Series Introduction

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.

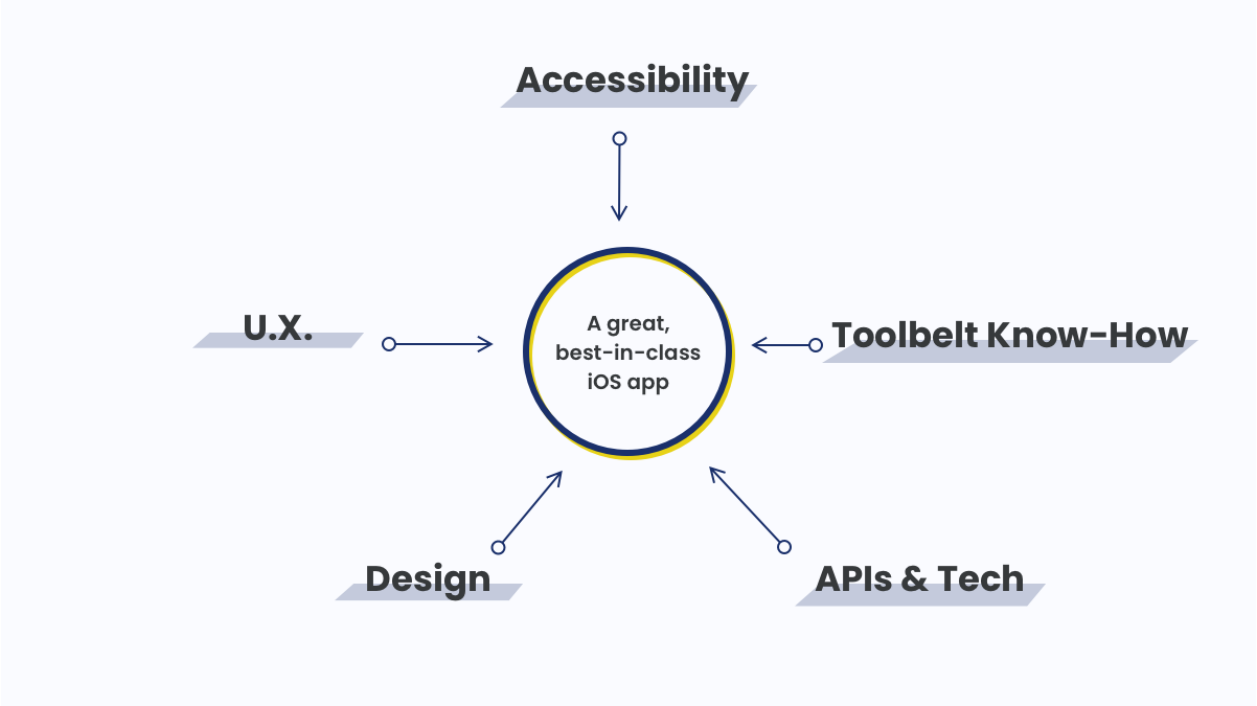
With that said, you'll find that each section can be written in one of two ways; Either a traditional, long-form explanation to explain the current section or topic, or a three-pronged approach covering the what, how and some tips on the subject. This is not so much a tutorial book, though you'll certainly 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 frame-

works) 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.

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:

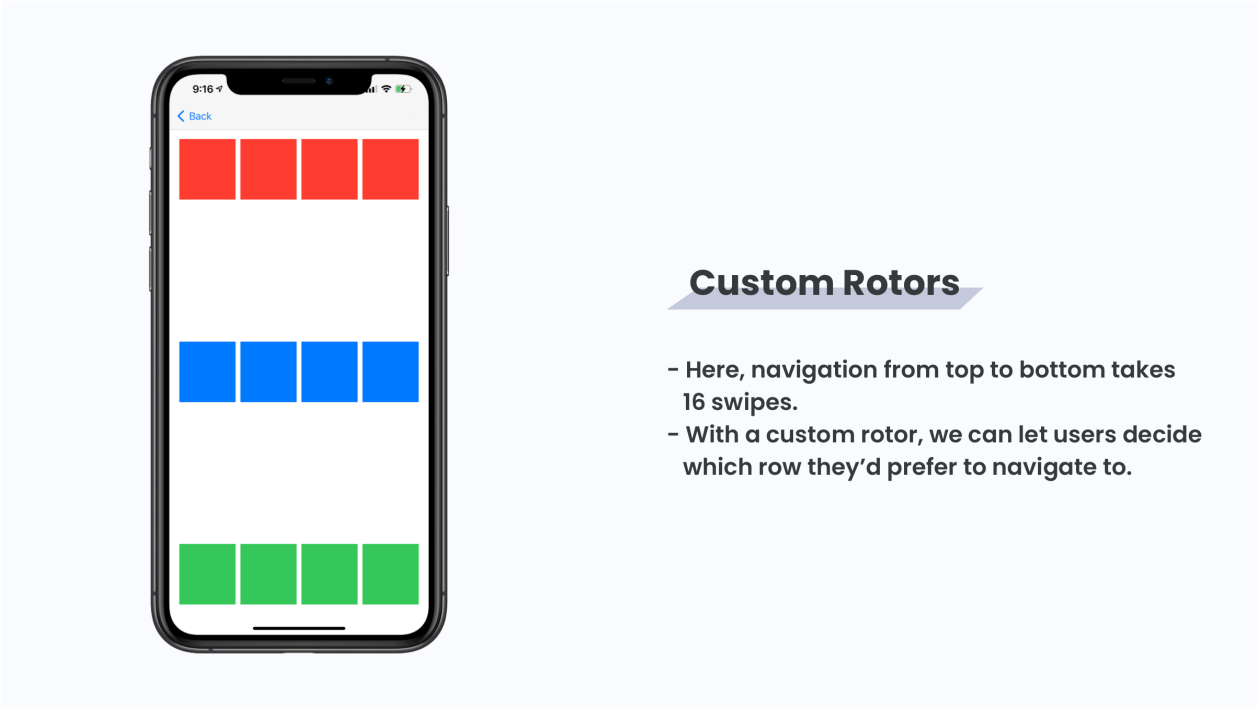
1. The name or type of the rotor
2. The next element that it should navigate to

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.

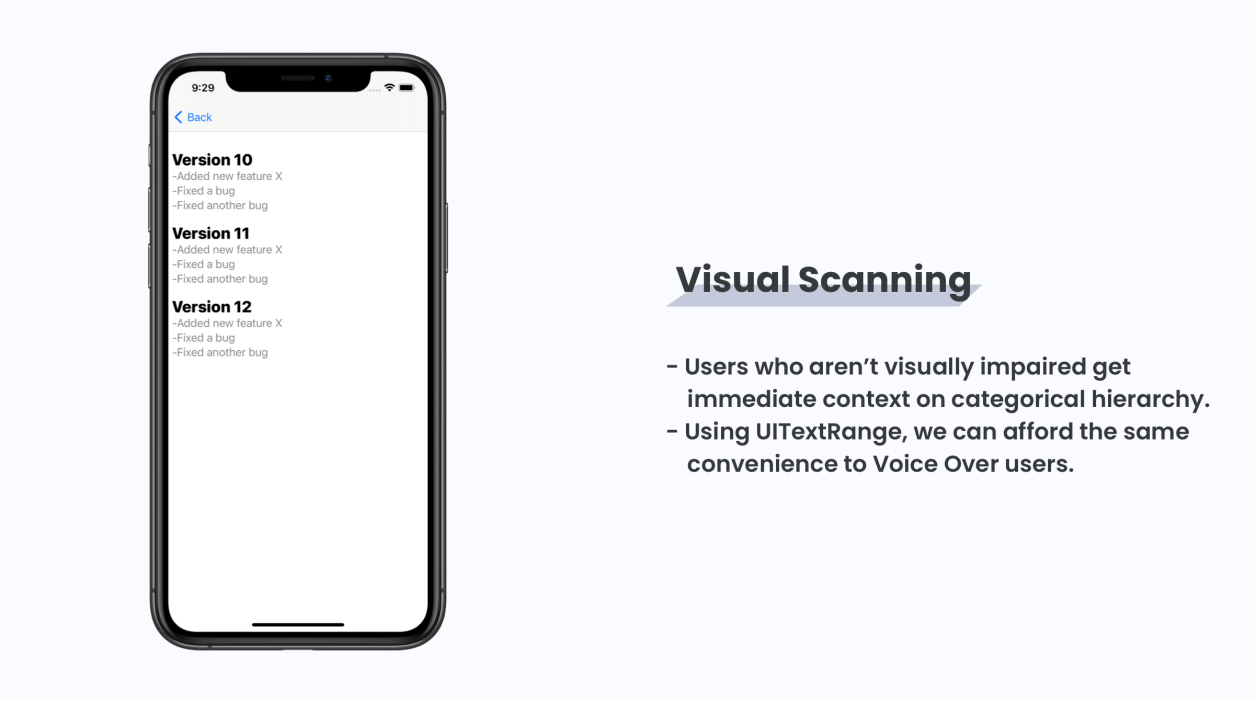
If we had implemented a single rotor instead of three (one for each row), we could have one single rotor called something like “Color Rows”, where each swipe up or down would take you to the next row and swipes left and right would navigate within them. It’s up to you to figure out which ways to implement 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[1]. This implementation requires a bit more tact that the one above,

[1] 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.

so if you can reconfigure your view setup to support the previous way of supporting a rotor control - by all means, do so.

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

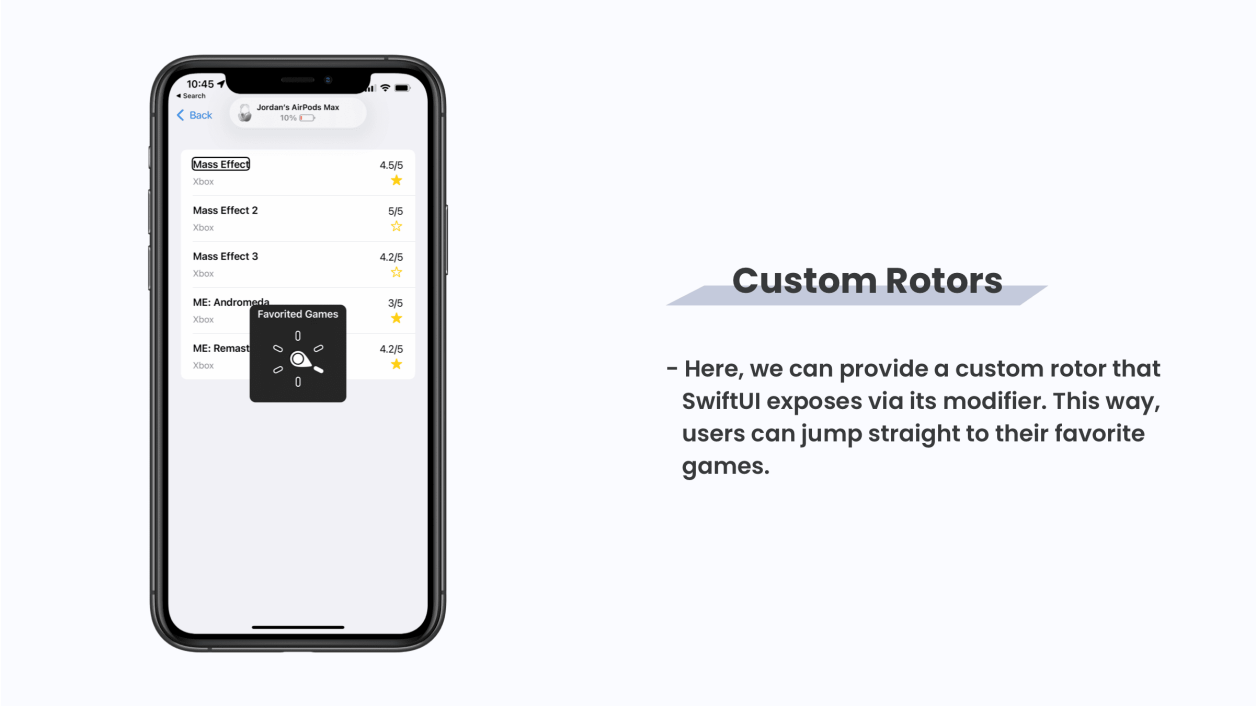
Consider an interface of video games that can be favorited. We could make a custom rotor to navigate *only* to favorited games. Looking at the model for video game, we could leverage the `isFavorited` property to handle this.

```
// 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.

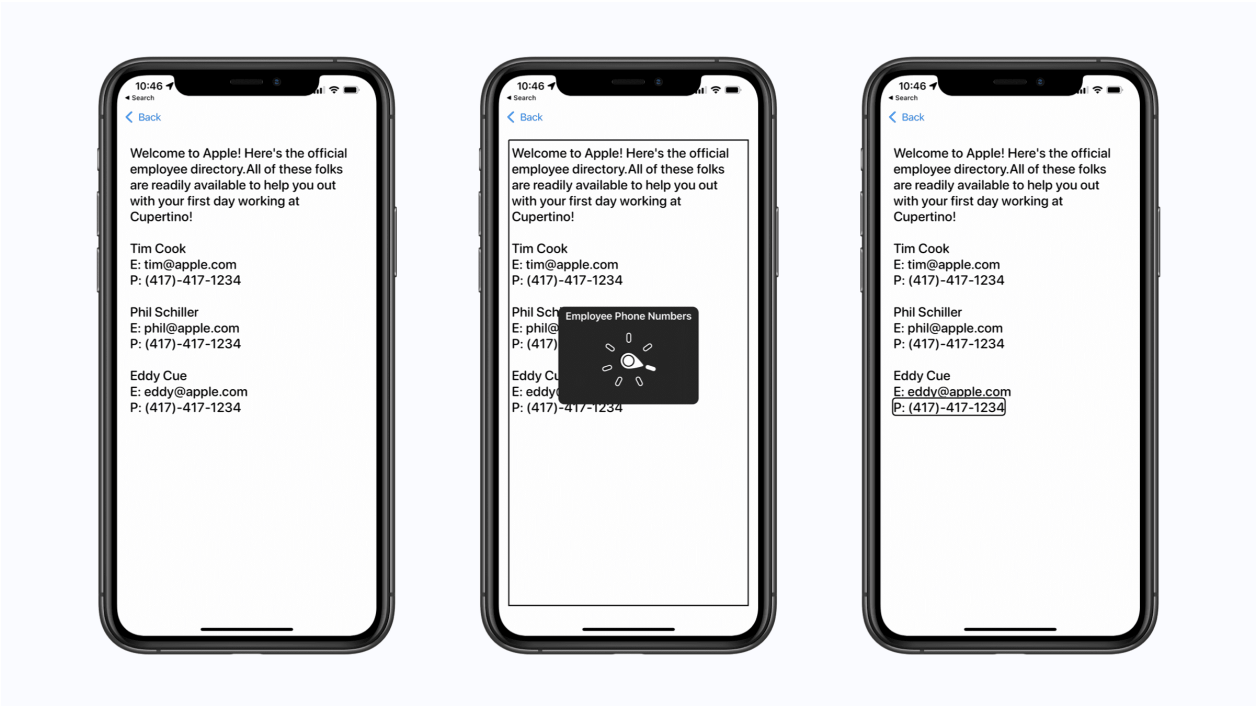
Each approach ties its utility to the amount of scale you require. For most views, the first two approaches should suffice. But for cases where the views don't have an implicit container using identity or they might span multiple views due to SwiftUI's inherent knack for composition — the last approach 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

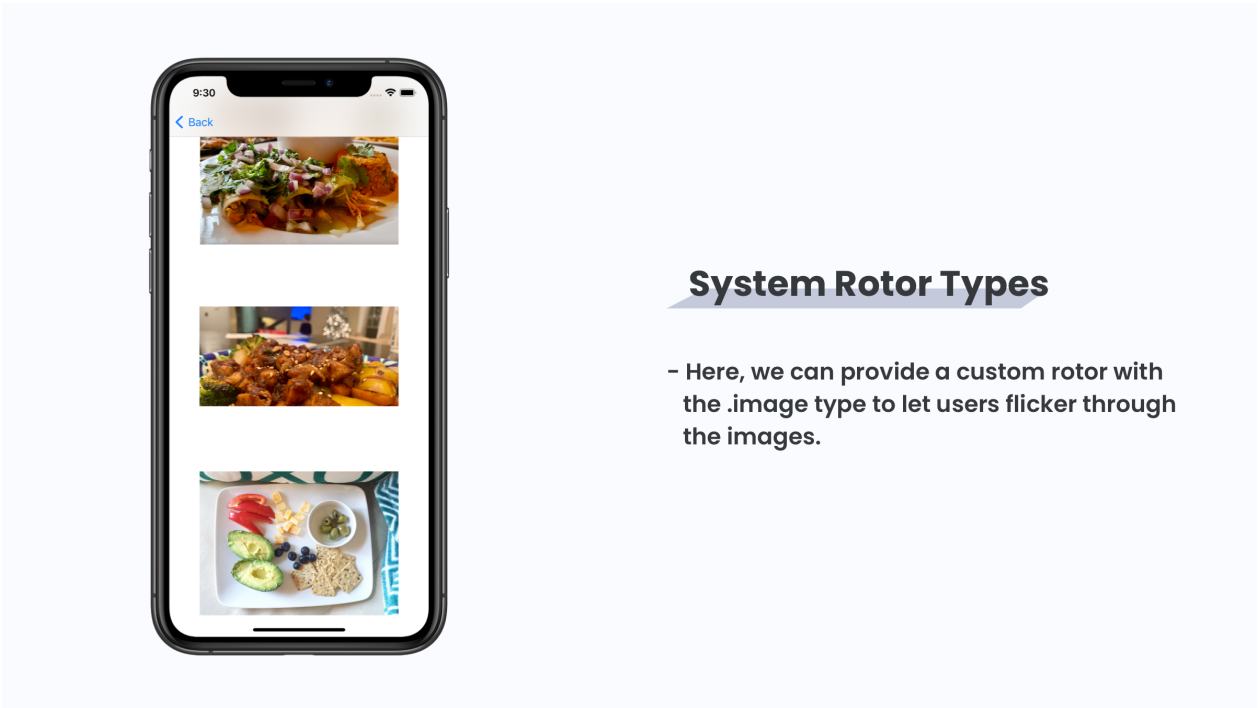
The rotor, in a sense, identifies where to go next categorically. For example, in most interfaces when the rotor is activated you'll likely see "Heading" as an option. As I pointed out above in "How it Works", that means you need heading level elements to have that accessibility trait so it'll be exposed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

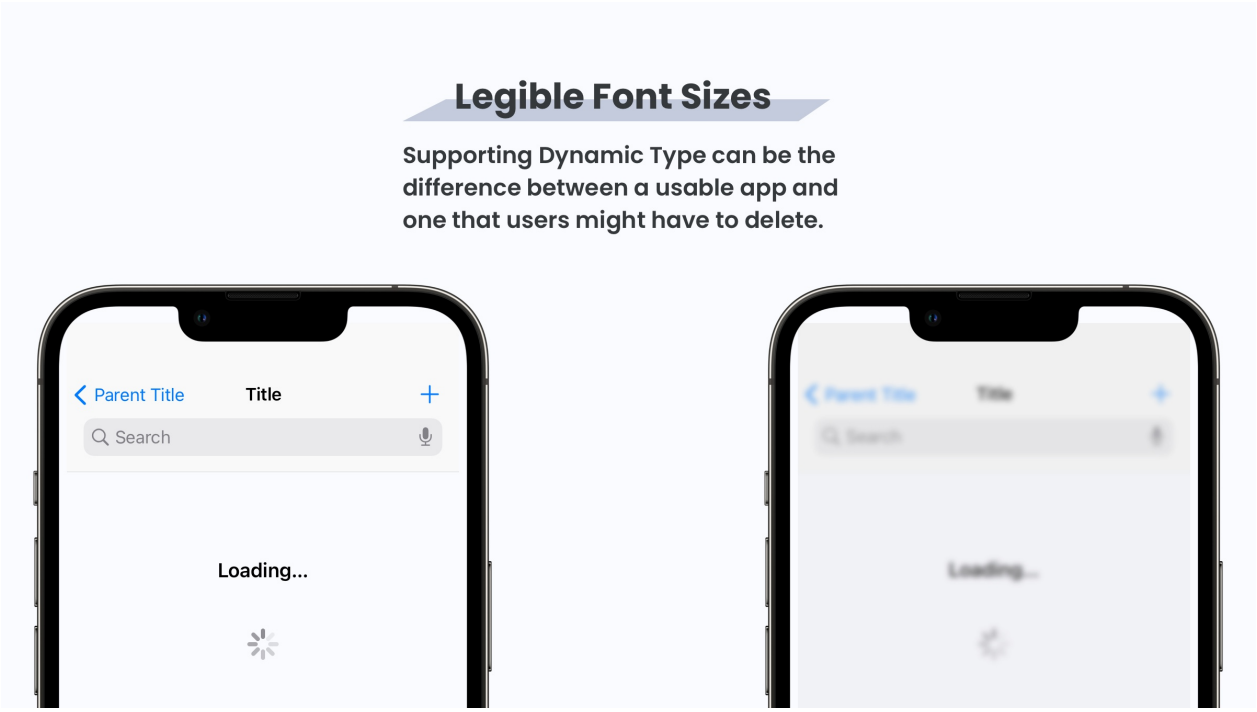
Three Key Takeaways

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.

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.

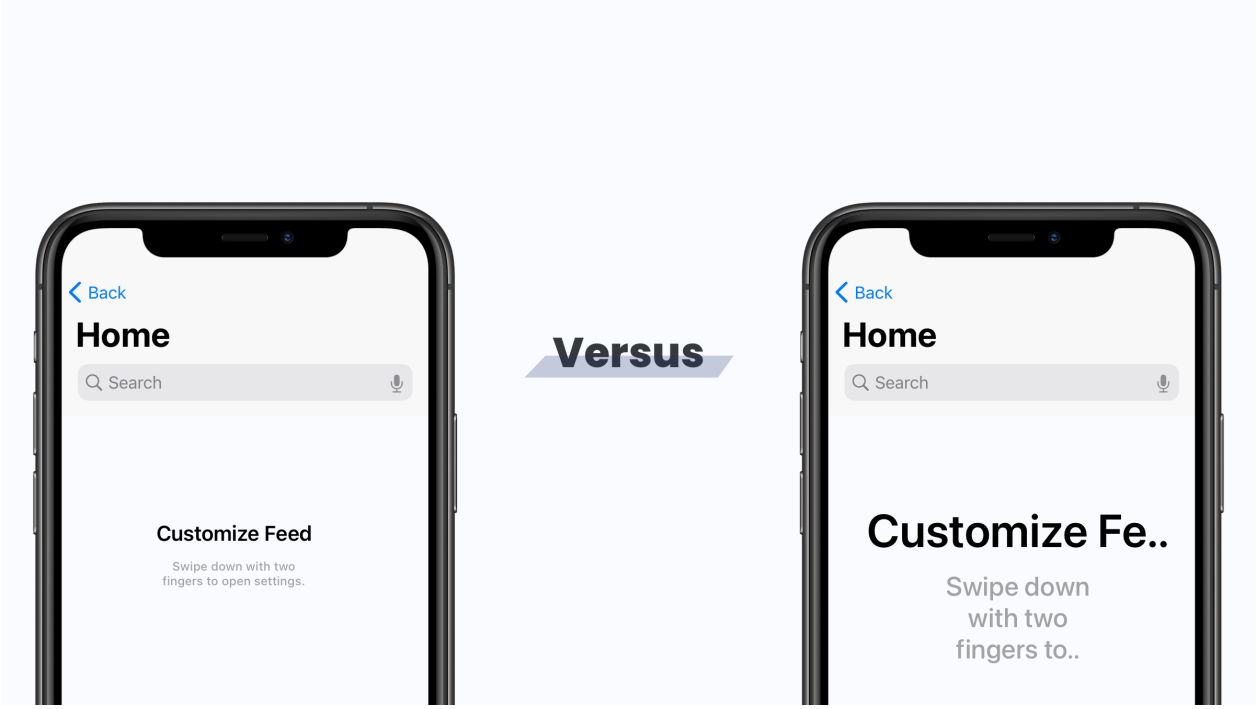
To make the process as simple as possible, I've always followed a simple set of rules when it comes to text in my own apps. These exist to ensure my text stays accessible, usable and readable:

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.

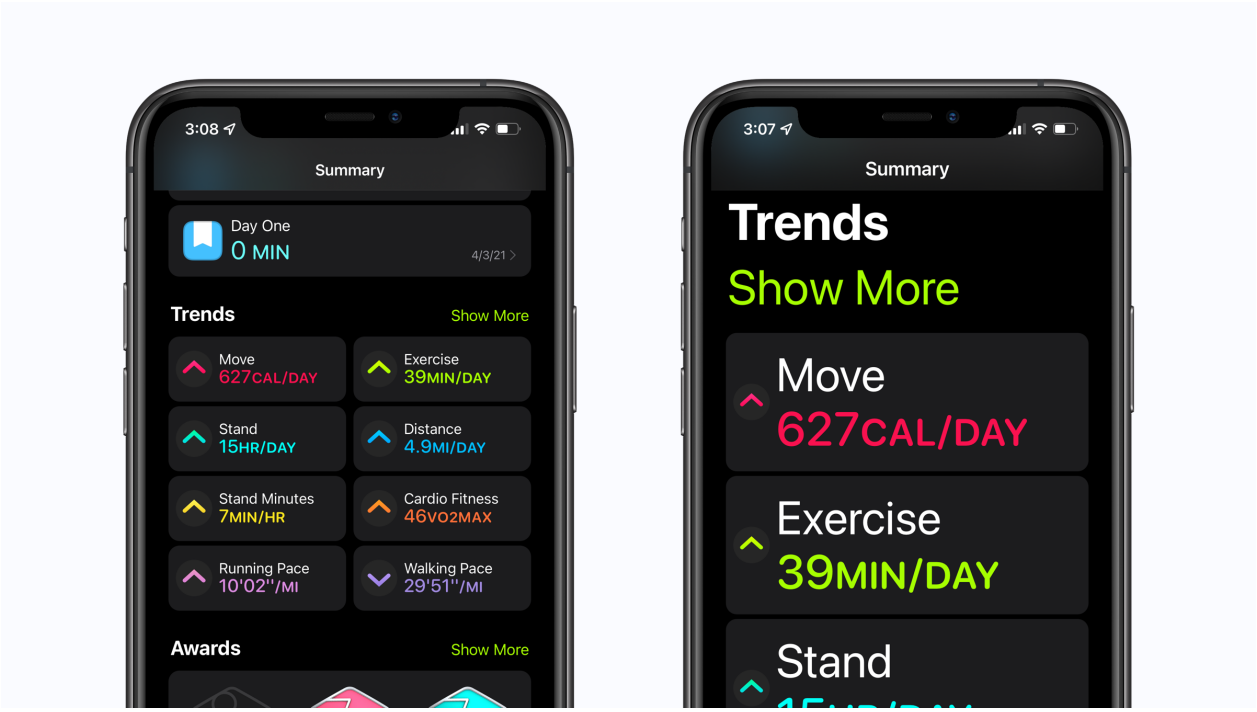


Clipping or truncating text is rarely a good idea.

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

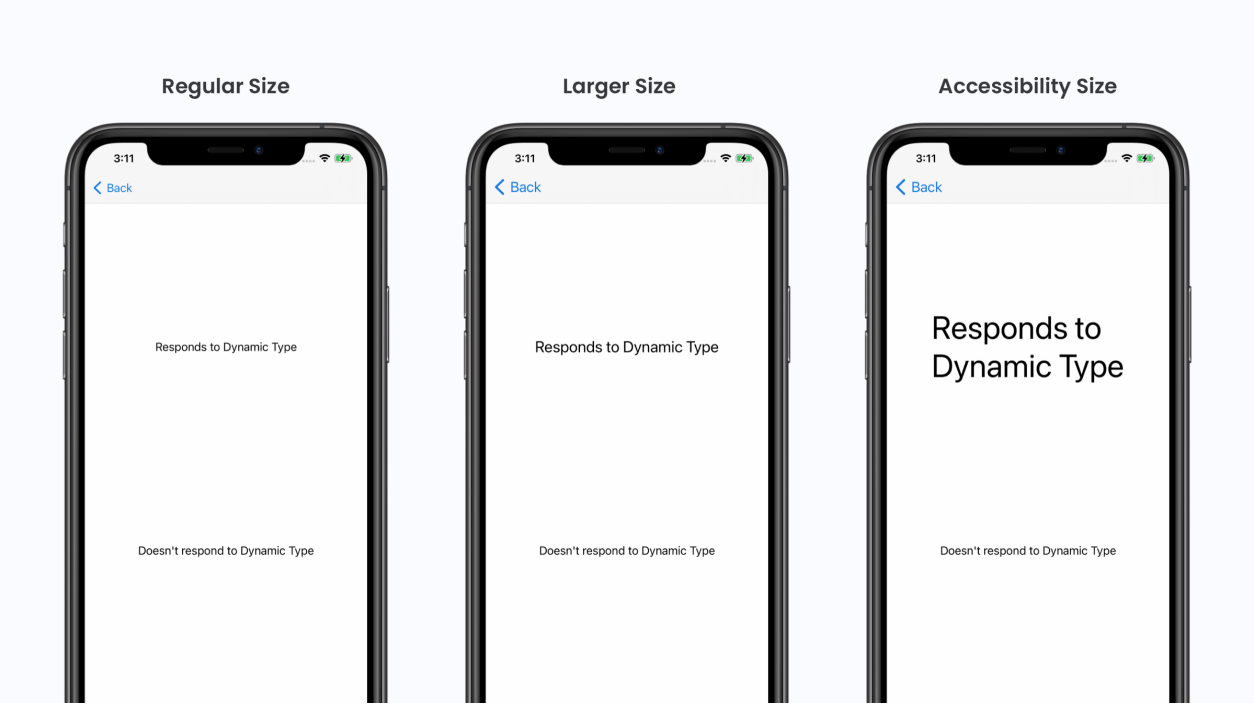
To enable dynamic type in UIKit's three main text controls (UILabel, UITextView and UITextField) you need to do two things:

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:

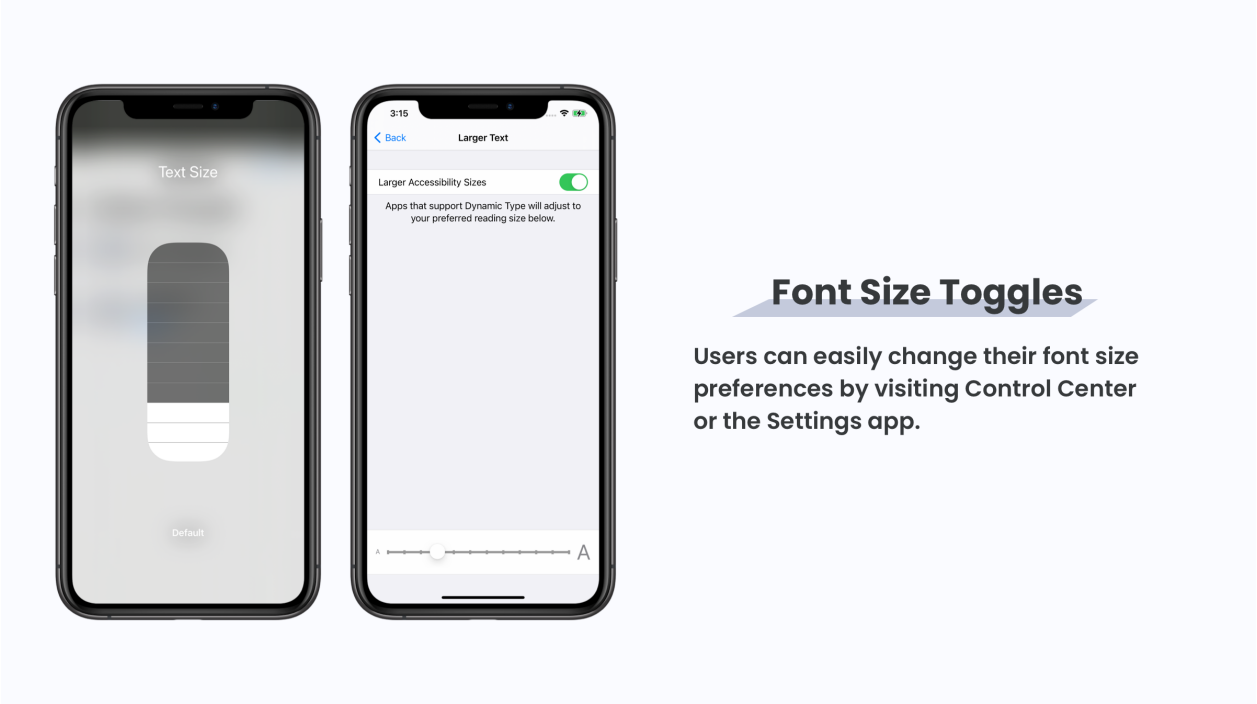
```
// UIKit -> Xcode -> DynamicTypeFig1ViewController.swift
let dynamicLabel = UILabel(frame: .zero)
dynamicLabel.numberOfLines = 0
dynamicLabel.text = "Responds to Dynamic Type"

// 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 overriding `traitCollectionDidChange(_ previousTraitCollection: UITraitCollection?)`
2. By observing `UIContentSizeCategory.didChangeNotification`

By overriding the `UITraitCollection` function 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 show-

ing before 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

override func traitCollectionDidChange(_ previousTraitCollection:
UITraitCollection?) {
    super.traitCollectionDidChange(previousTraitCollection)
    print("Changed from \(String(describing:
previousTraitCollection?.preferredContentSizeCategory)) to
\(traitCollection.preferredContentSizeCategory)")
}
```

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 logic, 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: UIFont) -> UIFont`

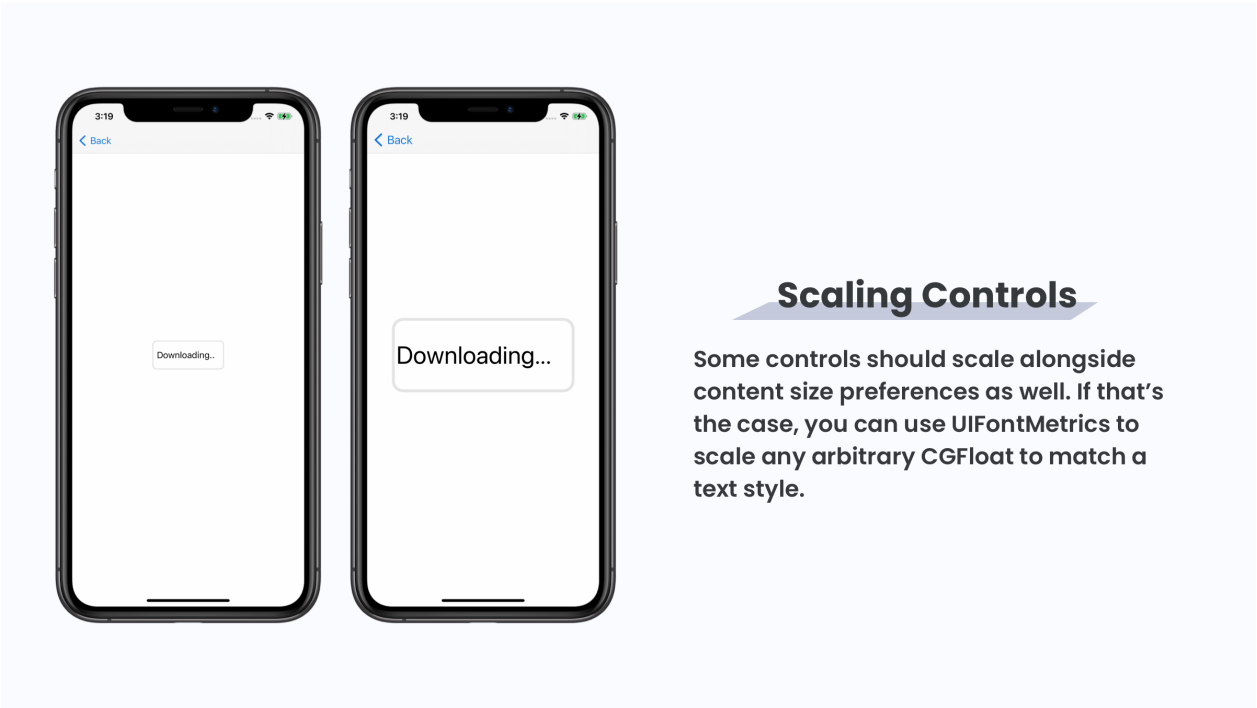
```
// UIKit -> Xcode -> DynamicTypeFig2ViewController.swift

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:

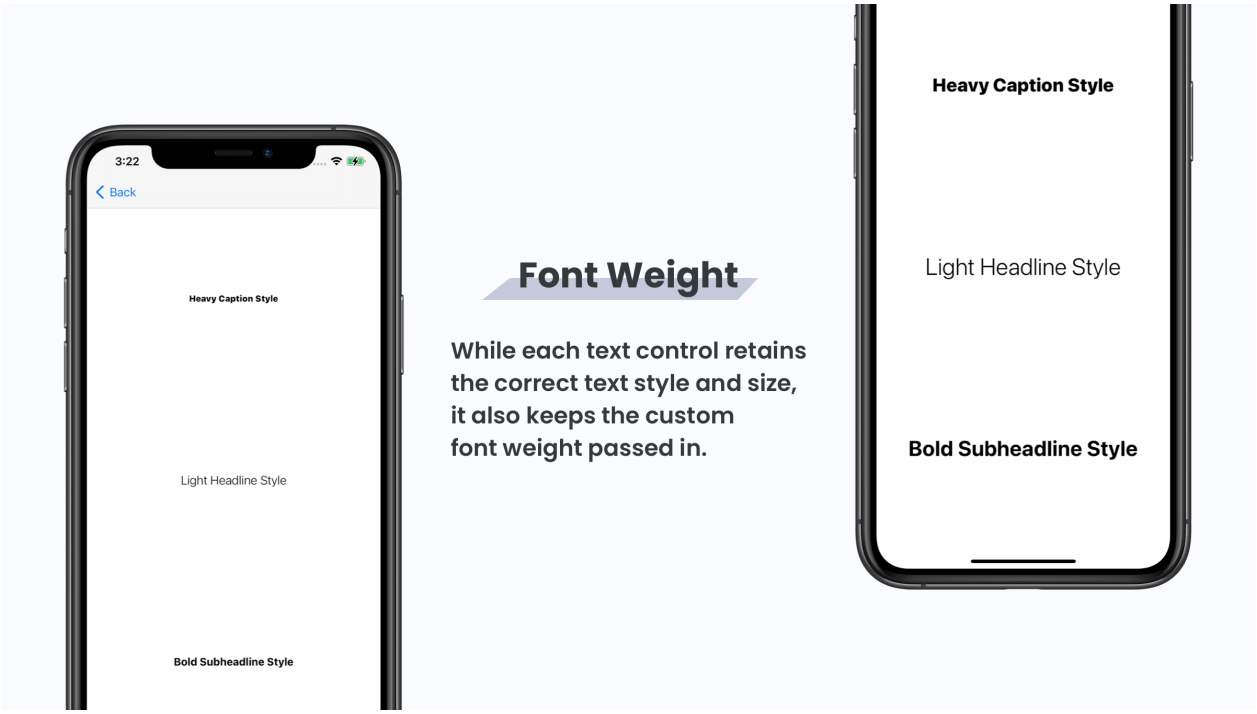
```
// This is heavier...
UIFont.preferredFont(forTextStyle: .headline)

// 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)
```

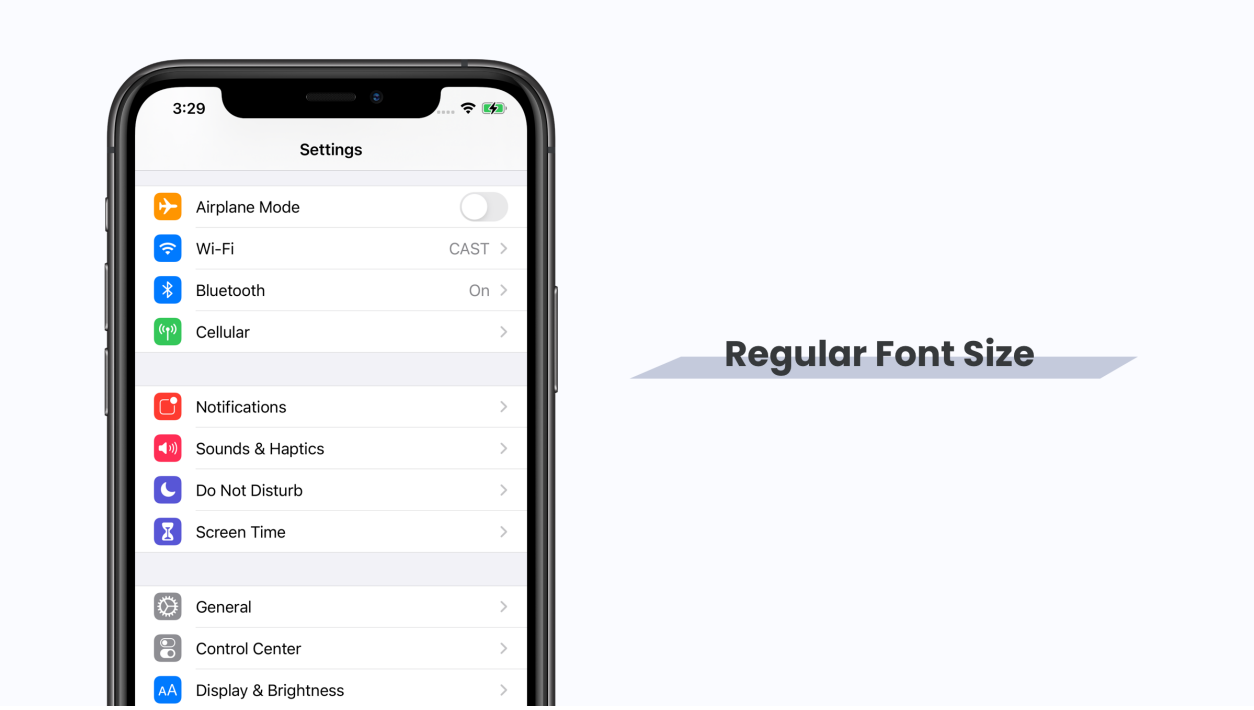
In SwiftUI, you can get this same behavior if you *don't* apply a specific font size but do request a particular weight using the `fontWeight(_:)` modifier. When it comes to Dynamic Type flexibility, SwiftUI makes it much easier:

```
// Still adapts to dynamic type
// You can stil request a text style *and* a font weight
// And it'll respond to content size changes
```

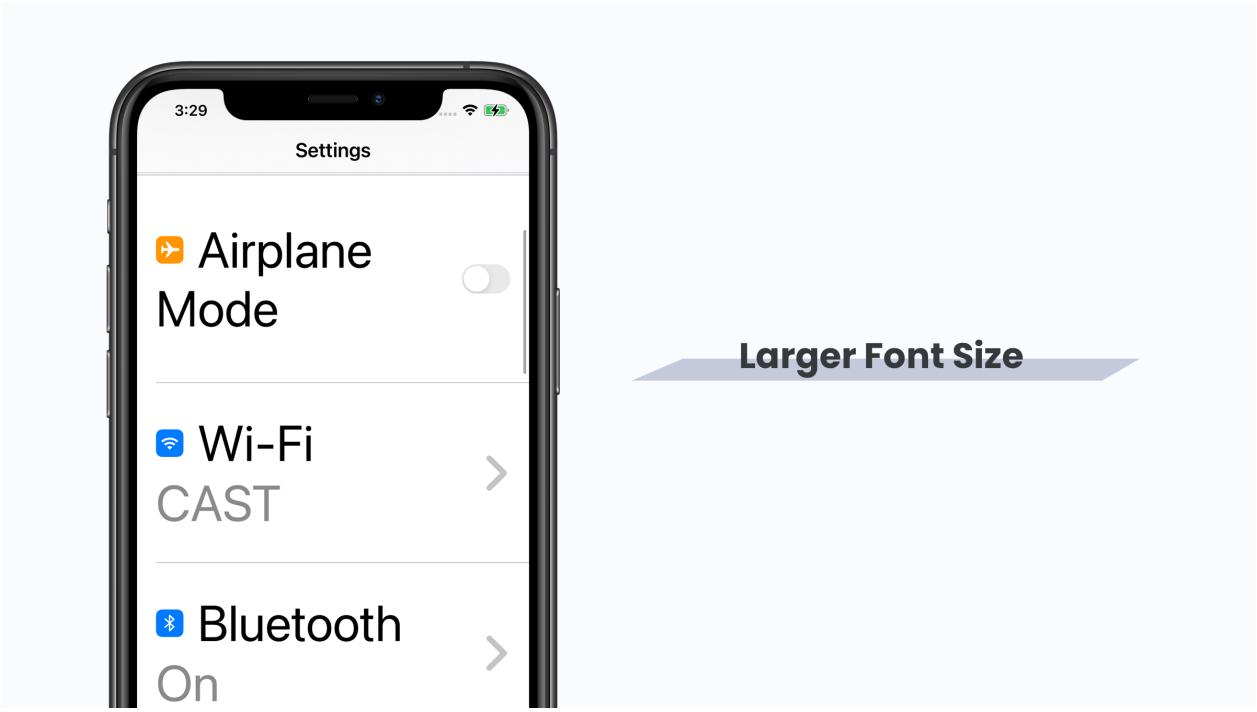
```
Text("Dynamic Type")
    .font(.body)
    .fontWeight(.thin)
```

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 to do the same check:

```
// UIKit -> Xcode -> DynamicTypeFig5ViewController.swift

override func traitCollectionDidChange(_ previousTraitCollection:
UITraitCollection?) {
    super.traitCollectionDidChange(previousTraitCollection)
    let isAccessibilitySize =
traitCollection.preferredContentSizeCategory.isAccessibilityCategory
}
```

Though, this could report a different preference than the one `UIApplication` will have since trait collections can be override.

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`:

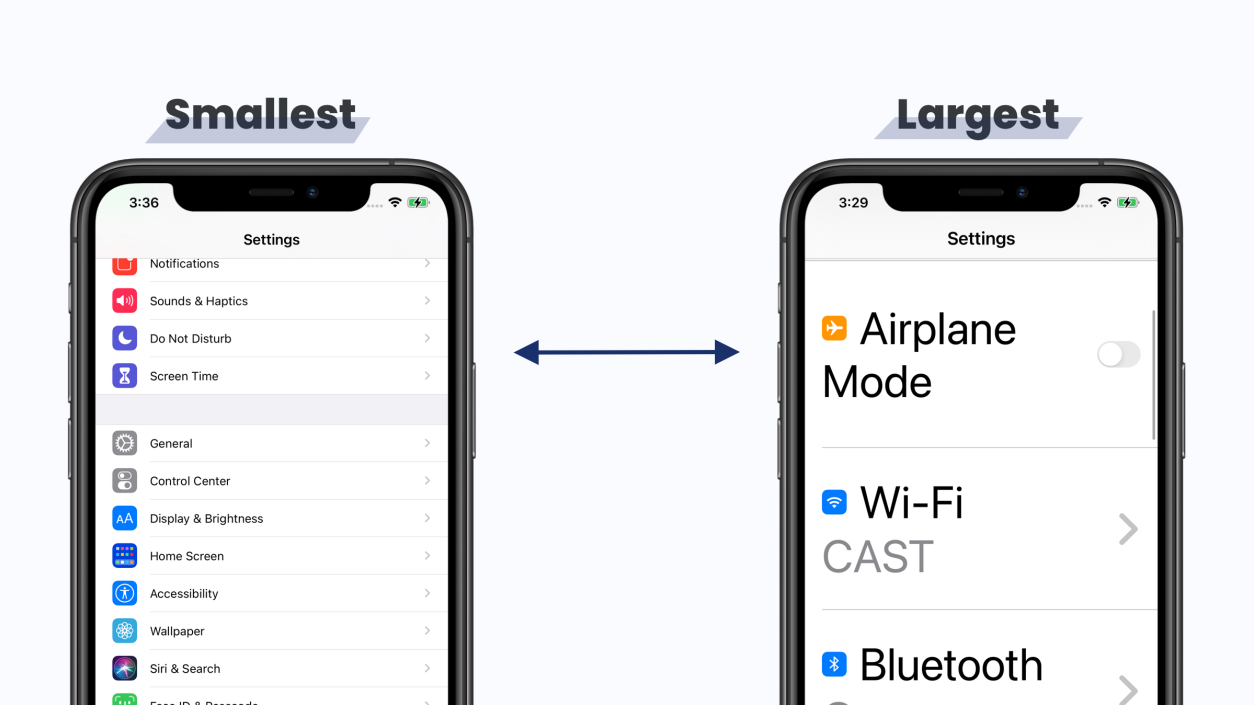
```
 3. <UIView:0x13b112930>: L <= (none->)L <= XXL
 2. <UIStackView:0x138f20310>: L <= (L->)L <= XXL
 1. <UILabel:0x138f23540>: L <= (L->)L <= XXL
--> L
```

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.

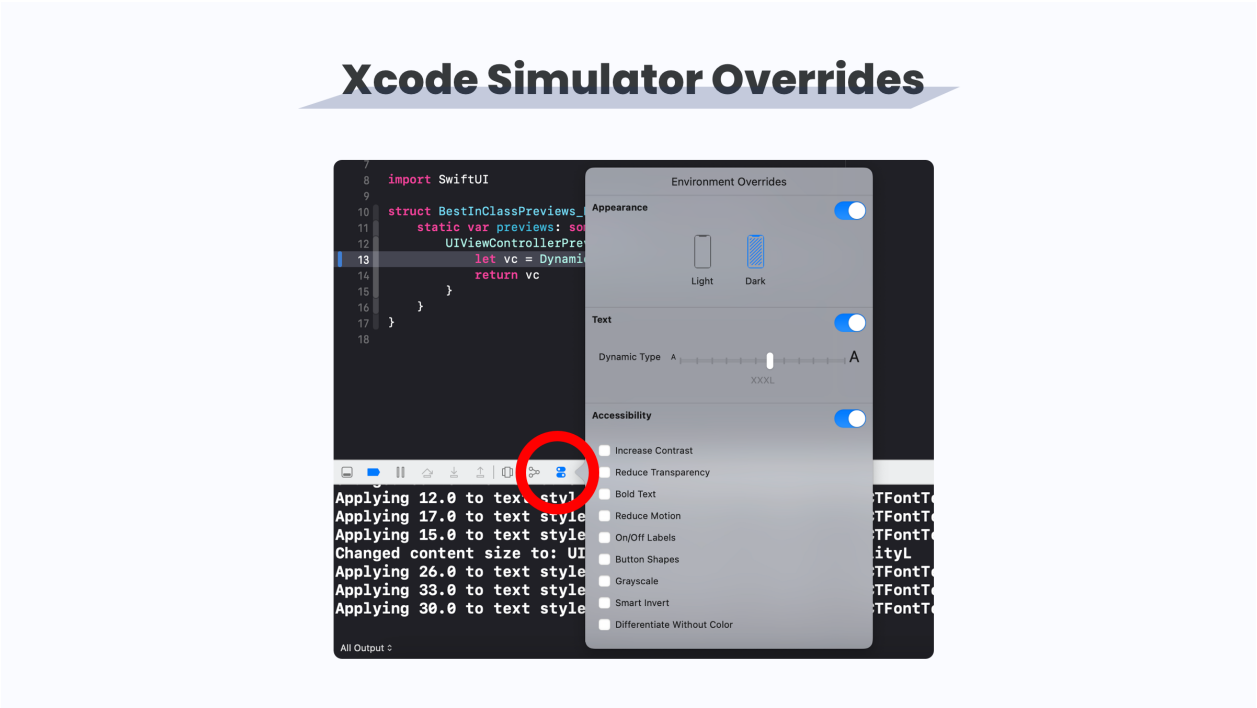
No matter which of the following methods you use, it helps to follow a "Lower Bound Upper Bound" rule. That is, test your app with a very small Dynamic Type setting, and then test with the largest option. If your app handles both gracefully, anything in the middle will do just fine as well.



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-

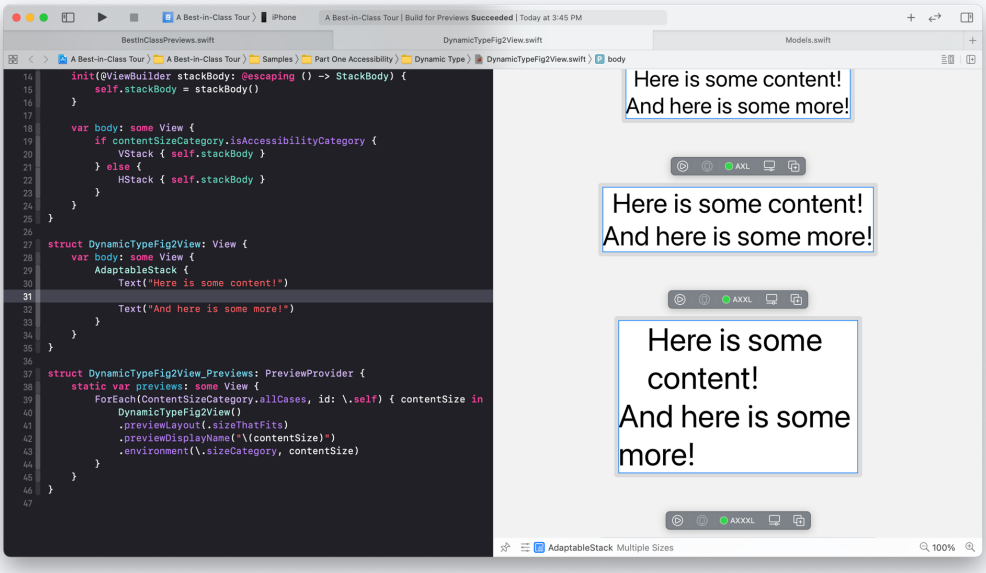
self "stuck" design wise with Dynamic Type, visiting one of their stock apps is usually a good place to start.

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!
AXXXL
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.

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.

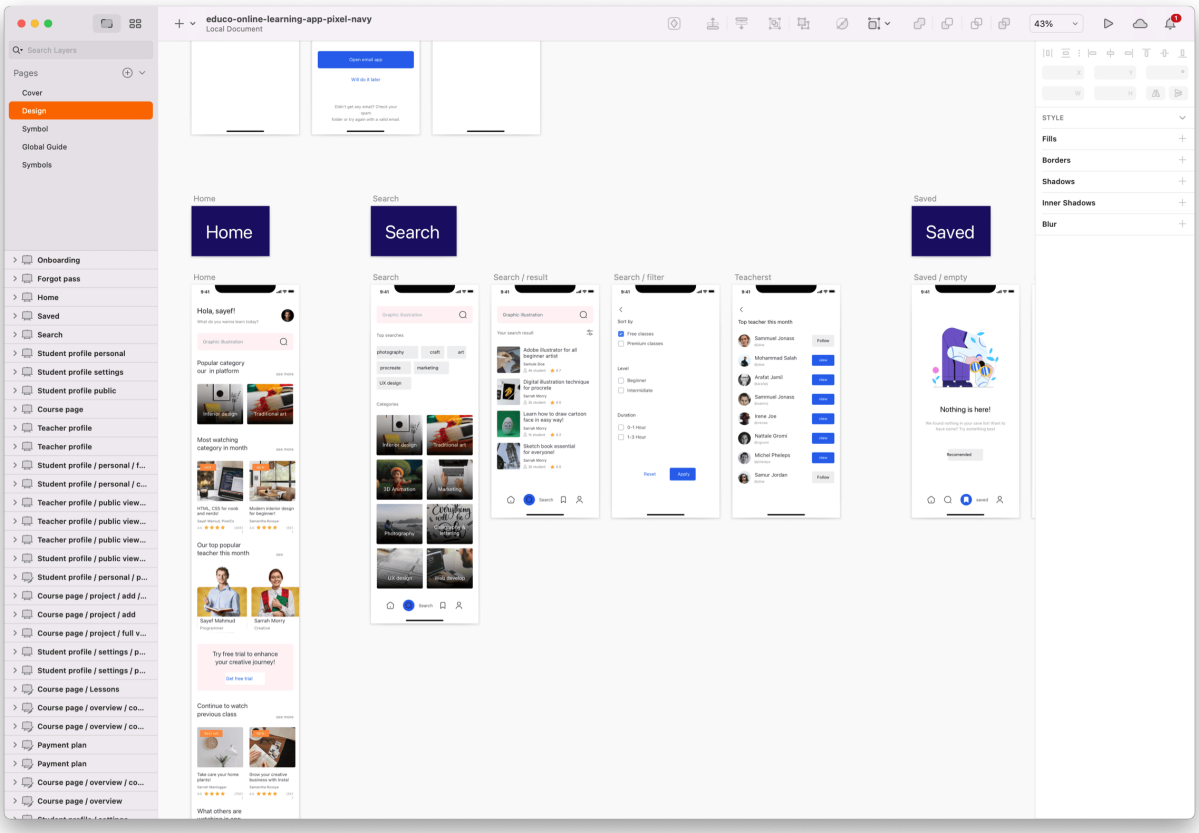
Of course, neither approach is wrong. It's about opening your eyes to both of them, though, and figuring out a way forward. A lot of times, we lean on our strengths at this point, too. Developers tend to dive right into code, whereas a designer at heart relishes the chance to create a high fidelity vision 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

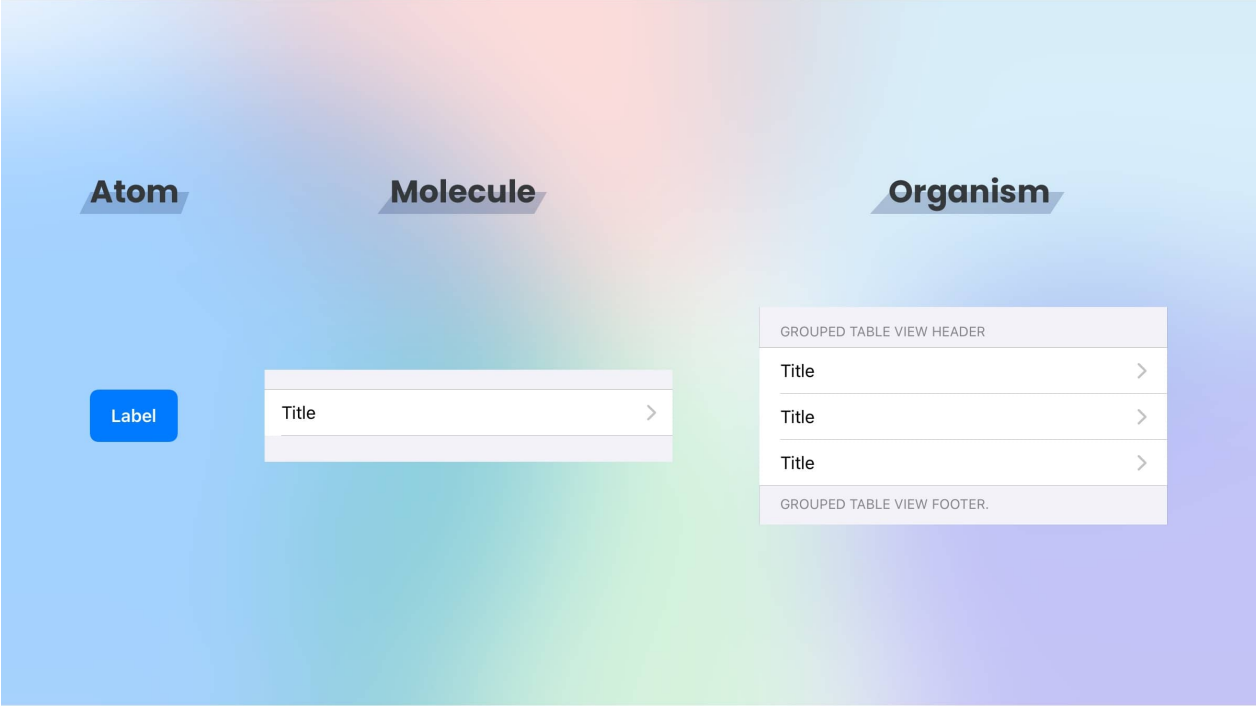
A high fidelity design is, in many ways, like a blueprint for developers. It tells them what to make. As such, it's a lot easier to reason about technical questions, too.

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!

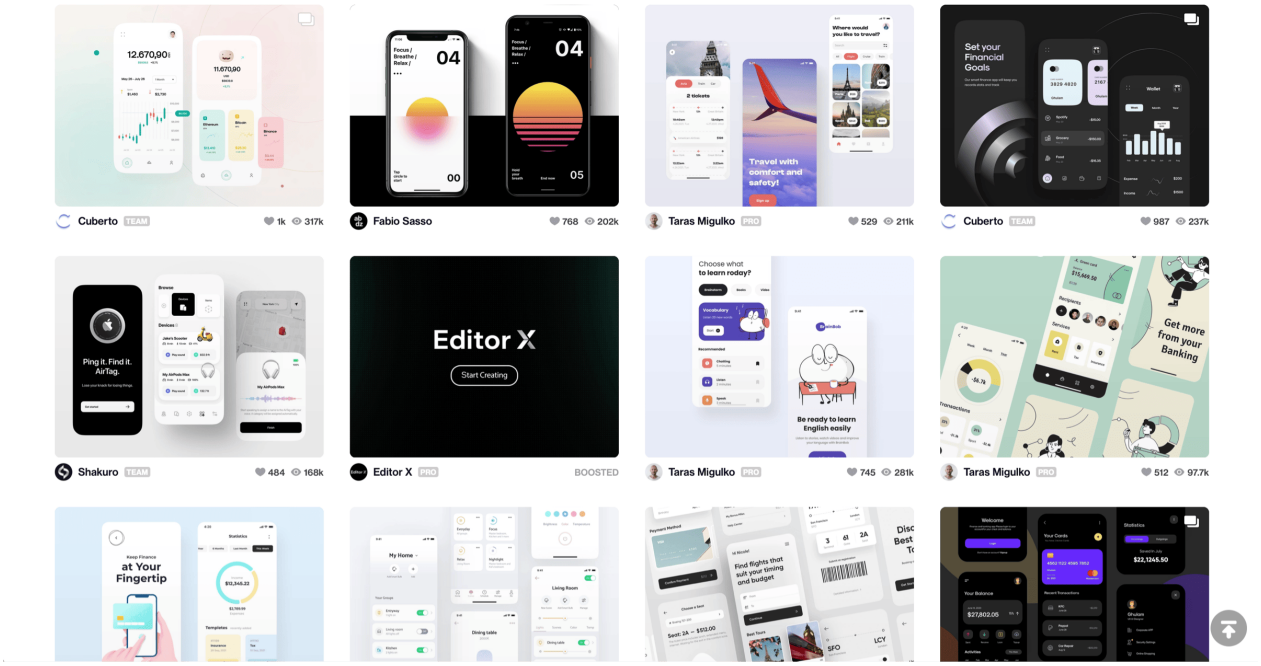
In fact, I'd argue that if you do well with high fidelity designs, then they *should* take quite a bit of time to make. It means you've thought about colors, active versus inactive states, flows and more.

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 my 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.

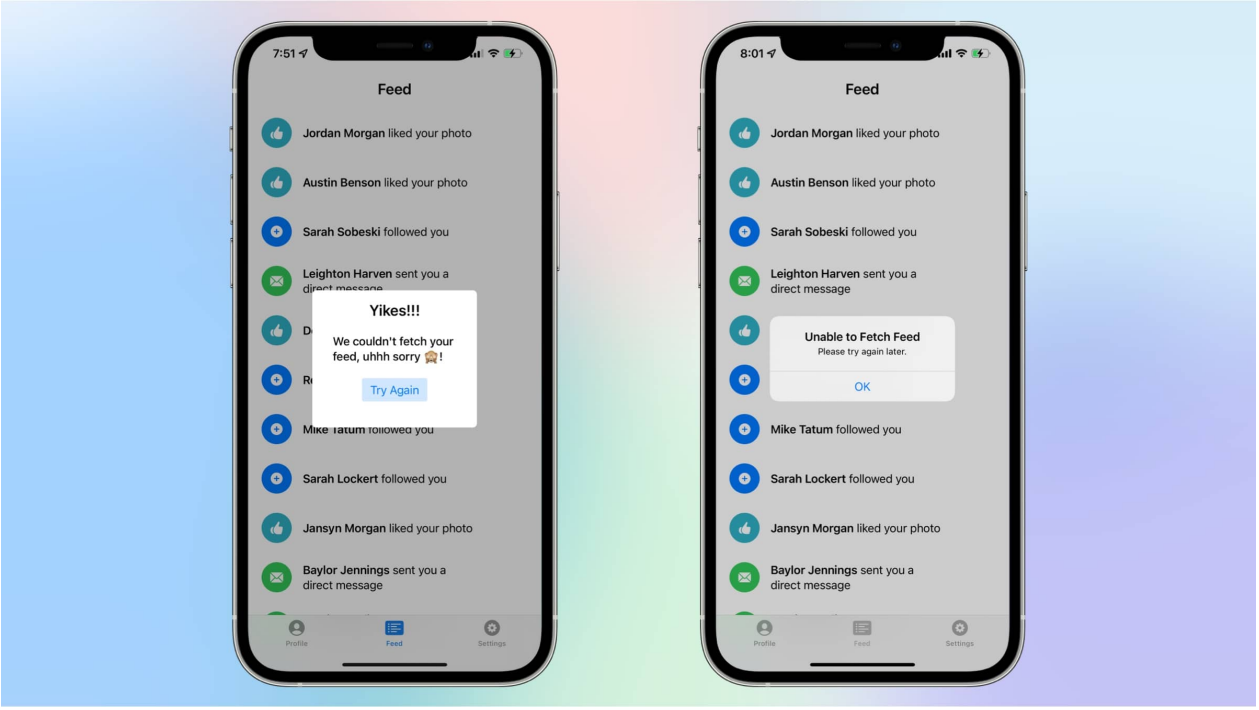
Holistically, it sets a tone for the app. Tone equals voice, and vice-versa.

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

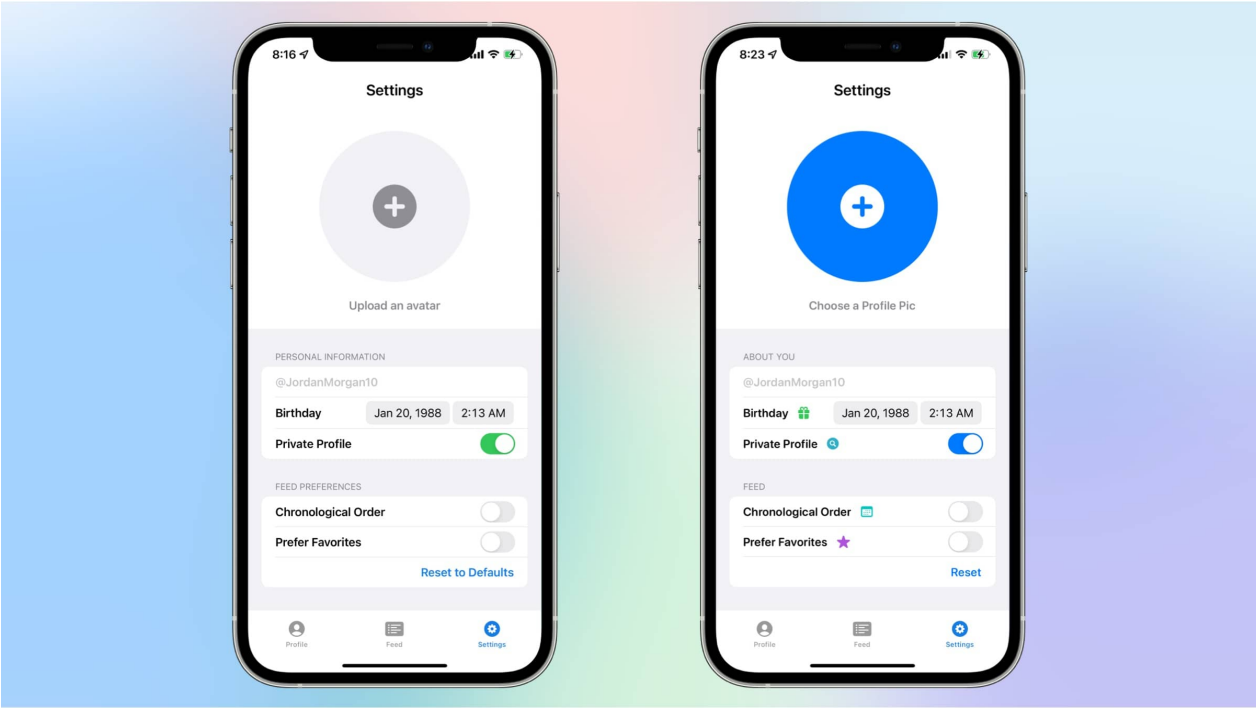
Be cognizant of the places your app displays any sort of tonality. You may be surprised at how often you have an opportunity to convey tone. It happens more than you think.

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:

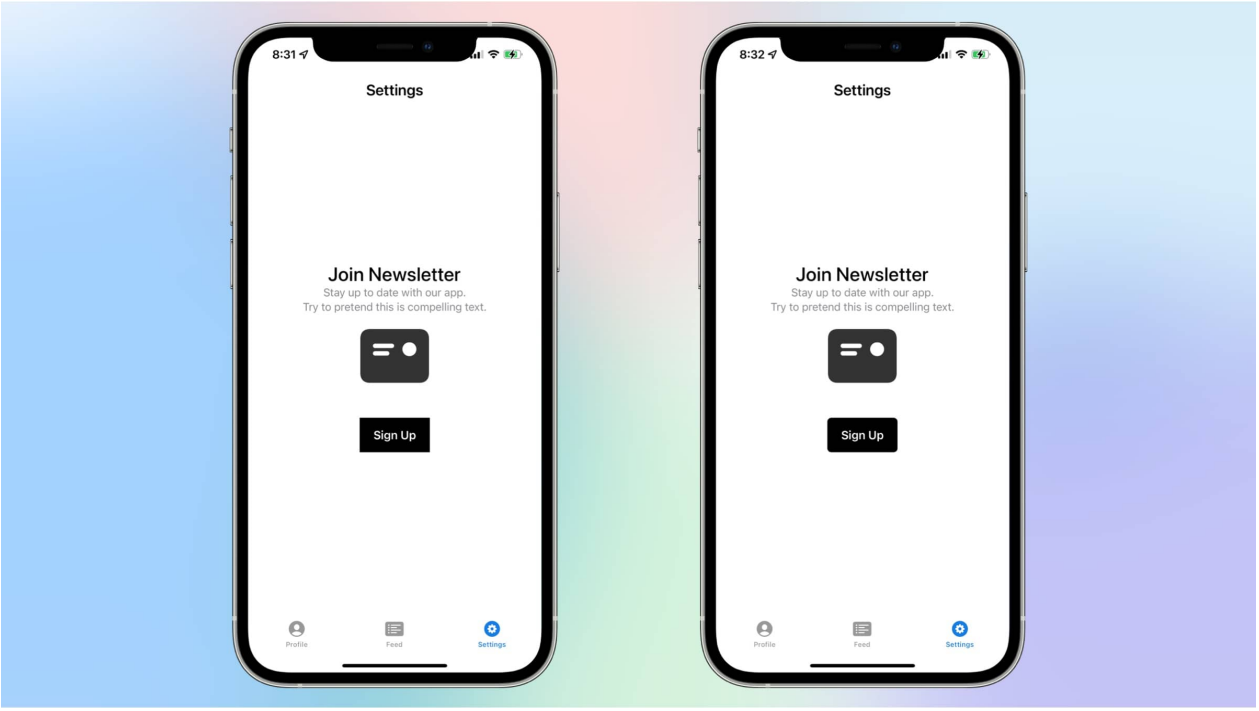


You get the same vibes as before, don't you? One feels like all business, while the other one feels playful and less serious. All that from simply changing colors, font and some text around.

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.

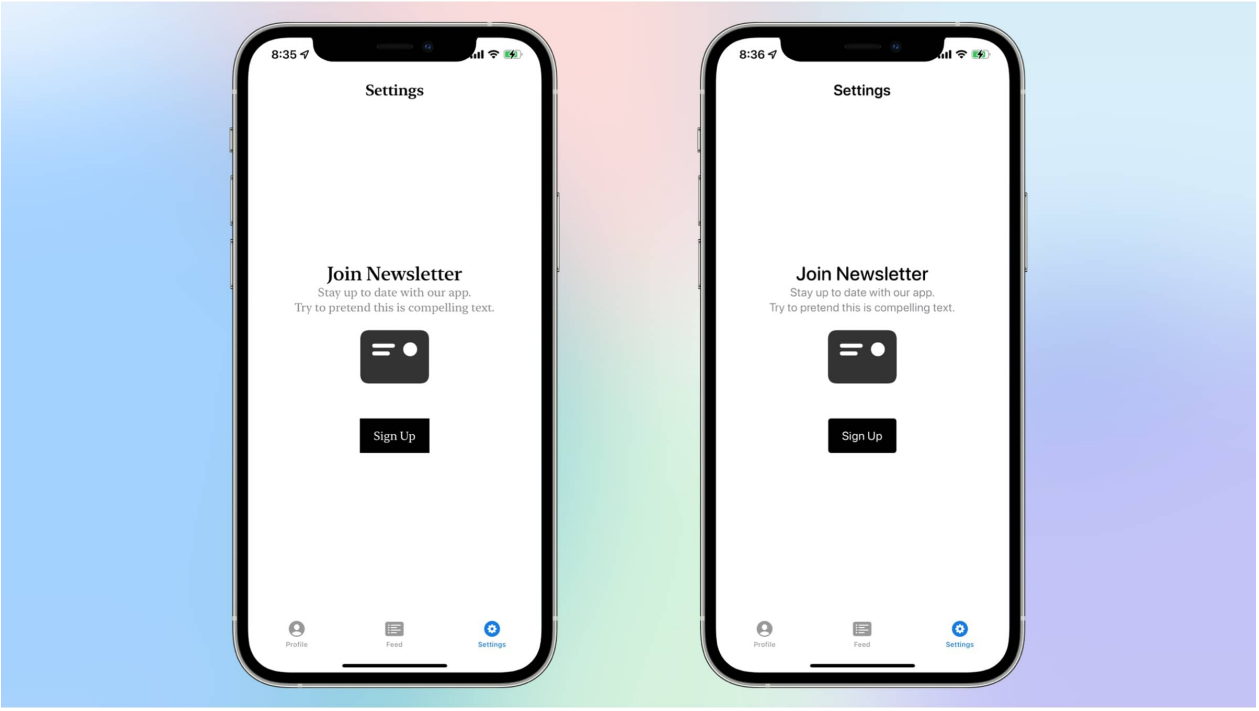
Colors. Typography. Copy. Navigation. We like to say these are functions of design, but I believe they are functions of an app's voice first. Because once you figure that out, the design comes next.

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.

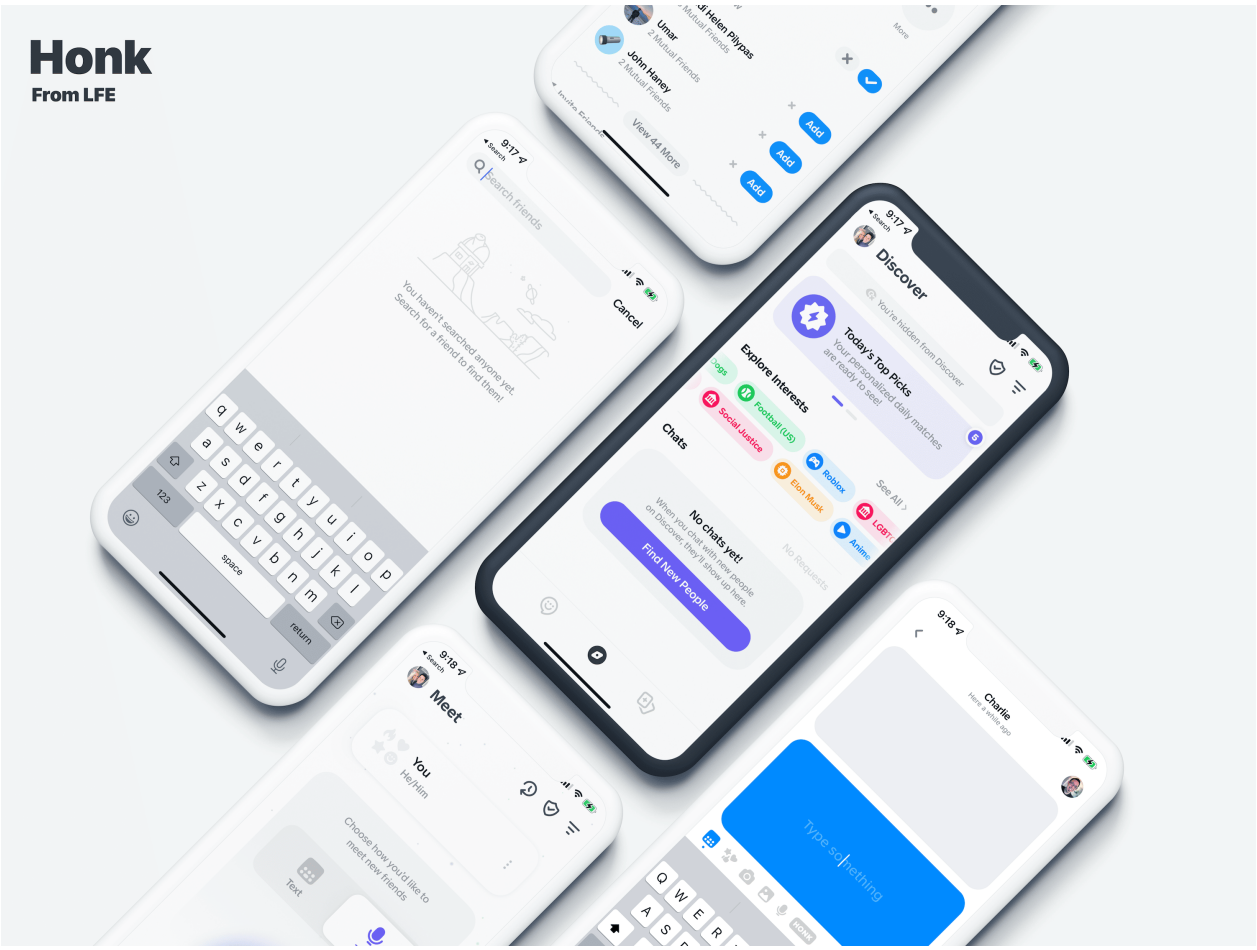
Types of Voices

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:

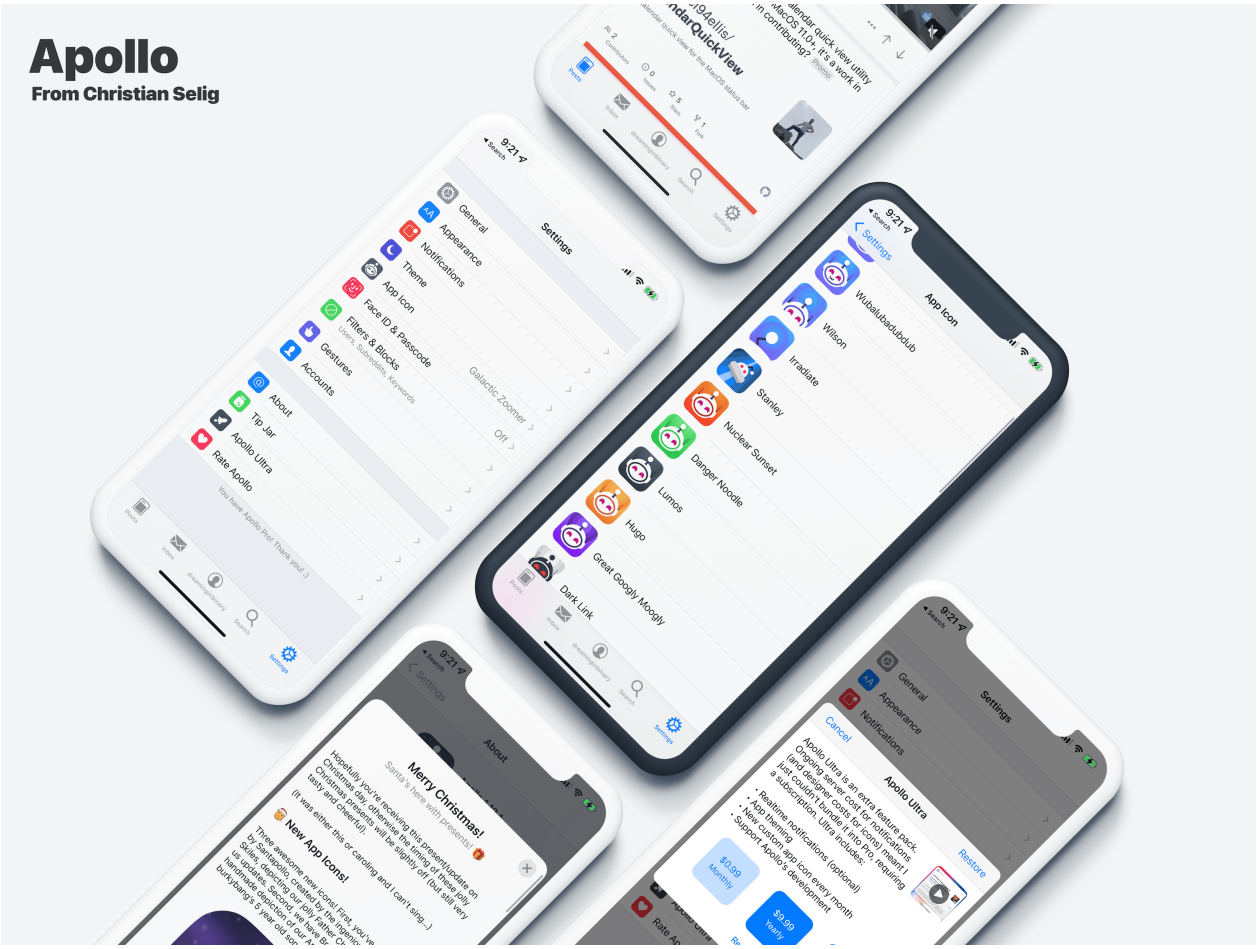
Playful

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:



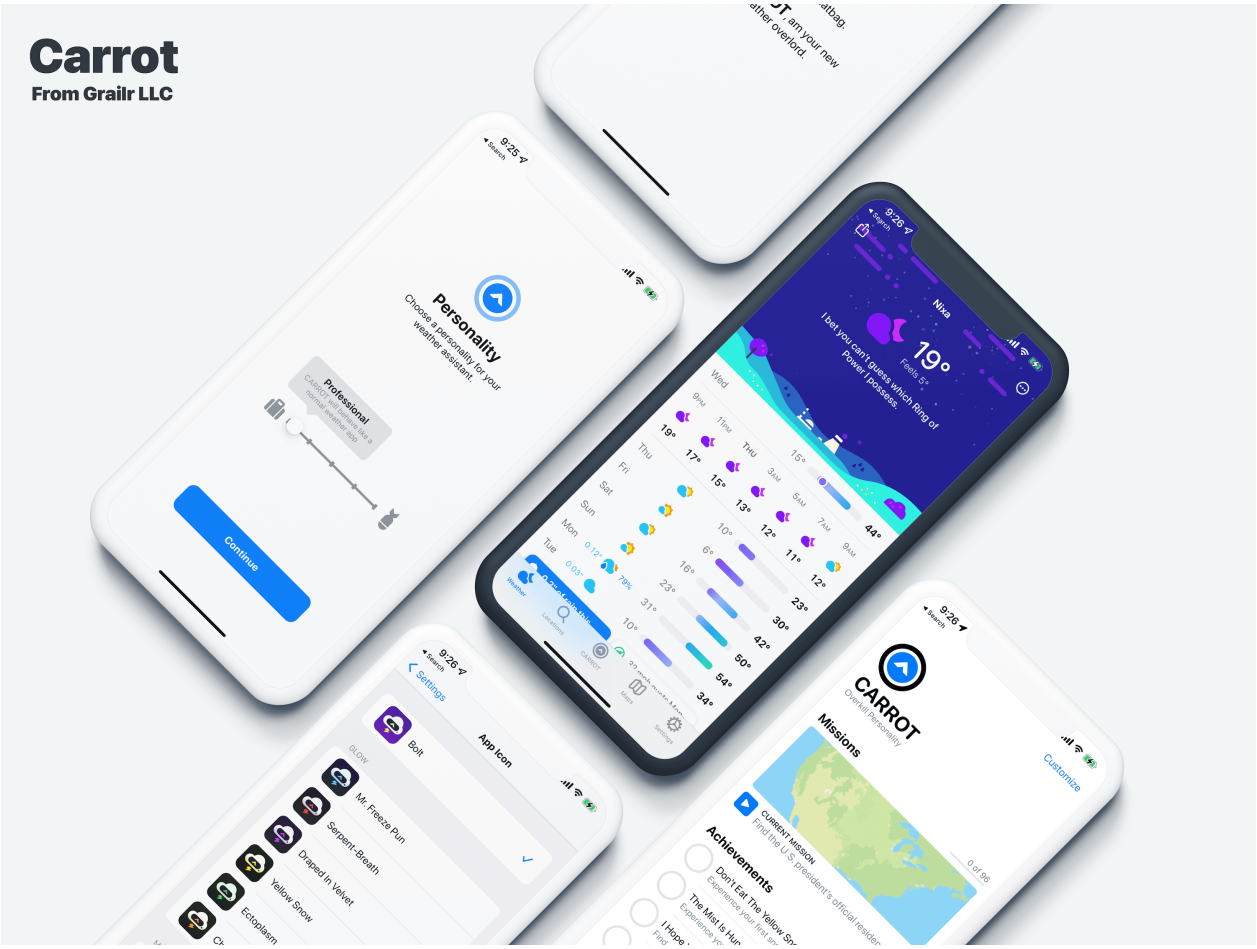
Friendly

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.



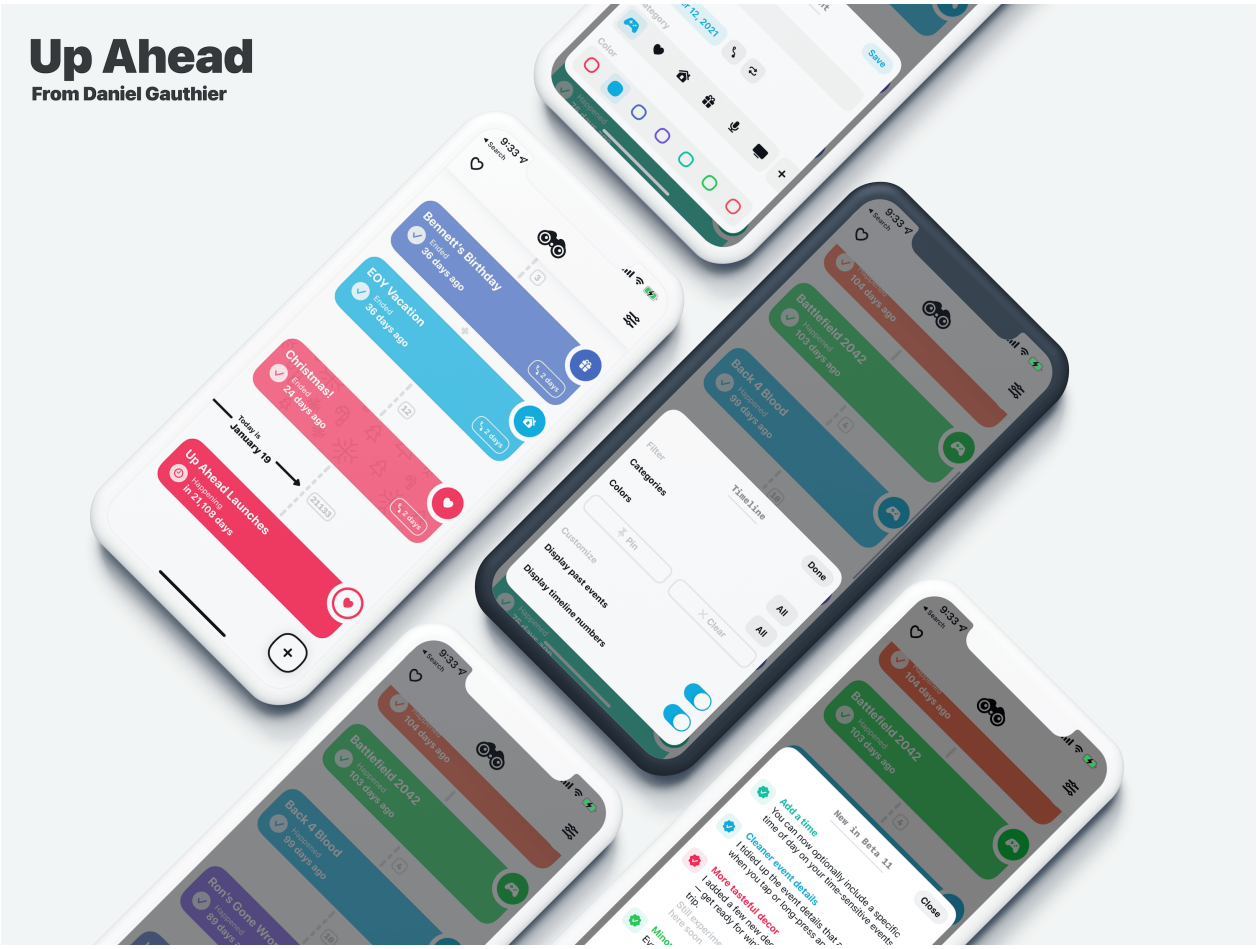
Sarcastic

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:



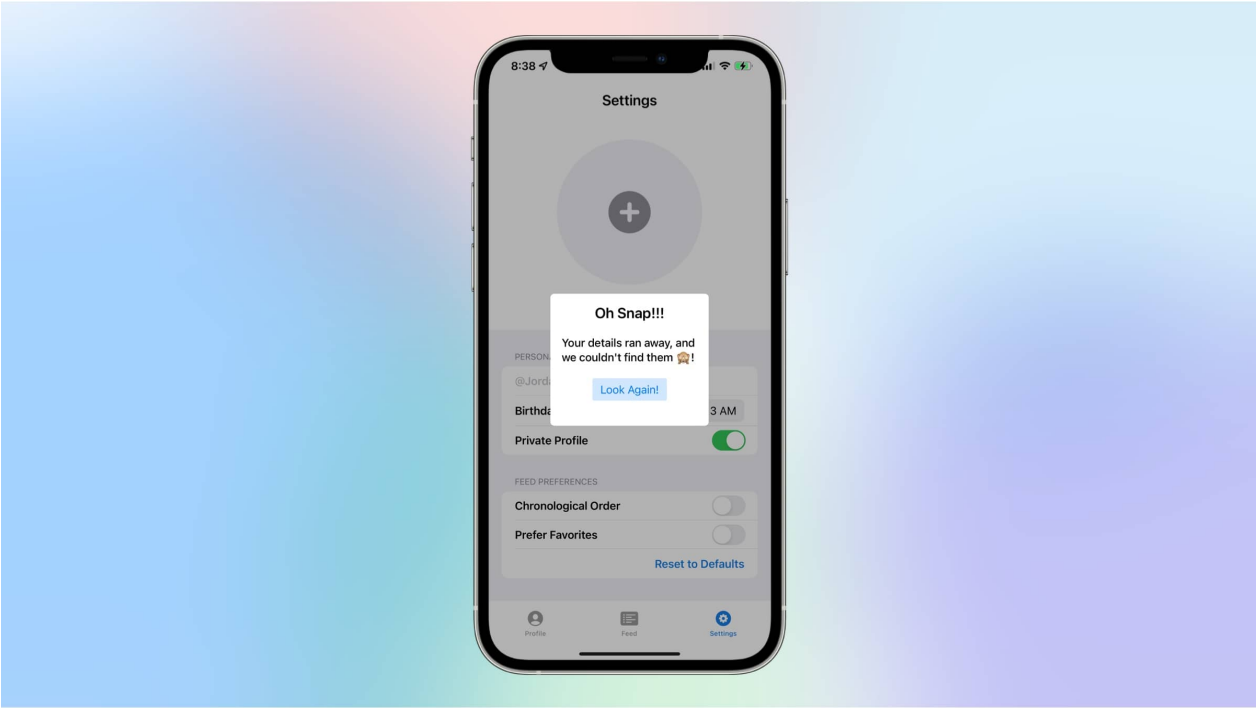
Something Else!

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.

Custom View Controller Transitions

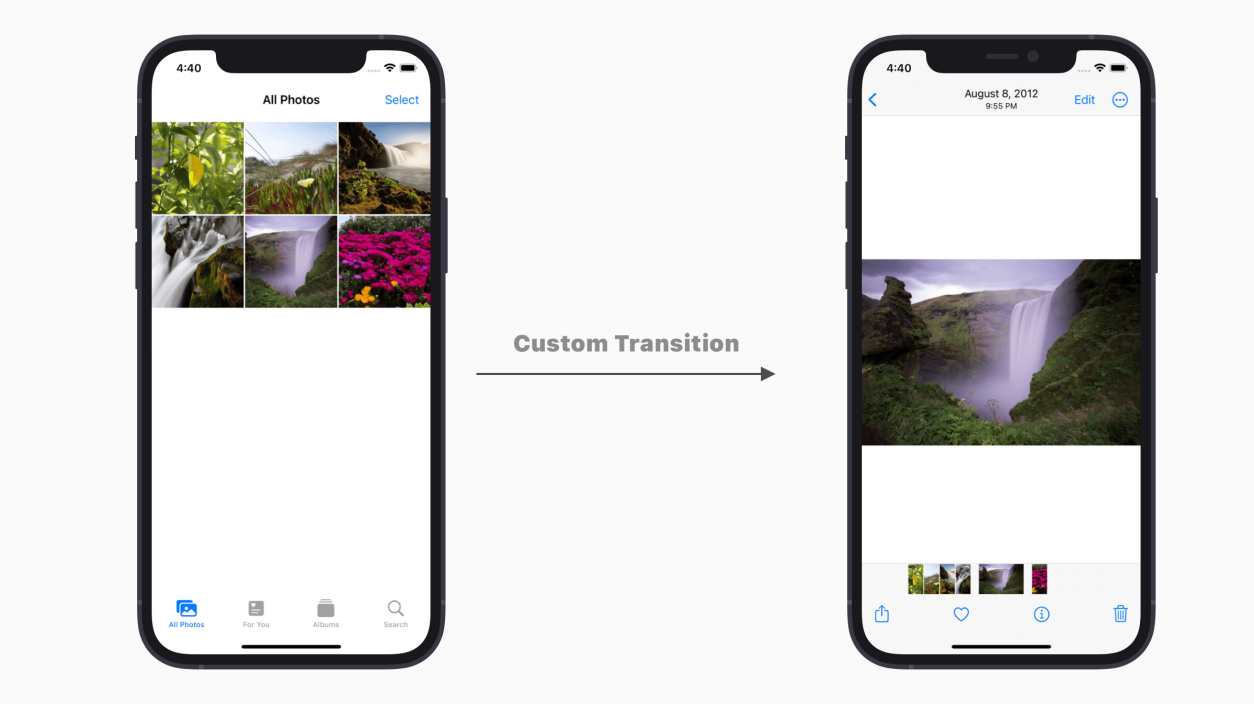
Custom view controllers add that *jena se qua* to any app. They just do. When done correctly, they delight and surprise people. When they are gratuitous or implemented unnecessarily, they get in the way. This chapter goes into what makes a good transition, when you might look to use them and how to implement 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.*

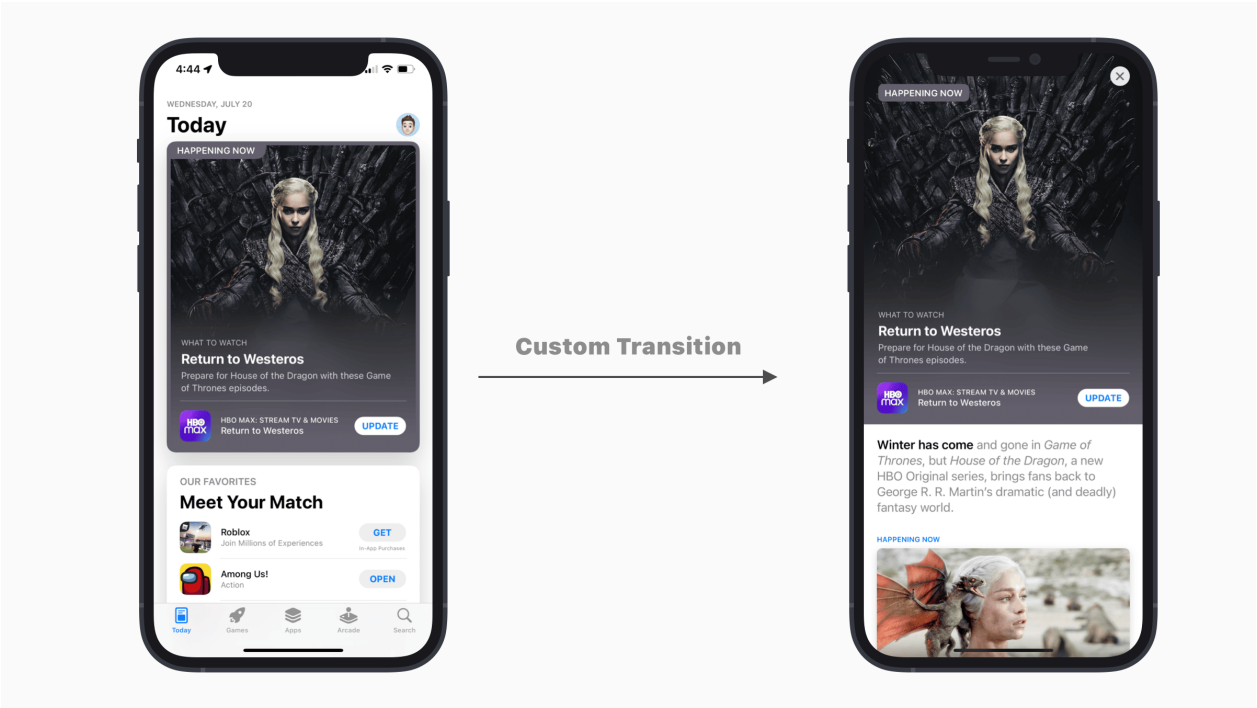
To kick off our discussion, what do I mean by a custom view controller transition? Within UIKit, anytime you present a view controller from another view controller — a view controller transition is occurring. For the most part, these are modal in nature — though that's not always the case when you're 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:

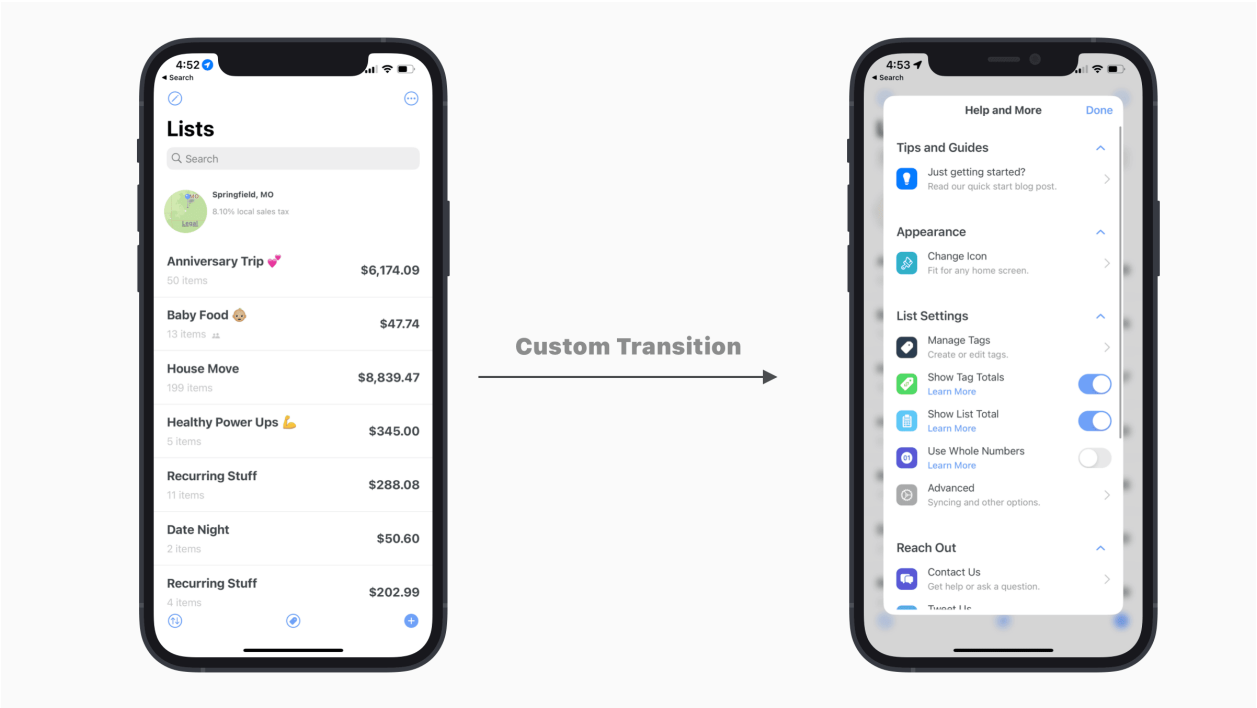


Further, a lot of these transitions can be interactive, meaning you can control their progress with a gesture, pause them while they are occurring or cancel them altogether. Many transitions are start and stop in nature, meaning that a user intends to view something, the system kicks off the transitions 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.

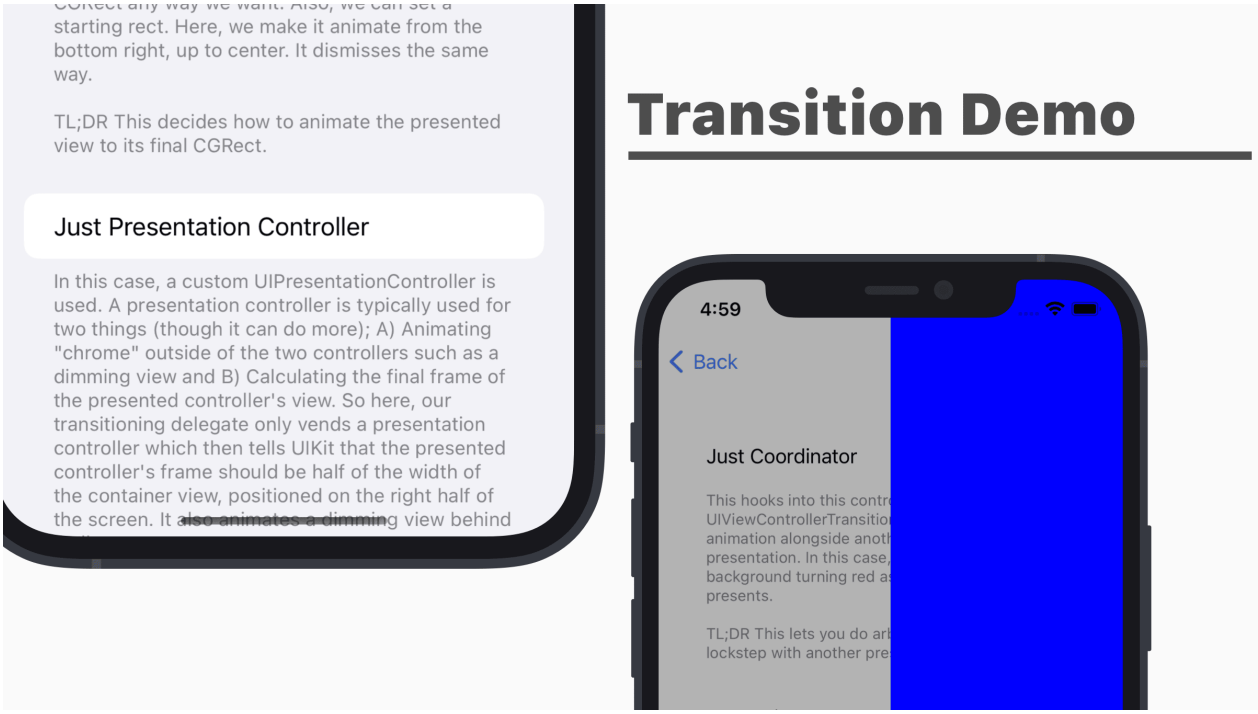
How Custom Transitions Work

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.

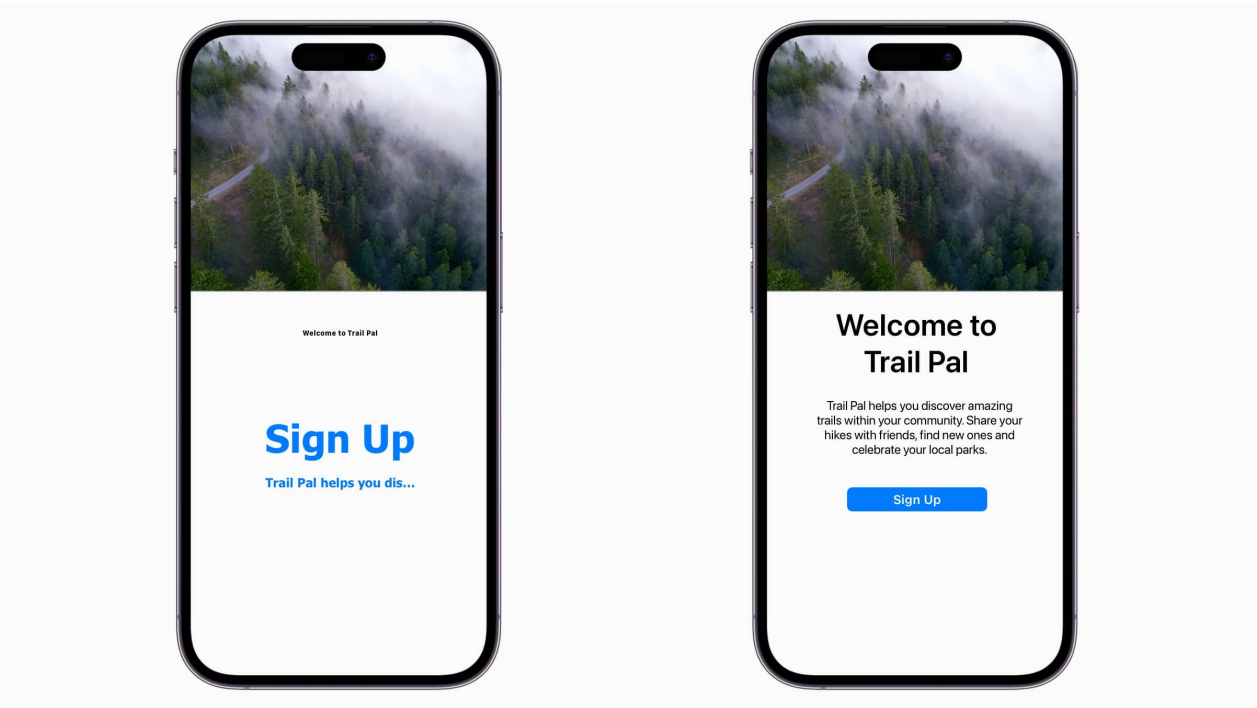
Three Key Takeaways

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.

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-

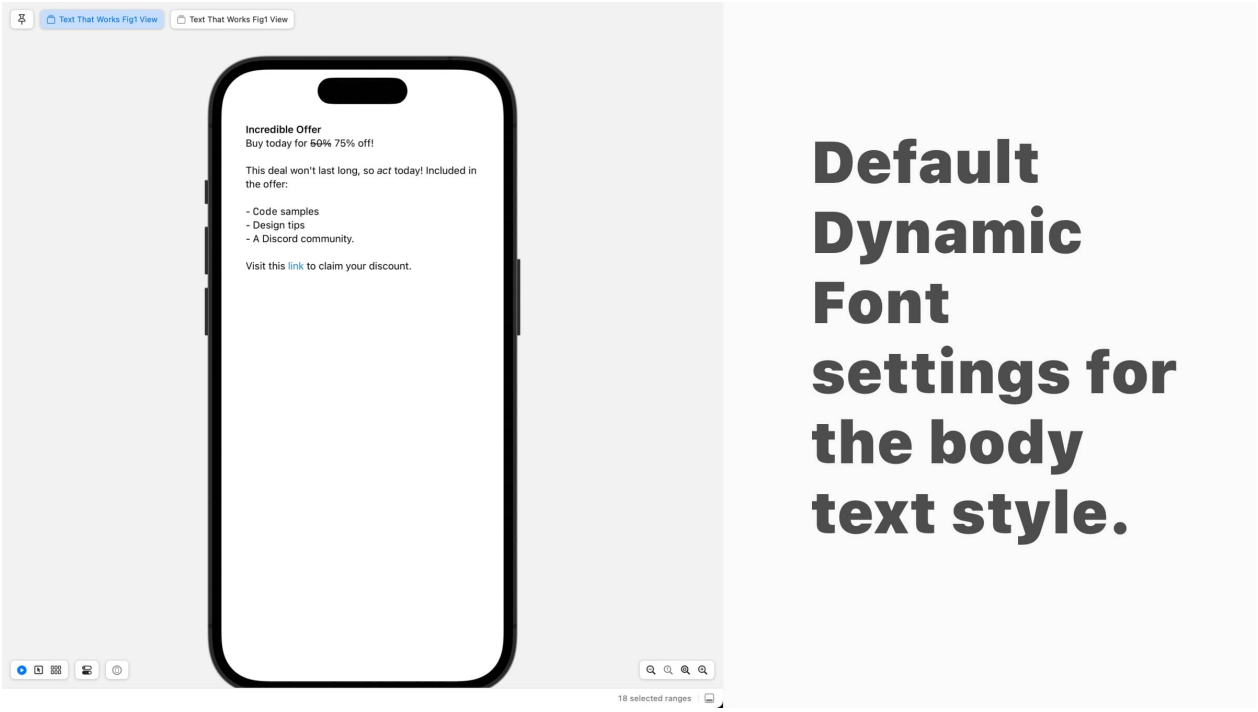
lar use case, what's dynamic about them? Well, the dynamic parts are enabled *by* the semantic meanings you use for fonts.

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

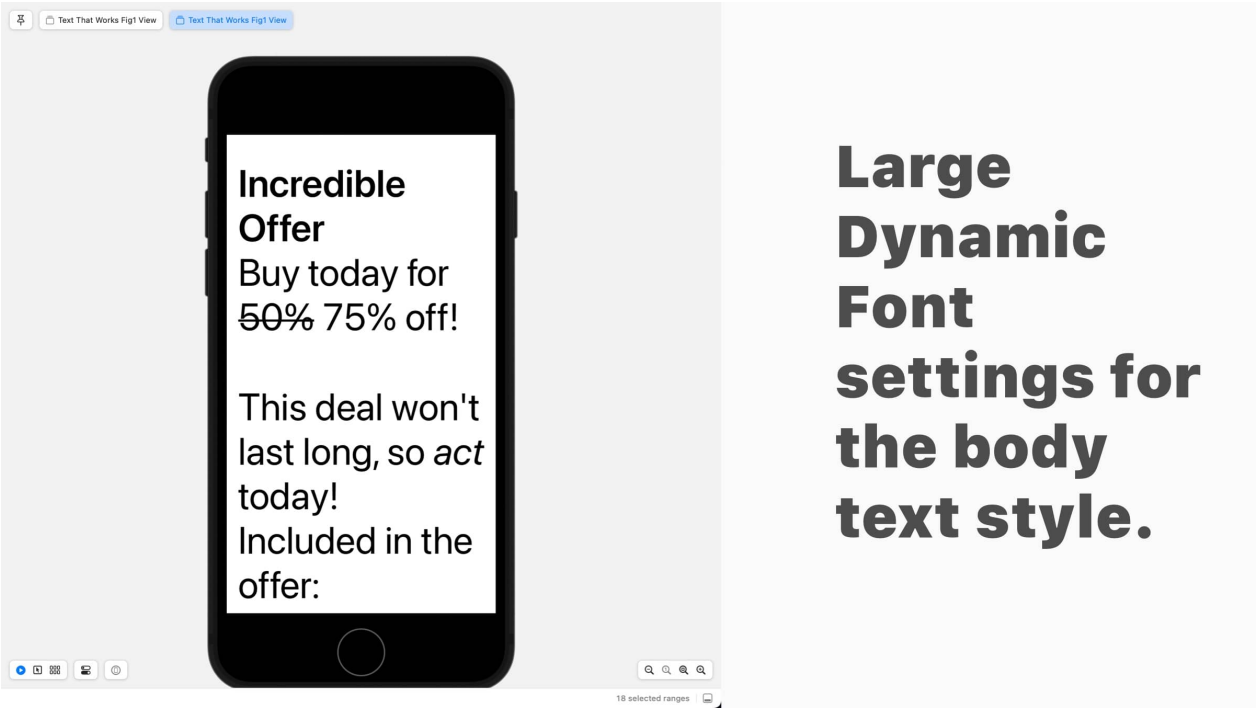
Continuing on with Dynamic Type, we just saw that any text might be fairly reasonably sized, or it could be very large. Consider the delta between these two examples. Here's some text at the regular `.body` size, basically an iPhone's out-of-the-box defaults:



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:

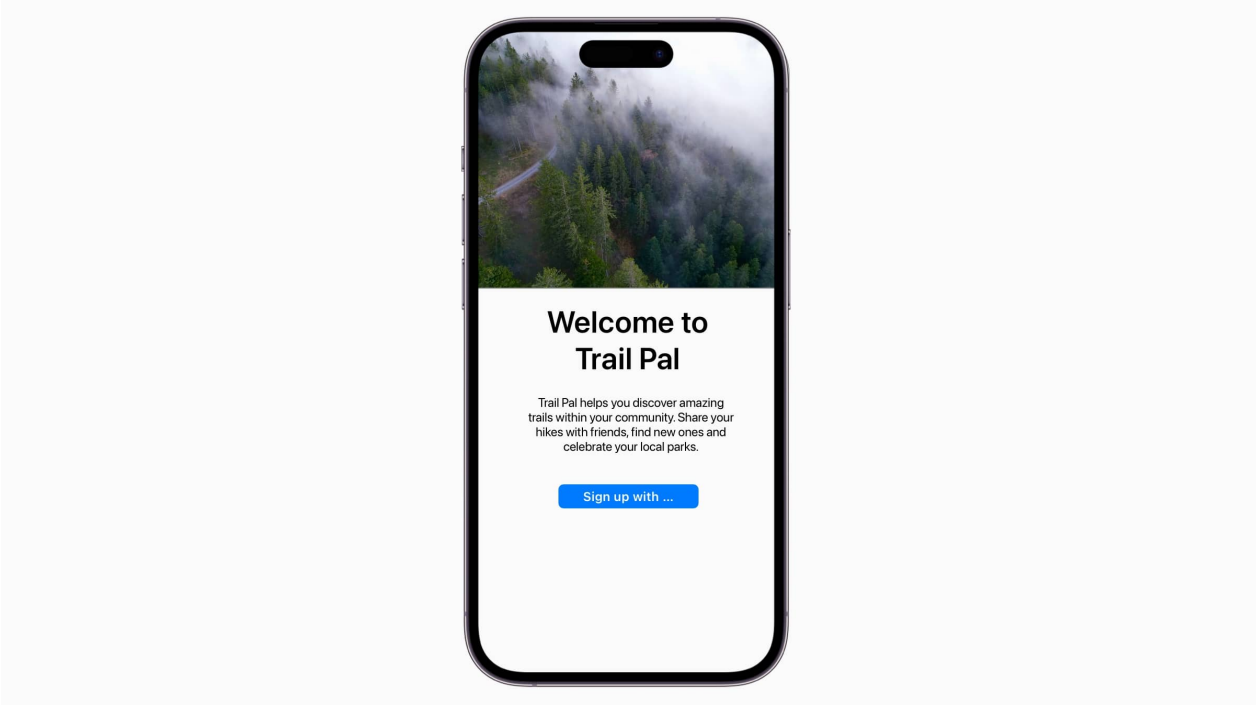


And, now you see why nearly every view should be a scrolling view. In fact, I start implementation with a scrolling view when I begin development, and I recommend you do the same. In SwiftUI or UIKit, our fix for this example is incredibly easy – simply wrap it in a `ScrollView` or `UIScrollView`. For example, in SwiftUI:

```
var body: some View {
    ScrollView {
        Text(markdown)
            .padding()
    }
}
```

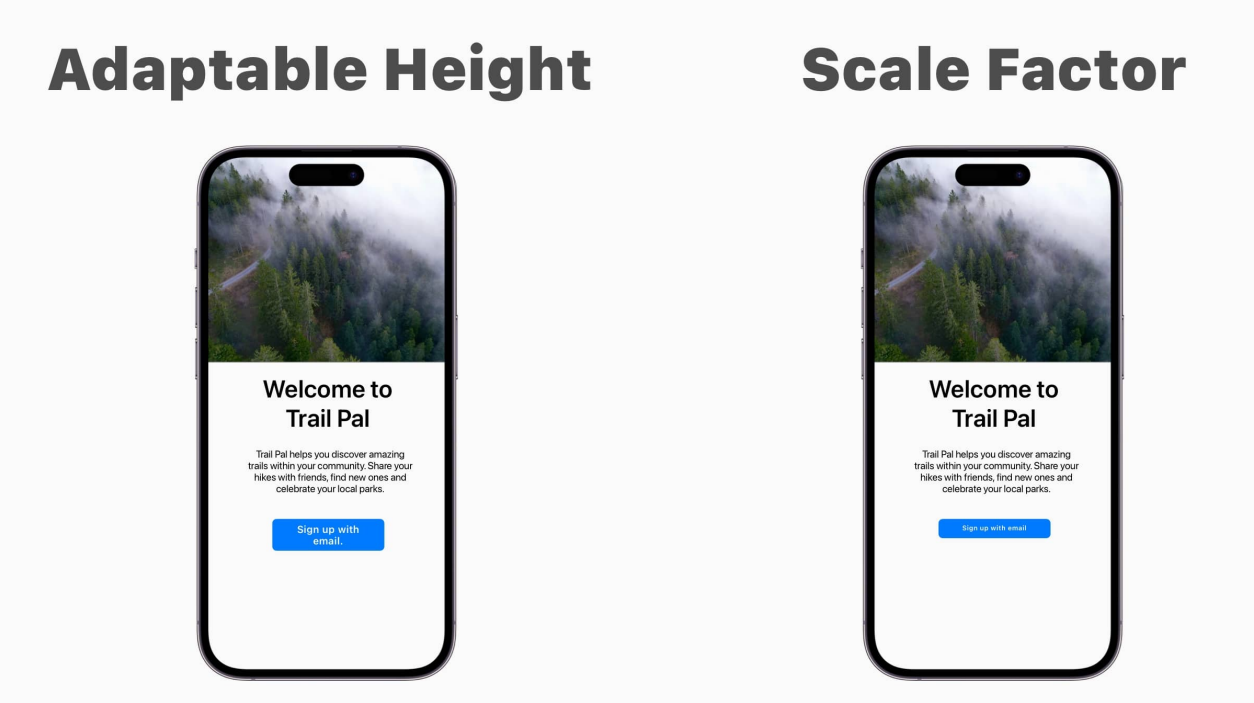
Avoid Clipping or Truncation, Promote Wrapping Text

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.

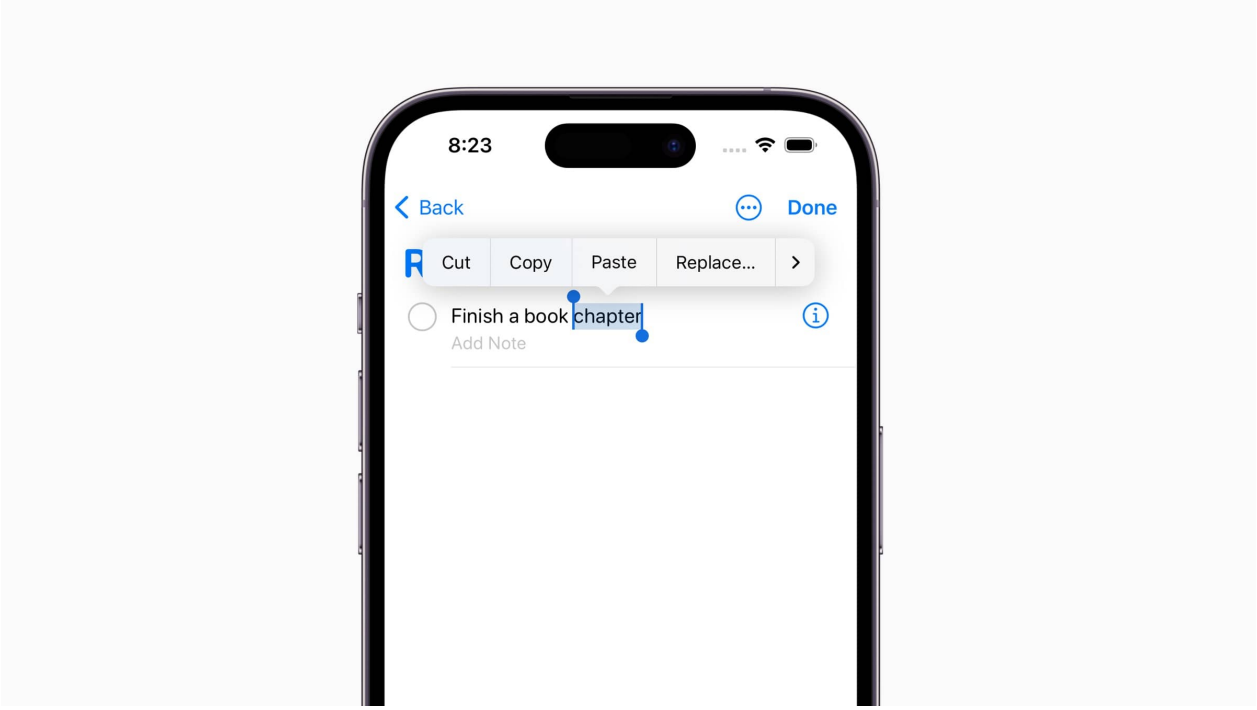
Again, for much more information over supporting Dynamic Type, please visit its relevant chapter. For a small dose of what you might be in for in terms of implementation with custom fonts to support Dynamic Type, it really comes down to `UIFontMetrics`:

```
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:

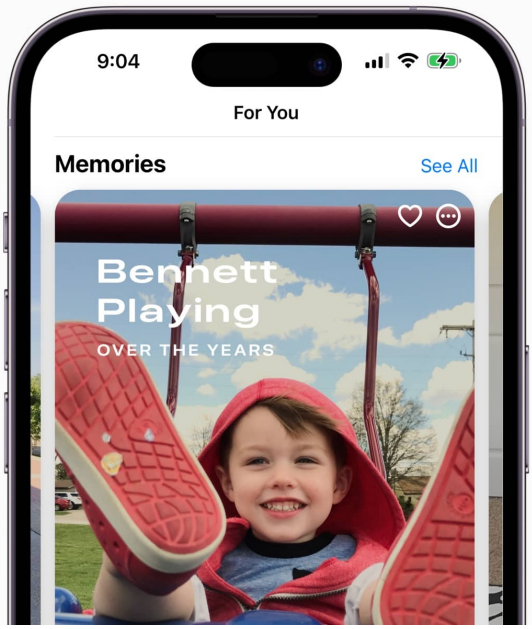
```
// Xcode -> SwiftUI -> TextThatWorksFig2View.swift

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.

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 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`.

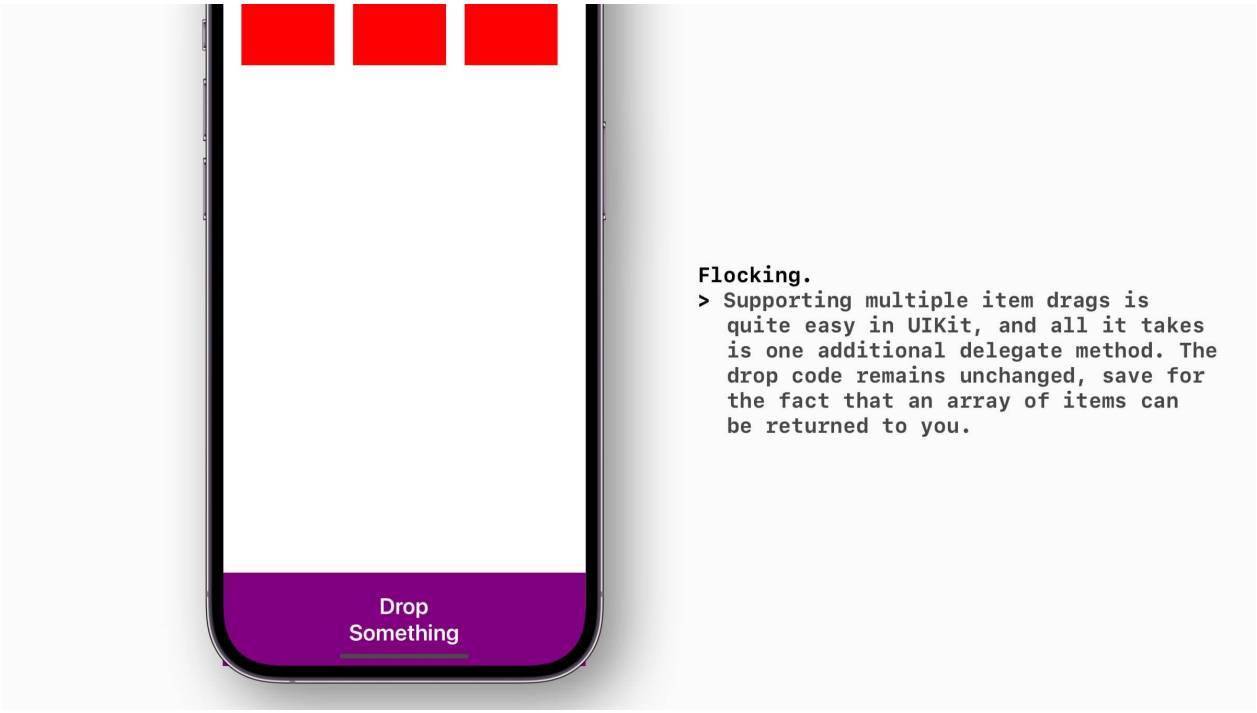
Supporting Multiple Item Drags

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:



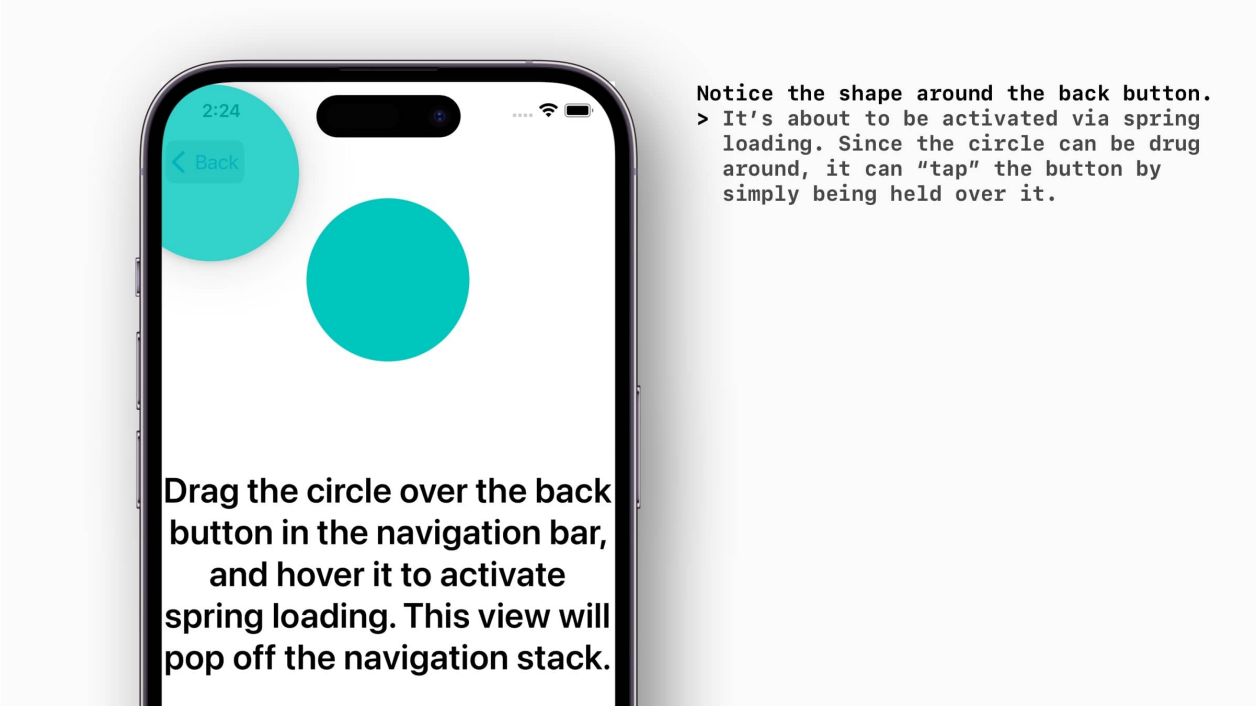
Support Spring Loading

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

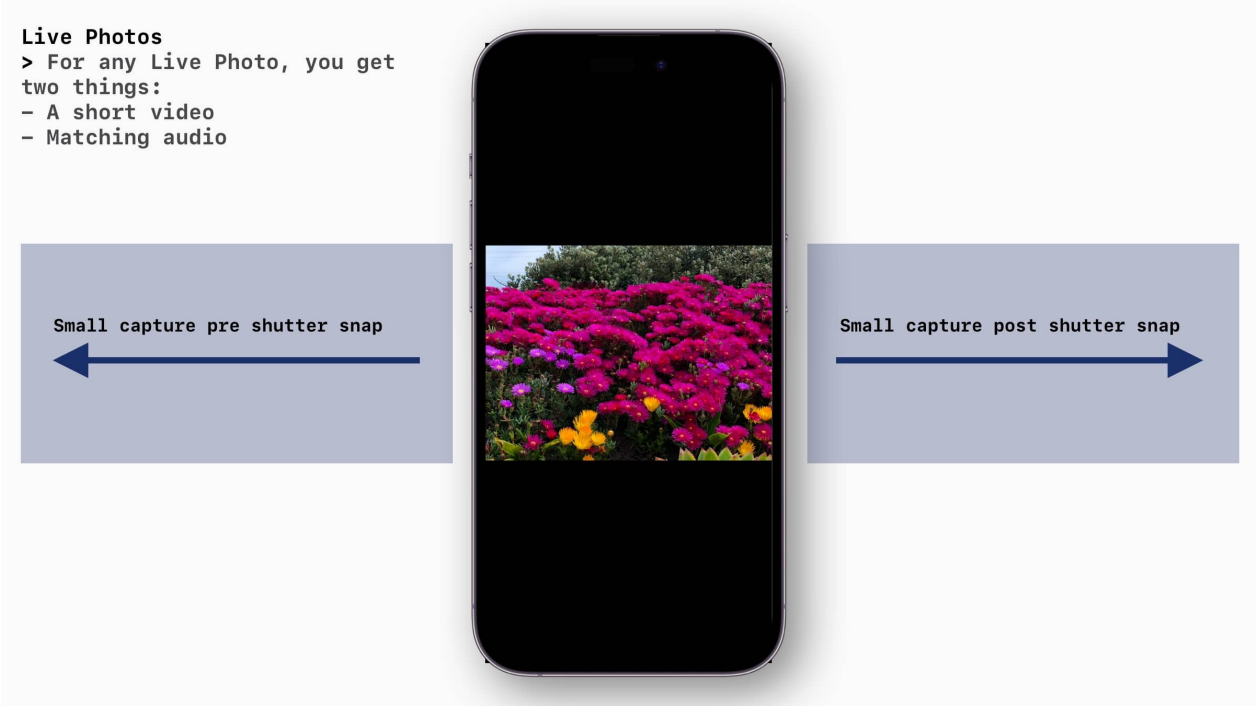
1. Support as many drag and drop interactions as you can in your app — this is a "the more, the better" situation.
2. Ensure you've designed your drag and drop interactions to be *obvious*, the outcome of a drop should yield no surprises.
3. When designing for drag and drop, ask yourself if there are places to add its three core flows (copy, move, or reorder).

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 little 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 UIK- it 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, property support their playback.