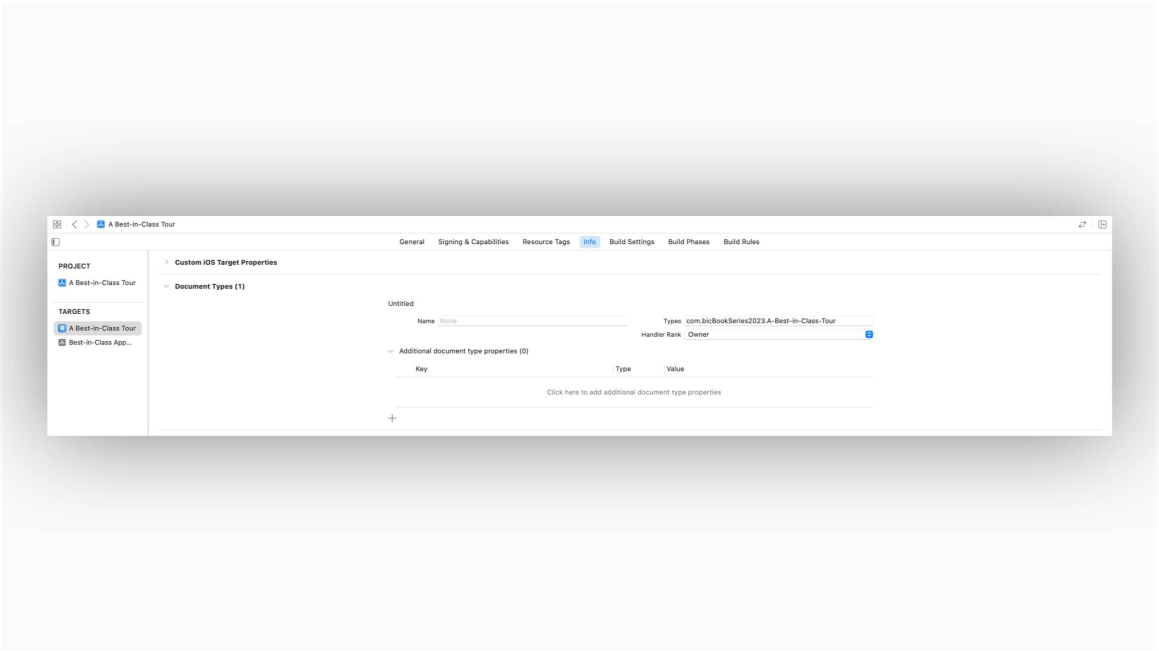


Now that we've identified a type we can export, we're going to tell the system that we can open it as well. By performing the following step, we're telling iOS "Hey, if you see this type I'm about to add – my app can open it, so let me try to." We do this in the "Document Types" section.

This part is a little less work – in fact, there's only two fields we care about:

1. **Types**: Here, we'll use the same identifier we entered from the previous type. Under the hood, this is how iOS will match up the `UTType` instances we've made.
2. **Handler Rank**: Even though the term is a little ambiguous, it simply tells iOS how your app relates to the type you're adding – is it a format you've created (that's the case here) or one you're importing from another app? Since we own the type, we'll go with "Owner" so the system gives us first dibs on importing it.

Another picture to demonstrate:



One last "gotcha" here. In your info.plist–you'll want to add "Supports opening documents in place" to "YES" – and if this warning doesn't go away, just close and reopen Xcode and that typically fixes it.

Now, let's turn to our actual model. In this case, we'll be using the `BookChapter` model I use in the sample project to list out all the chapters across the book series. Here, we need to conform to `Transferable` *and* make an extension from `UTType` to make using our custom type a little easier and less error-prone:

```
// Xcode -> AirDropConformance.swift
extension UTType {
    public static let bestInClassChapter = UTType(exportedAs:
"com.bicBookSeries2023.A-Best-in-Class-Tour")
}

extension BookChapter: Transferable {
    static var transferRepresentation: some TransferRepresentation {
        CodableRepresentation(contentType: .bestInClassChapter)
    }
```

```
}
```

Since the model already adopts `Codable`, we'll use `CodableRepresentation` to easily implement our method of transfer. It's good to point out here that `Transferable` can do *a lot* for you, in addition to our AirDrop use case here. It's used with drag and drop, copy and paste and more.

Also, notice this is how we associate our model with the custom `UTType` we've made. In the `contentType` parameter, which asks for a `UTType`, we've identified that this model represents the type that we created. By passing in the `bestInClassChapter` static type, the custom type and model are now linked together.

With this in place, we leverage `ShareLink` to AirDrop the model:

```
// Xcode -> SwiftUI -> AirDropFig1View.swift

var body: some View {
    List {
        ForEach(Book.allCases) { book in
            Section(book.rawValue) {
                ForEach(book.chapters()) { chapter in
                    HStack {
                        Text(chapter.rawValue)
                        Spacer()
                        // AirDrop support
                        ShareLink(item: chapter, preview:
SharePreview(chapter.rawValue))
                    }
                }
            }
        }
    }
}
```

As for the last step, we've got to handle the AirDrop from your app running on the *receiving* device via the `onOpenURL` modifier, and you're done. Using the URL passed in,

you then decode the model using a decoder as you would for any other .json response:

```
.onOpenURL { url in
    guard url.startAccessingSecurityScopedResource() else {
        return
    }

    do {
        let modelData = try Data(contentsOf: url)
        let model = try JSONDecoder().decode(Model.self,
                                             from: modelData)
    } catch {
        print("Error decoding AirDrop model:\
(error.localizedDescription)")
    }
}
```

For `UIKit`, it's a *bit* different. For starters, you'll use scene-based methods to handle the URL. The only core difference is in the implementation – the setup of your custom type all remains the same. But, when it comes to the code – you'll use a custom object conforming to `UIActivityItemSource`.

Since that protocol requires an object, and we're using a `Struct` with our book model – we'll write a wrapper to handle it:

```
// Xcode -> AirDropConformance.swift

class BookChapterContainer: NSObject, Codable, UIActivityItemSource
{
    let bookChapter: BookChapter

    init(_ chapter: BookChapter) {
        self.bookChapter = chapter
    }
```

```
    // MARK: Activity Item Source

    func activityViewControllerPlaceholderItem(_
activityViewController: UIActivityViewController) -> Any {
        return self.bookChapter.rawValue
    }

    func activityViewController(_ activityViewController:
UIActivityViewController, itemForActivityType activityType:
UIActivity.ActivityType?) -> Any? {
        return self.bookChapter
    }

    func activityViewController(_ activityViewController:
UIActivityViewController, dataTypeIdentifierForActivityType
activityType: UIActivity.ActivityType?) -> String {
        return UTType.bestInClassChapter.identifier
    }
}
```

The most important thing to note is that last delegate method – this is how we wire up our custom `UTType` to the model we're sharing in UIKit much like we did with `Transferable` earlier in SwiftUI.

From there, you simply invoke the activity item controller to AirDrop it:

```
// Xcode -> UIKit -> AirDropFig2ViewController.swift

extension AirDropFig2ViewController {
    func tableView(_ tableView: UITableView, didSelectRowAt
indexPath: IndexPath) {
        tableView.deselectRow(at: indexPath, animated: true)
        guard let chapter = datasource.itemIdentifier(for:
indexPath) else {
            return
        }

        // Create our model
```

```
        let container = BookChapterContainer(chapter)

        // Show the share sheet
        let activityController =
UIActivityViewController(activityItems: [container],
                                 applicationActivities: nil)
        present(activityController, animated: true)
    }
}
```

Finally, we can handle receiving the book chapter and open it:

```
// Xcode -> SceneDelegate.swift

func scene(_ scene: UIScene, openURLContexts URLContexts:
Set<UIOpenURLContext>) {
    guard let firstURL = URLContexts.first,
          firstURL.url.startAccessingSecurityScopedResource() else {
        return
    }

    do {
        let bookData = try Data(contentsOf: firstURL.url)
        let decodedBook = try
JSONDecoder().decode(BookChapterContainer.self, from: bookData)

        guard let rootController = window?.rootViewController as?
GlossaryViewController else {
            return
        }

        rootController.pushBookChapter(chapter:
decodedBook.bookChapter, with: .SwiftUISample)
    } catch {
        print("Error decoding AirDrop model: \
(error.localizedDescription)")
    }
}
```

…and we're done.

To recap, to AirDrop your models you need to register a custom `UTType`, let Xcode know you're aware of that type and can import and export it, associate your model to that type, and then share it and handle it via an incoming URL (no matter if you're using SwiftUI or UIKit).

Tips

Make Types Conform to Codable

As we've seen here, when sharing your models (especially in the case of SwiftUI) – having them conform to `Codable` out of the box makes the process incredibly easy. As a rule of thumb, I *always* make my model layer conform to `Codable` because eventually, I'll need to save it to the file system, handle it with .json or maybe even share it via AirDrop.

When we conform to `Codable`, we can turn around and conform to `Transferable` in essentially one line of code. To refresh your memory:

```
extension BookChapter: Transferable {
    static var transferRepresentation: some TransferRepresentation {
        CodableRepresentation(contentType: .bestInClassChapter)
    }
}
```

This also helps to handle our custom `UTType`, since the "Conforms To:" section becomes easy, too – just say it supports "public.json" and you're done. As a bonus, outside of AirDrop, this means power users could also open up the .json file of your mod-

el and take a peek at it. Of course, this means they could *also* really mess it up — but if they did, it likely wouldn't decode anyway.

Create a Custom a `UIActivityItemSource`

Switching gears to UIKit, again, be aware that you'll need an object. A lot of us are dealing in `Struct` instances at the model layer, but even if you aren't — it's probably a good idea to create a model just for AirDrop and sharing purposes that adopts `UIActivityItemSource`.

Occasionally, these can contain quite a bit of logic to get at the data, interpolate a String of properties create a title or description and more. If you look at our implementation above, even though it's not necessarily complicated, it is a good chunk of code. Best to separate out your logic for specific tasks, which sharing certainly is.

Three Key Takeaways

1. AirDrop allows for frictionless sharing between apps.
2. It's simple to share common types from your app, such as text, images or other URLs.
3. Though, if you'd like to share custom types – be sure to implement a custom Uniform Type Identifier for it.

CloudKit

If you want to provide syncing capabilities to your iOS app – CloudKit is likely the best choice.

I know that's a loaded statement, but I'm going to spend a bit of time here explaining why I think, for most cases, it's a great option. Before that, though, let's talk about what exactly CloudKit *is*.

At its core, CloudKit is a suite of APIs found within the CloudKit framework to enable syncing of your data between a user's devices within Apple's ecosystem. So, if you make a shopping list app, CloudKit can let you enable sync for it, such that all the data someone enters will show up on their iPad, iPhone, Mac, Apple Watch or Apple TV. It can go *anywhere*, so long as "anywhere" is an Apple device.

To get a bit pedantic, I'll also call out the fact that CloudKit also has a Javascript library, so it can certainly be available on the web, too. However, that's not our focus here – but nonetheless useful to be aware of.

Another powerful feature of CloudKit? You can enable any data to be *shared*, which is a massive advantage in terms of cloud providers. Want someone to share that shopping list with the friends or family? CloudKit allows for that, and without much effort.

Next, there's robust telemetry and dashboard reporting available. Apple offers an expansive CloudKit dashboard where you can manage schema changes, look into reporting, usage and issues occurring. It also allows for record creation and management (we'll look more at the concept of records in a moment). While CloudKit would be useful on its own, the accompanying dashboard could be a full-featured product in of itself.

Of course, like any Apple product – security and privacy are core tenants of CloudKit. You can't peek into a user's private database entries, and you can encrypt data as well. Personably identifiable information is tucked away, as it should be. This helps ease everyone's mind that data will be treated properly, both as developers and consumers.

Finally, it's extremely cost-effective. One of the barriers to entry as of this writing to cloud-based programming is the upfront cost. Setup would be another on. CloudKit largely eliminates both of those concerns.

Consider spikes in usage. What happens if your app goes viral and requests go from a few hundred a day to several thousand, or perhaps millions? It's unlikely to hurt your wallet. In fact, *I can't even find pricing* for public database usage as of this writing.

Plus, now you can also transfer CloudKit based apps should one of yours get acquired. Having been through this on the other side, it was a massive headache that thankfully no longer exists.

When you take all of these points into consideration, you can begin to appreciate why when it comes to syncing on iOS — you should probably go with CloudKit. There are simply too many compelling arguments for it, and not many against it.

So, why doesn't it seem like every single iOS app from indie developers are using it? Because, simply put — it's a *pain* to get started with. It's not immediately recognizable in terminology or API surface area, and there are quite a few moving parts. We're going to cover all of them here.

How We'll Tackle CloudKit

That said, I want to include the plan upfront because this chapter will likely work a little differently when compared against previous chapters. In short, I will spend our time here *explaining a lot* about how CloudKit works, and its primary APIs you need to know about to get going.

Why?

As I alluded to above, after working with CloudKit for several years and talking to other engineers about it, it has become evident to me that there is one primary barrier to entry. And, that is that it's simply a big, technical topic at the beginning and thus a little complex to get *started* with. Emphasis on the "started" part because once you get how all the pieces fit together — then building from there becomes many orders of magnitude easier.

Of course, everything is hard until it isn't. But that notion is certainly true of CloudKit — a little know-how and explanations, and you're off and running.

If you just dive in, and you don't know what a zone is, or a container, record, subscription, query — the list goes on, what happens? At that point, trying to implement

CloudKit becomes a maddening dance of documentation hopping that typically leads to moving on to something else. The goal of this chapter is to avoid that.

As such, I know this is a lot of reading and not a lot of doing, but I beg you to *just hang in there* and get through all of this material. Then, you'll be in a good spot to get started on implementing a syncing layer in your work.

Lastly, I have two other reasons I'm going this route. One is that, logistically, setting up CloudKit takes a few steps. Each of them make including code about it in a sample project a headache. I'd overcome that if it were necessary, but it really isn't.

Why?

Apple has incredible, in-depth sample projects of CloudKit hosted on GitHub! They are very useful and extensive, and you can find them all here. Simply search for "CloudKit" under their repositories. This is something I would've paid a lot of money for years ago when I was working with it myself.

Additionally, if you are familiar with my previous app, Spend Stack, I have also made it entirely open source. That means its own custom syncing engine is available for you to use, dissect and get familiar with. That code can be found here – specifically under `SSCloudKitManager`.

So – to recap, our approach here is to read this chapter, digest all of its pieces and then dive into Apple's sample code. Or, if you're feeling brave, Spend Stack's robust Objective-C syncing layer. Then, you'll be in a wonderful position to support CloudKit in your work.

Let's get going!

How it Works

To know CloudKit is to understand how each of its pieces fit together. Then, knowing when, and how, to use them. Let's start with the big picture items, and then we'll look each of their roles.

So let's get going. Here's a primer on the *most important piece* to understanding CloudKit – it's vocabulary.

Status

CloudKit is tied to one user at a time. And, that user is determined by the Apple ID which the device is currently signed in to.

As such, knowing the status of that user, insofar as it pertains to CloudKit, is critical. The status is represented by `CKAccountStatus`, and being an enumeration– it'll report back the status of the user's iCloud account:

```
public enum CKAccountStatus : Int {
    case couldNotDetermine
    case available
    case restricted
    case noAccount
    case temporarilyUnavailable
}
```

To work with CloudKit, you'll want the status to be represented as `available`. If it's not, you're in no position to sync data unless its to a public database:

```
let status = try? await CKContainer.default().accountStatus()
```

Container

Next, you'll need to know about the concept of a container. At a high level – it's precisely what it sounds like anytime you hear the word "container" in programming.

Here, it represents the entire umbrella for which your app, and *only* your app, can operate in as it pertains to CloudKit. In code, it's called `CKContainer`.

A container-based design is useful in that it eliminates cross-contamination of data from other apps within the CloudKit ecosystem. This is good to know, but it's not something you'll have to think about. It's abstracted away from you *unless* you do need to consider it. It is possible to have multiple apps use the same container, in the same way that multiple apps can use App Groups today. For most apps, however, you'll be using a single container.

The container offers access to distinct databases you'll be working with. Let's talk about those next.

Databases

The databases in CloudKit, `CKDatabase`, are where data is stored. Anytime you want to perform C.R.U.D. operations (create, read, update or delete) you'll be working with a database. As such, there are three main databases to know about:

1. **Public**: This database is available to everyone who uses your app. There are no special permissions and data isn't scoped to any particular user.
2. **Private**: This database is scoped to the user's iCloud account they are logged in with. Note that data can still be shared within this database, and in fact – that's how sharing works. More on that later.
3. **Shared**: The shared database is a little tricky to understand at first. If you are operating within it, it's actually *someone else's data* that is being shared with you. And, that data is housed within their private database. As such, the shared database is a peek into another private database. Based on the permissions you've been given, you can either read that data – or perform writes and deletes as well.

Zones

One or more zones (`CKRecordZone`) are housed within a database. Zones are a way to group data logically together. You can be as specific or as general as you see fit. Let's go back to our shopping list app.

You could make a zone for each list, or you could use the default zone for all of them. Either one is viable – but knowing which route to take largely depends on how you want to manipulate that data later. For example, you can delete, and share, entire zones all at once. Or, perhaps you have large data sets – so to cut down on expensive network calls, you would logically group things into zones and only fetch what you need.

By default, publishing data into the public or private database means that it'll land into the default zone. In the case of the public database, there is *only* the default zone, and others can't be added. In the private database, you can add as many as you want. And, for the shared database, its zones are determined by what was shared from a private database – meaning there is no concept of creating zones within one.

Records

A record represents data in your database. Conceptually, a record you work with could be thought of as a row of data in a table within a database. As such, `CKRecord`, will likely be modeled after your custom business models. So, if you've got a `Recipe` struct, then you've likely got a `Recipes` table in CloudKit and will have several records which represent them.

The record is what you will be working with more than anything in CloudKit, and thankfully – they are the easiest to use. A record will be associated with a type and record identifier. If you have an identifier for your model already, you can use it with a `CKRecord` too. In fact, this is what you *should* do.

```
struct Game: Identifiable {
    let id: String
}

// Creating a record...
let record = CKRecord(recordType: "Person",
                      recordID: .init(recordName: person.id))
```

Record Identifiers

Notice how for the `recordID` parameter, it *isn't* a simple string but instead a `CKRecordID`. This is for good reason, and to know why – recall our discussion over zones we just had. A record identifier has to really help CloudKit know "what" and "where", and that's why `CKRecordID` is used. It can store the "what" via its `recordName` (which is the unique identifier, don't let the "name" part fool you) and its `zoneID` represents the "where" part of the equation.

> *Note that in the sample above, the default zone was used for the record ID (that's the default one you get if you don't specify one yourself).*

With all of these pieces working together, you can easily move data where it should be. By using the combination of a certain database (i.e., private, shared or public) *and* which zone within that database it should be stored at – you can be precise. This amount of specificity is part of CloudKit's design, and once you are used to it – I think you'll see that it encourages thoughtful model layer design and relationships, too.

Change Tokens

Next, the change token – `CKServerChangeToken`. Anytime you get updates from CloudKit, it'll funnel down a *change token* along with it. This is essentially a stamp in time that says, "We are up-to-date from this point", and that's incredibly useful.

Imagine if someone was using your app on both an iPad and iPhone. They go on, day-to-day, on their iPhone and manipulating a lot of data. Of course, once they pick up their iPad a week later – the same data should be reflected there as well. And, that's where the change token comes in.

Once you do a fetch, you'll pass a cached server change token. And, in the case of the iPad, CloudKit will recognize that it hasn't synced for a week (or more) and it'll shuttle down all the data that it's missing.

This also means that implementing a "Sync Everything" type of functionality is straightforward – you'd simply pass in a `nil` server change token which tells CloudKit "Give me all the data you have."

Operations

Operations, `CKOperation`, are a way to manipulate batches of records. While there is what Apple calls *convenience* APIs to quickly perform record changes by using the functions off any `CKDatabase`, they are mostly meant for one-off situations:

```
let removedReocrd = try await database.deleteRecord(with:
recordToRemove)
```

The power and flexibility of CloudKit is found within its different operations, and those are what you'll mostly be using. There are different types of operations depending on what kind of operation you're trying to perform. There are operations for creating zones, subscriptions, records and more.

For example, to change a record – you could use the `CKModifyRecordsOperation`:

```
let modifyOp = CKModifyRecordsOperation(recordsToSave: [someRecord],
recordIDsToDelete: nil)

modifyOp.perRecordSaveBlock = { id, result, error in
```

```
        // Handle result
}

// Kicks off the operation
database.add(modifyOp)
```

The flow looks the same no matter the operation, and these types of operations are very configurable. You can determine what should happen for each record, handle errors, assign a quality of service to the operation and more. Then, you add them to a database and the system carries them out.

More recently, Apple has created a more Swift-friendly API for this, too. Now, each database instance has a `modify(X/Y/Z)` function which leverages Swift concurrency and result types. So, the above can be written as:

```
let _ = try await database.modifyRecordZones(saving: [someRecord],
deleting: [])
```

This is virtually the same as above, but you do lose some configurability. Either approach works, and unless you need fine grain control – this new API is much easier to use in my opinion.

Subscriptions

Finally, let's talk about subscriptions, which inherit from `CKDatabaseOperation`. Using subscriptions, you'll get a nice feed of changes in *near* realtime. Apple doesn't market it as such, but it's very close – you'll get changes within a few seconds of them occurring, in my experience.

There are three primary subscriptions you'll use: `CKDatabaseSubscription`, `CKRecordZoneSubscription`, and `CKQuerySubscription`.

Using these, you can subscribe to subscriptions for zone or database changes. Then, anytime something changes within them – you can react to those changes local-

ly. You can use notifications to respond to those changes (and even better, this doesn't require traditional permissions for notifications from the user, as these are "silent" notifications from CloudKit).

Here's what an incoming change would look like:

```
func application(_ application: UIApplication,
didReceiveRemoteNotification userInfo: [AnyHashable : Any],
fetchCompletionHandler
completionHandler: @escaping (UIBackgroundFetchResult) -> Void) {

    if let incomingNotification =
CKNotification(fromRemoteNotificationDictionary: userInfo) {
        switch incomingNotification.notificationType {
        case .query:
            print("Push notification came in for CKQuery, process
updates.")
        case .recordZone:
            print("Push notification came in for record zone,
processing updates.")
        case .database:
            print("Push notification came in for database,
processing updates.")
        default:
            print("Spend Stack - Received unknown CKNotification
type.")
        }
        completionHandler(.newData)
    } else {
        completionHandler(.noData)
    }
}
```

I'd also like to point out there are also queries, and query subscriptions, you can create. But, these are better suited towards the public database. For example, querying a specific piece of data and seeing when it changes. Typically, though, for most indie

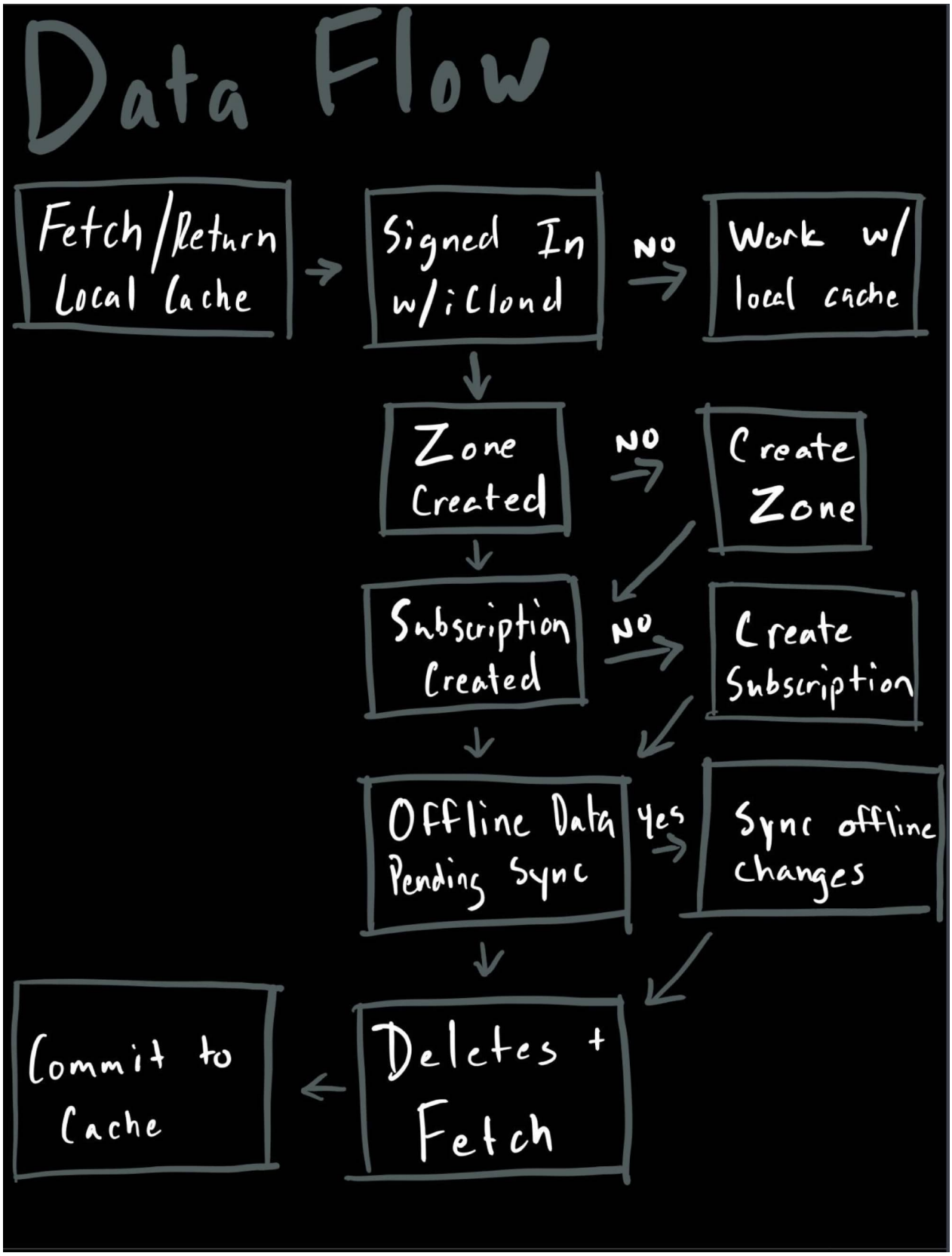
apps – you'll want to know about *every* change, thus – subscriptions are well suited to the task.

Putting It All Together

So, let's recap. I know that can seem like numerous different pieces (and, to be fair – it is!), but you're likely to use them all. To that end:

- You'll start with a container for your app.
- Then, you'll look at the user's status to ensure they are signed in to iCloud (assuming you want to use a private database, if it's only public – then it doesn't matter).
- Then, you'll use specific databases within that container.
- Within those databases, you'll create zones to house logical data, or just stick with the default zone.
- In those zones, you'll add records – which represent your data.
- To change those records, you'll use operations or the new Swift concurrency APIs housed on each database.
- Finally, using subscriptions, you'll react to those changes. When a change comes in, you'll use your server change token to fetch the updates.

To help better visualize how this should work, I've created a diagram that demonstrates what steps your syncing layer should work in:


Data Flow
Fetch/Return Local Cache
Signed In w/iCloud
NO
Work w/ local cache
Zone Created
NO
Create Zone
Subscription Created
NO
Create Subscription
Offline Data Pending Sync
Yes
Sync offline changes
Deletes + Fetch
Commit to Cache

Sharing Data

As mentioned previously, sharing is a major feature of CloudKit. Using it, you can share any data you want with other users of your app. This enables collaboration and other interesting features. To get started, you'll need to enable the `CKSharingSupported` boolean in your app's `Info.plist` first.

There are two different ways to share data in CloudKit app:

1. **Hierarchical**: This method has been around since CloudKit sharing was first released. Here, you associate child records to parent records — and then share the *root* record (i.e., parent) with someone else. Then, they'll receive all the relevant edits.
2. **Zone-based**: This newer method allows you to share an entire zone in one share. So, it could share several records all at once — not just a single parent-child relationship of them.

One isn't any better than the other, and knowing when to use hierarchical versus zone sharing all comes down to the way you've designed your database and the amount of data you want to share.

When you share a hierarchy versus zone, you've got to establish relationships among your records:

```
let parentRecord: CKRecord = getParent()
let childRecords: [CKRecord] = getChildren()

childRecords.forEach {
    $0.setParent(parentRecord)
}
```

Imagine the parent record represents a shopping list, and the child records are the items within it. So, every time you create an item in a list, for CloudKit purposes – you need to establish that the items are a child of the list.

Under the hood, this establishes a `CKReference` which enforces the relationship between the records. If you'd like, you can manually create the reference object yourself to specify what kind of action should occur if the parent record is removed. Stated differently, you can delete one root record and have a cascade-delete occur for any of its children:

```
let deleteAction = CKReference(record: aRecord,
                                    action: .deleteSelf)
```

This is useful to know even if you aren't sharing data. Deleting all child records of a parent is usually what should happen when the parent is deleted.

Next, a zone share is just like it sounds. You'll create a share using a zone's identifier to mark it as shareable:

```
let shoppingListZone = CKRecordZone(zoneName: "ShoppingLists")
let listsShare = CKShare(recordZoneID: shoppingListZone)

let (saves, _) = try? await database.modifyRecords(saving:
[listsShare])

for (recordID, result) in saves {
    // Handle saves
}
```

So, to recap – you can either use one or more hierarchical shares *within* a zone, or just share the entire zone. But, you can't perform both within a zone. For a real-world example, consider Apple's use of these shares.

For Reminders, Notes, and iCloud Drive – hierarchical sharing is used. Each of those structures lends themselves well to parent-child relationships, wherein each individual parent among many might be shared – but likely not all of them. For HomeKit secure video sharing, and HomePod multiuser functions, they use zone sharing since the entirety of each zone is relevant to everyone involved in the share.

All of these shares will end up in a `CKShare` object. It shows who owns the data (i.e., C.R.U.D. operations occur in their *private* database) and who is receiving it (i.e., C.R.U.D. operations happen in their *shared* database). Further, you can set permissions, too.

Want a shopping list to be shared but read-only? That's no problem. Within that share, you'll have participants, `CKShare.Participant`, which give you more information about everyone involved in the share along with their permission levels.

So, how exactly do you share things, though?

That all happens with CloudKit's turnkey solution, `UICloudSharingController`. You present this with a share record and the container it's being used for. Note that this isn't readily available in SwiftUI as a first class `View`, and you'll need to wrap it under a representable. Here's an example from that sample code I mentioned earlier:

```
struct CloudSharingView: UIViewControllerRepresentable {

    // MARK: Properties
    @Environment(\.presentationMode) var presentationMode
    let container: CKContainer
    let share: CKShare

    // MARK: UIViewControllerRepresentable

    func updateUIViewController(_ uiViewController:
UIViewControllerType, context: Context) {}
```

```
    func makeUIViewController(context: Context) -> some
UIViewController {
        let sharingController = UICloudSharingController(share:
share, container: container)
        sharingController.availablePermissions =
[.allowReadWrite, .allowPrivate]
        sharingController.delegate = context.coordinator
        sharingController.modalPresentationStyle = .formSheet
        return sharingController
    }

    func makeCoordinator() -> CloudSharingView.Coordinator {
        Coordinator()
    }

    final class Coordinator: NSObject,
UICloudSharingControllerDelegate {
        func cloudSharingController(_ csc: UICloudSharingController,
failedToSaveShareWithError error: Error) {
            debugPrint("Error saving share: \(error)")
        }

        func itemTitle(for csc: UICloudSharingController) -> String?
{
            "Sharing Sample Code"
        }
    }
}
```

When you present this cloud sharing controller and someone accepts the share, you get the callback `userDidAcceptCloudKitShareWith` in your scene delegate (if you're using SwiftUI – you'll need to use `UIApplicationDelegateAdaptor` to handle it as far as I can tell):

```
// Processing a user's acceptance of a share invitation
```

```
func windowScene(_ windowScene: UIWindowScene,
userDidAcceptCloudKitShareWith cloudKitShareMetadata:
CKShare.Metadata) {
    // Handle share using `CKAcceptSharesOperation` using
    // The passed in metadata
}
```

Tips

Development versus Production

When you are working in a development environment with CloudKit – records and their types can be made on the fly. So if I were to do this…

```
let listItem = CKRecord(recordType: "ListItem")
listItem["name"] = "Foo"

let privateDB = CKContainer.default().privateCloudDatabase
try? await privateDB.save(listItem)
```

…CloudKit would then create this type in your dashboard, found at https://icloud.developer.apple.com/. This is perfect for development because often times you're trying to figure out how all the pieces fit.

Though, when it's time to lock-in changes, you need to deploy your schema to production. This is also easily done in the dashboard, but if you forget to do it – any operation will err quickly in production. Luckily, these changes don't require an update to your app, but they are frustrating to forget.

Quality of Service

Though the new APIs added to quickly modify records are quick and easy to use, sometimes you want the full power of a customizable operation. In such cases, you might find that sometimes they can be a bit long-running. If that happens, it'll help to set an acceptable quality of service to let iOS know it should be more of a priority:

```
let modOp = CKModifyRecordsOperation(recordsToSave: [record])
modOp.qualityOfService = .userInitiated
```

By default, CloudKit will use `.utility`, and that can last up to seven days. Though uncommon – it *can* happen. So, if you're seeing things taking far too long under favorable network conditions, this is something to check.

Custom Sharing Experiences

I mentioned earlier that if you want to kick off a share with the sharing controller, you'll need to route things through `UICloudSharingController`. That is true, but you can also create your own sharing interface, as well. For example, using `CKShare` and the private database directly, you could add participants and give them permissions manually – creating the interface any way you want.

First, you'd need to find the relevant user. The easiest way is by their email (though you could use a phone number or record identifier too, if you knew them):

```
let receiverEmail = "jordan@bestinclassiosapp.com"
let container = CKContainer.default()
let participant = try await
container.shareParticipant(forEmailAddress: receiverEmail)
```

Then, (assuming you have set up a `CKShare` as before) – you could then add them to the share:

```
func add(_ participant: CKShare.Participant,
         permission: CKSharingParticipantPermissionOption,
         toShoppingList share: CKShare) throws {
    participant.permission = permission
    share.addParticipant(participant)

    let privateDB = CKContainer.default().privateCloudDatabase
    try await privateDB.save(share)
}
```

Finally, on the receiver's end – they could confirm it via the `url` property from the `CKShare` they'll receive:

```
func confirmShare(at shareURL: URL) async throws {
    let container = CKContainer.default()
    let metadata = try await container.shareMetadata(for: shareURL)
    try await container.accept(metadata)
}
```

Of course, the cloud sharing controller abstracts all of this away from you and does makes those associations automatically. But, if you'd benefit from a custom solution – there is certainly API available to do it.

Listening to Account Changes

The `CKAccountChanged` notification is critical to start your CloudKit syncing stack if you're using the private database. Always query for it once you start up your stack, and then only begin syncing if you get a good status back:

```
let defaultCenter = NotificationCenter.default
let notification = Notification.Name.CKAccountChanged

defaultCenter.addObserver(self,
                          selector: #selector(handleStatus),
                          name: notification,
```

```
                                object: nil)
```

Subscribe to any changes here, and also check the status when you fire things up:

```
let status = try await CKContainer.default().accountStatus()
```

iCloud Accounts that Aren't Setup

Speaking of account status queries, you might run into `temporarilyUnavailable`. If you hit this, the user has to finish setup in iOS. There's nothing you can do here, apart from possibly give them a hint in your interface that they should visit the Settings app to finish things.

Using iCloud's Unique Identifier

There is a way to leverage iCloud for a unique identifier for a user of your app, even if you have no backend system. I call this "Kinda using CloudKit while only kinda using CloudKit" — and by that, I mean, even if you sync zero data with it, you can still leverage the unique identifier:

```
Task {
    let container = CKContainer.default()
    let userRecord = try await container.userRecordID()
    print("We've got a user ID: \(userID.recordName)")
}
```

Why do this?

Perhaps you just need a notion of a user for lightweight things such as preferences. This can help you know that the same person is using your app across different devices.

There is nothing nefarious you can really do with this identifier anyways. The only thing it can show you is "Someone is signed in to iCloud, and here is a random identifier we gave them that isn't useful for anything else."

CoreData and CloudKit

Be aware that there is an "automatic" sync solution for CoreData and CloudKit. I haven't written about it here because I simply haven't used it, or know enough about CoreData to understand it works. However, it's good to know about — and the API you'd want to look further at is `NSPersistentCloudKitContainer`.

Don't Skip on Local Exports

Even though CloudKit enables sync, don't conflate it as a form of data loss prevention. It's still a user-friendly move to enable a way for someone to get their data out of your app.

For some more implementation details, check out the chapter "Getting Data In and Out" from the user experience book in this series.

Which Subscription to Use

Earlier I noted that there were three different types of subscriptions. For most apps, I think you'll want to stick with `CKDatabaseSubscription`. Unless you're *only* dealing with the default zone, which most apps probably won't.

Here's why: This subscription vends custom zone changes *and,* more importantly, changes from a *shared* database. A record zone subscription can't do this, and neither can a query subscription. Now, of course — some apps don't need this "tell me about every change" type of notifications, but I'd say the majority of them do. If you know

you're not supporting sharing, you can get away with a record zone subscription, *mostly*.

Using the setup I've described here, though, your flow will be like this:

1. Subscribe to database changes from your private *and* shared databases.
2. Get the record zone changes from the database subscription.
3. Then, pass those into a `CKFetchRecordZoneChangesOperation`.
4. Profit! Now you'll get both shared and custom zone record changes.

Three Key Takeaways

1. iCloud is a built-in, no-nonsense framework to provide seamless syncing to your iOS apps.
2. Leveraging the sharing APIs, you can also make sure the same data shows up for multiple people.
3. Remember that when using iCloud, the source of truth is always the server – so reconcile syncing conflicts as best as you can.

CKSyncEngine

In the previous chapter over CloudKit, we talked a lot about the entire CloudKit API. What records are, how zones work, subscriptions – really, the entire API surface area. And, we saw that getting those up and running wasn't *excruciating*, but it also wasn't simple.

It should be easier, and now it is – enter `CKSyncEngine`. If you're going to do CloudKit sync, **this is what you should be using for iOS 17 and up.**

To recap, CloudKit makes syncing *possible* but not necessarily *pleasant*. As developers, we have to juggle change tokens, zones, push notifications, and multiple operations just to keep a few records in sync. And on top of all that, we need to cache tokens and do some more housekeeping to make sure everything stays, well, in sync. To

me CloudKit is, technically speaking, setup quite elegantly, for lack of a better word. But operationally? It's undoubtedly complex.

I think Apple knew this, too. And so in iOS 17, they introduced `CKSyncEngine`. It's a high-level API built on top of CloudKit, designed to abstract away most of the boilerplate. And unsurprisingly — Apple uses it, at least in Freeform and `NSUbiquitousKeyValueStore` as far I know. So, it's battle tested.

Here, we'll go in on a deep overview of how `CKSyncEngine` works, how to adopt it in your app, and what patterns might emerge as you scale it. If you're already familiar with CloudKit basics (and if you read the last chapter, you are!), this will be a drop-in mental model shift. If not, again, check out the previous chapter. It introduces the core types you'll still need to understand (`CKRecord`, `CKContainer`, `CKRecordZone`, etc.).

The Basics

`CKSyncEngine` helps your app do three things:

1. Track what has changed locally and queue it for sync.
2. Sends those changed records to CloudKit for syncing when asked.
3. And, it reacts to changes from other devices, or iCloud itself.

This process is event-driven. You don't "push" records to CloudKit directly. Instead, you simply notify the engine what's changed, and from there it manages the lifecycle. This is a bit different than how it worked before, but it also means there are less steps for you to manage.

In practice, the flow will work like this:

- You instantiate a `CKSyncEngine`, configured with your container and zone.
- You adopt the `CKSyncEngineDelegate` protocol.
- You react to delegate callbacks by sending updated records or applying incoming changes.
- And, you make sure your own model layer is associated with its own record.

Everything else like push notification coordination, token management, subscriptions, and operation scheduling is handled for you. That's a massive load off in contrast to what you previously were responsible for.

CKSyncEngine in Practice

Since we covered so much theory and API in the last chapter, we'll take a different approach here. We'll opt to learn by doing.

On that note, let's imagine a small app where users track their favorite national parks. The data model is a simple `Park` object, and it needs to stay synced across a user's iPhone and iPad. Let's go down our setup list.

Engine Setup

First, we need our engine setup. You'll want to create and store your sync engine early, typically during app launch. In SwiftUI apps, that means wherever your `App` is defined. In this snip, the engine itself is housed within another class or struct to add some other helper functionality:

```
private(set) lazy var engine: CKSyncEngine = {
    let token: ChangeToken? = cachedToken()
```

```
    let config = CKSyncEngine.Configuration(database:
container.privateCloudDatabase,
                                            stateSerialization:
token,
                                            delegate: self)
    return .init(config)
}()
```

Engines are created from a configuration object, which needs a token to represent the sync state, a delegate to manage the changes and a database tow work with. More precisely:

- `stateSerialization`: This is a token you cache. It helps the engine know what's already been synced, so it can be much smarter about what it sends over the wire.
- `delegate`: Here, you pass whatever object conforms to `CKSyncEngine-Delegate`. In our upcoming example, the same class that holds the sync engine conforms to it. More on that shortly.
- `database`: Where the actual CRUD operations are occurring. This **can't** be the `default` or `public` database.

Defining Sync State

To let the engine know what changed locally, you queue a change via its internal state. For example, if you updated a few parks:

```
let recordIDs = parks.map { $0.syncRecord.recordID }
let changes = recordIDs.map { .saveRecord($0) }
```

```
engine.state.add(pendingRecordZoneChanges: changes)
```

The above code works for edits and creates, making it an easy API to adopt. And, if you deleted any:

```
let deletions = recordIDs.map { .deleteRecord($0) }
engine.state.add(pendingRecordZoneChanges: deletions)
```

Remember, these actions do **not** immediately trigger a sync. Instead, the engine schedules syncing according to system heuristics – battery, connectivity, usage patterns, etc. That's intentional, and it works much like other similar frameworks do, too. Think `WidgetKit`, where you *request* an update to a timeline instead of explicitly performing one.

CKSyncEngineDelegate

The sync engine delegate is where a lot of the "magic" of the framework happens. Here, we'll queue up changes, be notified of incoming ones and more. This is where most of your CloudKit logic lives. Here, you've got to implement two main things:

1. Provide data to send.
2. Respond to changes from iCloud.

Let's see how those work.

Zone Batch Changes

This method is called when the engine is ready to send data to iCloud. Your job is to take the list of pending changes (that you previously queued), match them to your lo-

cal data, and finally send them over over to CloudKit, complete with their updated `CKRecord` instances:

```
func nextRecordZoneChangeBatch(
  _ context: CKSyncEngine.SendChangesContext,
  syncEngine: CKSyncEngine
) async -> CKSyncEngine.RecordZoneChangeBatch? {
  let changes = syncEngine.state.pendingRecordZoneChanges.filter
{ context.options.scope.contains($0) }

  let batch = await
CKSyncEngine.RecordZoneChangeBatch(pendingChanges: changes)
{ recordID in
    guard let park = store.park(for: recordID.recordName) else {
      syncEngine.state.remove(pendingRecordZoneChanges:
[.saveRecord(recordID)])
      return nil
    }

    park.updateCKRecord()
    return park.syncRecord
  }

  return batch
}
```

So, in that code above:

- We're being asked to handle a sync. The engine knows something changed, and now it's our turn to turn that into actual CKRecords (representing those changes). The easiest way for me to understand this was to remember that this is how *other* devices pick up the synced data's state.
- Remember, "change" here is sort of all-encompassing: any CRUD operation is fair game. In fact, that's what this is meant for.

- Then, you filter out pending changes based on scope, build a batch, and update the `CKRecord` for each model.

No matter how you handle it, though, you're always in full control. You can choose what to send, and what to omit or otherwise changes things as you see fit.

Handling Events

Now, the next part is handling changes coming to *us* from *other* devices. This works by a catch-all notification system of a sync engine. You'll get notified of three main things:

1. Incoming changes from iCloud.
2. New sync tokens (which you must persist).
3. Account state changes (like the user signing out).

Here's what a "real" implementation could look like:

```
func handleEvent(_ event: CKSyncEngine.Event, syncEngine:
CKSyncEngine) async {
  switch event {
  case .stateUpdate(let state):
    store.saveSyncToken(state.stateSerialization)
  case .fetchedRecordZoneChanges(let changes):
    apply(changes: changes)
  case .accountChange(let change):
    // This is also called when the sync engine is fired up
    // So, you should create your zone here too.
    handleAccountChange(change)
  default:
    break
  }
}
```

Starting with `stateUpdate`, every sync products a new state token. You just need to save it. This helps the sync engine avoid unnecessary work, which means it will skip sync operations for things that have already done so. Another way to think about this? This is how you pull off delta syncing.

Fetching record zone changes is a big one. This edits from *another* device that you're reacting to (contrast that with the previous method, which was us going in the other direction). Here, we need to loop through those changes, convert each `CKRecord`'s data over into our local model layer and save everything locally:

```
func apply(changes: CKSyncEngine.Event.FetchedRecordZoneChanges) {
  var parks = store.load()

  for mod in changes.modifications {
    if let i = parks.firstIndex(where: { $0.id ==
mod.record.recordID.recordName }) {
      parks[i].updateWith(record: mod.record)
    } else {
      parks.append(Park(record: mod.record))
    }
  }

  for deletion in changes.deletions {
    parks.removeAll { $0.id == deletion.recordID.recordName }
  }

  store.save(parks)
}
```

So basically — cloud data came in, we applied those changes and persisted them locally. From completeness, here's all you can respond to here:

```
func handleEvent(_ event: CKSyncEngine.Event, syncEngine:
CKSyncEngine) async {
    print("☁️ sync engine event came in, processing...")
```

```
        switch event {
        case .stateUpdate(let stateUpdate):
            print("Sync token.")
        case .accountChange(let accountChange):
            print("Account change.")
        case .fetchedDatabaseChanges(let fetchedDatabaseChanges):
            print("Database changes.")
        case .fetchedRecordZoneChanges(let fetchedRecordZoneChanges):
            print("Record zone changes.")
        case .sentRecordZoneChanges(let sentRecordZoneChanges):
            print("Sent record zone changes.")
        case .didSendChanges,
             .willFetchChanges,
             .willFetchRecordZoneChanges,
             .didFetchRecordZoneChanges,
             .didFetchChanges,
             .willSendChanges,
             .sentDatabaseChanges:
            print("CKSyncEngine event: - \(event)")
            break
        @unknown default:
            print("Unknown CKSyncEngine event: \(event)")
        }
}
```

And finally, `.accountChange` is probably self-explanatory. Regardless, you'll get anytime anything with the account happens. That could be a sign out, in, permission change or something else along those lines. Of note, though, this gets called when you first activate a sync engine. So, that makes it a great place to create your custom zone, too:

```
mySyncEngine.state.add(pendingDatabaseChanges:
[ .saveZone(CKRecordZone(zoneName: MyZone))])
```

A Practical Example

Let's say we wanted to make a very simple "National Parks" app that simply saved our favorite park's names. In practice, it's just a model with a String property. But, it helps you see how it all fits – then, you can expand on it.

Here's the minimal parts at play might be:

1. `Park`: Your CloudKit-backed data model, it's in a local cache and is sync-aware.
2. `LocalStore`: Simple local persistence layer to cache data.
3. `SyncEngine`: Full sync engine setup, complete with delegate conformance and the sync logic.

To start, our model might look like this:

```
import SwiftUI
import CloudKit

class Park: NSObject, NSSecureCoding, Identifiable {
    static var supportsSecureCoding: Bool = true

    var id: String
    var name: String

    // MARK: CloudKit Properties
    static let zoneName = "Parks"
    static let recordType: CKRecord.RecordType = "Park"
    private var zoneID: CKRecordZone.ID?
    private var recordID: CKRecord.ID?
    private(set) var syncRecord: CKRecord?
    private var syncRecordData: NSData?

    func updateCKRecord() {
        syncRecord?["name"] = name
```

```
    }

    func update(with record: CKRecord) {
        name = record["name"] as? String ?? ""
        syncRecord = record
    }

    // To prep for CloudKit
    init(name: String) {
        self.name = name
        self.id = ProcessInfo.processInfo.globallyUniqueString

        // CloudKit
        let zoneID = CKRecordZone.ID(zoneName: Park.zoneName)
        let recordID = CKRecord.ID(recordName: id, zoneID: zoneID)
        let record = CKRecord.init(recordType: Park.recordType,
recordID: recordID)
        record.encryptedValues["textName"] = name

        self.syncRecordData = nil
        self.syncRecord = record
        self.zoneID = zoneID
        self.recordID = recordID
    }

    // For regular use
    init(name: String, id: String) {
        self.name = name
        self.id = id
    }

    // For syncs coming from other devices
    init(record: CKRecord) {
        id = record.recordID.recordName
        name = record["textName"] as? String ?? ""
        syncRecord = record
        let archiver = NSKeyedArchiver(requiringSecureCoding: true)
        record.encodeSystemFields(with: archiver)
        syncRecordData = archiver.encodedData as NSData
```

```
    }

    // MARK: NSCoding

    func encode(with coder: NSCoder) {
        coder.encode(name, forKey: "name")
        coder.encode(id, forKey: "id")

        if let record = syncRecord {
            let archiver = NSKeyedArchiver(requiringSecureCoding:
true)
            record.encodeSystemFields(with: archiver)
            syncRecordData = archiver.encodedData as NSData
            coder.encode(syncRecordData, forKey: "syncRecordData")
        }
    }

    required init?(coder: NSCoder) {
        guard let name = coder.decodeObject(of: NSString.self,
forKey: "name") as String?,
              let id = coder.decodeObject(of: NSString.self, forKey:
"id") as String? else {
            return nil
        }

        self.name = name
        self.id = id

        // CloudKit
        let zoneID = CKRecordZone.ID(zoneName: Park.zoneName)
        let recordID = CKRecord.ID(recordName: id, zoneID: zoneID)
        let syncRecordData = coder.decodeObject(of: NSData.self,
forKey: "syncRecordData") as NSData?

        self.zoneID = zoneID
        self.recordID = recordID
        self.syncRecordData = syncRecordData
        if let data = syncRecordData as? Data {
            do {
```

```
                let unarchiver = try
NSKeyedUnarchiver(forReadingFrom: data)
                unarchiver.requiresSecureCoding = true
                self.syncRecord = .init(coder: unarchiver)
            } catch {
                print(error.localizedDescription)
            }
        }
    }
}
```

And, the store:

```
import Foundation
import Observation
import Combine
import CloudKit

@Observable
class ParkLocalStore {
    static let parksKey: String = "AllParks"
    private(set) var parks: [Park] = []
    private let defaults: UserDefaults = .standard
    private var subs: [AnyCancellable] = []

    // MARK: CloudKit Sync Engine
    let cloudSync: ParkSyncEngine = .init()

    init() {
        fetchParks()

        NotificationCenter.default
            .publisher(for: .parkRemovePublishedParks)
            .sink { [weak self] _ in
                self?.parks.removeAll()
            }
            .store(in: &subs)

        NotificationCenter.default
```

```
            .publisher(for: .cloudSyncChangesFinished)
            .sink { [weak self] _ in
                self?.fetchParks()
            }
            .store(in: &subs)

        print("Store has \(self.parks.count) parks")
    }

    func save(park: Park) {
        if parks.firstIndex(where: { $0.id == park.id }) == nil {
            parks.append(park)
        }

        saveParks()
    }

    func remove(park: Park) {
        guard let parkIndex = parks.firstIndex(where: { $0.id ==
park.id }) else {
            return
        }

        let parkToRemove: Park = parks[parkIndex]
        parks.remove(at: parkIndex)

        Task {
            await cloudSync.queueDeletions(for: [parkToRemove])
        }

        saveParks(queueChanges: false)
    }

    private func fetchParks() {
        self.parks = ParkLocalStore.lastCachedParks()
    }

    private func saveParks(queueChanges: Bool = true) {
        ParkLocalStore.save(parks)
```

```
        if queueChanges {
            Task {
                await cloudSync.queueSaves(for: parks)
            }
        }
    }
}

// MARK: Simple local persistency.

extension ParkLocalStore {
    static func lastCachedParks() -> [Park] {
        let defaults = UserDefaults.standard
        guard let parksData = defaults.data(forKey:
ParkLocalStore.parksKey) else {
            return []
        }

        do {
            let decodedParks = try
NSKeyedUnarchiver.unarchivedArrayOfObjects(ofClass: Park.self, from:
parksData)
            return decodedParks ?? []
        } catch {
            print(error.localizedDescription)
            return []
        }
    }

    static func removeAllCachedParks() {
        let defaults = UserDefaults.standard
        defaults.removeObject(forKey: ParkLocalStore.parksKey)

NotificationCenter.default.post(name: .parkRemovePublishedParks,
object: nil)
    }

    static func save(_ parks: [Park]) {
```

```
        do {
            let defaults = UserDefaults.standard
            let encodedData: Data = try
NSKeyedArchiver.archivedData(withRootObject: parks,
requiringSecureCoding: true)
            defaults.set(encodedData, forKey:
ParkLocalStore.parksKey)
        } catch {
            print(error.localizedDescription)
        }
    }
}

// MARK: Extensions for Combine.

extension NSNotification.Name {
    static let parkRemovePublishedParks: NSNotification.Name
= .init(rawValue: "parkRemovePublishedParks")
    static let parkCloudSyncChangesFinished: NSNotification.Name
= .init(rawValue: "parkCloudSyncChangesFinished")
}
```

And, the sync engine itself:

```
import CloudKit

final actor ParkSyncEngine: NSObject {
    static let shared = ParkSyncEngine()

    private let container = CKContainer(identifier:
"iCloud.com.example.FavoriteParks")
    private var engine: CKSyncEngine!
    private var syncToken: CKSyncEngine.State.Serialization?

    override init() {
        super.init()
        let config = CKSyncEngine.Configuration(database:
container.privateCloudDatabase,
```

```
                                                    stateSerialization:
Self.loadToken(),
                                                    delegate: self)
        self.engine = .init(config)
    }

    private func saveToken(_ token:
CKSyncEngine.State.Serialization) {
        if let data = try? JSONEncoder().encode(token) {
            UserDefaults.standard.set(data, forKey: "sync-token")
        }
    }

    private static func loadToken() ->
CKSyncEngine.State.Serialization? {
        guard let data = UserDefaults.standard.data(forKey: "sync-
token"),
              let token = try?
JSONDecoder().decode(CKSyncEngine.State.Serialization.self, from:
data) else {
            return nil
        }
        return token
    }

    func queueSaves(for parks: [Park]) {
        let changes = parks.compactMap
{ $0.syncRecord?.recordID }.map
{ CKSyncEngine.PendingRecordZoneChange.saveRecord($0) }
        engine.state.add(pendingRecordZoneChanges: changes)
    }

    func queueDeletions(for parks: [Park]) {
        let deletions = parks.compactMap
{ $0.syncRecord?.recordID }.map
{ CKSyncEngine.PendingRecordZoneChange.deleteRecord($0) }
        engine.state.add(pendingRecordZoneChanges: deletions)
    }
```

```
    func setupZone() {
    let parks = ParkLocalStore.lastCachedParks()
        let recordIDs = parks.compactMap { $0.syncRecord?.recordID }
        let changes: [CKSyncEngine.PendingRecordZoneChange] =
recordIDs.map{ .saveRecord($0) }
        engine.state.add(pendingDatabaseChanges:
[ .saveZone(CKRecordZone(zoneName: Quote.zoneName)) ])
        engine.state.add(pendingRecordZoneChanges: changes)
    }
}

extension ParkSyncEngine: CKSyncEngineDelegate {
    func nextRecordZoneChangeBatch(_ context:
CKSyncEngine.SendChangesContext, syncEngine: CKSyncEngine) async ->
CKSyncEngine.RecordZoneChangeBatch? {
        let scope = context.options.scope
        let parks = ParkLocalStore.lastCachedParks()
        let changes =
syncEngine.state.pendingRecordZoneChanges.filter
{ scope.contains($0) }

        let batch = await
CKSyncEngine.RecordZoneChangeBatch(pendingChanges: changes)
{ recordID in
            guard let park = parks.first(where: { $0.id ==
recordID.recordName }) else {
                syncEngine.state.remove(pendingRecordZoneChanges:
[.saveRecord(recordID)])
                return nil
            }
            park.updateCKRecord()
            return park.syncRecord
        }

        return batch
    }

    func handleEvent(_ event: CKSyncEngine.Event, syncEngine:
CKSyncEngine) async {
```

```
        switch event {
        case .stateUpdate(let update):
            saveToken(update.stateSerialization)
        case .fetchedRecordZoneChanges(let changes):
            var parks = ParkLocalStore.lastCachedParks()

            for mod in changes.modifications {
                if let i = parks.firstIndex(where: { $0.id ==
mod.record.recordID.recordName }) {
                    parks[i].update(with: mod.record)
                } else {
                    parks.append(Park(record: mod.record))
                }
            }

            for deletion in changes.deletions {
                parks.removeAll { $0.id ==
deletion.recordID.recordName }
            }

            ParkLocalStore.save(parks)
        case .accountChange(let change):
            switch change.changeType {
            case .signIn:
            setupZone()
            default:
                print("Account change: \(change.changeType)")
            }
        default:
            break
        }
    }
}
```

Again, how closely you correlate your CloudKit syncing code to a model is up to you. Here, they are closely coupled – which certainly has its disadvantages. But, on the other hand, it makes the whole thing a bit easier to manage.

On a similar note, you'll quickly find that same tension between your local cache and the syncing code. Do you keep them closely coupled? Use a delegate? Notifications? Weak references? It's up to you and the way your app is architected.

Tips

Record IDs Should Match Model Identifiers

Each `CKRecord` has a `CKRecordID` associated to it. The `CKRecordID` instance has a property called `recordName`. This value should correspond to your model's identifier, so your server instance and your local instance will match up correctly.

You Still Need to Create a Zone

Even though `CKSyncEngine` manages change tokens and subscriptions for you, it does not create your record zone. You'll need to do that yourself, and make sure it's created on every device the first time the app runs. Do this early, and as I mentioned above, a great spot is in `accountChanged`.

Use Encrypted Values When Possible

Just like with vanilla CloudKit, `CKRecord` supports both `record["key"]` and `record.encryptedValues["key"]`. The latter ensures values are stored encrypted at rest, even on Apple's servers.

This is ideal for storing personal data that doesn't need to be queried via `NSPredicate` — names, toggles, notes, etc.

```
record.encryptedValues["isFavorite"] = true
```

Encrypted values require zero additional setup and can be read and write just like standard keys.

Sync Timing is Heuristic-Based

You don't control when sync occurs. Once you queue a change, the system waits for favorable conditions to send it. Using iOS' heuristics, it'll figure out the ideal time to kick it off. In practice, that means it considers things like battery state, network conditions, user activity and other similar data points..

Save Sync State Carefully

When you receive a `.stateUpdate` in your delegate, persist the `stateSerialization` token somewhere durable (UserDefaults, a local file, or keychain).

Without this, `CKSyncEngine` will re-fetch your entire record zone on next launch.

```
case .stateUpdate(let state):
    let data = state.stateSerialization
    try? data.write(to: syncTokenURL)
```

Don't skip this, it keeps your sync lightweight and fast.

Deletion Is Explicit

Unlike CoreData merge policies, `CKSyncEngine` won't infer intent. If you delete a record, you must explicitly mark it via a `.deleteRecord(recordID)` call. This is by design because the sync state is declarative, not observational.

However, recall that saves and updates all work through the same update record function.

You Can Manually Flush Sync

While most developers should simply let the engine run in the background, it is possible to manually trigger sync for pending changes:

```
engine.fetchChanges()
engine.sendChanges()
```

You might use this in pull to refresh situations, or something within your app's settings to try and trigger a sync if things are off.

Unsupported Zones

The default record zone is not supported with `CKSyncEngine`. You must use a custom zone you create and manage. This simplifies token tracking and scoping for the engine.

The public database is also not meant for `CKSyncEngine`. So remember, either a custom zone that's private, or the `.shared` database will do. The public and defaults zones will not work, to wit — I believe it throws an error if you attempt to do so. Or worse, fails silently.

Handle Account Changes

While we setup our zone earlier during account change, there are other things you should watch out here for, too. Always watch for `.accountChange` events. When the user signs out of iCloud (or switches accounts), your engine should clear state and prepare for reinitialization.

```
case .accountChange(let change):
    syncEngine = nil
    // Clear caches, reset store, etc.
```

Mapping Records to Models

One of the most important design decisions when working with a sync engine is how you model the relationship between your local data and its CloudKit record.

In my work, I've embedded the `CKRecord` directly into my model. I. store and maintain the record alongside local data so I can update it before sending, and then replace it when new data arrives.

In the park example, that's the approach used. It offers a few advantages;

- You can archive and cache the `CKRecord` using `NSKeyedArchiver` easily, allowing fast cold starts without re-fetching from the cloud.
- In my opinion, your sync logic becomes simpler because there is no need to rehydrate full records on the fly when syncing.
- Updates flow naturally because your model is always holding the source of truth.

This happens most often during initialization. When you receive a CKRecord from iCloud — either through `handleEvent(_:)` or batch sync — you can immediately archive its metadata to disk like so:

```
let archiver = NSKeyedArchiver(requiringSecureCoding: true)
record.encodeSystemFields(with: archiver)
syncRecordData = archiver.encodedData as NSData
```

Later, when the app relaunches or the record needs to be reconstructed, you can reverse the process using `NSKeyedUnarchiver`. This saves you from having to round-trip to CloudKit just to rebuild a `CKRecord`.

The tradeoff? There's quite a bit of coupling here, as your model becomes *very* CloudKit aware. But for sync-focused apps, that's usually worth it (to me, at least). If you adopt this pattern, just be sure to:

- Keep `CKRecord` fields up to date before sending (like `.update(text:updateRecord:)` does in the park app example).
- Handle nil or missing records gracefully (e.g. new unsynced objects).
- Serialize with `NSKeyedArchiver` using secure coding.

Source of Truth

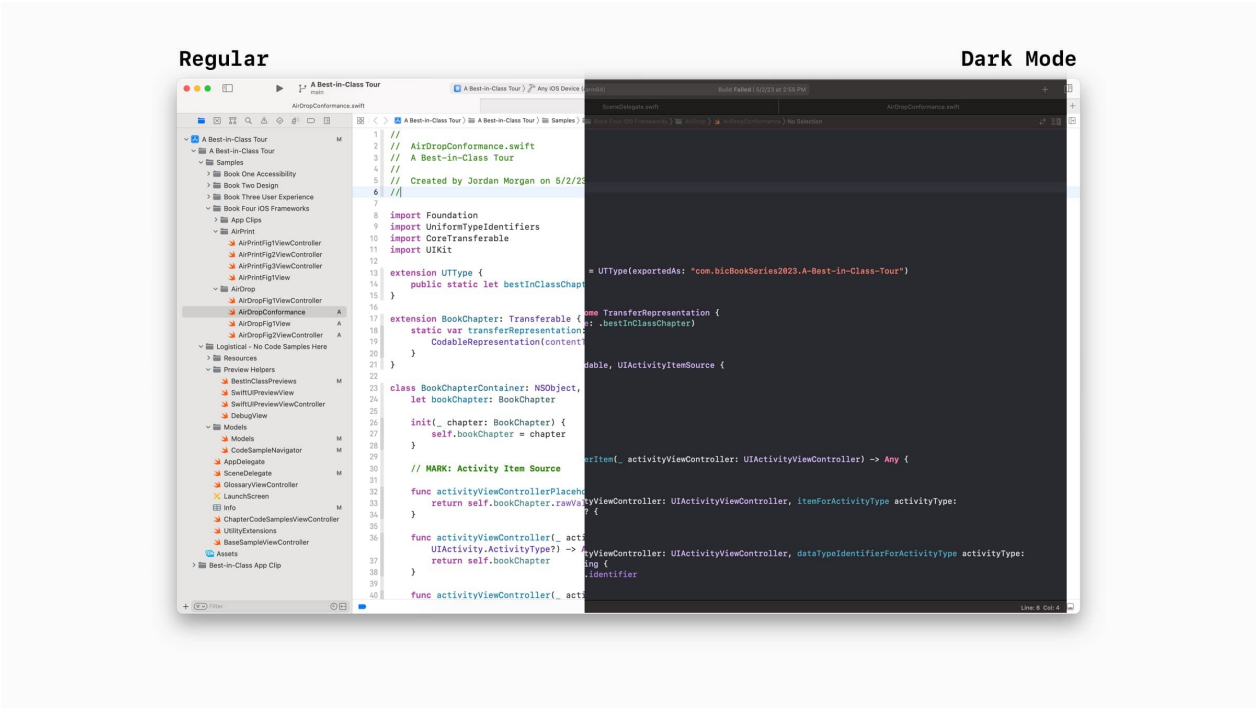
Use `CKSyncEngine` to coordinate between devices, and remember that when you do – the server is now the source of truth. Whatever is on the server should be treated as the "correct" data, and really – this is true of any sync service. Whether you use something else or go with CloudKit, make sure you keep your local data hydrated from the server.

Three Key Takeaways

1. `CKSyncEngine` turns sync into a declarative task: describe what changed, and then react to what comes in.
2. You no longer manually manage change tokens or push subscriptions.
3. It's the easiest and most reliable way to sync private CloudKit data today.

Dark Mode

In case you're not aware of what exactly Dark Mode is, it's a color setting that changes the entire interface to simply be darker:



Users can change it at their leisure, or they can follow a schedule which iOS controls on their behalf (essentially turning Dark Mode on around dusk). Dark Mode, aside from simply being an aesthetic preference, is also an accessibility aid. Some folks' vision can be sensitive to brighter colors, and Dark Mode can make their device more useable.

Even though we refer to it as Dark Mode, as does Apple's official documentation, what we're really talking about is a preference for how bright or dark the operating system's user interface is. And when that choice is made, apps should also respect it.

For iOS, Dark Mode arrived natively on iOS 13, but implementations had existed within apps long before them. Even now, it's not uncommon to find a "Theme" setting within most apps to toggle the color palette. These days, we should rely on the system APIs to support Dark Mode, and the good news is that it's probably the *easiest* system feature to support.

In fact, if you're familiar with the concept of semantic colors – which we covered in the "Colors" chapter in the design book in this series, then you're practically done. Of course, like many of Apple's frameworks, there is more under the surface if you look hard enough. To that end, we'll cover the basics of Dark Mode and go a bit deeper here.

How it Works

Whether Dark Mode is on is represented by a *user interface style*. In UIKit, that's `UIUserInterfaceStyle`:

```
let style = self.traitCollection.userInterfaceStyle
```

Specifically, `UITraitCollection` houses all sorts of useful information about the state of the display, its traits and more. You'll find scale factors of the device's screen, size classes which help adapt your interface on the fly and of course – the interface style representing the status of dark mode.

To be notified when it changes, you can override `override func traitCollectionDidChange(_ previousTraitCollection: UITraitCollection?)`. In this example, we simply update a label which reads out which user interface style is active – "regular" (i.e., light mode) or dark mode:

```
// Xcode -> UIKit -> DarkModeFig1ViewController.swift

class DarkModeFig1ViewController: BaseSampleViewController {
    private let styleLbl: UILabel = {
        let lbl = UILabel(frame: .zero)
        lbl.font = UIFont.preferredFont(forTextStyle: .headline)
        lbl.translatesAutoresizingMaskIntoConstraints = false
        lbl.textColor = .label
```

```
        lbl.textAlignment = .center
        return lbl
    }()

    override func viewDidLoad() {
        super.viewDidLoad()
        view.addSubview(styleLbl)

        NSLayoutConstraint.activate([
            styleLbl.centerXAnchor.constraint(equalTo:
view.centerXAnchor),
            styleLbl.centerYAnchor.constraint(equalTo:
view.centerYAnchor),
            styleLbl.widthAnchor.constraint(equalTo:
view.safeAreaLayoutGuide.widthAnchor),

styleLbl.heightAnchor.constraint(greaterThanOrEqualToConstant: 64)
        ])

        updateLbl()
    }

    private func updateLbl() {
        styleLbl.text = traitCollection.userInterfaceStyle == .light
? "Regular" : "Light"
    }

    override func traitCollectionDidChange(_
previousTraitCollection: UITraitCollection?) {
        updateLbl()
    }
}
```

In SwiftUI, instead of a catch-all trait collection, you simply use the environment. There, you'll find a *color scheme* which represents the same thing as `UIUserInterfaceStyle` in UIKit:

```
// Xcode -> SwiftUI -> DarkModeFig1View.swift
```

```
struct DarkModeFig1View: View {
    @Environment(\.colorScheme) private var scheme

    var body: some View {
        ZStack {
            if scheme == .light {
                Text("Light Mode Enabled")
            } else {
                Text("Dark Mode Enabled")
            }
        }
    }
}
```

It's possible to set the `colorScheme` yourself, but typically that's not the route you want to go. The reason? It'll change the scheme only on the direct view where the environment variable was injected, along with its child view. Though, any views that are higher up in the view tree won't respect the color scheme. In short, it'll lead to some unexpected results.

If you do need to change it in SwiftUI, the `preferredColorScheme(_:)` is what you want. It'll propagate the value up throughout the view tree. Aside from simply the color scheme, you may also be interested in the `colorSchemeContrast` environment variable. If it's `.increased`, you should consider that alongside any other color choices you make to ensure the colors are easy to read and view.

Testing dark mode is also a trivial matter, as Xcode previews allows you to turn it on via the "Device Settings" button. If you're using UIKit, you can still do the same thing by using the "Environment Overrides" button. Of course, on a real device – the fastest route is to toggle it via Control Center (or you could even ask Siri!).

Semantic Colors

Semantic colors are critical to support Dark Mode. By their nature, they will automatically adapt to it. So long as use the ones iOS has provided, they'll change to sensible values for you.

Colors (both `UIColor` and `Color`) that begin with `system` all adapt alongside Dark Mode and describe their purpose as well. For example, `.systemBackground`, `.systemLabel` and more. This extends even to basic colors. Using `.systemRed` adapts its color based on the user interface style, too.

Leaning on iOS' semantic colors is a good idea — and one big reason is that the concept extends to system controls. Look, we all love making custom controls — I certainly sprinkle a few in every project. But, it is what it is. System controls *always* adapt for you according to the user interface style. You likely have to ensure you're using semantic colors on anything custom you create on your own (and the rights ones for the correct purpose, at that).

For example, SwiftUI pops back and forth things like the background fill, and labels switch their font color too. Try running the demo app, and toggling dark mode back and forth to see it live. This is all displaying the power of semantic colors in action. It's less about saying "Use a black color for text" and instead thinking more in the terms of "No matter if Dark Mode is on or off, this is the best color for typography."

Custom Colors

Of course, most of us have to use our own colors eventually. The easiest way to still support Dark Mode is by leveraging asset catalogs. When you do this, you can specify different colors to use for both dark and light mode:

Any Appearance is used for light mode.
Dark, is well, for dark mode.



Aside from using asset catalogs, it's also possible to take a programmatic approach. For example, in UIKit, you can use `UIColor.init(dynamicProvider: @escaping (UITraitCollection) -> UIColor)` to vend a color based off the trait collection.

Here, we purposely do the opposite of what Dark Mode typically does – we show a white background when Dark Mode is on, and vice versa:

```
// Xcode -> UIKit -> DarkModeFig2ViewController.swift

class DarkModeFig2ViewController: BaseSampleViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        view.backgroundColor = .adaptiveFill
    }
}

extension UIColor {
    static var adaptiveFill: UIColor {
        UIColor.init { traitCollection in
            if traitCollection.userInterfaceStyle == .dark {
```

```
                return .white
            } else {
                return .black
            }
        }
    }
}
```

In SwiftUI, the same concept applies – though you'd query the aforementioned `colorScheme` (or just use a `UIColor` passed into `Color` itself):

```
@Environment(\.colorScheme) private var scheme

Text("Hey there!)
    .foregroundColor(scheme == .dark ? .white : .black)
```

No matter the case, strive for readability and always consider accessibility. At a minimum, the contrast ratio between colors should be no lower than 4.5:1. For custom foreground and background colors, you want to hit a ratio of 7:1, and that rings even more true for text that might run a bit smaller (such as captions or any kind of footnote).

For tips on how to test this easily, refer to the Accessibility book in this series for tips.

Tips

Leverage Materials

The system materials (or, as I like to call them, all the views that blurs things) work well with dark mode out of the box. Aside from that, it also encourages a core tenet of Apple's design language, specifically in terms of *depth*.

For example, sheets presenting modally from the bottom of the screen look especially slick when using materials. This is the same design pattern that Maps uses. In SwiftUI, it's easy to use materials to get this look:

```
VStack {
    content
}
.backgroundStyle(.thinMaterial)
```

In UIKit, you can achieve the same thing by using `UIVisualEffectView`:

```
let blur = UIBlurEffect(style: .light)
let fxView = UIVisualEffectView(effect: blur)
view.addSubview(fxView)
```

When it comes to text and materials, using *vibrancy* can ensure things still stay legible. Here, the label over the image will maintain legibility by using a vibrancy effect:

```
// Xcode -> UIKit -> DarkModeFig3ViewController.swift

class DarkModeFig3ViewController: BaseSampleViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

        let img = UIImage(named: "Healthy Food")
        let imgView = UIImageView(frame: view.bounds)
        imgView.image = img
        imgView.contentMode = .scaleAspectFit

        view.addSubview(imgView)

        let blur = UIBlurEffect(style: .prominent)
        let vibrancyFX = UIVibrancyEffect(blurEffect: blur,
                                          style: .label)
        let fxView = UIVisualEffectView(effect: vibrancyFX)
        fxView.frame = view.bounds
```

```
        let label = UILabel()
        label.text = "Healthy Food"
        label.textAlignment = .center
        label.font = .systemFont(ofSize: 46, weight: .black,
width: .expanded)
        label.sizeToFit()
        fxView.contentView.addSubview(label)
        label.frame = fxView.bounds

        view.addSubview(fxView)
    }
}
```

With that, the text label adapts according to the system materials being used, the content behind it and the vibrancy effect passed in to create compelling results. Since vibrancy effects amplify and adjust colors of any content layered behind a visual effect, it makes the content more vivid:



There are several styles of vibrancy you can use, and each one adapts differently based on the alpha components of the content being used. That's true for all of them, aside from `.label`, which will actually pass color through:

```
public enum UIVibrancyEffectStyle : Int {
    case label = 0
    case secondaryLabel = 1
    case tertiaryLabel = 2
    case quaternaryLabel = 3
    case fill = 4
    case secondaryFill = 5
    case tertiaryFill = 6
    case separator = 7
}
```

In SwiftUI, achieving the same thing is even easier. So long as you use the `.secondary` foreground style – SwiftUI will adapt it to use vibrancy, even if you don't have a material background. This recreates the example above:

```
// Xcode -> SwiftUI -> DarkModeFig2View.swift

struct DarkModeFig2View: View {
    var body: some View {
        ZStack {
            Image("Healthy Food")
            Text("Healthy Food")
                .font(.system(size: 46, weight: .black))
                .fontWidth(.expanded)
                .foregroundStyle(.secondary)
        }
    }
}
```

Trait Collections and Backgrounding Apps

You might be a bit puzzled if you use any sort of extensive logic with `UITraitCollection` and backgrounding apps. If you've written any code according to the interface style, you'll quickly notice that the interface style pops back and forth from both of the values of light and dark whenever you leave your app.

Why?

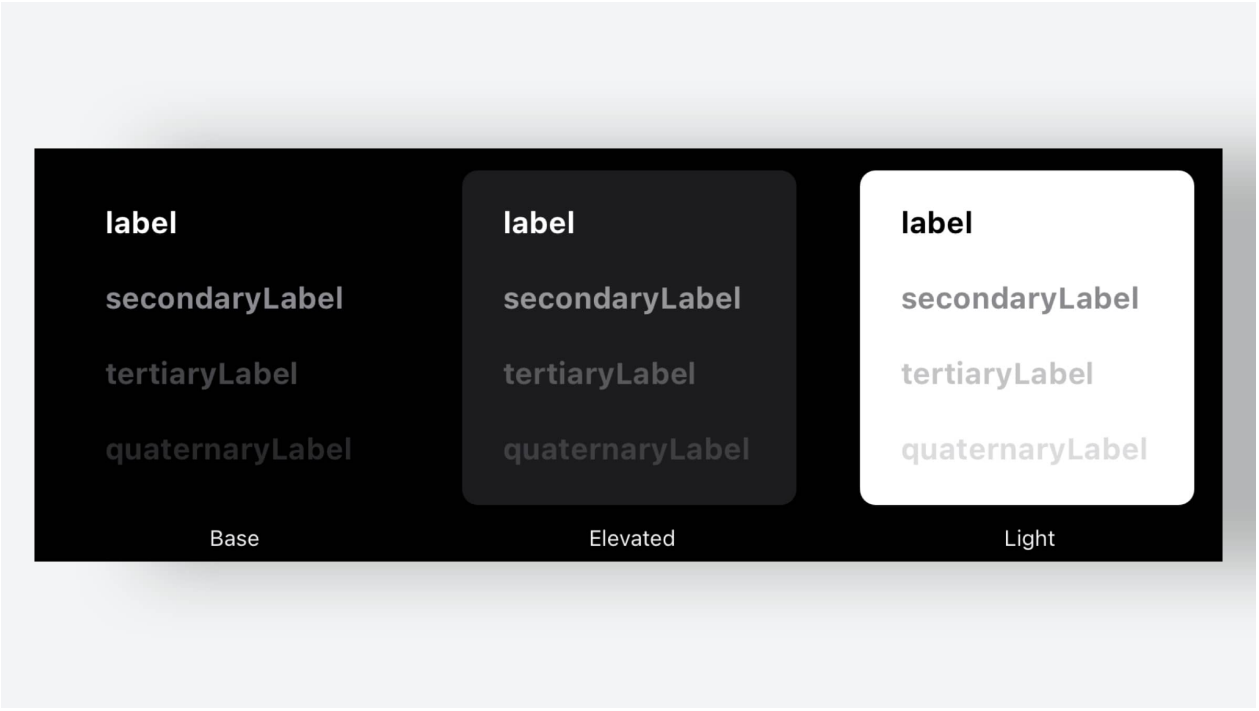
In short, it's mostly due to snapshotting. The system has to redraw the hierarchy to ensure that multitasking still looks correct if the user were to enter it with a different interface style from what was last used. To achieve that, iOS toggles the color scheme in the background for your app and performs the snapshot. You may not run into this – but if you do, it's confusing to make heads or tails of what's going on at first.

Elevation

Another component of Dark Mode is the concept of *elevation*. In short, we're talking about modality. To achieve the effect of depth, iOS slightly tweaks some colors based on their elevation, or more simply – their modality.

This makes sense because if you had the same black color for both a root view, and another modal view over it – you couldn't see where one stops and begins easily. The concept of elevation alleviates this.

Based on the elevation of the interface, some colors are dimmer to ensure background interfaces recede. Then, if they are elevated, colors would be brighter, making foreground views more prominent. This is good to be aware of, but as far I can tell, you can't change the elevation yourself (though it's possible to query in `UITraitCollection`). This snippet from Apple's Human Interface Guidelines demonstrates the differences quite well:



Don’t Forget Images

Just as we saw asset catalogs can be used for colors, the same is true of images. Using the same technique, you can supply a specific image for dark or light mode. This is great for situations such as onboarding, where branding assets may run especially bright to draw attention.

You can still achieve the same pop, and be easier on the eyes, by having your designer provide a dark mode asset of the same illustrations. There is no extra work you have to do, either. So long as it’s in an asset catalog, iOS will hot swap the images for you based on whether Dark Mode is active.

Three Key Takeaways

1. Dark Mode is a way for the entire system to display its interface in darker colors, your app included.
2. So long as you stick to stock components and semantic colors, you'll get much of this work for free.
3. If you roll your colors, be sure they adapt to the environment as well.

Drag and Drop

Drag and drop opens up so many powerful flows, and for my money — it's a critical feature to support in your apps. At its core, drag and drop usually opens up three primary interactions:

1. Move this thing from here to there (a *move*).
2. Take this data, but also put it somewhere else (a *copy*).
3. Take this thing, but reorder it within this container (a *reorder*).

Think about drag and drop on macOS, a core interaction on the platform for longer than some of you reading this have been alive. It's what made the operating system earn its "It just works", or "Things are easier to do" monikers. Want to take something from downloads and move it to the desktop? Just drag it from downloads, and drop it on the desktop. Done.

By supporting drag and drop – you're opening up the interactions that seem to make the most *sense* to people, and have them available when they want to use them. Here's what I mean by that; Whatever seems to be the *most* obvious way to do something, is likely the way it should work. That rings true of our downloads example I just mentioned.

"How can I move this from my downloads to the desktop? Hmmm, maybe I can just drag it." But now, you extend that line of thinking to your iOS apps.

When breaking down drag and drop, what we're really dealing with are two core pieces; a drag source and a drop destination. That's true whether you're using UIKit or SwiftUI, though their APIs represent them quite differently. As we go along, we'll ensure drag sources are reliable, informative and smooth – whereas we'll work to make sure that drag destinations are obvious, provide feedback and are lenient.

Before we look at how it works under the hood, it's a good time to think about how many drag and drop operations are found throughout iOS. There are multitudes of them, and it's not uncommon to see a tweet catch fire that says something similar to "What!! Why did nobody tell me you could do this 🤯! *A gif of some drag and drop things happening*"

To illustrate this – you can start a drag inside of Photos, hold down your finger and go home. Then, open up Messages and drop it right inside a chat. You could drag a photo, stay within the Photos app, and drop it into another album. On iPadOS, things get even more interesting with multiple windows – then you could easily move things via drag and drop all throughout the system and or to other apps.

How it Works

As always, we'll cover both UIKit and SwiftUI, and this is certainly a case where the APIs don't really have much in common across the two frameworks. But, there is one API that both can use – so let's start with that : `NSItemProvider`. An item provider is packed up alongside drags, and their drops, to represent the "what" of the item being used.

By using UTIs (uniform type identifiers) – the system can infer the payload of a drop and a consumer can operate on it. The power of item providers is that their corresponding UTI can be hierarchical in nature. Take a photo, for example, – there are so many formats, right? Is it .png, .jpeg – maybe a .raw photo?

No matter the format, there is a UTI that can be used to represent it. So if a specialized photo editing app took the drop – perhaps it would be most interested in the .raw representation of the file. Whereas something like Messages would likely do just fine with the .png or .jpeg versions. Type identifiers help apps work out how to best use what was actually dropped, and represent what kind of data started the drag.

So, to recap – when you assign to the array of UTIs in an item provider, you're creating a *promise*. And, that promise is saying "I can deliver these types of file formats to you, but it's up to you to choose with one you're most interested in." And, the word promise is very deliberate because when the drag starts – no operations of that file are happening – you're just *promising* that you can deliver them when it matters.

Now, let's dive into each platform to get a feel for how drag and drop works. Just a word of warning, UIKit has *a lot* going on – but it also makes it extremely flexible and powerful. SwiftUI, while as of iOS 16, has less to offer – but it's much simpler to implement.

UIKit

When someone begins a drag, UIKit constructs a drag item (`UIDragItem`). The part of your interface which constructs that drag is referred to as the interaction delegate (`UIDragInteractionDelegate`). In the drop zone, it's the opposite – that control is the destination, as it'll have to implement `UIDropInteractionDelegate`. You'll hear the word *interaction* a lot when dealing with drag and drop in UIKit, as it basically represents an entire drag and drop operation from start to finish.

As we just said, all of this starts with an item provider. If you're drag and dropping an `NSSting`, `NSURL`, `UIColor`, `UIImage`, or `NSAttributedString` – those all implement the necessary protocols for being an item provider for free. In your own custom models, you'll need to implement at least `NSItemProviderWriting` and possibly `NSItemProviderReading`.

So, let's look at the process from start to finish. We'll make a regular `UIView` be draggable, and it'll be dropped onto another `UIView` that'll change a text label text to whatever the drag view's text label's text is.

Starting with the drag view, we'll need to do three things:

1. Create an `NSItemProvider` to create the drag promise.
2. Adopt `UIDragInteraction` to let the system take the item provider, and put it into a `UIDragItem`.
3. Finally, ensure our view installs a `UIDragInteaction` into its `UIInteraction` array, and it includes the drag interaction delegate (which, again, is the view itself).

Here's that code in action:

```
// Xcode -> UIKit -> DragDropFig5ViewController.swift

class NumberDragView: UIView {
```

```
    private static let textNumber: String = "5"

    private lazy var numberLabel: UILabel = {
        /* Label setup ommitted for brevity */
        return numberLabel
    }()

    private let itemProvider: NSItemProvider = .init(object:
textNumber as NSString)

    override init(frame: CGRect) {
        /* Setup code ommitted for brevity */
        installDragInteraction()
    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    private func installDragInteraction() {
        let dragInteraction = UIDragInteraction(delegate: self)
        addInteraction(dragInteraction)
    }
}

extension NumberDragView: UIDragInteractionDelegate {
    func dragInteraction(_ interaction: UIDragInteraction,
itemsForBeginning session: UIDragSession) ->
[UIDragItem] {
        return [UIDragItem(itemProvider: itemProvider)]
    }
}
```

There, you can see all three steps occurring. There is an item provider, a drag interaction installed, and the view adopts `UIDragInteractionDelegate`. That's all it takes to make our view draggable – and if you ran the code as is, right now, you'd be able to drag it around.

Understanding Drag Sessions

To go a little further with we just did above, you'll need to understand that a `UIDragSession` was started. Once the gesture recognizer (which iOS installs and handles, by the way) kicks off a drag, the session for managing a drag activity is created. This way, iOS can invokes the drag interaction's `dragInteraction(_:itemsForBeginning:)` to start a drag that houses the promises found in the `UIDragItem`. Then, the drag session is populated from that – and the user can move it around and drop it where it needs to go.

Next, we want another view which accepts a drop – and the process is not all that different from what we just did, except we basically just switch around the protocol adoption for drops instead of drags.

Again, there are three core things to do:

1. Add a `UIDropInteraction` to your view.
2. Adopt `UIDropInteractionDelegate`
3. At the very least, implement `dropInteraction(interaction:, sessionDidUpdate session) -> UIDropProposal` and `dropInteraction(_ interaction:, performDrop session)`.

Here's what it looks like. When we drop the view above onto this one, it'll change its text label's text to the one that was dropped:

```
class NumberDropView: UIView {
    private lazy var numberLabel: UILabel = {
        /* Label setup omitted for brevity */
        return numberLabel
    }()

    override init(frame: CGRect) {
        /* Init code omitted for brevity */
```

```
        installDropInteraction()
    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    private func installDropInteraction() {
        let dropInteraction = UIDropInteraction(delegate: self)
        addInteraction(dropInteraction)
    }
}

extension NumberDropView: UIDropInteractionDelegate {
    // What can be dropped onto this view?
    func dropInteraction(_ interaction: UIDropInteraction, canHandle
session: UIDropSession) -> Bool {
        return session.canLoadObjects(ofClass: String.self)
    }

    // What should happen when its dropped here?
    func dropInteraction(_ interaction: UIDropInteraction,
sessionDidUpdate session: UIDropSession) ->
UIDropProposal {
        return UIDropProposal(operation: .copy)
    }

    // Finally, load in the dropped data
    func dropInteraction(_ interaction: UIDropInteraction,
performDrop session: UIDropSession) {
        let _ = session.loadObjects(ofClass: String.self)
{ textItems in
            guard let firstString = textItems.first else {
                return
            }

            self.numberLabel.text = "Dropped\n\(firstString)"
        }
    }
```

```
}
```

As you can see, I implemented three totals delegate methods – and have commented why on each of them. Basically, iOS gives us a chance to tell it ahead of time whether we can accept the drop. What we want to do with it once it's dropped (copy it) and how to consume it (pull out the string, and put it in our text label).

Understanding Drop Sessions

When a drag begins, the system also creates a `UIDropSession` which lets the system peek into the protocol functions we implemented above. That way, it'll know how it can handle the drop (or if it can at all). It updates the interface accordingly, and you'll notice we get the green plus symbol which shows for free (that is created based on what we returned for our `UIDropProposal`). Finally, when the drop occurs, it's consumed asynchronously and a view uses the data.

SwiftUI

Setting up drag and drop for SwiftUI is, in comparison, very little code when compared to UIKit. There are two core concepts to understand, and that's about it. The first is that you'll be using either `NSItemProvider` (as we did with UIKit) or (more preferably) the `Transferable` protocol to create the promise of data delivery. The second is that all of the drag and drop is set up with modifiers.

To construct promises for drag and drop, though, you should use `Transferable`, which we talked about in the `AirDrop` chapter. The `Transferable` property is where things are headed in SwiftUI, especially in relation to drag and drop, so I recommend to others that they use it over an item provider. Apple themselves mention this in the documentation:

> *In your modifiers, provide or accept types that conform to the Transferable protocol, or that inherit from the NSItemProvider class. When possible, prefer us-*

ing transferable items. Reference: https://developer.apple.com/documentation/swiftui/drag-and-drop

That said, after all we just went through to get drag and drop working in UIKit, you might be offended at the following code sample. In fact, we can wire up a drag view with just one modifier:

```
// Xcode -> SwiftUI -> DragDropFig1View.swift

struct DragDropFig1View: View {
    var body: some View {
        RoundedRectangle(cornerRadius: 10)
            .fill(Color.blue)
            .frame(width: 100, height: 100)
            .overlay {
                Text("Drag Me")
                    .foregroundColor(.white)
            }
            .draggable("Drag Me")
    }
}
```

…and that's it. That square can now be dragged all around, and once delivered – its payload would be a String reading "Drag Me". I'd like to note that this would work with a UIKit app, too. If it's looking for any Uniform Type Identifier dealing with text, it would accept a drop with a String from a SwiftUI app.

There are basically two different ways to make things draggable. One is to use the variants which use the `Transferable` protocol. This is what we did above, but you may not have caught it because I implicitly passed in something that adopts the protocol already, a String value:

```
.draggable("Drag Me")
```

You can make your own custom types adopt `Transferable` (again, we did this in the AirDrop chapter) or use several of the types that already do it for you (links, images, etc).

The next way is to use a modifier which accepts `NSItemProvider`:

```
// Xcode -> SwiftUI -> DragDropFig2View.swift

/* Same code as previous example to */
.onDrag {
    return .init(object: "Drag Me" as NSString)
}
```

This is the same as above, except that the `.onDrag` modifier wants you to return an `NSItemProvider`. It's also a closure, so if you need to run logic to construct the item provider you can do that here.

So, that covers the two ways to make things draggable, but if you type those modifiers into Xcode, you'll see that there are four of them. The other two variants allow you to return a custom preview, so if you wanted the drag item to look different from the view itself – this is where you could return something else.

Here, once the rectangle starts a drag – the drag item is a circle instead of a square:

```
// Xcode -> SwiftUI -> DragDropFig3.view

RoundedRectangle(cornerRadius: 10)
    .fill(Color.blue)
    .frame(width: 100, height: 100)
    .overlay {
        Text("Drag Me")
            .foregroundColor(.white)
    }
    .draggable("Drag Me") {
        Circle()
```

```
            .fill(Color.cyan)
            .frame(width: 100, height: 100)
            .overlay {
                Text("Drag in progress")
                    .font(.callout)
            }
    }
```

Next, drops. Drops, naturally, are a little more involved but not much more so. There are a few different ways to handle it:

1. Use a simple modifier to specify a `UTType` to accept, and operate on it.
2. Specify a `UTType` – but handle the details in a `DropDelegate`.
3. Finally, use a drop destination to specify types to accept that adopt `Transferable`, and act on those within a closure.

The first is the simplest to use, the second expands on that and allows for more call-backs during the drop operation, and the third is the most SwiftUI-oriented of the three. Let's look at each.

Method One

Here, we'll identify the `UTType` identifiers we're interested in, and then load in data from the item providers passed into us.

```
// Xcode -> SwiftUI -> DragDropFig3View.swift
@State private var targeted: Bool = false

var methodOneDropZone: some View {
    Rectangle()
        .fill(Color.red)
        .overlay {
            Text("Method\n1")
                .font(.title)
                .fontWeight(.black)
                .foregroundColor(.white)
```

```
        }
        .frame(maxHeight: 240)
        .onDrop(of: [UTType.text], isTargeted: $targeted)
{ providers in
            guard let firstProvider = providers.first,
                  firstProvider.canLoadObject(ofClass: String.self)
else {
                return false
            }

            // Load a String from the firstProvider
            return true
        }
}
```

Method Two

This method involves creating a `Struct` or object which conforms to `DropDelegate`, which looks an awful lot like UIKit's `UIDropInteractionDelegate`. Again, you specify the types you're interested in. This way offers the most flexibility, as we can easily do things like animate the drop destination when a drag enters or exits:

```
// Xcode -> SwiftUI -> DragDropFig3View.swift
@State private var expandZoneTwo: Bool = false

var methodTwoDropZone: some View {
    Rectangle()
        .fill(Color.purple)
        .overlay {
            Text("Method\n2")
                .font(.title)
                .fontWeight(.black)
                .foregroundColor(.white)
        }
        .frame(maxHeight: 240)
        .shadow(radius: expandZoneTwo ? 80 : 0)
        .animation(.default, value: expandZoneTwo)
        .onDrop(of: [UTType.text],
```

```
            delegate: DropZoneDelegate(animate: $expandZoneTwo))
}

struct DropZoneDelegate: DropDelegate {
    @Binding var animate: Bool

    func validateDrop(info: DropInfo) -> Bool {
        return info.hasItemsConforming(to: [UTType.text])
    }

    func performDrop(info: DropInfo) -> Bool {
        if let textProvider = info.itemProviders(for:
[UTType.text]).first {
            let _ = textProvider.loadTransferable(type: String.self)
{ result in
                switch result {
                case .success(let text):
                    print("Loaded \(text)")
                case .failure(let error):
                    print("Couldn't load text from drop: \
(error.localizedDescription)")
                }
            }
        }

        animate = false
        return true
    }

    func dropEntered(info: DropInfo) {
        animate = true
    }

    func dropUpdated(info: DropInfo) -> DropProposal? {
        return .init(operation: .copy)
    }

    func dropExited(info: DropInfo) {
        animate = false
```

```
    }
}
```

Method Three

Finally, method three allows you to not worry about the item providers altogether, and just operate on the array of items you get back based on the `UTType` you want, along with the location of where the drop occurs:

```
// Xcode -> SwiftUI -> DragDropFig3View.swift

var methodThreeDropZone: some View {
    Rectangle()
        .fill(Color.green)
        .overlay {
            Text("Method\n3")
                .font(.title)
                .fontWeight(.black)
                .foregroundColor(.black)
        }
        .frame(maxHeight: 240)
        .dropDestination(for: String.self) { items, location in
            let _ = items.first ?? ""
            return true
        }
}
```

Tips

Collection and Table View Drag and Drop Support

Both collection and table view offer dedicated delegate methods and protocols to implement drag and drop. The good news? They don't reinvent the wheel at all. The rea-

son we started looking at how to roll a drag and drop interaction all on your own is to ensure you get the concepts.

To that end – all that table and collection view are doing is exposing those same concepts, except they express them in terms of index paths. You can also easily support drag and drop reordering using a diffable data source and these protocols. Here's an end-to-end example with `UITableView` (though it's the same thing with collection view – the naming is just different).

Please open up this code sample and peek around yourself, and you'll find that it all looks the same as setting up the drop yourself. In fact, it's even a bit easier:

```
// Xcode -> UIKit -> DragDropFig1ViewController.swift

class MassEffectTableDataSource: UITableViewDiffableDataSource<Int,
MassEffectGame> {
    override func tableView(_ tableView: UITableView, canMoveRowAt
indexPath: IndexPath) -> Bool {
        return true
    }

    // Reodering
    override func tableView(_ tableView: UITableView, moveRowAt
sourceIndexPath: IndexPath, to destinationIndexPath:
IndexPath) {
        guard let fromGame = itemIdentifier(for: sourceIndexPath),
              sourceIndexPath != destinationIndexPath else
{ return }

        var snap = snapshot()
        snap.deleteItems([fromGame])

        if let toGame = itemIdentifier(for: destinationIndexPath) {
            let isAfter = destinationIndexPath.row >
sourceIndexPath.row
```

```
            if isAfter {
                snap.insertItems([fromGame], afterItem: toGame)
            } else {
                snap.insertItems([fromGame], beforeItem: toGame)
            }
        } else {
            snap.appendItems([fromGame], toSection:
sourceIndexPath.section)
        }

        apply(snap, animatingDifferences: false)
    }
}

class DragDropFig1ViewController: BaseSampleViewController {
    var videogames: [MassEffectGame] = MassEffectGame.data
    let tableView = UITableView(frame: .zero, style: .insetGrouped)

    lazy var datasource: MassEffectTableDataSource = {
        let datasource = MassEffectTableDataSource(tableView:
tableView, cellProvider: { (tableView, indexPath, model) ->
UITableViewCell? in
            let cell = tableView.dequeueReusableCell(withIdentifier:
"cell", for: indexPath)
            cell.textLabel?.text = model.name
            return cell
        })

        return datasource
    }()

    override func viewDidLoad() {
        super.viewDidLoad()

        tableView.register(UITableViewCell.classForCoder(),
forCellReuseIdentifier: "cell")
        view.addSubview(tableView)
        tableView.frame = view.bounds
```

```
        tableView.autoresizingMask =
[.flexibleWidth, .flexibleHeight]

        tableView.dragDelegate = self
        tableView.dropDelegate = self
        tableView.dragInteractionEnabled = true
        tableView.backgroundColor = .systemGroupedBackground

        var snapshot = datasource.snapshot()
        snapshot.appendSections([0])
        snapshot.appendItems(videogames, toSection: 0)
        datasource.applySnapshotUsingReloadData(snapshot)
    }
}

extension DragDropFig1ViewController: UITableViewDragDelegate {
    func tableView(_ tableView: UITableView, itemsForBeginning
session: UIDragSession, at indexPath: IndexPath) ->
[UIDragItem] {
        guard let item = datasource.itemIdentifier(for: indexPath)
else {
            return []
        }
        let itemProvider = NSItemProvider(object: item.id.uuidString
as NSString)
        let dragItem = UIDragItem(itemProvider: itemProvider)
        dragItem.localObject = item

        guard let cell = tableView.cellForRow(at: indexPath) else
{ return [dragItem] }

        let cellInsetContents = cell.contentView.bounds.insetBy(dx:
2.0, dy: 2.0)

        dragItem.previewProvider = {
            let dragPreviewParams = UIDragPreviewParameters()
            dragPreviewParams.visiblePath =
UIBezierPath(roundedRect:cellInsetContents, cornerRadius: 8.0)
```

```
            return UIDragPreview(view: cell.contentView, parameters:
dragPreviewParams)
        }

        return [dragItem]
    }
}

extension DragDropFig1ViewController: UITableViewDropDelegate {
    func tableView(_ tableView: UITableView, dropSessionDidUpdate
session: UIDropSession, withDestinationIndexPath
destinationIndexPath: IndexPath?) -> UITableViewDropProposal {
        return UITableViewDropProposal(operation: .move,
intent: .insertAtDestinationIndexPath)
    }

    func tableView(_ tableView: UITableView, performDropWith
coordinator: UITableViewDropCoordinator) {
        // If you don't use diffable data source, you'll need to
reconcile your local data store here.
        // In our case, we do so in the diffable datasource
subclass.
    }
}
```

Some things to note:

1. You need to subclass diffable datasource to get reordering to work. This isn't obvious at first, but that's where the methods are exposed to handle it.
2. You'll need to set a few flags, specifically – you'll have to assign to `dragDelegate`, `dropDelegate` and set `dragInteractionEnabled` to true.
3. Finally, implement the required methods in `UITableViewDropDelegate` and `UITableViewDragDelegate`.

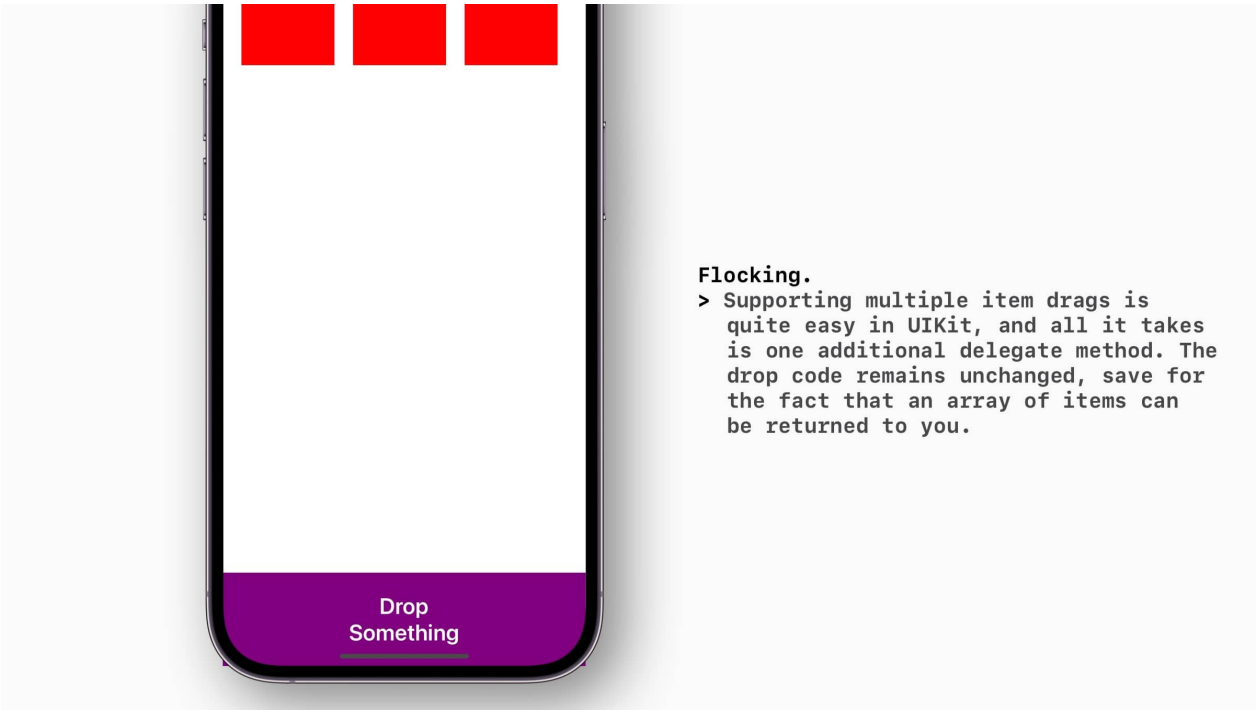
Supporting Multiple Item Drags

Also referred to as flocking, supporting multiple item drags is really just a matter of implementing one more delegated method in `UIDragInteractionDelegate`. By specifying more `UIDragItem` instances to return based off a touchpoint, you can add multiple items to a drag – in fact, there is no limit.

Photos is a great example. Start dragging around a photo, and while you're doing that – tap several other photos with a free finger. You'll see it just adds them to the existing drag interaction. To do this yourself, you'll want to implement the following (`itemsForAddingTo`):

```
extension DragDropFig2ViewController: UIDragInteractionDelegate {
    func dragInteraction(_ interaction: UIDragInteraction,
itemsForBeginning session: UIDragSession) -> [UIDragItem] {
        return [UIDragItem(itemProvider: .init(object: "" as
NSString))]
    }

    // This adds items to an existing drag interaction
    func dragInteraction(_ interaction: UIDragInteraction,
itemsForAddingTo session: UIDragSession, withTouchAt point:
CGPoint) -> [UIDragItem] {
        return [UIDragItem(itemProvider: .init(object: "" as
NSString))]
    }
}
```

Open up `DragDropFig2ViewController` to see this in action. Here, the drop zone below will count out how many red squares you dropped on top of it. While dragging a square, tap another one – add, you'll see that it gets added to the drag session:



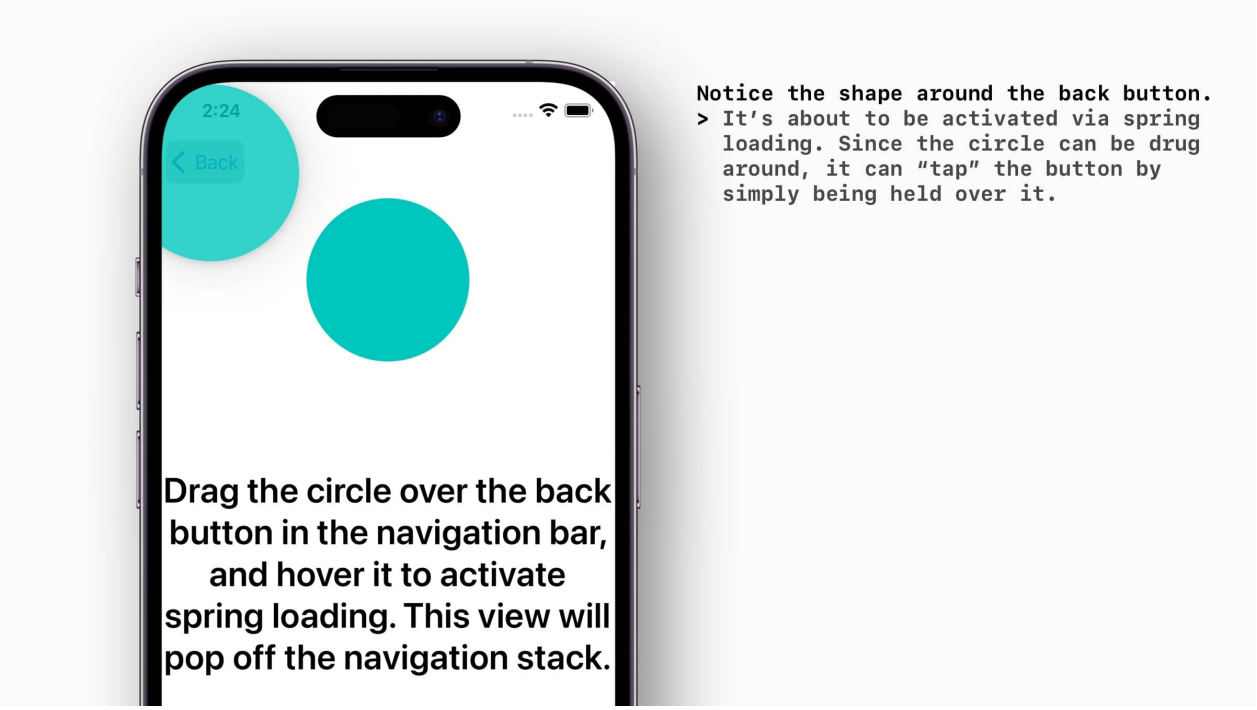
Support Spring Loading

Spring loading works in tandem with drag and drop. It lets people activate controls by dragging content over them – essentially working as a button tap. This is helpful when navigating around with drag and drop, such as wanting to pop a view from a navigation stack. The purpose of spring loading is to allow for easy navigation even when someone is in the middle of a drag interaction.

Various controls within UIKit, such as buttons, adopt this out of the box for you as they conform to `UISpringLoadedInteractionSupporting`. Of course, you can make a custom view support this behavior, too, by adding an interaction:

```
let spring = UISpringLoadedInteraction { interaction, context in
    // Handle interaction
}
view.addInteraction(spring)
```

If you open `DragDropFig3ViewController.swift`, you can try this out for yourself. Drag the purple circle over the back button, and after about a second – you'll that it blinks and pops the view off the navigation stack (which is what tapping the back button would do if it were tapped):



Add Clarity to Drop Zones

I think it helps to make droppable areas incredibly obvious. This is doubly true when it's outside a list context, where drag and drops are inherently understood by most people.

To that end, when something can be dropped, give a visual cue to the user who reveals it as a drop zone. There are delegate methods that are perfect for this in `UIDropInteractionDelegate`.

Let's enhance our initial number drop example from earlier in UIKit. Now, once a drop enters – we'll blow up the view a little and add a border. We'll undo that if the drop leaves, animating both of these interactions:

```
// Xcode -> UIKit -> DragDropFig5View.swift

// Make it obvious we can drop here
func dropInteraction(_ interaction: UIDropInteraction,
sessionDidEnter session: UIDropSession) {
    UIView.animate(withDuration: 0.25) {
        self.transform = .init(scaleX: 1.4, y: 1.4)
        self.layer.borderColor = UIColor.blue.cgColor
        self.layer.borderWidth = 8
    }
}

// Make it obvious when a drop leaves
func dropInteraction(_ interaction: UIDropInteraction,
sessionDidExit session: UIDropSession) {
    UIView.animate(withDuration: 0.25) {
        self.transform = .identity
        self.layer.borderWidth = 0
    }
}
```

And finally, if you'd like to perform any animations alongside the drop *itself*, there's a delegate function for that too:

```
// Perform when the drop occurs
func dropInteraction(_ interaction: UIDropInteraction, item:
UIDragItem, willAnimateDropWith animator: UIDragAnimating) {
    animator.addAnimations {
        self.transform = .identity
        self.layer.borderWidth = 0
    }
}
```

This is perfect because we'll want to animate the view back to its original state if the drop does occur within it – the same as we would if the user decided not to drop anything there at all (i.e., the animations we added in `dropInteraction(_ interaction:, sessionDidExit:)`.

In SwiftUI, we accomplished the same things using our `DropDelegate` listed in method two.

Customize Drag Sources in UIKit

Customizing the look of the items you drag around can make things feel a lot more natural. Plus, in my opinion, the rounded corners you can easily apply make the elements you're dragging around feel a little smoother too. The default view you'll likely get is a box, sharp-edged rectangle since several drags being in a list context.

To tweak this in UIKit, you can look to `UIDragPreviewParameters`. By supplying it with a `UIBezierPath`, you can easily round off the corners of the view you're dragging. In this sample, we do that for a table view cell by returning preview parameters inside a `previewProvider` closure found on the `UIDragItem`:

```
// Xcode -> UIKit -> DragDropFig1ViewController.swift

func tableView(_ tableView: UITableView, itemsForBeginning session:
UIDragSession, at indexPath: IndexPath) ->
[UIDragItem] {
    guard let item = datasource.itemIdentifier(for: indexPath) else
{
        return []
    }
    let itemProvider = NSItemProvider(object: item.id.uuidString as
NSString)
    let dragItem = UIDragItem(itemProvider: itemProvider)
    dragItem.localObject = item
```

```
    guard let cell = tableView.cellForRow(at: indexPath) else
{ return [dragItem] }

    let cellInsetContents = cell.contentView.bounds.insetBy(dx: 2.0,
dy: 2.0)

    // Customize the drag preview here
    dragItem.previewProvider = {
        let dragPreviewParams = UIDragPreviewParameters()
        dragPreviewParams.visiblePath =
UIBezierPath(roundedRect:cellInsetContents, cornerRadius: 8.0)
        return UIDragPreview(view: cell.contentView, parameters:
dragPreviewParams)
    }

    return [dragItem]
}
```

I want to point out that it's straightforward to get lost and overwhelmed by the API surface area on offer to customize how drag items look. So, here's a quick breakdown of the big players, and the good news is that, on their own, each is easy to use. It's just a matter of knowing what and when:

1. `UIDragPreviewParameters`: What we used above. Here, you can set the appearance and shape of the drag item.
2. `UIDragPreview`: Houses the parameters, and is used to identify the view to preview alongside those parameters.
3. `UIDragPreviewTarget`: The target of a source, or destination, of a drag item – expressing coordinates to help with animations.
4. `UITargetedDragPreview`: Uses the preview target to package up a customized drag preview, and can be used with parameters too.

Using a combination of these, you can change the drag preview view altogether to look like something entirely different than what the drag view appears as. In SwiftUI, we do this by using the trailing closure to provider a preview view (and there was an example of that in the SwiftUI section above).

Offer Undo and Redo

An accidental drag and drop can happen, so make it easy to revert things back. As soon as a drop occurs, it's not a bad idea to use a banner of some sort to say "X moved to Y" (or whatever applies) with an undo button in it. It's not always immediately obvious of how to *undo* a drag and drop operation — so making an obvious undo is not a bad idea.

Of course, you should still support the system's undo and redo capabilities. For more on that, be sure to read the entire chapter over it, "Undo and Redo" in the User Experience book in this series.

Three Key Takeaways

1. Support as many drag and drop interactions as you can in your app — this is a "the more, the better" situation.
2. Ensure you've designed your drag and drop interactions to be *obvious*, the outcome of a drop should yield no surprises.
3. When designing for drag and drop, ask yourself if there are places to add its three core flows (copy, move, or reorder).

Reacting to Low Power Mode

Effective energy management on iOS. Sounds a little ambiguous, doesn't it? How can we effectively manage much of anything when Apple goes to such great lengths to sandbox any little process or thread that could be abused?

It's objectively true that iOS devices are more powerful than ever — and that's a trend which doesn't appear to be slowing down anytime soon. Moore's Law continues to hold true, due in no small part to Apple's own proprietary silicon chips.

Even so, there are still several considerations for developers to think about, and chief among them is Low Power Mode. While this won't be the longest or most information dense chapter in the book series (the majority of it revolves around one structure, actually), it hopefully will be *impactful*. You'll have some things to change or optimize after reading it.

Let's dig in, and start talking about `ProcessInfo` and how it relates to Low Power Mode.

How it Works

Low Power Mode, introduced in iOS 9 and later on iPads and Mac alongside iPadOS 15 and macOS Monterey, was originally billed as a way to turn "levers and knobs we don't even know about" to dole out as much battery life as possible when things are tight. Basically, it tunes out anything that can cause battery drain and instead focuses the operating system only on the essentials.

Now, extrapolate that thought out to your app, what can you do to help out Low Power Mode's efforts? As we'll see shortly via some code samples — reacting to Low Power Mode is not too difficult. The trick, then, becomes knowing *what* to do when it's active.

To begin to form an understanding around what things we might change, we need to start by looking at what Apple does when Low Power Mode is activated. Such activities that are mutated in some form or fashion include, but are not limited to, the following:

- Reduced CPU and GPU performance.
- Discretionary and background activities, including networking, are paused.
- Reduced screen brightness.
- Reduced timeout for auto-locking the device.
- Mail fetch, motion effects, and animated wallpapers are disabled.
- Refresh rates on ProMotion displays.
- 5G (except for video streaming) on iPhone 12 models.

Take a look at that rap sheet and ask yourself—which of those things happen in my app? If you're doing any of those things (and you definitely are), it's likely they are good candidates to either pause or terminate if Low Power Mode kicks in.

For example:

- **Expensive animations?** Skip em'.
- **Tons of vibrancy and blur effects?** Make them opaque.
- **Lots of background networking and chatter?** Pause it.
- **Non-critical ocation services being used?** Turn them off.

Those are some obvious ones, but once you compile a list of processes to address, you can address them all with one helpful structure: `ProcessInfo`. Using it, we can look at many parts, aspects, and the state of the current process. One of those things is, of course, whether or not Low Power Mode is on.

By knowing when Low Power Mode is active, we can modify our apps according to the list of battery-heavy activities we listed above. Furthering the point, in the documentation for `ProcessInfo` Apple even hints that we should take note when it's on and then do our part:

> *The system has heuristics to improve battery life, performance, and responsiveness of applications for the benefit of the user.*

While it's a binary value you can read, I'm sure you might've guessed that querying for the state of Low Power Mode is quite elementary. And of course, you're absolutely right:

```
let lpmOn = ProcessInfo.processInfo.isLowPowerModeEnabled

if lpmOn {
```

```
    pauseNetworkCalls()
}
```

That, however, would not be very opportunistic. It's obviously not realistic to query such a value in `viewDidLoad()` in every view controller in our UIKit apps, or simply check it within SwiftUI's `onAppear`. Fortunately – Apple has notifications that can we register to, so we can properly react to Low Power Mode changes in real-time.

For instance, did you know that Low Power Mode will be switched off automatically by iOS should the device reach an eighty percent charge? Or, the user can simply toggle it on and off from Control Center. For that and several other reasons, I find it's best to abstract away some of `ProcessInfo`'s information.

By creating one class that vends `ProcessInfo` values, you can plug and play it with both UIKit and SwiftUI. We often need something more flexible than simply reading a value at an arbitrary time. As such, UIKit requires notification observers to be up-to-date, and SwiftUI doesn't offer it in the environment. So, this is my preferred method:

```
// Xcode -> Reacting to Low Power Mode -> ProcessUtil.swift

import Foundation
import Combine

class ProcessUtil: ObservableObject {
    private typealias NC = NotificationCenter

    @Published private(set) var lowPowerModeOn: Bool = false

    let info = ProcessInfo.processInfo
    let lowPowerModeChanged = NC.default.publisher(for:
NSNotification.Name.NSProcessInfoPowerStateDidChange)
    let thermalStateChanged = NC.default.publisher(for:
ProcessInfo.thermalStateDidChangeNotification)
```

```
    private var subs: [AnyCancellable] = []

    init() {
        lowPowerModeOn = info.isLowPowerModeEnabled

        lowPowerModeChanged.sink { [weak self] _ in
            guard let self = self else { return }
            self.lowPowerModeOn = self.info.isLowPowerModeEnabled
        }
        .store(in: &subs)
    }
}
```

Now, you can respond to Low Power Mode in a SwiftUI friendly way:

```
// Xcode -> SwiftUI -> LowPowerModeFig1View.swift

struct LowPowerModeFig1View: View {
    @StateObject private var processUtil = ProcessUtil()

    var body: some View {
        Form {
            Section("Process Info") {
                Text(processUtil.info.hostName)
                Text("Low Power Mode: \(processUtil.lowPowerModeOn ?
"Enabled" : "Disabled")")
            }
        }
    }
}
```

Occasionally, you might want the publisher itself, though, which you can get at from the `Notification`. That's why I hang it (and one for thermal state – more on that later) off of the object for ease of use:

```
.onReceive(processUtil.lowPowerModeChanged) { change in
    // React to change
```

```
}
```

…which saves you from a bit of a word salad situation otherwise:

```
.onReceive(NotificationCenter.default.publisher(for:
NSNotification.Name.NSProcessInfoPowerStateDidChange)) { change in
    // React to change
}
```

Recall that, again, all of this is in service of making battery life last longer. So remember to do things like reduce animations, pause location updates, use opaque colors instead of visual effects, forgoing aggressive network calls for cache updates, disabling any non-critical syncing or automatic backups processes and anything similar when it's active. Aside from Apple has listed as high-cost operations, *you* will know where the savings are at in your app. So, work to optimize them for Low Power Mode.

Tips

Leveraging ProcessInfo for More

As we've seen, by using `ProcessInfo` we can certainly play our part in the Low Power Mode equation to help extend battery life even more. However, this thread safe singleton can supply developers with all sorts of insightful information related to the current process by exposing unique process information agents.

As such, its utility goes beyond simply telling us about Low Power Mode, as the name implies – it contains just about everything you'd want to know regarding the current process:

```
let processInfo: ProcessInfo = .processInfo
let machineName = processInfo.hostName
```

```
//Prints out jordans-macbook-pro.local
print(machineName)
```

While the naming of the singleton is a little redundant, it's simply a vestige of its Foundation roots. I typically stub it out in a variable to make it a little easier on the eyes (if you aren't using a dedicated class for it, as I've done earlier in the chapter):

```
// Like this
let info: ProcessInfo = .processInfo

//Prints out jordans-macbook-pro.local
print(info.hostName)
```

From that single previous line of code, `ProcessInfo` will attempt to interpret environment variables and command-line arguments into Unicode — returning them back to us as friendly UTF-8 strings. If the process can't be successfully morphed into unicode or the ensuing C-String conversion fails—the process is ignored.

So, what else can we query? A close cousin to Low Power Mode is the thermal state of a user's device. Because, of course, when the device reaches critical mass and is close to exploding, the first thing that comes to mind is pausing some network calls:

```
Section("Thermal State") {
    switch processUtil.info.thermalState {
    case .nominal:
        Text("Nominal")
            .foregroundColor(.green)
    case .fair:
        Text("fair")
            .foregroundColor(.yellow)
    case .serious:
        Text("Serious")
            .foregroundColor(.red)
    case .critical:
```

```
            Text("Critical")
                .foregroundColor(.red)
                .bold()
        @unknown default:
            Text("Unknown")
        }
}
```

Just like we did with Low Power Mode, we can add a bit of functionality to our `ProcessUtil` class to use thermal states with SwiftUI or UIKit easily:

```
// Xcode -> ProcessUtil.swift

// Add in the published property
@Published private(set) var thermalState: ProcessInfo.ThermalState =
.nominal

// Listen for any changes in its initializer
thermalStateChanged.sink { [weak self] _ in
    guard let self = self else { return }
    self.thermalState = self.info.thermalState
}
.store(in: &subs)
```

For anything you need that `ProcessInfo` has, this is typically the pattern I follow.

Going beyond querying specific values, if you're wrestling with debugging the cause of a sudden termination—how about channeling your inner throwback sprit and busting out some Objective-C?

```
//A helpful little LLDB command to get the cause of termination
p (long)[[NSClassFromString(@"NSProcessInfo") processInfo]
_suddenTerminationDisablingCount]
```

`ProcessInfo` has a lot to offer—it's an unsung hero that can help with a lot more than just Low Power Mode.

Three Key Takeaways

1. Low Power Mode automatically disables expensive operations, system wide.
2. You should query for Low Power Mode, and react to it in your own app.
3. Use abstractions to make it easy to use `ProcessInfo` easily in both UIKit and SwiftUI.

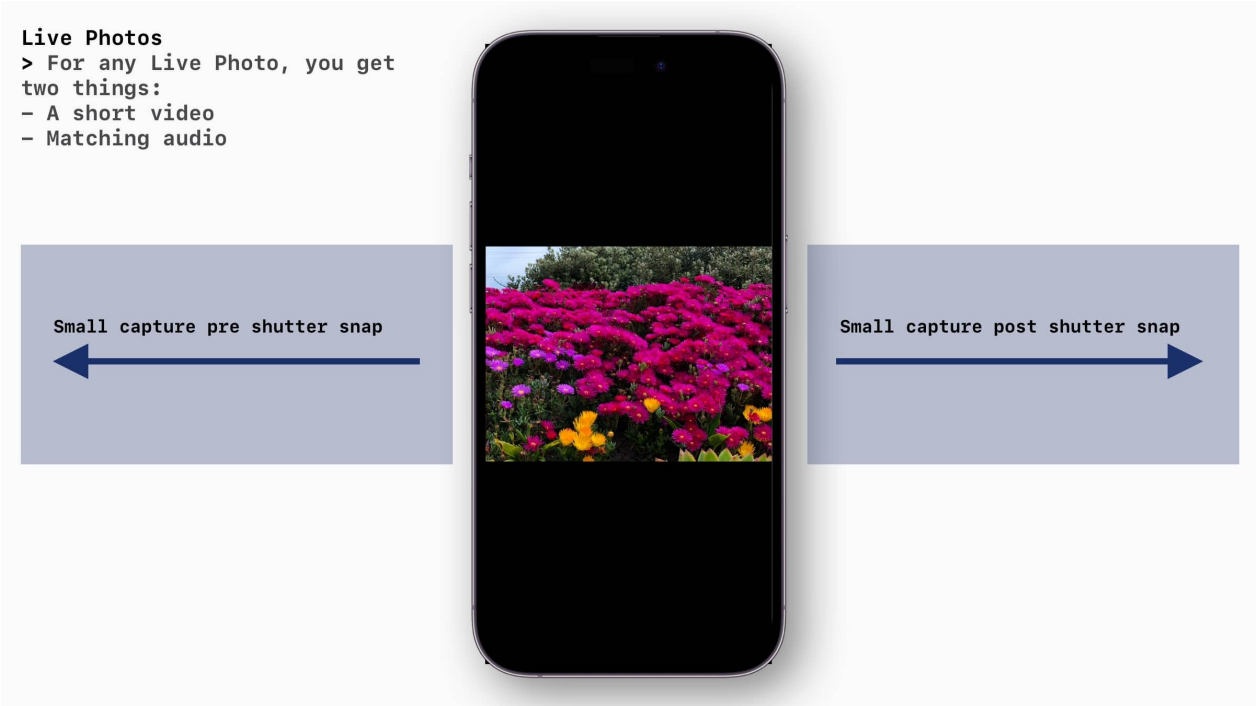
Live Photos

With the revelation of 3D Touch (God rest its soul) came a new media format, the Live Photo. The nascent paradigm fit in perfectly with the iPhone's newly minted sensors that could report how much force you were applying to the device's display. This allowed for the (now defunct, also – God rest its soul) peek and pop gestures.

These were some of my very favorite interactions on iOS, and it made the device feel incredibly reactive. Just push down on a link, and you would "peek" into the resulting page. Then, apply a bit more force, and it would "pop" into a full presentation. The same thing applied to Live Photos – you could play the entire video portion back with a slight push. I lament that this is all gone now (it's largely replaced by context menus and a very similar API and user experience) but none-the-less, Live Photos are here to stay.

At their core, Live Photos are a hybrid between a short video and a picture. Apple achieves this by capturing just a little bit of video *before* the photo was taken, and a lit-

tle bit *after* it was taken, too. The idea is that you get a better sense of the memory, it's "live" after all, than you might with a static image:



Though Live Photos maybe didn't have their cultural moment as much as Apple would've hoped, they are still the default format that pictures are taken in when using the Camera app. That means, by proxy – they are likely the most popular media format around today, considering the billions of iPhone users.

So, in our apps – we primarily need to do one thing well, and that is display them properly. While the majority of apps should rely on Apple's first part photo picker interfaces (which display them properly) – perhaps you have a photo import function of some sort or have a small media picker of your own.

In such cases, displaying them properly so that they can actually show as a bonafide Live Photo is key. If you simply displayed them as a static image, it could confuse users or, worse, disappoint them when they expected a Live Photo instead.

Secondly, it's possible to pick them apart and query their individual media aspects in addition to capturing some of your own. This is decidedly more involved, and is mostly a use case for media-based editing apps. But, it is possible – and you can even create your own Live Photos to boot.

How it Works

The Data Model

Interacting with Live Photos all revolves around one data model – `PHLivePhoto`. Similar to `PHAsset` which you may be familiar with, it contains the data to be used for display purposes only. It *doesn't* have the actual data which backs the Live Photo, just as a `UIImage` doesn't represent the data from the file from which it was loaded.

The tricky part with Live Photo support isn't displaying them, though – it's *getting* them or creating them which can be…a bit of a pain. The issue is that there are a few ways to do it, depending on what you're trying to achieve. If you're simply displaying a Live Photo, then there is a quick and painless route since iOS 14. If you need their individual media tracks of audio and video – then you've got some more work to do. Doubly so if you're trying to create your own Live Photo.

Continuing with that thought, if you want the assets backing a Live Photo – then `PHAssetResource` is where you'll head. By using `PHAssetResourceManager` – you can get an instance of the video and audio files which Live Photos are created from. While these resources are coupled with `PHAsset`, there is a way to grab the information directly if you've got a Live Photo already.

For example, if you wanted to export the raw sources for a Live Photo for a photo sharing app, you could do that with an existing `PHLivePhoto` you're using for display already:

```
let resources = PHAssetResource.assetResources(for: livePhoto)

// Now you could upload the sources to your server to reconstruct
later
```

We'll look at this a bit more in the "Tips" section.

Loading Live Photos

So, loading Live Photos. As I mentioned earlier, there are a few ways this can work. In fact, if you were to Google the situation right now – you may end up down a rabbit hole of dated information that proposes more work than you probably need. If all you're doing is loading in a Live Photo – I've got a modern solution for you. Let's start there.

`PHPickerViewController` is a bit of a successor to `UIImagePickerController`. Though they are both around and the latter hasn't been officially deprecated in any way – the former has a few advantages:

- More user-selected assets that are available.
- Improved stability around larger, complex assets.
- More validations around invalid inputs, and more.

Another key advantage, though? You can access Live Photos without requiring library access since it runs out of process. Given that this is available in SwiftUI too via the `PhotosPicker` – it's where you should start with fetching Live Photos:

```
// Xcode -> SwiftUI -> LivePhotosFig1View.swift
PhotosPicker("Choose Live Photo",
             selection: $livePhoto,
             maxSelectionCount: 1,
             selectionBehavior: .ordered,
             matching: .livePhotos,
             preferredItemEncoding: .automatic)
```

And in UIKit:

```
// Xcode -> UIKit -> LivePhotosFig1ViewController.swift

var config = PHPickerConfiguration()
config.filter = .livePhotos
config.selectionLimit = 1

let picker = PHPickerViewController(configuration: config)
picker.delegate = self
self.present(picker, animated: true)
```

In either case, you'll be given a `PHPickerResult` in UIKit, or a `PhotosPickerItem` in SwiftUI – and each of them have a similar way of handling loading. In fact, if you've come from the Drag and Drop chapter recently, then you might remember that in UIKit you leverage `ItemProvider` and in SwiftUI you typically use `Transferable`.

When we are dealing with a `PHPickerResult`, we can grab the item provider and load the Live Photo in directly:

```
// Xcode -> UIKit -> LivePhotosFig1ViewController.swift

guard let firstResult = results.first,
          firstResult.itemProvider.canLoadObject(ofClass:
PHLivePhoto.self) else {
    picker.dismiss(animated: true)
    return
}

firstResult.itemProvider.loadObject(ofClass: PHLivePhoto.self)
{ result
error in
    DispatchQueue.main.async {
        // Now we've got a Live Photo in `result`
        picker.dismiss(animated: true)
    }
```

```
}
```

In the case of SwiftUI, we turn to `Transferable`:

```
// Xcode -> SwiftUI -> LivePhotosFig1View.swift

selection.loadTransferable(type: PHLivePhoto.self) { result in
    switch result {
    case .success(let fetchedLivePhoto):
        print("Photo retrieved: \
(fetchedLivePhoto.debugDescription)")
    case .failure(_):
        print("Error occurred.")
    }
}
```

This method is the quickest and easiest way to get a Live Photo, and it's the one I recommend. That, and by using `PHAssetResource`, you can get at their contents if you need to by passing the found resources into `PHAssetResourceManager`. This should cover your Live Photo bases.

If you didn't want to go this route for whatever reason, you'd have to use `UIImagePickerController` and specify both `kUTTypeImage` and `kUTTypeLivePhoto` in its `mediaTypes` array. Then, the `editingInfo` dictionary will contain a `livePhoto` key containing a `PHLivePhoto`.

Further, if you get a hold of a `PHAsset` that's a Live Photo, you can fetch it using that, too. By using `PHImageManager`, you can setup a fetch request:

```
let fetchOps = PHLivePhotoRequestOptions()
fetchOps.isNetworkAccessAllowed = true
fetchOps.deliveryMode = .highQualityFormat
fetchOps.progressHandler = { progress, error, stop, info in
    // Update progress in UI
}
```

```
// This would come from the picker
let asset = PHAsset()

PHImageManager.default().requestLivePhoto(for: asset,
                targetSize: .init(width: 400, height: 400),
                contentMode: .aspectFit,
                options: fetchOps) { livePhoto, info in
    // Display Live Photo
}
```

Showing Live Photos

Fortunately, displaying Live Photos is relatively straightforward. One reason is because there is only one class to do it, and that's `PHLivePhotoView`. Using it is simple, simply assign to its `livePhoto` property, and you're done. It works nearly identical to UIKit's `UIImageView`.

We've seen a few snippets from the sample project, but here it is end-to-end using UIKit:

```
// Xcode -> UIKit -> LivePhotosFig1View.swift

class LivePhotosFig1ViewController: BaseSampleViewController {
    private lazy var photosView: PHLivePhotoView = {
        let pv = PHLivePhotoView(frame: view.bounds)
        return pv
    }()

    override func viewDidLoad() {
        super.viewDidLoad()

        let openPickerAC: UIAction = .init(image: .init(systemName:
"plus.circle.fill")) { _ in
            var config = PHPickerConfiguration()
            config.filter = .livePhotos
            config.selectionLimit = 1
```

```
            let picker = PHPickerViewController(configuration:
config)
            picker.delegate = self
            self.present(picker, animated: true)
        }

        navigationItem.rightBarButtonItem = .init(systemItem: .add,
primaryAction: openPickerAC, menu: nil)

        view.addSubview(photosView)
    }
}
extension LivePhotosFig1ViewController:
PHPickerViewControllerDelegate {
    func picker(_ picker: PHPickerViewController, didFinishPicking
results: [PHPickerResult]) {
        guard let firstResult = results.first,
                firstResult.itemProvider.canLoadObject(ofClass:
PHLivePhoto.self) else {
            picker.dismiss(animated: true)
            return
        }

        firstResult.itemProvider.loadObject(ofClass:
PHLivePhoto.self) { result, error in
            DispatchQueue.main.async {
                self.photosView.livePhoto = result as? PHLivePhoto
                picker.dismiss(animated: true)
            }
        }
    }
}
```

As you can see, once a selection is made we simply assign to the photo view with the `PHLivePhoto`, and we're set. In SwiftUI, there is no `LivePhotoView` equivalent, so we'd need to wrap it and make our own:

```
// Xcode -> SwiftUI -> LivePhotoView.swift
```

```
struct LivePhotoView: UIViewRepresentable {
    @Binding var livePhoto: PHLivePhoto?

    func makeUIView(context: Context) -> PHLivePhotoView {
        let phView = PHLivePhotoView()
        phView.delegate = context.coordinator
        return phView
    }

    func updateUIView(_ uiView: PHLivePhotoView, context: Context) {
        uiView.livePhoto = livePhoto
    }

    func makeCoordinator() -> Coordinator {
        Coordinator(parent: self)
    }

    class Coordinator: NSObject, PHLivePhotoViewDelegate {
        let parent: LivePhotoView

        init(parent: LivePhotoView) {
            self.parent = parent
        }

        func livePhotoView(_ livePhotoView: PHLivePhotoView,
canBeginPlaybackWith playbackStyle: PHLivePhotoViewPlaybackStyle) ->
{
            print("Live Photo can begin")
            return true
        }

        func livePhotoView(_ livePhotoView: PHLivePhotoView,
willBeginPlaybackWith playbackStyle: PHLivePhotoViewPlaybackStyle) {
            print("Live Photo will begin")
        }

        func livePhotoView(_ livePhotoView: PHLivePhotoView,
didEndPlaybackWith playbackStyle: PHLivePhotoViewPlaybackStyle) {
```

```
                print("Live Photo did end")
            }
        }
}
```

However, you may notice a few things, no matter if you're using UIKit or SwiftUI–namely, the result just looks like a regular image. While that's true, the user can tap down and hold, just like in the Photos app, and the Live Photo will play. There are ways to make this more obvious, though, and we'll look at those now in the "Tips" section.

Tips

The badge

One way to make Live Photo support more robust is to include the Live Photo badge you might've seen in Photos:



The `PHLivePhotoView` doesn't give us this out of the box – but we can access it. Here's how in SwiftUI:

```
// Xcode -> SwiftUI -> LivePhotoBadgeView
struct LivePhotoBadgeView: UIViewRepresentable {
    let livePhotoEnabled: Bool

    func makeUIView(context: Context) -> some UIView {
        let options: PHLivePhotoBadgeOptions =
livePhotoEnabled ? .overContent : .liveOff
        let badge = PHLivePhotoView.livePhotoBadgeImage(options:
options)
        let imageView = UIImageView(image: badge)

        return imageView
    }

    func updateUIView(_ uiView: UIViewType, context: Context) {
        // No op
    }
}
```

By using `PHLivePhotoBadgeOptions`, we can create a badge to show over our Live Photos so that users know they can press down to play them. If you've got this disabled for whatever reason, you can also pass `.liveOff` to get a different badge indicating playback isn't available.

Hint at Playback

Another nicety found in Photos on iOS is that when you swipe to a Live Photo, it hints at playback by playing a few frames. It's a wonderful effect, but unfortunately that's not built-in behavior, but we can reconstruct similar effects ourselves.

Thankfully, `PHLivePhotoView` has several hooks to control playback. For example, when one is being scrolled into view – we could play back just a *little* of its contents:

```
livePhotoView.startPlayback(with: .hint)
```

By starting playback with `.hint`, we can get an effect as seen in Photos. It plays a bit of the motion without any sound. If you want full playback, of course, you could just pass in `.full` here instead.

Remember, though, by default, any Live Photo view will support playback via its gesture. Though, I should mention you have access to that too, via `playbackGestureRecognizer`, and you could move it to an entirely different view if you saw fit.

Photo View Delegate

Earlier, when I wrapped the `PHLivePhotoView` in `LivePhotoView` – I also included a `Coorindator` object. This is because the `PHLivePhotoView` has various hooks into its lifecycle you can react to, if needed. This includes knowing the Live Photo started, stopped and is about to start. Plus, you can even add more time that's required to press down to kick off playback if you needed too.

Inspecting Live Photos

Earlier, I mentioned it's possible to interact with parts of a Live Photo. A common use case might be an image editing or social media app, which might require all parts of the Live Photo for uploading to later represent it elsewhere. Otherwise, it would just be a static image.

To get at these, here's a more robust code sample of what it would look like:

```
private func inspectLivePhoto() {
```

```
    guard let livePhoto = phLivePhoto else {
        return
    }

    let components = PHAssetResource.assetResources(for: livePhoto)

    let fetchOptions = PHAssetResourceRequestOptions()
    fetchOptions.isNetworkAccessAllowed = true

    components.forEach {
        PHAssetResourceManager.default().requestData(for: $0,
options: fetchOptions) { data in
            print("Handle data")
        } completionHandler: { error in
            print("Handle error")
        }
    }
}
```

Within the resources, you'll find a video via `PHAssetResourceType.pairedVideo` and the photo itself. Then, you can operate on each of them as you see fit (i.e. upload them to your server).

Bonus: Web Support

Yes – this is not a book series over web development in any form. But, solely for information purposes, I wanted to point out that Apple also provides a Javascript library to work with LivePhotos on the web. It's called LivePhotosKit JS and you can take a look at it here.

To use it, you can either grab it via Apple's content distribution network using a `<script>` tag: `<script src="https://cdn.apple-livephotoskit.com/lpk/1/livephotoskit.js"></script>` or using Node Package Manager:

```
npm install --save livephotoskit
```

Three Key Takeaways

1. Live Photos are a proprietary media format only on Apple platforms.
2. Find novel ways to incorporate them into your apps.
3. At the very least, properly support their playback.

Machine Learning

Machine Learning has taken over the iOS platform. What once seemed like a nascent technology that was teetering into a buzzword-like space is now a bonafide powerhouse that enables so much of what iOS does today. From recognizing recurring people in your photos and identifying them to detecting objects such as doors or sounds like a baby crying – all of this is powered by machine learning in some form.

Apple has built most of this in-house, and as a developer – fortunately it's not locked away from us. There are several, numerous APIs available. In fact, it's quite intimidating to step back and see what is out there to use: Core ML, Create ML, model files, CLI tools to convert data inputs into different formats, PyTorch – it can be a bit much to know where to start.

So in this chapter, I'm going to present a few different APIs that fall under the "Machine Learning" banner. The hope is that by surveying what is out there, it'll spark some creativity on your end to find the right applications for it. I think you'll find that just about any app could benefit from Machine Learning features, such as recognizing sound, augmenting photos and more.

Let's dive in.

How it Works

This chapter is a bit unique in its setup. Instead of showing how one API or frameworks works and then some tips, I'm going to instead introduce plenty of topics and show some examples of how they could be used. Hopefully, this will get your creative juices flowing and you can start to think about how to apply these APIs to your work.

And that's the thing – using machine learning on iOS isn't as simple as something like this : `import MachineLearning`. In fact, it's the opposite, there are several frameworks which all fall under the machine learning banner. Some of them are similar in API design, some are unique and unlike anything else. But, they all share the common underpinnings of Core ML and the technologies it's built on top of. Machine Learning on iOS is a large world, machine learning in general is even larger. Countless books could be dedicated to the topic.

So, we're going to make that large world a bit more understandable and *actionable* to us – I will give you a good overview of the things we might actually use in an app. To that end, we'll tackle machine learning on iOS by taking a look at the core categories they fall under:

- **Core ML**: A framework that allows developers to use machine learning models in our apps, and convert other ones into the Core ML format. This is essentially what powers most everything machine learning related on Apple platforms.
- **Create ML**: An application you can run on macOS to train new models to use with Core ML, as well as a framework in of itself.

- **Vision**: APIs to help you process and understand images or video by using computer vision.
- **Natural Language**: APIs to understand text in various interesting ways, such as classifying words.
- **Speech**: APIs to recognize speech patterns, sounds, and saliency features.
- **Sound**: APIs to recognize sounds as a particular class, such as a phone ringing or a baby crying.

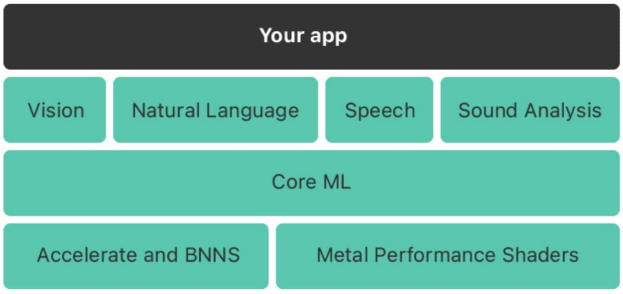
For each of them, we'll see what capabilities they afford to us to use within our apps. Let's start with the framework that powers it all, Core ML.

Core ML

This is *the* framework when it comes to machine learning on Apple platforms (not just iOS). Its primary role is to integrate machine learning models into apps. It applies its machine learning algorithms to a set of *training data* to produce a *model*. And, that's really the big takeaway here: Core ML takes inputs and results in an output.

Once the model is created, then it's ready to be used in apps. This is how Apple has been able to offer so many machine learnings APIs, they've created several models to power them all:

```
Apple's Machine Learning Stack
> Core ML powers it all.
```



Core ML itself is builds on top of the Accelerate and BNNS frameworks, as well as incorporating Metal Performance Shaders. By using these technologies, it's able to run very efficiently.

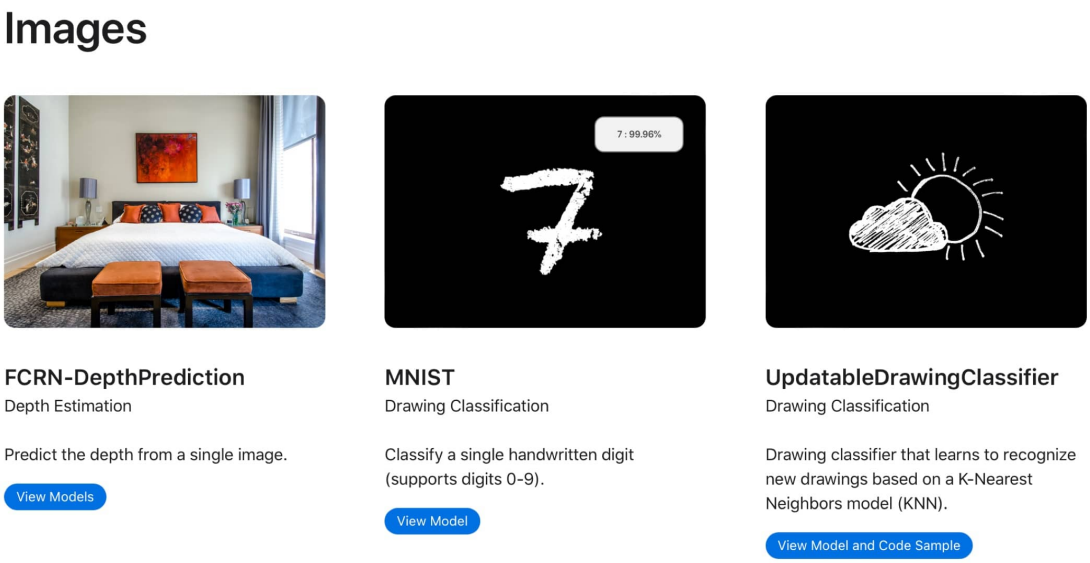
Impressive though they are, they aren't the *only* reason why they have a high degree of efficiency. Apple prides itself on running most (but not quite all) of its machine learning models on device. This offers several advantages:

- **Privacy**: The data the model will train on is often supplied by you. Think about it — how can Photos know when a certain image contains a family member? It's trained up a model to recognize that person's features. When this occurs on device and not on Apple's servers, you maintain privacy while enjoying the benefits of machine learning.
- **It's quick**: By eliminating network calls to perform analysis' required by machine learning models, things are just snappier. If the models were hosted

on a server somewhere, things wouldn't be as fast – and worse, it introduces another failure point. It wouldn't feel great, continuing with our Photos analysis example, if that feature didn't work if you had poor network conditions.

Apple offers several ready-made models to use right out of the box. These models will generate an interface for you in either Swift or Objective-C for use in your app. Though, I should mention – usually you don't have to worry about the model at all. It's often abstracted away from you from an API perspective.

It's good to be aware of how this all fits together, though. To that end, there are models to detect letters, answer questions based on some supplied content and more. For example, here are some of the ones on offer to examine images:

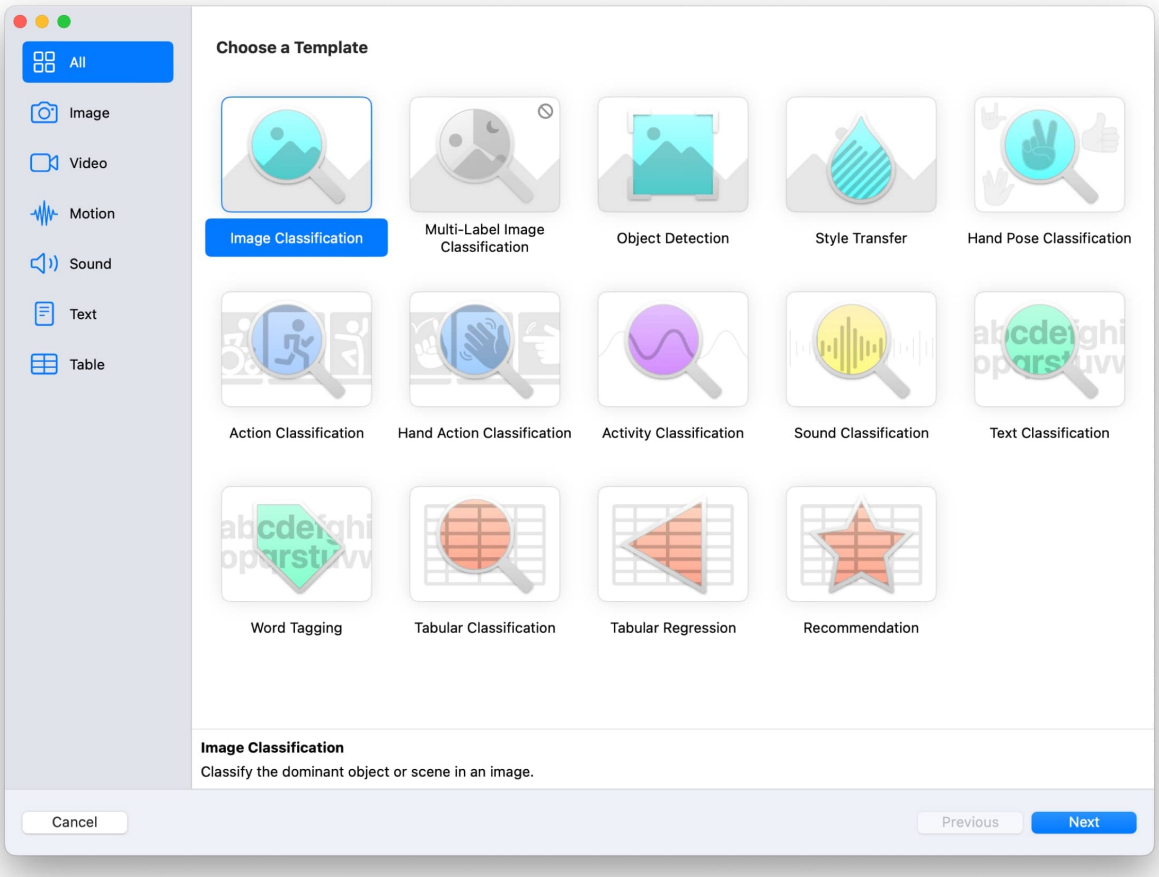


Of course, being an Apple technology – Core ML enjoys first class support within Xcode, too. You can use sample data files, encrypt models, get performance reports and profile Core ML APIs using Instruments.

Create ML

On the other hand, what if you need something outside those pre-trained models? Perhaps you've got your own unique use case. If you've got the data to train up the model, then Create ML can help you create one. Create ML can guide you from start to finish to create all kinds of custom models, and it'll export them into a format that Core ML understands.

It's also two parts, technically. Create ML is a standalone app on macOS to help you create models. It's also a framework, too.



As you can see from the image, there are several templates you can use to create a custom model. By providing the data to train the model against, you'll have all that you need to get started. Create ML is extremely capable – I suggest you open it and play around with one of the templates to get a feel for it.

Getting training data in is seamless as well. By using Continuity with your iPhone, you can provide video or audio assets to use as training data. Plus, you can save, resume or pause model training – you don't have to do it all in one shot. Create ML, in essence, has democratized the process of creating custom machine learning models. Anybody can do it now, forgoing what used to a delicate dance of installing third-party packages, modules, and processes.

Now, let's discuss the four pillars that are powered by Core ML.

Vision

Vision applications for machine learning is probably the most recognizable flavor of the four (Vision, Natural Language, Speech and Sound). Using frameworks such as Vision and VisionKit, you can use several under-the-hood models to perform plenty of interesting tasks. If you need to process, change or analyze video or images – the Vision framework is a perfect choice to use.

So, what kinds of things can we do here? Here's a few:

- **Classification**: Identify certain aspects or general content in an image as a whole.
- **Saliency**: Quantify the key parts of an image. Basically – it answers the question of where are people most likely to focus when they look at a given image.
- **Alignment**: Manage alignment of images.

- **Similarity**: How close in subject matter are two images?
- **Detection**: Label and identify objects within a photo. This is different from classification – one deals with the image as an entire unit, this one finds parts found within an specific image.
- **Tracking**: Find and track objects within a video.
- **Trajectory Detection**: Identify the trajectory of objects in motion from a video.
- **Contour Detection**: Traces the edges of objects in images and videos – kind of like a "magic lasso" in photo editing software.
- **Text Detection**: Find and identify regions of text.
- **Text Recognition**: Similar to the above, but this also extracts the text in addition to simply identifying their presence.
- **Face Detection**: Just like it sounds – find faces in images or videos.
- **Face tracking**: Again, similar to the above – but this tracks faces in real time.
- **Face Landmarks**: Finds facial features in an image, such as where the nose or eyes are.
- **Face Capture Quality**: Compares facial capture quality between a set of images.
- **Human Body Detection**: Finds regions that have, well, humans in them.
- **Body Poses**: Detects landmarks on people in images and videos.
- **Hand Poses**: The same as above, but unique to hand poses. You could use this to detect something like sign language, for example.
- **Animal Detection**: Finds the pets! Limited to cats and dogs.
- **Barcodes**: Finds and recognizes any barcode in an image.
- **Rectangular Detection**: Finds rectangular regions in photos.

- **Horizon Detection**: Analyzes where the horizon angle lies within a photo.
- **Optical Flow**: Analyzes the pattern of motion of objects between video frames.
- **Person Segmentation**: Generates a matte image for a person in an image — basically, it can pull out a person from a photo. This is similar to how the subject matter "lift" works on photos, where you can tap and hold on a person to lift it from a photo in your camera roll.
- **Document Detection**: Finds rectangular regions in images that have text in them.

As you can see, that is a…large list. Perhaps you can see why I think it's more effective to gain a general understanding of what's out there in terms of machine learning instead of diving into one particular topic. When you know what you *can* do, then you can dive into the *how* later on.

Natural Language

The Natural Language framework has many uses for handling bodies of text. For example, what words are in a sentence? Which are nouns or verbs? Here are some specific capabilities:

- **Tokenization**: Finds and loops through the words in strings.
- **Language Identification**: Recognize the language in text.
- **Named Entity Recognition**: This uses a linguistic tagger to name entities found within a string. For example, if you analyzed this sentence: "This book is popular in Missouri", it might say that "book" is a product and "Missouri" is a location.

- **Parts of Speech**: Identify verbs, sounds, adjectives and other parts of speech in a string.
- **Word Embedding**: This finds similar words or string within a proximity of their vectors. Basically, how close together are similar words?
- **Sentence Embedding**: The same as above, but this time – it operates on entire sentences.
- **Sentiment Analysis**: Attempts to determine the sentiment of a text – is it positive, negative or neutral?

Speech

One of the "smallest" of the four, but no less useful – the Speech framework allows you to recognize speech on any type of audio (live or prerecorded) to create transcriptions, confidence levels and alternative interpretations.

Sound

Finally, we've got the Sound Analysis framework. Like the Speech framework, it's also focused on one job: determining particular sounds in audio. For example, is a baby crying? Is an audience laughing?

Okay, so that's our tour of machine learning on iOS. As expected, it's a big topic with numerous applications. To get a feel for them, let's look at some code samples from each of the four primary sectors (Vision, Natural Language, Speech and Sound).

Examples

Vision

A lot of Vision's setup is the same. You'll create a handler, `VNImageRequestHandler`, and you'll load up that handler with different types of requests. From there, you'll execute it, look for any errors and then dispatch back to the main queue to do any further interface work as a result.

For example, consider `MachineLearningFig1ViewController.swift`. Here, I use Vision to detect my face in my Twitter avatar, and draw a rectangle around it:

```
// Xcode -> UIKit -> MachineLearningFig1ViewController.swift

private func performAnalysis() {
    guard let img = imageView.image, let liftAssetCGImage =
img.cgImage else {
        return
    }

    let imageRequestHandler = VNImageRequestHandler(cgImage:
liftAssetCGImage)
    let imageViewRect = imageView.bounds

    let faceDetectRequest = VNDetectFaceRectanglesRequest { req,
error in
        guard error == nil, let result = req.results?.first as?
VNFaceObservation else {
            print(error!.localizedDescription)
            return
        }

        let faceRect =
VNImageRectForNormalizedRect(result.boundingBox,

Int(imageViewRect.size.width),

Int(imageViewRect.size.height))

        DispatchQueue.main.async {
            let faceMark = UIView(frame: faceRect)
```

```
                faceMark.layer.borderColor = UIColor.red.cgColor
                faceMark.layer.borderWidth = 4.0

                self.imageView.addSubview(faceMark)
            }
        }

        do {
            try imageRequestHandler.perform([faceDetectRequest])
        } catch {
            print(error.localizedDescription)
        }
}
```

A few things to keep in mind, and these tips generally apply to using the Vision framework as a whole. The first is, as I mentioned above – you need to dispatch back to the main queue to do any interface work. In fact, if there is anything that would run on the main queue at all – you need to put those into a variable ahead of time. That's why I set the frame of the image view before the request:

```
let imageViewRect = imageView.bounds

// Then do request
```

If you didn't, you'd see the console light up with the main thread checker letting you know you're accessing interface properties off the main queue. Remember, both SwiftUI and UIKit run on the main queue. It also makes sense that Vision's requests *don't*, though. These can be expensive operations, so it's a good thing it offloads that work. You simply pop back to main when you're done with any task as we've done above:

```
DispatchQueue.main.async {
    let faceMark = UIView(frame: faceRect)
    faceMark.layer.borderColor = UIColor.red.cgColor
    faceMark.layer.borderWidth = 4.0
```

```
        self.imageView.addSubview(faceMark)
    }
```

Next, a lot of requests that return back some sort of coordinates you want to operate on are housed within a `boundingBox` property, which itself is a `CGRect`. However, you're likely used to the coordinate system that most interface frameworks deal in, that is – 0,0 is the top left of the control in question.

The bounding box doesn't work like this, and you'll be using it a lot, since most requests that have one inherit from `VNDetectedObjectObservation`.

If you look at the docs for it, you'll see that its coordinate system works a little differently:

> *The bounding box of the detected object. The coordinates are normalized to the dimensions of the processed image, with the origin at the image's lower-left corner.*

So, 0,0 here is the lower-left of a control. And its width and height are expressed via ratios. For example, half the width would be expressed as 0.5 instead of a raw point value. The idea is that you need to take this normalized bounding box and express it terms that make sense for your app. I've seen far too many solutions which write a clever function to make this kind of translation, and I'm telling you today – you don't have to do this.

Vision accounts for this scenario, and they've supplied a built-in function to do the conversion for you. In the sample above, that's what `VNImageRectForNormalizedRect` is doing. We give it the bounding box we retrieved from the request, and then give it the image's size, and it'll take care of it for us.

Take note, though, saying we give it the "image" size is a little misleading. For example, many apps want to do this kind of request and then apply the results to some `UIImageView`. So, in this case, what we *really* intend to use is the image view's size, *not* the actual pixel size of the image. Remember, most images are either smaller or larger than the image view or `Image` they are put into – they are seldom the same size unless you downsample the image yourself.

With that considered – we're able to put the red rectangle around the face in the image:

```
let faceRect = VNImageRectForNormalizedRect(result.boundingBox,

Int(imageViewRect.size.width),

Int(imageViewRect.size.height))

DispatchQueue.main.async {
    let faceMark = UIView(frame: faceRect)
    faceMark.layer.borderColor = UIColor.red.cgColor
    faceMark.layer.borderWidth = 4.0

    self.imageView.addSubview(faceMark)
}
```

Natural Language

The Natural Language is small in comparison to the other frameworks, and I don't mean that as a pejorative, either. It's focused on the task at hand, and only has what you'd need to identify a language in some text. In `MachineLearning-Fig1View.swift` – I've provided the same string in a few different languages. Using the Natural Language framework, we can identify the language for each of them:

```
private let testOne: String = "Please enjoy the best-in-class iOS
app book series!"
```

```
private let testTwo: String = "Per favore, goditi la migliore serie
di libri di app per iOS!"
private let testThree: String = "请享受一流的 iOS 应用书籍系列!"
```

The setup of the view is simple – each version of that string is shown along with its given language below it (once analysis is completed):

```
var body: some View {
    VStack(alignment: .leading) {
        Text(testOne)
        Text("Result: \(resOne)")
            .font(.caption)
        Divider()
            .padding(.bottom, 16)
        Text(testTwo)
        Text("Result: \(resTwo)")
            .font(.caption)
        Divider()
            .padding(.bottom, 16)
        Text(testThree)
        Text("Result: \(resThree)")
            .font(.caption)
        Spacer()
    }
    .padding()
    .toolbar {
        ToolbarItem(placement: .navigationBarTrailing) {
            Button("Analyze") {
                performAnalysis()
            }
        }
    }
}
```

To process these strings, we simply create an instance of `NLLanguageRecognizer` and call its `processString()` function. That will result in an optional `NLLanguage` assigned back to the instance of the recognizer. You can either display the language

identifier, or more likely – you can use `Locale` to get a more human-readable string from it (i.e., "English" instead of "en").

```
private func performAnalysis() {
    let recognizer = NLLanguageRecognizer()

    for (idx, value) in [testOne, testTwo, testThree].enumerated() {
        recognizer.processString(value)
        let languageID = recognizer.dominantLanguage?.rawValue ?? ""

        let locale = Locale(identifier: languageID)
        let result = locale.localizedString(forIdentifier:
languageID) ?? "N/A"

        switch idx {
        case 0:
            resOne = result
        case 1:
            resTwo = result
        case 2:
            resThree = result
        default:
            print("Unrecognized index.")
        }

        recognizer.reset()
    }
}
```

You can make the process more efficient if you know the languages you're looking for. For example, if you knew the languages you might be matching against, you can constrain the `processString()` function to *only* look for those – making things a bit quicker:

```
recognizer.languageConstraints = [.korean, .polish]
```

Speech

Similar to the Natural Language framework, the Speech framework mostly revolves around one class – `SFSpeechRecognizer`. Using it, you can recognize or transcribe audio in your app. This is similar to how voice dictation works on the system keyboard when you use it. Several languages are supported – but each instance of a recognizer only operates on one language at a time.

Of note – this is one instance of Apple's machine learning suite that *might* operate via Apple's servers. In short, it doesn't always happen on-device. Some languages work locally, but not all of them. It's better to assume that you'll require a network connection. In fact, when you implement the permissions request code and get a system alert – it'll mention this in the text, "This text is processed on Apple's servers."

For this reason – unlike the other machine learning APIs, you'll need to get consent from the user first. Much like other privacy alerts, the process works the same way with the Speech framework. First, you'll need to edit your app's .plist and add the `NSSpeechRecognitionUsageDescription` key. Give it a good description that plainly describes what you do with the audio.

Before I dive into implementation, I want to point out that there are two primary ways to transcribe audio: Live speech or audio, or via a URL asset. In our example, we'll look at live audio transcribing. Note that this requires you to run this on device – it won't work in the simulator. What we'll build is a view that transcirbes any text you speak, and it'll show that transcription in a `Text` control.

In the code sample, there is quite a bit of setup to get live audio capture up and running. Though, the main idea is similar – you create a request and give it a task to analyze. To that end, here is the main idea behind the code sample in `MachineLearningFig2View.swift`:

```
// Xcode -> SwiftUI -> MachineLearningFig2View.swift

// In `SpeechEngine`
private let recognizer = SFSpeechRecognizer()
private var task: SFSpeechRecognitionTask! = nil
private lazy var request: SFSpeechAudioBufferRecognitionRequest = {
    let req = SFSpeechAudioBufferRecognitionRequest()
    req.shouldReportPartialResults = true
    return req
}()

// Abbreviated - see `transcribeText()` for full sample
task = recognizer.recognitionTask(with: request) { result, error in
    var finished = false

    if let analyzedResult = result {
        self.publishedText =
analyzedResult.bestTranscription.formattedString
        finished = analyzedResult.isFinal
    }

    if error != nil || finished {
        self.audioEngine.stop()
        audioNode.removeTap(onBus: 0)

        var reason: String = "Finished."
        if let issue = error {
            reason = issue.localizedDescription
        }

        self.tearDown(with: reason)
    }
}
```

By creating a `SFSpeechRecognizer` and feeding it a `SFSpeechAudioBuffer-RecognitionRequest` (in this case, one suited for live feeds), we can start streaming in results. When it's finished, or there was an error – we respond to that and up-

date the interface. When you run this on device, once you get permissions to record audio and tap “Start Recording” – everything you say will show up in the text view.

There are plenty of interesting applications for this API. You can transcribe videos on-device, make those “caption generator” apps and more. It takes a bit to stand up in comparison to some other machine learning APIs, but it does follow some of the same conventions of creating a request, taking that request in and creating a task from it.

Sound

Finally, we end up in the sound domain – and the Sound Analysis framework. It, too, has one core job right now – and that’s sound classification. Out of the box, it boasts an impressive model that can recognize over 300 sounds. Or, you can create your own Core ML model to recognize other types of sounds, too.



If you're curious which sounds you can detect with the classifier, you can query them all using `SNClassifySoundRequest`. All it needs is the identifier of a classifier to use.

Currently, there's one such classifier that ships on-device. As such, to query it, use the extension Apple has provided on `SNClassifierIdentifier`:

```
extension SNClassifierIdentifier {
    /**
     @brief Denotes the first edition of the Apple-provided sound
classifier.
     */
    @available(iOS 15.0, *)
    public static let version1: SNClassifierIdentifier
}
```

From there, supply the sound request object and you're ready to start processing:

```
if let req = try?
SNClassifySoundRequest(classifierIdentifier: .version1) {
    let sortedSounds: [String] = req.knownClassifications.sorted()
    sortedSounds.forEach {
        print($0)
    }
}


// As of iOS 15, you get something like:
accordion
acoustic_guitar
air_conditioner
air_horn
aircraft
airplane
alarm_clock
ambulance_siren
applause
artillery_fire
babble
baby_crying
baby_laughter
bagpipes
```

```
banjo
...and on
```

You might be thinking that those titles don't lend themselves well to interfaces. If you need to make them user facing, I spotted some code Apple has used which uses a local bundle to get localized versions of them. But, I should note that I'm uncertain if this is *officially* supported (i.e., this bundle could move, change or be deleted in the future).

Regardless:

```
static func displayNameForLabel(_ label: String) -> String {
    let localizationTable = "SoundNames"
    let unlocalized = label
                      .replacingOccurrences(of: "_", with: " ")
                      .capitalized

    return Bundle
           .main
           .localizedString(forKey: unlocalized,
                            value: unlocalized,
                            table: localizationTable)
}
```

Then, we can do translations like this:

```
let identifierName = "acoustic_guitar"
let uiName = displayNameForLabel(identifierName) // Acoustic Guitar
```

So, we know what sounds we can query for. What's next? The flow is fairly easy to follow, is similar to the previous ones and looks like this:

1. Get a hold of a sound classifier.
2. Figure out what sounds you want to listen for.
3. Identify the audio file to use.

4. Create an observer conforming to `SNResultsObserving`.

Let's assume we've got a local audio – we could use that for step 3. I've supplied a `.wav` file of audience applause in the sample app. If you were using one of your own, you'd simply provide it the analyzer:

```
let videoURL = getVideoURL()
let analyzer = try SNAudioFileAnalyzer(url: videoURL)
```

Step 4 is where the processing takes place. The `SNResultsObserving` will be the connection point for the code that wants to know which sounds happened where, and the processing tasks that find that information out. It has one required function:

```
func request(_ request: SNRequest,
             didProduce result: SNResult)
```

And, like most of Apple's API designs, it also has two optional functions you can use to know tangentially related information. In this case, when things failed or finished:

```
optional func request(_ request: SNRequest, didFailWithError error:
Error)
optional func requestDidComplete(_ request: SNRequest)
```

The idea here is to handle the results of the observation requests. Since we're only concerned with applause sounds, the whole process might look like this:

```
class ApplauseDetector: NSObject, SNResultsObserving {
    private let applauseID: String = "applause"
    @Published var lastestTimeStamp: CMTime = .invalid

    func request(_ request: SNRequest, didProduce result: SNResult)
{
        guard let result = result as? SNClassificationResult,
              let _ = result.classification(forIdentifier:
applauseID) else {
```

```
            return
        }

        lastestTimeStamp = result.timeRange.start
    }
}
```

Then, putting it altogether, we could find timestamps for applause from any audio file. See `processAudio()` and `MachineLearningFig3View.swift` to see it all ina action:

```
class ApplauseDetector: NSObject, ObservableObject,
SNResultsObserving {
    private let applauseID: String = "applause"
    private var subs: [AnyCancellable] = []
    @Published private(set) var timestamps: [Double] =
[]
    let audioURL: URL

    init(audioURL: URL) {
        self.audioURL = audioURL
    }

    func processAudio() {
        guard let req = try?
SNClassifySoundReques
(classifierIdentifier: .version1),
              let analyzer = try?
SNAudioFileAnalyzer(url: audioURL)
else {
            return
        }

        // Add our classifier and result handler
        try? analyzer.add(req, withObserver: self)

        // Kick off detection
        analyzer.analyze()
```

```
    }

    // MARK: Observing Delegate

    func request(_ request: SNRequest, didProduce
result: SNResult) {
        guard let result = result as?
SNClassificationResult,
              let _ =
result.classification(forIdentifier:
applauseID) else {
            return
        }

        timestamps.append(result.timeRange.star
.seconds)
    }
}
```

Once you've got everything setup, you may find that you need to make some tweaks to get what you're after. Like many machine learning APIs, there is a confidence factor that you can play with:

```
let applause = result.classification(forIdentifier: applauseID)
let confidenceMet = applause.confidence > 0.7 // Or Whatever
```

Or, if you can give the request a duration the sound may occur, that'll help too:

```
// Four second window for applause
req.windowDuration = CMTime(seconds: 4, preferredTimescale: 1000)
```

In our case, there is no easy definition for how long applause is, so you can also use a list of ranges:

```
public enum SNTimeDurationConstraint {
    case enumeratedDurations([CMTime])
    case durationRange(CMTimeRange)
```

```
}
```

Further, if you don't have a set audio file, there is also an object for streaming audio and finding things on the fly. That's `SNAudioStreamAnalyzer`. You can think of it the same way that the Speech framework operates – there is an object to stream audio or use local audio.

Three Key Takeaways

1. There are several machine learning APIs on the iOS platform.
2. Getting a broad understand of what's available is key.
3. Don't "force" machine learning features into your app – instead, focus on finding compelling use cases related to your app.

Maps

Maps have robust support in iOS. Recently in SwiftUI, they've received updates for many powerful features such as Look Around. Whether you're showing a simple route to take on a walk, or creating a fully featured GPS app – it truly is an "if you can dream it, you can build it" type of situation.

Using Map APIs, we can display maps with realistic imagery, 3D perspectives and more. In addition, it's easy to add adornments to maps, too. Think of things like pins for specific points of interest – maybe parking spots or something similar. Or, you can hone in on specific geographic points and display them with elevation on the map to better emphasis them.

What I love about Maps on iOS is that it's easy to get started – it's just a few lines of code. The expressiveness of the API, especially in SwiftUI, embodies progressive disclosure design and opens up more power as you explore it. Let's get right to it and start with the framework that powers it all, MapKit.

How it Works

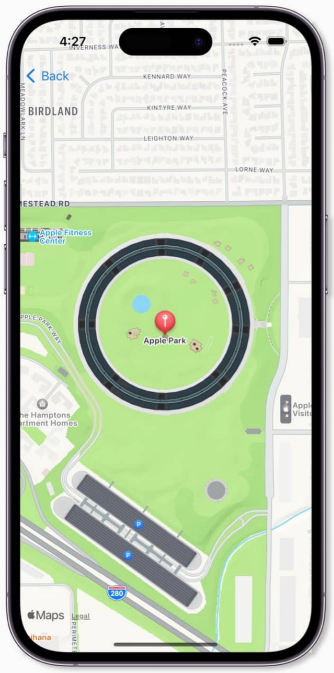
MapKit powers any Map experience on Apple devices. It's available on every platform – macOS, iPadOS, iOS, tvOS, watchOS, visionOS, and Mac Catalyst. In SwiftUI, you're primarily using the `Map` view. In iOS 17, `Map` became much more flexible, and has several API enhancements. As a result, we'll focus on those features and `Map`'s numerous improvements.

In addition, it now has a more natural result builder-based API to add things to a map. Hence, the progressive disclosure design I mentioned earlier. For example, if you wanted a map with a pin over Apple Park:

```
// Xcode -> SwiftUI -> MapFig1View.swift

struct MapFig1View: View {
    var body: some View {
        Map() {
            Marker("Apple Park",
                   coordinate: .applePark)
        }
    }
}
```

…and that is all the code required to get a fully-featured map in your app:

MapKit
> The `Map` view works with result builders to create an intuitive way to add markers and points of interest to any map.



In UIKit, you instead use a `MKMapView` – and we'd get a similar result as above:

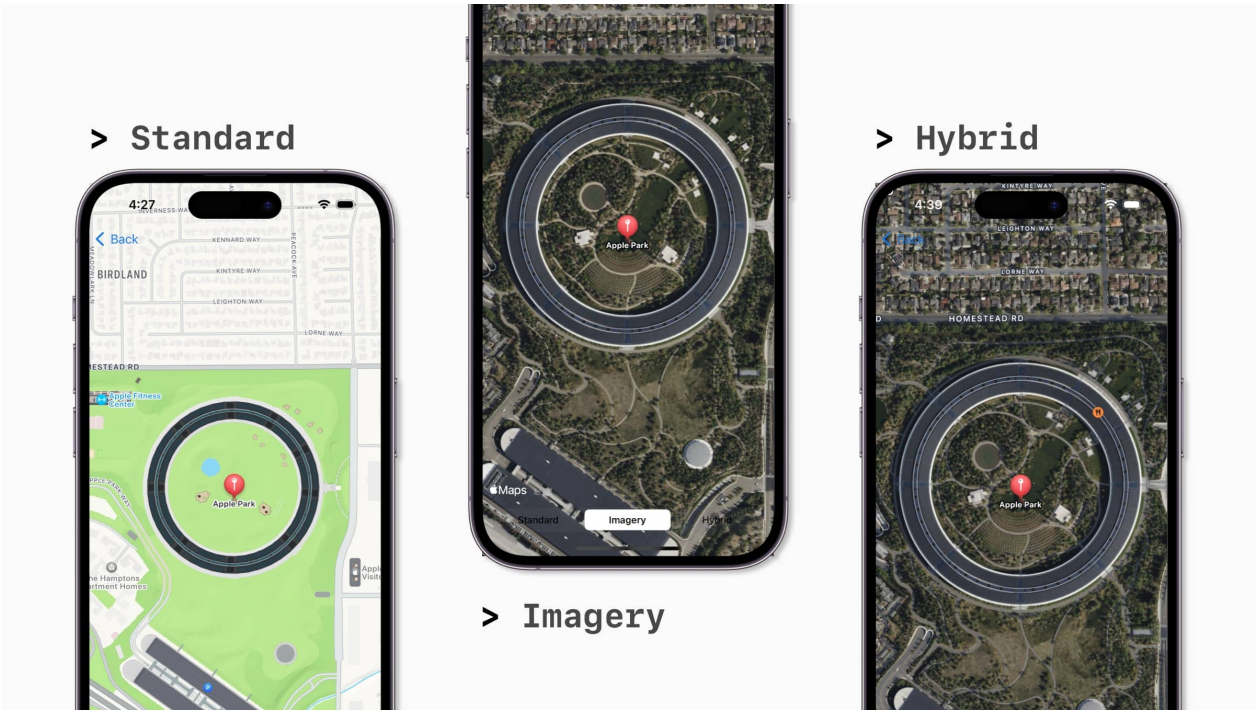
```
// Xcode -> UIKit -> MapFig1ViewController.swift

class MapFig1ViewController: BaseSampleViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

        let map = MKMapView(frame: view.bounds)
        view.addSubview(map)
        map.autoresizingMask = [.flexibleWidth,
                                .flexibleHeight]

        let ap = MKPointAnnotation()
        ap.title = "Apple Park"
        ap.coordinate = .applePark
        map.addAnnotation(ap)
    }
}
```

To change the look and feel of the map, there are different *emphasis* styles – and they are all are suited towards specific use cases. In UIKit, there are two primary ones – and those are the default and muted styles. The default style will be the one you probably use the most, with it – you get rich, saturated colors and it invites interactivity. The muted style presents a much less saturated color palette, and it's perfect for situations where the map is an ancillary part of the data showing – but not the main attraction.

In SwiftUI, we use a `MapStyle` struct along with the `.mapStyle` modifier. There, we've got three choices – each with rich customization options:

- **Standard**: The same as UIKit, a standard style that, to me, looks a bit like vector graphics. Rich, vibrant colors.
- **Imagery**: This renders the map using satellite imagery.
- **Hybrid**: The same as imagery, but it layers the roads and names using the standard style – giving you a good mix of both.

Here's what they look like:



You aren't constrained to any one selection, either. As we see in `MapFig2View`, you can switch them on the fly and SwiftUI will render the map again:

```
// Xcode -> SwiftUI -> MapFig2View.swift

struct MapFig2View: View {
    @State private var selection: Int = 0
    @State private var style: MapStyle = .standard
    var body: some View {
        Map() {
            Marker("Apple Park", coordinate:
.applePark)
        }
        .mapStyle(style)
        .safeAreaInset(edge: .bottom) {
            Picker("Map Style", selection:
$selection) {
                Text("Standard")
                    .tag(0)
                Text("Imagery")
```

```
                    .tag(1)
                Text("Hybrid")
                    .tag(2)
            }
            .pickerStyle(.segmented)
            .padding(.horizontal)
        }
        .onChange(of: selection) {
            switch selection {
            case 0:
                style = .standard
            case 1:
                style = .imagery
            case 2:
                style = .hybrid
            default:
                style = .standard
            }
        }
    }
}
```

Annotations

Annotations are an important part of any map. They call out specific points of interest, as defined by you, and they help people find what they're looking for by providing visual adornments over certain locations.

We've seen a few of these already – the pin over Apple Park represented one. When it comes to annotations, though, things work a lot differently here in UIKit and SwiftUI. SwiftUI, as I've mentioned – uses a result builder API to add markers.

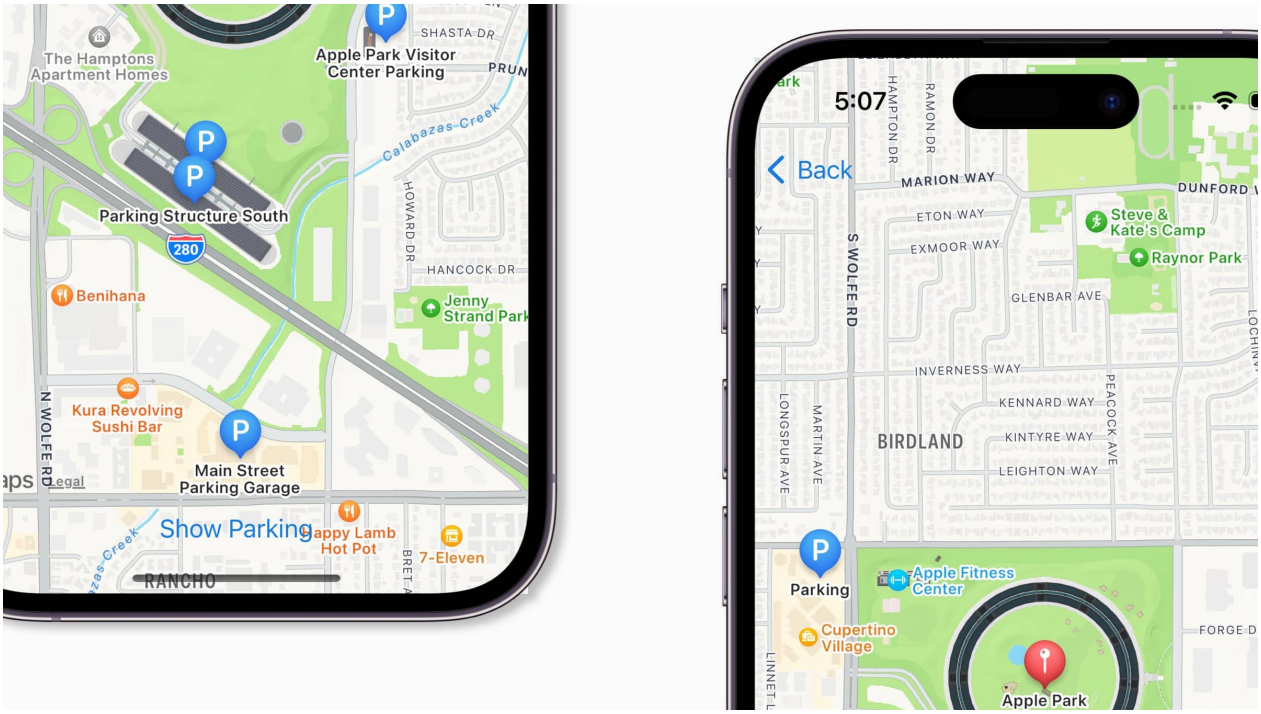
These `Marker` views are extremely versatile, too. They can be customized in several ways to bring attention to different parts of a map. Even better – sometimes the framework can customize them for you.

For example, let's say we wanted to find some parking near Apple Park – we could add `Marker` instances for each spot we find:

```
// Xcode -> SwiftUI -> MapFig3View.swift

struct MapFig3View: View {
    @State private var parking: [MKMapItem] = []

    var body: some View {
        Map() {
            Marker("Apple Park", coordinate: .applePark)

            ForEach(parking, id: \.self) { parking in
                Marker(item: parking)
            }
        }
        .toolbar {
            ToolbarItem(placement: .topBarTrailing) {
                Button("Show Parking",
                        systemImage: "parkingsign.circle.fill") {
                    showParking()
                }
            }
        }
    }

    // MARK: Private Functions

    private func showParking() {
        let request = MKLocalSearch.Request()
        request.naturalLanguageQuery = "Parking Spots"
        request.resultTypes = .pointOfInterest
        request.region = MKCoordinateRegion(center: .applePark,
                                            span: .searchSpan)

        Task {
            let search = MKLocalSearch(request: request)
            let response = try? await search.start()
```

```
                parking = response?.mapItems ?? []
            }
        }
    }
```

Once a search is performed, we'll not only get markers for each spot the framework found – but we also get a blue pin with a "P" added for us automatically. MapKit infers the type of information we're looking for, and in some cases – it'll automatically customize the markers as we're seeing here:



Take note of the `MKLocalSearch` found in `showParking()` method. Using its static `Request()` function, we can find points of interest and then use `Marker`'s initializer, which accepts a `MKMapItem` to display them on the map. That's what we've done here to display each result that we store in the `parking` array:

```
ForEach(parking, id: \.self) { parking in
    Marker(item: parking)
```

```
}
```

The great thing about MapKit is that a lot of it works between SwiftUI and UIKit without many changes required. For example, you use `MKMapItem` there as well – and the search code would remain essentially the same.

Of course, if we're showing annotations – we likely want to allow someone to tap on them as well. That's where a selection binding comes into play. The `Map` view accepts a binding to anything that conforms to `Hashable` – which means that so long as our marker represents the type we chose to use for selections, it'll all just automatically work.

That's a lot of words, so here's what it looks like in practice:

```
// Xcode -> SwiftUI -> MapFig4View.swift

@State private var selection: MKMapItem? = nil

var body: some View {
    Map(selection: $selection) {
        Marker("Apple Park", coordinate: .applePark)

        ForEach(parking, id: \.self) { parking in
            Marker(item: parking)
        }
    }
    .toolbar {
        ToolbarItem(placement: .topBarTrailing) {
            Button("Show Parking",
                        systemImage: "parkingsign.circle.fill") {
                showParking()
            }
         }
     }
}
```

Since our selection is a `MKMapItem` type, and our marker is also representing `MKMapItem` points – our selection binding will change based on which parking spot is tapped.

If you wanted to use something else, you could simply tag each view, too. This is great for cases where the type of data that the annotations are representing don't match one-to-one. For example, perhaps something is an arbitrary pin and another represents map items, like we've got here:

```
// Use an Int instead of map items
@State private var selection: Int? = nil

var body: some View {
    Map(selection: $selection) {
        Marker("Apple Park", coordinate: .applePark)
            .tag(0)
        ForEach(parking, id: \.self) { parkingSpot in
            Marker(item: parkingSpot)
                .tag(parking.firstIndex(of: parkingSpot) ?? 0)
        }
    }
}
```

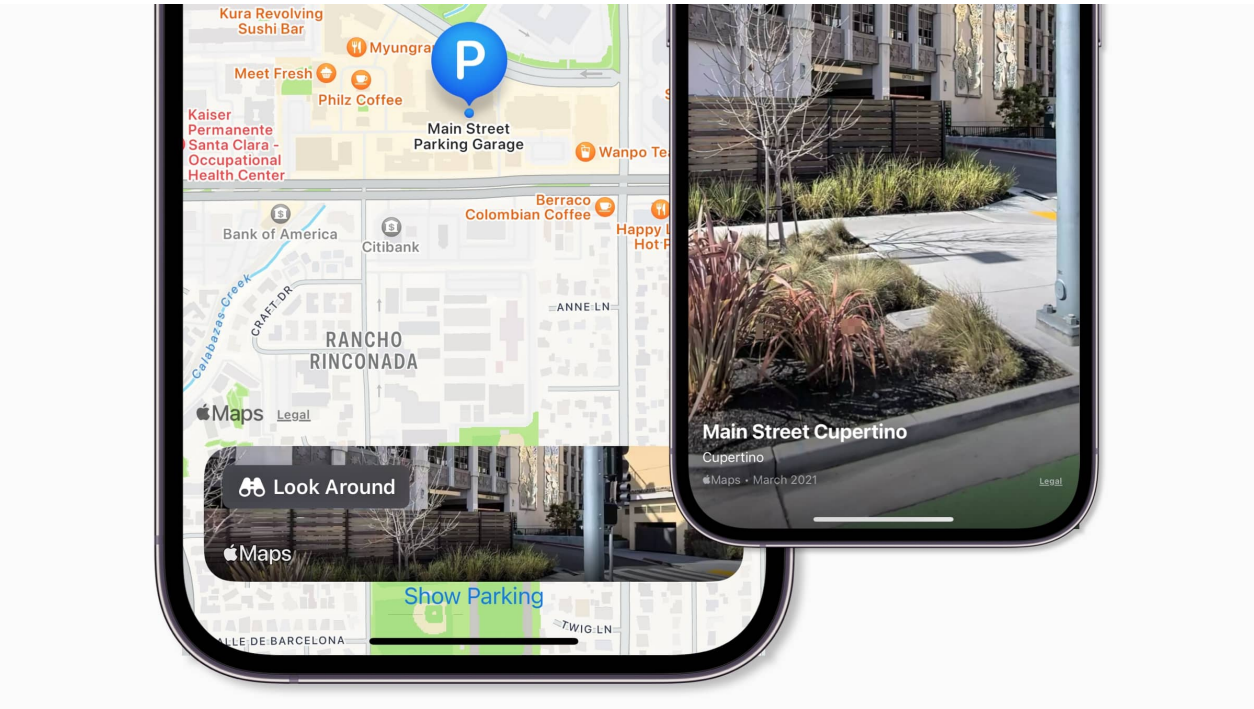
Tips

LookAround

Look Around is a nice little feature which shows what a certain location might look like from a first-person perspective. You can provide support for it in your app, too. By using `LookAroundPreview` – you can either show a small preview of it yourself in any

fashion, and you can allow it to fully expand into the immersive, full-screen view where users can tap around and move.

In `MapFig5View`, you can see we have a local state variable of type `MKLook-AroundScene`. When someone taps on a parking spot, we can see if the backing `MKMapItem` has Look Around support, and if it does – we'll show a small view at the bottom which people can expand:



We can run this by fetching the Look Around data by using `MKLookAroundScene-Request` – and when our selection binding changes, we'll fire it off:

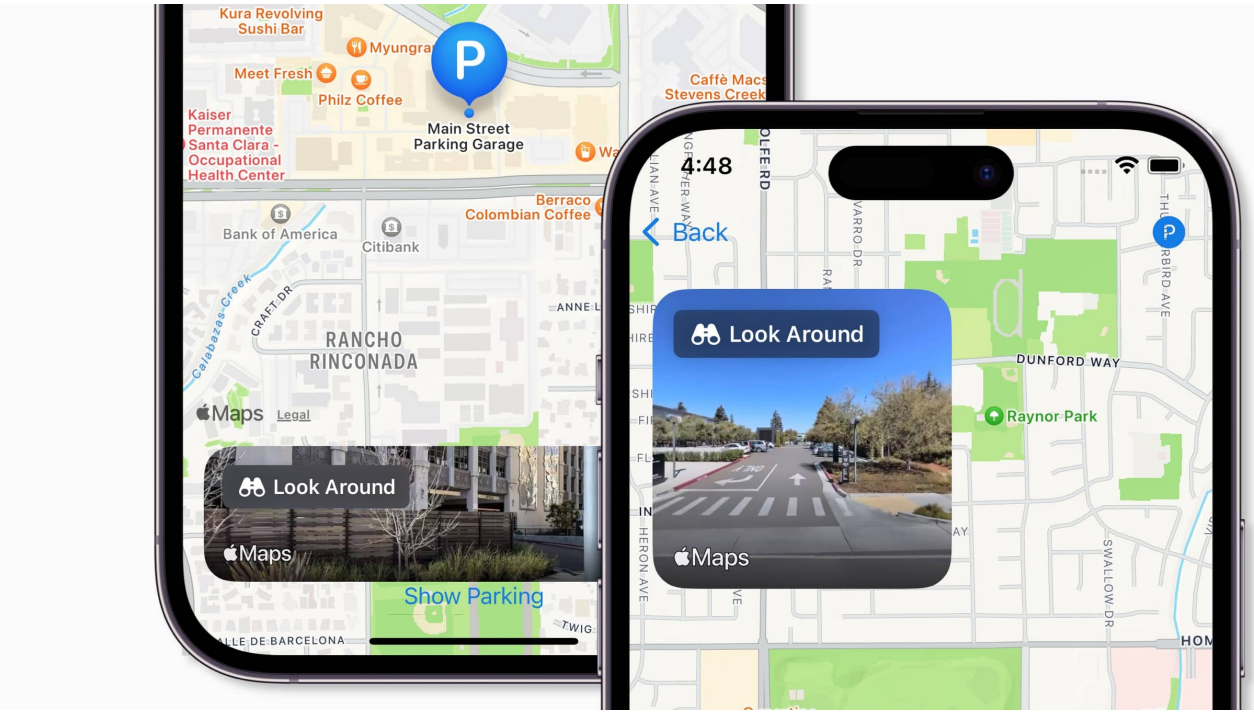
```
// Xcode --> SwiftUI --> MapFig5View.swift

private func fetchLookAround() {
    guard let selectedPoint = selection else {
        return
    }
```

```
        look = nil

        Task {
            let req = MKLookAroundSceneRequest(mapItem: selectedPoint)
            look = try? await req.scene
        }
    }
```

When the Look Around is fetched, we then show a preview that's tappable to expand into the full Look Around experience:

```
    .safeAreaInset(edge: .bottom) {
        VStack(spacing: 0) {
            if let _ = look {
                LookAroundPreview(scene: $look)
                    .frame(height: 100)
                    .clipShape(RoundedRectangle(cornerRadius: 22))
                    .padding(.horizontal)
            }
            Button("Show Parking") {
                showParking()
            }
        }
    }
```

What's great about `LookAroundPreview` is that'll match the size that you give it, too. For example, no matter if you want it to cover the entire view's size or just have a hard-coded frame – it'll do its best to adapt. Check out the smaller, square variant next to the longer, rectangular version below:



User Annotations and Map Controls

If you want to show where the user is on a `Map` view – it couldn't be simpler:

```
Map {
    UserAnnotation()
}
```

That's it! But, of course – this won't focus in on the user's location right away. For that, you'd want a control to focus the map on the user's actual location. There is a built-in button for this, too – and in fact, there are several built-in views for `Map` that you can use.

Here, we add a compass, a button to focus on the user's location (that would be the `UserAnnotation` mentioned above), a pitch toggle and a scale view:

```
// Xcode --> SwiftUI --> MapFig6View.swift
```

```
struct MapFig6View: View {
    var body: some View {
        Map {
            UserAnnotation()
        }
        .mapControls {
            MapCompass()
            MapScaleView(anchorEdge: .leading)
            MapUserLocationButton()
            MapPitchToggle()
        }
    }
}
```

Even better – these are just simple SwiftUI Views, so you can add them anywhere (even outside the `.mapControls` modifier).

Customizing Map Styles

The `mapStyle` modifier mentioned earlier has a lot of interesting customization options outside simply choosing a style. For example, we've been showing a lot of parking spot examples – but if you use one of the initializers in the `standard` style, we could just get those annotations for free:

```
// Xcode --> SwiftUI --> MapFig7View.swift

struct MapFig7View: View {
    private let poi: [MKPointOfInterestCategory] = [.parking]

    var body: some View {
        Map()
            .mapStyle(.standard(elevation: .realistic,
                                emphasis: .muted,
                                pointsOfInterest: .including(poi),
                                showsTraffic: true))
    }
```

```
}
```

Before you start in and do heavy lifting to place annotations or other similar marks, check to see if you can simply place them out-of-the-box by using the `MapStyle` initializer. Here's a list of what you can place for points of interest by the framework:

- AirPorts
- Aquariums
- ATMs
- Bakery
- Banks
- Beaches
- Brewery
- Cafés
- Campgrounds
- Car Rental Locations
- Electric Vehicle Chargers
- Fire Stations
- Fitness Centers or Gyms
- Food Markets
- Gas Stations
- Hospitals
- Hotels
- Laundry
- Libraries
- Marinas
- Movie Theaters

- Museums
- National Parks
- Night Life
- Parks
- Parking
- Pharmacies
- Police
- Post Offices
- Public Transit Locations
- Restaurants
- Restrooms
- Schools
- Stadiums
- Stores
- Theaters
- Universities
- Wineries
- Zoos

While I haven't been able to confirm this via the documentation, I believe that if you search for any of these types of locations – you'll also get the `Marker` customized displayed for free, too. I think this is why the parking locations from our previous examples are shown with a custom pin without any extra work on our part.

App Review

Take care not to hide the Apple Maps logo and legal link. While it's now shown if the MapView is 200×100 or smaller, in other cases – you'll want to apply about 7 points of padding around the sides of it, and about 10 points vertically.

If you don't follow these "rules", you might get hit with a rejection. Again, you don't have to call attention to it – but you can't purposely hide it. Personally, this is a rule I don't mind, as the link and logo provide assurance to people that the map data is coming from a credible source.

Three Key Takeaways

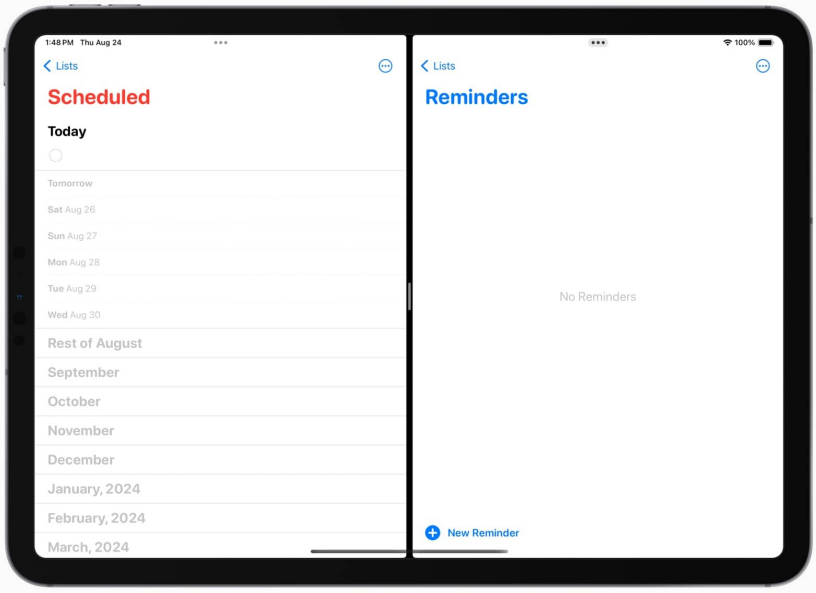
1. Use MapKit to bring geographical map features to your app.
2. Look for novel ways to incorporate maps into apps; it can be used for more than just GPS or navigational purposes.
3. Keep map experiences familiar, even if you do use them in different ways.

Multiple Windows and Scenes

Multiple windows haven't been around very long in the grand scheme of iOS. It wasn't until iOS 13 that they made their debut appearance on the platform. Today, their importance has only been more pronounced, with new system features coming seemingly each year and even nascent platforms such as visionOS supporting them. Further still – Stage Manager puts them front and center. Time will tell if iOS proper receives multiple window support – but for my money, *any* iPadOS app should have them.

To be clear – what do we mean by multiple windows? In essence, it's the ability to spawn multiple windows of your app – to multitask:

Multiple Windows
> Reminders with two windows.



Here, we've got two *windows* of Reminders and each of those windows has a view. They act independently, and depending on how you architect things – changes in one window may or may not be reflected in the other. Supporting these flows is expected on macOS, iPadOS and visionOS. On other platforms, the notion simply doesn't exist yet (iOS, tvOS and watchOS).

So, what does effective windowing look like on iPadOS? To me, it comes down multitasking workflows and size adaptability. Here are some things you should be thinking about:

1. Does my app resize correctly in Stage Manager and multitasking scenarios?
2. Is the data and overall state of my app accurately reflected in all of those windows?

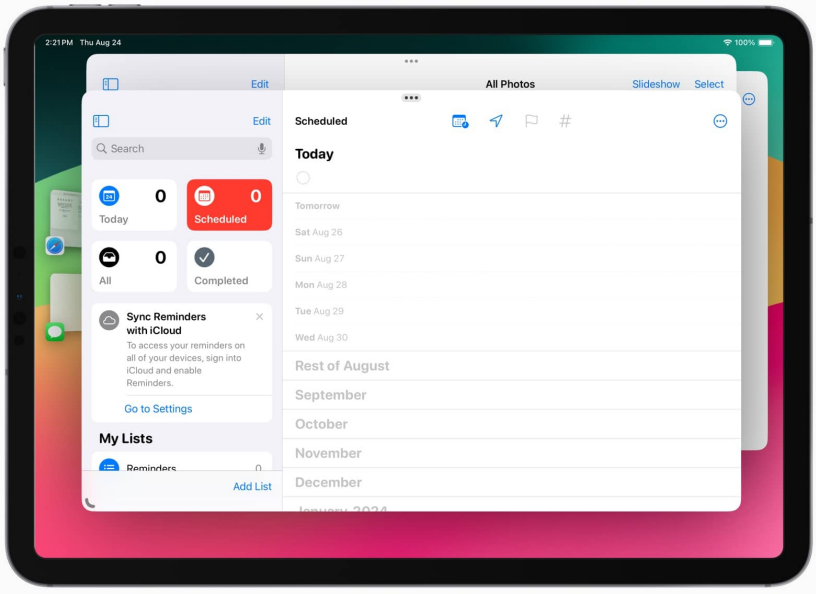
3. Can users do power flows, such as dragging and dropping things back and forth? Are there context menus to open new windows?
4. Does your app support unique windows to do certain flows that can open independently?

Those type of questions will start you down a successful path. In terms of user experience, there are four different ways your app could be used in a "multiple window" situation:

Stage Manager

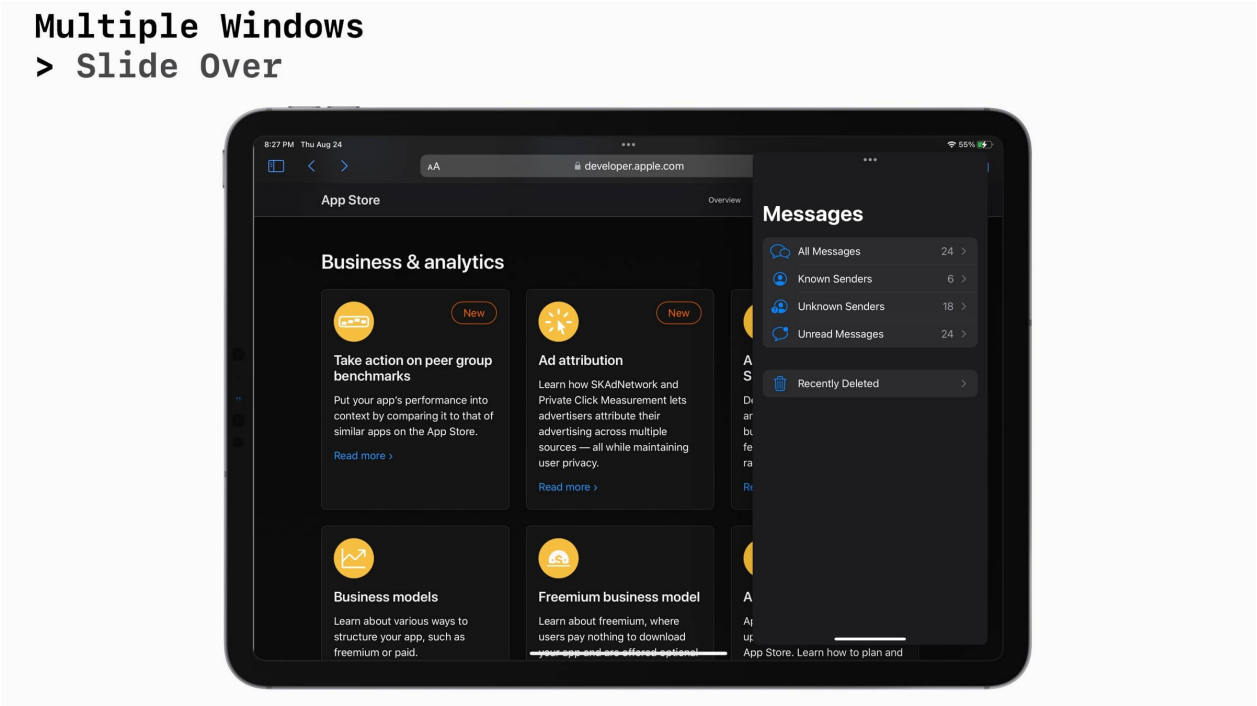
Stage Manager allows more of a traditional, macOS-like windowing experience on iPad. Users can arrange windows however they like, have several ones exist together and be resized to varying degrees:

Multiple Windows
> Stage Manager



Slide Over

Slide over occurs when you open another app over an existing one. You can also have it attach to a Split View slot from here as well.



Split View

Split View is probably what you commonly associate with multiple windows. Here, you share the device's screen with two apps, and you can slide the divider in the middle to portion the screen sizes accordingly to favor more space to one app or the other. We saw this in the first image above at the start of the chapter.

Picture in Picture

Exclusive to videos – this occurs when you are watching a video, such as in the TV app, and exit to the home screen. The small window persists to still play the video, notice that here in the bottom right:

Multiple Windows
> Picture in Picture



Also, you can have a combination of these occurring at the same, too – they aren't always mutually exclusive. For example, you could be in Split View and bring in an app via Slide Over.

From a workflow perspective, you can think of windows playing one of two roles. First, there is *always* a primary window. This is a window representing the core functionality of your app – you can get to anywhere in your app from this window. Additionally, there is might be an auxiliary window, and this one is likely more specialized. Perhaps it only operates on one specific task, such as editing a photo.

Further, you can suggest a presentation style for windows as well. There are three that you can use:

1. **Prominent**: This acts almost like a modal window – it displays front and center and dims surrounding areas to prevent interactions with anything else. It's good to get a focused flow done, and then should be dismissed.

For example, the Quick Look framework displays a window prominently to preview data. In the User Experience book in this series, we saw how this worked in the Quick Look chapter by using `QLPreviewSceneActivationConfiguration`.

2. **Standard**: Probably what you think of for multiple windows – it'll be shown side-by-side or alongside another window and it allows full interaction with it, and other windows.
3. **Automatic**: This isn't a specific style, but an API choice which lets the system decide what kind of presentation style will be used.

Before we dive into how it works, I should note there are some apps which have valid reasons not to participate in multiple windows. These are very specialized apps, typically – one I've personally seen is in the medical field where screen real estate is used for 3D models of the human body. For the 99% of us, though, it should be table stakes for any iPadOS app.

Let's dive into how it all works.

How it Works

UIKit

UIKit uses the concept of scene sessions (`UISceneSession`) to manage multiple windows. Your app will have one or more of these sessions running at once, and those are vended via an object conforming to `UISceneDelegate` and a configuration for the scene, represented by `UISceneConfiguration`. If you've been around for a while, this is all a distinct departure from the classic application delegate, which used to manage a lot of things about your app's lifecycle.

Now, there is a distinct split. You app delegate manages *scenes* instead of the app wholesale, and these scenes handles things that the app delegate *used* to handle – like opening URLs.

That's a lot of classes and words, so let's look more critically at it. To wit, this chapter is the primary reason the sample app itself is built on a UIKit lifecycle – we can see how it fits firsthand. As we'll see later, there's less to manage in SwiftUI to accomplish the same outcome.

It all starts with your app's `info.plist`, which will require an entry for `UIApplicationSceneManifest`. This contains information about the scenes that your app supports. In you look in the demo project, ours is simple:

```
<key>UIApplicationSceneManifest</key>
<dict>
    <key>UIApplicationSupportsMultipleScenes</key>
    <true/>
</dict>
```

If you don't have this key present, you'll want to manually add it. Any modern iOS app from the last few years should have this present already. From there, we'll go to our app delegate file to set up a scene configuration:

```
func application(_ application: UIApplication,
configurationForConnecting connectingSceneSession: UISceneSession,
options:
UIScene.ConnectionOptions) -> UISceneConfiguration {
    // Called when a new scene session is being created.
    // Use this method to select a configuration to create the new
scene with.
    let config = UISceneConfiguration(name: "Default Configuration",
sessionRole: connectingSceneSession.role)
    config.delegateClass = SceneDelegate.classForCoder()
    return config
```

```
}
```

We're creating a scene configuration and returning it – but what's important to note here is that we assigned to `delegateClass` on that configuration. This is what UIKit is going to look for when creating a new scene, and by proxy – your view controller.

Now, we get to the view setup in our scene delegate:

```
class SceneDelegate: UIResponder, UIWindowSceneDelegate {
    var window: UIWindow?

    func scene(_ scene: UIScene, willConnectTo session:
UISceneSession, options connectionOptions:
UIScene.ConnectionOptions) {
        guard let windowScene = (scene as? UIWindowScene) else
{ return }

        let rootController = GlossaryViewController()
        let rootNavController =
CustomNavController(rootViewController: rootController)
        window = UIWindow(windowScene: windowScene)
        window?.rootViewController = rootNavController
        window?.makeKeyAndVisible()

        // Scene restoration
        if let previousActivity = session.stateRestorationActivity {
            openBookChapter(with: previousActivity)
        }
    }
}
```

And with a `window` property afforded to us via adopting `UIWindowSceneDelegate`, we've got all we need to vend a view controller and have it show. I'd encourage you to open up the demo project if you need to follow it step by step. Personally, this was the easiest way to understand multiple windows in UIKit when they were initially announced.

You might be thinking at this point – all of that setup, and for what? A single view showing like we've always had? And you'd be right, but what it *also* shows is all of the infrastructure to support multiple windows. With this work, we can spawn multiple windows using our `SceneDelegate`, and we've got a plug-in point to open different kinds, too.

For example, notice the `UISceneConnectionOptions` that we were given. Using that, we could check user activities on that object to setup the scene differently. Or, we could use it for state and scene restoration. Further, maybe there are values in the `URLContexts` array. We can be flexible in this function, and configure things any which way.

While it is a bit to understand at first, it's thankfully easy to open new windows once you've got this infrastructure in place. By requesting scene activations, you do things like activate a scene with drag and drop, or perhaps programmatically open one with a context menu. Simply kick off the request using `UIApplication` by giving it a user activity representing the scene you want.

Here, I've made just a generic function to simply open another window of the app – nowhere specific, just another window:

```
// Xcode -> UIKit -> MultipleWindowsFig1ViewController.swift

let openAc = UIAction { _ in
    let activity = NSUserActivity(activityType:
"com.bic.demo.openWindow")
    activity.title = "Open Window"

    let options = UIWindowScene.ActivationRequestOptions()
    options.placement = nil // Let UIKit decide

    let req =
UISceneSessionActivationRequest(role: .windowApplication,
```

```
                                                    userActivity:
activity,
                                                    options: options)
    UIApplication.shared.activateSceneSession(for: req) { error in
        print(error)
    }
}
```

If you wanted to open app-specific data, you can add in anything you need via the user activity user info dictionary (such as a model identifier). You may also be wondering where the `com.bic.demo.openWindow` identifier came from. In your app's info.plist, you need to register the windows you wish to open new scenes with – and we've done that in the sample project:

```
<key>NSUserActivityTypes</key>
        <array>
            <string>com.bic.demo.openWindow</string>
        </array>
```

SwiftUI

There is, true to its nature, *much less* to think about in SwiftUI than in UIKit to get things to work. In fact, if you start a SwiftUI app today in Xcode – you get it done for you out of the box.

There are things it pays to know about, to be certain. For starters, apps in SwiftUI use a compositional approach to their architecture and multiple windows. As such, any SwiftUI app (that is, those using the SwiftUI lifecycle) are composed of three parts:

1. The `App`, or entry point for you iOS app.
2. A `Scene` which represents a window of your app.
3. Finally, the `View`, which you're used to – these are composed within a scene.

In any SwiftUI app, you'll see this structure:

```
struct SwiftUIApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

Notice the `WindowGroup` returned by the `body` – that's a *type of* `Scene`, and there are a few of them. Using those scene types, we can easily support multiple windows. For example, there is a `DocumentGroup` for document-based apps, and a `Settings` scene for, well – settings. Additionally, there is a `Window` for one-off window presentations and on macOS, there is a `MenuBarExtra` for showing views within the menu bar.

The good news is that these are composable, too. So, you can use multiple different scene types within a scene. Under the hood, SwiftUI will just take care of it all for you. You can have a window group to support multi-window scenarios along with a single window for something that represents something along the lines of global app state:

```
struct SwiftUIApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
        Window("Editing", id: "editingWindow") {
            EditingView()
        }
    }
}
```

To better illustrate the difference between `WindowGroup` and `Window` — think of a game. You probably only want to support one window in a game, so a window would make more sense than a window group. For most of the apps that you and I make, though, starting with window group is perfect.

If you're targeting multiple platforms with a SwiftUI app, you can also query whether or not multiple windows are supported via the environment:

```
@Environment(.\supportsMultipleWindows) private var supported
```

That'll return true for iPad apps using the SwiftUI lifecycle and the relevant property list keys we mentioned above, and for macOS apps. I believe the same is true of visionOS apps, as well, but I wasn't able to verify that for certain at the time of this writing. I'll update this text when I get the answer.

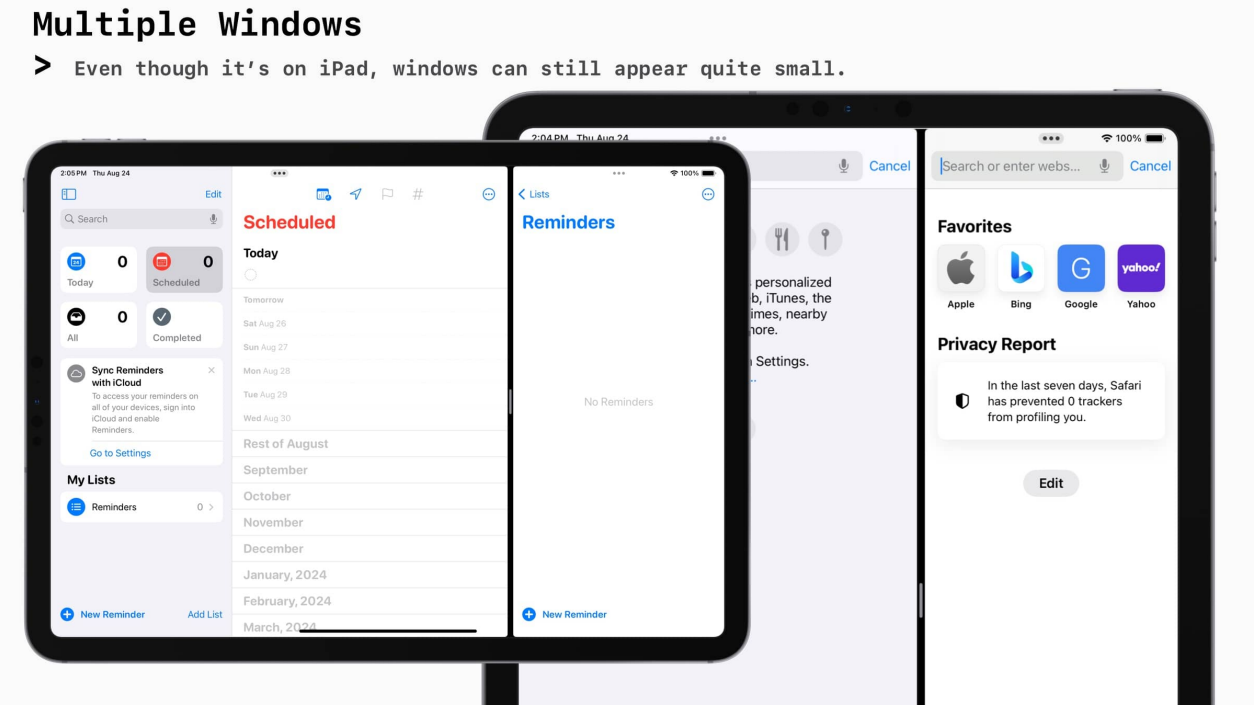
Tips

Don't Use Idiom Checks

Too often developers sprinkle checks like these throughout their code:

```
if UIDevice.current.userInterfaceIdiom == .pad {
    // Use some UI
} else {
    // Do different UI
}
```

This is not a great idea. As seem in the image below, there are no promises of a larger screen simply because your app is running on iPad:



Instead — opt to create adaptive views and let them resize accordingly. In UIKit, this means avoiding hard frame calculations or sizing and opting to use Auto Layout for most cases. In SwiftUI, things are inherently easier — but even there, there are tools such as `ViewThatFits` and size class information (as in UIKit) to help you adapt the screen real estate that you do have.

React to Data Changes Instantly

There will be situations where the user does some sort of action that will affect the data they have in another window. For example, consider a shopping list app. The user may have two windows open in Split View — the main "list of their lists" view and perhaps another individual list open.

So, what should happen if they delete the list showing in the single window from the *other* window (i.e. the one showing all of their lists)? The answer will vary, but personally, I like to show an alert and give them the option restore it if they wish.

Using Combine or whatever API you feel is necessary, look for the current data models in your app to be deleted or changed significantly. If the view is presented while that happens – offer them an option to restore it, close the view automatically or do generally whatever you feel the best user experience would be.

Thoughtfully Consider Singletons

Singletons can pose some critical issues from an architecture standpoint when it comes to multitasking. To be clear, I'm not against singletons, and in fact – I find them quite useful. When it comes to multiple windows, though, they can get you in trouble. Consider this mock singleton for a social media app:

```
struct UserStore {
    static let shared = UserStore()
    public var selectedUserProfileID: String = ""
}
```

Here, we're saying that the `UserStore` used throughout this app tracks which user profile is being used. The problem now, though, is that if the user has two windows open, and wants to sign out of one and use an entirely different account in another– they can't really do that. We have a singleton that powers the notion of a user being signed in, and that doesn't lend itself well at all to multiple windows.

With that setup – we're essentially bound to one profile being used at a time, which doesn't fit the multiple window and multitasking ethos.

Think about similar situations such as this before you stick things inside a singleton. There are things that are appropriate to be used here, however, such as settings that should always be honored. Things such as accessibility settings are likely to be reflected in *every* window of your app. But, things that should be unique per window instance don't belong on a singleton.

Context Menu for Opening a New Window

Context menus are perfect for opening new windows, and Apple typically offers the option in their own apps with the copy of "Open in a New Window."

In SwiftUI, this is done by using the `openWindow` environment value. Using this, you can open either a window group or single window, so long as the identifier you use is presented within your `App` declaration:

```
struct OpenRecipeWindowButton: View {
    let recipe: Recipe
    @Environment(\.openWindow) private var openWindow

    var body: some View {
        Button("Open \(recipe.name) in New Window") {
            openWindow(value: recipe.identifier)
        }
    }
}
```

Also, take note that we're using the model's identifier. Apple discourages sending the entire model itself – while the reason isn't explicitly stated, I assume there are performance considerations here. Passing the model *will work*, but opt for a flow where an identifier is used, and then the consumer fetches the model in question. Finally, also ensure that your model conforms to `Hashable` and `Codable` for state restoration purposes.

In our `OpenRecipeWindowButton` – you may have noticed that we're wanting to open up a detail about a recipe. How does SwiftUI associate that model to a window to open? The answer is through an initializer that uses the same type of identifier on the `WindowGroup`:

```
WindowGroup("Recipe Detail", for: Recipe.ID.self) { $recipeID in
    RecipeView(id: $recipeID)
```

```
}
```

So long as the value in this initializer matches the value we used for the `openWindow` function, everything will be hooked up for us – the window will open and have the identifier ready for us to use. For each unique value used here – we'll get another window. This is why we needed `Hashable` conformance as I mentioned earlier, this lets the framework know when we were opening new model instances (i.e. "recipe" models here) or have an existing one.

I want to call out that if you've got a document based window, you can open those via the environment as well – but note that's only available on macOS:

```
@Environment(\.newDocument) private var newDocument
@Environment(\.openDocument) private var newDocument
```

Leverage Drag and Drop

There are several instances where users will want to use your app in two different windows to shuttle data back and forth. Or, there are likely times when you consume data in your interface that a user would want to drop on it. Consider an app with the notion of a user profile – maybe the user would have the photos app open and might try to drag and drop a photo on the user profile photo control.

There are tons of use cases here – so a good rule to follow is to allow drops (and drags) in a control which has user editable data. To continue our thought here – if you're building the user profile button to choose a photo, allow it to accept drops of a photo. Same for text controls, URLs – you get the idea.

For more concrete implementation details, check out the Drag and Drop chapter in this book.

Pinch to Open

UIKit offers a system gesture to open a window in the prominent presentation style. You can achieve this with either the built in delegate option for `UICollectionView` or by installing it yourself with a system-provided `UIInteraction` — `UIWindowScene.ActivationInteraction`.

If you are using the collection view — implement `collectionView(_:sceneActivationConfigurationForItemAt:point:)` and return your activation configuration. From there, any pinch out on that collection view item triggers the window opening. If you install the interaction manually on any `UIView` — you get the same user experience there, too (i.e. pinch out to open the window).

Three Key Takeaways

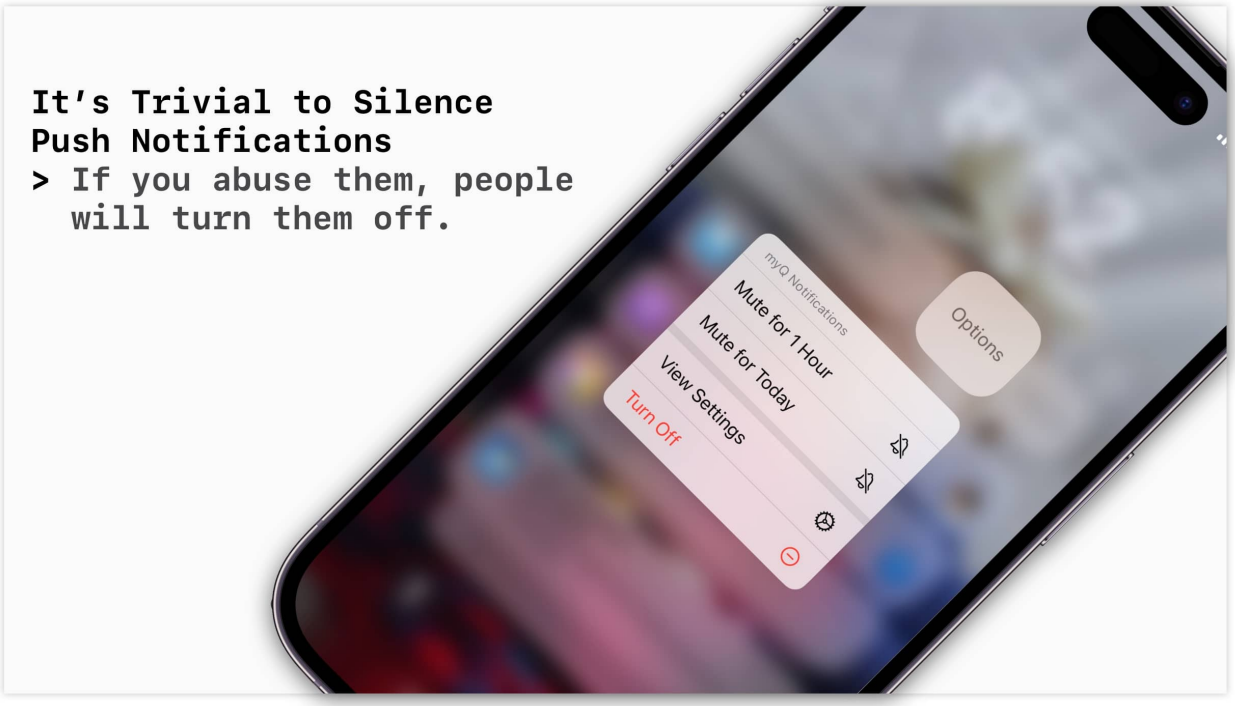
1. Multiple windows should be a part of any iPadOS app.
2. It helps to think in terms of macOS, instead of iOS, when thinking about architecture choices.
3. Multiple windows allows for many flows that power users will enjoy, and the lay user will enjoy discovering.

Notifications

Push Notifications are all about information and attention. When used properly, they deliver time-sensitive information to people. Or, they inform folks of important occurrences happening within their containing app.

To be honest, though — they get a bad rap. Most of the time, they downright annoy people. And that's because so many developers and companies deploy them the wrong way. Push Notifications aren't meant for promotional sales, "dark patterns" of drawing people back into an app artificially or similar cheap tactics.

To wit — I think that's one reason Apple has made it so easy to shut notifications down:



So — how do we make them, well, *not* that way? This chapter is all about how to get them right, and how to implement them end-to-end.

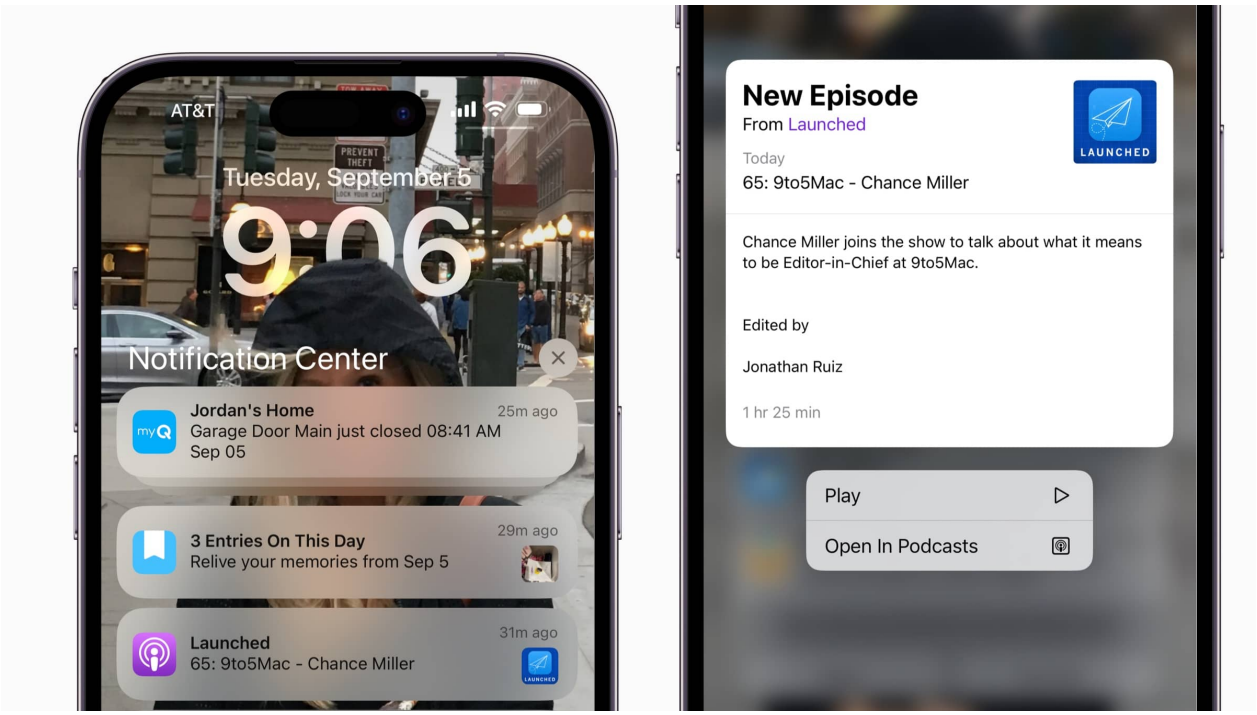
Push notifications deployed with empathy for your users can be an indispensable tool for any app. Along the same lines of Live Activities, informing people of the status of something related to your app can be a delightful experience. If you can build trust and show that your notifications matter, people will gladly begin to rely on them.

How it Works

Push Notifications, at their core, are smaller banners that contain timely information about your app (again, if done correctly). They can appear on iOS devices once they are delivered from the app, or a related server requests it via a POST call to APNS (Apple Push Notification Service). Where they end up actually showing depends on a few different factors, as we'll see in a bit.

Push Notifications can be delivered and scheduled either locally, or remotely. They don't require your app to be in the foreground, either – iOS will either schedule a local one or your server delivers a remote notification. No matter which route you take, they will require the user's permission. Further, they can be interactive – offering a few buttons to perform quick actions based off of the notification's contents.

I should note that for this chapter, we are focusing on *alert* pushes. This is probably what comes to mind when you think of notifications – these are the ones that are front and center that people will see:



There are also *background* notifications, which aren't meant to be visible to users. In fact, in most cases – you don't even need permissions for these. These are built to deliver data to your app, such as CloudKit background notifications notifying your app that data has changed – and a fetch should happen to keep things up to date.

A quick note here: I'll be showing code samples on how to setup remote notifications – but I personally don't have a server setup for this book series to deliver them. But, I think it's helpful to show you the steps within your app. We'll also deliver local notifications, and many of the steps are the same.

How to Setup Push Notifications

To setup pushes, you'll want to conform to `UNUserNotificationCenterDelegate`. A natural place to do this in UIKit-based apps is `didFinishLaunchingWithOptions::`

```
// Xcode -> AppDelegate.swift
func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions:
[UIApplication.LaunchOptionsKey: Any]?) -> Bool {
    // If you're going to use remote notifications
    UIApplication.shared.registerForRemoteNotifications()

    // Assign as delegate for `UNUserNotificationCenterDelegate`
    UNUserNotificationCenter.current().delegate = self

    return true
}
```

If you're using SwiftUI, there are two options:

1. Use an application delegate like you do with UIKit, via the adaptor: `@UIApplicationDelegateAdaptor(AppDelegate.self) var applicationDelegate`
2. Or, make an object that conforms to the delegate, and have it perform these functions when your app is launched.

Next, we need to get permission from the user to show alert push notifications:

```
// Xcode -> SwiftUI -> PushNotificationsFig1View.swift

private func promptNotificationPermissions() {
    let userNotifications = UNUserNotificationCenter.current()
    userNotifications.requestAuthorization(options:
[.alert, .sound]) { (success, error) in
        if let e = error {
            result = "No access: \(e.localizedDescription)"
        } else {
            result = "Push Notification access granted."
        }
    }
}
```

Once again, you'll want to use the current instance of `UNUserNotificationCenter` and request authorization.

This works like many of Apple's APIs, where a system standard alert will be presented:



Whatever the result here, it'll be reflected in the device's settings under Push Notifications. There are several different types of notification permissions you can ask for, however:

1. **Badge**: The red number badge you see on top of the application's icon.
2. **Sound**: The ability to play a sound alongside your notifications.
3. **Alert**: The most prominent permission – this represent the push notification alert and you'll likely want this each time you request permission.
4. **Critical Alert**: This ties into Focus Modes, which we'll see below – but it's meant to represent the most critical types of alerts that user's absolutely need to see. These are rare.
5. **Provides App Notification Settings**: The system will include a button to deep link to your app's own notification settings. We'll look at this under "Tips" later on.
6. **Provisional**: This represents a non interrupting alert that'll end up in Notification Center.

Understanding Alert Notification Payloads

When you receive either a remote or local push notification, you'll get an alert notification payload. It's represented as a dictionary, and you can add all sorts of metadata that helps make your alert understandable and actionable. Here's an example of what it might look like if we were constructing one to send to APNS:

```
{
    "aps" : {
       "alert" : {
            "title" : "New chapter available!",
            "subtitle" : "Learn about push notifications.",
```

```
            "body" : "The best-in-class book series has been
updated."
        },
        "sound" : "default",
        "badge" : 1,
    },
     "chapter" : "push_notifications",
     "releasedAt" : "August 21st"
}
```

Anything in the `aps` key should be structured in the format Apple has documented. So here, `alert` and its key, and `sound` and `badge` are options that Apple specifies – they aren't arbitrary.

However, *outside* of that dictionary is where you can include any additional meta-data about your push. So in this case, that's `chapter` and `releasedAt` – our own "proprietary" keys in the payload. There are several other affordances Apple gives us for push notifications. Buttons, images, identifiers to link them to previous Push Notifications for threading and more.

Okay, so we know what notification payloads look like, we've setup our app to ask for them and get permission – how do we handle any incoming ones? Again, we'll look at the user notification protocol.

The first function is where we operate on the incoming notification. The second is to allow for the banner notification to show at all – if we missed this one, nothing would show once we fire off the notification:

```
// Xcode -> AppDelegate.swift

func userNotificationCenter(_ center: UNUserNotificationCenter,
                            didReceive response:
UNNotificationResponse,
                            withCompletionHandler completionHandler:
@escaping () -> Void) {
```

```
    let userInfo = response.notification.request.content.userInfo

    guard let chapter = userInfo["chapter"] as? String else {
        // Either way, we to invoke the completion handler...
        completionHandler()
        return
    }

    // Show book chapter
    completionHandler()
 }

func userNotificationCenter(_ center: UNUserNotificationCenter,
willPresent notification: UNNotification,
withCompletionHandler completionHandler: @escaping
(UNNotificationPresentationOptions) -> Void) {
    completionHandler( [.banner, .badge, .sound])
}
```

In the `userInfo` dictionary of the first function, we can get at the keys we mentioned above. In most cases, you'll not need to worry about the actual push notification body itself – it's the metadata you're likely after. Remember, anything in the `aps` key is provided to send and display the actual push.

In our example, we had a book chapter we delivered – so in that case, we'd parse the chapter out and present it. Again, any of the key-values which we provide outside of the `aps` key help us send whatever data we need to operate on any type of push.

One important note here – you'll need to make sure you're calling the `completionHandler()` whether or not you were able to parse the data you were after. This tells iOS that we are done handling the notification.

Sending a Local Push

Now, let's try and send that notification we mapped out earlier when we were looking at notification payloads. This is a essentially a three step process:

1. Create the notification's content using `UNNMutableNotificationContent`.
2. Create a "trigger", which represents when the notification should fire. This could either be geofenced, a `Date` or a `TimeInterval`.
3. Finally, create a `UNNotificationRequest` which houses all of these things, and add it to the current instance of `UNUserNotificationCenter`.

Here's how the whole thing looks:

```
// Xcode -> SwiftUI -> PushNotificationsFig2View.swift

private func sendLocalPush() {
    // Create the notification
    let localPushContents = UNMutableNotificationContent()
    localPushContents.title = "The Accessibility Chapter"
    localPushContents.body = "Learn about AX Notifications"
    localPushContents.userInfo = ["chapter":
BookChapter.AXNotifications.rawValue]
    localPushContents.categoryIdentifier = "book_push"
    localPushContents.sound = UNNotificationSound.default

    // Create a trigger
    let trigger = UNTimeIntervalNotificationTrigger(timeInterval:
2.0, repeats: false)

    // Create the request
    let pushIdentifer = UUID().uuidString
    let localRequest = UNNotificationRequest(identifier:
pushIdentifer,
                                             content:
localPushContents,
                                             trigger: trigger)
    // Schedule the request with the system.
    let notificationCenter = UNUserNotificationCenter.current()
```

```
    notificationCenter.removeAllPendingNotificationRequests()

    notificationCenter.add(localRequest) { (error) in
        if let e = error {
            result = "Error: \(e.localizedDescription)"
        } else {
            result = "Scheduled!"
        }
    }
}
```

If you run this code either in the simulator or on your device with Push Notifications granted, you'll get the banner notification showing after two seconds. When it does, you can tap on it to open the chapter about accessibility notifications.

We've also slightly modified our app delegate to route that notification payload's metadata to open the book chapter for us as well:

```
// Xcode -> AppDelegate.swift

func userNotificationCenter(_ center: UNUserNotificationCenter,
                            didReceive response:
UNNotificationResponse,
                            withCompletionHandler completionHandler:
@escaping () -> Void) {
    let userInfo = response.notification.request.content.userInfo

    guard let chapter = userInfo["chapter"] as? String,
    let chapterEnum = BookChapter(rawValue: chapter) else {
        // Either way, we to invoke the completion handler...
        completionHandler()
        return
    }

    // Show book chapter...
    let keyWindow = UIApplication.shared.connectedScenes
        .filter({$0.activationState == .foregroundActive})
```

```
            .map({$0 as? UIWindowScene})
            .compactMap({$0})
            .first?.windows
            .filter({$0.isKeyWindow}).first

        guard let activeWindow = keyWindow,
              let rootController = activeWindow.rootViewController as?
CustomNavController,
              let glossaryController =
rootController.viewControllers.first as? GlossaryViewController else
{
            completionHandler()
            return
        }

        glossaryController.pushBookChapter(chapter: chapterEnum,
with: .UIKitSample)

        completionHandler()
}
```

Push Notifications are powerful, but they aren't that complicated. Remember, you need to do a little setup and conform to `UNUserNotificationCenterDelegate`, ask for user permissions and then either send remote or local notifications and handle them.

Tips

Trigger Types

As we saw above, you can trigger local notifications using `UNNotificationTrigger`. However, there are a few ways that you can use it.

Time Intervals

This is what we used above, `UNTimeIntervalNotificationTrigger`. With it, you can specify a set amount of time that should elapse before the notification attempts to send.

Calendar Based

Similar to time intervals triggers, these use `Date` structures instead. By passing one to `UNCalendarNotificationTrigger`'s initializer, you can specify a specific time or day for it to be sent.

Location Based

Finally, you can essentially make a geofence trigger for these types, `UNLocationNotificationTrigger`. You pass it a region, of type `CLRegion`, and it'll fire when the device enters or leaves that specific place. Keep in mind, you'll need "When In Use" authorization for this to work from the Core Location framework.

Thoughtfully Request Permission for Pushes

Too many onboarding flows just instantly show the Push Notification prompt. This isn't great, don't do this! Instead, always provide a reason as to why you want a Push Notification to show. Cliché though it may sound, simply tell them about the "value" they'll bring.

I typically try to follow a "Allow notifications so (benefit)" pattern:

- "Allow notifications so we can tell you when timers are up."
- "Allow notifications so you know when your friends are online."

It's not a hard and fast rule, but it can help you get started with writing copy.

Integrating with Focus Modes

People can integrate their notifications with their Focus Modes. Sometimes, that means your notification may not show up where you expect it to. You can have some say in how all of this will work via interruption levels. For example, we could modify our last code sample to use the `.passive` interruption level:

```
localPushContents.interruptionLevel = .passive
```

I should note that the notification, however, *is* delivered no matter the Focus Mode that's active — it's still instantly available. The Focus Mode just dictates how that notification is presented. You can do this server side too, but when scheduling local ones — be sure to assign an appropriate interruption level for notifications:

Passive.: Information people can view at their leisure, like a our book chapter example. This isn't mission critical information, but nice-to-have regardless.

Active: The default option, this is information people would appreciate having in a timely fashion such as sports scores.

Time Sensitive: This notification directly impacts the person and requires immediate action, such as a package delivery.

Critical: These are rare, but they represent information that's absolutely critical. Think health and safety notifications. Again, these notifications are not common. These can also come from governmental and public agencies.

In practice, here's how those would operate once they arrive on the device:

Breaking Through Focus
> How interruption levels work.

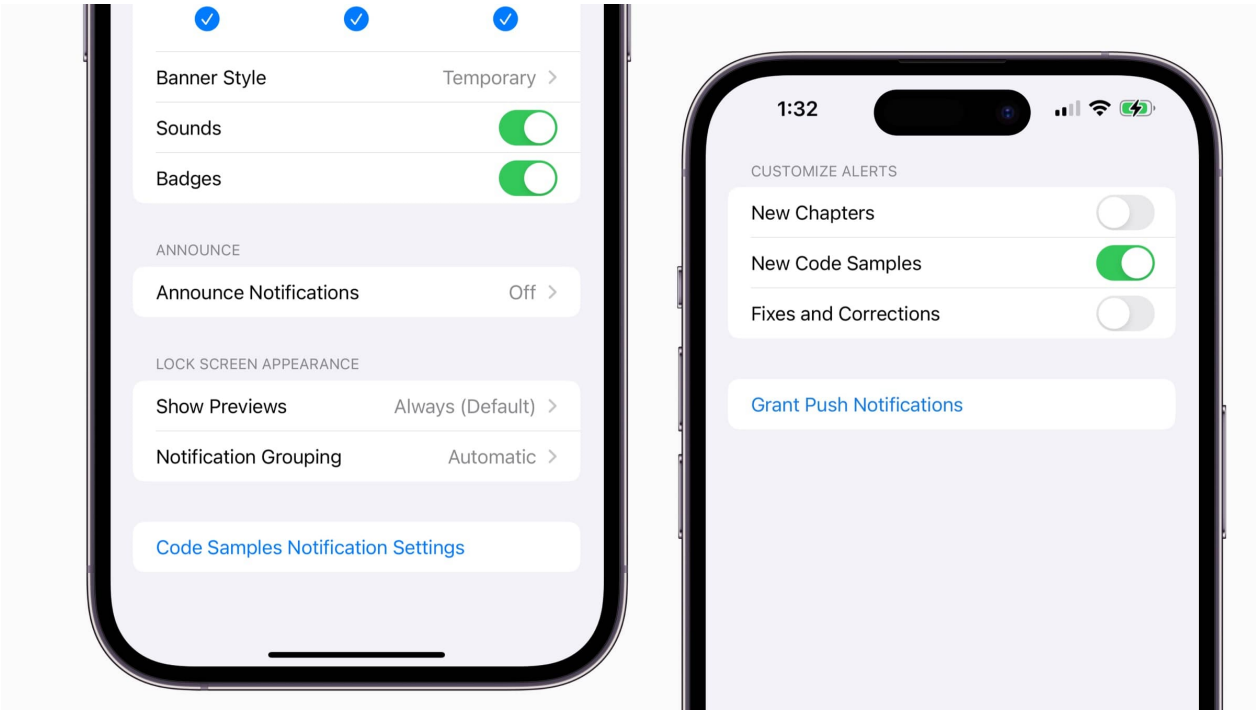
Interruption level	Overrides scheduled delivery	Breaks through Focus	Overrides Ring/Silent switch on iPhone and iPad
Passive	No	No	No
Active	No	No	No
Time Sensitive	Yes	Yes	No
Critical	Yes	Yes	Yes

Linking to Your Own Notification Settings

As we mentioned above – you can link directly to your apps push notifications settings. To be clear, these are settings you directly provide *within* your own app, not the system-provided Push Notifications options found in the Settings app. This is useful to help people get app-specific information about pushes, activate them and more.

When you request permissions for push notifications, add `.providesAppNotificationSettings` as part of the your requested options. Now, when the user views Push Notification settings within the actual Settings app, you'll get a button that prompts them to see your own provided app options too.

On the left side, we're in iOS' Settings app. You can see a button appears at the bottom, which will then call a function for the user Notification Center protocol – giving us the opportunity to display our own settings:



Specifically, implement `func userNotificationCenter(_ center: UNUserNotificationCenter, openSettingsFor notification: UNNotification?)` wherever you conform to `UNUserNotificationCenterDelegate` and again, present your settings view there:

```
// Xcode -> AppDelegate.swift

func userNotificationCenter(_ center: UNUserNotificationCenter,
openSettingsFor notification: UNNotification?) {
    let settingsView = MockPushSettingsView()
    let hostingController = UIHostingController(rootView:
settingsView)

    guard let keyWindow = UIApplication.shared.connectedScenes
        .filter({$0.activationState == .foregroundActive})
        .map({$0 as? UIWindowScene})
        .compactMap({$0})
        .first?.windows
        .filter({$0.isKeyWindow}).first else { return }
```

```
    // Present our custom settings
    keyWindow.rootViewController?.present(hostingController,
                                          animated: true)
}
```

Testing Push Notifications Locally

You can test out push notifications a variety of ways. One obvious approach is the one we've done here – simply implement the code, and test it. Beyond that, though, Apple offers a few different techniques, too.

Use `simctl`

When you install Xcode's tooling, you get access to `simctl`, a command line tool to help you manipulate the simulator. If you provide is a sample .json payload of a notification, you can fire it off from Terminal. I've included such a file in the sample project, `test_payload.json`.

If you provide its location of where it's found in your local file system – you can have it be delivered to the running simulator. I find it's easiest to simply copy and paste payloads to the desktop so you can easily fire this command off:

```
$ cd desktop
$ xcrun simctl push booted com.bicBookSeries2023.A-Best-in-Class-
Tour test_payload.json
```

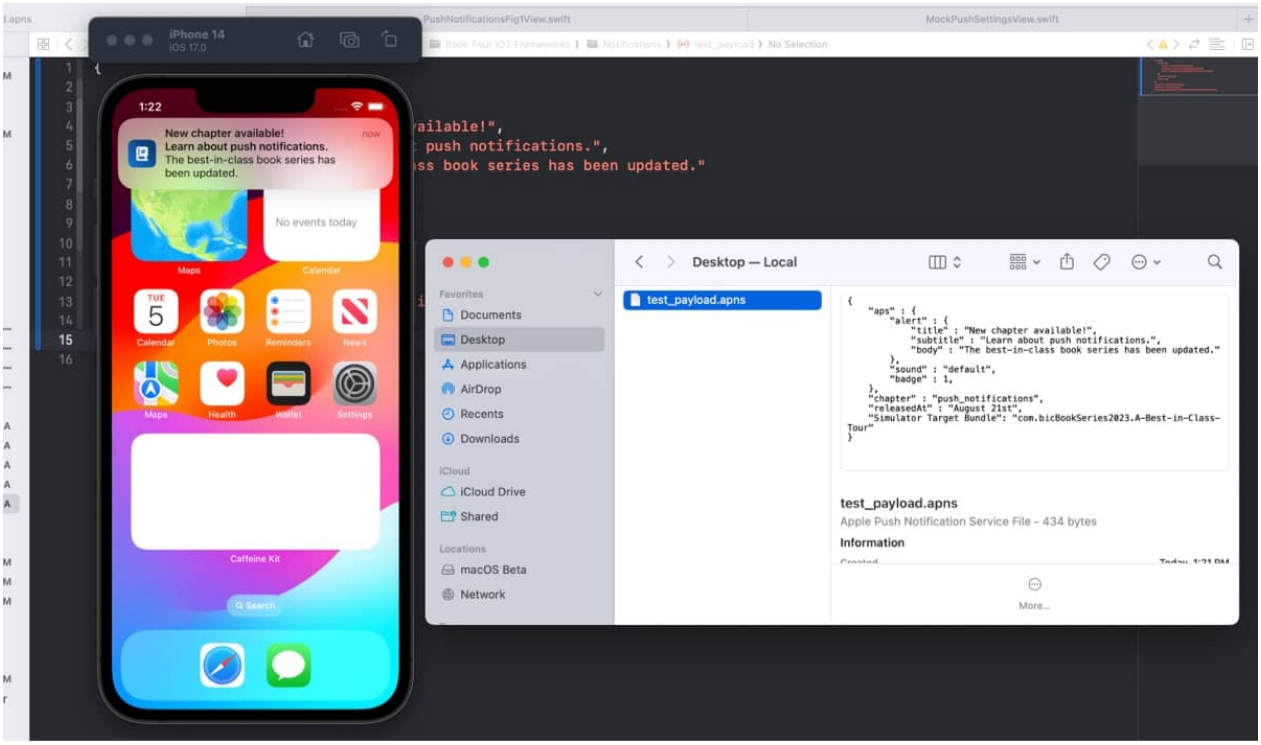
Here, `test_payload.json` is our file we dropped on the desktop and `com.bic-BookSeries2023.A-Best-in-Class-Tour` is the bundle identifier of the sample project. Using that, our notification should show on the device.

Drag and Drop .apns Files

Perhaps even easier is dragging and dropping .apns files onto the simulator. If you already have a test .json file, simply add one more key-value pair to do this:

```
"Simulator Target Bundle": "com.bicBookSeries2023.A-Best-in-Class-Tour"
```

Also, take care to give it the correct file extension, ".apns". In the newly added key, you'll use your app's bundle identifier. Now, if that's on the desktop – we can simply drop it right on top of the simulator and we're all set:



Three Key Takeaways

1. Push Notifications are a valuable way to let users engage with your app and stay informed on the go.
2. Don't abuse Push Notifications, as users will almost certainly turn them off altogether.
3. You can use Push Notifications either remotely or locally.

PencilKit

The Apple Pencil goes beyond simply being a stylus. While many tools came before the official Apple Pencil to try and replicate drawing on iPads – they all seem elementary in comparison to what Apple Pencil offers today. In short, the Apple Pencil packs so much technology and cohesion with iOS that it nearly feels like the real thing – pen (or pencil, marker, etc) to paper.

Even beyond that, iOS offers robust APIs to support drawing, marking things up, signing documents and more. All of that is powered with PencilKit. Whether you want to create a drawing app, support writing down words like you would on a real piece of paper, PencilKit has you covered.

When I think of PencilKit, there are two main components to it. The first is all of the support for drawing in general, and that's backed by a canvas control. Whether you want users to draw with their finger, Apple Pencil or some other tools – a flexible can-

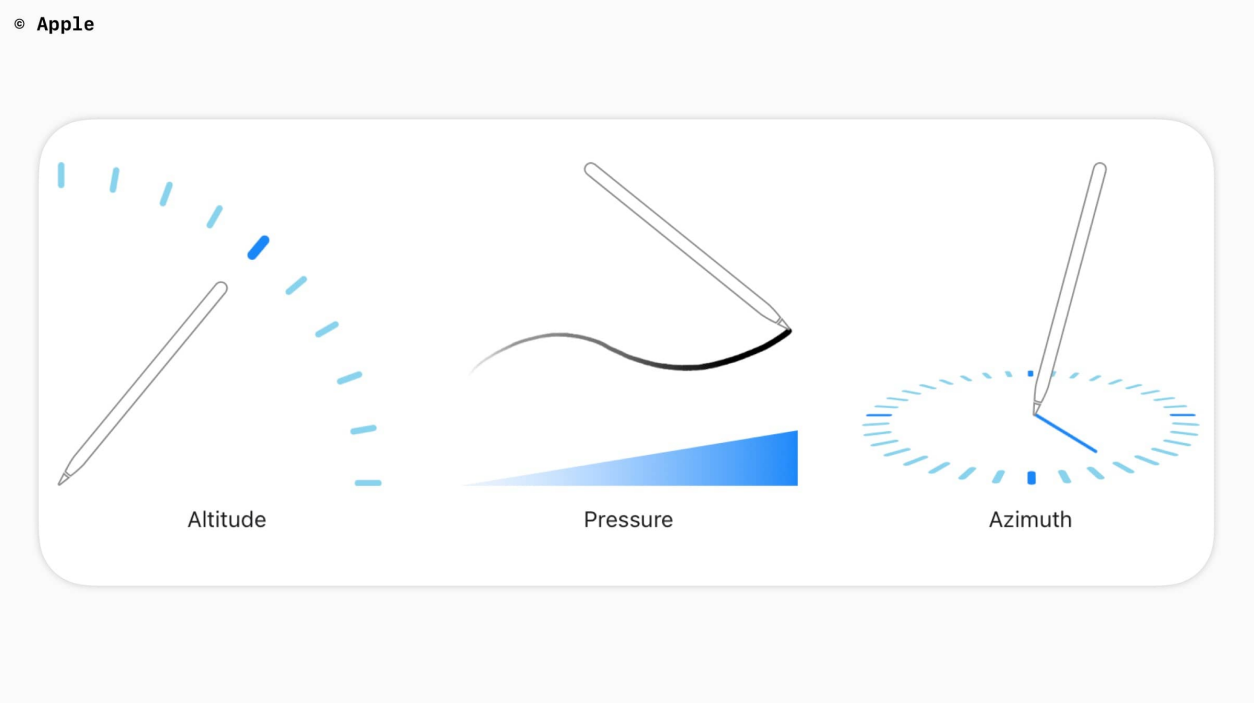
vas control will be just what you need. Next is the Apple Pencil itself — and there are numerous ways to use it outside of just drawing. Let's dive in.

Understanding Apple Pencil

So, why does Apple Pencil feel so much better than the stylus tools that came before it? For starters, it updates 240 times a second — giving it a lifelike level of precision. Further, with all of the advanced hardware packed inside of it, it can detect three important things:

1. **Altitude**: This means it can respond to tilting the pencil up or down, and side-to-side. Much the same way a pencil will produce different line strokes depending on altitude, so will Apple Pencil.
2. **Force**: How hard are you pressing down with a pencil? Again, this greatly affects the resulting pencil stroke. Apple Pencil mimics this real life interaction as well.
3. **Orientation**: Also referred to azimuth, this reflects the orientation of the Apple Pencil.

All of those factors are reported to PencilKit, which in turn produce strokes that feel like you're actually holding a real pencil, pen or marker. Here's a visual representation of what those three things look like while holding the Apple Pencil:



Adding to the real-world metaphor of an actual pencil – all interfaces (should) respond as soon as Apple Pencil interacts with it. If you have an iPad lying around with an Apple Pencil, you'll find that you can instantly start writing inside text controls immediately, there is no delay.

This adds to the responsiveness of the whole experience and keeps the illusion up that you're using a pencil, and you should treat it as such. The benefit? There is virtually no learning curve for users, they know how pencils work – and those same experiences should directly apply to Apple Pencil and any software which supports it.

Continuous Improvements

Perhaps best of all, Apple Pencil and its supporting APIs have only become more flexible and powerful. Before, you had to worry about some implementation details that could get in the way such as dynamically showing and hiding visual effect or blur views. No more.

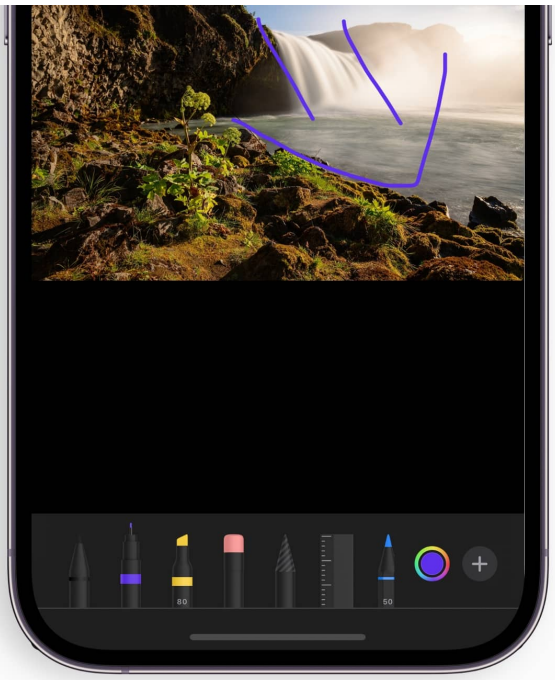
Aside from quality of life improvements, the additions have been coming at a steady pace, too. There is Smart Selection, support for adding empty space, tool picker enhancements and more – all coming at no extra cost to you. Plus, with Apple bringing its chip production in-house, it's added things like hover – which allow you to see a preview of where the Apple Pencil is currently hovering over a canvas.

All of these API enhancements continue to make Apple Pencil a compelling device to support from both a developer standpoint, and as an everyday user on iOS.

The Canvas

The next main component of PencilKit is its canvas. The canvas supports manipulation using the Apple Pencil, your finger or other third party stylus options. You can see it at work in several places across iOS – for example, Notes allows you to draw directly inside an existing note. Or, Photos, which supports markup:

```
PencilKit
> Aside from just a canvas,
  PencilKit is used all
  over iOS - as we see with
  Markup here in Photos.
```



We will look at implementation details below, but what you should notice is that the canvas is very flexible. It can be used for far more than simply a white rectangle to draw on.

How it Works

Now, let's see how these tools come to life. Thankfully, you can leverage the entirety of PencilKit with just a few objects:

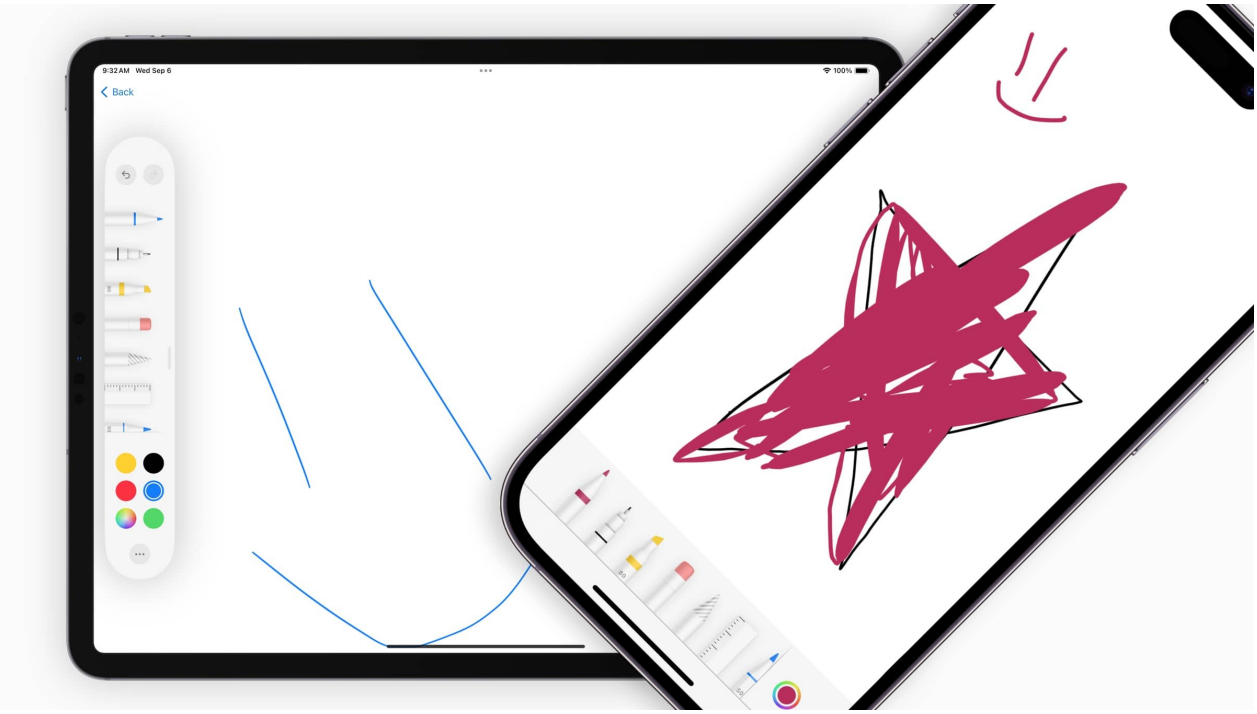
1. `PKCanvasView` is where you draw. Under the hood, it's built on top of a `UIScrollView`.
2. `PKDrawing` is the data model of that drawing. You can use this to generate images and thumbnails, or create raw data from it. That means you can persist them, send them off via AirDrop and more.
3. `PKToolPicker` is a user interface element to navigate between things you can use to draw. Using it, you can change colors, edit opacity, switch between pencils, markers and more.
4. And, `PKTools` are found within that tool picker. This object represents things like inks, erasers and lasso tools.

With that knowledge, you can create a canvas to draw on without much code:

```
// Xcode -> UIKit -> PencilKitFig1ViewController.swift
class PencilKitFig1ViewController: BaseSampleViewController {
    private let canvas: PKCanvasView = PKCanvasView(frame: .zero)
    private let toolPicker: PKToolPicker = PKToolPicker()

    override func viewDidLoad() {
        super.viewDidLoad()
        view.addSubview(canvas)
```

```
        canvas.autoresizingMask = [.flexibleWidth, .flexibleHeight]
        canvas.frame = view.bounds

        // Add tool picker
        toolPicker.addObserver(canvas)
    }

    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)
        toolPicker.setVisible(true, forFirstResponder: canvas)
        canvas.becomeFirstResponder()
    }
}
```

With that setup, you can draw with either your finger or an Apple Pencil. Depending on where you run it – you'll either get the tool picker at the bottom of the screen for compact size environments, or it'll display at a larger size with the ability to move it all around in regular size classes:



In SwiftUI – you're going to want to bridge the canvas view as there isn't a SwiftUI component for it yet. I do want to pause here and say that there is a `CanvasView`, but don't let the naming confuse you. That view is more geared towards efficient drawing with APIs such as Core Graphics.

Moving on, here's how we could simply turn around and use the canvas view controller we just created in SwiftUI:

```
// Xcode -> SwiftUI -> PencilKitFig1View.swift

struct PencilKitFig1View: View {
    var body: some View {
        PencilKitCanvasView()
    }
}
struct PencilKitCanvasView: UIViewControllerRepresentable {
    typealias UIViewControllerType = PencilKitFig1ViewController

    func makeUIViewController(context: Context) ->
UIViewControllerType {
        let drawingView = PencilKitFig1ViewController()
        return drawingView
    }

    func updateUIViewController(_ uiViewController:
UIViewControllerType,
                                context: Context) {

    }
}
```

Tools

Let's take a quick tour of the tools the tool picker offers, many of these will inherit from `PKTool`, but not all of them:

- **Inking**: Pen, marker or pencil. The width is calculated on the fly based off force, azimuth and more. In short, it scales in realtime as the user draws.
- **Eraser**: Uses vector (an object) or bitmap (a pixel) shapes to select content and remove it. Think of either a regular, circular eraser or something like shape selection to identify content to remove all in one go.
- **Lasso**: For selection — this is basically the smart lasso tool popularized by photo editing software. It intelligently selects content and encloses it, allowing you to then drag it around, manipulate it or simply erase it.
- **Ruler**: Not a `PKTool`, but rather a property on the canvas instance itself. You can toggle it with a simple boolean: `canvasView.rulerActive = true`. Then, users can either draw or mask against it.

It's a good time to not that the tool picker itself is *not* a view, and its visibility is based off of the responder chain. Even when wrapping a canvas view in SwiftUI, the tool picker's visibility will still rely on its first responder status to either hide or show:

```
let toolPicker = PKToolPicker.shared(for: window)
toolPicker.addObserver (canvasView)
toolPicker.setVisible(true, forFirstResponder: canvasView)

canvasView.becomeFirstResponder()
```

The tool picker used to only support once instance, but not you can show one per window. And, since it relies on the responder chain, dismissing it is as easy as

toggling its first responder status.

However, sometimes you want it to remain visible even when it's not the first responder. With an override, it's possible to keep it visible even when another control is first responder:

```
public func set(_ visible: Bool for responder: UIResponder) {
    // Logic to deteremine whether or not to keep its responder
status
}
```

As we saw in the screenshots of the tool picker running on iPad and iPhone, there is support for regular and small size classes. There aren't many material differences between the two, other than in regular size classes you can move it around and "dock" it to any edge of the screen. In compact size classes, it will remain docked at the bottom.

However, there is one key difference with the compact tool picker. And that is, undo and redo. In compact size classes, these are hidden – so you need to roll them on your own. Thankfully, implementation is quite easy since the canvas has undo and redo support built in anyways. You just need to provide the buttons.

Let's revisit the first UIKit example. I've added code in `viewDidLoad:` to account for this:

```
// Add undo and redo buttons
let undoBtn = UIBarButtonItem(systemItem: .undo,
primaryAction: .init { _ in
    self.canvas.undoManager?.undo()
})
let redoBtn = UIBarButtonItem(systemItem: .redo,
primaryAction: .init { _ in
    self.canvas.undoManager?.redo()
})

navigationItem.rightBarButtonItems = [undoBtn, redoBtn]

// Undo and redo logic
```

```
let sizeTraits: [UITrait] = [UITraitVerticalSizeClass.self,
UITraitHorizontalSizeClass.self]

registerForTraitChanges(sizeTraits) { (self: Self,
previousTraitCollection: UITraitCollection) in
    var showUndoRedo = false
    if self.traitCollection.horizontalSizeClass == .compact ||
self.traitCollection.verticalSizeClass ==
.compact {
        showUndoRedo = true
    }

    self.navigationItem.rightBarButtonItems?.forEach {
        $0.isEnabled = showUndoRedo
    }
}
```

Now, as trait collections change, the undo and redo button hide or show accordingly.

Another thing to consider with the tool picker is that its frame could change size. When it does, be sure you aren't obscuring any content. If you adopt the `PKToolPickerObserver` protocol, you can be notified of any frame changes that happen to it. Then, you may adjust insets or a control's frame, or even a drawing itself.

Though, even if you know that the frame of the tool picker changed, you need an easy to test if it's being obscured. There is a function to test for exact thing on `PKToolPicker`:

```
optional func toolPickerFramesObscuredDidChange(_ toolPicker:
PKToolPicker) {
    guard let obscuredFrame = toolPicker.frameObscured(in view:
myView) else {
        return
    }

    // obscuredFrame now contains a `CGRect` that's obscured in
`myView`
```

```
}
```

Tips

Use Canvas's Delegate

The `PKCanvas` has several delegate methods you can take advantage of to know things like when a drawing has completed, started and much more. If you run `PencilKitFig2ViewController` you can see these callbacks to understand where you might use them:

```
// MARK: Delegate
extension PencilKitFig2ViewController: PKCanvasViewDelegate {
    func canvasViewDrawingDidChange(_ canvasView: PKCanvasView) {
        print(#function)
    }

    func canvasViewDidEndUsingTool(_ canvasView: PKCanvasView) {
        print(#function)
    }

    func canvasViewDidFinishRendering(_ canvasView: PKCanvasView) {
        print(#function)
    }

    func canvasViewDidBeginUsingTool(_ canvasView: PKCanvasView) {
        print(#function)
    }
}
```

Knowing when tools show or hide is useful to provide feedback, especially if you're providing custom tooling in addition to the tool picker. Rendering also becomes im-

portant when restoring existing drawings from disk – sometimes this could take a bit of time, so it's key to hide and show loading elements.

Right and Left Hand

Some people will draw right handed, some will draw left handed. As such, you might consider a toggle to flip any custom interface elements you have over a canvas from side to side. This problem is particularly unique to drawing apps, really – but its utility could also extend beyond it (the concept of a "left handed mode" or similar).

If you run `PencilKitFig2View.swift` you can see this idea in action. Imagine that the red squares represent some sort of custom buttons you've included. The top menu in the navigation bar toggles between a right and left handed mode, which changes where the controls are pinned (either leading or trailing):

```
struct PencilKitFig2View: View {
    enum HandMode: Int, CustomStringConvertible, CaseIterable,
Identifiable {
        case rightHanded, leftHanded

        var description: String {
            switch self {
            case .rightHanded: return "Right Handed"
            case .leftHanded: return "Left Handed"
            }
        }

        var id: Self { self }

        var glyph: String {
            switch self {
            case .rightHanded: return "hand.point.right.fill"
            case .leftHanded: return "hand.point.left.fill"
            }
        }
```

```
    }

    @State private var handMode: HandMode = .rightHanded
    @State private var controlAlignment: Alignment = .leading

    var body: some View {
        ZStack {
            PencilKitCanvasView()
        }
        .overlay(alignment: controlAlignment) {
            VStack(spacing: 16) {
                Rectangle()
                    .fill(Color.red)
                    .frame(width: 44, height: 44)
                Rectangle()
                    .fill(Color.red)
                    .frame(width: 44, height: 44)
                Rectangle()
                    .fill(Color.red)
                    .frame(width: 44, height: 44)
            }
            .padding(.horizontal)
            .animation(.easeOut, value: controlAlignment)
        }
        .toolbar {
            Menu("Hand Mode") {
                ForEach(HandMode.allCases) { mode in
                    Button(mode.description, systemImage:
mode.glyph) {
                        self.handMode = mode
                    }
                }
            }
        }
        .onChange(of: handMode) {
            controlAlignment = handMode == .rightHanded ? .leading :
.trailing
        }
    }
```

```
}
```

Prefer Only Pencil Drawing

There are a few different drawing policies to be aware of. For example, `UIPencilInteraction.prefersPencilOnlyDrawing` will mean the canvas only accepts input from an Apple Pencil. This is a global setting that is toggled via the boolean.

However, you can set it directly on a canvas instance. You can change it by assigning to the `drawingPolicy` property. The policies are:

1. **Any**: Your finger, a stylus or Apple Pencil. The canvas accepts any input.
2. **Pencil**: Here, the pencil draws and the finger will only scroll or be used for smart selection.
3. **Default**: If the tool picker is showing, then it respects the global setting we mentioned above. If it's hidden, then it only uses the pencil.

Hide Effects View or Blurring

If you stumble across some of Apple's own PencilKit code or examples online, they may have logic to toggle a blurred navigation bar:

```
func updateNavigationBarBackground() {
    // Turn off the navigation bar background as it will affect
latency.
    UIGraphicsBeginImageContext(CGSize(width: 1, height: 1))
    UIColor.secondarySystemBackground.withAlphaComponent(0.95).set()
    UIBezierPath(rect: CGRect(x: 0, y: 0, width: 1, height:
1)).fill()

navigationBar.setBackgroundImage(UIGraphicsGetImageFromCurrentImageC
ontext(),
for: .default)
```

```
    UIGraphicsEndImageContext()
}
```

If you're targeting iOS 16 or above – this is no longer necessary. PencilKit has optimizations under the hood to account for this, which means you don't have to worry about potential performance hits when it comes to blurring or visual effect views.

Supporting Pencil Taps

The Apple Pencil has a neat little feature where if you double tap it, you can perform some arbitrary logic. However, Apple has default uses for these – and they can be toggled in the Settings app. Always use their preferred settings, or if you can't – do non-destructive mode switching via `UIPencilInteraction` and `UIPencilInteractionDelegate`:

```
// In viewDidLoad or similar...
let interaction = UIPencilInteraction()
interaction.delegate = self
view.addInteraction (interaction)

extension MyViewController: UIPencilInteractionDelegate {
    func pencilInteractionDidTap(_ interaction: UIPencilInteraction)
{
        switch UIPencilInteraction.preferredTapAction {
        case .ignore:
            print("doNothing()")
        case .switchEraser:
            switchEraser()
        case .switchPrevious:
            enablePreviousTool()
        case .showColorPalette:
            toggleColorPallete()
        case .showInkAttributes:
            presentInkingMetadata()
        @unknown default:
```

```
            print("Unknown interaction.")
        }
    }
}
```

Handling Dark Mode

Thankfully, PencilKit does handle dark mode automatically, but you may not want that. Whether it's due to stylistic purposes or anything else, you can use `overrideUserInterfaceStyle` to keep one color scheme or the other. Though, again, I quite enjoy how the inks and palette adapt without any extra work – but if someone is trying to draw an intricate portrait with specific colors, you can opt out of it.

Supporting the Screenshot Gesture

Apple Pencil has a neat little trick, wherein you swipe from the bottom left of the screen to take a screenshot. We can respond to that in our own code. When a screenshot is taken with Apple Pencil, you can hook into that via `UIScreenshotServiceDelegate`.

For example, maybe you want to present something as a .pdf instead of another format. This would be the perfect place to do it:

```
// UIScreenshotService on UIWindowScene
class DrawingViewController: UIScreenshotServiceDelegate {
    override func viewwillAppear() {
        super.viewWillAppear()
        let scene = view.window.windowScene
        scene.screenshotService.delegate = self
    }
}

// Handle a screenshot, send back .pdf data
func screenshotService(
```

```
UIScreenshotService,
generatePDFRepresentationWithCompletion: (Data?, Int, CGRect) ->
Void) {
    // Generate .pdf data
    // Handle indexOfPage
    // handle rectIncurrentPage in .pdf coordinates, which is bottom
left, not top right like view.
}
```

Indirect Interaction Support

One neat aspect of Apple Pencil and scribble support is that you can support "text entry" in empty white space. For example, Reminders works this way – if you start writing text in empty space with Apple Pencil, it'll not only support that, but it'll create a new reminder when you're finished.

You can provide the same experience using `UIIndirectScribbleInteraction`. It works like any other `UIInteraction` – simply install in on a view and adopt its protocol. This way, you could have any empty space accept scribble input and react to it accordingly.

Inspecting Strokes

Strokes are data models themselves, which means you can get annotations, support animations and more. You can access the individual paths and operate on them to several interesting things like trace back strokes to help someone learn how to write a letter correctly.

To inspect a drawing, you'll inspect its array of strokes. Each stroke *itself* has a path, or more simply – the shape of the stroke. Further, each one has an ink associated to it along with a transform representing its position along with a mask. Finally, the render bounds is a rectangle of all of those properties – but aggregated together.

Phew!

It sounds like a lot, but if you want to draw a path, it's likely you actually don't want the points as they are represented by default. Those are B-spline control points. Instead, you'd want to interpolate the spline. I know that's a lot of words and fun phrases, but in practice it means you likely *don't* want to do this:

```
for stroke in canvas.drawing.strokes {
    let path = stroke.path
    for point in path {
        // Use point
    }
}
```

And you *do* want to do this:

```
for stroke in canvas.drawing.strokes {
    let path = stroke.path
    for point in path.interpolatedPoints(by: .distance(10)) {
        // Use point
    }
}
```

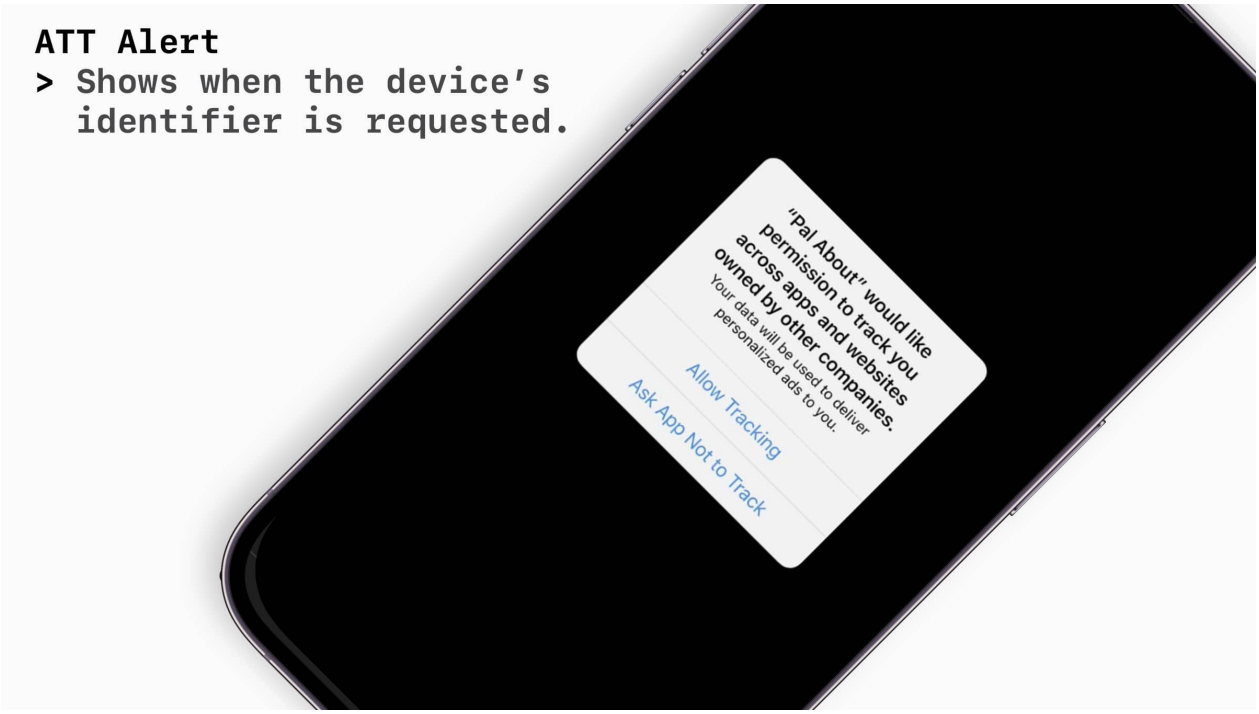
Three Key Takeaways

1. PencilKit is the best choice for any drawing needs in an iOS app.
2. Remember that Apple Pencil can be used for more than just drawing; it's used to mark up .pdf files and more.
3. Look for fun and novel ways to incorporate advanced features, such as hover.

Security and Privacy

Security and privacy are synonymous with iOS – as well they should be. You don't have to look far to find Apple themselves advertising the security and privacy aspects of iOS. Tim Cook himself has even declared these things a "fundamental human right", and I agree.

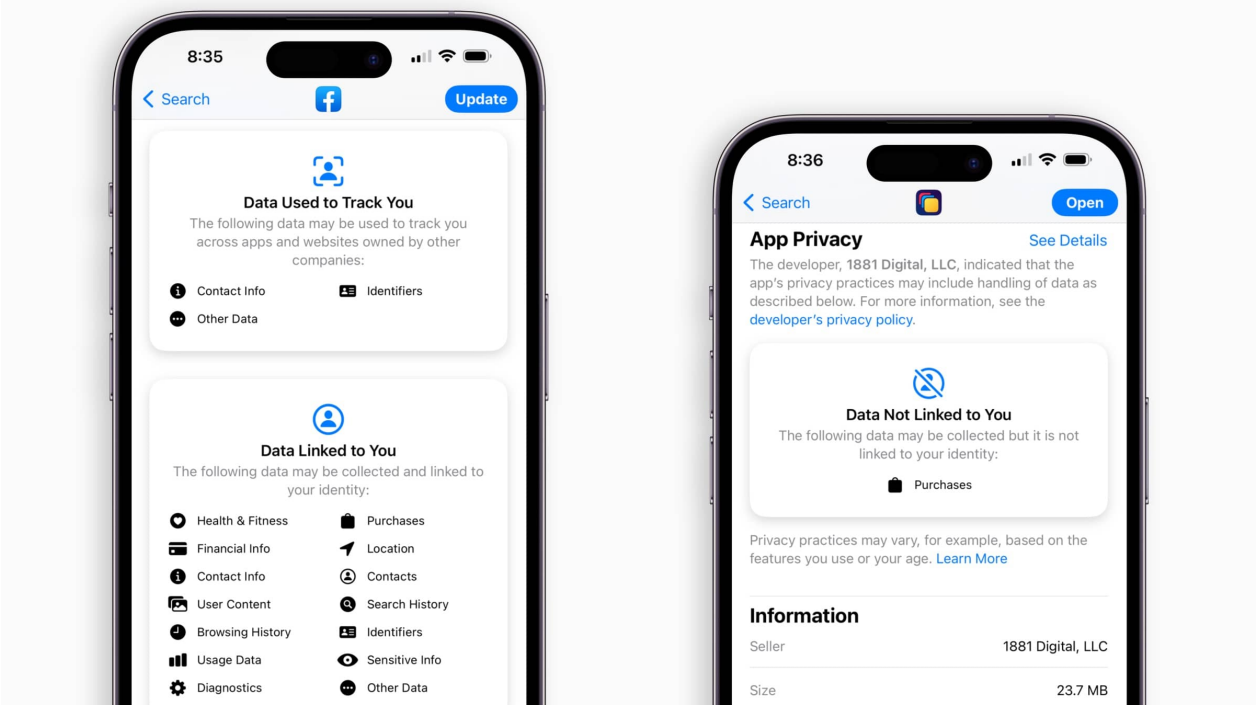
Don't take my word for it, though. You don't have to look far to see security and privacy at play in Apple's ecosystem. It wasn't even but a few years ago with iOS 14.5 that Apple effectively ended free access to device identifiers – which advertisers had long relied on for cross-platform tracking and ad targeting. Now, an alert pops up when that's requested, and it asks the user if they want to allow their device identifier to be used for tracking:



…and guess how many people tap "Allow Tracking".

Not many, and while it puts advertisers in a pinch, it does give more choice to people about how they are tracked online. It was, and still is, a controversial move among some – but it undoubtedly follows Apple's modus operandi. Going further, how app the App Store "nutrition labels" that debuted not too long ago? Now, what the app requests of your data is front and center. Anyone can see it before they even make the decision to download the app.

For some, it's a badge of pride – there's nothing to hide because there's nothing to gain from sucking up your data. For others, well – it falls on the other end of the equation:



And, I don't say that to come down on companies that *do* require data for their business to operate, but the point is that Apple's stance is clear: If you're using data, then people should:

- Know about it, and
- Be able to control how much you're willing to share.

This ideas even extends to Apple's own APIs. Recently, for example, iOS 15 added the ability to use CloudKit to provide encryption and key management for basically all primitive data types. Now, strings, dates and numbers can all be stored securely. And, frameworks like Local Authentication allow you to use on-device security measures, such as Touch ID and Face ID (as we'll see below). Even Core Location offers a location button API (`LocationButton` in SwiftUI, and `CLLocationButton` in UIKit) to easily and privately access the user's location temporarily.

Extending that thought our a bit more — nearly all of Apple's APIs that deal with the user's data require opt-in flows too. If you want to use any of the following data, you have to get user permission first to access it:

- Personal data such as the user's location, any health data, financial information, on-device contacts and other personally identifying information.
- Any "user-generated" content, which means things like emails, messages, calendar data, gameplay information, Apple Music activity, HomeKit data, and audio, video, and photo content.
- All protected resources. This means Bluetooth peripherals, home automation features, Wi-Fi connections, or local network access.
- Pretty much any device capabilities, such as the camera and microphone.
- In visionOS, you need permission to run in a Full Space, get ARKit data such as hand tracking, plane estimation, image anchoring, and world tracking.
- And, of course, the device's advertising identifier.

All of these things can be toggled at anytime, so you need to check for their authorization states before you access them in your code. A user could grant photo library access, but then turn it off before they come back to your app within iOS' Settings app under the "Privacy and Security" section.

Even when it *does* come to your personal information, Apple endeavors to process anything they possibly can on-device. This is essentially better in every way — data doesn't leave your device to go to some server, and it's more efficient and certainly quicker. When you open Photos and see that iOS recognizes faces and puts them to names, all of the processing to enable that occurred on your own device (in this case, typically at night when you're charing your phone).

Apple wants your data to stay on your device. And, the platform is better for it. I could go on with more examples, but hopefully the idea is clear: respect the user's data, don't take more than you need and always ask for it.

So, What Can We Do?

To that end, we are going to have a discussion over how to correctly frame your thinking when it comes to a user's personal data, and how to enforce a secure, respectful experience for your apps.

If you want a cheat sheet, though, Apple has provided one for us. There are four core "privacy pillars" that Apple endeavors to uphold, so we should try and do the same:

1. **Data Minimization**: Only ask, and take, what you need – but only after you have permission to do so.
2. **On-Device Processing**: Don't send to a server what you can operate on locally.
3. **Transparency and Control**: Make what you track or use completely, utterly obvious – and allow them to change any part of that equation.
4. **Security Protections**: People expect you to be secure and respectful of their data. So, avoid careless mistakes, data leaks or other similar situations by using the correct APIs and practices.

As always, when a chapter is structured a little bit differently I always call that out in the lede – and this chapter is no different. These is not one specific framework that encompasses security and privacy on iOS, rather – it's a mindset you apply to the code you wright *and* a combination of using the proper APIs to ensure data is secure and private.

For a more concrete example, consider tracking. I'm not oblivious to the fact that having some tracking in your app can be useful. It helps you make product decisions, know when a feature is used and more. There is, objectively speaking – a good and bad way to do it, though.

When it comes to tracking things, always ask the user. Be clear about what you collect. Provide a human readable privacy policy. Include options to easily opt out of it. The list goes on – but basically the idea is to develop your apps with a high degree of empathy for the user's information.

Now, let's look at some APIs we can use when it comes to security and privacy on iOS.

Tips

Always Check Permission States

Remember that list of things I mentioned you need user permission for *before* you can access them? Health data, sleep data, contacts and so on? As I alluded to earlier, don't assume that because they opted into before that you still have access. Users can turn these permissions off at anytime.

So, when you access these APIs (no matter which one) you should take the same approach – just assume you may not have permission. Consider an app which reads and writes caffeine data, here's how the retrieval of that data should be structured every time.

First, I find it helps to abstract away the actual authorization check to its own function – you'll be using it often:

```
func requestHealthPermissions () async -> Bool {
```

```
    guard !isMocked else { return false }
    let write: Set<HKSampleType> = [
        HKQuantityType(.dietaryCaffeine)
    ]

    let read: Set<HKObjectType> = [
        .quantityType(forIdentifier: .dietaryCaffeine)!
    ]

    let res: ()? = try? await health.requestAuthorization(toShare:
write, read: read)
    guard res != nil else {
        return false
    }

    return true
}
```

Then, be sure to check that during your app's launch and handle it accordingly. Again, this doesn't mean toss up a permission prompt right away – do it where it makes the most sense and provide context for why you're asking for permissions again.

Contextualized prompts always work better. For example, if you *had* access, but have lost it: "We've lost access to read your caffeine data." Or, if you're asking for the firs time: "We'll need access to your caffeine data stored on your device. We don't send this data anywhere, all of it says on your device."

From there, when you go to fetch the data – check that authorization status again. In our example, HealthKit has a convenient function to check if you have permission to use individual types of HealthKit data:

```
func getTodaysCaffeine() async throws -> Double {
    guard !isMocked, health.authorizationStatus(for:
HKQuantityType(.dietaryCaffeine)) ==
.sharingAuthorized else { return 0.0 }
```

```
    let sumToday = try await sumQuery().result(for: health)?
        .sumQuantity()?
        .doubleValue(for: HKUnit.gramUnit(with: .milli)) ?? 0.0

    return sumToday
}
```

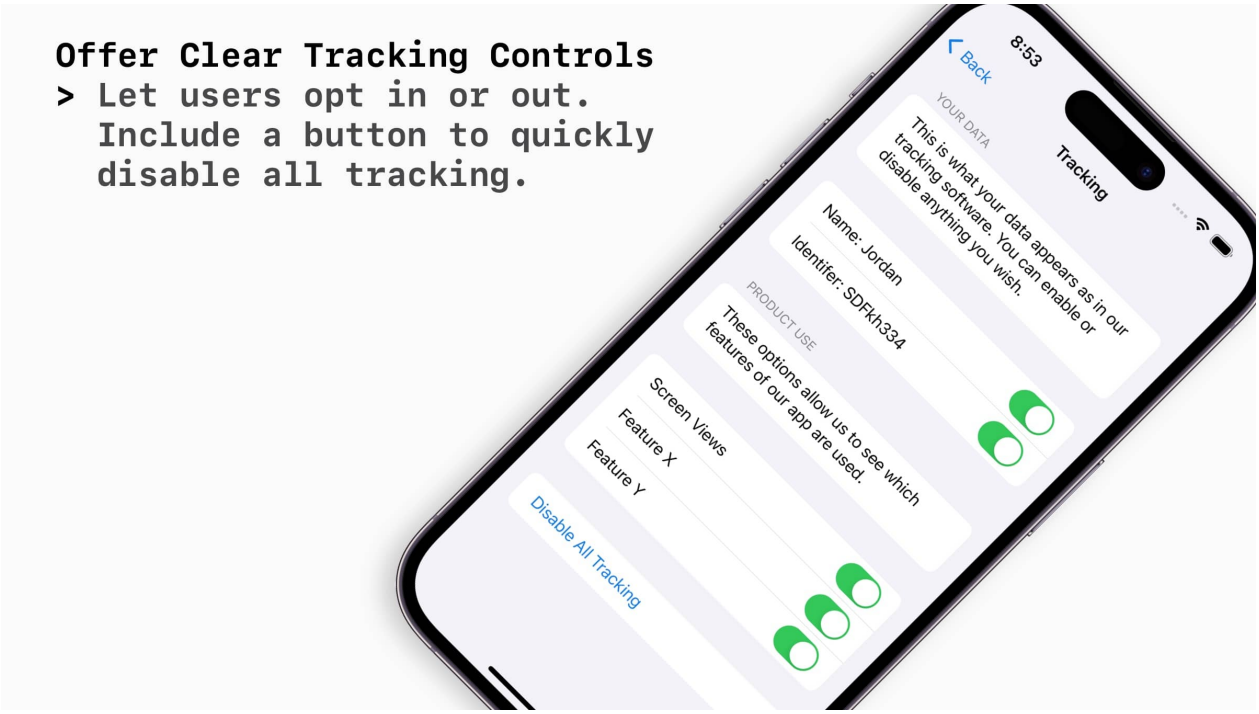
While the APIs across iOS look a little different, all of them follow the same pattern. Check for an authorization state, prompt for why you need any particular data and be sure to continually check your authorization state during any reads or writes to that data.

Offer Tracking Control Settings

If you have tracking capabilities in your app, try to do two things:

1. Explain exactly what you track, even outside of the App Store privacy details.
2. Offer the user an option to disable it.

A common place to put something like this is in your settings area, or the equivalent of that. Make it clear and simple to understand the state of what's being tracked, provide easy ways to disable particular fields and add a way to turn off all tracking altogether:



Use Context Types on Text Controls

When you set the correct content types on text controls, users can leverage Apple's privacy features built right into iOS. This includes features such as Hide my Email, Mail Privacy Protection, Password Autofill and more.

Whether it's a credit card number, address or anything else iOS probably has knowledge of regarding the user, there is probably a text content type for it. Be sure to always set it:

```
// SwiftUI
TextField("Email", text: $email))
    .textContentType(.emailAddress)

// UIKit
let tf = UITextField(frame: .zero)
tf.textContentType = .emailAddress
```

For more information on how to use text content types, view the chapter “Text that Works” in this book series.

Store Sensitive Data in the Keychain

Keychain Services is the right choice when it comes to storing sensitive user data locally. This is an encrypted database you can use, meaning it’s an incredibly secure option from an API perspective. Things such as notes, cryptographic keys, certificates and identifiers are ideal candidates to be stored in the user’s keychain.

Here’s an example using the keychain APIs to add a password and username, associated with this book series domain, to the keychain:

```
// Xcode -> SwiftUI -> SPFig3View.swift

enum KeychainAdditionError: Error {
    case invalidData
    case osError(status: OSStatus)
}

struct UserLoginInfo {
    var accountName: String
    var password: String

    static var empty: UserLoginInfo {
        return .init(accountName: "", password: "")
    }

    static let serviceDomain = "www.bestinclassiosapp.com"
}

struct SPFig3View: View {
    @State private var userData: UserLoginInfo = .empty
    @State private var result: String = "Awaiting Add Request"

    var body: some View {
```

```
        Form {
            Section {
                Text("Enter in credentials to securely store in the
keychain.")
            }
            Section {
                TextField("Account Name", text:
$userData.accountName)
                TextField("Password", text: $userData.password)
                Button("Add to Keychain") {
                    do {
                        try addToKeychain()
                    } catch {
                        result = error.localizedDescription
                    }
                }
            }
            Section("Status") {
                Text(result)
            }
        }
        .navigationTitle("Keychain Storage Example")
        .navigationBarTitleDisplayMode(.inline)
    }

    // MARK: Private Functions

    private func addToKeychain() throws {
        guard let passwordData = userData.password.data(using:
String.Encoding.utf8),
              userData.accountName.count > 2 else {
            return
        }

        let name = userData.accountName

        // Setup data
        let credentialInfo: [String: Any] = [kSecClass as String:
kSecClassInternetPassword,
```

```
                                        kSecAttrServer as String:
UserLoginInfo.serviceDomain,
                                        kSecAttrAccount as String: name,
                                        kSecValueData as String:
passwordData]

        // Request addition
        let keychainAdditionStatus = SecItemAdd(credentialInfo as
CFDictionary, nil)

        guard keychainAdditionStatus == errSecSuccess else {
            throw KeychainAdditionError.osError(status:
keychainAdditionStatus)
        }

        result = "Credentials added to keychain!"
    }
}
```

Local Authentication

Biometric authentication is a fantastic way to add an extra level of security to your app. Whether you are financial app with sensitive data, or simply want to add the ability to "lock" a note or something similar – the Local Authentication framework is where you should look.

Depending on the device, the Local Authentication framework will either prompt the user for Touch ID, Face ID, Optic ID or the device's password to proceed. Which one is abstracted away from you – you don't chose it, whichever is available is what's used. Here's a code sample showing you how can use biometric auth:

```
// Xcode -> SPFig1View.swift
import LocalAuthentication

class BiometricAuth {
```

```
    lazy var context: LAContext = {
        var ctx = LAContext()
        ctx.localizedReason = "View secure transactions."
        return ctx
    }()

    func canPerformAuth() -> Bool {
        var error: NSError?
        guard
context.canEvaluatePolicy(.deviceOwnerAuthenticationWithBiometrics,
                                          error: &error) else {
            print(error?.localizedDescription ?? "Unable to perform
local auth.")
            return false
        }

        return true
    }

    func performAuth() async -> Bool {
        do {
            try await
context.evaluatePolicy(.deviceOwnerAuthentication,
                                                localizedReason: "View
your recent transactions.")
            return true
        } catch let error {
            print(error.localizedDescription)
            return false
        }
    }
}
```

Though, as we've mentioned multiple times throughout this chapter, you need to provide a privacy reason for why you're even attempting to use biometric authentication anyways. Simply add the `NSFaceIDUsageDescription` to your info.plist along with your reason (i.e. "We'll use Face ID to view your transactions.").

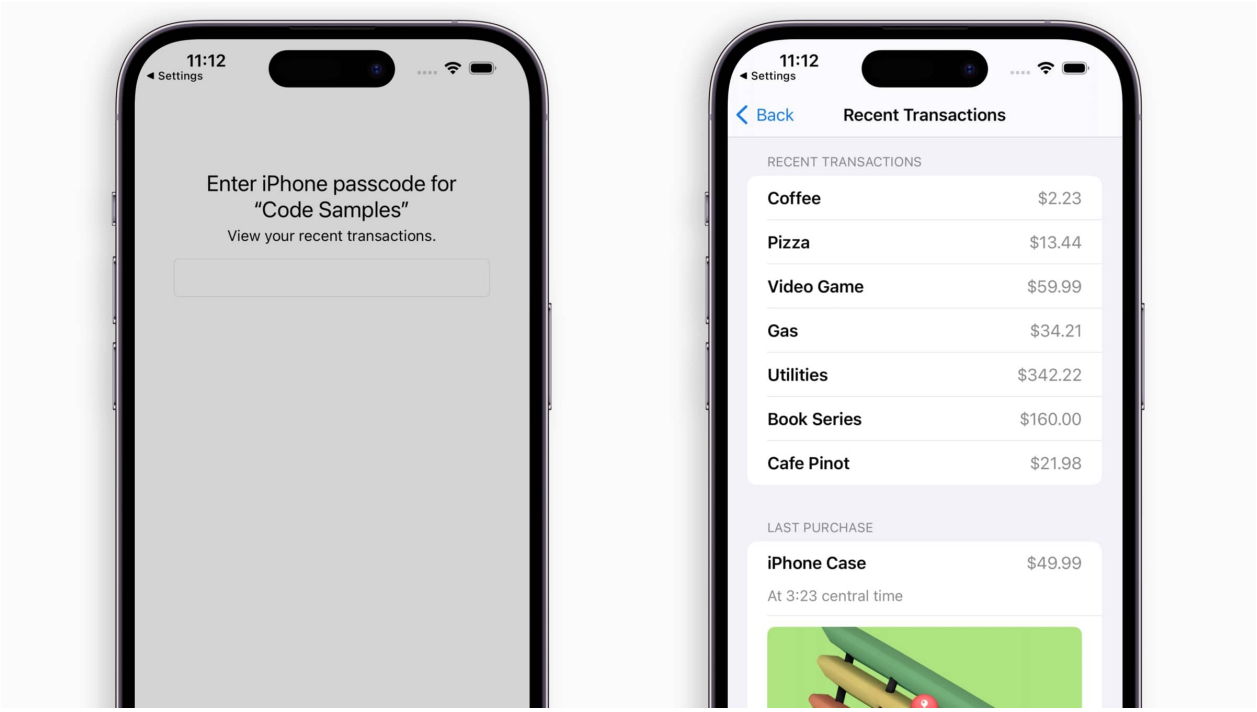
If you check out `SPFig1View.swift` – you'll see that we have a mock list of transactions from a fictional bank account. Of course, this is data you probably don't want front and center, so we could show a blur view over it and kick off biometric authentication once it's viewed. If it was successful, we'll remove the blur view:

```
// Xcode -> SwiftUI -> SPFig1View.swift

struct SPFig1View: View {
    private let auth: BiometricAuth = .init()
    @State private var isAuthenticated: Bool = false

    var body: some View {
        Form {
            Section("Recent Transactions") {
                ForEach(DemoTransactions.mockData) { transaction in
                    HStack {
                        Text(transaction.title)
                            .font(.headline)
                        Spacer()
                        Text(transaction.amount,
format: .currency(code: "USD"))
                            .foregroundStyle(Color(uiColor: .seconda
ryLabel))
                    }
                }
            }
            Section("Recent Transactions") {
                ForEach(DemoTransactions.mockData) { transaction in
                    HStack {
                        Text(transaction.title)
                            .font(.headline)
                        Spacer()
                        Text(transaction.amount,
format: .currency(code: "USD"))
                            .foregroundStyle(Color(uiColor: .seconda
ryLabel))
                    }
```

```
                }
            }
            Section("Last Purchase") {
                VStack {
                    HStack {
                        Text("iPhone Case")
                            .font(.headline)
                        Spacer()
                        Text(49.99, format: .currency(code: "USD"))
                            .foregroundStyle(Color(uiColor: .seconda
ryLabel))
                    }
                    .padding(.bottom, 8)
                    Text("At 3:23 central time")
                        .frame(minWidth: 0, maxWidth: .infinity,
alignment: .leading)
                        .foregroundStyle(Color(uiColor: .secondaryLa
bel))
                        .font(.subheadline)
                }
                Map() {
                    Marker("Apple Park",
                           coordinate: .applePark)
                }
                .frame(height: 200)
                .clipShape(RoundedRectangle(cornerRadius: 8))
            }
            Section("Last Purchase") {
                VStack {
                    HStack {
                        Text("iPhone Case")
                            .font(.headline)
                        Spacer()
                        Text(49.99, format: .currency(code: "USD"))
                            .foregroundStyle(Color(uiColor: .seconda
ryLabel))
                    }
                    .padding(.bottom, 8)
                    Text("At 3:23 central time")
```

```
                        .frame(minWidth: 0, maxWidth: .infinity,
alignment: .leading)
                        .foregroundStyle(Color(uiColor: .secondaryLa
bel))
                        .font(.subheadline)
                }
                Map() {
                    Marker("Apple Park",
                           coordinate: .applePark)
                }
                .frame(height: 200)
                .clipShape(RoundedRectangle(cornerRadius: 8))
            }
            Button("View More...") { }
        }
        .navigationTitle("Recent Transactions")
        .navigationBarTitleDisplayMode(.inline)
        .overlay {
            if !isAuthenticated {
                Rectangle()
                    .fill(.thinMaterial)
                    .ignoresSafeArea()
            }
        }
        .animation(.easeOut, value: isAuthenticated)
        .onAppear {
            Task {
                self.isAuthenticated = await auth.performAuth()
            }
        }
    }
}
```

You can even run this in the simulator, you'll get a password prompt:



A few notes here. If you are exclusively using Face ID, it helps to provide a button to kick-off the authentication. It can be jarring to have the scan happen right away, it feels more natural to let the user consciously press a button to begin authentication — they may not always want to.

Think of Touch ID — there, the user has to make the decision to scan their fingerprint. But if you suddenly begin Face ID, they no longer have any agency in the decision since you've made it for them.

Finally — consider scene changes. If the app is backgrounded, detect that and put the blur view back up to obstruct information. This way, it's not accidentally revealed in the app switcher.

Don't Add Third Party Code Without Thought

Simply put — some of the third party code vendors out there can be a real privacy issue. This is doubly true when its come to analytics software. In some ways, privacy and

data tracking are simply at odds with another. The former requires you to let go of data hoarding for the sake of respecting one's personal data. The latter requires you to consume as much data as possible to *know* one's personal information.

So, be sure to audit anything you add. Try to advocate for *less* tracking within your organization, and always question assumptions. Do we really need to know the user's email address, or can we simply tie a unique, generated identifier to them instead? Extrapolate that line of thinking across all of the software you write or any third party software you consider adding to your app.

Three Key Takeaways

1. Security and privacy are hallmark expectations on the iOS platform.
2. Instead of thinking of data to protect, simply frame your thinking differently – all data is data to protect.
3. Always make sure the user controls all of their data and information in your apps, offer exporting options as well.

Safari Services

Web browsing on iOS used to be a bit of a chore from a developer standpoint. There were several projects from developers who leveraged the WebKit framework to attempt to build out, well, Safari. And who could blame us? Safari is the browser that most people use on iOS, and so for many reasons – the majority simply want a Safari experience when opening websites within others apps.

Because there was no official way to use Safari *inside* of an app (you could always deep link *out* to Safari), what ended up happening was somewhat of a fragmented experience. While some simply made a barebones "web view", others tried to make things fully featured, akin to a legitimate browser, but with a different design than Safari had. And so, it was a little confusing – but now consider extrapolating that problem out to the massive, global user base that iOS has. Then, you can see how people might've been a little lost experiencing numerous permutations of a custom web browser inside the apps they used.

To them, it was simply a confounding experience – why is this web browser different from the one I typically use (i.e., Safari)? To developers, we knew why – because Apple didn't provide us with a Safari control, so we were just doing the best that we could.

Thankfully, this all changed with iOS 9 and the Safari Services framework. Using it, we gained access to the `SFSafariViewController` – and true to its name, it's a developer-friendly Safari browser view we can present. This carries several advantages, chief among them, though, is *familiarity* and *trust*. People know what they are using, and developers know what they are offering.

This chapter is all about how to leverage that view controller, both in SwiftUI and UIKit. Yes, that's right, as of now there is no SwiftUI-first browsing view. But, that's ok, we can easily wrap it.

Let's start by looking at its implementation.

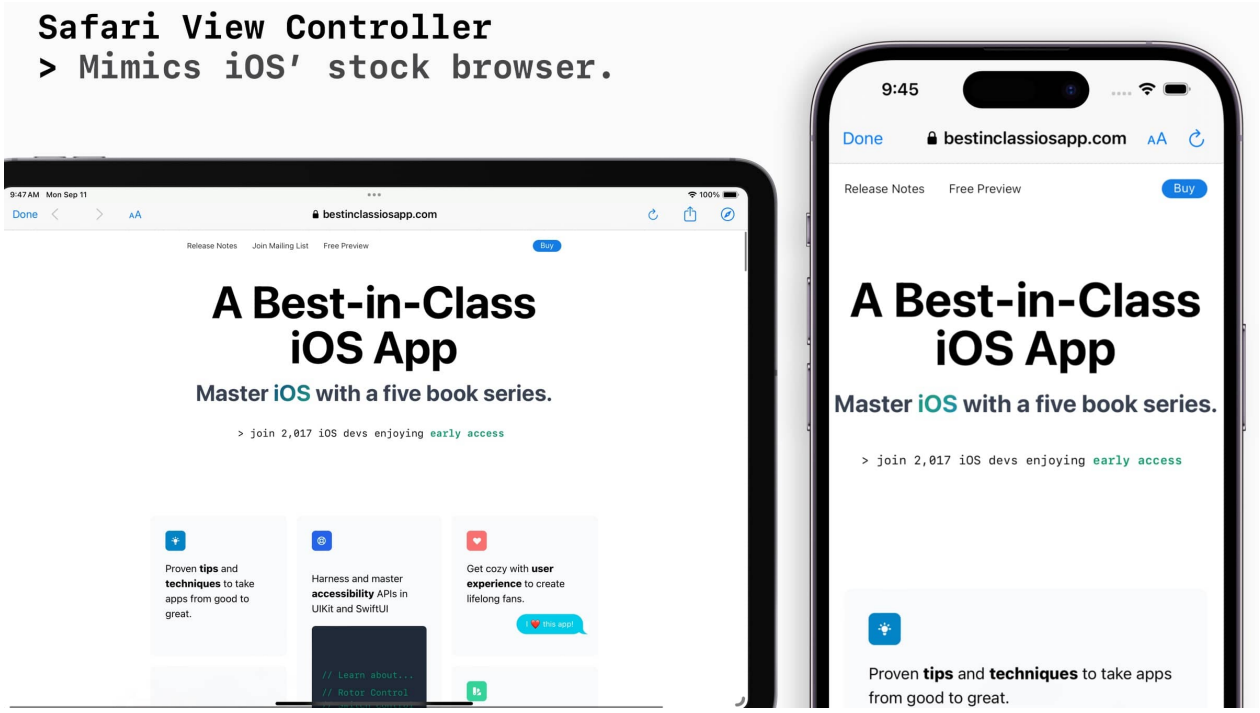
How it Works

The Safari view controller is trivial to use. At its core, simply provide it a valid URL and present it. Done! Here's a fully-featured safari view controller with all the bells and whistles without using much more than a few lines of code:

```
// Xcode -> UIKit -> SafariServicesFig1ViewController.swift

import UIKit
import SafariServices

class SafariServicesFig1ViewController: BaseSampleViewController {
    private lazy var urlTF: UITextField = .init(frame: .zero,
primaryAction: .init(handler: { _ in
```

```
        guard let address = URL(string: self.urlTF.text ?? "") else
{
            return
        }

        let webVC = SFSafariViewController(url: address)
        self.present(webVC, animated: true)
    }))

    override func viewDidLoad() {
        super.viewDidLoad()
        urlTF.placeholder = "Enter in URL"
        urlTF.textContentType = .URL
        urlTF.keyboardType = .URL
        urlTF.autocapitalizationType = .none
        view.addSubview(urlTF)

        urlTF.translatesAutoresizingMaskIntoConstraints = false
        NSLayoutConstraint.activate([
            urlTF.centerXAnchor.constraint(equalTo:
view.centerXAnchor),
            urlTF.centerYAnchor.constraint(equalTo:
view.centerYAnchor),
            urlTF.widthAnchor.constraint(equalTo:
view.readableContentGuide.widthAnchor),
            urlTF.heightAnchor.constraint(equalTo:
urlTF.heightAnchor)
        ])
    }
}
```

In this sample, simply type in a URL and the Safari browser will present, loading that content:



Now, you don't have to reinvent the wheel and roll your own web browser. But more importantly, we get all of the features Safari brings:

- Interactions within `SFSafariViewController` aren't visible to us, which means it's secure.
- Users can leverage AutoFill data, browsing history, and website data. You don't have to worry about offering any of that.
- Using `ASWebAuthenticationSession`, authenticating with services people have interacted with is much easier and frictionless.
- Plus, all of the things Safari offers that people expect. The ability to add content to reading lists, history, and more are available to use.

Over in SwiftUI, we'll need to wrap it. At the minimum, we'll need to pass in a URL to the wrapped controller:

```
import SwiftUI
```

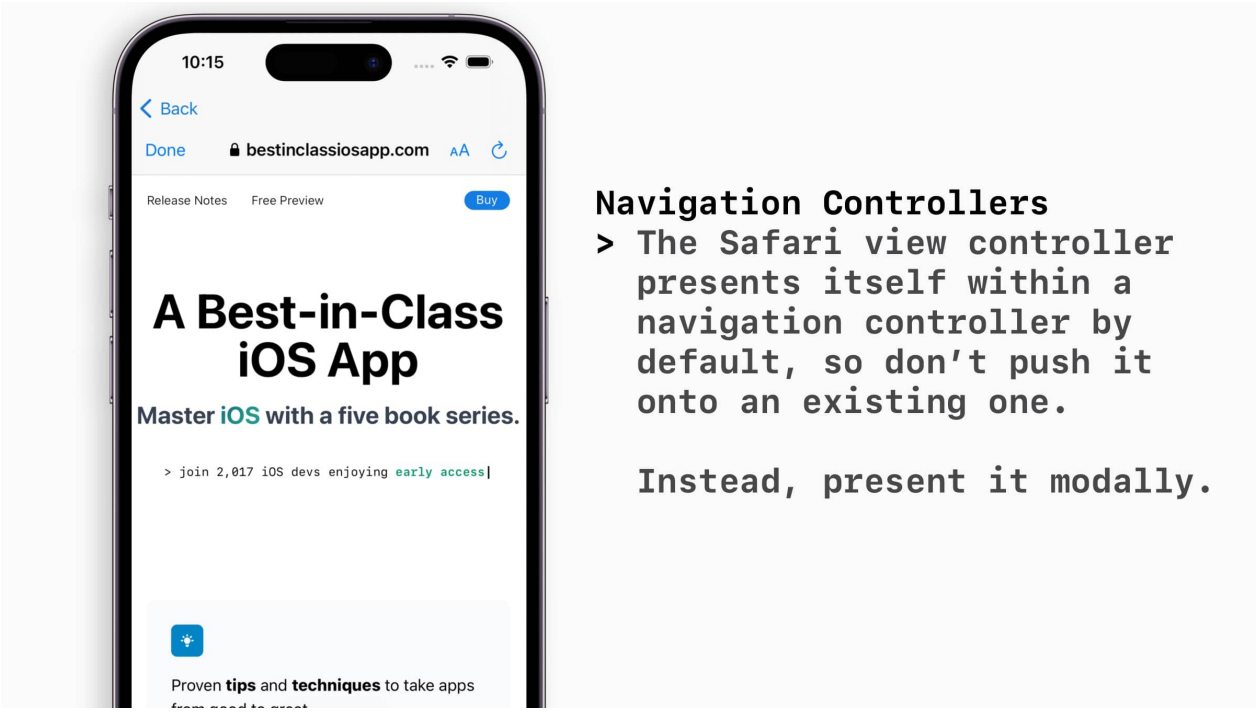
```
import SafariServices

struct SafariBrowserView: UIViewControllerRepresentable {
    let url: URL

    func makeUIViewController(context: Context) ->
SFSafariViewController {
        return SFSafariViewController(url: url)
    }

    func updateUIViewController(_ uiViewController:
SFSafariViewController,
                                context: Context) {

    }
}
```

From there, we're good to present it. This example mimics the UIKit one we just created, where someone enters a URL, and it's opened:

```
// Xcode -> SwiftUI -> SafariServicesFig1View.swift

import SwiftUI

extension URL: Identifiable {
    public var id: String {
        return UUID().uuidString
    }
}

struct SafariServicesFig1View: View {
    @State private var link: String = ""
    @State private var destination: URL? = nil

    var body: some View {
        VStack {
            TextField("", text: $link, prompt: Text("Enter in URL"))
                .textContentType(.URL)
```

```
                .keyboardType(.URL)
                .autocorrectionDisabled()
                .textInputAutocapitalization(.never)
                .onSubmit {
                    guard let enteredURL = URL(string: link) else {
                        print("Invalid URL.")
                        return
                    }

                    self.destination = enteredURL
                }
        }
        .padding()
        .fullScreenCover(item: $destination) { url in
            SafariBrowserView(url: url)
                .ignoresSafeArea()
        }
    }
}
```

However, there are a few things to know whether you're using UIKit or SwiftUI. Chief among them is that Safari view controller is embedded in a navigation controller by default. This means that if you present it via pushing onto a navigation stack, you're certain to get odd results. For example, look at the double navigation stack here:



So, that means avoiding presenting it like this in UIKit:

```
navigationController?.pushViewController(webVC,
                                         animated: true)
```

Or via navigation links in SwiftUI:

```
struct SafariContentView: View {
    private let bicLink: URL = URL(string: "https://
www.bestinclassiosapp.com")!

    var body: some View {
        NavigationStack {
            NavigationLink("Show Best-in-Class Website") {
                SafariWebView(url: bicLink)
            }
        }
    }
}
```

In either case, Safari view controller is meant to be presented modally. So use sheets and full-screen covers in SwiftUI, or UIKit APIs to present it as either a form sheet or any other modal presentation.

Tips

Configuration Options

The Safari view controller offers a class to customize how it's presented via `SFSafariViewController.Configuration`. For example, you could opt to always enter into reader mode (if available), collapse the navigation bars on scroll and more.

Further, you can tweak tinting options as well. Things such as bar tint and control tint can be set, though be aware that these are the classic Apple APIs prepended with "preferred", so the system may override your choices.

I've augmented our `SafariBrowserView` a bit to support these configurations:

```
struct SafariBrowserView: UIViewControllerRepresentable {
    let url: URL
    let options: SFSafariViewController.Configuration
    let barTint: Color
    let controlTint: Color

    init(url: URL,
         options: SFSafariViewController.Configuration = .init(),
         barTint: Color = .primary,
         controlTint: Color = .init(uiColor: .label)) {
        self.url = url
        self.options = options
        self.barTint = barTint
        self.controlTint = controlTint
    }
```

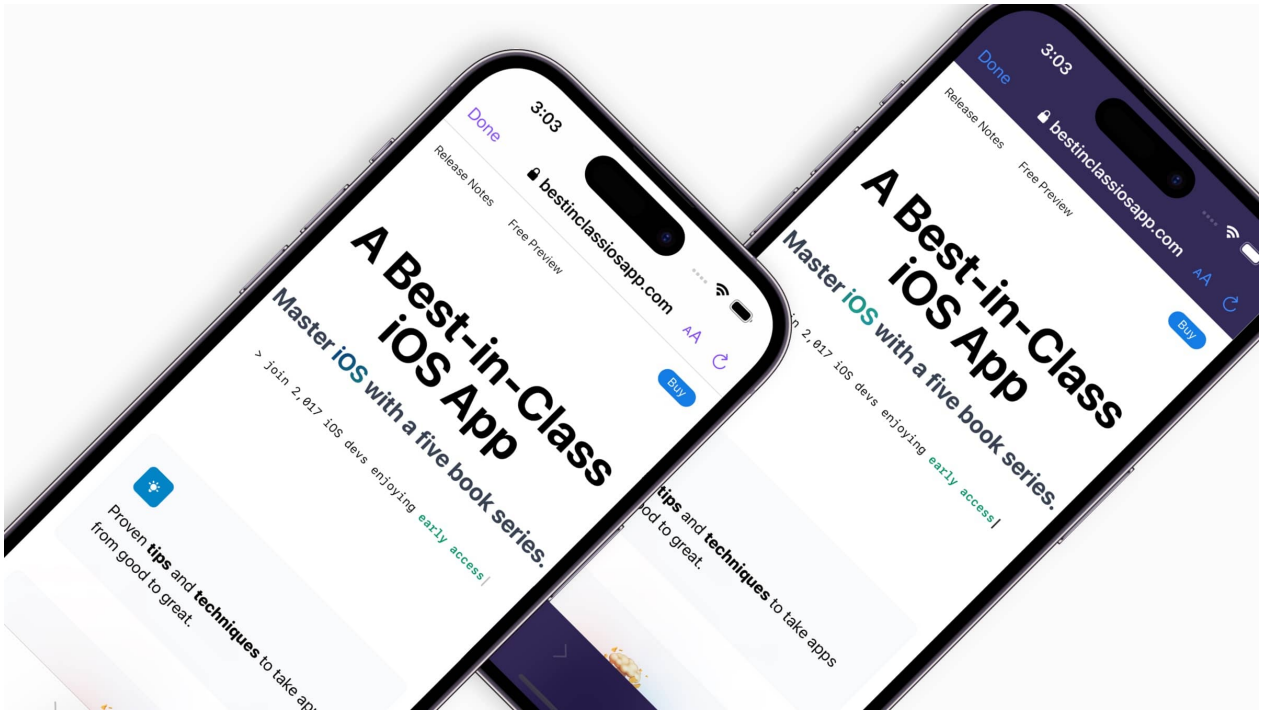
```
    func makeUIViewController(context: Context) ->
SFSafariViewController {
        let safariVC =  SFSafariViewController(url: url,
configuration: options)
        safariVC.delegate = context.coordinator
        safariVC.preferredBarTintColor = UIColor(barTint)
        safariVC.preferredControlTintColor = UIColor(controlTint)

        return safariVC
    }

    // Rest of implementation
}
```

To see how these work in practice, open up `SafariViewController-Fig2View.swift` to tweak them on the fly. You can also see how you could create a lightweight model to pass over to UIKit to assign to things like tint colors, too:

```
// Xcode -> SwiftUI -> SafariViewControllerFig2View.swift

@Observable
class SafariOptions {
    var showReader: Bool = false
    var collapseBarsOnScroll: Bool = false
    var preferredTintColor: Color = .blue
    var preferredControlTintColor: Color = .blue

    func createConfiguration() ->
SFSafariViewController.Configuration {
        let config = SFSafariViewController.Configuration()
        config.entersReaderIfAvailable = showReader
        config.barCollapsingEnabled = collapseBarsOnScroll
        return config
    }
}

struct SafariServicesFig2View: View {
    @State private var options: SafariOptions = .init()
```

```
    @State private var url: URL? = nil

    var body: some View {
        Form {
            Section("Safari Options") {
                Toggle("Use reader mode", isOn: $options.showReader)
                Toggle("Collapse Bars", isOn:
$options.collapseBarsOnScroll)
            }
            Section("Tinting") {
                ColorPicker("Bar Tint", selection:
$options.preferredTintColor)
                ColorPicker("Control Tint", selection:
$options.preferredControlTintColor)
            }
            Button("Present") {
                self.url = .init(string: "https://
www.bestinclassiosapp.com")!
            }
        }
        .fullScreenCover(item: $url) { destination in
            SafariBrowserView(url: destination,
                              options:
options.createConfiguration(),
                              barTint: options.preferredTintColor,
                              controlTint:
options.preferredControlTintColor)
            .ignoresSafeArea()
        }
    }
}
```

Just take care to choose something that's legible, it's easy to get carried away with changing tint colors. Usually, the system defaults are great – but sometimes you may want to match things up with your app or company's branding:



Preloading Content

If you know you've got a few websites you'd like to load, but they could take a bit for whatever reason – you can "prewarm" those connections. You'll get a token to store, which you can then use to cancel those if they are no longer relevant. It's quite easy to do, but only use it if you know you're loading something that's bound to take more than a few seconds – even on a solid connection:

```
let prewarmedURLs: [URL] = [.init(string: "https://
www.bestinclassiosapp.com")!]
let token = SFSafariViewController
            .prewarmConnections(to: prewarmedURLs)

self.token = token
```

So long as you keep a strong reference to that token, the framework will do its best to prewarm them. But, be aware that you could hit an HTTP timeout.

Use Safari View Controller's Delegate

The safari view controller delegate, `SFSafariViewControllerDelegate`, has several hook-in points to use. If we simply tweak the first code sample to assign to its `delegate` property, you can see them in action:

```
webVC.delegate = self

// As in extension
extension SafariServicesFig1ViewController:
SFSafariViewControllerDelegate {

    func safariViewControllerDidFinish(_ controller:
SFSafariViewController) {
        print(#function)
    }

    func safariViewController(_ controller: SFSafariViewController,
                              didCompleteInitialLoad
didLoadSuccessfully: Bool) {
        print(#function)
    }
}
```

You can see when pages load, perform some logic once they finish initial loading, tweak the share sheet by adding more `UIActivity` items and more. In SwiftUI, if we add a coordinator object – we can use those delegate actions, too. I've updated `SafariBrowserView` to do exactly that:

```
struct SafariBrowserView: UIViewControllerRepresentable {
    let url: URL

    func makeUIViewController(context: Context) ->
SFSafariViewController {
        let safariVC =  SFSafariViewController(url: url)
        safariVC.delegate = context.coordinator
```

```
        return safariVC
    }

    func updateUIViewController(_ uiViewController:
SFSafariViewController,
                                context: Context) {

    }

    func makeCoordinator() -> Coordinator {
        Coordinator(parent: self)
    }

    class Coordinator: NSObject, SFSafariViewControllerDelegate {
        let parent: SafariBrowserView

        init(parent: SafariBrowserView) {
            self.parent = parent

        }

        func safariViewController(_ controller:
SFSafariViewController,
                                  didCompleteInitialLoad
didLoadSuccessfully: Bool) {
            print(#function)
        }
    }
}
```

Of course, it's up to you to decide which parts of that delegate you need, but like any other delegate in UIKit – it can certainly be utilized in SwiftUI with coordinator objects.

Make URL Entry Easy

If you have some sort of user experience where they need to enter a URL, and then you display it within Safari view controller – be sure to make that entry as easy as pos-

sible. There are a few ways to do this, and you may have spotted them earlier in our examples.

In UIKit

- Set the text field's `.textContentType` to `URL` to provide automatic suggestions using iOS' heuristics.
- Hint that the text field is *for* URL entry with a sensible placeholder value, `urlTF.placeholder = "Enter in URL"` or similar.
- Assign the `.keyboardType` property to `URL` for easier entry.
- Make the default button on the keyboard load the URL. You can do this by supplying a `UIAction` within its `primaryAction` initializer:

```
private lazy var urlTF: UITextField = .init(frame: .zero,
                                            primaryAction: .init(handler:
{ _ in
    guard let address = URL(string: self.urlTF.text ?? "") else {
        return
    }

    let webVC = SFSafariViewController(url: address)
    self.present(webVC, animated: true)
}))
```

- Finally, set `.autocapitalizationType` to `.none` as that will avoid any odd results, such as `Http` versus `http` or `Bestinclassiosapp.com` versus `bestinclassiosapp.com`. Are these likely to break the URL loading? No. But does it just feel weird? Yes.

In SwiftUI

All the same practices apply as above. However, they are simply expressed via a SwiftUI API. Here are all the same results from UIKit written in SwiftUI:

```
TextField("", text: $link, prompt: Text("Enter in URL"))
    .textContentType(.URL)
    .keyboardType(.URL)
    .autocorrectionDisabled()
    .textInputAutocapitalization(.never)
    .onSubmit {
        guard let enteredURL = URL(string: link) else {
            print("Invalid URL.")
            return
        }

        self.destination = enteredURL
    }
```

Avoid Building Custom Browsers

There are not many good reasons to build your own browser. Tracking, really, is the primary reason you would. That's why you see Instagram, Facebook, TikTok, and other social networks offer their own.

Safari on iOS is used by the masses, and you get so much for free by using the safari view controller. For a refresher, that means things like log in and passwords are easier to manage for end users, they have their browsing history and cookies, it's a familiar user experience – the list goes on.

In fact, Apple clarifies that if you're building a custom browser, you're going in the wrong direction. Here's an except from the Human Interface Guidelines, in the "Web Browsing" section, making their stance clear in no uncertain terms:

> *Using a web view to let people briefly access a website without leaving the context of your app is fine, but Safari is the primary way people browse the web. Attempting to replicate the functionality of Safari in your app is unnecessary and discouraged.*

Outside nefarious reasons, though, Apple *does* have API to provide custom browsing experiences. For example, if you have one-off interactions that you want to occur within a web page that require you to access the web in full, use WebKit's `WKWebView`.

Don't Obscure the Safari Browser

While the Safari browser view controller could technically be used by developers to access some of the user's browsing data and sniff out browsing lifetime events – that's against App Store review guidelines. So, in short – don't do it!

Early on in its lifecycle, some developers would attempt to keep the browsing view controller either hidden or in the background to hook into certain events for tracking purposes. If Apple catches you doing this, your app will certainly get rejected – but it could also be worse, you could lose your developer account.

Three Key Takeaways

1. If you want to offer web viewing experiences, use `SFSafariViewController`.
2. There are few reasons to build a custom web browsing experience outside of tracking.
3. Take care when presenting it to not use another navigation controller, and show it modally.

Siri Shortcuts

Since the arrival of the app Workflow, the idea of automation on iOS took on a new meaning. The app allowed you to string together certain system actions to perform complex, or even dead simple, tasks. That could be appending several images together, creating a reminder and a note at the same time — there were few limits to what you could do.

In fact, the only limit was iOS itself, and the sandboxed environments in which apps are required to run. I think Apple realized this, along with Workflow's massive potential, which led to the app being acquired and eventually turned into what we know today, Siri Shortcuts. In fact, Apple has even said "The future of automation on Apple platforms is Shortcuts", so we'd better get familiar with them.

Shortcuts are contained, simple tasks that you can offer from your app. Their power really shows when you combine many of them together, have them run with an au-

tomation and the fact that iOS itself can use them in various places. Imagine offering a way to export your financial app's transactions, create them as a .pdf and email it to yourself each week without you having to do anything. Those are the kinds of things Shortcuts affords. Going forward, shortcuts that you vend from your app will be referred to as App Shortcuts.

Your App Shortcuts work across Apple's ecosystem. That means iOS, iPadOS, macOS, watchOS, HomePod, AirPods and more. Even better, they are available as soon as your app is installed (this didn't use to be the case). People can use your App Shortcuts from Siri's own suggestions, the Shortcuts app, Spotlight Search and even widgets.

In this chapter, we'll look at how to create these App Shortcuts. Before we dive in, I want to note that the world of supporting Shortcuts from a developer perspective can be a little confusing. There is SiriKit, the concept of donations, SiriKit domains, a WYSIWYG editor to create them – it's a bit much. Thankfully, there is an easier way – and that's what we will focus on; the App Intents framework.

First, let's get some domain level vernacular defined:

- An *intent* is a basically a Siri Shortcut. You can use predefined intents, or (more commonly) make your own for your app.
- The Siri Shortcuts App is where people can string together the custom Shortcuts you provide, via intents, to create custom workflows.
- Each one of your intents is referred to as an *action*. Again, these actions can be used in Spotlight, Siri, Shortcuts, widgets and more.

Predefined Intents

Now, I know I just told you that App Intents is the modern way to create custom intents. That's true, but I should note that SiriKit does provide predefined templates for Shortcuts. And, for some instances, it's the only way to create some specialized ones. If your intent would fall into any of these categories, use SiriKit:

Domain	Intent
Media	Adding items to a playlist, searching for media content.
Messaging	Sending messages via Messages or similar apps.
Payments	Sending and receiving them.
Car Commands	CarPlay other car-based actions, such as unlocking doors.
Lists and Notes	Reminders, editing Notes in the Notes app, etc.
Workouts	Kicking off workouts, pausing them, etc.
VoiP Calls	Starting VoiP calls.

But, again, for most of us — we want to create intents for our apps. So, we'll turn to the App Intents framework for that.

Further, I'm going to simplify this as much as possible because App Intents and Shortcuts is something where I truly believe less is more. If you're like me, learning them used to be intimidating. Though, if we can peel back the layers a bit and form a foundation, then you'll be set up to create both simple and advanced shortcuts.

How it Works

Providing shortcuts all begins with adopting the `AppIntent` protocol. When you provide a Struct instance that conforms to it, you have three main responsibilities:

1. Provide a title and description for the intent.
2. Declare any parameters the intent needs to run.
3. And, most importantly, implement the `perform` function, where you fire off the logic to perform your shortcut.

Before we go into more detail, here's a complete implementation of a shortcut:

```
struct SayPhraseIntent: AppIntent {
    // #1
    static var title: LocalizedStringResource = "Say a phrase."
    static var description = IntentDescription("Repeats a phrase you
type in.")

    // #2
    @Parameter(title: "Phrase")
    var phrase: String?

    init() {
self.phrase = nil
    }

    init(phrase: String) {
        self.phrase = phrase
    }

    // #3
    func perform() async throws -> some ProvidesDialog {
        guard let providedPhrase = phrase else {
            throw $phrase.needsValueError("What phrase do you want
to say?")
        }

        return .result(dialog: IntentDialog(stringLiteral:
providedPhrase))
    }
}
```

Our `SayPhraseIntent` is now a fully featured App Shortcut. When it's run, it'll ask for a `Phrase` value, and then speak it out. You can see each component of steps 1-3 that we listed right before the code sample implemented (I've attached a corresponding comment for each step number in the code, too). The great thing about this API is that it's written much like it's used by consumers – you can easily scan the code to understand what all it does.

For step number two, any parameter we provide must use the `@Parameter` property wrapper. Thankfully, this wrapper supports virtually any type you'd need, including custom models from your app:

- Bool, Int, Double, String, Duration, Date, Decimal, Measurement, and URL.
- Array and Set support.
- `IntentPerson` and `IntentFile`.
- Enumerations that conform to `AppEnum`, but they'll need known, static values at build time.
- For your custom types, conform them to `AppEntity` to use as a parameter.

Now, again, before iOS 16 – to get such a simple result required a lot more from you. You had to handle parameter resolution much more intentionally, create an intents definition file and more. While the App Intents framework is a big one, it is designed with progressive disclosure in mind. It can become much more complex if you require.

You may be wondering how the intent becomes available to the rest of the system. That's where `AppShortcutsProvider` comes into play. We need to make a Struct conform to that – and supply our intents to it. We'll add that right above the custom intent we made:

```
struct IntentProvider: AppShortcutsProvider {
    static var appShortcuts: [AppShortcut] {
```

```
        AppShortcut(intent: SayPhraseIntent(),
                            phrases: ["Say a Phrase in \
(.applicationName)."],
                            shortTitle: "Speak Phrase",
                            systemImageName: "speaker.wave.2.fill")
    }
}
```

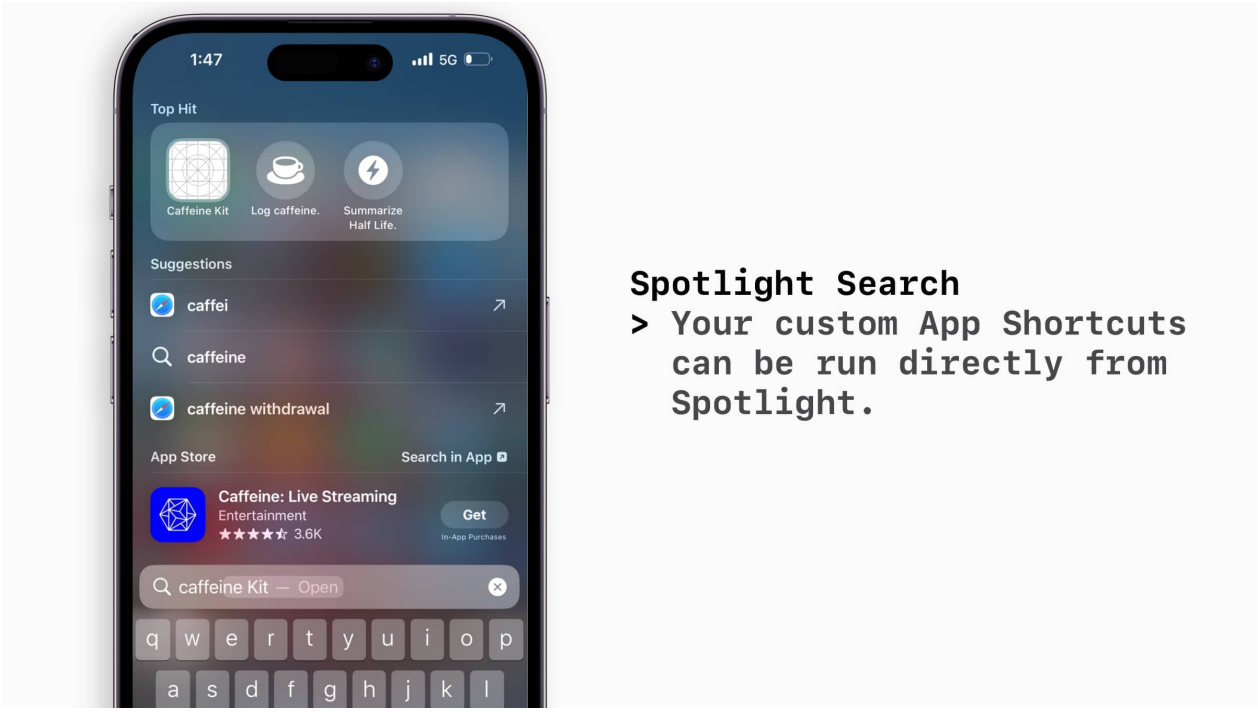
That's all it takes. It may seem a bit weird, but you don't even need to initialize this Struct – the framework will simply see that you have something conforming to the protocol and use the shortcuts you've provided there throughout the system. But it's impressive when you consider all of the places you can use your App Shortcut now:

- Menu Bar
- Share Extensions
- Terminal
- AppleScript
- Home Screen
- Suggestions
- Lock Screen
- Shortcuts Widgets
- Quick Actions
- Voice
- Apple Watch
- HomePod
- Automations
- Shortcuts App
- Keyboard
- Spotlight

- The Action Button on supported devices

By supplying our `appShortcuts` implementation (which is *not* an Array, it's a result builder which supplies `AppShortcut` instances), we're now ready to use the intent all over the Apple ecosystem.

For example, here's a few custom App Shortcuts from a caffeine tracking app I've been working on. At the top, you can see two App Shortcuts and the leading item simply opens the app. This is the kind of discovery you get without a lot of work:



If you run the demo app – you'd see the same result for App Shortcut we've created. Either type in its phrases or the app's name (Code Samples, in our case) to see them in spotlight.

Also, notice the SF Symbol iconography. That comes into play here, and it's why it's part of the AppIntent initializer. In our sample intent, I used a speaker:

```
AppShortcut(intent: SayPhraseIntent(),
            phrases: ["Say a Phrase in \(.applicationName)."],
            shortTitle: "Speak Phrase",
            systemImageName: "speaker.wave.2.fill")
```

So, with that little bit of code, you're ready to start making App Shortcuts. To recap our responsibilities, we've got to:

1. Create a Struct for each App Shortcut we want to provide, and they each will adopt the `AppIntent` protocol.
2. At a minimum, we need to implement the `perform()` function to fire off our logic.
3. Then, we need just *one* Struct that adopts `AppShortcutsProvider` – the system will automatically see it and vend your shortcuts provided there.

Oh, and perhaps best of all – these are easily debuggable. You can simply use LLDB within Xcode, which wasn't always the case before App Intents. Simply open the Shortcuts app on the simulator, add your custom intent to a new Shortcut and run it. You'll be able to breakpoint right on the `perform()` function.

Tips

Understanding Return Types for Custom Intents

When you conform to `AppIntent` and implement `perform()`, you're required to return something of type `some IntentResult` – and that can kind of get confusing, but it's certainly flexible. The type of result you need to return will depend on the actions you want the intent to perform.

For example, let's say our App Shortcut adds two numbers together:

```
public func perform() async throws -> some IntentResult &
ReturnsValue<Int> & ProvidesDialog {
    let total = numberOne + numberTwo
    return .result(value: total, dialog: "You added \(numberOne) and
\(numberTwo) together, the sum is \(total)")
}
```

Now, we're returning `some IntentResult & ReturnsValue<Int> & ProvidesDialog`. That's a lot to glance at, but what it means is that we have a value to return of an Int (`ReturnsValue<Int>`) and it also speaks out or provides a text result (`ProvidesDialog`).

If you find yourself confused, simply look at the documentation for `IntentResult` and glance at the types available. I find that's the easiest way to pinpoint what you need. To wit – here's that previous example fully implemented so you can compare it with our other, more simple, app intent:

```
struct AddNumbersIntent: AppIntent {
    static var title: LocalizedStringResource = "Add Two Numbers"
    static var description = IntentDescription("Returns the sum of
two numbers you provide.")

    @Parameter(title: "NumberOne")
    var numberOne: Int
    @Parameter(title: "NumberTwo")
    var numberTwo: Int

    init() {
        self.numberOne = 0
        self.numberTwo = 0
    }

    init(numberOne: Int, numberTwo: Int) {
        self.numberOne = numberOne
        self.numberTwo = numberTwo
    }
```

```
    public func perform() async throws -> some IntentResult &
ReturnsValue<Int> & ProvidesDialog {
        let total = numberOne + numberTwo
        return .result(value: total, dialog: "You added \(numberOne)
and \(numberTwo) together, the sum is \(total)")
    }
}
```

One confusing aspect of this is that if you omitted the `IntentResult` from the return type, it would still compile just fine:

```
public func perform() async throws -> some ReturnsValue<Int> &
ProvidesDialog
```

But, it would never run the `perform()` function. This is hard to figure out at first, but be sure to include it if you're returning something with multiple different `IntentResult` types as we are here:

```
public func perform() async throws -> some IntentResult &
ReturnsValue<Int> & ProvidesDialog
```

Also, I do want to call out something important here: **You need to provide a default initializer and one for the parameters that you provide**. If you don't, it may not work at all in different targets, or simply never run. I think part of the reason is that App Intents require default values at build time.

It can be frustrating to debug, but it's an easy fix. The Swift compiler and the framework simply can't synthesize these on our behalf quite yet, so we need to provide them.

Performing Parameter Resolution

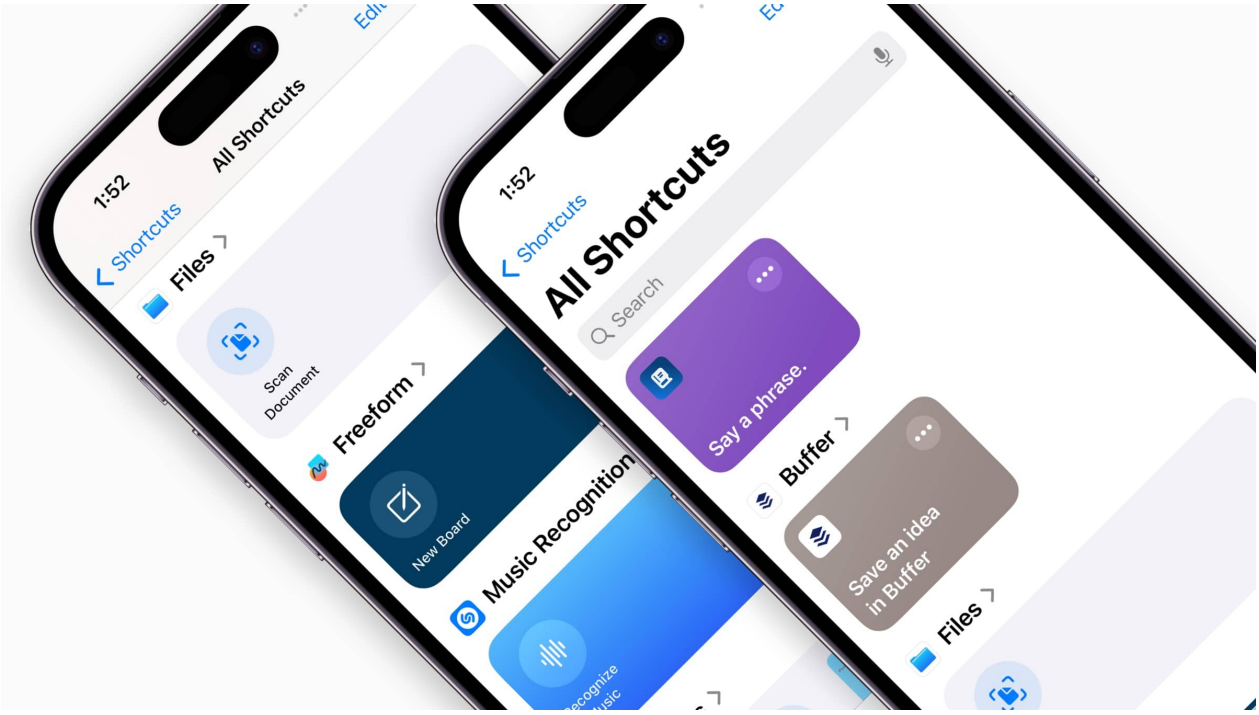
In some cases, you might need a value that's optional depending on your domain logic. It's not hard to prompt the user to provide one if it's absent, and you might've noticed us accounting for that in our `SayPhraseIntent` earlier:

```
func perform() async throws -> some ProvidesDialog {
    guard let providedPhrase = phrase else {
        throw $phrase.needsValueError("What phrase do you want to
say?")
    }

    return .result(dialog: IntentDialog(stringLiteral:
providedPhrase))
}
```

You can do that to enforce a value to be supplied, or even use it for disambiguation between existing values.

Supply a Color for Spotlight and More

In the Shortcuts app, you'll see that each Shortcut has a tile. Each one of those has a nice, gradient background and an icon:



You can influence which color is used for your custom intents, too. To do that, we need to look back at our instance conforming to `AppShortcutsProvider`. There, simply supply a color for `shortcutTileColor`:

```
static var shortcutTileColor: ShortcutTileColor {
    return .lightBlue
}
```

I should note that this is of type `ShortcutTileColor` – which means you do need to use something that Apple supplies to you. You can't use a custom color from your Asset Catalog, for example. There are a lot of them, though, so you should be able to get close to your branding:

```
public enum ShortcutTileColor {
    case red
    case orange
    case tangerine
    case yellow
```

```
    case lime
    case teal
    case lightBlue
    case blue
    case navy
    case grape
    case purple
    case pink
    case grayBlue
    case grayGreen
    case grayBrown
}
```

Architecting for Siri Shortcuts

You should really consider putting your model layer, API utilities or domain specific data fetching classes and similar code in their own framework. Why? Simply because these will be used across several targets. And that's even more true now than ever before because of widgets.

Widgets support interactivity, which means that you can have a button or toggle in them. The logic they fire are all powered by your custom intents. So, for example, we could have our "Say a Phrase" intent run from a widget button tap – and iOS will ask for the phrase we want and then speak it out loud.

If you've got those things in a place where they can be shared – you'll be in a good place to implement shortcuts, app specific functionality and more without a whole lot of fuss. You won't need shims, copy and pasted models or similar fallbacks.

Use App Intents for Common Actions

Since App Intents can be used for widget interactivity, Siri Shortcuts and more – I think it's wise to use those as the single point of entry for common logic in your app. Would

you rather maintain a code path for a widget that logs caffeine in a custom app intent *and* another path in your main app?

Following a bit of what I just talked about above – centralize things to make your life easier. App Intents are only becoming more and more prominent at an API level in the Apple ecosystem, and while I'm not dogmatic about architecture, I do tend to lean towards the DRY principles (don't repeat yourself).

As such, since our demo `SayPhraseIntent` might be used in our main app, a widget and Siri – lean on the intent to do all of their work. Again, `Button` accepts an intent in its initializer (assuming you've imported the App Intents framework), so code can look very similar across targets. Here, our intent is running in the main app, the same as it will be used later in the Widget chapter:

```
// Xcode -> SwiftUI -> ShortcutsFig1View.swift

struct ShortcutsFig1View: View {
    @State private var phrase: String = ""

    var body: some View {
        Form {
            TextField("Enter a phrase...", text: $phrase)
            Button(intent: SayPhraseIntent(phrase: phrase), label: {
                Text("Speak Phrase")
            })
        }
        .navigationTitle("App Shortcut Example")
        .navigationBarTitleDisplayMode(.inline)
    }
}
```

As we'll see later, we literally just copy and paste that button for later use in a widget.

Designing Shortcuts

Not everything should be an App Shortcut. In fact, you're limited in the number you can provide – and that's 10. You want to look for actions in your app that are meaningful and quick to perform. Apple recommends that you should benchmark each potential App Shortcut by these three criteria. Ask yourself if:

1. They are self contained, and can operate more or less in isolation.
2. They can be achieved without opening your app.
3. And, they are efficient, easy and uncomplicated.

Keep in mind that the API is certainly part of the design of a shortcut, too. You can reuse the same intent in so many places – such as Live Activity code. So, the more "single responsibility" standards an intent has, the better.

Control App Shortcuts Ordering

The way your custom App Shortcuts show in Spotlight and the Siri Shortcuts app is under your direct influence. Whichever order you provide them in the `appShortcuts` builder implementation is what's mirrored across the system. Again, from earlier:

```
static var appShortcuts: [AppShortcut] {
        return [AppShortcut(intent: SayPhraseIntent(),
                            phrases: ["Say a Phrase in \
(.applicationName)."],
                            shortTitle: "Speak Phrase",
                            systemImageName: "speaker.wave.2.fill")]
}
```

So, if we were to add a new App Shortcut to an update – it's probably a good idea to return that first before your other existing shortcuts. Or, if your existing ones are more relevant still, leave those at the front.

Handling App Dependencies

In many cases, you'll have some object in your app that you might need to use in an intent. In this case, you can use a property wrapper to let the App Intents framework know that there is an internal dependency. This is done via the `AppDependencyManager.shared` object and the `@Dependency` property wrapper.

The flow works like this: when your app is initialized, register dependencies that your app intents might need. A common use case would be some sort of data access layer:

```
init() {
    AppDependencyManager.shared
                        .add(dependency: myDataManager)
}
```

Then later, in your app intents – you can access it via a property wrapper:

```
struct MyIntent: AppIntent {
    @Dependency
    private var dao: DataAccessObject

    // Rest of intent code
}
```

This method works for both `AppIntent` or `EntityQuery` protocol conformers. Note that if you're using your intents across widgets and your app target, you'll likely need to register these dependencies in your `WidgetBundle` too.

Exposing Shortcuts

You can use a `SiriTipView` to hint to users that your app has custom App Shortcuts on offer:

```
SiriTipView(intent: SayPhraseIntent(),
```

```
            isVisible: $hasSeenTip)
```

That'll give you an adaptable, nice view that displays and manages dismissing all on its own too. That can also be programmatically controlled via the binding. In UIKit, it's virtually the same, except you use `SiriTipUIView`. You can also have it display specifically in light and dark mode, though its automatic style will adapt it on your behalf.

Further, to link to all of your app's actions in the Shortcuts app – use `ShortcutsLink()` in SwiftUI and `ShortcutsUIButton` in UIKit. Another neat little feature is that users can open up your app and say, "Siri, what can I do here?" – and it'll list out all of your provided App Shortcuts.

Marking Intents as Undiscoverable

In the age of interactive app intents, there will be times when your intent doesn't make sense to surface to the system. For example, if your widget simply toggles quotes via some `ToggleQuoteIntent` – that intent wouldn't do anything else. It has no use outside of that one widget.

In these cases, mark them as undiscoverable by assigning to the `isDiscoverable` static boolean:

```
struct ToggleQuoteIntent: AppIntent {
    static var isDiscoverable: Bool = false
    // Rest of app intent code
}
```

Now, system intelligence APIs won't try and surface that intent in places like Core Spotlight, short suggestions and more.

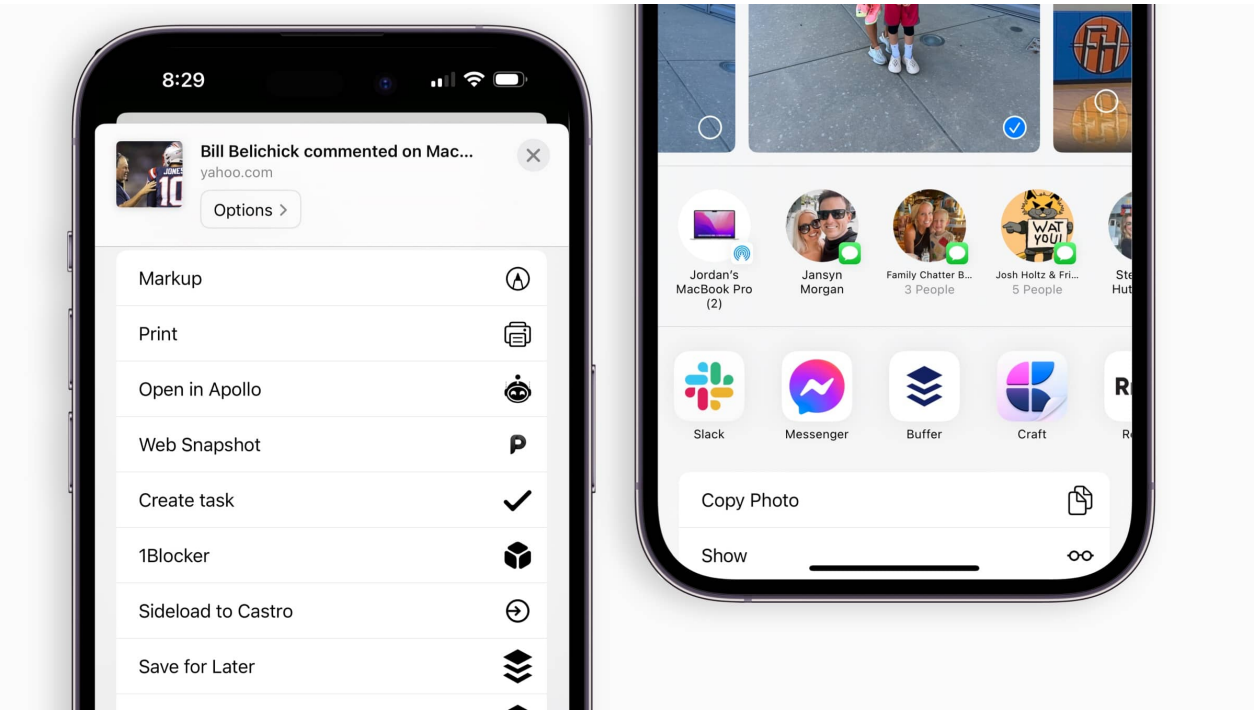
Three Key Takeaways

1. App Intents is the modern way to implement shortcuts on iOS.
2. Offer useful, quick interactions to make your app's utility extend across the system.
3. Consider architecting main flows as App Intents to share them across widgets, shortcuts and your own app.

Share and Action Extensions

Share and action extensions allow you to expand your app's fingerprint beyond its app target, and they've been around since iOS 8. You can use these hook-in points to do all sorts of interesting things across iOS, and while they are commonplace now — nascent developers may be unaware that these extension targets are how apps offer critical services. Whether you are sharing content to your app, or manipulating existing photos or videos — extensions allow you to do so outside your app.

For example, open up the share sheet in almost any context, and you're bound to see several action and share extensions available. In the following image, you're seeing actions extensions on the left side, and share extensions on the right:



Today, though, there isn't enough modern documentation over how to develop them. So in this chapter, we'll look at how they work and when to use one or the other. These extensions are important, and users will likely expect them to be available. Even if they aren't intuitively aware of what an "action" or "share" extension is, or what those words even mean, they probably *do* know about share sheets and all the things you can do in them.

First things first – these are new *targets* which are added to your primary app target. While they are many targets you can add to your app, and extensions for that matter – we're focusing on two of them: Share and Action extensions.

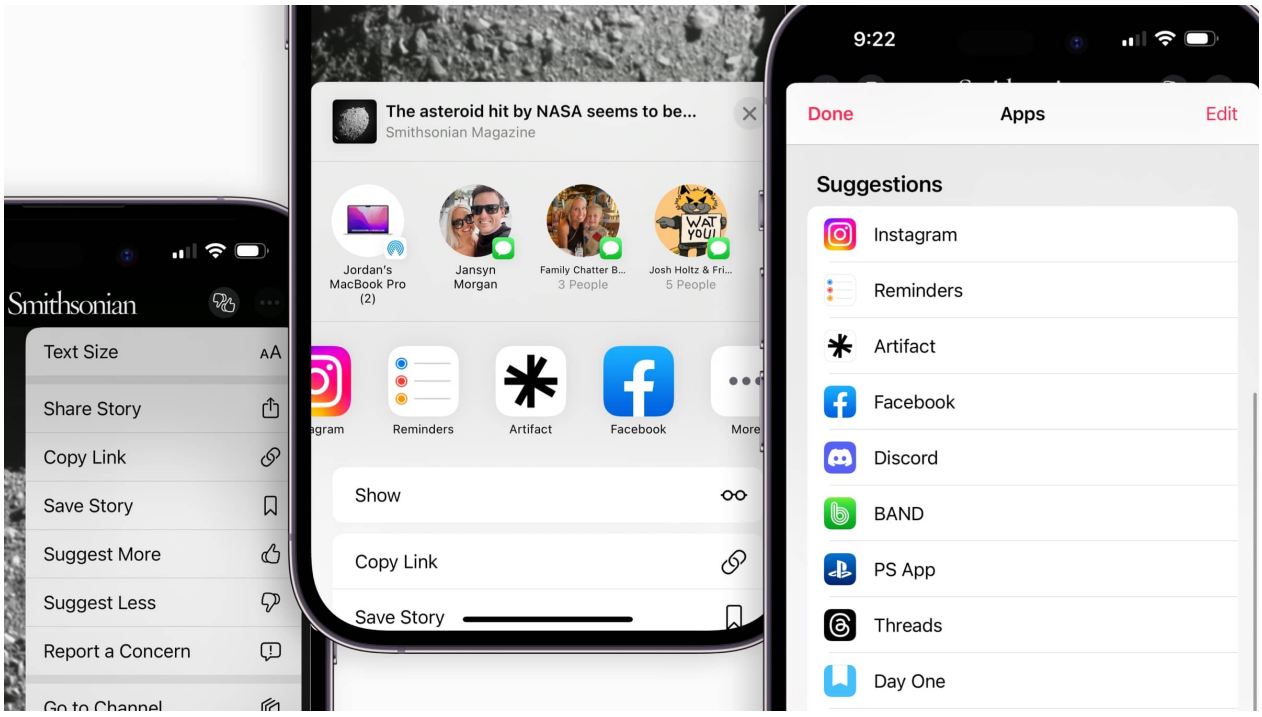
Share Extension

True to its name, a share extension is meant share information or data with another app, service or send that data in some meaningful way. So, for social media apps, this means sharing a photo to Instagram, or adding text to a feed or other similar ac-

tions. These extensions are more prominent than actions extensions, and you'll see your app's icon being used for them.

I do want to note, though, that nearly most these "share" extensions don't follow that definition. At least, not exactly. And, I think this is a case where it's okay to be liberal with Apple's definition of what a share extension should do. In fact, most apps don't offer any action extensions – only a share extension that acts *more* like an action extension should by its official definition.

Here's some share extensions I currently have on my iPhone. In Apple News, I've selected "Share Story" which presents the share sheet. In the middle row, you'll see all of the share extensions offered by third party apps. If I tap "Edit", I can add or remove other share extensions that are available:



Some of those follow the share extension definition to the letter. Facebook and Instagram post to social feeds. This was the original intended use case for share extensions

– they were social in nature. In fact, the Xcode template even imports the Social framework.

Though, look a little further and you'll see others that "share" in their own way. Day One "shares" the story to a new or existing journal entry. I'm glad it does, though, because share extensions are more prominent than action extensions – and Apple shows them up higher in the view hierarchy. I use them all the time.

So, if you need to get some data from somewhere else into your app or to a service – a share extension is a great choice.

Action Extension

Action extension work similarly – though they are meant to augment or otherwise transform existing information that's supplied to it. Then, they can return it to your app. Although, this is what they are by definition – in reality, many developers also use them as a "bonus" share extension in several ways. Both of them can do nearly the same things at a system level.

Keep in mind that these extensions appear with grayscale icons you provide, and they appear lower down in the share sheet. But, you can do effectively all of the same things with one that you can do with a share extension.

How They Work

These extensions are launched in their own separate process outside of your app. This is a controlled, somewhat constrained environment you'll be working in – so do what your extension requires and not much more. For example, memory comes at a premium in these extensions – something we'll cover in the tips section.

From a 10,000 foot view, though, extensions are fairly simple:

- They are launched from a user from the share sheet.
- You operate on some data.
- Then, it's closed and the user returns to what they were doing.

While most APIs are available in this context, there are some things you can't use. For example, `UIApplication.sharedApplication` won't be available – nor will microphone usage. Long running tasks also aren't suited to either extension type. Or, things like HealthKit or EventKit UI won't import. If you see `NS_EXTENSION_UNAVAILABLE` in the header files, it's unavailable for extension use.

Specifying Data Inputs

But, outside of that – you're free to implement any flow you'd like. However, starting with these extensions begins in its corresponding property list. Specifically, the `NSExtension` key. Here, you need to declare what kind of data you'll accept in it, along with other things like a default view controller.

As for the data it accepts, though, by default, it'll take in, well – anything:

```
<key>NSExtensionActivationRule</key>
<string>TRUEPREDICATE</string>
```

The `NSExtensionActivationRule` is where you specify what kind of data you intend to work with. In the sample project – we've indicated we're interested in working with a single photo for the `BICActionExtension`:

```
<key>NSExtensionActivationRule</key>
<dict>
    <key>NSExtensionActivationSupportsImageWithMaxCount</key>
    <integer>1</integer>
</dict>
```

You can get complex or simple with these rules. Uniform type identifiers come into play here – so you can specify that you only want to have your extension show for text, web pages or even specific types of images. In fact, iOS can even create a predicate for you so that you can support very niche requirements. Here's an example from Apple which shows a subquery, with another nested subquery, filtering down to a domain-specific UTI type, and getting only one of them:

```
SUBQUERY (
    extensionItems,
    $extensionItem,
    SUBQUERY (
        $extensionItem.attachments,
        $attachment,
        ANY $attachment.registeredTypeIdentifiers UTI-CONFORMS-TO
"org.appextension.action-one" ||
        ANY $attachment.registeredTypeIdentifiers UTI-CONFORMS-TO
"org.appextension.action-two"
    ).@count == $extensionItem.attachments.@count
).@count == 1
```

During development, you can switch back to default activation rule to let anything through. But, again, be sure to get this switched back before you ship. If you don't, you won't pass review. To help you remember, Xcode will provide a warning similar to this if you've got an activation rule in use that lets anything pass:

> *Embedded binary's NSExtensionActivationRule is TRUEPREDICATE. Before you submit your containing app to the App Store, be sure to replace all uses of TRUEPREDICATE with specific predicate statements or NSExtensionActivationRule keys. If any extensions in your containing app include TRUEPREDICATE, the app will be rejected*

So, once you specify what kind of data you want, where does it end up?

Operating on Data

For that, look to `NSExtensionContext`. This class will house everything you need to complete the entire lifecycle of an extension. It's responsible for signifying to the system when you are done, information about the extension's process and most importantly – it'll have the data you requested.

Any `UIViewController` has access to this context. Further, this is also why extensions are centered around them:

```
extension UIViewController : NSExtensionRequestHandling {
    @available(iOS 8.0, *)
    open var extensionContext: NSExtensionContext? { get }
}
```

Its `inputItems` array will house any items that match the rules setup in the prior step. Each `NSExtensionItem` instance will itself have an `items` array – and this is how you operate on things you receive in an extension. Again, let's look at our demo action extension and how we retrieve an image:

```
// Xcode -> BICActionExtension -> ActionViewController.swift

private let utType: UTType = .image
override func viewDidLoad() {
    super.viewDidLoad()

    guard let firstItem = extensionContext?.inputItems.first as?
NSExtensionItem,
          let firstItemProvider = firstItem.attachments?.first else
{
        cancelExtension(error: .noPhotosFound)
        return
    }
```

```
    guard
firstItemProvider.hasItemConformingToTypeIdentifier(utType.identifie
r) else {
        cancelExtension(error: .noPhotosFound)
        return
    }


    firstItemProvider.loadItem(forTypeIdentifier:
UTType.image.identifier,
                               options: nil,
                               completionHandler: { (imageURL,
error) in
        DispatchQueue.main.async {
            if error != nil {
                self.cancelExtension(error: .genericError(error! as
NSError))
                return
            }

            if let assetURL = imageURL as? URL,
               let fullImage =  UIImage(data: try! Data(contentsOf:
assetURL)) {
                print(fullImage)
            }
        }
    })
}
```

In `viewDidLoad:`, we go through the `extensionContext` property and grab the first `NSExtensionItem`. From there, it's about digging in through its `NSItemProvider` instances found in its `item` array. This is where uniform type identifiers come into play – when we find a match, we can load it using `NSItemProvider` APIs.

If you need a good breakdown on how these work, check out the "Drag and Drop" and "AirDrop" chapters for conversations over Transferable and how to work with item providers.

Now, *here* is where your custom app logic comes into play. You've identified what kind of data to work with, loaded an extension, and retrieved all of the data. You're free to carry out the purpose of your extension at this point.

Finishing an Extension Process

There is one final, important piece to the puzzle though. Once we've either finished our process, an error occurred *or* we simply need to cancel, we've got to let our `NSExtensionContext` know. You see this in our demo:

```
// Xcode -> BICActionExtension -> ActionViewController.swift

private func finishExtension() {
    extensionContext?.completeRequest(returningItems: [])
}

private func cancelExtension(error: BICExtensionError) {
    extensionContext?.cancelRequest(withError: error)
}
```

I find it's easiest to use a custom error to handle anything unexpected gracefully:

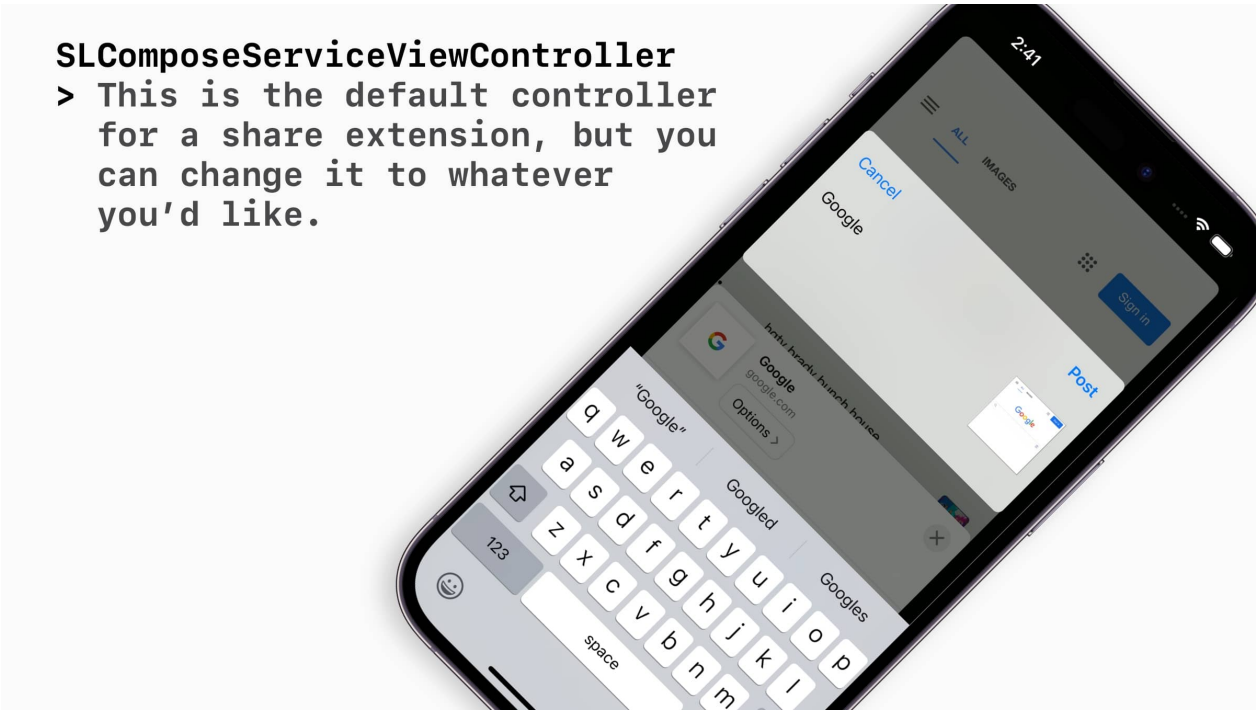
```
enum BICExtensionError: Error {
    case noPhotosFound
    case genericError(NSError)
    case cancelled
}
```

When you're done with the extension process, simply call either a cancellation with an optional error, or a completion function. Then, iOS will dismiss your extension – completing the extension lifecycle.

Share versus Actions Extensions

You may be wondering if there are many differences (aside from their intended use) between an action and a share extension. Thankfully, there really isn't much – just a few things. In fact, they're so short I can list them out:

1. The template for either of these will name the default view controller either `ActionViewController` or `ShareViewController`.
2. With an action extension, you *return* the items you operated on back to the host app: `extensionContext?.completeRequest(returningItems: self.extensionContext.inputItems)`. For `returningItems`, you can return something new, or simply the items you received from the extension context. In a share extension, you can just return an empty array.
3. Finally, with a `ShareViewController` – it'll inherit from `SLComposeServiceViewController` by default. While you *can* use it, I find in many cases – that's not what you want. Its design it a little dated, and it's not very flexible. In most cases, you'll simply want to change it to inherit from `UIViewController` so you can style it how you need. If you don't – you're boxed into using this user interface:



Tips

Avoiding Storyboards in an Extension

If you wish to use programmatic user interfaces and avoid storyboards in extensions, that's certainly possible. I haven't included a tip on how *to* use storyboards if like, because that's how the template comes configured.

First, remove the default `NSExtensionMainStoryboard` key. Replace it with `NSExtensionPrincipalClass` , and with a String value – use the class name of the view controller you'd like to use. If you're using Objective-C, the name is enough. If you're using Swift – prepend it with `$(PRODUCT_MODULE_NAME).` as we've done in the sample app (i.e., `$(PRODUCT_MODULE_NAME).ShareViewController`).

So you can see how both of these work, in the demo project - I've left the action extension with the default setup. It uses a storyboard. In the share extension, I've changed it to use a view controller without a storyboard.

Using SwiftUI in Extensions

Following the above – let's go a step further. What if we wanted to use SwiftUI in one of these extensions? That's certainly possible via hosting controllers. In our share extension, we get a photo like the action extension does – and then we display its size information.

The flow is the same as the action extension, except once we retrieve the image – we then turn around and show a SwiftUI view:

```
// Xcode -> BICShareExtension.swift

// Inside viewDidLoad:

let hostingController  = UIHostingController(rootView:
ImageSizeView(image: fullImage))
self.addChild(hostingController)
self.view.addSubview(hostingController.view)
hostingController.didMove(toParent: self)

hostingController.view.translatesAutoresizingMaskIntoConstraints =
false
hostingController.view.backgroundColor = UIColor.clear

NSLayoutConstraint.activate([
    hostingController.view.heightAnchor.constraint(equalTo:
self.view.heightAnchor),
    hostingController.view.widthAnchor.constraint(equalTo:
self.view.widthAnchor),
    hostingController.view.centerXAnchor.constraint(equalTo:
self.view.centerXAnchor),
```

```
    hostingController.view.centerYAnchor.constraint(equalTo:
self.view.centerYAnchor)
])
```

Or, if you don't want to add it as a subview – simply present it within `viewDidLoad`:

```
let hostingController = UIHostingController(rootView: MyView())
hostingController = .currentContext

self.present(hostingController,
             animated: true,
             completion: nil)
```

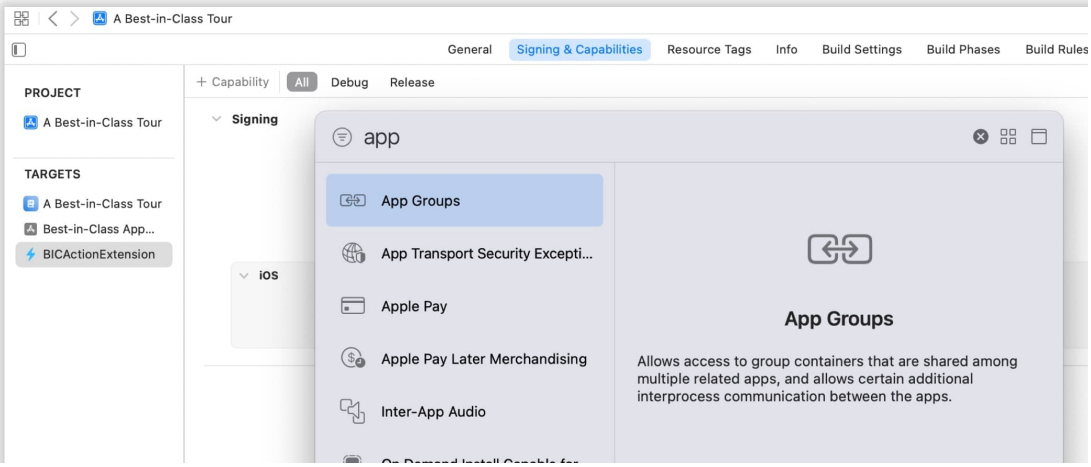
When you go with this approach, there are a few things to note:

- You'll need to either pass the extension context, or use Combine or some other mechanism to signal when your extension is done, cancelled or otherwise finished.
- You can present the SwiftUI view right away, or process the items first and then show it as we've done here. However, if you do the latter – it's good to show an activity indicator or something similar if processing items takes a bit.

Use Shared Defaults and App Groups

Since these extensions are new targets you add to your app, it pays to enable things like App Groups to access things such as shared user defaults:

App Groups
> User defaults, and other stores, can be used in extensions by leveraging App Groups.



This applies not only to share and action extensions, but new targets you add to your app in general. For example, WidgetKit and widget extensions. Whether or not you cache some lightweight identifiers or app-specific data, having it easily available in your targets is usually a requirement.

It's rare for these extensions to act in isolation from your primary app – they typically will rely on some sort of data or models associated with your primary app target. Tangentially related – I do this in conjunction with using separate Swift Packages or frameworks. Then, you can pull in all of your model layers or similar, and share data as needed.

Share Extension Limits

Apple only allows one share extension per app target, so make sure you're providing something that offers a critical workflow. Basically, don't waste it. Really consider what

would be the best experience. This limitation doesn't exist for action extensions, so you've got to make those share extensions count.

Further, it's *possible* to pivot it later to do something else. However, I'd try your best to avoid that as users will get used to the current flow that you have. If you need to augment a share extension's core functionality, consider an "adding to" approach as opposed to a "completely changing it" direction.

For example, let's say you were developing an app that saves photos to a service that you can tag for organizational purposes. Let's say your extension for such an app simply gave you the option to add a photo and a tag since its inception.

Years go by, and your app now offers the ability to post those photos to some social feed. You may be tempted to change your share extension altogether to support this. But then, you confuse users by switching how the extension works out from under them.

Your choice is to "add to" or "completely change" it. In these cases, it's better to add the ability to optionally post them to a feed, and keep the extension working as it always has. Users rely on these flows and develop a muscle memory from using them often.

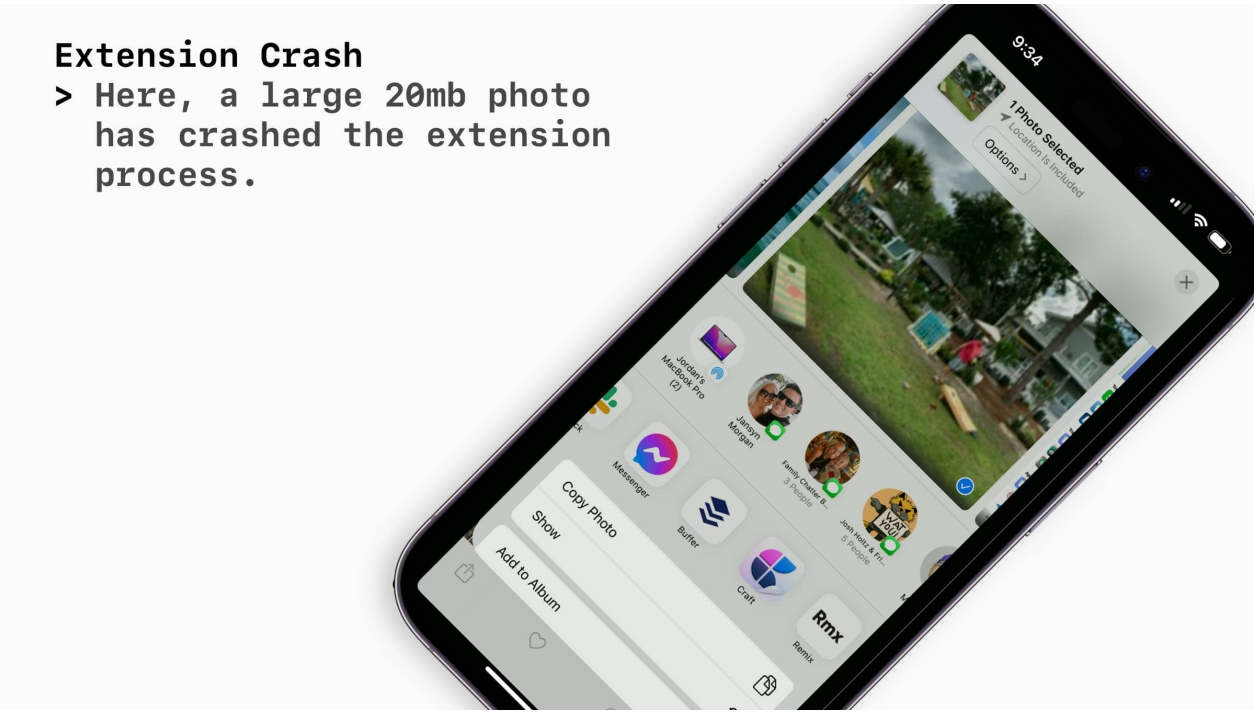
Be Aware of Memory Limits

While, as of now, the memory limits aren't officially documented for share extensions – just be aware of the fact that it's tight. I've seen 10 megabytes to 15 megabytes thrown around as the limit, but I've seen crashes occur anywhere in-between those, give or take a few megabytes on either end.

To demonstrate, try to use any share extension from a third party developer which accepts images. Now, go to your camera roll and try to use that extension with the

largest photo you can find. All too often, that'll crash the extension immediately because the memory allocation is simply too high for the extension process.

Here, I've commented out some optimization code I wrote for an extension I worked on to purposefully show the extension crashing:



In such cases, be sure to optimize, downsample or otherwise change any large assets you may get from an extension context to be lightweight and manageable in an extension context.

Hint at Extensions

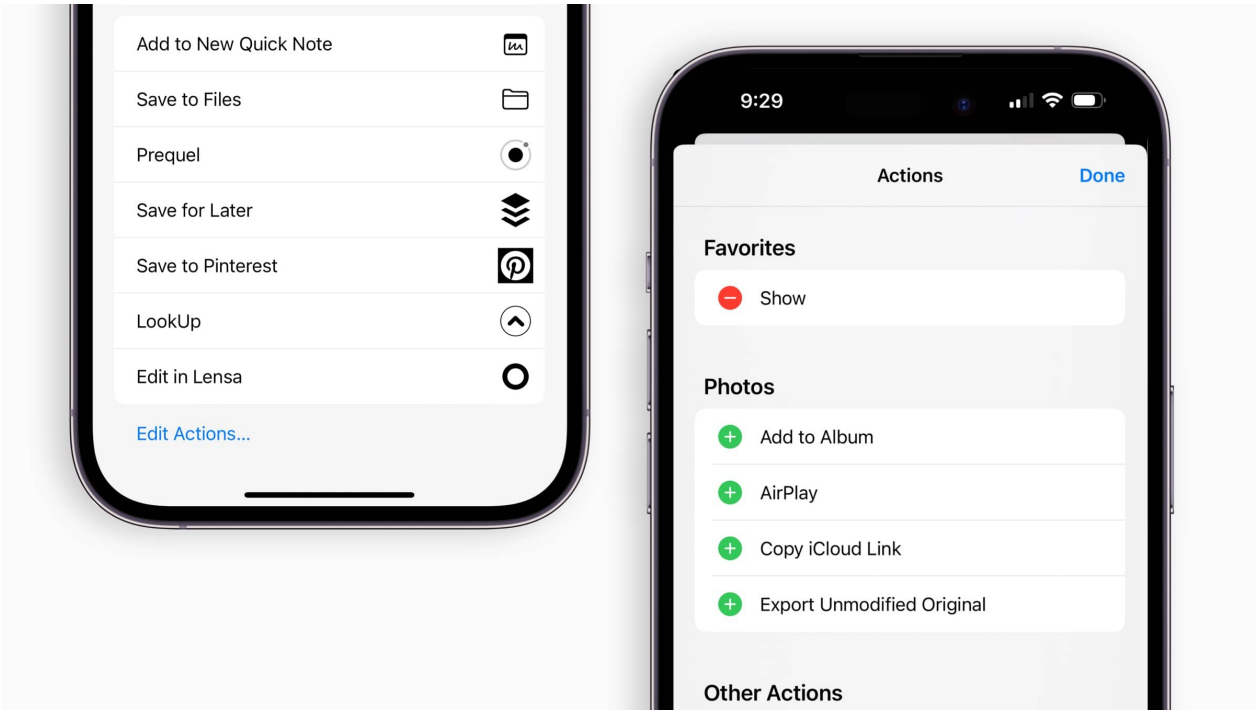
Extensions aren't always obvious for some users. While the people reading this book or the tech crowd will know to look for them, the day-to-day user probably knows they exist but don't know how to configure them for your app.

This is a case where you might use TipKit or some other modal view to let users know you have one. It's not a bad idea to include a quick, to-the-point set of instructions to install them either. You see this a lot with widgets, too.

For a share extension, I usually include something like this in terms of the "how-to" copy, along with some imagery in a horizontally scrolled page control:

1. Tap the share icon.
2. Scroll to "Edit".
3. Choose "App Name".

For action extensions, it's similar – but I'd argue those are even less prominent. For them, users have to scroll all the way down and tap "Edit Actions…":



Three Key Takeaways

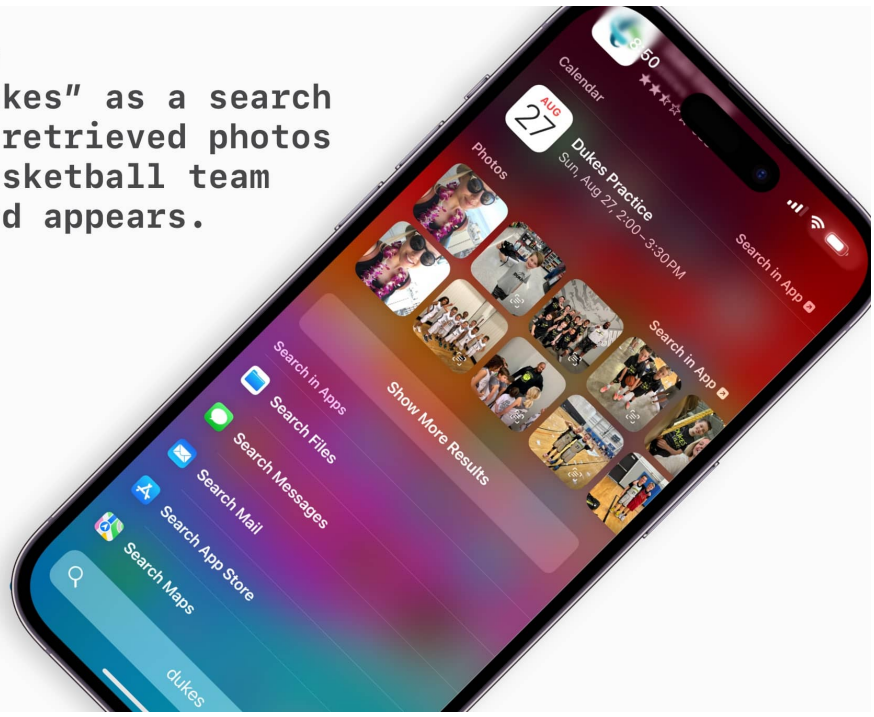
1. Action and Share extensions allow you to operate on data or information outside of your app's target.
2. Use them to provide quick, meaningful flows for your app.
3. Be aware of limitations of each of them, and provide a high-value task for your one share extension.

Spotlight Search

Spotlight is the primary way people search content in iOS. Each year, it seems to only become more and more capable. For example, you can search for text that may appear in an image in your camera roll directly from Spotlight. In the image below, I searched "Dukes", the name of my son's basketball team.

And, sure enough, all of the photos where that text appears were shown:

Spotlight Search
> I typed in "Dukes" as a search query, and it retrieved photos of my son's basketball team where that word appears.



This chapter is all about how to integrate into Spotlight search. We can supply our app's own data so that it can be searched for without having to open your app. In fact, you can index data with Apple's servers so that people can search for things in your app even when it's *not installed*. While our focus is Spotlight, these APIs can also aid in discoverability in other places, too. Safari search results, Siri, Handoff and even Reminders could surface your searchable content.

There are two "stores" where your content lives to become searchable:

1. **The on-device index**: This is never shared, nor synced, and remains directly on your device.
2. **Apple's server side index**: This is content you share that can be publicly searched by anyone, and it's driven by your app's website or web app.

We'll be focusing on the first one, the on-device index and setting up Spotlight search.

How it Works

There are two primary APIs to make your content searchable in Spotlight. Those are the Core Spotlight framework, and the do-it-all API `NSUserActivity`. Both of these will allow you to get content into Spotlight search – so naturally the questions becomes, which do you use? The answer might surprise you. In most cases, it's perfectly fine to use both of them.

To answer why that is, I'm going to show a full implementation of both approaches, and then we can have a better understanding of why you would use just one, or each of them together. Let's start with the Core Spotlight framework.

Core Spotlight Framework

The Core Spotlight framework supports search via three primary objects:

1. `CSSearchableItemAttributeSet`: This is where you define the uniform type identifier for the content you intend to share, along with metadata about it. Both approaches use this API.
2. `CSSearchableItem`: This represents the actual item you're indexing, and the related metadata about it comes the item attribute set above.
3. `CSSearchableIndex`: This is the store where you index things, update them or remove them.

Let's index a chapter from the book series – the one you're reading right now. Here, we index it with Core Spotlight and once we do, it's searchable within Spotlight:

```
// Xcode -> SwiftUI -> SpotlightSearch

struct SpotlightSearch {
    func index(chapter: BookChapter) async throws {
        let identifier = chapter.id.rawValue
```

```
        let attributeSet = CSSearchableItemAttributeSet(contentType:
.bestInClassChapter)
        attributeSet.title = "iOS APIs and Framework: \
(chapter.rawValue)"
        attributeSet.relatedUniqueIdentifier = identifier
        attributeSet.displayName = title

        let searchableItem = CSSearchableItem(uniqueIdentifier:
identifier,
                                              domainIdentifier:
"com.bicBookSeries2023.A-Best-in-Class-Tour",
                                              attributeSet:
attributeSet)

        try await
CSSearchableIndex.default().indexSearchableItems([searchableItem])
    }
}
```

There, we set metadata about the item, create the searchable item and add it to Core Spotlight's index. That's all it takes. I do want to call out `uniqueIdentifier` and `domainIdentifier` – the former is all about unique values for the searchable content, while the former represents anything specific to your app domain. So, if this was for a music based app – the `domainIdentifer` might be an album or song name.

Another "gotcha" that can occur is that if you omit the `displayName`, your item may not appear at all. This can be a pain to debug, and since you'll want it set anyway – be sure to always include it.

Once it's shown within Spotlight, though, it's up to you to handle it. For that, we'll look at the scene delegate in UIKit:

```
// Xcode -> SceneDelegate.swift

func scene(_ scene: UIScene, continue userActivity: NSUserActivity)
{
```

```
    if userActivity.activityType == CSSearchableItemActionType {
        // This came from Core Spotlight. Open the chapter.
        print(userActivity.description)
    } else {
        openBookChapter(with: userActivity)
    }
}
```

Handling it works the same way Handoff and other user activity APIs do. There is a unique `activityType` for content coming from Core Spotlight, and that's `CSSearchableItemActionType`. If the user activity has this type – it's from a Spotlight search.

In SwiftUI, we seem the same parallel with user activity handling. Again, the sample project is a UIKit-based app, but if it was SwiftUI – we'd use the user activity modifier somewhere close to the root view:

```
RootView()
.onContinueUserActivity(CSSearchableItemActionType) { userActivity
in
    handleActivity(userActivity)
}
```

User Activity

Next, let's accomplish the same thing, but with user activity:

```
// Xcode -> SpotlightSearch.swift

func userActivity(for chapter: BookChapter) -> NSUserActivity {
    let activity = NSUserActivity(activityType:
"com.bicBookSeries2023.A-Best-in-Class-Tour")

    activity.title = chapter.rawValue
    activity.userInfo = ["id": chapter.id]
    activity.persistentIdentifier = chapter.id.rawValue
    activity.isEligibleForSearch = true
```

```
    activity.isEligibleForPrediction = true
    activity.keywords = NSSet(array: [chapter.rawValue]) as! Set

    let identifier = chapter.id.rawValue
    let attributeSet =
CSSearchableItemAttributeSet(contentType: .bestInClassChapter)
    attributeSet.title = "iOS APIs and Framework: \
(chapter.rawValue)"
    attributeSet.relatedUniqueIdentifier = identifier

    activity.attributeSet = attributeSet

    return activity
}
```

As you can see, it's pretty similar to Core Spotlight. And, as we mentioned above – it gets it metadata from the same exact API as the search item did, `CSSearchableItemAttributeSet`.

From there, you simply assign it to a view controller, or if you're not using a view controller – as in SwiftUI – simply call `becomeCurrent()` on the user activity. In the sample project's case, we simply assign to `UIViewController` directly via its `userActivity` property. Then, it's available in Spotlight search:

```
// Xcode -> UIKit -> SpotlightFig1ViewController.swift

class SpotlightFig1ViewController: BaseSampleViewController {
    private let spotlight = SpotlightSearch()

    override func viewDidLoad() {
        super.viewDidLoad()
        let activity = spotlight.userActivity(for: .AirDrop)
        self.userActivity = activity
    }
```

```
}
```

The Differences Between the Two

When deciding which to use, be aware of the difference between the two:

1. User activity generally requires the user to "view" something before it is indexed. When using Core Spotlight, you can simply index anything and any point in your app's lifecycle. This is the primary difference.
2. User activity is more geared towards "navigational" purposes, too. Not GPS navigation (though it could be used for that, sure) but for your app's *content* navigation. You viewed something, and you want to go directly back to it later.
3. Generally, user activity is more powerful, which is understandable because it's a flexible API. Searchable items are *only* for indexing local content. User activity does that and a lot more. You can use it to weight the data in search results, they can markup web content, Siri Suggestions tie into and not to mention so many other parts of the system which use it.
4. If you want something publicly indexed on Apple's servers, you *have* to use user activity. Core Spotlight is always local indexing.

As I mentioned earlier, you can certainly use these two together. In fact, that's a powerful technique so you can get the benefits of both. In the case where Core Spotlight sees that you have an entry for some data from both APIs, it uses the Core Spotlight entry for Spotlight display data and the user activity is used for Siri Suggestions. This also brings an added benefit: your search weight is boosted by the system when you use both.

To link them together, simply set the `relatedUniqueIdentifier` on `CSSearchableItemAttributeSet` to be the same value as the searchable item's identifier. Here, I've refactored our `SpotlightSearch` Struct to link an activity to a searchable item. So, you'd first index the content via Core Spotlight, and then later set the user activity on a view controller or call its `becomeCurrent()` function.

At that point, they'll be linked by the `attributeSet.relatedUniqueIdentifier` value:

```
struct SpotlightSearch {
    func searchableItem(for chapter: BookChapter) ->
CSSearchableItem {
        let identifier = chapter.id.rawValue
        let attributeSet = CSSearchableItemAttributeSet(contentType:
.bestInClassChapter)
        attributeSet.title = "iOS APIs and Framework: \
(chapter.rawValue)"
        attributeSet.contentDescription = "A book chapter from the
best-in-class iOS app book series."
        attributeSet.relatedUniqueIdentifier = identifier

        let searchableItem = CSSearchableItem(uniqueIdentifier:
identifier,
                                              domainIdentifier:
"Book Chapter - \(chapter.id)",
                                              attributeSet:
attributeSet)

        return searchableItem
    }

    func index(chapter: BookChapter) async throws {
        let items = [searchableItem(for: chapter)]
        try await
CSSearchableIndex.default().indexSearchableItems(items)
    }
```

```
    func userActivity(for chapter: BookChapter) -> NSUserActivity {
        let activity = NSUserActivity(activityType:
"com.bicBookSeries2023.A-Best-in-Class-Tour")

        activity.title = chapter.rawValue
        activity.userInfo = ["id": chapter.id]
        activity.persistentIdentifier = chapter.id.rawValue
        activity.isEligibleForSearch = true
        activity.isEligibleForPrediction = true
        activity.keywords = NSSet(array: [chapter.rawValue]) as! Set

        let metadata = searchableItem(for: chapter)
        activity.contentAttributeSet = metadata.attributeSet

        return activity
    }
}
```

Again, an important thing to note here is that *both* APIs provide metadata the exact same way, via `CSSearchableItemAttributeSet`. So, linking them together, reusing them for similar content for both APIs, and simplifying code paths that generate them should be easy.

Tips

Generalized Content Types

When creating an attribute set – you've got to tie it back to some sort of uniform type identifier. Earlier in the book series – we made our own, so that's what we've used to represent book chapters:

```
let attributeSet =
CSSearchableItemAttributeSet(contentType: .bestInClassChapter)
```

However, you're still good to go if you don't have a custom one. Simply use `.content`, which represents any generalized user content from your app. For most cases, this is what you're surfacing anyways, so it will work just fine.

Provide Thumbnails

However, continuing with the point above – if you *do* have your own content type, a lot of times you've probably provided a custom thumbnail for that content type too. So, you can reuse that here as well.

Even if you don't have a custom content type, you can still provide a thumbnail. By default, it'll simply use your app's icon. You can set a thumbnail a few different ways on the attribute set:

```
let attributeSet =
CSSearchableItemAttributeSet(contentType: .bestInClassChapter)

// Use a url
attributeSet.thumbnailURL = thumbnailURL()

// Or data
attributeSet.thumbnailData = UIImage(named: "thumb")!.pngData()

// And provide a dark mode version
attributeSet.darkThumbnailURL =  thumbnailDarkModeURL()
```

Handle Changing Data

For the content you've indexed, be sure to update the store to keep it in lockstep with your app's current state. For example, if you added a document to the local index –

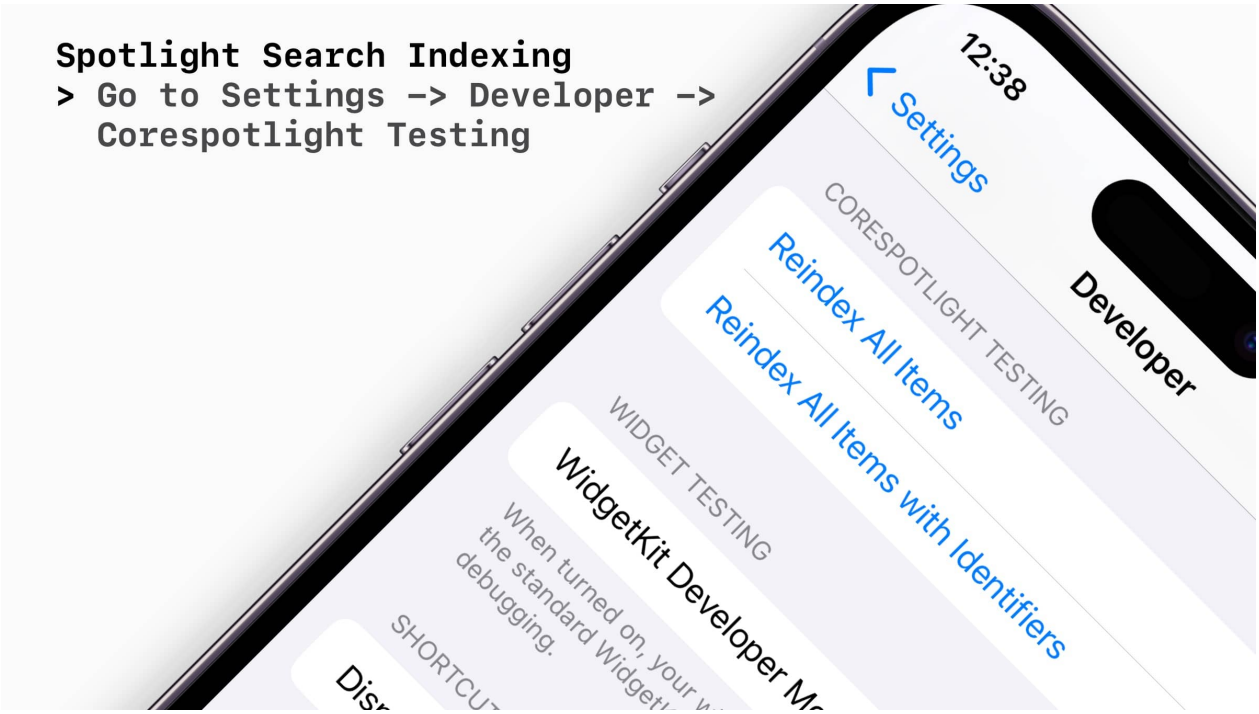
but you renamed it, deleted it or just simply updated it after a bit of time, you need to reflect that in the index:

```
let store = CSSearchableIndex.default()

// Delete
try await store.deleteSearchableItems(withIdentifiers: [docID])

// Update or add it
try await store.indexSearchableItems([docID])
```

Index Debugging

If you'd like to quickly reindex items, or simply reindex things with valid identifiers – you can skip doing that in your code and simply use the simulator or device developer settings:

1. Open the Settings app.
2. Go to "Developer".
3. Then, scroll down to "Corespotlight Testing"



Continuing Searches In-App

There may be scenarios where it would make sense to continue a Core Spotlight search within your app. A classic example of this may be Apple's Maps app. If you used Core Spotlight to search for a restaurant – it would make more sense if the data Maps surfaced in Core Spotlight for this specific query could be carried out within the app itself.

This is achievable using a somewhat obscure functionality of Core Spotlight, the `CSQueryContinuationActionType` global variable. The flow works like this:

1. You add a .plist value to your app, the `CoreSpotlightContinuation`, and set its boolean value to `true`.
2. Then, in your app delegate's `application(_:willContinueUserActivityWithType:)` implementation, you'll receive a `NSUserActivity`

with an `activityType` of `CSQueryContinuationActionType` when this type of Core Spotlight item is tapped.

3. In those cases, within the user info dictionary – you'll get the search term by getting the value of the `CSSearchQueryString` key.

Assuming you've got the property list key added with the correct value, the implementation would be something like this:

```
func scene(_ scene: UIScene, continue userActivity: NSUserActivity)
{
    if userActivity.activityType == CSQueryContinuationActionType,
       let searchTerm = userActivity.userInfo?[CSSearchQueryString]
as? String {
       // Show your search UI
    }
}
```

Three Key Takeaways

1. Use Spotlight suggestions to boost your app's content within Spotlight search.
2. Consider which of the two available APIs to use, or if both are appropriate.
3. Keep your indexed content up to date within the store.

Quick Actions

Home Screen Quick Actions provide a way to quickly allow users to perform meaningful tasks, directly from the Home Screen. These shortcuts (not to be confused with the actual Shortcuts APIs, I mean shortcuts in the literal sense of the word) let people get things done fast. Introduced in iOS 12, they felt incredible with 3D Touch enabled devices. Years later, even though 3D Touch technology has been sunset, many of its innovations such as this one live on.

Quick Actions are presented when someone taps and holds on an icon from the Home Screen. A typical context menu will present, and even if the app doesn't support Home Screen Quick Actions, you'll see a list of actions such as "Edit Home Screen" and "Remove App". Though, for most apps, a list of actions appear above that – and those are the quick actions the developers provided.

Here are some examples from Apple's own apps, from left-to-right: Messages, Wallet and the App Store:



There's a nice mix of immutable, always relevant actions. In Messages, there's a "New Message" action. No matter what happens in Messages, performing that action will always be relevant, and someone might always want to do it. It's a core flow of the app.

But beyond that, there's also context-specific actions. These are actions which are created based on user actions. Consider the Quick Actions I have showing in Messages once again. Those are all recently viewed threads in my Messages app. Actions like this are created dynamically, and actions like "New Message" are created statically.

Honestly, there isn't much more to say about Home Screen Quick Actions – and I think that's a good thing. It's a to-the-point API, and it's easily understood from both a developer and user perspective. Let's take a look at how they work.

How it Works

Quick Actions are powered by `UIApplicationShortcutItem`. This class represents any Home Screen Quick Action you can perform. At its core, though, it's simply a container. Using it, you'll provide a title, subtitle, an icon, its "type" (which you define, it simply helps disambiguate multiple actions from each other) and some domain-specific data in a `userInfo` dictionary.

For example, in the first image of this chapter showing off some of Apple's Home Screen Quick Actions – we can see where the data from the API correlates to what was shown on screen. Let's take the "New Message" action from messages: "New Message" was the title, and a compose icon was its icon.

Since we noted that there are two different types of actions we can use (ones provided at build time, and ones we create during runtime), that means there are two types of Home Screen Quick Actions:

Static Actions

These actions are ones which make sense to have available no matter the state of the app. Again, for Messages, we'd always want to possibly send a message. That workflow doesn't change based on how we use the app. Further, it doesn't require any dynamic data. Messages will always have a user signed in, based off the Apple ID that's used on the device, so this is a great choice for a statically provided quick action.

These static actions are derived from your app's `info.plist` – and you can provide up to four of them (that's dynamic actions included, too). In our case, we could make a static action to view all the code samples in the demo app. Given that, our info.plist entry will look like this:

```
<key>UIApplicationShortcutItems</key>
<array>
    <dict>
```

```
        <key>UIApplicationShortcutItemIconType</key>
        <string>UIApplicationShortcutIconTypeSearch</string>
        <key>UIApplicationShortcutItemTitle</key>
        <string>View All Code Samples</string>
        <key>UIApplicationShortcutItemType</key>
        <string>ShowAllSamplesQuickActionType</string>
    </dict>
</array>
```

Let's take a look at each key. First, `UIApplicationShortcutItemIconType`.

Since we're defining this at build time, this means we'll have to define our icon ahead of time, too. And for that, we've got two options. Either use one of the supplied icons from `UIApplicationShortcutItemType` or use an SF Symbol. For the former, we've got these types of iconography at our disposal:

- Compose
- Play
- Pause
- Add
- Location
- Share
- Search
- Prohibit
- Contact
- Home
- Mark Location
- Favorite
- Love
- Cloud

- Invitation
- Confirmation
- Mail
- Message
- Date
- Time
- Capture Photo
- Capture Video
- Task
- Task Completed
- Alarm
- Bookmark
- Shuffle
- Audio
- Update

We've used this approach in our static action. To use any of those, simply prepend `UIApplicationShortcutIconType` in front of it, and remove any white space. For example, "Capture Photo" would be `UIApplicationShortcutIconTypeCapturePhoto` in your .plist file. These are also available in code, too, for dynamic quick actions via `UIApplicationShortcutIcon`:

```
let icon = UIApplicationShortcutIcon(type: .favorite)
```

For SF Symbol use, you simply change the key to `UIApplicationShortcutItemIconSymbolName` in the .plist and provide the symbol's string as the value:

```
<key>UIApplicationShortcutItemIconSymbolName</key>
```

```
<string>rays</string>
```

And, via code, use a different initializer available on the shortcut icon type:

```
let icon = UIApplicationShortcutIcon(systemImageName: "rays")
```

Next, we've got `UIApplicationShortcutItemTitle` – and that's as simple as it sounds. We provide a string that succinctly describes what the action will do. In our case, "View Code Samples" should provide obvious context – using this will bring you to all the app's code samples.

Finally, there's `UIApplicationShortcutItemType` – and this is more of a identifier unique specifically for developers. We can use to know which kind of action we're handling later on, which is useful if we provide several types of them. Here, I just went with `ShowAllSamplesQuickActionType`. We'll look at this again soon.

Dynamic Actions

For dynamic actions, we skip the .plist entries and instead opt for code. Remember, actions here are the result of direct choices the user has made in your app. So, it only makes sense to provide these at runtime. The flow is, unsurprisingly, the same – except we simply create a `UIApplicationShortcutItem` ourselves.

In our case, let's provide a dynamic action to view a recently viewed chapter code sample. For implementation, that means we'd add a new shortcut item anytime a chapter is viewed. With our Struct below, we just need to call `setQuickActionFor(:)` when a chapter is selected:

```
// Xcode -> ChapterQuickActionVendor.swift

struct ChapterQuickActionVendor {
    static let homeAction = "ShowAllSamplesQuickActionType"
    static let showSampleType = "ShowSampleQuickActionType"
```

```
    static func setQuickActionFor(bookChapter: BookChapter,
interface: InterfaceFramework) {
        let application = UIApplication.shared
        let action = createAction(with: bookChapter, interface:
interface)
        var items: [UIApplicationShortcutItem] =
application.shortcutItems ?? []

        items.append(action)

        application.shortcutItems = items
    }

    private static func createAction(with chapter: BookChapter,
interface: InterfaceFramework) ->
UIApplicationShortcutItem {
        UIApplicationShortcutItem(type:
ChapterQuickActionVendor.showSampleType,
                                  localizedTitle: chapter.rawValue,
                                  localizedSubtitle: "View Code
Samples",
                                  icon:
UIApplicationShortcutIcon(systemImageName: "keyboard.fill"),
                                  userInfo: ["interface":
interface.rawValue as NSSecureCoding])
    }
}
```

Here, we create a shortcut item based off the chapter and interface framework passed to us. To make it show up on the Home Screen, we assign to the `UIApplication.shared.shortcutItems` array. Now, anytime a chapter is viewed, we'll use this Struct to add a dynamic Home Screen Quick Action:

```
// Xcode -> GlossaryViewController.swift

// MARK: Table View Delegate
```

```
func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
    guard let chapter = datasource.itemIdentifier(for: indexPath)
else {
        tableView.deselectRow(at: indexPath, animated: true)
        print("Could not find a chapter for the selected row.")
        return
    }

    // Update our user activity
    setSceneUserActivityFor(bookChapter: chapter, interface:
interfaceChoice)

    // Update Home Screen Quick Actions
    ChapterQuickActionVendor.setQuickActionFor(bookChapter: chapter,
interface: interfaceChoice)
}
```

And with that, we're all set. View a chapter, and then it'll immediately show on the Home Screen:



Handling Actions

Finally, there are two ways to handle these actions.

I do want to note that the easiest way to do this in SwiftUI is with an app delegate adapter. These are no SwiftUI-first ways to handle them as of now. Currently, the API also routes through a scene delegate, which we can't access in SwiftUI. At the end of this section, I do have an example showing you how to get it wired up in SwiftUI.

Before I show you where iOS will vend the actions, we're going to write a function to handle them first. In it, we'll perform the action if we can – and then true a `Bool` indicating if we could do it or not. You'll see why this is helpful later, as there are a few places these actions could come in:

```
// Xcode -> SceneDelegate.swift
```

```
private func handleQuickAction(shortcutItem:
UIApplicationShortcutItem) -> Bool {
    guard let rootNav = window?.rootViewController as?
CustomNavController else {
        return false
    }

    if shortcutItem.type == ChapterQuickActionVendor.homeAction {
        rootNav.popViewController(animated: false)
        return false
    } else if shortcutItem.type ==
ChapterQuickActionVendor.showSampleType {
        let chapter = BookChapter(rawValue:
shortcutItem.localizedTitle) ?? .AirDrop
        let interface = InterfaceFramework(rawValue:
shortcutItem.userInfo?["interface"] as? Int ?? 0) ?? .SwiftUISample

        guard let glossaryVC = rootNav.viewControllers.first as?
GlossaryViewController else {
            return false
        }

        glossaryVC.pushBookChapter(chapter: chapter, with:
interface)
    }

    return true
}
```

Here, we look at the type of action we have, its `shortcutItem.type` property, and act on it. Since String-based APIs are easy to get wrong, it's always a good idea to make these constants. We did exactly that when we wrote our `ChapterQuickActionVendor` earlier. After we check the type, if we could perform the action, we return either true or false. With that in place, we're ready to handle incoming actions.

To hook into those actions in the app's lifecycle, we first must consider whether the action resulted in app launch. If it did, we'll get access to it in our scene delegate. From there, we simply want to hang on to it – and use it later to perform the action. We do this here at the bottom:

```
// Xcode -> SceneDelegate.swift

class SceneDelegate: UIResponder, UIWindowSceneDelegate {
    var window: UIWindow?

    // An action we can use after app launch...
    private var executedQuickAction: UIApplicationShortcutItem? =
nil

    func scene(_ scene: UIScene, willConnectTo session:
UISceneSession, options connectionOptions:
UIScene.ConnectionOptions) {
        guard let windowScene = (scene as? UIWindowScene) else
{ return }

        let rootController = GlossaryViewController()
        let rootNavController =
CustomNavController(rootViewController: rootController)
        window = UIWindow(windowScene: windowScene)
        window?.rootViewController = rootNavController
        window?.makeKeyAndVisible()

        // Scene restoration
        if let previousActivity = session.stateRestorationActivity {
            openBookChapter(with: previousActivity)
        }

        // Quick Action?
        if let shortcutItem = connectionOptions.shortcutItem {
            executedQuickAction = shortcutItem
        }
    }
```

```
}
```

In the provided `connectionOptions`, there is an optional `shortcutItem`. If we have one, that means the app was launched from a quick action. So, we need to save it off – and later use it when the scene becomes active. This is similar to how we also handle user activities:

```
// Xcode -> SceneDelegate.swift

func sceneDidBecomeActive(_ scene: UIScene) {
    if let userActivity = window?.windowScene?.userActivity {
        userActivity.becomeCurrent()
    }

    if let shortcut = executedQuickAction {
        let _ = handleQuickAction(shortcutItem: shortcut)
    }
}
```

There, we check for an instance of the action, and it's there, we execute it – ignoring the `Bool` return value.

Okay – so what if the app was already in the foreground? In that case, things are a little easier. We don't have to hang on to a shortcut item ourselves, because it will be handed right to us in the scene delegate:

```
// Xcode -> SceneDelegate.swift

// MARK: Quick Actions
func windowScene(_ windowScene: UIWindowScene,
                 performActionFor shortcutItem:
UIApplicationShortcutItem,
                 completionHandler: @escaping (Bool) -> Void) {
    completionHandler(handleQuickAction(shortcutItem: shortcutItem))
}
```

The only material difference is that we have a `completionHandler` closure we must call to let the system know we've handed the action. Simply pass either `true` or `false`, depending on whether we were able to handle the action. With our previous work to return a `Bool` earlier, we're all set to handle that here (in addition to when the app was launched, too).

Using SwiftUI

To handle quick action in SwiftUI, we'll need an object conforming to `UIAppDelegateDelegate`. For years, an app delegate was the default mechanism for handling lifecycle events. Since then, it's given way to scene delegates in UIKit, and the `App` structure in SwiftUI-based apps.

To take care of quick actions, we'll need both a scene delegate and an app delegate. The app delegate wires up the scene delegate, while the scene delegate handles the actual quick actions. Here's how it would look:

1. Create an application delegate:

```
class SomeAppDelegate: NSObject, UIApplicationDelegate {
    func application(_ application: UIApplication,
                     configurationForConnecting
connectingSceneSession: UISceneSession,
                     options: UIScene.ConnectionOptions) ->
UISceneConfiguration {
        let configuration = UISceneConfiguration(name:
connectingSceneSession.configuration.name,
                                                 sessionRole:
connectingSceneSession.role)
        configuration.delegateClass = SceneDelegate.self
        return configuration
    }
```

```
}
```

2. Create a scene delegate:

```
class SceneDelegate: UIResponder, UIWindowSceneDelegate {
    private var executedQuickAction: UIApplicationShortcutItem? =
nil

    func scene(_ scene: UIScene,
               willConnectTo session: UISceneSession,
               options
               connectionOptions: UIScene.ConnectionOptions) {

        if let shortcutItem = connectionOptions.shortcutItem {
            executedQuickAction = shortcutItem
        }
    }

    func sceneDidBecomeActive(_ scene: UIScene) {
        if let shortcut = executedQuickAction {
            handleShortcutItem(shortcut)
        }
    }

    func windowScene(_ windowScene: UIWindowScene,
                     performActionFor shortcutItem:
UIApplicationShortcutItem,
                     completionHandler: @escaping (Bool) -> Void) {
        handleShortcutItem(shortcutItem)
        completionHandler(true)
    }
}

extension SceneDelegate {
    func handleShortcutItem(_ item: UIApplicationShortcutItem) {
        // Implementation of handling shortcut item
    }
```

```
}
```

3. Create a reference to the app delegate in your `App` structure:

```
@main
struct EliteHoopsApp: App {
    @UIApplicationDelegateAdaptor(SomeAppDelegate.self) var
appDelegate
}
```

While it is a little bit of code that I wish we could do without, until there is a SwiftUI-first way of handling these, this is the only way to do it.

Tips

Don't Worry About Counts with Dynamic Actions

When you create dynamic actions, you may tempted to truncate some type of data or Array. After all, we know we can only provide a maximum of four Home Screen Quick Actions – and that's static and dynamic actions added together. So, in our example, we provided one static action, "View All Code Samples".

For our dynamic ones wherein we added an item for each recently viewed chapter – you may naturally think that you'd have provide *only* three of the most recently viewed:

```
let recentlyViewed = Array(recentlyViewedCodeSamples.prefix(3))
```

That's unnecessary – because the framework will only take the maximum amount of Quick Action slots you have left. Providing more won't cause a crash or odd behavior,

so simply assign what you've got and iOS will display the most recent items you supply it given the amount of spots it has left.

Respond Gracefully to Edge Cases

Since Home Screen Quick Actions can be available immediately once your app is installed, take care to handle edges cases. For example, if you have a statically provided action to, say, view some sort of feed in a social app — well, that wouldn't make any sense if they haven't logged in.

Basically, this boils down to being smart about your app's state, and where in the app's journey the user currently is. Don't assume that because they've used a Home Screen Quick Action, that the action is ready to perform.

At the beginning of the chapter, when speaking about static Home Screen Quick Actions, I mentioned Messages and its "New Message" action. I said the following:

> *Messages will always have a user signed in, based off the Apple ID that's signed into the device, so this is a great choice for a statically provided quick action.*

That's not *tehcnically* true, I suppose. You could sign out of an Apple ID on your iPhone — and when you do, Messages will prompt you to sign in. So, that's a real-world example of the types of edge cases you'll want to consider.

Copy Matters

You don't have a lot of space when it comes to naming your Home Screen Quick Actions. Try to keep it short and avoid truncation while considering localization too. But, the most important part is making them completely *predictable*. The result of the action should be obvious before someone carries it out.

For example, titles like “Find Nearest Store” for a retail shop’s app is obvious. Where is the store that’s nearest my location? But, consider if it simply just said “Find” as the title. Find? Find what? Items at the store, the store itself, something else? The title is shorter, yes, but it’s not better. Be clear and concise.

Further, use the subtitle only if it adds essential context. The subtitle is not required, and I’d argue it’s just more to digest visually if you provide it. But for some actions, they could be critical or simply informational. For example, in an email app, the action’s title may be the inbox’s name, and the subtitle would be the number of messages unread. That gives more context to the user as to whether they may want to perform the action at all.

Update Items Dynamically

As we saw earlier, you can update your Home Screen Quick Actions on the fly. As information and data changes, you want to surface things that are most helpful from the Home Screen. For example, if you had a cooking app – then maybe you’d want to offer actions to view the most recently viewed recipes.

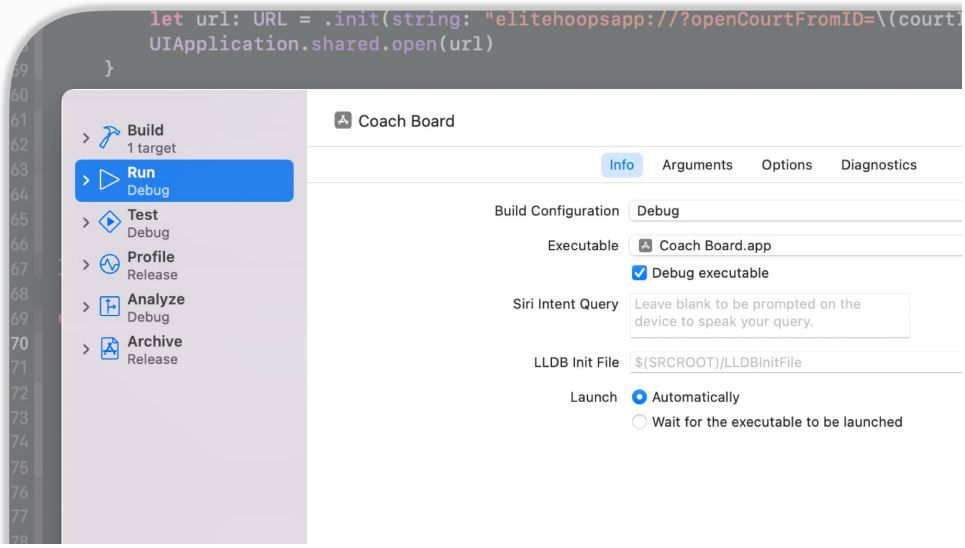
In the Messages app, you’ll see that very idea in action. Take a look at the Home Screen Quick Actions available for it now on your phone. Then, compare those actions again once you view a couple of conversations. They’ll update the messages you’ve recently viewed.

Debugging Quick Actions Which Launch an App

Sometimes, a quick action will require the app to be initialized and launched. In those cases, you might assume it would be difficult to debug them – how could debug an app’s launch? Xcode, thankfully, has a feature you can use for this exact scenario.

Choose **Edit Scheme** –> **Run** –> **Info** –> **Launch** –> Check "**Wait for the executable to be launched**":

Debugging Home Screen Quick Actions
> Use the "Wait for the executable to be launched" option under "Launch".



Now, lldb will attach to your app's process when it's launched. This will let you breakpoint at *any* point in the app's lifecycle – which makes it trivial to debug issues related to your quick actions resulting in an app launch.

So, where does it make sense to add a breakpoint in these situations? Most likely, you'll want to set one in your scene delegate, specifically at `scene(_:willConnectTo:options:)`.

Three Key Takeaways

1. Home Screen Quick Actions should offer shortcuts to perform high-value tasks in your app.
2. Be sure to keep them updated in your app as data changes.
3. You can provide up to four Home Screen Quick Actions, so find a nice mix of statically provided actions along with dynamic ones.

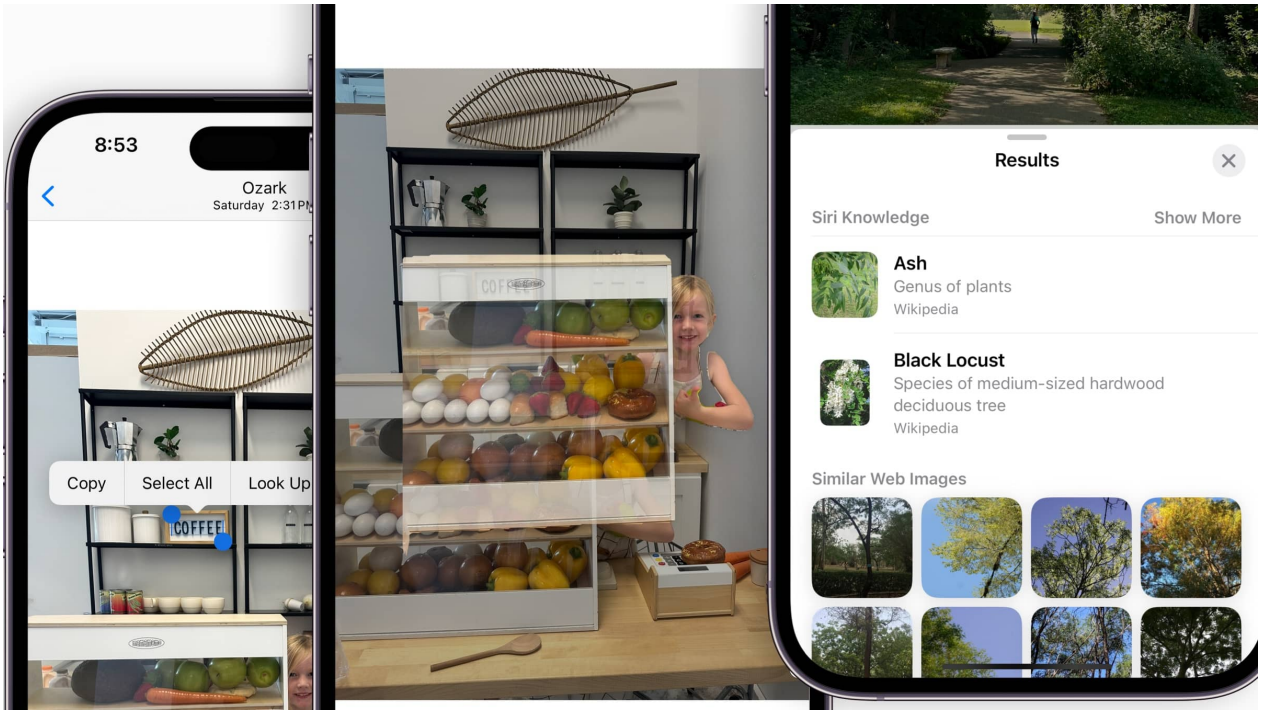
VisionKit

VisionKit allows our software to understand photos and videos in much the same way we do.

On the outset, that may not sound incredibly interesting, but in reality – this is compelling. When someone hands you a business card, what do you do? You scan over it. There is likely a name, maybe a photo, certainly an email address and phone number. Using VisionKit, we can have our apps ascertain that same information. That's where the possibilities really start to open up.

You can scan business cards, sure. But you can pick out URLs, detect animals in photos, centralize where a face is in within a photo and so much more. Using Apple's machine learning suite, we can start to leverage things like Live Text in our apps. VisionKit gives us high-level abstractions to do all of that and more.

You see this framework all over iOS, even if you don't realize it. Things like detecting text in an image, lifting the primary subject and content within an image or detecting plants are all powered by this technology:



Now, before we go on, I want to clear up something. There are two different frameworks named similarly on iOS – Vision, *and* VisionKit. Vision is a powerhouse of a framework, geared to help developers create apps which augment, detect and use data from photos and videos. Then, with VisionKit, we have higher-level abstractions and ready-made interfaces we can use to do similar things.

You can go deep and wide with Vision, but you can do common Vision-related tasks with pre-made interfaces, faster, using VisionKit. To that end, we'll be looking at the main heavy hitters of the VisionKit framework:

The Data Scanner

This control lets us quickly scan for data found within either an image or live video feed. Using it, we can find phone numbers, names, letters and more without having to set up all the Vision-related requests and objects. We can simply tell it what we want to find, and present it.

Image Analysis Interactions

This interaction class lets us analyze an image which allows us to surface any detected content types on top of existing images, trigger Visual Lookup and more. This is very similar to how the Live Text interface works. You pull up a photo in your camera roll with text, and you get the nice little icon to let you know you can copy text or data from it.

The Document Scanner

This is similar to the data scanner control, except that it is specifically geared towards scanning documents. So, think papers, .pdfs – that type of thing. This helps you create a digital representation of real-world documents, which opens up all sorts of interesting use-cases. You can scan them, organize them, easily search them and more.

Let's go through each of them one-by-one.

How it Works

Data Scanning

This ready-made view controller can be set up to scan data from a live video feed. This lets you focus on getting the data you're after without managing audio and video sessions, which historically speaking, has been a high barrier of entry for this type of task.

To get started, since we're using the device's camera – we'll need to include the `NSCameraUsageDescription` key in our info.plist:

```
<key>NSCameraUsageDescription</key>
<string>For our code samples.</string>
```

Of course, in the demo app's case – we simply want to include to run code samples. As always, include a short, descriptive reason for why you need permissions. For data scanning purposes, it would be helpful to be even more to-the-point, perhaps something like "We'll scan documents locally on your device", or similar.

Next, consider what type of data you're after. We have several options to use, all powered by `RecognizedDataType` instances. There are two primary domains it captures, and that's text and barcodes. But, within those types, there are subdomains of each. For example, you could say you want to capture only phone numbers, or only QR barcodes.

Let's breakdown what we can capture with each type.

Text

When it comes to text, we have two main choices to make. What languages are we interested in capturing, and what kind of text data do we want? If we can get as specific as possible, performance will improve. Outside simply narrowing down what the framework is looking for, Apple will also use a better-suited machine learning model for detection.

So, if we wanted to capture links and only in the English language – we'd do something like this:

```
.text(languages: ["en-US"], textContentType: .URL)
```

There are several text content types available for use:

- `dateTimeDuration`
- `emailAddress`

- `flightNumber`
- `fullStreetAddress`
- `shipmentTrackingNumber`
- `telephoneNumber`
- `URL`
- `currency`

Barcodes

For barcodes, each type is represented by `VNBarcodeSymbology`, and virtually any barcode you could think of is represented:

```
extension VNBarcodeSymbology {
    @available(iOS 11.0, *)
    public static let aztec: VNBarcodeSymbology

    @available(iOS 11.0, *)
    public static let code39: VNBarcodeSymbology

    @available(iOS 11.0, *)
    public static let code39Checksum: VNBarcodeSymbology

    @available(iOS 11.0, *)
    public static let code39FullASCII: VNBarcodeSymbology

    @available(iOS 11.0, *)
    public static let code39FullASCIIChecksum: VNBarcodeSymbology

    @available(iOS 11.0, *)
    public static let code93: VNBarcodeSymbology

    @available(iOS 11.0, *)
    public static let code93i: VNBarcodeSymbology

    @available(iOS 11.0, *)
    public static let code128: VNBarcodeSymbology
```

```
@available(iOS 11.0, *)
public static let dataMatrix: VNBarcodeSymbology

@available(iOS 11.0, *)
public static let ean8: VNBarcodeSymbology

@available(iOS 11.0, *)
public static let ean13: VNBarcodeSymbology

@available(iOS 11.0, *)
public static let i2of5: VNBarcodeSymbology

@available(iOS 11.0, *)
public static let i2of5Checksum: VNBarcodeSymbology

@available(iOS 11.0, *)
public static let itf14: VNBarcodeSymbology

@available(iOS 11.0, *)
public static let pdf417: VNBarcodeSymbology

@available(iOS 11.0, *)
public static let qr: VNBarcodeSymbology

@available(iOS 11.0, *)
public static let upce: VNBarcodeSymbology

// Revision 2
@available(iOS 15.0, *)
public static let codabar: VNBarcodeSymbology

@available(iOS 15.0, *)
public static let gs1DataBar: VNBarcodeSymbology

@available(iOS 15.0, *)
public static let gs1DataBarExpanded: VNBarcodeSymbology

@available(iOS 15.0, *)
```

```
    public static let gs1DataBarLimited: VNBarcodeSymbology

    @available(iOS 15.0, *)
    public static let microPDF417: VNBarcodeSymbology

    @available(iOS 15.0, *)
    public static let microQR: VNBarcodeSymbology

    @available(iOS 17.0, *)
    public static let msiPlessey: VNBarcodeSymbology
}
```

Using the data scanner is mostly an affair of deciding ahead of time what data you want to capture, and how accurate the results must be. As you'll see – the API requires you to make these decisions upfront in its initializer. From there, it's a matter of responding to results via protocol adoption.

Before we show an example, yes – this does mean there is no native SwiftUI data scanner. In fact, that holds true for the controls in this entire chapter, unfortunately. But, this is why Apple created ways to bridge UIKit to SwiftUI. Hopefully, in later releases, we will see a SwiftUI-first way to do these things – and I'll update things here if that happens.

With that, lets set up a data scanner to find the text type we mentioned earlier. That is, any URL that's in English. Here's what that would look like:

```
// Xcode -> UIKit -> VisionKitFig1ViewController.swift

import UIKit
import VisionKit

class VisionKitFig1ViewController: BaseSampleViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        presentDataScanner()
    }
```

```
    private func presentDataScanner() {
        let scannerVC = DataScannerViewController(
            recognizedDataTypes: [.text(languages: ["en-US"],
textContentType: .URL)],
            qualityLevel: .balanced,
            recognizesMultipleItems: false,
            isHighFrameRateTrackingEnabled: false,
            isHighlightingEnabled: true)

        scannerVC.delegate = self

        if scannerVC.isScanningAvailable {
            self.present(scannerVC, animated: true) {
                try? scannerVC.startScanning()
            }
        } else {
            // Delegate callback will fire
        }
    }
}
```

The initializer is where all the important choices have to be made. I'll step through each argument so that you can make your own informed choices:

1. `recognizedDataTypes`: This is what we called out in the beginning – what kind of data are you after? For us, that's URL text in English.
2. `qualityLevel`: The quality level which will impact the camera's resolution during capture. Do you want fast, possibly less accurate results? Do you want very accurate results at the cost of processing time? A mix of both?
3. `recognizesMultipleItems`: Should the scanner pick up all results within view – or focus on the one in the middle, otherwise known as the person's point of interest. This can be changed on the fly by the user tapping on the view.

4. `isHighFrameRateTrackingEnabled`: If you want to overlay your interface adornments on the feed, you'll want this to be enabled. If not, leave it to false.
5. `isHighlightingEnabled`: This is the yellow, target-like interface element that shows over matches. If you're not providing some sort of custom feedback for matches, I think it's a good call to leave this on.

There is still a little massaging we've got to do before presenting it. We need to set its delegate, first:

```
scannerVC.delegate = self
```

If we skip this, we can't do anything with the results. Next, you'll see I have this check:

```
if scannerVC.isScanningAvailable {
    // Presentation code...
}
```

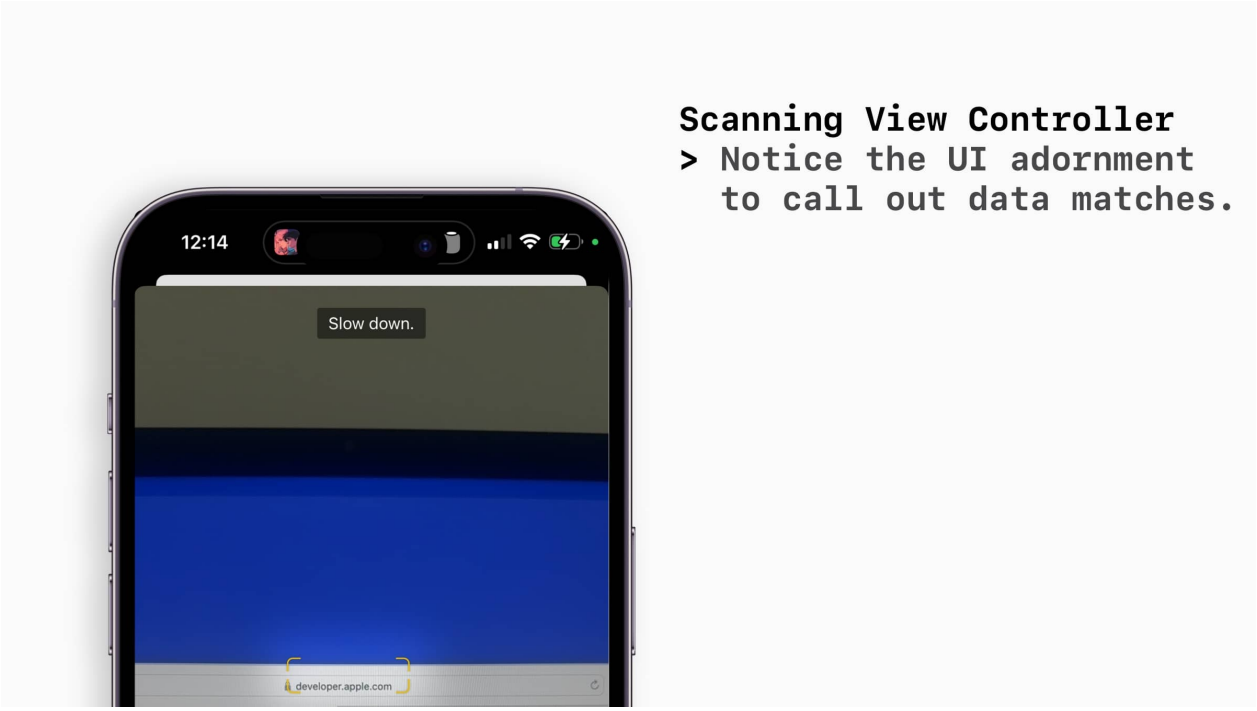
This is a custom extension I've made because you really need to check two things before presenting a scanner. Is it available from a hardware and system standpoint, and has the user granted access to the camera? Along those same lines, when dismissing a scanner – we have to call a function to turn off scanning ourselves.

Since these are two-step processes that we always must do, I've created extensions to house both of them:

```
// Xcode -> UIKit -> VisionKitFig1ViewController.swift

extension DataScannerViewController {
    var isScanningAvailable: Bool {
        return DataScannerViewController.isSupported &&
DataScannerViewController.isAvailable
    }
```

```
    func dismiss() {
        stopScanning()
        dismiss(animated: true)
    }
}
```

And with that, we've got a live scanner that should pick up an URLs in the English language. If you run the sample code and point it at any URLs, you should see a result similar to this:



If we needed to do more with the results we've found, the delegate methods are where we would insert any logic:

```
// Xcode -> UIKit -> VisionKitFig1ViewController.swift

extension VisionKitFig1ViewController:
DataScannerViewControllerDelegate {
    func dataScanner(_ dataScanner: DataScannerViewController,
```

```
                    didAdd addedItems: [RecognizedItem],
                    allItems: [RecognizedItem]) {
        // Items discovered
    }

    func dataScanner(_ dataScanner: DataScannerViewController,
didTapOn item: RecognizedItem) {
        switch item {
        case .text(let scannedURL):
            print("Found :\(scannedURL)")
        default:
            print("No op")
        }
    }

    func dataScanner(_ dataScanner: DataScannerViewController,
                    becameUnavailableWithError error:
DataScannerViewController.ScanningUnavailable) {
        print("Scanner is unavailable: \
(error.localizedDescription)")
    }
}
```

Image Interactions

`UIInteraction` can be found all throughout UIKit. In fact, if you search that term in this book series – you're going to find a lot of results. As such, `ImageAnalysisInteraction` works much the same way other interactions do:

- Create an interaction.
- Assign a delegate.
- Attach it to a view.

However, in this case – there's a bit more to consider since we need to pair that interaction with two things:

1. `ImageAnalyzer.Configuration`: This configuration will let VisionKit know what kind of data we're looking for in an image.
2. `ImageAnalyzer`: This will take in the configuration and perform the analysis. Once it's finished, we assign it back to the interaction and the process is complete.

Using the image interaction opens us up to offer four different types of functionality without a lot of work on our end. That is adding a Live Text experience to images, supporting subject lifting, offering Visual Lookup and finding machine-readable codes (i.e., barcodes). Let's say we wanted to have the Live Text user experience in our image views in an app. How would that look?

End-to-end, the process goes like this:

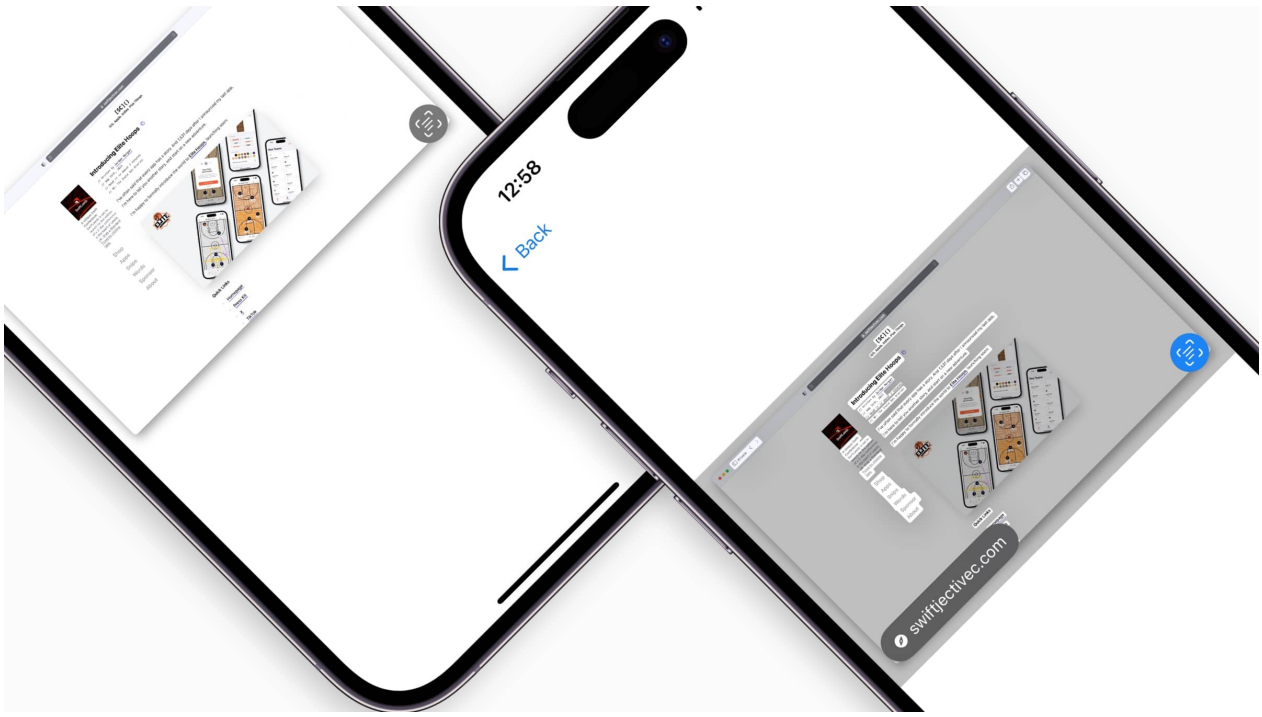
1. Create our image interaction, and set the preferred interaction type on it. If you don't, nothing will happen — and it defaults to an empty array.
2. Add that interaction to an existing image view.
3. Create an image analyzer configuration — and much like we set a preferred interaction typer on the `ImageAnalysisInteraction`, so to must be pass the data we want to analyze for here.
4. You pass the configuration and image to a `ImageAnalyzer` and have it perform analysis.
5. Finally, you set the result of the analysis *back to the starting image interaction* and you're done. VisionKit will take care of presenting any visual chrome for you.

That may sound like a lot, but the process isn't that heavy code-wise:

```
// Xcode -> UIKit -> VisionKitFig2ViewController.swift
```

```
// See the file in Xcode to see image view setup if needed...

override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)
    Task {
        await scanImage()
    }
}

private func scanImage() async {
    guard let img = imageView.image else {
        print("Image isn't present within image view.")
        return
    }

    // Add interaction with interaction types
    let liveTextInteraction = ImageAnalysisInteraction(self)
    liveTextInteraction.preferredInteractionTypes = [.textSelection]
    imageView.addInteraction(liveTextInteraction)

    // Create a config and analyzer
    let configuration = ImageAnalyzer.Configuration([.text])
    let analyzer = ImageAnalyzer()

    // Perform analysis, and set it back to the interaction
    do {
        let analysis = try await analyzer.analyze(img,
configuration: configuration)
        liveTextInteraction.analysis = analysis
        liveTextInteraction.preferredInteractionTypes = .automatic
        print("Text detected.")
    } catch {
        print(error.localizedDescription)
    }
}
```

With that, all set – when we run the sample app, the image will display and the analysis will be performed concurrently. When it finishes, we get the Live Text button that will appear to let us know data was found:



If you need more control over the interaction, or to otherwise respond to it – use the delegate methods. Here, we print out to the console anytime the Live Text button hides or shows:

```
// Xcode -> UIKit -> VisionKitFig2ViewController.swift

extension VisionKitFig2ViewController:
ImageAnalysisInteractionDelegate {
    func interaction(_ interaction: ImageAnalysisInteraction,
                     liveTextButtonDidChangeToVisible visible: Bool)
{
        print("Live text button visibility change.")
    }
```

```
}
```

Document Scanning

Finally, we come to the document scanner. This controller is meant to scan in data from any real-world documents. When you don't need a live video capture setting, this option is both faster, easier to use and provides a more concrete return type for the resulting data. You'll always receive a document class that represents the number of items scanned. This is the same controller that you might've seen in the stock Files, Messages, Notes, and Mail apps.

So, what exactly does this give us?

Well, for one – we don't have to do any interface setup. The document scanner is ready-made. And, again – fewer configuration options to worry about. Plus, it evenly "lights" the results, and corrects the perspective as well. If you've scanned a document in iOS before, you likely know what I'm talking about here. If a document you scanned was sideways and dimly lit – the scan appears more evenly lit, and rotated correctly. Plus, users can manually crop it too if they only wish to scan parts of a document.

However, this one does require more work from us to actually *use* the results. This is understandable because the result we get is a bit abstract – the API expects us to process it further to use it. And for that, we'll need to dip into the Vision framework.

From a setup standpoint, though, it requires the least work out of all of these options. Simply present it, assign a delegate – and that's it. The result we get from it will come in via a delegate method, and it'll be represented by a `VNDocumentCameraScan` instance. From there, we'll process results with Vision APIs. The only other "gotcha" of this API is that we must always be responsible for dismissing the scanning controller itself when we're finished.

With that said, let's take a look at a real world example where we retrieve the text from a scanned document:

```
// Xcode -> UIKit -> VisionKitFig3ViewController.swift

class VisionKitFig3ViewController: BaseSampleViewController {
    override func viewDidAppear(_ animated: Bool) {
        super.viewDidAppear(animated)

        if VNDocumentCameraViewController.isSupported {
            let docScanner = VNDocumentCameraViewController()
            docScanner.delegate = self
            present(docScanner, animated: true)
        } else {
            print("Scanning is not supported.")
        }
    }
}
```

Here, we present the controller if we're able to (i.e., `isSupported` returns true) and assign ourselves as the delegate. From there – we're mostly detached from the whole process. We don't need to do anything until the user finishes up and VisionKit delivers us the results.

Again – processing those results is where all the work is. And really, it depends on what you're trying to do. If you want, you can simply store the resulting images. Or, more commonly, you want to process the text that was found in each of them, or add interface elements at the retrieved text coordinates. All of that is possible.

For our example, let's process any text we find and put it in a text view. We need to have a local `VNRecognizeTextRequest` around to do this, so we create one in the sample view controller. When the view loads – that's where we add all of our logic to handle results within its closure:

```
// Xcode -> UIKit -> VisionKitFig3ViewController.swift

private var textRequest = VNRecognizeTextRequest()

private func setupTextDectionRequest() {
    textRequest = VNRecognizeTextRequest(completionHandler:
{ (request, error) in
        if let e = error {
            print("Error getting results: \
(e.localizedDescription)")
            return
        }

        guard let results = request.results as?
[VNRecognizedTextObservation],
             !results.isEmpty else {
            print("There were no results")
            return
        }

        // Stitch all the results together and put it in the text
view
        var allResults = ""
        for scannedText in results {
            guard let foundText = scannedText.topCandidates(1).first
else { continue }
            allResults += foundText + "\n"
        }

        DispatchQueue.main.async {
            self.textView.text = allResults
        }
    })

    textRequest.recognitionLevel = .accurate
    textRequest.usesLanguageCorrection = true
}
```

Here, we're checking for any errors and then processing the results into one string. I also want to call out two specific properties we set on the request:

```
textRequest.recognitionLevel = .accurate
textRequest.usesLanguageCorrection = true
```

The first one gives us more accurate results – and that's perfectly fine here because we aren't processing a live video feed. Remember, this text recognition result is completely abstracted away from the document scanner – it has no knowledge of where the images its processing are coming from. The second one, again, affords us better results. If we were scanning live video, then we'd likely want to use less accurate values for both properties.

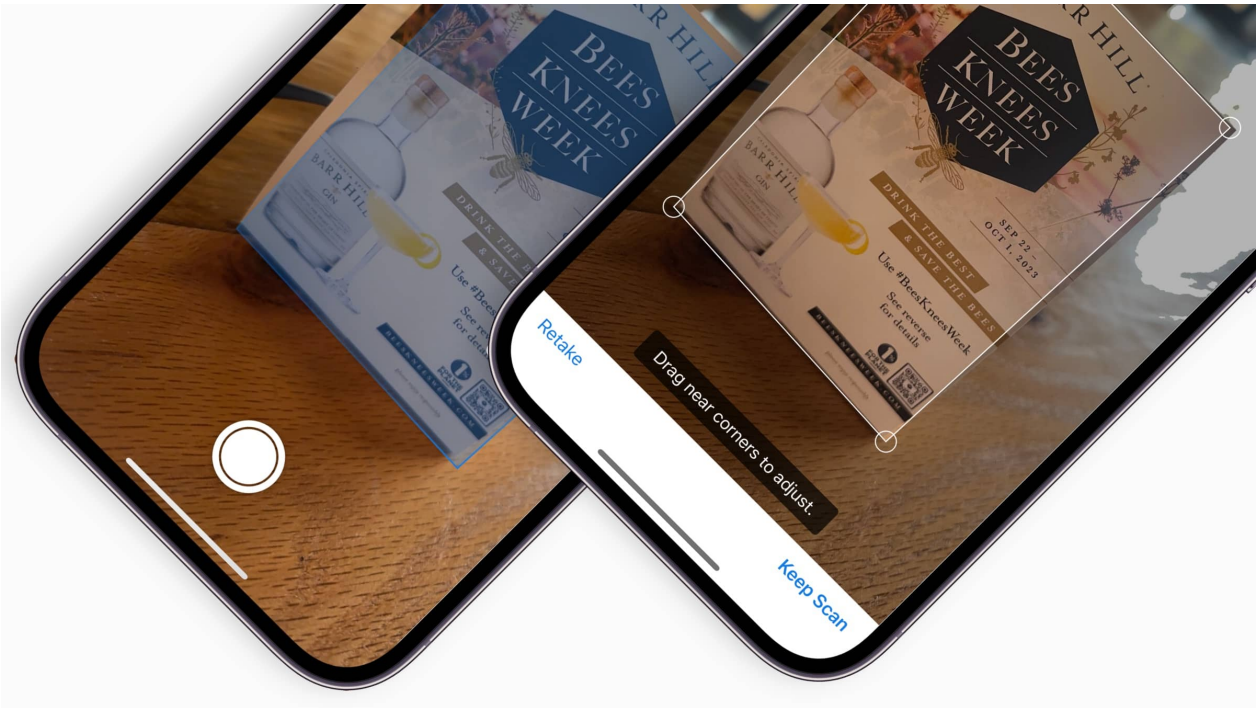
Finally, once we present the controller, we handle the results in the delegate function:

```
// Xcode -> UIKit -> VisionKitFig3ViewController.swift

extension VisionKitFig3ViewController:
VNDocumentCameraViewControllerDelegate {
    func documentCameraViewController(_ controller:
VNDocumentCameraViewController,
                                      didFinishWith scan:
VNDocumentCameraScan) {
        controller.dismiss(animated: true) {
            DispatchQueue.global(qos: .userInitiated).async {
                guard scan.pageCount > 0 else {
                    print("There were no pages returned.")
                    return
                }

                guard let cgImage = scan.imageOfPage(at: 0).cgImage
else {
                    print("Couldn't create Core Graphics image from
result.")
```

```
                    return
                }

                let request = VNImageRequestHandler(cgImage:
cgImage)

                do {
                    try request.perform([self.textRequest])
                } catch {
                    print(error)
                }
            }
        }
    }

    // See full code sample for error callbacks...
}
```

The scan, of type `VNDocumentCameraScan`, can have several pages of results. We're interested in the top one, so we ensure there's an image and then send it off to our text request we made from the Vision framework to process everything. If you run the sample code, you'll see the document scanner present right away.

Here's what it looks like scanning a pamphlet at the table I was at while writing this:



This is an interesting use case because there is a lot of text at different coordinate levels. There's not really a natural way to read it, but you'll notice the Vision processes it the best it can, top-to-bottom, and left-to-right for us. Here's what my first scan results came out like:



Very accurate, and only a few misfires. Now, if we were to use less accurate options – the results wouldn't be as great. But again, it's all about balancing your needs. And, that thought actually leads us into our first tip.

Tips

Find the Right Balance with Requests

The data scanner view controller and Vision requests offer a lot of setup options. This is a good thing – because the tasks they were created for are simply too varied to make any opinionated choices for you. So, experiment.

With the document scanner, try to start with accurate results by changing its quality level between `.balanced`, `.fast` and `.accurate`. You may be surprised at how much you can "get away with" using the most accurate configurations. iPhones are

powerhouse computers these days. But, don't assume you know — test it out on device every single time.

Further, try to stray away from generalizing your desired results and languages as much as possible. That is to say, try to look at the device locale and default to that language. Supply the types of data you're looking for. Unlike the advice in the above paragraph, tweaking these has lead to dramatic, noticeable results in my own testing.

Consider the Main Thread When Processing Results

Nearly all of these APIs run concurrently on a background thread. And, well, they should! Processing all of this data is expensive and takes a little bit of time. For that reason, you need to step back up to the main queue when you're done processing results.

You've seen this already in the code samples, for example:

```
DispatchQueue.main.async {
    self.textView.text = allResults
}
```

But, if you forget, the main thread checker in Xcode will remind you quickly. Shipping errors like this likely won't cause a crash, but it will stall your app for several seconds. Or worse, it'll hang indefinitely.

Three Key Takeaways

1. VisionKit allows you to work with images, videos and media assets in novel ways.
2. Look for ways to naturally include its features into your app, such as making text copy easier in images by supporting Live Text.
3. To go further for advanced use cases, consider looking into the Vision framework.

WidgetKit

WidgetKit has quickly become one of the most important frameworks on iOS. It started with iOS 14 and the ability to quickly add glanceable, useful information about your app on the Home Screen, Smart Stacks and the Today View:



If Apple had stopped there, they would still be incredibly impactful. But, they've done anything but stay the same. Now, we can find widgets on our Lock Screen, in StandBy mode, in Live Activities, watchOS complications and stacks, on the iPad Home Screen *and* Lock Screen and even macOS Sonoma supports your iOS app's widgets:



In short – they are significant, impactful, and people expect them to be supported in third-party apps. Now, with a framework that covers so much surface area, you may be expecting that there is a lot to know about WidgetKit.

And, well, you'd be right.

It's a big framework, true. I've said this phrase before, and I'll say it again – this is truly one of those frameworks which could constitute its own book. But, like most APIs, we can start with truly understanding the basics, and then slowly peel back how widgets work fundamentally. Stop me if you've read this already, but – we'll start with forming a foundation. And after that point, you can start to cast a wider net.

How it Works

Widgets are added via a new target, a WidgetKit extension. This means that all the usual rules apply — having your data or models in a framework or Swift package will help. You'll want to use shared app groups. All of that applies here.

Aside from being their own extension, though, widgets have a few core components to understand. The way they update is much different than what you're probably used to, they are built exclusively using SwiftUI, and they support many sizes, contexts, and features exclusive to each of those contexts too.

So, we need to cover the big-picture stuff. And, with widgets, that starts with keeping them updated. After all, the primary purpose of widgets is to show useful and *timely* information you can act on. What good would a weather widget be if it updated once a week?

Updates

WidgetKit renders the views they show in a separate process. While you don't specifically need to know the details of how they are archived, serialized and then de serialized — it helps to know that it happens. Because, if that *is* happening — then it stands to reason that iOS has an idea of the data it should show, and what it will look like, ahead of time. And, that's kind of how it works.

The gist of it is we have a budget that iOS gives us. Anytime we request an update to our widget, we start to eat into that budget. While how much we can update a widget isn't a hard and fast rule, we always want to do the least number of updates to be useful.

For some apps, that's once a day. For others — it could be once a minute. But again, the system will have the final say. Updates aren't guaranteed, and things like the widget's app being in the foreground, its last reload and the frequency that it's viewed

are all considered by the framework. Even further — iOS is constantly tuning update patterns based from the previous 24-hour usage window.

So, it's dynamic. But, there are times when we reload that it *doesn't* count against our update budge, and these instances are documented by Apple. They are when:

- The widget's containing app is in the foreground.
- The widget's containing app has an active audio or navigation session.
- The widget performs an app intent, such as when the user taps a button or toggles a switch.
- The widget performs an animation.
- The system locale changes.
- Dynamic Type or Accessibility settings change.

So that's the theory. In code, we have a *timeline* we provide. This timeline, powered by a `TimelineProvider`, vends entries — known as `TimelineEntry` objects. This timeline mechanism also vends a policy — and that tells WidgetKit how we'd ideally want to refresh our timeline once it hits the end.

So, if your timeline says that it should update for the next three hours, you need to say what happens after that. We've got three approaches to choose from:

- Request a new timeline after the last date passes. This is known as the `atEnd` policy.
- We could request a new timeline after a specific date we pass elapses, `.after(Date)`.
- Or, we could say that it doesn't need to be refreshed at all, `.never`. You might think this sounds odd — but there are several use cases for this, or

perhaps the containing app will manually refresh it based on a user's actions.

This part is typically the most challenging aspect to understand when getting started with WidgetKit. But, to put it all in a nice, digestible format – it goes a bit like this:

- You provide a timeline of how often to update.
- In that timeline, you pass in app data and use them within a widget's view.
- Then, at the end of the timeline – you start that process over.

I know this has been a bit of theory with not a lot of code, and I usually advise against teaching this way. But here, I think it's warranted. So, hang in there, read the rest of the core concepts here, and we'll put it in practice with a widget we'll create together.

The Interface

Next, we come to the interface itself. This part is easy – it's all in SwiftUI. Here's the stock one that comes with the WidgetKit extension:

```
struct WidgetExampleEntryView : View {
    var entry: Provider.Entry
    var body: some View {
        VStack {
            Text("Time:")
            Text(entry.date, style: .time)
            Text("Emoji:")
            Text(entry.emoji)
        }
    }
}
```

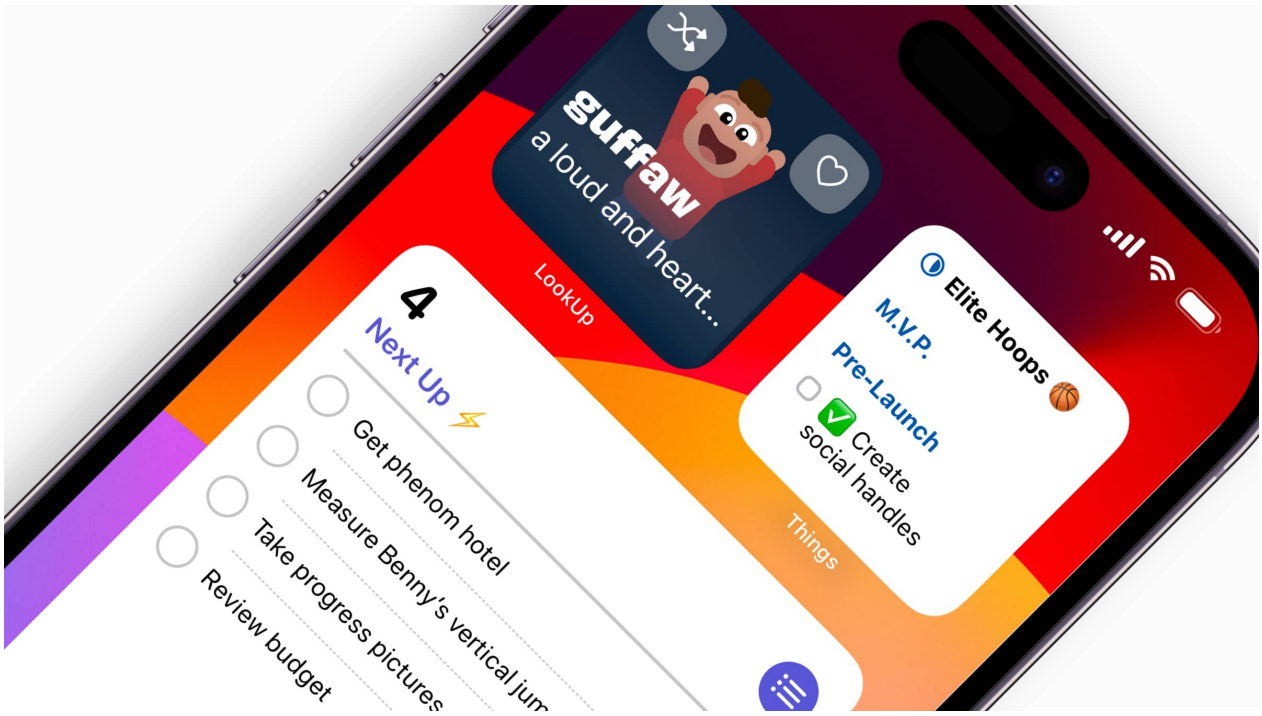
There's not much you *can't* do here that you normally can, but remember how widgets are fundamentally different from the views you build using SwiftUI for your apps.

Specifically, interactivity — only a button or toggle is supported. Lists, altogether, won't work.

So, if you run a widget and see a giant red X (seriously, this is what will show) you'll know you're using something that's not supported. Thankfully, though, widgets are more lively than they've ever been. If you're on iOS 17 or later, they have many smooth animations out of the box — and you can use most of the transition APIs, too.

Interactivity

Let's continue on that thread of interactivity for a moment. In iOS 17 and later, you can add interactivity to the mix. While widgets have always been able to launch your app and provide a link to it, now they can do more. Each of these buttons on these widgets operate on data, and then, using the timeline mechanism we discussed earlier, reload them while animating:



These button and toggle views are powered via App Intents. You pass in an existing intent to them, and they are run. If you need to brush up on how those work, refer to the App Shortcuts chapter. Later on, after we see a full code sample of a widget – I'll also include one that simply increments a counter to show how these work.

Sizes

We know that widgets are all over Apple platforms at this point. As such, there are several sizes they can appear as: small, medium, large, extra large, accessory circular, accessory rectangular, accessory corner and accessory inline. You can see which size your widget is currently being displayed at using an environment variable, too:

```
@Environment(\.widgetFamily) var family: WidgetFamily
```

Here's what each of those (aside from the accessory corner, which is exclusive to watchOS) looks like using Apple's Calendar widget. Please note that these aren't to exact scale, and any monospaced font is something I've added to note each widget's size:



With so much variety, you're probably wondering where all of these widget sizes can end up. While you don't need to commit anything to memory, here's a table for reference:

Size	iPhone	iPad	watchOS	macOS
Small	Home, Today, StandBy	Home, Today, Lock	N/A	Desktop, Notification
Medium	Home, Today	Home, Today	N/A	Desktop, Notification
Large	Home, Today	Home, Today	N/A	Desktop, Notification
Extra Large	N/A	Home, Today	N/A	Desktop, Notification
Acc. Circular	Lock Screen	Lock Screen	Stacks, Complications	N/A
Acc. Corner	N/A	N/A	Complications	N/A

Acc. Rectangular	Lock Screen	Lock Screen	Stacks, Complications	N/A
Acc. Inline	Lock Screen	Lock Screen	Complications	N/A

That's a lot of information. How widgets update, are built, their sizes, all the places they can be, interactivity and intelligence — I'm probably missing something here too. In fact, if you read Apple's article on how to get started with WidgetKit — here are the questions they propose you should ask yourself:

- Feature availability for each platform.
- Frameworks to use in addition to WidgetKit.
- Required appearances and available sizes for widgets, watch complications, and Live Activities.
- Technology that powers content updates.
- Animation.
- Interactivity with your app through deep links, buttons, and toggles.
- Configuration options for widgets and watch complications.
- Visibility in Smart Stacks.
- Functional constraints.

So, what do we do with all of this? Well, let's create a simple widget to feel out the process — and then go from there.

Our Widget

When developing widgets, we've seen that it helps to know all the parts in play and what they do. Thankfully, there are about four different things that, I think, are core to understand. Here are all the pieces we'll be using:

- The `Widget` struct. This represents a widget that our app provides. If we needed to vend several of them, we can use a `WidgetBundle`.
- The `View`. This is our actual widget user interface.
- A `TimelineEntry` type. This is basically the data model for a widget.
- A `TimelineProvider`. This supplies three different things: A placeholder `TimelineEntry` to show when the user is viewing widgets to add or when it's initially added. A snapshot of some entries, used to represent the current time and state of a widget. Finally, a timeline – the core piece of a widget's updating mechanism.

In a roundabout way, you can kind of sense a Model-View-Controller pattern. There is a view, a model from the entries and a controller, of sorts, with the timeline provider. Let's build a widget that shows a book chapter, which updates each hour.

I'm going to just put out all the code out there, and then we'll look at it in more detail. Our widget will display a random book from this series and a random chapter from it twenty-four times a day. Or, more simply – it'll refresh once per hour.

First, there' the `TimelineEntry`. Remember, this is our data model – and it must adopt the `TimelineEntry` protocol, which itself requires you to provide a `Date` property. This date will be used to know when to display the entry on the widget:

```
// Xcode -> SwiftUI -> WidgetExample.swift

struct ChapterEntry: TimelineEntry {
    let date: Date
    let book: Book
    let chapter: BookChapter

    static func randomData() -> (Book, BookChapter) {
        let randomBook = Book.allCases.randomElement()!
        let randomChapter = randomBook.chapters().randomElement()!
```

```
        return (randomBook, randomChapter)
    }

    static func placeholder() -> ChapterEntry {
        let (book, chapter) = ChapterEntry.randomData()
        return .init(date: .now, book: book, chapter: chapter)
    }
}
```

There, we can see two things: The date we must provide as part of the `TimelineEntry` protocol, and then our own app-specific data. I also like to provide a helper function to supply a snapshot or "demo data" version of the model to use later in our timeline provider.

Next, we need to set up a timeline. There, we'll adopt the `TimelineProvider` protocol and return our update frequency and models:

```
// Xcode -> SwiftUI -> WidgetExample.swift

struct Provider: TimelineProvider {
    func placeholder(in context: Context) -> ChapterEntry {
        ChapterEntry(date: Date(), book: .Accessibility, chapter:
.AXNotifications)
    }

    func getSnapshot(in context: Context,
                     completion: @escaping (ChapterEntry) -> ()) {
        let entry = ChapterEntry(date: Date(), book: .Accessibility,
chapter:
.AXNotifications)
        completion(entry)
    }

    func getTimeline(in context: Context,
                     completion: @escaping (Timeline<Entry>) -> ())
{
        var entries: [ChapterEntry] = []
```

```
        // One entry per hour, per day - then start over.
        let currentDate = Date()
        for hourOffset in 0 ..< 23 {
            let entryDate = Calendar.current.date(byAdding: .hour,
                                                  value: hourOffset,
                                                  to: currentDate)!

            let (book, chapter) = ChapterEntry.randomData()

            let entry = ChapterEntry(date: entryDate,
                                     book: book,
                                     chapter: chapter)
            entries.append(entry)
        }

        let timeline = Timeline(entries: entries,
                                policy: .atEnd)
        completion(timeline)
    }
}
```

The most important part I want you to notice here is the `getTimeline(in:completion)` function. This is what the widget will use to update, and here you can see we've supplied twenty-four entries, each with a random book and a random chapter from it:

```
// One entry per hour, per day - then start over.
let currentDate = Date()
for hourOffset in 0 ..< 23 {
    let entryDate = Calendar.current.date(byAdding: .hour,
                                          value: hourOffset,
                                          to: currentDate)!

    let (book, chapter) = ChapterEntry.randomData()

    let entry = ChapterEntry(date: entryDate,
                             book: book,
                             chapter: chapter)
    entries.append(entry)
```

```
}
```

From there, we wrap things up in a `Timeline` object which houses the entries and tells WidgetKit when we intend to recreate a new timeline of entries:

```
let timeline = Timeline(entries: entries,
                                  policy: .atEnd)
```

So, all we've done here is tell WidgetKit that we want the widget refreshed once an hour, per day – and then to recreate another timeline and get us more random data.

Next, we have the `Widget` instance itself. Here, we need to pass a few things. What the description should read in the widget gallery, its title, kind, the sizes we support (all of them by default, unless you say otherwise as we've done here), the provider it uses and the view. This fully describes the widget in its entirety.

Furthermore, here we've used a *static* configuration here, which means there are no additional options the user can change about the widget. It that weren't the case, for example – if you wanted to let the user *choose* which book and chapter to show, we'd need a dynamic configuration.

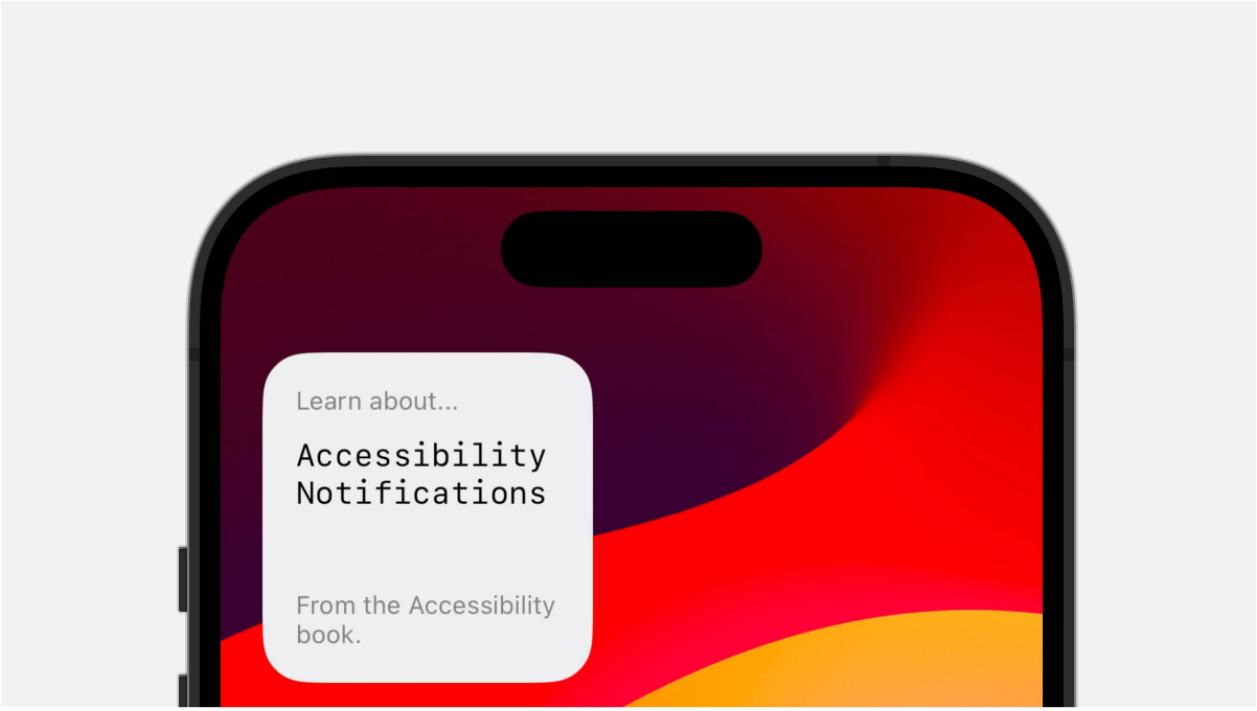
```
struct WidgetExample: Widget {
    let kind: String = "WidgetExample"
    var body: some WidgetConfiguration {
        StaticConfiguration(kind: kind,
                            provider: Provider()) { entry in
            WidgetExampleEntryView(entry: entry)
                .containerBackground(.fill.tertiary, for: .widget)
        }
        .supportedFamilies([.systemSmall])
        .configurationDisplayName("Chapter Widget")
        .description("Shows a random book and chapter from the Best-
in-Class
series.")
    }
```

```
}
```

Finally, we come to the easiest part – the view itself. This will be initialized with a timeline entry for us, so then we can turn around and use its data:

```
struct WidgetExampleEntryView : View {
    var entry: Provider.Entry

    var body: some View {
        VStack(alignment: .leading) {
            Text("Learn about...")
                .font(.caption)
                .foregroundStyle(Color(uiColor: .secondaryLabel))
                .padding(.bottom, 4)
            Text(entry.chapter.rawValue)
                .font(.system(.subheadline, design: .monospaced,
weight:
.semibold))
            Spacer()
            Text("From the \(entry.book.rawValue) book.")
                .font(.caption)
                .foregroundStyle(Color(uiColor: .secondaryLabel))
        }
    }
}
```

And with that, we have a simple – but fully functional – widget:



Interactive Widget

As I mentioned earlier, let's also show how an interactive widget would look. This example simply has a widget that shows a count from an `Int`. The count is stored away in user defaults (again – using shared App Groups) and it's incremented by the widget's button.

Before we continue, I don't have App Groups enabled for the sample project out of the box due to requiring team identifiers and the like. So, if you want to run this – enable those on the primary app target and the widget extension target first, so you'll have access to shared user defaults.

Now, to the code sample. There are two parts. First, we have an App Intent, a light-weight model and a view to show the current count *in the main app target*:

```
// Xcode -> SwiftUI -> InteractiveWidgetContentView.swift

import Foundation
import AppIntents
```

```
import SwiftUI

class Counter {
    private static let sharedDefaults: UserDefaults =
UserDefaults(suiteName: "CHANGE_TO_YOUR_APP_GROUP_NAME")!

    static func incrementCount() {
        var count = sharedDefaults.integer(forKey: "count")
        count += 1
        sharedDefaults.set(count, forKey: "count")
    }

    static func currentCount() -> Int {
        sharedDefaults.integer(forKey: "count")
    }
}

struct ExampleIntent: AppIntent {
    static var title: LocalizedStringResource = "Increment Count"
    static var description = IntentDescription("Increments a shared
count with the main app.")

    func perform() async throws -> some IntentResult {
        Counter.incrementCount()
        return .result()
    }
}

struct InteractiveWidgetContentView: View {
    @Environment(\.scenePhase) private var phase
    @State private var count: Int = 0

    var body: some View {
        VStack {
            Text("Count: \(count)")
        }
        .padding()
        .onChange(of: phase) {
            count = Counter.currentCount()
```

```
        }
    }
}
```

And then, in the widget extension:

```
// Xcode -> SwiftUI -> InteractiveWidgetExample.swift

import Foundation
import WidgetKit
import SwiftUI
import AppIntents

struct Provider: TimelineProvider {
    func placeholder(in context: Context) -> SimpleEntry {
        SimpleEntry(date: Date(), count: "\
(Counter.currentCount())")
    }

    func getSnapshot(in context: Context, completion: @escaping
(SimpleEntry) -> ()) {
        completion(SimpleEntry(date: Date(), count: "\
(Counter.currentCount())"))
    }

    func getTimeline(in context: Context, completion: @escaping
(Timeline<Entry>) -> ()) {
        let timeline = Timeline(entries: [SimpleEntry(date: Date(),
                                                      count: "\
(Counter.currentCount())")],
                                policy: .never)
        completion(timeline)
    }
}

struct SimpleEntry: TimelineEntry {
    let date: Date
    let count: String
}
```

```
struct WidgetsEntryView : View {
    var entry: Provider.Entry
    var body: some View {
        VStack {
            Text("Count:")
            Text(entry.count)
            Button(intent: ExampleIntent()) {
                Text("Increment Count")
            }
        }
        .containerBackground(.fill.tertiary, for: .widget)
    }
}

struct Widgets: Widget {
    let kind: String = "Widgets"
    var body: some WidgetConfiguration {
        StaticConfiguration(kind: kind, provider: Provider())
{ entry in
            WidgetsEntryView(entry: entry)
        }
        .configurationDisplayName("Count Widget")
        .description("An interactive widget incrementing counts.")
    }
}
```

Notice how everything fits together. We have our timeline provider vending entries with a count and a date. A timeline that reloads when the intent fires with a reload policy of `.never` since we'll either reload it within the app, or it'll occur when the button is tapped.

The App Intent, `ExampleIntent`, is used both in the app target and the widget – something I alluded to in an earlier chapter over App Shortcuts. While this is a simple example, you can see all the pieces fitting together.

What's Next?

Armed with a simple example with static data and an interactive one – you can start to see a foundation forming. And, that's key because WidgetKit may be one of the largest frameworks the platform has.

So, now, you should try to expand on that. What happens if you wanted to provide a bigger widget for any of these, for example? We could use the aforementioned environment key to provide different views based on that. But, what if the user isn't signed in to view data?

These are all questions you'll be faced with in production scenarios, but they are all feasible due to the flexible API design. For example, here's what one might look like in an app which has to answer those types of questions. I've adapted this code based on a widget I developed which has been used on over 100,000 devices:

```
struct UpcomingWeatherView : View {
    @Environment(\.widgetFamily) var family: WidgetFamily
    let hasPasswordSet = WidgetData.hasPasswordSet
    var entry: UpcomingWeatherTimelineProvider.Entry

    @ViewBuilder
    var body: some View {
        if (!hasPasswordSet && !entry.isShowingInPreview) {
            SignInView()
                .padding()
        } else if entry.forecast.isEmpty {
            NoForecastFoundView()
                .padding()
        } else {
            switch family {
            case .systemSmall:
                UpcomingWeatherSmallWidgetView(entry: entry)
            case .systemMedium:
                UpcomingWeatherMediumWidgetView(entry: entry)
            case .systemLarge:
                UpcomingWeatherMediumWidgetView(entry: entry)
```

```
            default:
                UpcomingWeatherMediumWidgetView(entry: entry)
            }
        }
    }
}
```

Those that are more complex, now you have the know-how to follow it. As we said at the jump – start small, then go deep and wide.

Tips

Objective-C Shims

If you need to access `WidgetCenter` in your app, and you're using an Objective-C codebase – you're going to run into a wall quickly. Widget center APIs are *only* available in Swift. As such, you'll need to make a Swift shim to bridge over any functionality.

Here's an example which exposes `WidgetCenter` and its function to request a reload of all widget timelines:

```
@objc
class SwiftShims:NSObject {
    @objc
    class func reloadAllWidgets() {
        WidgetCenter
        .shared
        .reloadAllTimelines()
    }
 }
```

Then, in Objective-C – you're free to invoke it:

```
#import "AppName-Swift.h"
```

```
// Later on...
[SwiftShims reloadAllWidgets];
```

And that works just fine. However, speaking of reloading widgets – just reloading them all wholesale isn't always the best idea. Remember, we've got a budget we're allowed, and we want to stay within it – or our reloads run the risk of being ignored.

So, to that end – consider reloading just the ones you need. We'll look at how to do that in the next tip.

Selectively Reload Widgets

If you've got several widgets on offer, and you need to reload them from your app due to data changes, user actions or something similar – consider loading *just* the one you require. As we saw above, it's easy to just call `reloadAllTimelines()` but that's not always wise, and it can deplete reload budgets.

Instead, just pass in the widget's kind to the widget center to request a reload. This is much more budget friendly, and the more we are a good citizen of the API, the more likely we are to be granted future reloads:

```
func reloadWidgetOf(type: String) {
    WidgetCenter.shared.getCurrentConfigurations { result in
        guard case .success(let widgets) = result else {
            print("No widgets were found.")
            return
        }
        if let match = widgets.first(where: { $0.kind == type }) {
            WidgetCenter.shared.reloadTimelines(ofKind: match.kind)
        }
    }
}
```

Take an Iterative Approach to Widgets

Since widgets can end up in so many places, the inclination is to be everything all at once. If you start on a single widget – it needs to support all sizes, be in all places, and work for every single use case that the system uses a widget.

In my opinion, that's the wrong approach. Start small, get that experience right – and then start crafting something more. Here's what I ask myself when I start on a WidgetKit extension:

1. What quick, useful information can I surface?
2. Of that information, should it be manipulated (i.e., should this be an interactive widget)?
3. What sizes make the most sense?

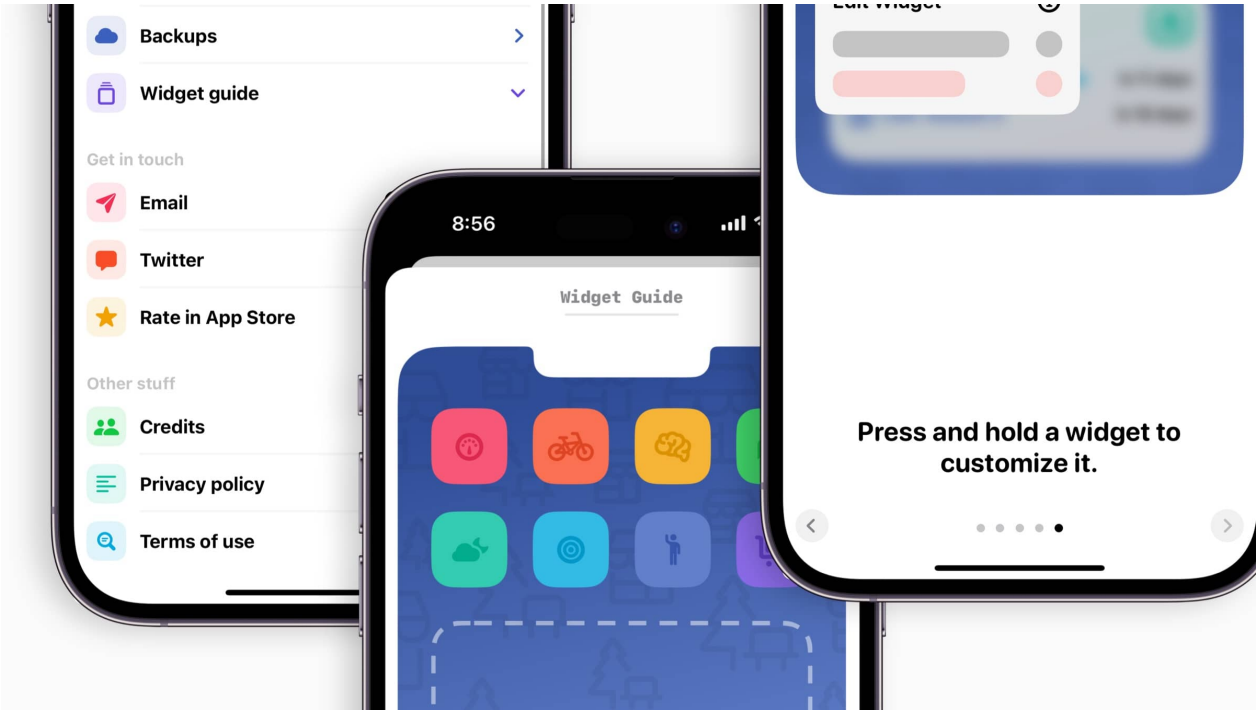
And that's it – I stop there. Because from those three questions, I can come up with a compelling M.V.P., if you will, of a widget. And later, I'll expand on it. Perhaps give it more size support, and with that more (or less!) information. Maybe I'll make it configurable or add more options.

But, start small. *Always* start small with widgets. And, this is a credit to their API, once you do – it's really not too much work to bring it to more places across iOS that support or use WidgetKit, like StandBy mode.

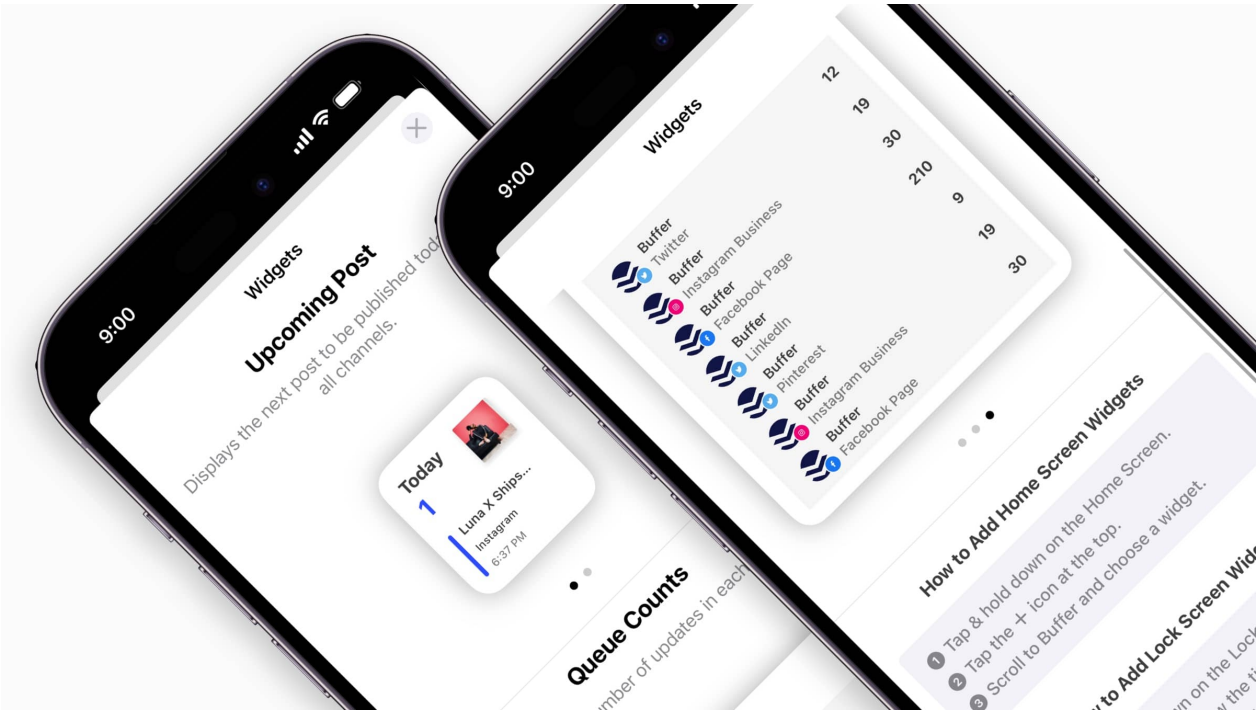
Widget Tips and Callouts

Unfortunately, even though widgets have reached a massive audience, and I'd consider them mainstream among iOS' user base, they still can be tricky to set up and configure. For that reason, I think it's a good idea to provide some sort of call out, tool tip or something similar to let people know that your app has widgets, and even how to add them.

Here's a fun example from Up Ahead:



Or, in Buffer – it shows the available widgets (much like the gallery does on iOS) and simply lists the steps you need to take to add them:



Increasing Visibility in Smart Stacks

You can donate intents to the system to let iOS know when a particular widget may be best suited to show in a Smart Stack. However, there are three different ways to do this – all best suited for different situations. I'll explain each of them here, and then you can decide which one makes the most sense for your situation:

Donate Intents via Intents Manager

In this case, we are manually donating an intent which powers a widget to the `IntentDonationManager`:

```
func donateIntent() async {
    let manager = IntentDonationManager.shared
    let intent = ExampleIntent()

    do {
        let _ = try await manager.donate(intent: intent)
    } catch {
```

```
        print(error.localizedDescription)
    }
}
```

Using this method, you can donate the intent as you see fit while users navigate and interact with your app. However, using this method – you are extending your app intent's fingerprint beyond *just* Smart Stacks. Using this method, the system might surface it in Spotlight search, suggest it as a shortcut on the Lock Screen, the Siri watch face and more.

Another note here – you can skip the `IntentDonationManager` altogether and simply call `donate()` on the `AppIntent` itself:

```
func donateIntent() async {
    let intent = ExampleIntent()

    do {
        let _ = try await intent.donate()
    } catch {
        print(error.localizedDescription)
    }
}
```

Anytime someone is using the same intent over and over, and that intent also ties into a widget – you should donate it. Aside from simply extending that intent across the system to other places, it also increases discoverability. If the user has the widget the intent is used by in a Smart Stack, then it's more likely to be show on top. If they don't have the widget in a Smart Stack – then the system will likely suggest it. Win-win.

Relevant Intents

This next method does the opposite – you suggest intents that are likely to be relevant in the *future* instead of *now*. If you are developing a sports app, and the user has

a favorite team who is about to play later that day – then this is the ideal API to use to suggest your widget which displays the team's score when they are playing:

```
func donateIntent() async {
    // The intent that'll be relevant later today
    let intent = FavoriteTeamScoreIntent()

    // Construct the game's start and end time
    // Keep in mind - this code would be firing
    // Before the game, sometime in the morning
    let now = Date()
    var calendar = Calendar.current
    calendar.timeZone = TimeZone.current
    var dateComponents =
calendar.dateComponents([.year, .month, .day], from: now)
    dateComponents.hour = 19
    dateComponents.minute = 0
    let startTime = calendar.date(from: dateComponents)!
    dateComponents.hour = 21
    let endTime = calendar.date(from: dateComponents)!

    // Create a relevant intent from our game intent
    let gameIntent = RelevantIntent(intent,
                                    widgetKind: "WidgetExample",
                                    relevance: .date(from:
startTime, to: endTime))

    // Finally, update the relevant intent
    let manager = RelevantIntentManager.shared

    do {
        let _ = try await
manager.updateRelevantIntents([gameIntent])
    } catch {
        print(error.localizedDescription)
    }
}
```

Using `relevance` on `TimelineEntry` instances

Finally, in the same vein as donating a relevant intent, you can also use a scoring system that you specify to accomplish the same thing. Using `TimelineEntryRelevance` – you can tell the system how important a specific widget is now or in the future.

It works like – you setup an absolute scoring system, it could be 1-10, 1-100, whatever – and you assign a score depending on how relevant that timeline entry is for the widget it represents. Again, consider a sports app. Based on if the team is playing today, this week or if they don't play at all, we could assign an appropriate score for the entry:

```
struct FavoriteTeamEntry: TimelineEntry {
    let date: Date
    let status: GameStatus
    let team: Team
    var relevance: TimelineEntryRelevance? {
        switch status {
        case .isGameday:
            return .init(score: 10)
        case .isGameWeek:
            return .init(score: 5)
        case .isByeWeek:
            return .init(score: 0)
        }
    }
```

Three Key Takeaways

1. Use widgets to surface timely information from your app.
2. Offer useful, isolated tasks as interactive widgets.
3. Not all widgets need to support every size, always consider the design and use case first.

URL Schemes

Creating custom URL schemes opens up the ability to reference data and resources within apps, either from *your* app or (most commonly) allowing other apps to. In some ways, I've found that it helps to think of custom URL schemes as an extension point.

When someone wants to open, say, Twitter from their app and show a particular tweet – they do so via a custom URL scheme. This means we aren't required to add a Twitter S.D.K. to our project, or any extra third-party code. Instead, Twitter offers a public URL scheme that's documented. Twitter's own URL scheme looks like this: `twitter://`.

From there, developers simply append different parameters and paths to do all sorts of things. These links work similar to how web links work. Parameters begin with a `?` and more added using `&`. For example, we could open a certain tweet within the Twitter app from our apps. The URL would be constructed as follows:

```
twitter://status?id=[Tweet ID]
```

And then, using `UIApplication`'s `openURL` function, developers request to open the URL:

```
let bicAnnounceTweet = URL(string: "twitter://status?
id=1397894126432030722")!

UIApplication.shared.open(bicAnnounceTweet)
```

If a third-party app executed this code, and Twitter is installed on the device – it would open up and display that tweet from my profile. Twitter's custom URL schemes allow us to do this kind of thing.

Now, both Twitter and other apps can deep link straight into a particular tweet using Twitter's app. Thus, you can see what I meant by starting the chapter like this:

> *Creating custom URL schemes opens up the ability to reference data and resources within apps, either from your app or other ones.*

How it Works

Let's peel back how custom URL schemes work. They aren't that complicated, but there are some odd things you can run into if you haven't had much experience with them. Plus, you may have heard the terms "deep linking", "URL schemes" and "custom URL schemes" all used interchangeably. Here's what that all really means:

1. Apps register custom URL schemes within Xcode.
2. They specify a role, either an editor or viewer. If you are creating the custom scheme, you are the editor. Otherwise, consuming existing schemes means that you are the viewer.

3. Specify an identifier, but really you should simply provide a reverse DNS string, as you typically would for iOS-based identifiers. Include your company's domain and the app's name in it, so `com.bestinclassiosapp.codesamples` might be one I'd use for the demo project. You'll notice that this process is very similar to how we register custom uniform type identifiers – they work much the same way.
4. From there, iOS will use URL opening and handling APIs. Which one you need depends on the type of app you've made.
    4.1 Do you still use the app delegate lifecycle? Then, the URL functions on an app delegate will be called.
    4.2 Are you using scenes and UIKit? Here, scene-based URL functions are used.
    4.3 Finally, in a SwiftUI-based app, we can use the URL open modifiers.

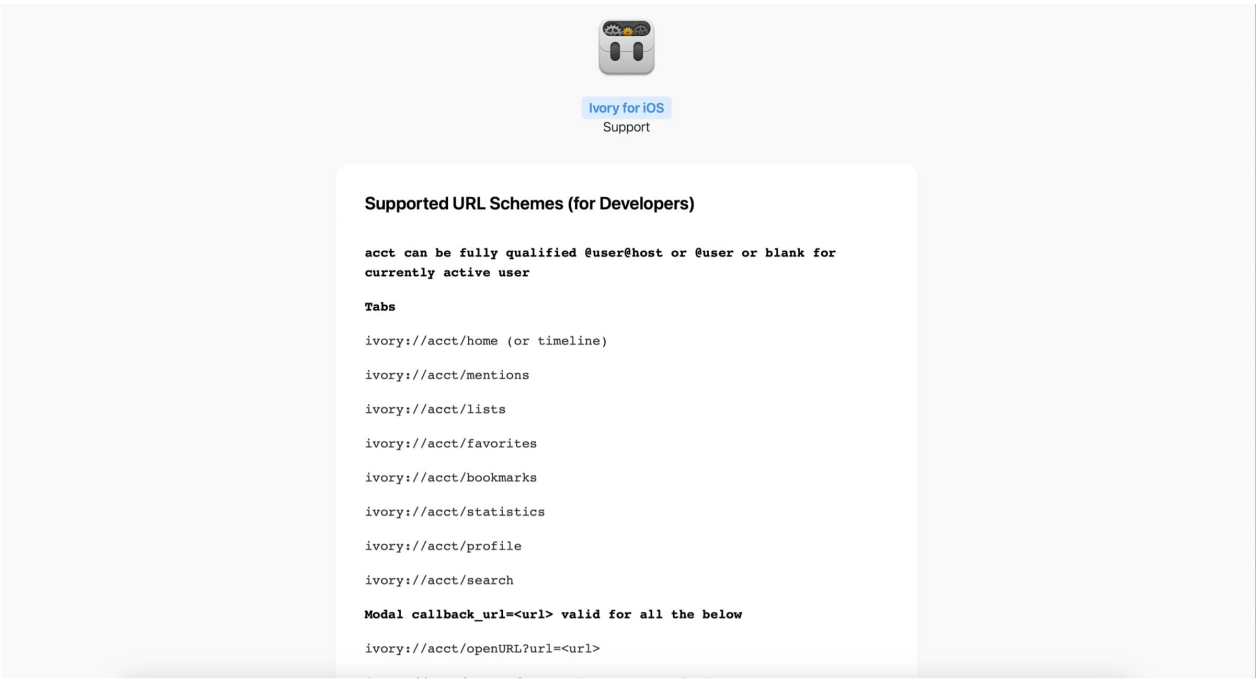
Now, given that – you may be wondering how our example with Twitter works if we haven't said that we are a "viewer" of the `twitter://` URL scheme. The reason is that you only need to register custom schemes in Xcode if you're trying to *handle* them yourself in your app. Here, we aren't doing that – we simply wish to open the URL and iOS takes care of routing it to the right place for us (i.e., Twitter's app).

Another one of those "gotcha" moments is how iOS disambiguates these URLs. For example, what happens if several apps register the same scheme, and they all say that they are the "editor" of it? Well, that's one reason why we need to supply an identifier.

This helps iOS identify each of them uniquely. But, unfortunately – sometimes the resulting behavior can be undefined. We have no API or mechanism in place to "rank"

higher than another app who might register the same one, or otherwise augment what iOS will do in such cases.

There are very few instances where some apps will attempt to "highjack" custom URL schemes so that their app will open them instead. Again, there's not much we can do about it. Generally, that's not an issue. In fact, most apps will publicly document their custom URL schemes so that other apps can use them:



Apple documents several of the custom schemes they support. Apps who document their schemes *want* other apps to use them! It just helps make their own apps more useful and *used* by more people. Here are some common deep links powered by custom URL schemes you might use from Apple's stock apps:

Compose an email to a certain address

```
mailto:[email address]
```

Compose an email with a subject and message

```
mailto:[email address?&subject=[subject]&body=[body]
```

Call a number

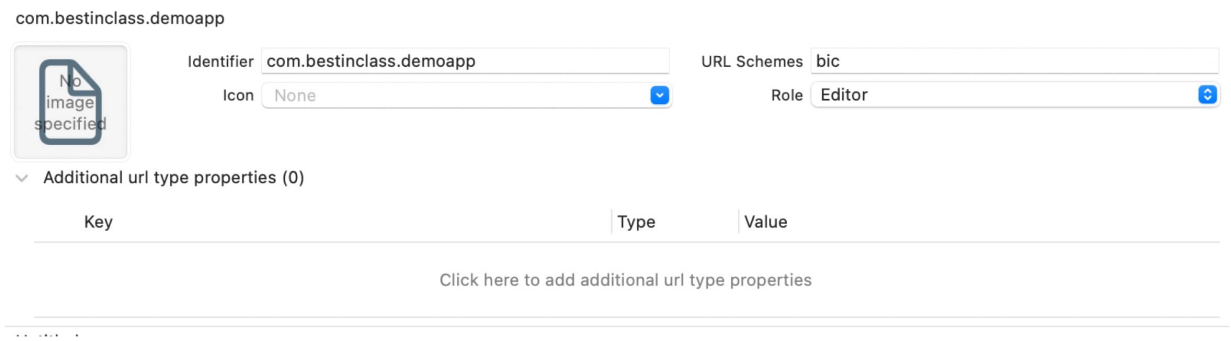
```
tel:[the phone number]
```

Open Messages for a certain number

```
sms:[the phone number]
```

So, if you supply your custom URL schemes for public consumption, I'd recommend documenting them on the app's website. Or, consider even adding information about them in your app.

Let's see how this all works. We can set up a custom URL scheme of our own. Ours will simply open a chapter from the book. First, we've got to register the URL scheme:



From there, we can construct a URL and call `openURL` to invoke it. In the following sample, the top section will open the Twitter URL we started the chapter talking about. Below it, you can select any chapter in the book, and we'll construct a URL to handle ourselves:

```
// Xcode -> SwiftUI -> URLSchemesFig1View.swift

struct URLSchemesFig1View: View {
```

```
    private let bicAnnounceTweet = URL(string: "twitter://status?
id=1397894126432030722")!
    @State private var chapter: BookChapter = .AXNotifications

    var body: some View {
        Form {
            Section("Opening an external custom URL Scheme") {
                Button("Open Best-in-Class announcement Tweet") {
                    UIApplication.shared.open(bicAnnounceTweet)
                }
            }
            Section("Opening our own custom URL Scheme") {
                Button("Open Selected Chapter") {
                    let chapterURL = URL(string: "bic://open?
chapter=\(chapter.rawValue)")!
                    UIApplication.shared.open(chapterURL)
                }
                Picker("Chapter to open:", selection: $chapter) {
                    ForEach(BookChapter.allCases) { c in
                        Text(c.rawValue)
                    }
                }
                .pickerStyle(.navigationLink)
            }
        }
    }
}
```

If you choose a book chapter and try to open it, our scene delegate will be supplied the fully qualified URL, and we can operate on it. If we set a breakpoint there, we could see the URL and any options iOS provides to us:

```
Printing description of firstURL:
<UIOpenURLContext: 0x280c2a9a0; URL: bic://open?
chapter=Accessibility%20Notifications; options:
<UISceneOpenURLOptions: 0x28199e940; sourceApp:
```

```
com.bicBookSeries2023.A-Best-in-Class-Tour; annotation: (null);
openInPlace: NO; _eventAttribution: (null)>>
```

From then, we just open it. I've created a function which returns a `Bool` to indicate we handled it, and false otherwise:

```
// Xcode -> SceneDelegate.swift

private func openURLContextIfPossible(_ context: UIOpenURLContext)
-> Bool {
    // Custom URL Schemes
    let requestingApp = context.options.sourceApplication ?? "N/A"
    let url = context.url

    // Get the path and parameters
    if let components = NSURLComponents(url: url,
                                        resolvingAgainstBaseURL:
true),
       let parameters = components.queryItems  {

        if let selectedChapter = parameters.first(where: { $0.name
== "chapter"
})?.value,
           let model = BookChapter(rawValue: selectedChapter) {
            print("Opening chaper from custom URL scheme. Sent from
\(requestingApp)")
            guard let rootController = window?.rootViewController
as?
CustomNavController,
                  let glossaryVC =
rootController.viewControllers.first as?
GlossaryViewController else {
                print("Unable to open book chapter. Unexpected view
hierarchy.")
                return false
            }

            rootController.popToRootViewController(animated: false)
```

```
            glossaryVC.pushBookChapter(chapter: model,
with: .SwiftUISample)
            return true
        }
    }

    return false
}
```

The `UIOpenURLContext` houses the URL itself, options and more. The reason we need to see if we're handling a custom URL scheme, though, is in our case – we already used that function previously to handle AirDrop. So, now we can check if we're handling a custom URL scheme, or handling AirDrop payloads:

```
// Xcode -> SceneDelegate.swift

func scene(_ scene: UIScene, openURLContexts URLContexts:
Set<UIOpenURLContext>) {
    guard let firstURL = URLContexts.first else {
        return
    }

    // Custom URL Schemes handling if needed
    guard !openURLContextIfPossible(firstURL) else {
        return
    }

    // AirDrop
    do {
        guard firstURL.url.startAccessingSecurityScopedResource()
else {
            return
        }

        let bookData = try Data(contentsOf: firstURL.url)
        let decodedBook = try
JSONDecoder().decode(BookChapterContainer.self, from:
```

```
bookData)

        guard let rootController = window?.rootViewController as?
GlossaryViewController else {
            return
        }

        rootController.pushBookChapter(chapter:
decodedBook.bookChapter, with:
.SwiftUISample)
    } catch {
        print("Error decoding AirDrop model: \
(error.localizedDescription)")
    }
}
```

Custom URL schemes are a great way to offer functionality from your app. Anytime you need to deep link into, or from, your app – this is the right tool to use.

Tips

Consider Attack Vectors

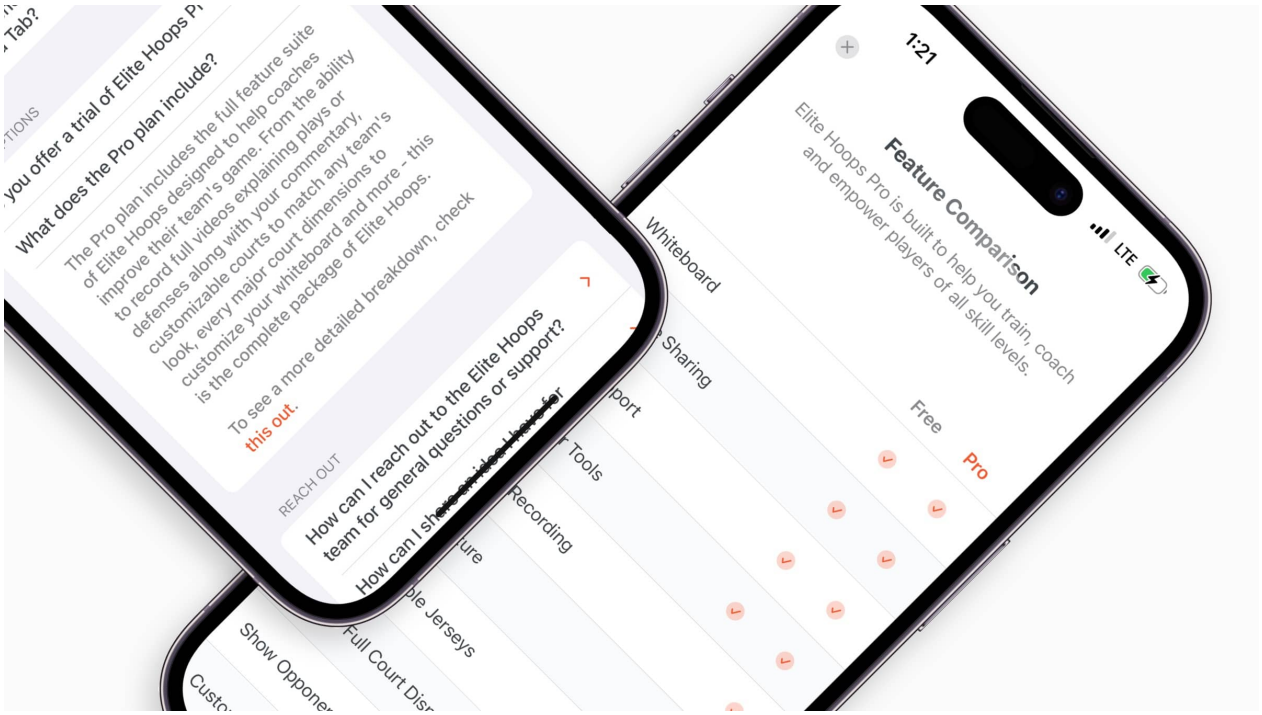
A URL scheme could be used as an attack vector in your app. It's important to make sure every single URL you're handling isn't malformed, and you should always validate the parameters they have. Further, foolproof URL handling to make sure that even if someone were to access undocumented custom URL schemes – that they couldn't do damage or access user data.

That means creating custom URL schemes to delete or remove an account, for example, isn't advised. When you're creating custom URL schemes, just assume that everyone could potentially be calling them.

Deep Links and Markdown

Using SwiftUI's `Text` view, in tandem with an `AttributedString` – you can write Markdown-based text. This is a great time to use custom URL schemes, this way – you can provide inline links in the text to handle. I do this in my app, where I show a `Text` control using the attributed string and its initializer, which takes in a `markdown` parameter.

Notice the orange words on the left, and the right is what presents when it's tapped:



To accomplish this, I use custom URL schemes. I have one already setup in my app, and a small utility class to handle them:

```
enum DeepLinks: String, Identifiable {
    case featureComparison = "featureComparison", paywall =
"paywall"
```

```
    enum Action: String, Identifiable {
        case open = "open"
        var id: Self { self }
    }

    var id: Self { self }
}

class DeepLinker: ObservableObject {
    @Published var linkedView: DeepLinks? = nil

    func destinationFor(queryItems: [URLQueryItem]) -> DeepLinks? {
        guard !queryItems.isEmpty else { return nil }

        for item in queryItems {
            if item.name == DeepLinks.Action.open.rawValue {
                if item.value ==
DeepLinks.featureComparison.rawValue {
                    return .featureComparison
                } else if item.value == DeepLinks.paywall.rawValue {
                    return .paywall
                }
            }
        }

        return nil
    }
}
```

I use this at the `App` level – where I look for incoming links and handle them accordingly. I call this function from the `openURL` modifier:

```
private func handle(_ url: URL) {
    guard let components = URLComponents(url: url,
resolvingAgainstBaseURL: true),
        let params = components.queryItems,
        let destination = deepLinker.destinationFor(queryItems:
params) else {
```

```
            Logs.deepLinks.warning("Couldn't parse incoming URL: \
(url)")
            return
        }

        let stringURL = url.absoluteString
        Logs.deepLinks.info("Handling \(stringURL)")
        deepLinker.linkedView = destination
}
```

If I was able to parse one out, setting the `linkedView` property will trigger a sheet to show:

```
.fullScreenCover(item: $deepLinker.linkedView, content: { deepLink
in
    switch deepLink {
    case .paywall:
        PaywallView()
    case .featureComparison:
        WhatsIncludedView()
    }
})
```

Universal Links versus Deep Links

Custom URL schemes are a great way to perform deep links. But, that's *all* they were meant for. If you're trying to do this and more – look to Universal Links. These are created by you, and via associated domains, it can reference your app's website and your app at the same time. Plus, they are standard HTTP or HTTPS links, making them easy to use on the web and in our apps.

Think of them as custom URL schemes with superpowers. A link from your website could deep link directly into your app, or vice versa. Your app's website, and its con-

tent, can be used if your app isn't installed. There's SEO benefits, App Clip support and more.

Three Key Takeaways

1. Custom URL schemes allow you to deep link to parts of your app.
2. You can create your own URLs, or consume publicly documented custom URL schemes to open other apps.
3. Be sure to consider security implications, and always validate the URLs you receive.

Always Onward

Congratulations on finishing the largest and most technical book from the Best-in-Class iOS App series! The world of iOS frameworks and APIs is ever-expanding, and that's not likely to change anytime soon. As that technological world continues to expand, I expect this book to grow along with it.

As I look back on writing this series, I can't help but think of one word: *curiosity*. I think that's an incredibly undervalued trait of any designer or software developer. What else could keep us coming back, yearning to learn more and more, if not curiosity?

This industry is unique in that we are constantly learning, and relearning, concepts all the time. If you are a plumber, I'd venture to say that the tools and concepts you've learned do not drastically change over the course of your career. Sure, advancements happen, new ways of doing old jobs emerge — but fundamentally, your craft and its

skills are hardened and sharpened. Likely, brand new, industry-changing, seismic shifts don't occur.

Not so with us.

SwiftUI is but a nascent framework in comparison to UIKit – and the same is true of UIKit when compared to AppKit! That's a seismic, industry-changing shift. And so it is, we all start learning from scratch again. But, we keep coming back, and we keep learning. Because we stay *curious*.

As your curiosity and excitement follow along with each release of iOS, I hope you'll come back to this book again and again. No doubt, I'll be fulfilling my curiosity about the world of iOS within these pages. And I can't wait to keep doing so for the rest of my career, as long as I'm able to.

Keep learning, keep going – always onward.

Thanks for reading.

The Journal: Quick Tips from over a Decade of iOS Development

I remember learning about Xcode's schemes at my first job straight out of college. I had no idea what they were when I started – or why I might even make my own or otherwise change them. But, with an experienced developer there to help me out, I picked up a valuable skill that day. We setup a custom scheme that saved me time, but perhaps more importantly – I learned a new skill that day. One that I still use all the time, over a decade later.

That's not an earth shattering or inspirational story, but what it *is*, however, is realistic. This kind of thing happens everyday. Developers, both experienced and brand new, learn some tips of the trade from each other all the time. And really, there's nothing quite like it. That moment of, "Ohhhhhh!", when realizing a new flow can save you tons of time, or maybe learning about a different API or whatever it is, feels fantastic.

That's what I want this little journal should be. And, that moment where I learned about schemes – those moments are what I want for everybody to have while reading this. So, inside here, you'll find very short journal entries. Every entry is a tip I want you to know about, and all of them are presented in a 1,000 words or less. Hardly any pictures, no huge code samples – just simple, actionable stuff.

Read them while you're waiting on a bus. Or, maybe you have a five minute break throughout the day. Whatever it is. Technical reading can be dense (I would know, I've

written an entire book series) but this journal should be the opposite. Short, concerted and digestible.

If you've got the time, I've got some tips. Maybe you know some of them, or maybe (hopefully!) you'll learn a few things along the way. Either way, thanks for reading!

Environment Variables and Launch Arguments

Xcode's schemes represent a configuration, of sorts, of your app's target. To understand them, it helps to consider the world of video games, of all things. These days, a game seemingly can't be released without offering several editions. We've got the always-present collector's edition, the standard release and typically countless more permutations.

All different versions of a just about the same thing – the actual video game.

Schemes work the same way. We have our app, but can run it under different conditions. And, to augment these conditions – we can use launch arguments and environment variables. Want to double the length of all of your strings to test pesky layouts caused by more verbose languages? Or, how about you turn on all of your app's paid features without changing your code?

All that and more is trivial with launch arguments and environment variables.

To achieve this, we edit the current scheme – where we can begin to edit both launch arguments and environment variables:

```
Xcode -> [Click on the target] -> Edit Scheme...
```

From there, we'll select:

```
Run -> Arguments
```

At this point, we've got two choices:

- Add, edit or remove **Launch Arguments**
- Add, edit or remove **Environment Variables**

What is the difference between the two? Functionally, they can be used for very similar purposes. In practice…

- Launch arguments aren't created in Xcode with a key-value pair. Environment variables are.
- Launch arguments begin with a dash, "-". Environment variables can be named however you see fit.
- Launch arguments override values, or add them altogether, in `UserDefaults` for the duration of the current process. Once it's ended, it's removed. Environment variables are accessed with a strongly typed dictionary, with both string types for the key and their value.

Here's an example of a *launch argument*:

```
-NSDoubleLocalizedString YES
```

More precisely, the correct syntax here:

1. Begin with a dash, "-"
2. Then, the name, "NSDoubleLocalizedString"
3. A space.

4. Then, its value, "YES".

Which gives us our launch argument, `-NSDoubleLocalizedString YES`. I should note, however, this is *mostly* the case. There are some launch arguments out there which don't require a value.

On the other hand, an *environment variable* would have a name, perhaps "TZ", and a value, "UTC". Within Xcode, there are specific fields for both of these – so creating them is a bit easier and less error-prone. Each of the things we set up here are specific to the *scheme* we've added them to, which makes them extremely useful to quickly test very specific scenarios.

A quick note here – SwiftUI has environment variables too, so the naming here can be a little ambiguous. In SwiftUI, if you mention environment variables – you're talking about values propagated throughout a SwiftUI app by the framework:

```
@Environment("the_var") private var name: Type
```

When we talk about environment variables outside of a SwiftUI context, we're referring to the flags we set up in Xcode for our schemes. So, they have the same name, but different purposes.

Launch Arguments

Any of our launch arguments can be accessed via `UserDefaults`. For example, if added one called `-EnablePro YES`, I could find it with the key `EnablePro` at runtime:

```
let arg = UserDefaults.standard["EnablePro"]
```

In fact, you may be surprised to see how many launch arguments Apple is using whether you've enabled them or not. You can dump all the defaults quite easily to see them:

```
let defaults = UserDefaults.standard.dictionaryRepresentation()

for key in defaults.keys {
    print("-- \(key): \(String(describing: defaults[key]))")
}
```

Here's some (including the one we just added) which show for one of my own apps:

```
-- mpDebugTrackedKey: Optional(1)
-- AppleLanguagesDidMigrate: Optional(20E247)
-- AppleKeyboards: Optional(<__NSCFArray 0x600003ce8880>(
-- AKLastLocale: Optional(en_US)
-- AppleLanguages: Optional(<__NSCFArray 0x600003ce88c0>(
-- AppleKeyboardsExpanded: Optional(1)
-- AppleLocale: Optional(en_US)
-- AddingEmojiKeybordHandled: Optional(1)
-- EnablePro: Optional(YES)
-- ApplePasscodeKeyboards: Optional(<__NSCFArray 0x600003ce8840>(
-- NSInterfaceStyle: Optional(macintosh)
-- NSLanguages: Optional(<__NSArrayI 0x600002983ce0>(
-- AppleLanguagesSchemaVersion: Optional(3000)
-- PKLogNotificationServiceResponsesKey: Optional(0)
```

While they are typically used for command line-based programs, such as `xcode-build`, we can certainly make use of them in our app. For many launch arguments that Apple provides – we don't necessarily need to know that they are in `UserDefaults` – they'll just augment some runtime behavior for us.

Consider `NSDoubleLocalizedString` – when present, it'll repeat any localizes string it finds within the user interface. So, this…

```
let title = NSLocalizedString("headerTitle", comment: "Welcome to a
Best-in-Class iOS App")
```

…would show up in the interface as: "Welcome to a Best-in-Class iOS App Welcome to a Best-in-Class iOS App" – very useful for testing how interfaces handle long text. Languages such as German will have more characters than English, so with one launch argument, you can get a feel for how your app is holding up for localization and internationalization purposes.

Or, how about `-NSShowNonLocalizedStrings YES` – which capitalizes any non-localized string it finds. Further, say you need to test a different language – use `-AppleLanguages (es)` to change the device language on the fly. In fact, we saw this key above when we dumped user defaults. No need to go to the Settings app to switch up languages, simply override the launch argument.

And, that's just languages and strings. There are lots of other (sometimes, undocumented, unfortunately) launch arguments Apple has available for us to use. Take focus testing. Use `-UIFocusLoopDebuggerEnabled YES` and a debugging overlay will show while your app is running to help you determine which view has focus. Or, how about `-UITraitCollectionChangeLoggingEnabled YES`, which lets developers see exactly when a trait collection changes and all the details about that change.

Again, no matter which one you use (or add yourself), if you were to peek at `UserDefaults` when your app is running when using them, you'd see them there. Most of the time, you don't need to access the value in the defaults, but it helps to know that it's there. Why? Because that means you can use launch arguments for plenty of other things:

- Debug Flags: Pass in some flags to help you debug things like in-app purchase flows, gating features and more.

- Profiling: You can use them for profiling purposes as well, enabling or disabling launch metrics or toggling any performance logging you may have in place.
- Flow Tests: Got a new feature in the works, but don't want to tap on three different buttons to get there? Enable a flag to set any relevant data and present the view right away.

Environment Variables

Environment variables are a little more structured. You give them a specific string value, and you can find any of them within `ProcessInfo`:

```
let flags: [String:String] = ProcessInfo.processInfo.environment
```

Since these are tied to the process and scheme, you don't have to worry about them slipping into production. While you can do similar things with both a launch argument and environment variable – you should think about them separately. And, going back to our launch argument to enable "pro" – that's better served as a environment variable in my opinion.

Specifically, I like to think of environment variable as something I can reach into during my app's execution at any point. Launch arguments, on the other hand, are just that – things I would check for at launch if I added them. Or, if it's an Apple supplied launch argument, then I don't think about it all – they'll do their thing on their own.

As I mentioned earlier, I love using environment variables to give me "pro" access to features while I'm developing my apps. I add something like "ForcePro", and "true" as an environment variable. Then, in my app – I can turn around and check for that:

```
func checkIsPro() -> Bool {
    if let flag = ProcessInfo.processInfo.environment["ForcePro"],
      flag == "true" {
        return true
    }

    return userManager.entitlements.contains("pro")
}
```

In fact, I use environment variables quite a bit – so I abstract out their values into a tiny little struct. Here's one, verbatim, that I use in my app:

```
struct EnvironmentConfig {
    static var shouldForceProStatusOnUser: Bool {
        if let value =
ProcessInfo.processInfo.environment["SHOULD_FORCE_PLUS"],
        value == "1" {
            return true
        } else {
            return false
        }
    }

    static var shouldAddDemoData: Bool {
        if let value =
ProcessInfo.processInfo.environment["SHOLD_CREATE_DEMO_DATA"],
        value == "1" {
            return true
        } else {
            return false
        }
    }

    static var shouldAddDemoTeam: Bool {
        if let value =
ProcessInfo.processInfo.environment["SHOULD_CREATE_DEMO_TEAM"],
        value == "1" {
            return true
```

```
        } else {
            return false
        }
    }
}
```

Of course, Apple has several environment variables you can use. Like launch arguments, many of them aren't as easy to find as I would like. It's almost as if they are more of a trade secret, passed down from developer to developer.

As such, it's not uncommon to stumble across one while reading documentation over a particular API from Apple. For example, the Metal framework has many of them available for debugging purposes.

Here are a few of them I've used:

Key	Value	Purpose
NSZombieEnabled	YES	Used to find objects you're accessing after they've been released.
LIBDISPATCH_COOPERATIVE_POOL_STRICT	1	Detect unsafe blocking primitives.
_XCAppClipURL	A URL	Passes the URL to an App Clip.
CI_PRINT_TREE	Graph/Output/Options	Visualizes Core Image graphs.

As we've seen, launch arguments and environment variables are a wonderful tool to use in your iOS travels. While they are a phenomenal choice for debugging, feel free to let your imagination run wild to set your app up in different states, toggling features or changing languages.

Most of all, stay curious – there are several of them out there, nestled away in Apple's documentation. Maybe the next one you find could significantly change your workflow?

Property List Keys

Property List keys are the linchpin of any iOS app. Around since the dawn of iPhoneOS, property list keys are often found within the well-known `info.plist` file in Xcode projects. In modern Xcode templates, such as ones with SwiftUI, this may not show by default. However, adding something like a privacy key (such as permissions to access the camera) will add one for you.

No matter if it's there by default or not, still today, iOS apps rely on these property list keys for all sorts of uses. You won't get far into developing an app without using one. These keys are accessed via numerous iOS frameworks at runtime, and in some cases – their absence could even cause a crash (i.e., attempting to access a hardware resource you don't have a property list key for).

Conceptually, property list keys are nothing more than metadata. They are created as key-value pairs, so in *reality* – they are nothing more than a dictionary. In fact, if you view them as source code – you'll find that they are simply .xml under the hood:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://
www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
<! -- plist contents -->
</dict>
```

```
</plist>
```

Since they are used for so many purposes, you could even use different .plist files for debug and release builds. In Xcode's build settings, you may define separate paths for each of them. While property lists, and their makeup – aren't terribly interesting, there are several interesting keys available to use that you may not know about.

That's what this journal entry is all about. Without further ado, let's look at some of them:

CoreSpotlightContinuation

Value: Boolean.

Purpose: Enables Core Spotlight continuation, so that your app can further handle searches inside your app that have come from Spotlight.

INAlternativeAppNames

Value: Array.

Purpose: A list of other names that Siri can recognize your app by, which helps disambiguate requests.

UIViewControllerBasedStatusBarAppearance

Value: Boolean.

Purpose: Determines whether the view controller present determines the status bar style.

UIWhitePointAdaptivityStyle

Value: String. Accepted values are `UIWhitePointAdaptivityStyleStandard`, `UIWhitePointAdaptivityStyleReading`,`UIWhitePointAdaptivityStyle-Photo`,`UIWhitePointAdaptivityStyleVideo` and `UIWhitePointAdaptivity-StyleGame`.

Purpose: Changes True Tone behavior on devices which support it, providing a better viewing experience if your app doesn't mesh well with True Tone color shifts.

UIViewControllerBasedStatusBarAppearance

Value: Boolean.

Purpose: Determines whether the view controller present determines the status bar style.

LSSupportsOpeningDocumentsInPlace

Value: Boolean.

Purpose: If set to `YES`, this means your app can handle the original version of a file that's passed to it instead of a copy.

CFBundleDisplayName

Value: String.

Purpose: The name of your app, visible to users, displayed on the Home Screen and recognized by Siri.

CFBundleIdentifier

Value: String.

Purpose: The value of your app's bundle identifier.

CFBundleShortVersionString

Value: String.

Purpose: The value of your app's current version, useful for displaying within a Settings screen or perhaps including with debugging files.

NSExtensionActivationRule

Value: Dictionary.

Purpose: For extensions, this specifies the "rules" and conditions for your app extension to be shown in the share sheet.

NSExtensionPrincipalClass

Value: String.

Purpose: The name of the view controller which the extension will use to present itself, useful for changing out the default one that Xcode supplies you from the extension templates.

UIDocumentPickerModes

Value: Array or strings. Acceptable values are `UIDocumentPickerModeImport`, `UIDocumentPickerModeOpen`, `UIDocumentPickerModeExportToService`, and `UIDocumentPickerModeMoveToService`.

Purpose: Identifies the document picker modes that you support.

NSExtensionJavaScriptPreprocessingFile

Value: String.

Purpose: Use this to supply a Javascript file that can be used in a Share extension.

ITSAppUsesNonExemptEncryption

Value: Boolean.

Purpose: Indicates that your app doesn't use encryption, allowing you to bypass the prompt in App Store Connect asking you if it does.

.plist Pro

To get a feel for all of the many purposes that your app's info.plist serves, simply go to Xcode and open yours. Clicking the plus icon will present a dropdown with hundreds of choices for you. Thankfully, these are human-readable values instead of the actual keys, like we listed above.

Further, you can specify a property list key to *only* apply to a specific platform or device. For example, `UISupportedInterfaceOrientations~ipad UIInter-`

`faceOrientationPortrait` would mean that for just the iPad app, your app only supports portrait mode (a bad idea by the way, but hey—it's an example).

That syntax looks like this: `[propety list key]-[Apple platform]~[Apple Device]`. For the platform key, we can use `iphoneos` or `macos`, and the supported devices are `iphone` and `ipad`. Though, I expect this to change soon with visionOS.

All told, the next time you go about your development tasks — remember the unsung hero of your apps, controlling everything from system version support to how extensions activate. Our good friend property list, and all of its keys.

simctl

I remember the day I first ran a command in Terminal. At that moment, I truly felt like a bonafide programmer. With my fellow students huddled around my laptop at college, I swiftly pecked out a command with steadfast confidence while acting as if I'd done it a million times before.

For the modern iOS developer, spending time in Terminal is commonplace. Today, we'll look at a utility one can use in Terminal that's been around for a few years but has largely eluded the spotlight:

simctl—a lovely little tool to aid with the iOS simulator.

So, what is it? First released alongside Xcode 6, `simctl` is a command-line utility (itself housed within `xcrun`) that can be used in Terminal to do all sorts of helpful things with the iOS simulator. It supports simple tasks such as device bookkeeping to more immediate time savers such as recording video.

If you've ever seen Mac apps which augment the simulator in some way – well, this is how they do it. Calling into `simctl` to pull off their required tasks from changing the device time, to resetting it and more.

This simple C.L.I. wrapper essentially parlays the power of Xcode's tools over to Terminal. Though such tools are available for individual download, in the modern era we simply have them included as part of the Xcode package.

To check which version of `simctl` you're using, simply run this command:

```
xcode-select --print-path
```

During summer development, when the beta version of Xcode is no doubt on your machine – you might see `Xcode-Beta` in the result. Under other circumstances, it should produce something like this:

```
/Applications/Xcode.app/Contents/Developer
```

No matter which version you're pointing to, if you need to switch back and forth, that's no problem:

```
xcode-select -switch (the path to Xcode)/Xcode.app
```

Anyway – you've got it installed. You know which version you're using. Great. So, what can it do?

Examples

One of my favorite uses of `simctl` is to capture a quick video from the iOS simulator:

```
xcrun simctl io booted recordVideo (filename).(extension)
```

When used, a high-fidelity recording begins capturing everything on the main frame buffer display of the simulator. When you're ready to stop, simply return to Terminal and enter CTRL+C.

You can save the resulting file as either a .mov, .h264, .mp4 or .fmp4. The opportunity is there to explore unconventional solutions, too. Take, for instance, the ability to replace a file destination with a URL to establish a server socket to pipe video over.

The equivalent does exist for screenshots as well, with built-in support for .png, .tiff, .bmp, .gif and .jpeg:

```
xcrun simctl io booted screenshot myScreenShot.png
```

But, hey—CMD+S is also supported in the simulator for screenshots — and those are .png by default.

Device Selection

You may have noticed the "booted" argument used in the previous commands. All of simctl's sub commands have a device parameter, and each device ran via the simulator has its own unique identifier too, otherwise known as a UDID. To view these yourself, feel free to employ—what else—a `simctl` command:

```
xcrun simctl list
```

This produces a comprehensive list of device pairs, types, their availability, and runtimes available for the simulator to use. On mine, I can clearly see the iPhone 14 Pro max I've got currently running among the output:

```
iPhone 14 Pro Max (D1F67F00-FA3D-42B7-9E2F-FEF23809D4A0) (Booted)
```

That means I could just as well take a screenshot by supplying its UDID for the device parameter:

```
xcrun simctl io D1F67F00-FA3D-42B7-9E2F-FEF23809D4A0 screenshot screen.png
```

As you've likely concluded, passing "booted" for the device argument automatically supplies the booted device's UDID on our behalf. If there is more than one device running, `simctl` will just pick one of them.

For the remainder of this journal entry, I'll use booted where the device parameter is expected.

Diagnosing Issues

Let's face it, nobody likes log files. Developers always ask for them, and consumers have no idea what they are. And, when we do use them, they are usually bereft of useful formatting or digestible text.

Yet, as developers, *we just can't stop logging everything*. Typically for good reason, too. That said, I'd be remiss if I didn't mention you can generate an incredibly detailed log of anything that's happened in your simulator session with `simctl` :

```
xcrun simctl diagnose
```

After you move past the wall of text warning you that the dump contains *only* your personal information, iCloud Accounts, Apple ID, name, username, email address and settings, file paths, downloads, IP addresses, network connection information and installed applications—what you can find in it has some neat stuff:

- System Logs
- Simulator Logs
- Device and Environment .plist settings
- And a whole load of more information

But, yes – buyer beware. Keep that log stashed locally, unless the idea of a total stranger standing over your shoulder while you use a MacBook doesn't frighten you at all.

URL Schemes

Another increasingly common part of iOS development is testing url schemes. With deep linking becoming a powerful tool to incorporate both as a consumer and a provider, `simctl` can help here too:

```
xcrun simctl openurl booted https://www.bestinclassiosapp.com/
```

And just like that, the active device pops open Safari with the URL. Of course, if you know another app's custom URL scheme –those are fair game, too:

```
xcrun simctl openurl booted sms: #Open Messages
```

Though that previous command simply opens the Messages app, parameters are fully supported as well. For example, if you wanted to send a friend a text message – fire away:

```
xcrun simctl openurl booted sms:1-417-323-2345
```

Odds and Ends

Though `simctl` has a robust toolset that's worth your time, here are some more of my personal highlights:

Adding Media

```
xcrun simctl addmedia booted (path to file, or files)
```

Printing Environment Variables

```
xcrun simctl getenv booted (variable name)
```

Forcing an iCloud Sync

```
xcrun simctl icloudsync booted
```

Reset Device Content and Settings

```
xcrun simctl erase booted
```

Install an App On Device

```
xcrun simctl install booted (The path to the app)
```

...And Launching Apps (via bundle ID)

```
xcrun simctl launch booted (ID)
```

While command line utilities aren't the heroes of iOS development, it cannot be argued that they are indeed useful. Bolstering our tool belt as software engineers cannot be understated. Much like buying insurance, it's not so incredible day in and day out—until the day comes when you really need it.

vmmap

Let's turn our attention to yet another command line interface tool that comes with Xcode's tool, `vmmap`. This tool helps developers see exactly how the virtual memory of their app's process is being created and used. Why is that useful? Well, it can help us understand why memory is being used, *where* it's being used and most importantly – what kind of memory is being used. That information can give us valuable insight to make corrections.

It's straightforward to use, in fact – it only requires one argument, which accepts three different values. Any one of these will work:

- The process identifier.

- The process name.
- Or, the name of a .memgraph file.

The first two options are found easily while debugging in Xcode. If you go to `Debug -> Attach to Process` you'll see all the current running processes from Xcode. Among them, you'll find your app, its extensions and other similar processes. So, if your process name was `Best-in-Class-App`, we could use `vmmap` by running this from within Terminal:

```
vmmap --summary Best-in-Class-App
```

A .memgraph file is a little different. These are memory graph files, which – in of themselves – are very useful. Anytime I use `vmmap`, I try to use it with a .memgraph file. Memory graph files are useful for diagnosing all sorts of issues, chief among them are memory leaks. However, when used in tandem with `vmmap`, you can uncover all sorts of interesting things.

Before I show you a quick example, here's a reference of all the options that `vmmap` accepts:

Argument	**Purpose**
-s/-sortBySize	Sort regions by size (dirty + swapped).
-w/-wide	Print wide output
-v/-verbose	Equivalent to -w -submap -allSplitLibs -noCoalesce.
-pages	Print region sizes in page counts rather than kilobytes.
-interleaved	Print all regions in order, rather than non-writable then writable.
-submaps	Print info about submap.
-allSplitLibs	Print info about all system split libraries, even those not loaded by this process.

-noCoalesce	Do not coalesce adjacent identical regions (default is to coalesce for more concise output).
-summary	Only print overall summary, not individual regions.
-stacks	Show region allocation backtraces if target process uses MallocStack-Logging (implies -interleaved -noCoalesce).
-fullStacks	Show region allocation backtraces with one line per frame.
-forkCorpse	Generate a corpse fork from process and run vmmap on it.

Anytime I use `vmmap`, I typically use it with the `--summary` flag. This, well, summarizes your app's memory situation in a way that's a bit easier to understand. By coalescing information together, instead of just showing individual regions, you can get a better understanding of where things stand.

```
vmmap --summary demo.memgraph
```

With that command, Terminal will spill the details of the size in memory of the region, how much of that memory is dirty, swapped or compressed. If you're a bit unfamiliar with what all of that means — it's the dirty and swapped memory that you're probably needing to address.

Why? Well, think about your app's memory, how it uses it and how much it has left to consume. We know memory is important, and iOS always works better when you are efficient with it in your app. But `vmmap` is a bit hard to make sense of if you don't understand a few fundamental concepts.

To that end, each page of memory is about 16 kilobytes in size. And, as we do things in our app, we start to fill up those pages. Our app is only allowed so many of them, too. This is similar to an actual notebook pad, where the page becomes full as you write on them — and in doing so, the notebook itself becomes full. But, the memory we use in our app is not all equal.

And, that's where it's tricky. We have clean, dirty and compressed memory. Clean memory is fairly easy to understand – it's simply memory we've allocated for use. If you think back to Objective-C, you see this all the time:

```
UILabel *test = [[UILabel alloc] initWithFrame: CGRectZero];
```

The first message in that call says it all, `alloc` – "allocate memory for this label". And, that's what iOS does using clean memory. But, allocating it wouldn't mean much if we didn't actually use it – and that's when we get into dirty memory. This is memory we're now using, which we allocated prior. We use too much of this, and we're in trouble.

Finally, there's compressed memory, which was added back in iOS 7. This is memory that hasn't been accessed in quite a while. If it hasn't been used, iOS compresses it down and uncompresses it later. It's a bit dynamic in nature. It's these two types of memory footprints we want to track.

Which leads us back to `vmmap`, since it can list the amount of any region's memory. It'll show how much of that memory is dirty, swapped or compressed.

Let's end with a quick example that I always use to show how `vmmap` is useful. Say I have an app which simply shows an image view from loading content from your app's bundle:

```
<UIImage: 0x600003d41a40>, {1718, 2048}
```

If we haven't downsampled it, we can see how much memory we're wasting unnecessarily. The reason is beyond the journal, but go to my website and read my article "Optimizing Images" to learn more. To distill it down, in some cases an image's memory hit comes from its dimensions – not its file size. Continuing, if we create a memory graph for that app, we can run it through `vmmap` to see its memory footprint:

```
vmmap --summary test.memgraph
```

We'll get a lot of data back, but I hone in on `Physical footprint`:

```
Physical footprint:         69.5M
Physical footprint (peak):  69.7M
```

So, we're at about 70 megabytes. Since these text results are so large, I also use `grep` to find frameworks I suspect are the issue. Here, that's `ImageIO`:

```
                                 VIRTUAL RESIDENT    DIRTY  SWAPPED VOLATILE   NONVOL    EMPTY   REGION
REGION TYPE                         SIZE     SIZE     SIZE     SIZE     SIZE     SIZE     SIZE    COUNT (non-coalesced)
===========                      ======= ========    =====  ======= ========   ======   =====  =======
Activity Tracing                    256K       36K      36K       0K       0K      36K       0K        2
CoreAnimation                         8K        8K       8K       0K       0K       8K       0K        2
CoreServices                        488K      344K     344K       0K       0K       0K       0K        2
Foundation                            4K        4K       4K       0K       0K       0K       0K        2
Image IO                          13.4M     13.4M    13.4M       0K       0K       0K       0K        2
```

There, we can clearly see we have about 14 megabytes of dirty memory:

```
vmmap --summary test.memgraph | grep "Image IO"

Image IO  13.4M   13.4M   13.4M    0K  0K  0K   0K  2
```

Now, if we go back to our code and downsample the image to save on memory – we can see the savings using another .memgraph file and `vmmap`:

```
vmmap -summary testOptimized.memgraph

Physical footprint:         56.3M
Physical footprint (peak):  56.7M
```

And just like that, `vmmap` helped us find dirty memory and confirm we've fixed it.

Using command line tools is a great way to check things over quickly, but please keep in mind that Instruments is also a fantastic way to perform similar debugging scenarios. Two templates, Allocations and VM Tracker, can put a nice, graphical display over a lot of the information that `vmmap` provides.

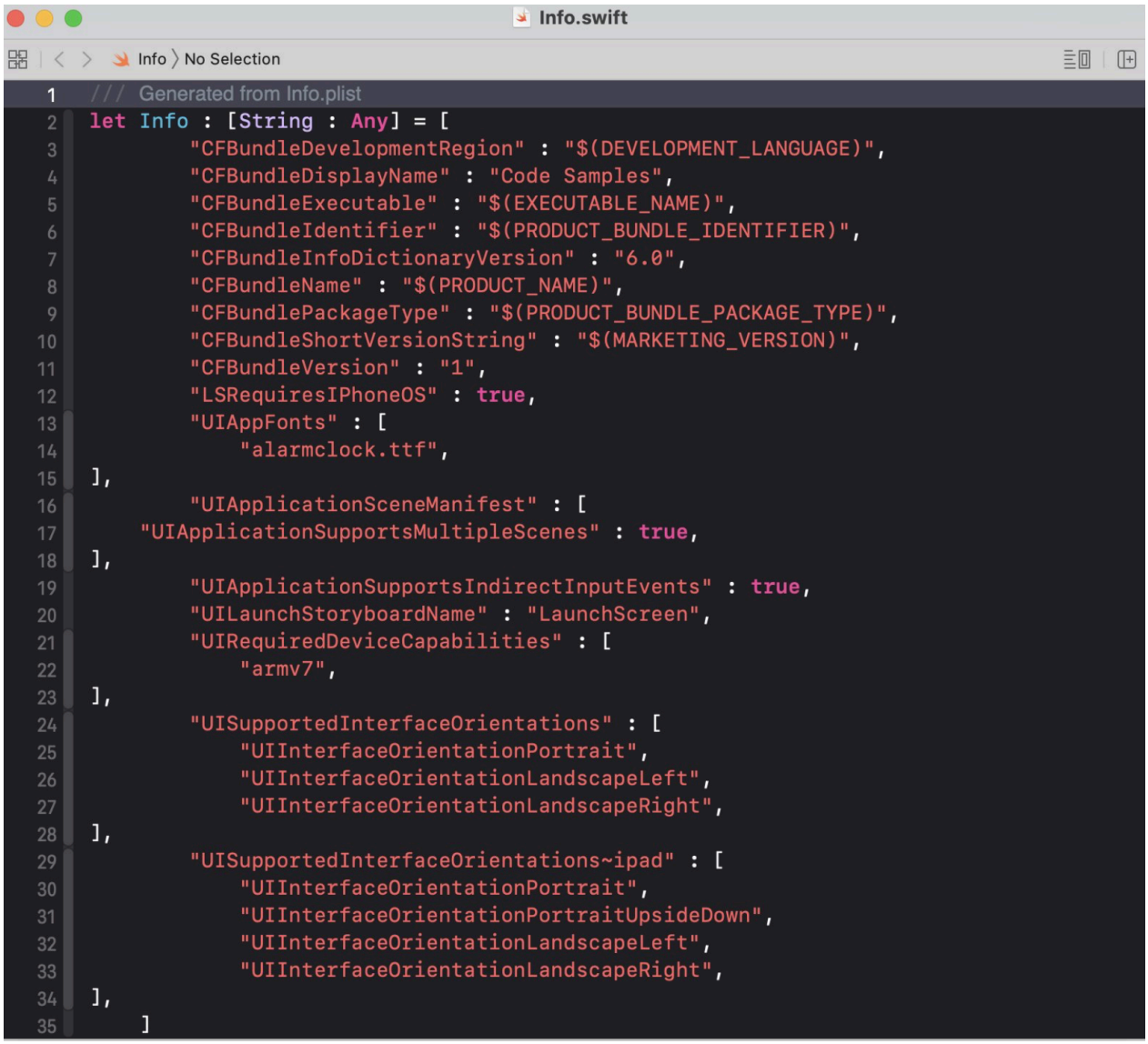
plutil

Today, we turn our attention to another useful command line tool, `plutil`. Like the others we've discussed, these are found within Xcode Tools. This little tool helps you manipulate .plist files — one of the most important files in a Xcode project. Using this, we can check whether a .plist file is valid, edit them and convert them back and forth into different formats.

There are times when you might want to access .plist files in code, or have them exported or used in a different format. You can easily do that with this tool. Take my book series and the demo project:

```
$plutil -convert swift info.plist
```

Here, I've noted that I want to convert my app's info.plist file using Swift into a dictionary. Here's the result:



I could even do the same thing with Objective-C:

```
$plutil -convert objc info.plist
```

Which results in:

```objc
#import "Info.h"

/// Generated from Info.plist
__attribute__((visibility("hidden")))
NSDictionary * const Info = @{
    @"CFBundleDevelopmentRegion" : @"$(DEVELOPMENT_LANGUAGE)",
    @"CFBundleDisplayName" : @"Code Samples",
    @"CFBundleExecutable" : @"$(EXECUTABLE_NAME)",
    @"CFBundleIdentifier" : @"$(PRODUCT_BUNDLE_IDENTIFIER)",
    @"CFBundleInfoDictionaryVersion" : @"6.0",
    @"CFBundleName" : @"$(PRODUCT_NAME)",
    @"CFBundlePackageType" : @"$(PRODUCT_BUNDLE_PACKAGE_TYPE)",
    @"CFBundleShortVersionString" : @"$(MARKETING_VERSION)",
    @"CFBundleVersion" : @"1",
    @"LSRequiresIPhoneOS" : @YES,
    @"UIAppFonts" : @[
        @"alarmclock.ttf",
    ],
    @"UIApplicationSceneManifest" : @{
        @"UIApplicationSupportsMultipleScenes" : @YES,
    },
    @"UIApplicationSupportsIndirectInputEvents" : @YES,
    @"UILaunchStoryboardName" : @"LaunchScreen",
    @"UIRequiredDeviceCapabilities" : @[
        @"armv7",
    ],
    @"UISupportedInterfaceOrientations" : @[
        @"UIInterfaceOrientationPortrait",
        @"UIInterfaceOrientationLandscapeLeft",
        @"UIInterfaceOrientationLandscapeRight",
    ],
    @"UISupportedInterfaceOrientations~ipad" : @[
        @"UIInterfaceOrientationPortrait",
        @"UIInterfaceOrientationPortraitUpsideDown",
        @"UIInterfaceOrientationLandscapeLeft",
```

For Objective-C, we could also generate a header file by appending the `-header` flag. Beyond Objective-C and Swift, we can also convert .plist files into XML, binary or JSON. This can be nice when using third-party tools which put a user interface over property list files for easier maintenance.

Personally, I find that I like to convert property list files to .json for easier editing:

```
$plutil -convert json info.plist -r
```

Then, after my edits are done – I simply put it back to XML:

```
$plutil -convert xm1 info.plist
```

Beyond conversions, we can even add or remove keys from property lists:

```
$plutil -insert/-replace/-remove
```

Or, perhaps after editing one — we need to ensure it's still in a valid format:

```
$plutil -lint info.plist
info.plist OK
```

There's not much else to say about `plutil` — but not to its detriment. It's an easy-to-understand tool with one, focused job. And these are things that I absolutely adore finding and learning about! The next time you need to get dirty with a property list file, consider the command line and Terminal for the job.

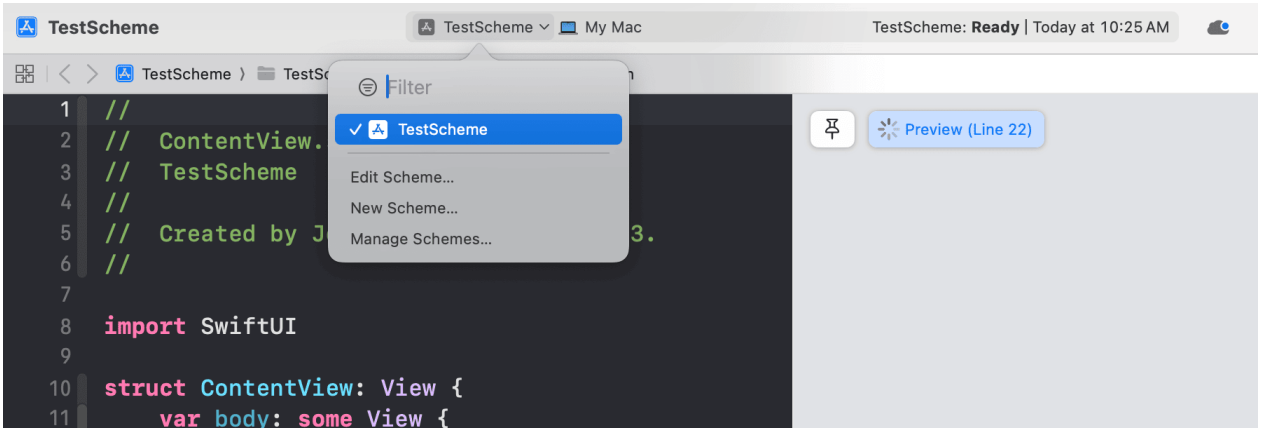
Leveraging Schemes

Xcode's schemes are a wonderful utility to get several tedious tasks done easily. Need to test in a different language? Skip going over to the device's Settings app and changing it there — simply make a new scheme and set the desired language as a launch argument (we learned about those earlier).

Every scheme you create can do different things. By default, you'll get one for each new project. So, you'll start with the primary app's target — and the scheme you get along with it. As you add new targets, though, like a widget extension — Xcode will create a scheme for that too. In addition, a build scheme can have multiple targets that it manages.

Since build schemes can do so much – I want to give you a tour of a build scheme in this journal entry. We'll look at every little tab you might see when you peek into one.

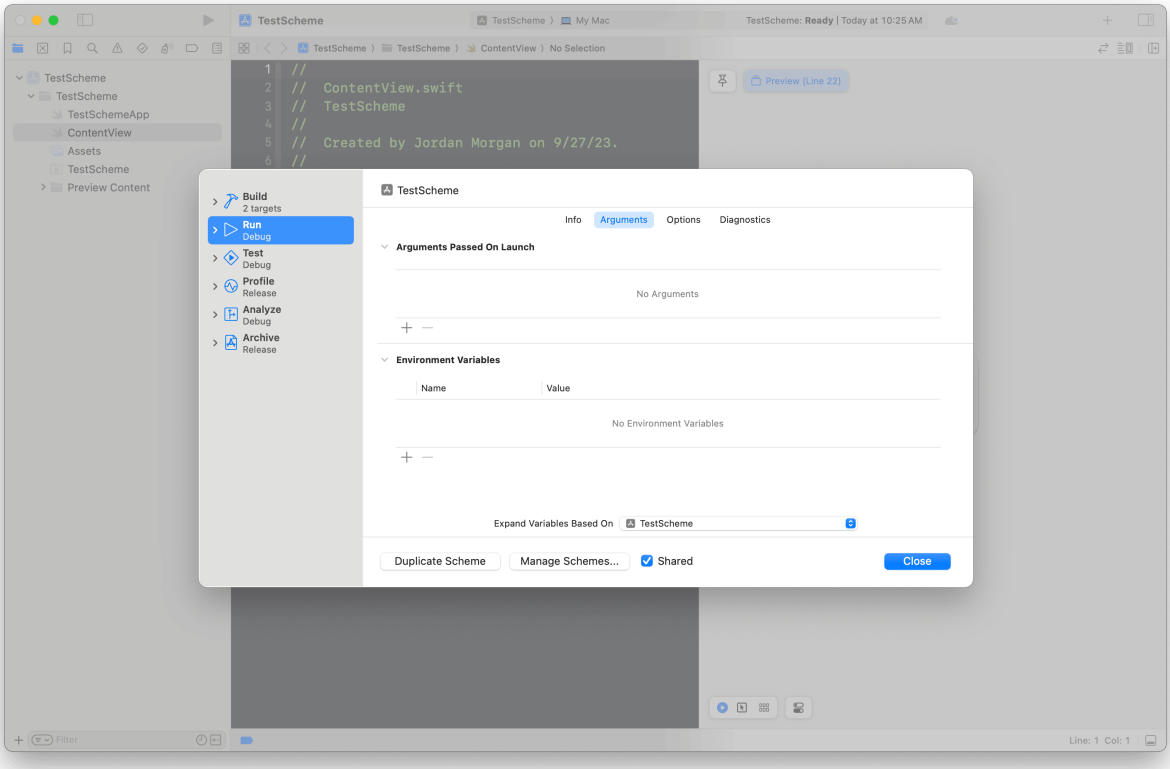
If I were to create a brand-new Xcode project for a Mac app, we can see the scheme that was created for us at the top:



Schemes are used a few different ways, and it helps to understand what each of their options represent. If you open a scheme, you'll typically see six different tabs on the left:

- Build
- Run
- Test
- Profile
- Analyze
- Archive

Again, if we open the scheme from above – we can clearly see all of those options:



For the *Build* option, we can configure tasks or tweak how the scheme builds each target. The build order option lets us build a target's contained dependencies in parallel, or we can build them in the manual order they are listed as within the target.

I should note, you probably don't want to use that last one — it can make builds take longer. In recent versions of Xcode, it's still supported — but it's deprecated altogether. Building in parallel is typically the right call for most of us. There are legacy build setups which may require you to turn it off, but you'll likely know that ahead of time if you are dealing with such an issue.

As for the build options, "Find Implicit Dependencies", which is on by default, helps Xcode gain more information about a target's dependencies during builds. Later on, it can use that information to speed up builds, and better schedule any build-related tasks it has to complete.

Next, under *Run*, we can create tasks to perform before or after a build. Perhaps more importantly, we can use launch arguments and environment variables here. Refer to that journal entry to see all the magical things you can do with those.

For the "Info" section under *Run*, we can specify certain LLDB files to use during debugging. If you've got a certain setup that you like to use, then adding it here would make it available for use during debug sessions.

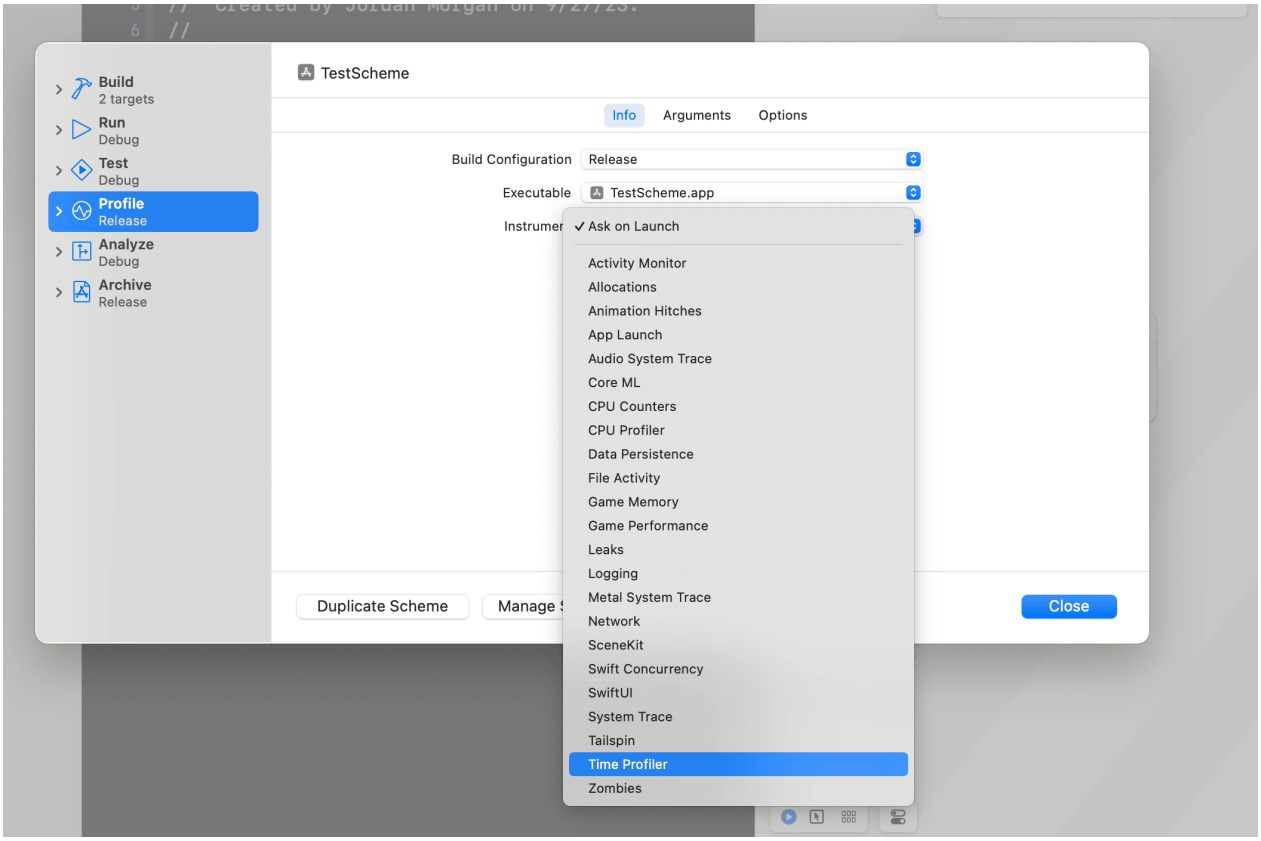
There's also options to use different build configurations – such as debug or release. These are helpful in certain contexts, such as testing push notifications. If you pop on the release build configuration, you'll get pushes from APNS, for example.

Finally, we've got "Diagnostic" options. Here, we can enable or disable things like the main thread checker, turn on memory management tools or runtime debugging utilities such as the address sanitizer. These can heavily affect build times and LLDB, so remember to turn them off when you're finished.

Moving along, we've got the *Test* option. This, in of itself, is another target of the scheme. When you create a new Xcode project, you get a testing suite by default (unless you uncheck the box which adds it). Since this deals with the `XCTest` framework, the options are a little more dialed in. You can still change the build configuration, but here you can manage your testing plan.

Next, we've got options to use when you profile your target – the aptly named *Profile* tab. If you got to `Product -> Profile` in Xcode, the options you set here will be used. Profiling an app runs through yet *another* app, Instruments, which is used for virtually any debugging purpose you can imagine in iOS development.

This build scheme option is handy because you can set a default template for Instruments to use anytime you begin profiling. For me? I use Time Profiler a lot – so in new projects, this is one of the first options I set (the default option is to simply open the templates menu in Instruments):



The arguments tab works much the same way, but the "Options" tab allows you to launch the app with state restoration, set a working directory and more.

Okay, that's four down – two to go. Let's check out the *Analyze* tab next. This, too, is run when you go to `Product -> Analyze` in Xcode. Running an analysis can uncover all sorts of issues such as logical coding errors, potential Objective-C bugs, security issues and more. When it's run – the option (not plural, just one option) is used. And that's simply the build configuration you want: Debug, or Release.

Since that one is easy to understand, we're left with the final option – *Archive*. Again, the options we chose here are used when we go to archive our app, and that's something we must do anytime we submit it to the App Store. There's not much here,

like the last option, apart from a build configuration toggle, the ability to name our archive and to reveal it from within the organizer.

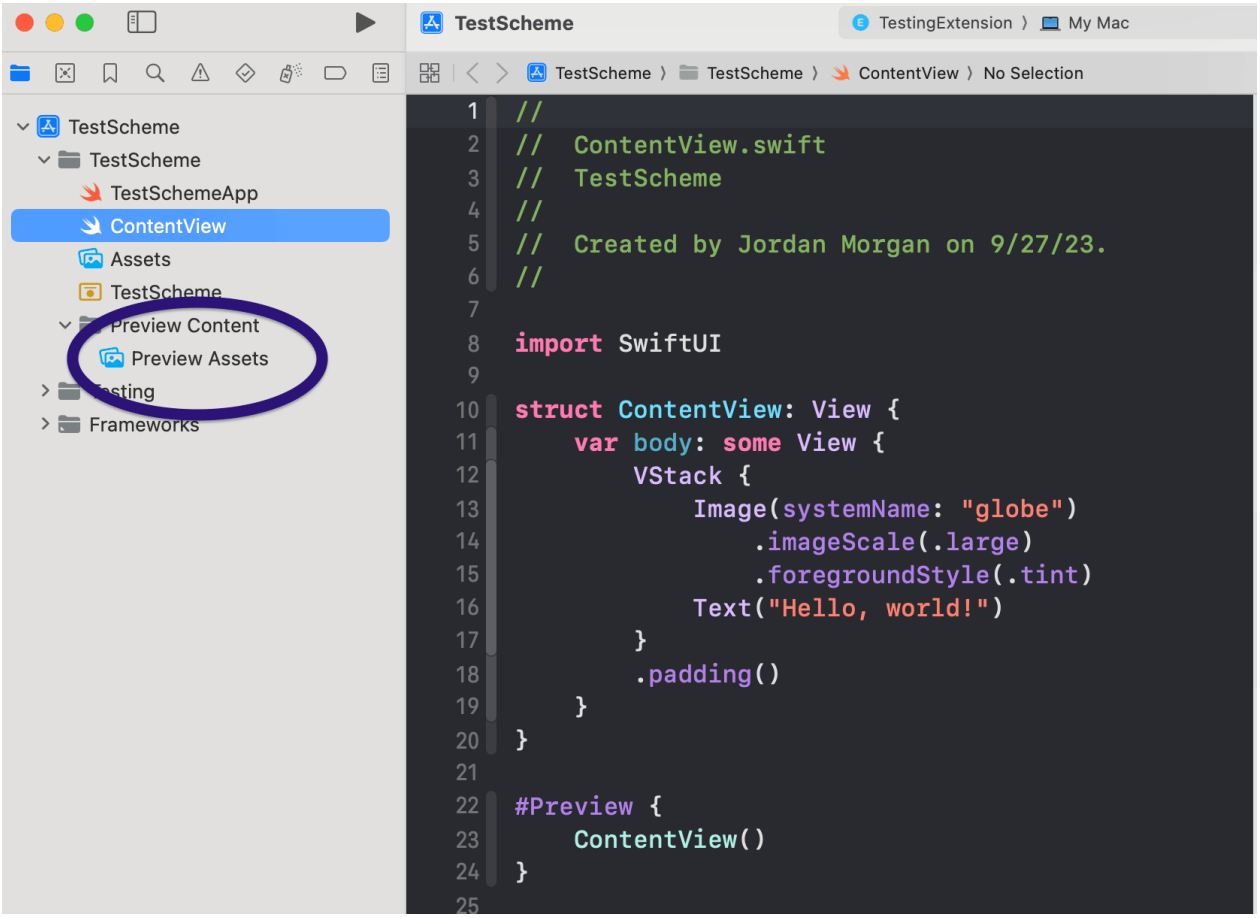
So – what do you do with all of that information? Well, typically, larger organizations like to set up particular build schemes for their apps to streamline development. Indies love to have different schemes for different stores, such as macOS or distributing their app from their own website. Me? I love using a scheme to send in launch arguments and change the language of the simulator most commonly. Build schemes can do things both big and small.

Whatever your use case, a custom build scheme (or editing the one you already have) can save you a significant amount of time.

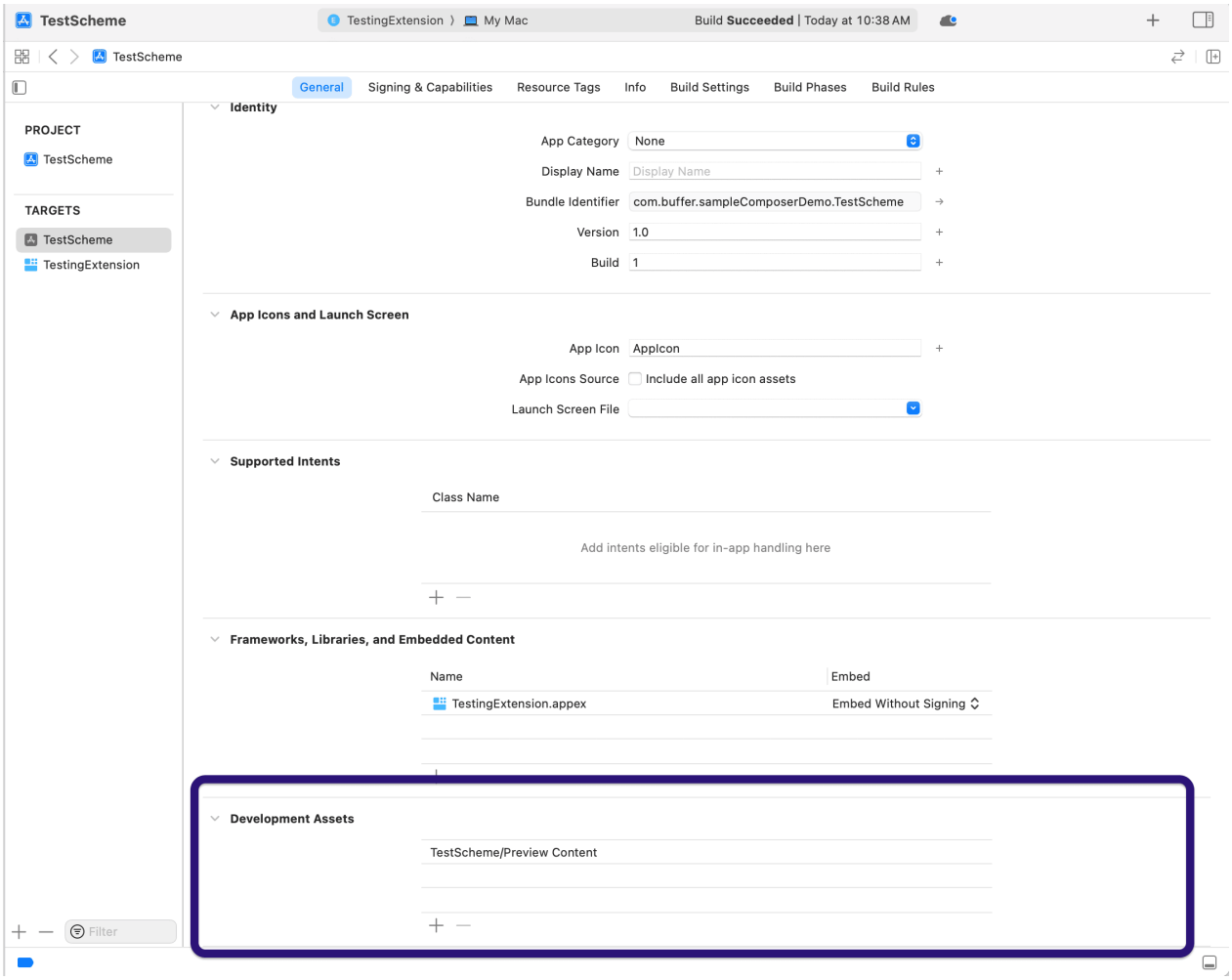
Development Assets

Xcode Previews changed the game for iOS developers. Freeing us from the code, build, run and navigate pattern of testing our work – we now just see our code come to life as we write it. Since Xcode Previews have brought about a rich new way to see our interfaces, we changed our development techniques along with it. For example, the value of mock objects or testing data has become more prevalent.

Apple realized this, and they've extended the idea of testing data over to *testing assets*. Now, modern versions of Xcode give you a *development asset catalog* by default:



If you're working on an existing project, or a new target, that doesn't have one – you can add them here under your target's "General" settings towards the bottom:



Now, anything you add here won't be shipped with your app – so this opens you up to create realistic previews and imagery throughout your previews. When your app bundle is archived – development asset catalogs are ignored. The assets found within these catalogs are referred to in code like any other asset might be:

```
Image("MyDevAssetName")
```

Though, there are uses simply beyond only media assets. You can place entire files in here, too. So, that means your mocks or .json files that you might use with `Codable` models can all live in the development asset catalog. Even raw data assets are fair game. I have a local .gif data asset I use to test any .gif handling code I have:

```
let gifData = NSDataAsset(name: "Dev_Gif", bundle: .module)!.data
```

Just remember – these are configured *per target*. As a rule, I tend to only reference development assets within Xcode Preview code:

```
#Preview {
    Image("MyDevAssetName")
}
```

This little feature can save you so much space on your app's binary. In fact, it's so incredibly trivial to use that I nearly didn't include it within this journal. But that would have been a shame, wouldn't it? Are tips no less valuable because they are easier to explain?

I think not, in fact – I think those are the best ones. Go forth and add all the preview assets you want during development. Rest easy knowing that Xcode will strip them from your app bundle during the achieve process, and they'll never see the bright lights of production releases.

How to Focus

I don't mean to brag here, I really don't – but the question I get most from people is, "How do you do all the things that you do?" It's a fair ask, I suppose. At the time of this writing, I:

- Have a full-time job.
- Write a book series.
- Maintain a blog with sponsorships.
- Release my apps.

- And have a family, complete with three kids who are a little hyper during all waking hours.

How do I do all of those things, then? Am I some sort of time management guru? Hardly. Or, am I one of those #JustHustleHarder guys? Do I spew the benefits of cold showers at 5:00 a.m. and getting a ten-mile run in before the sun even rises to anyone who will listen?

Absolutely not, I'm lucky to be half shaven with clothes on by the time my kids come in and invade my space. To wit, my alarm clock is hardly necessary. Typically, my peaceful wakeup call is my son tapping my nose asking me for cereal.

What gives then? Well, before I go on to talk about the *how* (unsurprisingly, it's focus) I think context helps too:

- I have a job where I have four-day work weeks.
- That job also offers me a sabbatical.
- I'm fine with going slow on projects of my own (this book series took several years to write, for example).
- My wife stays at home, and does *a lot*.

When you add in more context to who I am and what I do, it's obvious that I am no superhero — I simply have a lot of help. And, yes, I can focus pretty well when I need to. That's about it.

I can't do much about your personal circumstances — though I wish nothing but the most advantageous ones for you — but I can tell you how I think about focusing on things. So, let's talk about that, then.

Focusing is *hard*.

Like, really hard. Take it from someone who has had countless days on either end of the focus spectrum. Times when I couldn't do *anything* and days when I couldn't be stopped. And, days like the latter? I could wield focus like a secret weapon, devouring any task that dare stand in my way. Writing a book series seemed like a walk in the park during those times.

Or, how about when I'd *give anything* to focus at all? No matter the amount of espresso shots flowing through my veins, focus was fleeting. Before you know it, I noticed that Twitter was calling – and now I'd be mindlessly scrolling through a feed that has no end.

On focused days, you're unstoppable. When you're unfocused, even the most trivial task feels gargantuan. Can we even the odds, then, and try to force ourselves to focus?

Not really. The first thing to admit to yourself is that somedays, you just won't have it. But! There are some steps that I think you can take, some ways to tweak your environment and little habits to build which might help you be focused more often.

That's what this journal entry is – how I try to set myself up to focus most of the time. It doesn't always work, but hey – it *mostly* does. And if you're realistic with yourself, being focused *most of the time* is probably better than where most people end up at.

So here it is. All of my super secret ways to do things. They are all pretty obvious in hindsight, and there are only three of them. The order matters here, too. They are listed top-to-bottom, starting with what I consider most important, to still important, to hey – if you can't do them, you can't do them:

Mindset

If your mindset isn't at an ideal place, or least even an okay one, at the start of anything, you're toast. And, this is critical – because the way that you think about things shapes how you *feel* about them. And, you perform wildly different at tasks largely based on how you feel about them. It's just how we're wired.

Scared before a big speech in front of many people? Your palms sweat and maybe you talk a little too fast.

Are you calm before a big speech in front of many people? As calm as you *can* be? Because public speaking is still going to be intimidating to folks, no matter their mindset. But if you're as calm as *you* can be, whatever that looks like, you're going to do better.

Maybe your speech slows down a bit. You think a little more clearly. Why? Because you feel differently about it. It might still be scary, but it's a little *less* scary. And the delta between "This is absolutely terrifying" and "This is still scary, but I'm gonna do it!" is infinite.

So, start here first with focus. Frame your mindset. Want to ship an app? Don't think "I have a full-time job" or "I have too many responsibilities" first. Those things are, no about, all true – but we have so much more time than we think. Yes, put all your responsibilities first – but then build some mental resolve to *find* tiny little cracks in your schedule to do things you might be passionate about.

Do you work from 9-5? Try to start an hour early, and then you have 5 hours a week to work on your hobby. Maybe flip – do an hour later! What could you do with 20 hours in a month?

No matter how you do it, it all starts with how you think about it. If you think it'll never happen, and it'll never get done – congratulations! – you're absolutely right. When you don't believe you can do much, focusing on anything becomes difficult.

But, if you think you can *eventually* get something done, slowly and at a healthy pace that respects burnout and your mental capacity – then hey, you probably will get it done. On days when this book series nearly overwhelmed me, I remembered that it was a marathon. It'll take a long time. And critical things such as my job and my family will always come before it – but over time, I will finish it.

And I did.

Constraints

Time box yourself! Give yourself a date! Do all that you can while following the advice above to really get things moving.

While writing the book series, I promised an update every two weeks. There were small exceptions to that. If I was sick or on vacation, then that didn't apply. But, over two and a half years – I stuck to it.

Guess how this book series would've gone if I had not done that? The answer is obvious – I would not have finished it. Though we don't tend to enjoy deadlines at work, those are *external* constraints. And, yeah – those aren't fun most of the time.

But *internal* constraints, ones that you place on yourself, can be powerful. Future you is telling present you to get some progress towards something that's near and dear to your heart! Listen to future you! Even if they annoy present you occasionally.

Worst Comes First

And finally, no matter how fun your hobby is that you're trying to focus on – there will be tasks that simply are not fun. There is a term often used, "Eat your frog", which applies here. It basically means this: Do the worst, least fun thing first.

Want to ship your app, but you need to do App Store assets? Do that now! Don't sit on it! When you knock out the things that you *have* to do, you open yourself up to do things you *want* to do – and with more focus, too.

What's more fun than working on stuff you actually want to get done? Doing those things while knowing that something you are dreading doing isn't hanging over your head. It's so freeing, that in a way – I look forward to starting my side project time by knocking out a task I have no interest in first.

Because after that? It's off to the races.

And Finally

No matter what focus looks like for you – give yourself a break. Focusing with real life responsibilities and circumstances knocking you around is hard. It doesn't always work. But, it can occasionally. And with these tips, I hope you find a way to do something that's important to you. Try to focus, but show grace to your situation and tamp down unrealistic expectations.

Forget about what everyone else can do, is doing or has done. What can *you* do with some more focus? I bet you'll surprise yourself.