



THE ULTIMATE SWIFTUI LAYOUT COOKBOOK

**FIRST EDITION
KARIN PRATER**

1. Working with SwiftUI in Xcode	8
1.1 Showing previews in Xcode.....	8
1.2 Working with the Canvas in Xcode	13
1.3 Quick and Efficiently Edit SwiftUI Views	18
1.4 Debugging layout issues	20
1.5 SwiftUI Tree of Doom	23
1.6 Typical problems with Xcode and swiftui and how to fix them.....	25
2. Primitive Layout Components.....	27
2.1 VStack, HStack and ZStack	27
2.2 Divider and Spacer	30
2.3 Group	33
2.4 GroupBox	34
2.5 ControlGroup	36
3. Layering Views.....	38
3.1 Background Modifier.....	38
3.2 Overlay modifier.....	42
3.3 ZStack vs background/overlay	44
3.4 Color view.....	46
3.5 Gradients	48
3.6 Materials.....	50
4. Positioning Views	53
4.1 How to position views.....	53
4.2 Alignment Guides.....	57
4.3 Custom Alignment Guides.....	60
4.4 Grid View.....	66
4.5 Position and Offset Modifiers	70
5. Sizing Views.....	73
5.1 How the layout system sizes and positions views	73
5.2 Fixed and Flexible Frames	78
5.3 FixedSize	82
5.4 Layout Priority	84
5.5 Sizing Text Views	86
5.6 Sizing Images	89

5.7 Upscaling images and Bitmap vs Vector graphics	93
5.8 Sizing System Icons.....	95
5.9 AsyncImage.....	97
5.10 Aspect Ratio	103
5.11 Scale Effect	104
5.12 Content Edges: Safe area, Padding and Margins.....	106
5.13 Container Relative Frame	109
5.14 CornerRadius, Clip and Mask	113
Challenge 🖐️ Superhero Detail View	116
6. Reusable Layout Components.....	118
6.1 Making Your SwiftUI Views More Reusable	118
6.2 Reusable View Modifiers.....	121
6.3 ButtonStyle	124
6.4 Custom Container Views	126
6.5 Custom Containers with Dynamic Data	128
7. Custom Layout	131
7.2 GeometryReader	131
7.3 Example: Custom Container with GeometryReader.....	139
7.4 PreferenceKeys.....	141
7.5 Bounds Measurement with PreferenceKeys and GeometryReader	145
7.6 Layout Protocol.....	149
7.7 Layout Example: Equal Width HStack and VStack	156
7.8 Layout Example: Flow Layout	157
7.9 Layout Example: Radial Layout.....	159
7.10 Custom Layout with Layout Priority	161
7.11 Custom Layout for Image Gallery	163
8. Dynamic Data	169
8.1 ForEach	169
8.2 identifiable Data.....	172
8.3 Making Enums Identifiable	173
8.4 ForEach with Binding	176
8.5 LazyVStack and LazyHStack.....	178
8.6 Lazily Showing Images	180

8.7 Smooth ScrollViews with Images	185
8.8 LazyVGrid and LazyHGrid.....	191
8.9 Image Gallery with LazyVGrid and LazyHGrid	196
8.10 Infinitive Loading View	200
Challenges 🖐️	203
9. ScrollView	210
9.1 Why Use ScrollView?.....	210
9.2 Customizing The Appearance of ScrollView.....	211
9.3 Scroll Direction	215
9.4 Scroll Content Size	216
9.5 Scroll Behaviour	219
9.6 Programmatic Scrolling with ScrollViewReader.....	221
9.7 ScrollView Position	224
9.8 Synchronizing Multiple ScrollViews	226
9.9 Default Scroll Position.....	232
9.10 ScrollView Animations with ScrollTransition	235
9.11 Animations with VisualEffect	241
9.12 Parallax Example	245
9.13 Background Parallax Effect	248
9.14 Pinned Views	251
Challenge 🖐️ ScrollView	254
Custom Picker View Challenge	256
10. Adaptive Layout	261
10.1 Why You Need Adaptive Layout.....	261
10.2 What is the Available Space	264
10.3 Interface Size Classes.....	268
10.4 Environment Values	271
10.5 Environment vs PreferenceKeys	272
10.6 Dynamic Type Size	276
10.7 Scaled Metric.....	281
10.8 Conditional View Modifiers.....	283
10.9 AnyLayout - Switching Between Layout Containers.....	285
10.10 ViewThatFits	287

10.11 ViewThatFits Example 1	289
10.12 ViewThatFits Example 2.....	291
10.13 ViewThatFits Example 3	293
10.14 Keyboard Layout Adjustments.....	295
10.15 Keyboard & Background Image	298
10.16 Keyboard & Forms.....	301
10.17 Keyboard toolbar	302
10.18 Summary Responsive Design	305
10.19 Summary Adaptive Design	305
11. Special System Containers	306
11.1 OverView of System Containers.....	306
11.2 Adding macOS Target	307
11.3 List	310
11.3.1 ListStyle	315
11.3.2 List Row Background	317
11.3.3 List Row Insets and Separators	319
11.3.4 Move and Delete.....	322
11.3.5 List Selection	326
Challenge 🖐️ NavigationSplitView with Lists	330
11.4 Structuring Lists.....	335
11.4.1 Sections	335
11.4.2 Collapsible Sections.....	338
11.4.3 Hierarchical Lists.....	341
11.5 Form	344
11.5.2 Example: Settings View	347
11.5.3 Example: Registration Form	350
11.5.4 Example: Inspector	352
11.6 Table	355
11.6.1 Creating a Table with SwiftUI	356
11.6.2 Table Styling	358
11.6.3 Edit Table Data.....	361
11.6.4 Selecting Table Rows	363
11.6.5 Sorting and Filtering	366

11.6.6 DisclosureTableRow	371
11.6.7 Cross-platform.....	374
Challenge 🙌 Layout Variations.....	377

Welcome to The Ultimate SwiftUI Layout Book!

I made this book to help you find your way around SwiftUI's layout parts easily. You can learn at your speed, jumping into different sections as needed, like a cookbook for coding. It's packed with examples that you can use right away, saving you time.

How to use the Book and Course together

If you want to dive deeper, check out my course, where I walk you through everything step by step. Everything in the book matches up with the [SwiftUI Layout Course](#), making it easy to follow along.

Sometimes, reading is not enough, and seeing a code walkthrough in a video is much better for understanding. That is why I added links to all major section headings. If you get stuck in a section, just click on the heading and watch the corresponding video. This is especially useful for e.g. interactions or animations like in [9.12 Parallax Example](#)

The image is a composite of three elements related to the SwiftUI course:

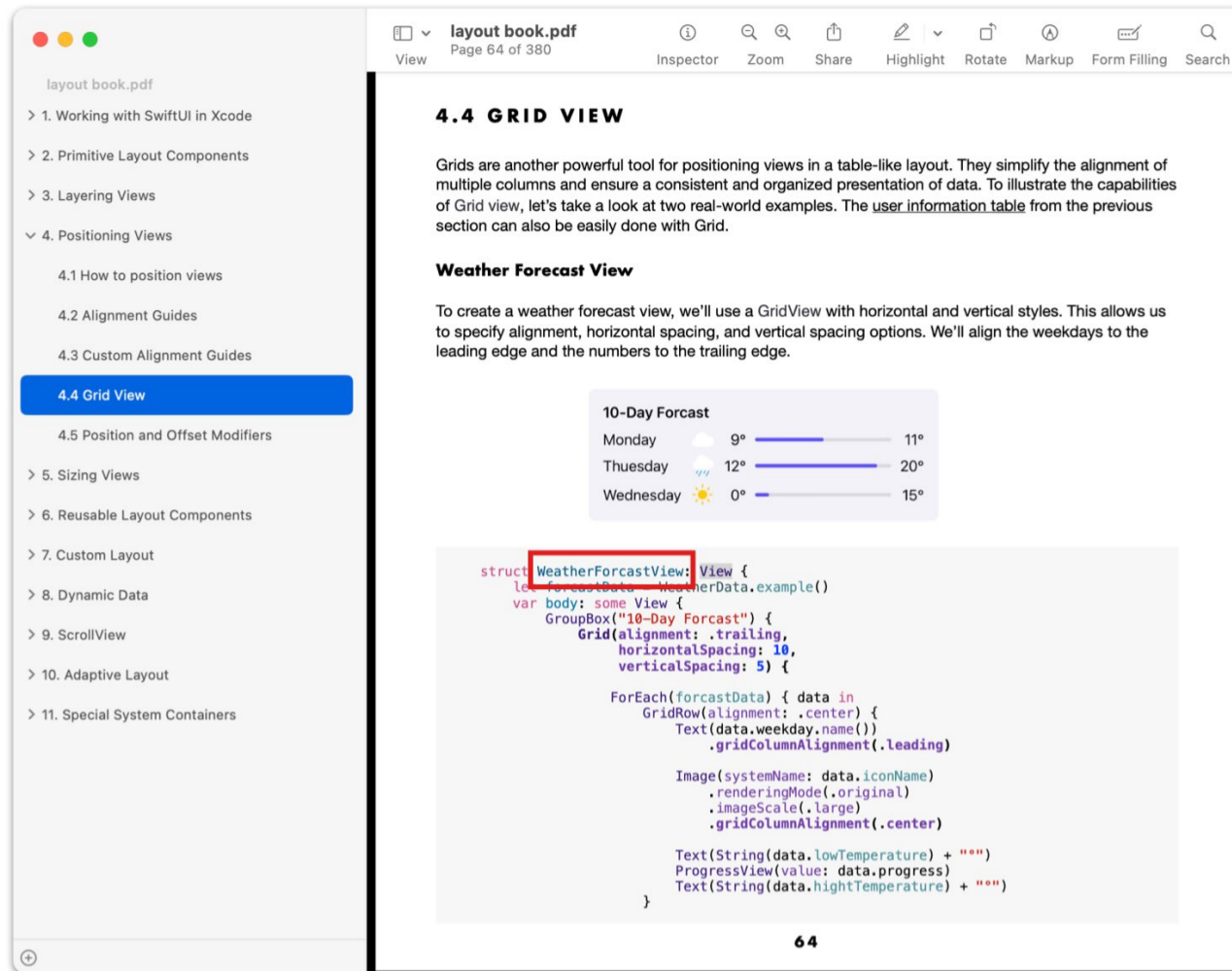
- Table of Contents (Left):** A list of sections under '9. ScrollView'. Section '9.12 Parallax Example' is highlighted in blue.
- Course Page (Middle):** A page titled '9.12 Parallax Example' with a red box around the heading. Below the text, there are four mobile app screenshots showing a parallax effect. A progress bar indicates '3% COMPLETE 4/128 Steps'.
- Video Player (Right):** A video player showing a code editor with SwiftUI code for a parallax effect. The code includes `ParallaxEffectView` and `ParallaxEffectViewModifier`. A video player interface is overlaid on the code.

A red arrow points from the '9.12 Parallax Example' section in the table of contents to the video player.

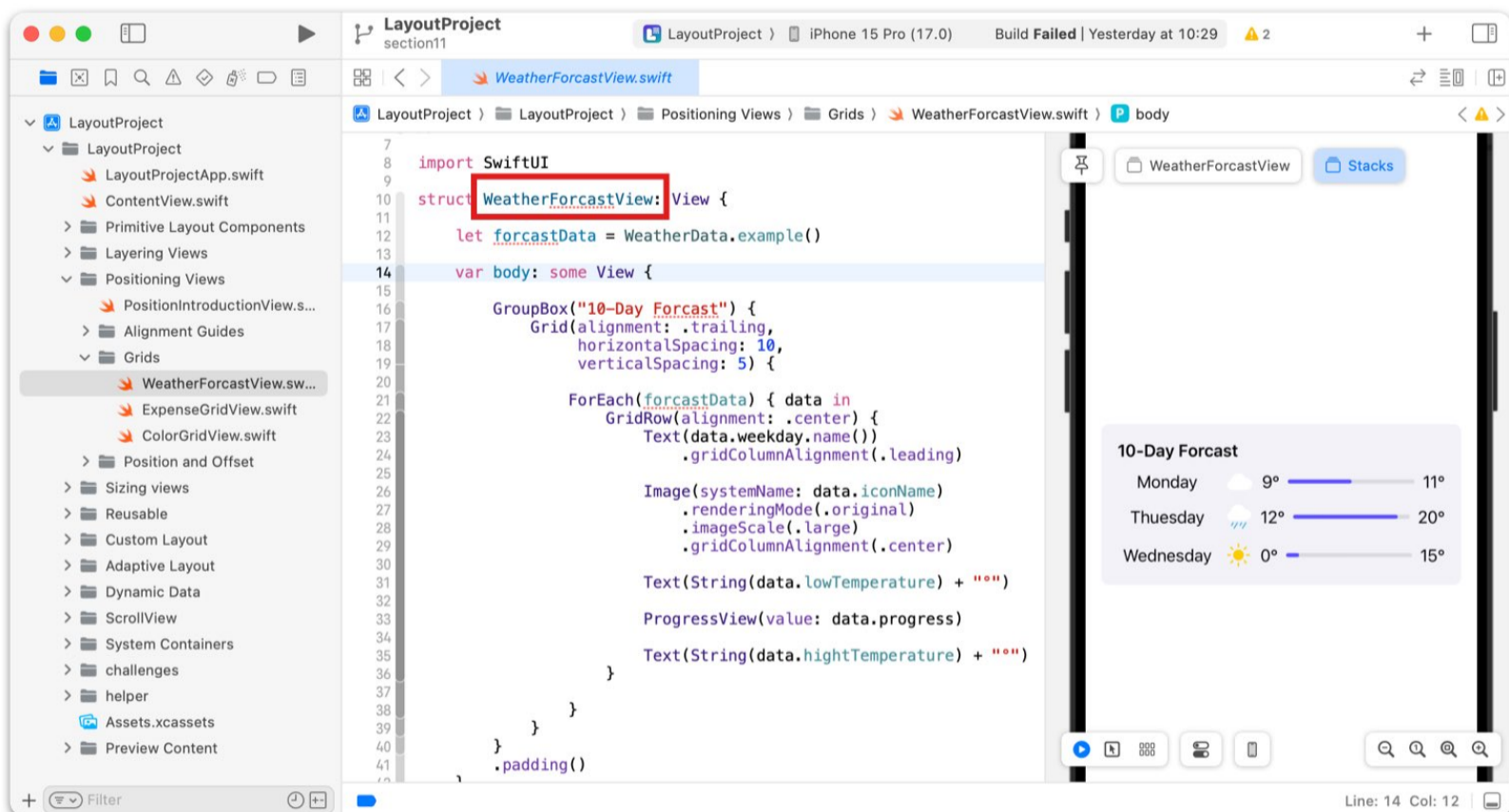
Don't have access to the Layout course yet? Upgrade and use code **"LAYOUTMASTERY"** to get \$20 off.

How to work with the Project Files

The structure of this book is the same as the companion project. If you want to find the code in the project, you can also use the file names from the code snippets:



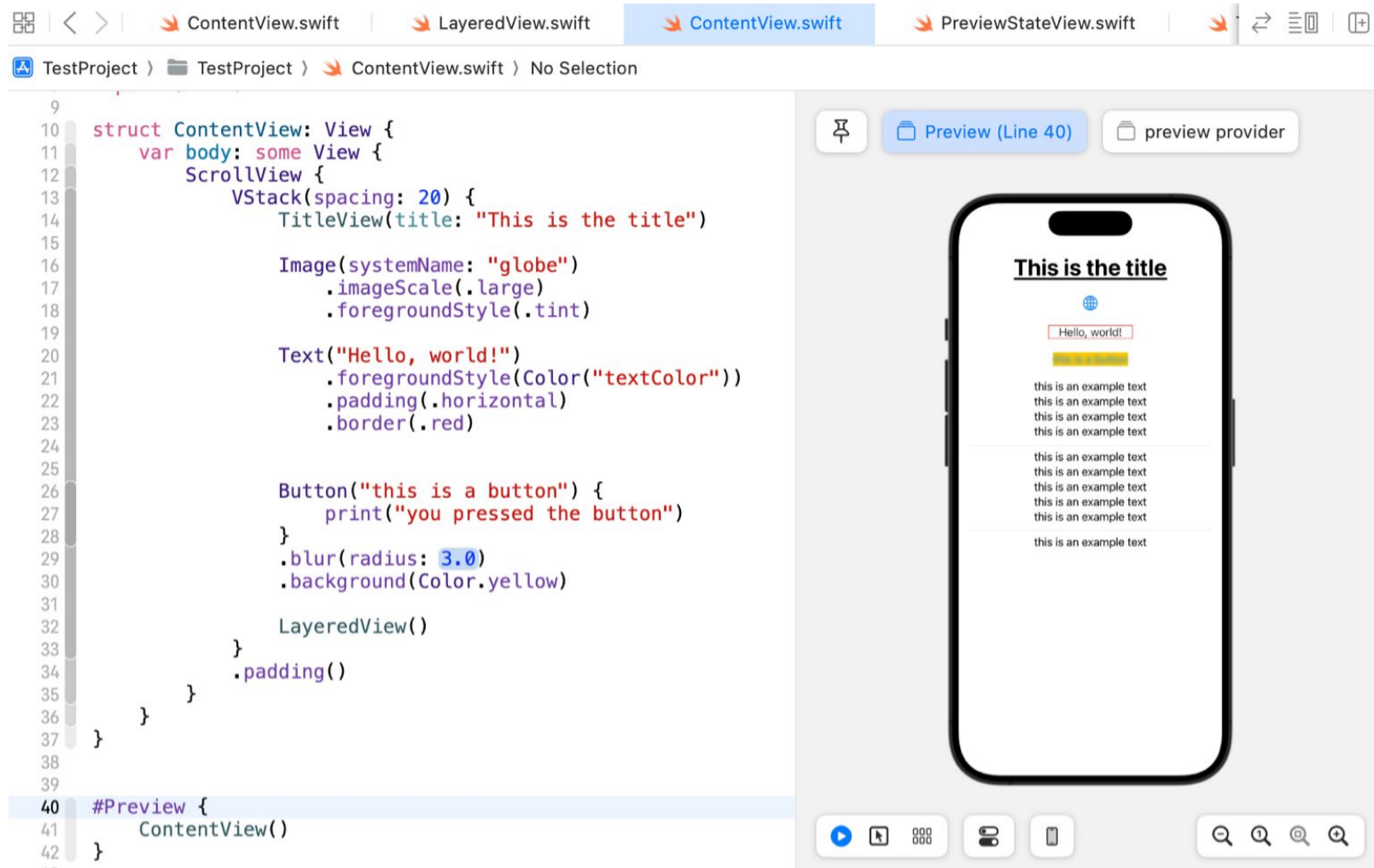
and search in Xcode for the corresponding file:



1. WORKING WITH SWIFTUI IN XCODE

1.1 SHOWING PREVIEWS IN XCODE

In this section, I will guide you on effectively working with Xcode for SwiftUI. Previews have changed with Xcode 15 and use now the Preview macro, whereas before you had to use the PreviewProvider. I will give you examples of both of these features.

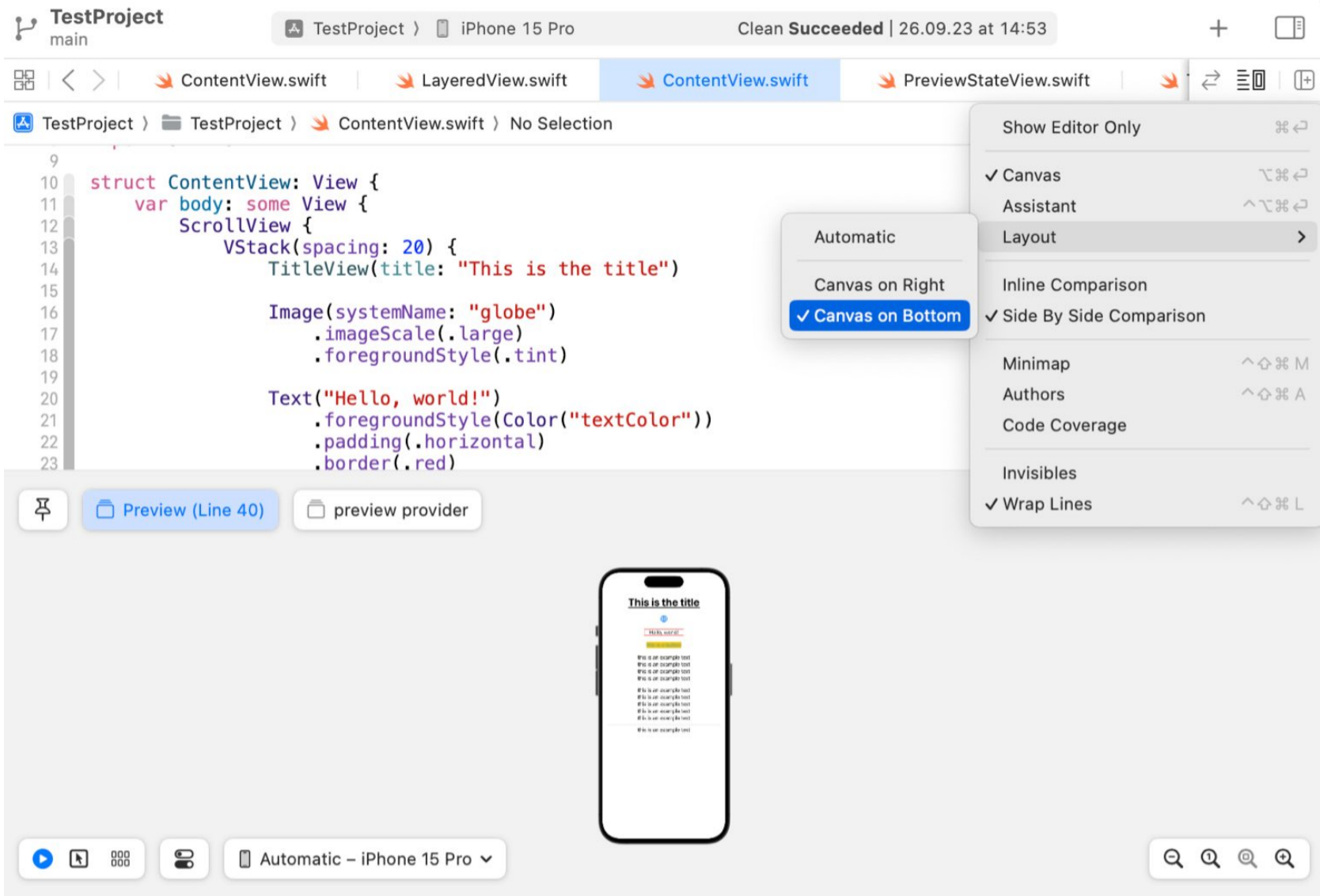


Showing and Hiding the Canvas

By default, the canvas is not shown on the right side. To show or hide the canvas, go to the top-right corner and click on the inspector. You can also use the keyboard shortcut **Option + Command + Return**.

Sometimes, when you make changes to your code, the preview may not refresh properly or may not be visible. In such cases, you can use the keyboard shortcut **Option + Command + P** to recreate the preview.

By default, the layout is set to automatic. You can choose between having the canvas on the right or below the editor by selecting the appropriate option. This allows you to maximize the space based on your preferences and the size of the views.



Preview Macro

The new preview macro has made it simpler and shorter in Xcode previews and is available for Xcode 15:

```
#Preview {
    ContentView()
}
```

When you generate a new file, such as using the SwiftUI view template, Xcode automatically generates a preview section for that file.

If your project's minimum deployment target is lower than iOS 17 or macOS 14, you need to add a version check before the preview:

```
@available(iOS 17.0, macOS 14.0, tvOS 17.0, watchOS 10.0, *)
#Preview {
    ContentView()
}
```

When working with multiple previews, it is important to ensure that you pass the correct arguments to the views. You can generate multiple previews by using the “preview” macro multiple times. Each preview

can only specify one view. If you need to test different input values, you can create multiple previews with varying arguments.

```
import SwiftUI

struct TitleView: View {
    let title: String
    var body: some View {
        Text(title)
            .font(.largeTitle)
            .bold()
            .underline()
    }
}

#Preview("short title") {
    TitleView(title: "Hello world")
}

#Preview("Long title") {
    TitleView(title: "This is a very, very, very long title")
}
```

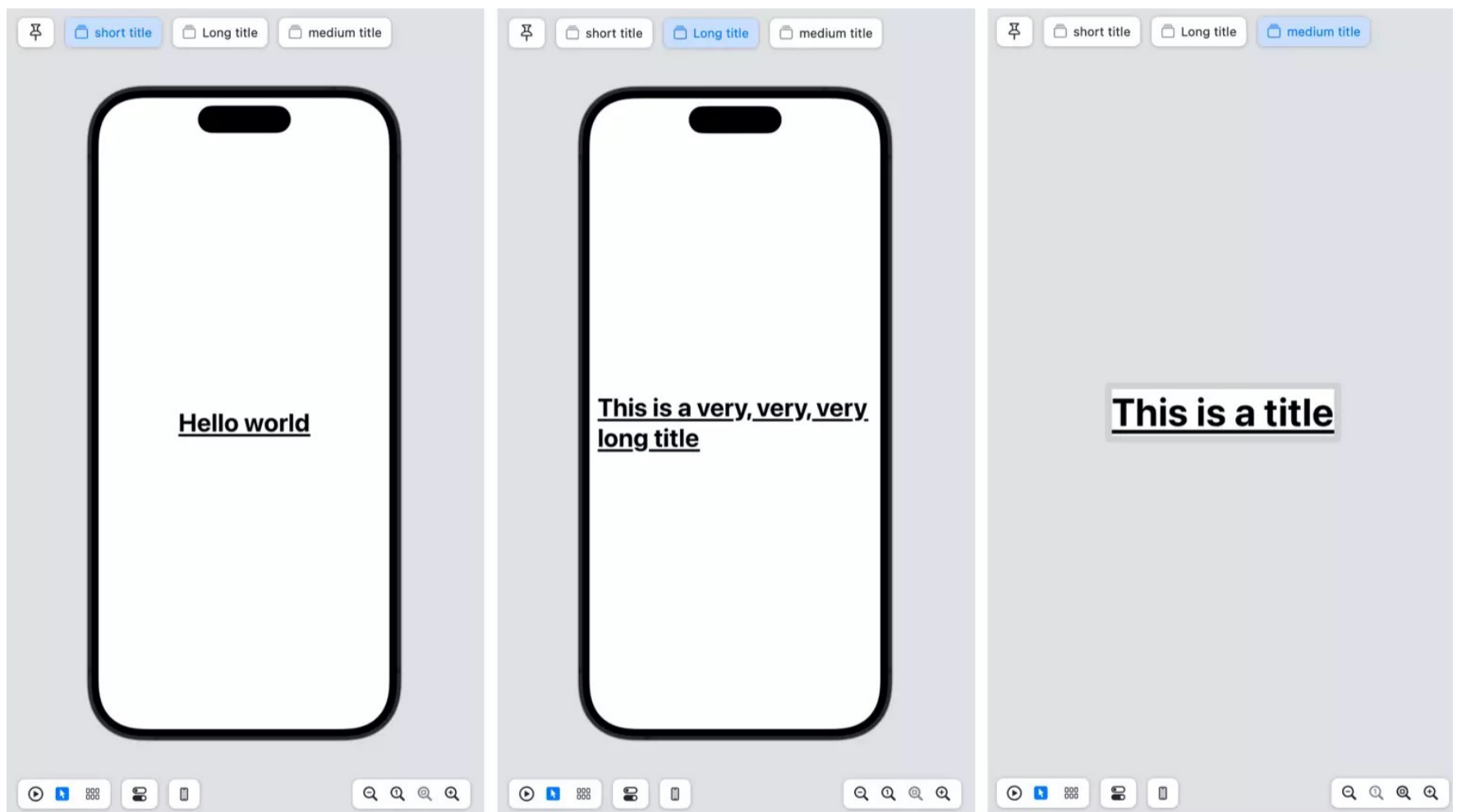


To organize your previews, you can give them names. This helps in distinguishing between different previews and provides a clear description of their purpose.

Additionally, you can set **traits**. In the below example, I used a “sizeThatFitsLayout”. This is useful when you have very small views and you don’t want to show them on a device. Note that this only works when you are in the selectable preview.

```
#Preview("short title") {
  TitleView(title: "Hello world")
}
#Preview("Long title") {
  TitleView(title: "This is a very, very, very long title")
}

#Preview("medium title", traits: .sizeThatFitsLayout) {
  TitleView(title: "This is a title")
}
```



Here is a list of all the available traits:

- `fixedLayout(width: CGFloat, height: CGFloat)` and `sizeThatFitsLayout`
- `portrait` and `portraitUpsideDown`
- `landscapeLeft` and `landscapeRight`

PreviewProvider

You may come across the preview provider if you are working with an older project that started before Xcode 15.

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

You can show multiple previews in the canvas:

```
struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        Group {
            ContentView()
                .previewLayout(.sizeThatFits)
            ContentView()
        }
    }
}
```

The preview can be customized by adding preview modifiers:

```
struct ContentView_Preview: PreviewProvider {
    static var previews: some View {
        ContentView()
            .previewLayout(.sizeThatFits)
            .previewDisplayName("preview provider")
    }
}
```

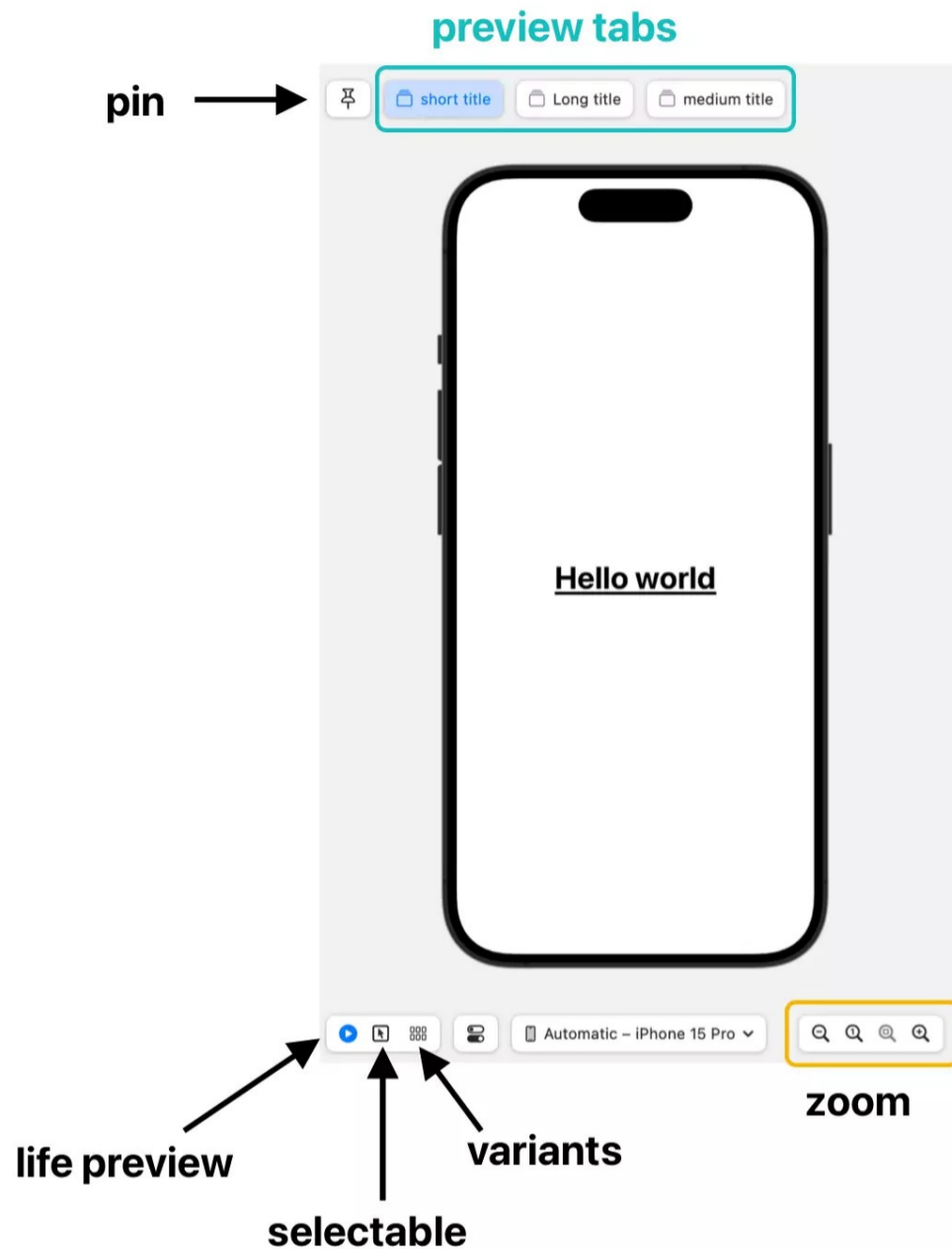
Modifier for minimum size	<code>.previewLayout(.sizeThatFits)</code>
Modifier for fixed size	<code>.previewLayout(.fixed(width: 600, height: 200))</code>
Set specific device type	<code>.previewDevice(PreviewDevice(rawValue: "iPhone 8"))</code>
Set display name tab	<code>.previewDisplayName("show Iphone 8")</code>

Change Environment variables:

Change to dark mode	<code>.environment(\.colorScheme, .dark)</code>
Change dynamic text	<code>.environment(\.sizeCategory, .accessibilityLarge)</code>

1.2 WORKING WITH THE CANVAS IN XCODE

In this section, we will explore the various features and options available in the Canvas in Xcode. The Canvas provides a real-time preview of your SwiftUI layout, allowing you to quickly iterate and test your designs.



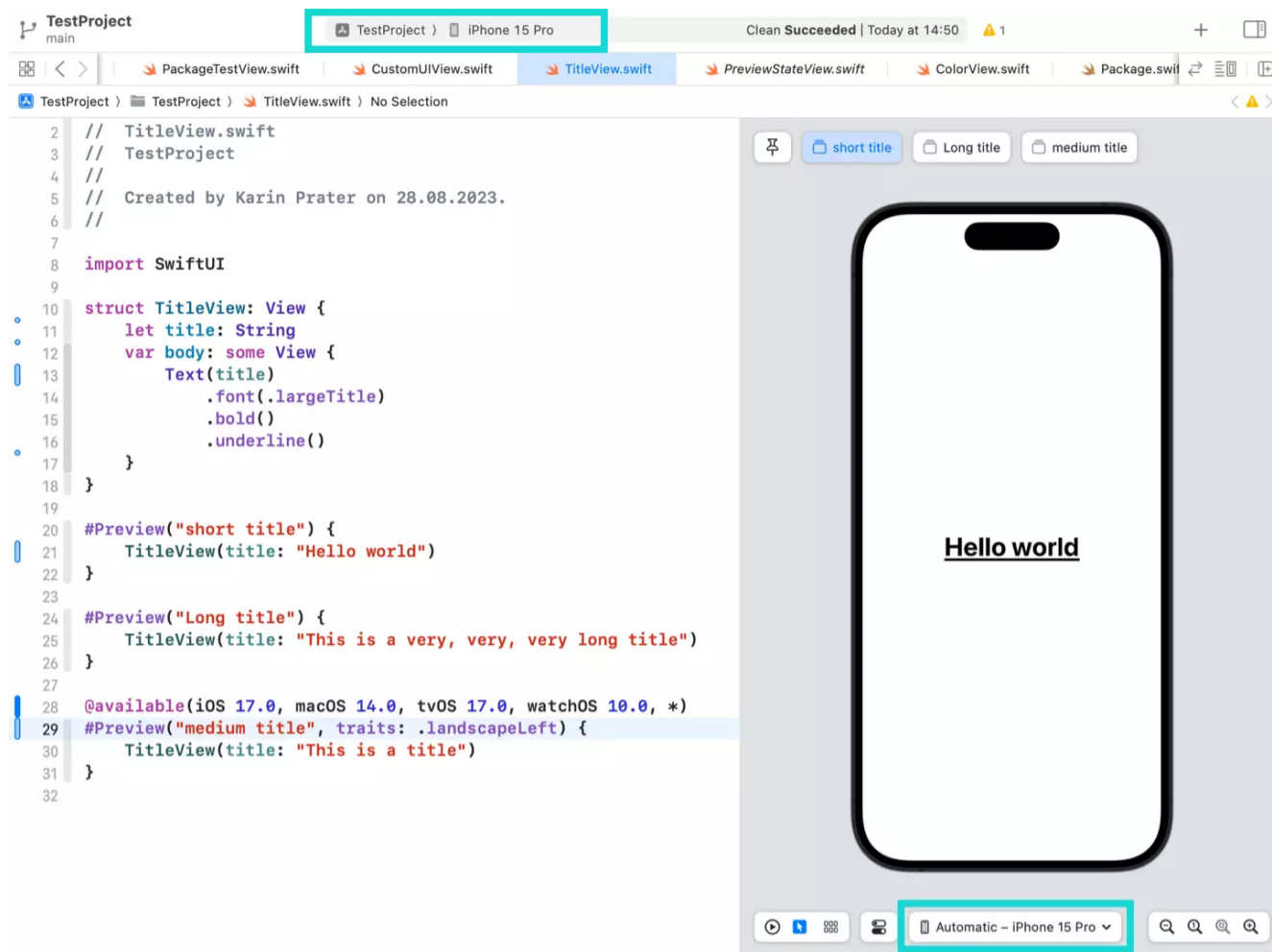
At the bottom of the Canvas, you will find a range of options to enhance your previewing experience. Let's take a closer look at each of these features:

Zooming and Fit on Screen

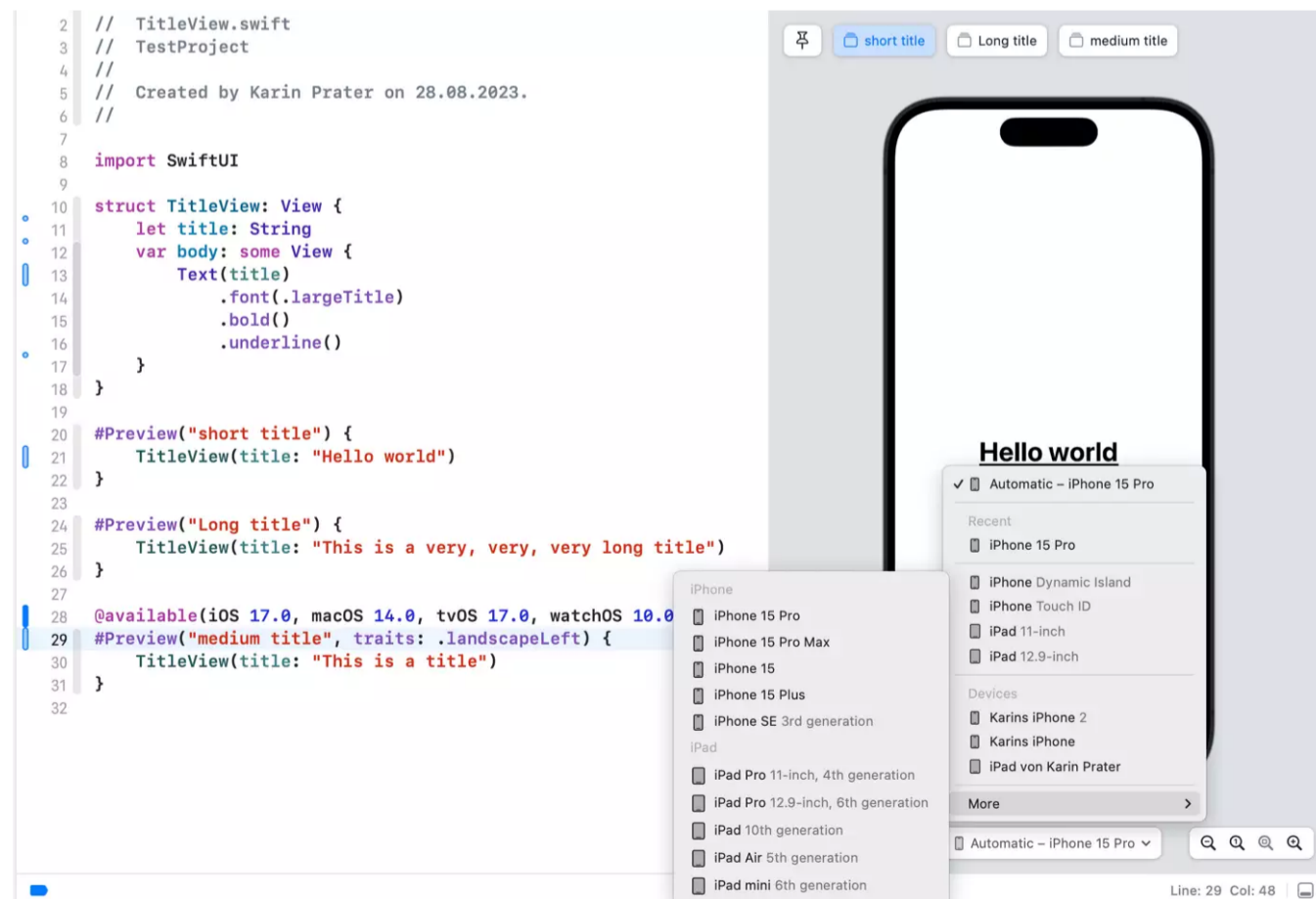
On the right side of the Canvas, you can find options to zoom in or fit the layout on the screen. These options help you view your design more closely or fit it to the available space.

Device Preview

The toggle next to the zoom options allows you to select the device for the preview. By default, the preview matches the device selected for your run target.



However, you can choose to manually select a specific device or let Xcode automatically switch the preview based on your run target.

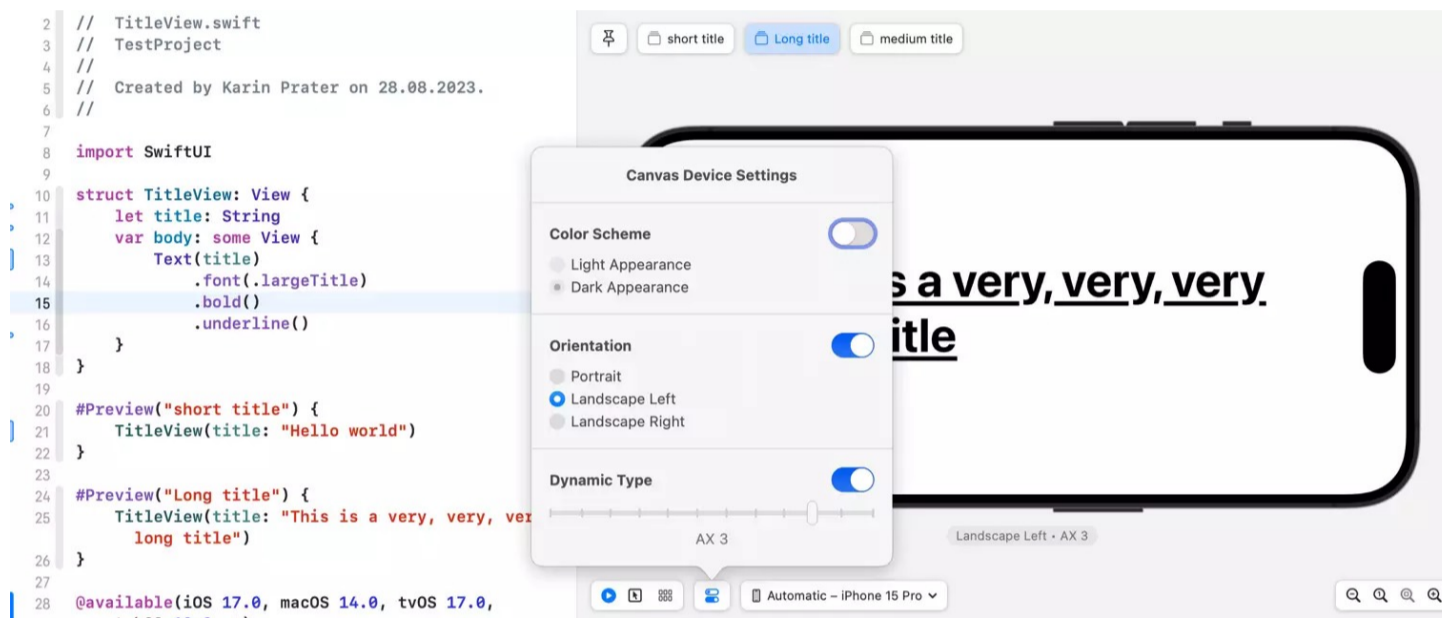


Real Device Preview

In addition to the built-in device previews, you can also use your actual iOS device to preview your SwiftUI layout. To enable this feature, you need to install the Xcode Previews app on your device and allow it in developer mode. Please note that there may be some connectivity issues, so ensure your device is plugged in for a reliable connection.

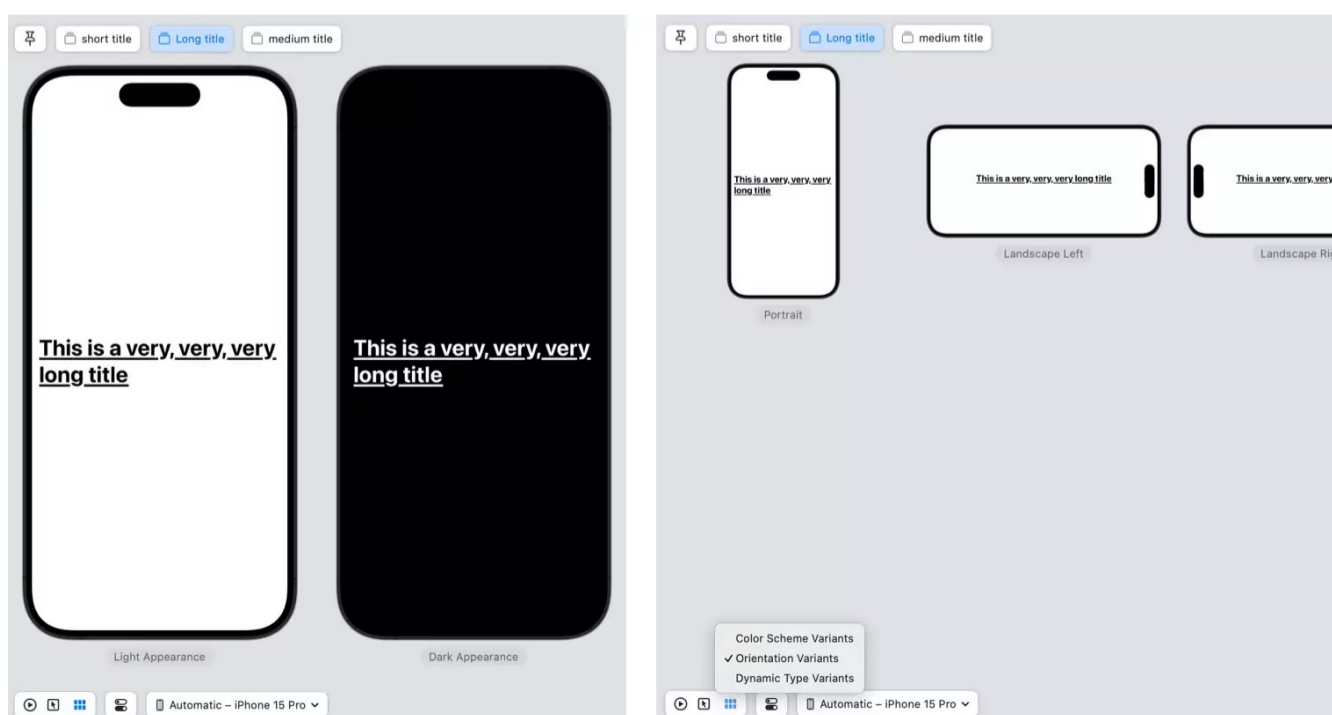
Device Settings

The “Device Settings” option allows you to customize the preview environment further. You can set specific color schemes, test landscape or portrait orientations, and even experiment with dynamic type sizes. These settings help you ensure your layout adapts well to different scenarios.



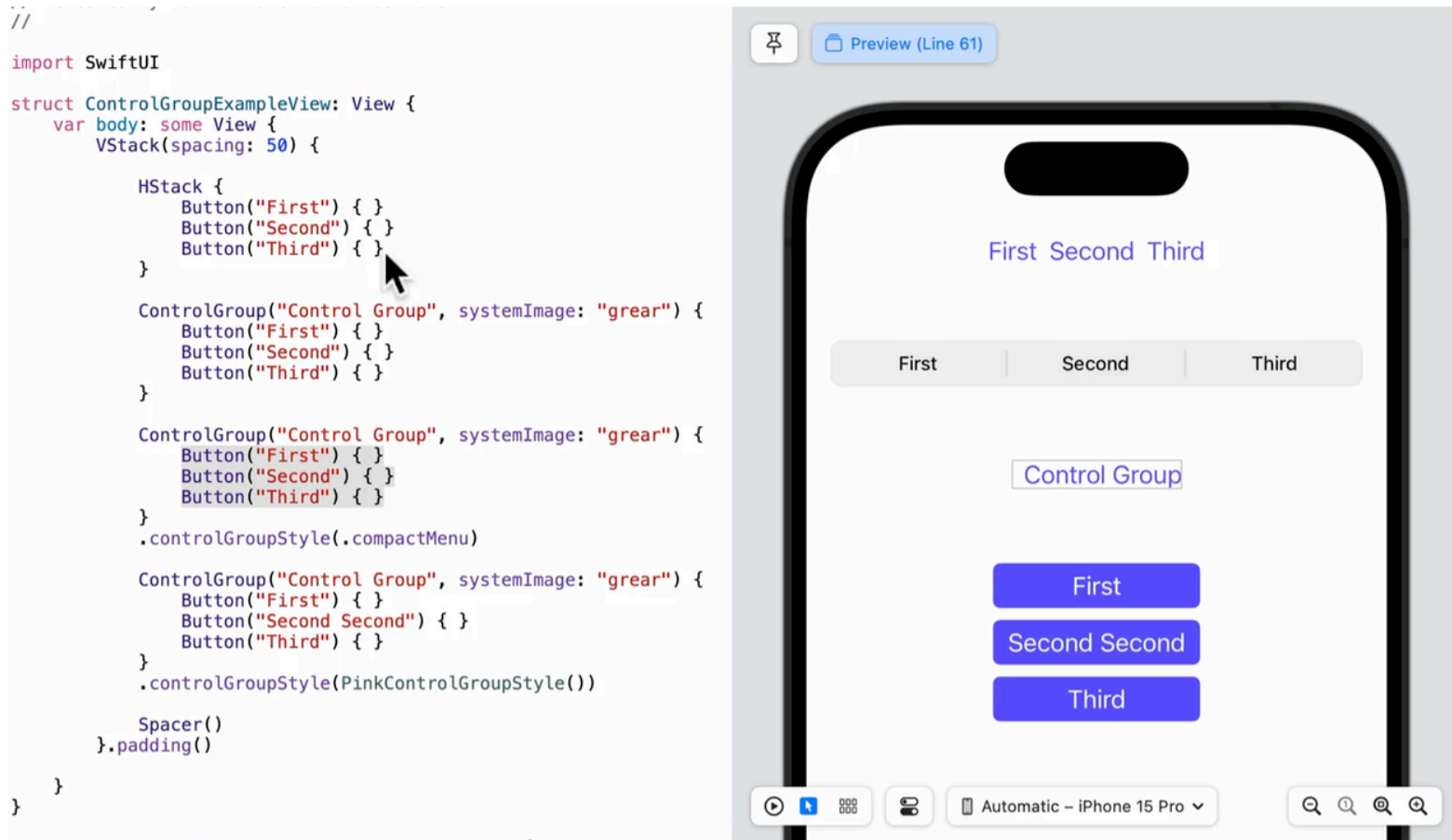
Variants

To make testing and debugging more efficient, Xcode offers variants for **color schemes, orientations, and dynamic type sizes**. By enabling variants, you can compare different options side by side and quickly identify any issues or inconsistencies in your layout.



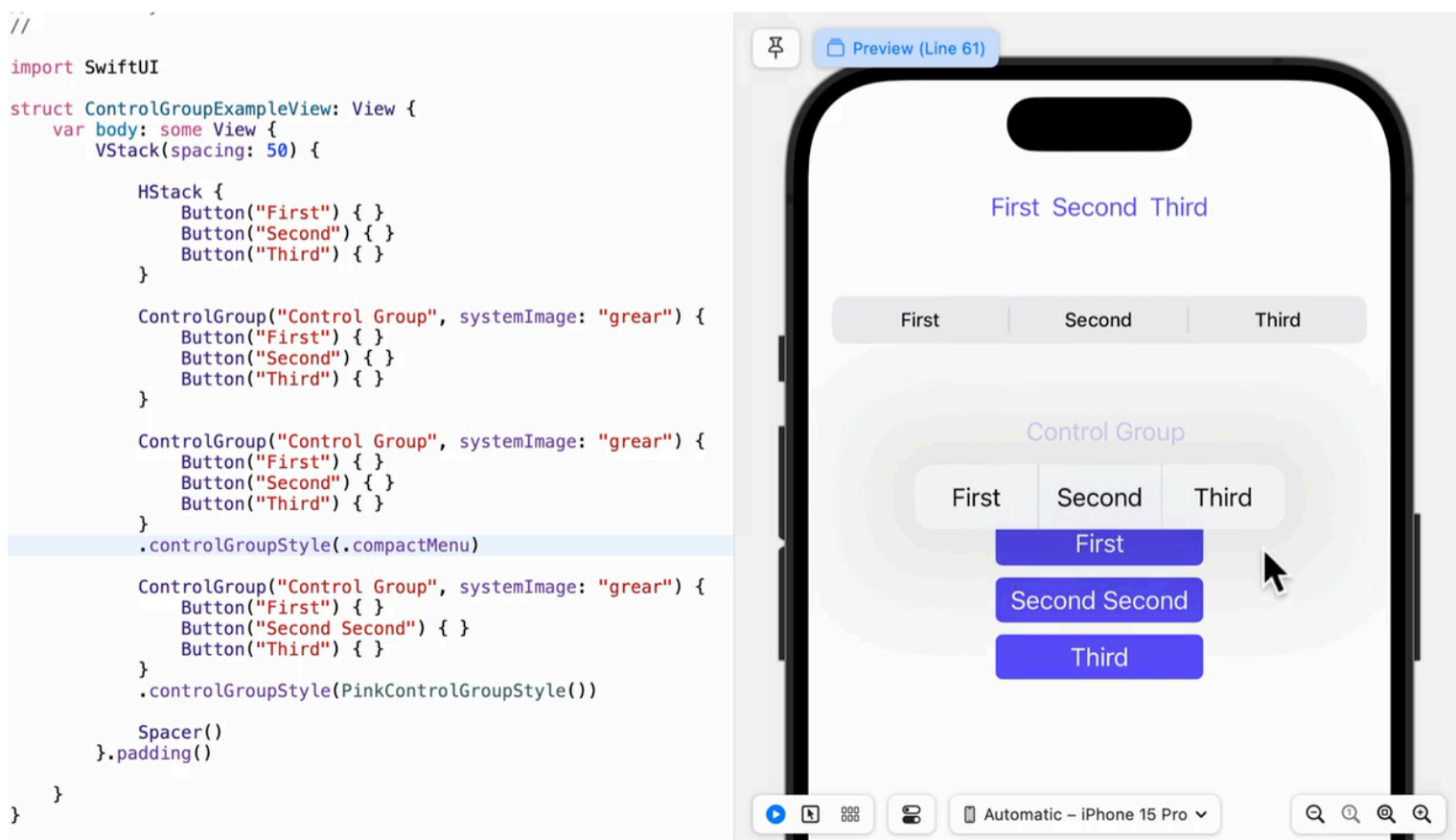
Selectable Preview

In the canvas area choose the second button in the bottom left corner to use the selectable preview feature. You can double-click on a specific view to highlight the corresponding code in the editor. This feature is particularly useful when working with complex views or collections, as it helps you identify which code snippets correspond to which views.



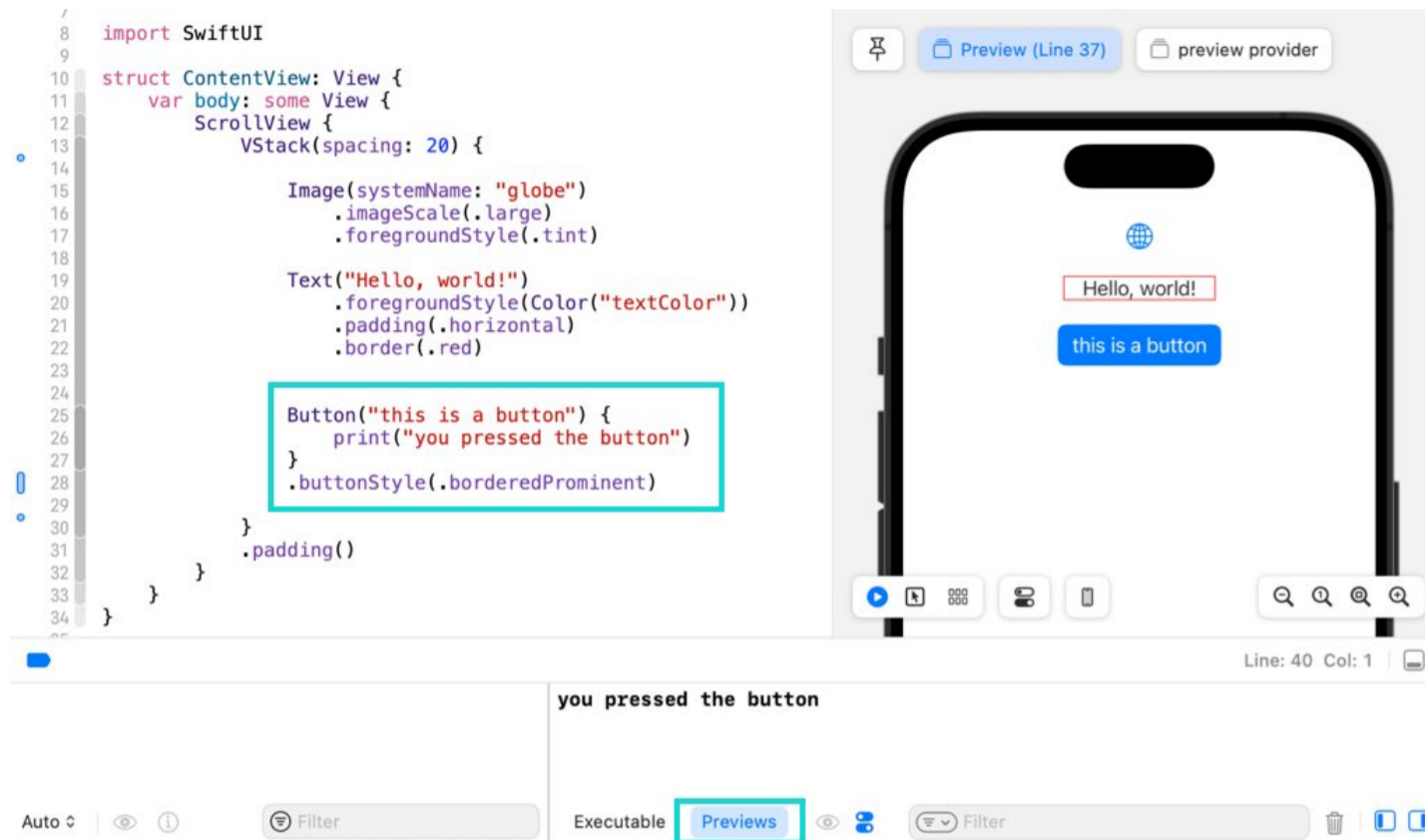
Live Preview

The live preview feature is handy for testing animations and interactions in real time. It allows you to see how your views respond to user interactions, such as tapping a button or scrolling a scroll view.



Debugging with Print Statements

To aid in debugging, the live preview also supports the use of print statements. You can add print statements to your code and observe the output in the debug area of the preview. This helps you verify if certain actions are being executed or if specific code paths are being triggered.



Pin Previews

In Xcode, you have the option to pin previews. This allows you to work in the context of a specific view, even when navigating between different files. Pinned previews are displayed at the top and can be easily accessed for quick reference. If you want to remove pinned previews, simply tap on the pin button again.

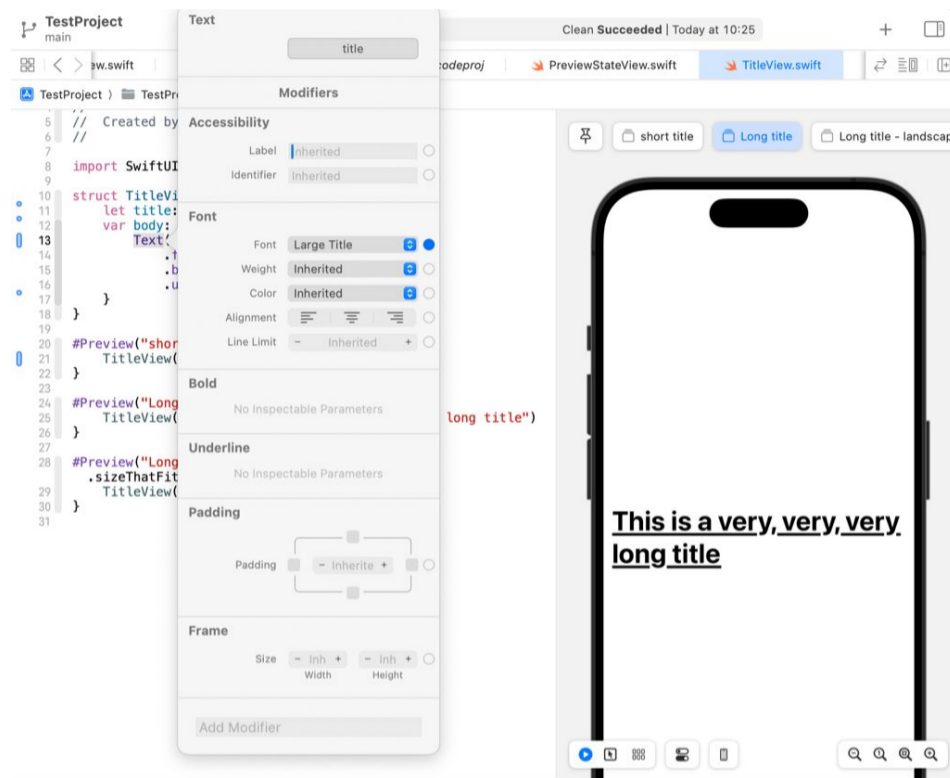


1.3 QUICK AND EFFICIENTLY EDIT SWIFTUI VIEWS

One of the challenges we often face with SwiftUI is locating the tools we need to make changes to our views. It can be frustrating trying to figure out what can be modified and how to modify it.

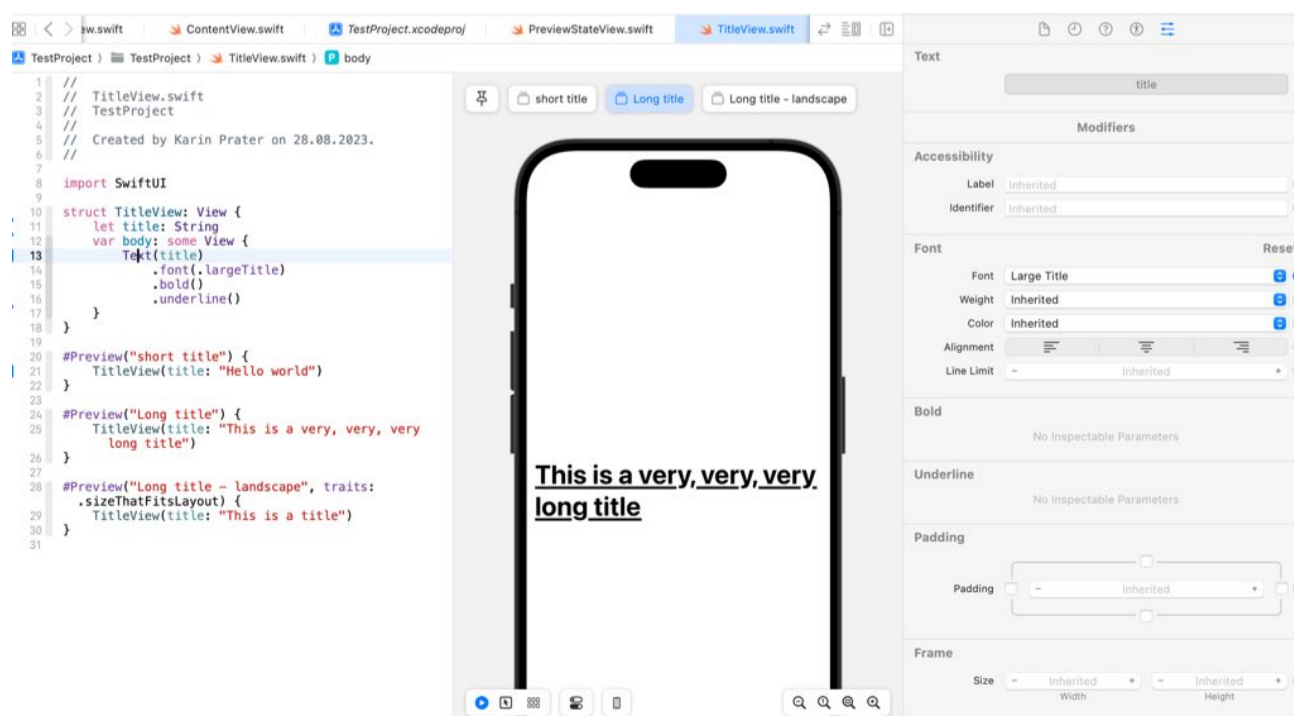
SwiftUI Inspector

Simply control-click on an element and select “Show SwiftUI Inspector.” This brings up a panel where you can quickly modify properties such as accessibility labels, paddings, frames, and even add additional modifiers like blur effects. You can also find these modifications in the editor, where they are represented as lines of code.



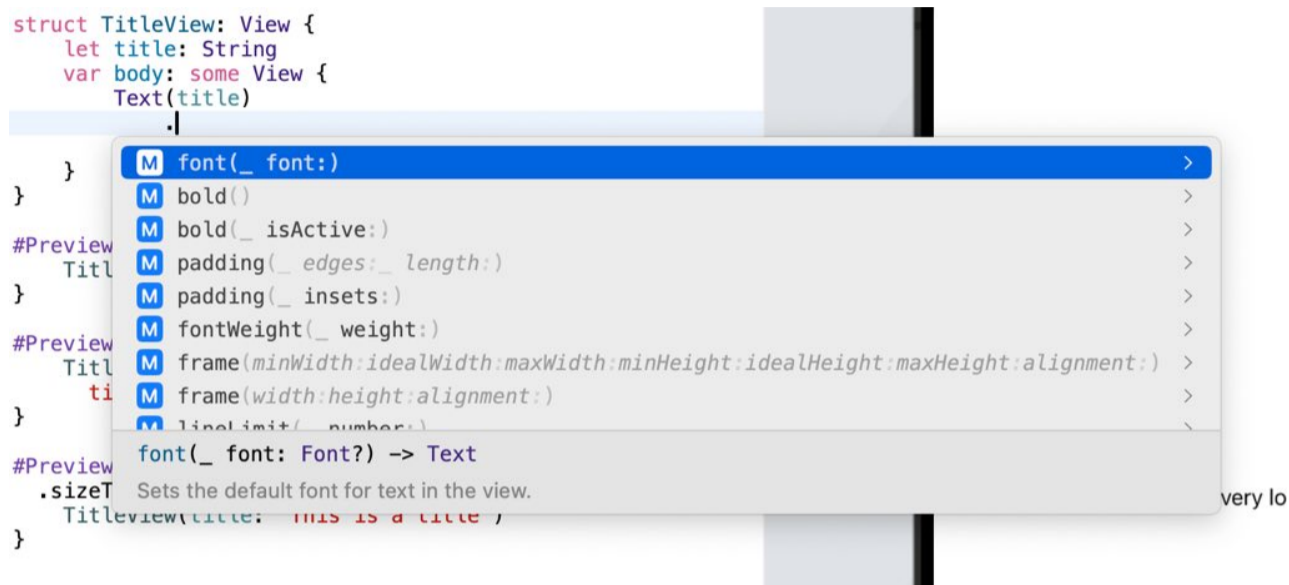
Attribute Inspector area

Another useful tool is the Attribute Inspector area, where you can find a list of arguments and modifiers for a particular view. Here, you can scroll through and easily make changes to properties such as accessibility labels, padding, and more.



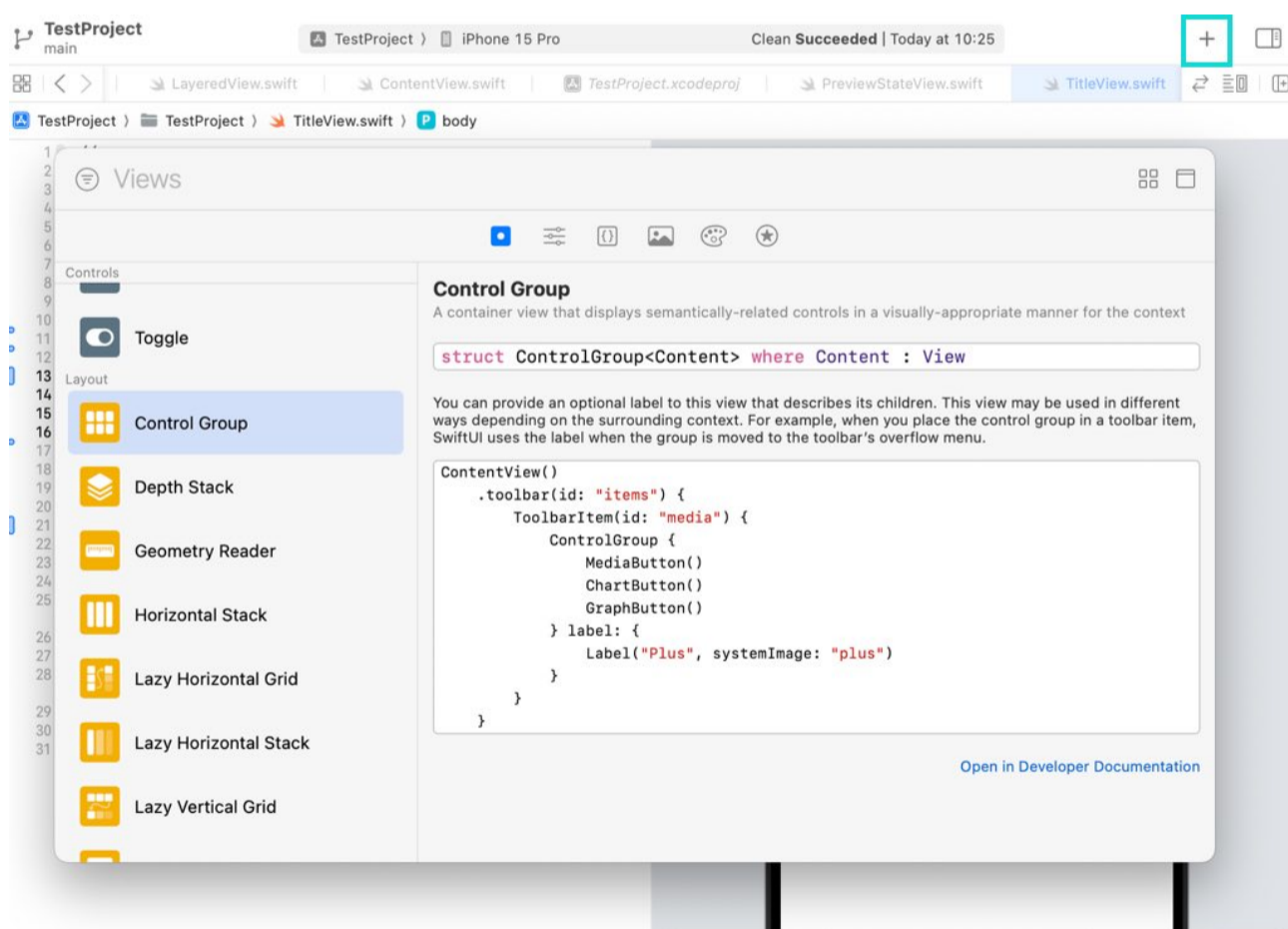
Xcode Auto Suggestions

Xcode has also become smarter in suggesting modifiers based on the context. For example, when working with a button, Xcode may suggest using a frame, navigation title, or padding. Similarly, when working with text, it may suggest using a multiline text alignment. These suggestions can save you time and effort in finding the right modifiers for your views.



Xcode Library

If you prefer a visual approach, you can utilize the Xcode library by clicking on the plus button. Here, you'll find a collection of icons, symbols, assets, colors, images, and code snippets. The library is organized into categories such as modifiers, effects, layout, text, images, list, navigation, and styling. This allows you to quickly browse through different options and easily add them to your code.



By familiarizing yourself with these built-in tools and resources, you can save valuable time that would otherwise be spent searching and googling for solutions. The documentation, in particular, can provide inspiration and helpful code snippets to enhance your SwiftUI skills.

1.4 DEBUGGING LAYOUT ISSUES

Debugging layout issues with SwiftUI can be a challenging task. I'm here to guide you through some helpful strategies that will make the process much easier.

Using the Selectable Preview

One useful tool for debugging layout issues is the inspector. By selecting a view, you can access information about its size and other properties. For example, you can identify if there is excessive padding causing unexpected spacing between views. By removing or adjusting the padding, you can resolve the issue and achieve the desired layout.

Adding Borders And Background Colors

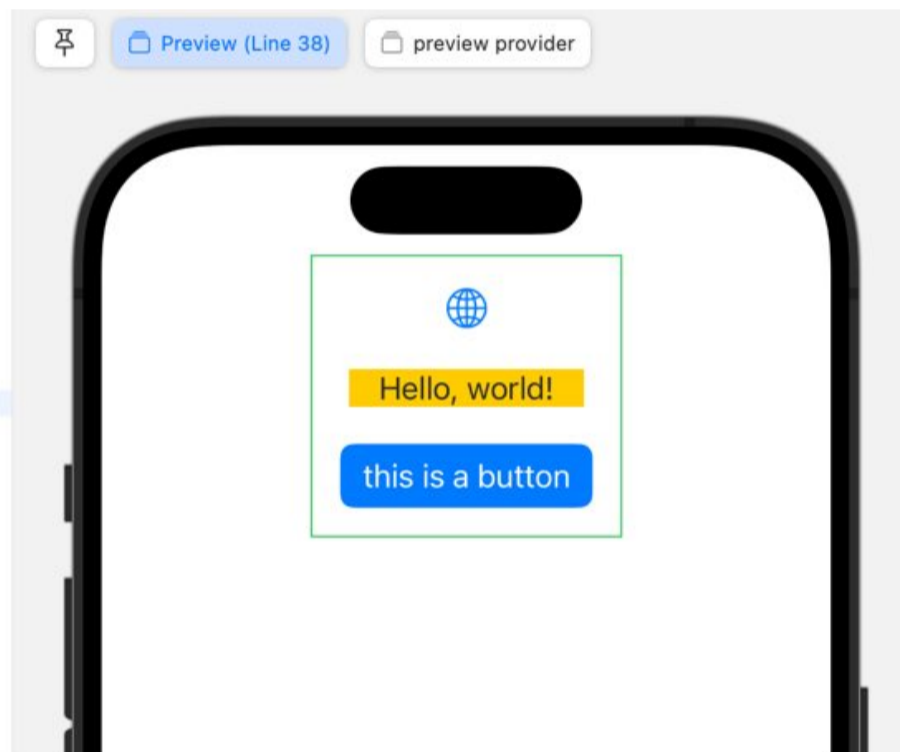
Additionally, if you have multiple views that could be causing the problem, such as text or buttons, you can add borders or background colors to visually differentiate them. This allows you to pinpoint the specific view that needs adjustment. By zooming in and examining the highlighted view, you can identify any padding modifiers, frames, or offsets that may be causing layout inconsistencies.

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        ScrollView {
            VStack(spacing: 20) {
                Image(systemName: "globe")
                    .imageScale(.large)
                    .foregroundStyle(.tint)

                Text("Hello, world!")
                    .foregroundStyle(Color("textColor"))
                    .padding(.horizontal)
                    .background(.yellow)

                Button("this is a button") {
                    print("you pressed the button")
                }
                .buttonStyle(.borderedProminent)
            }
            .padding()
            .border(Color.green)
        }
    }
}
```

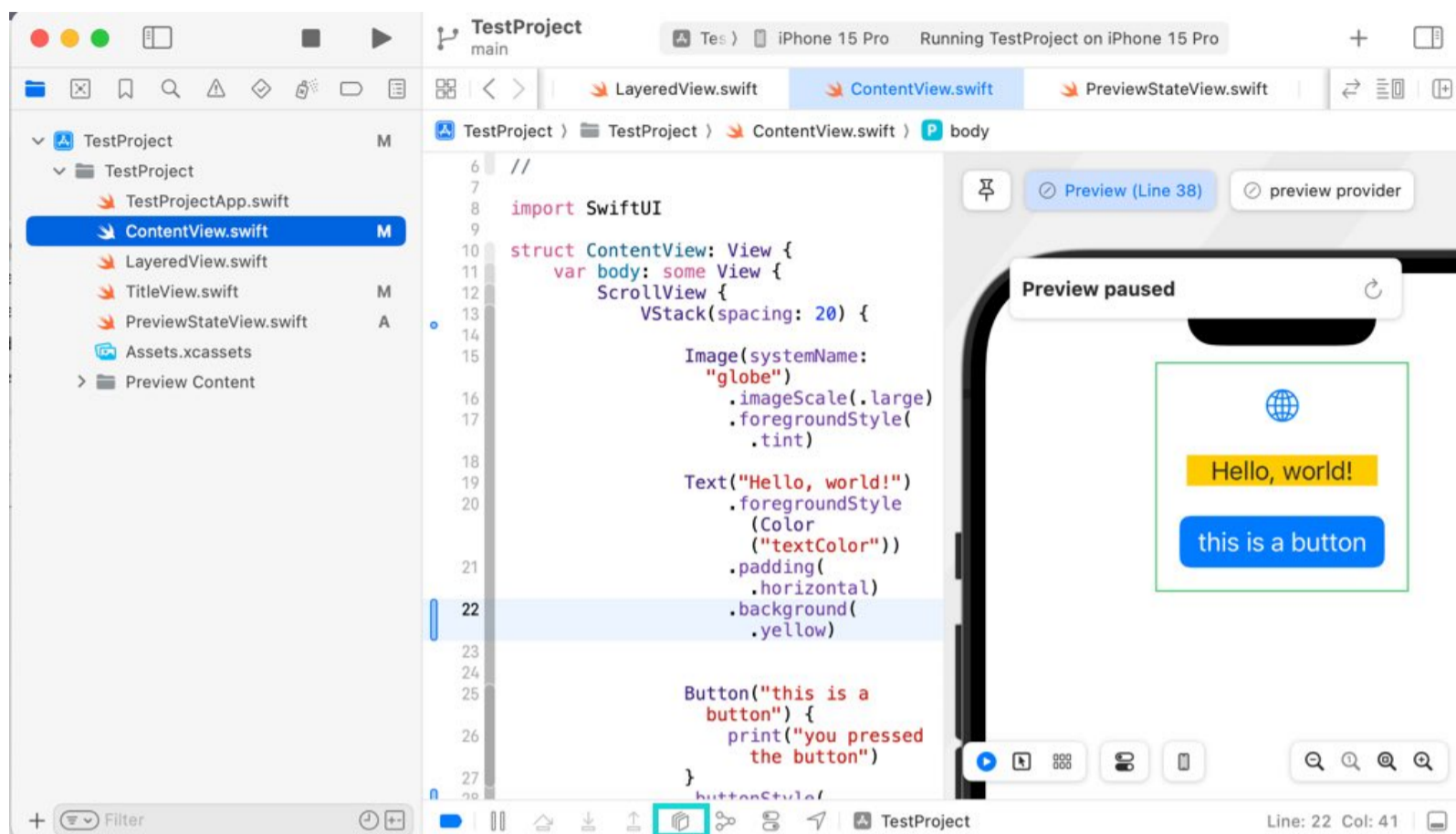


Debug View Hierarchy

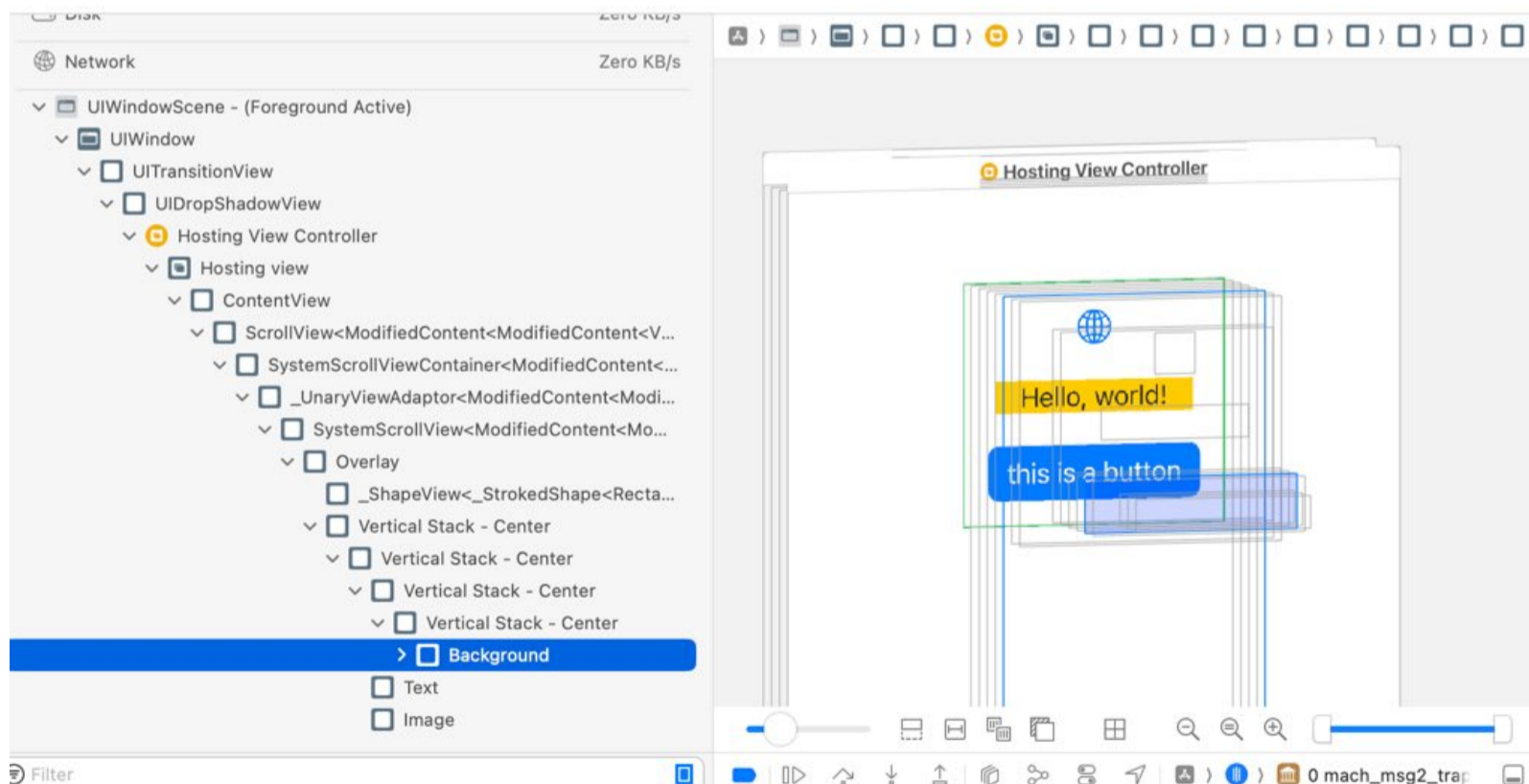
Understanding the view hierarchy is crucial for debugging layout issues. SwiftUI provides a debug view hierarchy feature that allows you to visualize the stack of views used in your project. By using this feature, you can navigate through the hierarchy and gain insights into how views are structured.

For example, if you want to locate where a specific title is defined, you can use the debug view hierarchy. By selecting the title, you can trace back to its parent views and identify the content view responsible for its creation. This feature is especially helpful when dealing with layered views or complex layouts.

Run your project and select the “Debug View Hierarchy” button at the bottom of the Xcode window:



Xcode will pause the simulator and open the Debug View Hierarchy. In the left navigator pan you can select the views and layers:



Monitoring View Updates

Sometimes, you may encounter performance issues or frequent redrawing without understanding which views are causing the problem. In such cases, it's useful to monitor view updates and identify which views are being recreated. You can achieve this by using the `print changes` statement:

```
struct TitleView: View {
    let title: String

    var body: some View {
        Self._printChanges()
        return Text(title)
            .font(.largeTitle)
            .bold()
            .underline()
    }
}
```

Thus, you can track how many times the view is recreated or updated. This can provide valuable insights into the impact of changes on specific views.

By observing the print statements in the debugger, you can determine which views are being updated and how often. This helps you understand the relationship between view updates and potential layout issues.

1.5 SWIFTUI TREE OF DOOM

When working with SwiftUI, you may encounter situations where your **views become large and complex**, leading to slow previews or unhelpful error messages from Xcode. This is commonly referred to as the “tree of doom” in SwiftUI, where nesting levels can become overwhelming. Here are a few strategies to work against this.

Extracting Subviews

To simplify the view hierarchy, you can use the “Extract Subview” feature by Ctrl-clicking on the problematic view. This will extract the view into a separate subview. You can then rename it to something more meaningful. Personally, I prefer moving these subviews to separate files rather than leaving them within the same file. This way, I can easily navigate through each view file and see its preview directly. It also prevents the subviews from getting lost amidst the main view code.

Small Views

Remember, it’s always beneficial to break down your views into smaller, more manageable pieces. Personally, I find it helpful to keep the body of a view smaller than 100 lines. However, you should experiment and find what works best for you, as everyone’s preferences and project requirements may vary.

Code Highlighting

If you encounter long containers, such as a VStack with numerous subviews, it can be challenging to identify where the container ends. To solve this, you can utilize Xcode’s highlighting feature. By clicking on the curly braces at the beginning of the container, Xcode will highlight the corresponding closing curly brace, indicating the end of the container. This helps in making changes or adding modifiers to the container.

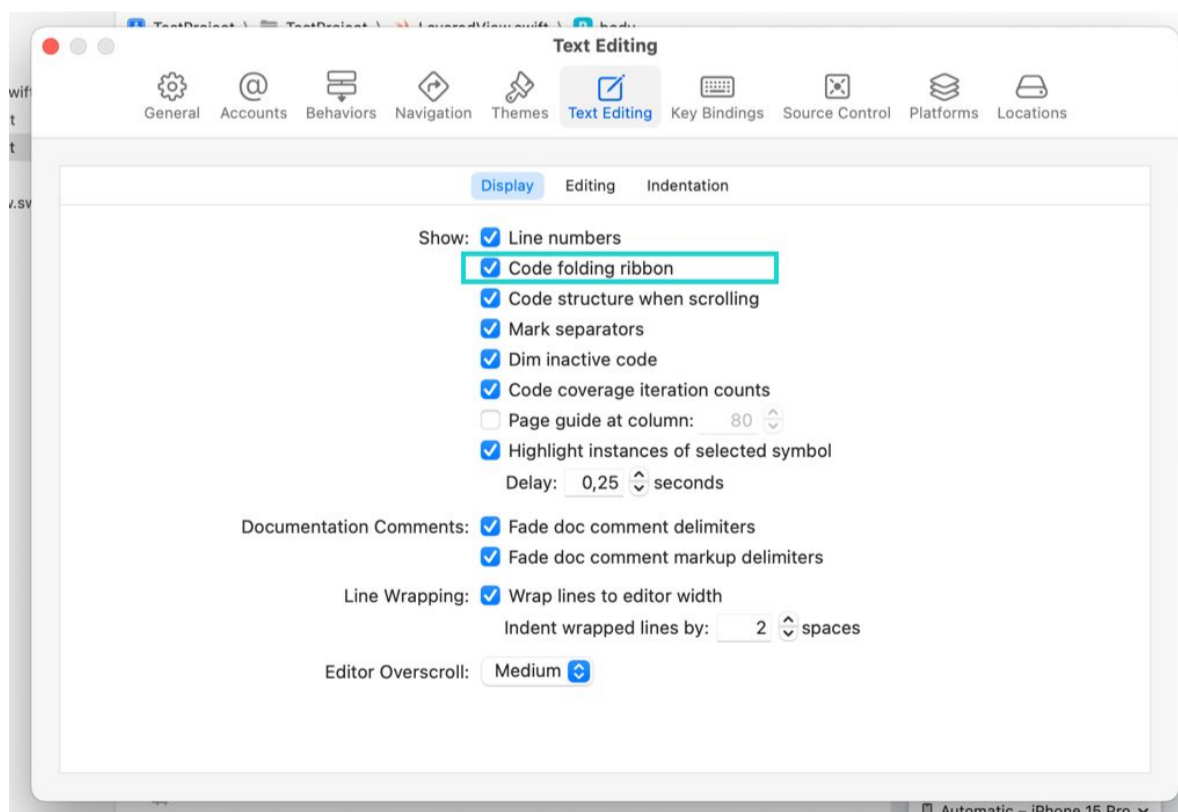
```
7
8  import SwiftUI
9
10 struct LayeredView: View {
11
12     let text = "this is an example text"
13     var computedValue: String {
14         "check something"
15     }
16
17     var body: some View {
18         VStack {
19
20             Text(text)
21             Text(text)
22             Text(text)
23             Text(text)
24             // Text(text)
25
26             VStack {
27                 Divider()
28                 Text(text)
29                 Text(text)
30                 Text(text)
31                 Text(text)
32                 Text(text)
33             }
34
35             Divider()
36             Text(text)
37         }
38     }
39 }
40
41 #Preview {
42     LayeredView()
43 }
44
```

Folding Ribbons

In cases where your code extends beyond the visible area, you can make use of the code folding ribbons. These ribbons allow you to hide or show specific sections of your code, making it easier to focus on the relevant parts.

```
7
8 import SwiftUI
9
10 struct LayeredView: View {
11
12     let text = "this is an example text"
13     var computedValue: String {
14         "check something"
15     }
16
17     var body: some View {
18         VStack {
19             // ...
20         }
21     }
22 }
23
24 #Preview {
25     LayeredView()
26 }
27 }
```

To enable or disable the ribbons, go to Xcode settings, specifically the “Text Editor” area, and toggle the “Show Code Folding Ribbons” option.



By simplifying your complex SwiftUI views, you can enhance the performance of previews, reduce errors, and make your code more maintainable. So, don't just rely on Xcode's default behavior, take control of your code and make it more understandable and efficient.

1.6 TYPICAL PROBLEMS WITH XCODE AND SWIFTUI AND HOW TO FIX THEM

Sometimes, while working with Xcode and SwiftUI, you may encounter certain issues that can be a bit frustrating. Unfortunately, Xcode doesn't always provide the most informative error messages. In this section, I will walk you through some common scenarios for errors, explain why they occur, and show you how to resolve them.

Invalid Redefinition of View Names

One error that you might come across is when you have a view with the same name declared multiple times. For instance, if you copy a struct called "LayeredView" to your ContentView, you will get an error message saying *"invalid redeclaration of LayeredView."* This error occurs because you have already declared a view with the same name. To fix this, you can use the search function to locate the duplicate view and rename it accordingly.

Uninitialized Properties

Another issue you might encounter is when you have a property declared within a struct but it is not initialized. Xcode will display an error message stating *"property declared as object return type but has no initializer."* To resolve this error, you need to provide an initial value for the property. Even if it's just a placeholder like a Text view, it will help Xcode infer the underlying type correctly.

Missing Return Value for Text Views

Similarly, you might face problems when using Text views. If you don't return anything within the body property, an error will be thrown. For example:

```
import SwiftUI

struct TitleView: View {
    let title: String

    var body: some View {
        |
    }
}
```

Property declares an opaque return type, but...

Missing return in accessor expected to return 'some View'

To fix this, make sure to return a view within the body property. You can use a Text view with some content, even if it's temporary. However, it's best practice to declare constants or computed values outside of the body property for better code organization.

Missing Environment Objects in SwiftUI Previews

If you use environment objects in your views or subviews, make sure to also add them in the preview. Similarly, you need to inject a context when working with CoreData or SwiftData:

```
#Preview {
    ContentView()
        .environment(MyViewModel())
        .environment(\.managedObjectContext, NewContext())
}
```

Troubleshooting Tips

If you encounter persistent issues and can't figure out the problem, here are a few troubleshooting tips:

1. Uncomment the views you just created and implement them one by one to identify any potential errors.
2. **Clean the build folder** by going to Product > Clean Build Folder and then rebuild your project.
3. Sometimes, **running the project** instead of relying solely on the preview can help resolve issues.
4. In extreme cases, **restarting Xcode or even your Mac** might be necessary to resolve stubborn issues.

Overall, Xcode previews with the Canvas feature are incredibly useful and can save you a lot of time. In the upcoming lessons, you will see how these previews can streamline your development process by eliminating the need to build and run your project repeatedly.

2. PRIMITIVE LAYOUT COMPONENTS

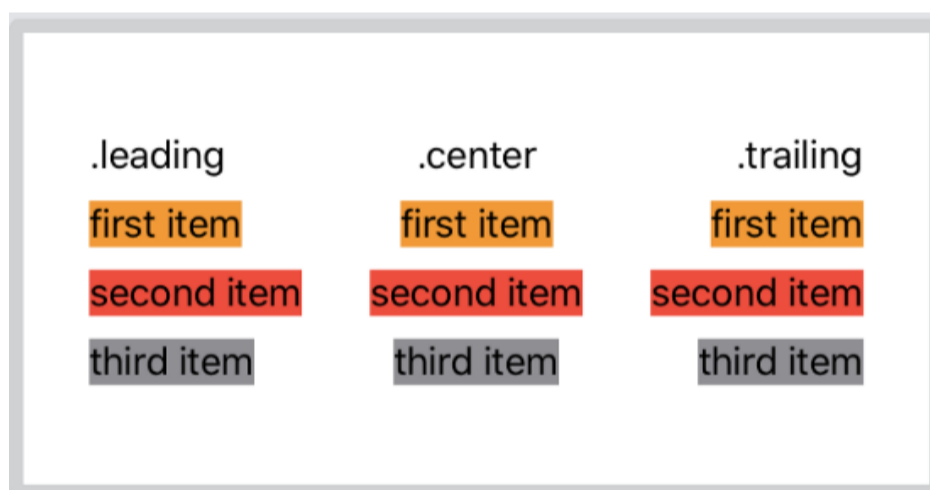
2.1 VSTACK, HSTACK AND ZSTACK

VStack - Vertical Stacking

VStack is a container view that arranges its child views vertically. To create a VStack, you can use the “Embed in VStack” option after control-clicking on the view. By default, a VStack adjusts its size to fit its child views. The size of the VStack is determined by the space occupied by its children.

You can customize the alignment and spacing of the child views within the VStack. For example, you can align the views to the leading edge and set a spacing of 20 between them. Additionally, you can nest multiple VStacks to create more complex layouts.

```
VStack(alignment: .center,  
      spacing: 10) {  
  Text("first item")  
    .background(Color.yellow)  
  
  Text("second item")  
    .background(Color.red)  
  
  Text("third item")  
    .background(Color.gray)  
}
```



HStack - Horizontal Stacking

HStack is a container view that arranges its child views horizontally. Similar to VStack, you can use the “Embed in HStack” option to create an HStack. The alignment property of an HStack determines how the child views are aligned vertically.

You have various alignment options available such as center, top, bottom, first text baseline, and last text baseline. These options allow you to align the child views based on their text baselines or other criteria. By changing the font size or adding different-sized views, you can observe the effects of alignment within an HStack.

```

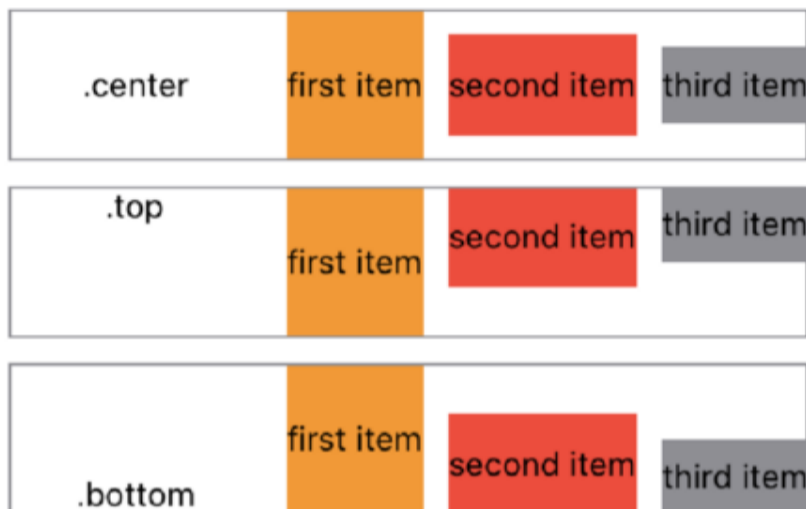
HStack(alignment: .firstTextBaseline,
      spacing: 10) {

  Text("first item")
    .background(Color.yellow)

  Text("second item")
    .background(Color.red)

  Text("third item")
    .background(Color.gray)
}

```



ZStack - overlay stacking

ZStack is a container view that stacks its child views on top of each other, creating a layered effect. The order in which the child views are added to the ZStack determines their stacking order, with the first view being the furthest behind.

You can adjust the stacking order using the **zIndex view modifier** or by changing the order of the child views. Additionally, you can customize the alignment of the child views within the ZStack, such as top, leading, center, or combinations of them.

```

ZStack(alignment: .center) {
  Text("first item")
    .padding(.vertical, 20)
    .background(Color.yellow)
    .zindex(2)

  Text("second item")
    .padding(.vertical, 10)
    .background(Color.red)

  Text("third item")
    .background(Color.gray)
}

```

You can also use the alignment property to align the views within the ZStack.

For Text views, you might also want to use **.leadingLastTextBaseline**, and **.trailingFirstTextBaseline** etc.



ZStacks are particularly useful when you want to overlay views on top of each other. You can use them to create visually appealing effects, such as combining images with text or applying gradients and backgrounds.

```

struct CatExampleView: View {
    var body: some View {
        ZStack(alignment: .bottomLeading) {
            ResizableImageView(imageName: "cat_1")

            Text("Cats are awesome")
                .font(.title).bold()
                .background(Color.white)
                .padding()
        }
    }
}

```



2.2 DIVIDER AND SPACER

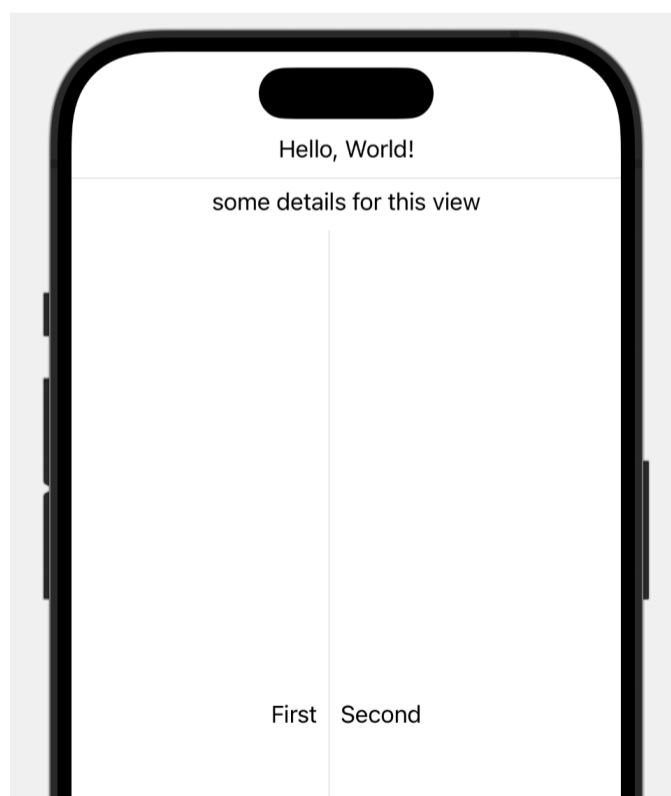
In this section, we will explore two fundamental layout views in SwiftUI: dividers and spacers. These views play a crucial role in organizing and structuring your user interface.

Imagine you have a `VStack` with various pieces of information. You want to visually separate two specific views within this stack. To achieve this, you can simply add a `Divider`. This will create a thin line between the two views, providing a clear visual distinction. Depending on whether you place the divider in a `VStack` or an `HStack`, it will appear as a horizontal or vertical line respectively. SwiftUI is smart enough to adapt the appearance of the divider based on its container stack.

```
struct DividerExampleView: View {
    var body: some View {
        VStack {
            Text("Hello, World!")

            Divider()
            Text("some details for this view")

            HStack {
                Text("First")
                Divider()
                Text("Second")
            }
        }
        // .fixedSize()
    }
}
```



Dividers, being a “greedy” view, strive to occupy as much space as possible. This means they expand to fill the available space in the layout. For instance, if you want to minimize the height of the divider and have it only as tall as the two text views, you can use the `fixedSize()` modifier.

```
HStack {  
  Text("First")  
  Divider()  
  Text("Second")  
}  
.fixedSize(horizontal: false, vertical: true)
```

First | Second

By applying the **fixedSize()** modifier, the divider's height is constrained to match the height of the views it separates, resulting in a more compact appearance.

Moving on to **Spacer**, they are incredibly useful for controlling the distribution of space within a layout. A spacer view takes up all the available space in a given axis and pushes the surrounding views accordingly.

Imagine you have a vertical stack and you want to position it at the top of the screen instead of the default center alignment. To achieve this, you can use a spacer to expand the stack's height and push it to the top.



```

struct SpacerExampleView: View {
    let superhero = SuperHero.example
    var body: some View {
        VStack(alignment: .leading) {
            Text(superhero.name)
                .font(.title)
            Text(superhero.biography)

            Spacer()
        }
    }
}

```

By adding the `Spacer` view, it occupies the remaining space at the bottom of the screen and pushes the stack upwards, aligning it with the top edge.

Spacers can also be used to adjust the spacing between views. For example, if you have three buttons arranged horizontally, you can add spacers between them to control their positioning.

```

HStack(spacing: 0) {
    Spacer(minLength: 0)
    Button("First") { }
    Spacer(minLength: 10)
    Button("Second") { }
    Spacer(minLength: 10)
    Button("Third") { }
    Spacer(minLength: 0)
}

```



In this case, the spacers distribute the available space evenly between the buttons, resulting in a visually appealing layout. Alternatively, you can use the **Color view as a spacer** by setting its background color to match the desired spacing.

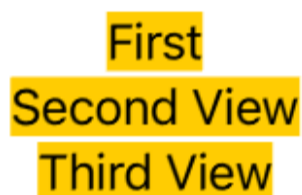
Using dividers and spacers in your SwiftUI layouts allows for greater flexibility and adaptability across different screen sizes.

2.3 GROUP

In SwiftUI, groups are container views that do not handle the layout of their children. Instead, the layout is determined by the container view they are placed in, such as an HStack or VStack. However, groups offer a convenient way to apply the same view modifier to all their children individually.

Where groups shine is their ability to apply view modifiers to multiple child views simultaneously. Let's say you want to add a yellow background to all the text views. With a group, you can simply apply the background modifier to the group:

```
Group {
    Text("First")
    Text("Second View")
    Text("Third View")
}
.background(Color.yellow)
```



The diagram shows three text views stacked vertically: "First", "Second View", and "Third View". Each text view is highlighted with a yellow rectangular background, demonstrating the effect of applying a background modifier to a group of views.

Unlike when using a VStack or HStack, where the background modifier would be applied to the entire stack, the group allows you to apply the same modifier to each individual child view. This can be incredibly convenient, especially when you want to avoid duplicating code.

Another use case for groups is when you have conditional code that requires specific view modifiers. For example, let's say you want to display a different view based on whether a user is logged in or not:

```
struct GroupExampleView: View {
    let isLoggedIn: Bool
    var body: some View {
        VStack {
            Group {
                if isLoggedIn {
                    Text("Thank you for signing up")
                } else {
                    Text("You need to log in to get access")
                }
            }
            .foregroundColor(Color.blue)
        }
    }
}
```

In this case, applying the foregroundColor modifier directly to the conditional code would result in a crash. However, **by wrapping the condition in a group, you can apply the modifier** without any issues.

To summarize, groups in SwiftUI do not handle the layout of their children. Instead, they rely on the container view they are placed in to handle the layout. However, groups provide a convenient way to apply the same view modifier to all their children individually, making your code more efficient and avoiding unnecessary repetition.

2.4 GROUPBOX

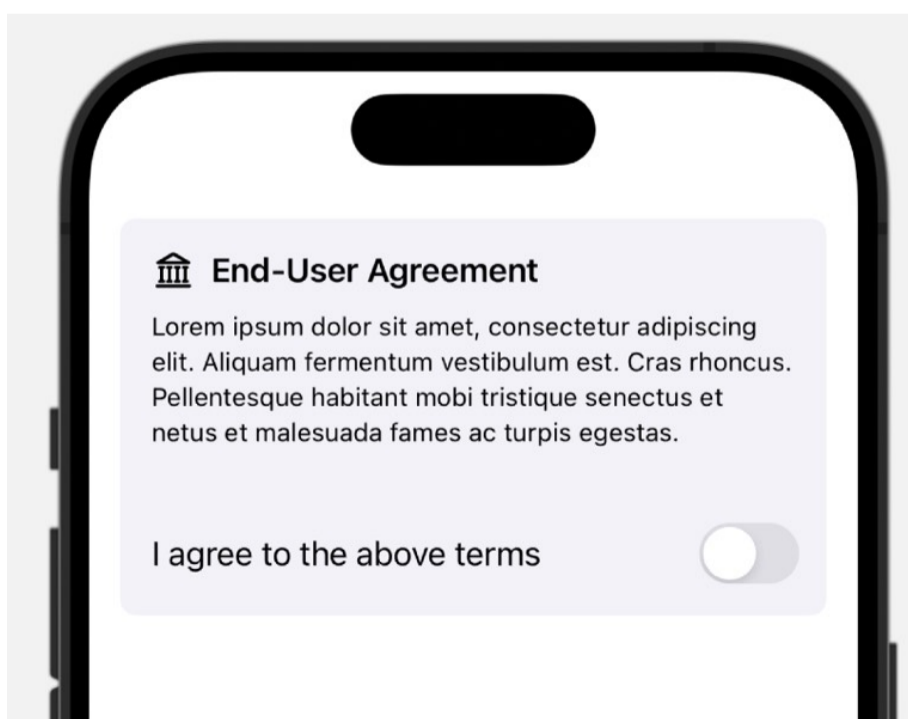
In SwiftUI, there are various container views available for organizing and styling your interface. One such container is the `GroupBox`. Unlike the basic `Group` view, which doesn't apply much styling, the `GroupBox` allows you to create **card-like layouts** with ease.

To use a `GroupBox`, you simply define a title or label and the content you want to display. You can also apply some padding to enhance the styling. The label is typically displayed in a headline font style, giving it a prominent appearance.

```
struct GroupBoxExampleView: View {
    @State private var userAgreed: Bool = false
    let agreementText: String = "Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Aliquam fermentum vestibulum est. Cras rhoncus. Pellentesque habitant mobi
tristique senectus et netus et malesuada fames ac turpis egestas."

    var body: some View {
        GroupBox(label: Label("End-User Agreement",
                              systemImage: "building.columns"),
                 content: {
                    Text(agreementText)
                      .font(.footnote)

                    Toggle(isOn: $userAgreed) {
                        Text("I agree to the above terms")
                    }
                })
        .groupBoxStyle(.automatic)
        .padding()
    }
}
```



In SwiftUI, many container views have specific styling modifiers tailored to their functionality. You can also create your own custom styling for the `GroupBox`. By conforming to the **`GroupBoxStyle`** protocol, you can define a unique appearance and interaction for all `GroupBox` instances within your view hierarchy.

```
struct OrangeGroupBoxStyle: GroupBoxStyle {
    func makeBody(configuration: Configuration) -> some View {
        VStack(alignment: .leading) {
            configuration.label
                .font(.title)
            configuration.content
        }
        .padding()
        .background(
            RoundedRectangle(cornerRadius: 5.0)
                .fill(Color.orange)
                .shadow(radius: 5)
        )
    }
}
```

In this custom styling example, we can use a `VStack` to create multiple `GroupBox` instances with different titles and contents. By applying our orange group box style to these instances, we can see the visual transformation.

```
GroupBox(titleText) {
    Text(agreementText)
}
.groupBoxStyle(OrangeGroupBoxStyle())
```



While `GroupBox` may not be one of the most commonly used views, it can be invaluable in creating visually appealing card-like layouts. Especially in complex apps with numerous subviews, using `GroupBox` can help break down information and provide a more intuitive user experience.

2.5 CONTROLGROUP

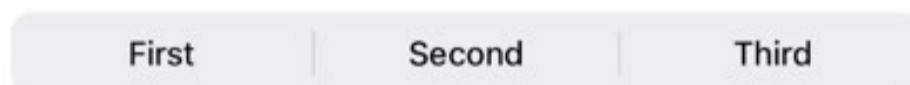
In SwiftUI, control groups are a powerful tool for adding specific styling to control views, such as buttons, that you want to group together. Control groups allow you to create a cohesive and visually appealing layout for your user interface.

To create a control group, you can use the `ControlGroup` view. Let's compare putting buttons in a `HStack` vs. a `ControlGroup`:

```
HStack {
    Button("First") { }
    Button("Second") { }
    Button("Third") { }
}

ControlGroup("Control Group", systemImage: "gear") {
    Button("First") { }
    Button("Second") { }
    Button("Third") { }
}
```

First Second Third

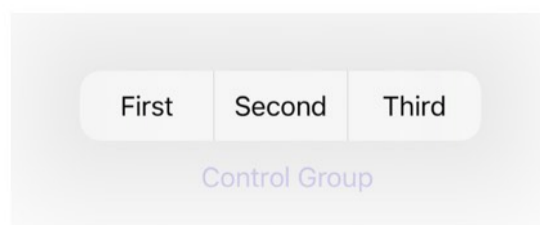


In this example, we have three buttons grouped together within a control group. By wrapping the buttons in a control group, they are visually styled as a cohesive unit. You can still interact with each button individually, but they appear as a single entity.

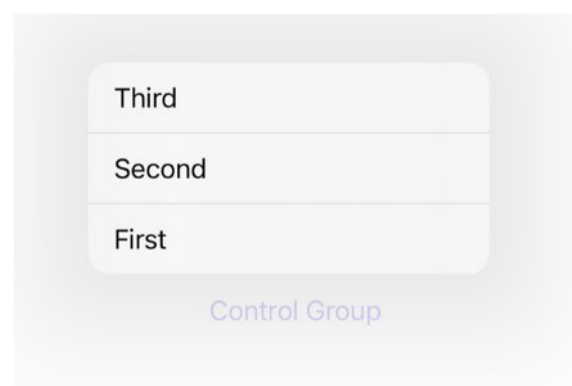
Control groups also offer various styling variations. For example, you can use system-provided styles like `compactMenu` to display the control group as a menu:

```
ControlGroup("Control Group", systemImage: "gear") {
    ...
}
.controlGroupStyle(.compactMenu)
```

Control Group



.compactMenu



.palette

Control groups are particularly useful when you have a set of controls that you want to reuse in multiple places within your app. They automatically adjust their layout based on their placement, making it easy to maintain a consistent design throughout your user interface.

You can also customize the styling of a control group by creating a custom control group style. Here's an example:

```
struct EqualSizControlGroupStyle: ControlGroupStyle {
    func makeBody(configuration: Configuration) -> some View {
        VStack {
            configuration.content
                .foregroundColor(.white)
                .padding(.vertical, 5)
                .padding(.horizontal, 10)
                .frame(maxWidth: .infinity)
                .background(
                    RoundedRectangle(cornerRadius: 5)
                        .fill(Color.accentColor)
                )
        }
        .fixedSize(horizontal: true, vertical: false)
    }
}
```

Here is how you can use your custom Group Styling:

```
ControlGroup("Control Group", systemImage: "gear") {
    Button("First") { }
    Button("Second Second") { }
    Button("Third") { }
}
.controlGroupStyle(EqualSizControlGroupStyle())
```

This group will place all its children in a VStack and add background rectangles that have all the same width:



3. LAYERING VIEWS

In this section, we will explore the concept of layering views in SwiftUI. Layering views allows us to create visually appealing and dynamic user interfaces by combining multiple elements together. I will cover various techniques and modifiers that enable us to control the order and appearance of views within our layouts.

Throughout this section, I will dive into the following key topics:

- **Background Modifier:** Learn how to set a background color or image for your views, providing a solid foundation for your UI elements.
- **Overlay Modifier:** Discover how to overlay additional views on top of existing ones, allowing for creative design choices and visual enhancements.
- **ZStack** container can be used to layer views. I covered this in [a previous section](#)
- **ZStack vs Background/Overlay:** Understand the differences between using a ZStack and applying background/overlay modifiers, and when to choose one over the other.
- **Color View and Gradients:** Explore different ways to apply colors to your views, from solid colors to gradients. These views are often used for backgrounds and overlays.
- **Materials:** Discover how to apply materials effects to your view background.

3.1 BACKGROUND MODIFIER

The background modifier in SwiftUI allows to add a background to a view. We can easily add a background to this text by applying the **background** modifier and specifying a view, such as a color, image, or shape for the background:

```
Text("Hello, World!")
    .padding()
    .background {
        Color.cyan
    }

Text("Cats are awesome")
    .padding()
    .background {
        Image("abstract-pool-water")
            .resizable()
            .scaledToFill()
    }
    .clipped()

Text("More")
    .padding(.horizontal)
    .background {
        Capsule().fill(Color.cyan.gradient)
    }
```



The great thing about the background modifier is that it fills out the background of the view it is attached to without increasing the size of the view itself, unlike the `ZStack`. This makes it perfect for color backgrounds. If you want to **make the view larger, you can use other modifiers like padding or frame**.

In addition to colors, you can also use **images** as backgrounds. By using the `resizable` modifier, you can resize the image to fit the available space in the background. You can also use the `scaleToFill` modifier to maintain the aspect ratio of the image while filling the background. **To prevent the image from overflowing, you can use the clipped modifier**.

Shapes can also be added as backgrounds. For example, you can add a capsule with a gradient fill using the background modifier.

SuperHero Example Card

Now, let's move on to a more interesting example. Imagine we want to create a superhero card view. We can define a `SuperHeroView` struct conforming to `View` and add an image of the superhero along with their name. By applying the background modifier, we can add a color or gradient background to the view. To achieve a card-like appearance, we can use the `cornerRadius` modifier or a rounded rectangle shape.

```
struct SuperHeroView: View {
    let superhero = SuperHero.example2
    var body: some View {
        ResizableImageView(imageName: superhero.imageName)
            .padding([.leading, .top])
            .background(alignment: .topLeading) {
                Text(superhero.name)
                    .font(.largeTitle)
                    .bold()
                    .foregroundColor(.white)
                    .padding()
            }
            .background(
                RoundedRectangle(cornerRadius: 15)
                    .fill(Color.cyan.gradient)
            )
            .compositingGroup()
    }
}
```

```
        .shadow(radius: 10)  
        .padding()  
    }  
}
```



The size of the view depends on the image size by default. You can also add multiple background modifiers to further enhance the visual appeal of the card.

It's worth mentioning that when applying the background modifier, **the order of modifiers becomes crucial**. For example, if you want to add text on top of the image, you need to ensure that the text is placed before the background modifier.

Additionally, you can use the **alignment parameter** to control the alignment of the background within the view. I used this parameter to align the superhero text to the top leading edge.

To handle safe areas, the background modifier provides the **ignoresSafeAreaEdges** parameter. This allows you to extend the background into the safe areas.

```
.background(Color.cyan, ignoresSafeAreaEdges: .top)
```

Background Styles and Shapes

In iOS 15 and macOS 12, Apple introduced new background styles and shapes. You can use these to create more visually appealing backgrounds. There are more convenient ways to place shapes behind a view with the background modifier where you can give a ShapeStyle (e.g. a color or gradient) and a shape:


```
Text("Capsule with a gradient background")
    .foregroundColor(.white)
    .padding()
    .background(Color.cyan.gradient, in: Capsule())
```



Capsule with a gradient background

This can be separated out into two modifiers:

```
VStack {
    Text("background style")
        .padding()
        .background(in: RoundedRectangle(cornerRadius: 5))
}
.padding()
.background(Color.yellow)
```



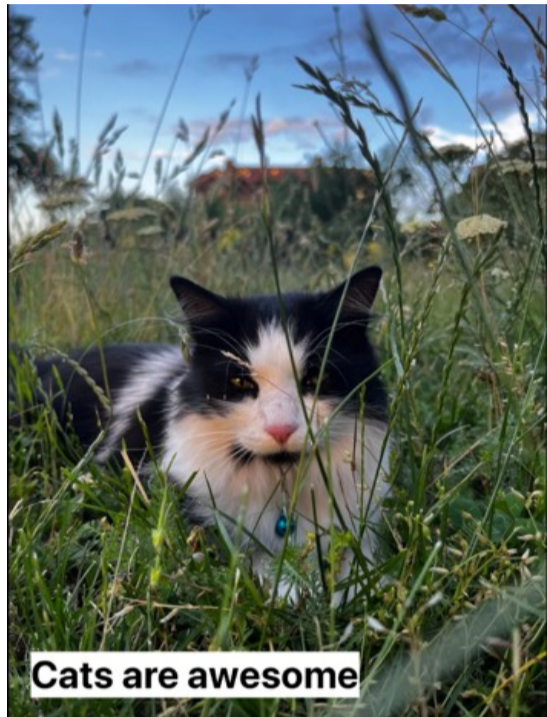
background style

You can set the background style independently for example to change the color of a GroupBox:

```
GroupBox {
    Text("GroupBox")
}
.backgroundStyle(Color.cyan.gradient)
```

3.2 OVERLAY MODIFIER

In SwiftUI, we have the overlay modifier, which allows us to place views on top of another view. Similar to the background modifier that places views behind a certain view, overlay lets us layer views on top.



```
ResizableImageView(imageName: "cat_1")
    .overlay(alignment: .bottomLeading) {
        Text("Cats are awesome")
            .font(.title).bold()
            .background(Color.white)
            .padding()
    }
```

Let's consider another example using a superhero image. Suppose we have a separate `SpiderManProfileImageView` that displays a profile image of Spider-Man. We can use the overlay modifier to add a white border around this view:

```
struct SpidermanProfileImage: View {
    var body: some View {
        Image("spiderman_profil")
            .resizable()
            .scaledToFill()
            .frame(width: 200, height: 200)
            .clipShape(Circle())
            .shadow(radius: 5)
            .overlay {
                Circle().stroke(Color.white, lineWidth: 5)
            }
    }
}
```

In this example, we specify the frame size once for both the clipped circle shape and the overlay. They perfectly align with each other.



Just like with the background modifier, we have options for styling and alignment. Most of the time, you'll use the overlay modifier with content or alignment. Additionally, you can overlay a whole shape, such as a circle:

```
.overlay(alignment: .bottomLeading) {  
  Text("Spider-Man")  
}  
  
.overlay(Color.white.opacity(0.5), in: Circle())  
  
.overlay(Color.yellow, ignoresSafeAreaEdges: .top)
```

3.3 ZSTACK VS BACKGROUND/OVERLAY

In SwiftUI, there are different techniques available to achieve layering of views: background, overlay, and stack. Each of these techniques has its own purpose and considerations. Let's explore the differences between them and when to use each approach. To illustrate these concepts, let's use an example from a previous lesson.

```
Text("Cats are awesome")
    .padding()
    .background {
        Image("abstract-pool-water")
            .resizable()
            .scaledToFill()
    }
    .clipped()
```



The main view is the text view saying "Cats are awesome". In its background, we have an image view that shows a water pool. The size of the view is defined by the text itself and the padding.

The **background, on the other hand, doesn't influence the size of the view** but uses the size of the view it is attached to and passes it down to its child views. As a result, the image only receives the same size as the text.

Now, let's explore how the layout changes when we overlay the text and image **using a ZStack**. We'll need to adjust the image to fit and position the text on top of it.

```
ZStack {
    Image("abstract-pool-water")
        .resizable()
        .scaledToFit()
    Text("Cats are awesome")
        .padding()
}
```



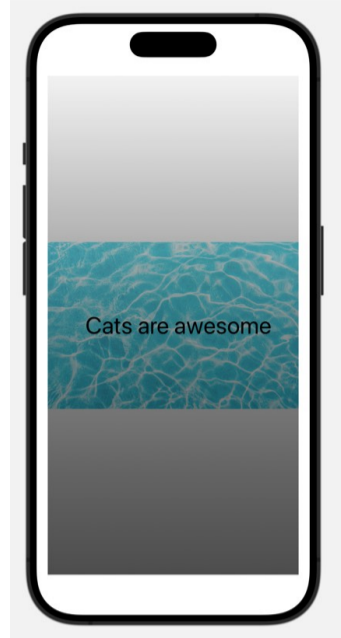
The ZStack considers the size of all its child views. In this case, the image is scaled to fit, making it as large as possible within the horizontal direction. **The ZStack adapts its size to accommodate the largest child view, which, in this case, is the image view.**

When choosing between background, overlay, and ZStack, the decision depends on how you want to control the size of your views. Lets add a color gradient to the above ZStack:

```
ZStack {
    Image("abstract-pool-water")
        .resizable()
        .scaledToFit()

    LinearGradient(colors: [Color(white: 0.9, opacity: 0.5),
                            Color(white: 0, opacity: 0.7)],
                 startPoint: .top,
                 endPoint: .bottom)

    Text("Cats are awesome")
        .font(.largeTitle)
        .padding(.leading)
}
.padding()
```

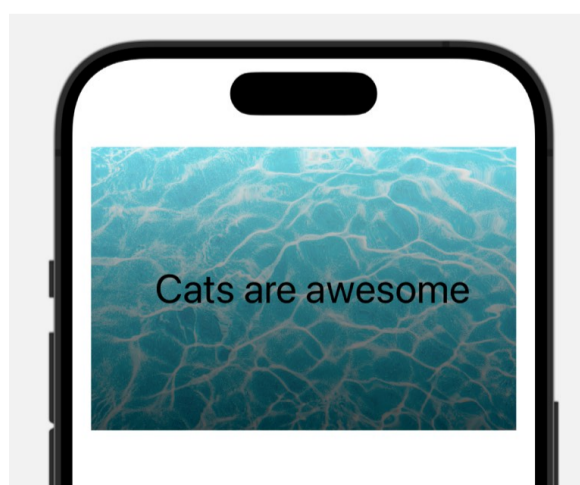


However, you may notice that the linear gradient takes up a lot of space and tries to be as big as possible. Consequently, the ZStack adjusts its size to accommodate the largest child view, which, in this case, is the linear gradient.

To restrict the size of the gradient to match the image, using an **overlay modifier is a better solution**. By moving the gradient inside the overlay, its size no longer influences the layout, and the image and gradient can have the same size.

```
ZStack {
    Image("abstract-pool-water")
        .resizable()
        .scaledToFit()
        .overlay {
            LinearGradient(colors: [Color(white: 0.9, opacity: 0.5),
                                    Color(white: 0, opacity: 0.7)],
                         startPoint: .top,
                         endPoint: .bottom)
        }

    Text("Cats are awesome")
        .font(.largeTitle)
        .padding(.leading)
}
.padding()
```



Another solution is to use the `fixedSize` modifier to the `ZStack` and restrict the size in the vertical direction. This way the gradient would not expand more than the image size:

```
ZStack {
    Image(...)
    LinearGradient(...)
    Text(...)
}
.fixedSize(horizontal: false, vertical: true)
```

In some cases, using an `overlay` modifier instead of a `ZStack` makes more sense, especially when you want to ensure that the layout is determined by a specific view's size. The `background` and `overlay` modifiers allow you to align or resize views relative to other views without affecting the overall layout.

To summarize, when layering views in SwiftUI, consider whether you want views to be properly sized and take up the space they need or if they are secondary and should align or resize with other views. Views that need to be properly sized can be placed in the `background` or `overlay`, while views that should align or resize with other views can be placed in a `ZStack`.

3.4 COLOR VIEW

Colors play a crucial role in creating visually appealing and dynamic user interfaces, and SwiftUI makes it incredibly easy to work with them. Let's start by understanding a fundamental concept: **colors are views themselves**. This means that we can treat colors just like any other view and use them within our view hierarchies. For example, we can add a blue color to a `VStack` as a view, and **it will expand to occupy as much space as possible**. We can also use other colors like cyan and indigo in a similar manner.

To restrict the expansion of colors, we can **apply a frame modifier to set a specific height or width**. For instance, we can limit the height of our color views to 100 by adding a frame modifier with a height of 100.

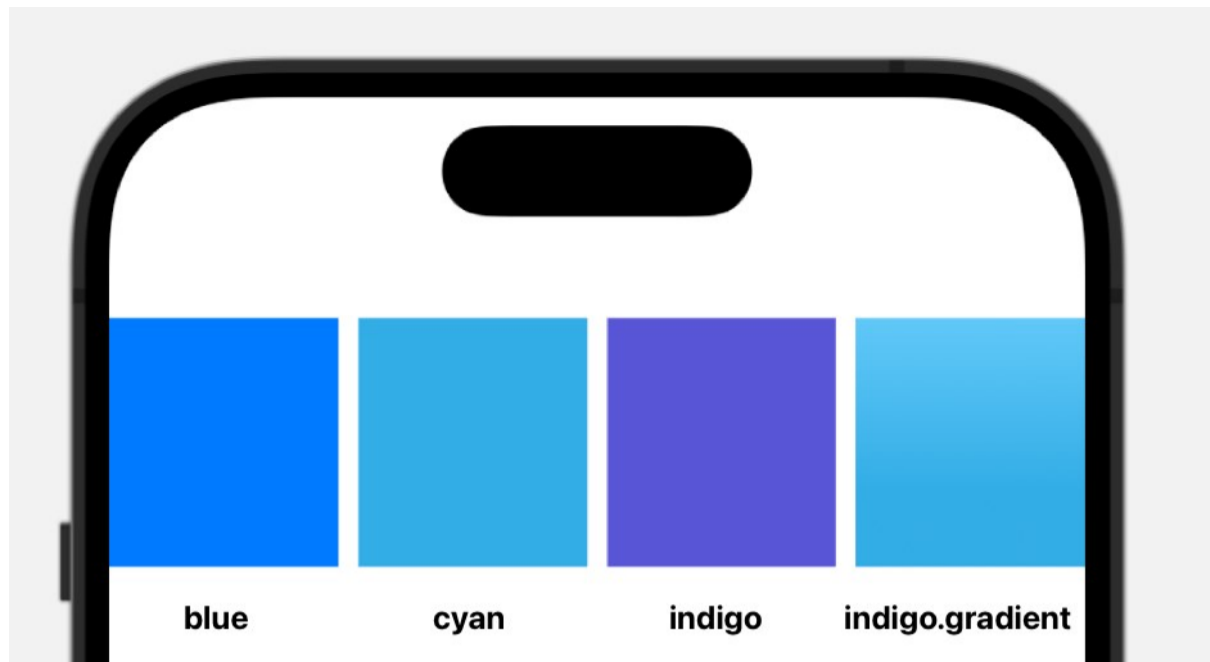
```
Color.cyan
    .frame(height: 100)
```

It's worth noting that colors conform to the `ShapeStyle` protocol, allowing us to use them as backgrounds with the `background` modifier. However, if we use a color gradient like

```
Color.cyan.gradient
```

it becomes an `AnyGradient` and is not considered a view. If you want to use a gradient as a view, use it to fill a shape like:

```
Rectangle().fill(Color.cyan.gradient)
```



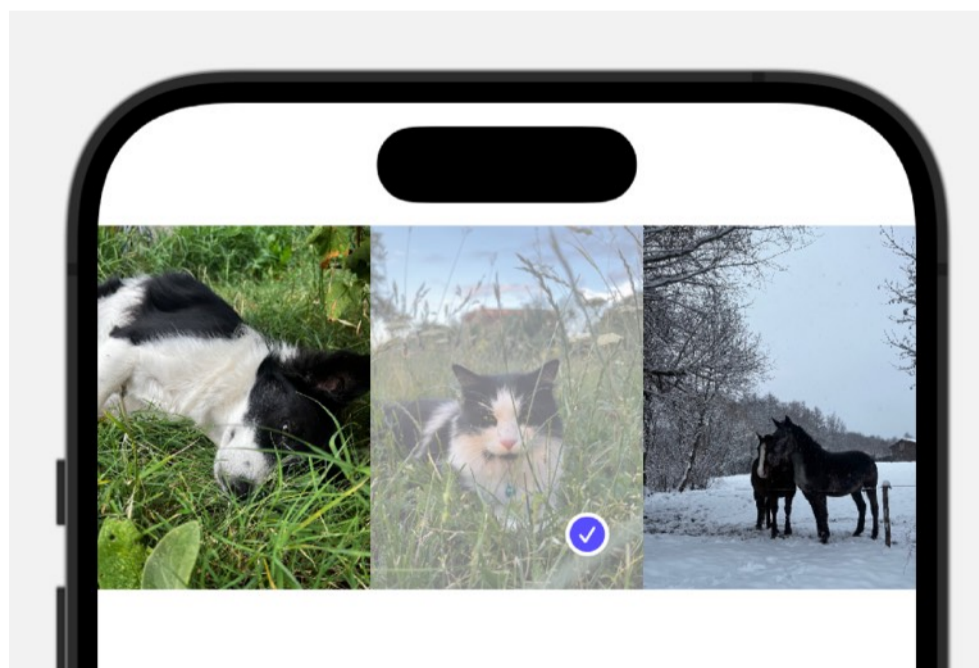
Another use case for colors is filling shapes with a filled shape style. For example, we can use a gradient as a shape style within a background to create visually appealing effects.

In addition to using predefined colors, SwiftUI also provides various options for creating custom colors. We can generate colors using hue, saturation, brightness, red, green, and blue values, or even create semi-transparent colors with a specific opacity. These custom colors offer great flexibility in designing our interfaces.

```
Color(hue: 0.7, saturation: 1, brightness: 1) // bright blue
Color(hue: 1, saturation: 1, brightness: 0.9, opacity: 0.5) // opace pink
Color(red: 1, green: 0, blue: 0) // red
Color(white: 0, opacity: 0.5) // opace gray
```

Example: Image Selection Screen

To demonstrate the usage of colors, let's consider an example. Imagine we have a board displaying multiple images, and we want to highlight the selected images by adding a semi-transparent overlay.



We can achieve this by creating a reusable subview, which takes an image name and a boolean value indicating whether it is selected or not. By leveraging the overlay modifier, we can apply the semi-transparent color to the selected images.

```
struct ImageSelectionView: View {
    let imageName: String
    let isSelected: Bool
    var body: some View {
        ResizableImageView(imageName: imageName)
            .overlay(alignment: .bottomTrailing) {
                if isSelected {
                    ZStack(alignment: .bottomTrailing) {
                        Color(white: 1, opacity: 0.5)

                        Image(systemName: "checkmark.circle.fill")
                            .foregroundColor(.accentColor)
                            .padding(1)
                            .background(Color.white, in: Circle())
                            .padding()
                    }
                }
            }
    }
}
```

To further enhance the selected images, we can add a checkmark icon to indicate their selection status. By using the overlay modifier with a **bottomTrailing** alignment, we can position the checkmark icon precisely where we want it. Additionally, we can add a circle shape behind the image to create a visual distinction.

```
HStack(spacing: 0) {
    ImageSelectionView(imageName: "dog_1",
                      isSelected: false)
    ImageSelectionView(imageName: "cat_1",
                      isSelected: true)
    ImageSelectionView(imageName: "horse_1",
                      isSelected: false)
}
```

By combining overlays, backgrounds, and alignments, we can create visually appealing and interactive user interfaces. Experimenting with different variations of these techniques will allow you to fine-tune the appearance of your views.

Remember, colors can significantly impact the size and layout of your views. Using overlays and backgrounds judiciously will help maintain the desired visual balance without compromising the overall design.

3.5 GRADIENTS

Gradients like colors are 'greedy' views and will expand to the available space. SwiftUI provides different types of gradients such as linear, radial, angular, and elliptical gradients. Here are some examples:


```

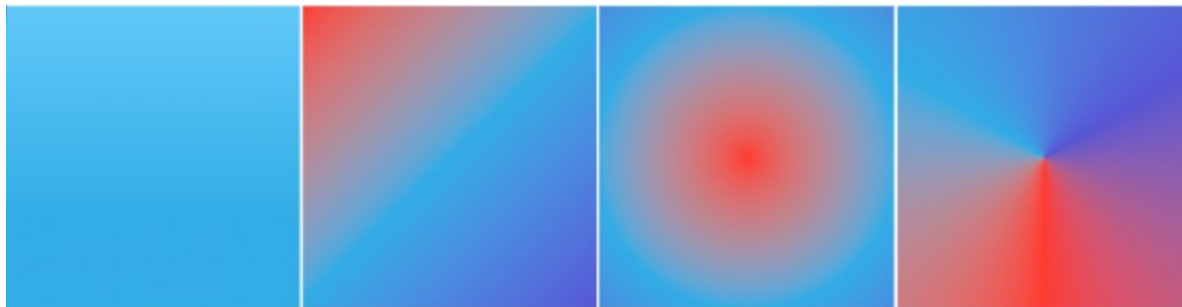
HStack(spacing: 1) {
  Rectangle().fill(Color.cyan.gradient)

  LinearGradient(colors: [Color.red, Color.cyan, Color.indigo],
    startPoint: .topLeading,
    endPoint: .bottomTrailing)

  RadialGradient(colors: [Color.red, Color.cyan, Color.indigo],
    center: .center,
    startRadius: 0,
    endRadius: 100)

  AngularGradient(colors: [Color.red, Color.cyan, Color.indigo, Color.red],
    center: .center, angle: .degrees(90))
}
.frame(height: 100)

```



Example: Making Text more Readable

A common use case is to add text on top of an image. Oftentimes this makes the text very difficult to read.

```

ZStack(alignment: .bottomLeading) {
  Image("cat_4")
    .resizable()
    .scaledToFit()

  Text("Cats are awesome")
    .font(.largeTitle)
    .foregroundStyle(.white)
    .padding(.leading)
}

```

You can use a gradient that is placed behind the text to increase the contrast. In the below example, I restricted the size of the gradient to a height of 100 points:

```

ZStack(alignment: .bottomLeading) {
  Image(...)

  LinearGradient(colors: [Color(white: 0, opacity: 0),
    Color(white: 0, opacity: 0.5)],
    startPoint: .top,
    endPoint: .bottom)
    .frame(maxHeight: 100)

  Text(...)
}

```

This is especially useful if you have high-contrast images like the below winter image:



with opace black gradient

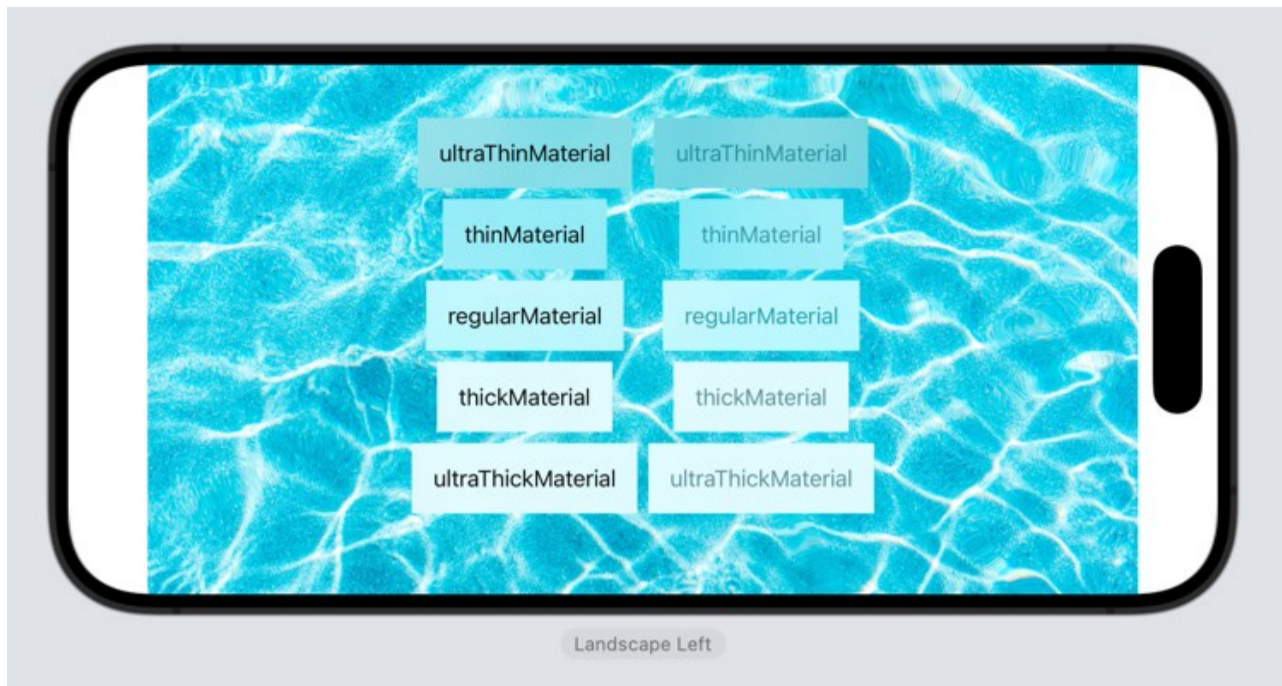
3.6 MATERIALS

Materials in SwiftU can be used to enhance the legibility, readability, and contrast of your views. Materials provide an alternative approach to using colored backgrounds, allowing you to achieve a similar effect with ease.

To better visualize the effect of materials, let's use a `ZStack` as our example and add an image with high contrast or distinct features, such as a water pool. I am placing text above with a ultra-thin material:

```
ZStack {  
    Image("abstract-pool-water")  
        .resizable()  
        .scaledToFill()  
  
    Text("ultraThinMaterial")  
        .padding()  
        .background(.ultraThinMaterial)  
}
```

By using materials, you can achieve an iced glass effect where the details of the background shimmer through. We can then apply different materials to the background to see their impact. There are five materials available: thin, regular, ultra thin, thick and ultra thick.



From top to bottom, the materials range from the most transparent to the thickest. You can think of them as different glass sheets with varying degrees of opacity. The choice of material depends on the desired contrast and the foreground colors of your text. For example, a thicker material may provide better contrast for certain text colors.

It's worth noting that materials also work seamlessly in dark mode. When switching to dark mode, the materials adapt automatically, providing a darker appearance. This can be particularly useful for maintaining legibility across different color schemes. You can even use different images or add darker sheets or gradients to achieve the desired effect in dark mode.

```

struct MaterialExampleView: View {
    @Environment(\.colorScheme) var colorScheme
    var body: some View {
        ZStack {
            Image("abstract-pool-water")
                .resizable()
                .scaledToFill()

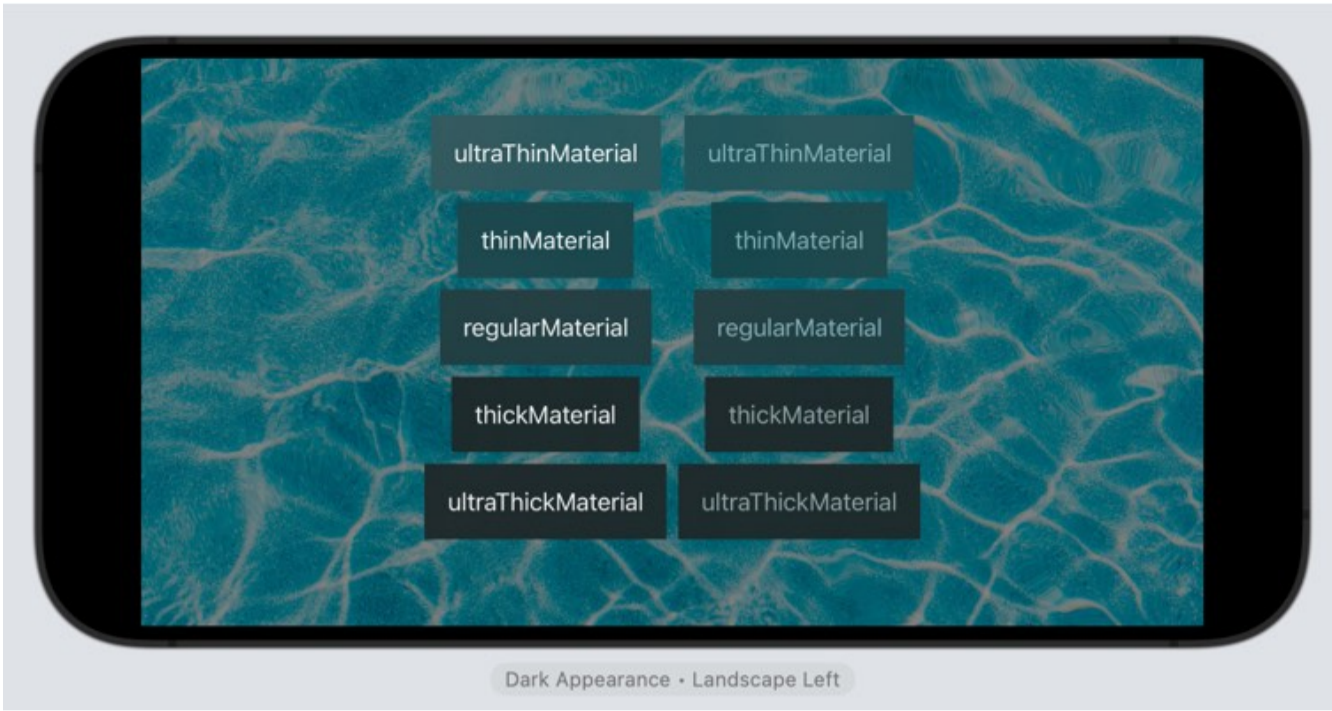
            if colorScheme == .dark {
                Color.black.opacity(0.5)
            }

            HStack {
                Text(...)

                Text(...)
                    .foregroundColor(.secondary)
            }
        }
    }
}

```

Remember, when presenting information, always ensure that the contrast between the text and the background is sufficient for users to read comfortably. This consideration is especially important for individuals with visual impairments. Materials, along with other techniques like gradients and semi-transparent colors, can greatly enhance the readability of your views.



4. POSITIONING VIEWS

4.1 HOW TO POSITION VIEWS

In this section, we will explore various techniques for positioning views within your layout in SwiftUI. By using different layout containers and view modifiers, we can easily achieve the desired positioning effects.

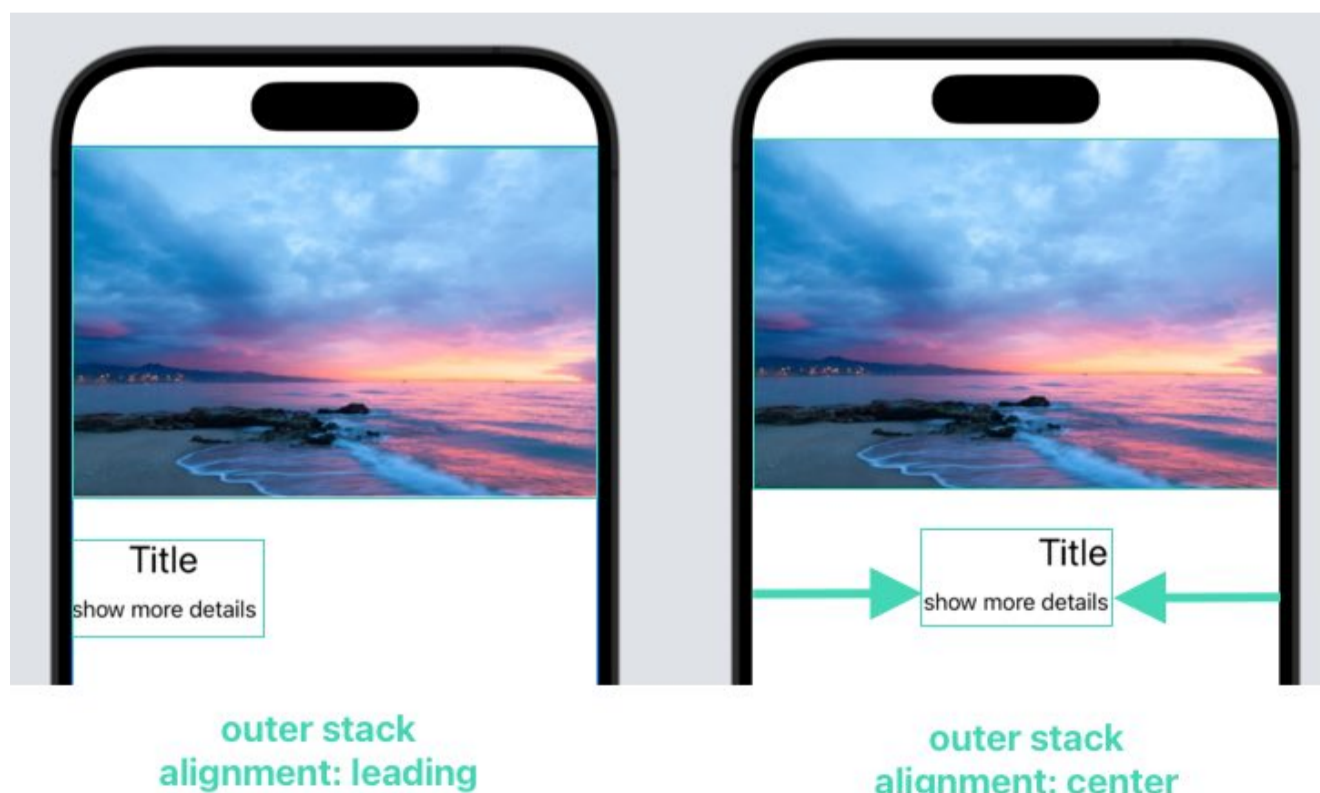
Using Primitive Layout Containers

SwiftUI provides us with primitive layout containers like VStack, HStack, and ZStack, which are essential tools for positioning views. For example, when using a VStack, we can utilize the alignment and spacing properties to control the positioning of the views within the container.

```
VStack(alignment: .leading,  
      spacing: 10) {  
    Text("Title")  
      .font(.title)  
  
    Text("show more details")  
}
```



The alignment properties come into play when we have multiple views that need to be aligned or **positioned relative to each other**. If you have multiple stacks you can the alignment becomes for complex:



In this example I am using 2 VStacks that are nested. Therefore I can use 2 alignment properties. Depending on how I want to align the image to the VStack with the texts to each other or how to align the 2 texts to each other. This is the code that produces the result above on the right:

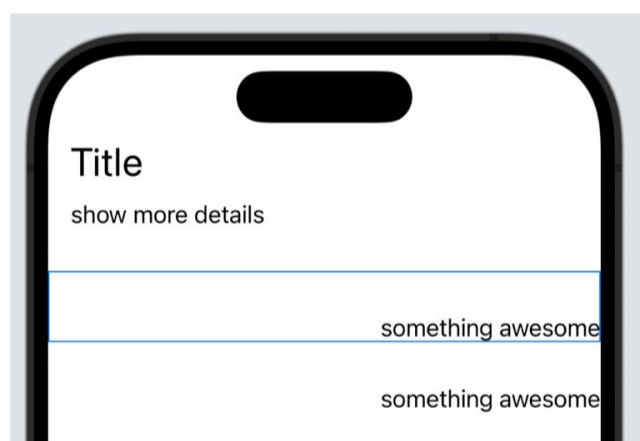
```
VStack(alignment: .center, spacing: 30) {  
  ResizableImageView(imageName: "beach")  
  
  VStack(alignment: .trailing,  
    spacing: 10) {  
    Text("Title")  
      .font(.title)  
    Text("show more details")  
  }  
}
```

Moving Views with Spacers

To add spacing between views or align them to specific positions, we can utilize spacers within the layout containers. In the following example, the spacer takes up the space on the leading side of the text view, which in turn is pushed to the trailing edge:

```
HStack {  
  Spacer()  
  Text("something awesome")  
}  
.padding(.top, 30)
```

Spacer expands to fill available space and pushes views to the edges.



If you place the Spacer in a VStack, it will "push" the other child views in the stack in the vertical direction. In the following example the image is pushed to the top of the screen:

```
VStack {  
  ResizableImageView(imageName: "beach")  
  Spacer()  
}
```



Fixed and Flexible Frames

You can use frames around views. The frame modifier has an argument for alignment, which allows to align the view inside the frame. The following example gives the same result as the HStack + Spacer example from above:

```
Text("something awesome")  
  .frame(maxWidth: .infinity,  
         alignment: .trailing)
```

Similarly, I can use a flexible frame to move an image to the top of the screen:

```
ResizableImageView(imageName: "beach")  
  .frame(maxWidth: .infinity,  
         alignment: .top)
```

Fine-Tuning with Padding

Padding is a useful tool for adjusting the position of individual views. By applying padding to a view, we can change its position within the layout.

```
struct SuperHeroHeaderView: View {  
  let superhero = SuperHero.example  
  var body: some View {  
    VStack {  
      ResizableImageView(imageName: superhero.imageName)  
        .padding([.leading, .bottom])  
        .background(Color.indigo.gradient)  
    }  
  }  
}
```

```

        VStack(alignment: .leading) {
            Text(superhero.name)
                .font(.largeTitle)
                .bold()

            Text(superhero.biography)
        }
        .padding()
    Spacer()
}
}
}
}
}

```

For example, I added padding around the text to move it away from the edges. Then I also added padding to the image but only to the leading and bottom edges:



By utilizing the various layout containers, spacers, padding, and advanced positioning techniques like alignment guides, grids, and view modifiers, we can easily position views within our SwiftUI layouts. These tools provide us with the flexibility and control needed to create visually appealing and well-structured user interfaces.

4.2 ALIGNMENT GUIDES

In this section, we will explore alignment guides in SwiftUI, which allow you to fine-tune the alignment of views or override the default alignment system.

```
HStack(alignment: .lastTextBaseline) {  
    Text("Delicious")  
    Image("avocado_large")  
    Text("Avocado Toast")  
        .font(.largeTitle)  
}
```



I used a `lastTextBaseline` alignment but I want to deviate from it. To override the default alignment system, we need to use the alignment guide modifier. First, we specify which alignment guide we want to modify, which in this case is the last text baseline. Then, we define the dimension of the alignment guide:

```
HStack(alignment: .lastTextBaseline) {  
    Text("Delicious")  
  
    Image("avocado_large")  
        .alignmentGuide(.lastTextBaseline, computeValue: { dimension in  
            dimension.height * 0.8  
        })  
  
    Text("Avocado Toast")  
        .font(.largeTitle)  
}
```



By using this alignment guide, the last text baseline of “Delicious” and “Avocado Toast” will **align with the specified height**, which is 80% through the avocado image.

If we prefer to use points instead of percentages, we can specify a value such as 20 points to move the image up or down.

```
Image("avocado_large")  
    .alignmentGuide(.lastTextBaseline, computeValue: { dimension in  
        dimension.height - 20  
    })
```

Additionally, we can use different alignment guides like center or top, depending on our layout requirements. Modifying the above example, I am aligning the 2 Text views text base line to the images top edge:

```
Image("avocado_large")
    .alignmentGuide(.lastTextBaseline, computeValue: { dimension in
        dimension[VerticalAlignment.top]
    })
```



VStack with Alignment Guides

Now, let's explore alignment guides within a VStack. We'll create another VStack with a leading alignment and add two text views:

```
VStack(alignment: .leading) {
    Text("Moved")
        .alignmentGuide(.leading, computeValue: { dimension in
            dimension.width * 0.75
        })
    Text("Delicious")
    Image("avocado_large")
        .alignmentGuide(.leading, computeValue: { dimension in
            dimension[HorizontalAlignment.center]
        })
}
```



In this VStack, we can observe how the alignment guides work with the leading alignment. By using the alignment guide, we can align the views based on a specific dimension, such as width times 0.75.

You can modify individual views within the stack using alignment guide modifiers. For example, you can apply the alignment guide to the image and align it to the leading edge, or you can leave the text view as the default alignment. The alignment guides allow for precise control over the positioning of views within a stack.

ZStack with Alignment Guides

Lastly, we'll explore alignment guides within a ZStack. In this example, we'll add a resizable image of a spider as the background. We'll also include a profile image of a superhero in the foreground. You can see the results for the default **bottomLeading** alignment on the left and for the custom on the right:



By using alignment guides, we can align the center of the profile image to the bottom edge of the spider image. To achieve this alignment, we need to override the bottom alignment guide and set it to `dimensions.height` multiplied by 0.5.

```
ZStack(alignment: .bottomLeading) {  
    ResizableImageView(imageName: "spider")  
    SpidermanProfileImage()  
        .alignmentGuide(.bottom, computeValue: { dimension in  
            dimension.height * 0.5  
        })  
        .padding(.leading, 10)  
}
```

Alignment guides are incredibly useful when fine-tuning the alignment of views within the same stack. In the next section, we will explore how to align views that are in different stacks and how to create custom alignment guides.

4.3 CUSTOM ALIGNMENT GUIDES

In this section, we will explore how to align views to each other that are contained in different containers using custom alignment guides. By creating our own alignment guides, we can achieve precise **alignment across multiple stacks**. Let's dive into some examples to better understand this concept.

Example 1: Aligning Horizontal Stripes

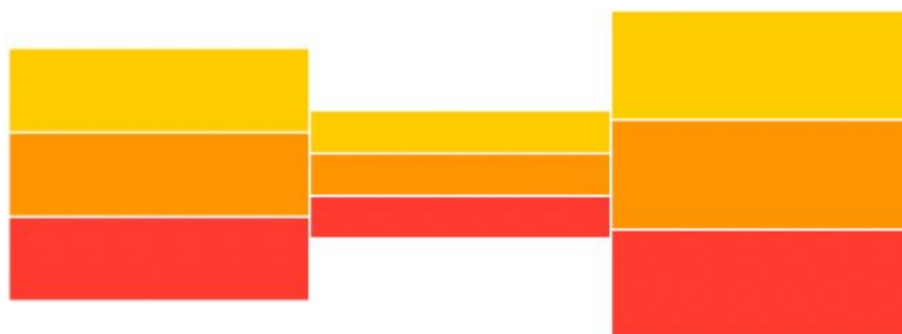
To demonstrate custom alignment guides, let's start with a basic example of horizontal stripes. We will create a `VStack` with three colors: yellow, orange, and red. Then, we will use an `HStack` to display these stripes. Initially, the stripes will fill up the entire space, but we want to align them to the first third of the views inside the `HStack`.

```
struct HorizontalStripesView: View {
  var body: some View {
    VStack(spacing: 1) {
      Color.yellow
      Color.orange
      Color.red
    }
  }
}

struct CustomAlignmentGuideExampleView: View {
  var body: some View {
    HStack(spacing: 1) {
      HorizontalStripesView()
        .frame(height: 100)

      HorizontalStripesView()
        .frame(height: 50)

      HorizontalStripesView()
        .frame(height: 130)
    }
  }
}
```



To achieve this, we need to write our own custom alignment guide. We declare a private struct called `FirstThird` conforming to `AlignmentID`. This alignment guide will align the views to the bottom of the yellow color. We then extend the existing `VerticalAlignment` with our custom alignment guide.

```

struct FirstThirdAlignment: AlignmentID {
    static func defaultValue(in context: ViewDimensions) -> CGFloat {
        context.height / 3
    }
}

extension VerticalAlignment {
    static let firstThird = VerticalAlignment(FirstThirdAlignment.self)
}

```

Finally, we can use our custom alignment guide in the HStack by setting the alignment to FirstThird. This will align the stripes to the first third, creating a visually pleasing layout.

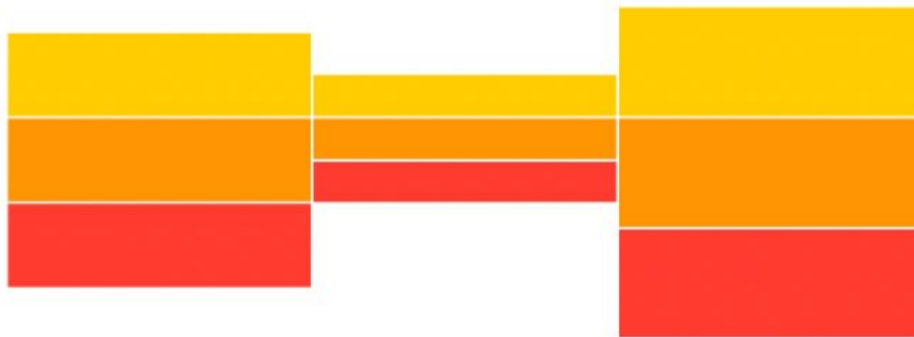
```

HStack(alignment: .firstThird, spacing: 1) {
    HorizontalStripesView()
        .frame(height: 100)

    HorizontalStripesView()
        .frame(height: 50)

    HorizontalStripesView()
        .frame(height: 130)
}

```



Example 2: Aligning Image and Title

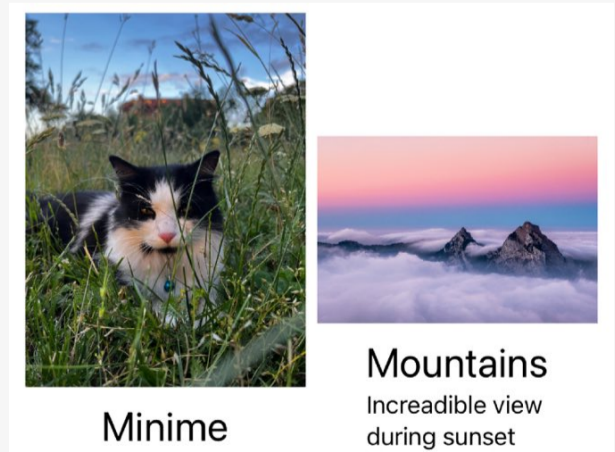
In this example, we have an HStack with two VStacks inside. Each VStack contains an image and a text.

```

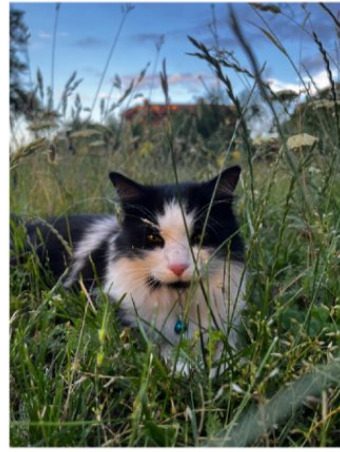
HStack {
    VStack {
        ResizableImageView(imageName: "cat_1")
        Text("Minime")
            .font(.title)
    }

    VStack {
        ResizableImageView(imageName: "mountain")
        Text("Mountains")
            .font(.title)
        Text("Incredible view during sunset")
    }
}

```



By default, the VStacks are aligned in the center, but we want to align them so that the bottom edge of the images is aligned to each other.



Minime



Mountains

Incredible view
during sunset

To achieve this, we need to create our own custom alignment guide again. We declare a private struct called `ImageTitleAlignment` conforming to `AlignmentID`. This alignment guide will align the views based on the dimensions of the images. We extend the existing `VerticalAlignment` with our custom alignment guide.

```
struct ImageTitleAlignment: AlignmentID {
    static func defaultValue(in context: ViewDimensions) -> CGFloat {
        context[.bottom]
    }
}

extension VerticalAlignment {
    static let imageTitleAlignment = VerticalAlignment(ImageTitleAlignment.self)
}
```

Using our custom alignment guide, we can align the images to each other by attaching the alignment guide to the appropriate views. This allows us to achieve precise alignment between the image and the title, creating a visually appealing layout.

```
HStack(alignment: .imageTitleAlignment) {
    VStack {
        ResizableImageView(imageName: "cat_1")
            .alignmentGuide(.imageTitleAlignment, computeValue: { dimension in
                dimension[.bottom]
            })
        Text("Minime")
            .font(.title)
    }

    VStack {
        ResizableImageView(imageName: "mountain")
            .alignmentGuide(.imageTitleAlignment, computeValue: { dimension in
                dimension[.bottom]
            })
        Text("Mountains")
            .font(.title)
        Text("Incredible view during sunset")
    }
}
```

Example 3: Creating a User Table

In this example, we want to create a user table with rows containing the user's name, street, zip code, and more details. Each row is represented by an HStack, and we want to align the title property of each row to the trailing edge of the table.



```
struct UserInformationView: View {
    var body: some View {
        VStack(alignment: .center, spacing: 20) {
            Image(systemName: "person.fill.questionmark")
                .font(.system(size: 40))
                .foregroundColor(.accent)

            PersonRowView(title: "Full Name:",
                          value: "John Doe")
            PersonRowView(title: "Street:",
                          value: "One Apple Park Way")
            PersonRowView(title: "Zip:",
                          value: "95014 Cupertino")
            PersonRowView(title: "Details:", value: "")
        }
    }
}
```

To achieve this, we once again create a custom alignment guide. We declare a private struct called CustomTrailing conforming to AlignmentID. This alignment guide aligns the views based on the trailing edge. We extend the existing HorizontalAlignment with our custom alignment guide.

```
struct CustomTrailing: AlignmentID {
    static func defaultValue(in context: ViewDimensions) -> CGFloat {
        context[.trailing]
    }
}

extension HorizontalAlignment {
    static let customTrailing: HorizontalAlignment =
        HorizontalAlignment(CustomTrailing.self)
}
```

By applying our custom alignment guide to the appropriate views, we can align the title properties of each row to the trailing edge of the table. This creates a clean and organized layout, resembling a table.

```
struct PersonRowView: View {
  let title: String
  let value: String

  var body: some View {
    HStack {
      Text(title)
      .alignmentGuide(.customTrailing, computeValue: { dimension in
        dimension[.trailing]
      })
      Text(value)
    }
  }
}
```

```
struct UserInformationView: View {
  var body: some View {
    VStack(alignment: .customTrailing, spacing: 20) {
      Image(...)

      PersonRowView(...)
      PersonRowView(...)
      PersonRowView(...)
      PersonRowView(...)
    }
  }
}
```

Example 4: Movie Detail View

In our final example, we create a movie detail view with a background image, title, star rating, action, and more details.

```
VStack {
  ZStack {
    ResizableImageView(imageName: "spider")

    HStack(spacing: 20) {
      Image("spiderman_profil")
      .resizable()
      .scaledToFill()
      .frame(maxWidth: 150, maxHeight: 200)
      .clipShape(RoundedRectangle(cornerRadius: 5))

      VStack(alignment: .leading, spacing: headerSpacing) {
        Text(title)
          .font(.title).bold()
          .foregroundColor(.white)

        VStack(alignment: .leading) {
          Text("★★★★")
          Text("Action")
            .font(.headline)
        }
      }
    }
  }
}
```



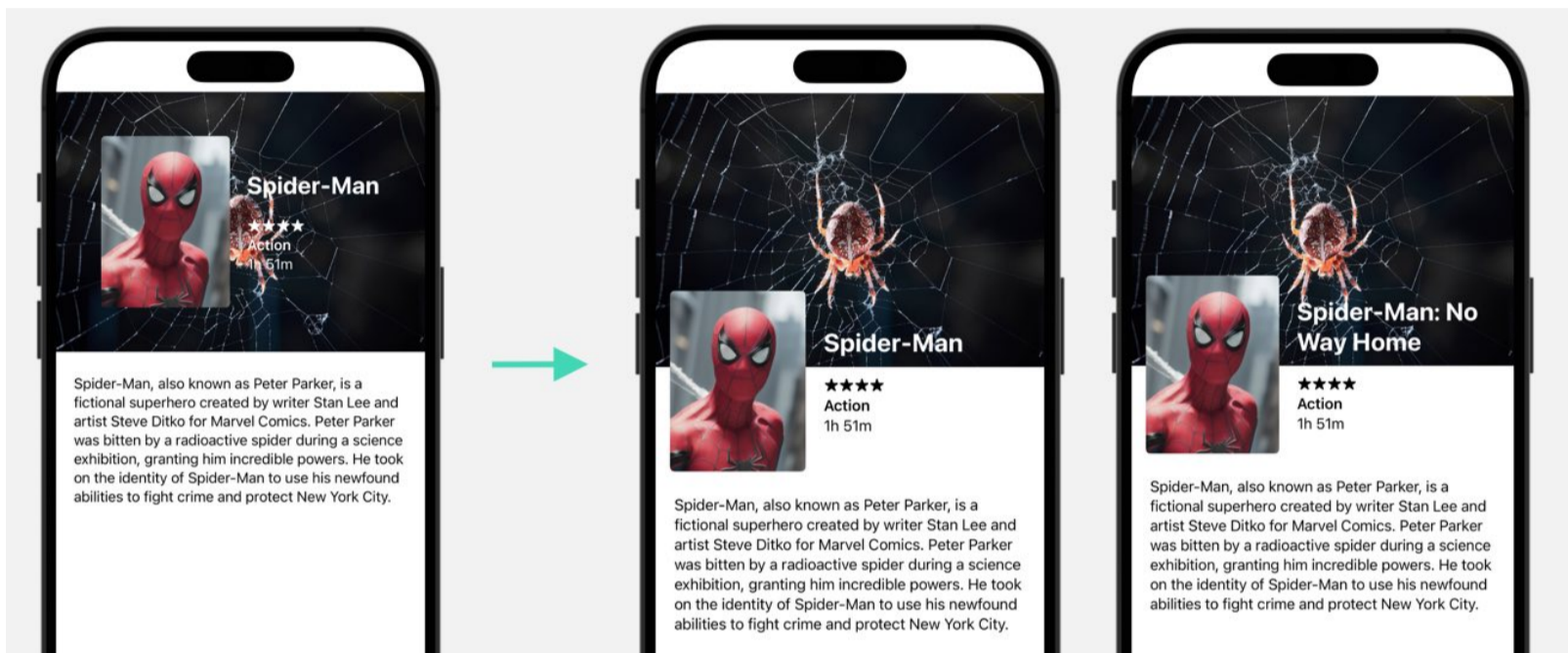
```

        Text("1h 51m")
      }
    }
  }
  .padding(.horizontal)
}

Text(superhero.biography)
  .padding()
}

```

We want to align the Spider-Man image to the bottom between the title and the star rating, regardless of the number of lines in the title.



To achieve this, we use a ZStack to overlay the views. We create a custom alignment guide called MovieAlignment that aligns the views based on the vertical center.

```

extension Alignment {
    static let movieAlignment = Alignment(horizontal: .leading,
                                          vertical: .imageTitleAlignment)
}

```

By attaching the alignment guide to the appropriate views, we can align the Spider-Man image precisely between the title and the star rating.

```

VStack {
    ZStack(alignment: .movieAlignment) {
        ResizableImageView(imageName: "spider")

        HStack(spacing: 20) {
            Image(...)
                .alignmentGuide(.imageTitleAlignment, computeValue: { dimension in
                    dimension[VerticalAlignment.bottom] + headerSpacing / 2
                })

            VStack(alignment: .leading, spacing: headerSpacing) {
                ...
            }
        }
    }
    .padding(.horizontal)
}

```

```

    }
    Text(..)
}

```

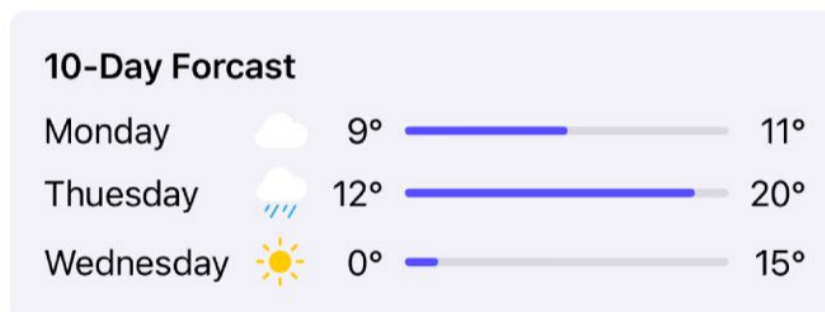
By utilizing custom alignment guides, we can achieve balanced and visually pleasing layouts. These alignment guides provide a **similar experience to Auto Layout in UIKit**, making it easier to align views within different stacks.

4.4 GRID VIEW

Grids are another powerful tool for positioning views in a table-like layout. They simplify the alignment of multiple columns and ensure a consistent and organized presentation of data. To illustrate the capabilities of Grid view, let's take a look at two real-world examples. The [user information table](#) from the previous section can also be easily done with Grid.

Weather Forecast View

To create a weather forecast view, we'll use a `GridView` with horizontal and vertical styles. This allows us to specify alignment, horizontal spacing, and vertical spacing options. We'll align the weekdays to the leading edge and the numbers to the trailing edge.



```

struct WeatherForecastView: View {
    let forecastData = WeatherData.example()
    var body: some View {
        GroupBox("10-Day Forecast") {
            Grid(alignment: .trailing,
                horizontalSpacing: 10,
                verticalSpacing: 5) {

                ForEach(forecastData) { data in
                    GridRow(alignment: .center) {
                        Text(data.weekday.name())
                            .gridColumnAlignment(.leading)

                        Image(systemName: data.iconName)
                            .renderingMode(.original)
                            .imageScale(.large)
                            .gridColumnAlignment(.center)

                        Text(String(data.lowTemperature) + "°")
                        ProgressView(value: data.progress)
                        Text(String(data.highTemperature) + "°")
                    }
                }
            }
        }
    }
}

```

```

    }
  }
}

```

In this code snippet, we iterate over the `weatherData` array and create a `GridRow` for each data point. I am aligning the weekdays to the leading edge using

```
.gridColumnAlignment(.leading)
```

the image icon is aligned in the center:

```
.gridColumnAlignment(.center)
```

and use the default alignment for the other views. The `gridColumnAlignment` modifier allows us to specify a different alignment for a specific column.

Expense Grid View

For the expense grid view, we'll again use a `GridView` with `ForEach` to iterate over the expense data. We'll create grid rows to display the expense details, including fixed expenses, variable expenses, and the total for each month.

	Fixed	Variable	All
Jan	100.00	130.00	230.00
Feb	100.00	10.00	110.00
Mar	100.00	0.00	100.00
Apr	100.00	1230.00	1330.00
May	100.00	990.00	1090.00
Jun	100.00	230.00	330.00
Jul	100.00	0.00	100.00
Aug	160.00	0.00	160.00
Sep	160.00	350.00	510.00
Oct	160.00	480.00	640.00
Nov	160.00	0.00	160.00
Dec	160.00	0.00	160.00
Total			\$13500.00

In this example, we use the `expenseData` array to populate the `GridView`. We create a `GridRow` for each data point and display the necessary expense details.

```

struct ExpenseGridView: View {
    let expenseData = ExpenseData.examples()
    let totalExpense: Double = 13500

    var body: some View {
        Grid(alignment: .trailing) {
            GridRow {
                Color.clear
                    .gridCellUnsizeAxes([.vertical, .horizontal])

                Text("Fixed")
                Text("Variable")
                Text("All")
                    .bold()
            }

            Divider()
                .gridCellUnsizeAxes([.vertical, .horizontal])

            ForEach(expenseData) { data in
                GridRow {
                    Text(month(for: data.month))
                    Text(String(format: "%.2f", data.fixedExpenses))
                    Text(String(format: "%.2f", data.variableExpenses))
                    Text(String(format: "%.2f", data.totalExpenses))
                }
            }

            Divider()
                .gridCellUnsizeAxes([.vertical, .horizontal])

            GridRow {
                Text("Total")
                    .bold()

                Color.clear
                    .gridCellUnsizeAxes([.vertical, .horizontal])
                    .gridCellColumns(2)

                Text("$" + String(format: "%.2f", totalExpense))
                    .bold()
            }
        }
    }

    let formatter = DateFormatter()

    func month(for number: Int) -> String {
        formatter.shortMonthSymbols[number - 1]
    }
}

```

For the header, I need to add an empty placeholder for the first cell. This can be done with

```

GridRow {
    Color.clear
        .gridCellUnsizeAxes([.vertical, .horizontal])

    Text(...)
}

```

For the summary row at the bottom, I need again an empty placeholder. By using the `gridCellColumnsCount` modifier, we can span views across multiple columns, allowing for more flexibility in the layout.

```
Color.clear
    .gridCellUnsizeAxes([.vertical, .horizontal])
    .gridCellColumns(2)
```

You have the possibility to align all cells in the same column with

```
.gridColumnAlignment(.trailing)
```

or to only align individual views with

```
.gridCellAnchor(.leading)
```

4.5 POSITION AND OFFSET MODIFIERS

In certain cases, we may need to use view modifiers like position and offset to achieve custom positioning effects. These modifiers are particularly useful for drawings or creating unique visual effects.

Offset Modifier

The offset modifier allows you to shift a view from its original position. You can specify the offset using either `CGSize` or `x` and `y` values.

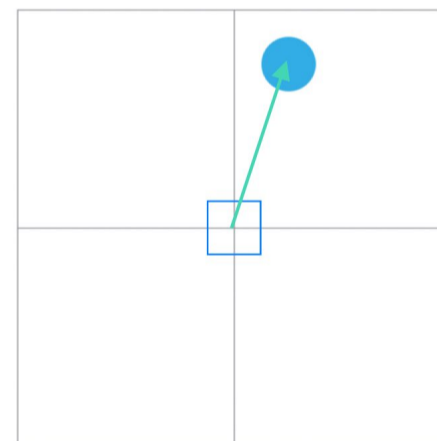
To better understand the concept, let's start with a simple exercise. We'll create a crosshair view to visualize the center of our layout. We can achieve this by using a `ZStack` and overlaying two lines, one horizontal and one vertical. By default, the alignment of the crosshair view will be centered.

```
struct CrossHairView: View {
    var body: some View {
        ZStack {
            Color.gray.frame(width: 1)
            Color.gray.frame(height: 1)
        }.border(Color.gray)
    }
}
```

Once we have the crosshair view set up, we can use it to visualize the original position of other views. For example, we can create a circle or use a text view with a small font size. By applying the offset modifier, we can shift the view in different directions, such as moving it down or to the right.

```
ZStack {
    CrossHairView()

    Circle()
        .fill(Color.cyan)
        .frame(width: 50, height: 50)
        .offset(x: 50, y: -150)
}
.frame(height: 400)
```



It's important to note that when using the offset modifier, we are only shifting the visible part of the view, not the entire layout. The layout system still reserves the necessary space for each view, which you can see from the blue border when the selectable preview is used.

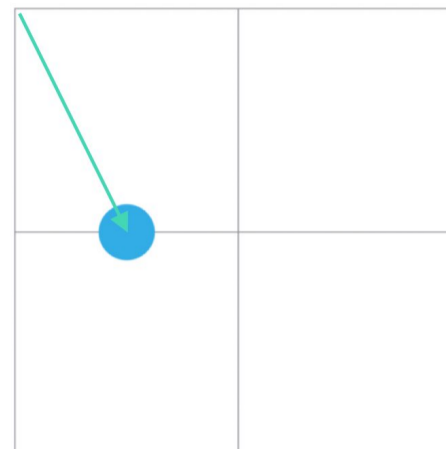
Position Modifier

The position modifier is another way to adjust the position of a view. However, instead of using an offset, we specify the exact position in the parent's coordinate system. This can be a bit confusing at first, but it becomes clearer with an example.

Let's say we have a ZStack as our parent view, and we want to position a subview within it. By using the position modifier, we can specify the exact coordinates in the parent's coordinate system. For instance, setting x: 100 and y: 200 will move the center of the view 100 points to the right and 200 points down from the top-left corner of the parent view.

```
ZStack {
    CrossHairView()

    Circle()
        .fill(Color.cyan)
        .frame(width: 50, height: 50)
        .position(x: 100, y: 200)
}
.frame(height: 400)
```



The position modifier becomes particularly useful when we need precise control over the position of a view, such as when creating drawings or animations. However, it's important to note that using the **position modifier can affect the layout of other views**, as it directly influences the positioning within the parent view.

Example: Animated Background Graphics

In this subsection, I'll walk through an example that demonstrates the use of offset and blur modifiers to create a visually appealing animated background. This example will showcase how these modifiers can be combined with animations to add a touch of elegance to your app.

```
struct FancyBackgroundView: View {
    @State private var animate: Bool = false
    var body: some View {
        ZStack {
            Color.backgroundColor2
            Circle()
                .fill(Color.accentColor)
                .frame(width: 200)
                .offset(x: animate ? 100 : -100,
                    y: animate ? -50 : 200)
                .blur(radius: animate ? 100 : 120)

            Circle()
                .fill(Color.cyan)
                .blur(radius: animate ? 100 : 120)
                .frame(width: 150)
                .offset(x: animate ? -200 : 100,
                    y: animate ? 100 : -200)
        }
        .edgesIgnoringSafeArea(.all)
        .onAppear {
            withAnimation(Animation.easeInOut(duration: 5).repeatForever()) {
                animate = true
            }
        }
    }
}
```

I'll start by using a ZStack to layer our background elements. You can add a solid color as the base background using the background modifier or the Color view directly.

Next, we can incorporate various shapes, such as circles, to add visual interest to the background. By **specifying the fill color, frame size, and position offset**, we can create multiple circles in different corners of the screen.

To bring the animation to life, we'll introduce a @State property called animate to toggle the animation on and off. We'll use the **withAnimation** block to specify the animation type, duration, and any additional animation options. By animating the position offset and blur radius values, we can create a mesmerizing effect as the circles move and blur in and out.

Remember, it's important to strike a balance with animations. While they can add a delightful touch to your app, overusing them can lead to a cluttered and distracting user experience. It's best to test and adjust the animation duration and effects to achieve the desired visual impact without overwhelming the user.



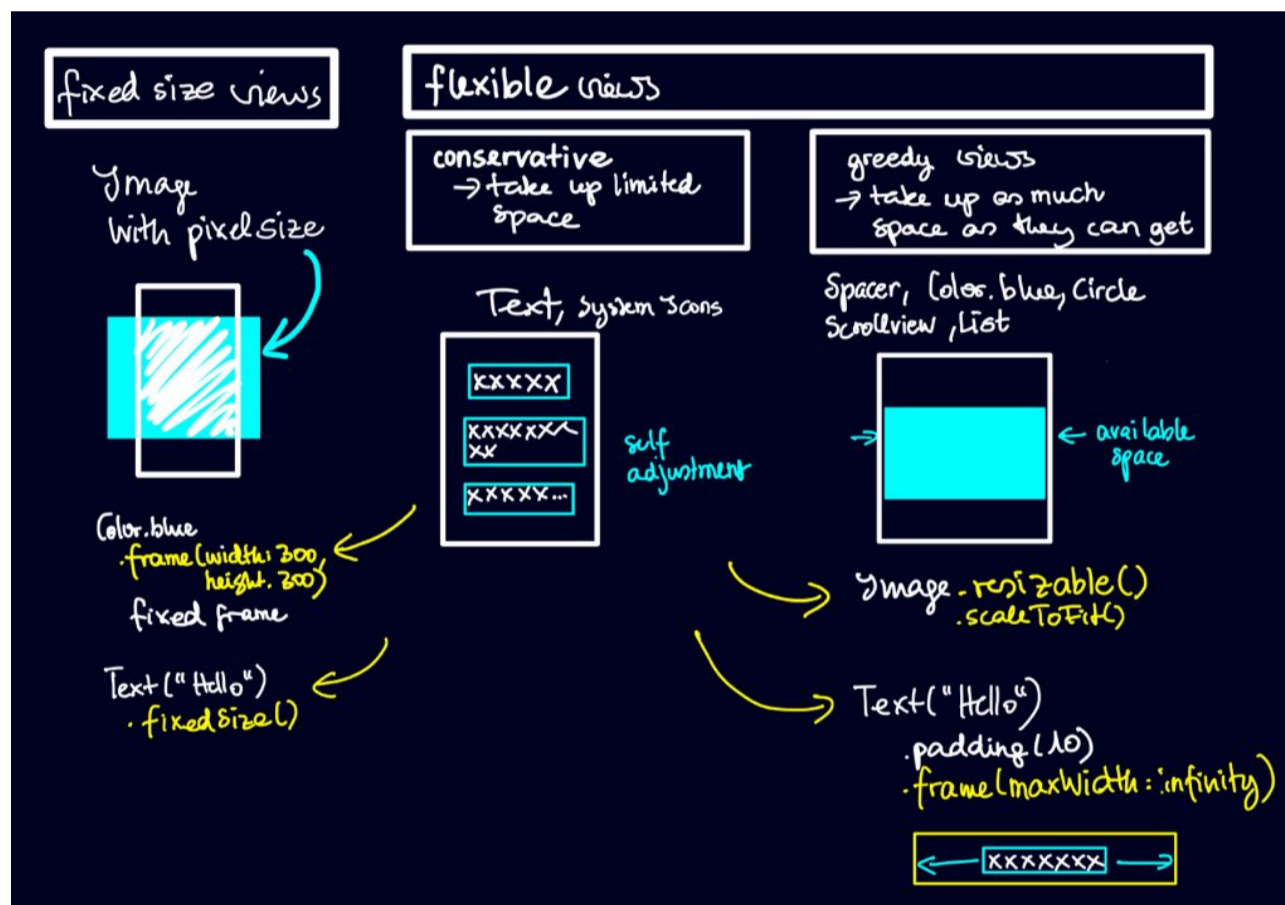
5. SIZING VIEWS

5.1 HOW THE LAYOUT SYSTEM SIZES AND POSITIONS VIEWS

In this section, we will explore how to size views in SwiftUI. Most of the time, the layout system handles the sizing of views internally. However, there are situations where you may want to customize the sizing behavior. To understand this, let's first delve into how the layout system works.

Different Types of Views

In SwiftUI, views can be categorized into three main types based on their sizing behavior:



- **Fixed Size Views:** These views have a strict, predetermined size. For example, if you place an *image* from your assets in a view, it will insist on its original size, even if it overflows the screen. These fixed-size views maintain their predetermined dimensions.
- **Flexible Views:** These views adjust their size based on available space. There are two subcategories within flexible views:
 - **Conservative Views:** These views take up only as much space as they need to fit their content. For example, a *Text view* will adjust its size based on the length of the text. If it doesn't fit on one line, it will wrap to multiple lines. If space is limited, it may truncate or compress itself to fit.
 - **Greedy Views:** These views strive to occupy as much space as possible. For example, a spacer in a horizontal stack will expand horizontally until it reaches the screen's limit. Examples of greedy views include *Spacer, Color, Shape, ScrollView, List, and TextEditor*.

Modifying View-Sizing Behavior

You can modify the sizing behavior of views using view modifiers. They can make views use a fixed size or make them use a more flexible sizing. Here are a few commonly used modifiers:

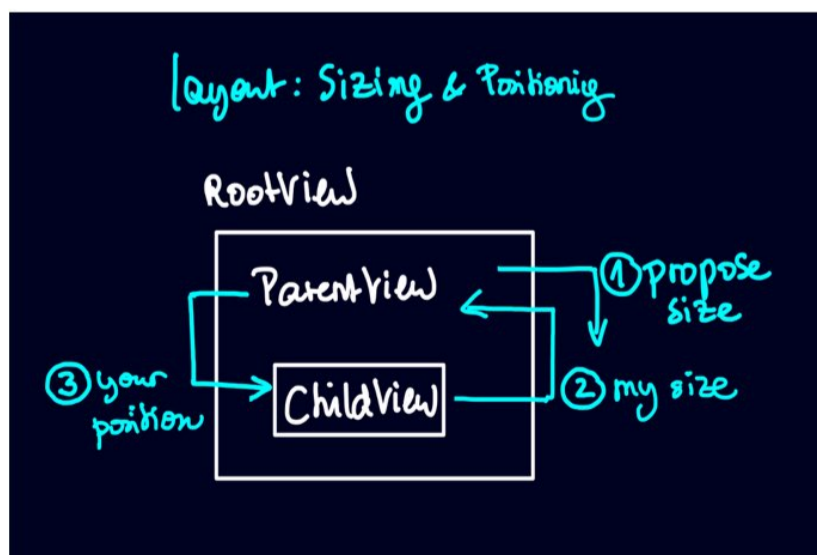
- **Resizable**: This modifier allows you to stretch or shrink an image to fit the available space. Thus, making the image a greedy view.
- **ScaleToFit**: When applied to an image, this modifier scales the image proportionally to fit within the available space.
- **Fixed Frame**: This modifier acts as a container specifying a fixed size for a view.
- **Flexible Frame**: You can set min, and maximum frame sizes which makes the frame container a greedy view that expands to fill the available space.
- **FixedSize**: This modifier ensures that a view maintains its intrinsic size. SwiftUI will respect the size and will use it for the layout. This can cause views to expand beyond the visible areas of the screen.

By applying these modifiers, you can adjust the sizing behavior of views to suit your specific needs.

Understanding the Layout Process

Container views, such as `VStack`, have the responsibility of laying out their subviews or children and distributing available space. This becomes particularly interesting when you have limited screen space and need to prioritize content.

In SwiftUI, the layout system is responsible for sizing and positioning views on the screen. It follows a three-step process to determine the size and position of each view within its parent container. Let's dive into each step to get a better understanding of how it works.



Step 1: Propose Size

The first step in the layout process is when the parent view proposes a size to its child views. The parent view examines the available space it has and suggests a size to its children. This proposed size is based on the parent view's own size and any constraints or modifiers applied to it.

Step 2: Subview Decides its Own Size

Once the child view receives the proposed size from its parent, it decides how much space it wants to occupy. This decision is influenced by the type of view and any modifiers applied to it. There are three main categories of views in terms of sizing behavior:

Fixed Size Views: These views have a specific size that they insist on occupying. They will not change for different proposed sizes.

Flexible Views: Flexible views will adjust their size based on the proposed size.

Conservative Views: These views take up only as much space as they need to fit their content. It will return the size to the container that it needs and not more

Greedy Views: Greedy views try to occupy as much space as possible. They will return to their parent all the space that they have been offered.

After the child view determines its desired size, it communicates this information back to the parent view. Thus the parent view adjusts its own size based on the sizes of its children and communicates it to its own parent.

Step 3: Positioning

The parent view then uses this size to position the child view within its own bounds. By default, SwiftUI centers the child view within the available space. However, you can modify the positioning behavior using view modifiers like alignment and padding.

It's important to note that the layout process follows a **top-to-bottom approach**, starting from the root view and cascading down to its children. Each parent view sizes and positions its children, who in turn do the same for their own children, if any.

Laying out Views that Compete for Space

When working with container views like HStack, the layout system determines how to size and distribute space among the child views. The order in which views are sized and positioned is crucial. *SwiftUI prioritizes the least flexible views first, allowing them to claim the space they need. Then, the remaining space is distributed among the more flexible views.* Let's consider a scenario where multiple color views compete for space within an HStack:

```
HStack(spacing: 0) {
    Color.red
    Color.orange
    Color.yellow
    Color.green
    Color.blue
}
.frame(width: 100, height: 100)
```



All 5 color views are considered “greedy” because they want to take up as much space as possible. When the layout system encounters this scenario, it follows a set of rules to distribute the available space among the competing views. In the case of the HStack, it **distributes the total width equally** between its child views. It first subtracts the spacing between the child views and the remaining width is divided by the number of views:

$$\text{Size of individual color view} = (\text{width} - (\text{spacing} * (\text{number of children} - 1))) / \text{number of children}$$

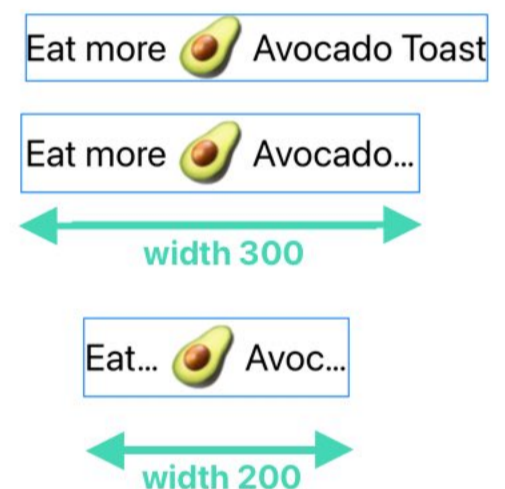
In the above example, we have:

$$\text{Size of individual color view} = (100 - (0 * 4)) / 5$$

As a result, each view within the HStack will have the full height available of 100 points and 1/3 of the width, allowing them to be equally spaced within the available space. This behavior ensures that all the views have a fair share of the available width.

Lets look at an example where an HStack has to **distribute the space between a fixed-size view and flexible views**:

```
HStack {
  Text("Eat more")
  Image("avocado_large")
  Text("Avocado Toast")
}
.font(.title)
.frame(width: 200, height: 60)
```



The **layout will first offer the space to the least flexible-sized view** which is the image. Then it will distribute the remaining space between the other two texts, which will be proposed half each. The “Avocado Text” is longer and will be truncated first.

You can also use the layoutPriority modifier to **prioritize one view over another when space is limited**. In the following example the “Avocado Toast” is offered the space over the other text:

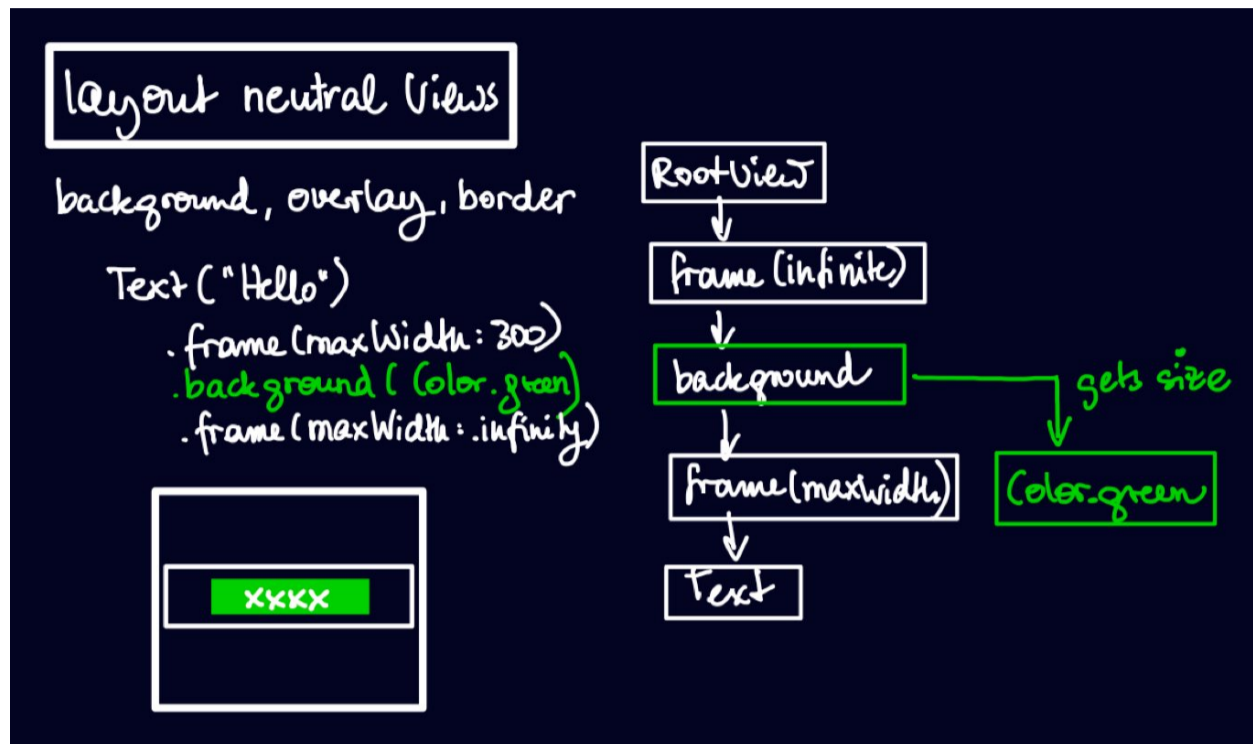
```
HStack {
  Text("Eat more")
  Image("avocado_large")
  Text("Avocado Toast")
  .layoutPriority(1)
}
.font(.title)
.frame(width: 300, height: 60)
```



In the upcoming sections, we will explore various view modifiers like layoutPriority that allow you to customize the sizing behavior of views. These modifiers, such as resizable, and frame, provide you with the flexibility to fine-tune the sizing and positioning of your views.

Layout Neutral Views

Some view modifiers, such as background, overlay, and border, are layout-neutral. These modifiers do not influence the layout behavior but instead, pass the proposed size to their children. They are useful for visualizing the occupied area of views.



Background has a secondary child view which is in this example the green color view. The background takes the view its primary child (the frame and text) wants and passes it to its secondary child. The green color gets the size proposed and because it is a greedy view, it takes it completely up. Therefore the green color in the background is the same exact size as the background's wrapped view.

Understanding the layout system and the different types of views, as well as how to modify their sizing behavior, is crucial for creating well-designed SwiftUI layouts. In the upcoming sections, we will explore these concepts in more detail and cover additional topics such as adaptive layout and custom layouts.

5.2 FIXED AND FLEXIBLE FRAMES

In this section, we will explore the various ways to specify the size of a view using the frame modifier in SwiftUI. The frame modifier allows us to control the dimensions and alignment of a view within its container.

Flexible Frames

One example of using the frame modifier is to create flexible frames. By setting the maximum width or height to infinity, we can allow the view to stretch and adapt to the available space. For example, to move a view to the trailing edge, we can use a flexible frame with a maximum width of infinity:

```
Text("Hello, World!")  
  .frame(maxWidth: .infinity)  
  .background(Color.yellow)
```



By default, the content within the frame is centered. We can change the alignment by using the frame modifier's alignment parameter. For example, to align the text to the trailing edge, we can use:

```
Text("Hello, World!")  
  .frame(maxWidth: .infinity, alignment: .trailing)  
  .background(Color.yellow)
```



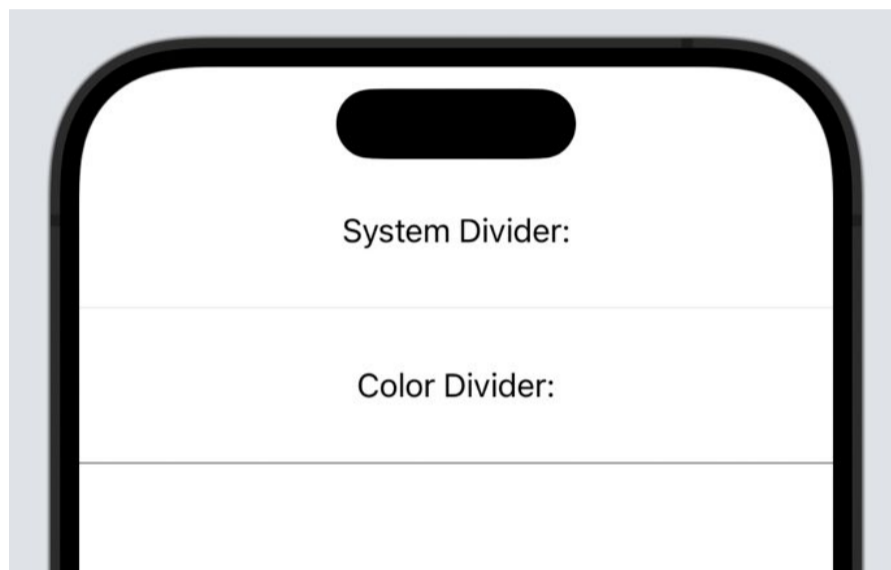
The frame modifier can expand vertically and horizontally. The alignment options are the same as for ZStack:

```
.frame(maxWidth: .infinity,  
      maxHeight: .infinity,  
      alignment: .bottomTrailing)
```

Fixed Frames

In addition to flexible frames, we can also set specific dimensions for a view using the frame modifier. This is useful when we want to control the size of images or drawings. For example, to create a custom divider, we can use the frame modifier with fixed width and height values:

```
Divider()  
Color.gray  
  .frame(height: 1)
```



Clipping and Masking

The frame modifier can also be used in combination with clipping and masking to achieve specific sizing effects. For example, we can use a mask of a circle view to create a circular image:

```
Image("spiderman_profil")  
  .resizable()  
  .frame(width: 100, height: 100)  
  .mask(Circle())
```



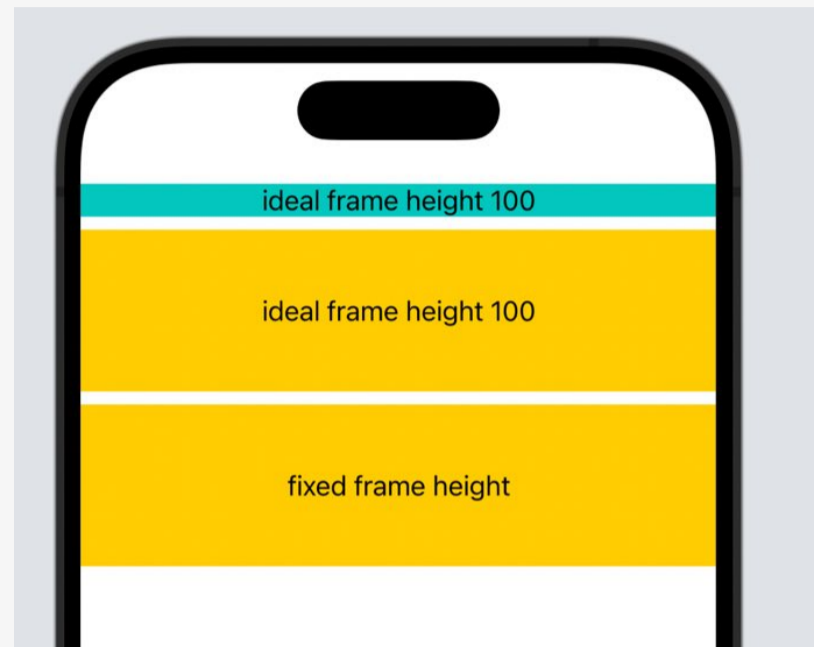
Ideal Frames

The frame modifier provides additional parameters such as, ideal width, and ideal height. These parameters allow us to define the ideal size of a view which is used as the intrinsic size of the view. For example, we can create a view with an ideal width of 100 and place it inside a ScrollView:

```
ScrollView {
  Text("ideal frame height 100")
    .frame(maxWidth: .infinity,
           maxHeight: .infinity)
    .background(Color.mint)

  Text("ideal frame height 100")
    .frame(minWidth: 0,
           idealWidth: 100,
           maxWidth: .infinity,
           minHeight: 0,
           idealHeight: 100,
           maxHeight: .infinity)
    .background(Color.yellow)

  Text("fixed frame height")
    .frame(height: 100)
    .frame(maxWidth: .infinity)
    .background(Color.yellow)
}
```

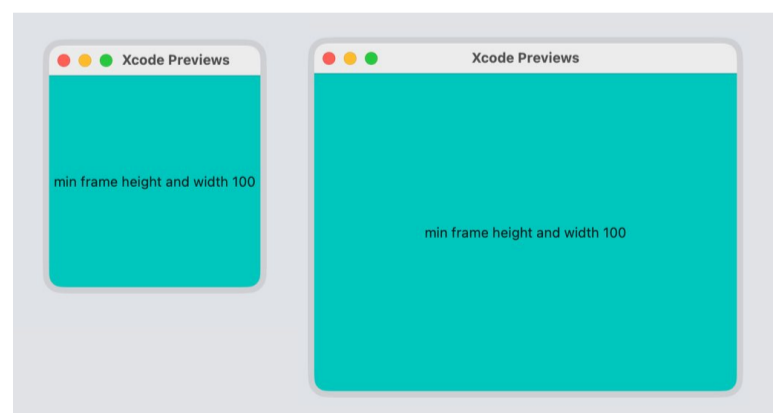


The ideal width/ height will be used if a fixedSize is applied. ScrollView will attach a fixed size to its children in the scroll direction. The frame's ideal height will be used. Similarly, you can force a height of 100 points with a fixed frame value.

Minimum Width and Height

In addition to maximum height and width, the frame modifier in SwiftUI also allows us to set minimum height and width values for a view. These parameters define the minimum size that a view should occupy within its container. For example, you might want to set a minimum size of a view on macOS, where the user can drag the window to make it smaller and larger. The minimum values make sure that the window will not shrink smaller than the values you have given:

```
Text("min frame height and width 100")
  .frame(minWidth: 200,
         minHeight: 200)
  .background(Color.mint)
```



Alignment In Multiple Frames

We can apply multiple frame modifiers to a view to achieve complex layouts. When using multiple frames, it's important to understand how the alignment is affected. The alignment of a view is determined by its immediate container. For example, if we have a VStack with multiple child views, the alignment of the VStack will affect the alignment of its child views.

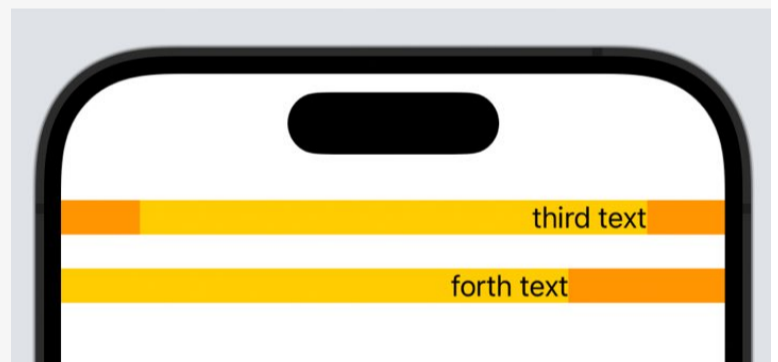
However, if we apply a frame modifier to a child view, that will take up the full width in the stack, the alignment of the text view will be handled solely by the frame.

```
VStack(alignment: .leading) {  
    Text("First Text")  
  
    Text("Second Text")  
        .frame(maxWidth: .infinity,  
               alignment: .trailing)  
        .background(Color.yellow)  
}
```



Each text view is aligned inside its immediate frame container. When one frame is nested inside another frame, the parent frame is responsible to align the child frame respectively:

```
VStack {  
    Text("third text")  
        .frame(maxWidth: 300,  
               alignment: .trailing)  
        .background(Color.yellow)  
        .frame(maxWidth: .infinity,  
               alignment: .center)  
        .background(Color.orange)  
  
    Text("forth text")  
        .frame(maxWidth: 300,  
               alignment: .trailing)  
        .background(Color.yellow)  
        .frame(maxWidth: .infinity,  
               alignment: .leading)  
        .background(Color.orange)  
}
```



The frame modifier in SwiftUI provides powerful tools for sizing and aligning views within their containers. Whether we need flexible frames that adapt to available space or fixed frames for precise control, the frame modifier allows us to achieve our desired layouts.

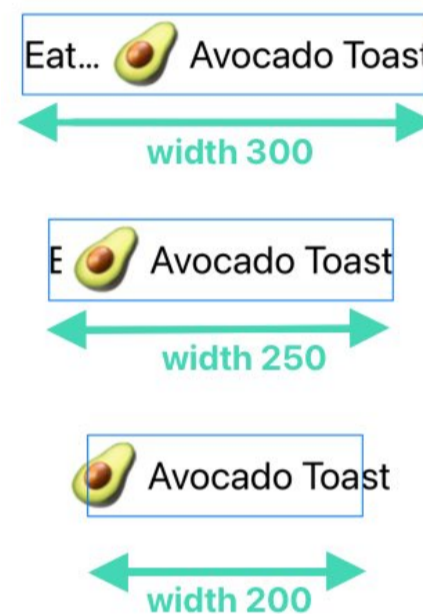
5.3 FIXEDSIZE

In this section, we will explore the concept of sizing views in SwiftUI. One important aspect of sizing views is the fixed size, which can be thought of as the intrinsic size that a view needs. Let's dive into some examples to understand this concept better.

Using the Intrinsic Size of Text Views

When working with views, it's crucial to consider their necessary size. When you apply the fixed size modifier you are telling the SwiftUI layout to use the intrinsic size of this view. In the following example, I enforce the intrinsic size of the "Avocado Toast" view:

```
HStack {
    Text("Eat more")
    Image("avocado_large")
    Text("Avocado Toast")
        .fixedSize()
}
.font(.title)
.frame(width: 250, height: 60)
```



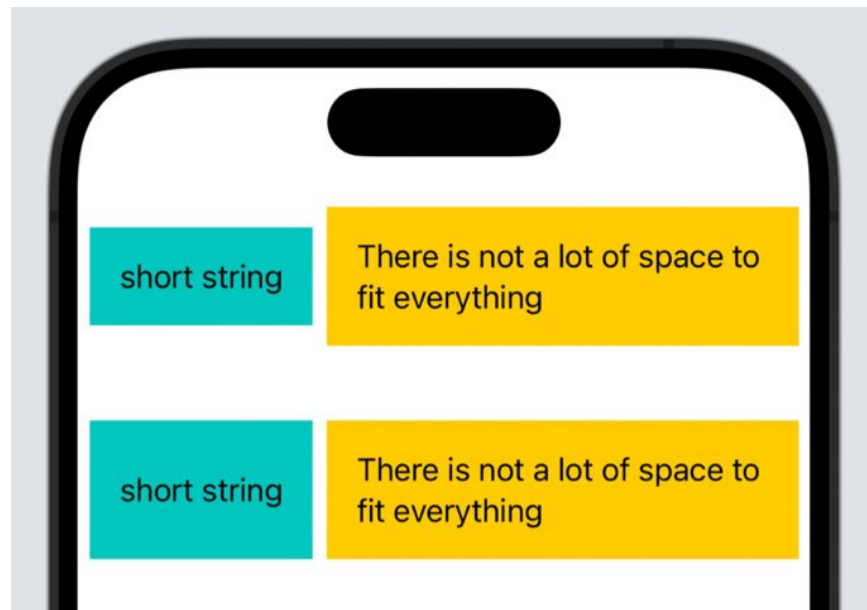
This means that the view will always display the full text, without being cut off. However, keep in mind that when using fixed size, the HStack needs to distribute the available space between its children. In this case, both the image and the AvocadoToastTextView have strict sizing, and the remaining space is allocated to the "Eat more!" text.

For a smaller space of the stack, the image and "Avocado Toast" texts are still using their intrinsic size, which can be larger than the size of the stack. Thus the views are overflowing the available area.

Example: Equal Size Views

In this example, we want to size two Text views within an HStack. One view contains a short string, while the other has a longer string. To ensure both views have equal height, we can apply the fixedSize modifier.

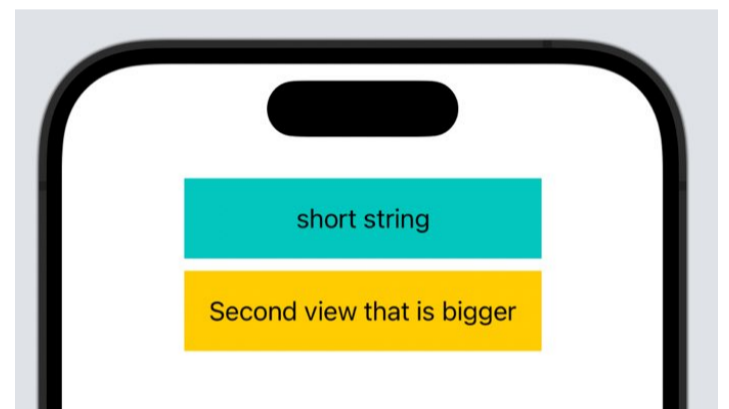
```
HStack {
    Text("short string")
        .padding()
        .frame(maxHeight: .infinity)
        .background(Color.mint)
    Text("There is not a lot of space to fit everything")
        .padding()
        .frame(maxHeight: .infinity)
        .background(Color.yellow)
}
.fixedSize(horizontal: false, vertical: true)
```



By using `fixedSize`, we restrict the vertical size of the `HStack` to the intrinsic height of the larger view. The flexible frame stretches dynamically in the vertical direction. This ensures that both views have the same height, even if the longer string doesn't fit entirely.

Similarly, you can do the same in the horizontal direction, which I used in an earlier section for [ControlGroup styling](#):

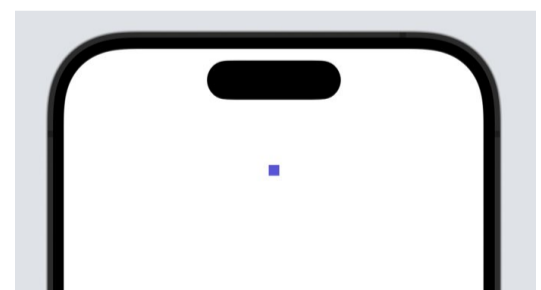
```
VStack {
  Text("short string")
    .padding()
    .frame(maxWidth: .infinity)
    .background(Color.mint)
  Text("Second view that is bigger")
    .padding()
    .frame(maxWidth: .infinity)
    .background(Color.yellow)
}
.fixedSize(horizontal: true, vertical: false)
```



Example: Undefined Intrinsic Size

Some views, like the `Color` view, don't have a clearly defined intrinsic size. In such cases, applying the `fixedSize` modifier can have unexpected results. A color view will become very small when you use a fixed size:

```
Color.indigo
  .fixedSize()
```



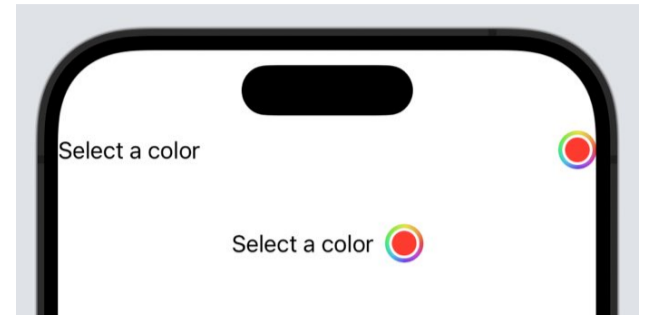
Similarly, you get really small views for resizable images, shapes, and spacers.

Example: Spacer and Control Views

Certain control views, like ColorPicker and DatePicker, also benefit from the `fixedSize` modifier in specific scenarios.

```
ColorPicker("Select a color",
           selection: $selectedColor)

ColorPicker("Select a color",
           selection: $selectedColor)
           .fixedSize()
```



These views use a spacer in an `HStack`. Applying the `fixedSize` modifier will shrink the `Spacer` and thus the stack in the horizontal direction. This can be useful in layouts where we want to eliminate unnecessary spacing. Similarly, control views like `ColorPicker` and `DatePicker` can have their spacing adjusted by applying `fixedSize`.

5.4 LAYOUT PRIORITY

In this section, we will dive deeper into the concept of layout priority. Let's take a look at an example to understand how it works. Imagine we have a simple layout with two text views and an image, all placed within a `VStack`. To demonstrate layout priority, we can add frames with a width of 300 to this view and apply borders around everything.

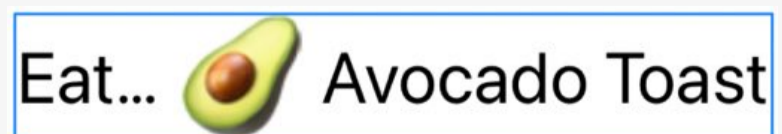
```
HStack {
  Text("Eat more")
  Image("avocado_large")
  Text("Avocado Toast")
}
.font(.title)
.lineLimit(1)
.frame(width: 300)
.border(Color.blue)
```



By default, all three views have the same layout priority. This means that the image, being more strict in its size, will be sized first. The remaining space is then divided equally between the two text views. However, if one of the text views, let's say "Eat More," is smaller in size, it will be able to fit itself while the "Avocado Toast" text will be truncated.

Now, let's explore the power of layout priority. By assigning a higher layout priority to the "Avocado Toast" text, we prioritize its size over the other text views. The image still maintains its strict size, so it will still be sized first. This type of layout priority is particularly useful when working with texts that adjust to available space. It allows you to determine which text is more important for the user to read.

```
HStack {
  Text("Eat more")
  Image("avocado_large")
  Text("Avocado Toast")
  .layoutPriority(1)
}
```



```
.font(.title)
.lineLimit(1)
.frame(width: 300)
.border(Color.blue)
```

You can apply multiple layout priorities to different views within a VStack or a Stack. For example, if you have another text view, you can set the layout priorities to 2, 1, and 0 to define the order of sizing priority.

However, applying layout priority to certain views, such as colors, may yield unexpected results. For instance, if you have a HStack with five colors, all with the same layout priority, the layout system will distribute the available space equally among them.colors.

```
HStack(spacing: 0) {
  Color.red
  Color.orange
  Color.yellow
  Color.green
  Color.blue
}
.frame(width: 100, height: 100)
.mask(Circle())
```



But by assigning a layout priority of 1 to the blue color, it becomes the highest priority and receives all the available space, leaving none for the other

```
HStack(spacing: 0) {
  Color.red
  Color.orange
  Color.yellow
  Color.green
  Color.blue
  .layoutPriority(1)
}
.frame(width: 100, height: 100)
.mask(Circle())
```



To address this, you can add frames to the views. For example, setting a minimum width of 10 ensures that the green color receives exactly the minimum space. Adding a maximum width wouldn't make a difference in this case. You can also define a frame with minimum, ideal, and maximum widths, such as 10, 20, and 100. This allows the view to take whatever space is offered up to its maximum width.

```
HStack(spacing: 0) {
  Color.red
  Color.orange
  Color.yellow
  Color.green
  .frame(minWidth: 20)
  Color.blue
  .frame(minWidth: 10,
        idealWidth: 20,
        maxWidth: 50)
  .layoutPriority(1)
}
```



```
.frame(width: 100, height: 100)
.mask(Circle())
```

In the case of conflicting layout priorities, the view with the highest priority will be sized first and it will be offered all the space. This behavior may seem a bit unfortunate, as it gives everything to the highest priority view. But when it comes to texts, it actually makes sense. For example, when using 2 text views type, the layout system prioritizes giving all the space to the most important text, gradually shrinking the important text until it is completely truncated.

Applying layout priority to views within a VStack or a Stack is incredibly useful for making your views adaptable to different screen sizes, dynamic types, and even landscape or portrait modes. It ensures that your content adjusts to the available space and provides a seamless user experience.

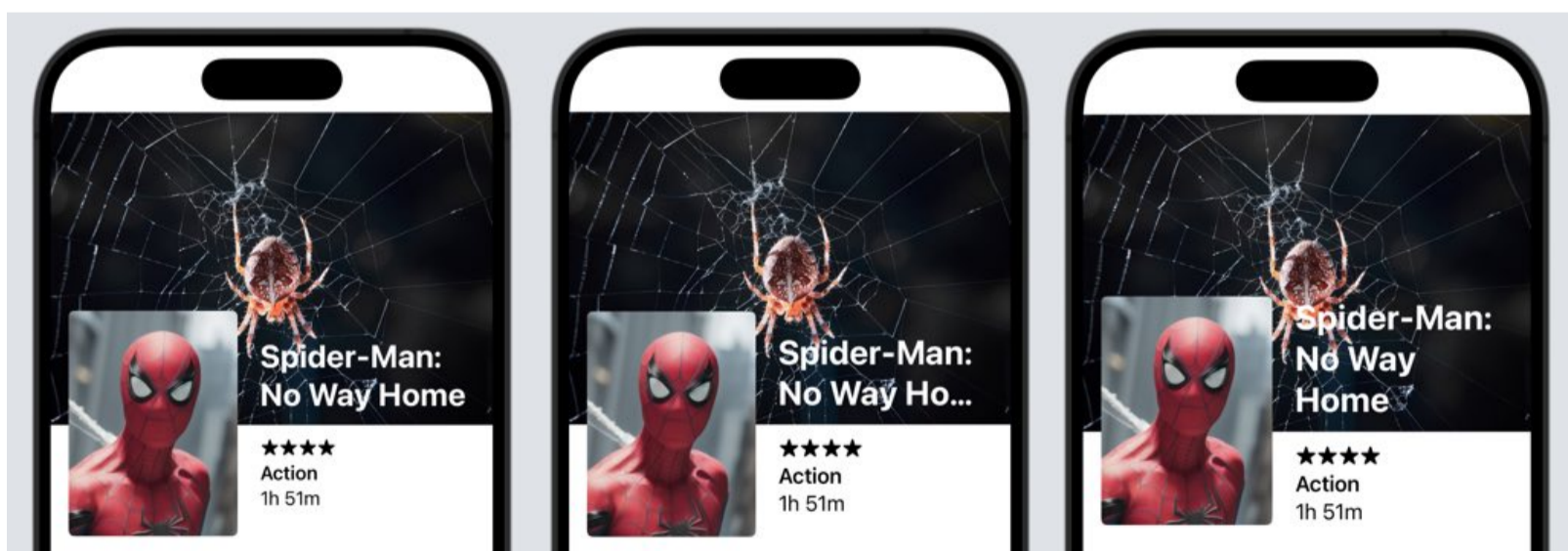
5.5 SIZING TEXT VIEWS

In this section, we will dive deeper into sizing text views in SwiftUI. Text views are conservative views that try to adjust automatically, making them quite flexible in terms of layout. By default, text views aim to fit themselves within the available space.

Restricting Text Views

In a previous lesson, we discussed using the `layoutPriority` modifier to specify which views should adapt their size. Additionally, you can restrict a text view's size by using **frames**, such as **fixed width** or **maximum width**. You can also limit the height by setting it to a certain value or using the `lineLimit` modifier to display only a single line.

For instance, in the `MovieDetailView`, the title of the movie is defined with a large and bold font. To ensure it fits well within the layout, we can add a **line limit of two**. By experimenting with different line limits and dynamic types, we can find the optimal display for our text.



```

VStack {
  ZStack {
    ResizableImageView(imageName: "spider")

    HStack(spacing: 20) {
      Image(...)

      VStack(alignment: .leading, spacing: headerSpacing) {
        Text(title)
          .font(.title).bold()
          .foregroundColor(.white)
          .lineLimit(2)

        VStack(alignment: .leading) {
          Text("★★★★")
          Text(...)
        }
      }
    }
  }
  .padding(.horizontal)
}

Text(superhero.biography)
  .padding()
}

```

Dealing with Multi-Line Text

When working with multi-line text, it's important to consider the readability and alignment of the content. SwiftUI provides modifiers to control the alignment and truncation of multi-line text views.

- **Alignment:** By default, text views are aligned to the leading edge. You can modify this alignment using the `multilineTextAlignment` modifier to center or trail the text.
- **Readability:** Leading alignment is often recommended for multi-line text as it reduces friction for the reader. When the text jumps to the next line, the reader's eyes naturally return to the same position. Apple's UX designers have wisely set the default alignment to leading, ensuring a better user experience.

```

Text(text)
  .multilineTextAlignment(.trailing)

```

Architecto expedita suscipit praesentium. Et et officiis libero cumque quisquam ut nulla inventore voluptas molestiae illo aut quos qui. Fugiat cumque et sed molestias reprehenderit quis dolores incidunt animi ipsa facere dolores fuga vero aspernatur. Ipsa eaque atque sit explicabo quasi est nam fugit vitae rem. Nihil soluta culpa nam occaecati. In commodi quibusdam eos q...

leading

Architecto expedita suscipit praesentium. Et et officiis libero cumque quisquam ut nulla inventore voluptas molestiae illo aut quos qui. Fugiat cumque et sed molestias reprehenderit quis dolores incidunt animi ipsa facere dolores fuga vero aspernatur. Ipsa eaque atque sit explicabo quasi est nam fugit vitae rem. Nihil soluta culpa nam occaecati. In commodi quibusdam eos q...

center

Architecto expedita suscipit praesentium. Et et officiis libero cumque quisquam ut nulla inventore voluptas molestiae illo aut quos qui. Fugiat cumque et sed molestias reprehenderit quis dolores incidunt animi ipsa facere dolores fuga vero aspernatur. Ipsa eaque atque sit explicabo quasi est nam fugit vitae rem. Nihil soluta culpa nam occaecati. In commodi quibusdam eos qui ut exercitationem exercitationem non eu...

trailing

Text Truncation Mode

Text will automatically be truncated if it does not fit the available space. Truncated text will be indicated by three dots. By default, the truncation is at the end of the text. But you can change it with the `truncationMode` modifier:

```
Text(superhero.biography)
  .lineLimit(2)

Text(superhero.biography)
  .lineLimit(2)
  .truncationMode(.middle)

Text(superhero.biography)
  .lineLimit(2)
  .truncationMode(.head)
```

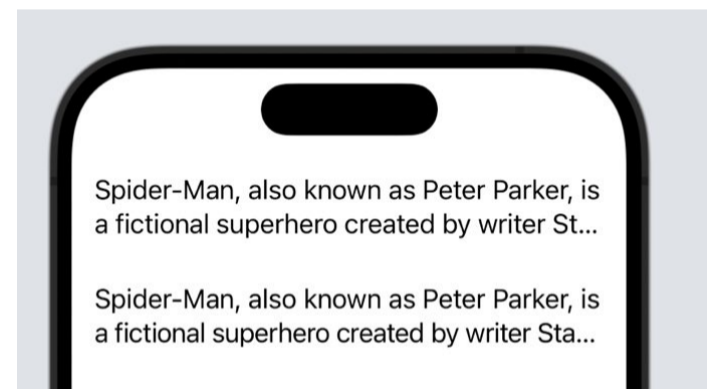


Allows Tightening

You can also allow tightening, which will compress the space between characters, if it is limited in space:

```
Text(superhero.biography)
  .lineLimit(2)

Text(superhero.biography)
  .lineLimit(2)
  .allowsTightening(true)
```

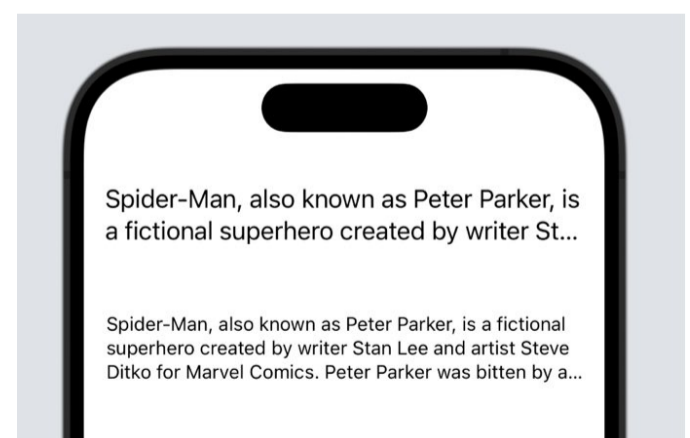


Minimum Scale Factor

You can also allow the text to scale down to fit in the available space. The following minimum scale factor allows the text to shrink to 75%:

```
Text(superhero.biography)
  .frame(height: 70)

Text(superhero.biography)
  .frame(height: 70)
  .minimumScaleFactor(0.75)
```

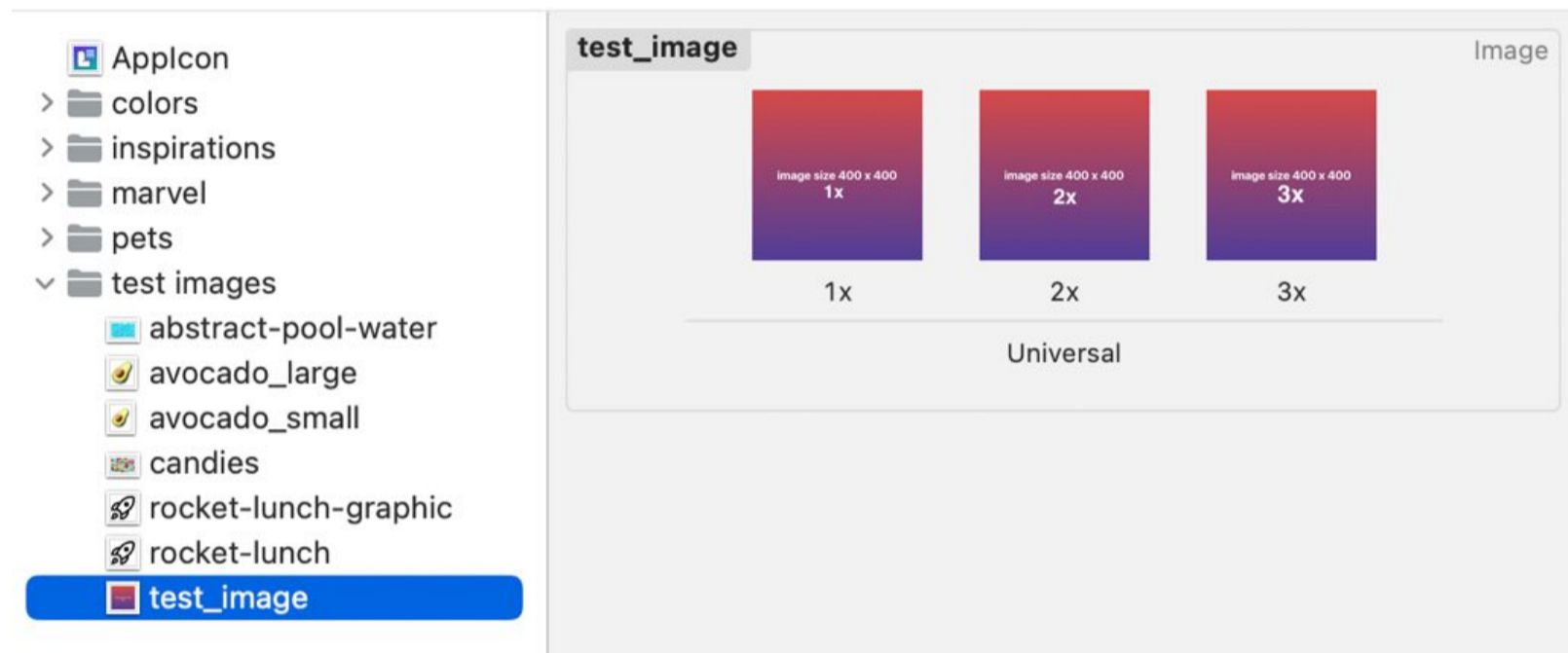


5.6 SIZING IMAGES

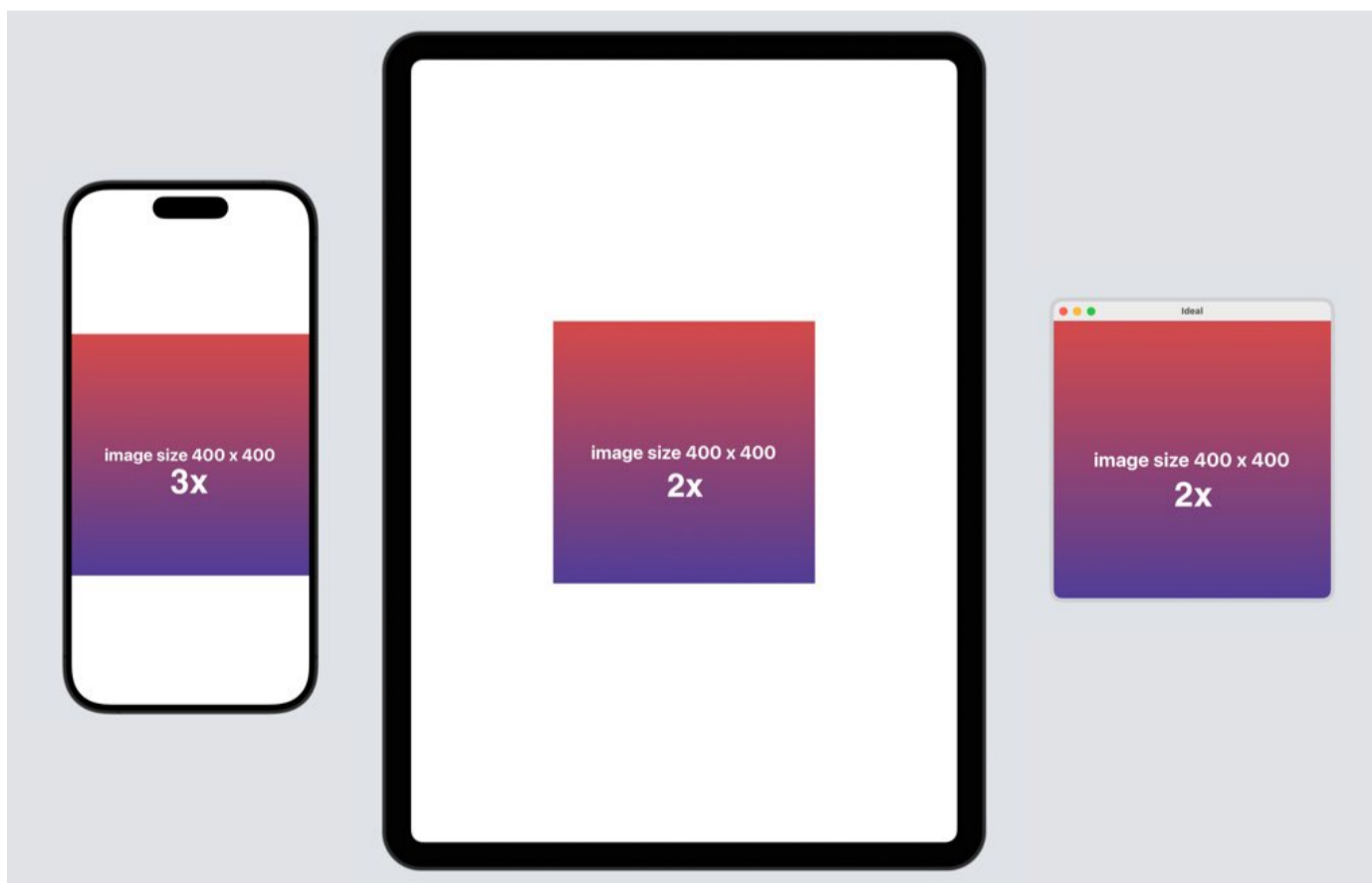
Working with images in an iOS app is a common task, but sizing them correctly can be a bit tricky. In this section, we will explore different types of images, how to size them, frame them correctly, and even handle async images downloaded from the internet.

Image Size and Resolutions

In the assets folder, I have provided a set of images with different resolutions. Typically, we have 1x, 2x, and 3x images for different screen resolutions.



For example, iPhones usually use 3x images due to their higher pixel density, while iPads and Macs use 2x images. I used images that show a text with their size. “image size 400 by 400” means the image is upscaled to 800 by 800 points for 2x resolution and 1,200 by 1,200 points for 3x resolution.



Using Images in SwiftUI

To use an image, simply call the Image view and provide the image name from the asset catalog. For example, Image("test") will display the image named "test". By default, SwiftUI selects the appropriate image based on the device's screen resolution.

```
struct ImageFitScalingView: View {
    var body: some View {
        Image(.test)
    }
}
```

Resizing Images

Images have a fixed size per default. In order to make them scale, you need to apply the resizable() modifier. However, this modifier doesn't preserve the image's aspect ratio, resulting in stretching.

```
struct ImageScalingView: View {
    var body: some View {
        Image(.cat1)
            .resizable()
    }
}
```



To maintain the aspect ratio, we can use the `aspectRatio` modifier with the `fit` content mode. This ensures the image fits exactly to the edges of the screen while preserving its natural aspect ratio. Similarly, you can also use **scaleToFit**:

```
struct ImageScalingView: View {
    var body: some View {
        Image(.cat1)
            .resizable()
            //aspectRatio(nil, contentMode: .fit)
            //aspectRatio(contentMode: .fit)
            .scaledToFit()
    }
}
```

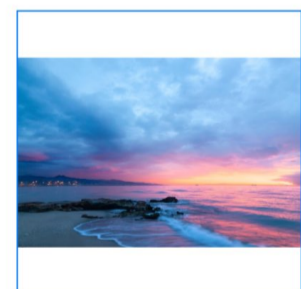
In some cases, we may want to **fill the entire screen with an image**. To achieve this, we can use the `aspectRatio` modifier with the `fill` content mode. This scales the image to fill the frame, potentially overflowing the edges. To include the safe area as well, we can use the `ignoresSafeArea` modifier.

```
struct ImageScalingView: View {
    var body: some View {
        Image(.cat1)
            .resizable()
            //aspectRatio(nil, contentMode: .fill)
            //aspectRatio(contentMode: .fill)
            .scaledToFill()
    }
}
```

Reframing Images

Sometimes, we may want to set a specific frame for an image. We can achieve this by combining the `frame` modifier with the `resizable` modifier. However, the aspect ratio of the image might not fit with the frame. If you use **scaleToFit**, the image will be scaled until one of its edges touches the frame:

```
Image(.beach)
    .resizable()
    .scaledToFit()
    .frame(width: 200, height: 200)
    .border(Color.blue)
```



To make the image fill the frame use the `scaleToFill` modifier. This will however overflow the frame. If we want to clip the image to its frame, we can add the `clip` modifier.

```
Image(.beach)
    .resizable()
    .scaledToFill()
    .frame(width: 200, height: 200, alignment: .trailing)
    .border(Color.blue)
```



frame alignment center
scaleToFill



frame alignment trailing
scaleToFill



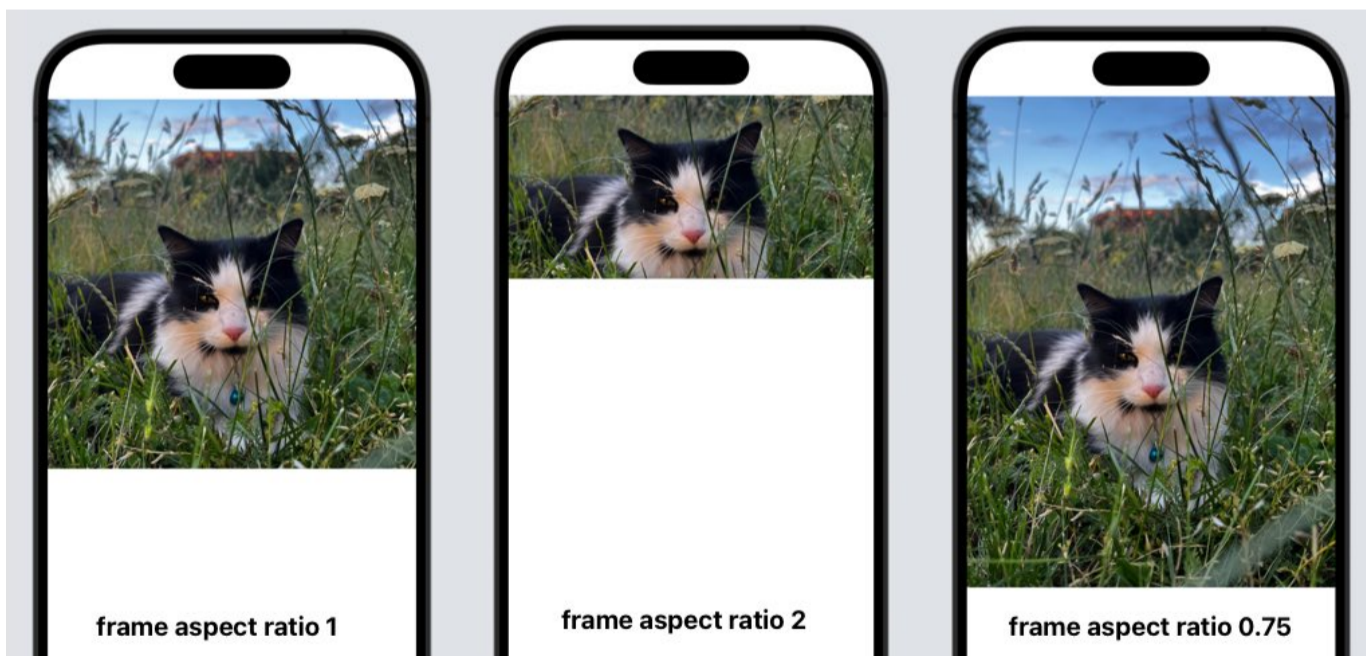
frame alignment trailing
scaleToFill
clipped

To fine-tune which areas of the image are visible within the frame, we can use the alignment property of the frame modifier. By default, it is set to center, but we can also use leading, trailing, top, or bottom depending on the desired result.

Ideally, all our images would have the same aspect ratio, eliminating the need for fixing and clipping. However, when working with images downloaded from the server, we may encounter various aspect ratios. By using the frame and clip modifiers, we can handle different aspect ratios and adjust which areas of the image are visible within the frame.

Setting a Frame Aspect Ratio

One thing I always struggled with, is setting a frame aspect ratio that changes the size of the image but keeps its natural image aspect ratio. Here are 3 examples where I set the frames aspect ratio to 1, 2, and 0.75. These images scale to fit the screen size:



frame aspect ratio 1

frame aspect ratio 2

frame aspect ratio 0.75

In order to get this working, I have to use another view as the placeholder. I am using a color view because it is a greedy view and resizes. To this view, I can attach the aspectRatio modifier with the desired value. This is occupying the space I want. The image itself is added in the overlay to this placeholder and thus gets the same size as the placeholder.

I use a scale to fill and clip to make sure it only is displayed in the area of the placeholder. This is quite a bit of complexity but gets the job nicely done.

```
struct ImageAspectView: View {  
    let imageName: String  
    let frameAspectRatio: CGFloat  
  
    var body: some View {  
        Color.cyan // Placeholder  
        .aspectRatio(frameAspectRatio, contentMode: .fit)  
        .overlay {  
            Image(imageName)  
                .resizable()  
                .aspectRatio(nil, contentMode: .fill)  
        }  
        .clipped()  
    }  
}
```

5.7 Upscaling images and Bitmap vs Vector graphics

When working with images, there may be situations where you need to upscale them to fit a certain frame. Let's take a look at an example with a 25 by 30-pixel large png image:

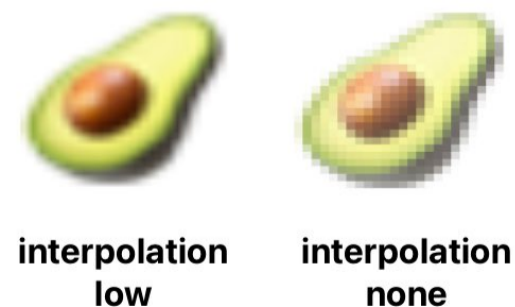
```
Image(.avocadoSmall)  
    .resizable()  
    .frame(width: 100, height: 100)
```



In this example, we have a small image of an avocado that we want to upscale. By using the `resizable()` modifier and setting a frame width and height, we can increase the size of the image. However, when upscaling, you may notice that the image becomes pixelated.

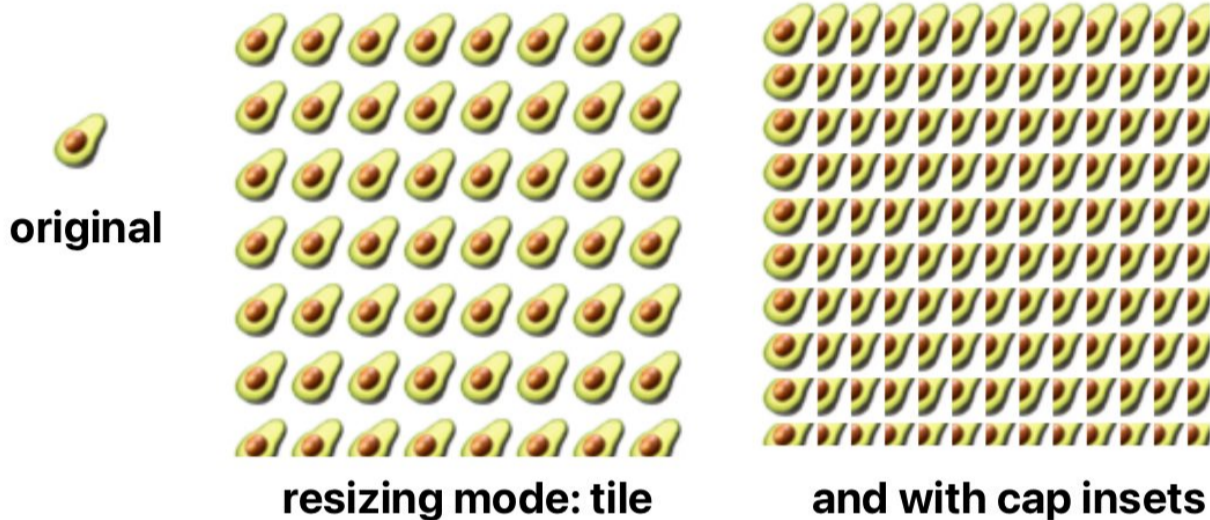
To address this, SwiftUI provides an interpolation parameter that allows you to control how the image is upscaled. The default interpolation mode tries to adjust the image using some interpolation techniques. However, if you want to maintain the pixelated look, you can set the interpolation mode to `none`.

```
Image(.avocadoSmall)  
    .resizable()  
    .interpolation(.none)  
    .frame(width: 100, height: 100)
```



Additionally, you can use the tile option to repeat the image instead of stretching it to fill the frame. This can be useful for creating visual effects or backgrounds.

```
Image(.avocadoSmall)
  .resizable(capInsets: .init(top: 10, leading: 10, bottom: 0, trailing: 0),
            resizingMode: .tile)
  .frame(width: 200, height: 200)
```



Vector Graphics

Bitmap graphics, such as JPEGs, store image information bit by bit or point by point. On the other hand, vector graphics store drawing instructions. Let's explore the advantages of vector graphics:

- **Resolution Independence:** Vector graphics are rendered during runtime, allowing for infinite scalability. This means you are not limited by image sizes anymore.
- **Sharp Edges:** Vector graphics maintain sharp edges regardless of the size they are rendered.

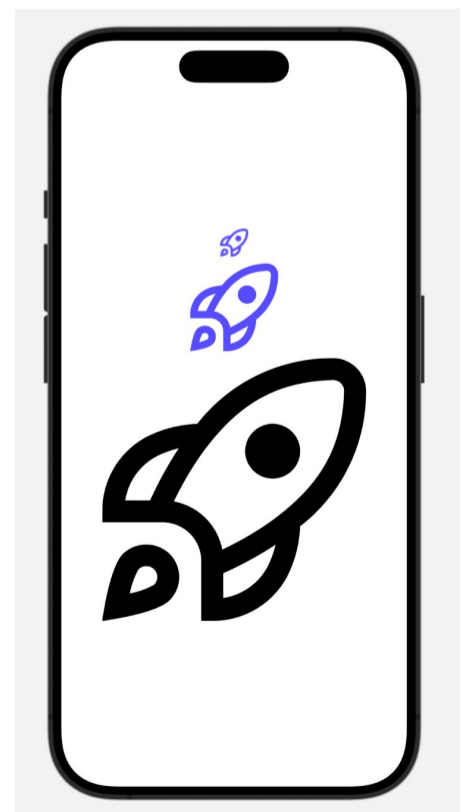
To demonstrate this, let's use a PNG (bitmap) image and a SVG (vector) image:

```
Image(.rocketLunch) // png
  .renderingMode(.template)
  .foregroundColor(.accent)

Image(.rocketLunchGraphic) // svg
  .resizable()
  .renderingMode(.template)
  .foregroundColor(.accent)
  .scaledToFit()
  .frame(width: 100)

Image(.rocketLunchGraphic) // svg
  .resizable()
  .scaledToFit()
  .frame(width: 300)
```

In this example, the PNG image is displayed at its original size of 32x32 pixels, while the SVG image takes on the size defined in the drawing



instructions. As you can see, the SVG image maintains sharp edges, thanks to its vector nature. This ensures that the image always looks sharp, regardless of the size.

By changing the image to use the render mode of template you can also change the foreground color gradient of the image. This works also with png format:

```
struct GradientSpidermanView: View {
    var body: some View {
        Image("spiderman")
            .resizable()
            .renderingMode(.template)
            .foregroundColor(LinearGradient(colors: [Color.pink, Color.purple],
                startPoint: .topLeading,
                endPoint: .bottomTrailing))

            .scaledToFit()
            .padding([.leading, .top])
            .background(Color.backgroundColor2)

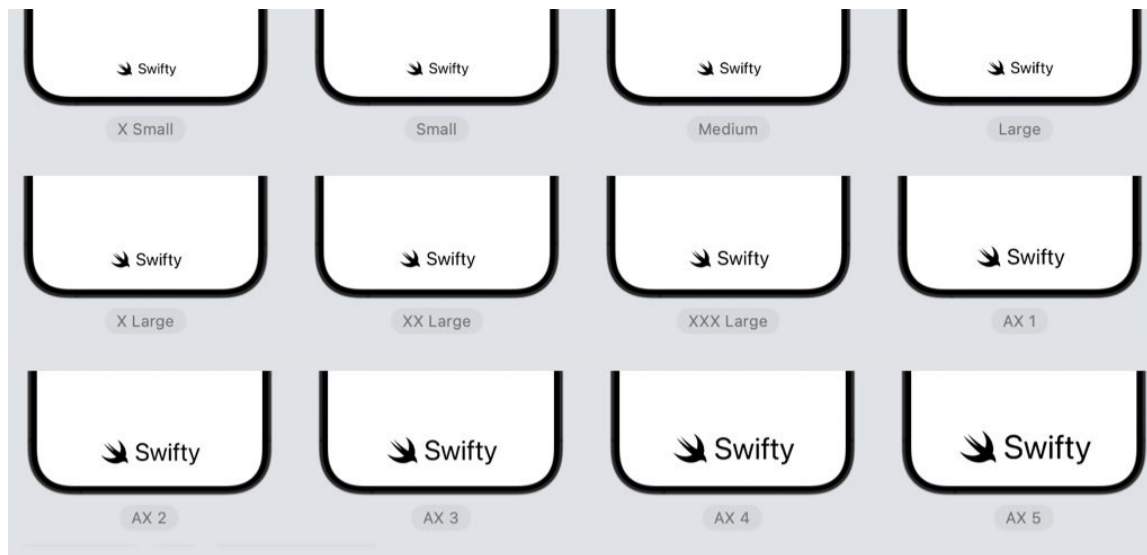
        Spacer()
    }
}
```



5.8 Sizing System Icons

System icons behave a little differently than images from the assets, mainly because they can be used together with text. To demonstrate this, let's start by creating a label that combines a text and a system icon. We can achieve this by using a Label view. For example, we can create a label with the text "Swift" and the Swift icon:

```
Label("Swift", systemImage: "swift")
```



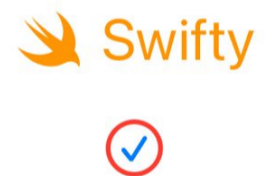
By default, the system icon is aligned to the text and scales with it. This means that if you use dynamic type variants, both the image and the text will scale nicely together.

Image Styling

Additionally, since system icons behave like text, you can use the **foregroundColor** or **foregroundStyle** modifiers to change their color. For example, you can apply the accent color or any other color you prefer.

```
Label("Swift", systemImage: "swift")
    .foregroundColor(.orange)

Image(systemName: "checkmark.circle")
    .foregroundStyle(.blue, .red)
```



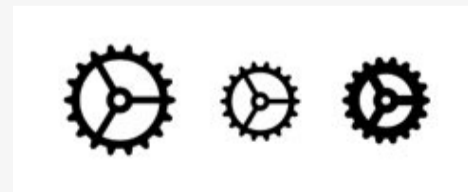
Resizing System Icons

To resize system icons, you have several options. You can specify a specific font size or use a system size. You can also apply different font weights to the image, like bold or black, to make it stand out. For example:

```
Image(systemName: "gear")
    .font(.title)

Image(systemName: "gear")
    .font(.system(size: 20))

Image(systemName: "gear")
    .font(.system(size: 20))
    .fontWeight(.bold)
```



If you want to make your icons more visually appealing, you can adjust their size using the **imageScale** modifier. This allows you to make the icons larger or smaller than the default size. You can choose from options like **.small**, **.medium (default)**, or **.large**. By adjusting the image scale, you can deviate from the text size slightly. This also adapts to dynamic type variants.


```
Image(systemName: "swift")
    .imageScale(.small)
Image(systemName: "swift")
    .imageScale(.medium)
Image(systemName: "swift")
    .imageScale(.large)
```



Alternatively, you can use the frame modifier to set a specific size for the system icon. This approach treats the icon as an image and allows you to define a fixed width and height. For example:

```
Image(systemName: "lasso.badge.sparkles")
    .font(.system(size: 20))
    .border(Color.yellow)

Image(systemName: "lasso.badge.sparkles")
    .resizable()
    .frame(width: size, height: size)
    .border(Color.yellow)
```



The sizes of the images vary depending on whether you treat them like text (with font) or as an image with resizable. The 2 examples above are mainly different in height. The icon image text uses the text height.

5.9 AsyncImage

In this section, we will explore how to effectively size and display images that are downloaded from a server. With the release of iOS 15, Apple introduced the AsyncImage view, which simplifies the process of downloading and displaying images asynchronously. The most basic use of AsyncImage is:

```
AsyncImage(url: URL(string: "https://picsum.photos/id/12/200"))
```

In this example, we use the AsyncImage initializer that takes a URL to handle the downloaded image. We provide a URL to the Lorem Picsum service, which uses a dummy image with a size of 200 by 200 pixels. SwiftUI will use 200 by 200 points to display the image, which makes it look very blurry on a high resolution iPhone.

If you want to resize a downloaded image, you need to use a different AsyncImage initializer:

```
AsyncImage(url: URL(string: "https://picsum.photos/id/12/600")) { image in
    image.resizable()
} placeholder: {
    ProgressView()
}
.frame(width: 200, height: 200)
```

The closure receives the downloaded image as a parameter and we can apply any necessary modifications, such as **making it resizable and setting its frame**. I am using a 600 by 600 pixel image and set its frame to 200 by 200 points.

In the blow screen, you can see the more blurry image on top that only has a resolution of 200 by 200 and the **higher resolution sharper image** on the bottom:



Displaying Images for the Device Screen Resolution

To ensure that the downloaded image is displayed with the correct size, we need to consider the **screen's resolution**. We can use the Environment property **displayScale** to determine the scale factor of the screen. For example, on an iPhone, the scale factor would be 3. We can then multiply the desired size by the scale factor to obtain the appropriate image size.

```
struct AsyncImageExampleView: View {  
    let urlString = "https://picsum.photos/id/12/"  
    @Environment(\.displayScale) var scale  
    let size: CGFloat = 200  
    var urlString: String {  
        urlString + "\(Int(size * scale))"  
    }  
    var body: some View {
```

```

        AsyncImage(url: URL(string: urlString)) { image in
            image.resizable()
        } placeholder: {
            ProgressView()
        }
        .frame(width: size, height: size)
    }
}

```

In this updated example, we calculate the appropriate image size based on the screen's scale factor. We construct the URL using the calculated size and pass it to the AsyncImage initializer. The downloaded image is then displayed with the correct size.

Handling Placeholder and Error States

AsyncImage provides additional functionality for **handling placeholder and error states**. We can specify a placeholder view to be displayed while the image is being loaded, and we can handle loading errors using the **image phase** of the AsyncImage initializer.

```

struct AsyncImageExampleView: View {

    let baseURLString = "https://picsum.photos/id/12/"

    @Environment(\.displayScale) var scale
    let size: CGFloat = 200
    var urlString: String {
        baseURLString + "\(Int(size * scale))"
    }

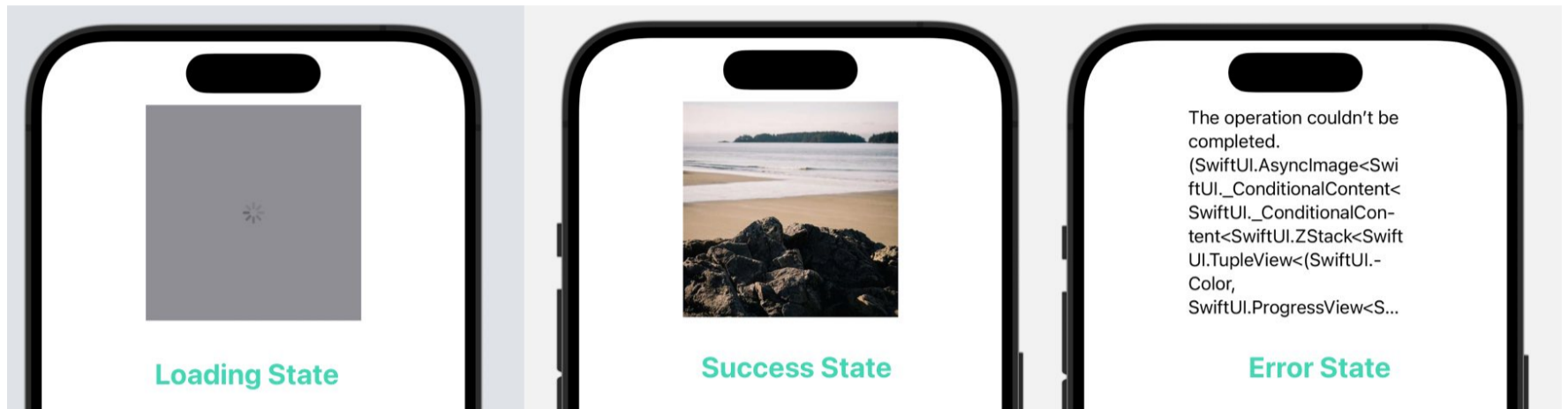
    var body: some View {
        AsyncImage(url: URL(string: urlString),
                  scale: 3,
                  transaction: .init(animation: .bouncy)) { phase in
            switch phase {
            case .empty:
                ZStack {
                    Color.gray
                    ProgressView()
                }
            case .success(let image):
                image.resizable()
            case .failure(let error):
                Text(error.localizedDescription)
                // use placeholder for production app
            @unknown default:
                EmptyView()
            }
        }
        .frame(width: size, height: size)
    }
}

```

In this updated example, we handle different phases of the image-loading process using a switch statement. We display a ProgressView as a placeholder while the image is being loaded, show the

downloaded image on success, and display an error message on failure. We also handle any unknown cases with an EmptyView.

The frame is added around the AsyncImage. This assures that the view has in **all states the same size**. This is important when you want to lazily load images in a list or scroll view and want to ensure that the **scrolling is smooth and without jumps**.

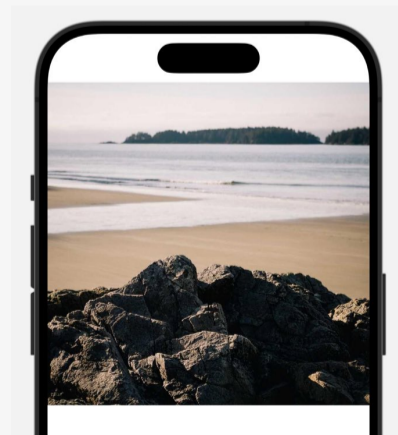


Downloading Images that Fit Perfectly

In some cases, we may want to download an image that fits perfectly within a given area, maintaining its aspect ratio. I want to download an image that is large enough so looks sharp that is the right resolution and that takes up exactly the space on the screen. But not more, since I don't want to download more data than necessary. This is especially important when your user is on mobile data.

To achieve this, we can use the **GeometryReader** view to **determine the available size** and calculate the appropriate image size. The following code would calculate the size of a **square image** since I am using the same value for the height and width of the image:

```
struct SizedAsyncImageExampleView: View {  
  
    let urlString = "https://picsum.photos/id/12/"  
    @Environment(\.displayScale) var scale  
  
    var body: some View {  
        GeometryReader(content: { geometry in  
            AsyncImage(url: url(in: geometry.size.width)) { phase in  
                ...  
            }  
            .frame(width: geometry.size.width,  
                  height: geometry.size.width)  
        })  
    }  
  
    func url(in width: CGFloat) -> URL? {  
        let imageWidth = width * scale  
        let urlString = urlString + "\(Int(imageWidth))"  
        return URL(string: urlString)  
    }  
}
```





Next, to correctly download the **image and keep its aspect ratio**, I need to know the image dimensions. For the Lorem Ipsum AP, I am downloading this information from the server first. Here is the basic information of the data model:

```
struct PicsumPhoto: Codable, Identifiable {
    let id: String
    let author: String
    let width: CGFloat
    let height: CGFloat
    let url: String
    let downloadUrl: String

    var aspectRatio: CGFloat {
        width / height
    }

    static func example() -> PicsumPhoto {
        PicsumPhoto(id: "63",
                    author: "Justin Leibow",
                    width: 5000,
                    height: 2813,
                    url: "https://unsplash.com/photos/ZJsseAxEcqM",
                    downloadUrl: "https://picsum.photos/id/63/5000/2813")
    }
}
```

I am getting the width and height of the full image and using it to calculate the aspect ratio. This information is used in the SwiftUI view when I calculate what image I want to download in the url function:

```
struct AspectRatioSizedAsyncImageExampleView: View {

    let photo = PicsumPhoto.example()
    let urlString = "https://picsum.photos/id/"
    @Environment(\.displayScale) var scale

    var body: some View {
        GeometryReader(content: { geometry in
            AsyncImage(url: url(in: geometry.size.width),
                      scale: 3,
                      transaction: .init(animation: .bouncy)) { phase in
                ...
            }
            .frame(width: geometry.size.width,
                  height: geometry.size.width / photo.aspectRatio)
        })
    }
}
```

```

    })
  }

  func url(in width: CGFloat) -> URL? {
    let imageWidth = width * scale
    let imageHeight = imageWidth / photo.aspectRatio
    let urlString = "\(baseURLString)\(photo.id)/\(Int(imageWidth))/\
(Int(imageHeight))"
    return URL(string: urlString)
  }
}

```

In this example, we use the GeometryReader to determine the available width and calculate the corresponding height based on the aspect ratio of the image. We then construct the URL using the calculated width and height and pass it to the AsyncImage initializer. The downloaded image is displayed with the correct aspect ratio and fits perfectly within the available area.

Loading Mechanism

SwiftUI will take care of loading the image **asynchronously** using the provided URL. It manages the download and caching of the image, so you don't have to write custom networking code for image loading. **If the view disappears before the download is complete, the URL request is canceled.**

Animations

You can use the transaction parameter of AsyncImage to add an animation when the image is shown. This will add a fade in animation:

```

AsyncImage(url: ...,
           transaction: .init(animation: .bouncy(duration: 1))) { phase in
    ...
}

```



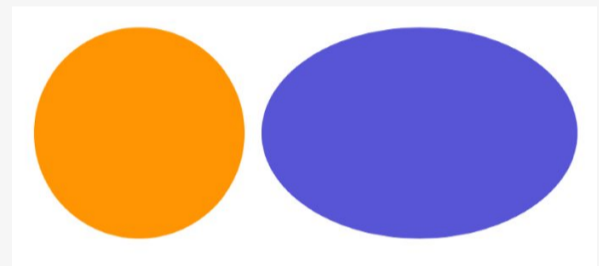
5.10 ASPECT RATIO

In this section, we will explore the concept of aspect ratio in SwiftUI. Aspect ratio allows us to control the proportions of our views, ensuring they maintain a specific width-to-height ratio.

Aspect ratio can be attached to any view, but it may not work for all views. For instance, if we try to use an **aspect ratio of 1.5 with a fill mode on a Text view, the view won't change**. This is because Text views are designed to preserve their own aspect ratio to avoid distortion.

However, aspect ratio works well with **shapes like rectangles, ellipses and circles**. Circles naturally maintain an aspect ratio of 1, so applying an aspect ratio modifier won't change their appearance. On the other hand, if we use an ellipse, it can stretch in both directions. To prevent stretching, we can set an aspect ratio of 1 to make the ellipse appear as a circle again.

```
Ellipse()  
    .fill(Color.orange)  
    .aspectRatio(1, contentMode: .fit)  
    .frame(height: 100)  
  
Ellipse()  
    .fill(Color.indigo)  
    .aspectRatio(1.5, contentMode: .fit)  
    .frame(height: 100)
```



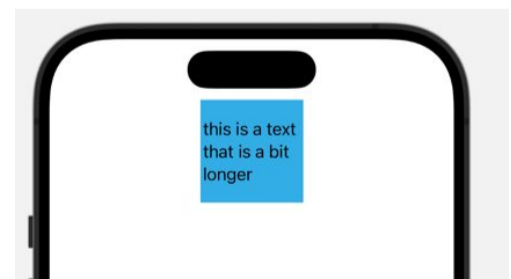
Aspect ratio can also be used with colors, specifically with stretchy views that expand in two directions. For example, if we use a color with an aspect ratio of 2, it will occupy the available space while maintaining the desired aspect ratio.

```
Color.cyan  
    .aspectRatio(2, contentMode: .fit)
```



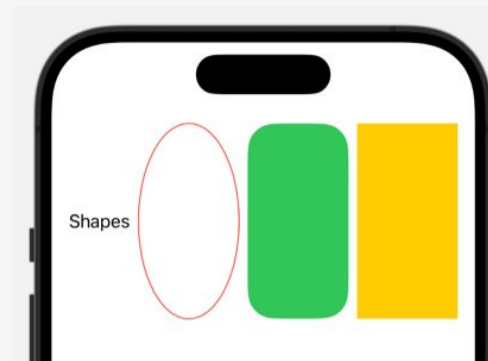
Instead of adding aspect ratio to individual views, we can also apply it to a whole stack. For instance, we can use a ZStack with colors and add a text view. By applying an aspect ratio to the ZStack, both the colors and the text will have the same aspect ratio.

```
ZStack {  
    Color.cyan  
    Text("this is a text that is a bit longer")  
}  
.aspectRatio(1, contentMode: .fit)  
.frame(height: 100)
```



Additionally, aspect ratio can be applied to an HStack. Let's create an HStack with three shapes: an ellipse, a rounded rectangle with a corner radius of 25, and a rectangle. We can then set an aspect ratio of 4 to the HStack, making its width four times its height.

```
HStack {
  Text("Shapes")
  Ellipse()
    .stroke(Color.red)
  RoundedRectangle(cornerRadius: 25.0)
    .fill(Color.green)
  Rectangle()
    .fill(Color.yellow)
}
.aspectRatio(2, contentMode: .fit)
.padding()
```



In some cases, when **working with images**, aspect ratio may not be the best option. By default, images in SwiftUI are resizable and will stretch to fill the available space. If we want to preserve the image's aspect ratio, we can use the `scaleToFit` or `scaleToFill` modifiers instead of aspect ratio.

However, if we need to set a specific frame size for an image while maintaining its aspect ratio, we can use a neat trick. We can create a **self-resizing image view** that respects the aspect ratio of the image, but allows us to set a separate frame aspect ratio. This way, we can handle the layout and drawing of the image separately.

5.11 SCALE EFFECT

The `scaleEffect` modifier allows us to adjust the scale of a view, influencing its layout and how space is distributed. Let's create an HStack and add three views to it.

```
HStack {
  Text("Hello, World!")
  Text("Make me larger")
    .foregroundColor(.accent)
    .padding()
    .background(Color.yellow)
  Text("After the shape")
}
```

Hello, World! Make me larger After the shape

By using Text views within the HStack, the space is distributed evenly, and we can see all three text views. Now, let's apply the `scaleEffect` modifier to the "Make me larger" view. You can specify a scale value for both the horizontal and vertical directions, independent of each other.

```
HStack {
  Text("Hello, World!")
    .zIndex(1)
  Text("Make me larger")
}
```

Hello, World! Make me larger After the shape


```

        .foregroundColor(.accent)
        .padding()
        .background(Color.yellow)
        .scaleEffect(x: 1.5, y: 1.5)
    Text("After the shape")
}

```

As you can see, the view is scaled up from the center, and the view moves outward from the center. You can also use other anchor points like `.leading` or `.topTraiing`:

```

.scaleEffect(x: 1.5, y: 1.5, , anchor: .leading)

```

However, there are a few things to keep in mind when using the `scaleEffect` modifier.

- layout neutral changes: `scaleEffect` only scales the visible part of the view and doesn't change the layout.
- The original "Make me larger" text still occupies the same area. But it becomes larger, the view can **overlay other views** next to it.
- It simply scales up without rendering additionally. Thus your view can **look blurry**. It's best not to overdo the scaling. A scale of 1.5 and 1 is often a good choice.

Use Cases

Scale effect modifier can be used for resizing views, it's often more suitable for **animation effects**. For instance, you can create fancy background animations. It is a good fit if you want to add scroll animations because the layout is not changed, the views inside the scrolling spacing stays constant and the scrolling can remain smooth. By animating the scale, you can make views appear smaller when entering the scroll view and scale them up to their appropriate size, creating visually appealing scroll animations.

5.12 CONTENT EDGES: SAFE AREA, PADDING AND MARGINS

In this section, we will explore how to add white space and adjust the layout of individual views using content edges or insets. We will cover concepts such as the safe area, padding, and margins.

Understanding the Safe Area

The safe area refers to the space on the screen that is not occupied by system elements like the time, Wi-Fi connection, and battery life indicators. By default, views do not use these areas to avoid overlapping with important system information.




To visualize the safe area, you can run your app on the simulator and observe the top and bottom areas that are not occupied. If you want to fill these areas with your content, you can use the `edgesIgnoringSafeArea` modifier. In iOS 14 and later, you can use `ignoresSafeArea(.all)` to ignore all safe areas or specify a specific edge like `.bottom` to only use the bottom area.

```
Image(.cat1)
    .resizable()
    .scaledToFill()
    .ignoresSafeArea(.container, edges: .all)
```

Adding Padding to Views

Padding is a way to add space around a view. You can apply padding to any view using the `padding()` modifier. The amount of padding applied depends on the device and its available space. On an iPhone, the padding will be smaller compared to an iPad.

```
Text("Hello, World!")  
  .padding()  
  .background(Color.yellow)
```



To set a specific padding value, you can use the `padding(_:)` modifier. For example, `padding(50)` adds 50 points of padding around the view in all directions.

```
Text("Hello, World!")  
  .padding(50)  
  .background(Color.yellow)
```



You can also specify the direction of padding using arguments like `.horizontal` or `.leading`. If you need to add multiple paddings, you can either provide an array of edges or chain multiple `padding(_:)` modifiers.

```
Text("Hello, World!")  
  .padding([.top, .horizontal], 50)  
  .background(Color.yellow)
```

```
Text("Hello, World!")  
  .padding(.vertical, 10)  
  .padding(.horizontal, 20)  
  .background(Color.yellow)
```



Padding is particularly useful when you want to create white space around text or prevent views from touching the edges.

Content Margins for Scrollable Views

Content margins allow you to add edges or margins to scrollable content, such as scroll views or text editors. They provide more control over the layout of the content within these views. The `contentMargin(_:)` modifier can be used with a scroll view or a text editor.

```
ScrollView {  
  ResizableImageView(imageName: "cat_2")  
  
  VStack(alignment: .leading, spacing: 10) {  
    Text(Lorem.title)  
      .font(.title)  
    Text(Lorem.paragraphs(3))  
  }  
}
```

```
.contentMargins(20)
.contentMargins(5, for: .scrollIndicators)
```

On the left, you see the scroll view without the content margins, and on the right with a margin of 20 points. Additionally, I added 5 points to the scroll indicator, so that it does not overlap the views anymore:



Content margins can be set in both horizontal and vertical directions using arguments like `.horizontal` and `.vertical`. You can also specify specific values in points for more precise control over the margins.

```
.contentMargins(.trailing, 20)
```

Content margins are especially useful when you want to fine-tune the layout of scrollable content or add margins to text editors:

```
struct TextEditorInsetExampleView: View {
    @State private var text = "something amazing ...
something is coming here"

    var body: some View {
        TextEditor(text: $text)
            .border(Color.black)
            .contentMargins(.horizontal, 20)
            .contentMargins(.vertical, 5)
    }
}
```



5.13 CONTAINER RELATIVE FRAME

In iOS 17, we now have a new way to size views called container relative frame. This allows us to size views relative to the container they are inside. The great thing about Container Relative Frame is that it adapts well to different screen sizes. Also, you can replace a lot of use cases for GeometryReader with it.

For example, I want to frame an image and set its height to 33% of the available space



Using the container relative frame, you can specify the axis and length to determine the direction and size of the view relative to the container. For example, if I want to scale something vertically, I can set the axis to vertical and return the length divided by two. This means the view will take up 50% of the available space.

```
VStack(spacing: 0) {  
  Image(.cat1)  
    .resizable()  
    .scaledToFill()  
    .containerRelativeFrame(.vertical) { length, axis in  
      return length / 2  
    }  
    .clipped()  
  
  Color(white: 0.8)  
}
```

Example: ScrollView with Onboarding

Container Relative Frame becomes particularly useful when working with scroll views. Let's consider a scenario where we have a horizontal scroll view with three colors: blue, red, and yellow. By default, the scroll view applies a fixed size to its content, resulting in small color blocks. However, we can use Container container-relative frame with the axis set to horizontal to make each color block take up the entire screen.

```

struct ScrollColorExampleView: View {
    let colors = [Color.blue, Color.red, Color.yellow]
    var body: some View {
        ScrollView(.horizontal) {
            HStack {
                ForEach(colors, id: \.self) { color in
                    color
                    .containerRelativeFrame([.horizontal, .vertical])
                }
            }
        }
    }
}

```



Now, when we scroll horizontally, each color block fills the screen, providing a convenient and adaptive layout. This saves us from using a geometry reader or setting fixed frames for each color block.

Container Relative Frame can also be handy when creating **onboarding sequences or slideshows**. For example, you can use it to slide through a series of images.

Example: Container Relative Frame for Grid Layouts

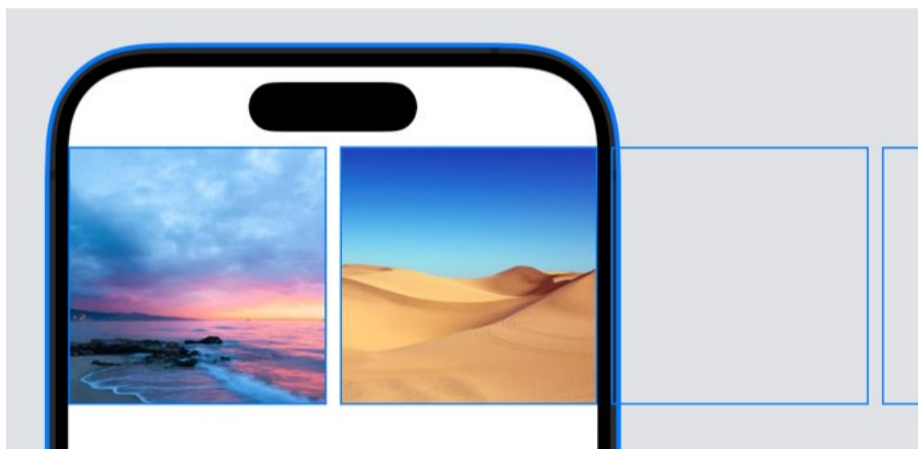
Additionally, Container Relative Frame has a parameter called count and span, which is useful for creating grid views. By specifying the count and span, you can control the number of items displayed in a stack.

```

let inspriations = NatureInspiration.examples()
let spacing: CGFloat = 10

ScrollView(.horizontal) {
    LazyHStack(spacing: spacing) {
        ForEach(inspriations) { inspiration in
            ImageAspectRatio(imageName: inspiration.imageName,
                             frameAspectRatio: 1)
            .containerRelativeFrame(.horizontal,
                                   count: 2
                                   spacing: spacing)
        }
    }
}

```



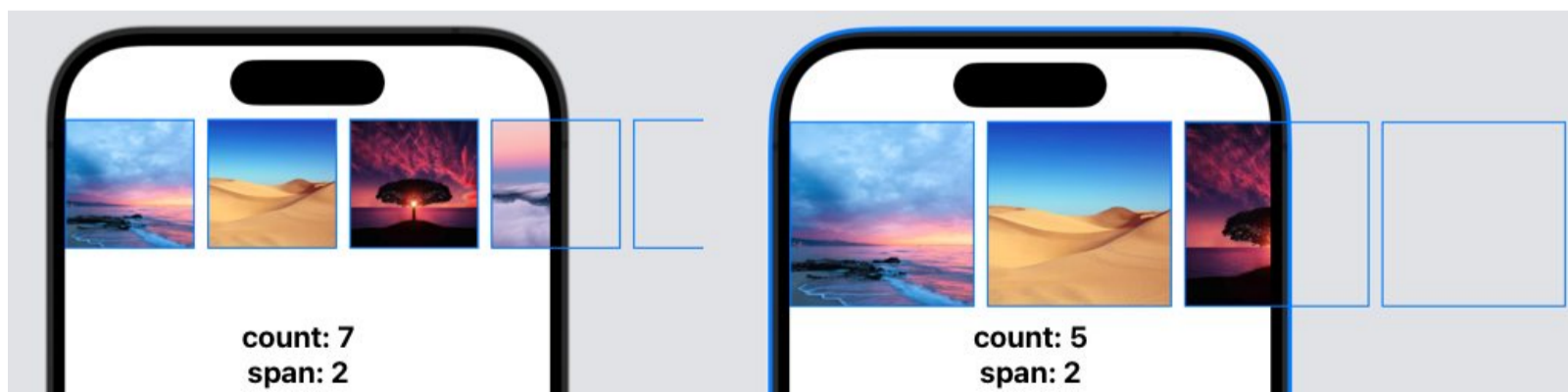
I used a scroll view with a horizontal axis and a ForEach loop to display a collection of images. To ensure the images maintain a square aspect ratio, I used a resizable **Image Aspect Ratio view**.

I set the container relative frame to 2 images with the count property. This allowed me to display exactly 2 images on the screen at once, regardless of the screen size. I could even adjust the span to show fractions of images, such as one and a half or two and a half. This flexibility ensures a consistent and adaptive layout across various devices.

```

.containerRelativeFrame(.horizontal,
                        count: 7,
                        span: 2,
                        spacing: spacing)

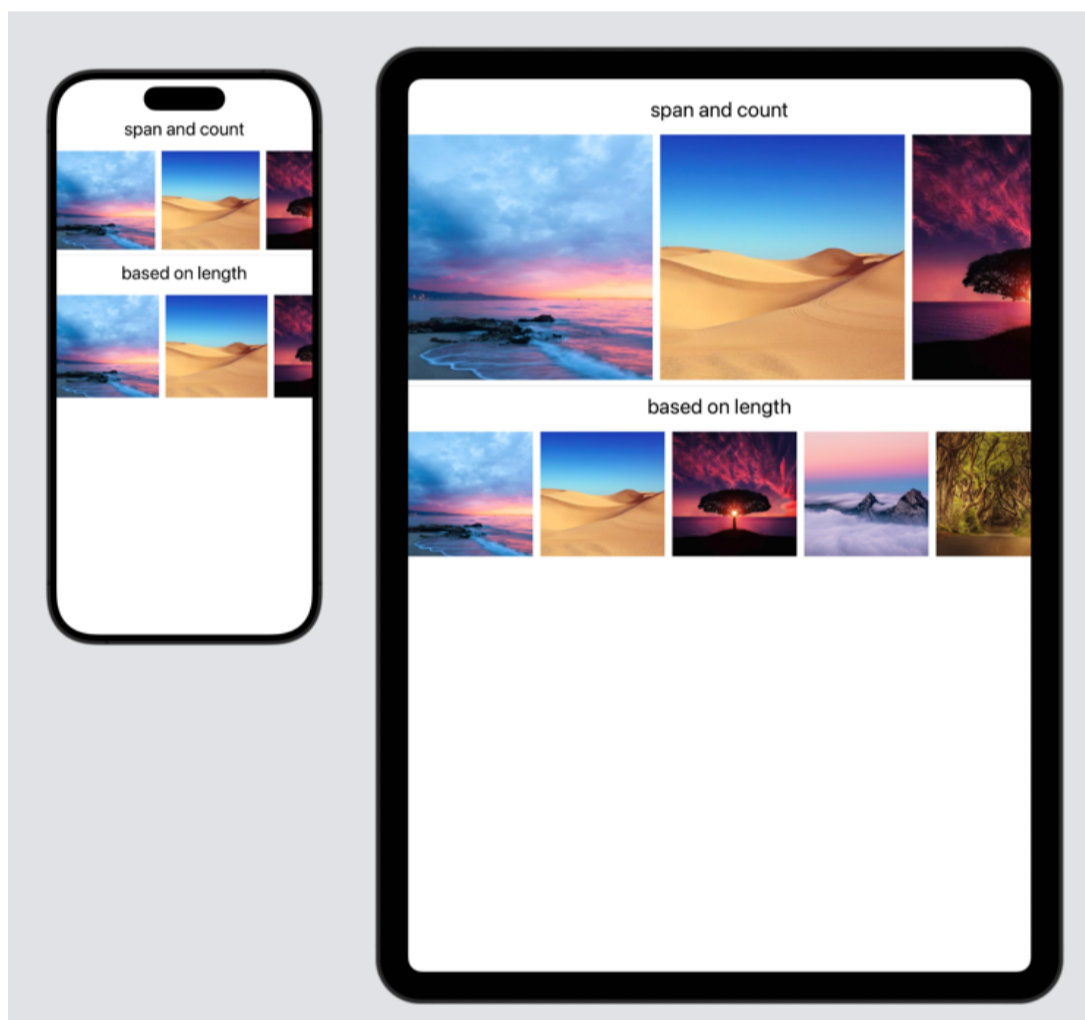
```



You can set more fine-grained frame rules with the container length:

```
.containerRelativeFrame(.horizontal, alignment: .leading) { length, axis in
    if axis == .horizontal {
        if length > 500 {
            return length * 0.20
        } else {
            return length * 0.40
        }
    } else {
        return length
    }
}
```

By checking the length and adjusting the return value, I could control the number of images displayed based on the screen width. This approach is particularly useful when you want to show more or fewer items on much larger screens like the iPad.



Container Relative Frame is a powerful addition to SwiftUI's layout capabilities. It allows for flexible and adaptive views without the need for fixed values. Whether you're working with scroll views or other container views like windows, columns, tabs, or lists, Container Relative Frame provides a neat solution. It's a valuable tool, especially when dealing with complex layouts like scroll views. However, if you don't require a scroll view, using flexible frames may suffice.

For more examples and in-depth explanations of Container Relative Frame, be sure to check out the scroll view section in this course.

5.14 CORNERRADIUS, CLIP AND MASK

In this section, we will explore how to manipulate the size and shape of views in SwiftUI. Specifically, we will focus on using corner radius, clip, and mask modifiers to achieve the desired effects.

Corner Radius

The simplest way to modify the shape of a view is by applying a corner radius. By using the `cornerRadius` modifier, you can round the corners of any view.

```
Text("Hello, World!")
    .padding()
    .background(Color.yellow)
    .cornerRadius(15)
```



Hello, World!

In this example, we apply a corner radius of 15 to a text view with a yellow background. You can also apply corner radius to other types of views, such as shapes or images.

Clip Shape

To achieve more complex shapes, we can use the `clipShape` modifier. This modifier allows us to clip a view to a specific shape. For instance, we can create a rounded rectangle or a circle:

```
Text("Hello, World!")
    .padding()
    .background(Color.yellow)
    .clipShape(RoundedRectangle(cornerRadius: 15))

Text("Hello, World!")
    .padding()
    .background(Color.yellow)
    .clipShape(Capsule())
```



Hello, World!



Hello, World!

```
Image(.cat1)
    .resizable()
    .scaledToFill()
    .frame(width: 100, height: 100, alignment: .center)
    .clipShape(Circle())
```

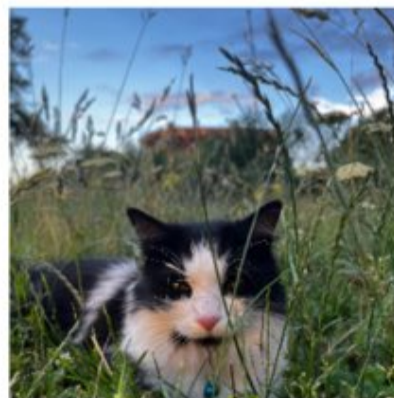




Clipped Views

Sometimes, views may overflow the layout area, causing unwanted effects. To ensure that a view stays within its designated area, we can use the clipped modifier. This modifier restricts the view to the layout area, cutting off any overflowing portions. For example:

```
Image(.cat1)
    .resizable()
    .scaledToFill()
    .frame(width: 100, height: 100, alignment: .center)
    .clipped()
```



In this case, the image is scaled up and **overflows the layout area**. By applying the clipped modifier, we ensure that only the dedicated layout area is visible.

Mask Modifier

For more advanced view manipulation, we can use the mask modifier. This modifier allows us to cut out a view using another view as a mask. You could use a gradient:

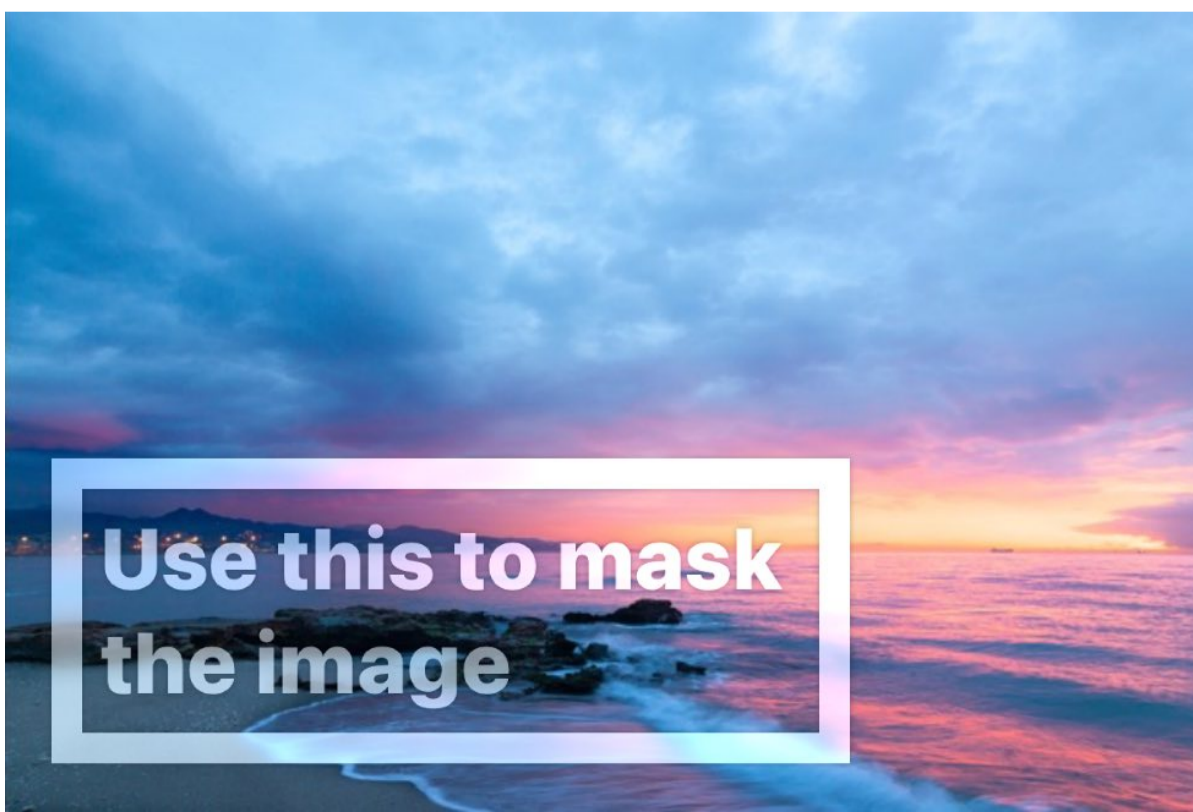
```
LinearGradient(gradient: Gradient(colors: [.pink, .indigo, .cyan]),
               startPoint: .top, endPoint: .bottom)
    .frame(height: 40)
    .mask {
        Text("Colorful Text")
            .font(.title)
            .fontWeight(.heavy)
    }
```



A more complex example with cutting out part of an image. I am using here a text with a border:

```
ZStack {
    Image(.beach)
        .resizable()
        .scaledToFit()

    Image(.beach)
        .resizable()
        .scaledToFit()
        .blur(radius: 3.0) // icing glass effect like materials
        .mask(alignment: .bottomLeading) {
            Text("Use this to mask the image")
                .font(.title)
                .fontWeight(.heavy)
                .padding()
                .border(Color.black, width: 10)
                .padding()
        }
        .opacity(0.8)
        .brightness(0.60)
        .shadow(radius: 1)
}
```



In this example, we use a linear gradient as a mask to create a **semi-transparent effect** on the image. We can also stack multiple views with different opacities and brightness levels to achieve interesting effects.

CHALLENGE 🙌 **SUPERHERO DETAIL VIEW**

Your goal is to create a layout that features the superhero image in the background. The image is a PNG with a transparent background. You can fill the entire area with a black gradient. The main content is a ScrollView with a title, biography, and quotes of the superhero data.

To achieve the desired layout, you will need to make use of alignment, padding, and material effects for the quotes. We also want to incorporate alternating styling with a bit of distance from the edges.

Main Component: ZStack

The main component of the layout is a ZStack. I place the background image first in the ZStack, and on top of it, we add a scroll view containing the text information. The image is scaled to fit and has a linear gradient applied to it, creating a shaded effect towards the top. To achieve the gradient background, I use a black color. I used 2 flexible frames to size the image and black gradient background.

```
struct MarvelView: View {
    let superHero: SuperHero
    var body: some View {
        ZStack(alignment: .bottom) {
            Image(superHero.imageName)
                .renderingMode(.template)
                .resizable()
                .scaledToFit()
                .foregroundStyle(LinearGradient(colors: [Color.white, Color.black],
                                                startPoint: .top,
                                                endPoint: .bottomLeading))
                .frame(maxWidth: .infinity, maxHeight: 600, alignment: .topTrailing)
                .frame(maxHeight: .infinity, alignment: .top)
                .background(Color.black.gradient)

            ScrollView {
                ...
            }
        }
    }
}
```

Scroll View: VStack and Offset

Within the scroll view, I use a `VStack` to arrange the different components vertically. To create the offset at the top, we add a large padding to the top area. This ensures that the text doesn't start right at the top of the scroll view. Instead of using the offset modifier, I simply add extra space at the top.

```
ScrollView {
  VStack(alignment: .leading, spacing: 10) {
    Text(superHero.name)
      .font(.largeTitle)
      .bold()

    Text("Biography")
      .bold()
      .padding(.top)
    Text(superHero.biography)

    Text("Quotes")
      .bold()
      .padding(.top)

    VStack(alignment: .leading) {
      ForEach ...
    }
  }
  .padding(.leading)
}
.foregroundColor(.white)
.padding()
.padding(.top, 200)
}
```

I also apply padding around everything within the scroll view to prevent the text from touching the edges.

Alignment and Quotes Styling

To properly align the title with the biographies, we use leading alignment within the `VStack`. For the quotes, we use an ultra-thin material as the background and add a larger padding of 50 to either the leading or trailing edge. The choice between leading or trailing edge depends on the index of the quote. To achieve this, I use the `enumerate` function on the quotes array, which provides us with the index. Based on whether the index is even or odd, we add the padding to the leading or trailing edge accordingly.

```
ForEach(Array(superHero.quotes.enumerated()), id: \.offset) { (index, quote) in
  Text(quote)
    .italic()
    .padding(15)
    .background(.ultraThinMaterial)
    .cornerRadius(5)
    .padding(index.isOdd ? .leading : .trailing, 50)
    .frame(maxWidth: .infinity,
           alignment: index.isOdd ? .trailing : .leading)
}
```

6. REUSABLE LAYOUT COMPONENTS

In this section, we will explore how to write reusable components in SwiftUI to effectively organize and structure your code. While this topic may not be directly related to layout, it plays a crucial role in enhancing code reusability and maintainability.

By organizing your views with **functions** and **subviews**, utilizing **view builders**, creating **custom modifiers**, and writing **custom container views**, you can enhance code reusability and improve the overall structure of your SwiftUI projects.

It is important to note that there is no right or wrong way to structure your code. The goal is to make it easily understandable and manageable for yourself and future developers. Experiment with different approaches and find what works best for your specific requirements.

6.1 MAKING YOUR SWIFTUI VIEWS MORE REUSABLE

In order to organize and work easier with your code, you might want to shorten your views and reorganise them into smaller more manageable chunks. Let's use the following as a simple example:

```
struct ReusableComponentsExampleView: View {
    let isLoggedIn: Bool = false

    var body: some View {
        VStack(alignment: .leading) {
            Text("How to organize and structure your code")
                .font(.title)

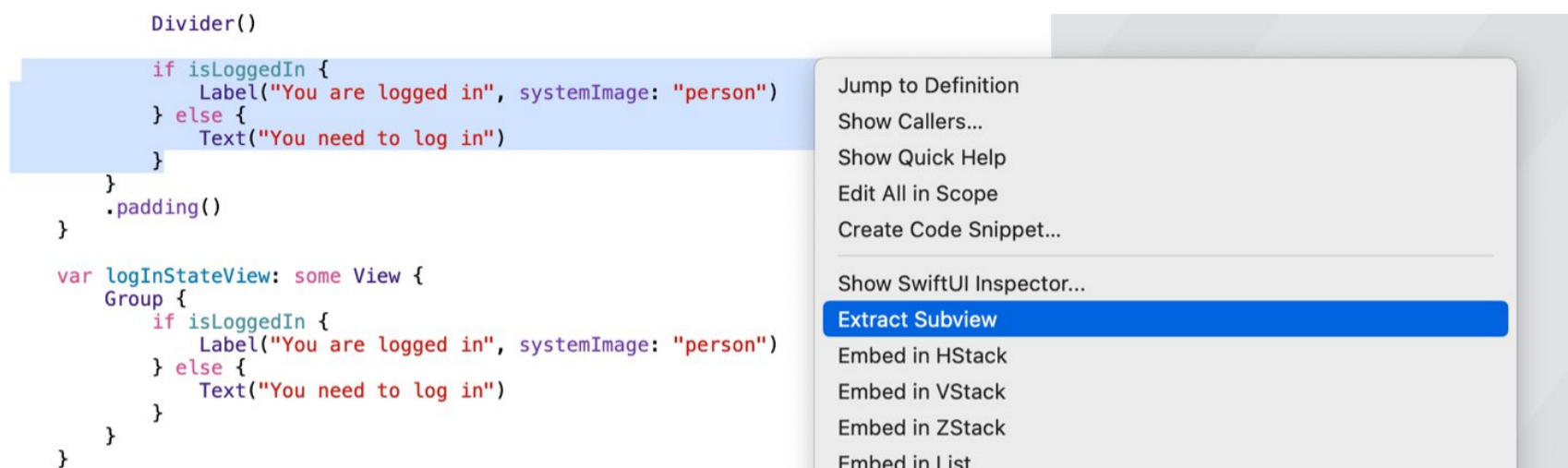
            Divider()

            if isLoggedIn {
                Label("You are logged in", systemImage: "person")
            } else {
                Text("You need to log in")
            }
        }
        .padding()
    }
}
```

The if-else statement makes it more complex and I want to extract this into its separate entity.

View Organization with Subviews

Typically, you would extract subviews by selecting the desired code and creating a new view. You can then pass any necessary properties to the subview, making it more flexible and reusable.



```
struct MySubView: View {
    let isLoggedIn: Bool

    var body: some View {
        if isLoggedIn {
            Label("You are logged in", systemImage: "person")
        } else {
            Text("You need to log in")
        }
    }
}
```

Utilizing View Builders

In SwiftUI, you often make use of `@ViewBuilder` with closures that return SwiftUI Views. `@ViewBuilder` is a parameter attribute that allows to return of multiple views from a closure or for closures with conditionals. It can be used for computed properties, functions, or container views with view passing closures.

For instance, you can create custom functions or computed properties that return a **some View** type. By using the `@ViewBuilder` attribute, you can return multiple views within the closure. This allows for more flexibility and cleaner code:

```
@ViewBuilder
func createLoginInfo() -> some View {
    Label("You are logged in", systemImage: "person")
    Text("Take advantage of pro features")
}
```

Extracting Views as Computed Properties

If I continue with the example from above and extract the conditional statement as a computed property:

```

struct ReusableComponentsExampleView: View {
    let isLoggedIn: Bool = false

    var body: some View {
        VStack(alignment: .leading) {
            Text("How to organize and structure your code")
                .font(.title)

            Divider()
            logInStateView
        }
        .padding()
    }

    @ViewBuilder
    var logInStateView: some View {
        if isLoggedIn {
            Label("You are logged in", systemImage: "person")
        } else {
            Text("You need to log in")
        }
    }
}

```

I have to add the @ViewBuilder in front of the computed property because I am returning 2 different views from the conditional.

Alternatively to @ViewBuilder you can also embed the conditional in a Group. This will return a single view Group:

```

var logInStateView: some View {
    Group {
        if isLoggedIn {
            Label("You are logged in", systemImage: "person")
        } else {
            Text("You need to log in")
        }
    }
}

```

Extracting Views as Functions

Similar to computed properties you can also write functions that generate views. The same view from above could be using instead a function:

```

struct ReusableComponentsExampleView: View {
    let isLoggedIn: Bool = false

    var body: some View {
        VStack(alignment: .leading) {
            Text("How to organize and structure your code")
                .font(.title)

            Divider()

```



```

        loginStateView()
    }
    .padding()
}

@ViewBuilder
var func loginStateView() -> some View {
    if isLoggedIn {
        Label("You are logged in", systemImage: "person")
    } else {
        Text("You need to log in")
    }
}
}

```

6.2 REUSABLE VIEW MODIFIERS

View modifiers are a powerful tool in SwiftUI that allows you to apply various modifications to your views. While SwiftUI provides a range of built-in modifiers, you can also create your own custom modifiers to encapsulate specific sets of modifications.

By creating custom modifiers, you can keep related modifications together and apply them easily to different parts of your app. This promotes code reusability and simplifies the process of applying consistent styling or behavior across your views.

Example 1: Text Styling

Let's start with a simple example of text styling. Imagine you have a box with some text inside, and you want to apply multiple view modifiers such as foreground color, bold font, padding, and background color. Initially, you might end up cluttering your code by adding these modifiers individually to each view.

```

Text("Hello, World!")
    .foregroundColor(.white)
    .bold()
    .padding()
    .background(Color.cyan)

```

To avoid this repetition and make your code more reusable, you can create your own view modifier. Since we want to create a view modifier, we won't conform to the View protocol. Instead, we'll conform to a different protocol specifically designed for view modifiers.

```

struct BoxViewModifier: ViewModifier {

    func body(content: Content) -> some View {
        content
            .foregroundColor(.white)
            .bold()
            .padding()
            .background(Color.pink)
    }
}

```

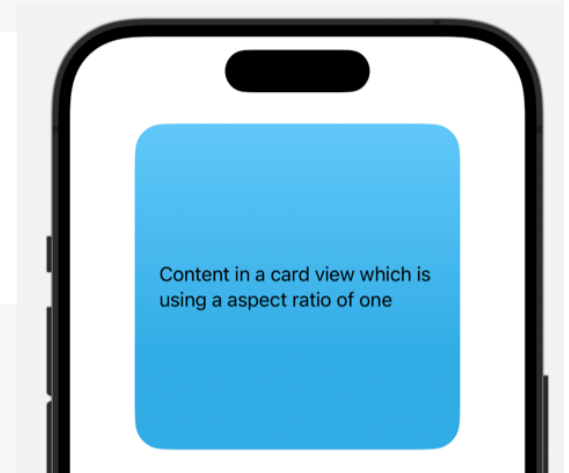
You can then apply this custom ViewModifier like so:

```
Text("Hello, World!")
    .modifier(BoxViewModifier())
```

A shorter form that would be more SwiftUI-like is to write a function to view. View modifiers in SwiftUI are defined as functions that modify the view itself and return some View:

```
extension View {
    func boxStyling() -> some View {
        self.modifier(BoxViewModifier())
    }
}

Text("Hello, World!")
    .boxStyling()
```



Instead of a separate ViewModifier I could have instead directly added all the view modifiers like so:

```
extension View {
    func boxStyling() -> some View {
        self.foregroundColor(.white)
            .bold()
            .padding(hori == .compact ? 10 : 30)
            .background(Color.pink)
    }
}
```

The advantage of custom ViewModifiers is that they can own state and access environment properties. For example, if I wanted to change the padding depending on the horizontal size class, I could accomplish this with a custom view modifier like so:

```
struct BoxViewModifier: ViewModifier {
    @Environment(\.horizontalSizeClass) var hori

    func body(content: Content) -> some View {
        content
            .foregroundColor(.white)
            .bold()
            .padding(hori == .compact ? 10 : 30)
            .background(Color.pink)
    }
}
```

Example 2: Square Card Box Modifier

I want to make the following behavior for a squad card reusable:

```
ZStack {
    RoundedRectangle(cornerRadius: 25.0)
        .fill(Color.cyan.gradient)
    Text("Content in a card view which is using
        a aspect ratio of one")
        .padding()
}
.aspectRatio(1, contentMode: .fit)
.padding()
```

I am creating a function for Text view, but you can also make this work for all views by writing an extension to view. In this example self refers to the text view itself:

```
extension Text {
    func squareBoxStyling() -> some View {
        ZStack {
            RoundedRectangle(cornerRadius: 25.0)
                .fill(Color.cyan.gradient)
            self.padding()
        }
        .aspectRatio(1, contentMode: .fit)
    }
}
```

And I can use the modifier to any Text view:

```
Text("Content in a card view which is using a aspect ratio of one")
    .squareBoxStyling()
    .padding()
```

Example 3: Image Resizing

Often times you can choose between a custom modifier and a subview. In a [previous section](#), I introduced the resizing image view:

```
struct ImageAspectRatioView: View {

    let imageName: String
    let frameAspectRatio: CGFloat

    var body: some View {
        Color.cyan // Placeholder
            .aspectRatio(frameAspectRatio, contentMode: .fit)
            .overlay {
                Image(imageName)
                    .resizable()
                    .aspectRatio(nil, contentMode: .fill)
            }
    }
}
```

```

        .clipped()
    }
}

```

I could achieve the same result with a modifier function:

```

extension Image {
    func resizeToFit(frameAspectRatio: CGFloat,
                    cornerRadius: CGFloat = 0) -> some View {
        Color.cyan
            .aspectRatio(frameAspectRatio, contentMode: .fit)
            .overlay {
                self.resizable()
                    .aspectRatio(nil, contentMode: .fill)
            }
            .clipped()
            .cornerRadius(cornerRadius)
    }
}

```

I am modifying an image and need to use the resizable modifier which is only allowed for images. Therefore the function is attached in an extension to Image.

With the help of this image modifier, my resizing functionality becomes:

```

Image("fish_3")
    .resizeToFit(frameAspectRatio: 1)

```

6.3 BUTTONSTYLE

In SwiftUI, you can make view modifiers reusable by using custom view modifiers, such as card styles. Additionally, some system-provided views like button toggles, sliders, lists, or forms have special modifiers that only apply to their specific use case. For example, a list styling modifier for lists or a button style for buttons.

In some cases, these system-provided views expose the protocol behind them, allowing you to write your own styling modifiers. This is true for buttons and toggles, but not for all views like pickers, menus, or lists. However, it can be quite advantageous. Let me give you an example:

```

struct CustomButtonStyle: ButtonStyle {
    @Environment(\.isEnabled) var isEnabled

    func makeBody(configuration: Configuration) -> some View {
        configuration.label
            .foregroundColor(.white)
            .padding(.horizontal, 10)
            .padding(.vertical, 5)
            .frame(maxWidth: 400)
            .background {
                RoundedRectangle(cornerRadius: 5)
                    .fill(configuration.role == .destructive ? .pink : .accent)
            }
    }
}

```

```

        .shadow(radius: shadow(for: configuration.isPressed))
    }
    .scaleEffect(CGSize(width: configuration.isPressed ? 0.9 : 1,
                        height: configuration.isPressed ? 0.9 : 1))
    .saturation(isEnabled ? 1 : 0)
}

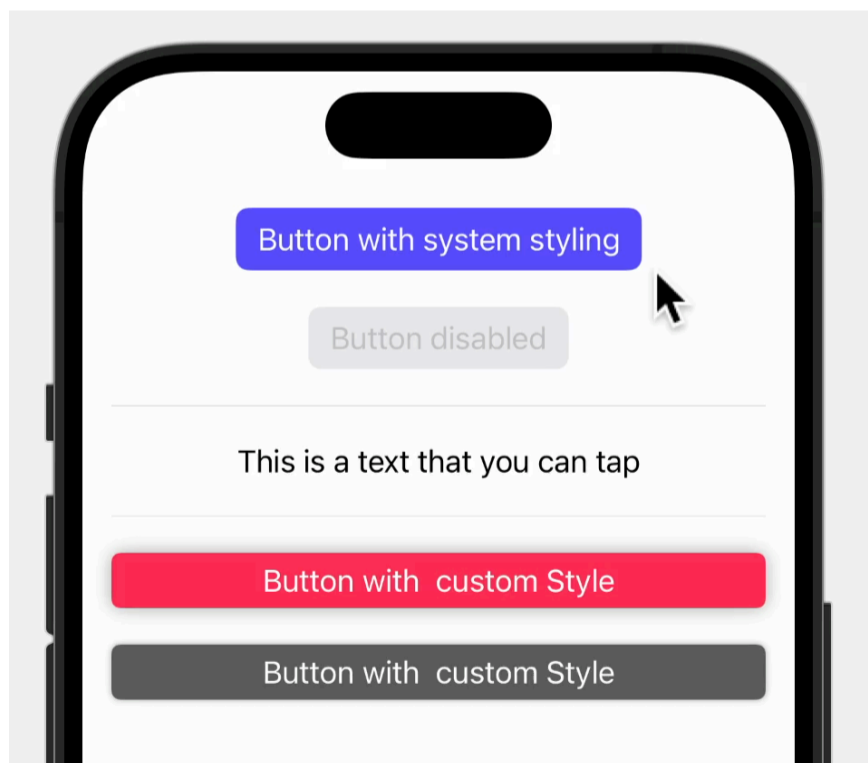
func shadow(for isPressed: Bool) -> CGFloat {
    guard isEnabled else { return 2 }
    if isPressed {
        return 0
    } else {
        return 5
    }
}
}
}

```

The button configuration has 3 main parameters:

- the **label** which is the display text for the button
- **isPressed**: a boolean that is true when the user presses or holds the button. Use this property to animate the button interaction
- **role**: the role of the button that can be default, destructive, or cancel.

Additionally, I used the **isEnabled** property from the environment to make the button greyed out when it is disabled.



When you test this in live preview and tap the button, it will scale down slightly and provide a press animation. This feedback is important for users to understand that they have pressed something. If you use an onTap gesture instead, you would not be able to add this important animation.

You can then use this styling for your buttons:

```

Button(role: .destructive, action: {

}, label: {
    Text("Button with custom Style")
})
.buttonStyle(CustomButtonStyle())

Button(role: .destructive, action: {

}, label: {
    Text("Button with custom Style")
})
.buttonStyle(CustomButtonStyle())
.disabled(true)

```

6.4 CUSTOM CONTAINER VIEWS

When you come across good strategies, such as adapting for different screen sizes or device orientations, you may want to make them more reusable. By writing your own containers that handle this logic, you can create highly generic components that can be used in different parts of your app.

Example 1: Replicating a Button

Now, we will take it a step further and create a custom container view that is more generic. To illustrate this, we will start by replicating a button.

Note: While replicating a button may not be the most practical use case for your projects, it serves as a valuable exercise in understanding how SwiftUI internally handles such scenarios.

To begin, let's look at the definition of SwiftUI button:

```

@available(iOS 13.0, macOS 10.15, tvOS 13.0, watchOS 6.0, *)
public struct Button<Label> : View where Label : View {

    /// Creates a button that displays a custom label.
    ///
    /// - Parameters:
    ///   - action: The action to perform when the user triggers the button.
    ///   - label: A view that describes the purpose of the button's `action`.
    public init(action: @escaping () -> Void, @ViewBuilder label: () -> Label)

    /// The content and behavior of the view.

    @MainActor public var body: some View { get }

    /// The type of view representing the body of this view.

    public typealias Body = some View
}

```

Button has 2 parameters for the action closure and the label closure. It uses a Generic type **Label** that conforms to the view protocol. This allows to use any view as the label of a button:

```

Button(action: {
    // do something
}, label: {
    Text("Button") // the label is a Text type
})

Button(action: {
    // do something
}, label: {
    Image(systemName: "gear") // the label is a Image type
})

Button(action: {
    // do something
}, label: {
    Label("Button", systemImage: "gear") // the label is a Label type
})

```

Replicating the system Button behavior with a similar initializer:

```

struct CustomButton<Label> : View where Label : View {

    let action: () -> Void
    let label: () -> Label

    init(action: @escaping () -> Void, @ViewBuilder label: () -> Label) {
        self.action = action
        self.label = label
    }

    var body: some View {
        VStack {
            label()
        }
        .foregroundColor(.accent)
        .onTapGesture {
            action()
        }
    }
}

```

In this example, we define a CustomButtonView struct conforming to the View protocol. It has two properties: action, which is a closure representing the button's action, and label, which is a closure returning a view representing the button's label.

By utilizing the @ViewBuilder parameter attribute, we can return multiple views within the label closure. This allows for greater flexibility.

Example 2: Creating a Custom Container View

Now that we understand how to create a custom button, let's explore a custom container view example. I will reimagine the ViewModifier we previously created, called BoxViewModifier, as a container view. It replicates a VStack with a custom styling:

```

struct CardContainer<Content> : View where Content : View {

    let content: () -> Content

    init(@ViewBuilder content: @escaping () -> Content) {
        self.content = content
    }

    var body: some View {
        ZStack {
            RoundedRectangle(cornerRadius: 25.0)
                .fill(Color.cyan.gradient)

            VStack(alignment: .center,
                  spacing: 10,
                  content: content)
                .padding()
        }
        .aspectRatio(1, contentMode: .fit)
    }
}

```

In this case, I define a CardContainerView struct conforming to the View protocol. It takes a generic Content type, which represents the views to be contained within the container.

By utilizing the @ViewBuilder attribute, you can pass multiple views as the content parameter:

```

CardContainer {
    Text("Hello, World!")
    Text("second")
}

```

6.5 CUSTOM CONTAINERS WITH DYNAMIC DATA

Additionally, you can create container views that work with dynamic data, such as displaying an array of items using a ForEach or a List. While this involves more advanced concepts and generics, it provides a blueprint for creating reusable container views tailored to your specific needs.

I will use the following list view as an example, where I have an array of my NatureInspiration data. List takes the array and for each element ask what view to show:

```

struct ContentView: View {

    let inspirations = NatureInspiration.examples()

    var body: some View {
        GeometryReader { geometry in
            List(inspirations) { inspiration in
                Image(inspiration.imageName)
                    .resizable()
                    .scaledToFill()
                    .frame(width: geometry.size.width,
                          height: geometry.size.width)
            }
        }
    }
}

```



```

        .clipped()
        .listRowSeparator(.hidden)
        .listRowInsets(.init(top: 0, leading: 0, bottom: 0, trailing: 0))
    }
    .listStyle(.plain)
}
}
}
}
}

```

I am using GeometryReader, List, and a few list customizers that make this view look more complex. I would like to make this behavior more reusable and create a custom container view that holds most of the logic.



To understand how to pass dynamic data, let's look at this initializer of List:

```

extension List {
    @MainActor public init<Data, RowContent>(_ data: Data, @ViewBuilder rowContent:
    @escaping (Data.Element) -> RowContent) where Content == ForEach<Data,
    Data.Element.ID, RowContent>, Data : RandomAccessCollection, RowContent : View,
    Data.Element : Identifiable
}

```

There is a lot of Generics and Protocol conformance involved. Here is a breakdown:

- Data conforming to **RandomAccessCollection**. This is the array of data we are passing to the List. List wants a collection and each element can be accessed very quickly by the index. This helps SwiftUI performance and allows for smooth scrolling.
- Data.Element conforming to **Identifiable**. Each element in the Data collection should conform to Identifiable.

- RowContent conforming to **View**. For each element in the data collection we have to return a RowContent that is a view and can be displayed on screen.

If I use the same generics and protocol requirements for a custom view, this could look like so:

```
struct CustomContainer<Data, RowContent>: View where Data: RandomAccessCollection,
    Data.Element: Identifiable, RowContent: View {

    private let data: [Data.Element]
    private let rowContent: (Data.Element) -> RowContent

    init(_ data: Data,
         @ViewBuilder rowContent: @escaping (Data.Element) -> RowContent) {
        self.data = data.map({ $0 })
        self.rowContent = rowContent
    }

    var body: some View {
        GeometryReader { geometry in
            List(data) { element in
                rowContent(element)
                    .frame(width: geometry.size.width,
                          height: geometry.size.width)
                    .clipped()
                    .listRowSeparator(.hidden)
                    .listRowInsets(.init(top: 0, leading: 0, bottom: 0, trailing: 0))
            }
            .listStyle(.plain)
        }
    }
}
```

In this custom container, we have two generic parameters: Data and RowContent. Data represents the array of data points we want to pass in, and RowContent represents the view that will be displayed for each data element.

To use our custom container view, we can simply pass in the data and the view that is shown for each data point. For example, we can replace the List view in our previous example with the CustomContainer view.

```
struct ContentView: View {
    let inspirations = NatureInspiration.examples()

    var body: some View {
        CustomContainer(inspirations) { inspiration in
            Image(inspiration.imageName)
                .resizable()
                .scaledToFill()
        }
    }
}
```

7. CUSTOM LAYOUT

In this section, I will show you how to create custom layouts in SwiftUI that cater to your specific needs. So far, I have demonstrated the use of built-in containers like stacks and grids. However, there are situations where you require more control and flexibility.

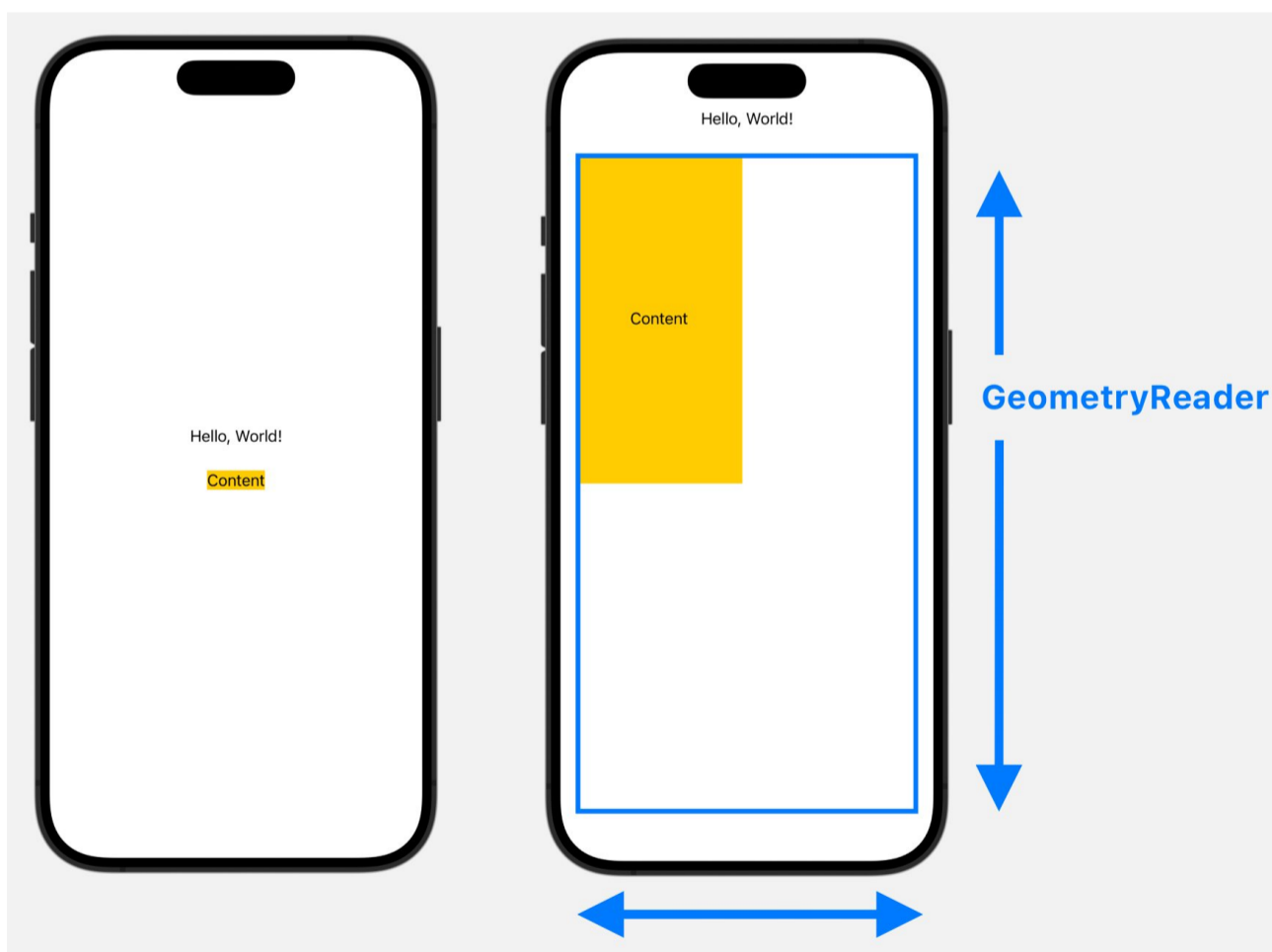
7.2 GEOMETRYREADER

The first option I want to discuss is the GeometryReader. With this, you can obtain the size and area of a view and make decisions on how to display and size its contents.

If you look at this simple example:

```
VStack {  
    Text("Hello, World!")  
    Text("Content")  
        .background(Color.yellow)  
}
```

And add a GeometryReader around the “Content” text. GeometryReader provides us with a closure where we can access the **geometry proxy** that contains information about the available space. For example, you can use the **geometry.size.width** and **height** property to set the frame around the text:



```

VStack {
  Text("Hello, World!")
  GeometryReader(content: { geometry in
    Text("Content")
      .frame(width: geometry.size.width / 2,
             height: geometry.size.height / 2)
      .background(Color.yellow)
  })
  .border(Color.blue, width: 5)
  .padding()
}

```

When we add the GeometryReader to our view, we can see that it changes the layout significantly. By default, the GeometryReader is a greedy view, meaning it takes up most of the available space. This can cause issues when we want to change the size of the view it wraps without affecting other views.

The GeometryReader will place all its child views at the top leading edge. You can use layout components inside the GeometryReader like HStack and VStack to distribute views.

```

struct GeometryReaderExampleView: View {
  let spacing: CGFloat = 10

  var body: some View {
    GeometryReader(content: { geometry in
      VStack(spacing: 0) {
        HStack(spacing: 10) {
          Color.yellow
            .frame(width: (geometry.size.width - 10) / 2,
                  height: geometry.size.height / 2)
          Color.orange
            .frame(width: (geometry.size.width - 10) / 2,
                  height: geometry.size.height / 2)
        }

        Image("dog_3")
          .resizable()
          .scaledToFill()
          .frame(height: geometry.size.height / 2)
          .clipped()
      }
    })

    .border(Color.blue, width: 5)
    .padding()
  }
}

```

The advantage of GeometryReader is that I can size the views depending on the available space. This allows me to adjust the layout for different screen sizes and device orientations:



By using the GeometryReader in combination with other views, we can create dynamic and flexible layouts. For example, we can create a grid layout by adding multiple views with different images and adjusting their sizes based on the available space.

GeometryReaderProxy

The GeometryProxy object contains properties such as `size`, `safeAreaInsets`, and `frame(in:)`. These properties allow you to access and manipulate the geometry of the container view.

Here are some commonly used properties:

- **geometry.size**: The size of the GeometryReader area including the width and height.
- **geometry.frame(in: .global)**: The frame of the container in the global coordinate space. The global coordinate space refers to the coordinate system of the entire screen. This means that the origin (0,0) is at the top-left corner of the screen, and the coordinates increase as you move down and to the right.
- **geometry.frame(in: .local)**: The local coordinate space refers to the coordinate system of the GeometryReader itself. This means that the origin (0,0) is at the top-left corner of the GeometryReader and the coordinates increase as you move down and to the right.
- **geometry.frame(in: .named(_: String))**: The frame in a specific coordinate space.

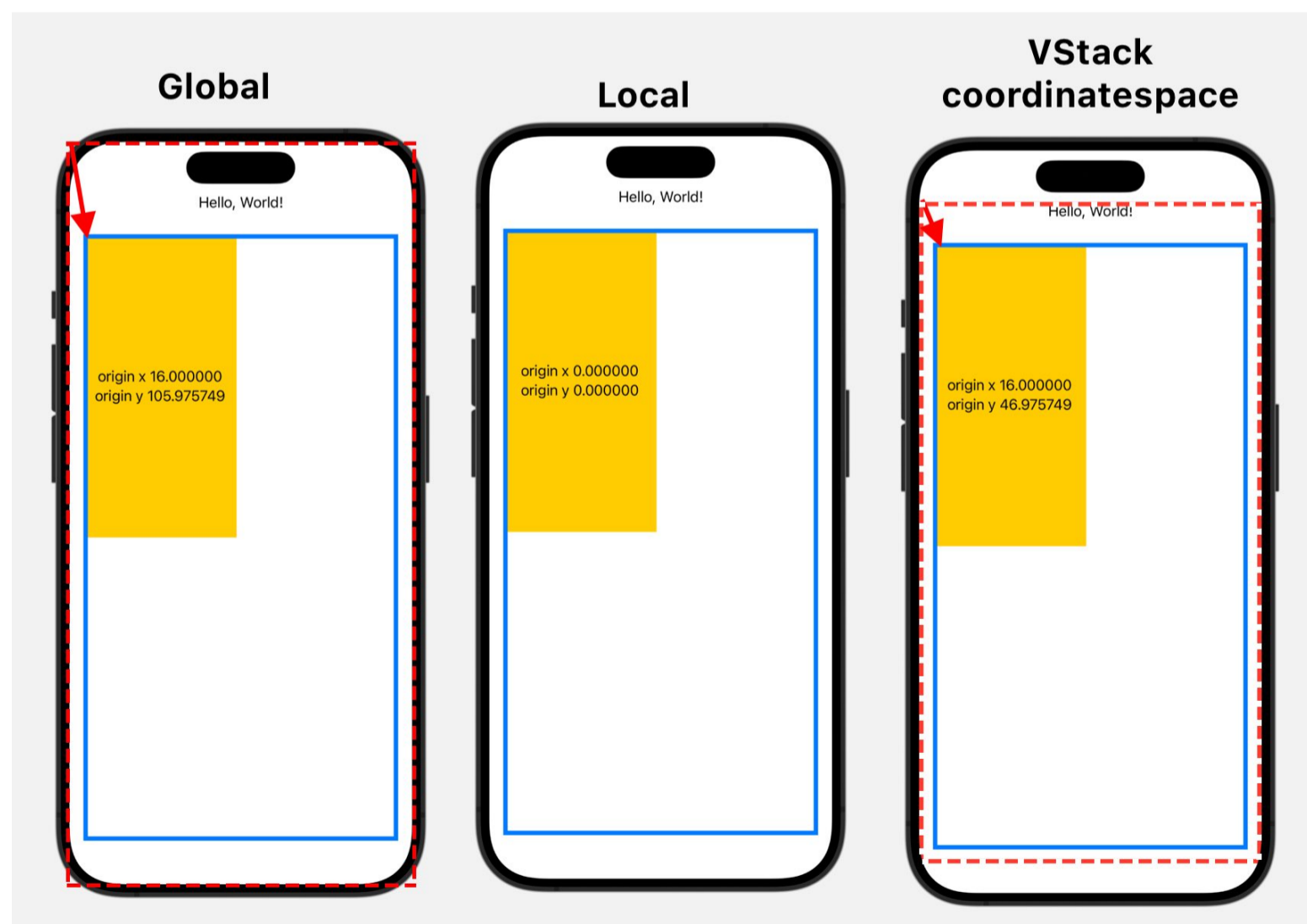
As an example let's show the min x and min y coordinates with a Text:

```
struct GeometryReaderExampleView: View {
    let namespace: String = "namespace"

    var body: some View {
        VStack {
            Text("Hello, World!")

            GeometryReader(content: { geometry in
                VStack {
                    Text("origin x \ \(geometry.frame(in: .named(namespace)).minX)")
                    Text("origin y \ \(geometry.frame(in: .named(namespace)).minY)")
                }
                .frame(width: (geometry.size.width - 10) / 2,
                    height: geometry.size.height / 2)
                .background(Color.yellow)
            })
            .border(Color.blue, width: 5)
            .padding()
        }
        .coordinateSpace(name: namespace)
    }
}
```

By setting the coordinateSpace name around the VStack and asking for the frame in reference to this coordinate system, I can get the distance from the organ of the VStack to the beginning of the GeometryReader. In this example the GeometryReader is 16 points in the x direction and 47 points in the y direction:



How not to use GeometryReader

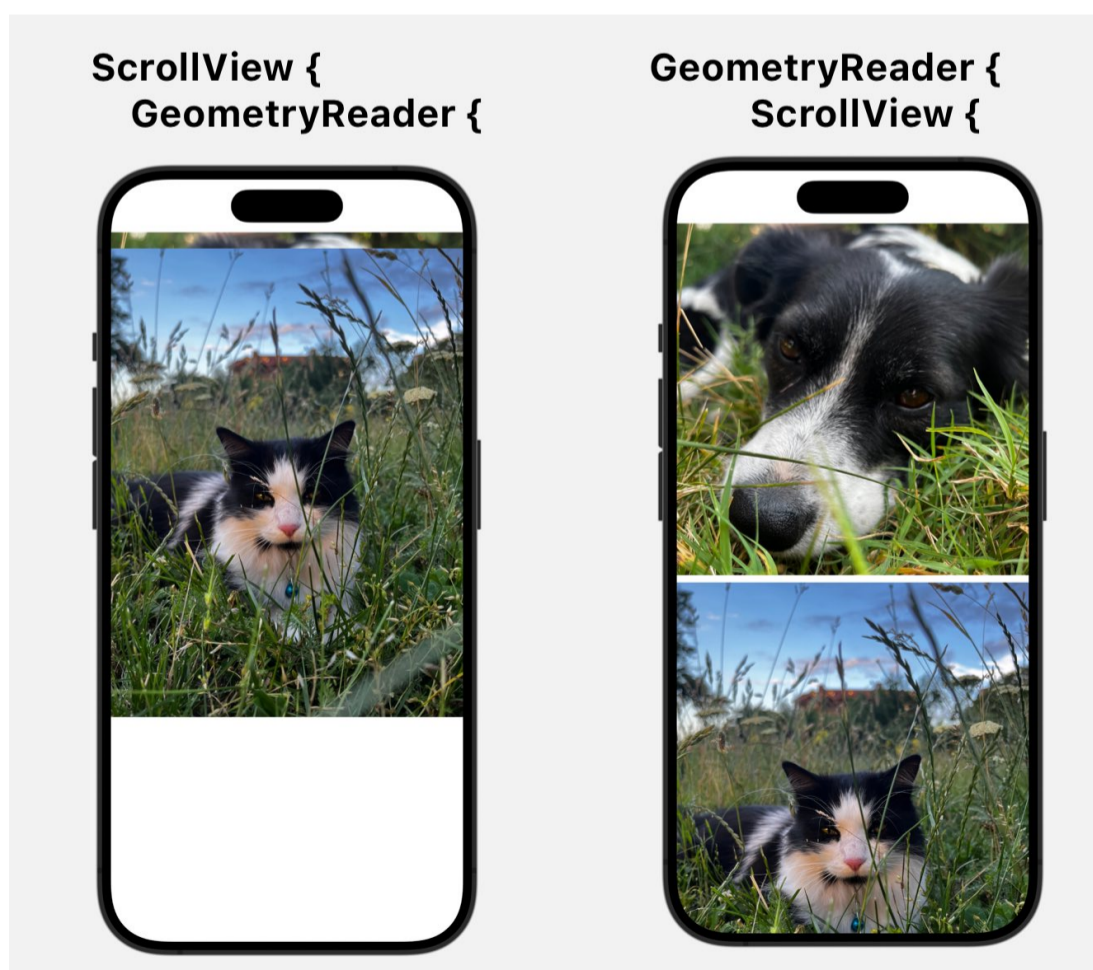
However, it's important to note that the GeometryReader can sometimes cause layout issues, especially when used in combination with scroll views or lists.

For example, I am placing a GeometryReader inside a ScrollView to size an image:

```
struct BadGeometryReaderExampleView: View {
    var body: some View {
        ScrollView {
            GeometryReader(content: { geometry in
                Image("dog_3")
                    .resizable()
                    .scaledToFill()
                    .frame(width: geometry.size.width,
                        height: geometry.size.width)
                    .clipped()
            })

            Image("cat_1")
                .resizable()
                .scaledToFit()
        }
    }
}
```

ScrollView will use the ideal size of its children in the scroll direction. GeometryReader does not have an ideal size and the fallback value of 10 points is used for its height. In the below screenshot, you can see that on the left the first image is very small because of its 10 height. On the right you can see that it looks much better by using the GeometryReader outside the ScrollView:



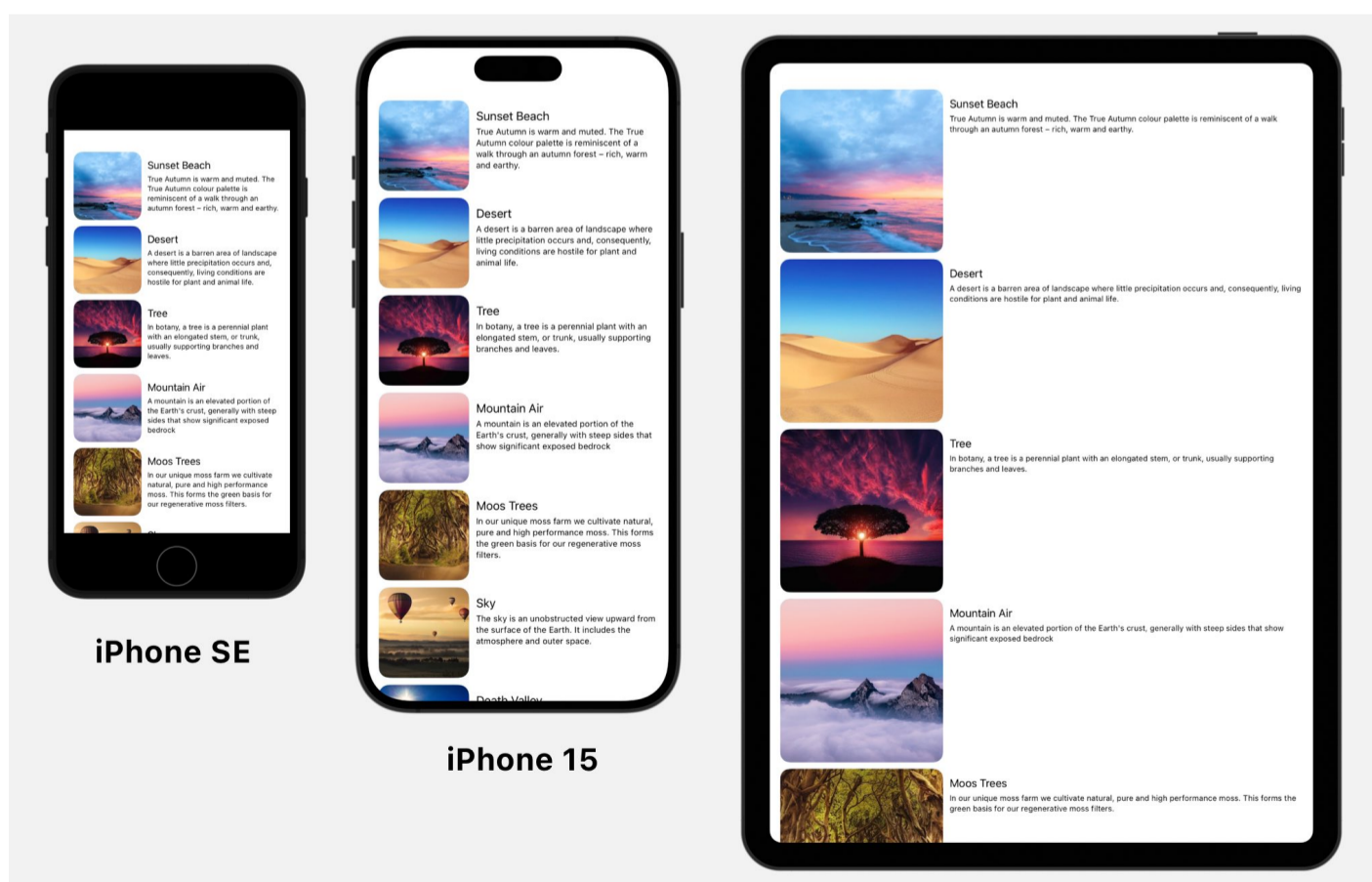
In most cases, when using a ScrollView with a GeometryReader, it's better to place the GeometryReader outside the scroll view. This way, the GeometryReader can stretch to fit the available space. The scroll view is also a greedy view and will fill out the whole GeometryReader content area:

```
struct GeometryReaderExampleView: View {
    var body: some View {
        GeometryReader(content: { geometry in
            ScrollView {
                Image("dog_3")
                    .resizable()
                    .scaledToFill()
                    .frame(width: geometry.size.width,
                        height: geometry.size.width)
                    .clipped()

                Image("cat_1")
                    .resizable()
                    .scaledToFit()
            }
        })
    }
}
```

Example: GeometryReader for List Row Sizing

We will explore how to create custom layouts in SwiftUI. We will create a list view with images, titles, and descriptions. The challenge is to make the images scale with the screen size and always occupy one-third of the width.



To ensure that the images always occupy 30 percent of the width, I used the GeometryReader and multiplied its width by 30%. This way, the images dynamically adjust to different screen sizes. You can test this on smaller screens like an iPhone SE.

```
struct GeometryReaderInspirationListView: View {
    let inspirations = NatureInspiration.examples()

    var body: some View {
        GeometryReader(content: { geometry in
            ScrollView {
                LazyVStack(alignment: .leading, spacing: 10, content: {
                    ForEach(inspirations) { inspiration in

                        HStack(alignment: .top, spacing: 10) {
                            ImageAspectView(imageName: inspiration.imageName,
                                frameAspectRatio: 1,
                                cornerRadius: 15)
                                .frame(width: geometry.size.width * 0.3)

                            VStack(alignment: .leading, spacing: 5) {
                                Text(inspection.name)
                                Text(inspection.description)
                                    .font(.caption)
                                    .lineLimit(4)
                            }
                                .padding(.vertical, 12)
                        }
                    }
                })
            })
        })
    }
}
```

The alternative solution is quite similar. Instead of changing the frame, you would set the `containerRelativeFrame` in the horizontal direction to scale it accordingly. In the following, I set the size of the image to 30% of the enclosing frame:

```
HStack(alignment: .top, spacing: 10) {
    ImageAspectView(imageName: inspiration.imageName,
        frameAspectRatio: 1,
        cornerRadius: 15)
        .containerRelativeFrame(.horizontal) { length, axis in
            length * 0.3
        }

    VStack(alignment: .leading, spacing: 5) {
        Text(inspection.name)
        Text(inspection.description)
            .font(.caption)
    }
        .padding(.vertical, 12)
}
```

Example: Adaptable Layout with LazyVGrid

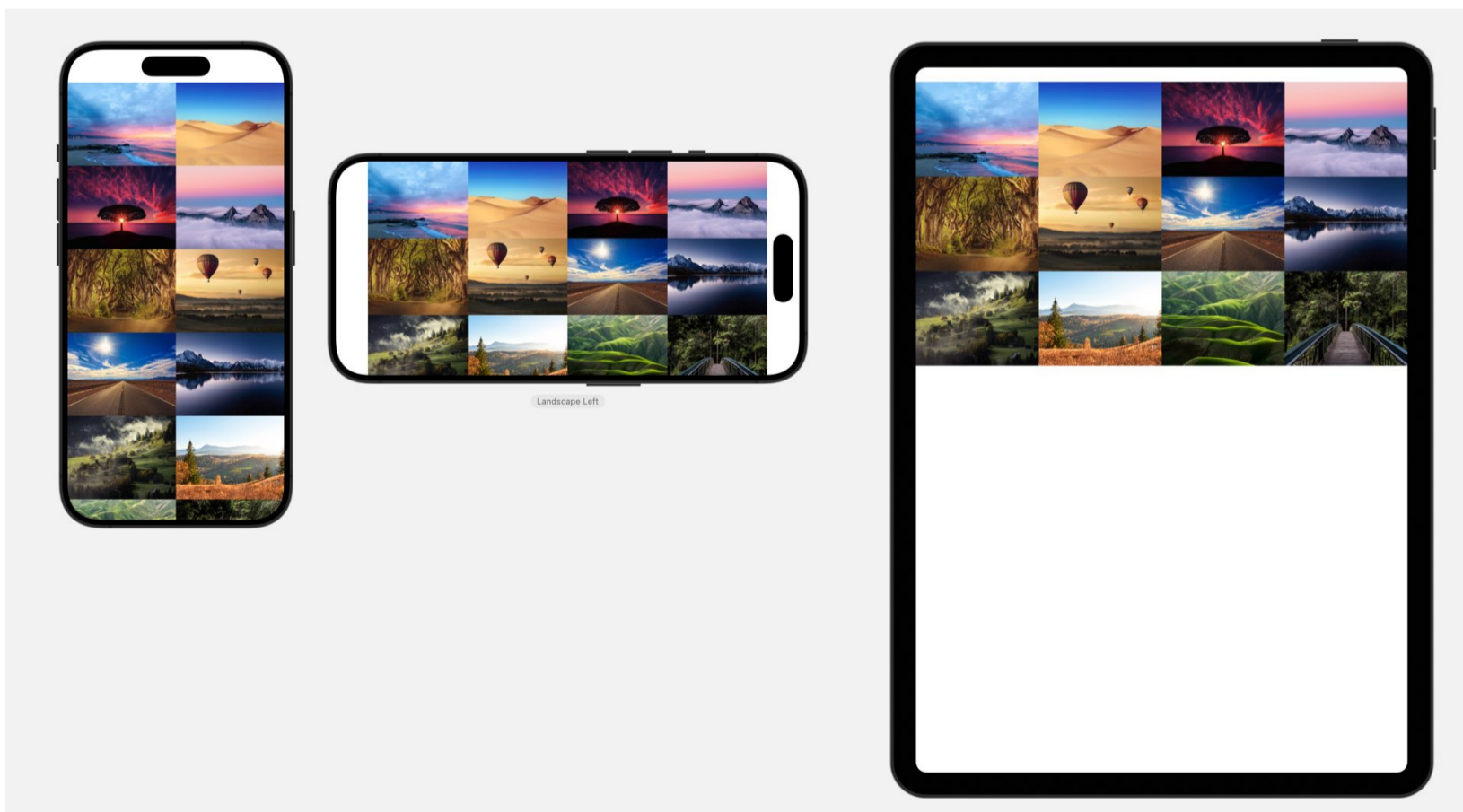
Let's explore an example that demonstrates how to use LazyVGrid in combination with GeometryReader to create a flexible grid layout. This example will showcase how you can customize the number of columns based on the available screen width.

We will use an array of images as our data source for the grid:

```
struct ImageGalleryView: View {
    let inspirations = NatureInspiration.examples()

    var body: some View {
        GeometryReader(content: { geometry in
            ScrollView {
                LazyVGrid(columns: gridItems(for: geometry.size.width), spacing: 0) {
                    ForEach(inspirations) { inspiration in
                        ImageAspectView(imageName: inspiration.imageName,
                                        frameAspectRatio: 1.3)
                    }
                }
            }
        })
    }
}

func gridItems(for width: CGFloat) -> [GridItem] {
    let numberOfColumns = Int(round(width / 200))
    let item = GridItem(.flexible(minimum: 150, maximum: 350),
                        spacing: 0)
    return Array(repeating: item,
                count: numberOfColumns)
}
```



In this example, I use `GeometryReader` to access the available width of the screen. I then pass this width to the `gridItems` function, which calculates the number of columns based on the width and returns an array of `GridItem` objects. Each `GridItem` represents a column in the grid and specifies its minimum and maximum width.

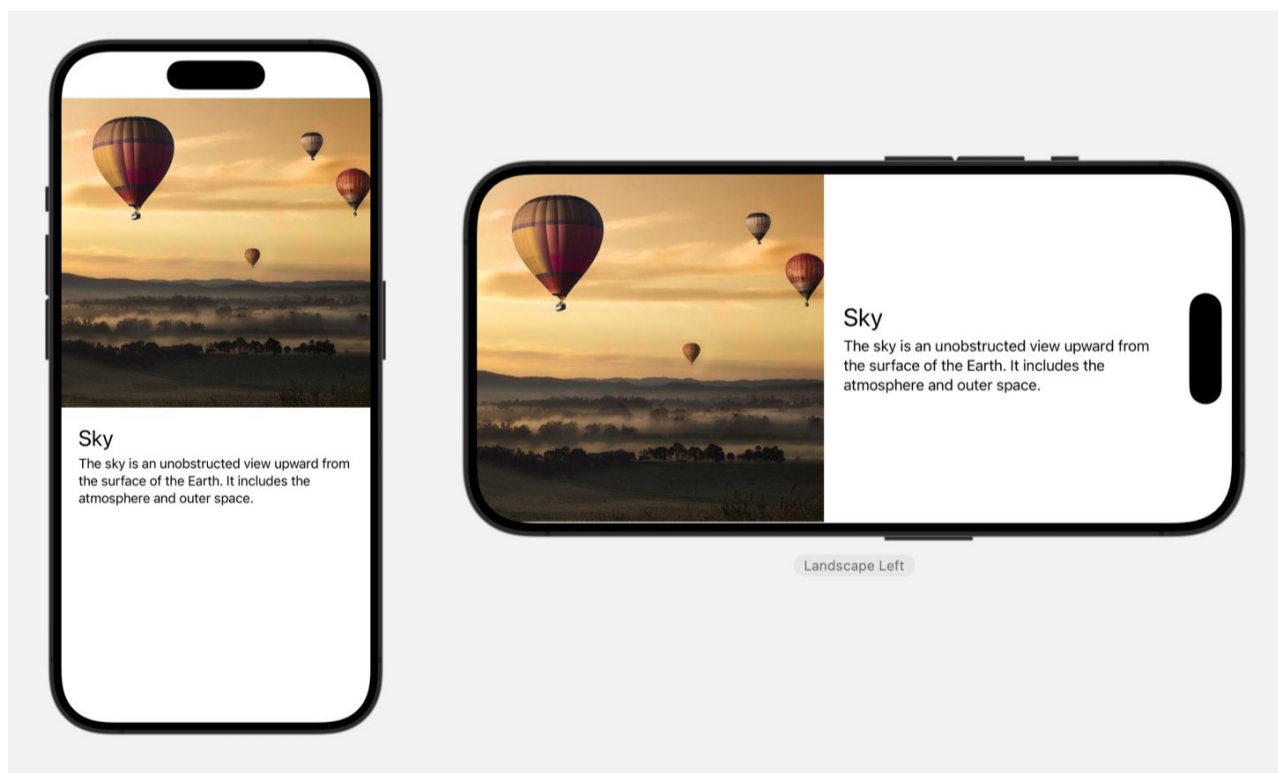
By adjusting the `generateGridItems` function, you can easily customize the number of columns and the width range for each column. This flexibility allows you to create dynamic grid layouts that adapt to different screen sizes and orientations.

On an iPhone 15 in landscape mode, 4 columns will be shown, whereas for portrait mode only 2 columns are used. This approach also works great for the iPad where a much larger screen offers to show more grid elements at the same time.

Using `GeometryReader` together with a `Grid` layout is a very useful way to create adaptive layouts.

7.3 EXAMPLE: CUSTOM CONTAINER WITH GEOMETRYREADER

In this section, we will explore how to create a custom layout using SwiftUI's `GeometryReader`. We will focus on a specific challenge: displaying an inspiration detail view that adjusts its layout based on the device's orientation or available space.



In landscape mode, we want to change the `HStack` layout to a `VStack`. To achieve this, we can use `GeometryReader` to check the available space and conditionally display either a `VStack` or an `HStack`.

To make our layout even more adjustable, I am creating a custom layout container that handles the conditional logic. I can pass in a limit value. If the available space is larger than this limit, a VStack is used and if it is smaller a HStack:

```
struct GeometryStack<Content: View>: View {
  let content: () -> Content
  let limit: CGFloat

  init(limit: CGFloat = 400,
        @ViewBuilder content: @escaping () -> Content) {
    self.limit = limit
    self.content = content
  }

  var body: some View {
    GeometryReader(content: { geometry in
      if geometry.size.width > limit {
        ScrollView {
          HStack(alignment: .center,
                 spacing: 10,
                 content: content)
        }
      } else {
        ScrollView {
          VStack(alignment: .center,
                 spacing: 10,
                 content: content)
        }
      }
    })
  }
}
```

Additionally, I added scroll views around both the HStack and VStack to make it even more adaptable to different screen sizes.

I can then use this container around the views that I want to use for the detail view, which is an image and 2 text views:

```
struct InspirationDetailView: View {

  let inspiration: NatureInspiration

  var body: some View {
    GeometryStack {
      ImageAspectView(imageName: inspiration.imageName,
                       frameAspectRatio: 1)

      VStack(alignment: .leading, spacing: 5) {
        Text(inspiration.name)
          .font(.title)
        Text(inspiration.description)
      }
      .padding(12)
    }
    .edgesIgnoringSafeArea([.bottom, .leading])
  }
}
```

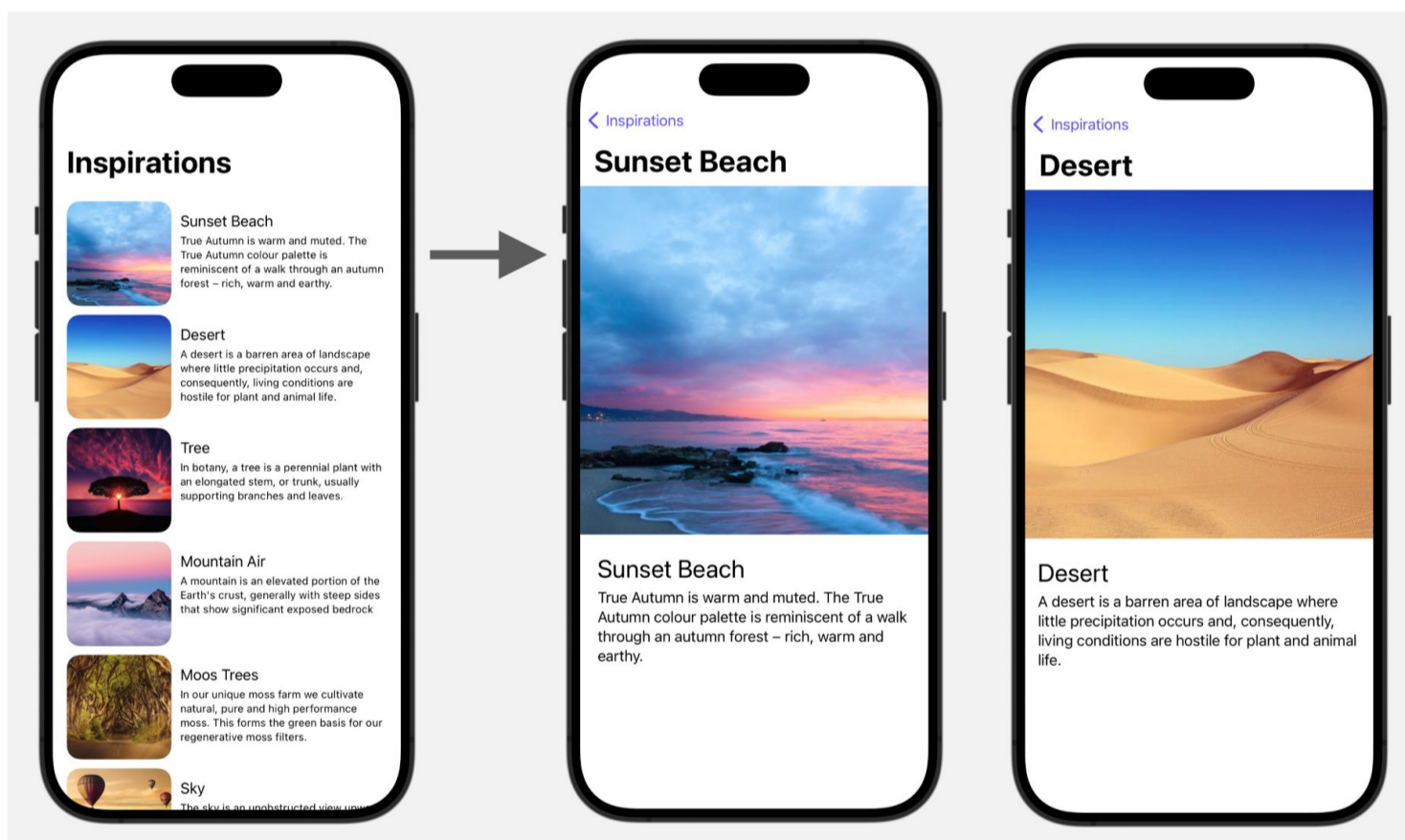
7.4 PREFERENCEKEYS

In this section, we will explore the concept of preference keys and preference values, which are useful for passing information within the view hierarchy. There are two directions in which you can pass values:

- top to bottom, where values are injected from the top and passed down to child and sub-child view which is passed in the **Environment**
- and the opposite direction, where a child view wants to pass a value to its parent and further up the view hierarchy. This is done with **PreferenceKeys**

Example: Navigation Title

Let's start by looking at an example where preference keys are used. We will use the `NavigationStack` and `NavLink` to demonstrate this. Imagine we have a list of inspirations, and when we tap on one of them, we navigate to a detail view. I also want to display a navigation title for the inspirations:



```
struct PreferenceExampleView: View {  
  
    let inspirations = NatureInspiration.examples()  
  
    var body: some View {  
        NavigationStack {  
            ScrollView {  
                LazyVStack(alignment: .leading, spacing: 10, content: {  
                    ForEach(inspirations) { inspiration in  
                        NavLink {  
                            InspirationDetailView(inspiration: inspiration)  
                                .navigationTitle(inspiration.name)  
                        } label: {  
                            InspirationRow(inspiration: inspiration)  
                        }.buttonStyle(.plain)  
                    }  
                }  
            }  
        }  
    }  
}
```

```

        })
        .padding()
    }
    .navigationTitle("Inspirations")
}
}
}
}
}

```

To show a title in the navigation bar, you can use the `navigationTitle` modifier, which needs to be added inside the `NavigationStack`. If you would add it outside like so:

```

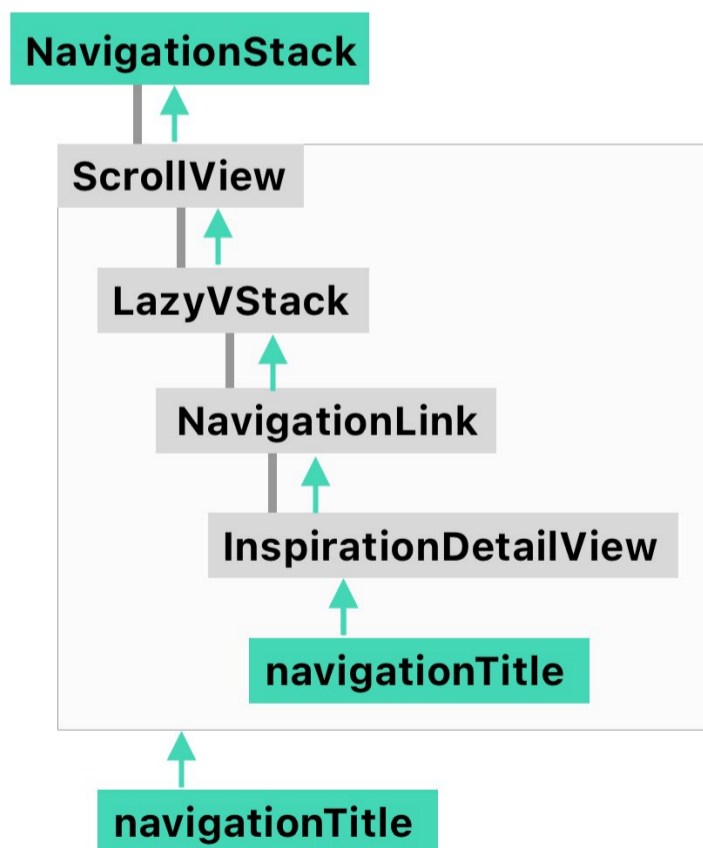
NavigationStack {
    ""
}
.navigationTitle("Inspirations")

```

the navigation title would not be shown. This is because the navigation stack and navigation views propagate the title from inside the views to the parent in the hierarchy. This approach allows for multiple views to contribute to the navigation title, with the innermost title taking precedence.

By using `PreferenceKeys` to pass the navigation title, you can add multiple different values from within your view hierarchy. When I have a navigation link open the preference value from that view is passed and used for the navigation title. Whereas when I show the root view, the navigation title from that view is used.

To achieve this kind of value propagation from child to parent views, we can use preference keys:



Example: Custom Container with PreferenceKey

Let's dive deeper into how Preferences works by creating a similar container view. First, we define our own preference key, which in this case is a string preference key to pass string values.

```
struct StringPreferenceKey: PreferenceKey {
    static var defaultValue: String? = nil

    static func reduce(value: inout String?, nextValue: () -> String?) {
        guard let nextValue = nextValue() else { return }
        value = nextValue
    }
}
```

The purpose of the reduce function is to determine how to combine these values. You have the flexibility to choose how the values are accumulated, whether you want to keep all of them, only keep the first or last value, or perform any other custom logic.

In the example we discussed earlier, the reduce function simply replaces the current value with the next value. This means that as new values are encountered, the previous value is overwritten, and only the latest value is stored.

It's important to note that the reduce function is called for each view that contributes a value to the preference key. SwiftUI automatically handles the accumulation process, iterating through the views and passing the values to the reduce function.

If no value for the given PreferenceKey is passed from a child view, SwiftUI will use the value you specify by the defaultValue property. In this example, we will have a nil value.

You can then pass a value with preferences with the preference modifier:

```
Text("Container with Preferences")
    .preference(key: StringPreferenceKey.self, value: "My Title")
```

To make it more convenient, we can extend the View type with a function that handles the preference key and value.

```
extension View {
    func myContainerTitle(_ title: String) -> some View {
        self.preference(key: StringPreferenceKey.self, value: title)
    }
}
```

In the container view, we can access the preference value using the `onPreferenceChange` modifier. We can then store this value in a state property and use it within the view.

```
struct PreferenceContainerView<Content> : View where Content : View {  
  
    @State private var title: String? = nil  
    let content: () -> Content  
  
    init(@ViewBuilder content: @escaping () -> Content) {  
        self.content = content  
    }  
  
    var body: some View {  
        ZStack {  
            RoundedRectangle(cornerRadius: 25.0)  
                .fill(Color.cyan.gradient)  
  
            VStack(alignment: .leading,  
                spacing: 10) {  
  
                if let title {  
                    Text(title)  
                        .font(.title)  
                        .bold()  
                }  
  
                content()  
            }  
            .padding()  
        }  
        .aspectRatio(1, contentMode: .fit)  
        .onPreferenceChange(StringPreferenceKey.self, perform: { value in  
            self.title = value  
        })  
    }  
}
```

By using preference keys, we can easily pass values from child views to their parent views and further up the view hierarchy. This approach allows for flexible customization and dynamic updates based on the values passed.

```
PreferenceContainerView {  
    Text("First Text")  
        .myContainerTitle("First")  
  
    HStack {  
        Text("Container with Preferences")  
            .myContainerTitle("Second title")  
  
        VStack {  
            Text("Child")  
                .myContainerTitle("Child title")  
        }  
    }  
}
```



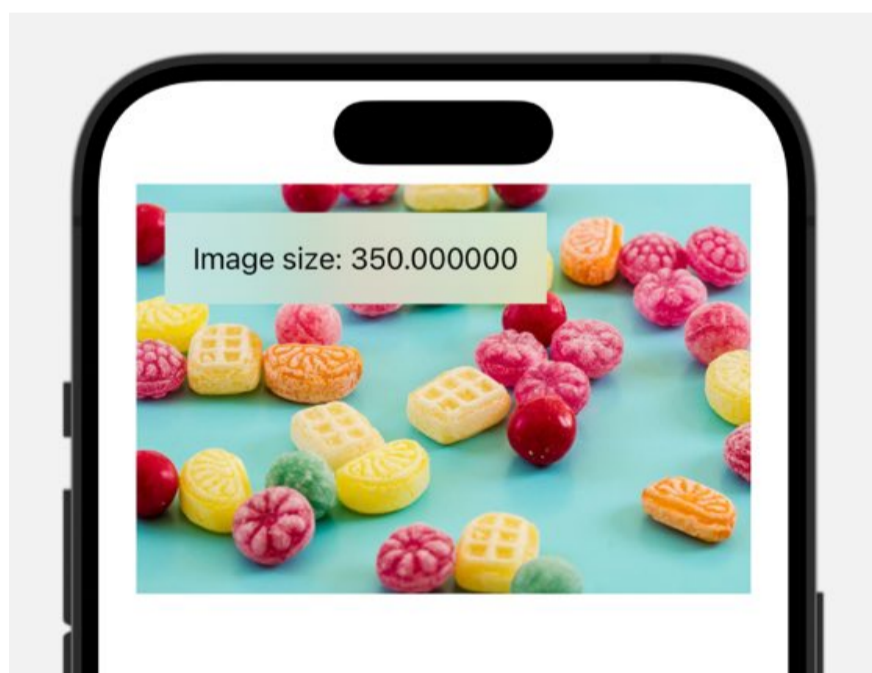
While preference keys may seem a bit complex at first, understanding how they work can greatly enhance your SwiftUI layout skills. It's important to remember to place the necessary modifiers and values in the correct positions to ensure the desired behavior. With preference keys, you have the advantage of multiple positions to attach and pass values, providing a powerful tool for creating custom layouts.

7.5 BOUNDS MEASUREMENT WITH PREFERENCEKEYS AND GEOMETRYREADER

In this section, we will focus on measuring the bounds of views using GeometryReader and PreferenceKeys. We want to know the size of a specific view and display it in another view without affecting the layout.

To measure the size of this view, we can leverage the fact that modifiers like background and overlay do not influence the size of the views they are applied to. By using the overlay modifier with a GeometryReader, we can obtain the size of the view:

```
Image(.candies)
    .resizable()
    .scaledToFit()
    .frame(width: 350)
    .overlay {
        GeometryReader(content: { geometry in
            Text("Image size: \(geometry.size.width)")
                .padding()
                .background(.thinMaterial)
                .padding()
        })
    }
}
```



I have the size of the view inside the overlays secondary child view. To pass this size value up the view hierarchy, I will use PreferenceKeys.

First, we need to define a preference key. Let's call it BoundsPreferenceKey:

```
struct BoundsPreferenceKey: PreferenceKey {
    static var defaultValue: CGRect = .zero

    static func reduce(value: inout CGRect, nextValue: () -> CGRect) {
        value = nextValue()
    }
}
```

Next, I am creating a view modifier that holds the logic with the GeometryReader in the background:

```
struct BoundsMeasurement: ViewModifier {
    @Binding var bounds: CGRect
    let namespace: String

    func body(content: Content) -> some View {
        content
            .background {
                GeometryReader(content: { geometry in
                    Color.clear
                    // pass value from the child view:
                    .preference(key: BoundsPreferenceKey.self,
                               value: geometry.frame(in: .named(namespace)))
                })
            }
        // access value from preferences
        .onPreferenceChange(BoundsPreferenceKey.self, perform: { value in
            self.bounds = value
        })
    }
}
```

I can inject this size of the GeometryReader as a preference. We'll use the onPreferenceChange modifier to capture the value and store it in a state variable.

By encapsulating this functionality in a view modifier, we can make it more reusable:

```
extension View {
    func measureBounds(bounds: Binding<CGRect>,
                      namespace: String) -> some View {
        self.modifier(BoundsMeasurement(bounds: bounds,
                                         namespace: namespace))
    }
}
```

Now, we can use the MeasureBounds modifier to measure the bounds of any view:

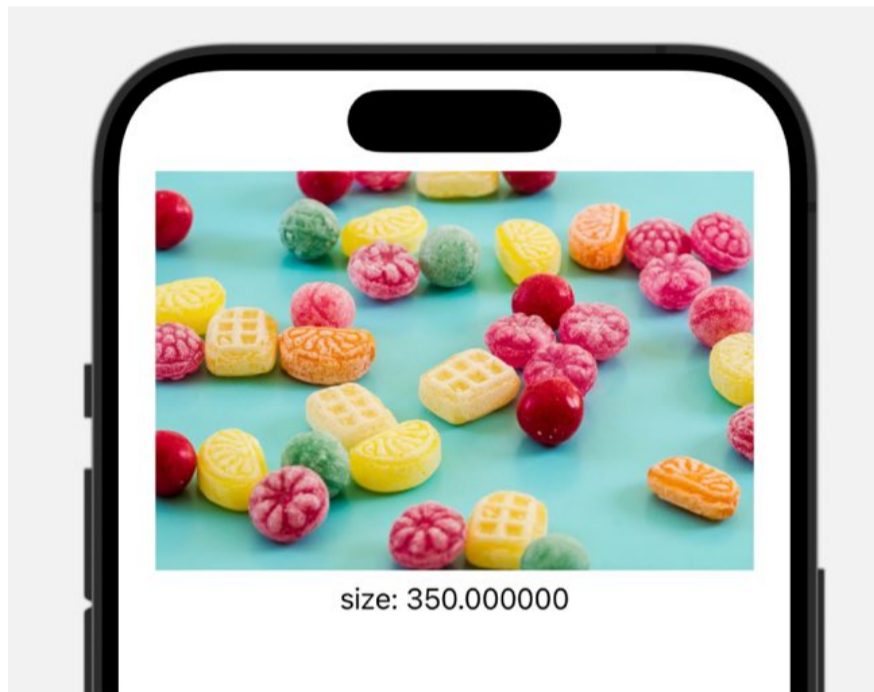
```
struct BoundsMeasurementExampleView: View {
    @State private var bounds = CGRect.zero
```

```

var body: some View {
    VStack {
        Image(.candies)
            .resizable()
            .scaledToFit()
            .frame(width: 350)
            .measureBounds(bounds: $bounds, namespace: "mynamespace")

        Text("size: \(bounds.width)")
    }
}

```



This approach allows us to measure the bounds of a view without affecting the layout and pass the information up the view hierarchy for further use.

Example: ScrollView Animation

I will use the size measurement approach from the previous section to add animations to a scroll view. I want to fade out the view that is leaving the scroll area on the top.

```

struct ScrollOffsetExampleView: View {
    let inspirations = NatureInspiration.examples()
    let namespace = "scrollviewspace"

    var body: some View {
        ScrollView {
            LazyVStack(spacing: 0) {
                ForEach(inspirations) { inspiration in
                    FadeOutImageView(name: inspiration.imageName,
                                     namespace: namespace)
                }
            }
        }.coordinateSpace(name: namespace)
    }
}

```

I create a custom sub-view for each image in the scroll view and use the `measureBounds` modifier to get access to the frame. I want to know the current scroll position of each image in the scroll view. The frame minimum Y position is the top position in the coordinate space of the scroll view. I had to add a `coordinateSpace` modifier around the `ScrollView` and pass a name. The same name is used in the namespace of `GeometryReader` to get the coordinates in reference to the scroll view.

This allows me to use the frame minimum Y position to change the opacity of the image. If the image position is negative, the image is no longer inside the scrollview and the opacity is set to 0.

```
struct FadeOutImageView: View {
    let name: String
    @State private var frame = CGRect.zero
    let namespace: String

    var opacity: CGFloat {
        guard frame.minY < 0 else { return 1 }

        let offset = abs(frame)
        let min = min(offset, frame.height)

        return 1 - min / frame.height
    }

    var body: some View {
        Image(name)
            .resizable()
            .scaledToFit()
            .opacity(opacity)
            .measureBounds(bounds: $$frame, namespace: namespace)
    }
}
```



Note that with iOS 17, you can use `scrollTransition` and `visualEffect` to achieve the same behavior with a much easier approach.

7.6 LAYOUT PROTOCOL

In iOS 16, the Layout Protocol was introduced. This means that you now have the ability to create your own layout containers and components, similar to the existing `VStack` and `HStack`. This opens up a world of possibilities, from basic layouts to more complex and sophisticated ones.

In many cases, you can replace `GeometryReader`, which has its limitations, with the new layout protocol. However, be aware that this approach is more complex as it requires you to manually layout and size your views.

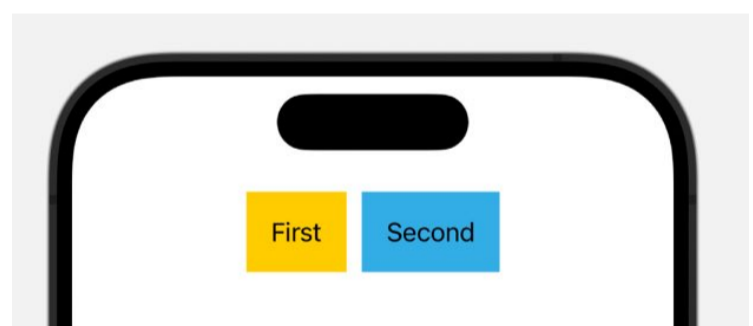
In order to dive into this topic, let's revisit what we discussed in the [layout system section](#), where we explored how views are sized. When working with the layout protocol, we get a glimpse behind the scenes of the layout process in SwiftUI. There are three main steps involved:

- First, the parent view receives a size from its parent. If it's the only view, it will take up the entire screen and pass down a **proposed size to its children**.
- **The child view then decides how large it wants to be** and passes that information back up to the parent.
- Once the parent knows the size of its children, it can also **pass its own values back to its parent**.
- Finally, after all the sizes have been determined, the **views are positioned using alignments** and alignment directions.

Example: Recreating HStack

To demonstrate the process of creating a custom layout, let's take a look at an example where we recreate an `HStack`. It would achieve the following layout with 2 small text views:

```
HStack(spacing: 10) {
  Text("First")
    .padding()
    .background(Color.yellow)
  Text("Second")
    .padding()
    .background(Color.cyan)
}
```



We'll start by defining a struct called `CustomHStack` that conforms to the layout protocol. This protocol has two requirements: `sizeThatFits` and `placeSubviews`.

```
struct CustomHStack: Layout {

  func sizeThatFits(proposal: ProposedViewSize,
                   subviews: Subviews,
                   cache: inout ()) -> CGSize {
    // return the ideal size of this layout container
  }
}
```

```

func placeSubviews(in bounds: CGRect,
                  proposal: ProposedViewSize,
                  subviews: Subviews,
                  cache: inout ()) {
    // position views
}

```

The **sizeThatFits** function is called to determine how large the container wants to be. We need to consider the views inside the container, which is why we receive the subviews and the proposed size from the parent. In this function, we can calculate the ideal size of our container CustomHStack.

The **placeSubviews** function is responsible for positioning the subviews within the container. For example, you can iterate over all subviews and use call `place(at: CGPoint, proposal: ProposedViewSize)`.

How to calculate the container Size

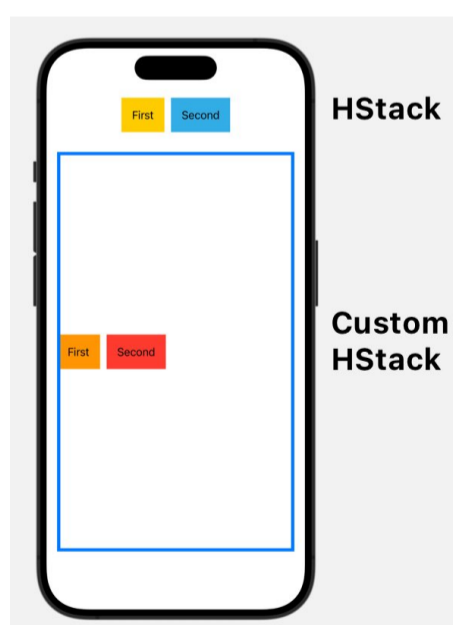
First, you need to return a size for your container. If I don't have any subview, I am returning a zero size for my container. If you want to implement a **greedy container** and take all the space that is been offered, you can call:

```

func sizeThatFits(proposal: ProposedViewSize,
                  subviews: Subviews,
                  cache: inout ()) -> CGSize {
    guard !subviews.isEmpty else { return .zero }

    return proposal.replacingUnspecifiedDimensions()
}

```



Be aware that if you add **fixedSize** or embed your layout stack inside a **ScrollView** the proposed size will have nil values, that are referred to as unspecified. By calling `replacingUnspecifiedDimensions` these values are replaced by 10 points which is the default value. From the definition of this function you can see the 10 points:

```
@inline public func replacingUnspecifiedDimensions(by size: CGSize =
CGSize(width: 10, height: 10)) -> CGSize
```

In this example, I want to create a conservative container that only asks for the space it needs to fit its child views. I will write a separate function to calculate the intrinsic size:

```
func sizeThatFits(proposal: ProposedViewSize,
                 subviews: Subviews,
                 cache: inout ()) -> CGSize {
    guard !subviews.isEmpty else { return .zero }

    return intrinsicSize(subviews: subviews, for: proposal)
}

func fullIntrinsicSize(subviews: Subviews,
                      for proposal: ProposedViewSize) -> CGSize {
    let viewSizes = calculateSizes(subviews: subviews,
                                   for: proposal)
    let height = viewSizes.max { $0.height < $1.height }?.height ?? 0
    let width = viewSizes.reduce(0) { $0 + $1.width }

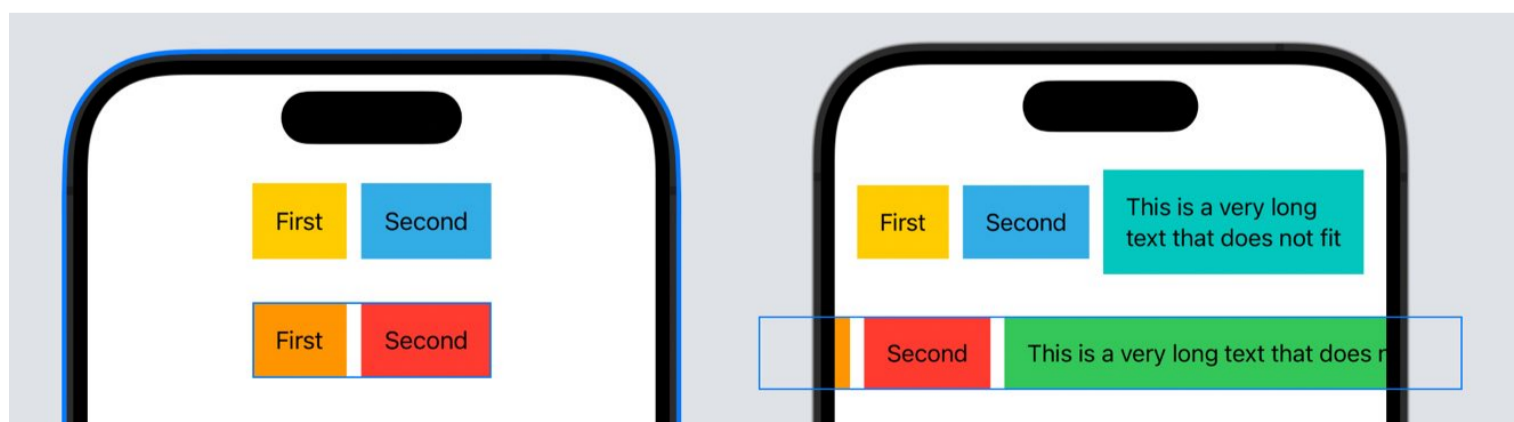
    return CGSize(width: width + totalSpacing, height: height)
}
```

I am using the size of the individual subviews. The height of an HStack is decided by the largest child view. The width of the Hstack is a sum of the width of all child views plus the total spacing.

The tricky part is actually to decide how large each child view should be. As an example, I am using the ideal size of each child view.

```
func calculateSizes(subviews: Subviews,
                  for proposal: ProposedViewSize) -> [CGSize] {
    var viewSizes = [CGSize]()

    for subview in subviews {
        let idealSize = subview.dimensions(in: .unspecified)
        viewSizes.append(CGSize(width: idealSize.width,
                                height: idealSize.height))
    }
    return viewSizes
}
```



This approach replicates an HStack with small views inside. However, when the text views get larger, the container will size too big and overflow the screen. In order to replicate the size distribution, you need to consider how each subview will fit into the available space.

Each subview has a dimensions function that you can call to get certain size behaviours. You can get the maximum, minimum, and ideal sizes:

```
let maxSize = subview.dimensions(in: .infinity)
    // same as subview.sizeThatFits(.infinity)
let idealSize = subview.dimensions(in: .unspecified)
    // same as subview.sizeThatFits(.unspecified)
let minSize = subview.dimensions(in: .zero)
    // same as subview.sizeThatFits(.zero)
```

The dimensions function will return a ViewDimensions type. Whereas the sizeThatFits function returns CGSize.

You can also ask for **the size that fits in a specified proposed size**. Let's say I have a larger multiline text and I want to know how it fits in the container available space:

```
let sizeThatFits = subview.sizeThatFits(.init(width: proposal.width,
    height: proposal.height))

// same as subview.dimensions(in: ProposedViewSize(width: proposal.width,
    height: proposal.height))
```

A text view would return the same size for all 3 proposals. A Color view would return 0 for the minimum, 10 for the ideal, and inf for the maximum size. You can call these functions multiple times for each subview to determine its sizing behavior e.g. greedy view vs fixed size view.

	Color	Text	.frame(min, ideal, max)
min size	0	e.g. 68	Frame minimum value
ideal size	10	e.g. 68	Frames ideal value
max size	Inf	e.g. 68	Frames max value
size that fits	Returns proposed size	Returns size that fits in proposed size	Minimum of proposed size and frame ideal size

In this case, I would propose each child view the full available space, which would lead again to the container being too big. Instead, I want to offer each subview a portion of the available space:


```

func calculateSizes(subviews: Subviews,
                   for proposal: ProposedViewSize) -> [CGSize] {
    let availableWidth = proposal.width ?? 0 - totalSpacing(subviews: subviews)
    let individualWidth = availableWidth / CGFloat(subviews.count)
    let individualProposal = ProposedViewSize(width: individualWidth,
                                              height: proposal.height)

    var viewSizes = [ViewSizeResult]()
    for subview in subviews {
        let sizeThatFits = subview.sizeThatFits(individualProposal)
        viewSizes.append(sizeThatFits)
    }
    return viewSizes
}

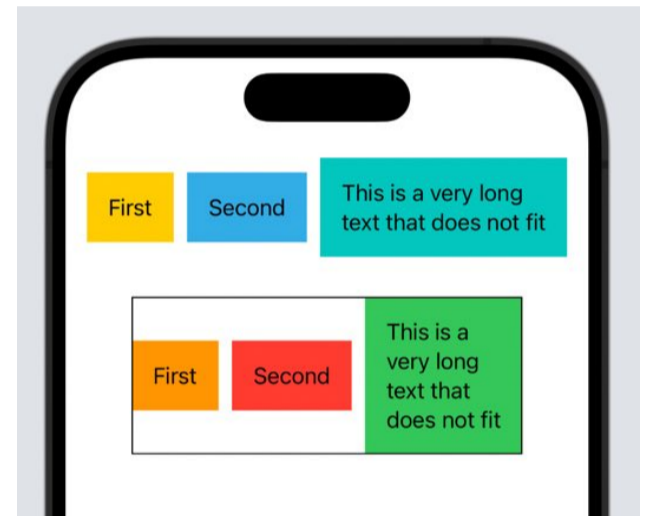
```

I proposed each subview an equal proportion of the available space. The resulting layout will fit on the screen but it will not fill the whole available space:

```

CustomHStack(spacing: 10) {
    Text("First")
        .padding()
        .background(Color.orange)
    Text("Second")
        .padding()
        .background(Color.red)
    Text("This is a very long text
        that does not fit")
        .padding()
        .background(Color.green)
}

```



To further optimize the layout for space, you can add an additional round of redistributing space to subviews that want more space. You can find out if a view fits by comparing its sizes:

```

let idealSize = subview.sizeThatFits(.unspecified)
let sizeThatFits = subview.sizeThatFits(individualProposal)

let doesFit = idealSize == sizeThatFits

```

I would then filter the subviews that do not fit and redistribute the remains space:

```

struct ViewSizeResult {
    let index: Int
    var size: CGSize
    var isFit: Bool
}

func calculateSizes(subviews: Subviews,
                   for proposal: ProposedViewSize) -> [CGSize] {
    let availableWidth = proposal.width ?? 0 - totalSpacing(subviews: subviews)
    let individualWidth = availableWidth / CGFloat(subviews.count)
    let individualProposal = ProposedViewSize(width: individualWidth,
                                              height: proposal.height)

    var viewSizes = [ViewSizeResult]()
    for (index, subview) in subviews.enumerated() {
        let idealSize = subview.sizeThatFits(.unspecified)
        let sizeThatFits = subview.sizeThatFits(individualProposal)
    }
}

```

```

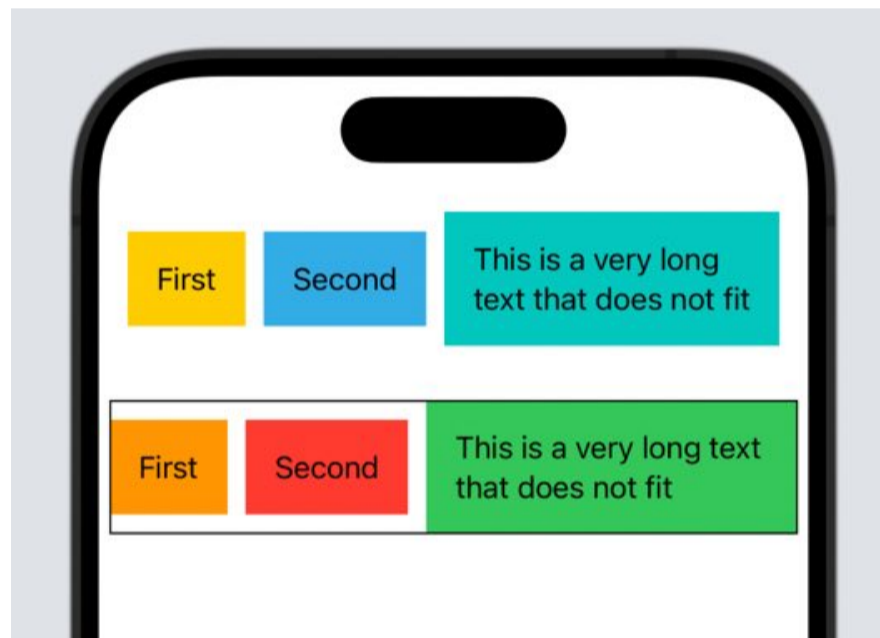
    let doesFit = idealSize == sizeThatFits
    viewSizes.append(ViewSizeResult(index: index,
                                    size: doesFit ? sizeThatFits : .zero,
                                    isFit: doesFit))
}

let widthThatIsUsedByFitViews = viewSizes.reduce(0) { $0 + $1.size.width }
let remainingWidth = availableWidth - widthThatIsUsedByFitViews
let countRemainingViews = viewSizes.filter { !$0.isFit }.count
let remainingWidthOffering = remainingWidth / CGFloat(countRemainingViews)
let remainingOffering = ProposedViewSize(width: remainingWidthOffering,
                                          height: proposal.height)

for viewSize in viewSizes {
    if !viewSize.isFit {
        let size = subviews[viewSize.index].sizeThatFits(remainingOffering)
        viewSizes[viewSize.index].size = size
    }
}

return viewSizes.map { $0.size }
}

```



In the below example, my custom layout behaves very similarly to the system HStack. I only added one more round of space redistribution. But you can imagine that you can add a lot more complexity if you use more subviews in different situations.

How to position the subviews inside the Layout container

In order to position the views, you have to use the `placeSubviews` function and iterate over all subviews. Each subview has a function `place(at:)` that you can call to position the view. Additionally you pass in a proposed size to this subview:

```

func placeSubviews(in bounds: CGRect,
                  proposal: ProposedViewSize,
                  subviews: Subviews,
                  cache: inout ()) {

```

```

guard !subviews.isEmpty else { return }

var x = bounds.minX
let viewSizes = calculateSizes(subviews: subviews,
                              for: proposal)
for (index, subview) in subviews.enumerated() {
    let size = viewSizes[index]
    subview.place(at: CGPoint(x: x + size.width / 2,
                              y: bounds.midY),
                 anchor: .center,
                 proposal: ProposedViewSize(width: size.width,
                                             height: size.height))

    x += size.width + spacing
}
}

```

The position needs to be given in absolute position of the screen. You can get the start position of your Layout container by using the bounds minimum X and Y coordinates.

In the example of an HStack I would move each view to the next horizontal direction. In the above example I aligned the views in the HStack with a center alignment. But you can also pass in an alignment property to your Layout container that you use in the placeSubviews function:

```

struct CustomHStack: Layout {

    let alignment: VerticalAlignment
    let spacing: CGFloat

    ...

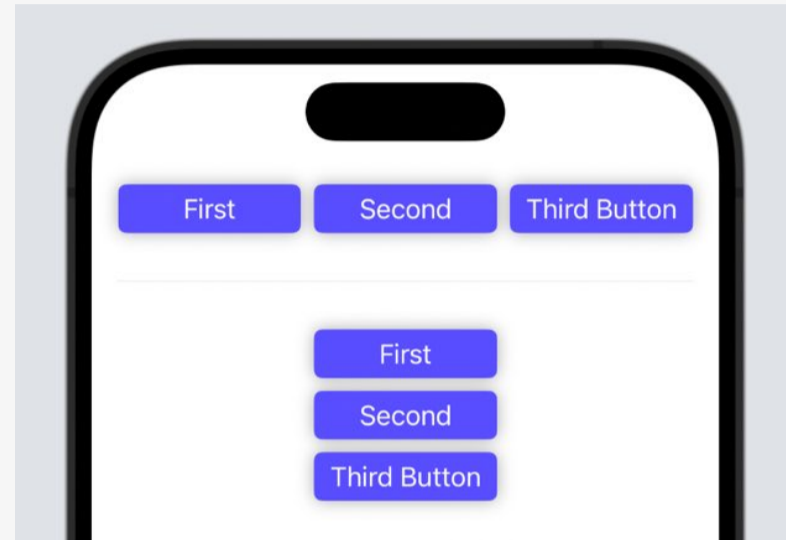
}

```

7.7 Layout Example: Equal Width HStack and VStack

In this section, I will show you a custom layout that creates a stack with equal-sized child views. This example is from WWDC 2023, where they introduced layouts that ensure all child views have the same size in either height or width.

```
EqualWidthVStack() {  
    Button(action: {  
  
    }, label: {  
        Text("First")  
    })  
  
    Button(action: {  
  
    }, label: {  
        Text("Second")  
    })  
  
    Button(action: {  
  
    }, label: {  
        Text("Third Button")  
    })  
}  
.buttonStyle(CustomButtonStyle())
```



The `EqualWidthVStack` and `EqualWidthHStack` calculate their size by using the size of the largest subview. For example, the horizontal stack uses the max subview size for its size:

```
func maxSize(subviews: Subviews) -> CGSize {  
    let subviewSizes = subviews.map { $0.sizeThatFits(.unspecified) }  
    let maxSize: CGSize = subviewSizes.reduce(.zero) { currentMax, subviewSize in  
        CGSize(  
            width: max(currentMax.width, subviewSize.width),  
            height: max(currentMax.height, subviewSize.height))  
    }  
  
    return maxSize  
}  
  
func fullIntrinsicSize(subviews: Subviews) -> CGSize {  
    let maxSize = maxSize(subviews: subviews)  
    let totalSpacing = totalSpacing(subviews: subviews)  
  
    return CGSize(width: maxSize.width * CGFloat(subviews.count) + totalSpacing,  
                  height: maxSize.height)  
}
```

This layout does not consider the proposed size it gets from its parent. If you use the horizontal stack, this can lead to the layout being larger than the screen size. This behavior is intentional and allows us to define different ways of using this layout.

By using `ViewThatFits`, you can switch between these two layouts. If the horizontal layout does not fit, the smaller vertical version will be used:

```

struct EqualSizeButtonExampleView: View {
    var body: some View {
        ViewThatFits {
            EqualWidthHStack() {
                buttons
            }
            .buttonStyle(CustomButtonStyle())

            EqualWidthVStack() {
                buttons
            }
            .buttonStyle(CustomButtonStyle())
        }
        .padding()
    }

    @ViewBuilder
    var buttons: some View {
        Button(action: {

        }, label: {
            Text("First")
        })

        Button(action: {

        }, label: {
            Text("Second")
        })

        Button(action: {

        }, label: {
            Text("Third Button")
        })
    }
}

```

7.8 Layout Example: Flow Layout

In this section, we will explore the flow layout, which is a highly requested feature in SwiftUI. You can accomplish a flow layout with the new Layout Protocol. This allows you to create a collection view-like layout where items flow horizontally and wrap to the next line when necessary.

In the following example, I am creating a picker view where you can select your favorite pets. Inside the FlowLayout container, I am using a ForEach to iterate over all the available options:

```

struct FlowLayoutExampleView: View {

    let pets = ["cat", "dog", "fish", "horse", "snack", "bird", "rat"]
    @State private var selection = Set<String>()

    var body: some View {
        VStack(alignment: .leading) {
            Text("Select a pet")
                .bold()
            FlowLayout(alignment: .leading, spacing: 10) {

```

```

ForEach(pets, id: \.self) { pet in
    Text(pet)
        .foregroundColor(.white)
        .padding(5)
        .background(color(for: pet))
        .cornerRadius(3.0)
        .onTapGesture {
            if selection.contains(pet) {
                selection.remove(pet)
            } else {
                selection.insert(pet)
            }
        }
    }
}

func color(for pet: String) -> Color {
    selection.contains(pet) ? Color.indigo : Color.gray
}
}

```



This layout uses the ideal size of each element:

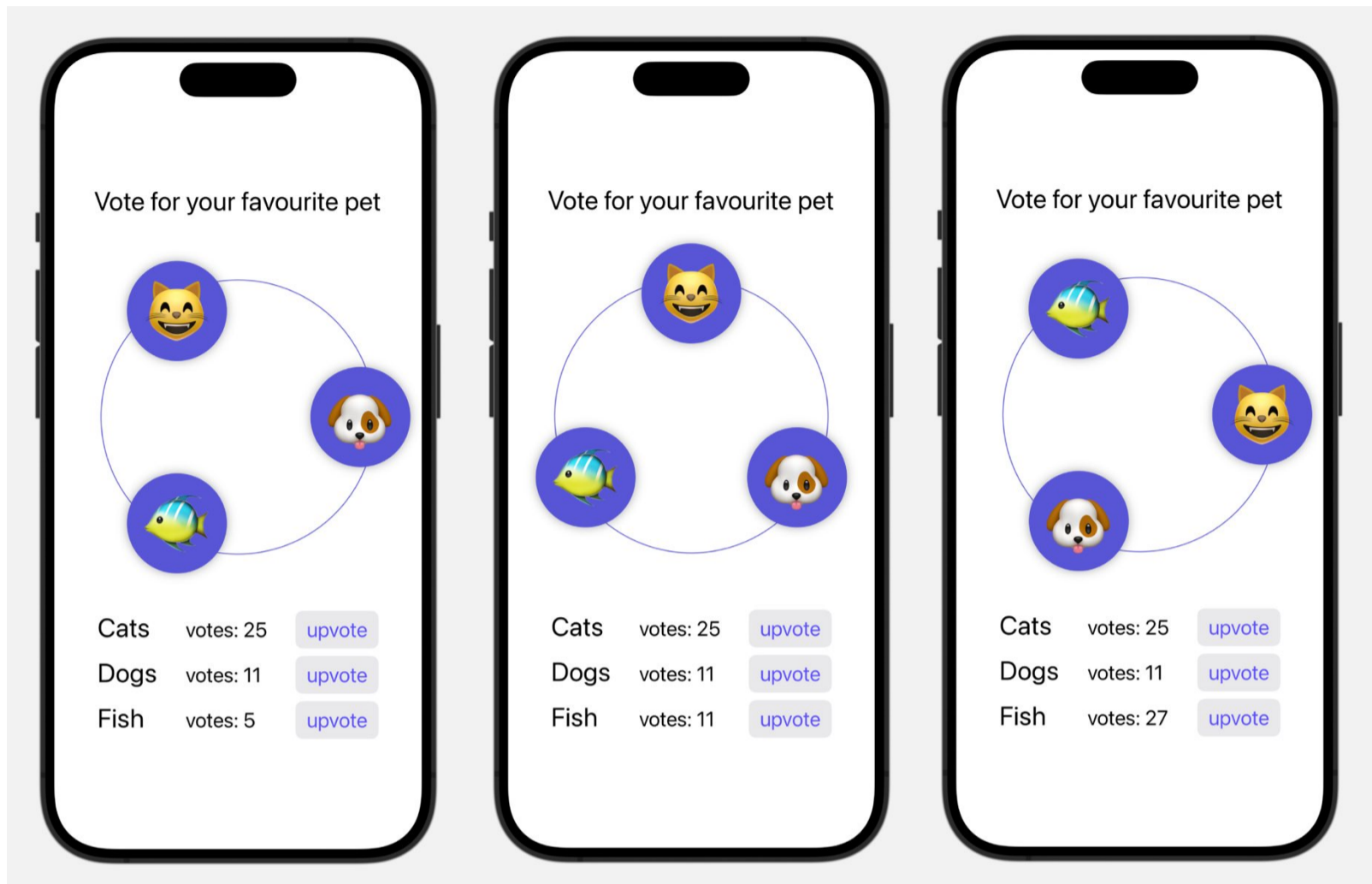
```
let idealSize = subview.sizeThatFits(.unspecified)
```

It then uses this size and the available space to distribute all subviews over multiple rows.

Note that you should not use this layout for large collection views because it is not lazily created. The Layout Protocol is suitable for small and precise sizing and positioning of views.

7.9 Layout Example: Radial Layout

Another example from the WWDC2023 is the following radial layout. As an example, I am using it to show the voting results for the favorite pet. The pet with the highest vote is shown on top. This layout only works for 3 items in the layout:



```
MyRadialLayout() {  
    ForEach(pets) { pet in  
        PetBubbleView(pet: pet, allPets: pets)  
    }  
}  
.background(Circle().stroke(Color.indigo).padding(50))  
.aspectRatio(1, contentMode: .fit)
```

The layout container needs to know the rating for each of the 3 elements. You can pass values from your subview to your layout by using `LayoutValueKey` which is a key that the layout uses to read the rank for a subview:

```
private struct Rank: LayoutValueKey {  
    static let defaultValue: Int = 1  
}
```

From the subview you can pass the layout value. This is a convenience view modifier that handles this:

```

extension View {
    /// Sets the rank layout value on a view.
    func rank(_ value: Int) -> some View {
        layoutValue(key: Rank.self, value: value)
    }
}

```

In the subview, you can add that rank value for each pet:

```

struct PetBubbleView: View {
    let pet: Pet
    let allPets: [Pet]

    var body: some View {
        Text(emoji(for:pet))
            .font(.system(size: 70))
            .padding()
            .background(Circle().fill(.indigo)
                .shadow(radius: 5))
            .rank(rank(pet))
    }

    func emoji(for pet: Pet) -> String {
        switch pet.type {
            case .cat: "🐱"
            case .dog: "🐶"
            case .fish: "🐟"
            case .horse: "🐎"
        }
    }

    func rank(_ pet: Pet) -> Int {
        allPets.reduce(1) { $0 + (($1.votes > pet.votes) ? 1 : 0) }
    }
}

```

The RadialLayout container can then access these values from each subview:

```

func placeSubviews(in bounds: CGRect,
    proposal: ProposedViewSize,
    subviews: Subviews,
    cache: inout Void) {

    let ranks = subviews.map { subview in
        subview[Rank.self]
    }

    for (index, subview) in subviews.enumerated() {
        // use ranks[index] to place subview
    }
}

```

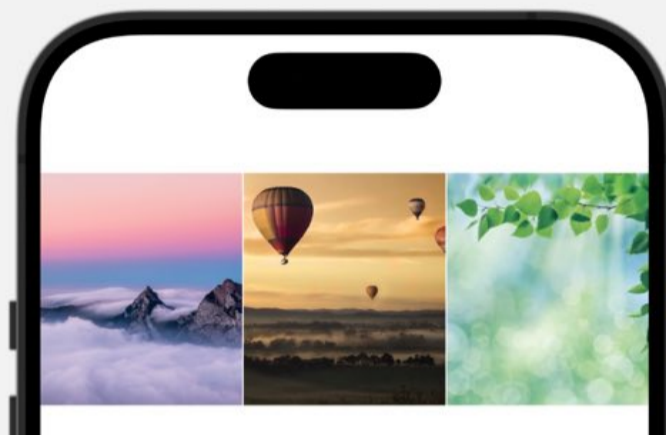

By creating your own layout keys and using them in the layout, you can influence the arrangement of views in unique ways. The radial layout is just one example of the possibilities that custom layouts offer.

7.10 Custom Layout with Layout Priority

In this section, we will explore a different way of using layout priority in SwiftUI. To illustrate this concept, let's consider an example of an HStack with 3 images:

```
HStack(spacing: 1) {
    ImageFillView(imageName: "mountain")
    ImageFillView(imageName: "sky")
    ImageFillView(imageName: "leaves")
}
.frame(height: 150)

struct ImageFillView: View {
    let imageName: String
    var body: some View {
        Color.cyan
            .overlay {
                Image(imageName)
                    .resizable()
                    .aspectRatio(nil, contentMode: .fill)
            }.clipped()
    }
}
```



All images have the same sizing. If you want to give one image more width, you could try to use layout priority:

```
HStack(spacing: 1) {
    ImageFillView(imageName: "mountain")
        .layoutPriority(0.25)
    ImageFillView(imageName: "sky")
        .layoutPriority(0.25)
    ImageFillView(imageName: "leaves")
        .layoutPriority(0.5)
}
.frame(height: 150)
```

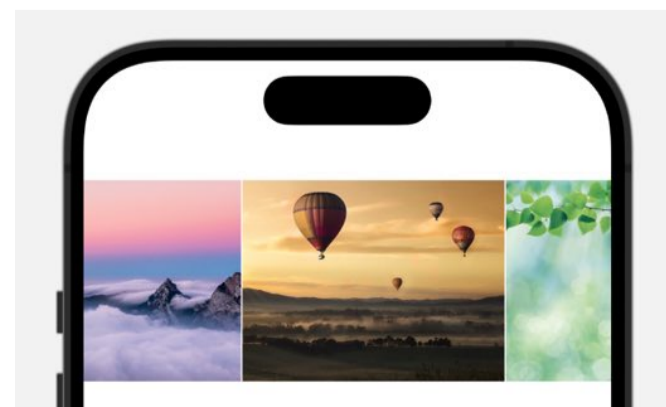


In this case, all the space is proposed first to the view with the highest layout priority. The image will take all the offered space and you can only see this one view.

Instead, I want to have a layout container that uses the layout priority as a percentage of the available width:

```
PriorityWidthHStack(spacing: 1) {
    ImageFillView(imageName: "mountain")
        .layoutPriority(0.3)

    ImageFillView(imageName: "sky")
        .layoutPriority(0.5)
}
```



```

        ImageFillView(imageName: "leaves")
            .layoutPriority(0.2)
    }
    .frame(height: 150)

```

Each subview has a priority property that I use together with the available space to calculate the subviews width:

```

struct PriorityWidthHStack: Layout {
    ...
    func placeSubviews(in bounds: CGRect,
                      proposal: ProposedViewSize,
                      subviews: Subviews,
                      cache: inout Void) {
        guard !subviews.isEmpty else { return }
        guard let containerWidth = proposal.width else { return }

        let totalSpacing = totalSpacing(subviews: subviews)
        let availableWidth = containerWidth - totalSpacing

        var x = bounds.minX

        for (index, subview) in subviews.enumerated() {
            let viewWidth = subview.priority * availableWidth
            let viewSize = subview.sizeThatFits(.init(width: viewWidth,
                                                    height: bounds.height))

            let placementProposal = ProposedViewSize(width: viewWidth,
                                                    height: containerHeight)

            let anchorPoint = anchorPoint(in: bounds,
                                         availableViewWidth: viewWidth,
                                         intrinsicViewSize: viewSize,
                                         x: &x)

            subviews[index].place(
                at: anchorPoint,
                anchor: .center,
                proposal: placementProposal)
            x += viewWidth + spacing
        }
    }
}

```

However, it's worth noting that this approach may lead to confusion when switching between different stacks. Alternatively, you could create another LayoutValueKey, similar to the radial example, where a rank was added. This approach would provide a clearer understanding of the layout priorities.

Alternative Solution: containerRelativeFrame

An easier solution for this specific problem would have been to use the new containerRelativeFrame which is available for iOS 17:

```

HStack(spacing: 1) {

```

```

ImageFillView(imageName: "mountain")
    .containerRelativeFrame(.horizontal) { length, axis in
        length * 0.3
    }

ImageFillView(imageName: "sky")
    .containerRelativeFrame(.horizontal) { length, axis in
        length * 0.5
    }

ImageFillView(imageName: "leaves")
}
.frame(height: 150)

```

Note that Xcode gave me an error when I added containerRelativeFrame to all 3 images.

7.11 Custom Layout for Image Gallery

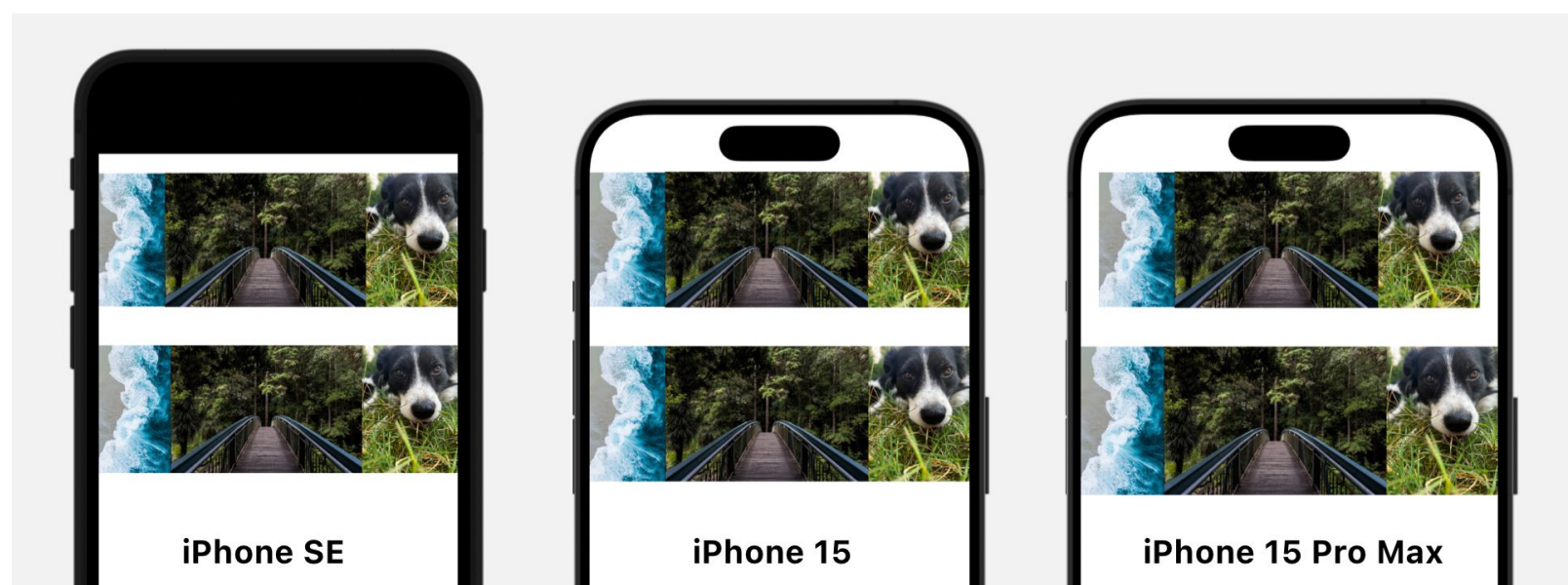
In this section, we will explore how to create a custom image gallery view using SwiftUI. The goal is to have multiple images with different aspect ratios displayed on the same row, all with the same height and fitting perfectly within the row.

To achieve this, we can start by adjusting and setting specific frame width values for each image to ensure equal scaling. However, a better approach is to use an HStack that automatically distributes the space without the need for manual adjustments.

```

HStack(spacing: 0) {
    Image("beach-view")
        .resizable()
        .scaledToFit()
        .frame(width: 78)
    Image("bridge")
        .resizable()
        .scaledToFit()
        .frame(width: 210)
    Image("dog_2")
        .resizable()
        .scaledToFit()
        .frame(width: 105)
}

```



In the following screenshots, you can see that a fixed frame size for the images (top row) does not adapt well to different screen sizes. The second image row adjusts the row height and individual image size to fit on on row. This depends solely on the aspect ratios of the images and the screen width:

The container view that accomplishes this adaptive layout does not rely on fixed frame sizes:

```
ImageCollectionHStack(spacing: 0) {  
    Image("beach-view")  
        .resizable()  
        .scaledToFit()  
    Image("bridge")  
        .resizable()  
        .scaledToFit()  
    Image("dog_2")  
        .resizable()  
        .scaledToFit()  
}
```

This special layout container uses the aspect ratios of the images to determine the size and position of the images:

```
struct ImageCollectionHStack: Layout {  
  
    let spacing: CGFloat  
  
    func sizeThatFits(proposal: ProposedViewSize,  
                     subviews: Subviews,  
                     cache: inout ()) -> CGSize {  
        let width = proposal.width ?? 0  
        let height = calculateHeight(proposal: proposal,  
                                     subviews: subviews)  
  
        return CGSize(width: width, height: height)  
    }  
  
    func calculateHeight(proposal: ProposedViewSize,  
                        subviews: Subviews) -> CGFloat {  
        let subViewSizes = subviews.map { $0.sizeThatFits(.unspecified) }  
        let subViewAspectRatios = subViewSizes.map { $0.width / $0.height }  
        let sumSubViewAspectRatios = subViewAspectRatios.reduce(0) { $0 + $1 }  
  
        let totalSpacing = spacing * CGFloat(subviews.count - 1)  
        let availablWidth = proposal.width ?? 0  
  
        return (availablWidth - totalSpacing) / sumSubViewAspectRatios  
    }  
  
    func placeSubviews(in bounds: CGRect,  
                      proposal: ProposedViewSize,  
                      subviews: Subviews,  
                      cache: inout ()) {  
        guard !subviews.isEmpty else { return }  
        let subviewHeight = calculateHeight(proposal: proposal,  
                                             subviews: subviews)  
  
        var x = bounds.minX  
  
        for (index, subView) in subviews.enumerated() {
```

```

    let viewSize = subView.sizeThatFits(.unspecified)
    let subviewWidth = subviewHeight * viewSize.width / viewSize.height
    let placementProposal = ProposedViewSize(width: subviewWidth,
                                              height: subviewHeight)

    subviews[index].place(at: CGPoint(x: x + subviewWidth / 2,
                                       y: bounds.midY),
                          anchor: .center,
                          proposal: placementProposal)
    x += subviewWidth + spacing
  }
}
}

```

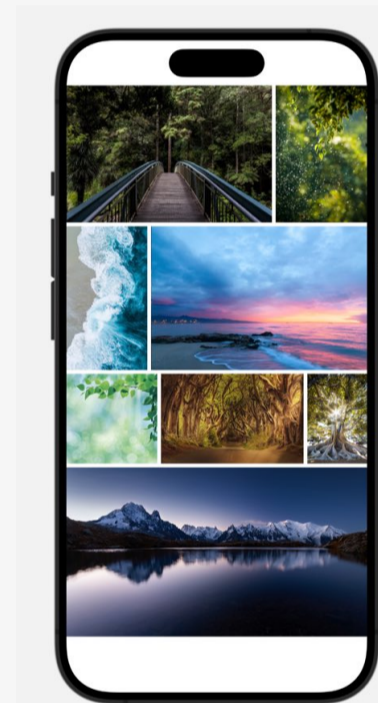
To see the full potential of this custom layout, let's create a ScrollView with a LazyVStack and set the spacing to one. Within this view, we can display the image collection using a ForEach loop. For simplicity, we can create an array with two dimensions, representing the number of images in each row:

```

struct ImageCollectionView: View {
    let images = [
        ["bridge", "green-tree-sun"],
        ["beach-view", "beach"],
        ["leaves", "trees-moos", "tree-sun"],
        ["mont-blanc"]
    ]

    let spacing: CGFloat = 5
    var body: some View {
        ScrollView {
            LazyVStack(spacing: spacing) {
                ForEach(images, id: \.self) { row in
                    ImageCollectionHStack(spacing: spacing) {
                        ForEach(row, id: \.self) { name in
                            Image(name)
                                .resizable()
                                .scaledToFit()
                        }
                    }
                }
            }
        }
    }
}

```



Example: Image Gallery View with Async Loading

However, when using async images or images downloaded from a server, we may not have the images available during layout. In this case, we only need the aspect ratio of each image to make the layout work. Similar to the radial layout example, we can use a custom layout value key called `aspectRatioKey` to calculate the layout.

```

private struct ImageAspectRatio: LayoutValueKey {
    static let defaultValue: CGFloat = 1
}

extension View {
    func imageAspectRatio(_ value: CGFloat) -> some View {
        layoutValue(key: ImageAspectRatio.self, value: value)
    }
}

```

```
}
```

I can then modify the Layout to use the aspect ratio from the subviews:

```
struct AspectRatioImageCollectionHStack: Layout {
    ...
    func calculateHeight(proposal: ProposedViewSize,
                        subviews: Subviews) -> CGFloat {
        let subViewSizes = subviews.map { $0.sizeThatFits(.unspecified) }

        let subViewAspectRatios = subviews.map { subview in
            subview[ImageAspectRatio.self]
        }

        let sumSubViewAspectRatios = subViewAspectRatios.reduce(0) { $0 + $1 }
        let totalSpacing = spacing * CGFloat(subviews.count - 1)
        let availablWidth = proposal.width ?? 0

        return (availablWidth - totalSpacing) / sumSubViewAspectRatios
    }
}
```

The data that is loaded from the server has information about the width and height of each image. This gives me the aspect ratio for the image layout:

```
struct PicsumPhoto: Codable, Identifiable, Hashable {
    let id: String
    let author: String
    let width: CGFloat
    let height: CGFloat
    let downloadUrl: String

    var aspectRatio: CGFloat {
        width / height
    }
}
```

The view model to load the image data returns an array of PicsumPhoto. To prepare for a 2 dimensional grid I am reordering these photos into a nested array:

```
class PicsumPhotoLoader: ObservableObject {

    @Published var photos: [PicsumPhoto] = [] {
        didSet {
            reorder()
        }
    }

    @Published var photoRows: [[PicsumPhoto]] = []

    func loadImage() {
        ...
    }
}
```

```
}
```

For the image gallery view, I am using a ScrollView and LazyVStack. A ForEach iterates through all photos rows. For each row I am using the AspectRatioImageCollectionHStack that generates the layout with the help of the image aspect ratio passed as LayoutValueKeys:

```
struct AsyncImageCollectionView: View {

    @StateObject var photoLoader = PicsumPhotoLoader()
    let spacing: CGFloat = 1

    var body: some View {
        ScrollView {
            LazyVStack(spacing: spacing) {
                ForEach(photoLoader.photoRows, id: \.self) { row in

                    AspectRatioImageCollectionHStack(spacing: spacing) {
                        ForEach(row) { photo in
                            AspectRatioSizedAsyncImage(photo: photo)
                                .imageAspectRatio(photo.aspectRatio)
                        }
                    }
                }
            }
        }
    }
}

fileprivate struct AspectRatioSizedAsyncImage: View {

    let photo: PicsumPhoto
    let urlString = "https://picsum.photos/id/"
    @Environment(\.displayScale) var scale

    var body: some View {
        GeometryReader(content: { geometry in
            AsyncImage(url: url(in: geometry.size.width),
                scale: scale,
                transaction: .init(animation: .bouncy)) { phase in
                switch phase {
                    case .empty:
                        ZStack {
                            Color.gray
                            ProgressView()
                        }
                    case .success(let image):
                        image.resizable()
                            .scaledToFit()
                    case .failure(let error):
                        Text(error.localizedDescription)
                        // use placeholder for production app
                    @unknown default:
                        Text("default")
                }
            })
        })
        .imageAspectRatio(photo.aspectRatio)
    }
}

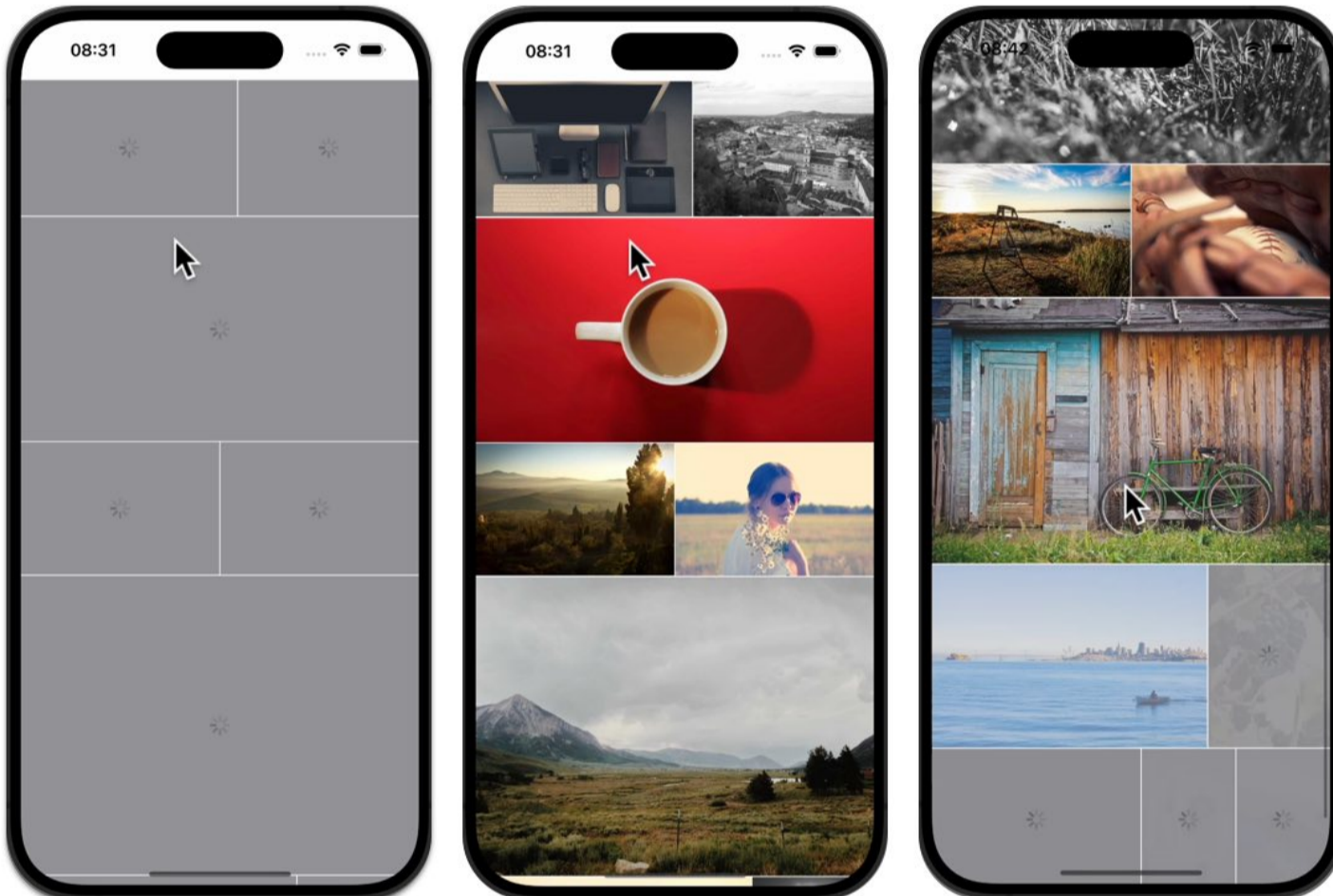
func url(in width: CGFloat) -> URL? {
```

```

        let imageWidth = width * scale
        let imageHeight = imageWidth / photo.aspectRatio
        let urlString = "\($baseUrlString)\($photo.id)/\($Int(imageWidth))/\($Int(imageHeight))"
        return URL(string: urlString)
    }
}

```

By using a LazyVStack, the rows are created dynamically as needed. This ensures efficient memory usage and smooth scrolling. The layout is properly sized, and images are loaded to the appropriate size, resulting in a seamless user experience. Each row uses a custom layout that is created row by row.



8. DYNAMIC DATA

8.1 FOREACH

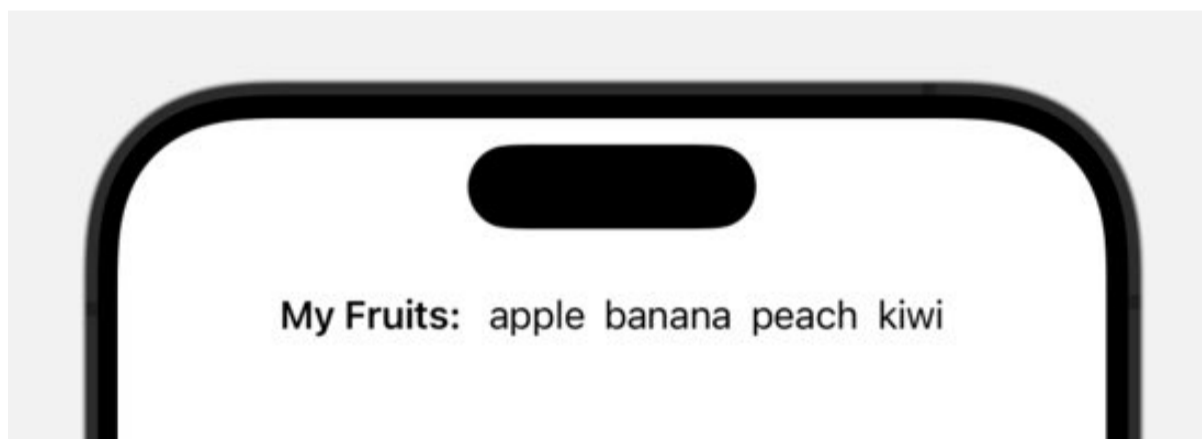
In this section, we'll dive into dynamic data in SwiftUI. We'll explore the `ForEach` container, which allows us to create a loop for a collection and show a view for each element in the collection. This is particularly useful when working with large data sets, such as arrays or dictionaries.

Arrays of Strings

Let's start with a simple example using an array of strings. Suppose we have an array called `fruits` containing strings like "apple," "banana," "peach," and "kiwi." If we want to display each of these fruits in a view, we can use the `ForEach` container:

```
struct ForEachExampleView: View {
    let fruits = ["apple", "banana", "peach", "kiwi"]
    var body: some View {
        HStack {
            Text("My Fruits: ")
                .bold()
            ForEach(fruits, id: \.self) { fruit in
                Text(fruit)
            }
        }
    }
}
```

Here, we pass in the `fruits` array as the data argument to `ForEach`. The `id` parameter tells SwiftUI how to identify each element in the array. In this case, we use `\.self` to use the string itself as the identifier. Now, SwiftUI will display each fruit as a `Text` view:



Ranges of Numbers

Now, let's say you want to display a range of numbers. You can use `ForEach` with a range like this:

```
ForEach(0..<10) { index in
    Text("index \(index)")
}
```

With ranges, you don't need to provide an id property because each number is unique and ForEach uses the numbers themselves as identifiers.

Enumerating and Identifying Elements

Sometimes, you might want to display both the index and the value from an array. You can use the enumerated() method to achieve this:

```
struct ForEachExampleView: View {
    let fruits = ["apple", "banana", "peach", "kiwi"]

    var body: some View {
        VStack {
            ForEach(Array(fruits.enumerated()),
                    id: \.element) { index, fruit in
                Text("\(index + 1). \(fruit)")
            }
        }
    }
}
```

I can call enumerated on the String array, but the resulting EnumeratedSequence<[String]> does not conform to RandomAccessCollection. Instead, you can convert it back to an array. I use the element itself (the string value) as the identifier with \.element.

In the ForEach closure, you then have access to the index and string values. In this example, I create a numbered list:



Handling Unordered Collections: Dictionaries and Sets

Moving on, let's discuss working with collections that are not naturally sorted, such as dictionaries and sets. These collections require some additional steps to ensure proper ordering and identification.

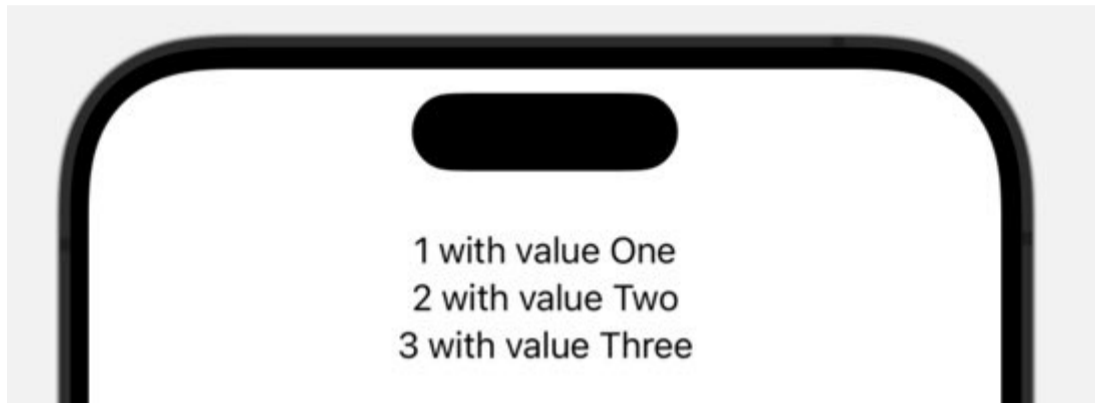
Dictionaries are not inherently ordered, so you need to sort them before using them in a ForEach. For dictionaries, we can sort them based on the keys using the sorted(by:) method. Here's an example:

```

struct UnorderedDataView: View {
    let numberDictionary: [Int: String] = [1: "One",
                                           2: "Two",
                                           3: "Three"]

    var body: some View {
        VStack {
            ForEach(numberDictionary.keys.sorted(), id: \.self) { key in
                Text("\(key) with value \(numberDictionary[key] ?? "")")
            }
        }
    }
}

```



In this case, we create an array from the keys of the numberDictionary and sort them using sorted(). We use \.self as the identifier, indicating that the key itself is the identifier. Now, we can display each key-value pair in the dictionary.

Sets, on the other hand, are not ordered collections. To work with sets, we can either sort them or create an array from them. Here's an example of using a sorted set:

```

struct UnorderedDataView: View {
    @State private var uniqueNumbers: Set = [1,2,3,4,5]
    var body: some View {
        VStack {
            ForEach(uniqueNumbers.sorted(), id: \.self) { number in
                Text("number \(number)")
            }
        }
    }
}

```

Alternatively, we can create an array from the set and then use the array in the ForEach container.

```

ForEach(Array(uniqueNumbers), id: \.self) { number in
    Text("number \(number)")
}

```

Both approaches technically work, but keep in mind that calling Array() does not preserve any particular order whereas sorted will result in always the same predefined order:

Sorted Set	Random Order
number 1	number 4
number 2	number 5
number 3	number 1
number 4	number 2
number 5	number 3

When working with dynamic data in SwiftUI, remember to always provide a unique identifier for each element using the `id` parameter in `ForEach`. Additionally, ensure that your collections are ordered when necessary. By doing so, you enable SwiftUI to manage and animate your views effectively, leading to a smooth and responsive user experience.

8.2 IDENTIFIABLE DATA

In the previous section, I've shown you how to use `ForEach` with basic types like strings or integers. But what happens when you're dealing with custom types? Let's say you're not working with strings but with a model type, such as `Fruit`. How do you handle that?

First, I'll define a custom type called `Fruit`:

```
struct Fruit {
    var name: String
    var color: String
    var isFavorite: Bool
}
```

To display these fruits, I'll use a `ForEach` loop with the `id` property:

```
let fruits = [
    Fruit(name: "Apple"),
    Fruit(name: "Banana"),
    Fruit(name: "Kiwi"),
    Fruit(name: "Cherry")
]

ForEach(fruits, id: \.name) { fruit in
    Text(fruit.name)
}
```

Instead of specifying the `id` every time, which can be cumbersome, you can conform to `Identifiable` directly in your model:

```
struct Fruit: Identifiable {
```

```
var id: String { name }
var name: String
var color: String
var isFavorite: Bool
}
```

Handling Unique Identifiers

Using the name as an identifier can lead to issues if you have fruits with the same name. When adding a new fruit, if two fruits have the same name, SwiftUI may treat them as the same view because they share the same identifier. This can lead to wired updating issues. To avoid this, use a unique identifier like UUID:

```
struct Fruit: Identifiable {
    let id = UUID()
    var name: String
    var color: String
    var isFavorite: Bool
}
```

Now, even if you add two fruits with the same name, SwiftUI recognizes them as separate entities because they have unique identifiers.

When working with dynamic data in SwiftUI, it's crucial to ensure each piece of data has a unique identifier. This allows SwiftUI to track and update the UI correctly. By using the Identifiable protocol and a unique identifier like UUID, you can avoid common pitfalls and ensure your views render as expected.

Remember, set the identifier when you create your data and make sure it stays the same throughout the lifecycle of the data. This will save you from unexpected behavior and keep your SwiftUI views in perfect order.

8.3 MAKING ENUMS IDENTIFIABLE

In this section, I'm going to walk you through how to make enums identifiable in SwiftUI. This is particularly useful when you're dealing with picker views, as they often present a set of options derived from enum cases to the user.

Example 1: Weather Picker

Let's start by creating a new file for our enum example view. We'll define an enum called WeatherType with a raw value of type String. Here's how it looks:

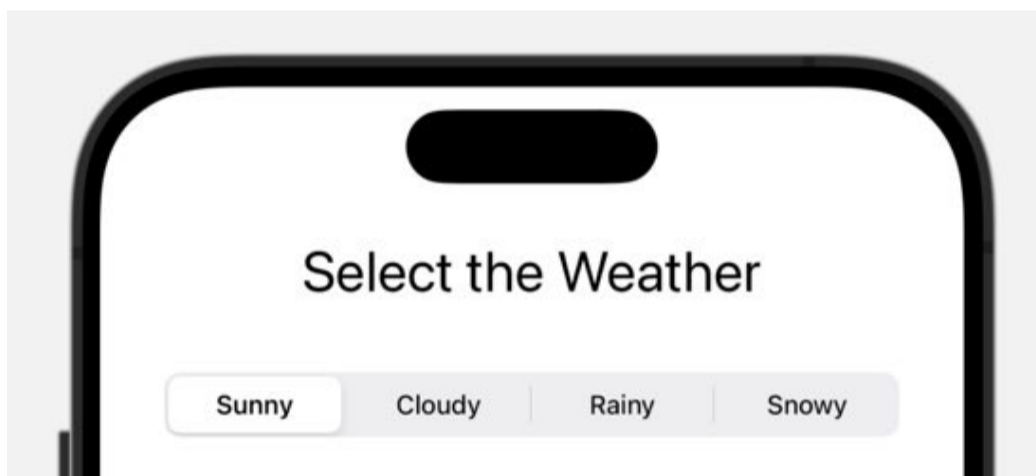
```
enum WeatherType: String {
    case sunny = "Sunny"
    case cloudy = "Cloudy"
    case rainy = "Rainy"
    case snowy = "Snowy"
}
```

Now, let's create a picker in our view. We'll need a `@State` property to keep track of the selected weather type. In our view, we'll add a picker with a title and use the segmented picker style to display all options at once:

```
struct EnumExampleView: View {
    @State private var selectedWeatherType = WeatherType.sunny

    var body: some View {
        VStack {
            Text("Select the Weather")
                .font(.title)

            Picker("Select", selection: $selectedWeatherType) {
                // Picker content will go here
            }
                .pickerStyle(.segmented)
        }
    }
}
```



Instead of manually typing out each case, we can leverage the `CaseIterable` protocol to access an array of all the enum cases.

```
enum WeatherType: String, CaseIterable {
    ...
}
```

This makes our code cleaner and more maintainable:

```
Picker("Select", selection: $selectedWeatherType) {
    ForEach(WeatherType.allCases) { type in
        Text(type.rawValue)
    }
}
```

However, to use `ForEach`, our enum needs to conform to the `Identifiable` protocol. We can satisfy this requirement by adding an `id` property that returns the raw value, which is unique for each case:

```
enum WeatherType: String, CaseIterable, Identifiable {
    var id: String { self.rawValue }
    ""
}
```

However, if you try to select an option in the picker you will notice that you cannot. To ensure that the picker can properly identify and select the options, we'll add a tag modifier to each option:

```
Picker("Select", selection: $selectedWeatherType) {
    ForEach(WeatherType.allCases) { type in
        Text(type.rawValue)
            .tag(type)
    }
}
.pickerStyle(.segmented)
```

Example 2: Eating Preference Enum

As a second example, let's consider an EatingPreference enum without raw values. We'll make it CaseIterable and Identifiable as well:

```
enum EatingPreference: CaseIterable, Identifiable {
    case sweet
    case spicy
    case salty

    var id: Self { return self }

    var displayName: String {
        switch self {
            case .sweet: "Sweet"
            case .spicy: "Spicy"
            case .salty: "Salty"
        }
    }
}
```

Notice that for the id property, we're returning the enum instance itself (self). This is possible because enum cases are unique and hashable.

Now, let's use this enum in another picker:

```
@State private var selectedEatingPreference: EatingPreference = .salty

Picker("Eating", selection: $selectedEatingPreference) {
    ForEach(EatingPreference.allCases) { preference in
        Text(preference.displayName)
    }
}
.pickerStyle(SegmentedPickerStyle())
```

In this case, we don't need to add a tag because the enum itself is used for identification, which simplifies the code.

8.4 FOREACH WITH BINDING

In this section, we're going to explore how to add binding to a `ForEach` loop in SwiftUI. This feature became available starting with iOS 15, and it's a game-changer for working with dynamic data. Imagine you have an array of fruits, and you want to let users pick a color for each fruit and toggle their favorite status. Let's dive into how you can achieve this with SwiftUI's `ForEach` and binding.

First, you need to ensure your data model is mutable. Here's what your **Fruit** model should look like:

```
struct Fruit: Identifiable {  
  
    var name: String  
    var color: Color  
    var isFavorite: Bool  
    let id: UUID  
  
    init(name: String,  
         color: Color = .yellow,  
         isFavorite: Bool = false) {  
        self.name = name  
        self.color = color  
        self.isFavorite = isFavorite  
        self.id = UUID()  
    }  
}
```

Notice that `id` is a constant (`let`) because it should not change, while `name`, `color`, and `isFavorite` are variables (`var`) because you want to allow the user to modify these properties.

Now, let's create a SwiftUI view where we show a list of fruits. You'll start with an array of `Fruit` objects and use a `ForEach` to iterate over them:

```
struct BindingForEachExampleView: View {  
    @State var fruits = [Fruit(name: "apple"),  
                        Fruit(name: "banana"),  
                        Fruit(name: "cherry"),  
                        Fruit(name: "kiwi")]  
  
    var body: some View {  
        VStack(alignment: .leading) {  
            Text("What colors have these Fruits?")  
                .font(.title)  
  
            ForEach($fruits) { $fruit in  
                // edit fruit information  
            }  
        }  
    }  
}
```

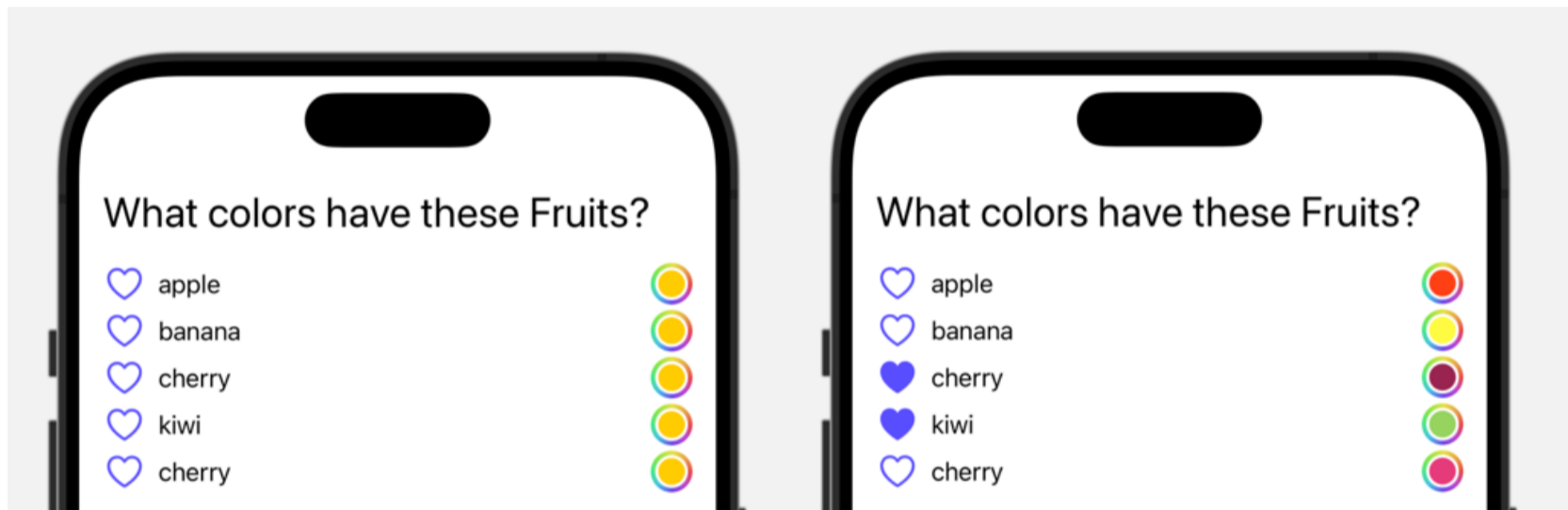

In the `ForEach`, you use `$fruits` to pass a binding to each `Fruit` object. This allows you to create a `TextField` and a `ColorPicker` that directly bind to the name and color properties of each `Fruit`. The `Toggle` binds to the `isFavorite` property.

```
ForEach($fruits) { $fruit in
  HStack {
    Toggle("isFavorite",
           isOn: $fruit.isFavorite)
      .toggleStyle(FavoriteToggleStyle())

    TextField("Fruit", text: $fruit.name)

    ColorPicker(selection: $fruit.color) {
      Text("Color")
    }
      .labelsHidden()
  }
}
```

Any changes you make in the `TextField` or `ColorPicker` will be reflected immediately in the `fruits` array. This is because you're using bindings, which create a two-way connection between the UI elements and the data.



You can use bindings with `ForEach` and `List`. Using `List` for the above example:

```
List($fruits) { $fruit in
  HStack {
    Toggle("isFavorite",
           isOn: $fruit.isFavorite)
      .toggleStyle(FavoriteToggleStyle())
    ""
  }
}
```

You can customize the **toggle style** to make it more visually appealing. Here's an example of a custom `FavoriteToggleStyle`:

```

struct FavoriteToggleStyle: ToggleStyle {
    func makeBody(configuration: Configuration) -> some View {
        Button(action: {
            configuration.isOn.toggle()
        }) {
            Image(systemName: configuration.isOn ? "heart.fill" : "heart")
                .foregroundColor(.accentColor)
                .font(.system(size: 24))
        }
    }
}

```

8.5 LAZYVSTACK AND LAZYHSTACK

When you're aiming to boost your app's performance, making it snappy and battery-efficient is key. That's where lazy containers come into play. Unlike their name suggests, they're not lounging around all day. They're hard at work, but only when they need to be—specifically when the user is interacting with the screen.

Imagine you have an app with a gallery of images. You wouldn't want to load every single image if the user is only viewing a few at a time. That's where SwiftUI's lazy containers come in handy. There are four main types: **LazyVStack**, **LazyHStack**, **LazyVGrid**, and **LazyHGrid**. Even SwiftUI's **List** is lazy by default.

Understanding Lazy Loading

Let's dive into an example. I'll use a simple array of emojis to illustrate how lazy loading works. Picture a **ScrollView** filled with emojis—there's a lot of them, and they certainly won't all fit on the screen at once. This is a perfect scenario for a **ScrollView**.

Here's how you might set it up:

```

struct LazyVStackExampleView: View {
    let emojis = Emoji.examples()

    var body: some View {
        ScrollView {
            LazyVStack {
                ForEach(emojis) { emoji in
                    EmojiView(emoji: emoji)
                }
            }
        }
    }
}

```

In this example, **EmojiView** is a custom view that takes an emoji character and displays it. The magic happens when you start scrolling. As new emojis come into view, SwiftUI initializes and renders them just in time. This is similar to how **UITableView** works in **UIKit** with reusable cells.

When Views Are Created and Appear

To understand when each emoji view is created and appears, you can add print statements in two places:

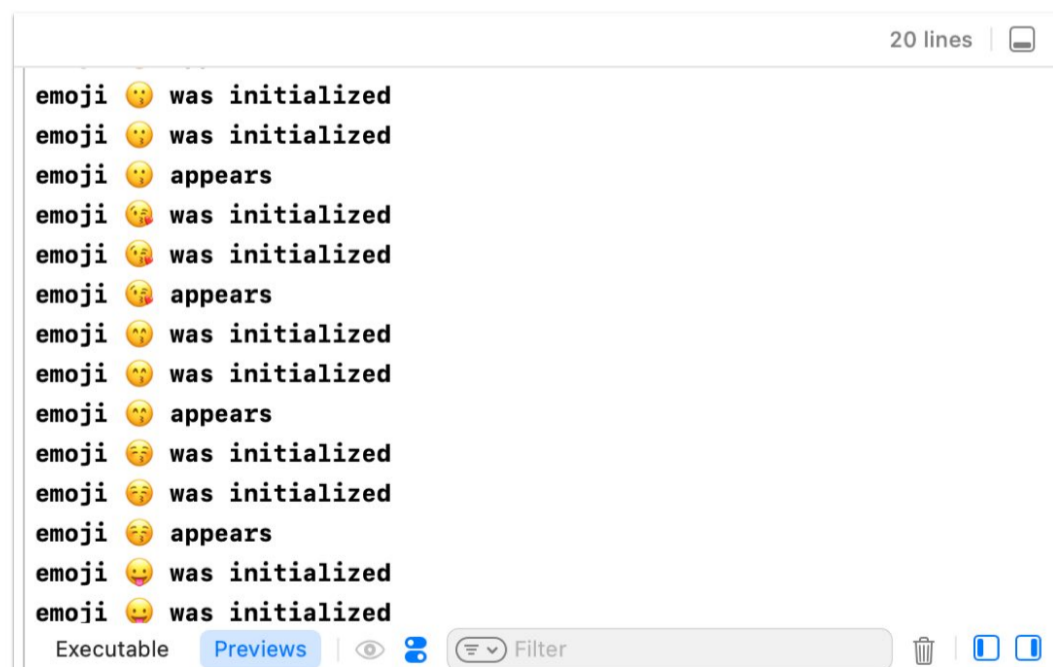
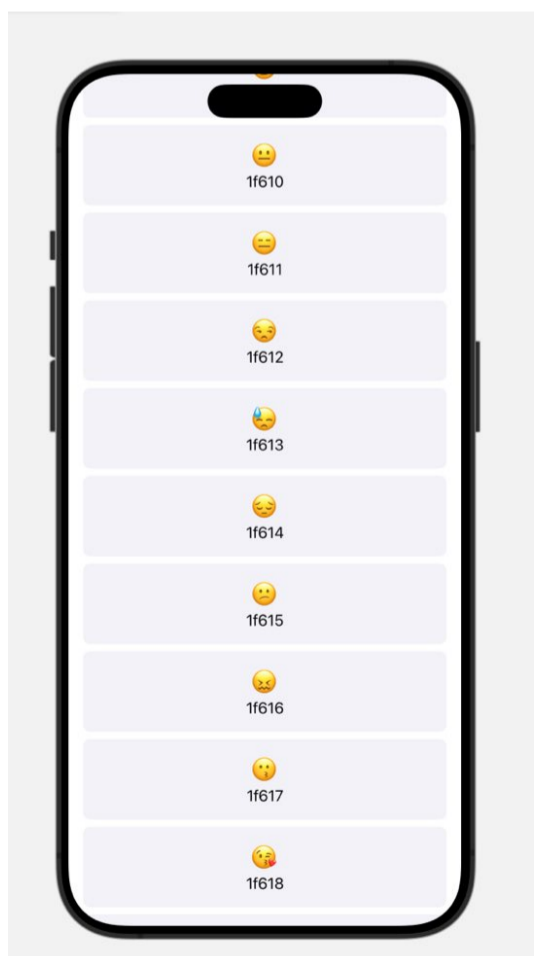
- in the initializer of EmojiView
- In the `.onAppear` modifier

```
struct EmojiView: View {
    let emoji: Emoji

    init(emoji: Emoji) {
        print("emoji \(emoji.emojiSting) was initialized")
        self.emoji = emoji
    }

    var body: some View {
        GroupBox {
            Text(emoji.emojiSting)
                .font(.title)
            Text(emoji.valueString)
                .frame(maxWidth: .infinity)
        }
        .onAppear {
            print("emoji \(emoji.emojiSting) appears")
        }
    }
}
```

When you run this code, you'll notice that the views are created and appear only as needed. This is the essence of lazy loading—views are not created all at once but rather as the user scrolls through the content.



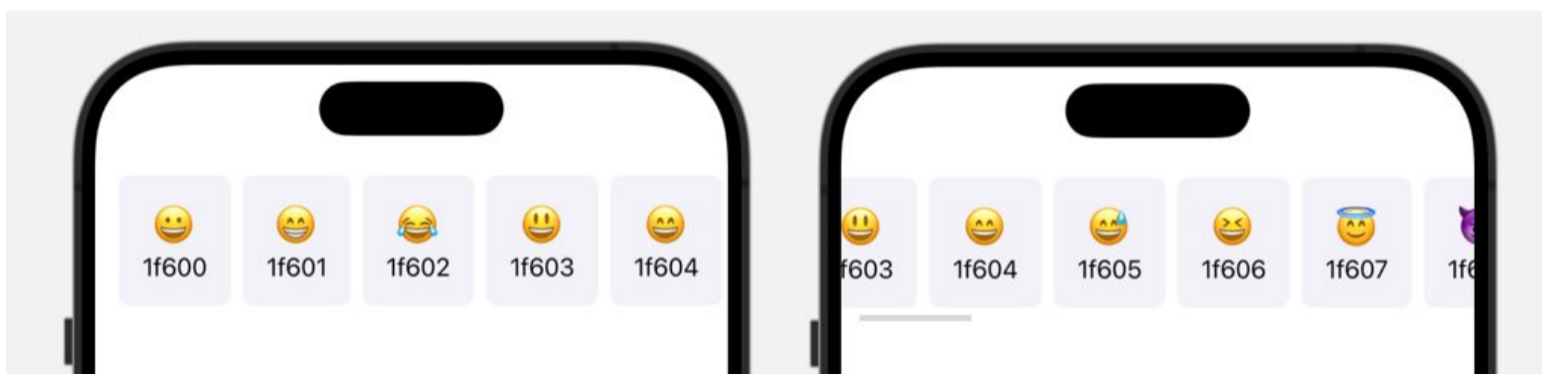
LazyHStack Example

Now, let's look at a horizontal scrolling example using LazyHStack. The setup is similar to LazyVStack, but you'll need to specify the scroll direction:

```
struct LazyHStackExampleView: View {
    let emojis = Emoji.examples()

    var body: some View {
        ScrollView(.horizontal) {
            LazyHStack {
                ForEach(emojis) { emoji in
                    EmojiView(emoji: emoji)
                }
            }.padding()
        }
    }
}
```

With LazyHStack, you can scroll horizontally through your content, and just like with LazyVStack, SwiftUI will only create views as they're needed.



Using lazy stacks is crucial when dealing with large datasets, complex calculations, or resource-intensive operations like loading images. Whether the images are local or fetched from the web, lazy loading ensures that your app remains efficient and responsive.

Remember, the goal is to create views only when necessary. This not only improves performance but also conserves memory usage, leading to a better user experience. So, when you're faced with a long list of data or a gallery of images, think lazy. It's the smart way to load content dynamically.

8.6 Lazily Showing Images

When dealing with images in your SwiftUI app, especially when you have a large number of them, it's crucial to consider performance. In this section, I'll guide you through the process of displaying images efficiently, focusing on lazy loading to enhance your app's performance.

Loading Images from Assets

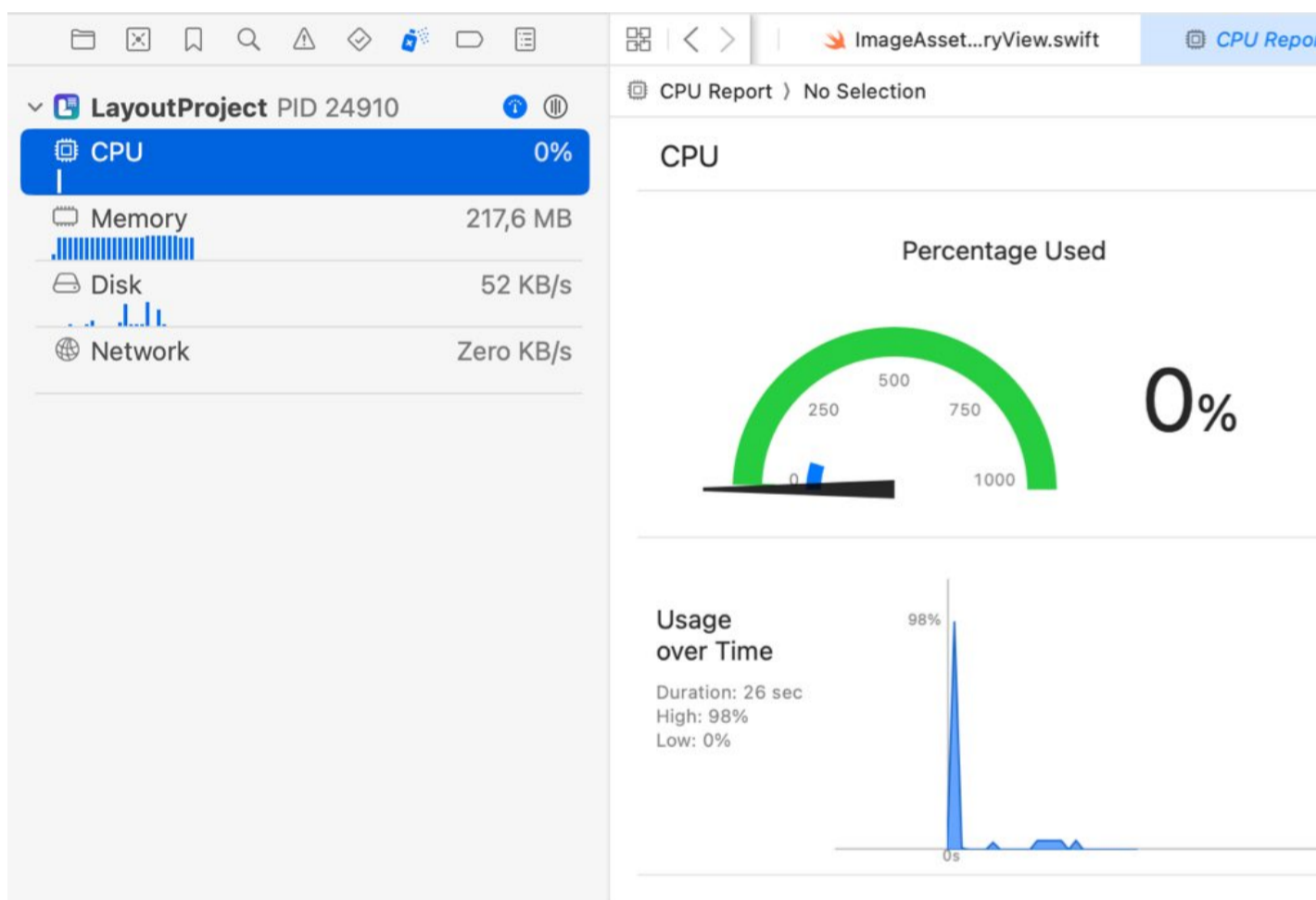
Let's start with images you've added to your asset catalog. I'll demonstrate this with an ImageGalleryView. Here's a simple setup using a ScrollView and a VStack:

```

struct ImageAssetsGalleryView: View {
    let inspirations = NatureInspiration.examples()
    var body: some View {
        ScrollView {
            VStack {
                ForEach(inspirations) { inspiration in
                    Image(inspiration.imageName)
                        .resizable()
                        .scaledToFit()
                }
            }
        }
    }
}

```

In this example, all images are loaded and rendered as soon as the view appears. This can cause a significant spike in CPU usage and memory consumption. If you run this in your app and check the debug area, you might see a CPU spike to 98% and memory usage of 200MB. This is not ideal, especially if you have more images or larger data sets.



Using LazyVStack for Better Performance

To improve performance, replace `VStack` with `LazyVStack`. This lazy container only loads and renders images as they come into view, which is much more efficient:

```

struct ImageAssetsGalleryView: View {
    let inspirations = NatureInspiration.examples()

    var body: some View {
        ScrollView {
            LazyVStack {
                ForEach(inspirations) { inspiration in

```



```

        GeometryReader(content: { geometry in
            AsyncImage(url: url(in: geometry.size.width),
                scale: 3,
                transaction: .init(animation: .easeIn)) { phase in
                switch phase {
                case .empty:
                    ZStack {
                        Color(white: 0.8)
                        ProgressView()
                    }
                case .success(let image):
                    image
                        .resizable()
                        .scaledToFit()
                case .failure(let error):
                    Text(error.localizedDescription)
                    // use placeholder for production app
                @unknown default:
                    fatalError()
                }
            })
        }.aspectRatio(aspectRatio, contentMode: .fit)
    }

    func url(in width: CGFloat) -> URL? {
        let imageWidth = Int(width * scale)
        let imageHeight = Int(width * scale / aspectRatio)
        let urlString = "https://picsum.photos/id/\(photo.id)/\(imageWidth)/\
(imageHeight)"
        return URL(string: urlString)
    }
}

```

This view takes a `PicumPhoto` object and loads the image from the provided URL. It uses a placeholder color while the image is loading.

Next, use this view in a `ScrollView` with a `LazyVStack`:

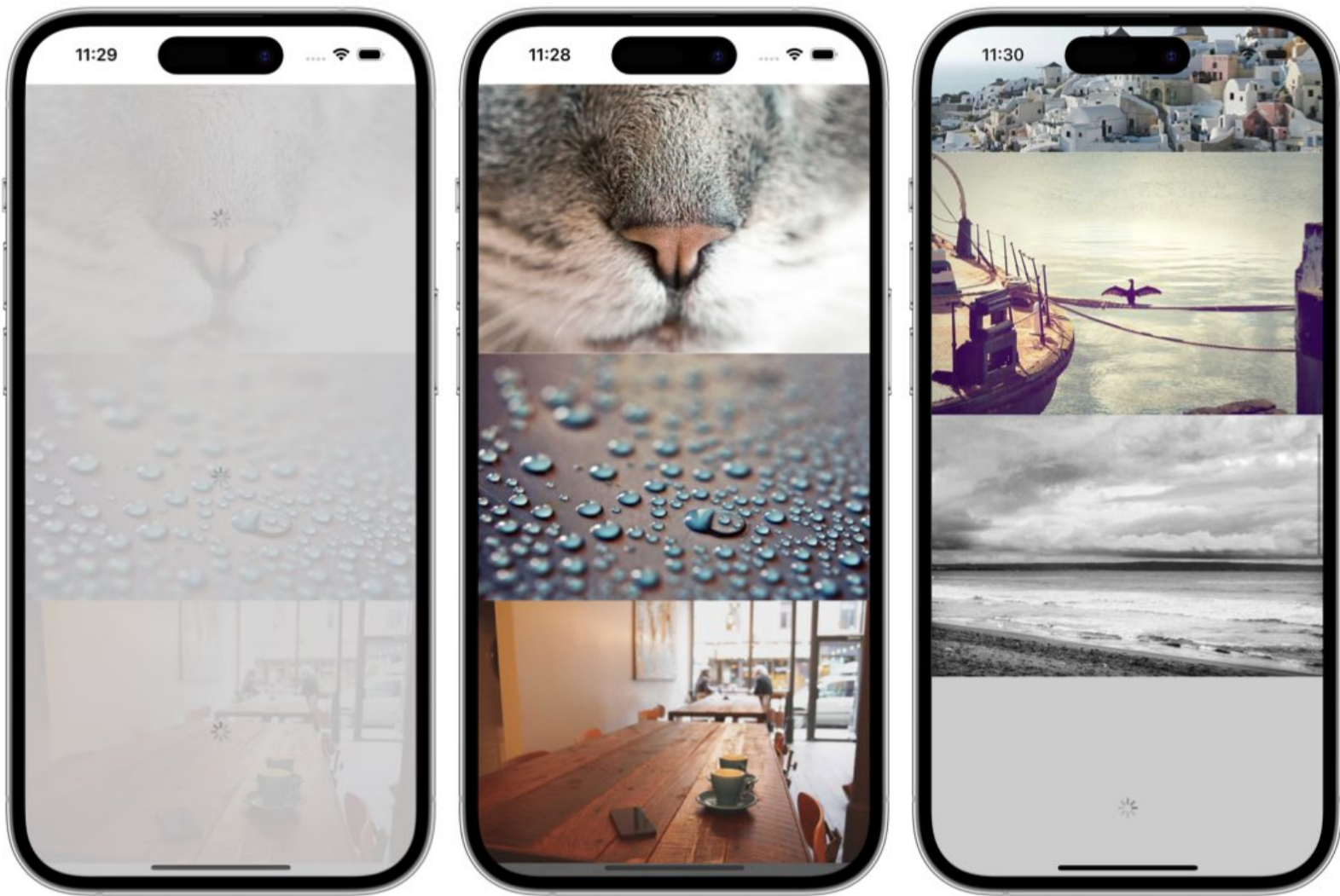
```

struct ImageAsyncGalleryView: View {
    @StateObject private var photoLoader = PicsumPhotoLoader(page: 3,
                                                            photosPerPage: 20)

    var body: some View {
        ScrollView {
            LazyVStack(spacing: 0) {
                ForEach(photoLoader.photos) {
                    PicsumPhotoImage(photo: $0)
                }
            }
        }
    }
}

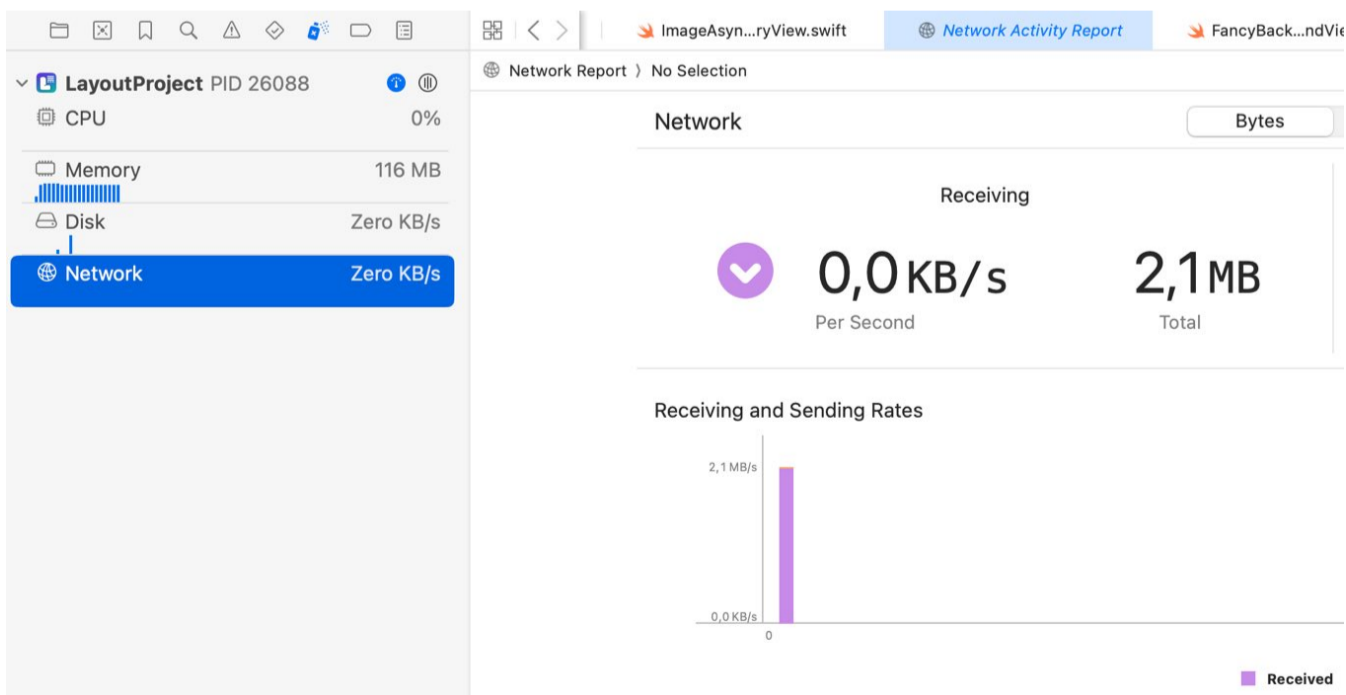
```

By using LazyVStack, you ensure that images are only loaded as they come into view, which is much more efficient than loading all images at once.

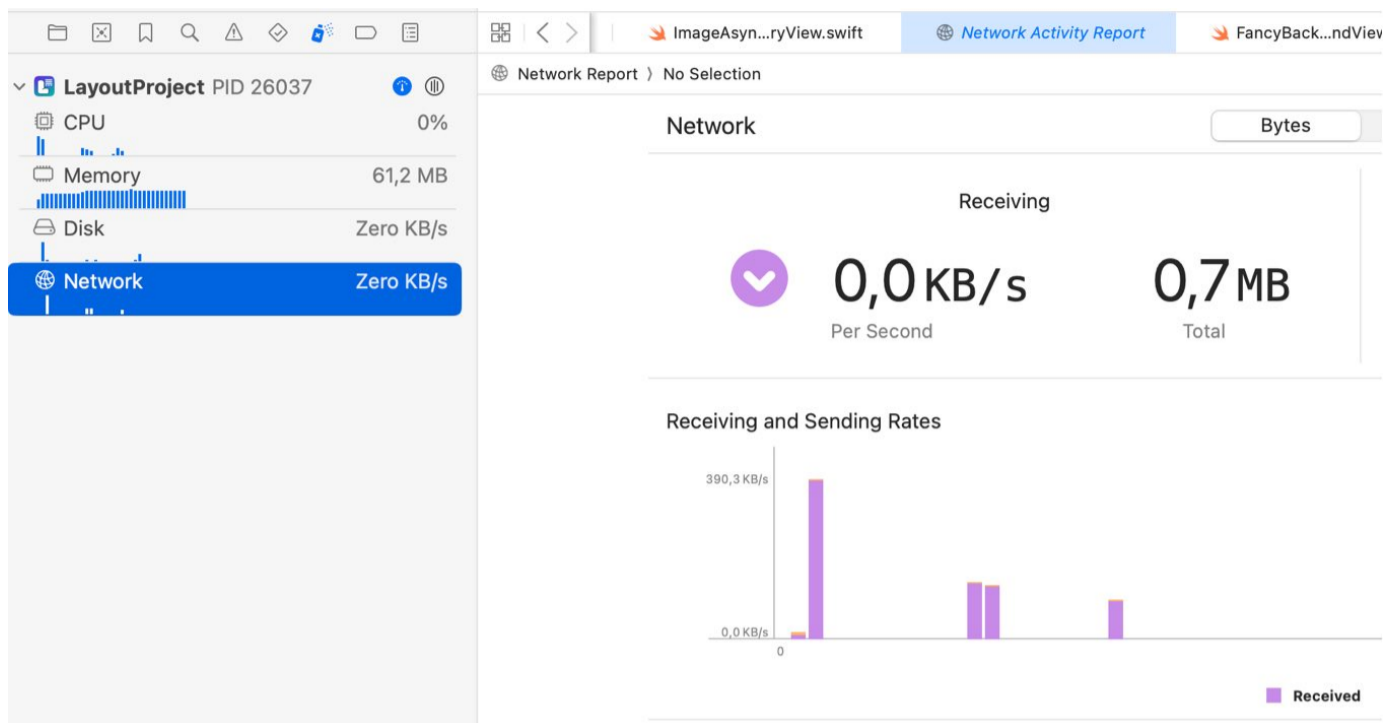


Performance Implications

When you run the app and monitor the debug area, you'll see the performance benefits of lazy loading. With a VStack, you might have a CPU spike and a significant amount of data downloaded at once:



With a LazyVStack, the CPU usage and data downloaded (here 0.3MB instead of 2MB) are much lower during launch, as images are only loaded when necessary. Once you scroll further more images are loaded, which you can see from additional peaks in the debug network graph:



Remember, if users scroll up and down frequently, AsyncImage may re-download images that have left the view. You can address this by implementing a caching policy or accepting this behavior, as most users tend to scroll in one direction.

By only showing and downloading images as needed, you greatly improve the performance of your app. This approach reduces network consumption, minimizes CPU usage, and ensures smooth scrolling and fast image loading during app launch.

In summary, lazy loading images is a powerful strategy for optimizing the performance of your SwiftUI app. It's a simple change that can have a significant impact on how your app feels and behaves, especially when dealing with a large number of images or dynamic data.

8.7 Smooth ScrollViews with Images

When dealing with async images in a ScrollView, it's crucial to ensure a smooth user experience. One key aspect is to provide an initial size for image placeholders. Let's dive into what can go wrong and how to fix it.

The Problem with Stuttering Image Galleries

Imagine you have an ImageGalleryView. You might notice that as images load, the layout shifts, causing a stuttering effect.

```
struct ImageAsyncGalleryView: View {
    @StateObject private var photoLoader = PicsumPhotoLoader(page: 3,
                                                              photosPerPage: 20)
    @Environment(\.displayScale) var scale

    var body: some View {
```

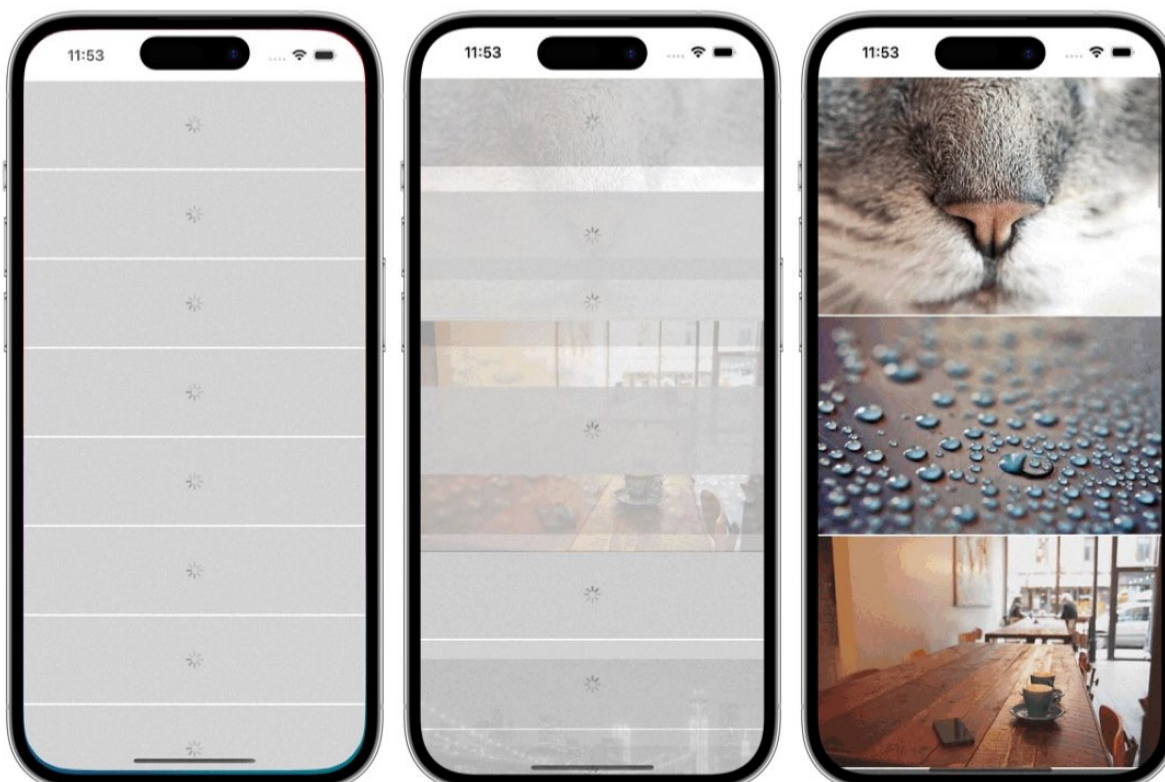
```

GeometryReader { geometry in
  ScrollView {
    LazyVStack(spacing: 2) {
      ForEach(photoLoader.photos) { photo in
        AsyncImage(url: url(in: geometry.size.width, photo: photo),
          scale: 3,
          transaction: .init(animation: .easeIn)) { phase in
          switch phase {
            case .empty:
              ZStack {
                Color.gray
                ProgressView()
              }.frame(height: 100)
            case .success(let image):
              image.resizable()
                .scaledToFit()
            case .failure(let error):
              Text(error.localizedDescription)
                .frame(height: 50)
          }
        }
      }
    }
  }
}

func url(in width: CGFloat, photo: PicsumPhoto) -> URL? {
  let imageWidth = Int(width * scale)
  let imageHeight = Int(width * scale / photo.aspectRatio)
  let urlString = "https://picsum.photos/id/\(photo.id)/\(imageWidth)/\
(imageHeight)"
  return URL(string: urlString)
}
}

```

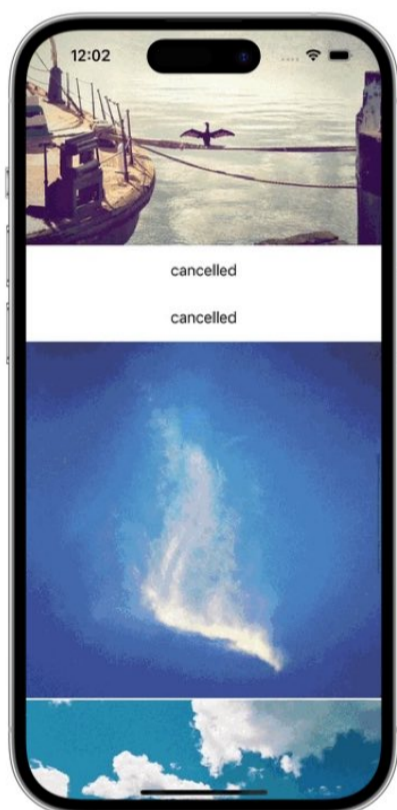
This happens because the placeholder sizes don't match the final images, leading to constant rearrangement as new images load. Initially 8 placeholder images with a height of 100 points fit on screen. When the images are finished loading only 3 images fit on screen:



This stuttering effect becomes worse when you scroll further down as the smooth scrolling is interrupted by images being resized and moving in the ScrollView.

AsyncImage loads the image as long as the view is visible. When the image disappears before the download is finished, an error occurs. You get this in the failure case of AsyncImage:

```
AsyncImage(url: url(in: geometry.size.width, photo: photo)) { phase in
  switch phase {
    case .empty:
      ""
    case .success(let image):
      ""
    case .failure(let error):
      Text(error.localizedDescription)
        .frame(height: 50)
    @unknown default:
      fatalError()
  }
}
```



During the initial loading, I was loading too many images. Once the images are displayed, SwiftUI discards the unnecessary images. They disappear from the screen and the download tasks are canceled if they did not finish before. If you scroll through the image gallery, you can find these canceled rows.

The Solution for Smooth Scrolling

To prevent stuttering, use placeholders that match the aspect ratio of the final images. This way, images can fade in without affecting the layout.

For example, I can use the size of the images that I get from the server:

```
struct PicsumPhoto: Codable, Identifiable, Hashable {
    let id: String
    let author: String
    let width: CGFloat
    let height: CGFloat
    let url: String
    let downloadUrl: String

    var aspectRatio: CGFloat {
        width / height
    }
}

...
}
```

And use it in the above example to size the placeholder:

```
AsyncImage(url: url(in: geometry.size.width, photo: photo)) { phase in
    switch phase {
        case .empty:
            ZStack {
                Color.gray
                ProgressView()
            }
            //.frame(height: 100)
            .aspectRatio(photo.aspectRatio, contentMode: .fit)
        case .success(let image):
            image.resizable()
                .scaledToFit()
        case .failure(let error):
            Text(error.localizedDescription)
                .frame(height: 50)
    }
}
```

The downloaded images also use the aspect ratio to fetch the appropriately sized image:

```
func url(in width: CGFloat) -> URL? {
    let imageWidth = Int(width * scale)
    let imageHeight = Int(width * scale / aspectRatio)
    let urlString = "https://picsum.photos/id/\(photo.id)/\(imageWidth)/\
(imageHeight)"
    return URL(string: urlString)
}
```

Smooth Image Gallery with Reusable Component

You can also write a reusable image component. In the following example, I am not using a `GeometryReader` outside the `ScrollView`:

```
struct ImageAsyncGalleryView: View {
    @StateObject private var photoLoader = PicsumPhotoLoader(page: 3,
                                                                photosPerPage: 20)
```

```

var body: some View {
    ScrollView {
        VStack(spacing: 0) {
            ForEach(photoLoader.photos) {
                PicsumPhotoImage(photo: $0)
            }
        }
    }
}

```

Instead, I use the GeometryReader inside the PicsumPhotoImage component to read the image area. I am adding the aspect ratio with the photo value to the GeometryReader. Inside is the AsyncImage with the placeholder. All states of the async image use the same aspect ratio and have thus the same size:

```

struct PicsumPhotoImage: View {
    let photo: PicsumPhoto
    let aspectRatio: CGFloat
    @Environment(\.displayScale) var scale

    init(photo: PicsumPhoto, aspectRatio: CGFloat? = nil) {
        self.photo = photo
        self.aspectRatio = aspectRatio ?? photo.aspectRatio
    }

    var body: some View {
        GeometryReader(content: { geometry in
            AsyncImage(url: url(in: geometry.size),
                scale: 3,
                transaction: .init(animation: .easeIn)) { phase in
                switch phase {
                case .empty:
                    ZStack {
                        Color(white: 0.8)
                        ProgressView()
                    }
                case .success(let image):
                    image
                        .resizable()
                        .scaledToFit()
                case .failure(let error):
                    Text(error.localizedDescription)
                    // use placeholder for production app
                @unknown default:
                    fatalError()
                }
            })
        })
        .aspectRatio(photo.aspectRatio, contentMode: .fit)
    }

    func url(in size: CGSize) -> URL? {
        let imageWidth = Int(size.width * scale)
        let imageHeight = Int(size.height * scale)
        let urlString = "https://picsum.photos/id/\(photo.id)/\(imageWidth)/\
(imageHeight)"
        return URL(string: urlString)
    }
}

```

Caching and Async Image Pitfalls

The AsyncImage view doesn't handle caching, and it can cancel image loads if the view disappears before loading completes. To address this, you can create a custom CachedAsyncImage that implements caching.

```
struct CacheAsyncImage<Content>: View where Content: View {

    private let url: URL?
    private let scale: CGFloat
    private let transaction: Transaction
    private let content: (AsyncImagePhase) -> Content

    init(url: URL?,
         scale: CGFloat = 1.0,
         transaction: Transaction = Transaction(),
         @ViewBuilder content: @escaping (AsyncImagePhase) -> Content){
        self.url = url
        self.scale = scale
        self.transaction = transaction
        self.content = content
    }

    var body: some View{
        if let url, let cached = ImageCache[url] {
            let _ = print("cached: \(url.absoluteString)")
            content(.success(cached))
        } else {
            let _ = print("request: \(url?.absoluteString ?? "")")
            AsyncImage(
                url: url,
                scale: scale,
                transaction: transaction) { phase in
                cacheAndRender(phase: phase)
            }
        }
    }

    func cacheAndRender(phase: AsyncImagePhase) -> some View{
        if case .success (let image) = phase, let url {
            ImageCache[url] = image
        }
        return content(phase)
    }
}
```

You'll need a caching mechanism to store and retrieve images efficiently. Here's a simplified version of an image cache:

```
class ImageCache {

    struct ImageData {
        let image: Image
        let timeStamp: Date
    }

    static private var cache: [URL: ImageData] = [:]
    static private let maxCacheNumber = 10
    static subscript(url: URL) -> Image?{
```

```

        get{
            ImageCache.cache[url]?.image
        }
        set{
            if ImageCache.cache.count >= maxCacheNumber,
                let first = ImageCache.cache.sorted(by: { $0.value.timeStamp <
                    $1.value.timeStamp }).first?.key {
                    print("remove")
                    ImageCache.cache.removeValue(forKey: first)
                }
            if let newValue {
                ImageCache.cache[url] = ImageData(image: newValue, timeStamp: Date())
            }
        }
    }
}

```

Replace `AsyncImage` with `CachedAsyncImage` in your views to benefit from caching.

```

// Usage in your SwiftUI view
CachedAsyncImage(url: url(for: size)) { image in
    image.resizable()
} placeholder: {
    Color.gray
}

```

While the provided cache example is basic, you might want to create a more robust solution using `URLSession` and `DataTask`. Always monitor your app's memory and network usage, and refine your approach for the best performance.

8.8 LAZYVGRID AND LAZYHGRID

In this section, we're going to dive into grid layouts using SwiftUI's **LazyVGrid** and **LazyHGrid**. Unlike a one-dimensional list, grids allow us to display content in multiple columns or rows, which is perfect for something like an image gallery. We'll start with a simple example using emoji data to get a feel for how these grids behave before moving on to more complex scenarios.

Understanding Lazy Grids with Emoji Data

First, let's set up an example view using the emoji data provided in the emoji examples area. We'll begin with a **LazyVGrid**. To define a grid, you need to specify the number of columns, their sizing, alignment, spacing, and the content itself. Here's how you can define a `LazyVGrid`:

```

struct LazyVGridExampleView: View {
    let emojis = Emoji.examples()

    var body: some View {
        ScrollView {
            LazyVGrid(columns: [GridItem(.adaptive(minimum: 80), spacing: 10)],

```

```

        alignment: .center,
        spacing: 10) {

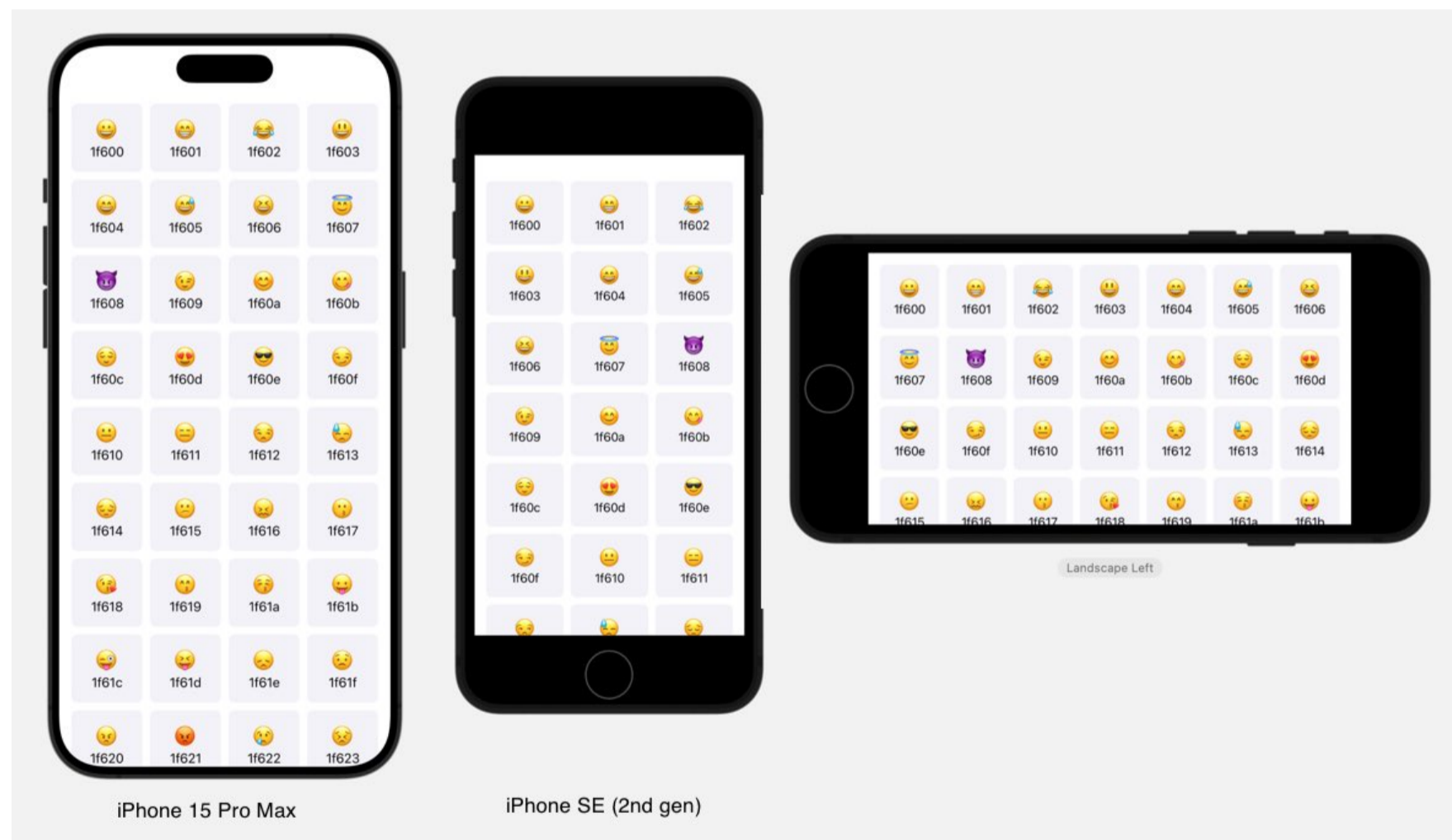
        ForEach(emojis) {
            EmojiView(emoji: $0)
        }
    }
    .padding()
}
}
}

```

In this example, columns is an array of GridItem that you need to define. There are three types of GridItem you can use: **.fixed**, **.flexible**, and **.adaptive**.

Adaptive Grid Items

Adaptive grid items are particularly useful for creating responsive layouts. You set a minimum size, and the grid will automatically fit as many items as possible across the available space. For instance `.adaptive(minimum: 80)` will ensure that each item is at least 80 points wide. It will adapt to the screen width and device orientation:



Fixed Grid Items

With `.fixed`, you set an exact size for each grid item. If you specify `.fixed(150)`, each column will be exactly 150 points wide. The number of elements in the columns array determines the number of columns shown. In the following, I am showing 5 columns with the same 150 points column width:


```

struct LazyVGridFixedExampleView: View {
    let emojis = Emoji.examples()
    let columns = Array(repeating: GridItem(.fixed(150)), count: 5)

    var body: some View {
        ScrollView([.horizontal, .vertical]) {
            LazyVGrid(columns: columns,
                    alignment: .center,
                    spacing: 10, content: {
                ForEach(emojis) {
                    EmojiView(emoji: $0)
                }
            })
            .padding()
        }
    }
}

```

The resulting grid does not fit on the screen. Therefore, I used a ScrollView that scrolls in vertical and horizontal directions.



Flexible Grid Items

Flexible grid items allow for a range of sizes. For example, `.flexible(minimum: 50, maximum: 200)` means that the grid item can size itself flexibly within the given range, depending on the available space. The following columns will show 2 columns with the first column being larger than the second:

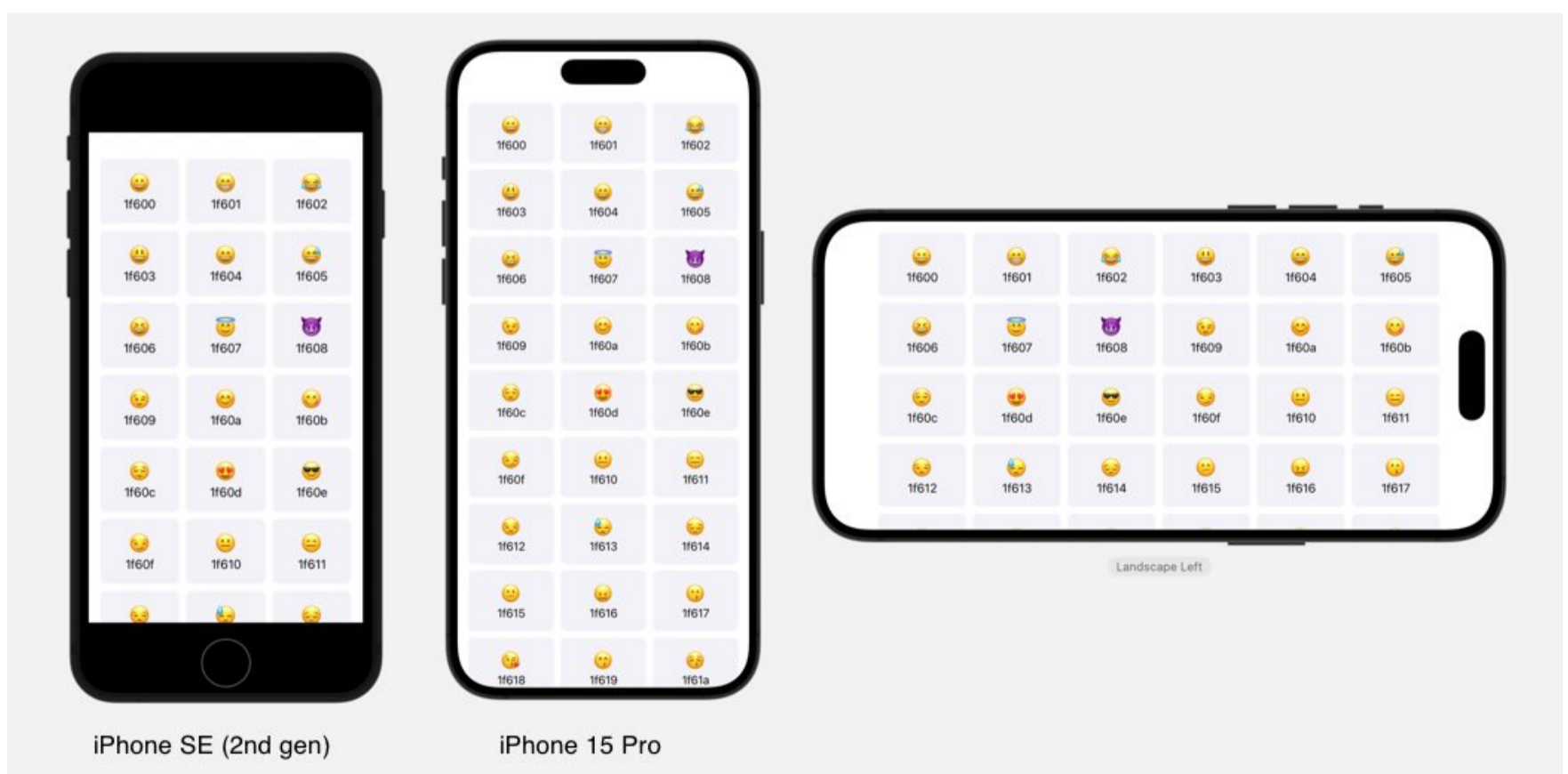
```
let columns = [GridItem(.flexible(minimum: 100, maximum: 500)),
               GridItem(.flexible(minimum: 50, maximum: 200))]
```

This will always use 2 columns even if you could fit more on screen. You can also use GeometryReader to decide how many columns you want to show:

```
struct LazyVGridFlexibleExampleView: View {
    let emojis = Emoji.examples()
    var body: some View {
        GeometryReader { geometry in
            ScrollView {
                LazyVGrid(columns: columns(width: geometry.size.width),
                          alignment: .center,
                          spacing: 10, content: {
                    ForEach(emojis) {
                        EmojiView(emoji: $0)
                    }
                })
            }
        }.padding()
    }
}

func columns(width: CGFloat) -> [GridItem] {
    let columnCount = width < 450 ? 3 : 6
    return Array(repeating: GridItem(.flexible(minimum: 70, maximum: 150),
                                           spacing: 10,
                                           alignment: .top),
                 count: columnCount)
}
```

I use the GeometryReader width to decide how many columns to show. For a smaller screen width than 450 points, I am only showing 3 columns. For larger areas like in landscape mode, I show 6 columns.



Adjusting Grid Spacing and Alignment

You can adjust the spacing between rows in a LazyVGrid by setting the spacing parameter.

```
LazyVGrid(columns: columns,  
          alignment: .center,  
          spacing: 10) {  
  ...  
}
```

If you want to change the spacing between columns, you need to adjust the spacing within the GridItem.

```
GridItem(.adaptive(minimum: 80), spacing: 10)
```

To prevent your content from touching the edges of the screen, it's a good idea to add padding around your grid: `.padding()`

```
ScrollView {  
  LazyVGrid(columns: columns) {  
    ...  
  }  
  .padding()  
}
```

Exploring LazyHGrid

Now, let's switch gears and look at LazyHGrid, which is similar to LazyVGrid but for horizontal grids. Instead of columns, we define rows. The principles are the same but keep in mind that the fixed size now applies to the height of each row.

Here's a basic setup for a LazyHGrid, where I show 3 rows with a fixed height of 100 points:

```
struct LazyHGridExampleView: View {  
  let emojis = Emoji.examples()  
  let spacing: CGFloat = 10  
  
  var rows: [GridItem] {  
    Array(repeating: GridItem(.fixed(100), spacing: spacing),  
          count: 3)  
  }  
  
  var body: some View {  
    ScrollView(.horizontal) {  
      LazyHGrid(rows: rows, spacing: spacing) {  
        ForEach(emojis) { emoji in  
          Text(emoji.emojiString)  
            .font(.system(size: 45))  
        }  
      }  
    }  
  }  
}
```

```

        .padding()
        .frame(maxHeight: .infinity)
        .background(RoundedRectangle(cornerRadius: 15)
            .fill(Color(white: 0.9)))
    }
}
.padding(spacing * 2)
}
}
}

```



Lazy grids in SwiftUI are incredibly versatile and can be tailored to fit a wide range of layout needs. Whether you're working with fixed, flexible, or adaptive grid items, you can create layouts that look great on any device and orientation. In the next lesson, we'll take these concepts further by creating an image grid that will showcase the true power of lazy grids in SwiftUI.

8.9 IMAGE GALLERY WITH LAZYVGRID AND LAZYHGRID

In this section, I'll show you how to create an image gallery using LazyVGrid in SwiftUI. Let's start by creating a new view called ImageGridGalleryView in the image gallery folder.

Image Gallery with Images from the Assets Catalog

First, we'll use images from the assets. Remember the natureInspirations examples array? We'll be using that. With LazyVGrid, we have three layout options: adaptive, flexible, or fixed. For an image gallery, fixed sizes aren't ideal, so let's opt for adaptive to take advantage of its flexibility.

Here's how you can set it up:

```

struct ImageGridGalleryView: View {
    let inspirations = NatureInspiration.examples()
    var body: some View {
        ScrollView {
            LazyVGrid(columns: [GridItem(.adaptive(minimum: 180)),

```

```
                spacing: 0)],
            alignment: .center,
            spacing: 0, content: {
                ForEach(inspirations) {
                    ImageAspectRatio(imageName: $0.imageName,
                                     frameAspectRatio: 1.2)
                }
            })
        }
    }
}
```

Notice that I've set the spacing to zero to fill out the grid nicely. However, due to different aspect ratios of images, you might encounter some sizing issues. To address this, use `ImageAspectRatio` with aspect ratio set to 1.2, creating a uniform rectangular layout.

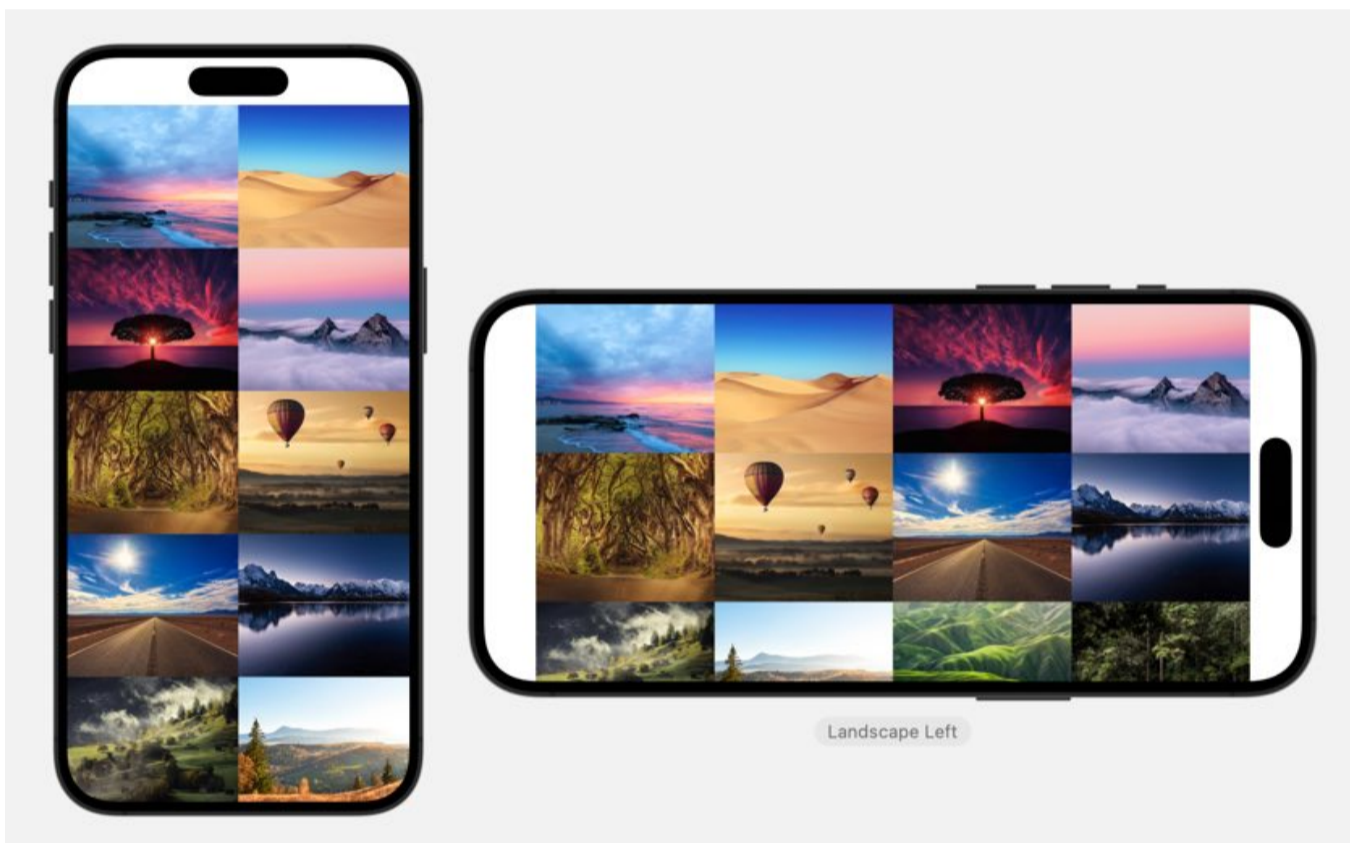


Image Gallery with AsyncImage

Now, let's explore LazyHGrid for a horizontal image gallery. We'll use the same natureInspirations data but with a twist: instead of static images, we'll load them asynchronously.

Here's a snippet to get you started:

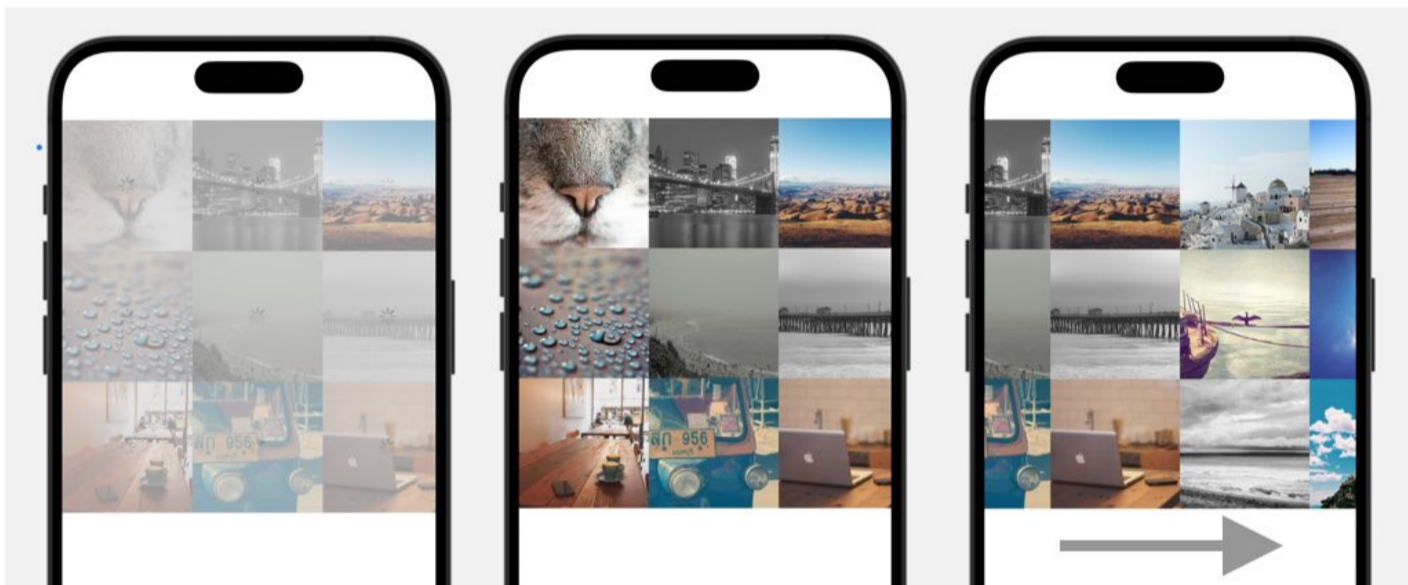
```
struct ImageHGridGalleryView: View {
    @StateObject private var photoLoader = PicsumPhotoLoader(page: 2,
                                                            photosPerPage: 40)
    let rows = Array(repeating: GridItem(.fixed(150), spacing: 0),
                    count: 3)
```

```

var body: some View {
    ScrollView(.horizontal) {
        LazyHGrid(rows: rows,
            alignment: .center,
            spacing: 0, content: {
            ForEach(photoLoader.photos) {
                PicsumPhotoImage(photo: $0,
                    aspectRatio: 1)
            }
        })
    }
}

```

In this example, I've set the rows to a fixed size of 150 and removed the spacing to create a uniform layout. I want to achieve a square image pattern:



Therefore I am setting the aspect ratio of the asynchronously loaded images to 1. To achieve this you can modify the PicsumPhotoImage component from the previous section. As an extra argument, you can use the aspectRatio parameter. If you don't pass it in the initialiser, the default inherited aspect ratio is used that is provided in the PicsumPhoto instance:

```

struct PicsumPhotoImage: View {
    let photo: PicsumPhoto
    let aspectRatio: CGFloat
    @Environment(\.displayScale) var scale

    init(photo: PicsumPhoto, aspectRatio: CGFloat? = nil) {
        self.photo = photo
        self.aspectRatio = aspectRatio ?? photo.aspectRatio
    }

    var body: some View {
        GeometryReader(content: { geometry in
            AsyncImage(url: url(in: geometry.size),
                scale: 3,
                transaction: .init(animation: .easeIn)) { phase in
                switch phase {
                case .empty:
                    ZStack {

```

```

        Color(white: 0.8)
        ProgressView()
    }
    case .success(let image):
        image
            .resizable()
            .scaledToFit()
    case .failure(let error):
        Text(error.localizedDescription)
        // use placeholder for production app
    @unknown default:
        fatalError()
    }
}
})
}
}
}

func url(in size: CGSize) -> URL? {
    let imageWidth = Int(size.width * scale)
    let imageHeight = Int(size.height * scale)
    let urlString = "https://picsum.photos/id/\(photo.id)/\(imageWidth)/\
(imageHeight)"
    return URL(string: urlString)
}
}
}

```

Tips for Using Lazy Grids

Using LazyVGrid and LazyHGrid is straightforward, but the challenge lies in choosing between fixed, flexible, or adaptive configurations. For a vertical grid, an adaptive approach is useful to fill the entire screen width and scroll vertically.

Another important aspect is resizing the images to fill the grid cells correctly. You might recall our discussions on image resizing; those concepts are now paying off as we have reusable views that maintain the aspect ratio while fitting the designated space.

Asynchronously loading images from a server to fill the grid is another advanced technique that enhances the user experience by ensuring images are displayed as soon as they're downloaded.

Lastly, you can combine LazyVGrid and LazyHGrid in various ways to create complex layouts. For instance, you can nest multiple horizontal scroll views within a vertical scroll view or mix image galleries with other UI elements to craft a rich, dynamic interface.

Remember, these grids are particularly powerful for image galleries, allowing you to create visually appealing layouts that are both functional and engaging.

8.10 INFINITIVE LOADING VIEW

When you're working with a list of images that loads asynchronously, you might have initially set up your app to load a fixed number of images, say 20 photos per page. But what happens when the user scrolls to the end of these images? Typically, you'd expect the list to keep scrolling and loading more images indefinitely. This is known as an infinite scroll view, and in this section, I'll show you how to implement this in SwiftUI.

Understanding the Infinite Scroll Concept

Firstly, you need to decide when to trigger the loading of more images. As a user, you wouldn't expect the list to abruptly stop. Instead, you'd want to continue scrolling and see a loading indicator that more content is on the way.

Setting Up the Loading Indicator

You'll want to display a loading indicator at the bottom of your list. This can be similar to the placeholder view you might have used before. For instance, you can use a progress view as a card that appears at the end of your list.

In the below image sequence, you can see the large placeholder first that should start the download of more pages. Next, you can see the loading placeholders for the individual images. Finally, all downloaded images are displayed:



To make sure it fills the entire height of the screen, you can use `containerRelativeFrame`. When this large initial loading indicator appears, that's your cue to start fetching the next set of images.


```

struct InfinitePhotosView: View {
    @StateObject private var photoLoader = InfinitePhotoLoader(page: 1,
                                                                photosPerPage: 10)

    var body: some View {
        ScrollView {
            LazyVStack(spacing: 2) {
                ForEach(photoLoader.photos) {
                    PicsumPhotoImage(photo: $0)
                }

                ZStack {
                    Color(white: 0.9)
                    ProgressView()
                        .controlSize(.extraLarge)
                }
                .containerRelativeFrame(.vertical)
                .onAppear {
                    photoLoader.loadImages()
                }
            }
        }
    }
}

```

However, you don't want to start loading everything all at once, so you'll need to modify your view model to handle this new behavior.

Creating the Infinite Photo Loader ViewModel

Create a new Swift file named `InfinitePhotoLoader.swift` and set up a new `ObservableObject` class called `InfiniteLoader`. This class will need properties to keep track of the current page, the number of photos per page, and whether it's currently loading images.

Here's a basic structure for your `InfiniteLoader` class:

```

class InfinitePhotoLoader: ObservableObject {

    @Published var photos: [PicsumPhoto] = []

    var page: Int
    let photosPerPage: Int

    @Published var isLoading = false

    init(page: Int = 1, photosPerPage: Int = 10) {
        self.page = page
        self.photosPerPage = photosPerPage
    }

    func loadImages() {

        guard !isLoading else { return }
        isLoading = true

        let urlString = "https://picsum.photos/v2/list?page=\(page)
                        &limit=\(photosPerPage)"

        page += 1
        guard let url = URL(string: urlString) else { return }
    }
}

```

```

        URLSession.shared.dataTask(with: url) { data, response, error in
            if let data,
                let photos = try? JSONDecoder().decode([PicsumPhoto].self,
                                                         from: data) {
                DispatchQueue.main.async {
                    self.photos.insert(contentsOf: photos, at: self.photos.count)
                    self.isLoading = false
                }
            }
        }.resume()
    }
}

```

Loading More Images

When you reach the end of the currently loaded images, you'll want to load more. To do this, increment the `currentPage` property each time you fetch a new set of images. Make sure to append the new images to the existing array rather than replacing them.

Implementing the Infinite Scrolling

In your `InfinitePhotosView`, use the `onAppear` modifier on the loading indicator view to trigger the `loadImages` function of your `InfiniteLoader` view model. This will ensure that more images are loaded when the user reaches the end of the list.

Here's an example of how you might implement this:

```

ScrollView {
    LazyVStack(spacing: 2) {
        ForEach(photoLoader.photos) {
            PicsumPhotoImage(photo: $0)
        }

        ZStack {
            Color(white: 0.9)
            ProgressView()
                .controlSize(.extraLarge)
        }
        .containerRelativeFrame(.vertical)
        .onAppear {
            photoLoader.loadImages()
        }
    }
}

```

By using the **onAppear** modifier on a loading indicator, you can create an infinite scrolling experience without having to manually check the scroll position. Alternatively, you can also use the newer **task** modifier that works with `async await`.

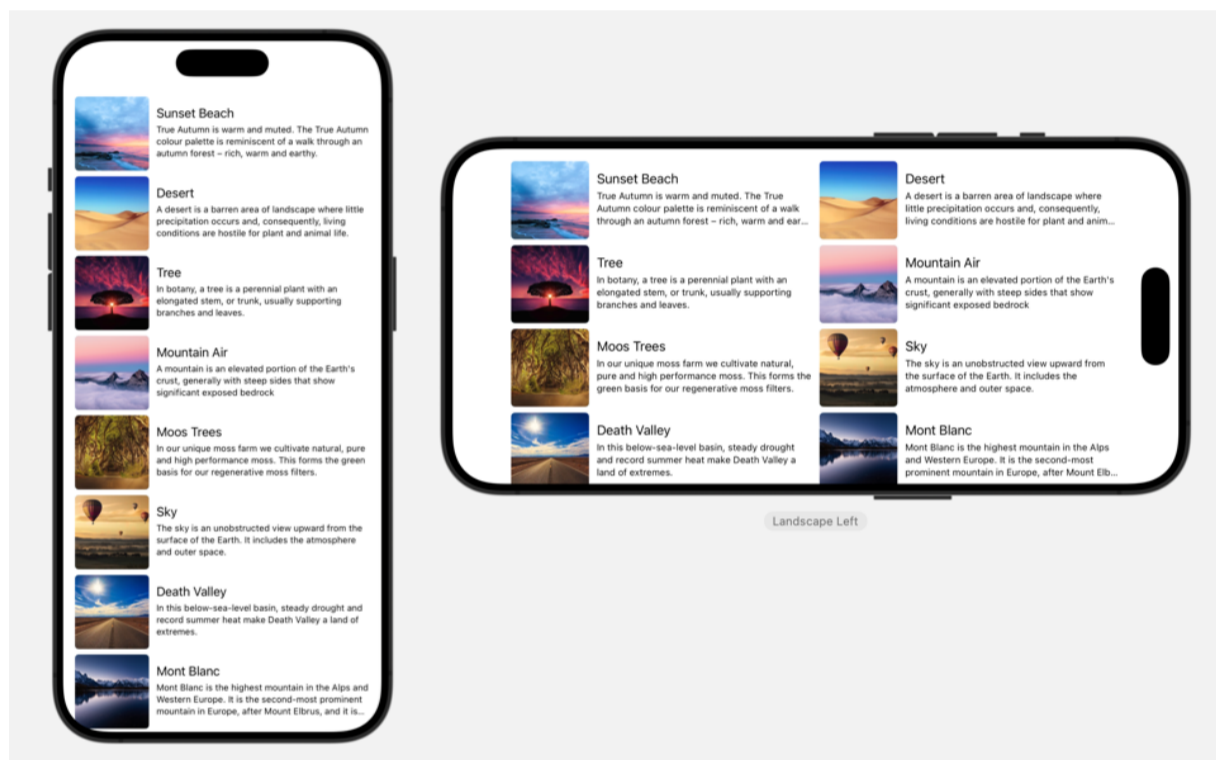
Remember to update your view model to handle the loading state and keep track of which page you're on. Although the API you're using might not provide information on the total number of pages available, the key takeaway is knowing when to fetch more data to keep the UI continuously populated with new images as the user scrolls.

CHALLENGES 🖐️

In our journey through SwiftUI, we've explored the power of LazyVGrid and LazyHGrid for creating image and emoji grid views. But did you know that these grids can also be harnessed for other adaptive layouts?

Challenge: Grid Columns for Adaptive Layout

Imagine you have a list that showcases nature-inspired images, complete with titles and descriptions. These elements are neatly arranged in an HStack, forming individual cells. When you rotate your device to landscape mode, you want the layout to adapt and display two columns instead of one. This not only utilizes the space more efficiently but also prevents the text from stretching too wide, which could compromise the aesthetics of your layout.



To incorporate this InspirationRowView into my list, I **replaced the LazyVStack with a LazyVGrid**. To ensure that only one column is displayed in portrait mode, I set the minimum column width to a high value, such as 250 points. This width is too large for any iPhone to accommodate two columns in portrait mode, so it defaults to a single column.

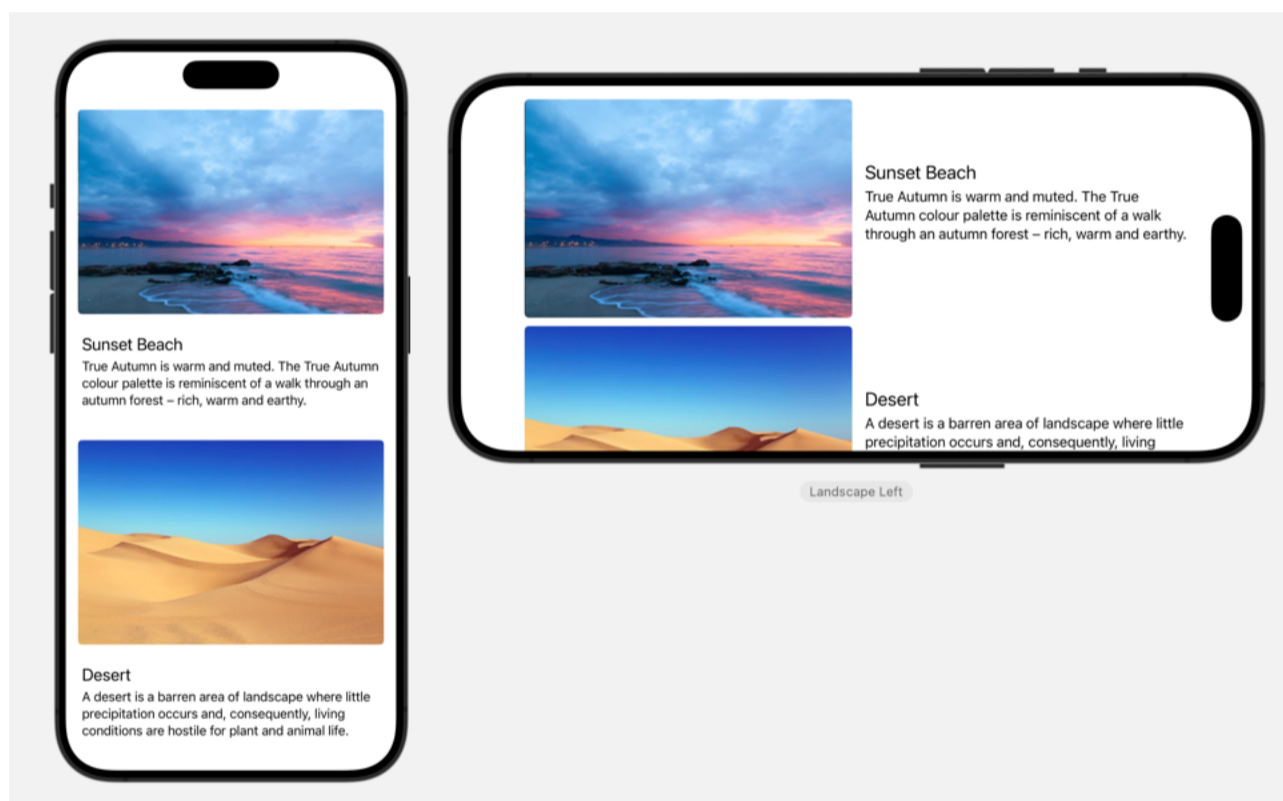
```
struct InspirationColumnExampleView: View {  
    let inspirations = NatureInspiration.examples()  
  
    var body: some View {  
        ScrollView {  
            LazyVGrid(columns: [GridItem(.adaptive(minimum: 300), spacing: 10)]) {  
                ForEach(inspirations) { inspiration in  
                    InspirationRowView(inspiration: inspiration)  
                }  
            }  
        }  
        .padding()  
    }  
}
```

However, when you switch to landscape mode, the device has enough space to fit two columns. This simple adjustment with LazyVGrid allows for a layout that gracefully adapts to different device orientations. It's also worth noting that this approach scales well on larger devices like iPads or Macs, where you might see three columns on an 11-inch iPad, for example.

Challenge: Layout Switch VStack to HStack

In this section, we're going to explore a slightly unconventional use of LazyVGrid in SwiftUI. Imagine you have a collection of items, and each item has an image, a title, and a description. You've laid them out in a ScrollView, and it looks great in portrait mode on your iPhone. But what happens when you switch to landscape mode? If you keep the same layout, it might look off. The images could be too large, or the text might stretch too far across the screen. It's not ideal.

So, what you want is a layout that adapts to the orientation of the device. In landscape mode, it would be much nicer to have the title and text next to the image, rather than below it.



This is where LazyVGrid comes into play. I started with a ScrollView, nothing new there. Then, I used a LazyVGrid. But here's the twist: I used the `.adaptive` grid item to show not just one view for each item, but two. One is the image, and the other is a VStack containing the text and title.

The LazyVGrid will now attempt to place these two elements—the image and the VStack text—separately. Both elements are set to be adaptive, which means that on an iPhone in portrait mode, it can only fit one column at a time because of the minimum width of 250 points.

```
struct LargeInspirationColumnExampleView: View {  
  
    let inspirations = NatureInspiration.examples()  
    let columns = [GridItem(.adaptive(minimum: 350),  
                            spacing: 10)]  
  
    var body: some View {
```

```

    ScrollView {
        LazyVGrid(columns: columns,
            alignment: .leading,
            spacing: 10) {
            ForEach(inspirations) { inspiration in
                Image(inspiration.imageName)
                    .resizable()
                    .scaledToFit()
                    .clipShape(RoundedRectangle(cornerRadius: 5))

                VStack(alignment: .leading, spacing: 5) {
                    Text(inspiration.name)
                        .font(.title2)
                    Text(inspiration.description)
                }
                    .padding(.bottom)
                    .padding(.vertical)
                    .padding(.leading, 5)
            }
        }
    }
}

```

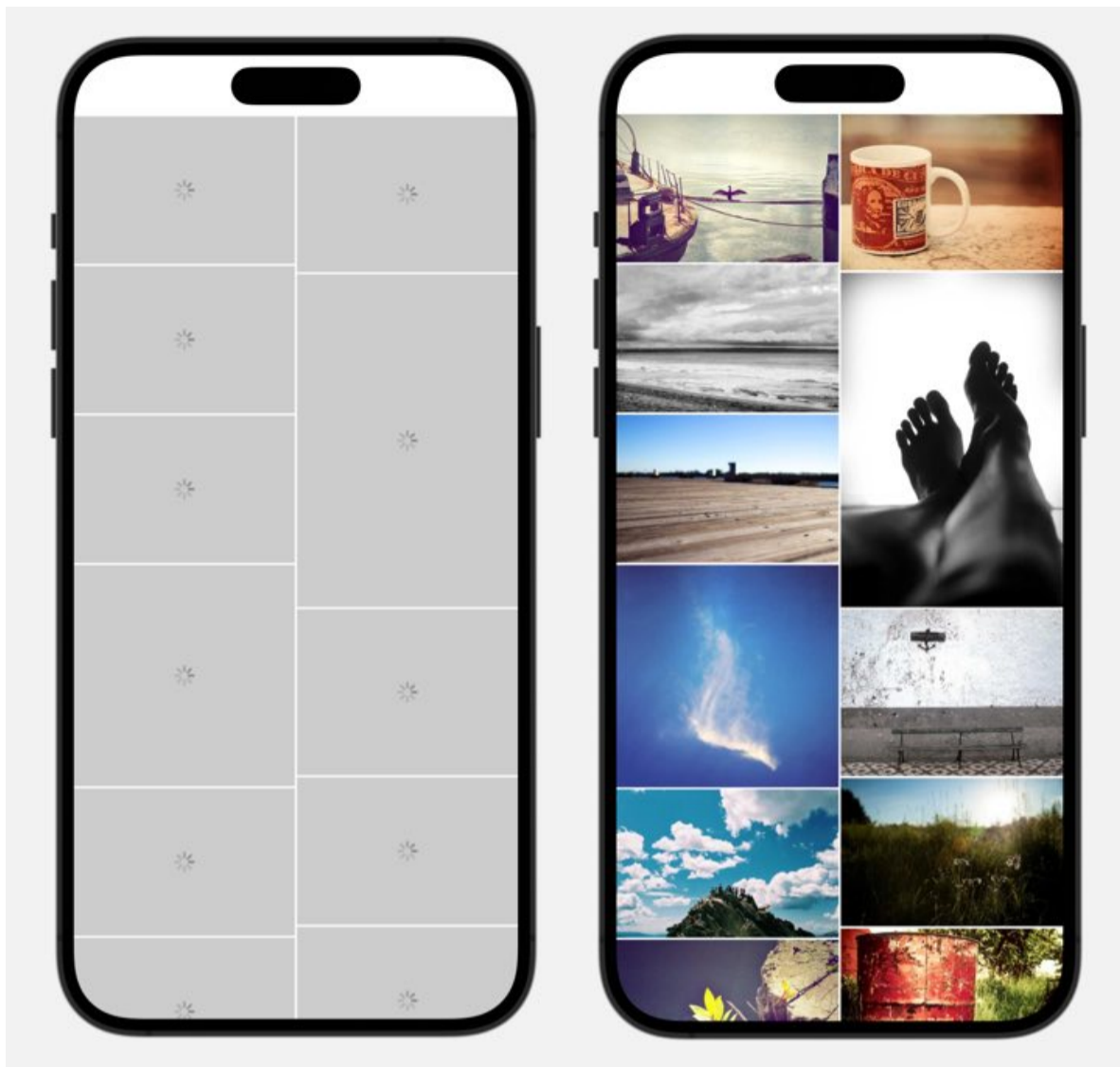
However, when you rotate the device to landscape mode, it will show 2 columns. It doesn't really care what you put inside; it just takes the first view that it has inside for its first element in the first column, then uses the second element for the second column, and so on. It simply distributes these elements across the grid.

With this simple trick of using the LazyVGrid, you get a very nice, adaptive layout. The elements move around smoothly with the animations, creating a dynamic and responsive design that looks good in both portrait and landscape orientations.

Remember, the key here is to think about how your layout should adapt to different screen sizes and orientations. By using LazyVGrid with adaptive columns, you can create a flexible layout that adjusts to the available space, providing a great user experience no matter how the device is held.

Challenge: Waterfall Image Gallery

In our journey through SwiftUI layouts, we've explored a variety of ways to present images. Now, I want to introduce you to another layout style that you can create based on what I've already shown you. This is a waterfall, Pinterest-style layout. It's a lazily loaded layout with two columns, where each column maintains the same aspect ratio—a hallmark of the waterfall layout. This layout uses AsyncImage for efficient, lazy loading.



To make the layout lazy, I've used an HStack containing two lazy VStacks. While the HStack itself isn't lazy and creates both columns immediately, the images within the lazy VStacks are loaded as needed.

```

struct LazyWaterfallImageGridExampleView: View {

    @StateObject var firstPhotoLoader = PicsumPhotoLoader(page: 6)
    @StateObject var secondPhotoLoader = PicsumPhotoLoader(page: 4)

    let spacing: CGFloat = 2

    var body: some View {
        ScrollView {
            HStack(alignment: .top, spacing: spacing) {
                LazyVStack(spacing: spacing) {
                    ForEach(firstPhotoLoader.photos) { photo in
                        PicsumPhotoImage(photo: photo)
                    }
                }

                LazyVStack(spacing: spacing) {
                    ForEach(secondPhotoLoader.photos) { photo in
                        PicsumPhotoImage(photo: photo)
                    }
                }
            }
        }
    }
}

```

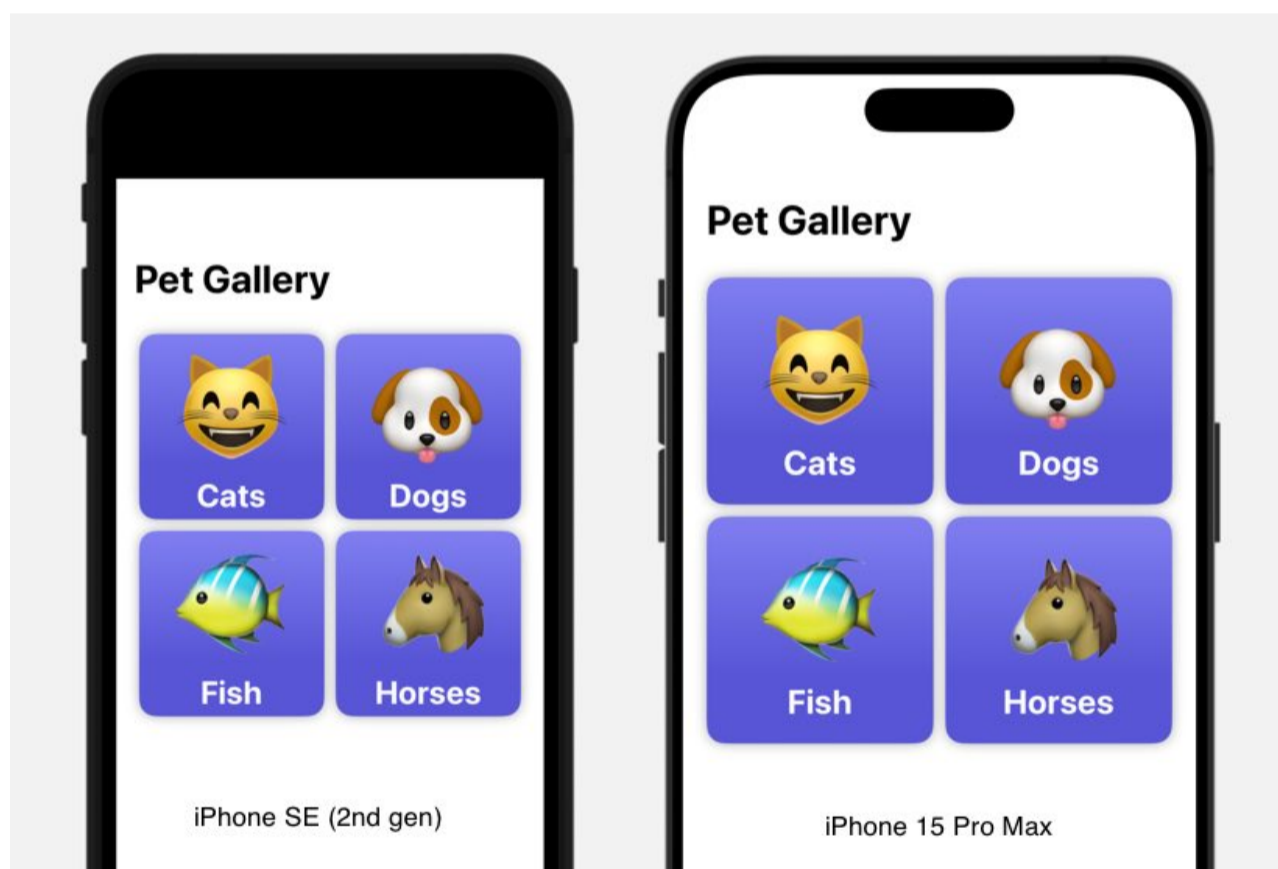
This approach is similar to what we've done before, using `ForEach` to display images with their natural aspect ratios. To handle data for two columns, I've simply created two different arrays of example data.

Here I use `PicsumPhotoImage`, which loads images asynchronously, handling caching and other complexities. I've set the spacing to 2 to create a neat grid layout. To populate the columns with different images, I've assigned different pages of data to each, which might seem a bit hacky, but it's effective and straightforward.

As we reach the bottom of the scroll view, you could trigger a reload of additional pages to keep the content fresh. Ideally, you'd fetch images with varying aspect ratios from the server to add diversity to the layout. However, as you can see, this setup is quite simple—much easier than some of the more complex layouts we've discussed.

Challenge: Grid Cards

In this layout challenge, I will change the current layout into a grid style, using the icons. I want to ensure that the indigo background has an aspect ratio of 1, so it scales nicely across different screen sizes. Remember, no fixed frames are allowed—your layout must be adaptable.



For my solution, I chose `LazyVGrid` because I may add more pet categories in the future. A vertical layout is more space-efficient when there's nothing else in the view. The key here is to define the columns. I've decided on two columns that adjust to screen sizes, so I use an array of `GridItems` with the flexible modifier to allow them to stretch.

```
struct EmojiPetPalGalleryView: View {
    let pets: [Pet]
    private let padding: CGFloat = 10
```

```

private var columns: [GridItem] {
    Array(repeating: GridItem(.flexible(minimum: 100),
        spacing: padding),
        count: 2)
}

var body: some View {
    NavigationStack {
        ScrollView {
            LazyVGrid(columns: columns, spacing: padding) {
                ForEach(pets, id: \.type) { pet in
                    PetCardView(pet: pet)
                }
            }
            .padding(padding * 2)
        }
        .navigationBarTitle("Pet Gallery")
    }
}
}

```

I encapsulate the emoji and text within a `PetCardView`. To achieve the desired aspect ratio for each card, I use a `ZStack` with a `RoundedRectangle` and a `VStack` with the emoji and text on top. Outside the `ZStack`, I apply the `aspectRatio` modifier.

```

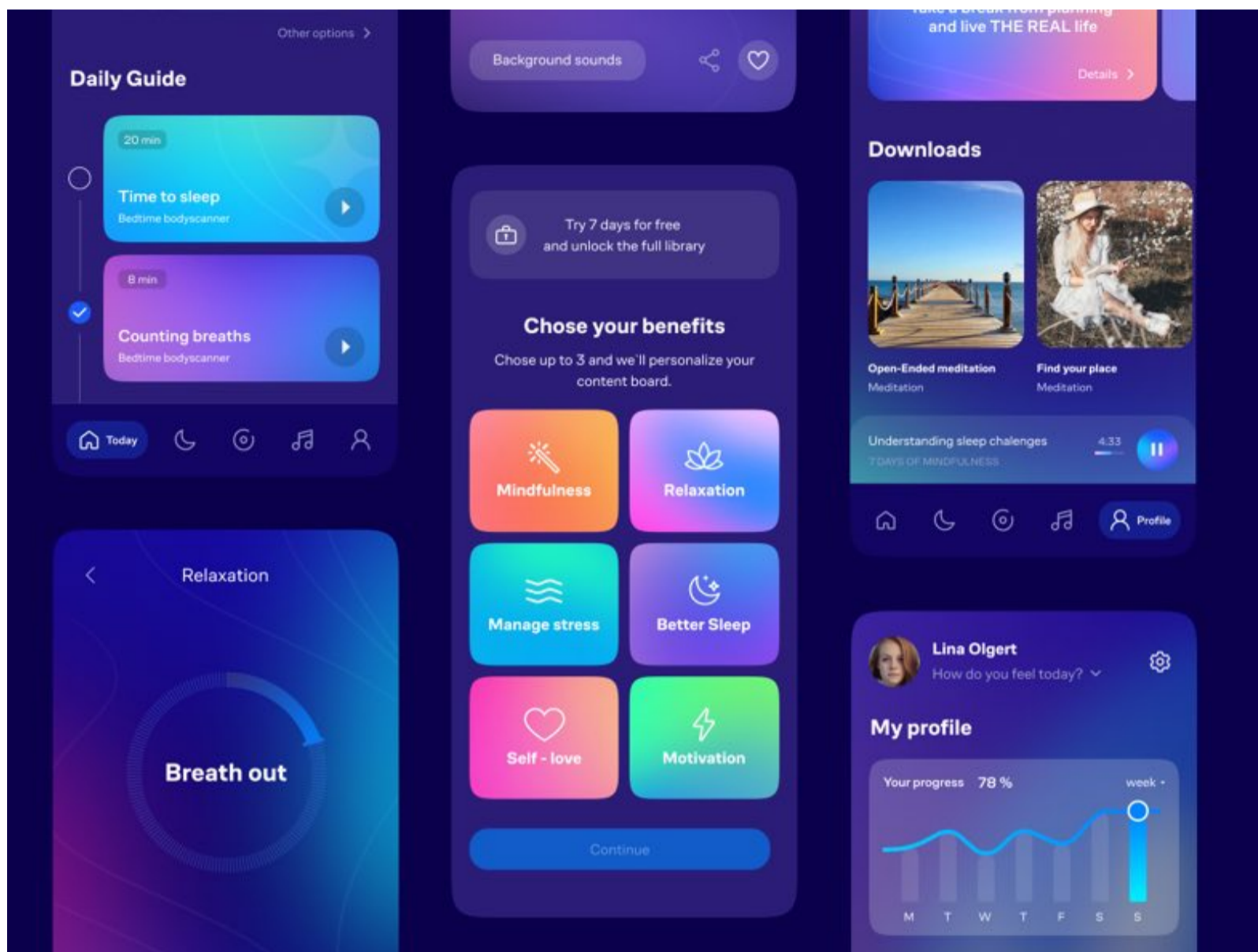
struct PetCardView: View {
    let pet: Pet
    var body: some View {
        ZStack {
            RoundedRectangle(cornerRadius: 15)
                .fill(Color.indigo.gradient)
                .shadow(radius: 5)
            VStack {
                Text(emoji(for: pet))
                    .font(.system(size: 100))

                Text(pet.type.displayName)
                    .foregroundColor(Color.white)
                    .font(.title)
                    .bold()
            }
        }
        .aspectRatio(1, contentMode: .fill)
    }
}

func emoji(for pet: Pet) -> String {
    switch pet.type {
        case .cat: "🐱"
        case .dog: "🐶"
        case .fish: "🐟"
        case .horse: "🐎"
    }
}
}

```


Grid layouts are common in app design. For example, a meditation app might use a grid to display different practice categories, each with an icon and text. Incorporating such card-style grids in your apps can help you make the most of the available space and enhance the user experience.



9. SCROLLVIEW

Throughout this book, you've encountered ScrollView multiple times. However, it's time to delve deeper into this versatile component. ScrollView has a plethora of functionalities, and with the introduction of new APIs in iOS 17, it has become even more powerful. New features are paging, programmatic scrolling, and scroll transition effects.

9.1 WHY USE SCROLLVIEW?

ScrollView is your go-to when you have more content than what can be displayed on a single screen. It's incredibly flexible, allowing for both horizontal and vertical scrolling.

Imagine you're dealing with **dynamic data**, like a list that could potentially have hundreds of entries. In its simplest form, ScrollView tries to accommodate its size to the content within. For large datasets, you'd typically use a LazyVStack, as I've discussed in previous sections.

Another scenario where ScrollView shines is when you want to ensure your content is accessible and fits on all screen sizes, including those with **accessibility settings** enabled. Here's a simple example:

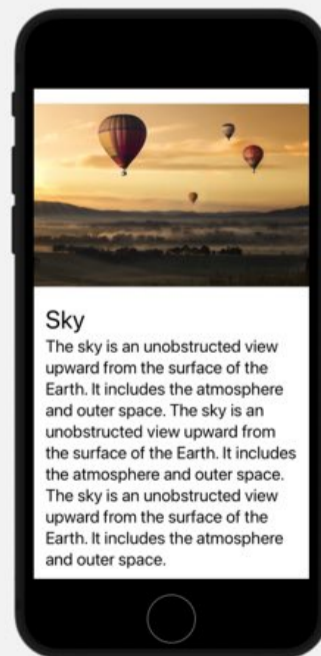
```
struct DetailView: View {
    let inspiration = NatureInspiration.example1()
    var body: some View {
        ScrollView {
            VStack {
                ResizableImageView(imageName: inspiration.imageName)
                Text(inspiration.name)
                    .font(.title)
                Text(inspiration.description)
            }
        }
        .padding()
    }
}
```



iPhone 15 Pro



AX 2



XXX Large

iPhone SE (2nd gen)

In this `DetailView`, even if the content is minimal, you might have instances where the text is lengthy. Wrapping your content in a `ScrollView` ensures it remains accessible and fits on the screen, regardless of the amount of text or the device's screen size.

The primary use case for `ScrollView` is to **create adaptable views** that cater to various screen sizes and accommodate different accessibility font sizes.

9.2 CUSTOMIZING THE APPEARANCE OF SCROLLVIEW

When working with `ScrollView` in SwiftUI, you have a variety of options to customize its appearance to fit your design needs. In this section, I'll guide you through the process of tweaking `ScrollView` properties such as disabling scrolling, showing or hiding scroll indicators, adding padding, and preventing content clipping.

As an example, I will use the following horizontal scroll view with image cards:

```
struct ScrollCustomExampleView: View {
    let inspirations = NatureInspiration.examples()
    let spacing = 10
    var body: some View {
        ScrollView(.horizontal) {
            HStack(spacing: spacing) {
                ForEach(inspirations) { inspiration in
                    InspirationCard(inspiration: inspiration)
                        .containerRelativeFrame(.horizontal, count: 4, span: 3,
                                                spacing: spacing)
                }
            }
        }
        .padding()
    }
}

struct InspirationCard: View {
    let inspiration: NatureInspiration
    let frameAspectRatio: CGFloat

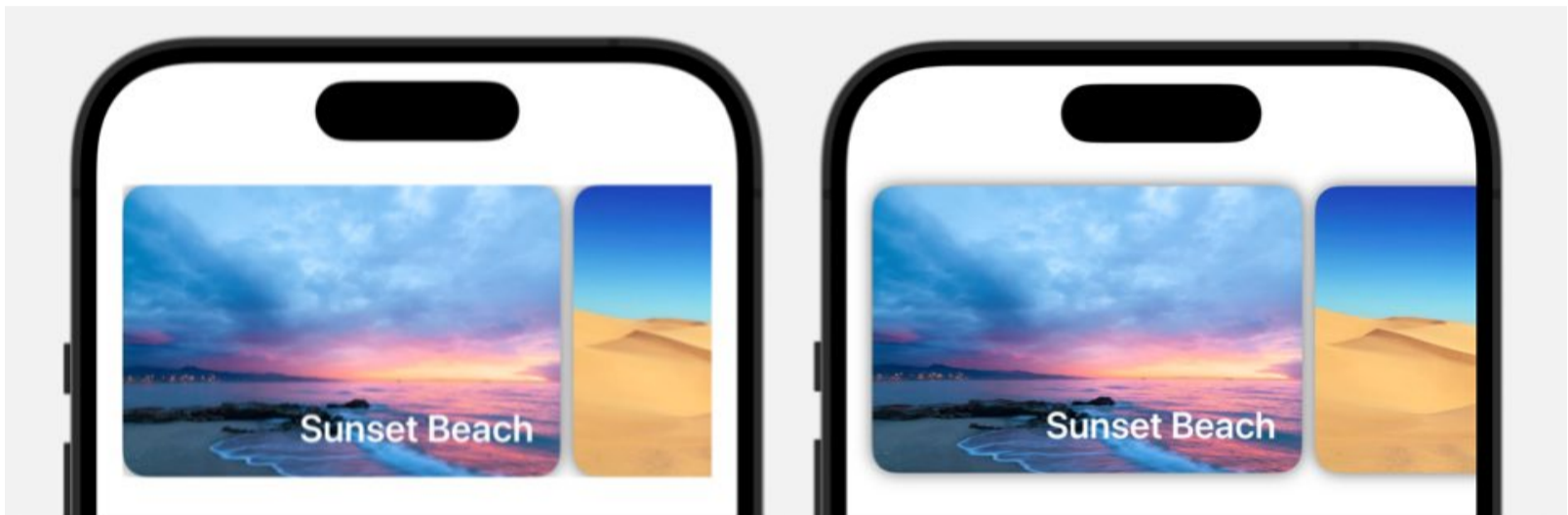
    init(inspiration: NatureInspiration, frameAspectRatio: CGFloat = 1.5) {
        self.inspiration = inspiration
        self.frameAspectRatio = frameAspectRatio
    }

    var body: some View {
        ImageAspectView(imageName: inspiration.imageName,
                        frameAspectRatio: frameAspectRatio,
                        cornerRadius: 15)
            .shadow(radius: 5)
            .overlay(alignment: .bottomTrailing) {
                Text(inspiration.name)
                    .bold()
                    .foregroundColor(.white)
                    .padding()
            }
    }
}
```

Preventing Content Clipping

In previous versions of iOS, shadows or other effects might get clipped at the edges of a ScrollView. Starting with iOS 17, you can prevent this clipping using the `.scrollClippingDisabled(_:)` modifier:

```
ScrollView(.horizontal) {  
    ...  
}  
.padding()  
.scrollClippingDisabled(true)
```



This ensures that visual effects like shadows are fully visible, even at the edges of the ScrollView.

Adding Padding and Margins

To add space around your content within the ScrollView, you can use padding or the newer `contentMargins` modifier. You can use padding to the views inside ScrollView:

```
ScrollView(.horizontal) {  
    HStack {  
        ...  
    }  
    .padding(20)  
}
```

Or, using `contentMargins`, which adds space around the content inside the ScrollView:

```
ScrollView(.horizontal) {  
    HStack {  
        ...  
    }  
}  
.contentMargins(20)
```

Both of the above modifications will result in the same layout.



However, `contentMargins` also allows you to set the scroll indicator margins separately:

```
.contentMargins(20)  
.contentMargins(5, for: .scrollIndicators)
```



In the above example, the scroll indicator initially overlaps with the image cards. I can change this with the help of the content margins and the scroll indicator is placed below the images.

Scroll Indicators

By default, `ScrollView` shows scroll indicators, but you can easily hide them if you prefer a cleaner look. Use the `showsIndicators` parameter in the `ScrollView` initializer or the `.scrollIndicatorVisibility(_:)` modifier for more control:

```
ScrollView(.horizontal, showsIndicators: false) {  
    ""  
}
```

Or, for iOS 16 and later:

```
ScrollView(.horizontal) {  
    ...  
}  
.scrollIndicators(.hidden)
```

If you want to make the scroll indicators more prominent and flash them to draw attention to the scrollable area, you can use `scrollIndicatorsFlash` which is available for iOS 17+. You can use this modifier to control whether the scroll indicators of a scroll view briefly flash when the view first appears:

```
ScrollView(.horizontal) {  
    ...  
}  
.scrollIndicatorsFlash(onAppear: true)
```

or flash scroll indicators when a value changes, use `scrollIndicatorsFlash(trigger:)` instead. Here is an example that flashes the indicator every time the number value changes, which happens when you press the button:

```
struct ScrollCustomExampleView: View {  
    @State private var number: Int = 1  
  
    var body: some View {  
        ScrollView(.horizontal) {  
            ...  
        }  
        .scrollIndicatorsFlash(trigger: number)  
  
        Button {  
            number += 1  
        } label: {  
            Text("increase number \ \(number)")  
        }  
    }  
}
```

Disabling Scrolling

Sometimes, you might want to present your content in a `ScrollView` but disable the scrolling temporarily. This can be done by setting the `isScrollEnabled` modifier to `false`. Here's how you do it:

```
ScrollView(.horizontal) {  
    ...  
}  
.scrollDisabled(true)
```

With this modifier, your `ScrollView` will no longer respond to scroll gestures.

9.3 SCROLL DIRECTION

In SwiftUI, customizing the scroll direction of a `ScrollView` is straightforward. By default, a `ScrollView` is vertical, but you can easily change it to horizontal or even allow for both directions. Let's dive into how you can set this up.



Vertical Scroll Example View

The default scroll direction is vertical. You don't need to specify anything. It applies a default alignment and spacing which you can change by embedding the content of the `ScrollView` in a `VStack` or `LazyVStack`:

```
ScrollView {
    LazyVStack(alignment: .leading, spacing: 10) {
        ForEach(inspirations) { inspiration in
            InspirationCard(inspiration: inspiration)
        }
    }
    .padding(20)
}
```

Horizontal Scroll Example View

To create a horizontal `ScrollView`, you'll need to specify the axis. This is done using the `axes` argument within the `ScrollView` initializer. Here's an example:

```
ScrollView(.horizontal) {
    LazyHStack {
        ForEach(inspirations) { inspiration in
            InspirationCard(inspiration: inspiration)
        }
    }
}
```

```

        }
    }
    .padding()
}
.frame(height: 200)

```

Remember, when you're using a horizontal `ScrollView`, you'll typically want to use an `HStack` to lay out your content horizontally.

Bidirectional Scroll Example View

Sometimes, you might want to allow scrolling in both directions. This can be particularly useful for large tables or when displaying a lot of data. To enable bidirectional scrolling, you'll use a set of axes:

```

struct BidirectionalScrollExampleView: View {
    let emojis = Emoji.examples()
    let columns = Array(repeating: GridItem(.fixed(150)), count: 5)
    var body: some View {
        ScrollView([.horizontal, .vertical]) {
            LazyVGrid(columns: columns,
                    alignment: .center,
                    spacing: 10, content: {
                ForEach(emojis) {
                    EmojiView(emoji: $0)
                }
            })
        }
        .padding()
    }
}

```

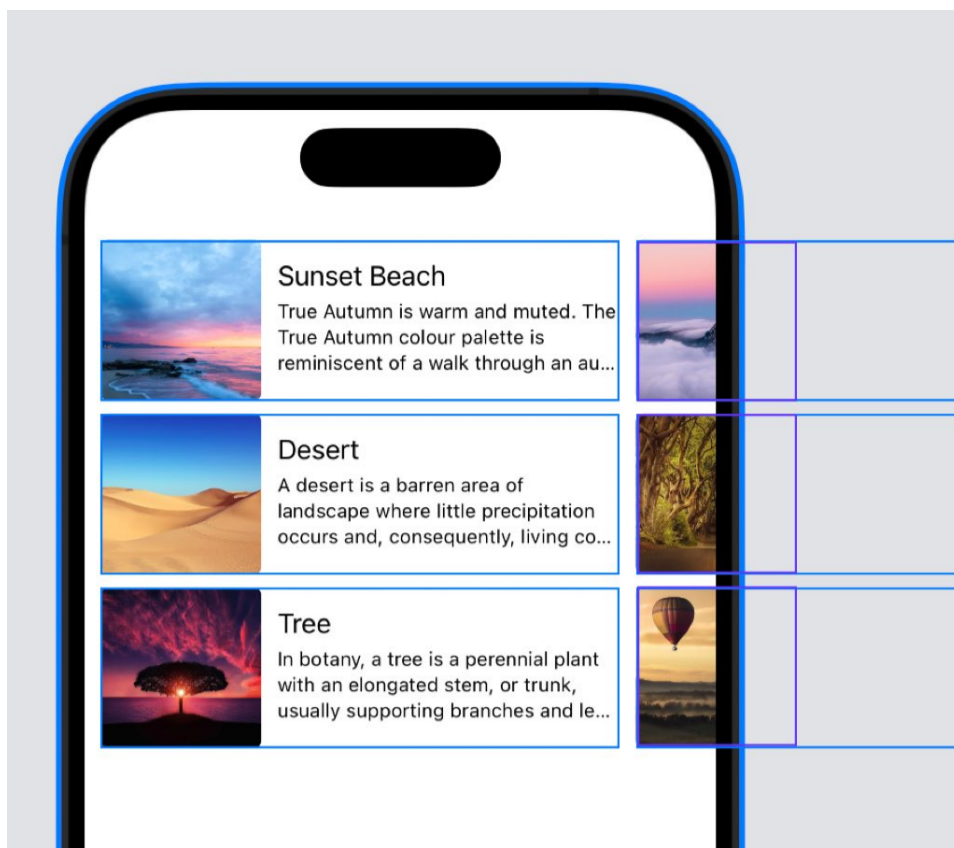
9.4 SCROLL CONTENT SIZE

When working with `ScrollView` in SwiftUI, sizing the content correctly can be a bit of a challenge. The `ScrollView` has its own sizing behavior, which can complicate things, especially when dealing with images that you want to scale to fit the available space. In this section, I'm going to walk you through two strategies to handle this: using [GeometryReader](#) and the [containerRelativeFrame](#) modifier, which is new in iOS 17.

Practical Example

Let's say you have a horizontal `ScrollView` with 3 rows of images, and you want to show just a bit of the next column to indicate that the view is scrollable. You might be tempted to set a fixed width, but this won't adapt well to different screen sizes.

For example I want to show half of the images that are shown in the second column:



Using GeometryReader

First up, let's talk about GeometryReader. This is a view that provides you with the size and position of its parent view. It's particularly useful when you need to size content within a ScrollView because it lets you make decisions based on the available space. Here's how you can use GeometryReader:

```
struct ScrollContentGeometryReaderExampleView: View {
    let inspirations = NatureInspiration.examples()
    let padding: CGFloat = 10
    let imageSize: CGFloat = 100

    var rows: [GridItem] {
        Array(repeating: GridItem(.fixed(imageSize)), count: 3)
    }

    var body: some View {
        GeometryReader { proxy in
            ScrollView(.horizontal) {
                LazyHGrid(rows: rows, spacing: padding) {
                    ForEach(inspirations) { inspiration in
                        InspirationRowView(
                            inspiration: inspiration,
                            imageSize: imageSize
                        )
                        .frame(
                            width: proxy.size.width - padding * 2
                                - imageSize * 0.5
                        )
                    }
                }
            }
            .padding(padding)
        }
    }
}
```

Here, I'm subtracting the padding from the ScrollView's width and half of the next image size.

Remember to account for any padding or spacing you've added when calculating sizes. For example, if you want to show a grid of images and you've added padding around them, you'll need to subtract this padding from the width provided by the `GeometryReader` to get the correct size for your images.

Using `containerRelativeFrame`

The second strategy involves the `containerRelativeFrame` modifier. This is a powerful tool that allows you to size your content relative to its container. It's particularly handy when you're dealing with a `ScrollView` because it lets you reference the size of the `ScrollView` directly.

Here's the above example implemented with `containerRelativeFrame`:

```
struct ScrollContainerRelativeFrameExampleView: View {
    let inspirations = NatureInspiration.examples()
    let padding: CGFloat = 10
    let imageSize: CGFloat = 100

    var rows: [GridItem] {
        Array(repeating: GridItem(.fixed(imageSize)), count: 3)
    }

    var body: some View {
        ScrollView(.horizontal) {
            LazyHGrid(rows: rows, spacing: padding) {
                ForEach(inspirations) { inspiration in
                    InspirationRowView(
                        inspiration: inspiration,
                        imageSize: imageSize)
                    .containerRelativeFrame(.horizontal) { length, axis in
                        length - padding * 2 - imageSize * 0.5
                    }
                }
            }
        }
        .padding(padding)
    }
}
```

You can adjust the width and height parameters to control how much space the content should occupy relative to its container.

`ContainerRelativeFrame` offers a more straightforward way to size content relative to its container. Remember, `containerRelativeFrame` is only available in iOS 17 and later, so if you're supporting earlier versions, you'll need to stick with `GeometryReader`.

9.5 SCROLL BEHAVIOUR

When working with ScrollViews in SwiftUI, a common feature you might want to implement is a snapping or paging behavior. This is where the content automatically aligns itself to a certain point on the screen when the user stops scrolling. Without this, users have to manually align the content, which can be a cumbersome experience.

Implementing Paging with TabView

Before diving into the latest iOS features, let's explore an alternative for earlier iOS versions using a TabView with page tab view style. This approach is particularly useful for creating onboarding screens or any scenario where you want to present full-screen content that the user can page through.



Here's how you can set it up:

```
struct TabPageExampleView: View {
    let inspirations = NatureInspiration.examples()

    var body: some View {
        TabView {
            ForEach(inspirations) { inspiration in
                InspirationCard(inspiration: inspiration)
                    .padding()
            }
        }
        #if os(iOS)
        .tabViewStyle(.page(indexDisplayMode: .always))
        .indexViewStyle(.page(backgroundDisplayMode: .always))
        #endif
    }
}
```

By setting the `tabViewStyle` to `PageTabViewStyle` and the `indexDisplayMode` to `.always`, you enable the paging behavior and display the page indicators. This makes it clear to the user which page they're on and allows them to tap the indicators to quickly navigate between pages.

Paging with ScrollView in iOS 17

Now, let's talk about the `scrollTargetBehavior`, a new feature introduced in iOS 17 that allows for similar paging behavior within a `ScrollView`:

```
ScrollView(.horizontal) {
    LazyHStack(spacing: 0) {
        ForEach(inspirations) { inspiration in
            InspirationCard(inspiration: inspiration)
                .padding()
                .containerRelativeFrame(.horizontal)
        }
    }
}
.scrollTargetBehavior(.paging)
```

With `.paging`, the `ScrollView` will snap to each page, aligning it to the visible area. If you want to align based on individual views rather than full pages, you can use `.viewBased` instead.

View-Based Paging

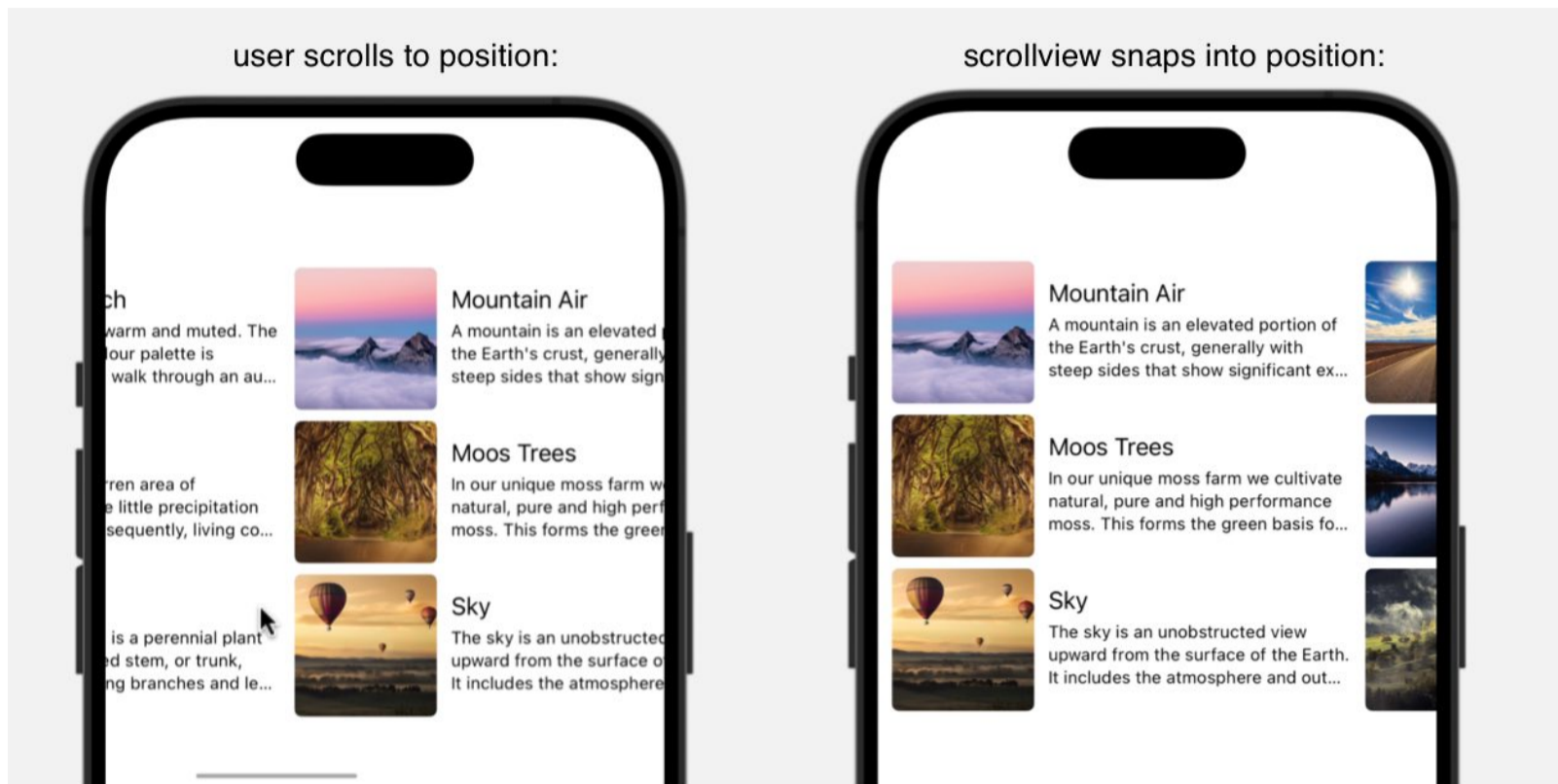
For a more granular control, where you want to snap to specific views within your `ScrollView`, use the `.viewBased` option. This is particularly useful when you have a grid or a collection of items that you want to align individually. Here's how you can implement view-based paging with the grid example from the earlier section:

```
ScrollView(.horizontal) {
    LazyHGrid(rows: rows, spacing: padding) {
        ForEach(inspirations) { inspiration in
            InspirationRowView(inspiration: inspiration,
                               imageSize: imageSize)
                .containerRelativeFrame(.horizontal) { length, axis in
                    length - padding * 2 - imageSize * 0.5
                }
        }
    }
    .padding(padding)
    .scrollTargetLayout()
}
.scrollTargetBehavior(.viewAligned)
```

To enable the view-based behavior, you'll need to do two things:

- Add a `scrollTargetLayout` modifier to the views you want to align to.
- Wrap your `ScrollView` with a `scrollTargetBehavior` modifier.

Now, when the user scrolls and releases, the ScrollView will snap to the start of each individual view/column, providing a more precise and controlled scrolling experience.



9.6 PROGRAMMATIC SCROLLING WITH SCROLLVIEWREADER

When dealing with long lists in a ScrollView, a common requirement is to programmatically scroll to a specific position. In iOS 17, SwiftUI introduced a new scroll position modifier that allows both programmatic scrolling and reading the scroll position. This feature is incredibly handy, and I'll be showing you plenty of examples. However, for versions below iOS 17, we have to rely on the ScrollViewReader, which only supports programmatic scrolling without the ability to read the current scroll offset.

Let's dive into the ScrollViewReader. You'll need to wrap your ScrollView with a ScrollViewReader to access its functionality. The ScrollViewReader provides you with a proxy that has methods for programmatic scrolling. Here's how you can use it:

```
struct ScrollViewReaderExampleView: View {
    let emojis = Emoji.examples()

    var body: some View {
        ScrollViewReader(content: { proxy in
            ScrollView {
                LazyVStack {
                    ForEach(emojis) { emoji in
                        GroupBox {
                            Text(emoji.emojiSting)
                                .font(.title)
                            Text("\(emoji.value)")
                                .frame(maxWidth: .infinity)
                        }
                    }
                }
            }
        })
    }
}
```

```

    }
    Button("scroll to 🤬", action: {
        proxy.scrollTo(128545)
    })
}
}
}

```

In this example, I'm using emojis as the data to scroll through. I've made the emojis identifiable by their value.

```

struct Emoji: Identifiable {
    let value: Int

    var emojiString: String {
        guard let scalar = UnicodeScalar(value) else { return "?" }
        return String(Character(scalar))
    }

    var id: Int {
        return value
    }
}

```

You use the same id when you call the ScrollViewReader's proxy to scrollTo position. When you use the scrollTo method, the ScrollView jumps to the specified ID. If you want a smooth, animated scroll, you can wrap the scrollTo call in an withAnimation block:

```

Button("Scroll to 🤬") {
    withAnimation(.easeInOut(duration: 2)) {
        proxy.scrollTo(128545)
    }
}

```

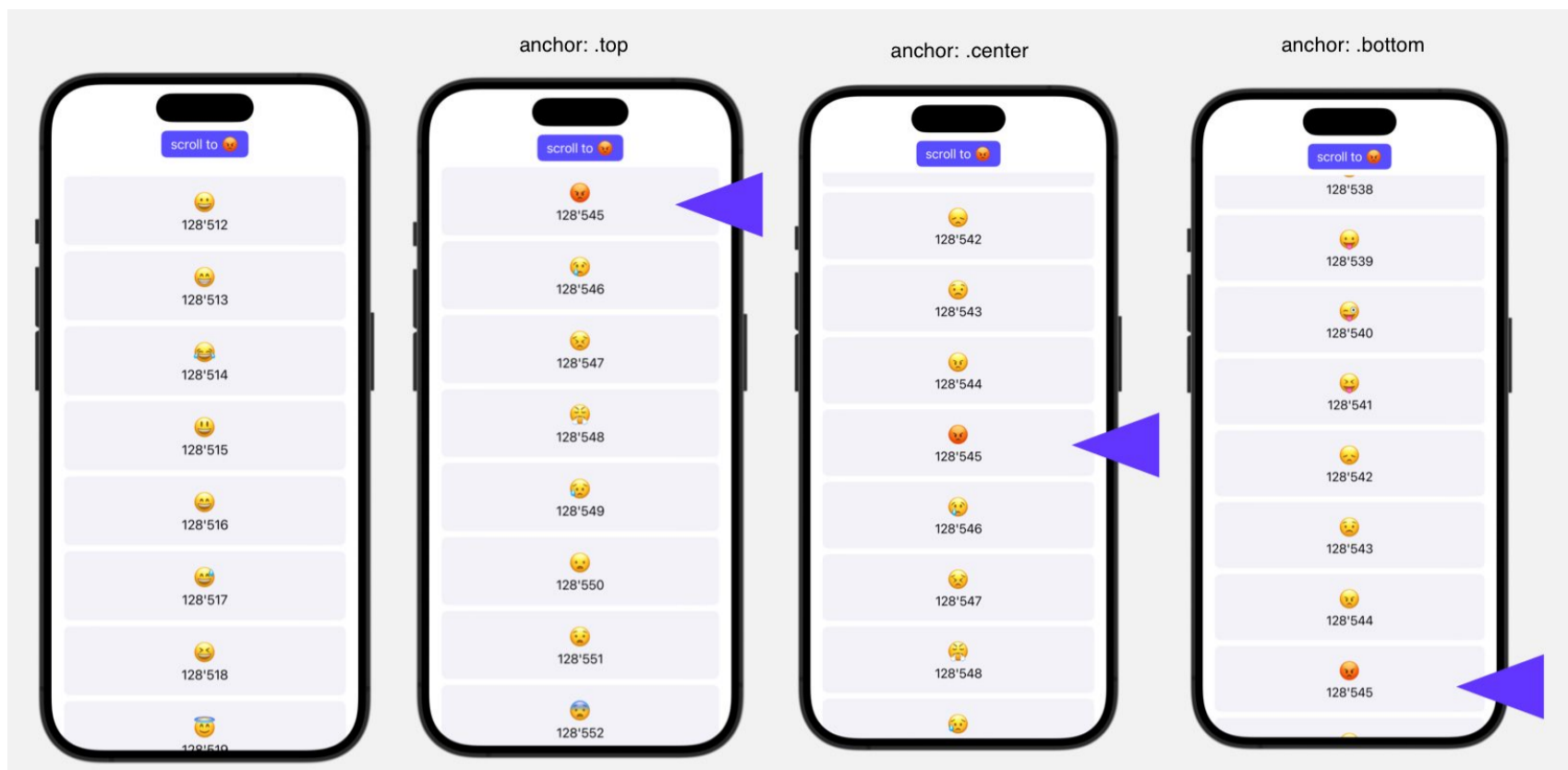
By default, scrollTo will scroll just enough to make the view visible. If you're below the target view, it will appear at the top of the ScrollView. If you're above it, it will stop as soon as the view comes into view. You can control this behavior using the anchor parameter to specify where you want the view to align within the ScrollView.

For example, if you want the view to always appear at the top, you can set the anchor to .top. If you prefer it to be centered, use .center.

```

Button("Scroll to 🤬") {
    withAnimation(.easeInOut(duration: 2)) {
        proxy.scrollTo(128545, anchor: .center)
    }
}

```



How does ScrollView know which view to scroll to?

When you're working with a ScrollView and you want to programmatically scroll to a specific view, you might wonder how the ScrollView knows which view to target. The answer lies in the use of identifiers, or IDs, which are unique values assigned to the views within the ScrollView.

For **dynamic content**, such as a list generated from an array, each element in the array should conform to the **Identifiable** protocol. This means each element has a unique id property. SwiftUI uses this id to distinguish between views:

```
ForEach(emojis) { emoji in
    Text(emoji)
        .id(emoji.id)
    // per default this is added with identifiable data in ForEach
}
```

For **static content** or when you're not using identifiable data, you can manually assign IDs to views using the `.id(_:)` modifier. This modifier takes a hashable value that you designate as the identifier for that view. In the following, I added a text view with an id of "11":

```
ScrollView {
    LazyVStack {
        ForEach(emojis) { emoji in
            ...
        }
        Text("Finish")
            .id(11)
    }
}
```

You can then scroll to this view with:

```
Button("Scroll to end") {
    proxy.scrollTo(11, anchor: .center)
}
```

SwiftUI automatically assigns an id to each view. You can use the internal ids by using the namespace:

```
@Namespace var bottomID

Text("Finish")
    .id(bottomID)
```

Remember, the `ScrollViewReader` doesn't provide information about the current scroll position. For that, we'll explore the new scroll position modifier in the next lesson, which is available in iOS 17 and later.

9.7 SCROLLVIEW POSITION

In this section, we're going to explore the `ScrollView` position, a feature introduced in iOS 17 that allows us to control and observe the scroll position within a `ScrollView`. I'll be reusing the same view from the previous example so that you can compare the old and new methods side by side.

```
struct ScrollPositionExampleView: View {
    let emojis = Emoji.examples()
    @State private var scrollID: Int? = nil

    var body: some View {
        ScrollView {
            LazyVStack {
                ForEach(emojis) { emoji in
                    GroupBox {
                        Text(emoji.emojiSting)
                            .font(.title)
                        Text("\ (emoji.value)")
                            .frame(maxWidth: .infinity)
                    }
                }
            }
            .padding()
            .scrollTargetLayout()
        }
        .scrollPosition(id: $scrollID, anchor: .center)
    }
}
```

Now, let's get to the main point: scrolling. This is done using the `scrollPosition` modifier, which requires a binding. A binding is two-way; you can set it and read from it. This is why it works so well.

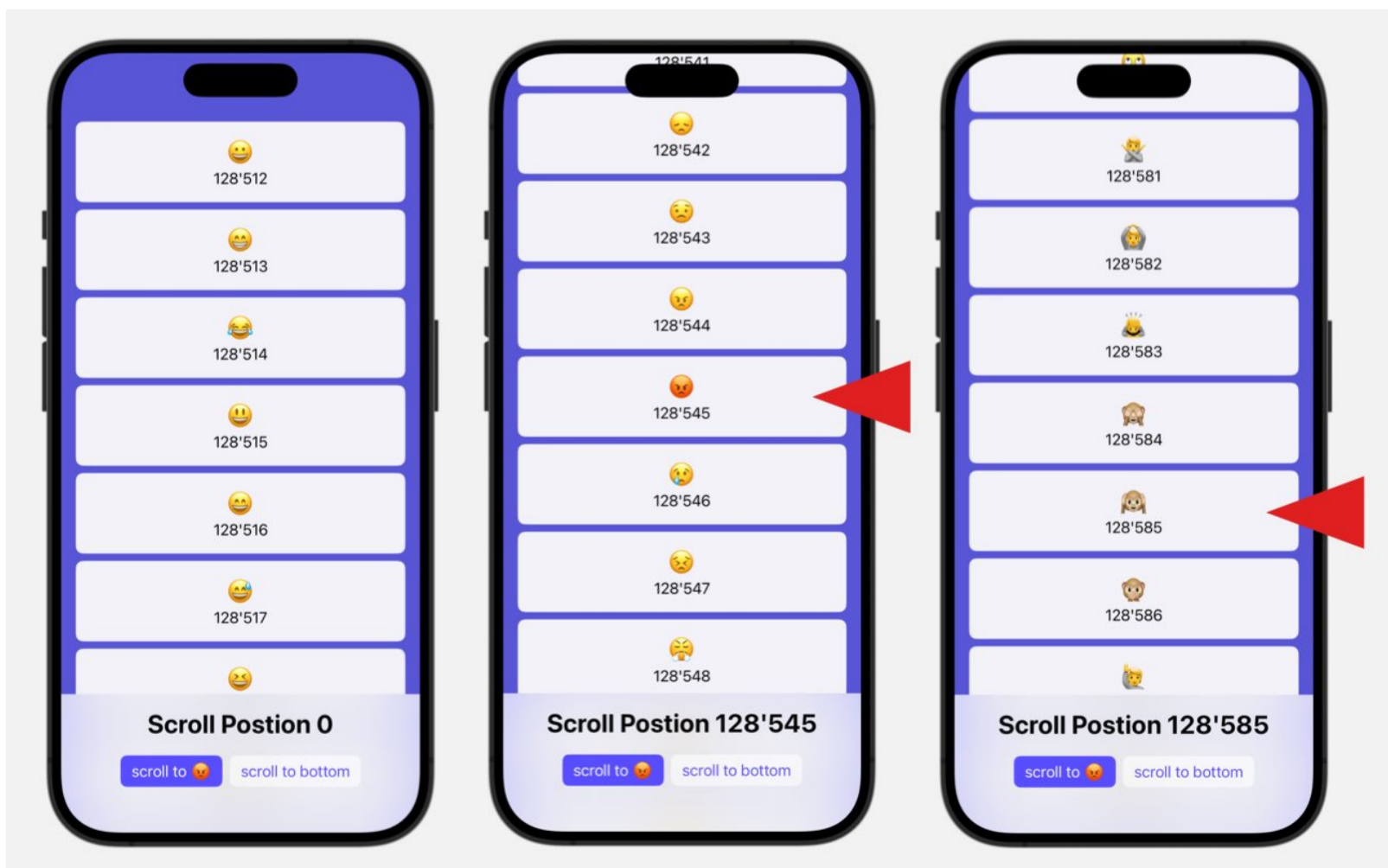
Displaying the Scroll Position

To display the current scroll position, I'll add a title text within the safe area that shows the scrollID. Since scrollID is optional, I'll display zero if it's nil.

```
struct ScrollPositionExampleView: View {
    let emojis = Emoji.examples()
    @State private var scrollID: Int? = nil

    var body: some View {
        ScrollView {
            LazyVStack {
                ...
            }
            .padding()
            .scrollTargetLayout()
        }
        .scrollPosition(id: $scrollID, anchor: .center)
        .background(Color.indigo)
        .safeAreaInset(edge: .bottom, content: {
            Text("Scroll Postion \(scrollID ?? 0)")
                .font(.title).bold()
                .frame(maxWidth: .infinity)
                .padding()
                .background(.thinMaterial)
        })
    }
}
```

If you want to observe changes to the scroll position as you manually scroll, you'll need to use the scrollTargetLayout modifier. Without it, the property won't update as you scroll.



I used the anchor of center which will track the scroll view row in the center. You can see the same id shown in the rows as in the safe area insets at the bottom.

Programmatically Scrolling

To scroll to a specific item, I'll change the scrollID state property. For example, if I want to scroll to the item with the emoji 🤨 which has an id of 128545, I'll set scrollID to that value:

```
@State private var scrollID: Int? = nil

...

Button("scroll to 🤨", action: {
    withAnimation(.easeInOut(duration: 2)) {
        scrollID = 128545
    }
})
```

Since iOS 17 you can now use the scrollPosition modifier to control and observe the scroll position in your ScrollView. In the next lesson, I'll show you more advanced examples, including how to sync two scroll views together using the scroll position.

9.8 Synchronizing Multiple ScrollViews

In this section, I'm going to walk you through an advanced layout technique in SwiftUI where we can synchronise the scrolling behaviour of multiple views. I am going to use the scroll position modifier for this.

Example: Image ScrollView with Preview Grid

Imagine you have a main ScrollView displaying content horizontally and below it you show a grid of all images that act as a preview. We want these two to be in sync, so when you interact with one, the other reflects the changes accordingly. For example, the preview grid should highlight the currently shown image on top. When the user taps on one of the preview images, the top scroll view should scroll to this image.

Let's start by setting up the top scroll view. The images should fit the width of the screen which I accomplish with containerRelativeFrame:

```
struct DoubleScrollExampleView: View {
    let inspirations = NatureInspiration.examples()
    @State private var id: UUID? = nil

    var body: some View {
        VStack {
            ScrollView(.horizontal) {
                LazyHStack(spacing: 0) {
                    ForEach(inspirations) { inspiration in
                        ImageAspectView(imageName: inspiration.imageName,
```

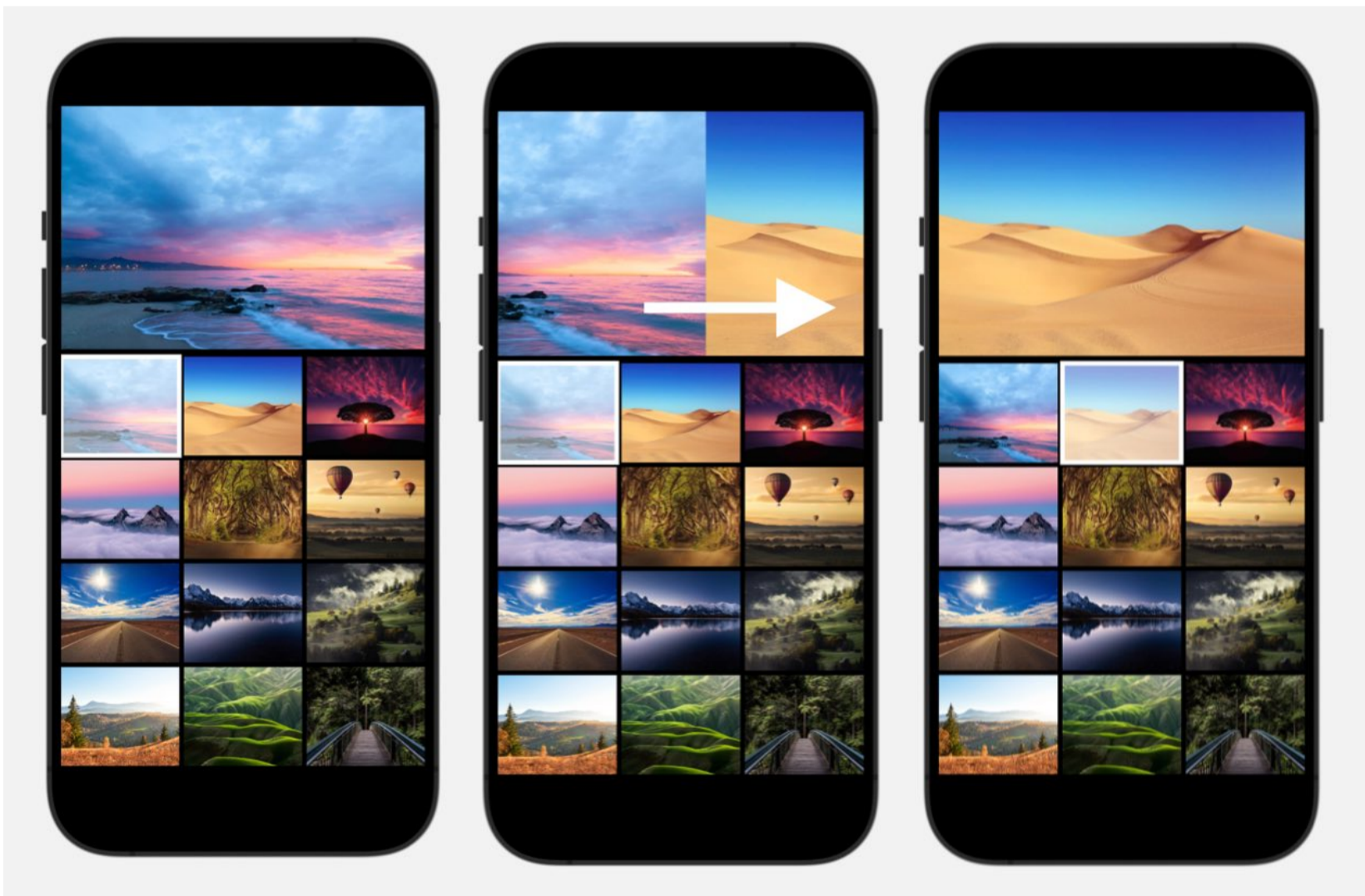
```

        frameAspectRatio: 1.5)
        .containerRelativeFrame(.horizontal)
    }
}
.scrollTargetLayout()
}
.scrollTargetBehavior(.paging)
.scrollPosition(id: $id)
.scrollIndicators(.hidden)
.fixedSize(horizontal: false, vertical: true)
.onAppear {
    id = inspirations.first?.id
}

// >>> preview grid
}
}

```

I use the `scrollPosition` modifier to access the current scroll position. During `onAppear` of this scroll view the state property for the scroll position is set to the first image in the data array. Additionally, the scrollview uses paging target behavior.



Now, let's implement a smaller grid image for preview purposes. This grid will be a visual representation of the thumbnails that the user can interact with. To highlight the currently visible image, we'll add an overlay to the thumbnail grid.

```

ScrollView {
    LazyVGrid(columns: [GridItem(.adaptive(minimum: 100),
        spacing: 5)],

```

```

        alignment: .center,
        spacing: 5, content: {
    ForEach(inspirations) { inspiration in
        ImageAspectRatio(imageName: inspiration.imageName,
            frameAspectRatio: 1.2)
        .overlay {
            if inspiration.id == self.id {
                Color(white: 1, opacity: 0.5)
                Rectangle().stroke(Color.white, lineWidth: 5)
            }
        }
    }
    .onTapGesture {
        // set selected image
        id = inspiration.id
    }
}
})
}
}

```

To synchronize the scroll positions, we'll add tap gestures to each thumbnail image that will programmatically scroll the main image into view. This can be done by setting the state property `id` to the tapped image. This state property is connected to the top scroll view via the `scrollPosition` modifier.

We can also add the option to animate the scrolling for a smoother user experience. This can be done using the `withAnimation` block:

```

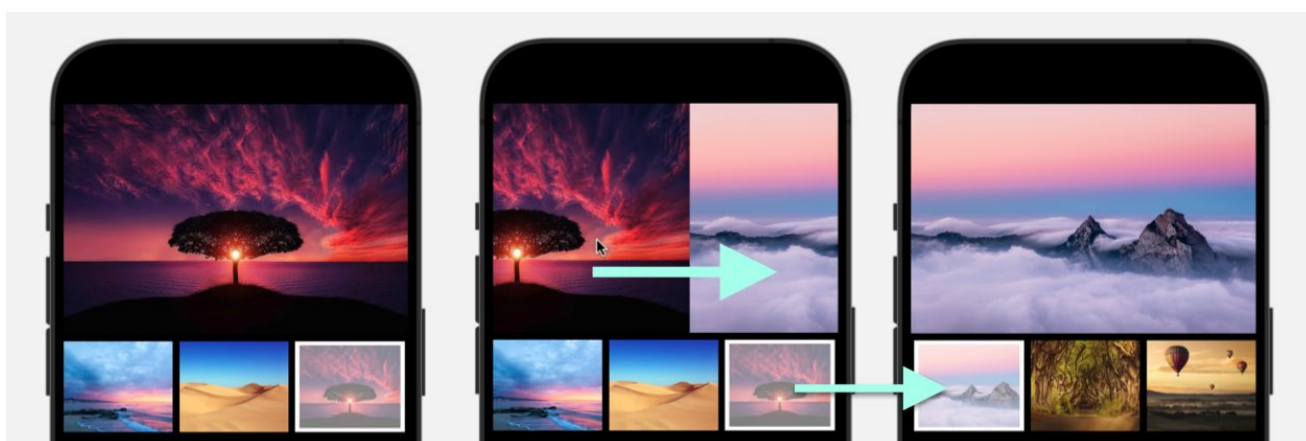
.onTapGesture {
    withAnimation {
        id = inspiration.id
    }
}
}

```

By following these steps, we've set up a basic synchronized scrolling system between our main content view and our thumbnail preview grid.

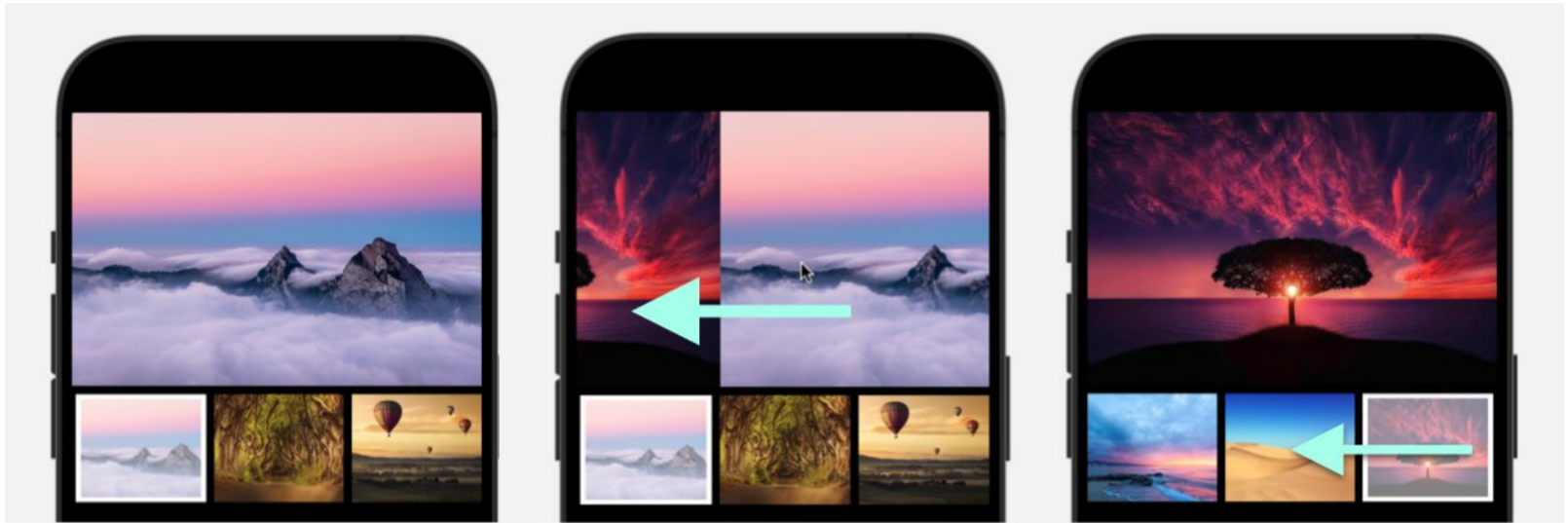
Example: Scrolling a ScrollView when another ScrollView is scrolled by the user

I am going to modify the previous example. I still want to have the top horizontal scroll view. Additionally, I want another horizontal scroll view below that shows 3 image previews at a time.



When the user scrolls in the top scroll view, I want to show the corresponding section in the lower preview scroll view. That means when I scroll right and the top image moves further than the currently shown images below, I will programmatically scroll the bottom scroll view by 3 positions.

Similarly, when I scroll left, I also want to sync the bottom preview scroll view and move the scroll position programmatically by 3 positions:



I first need to change the preview image area to use a horizontal scroll view:

```
struct SyncTwoScrollViewsExample: View {

    let inspirations = NatureInspiration.examples()
    @State private var topSelectedImage:NatureInspiration? = nil
    @State private var bottomSelectedImage:NatureInspiration? = nil

    let numberOfColumns = 3

    var body: some View {
        VStack {
            ScrollView(.horizontal) {
                LazyHStack(spacing: 0) {
                    ...
                }
                .scrollTargetLayout()
            }
            .scrollTargetBehavior(.paging)
            .scrollPosition(id: $topSelectedImage)

            ScrollView(.horizontal) {
                LazyHStack {
                    ForEach(inspirations) { inspiration in
                        ImageAspectView(imageName: inspiration.imageName,
                                        frameAspectRatio: 1.2)
                            .id(inspection)
                            .overlay {
                                if topSelectedImage == inspiration {
                                    Color(white: 1, opacity: 0.5)
                                    Rectangle().stroke(Color.white,
                                                       lineWidth: 5).padding(2)
                                }
                            }
                    }
                }
                .onTapGesture {

```



```

        .scrollPosition(id: $bottomSelectedImage)
    }
    .background(Color.black)
    .onAppear {
        topSelectedImage = inspirations.first
        bottomSelectedImage = inspirations.first
    }
    .onChange(of: topSelectedImage) { oldValue, newValue in
        guard let newValue,
              let topIndex = inspirations.firstIndex(of: newValue) else {
            return
        }

        let index = topIndex % numberOfColumns

        if index == 0 {
            // programmatically scroll right >>
            withAnimation {
                self.bottomSelectedImage = newValue
            }
        } else if index == numberOfColumns - 1 {
            // programmatically scroll left <<
            withAnimation {
                let leadingIndex = topIndex - numberOfColumns + 1
                self.bottomSelectedImage = inspirations[leadingIndex]
            }
        }
    }
}
}
}
}
}
}
}
}
}
}
}

```

To determine when to scroll the other ScrollView, we'll calculate the current column index based on the selected image.

```

let index = topIndex % numberOfColumns

```

If the index is zero, I know that the selected image should be the image on the right. If I set the bottom selected image to that image, I scroll to the next section.

If the index is the last column index, I can scroll left. I need to find the image of the first column in this section:

```

let leadingIndex = topIndex - numberOfColumns + 1
self.bottomSelectedImage = inspirations[leadingIndex]

```

I used the `numberOfColumns` from the bottom scroll view to find the correct positions. Therefore the above implementation also works for other scroll view settings like 4 or 5 columns.

In this example, we've explored advanced techniques to synchronize multiple ScrollViews. We've seen how to manage state, use the `onChange` modifier, and calculate scroll positions to customize the scroll synchronisation.

Key Takeaways

- **State Tracking:** We've learned how to use state properties to track scroll positions and selected items.
- **OnChange Modifier:** We've utilized the `onChange` modifier to synchronize scrolling between multiple `ScrollViews`.
- **Identifiable Views:** We've discussed the importance of using IDs to identify views for programmatic scrolling.
- **SwiftUI's ScrollView API:** We've recognized the power and flexibility of SwiftUI's `ScrollView` API for creating complex, synchronized layouts.

9.9 Default Scroll Position

When you're working with a `ScrollView` in SwiftUI, you might want to control the initial scroll position when the view appears. This is particularly useful if you want your users to start at a specific point in your content, rather than at the top. As of iOS 17, SwiftUI provides a handy modifier for this purpose: `defaultScrollAnchor`. Let's dive into how you can use this to set the default scroll position.

Using `defaultScrollAnchor`

Imagine you have a `ScrollView` filled with emojis, and you want to control where the view starts when it first appears. By default, a `ScrollView` starts at the beginning. However, with the `defaultScrollAnchor` modifier, you can change this behavior. Here's how you can use it:

```
ScrollView {  
    ...  
}  
.defaultScrollAnchor(.top)
```

The `defaultScrollAnchor` modifier accepts a `UnitPoint` value, such as `.bottom`, `.top`, `.leading`, `.trailing`, or `.center`. The direction of your `ScrollView` (horizontal or vertical) will determine which values make sense to use. For a vertical `ScrollView`, you'll typically use `.top`, `.bottom`, or `.center`.

You can also specify a percentage to fine-tune the initial position:

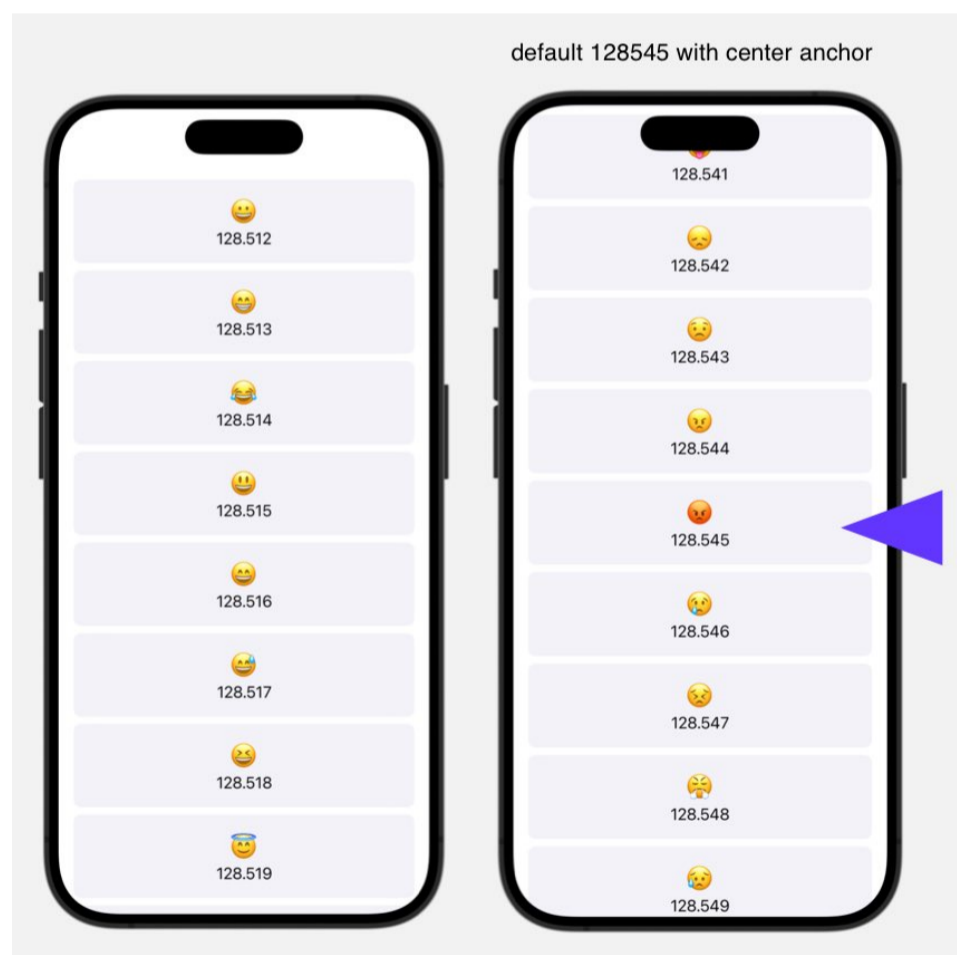
```
ScrollView {  
    ...  
}  
.defaultScrollAnchor(.init(x: 0, y: 0.3))  
// Scrolls to 30% in the y-direction
```


Scrolling to a Specific View

What if you want to scroll to a specific view within your `ScrollView`, like a particular emoji? In that case, `defaultScrollAnchor` might not be the best choice. Instead, you can use the same strategy as before with the `scrollPosition` modifier. Here's an example:

```
struct InitialScrollPositionExampleView: View {
    let emojis: [Emoji]
    @State private var scrollID: Int? = nil

    var body: some View {
        ScrollView {
            LazyVStack {
                ""
            }
        }
        .scrollPosition(id: $scrollID, anchor: .center)
        .onAppear {
            scrollID = 128545
        }
    }
}
```



In this example, you start with a `nil` value for `scrollID`. Then, in the `.onAppear` modifier, you set `scrollID` to the ID of the emoji you want to scroll to. This will cause the `ScrollView` to start at the specified emoji.

You might be tempted to set the `scrollID` in the initializer of your view, but this approach won't work as expected. The `onAppear` method is a reliable place to set the initial scroll position when dealing with dynamic data.

9.10 SCROLLVIEW ANIMATIONS WITH SCROLLTRANSITION

In this section, I'm diving into scroll animations in SwiftUI. iOS 17 introduced some powerful new modifiers, including `scrollTransition`. This modifier makes it very easy to animate views appearing and disappearing inside `ScrollView`. But that's not all—if you're aiming for more intricate effects, like parallax, you'll want to explore the `visualEffects` modifier.

The real challenge here is restraint. It's tempting to go wild with animations, but the key is subtlety. Let's start with `scrollTransition` and see how to create tasteful animations.

Imagine you have an array of inspiration cards laid out in a vertical stack. You're using a `ScrollView` in its default vertical orientation. Now, you want these cards to fade out gracefully as they leave or enter the viewport. This is where `scrollTransition` comes into play.

You can attach the `scrollTransition` modifier to each view within your `ScrollView`. Here's how you might set up a fading animation:

```
struct ScrollTransitionExampleView: View {
    let inspirations = NatureInspiration.examples()
    var body: some View {
        ScrollView {
            ForEach(inspirations) { inspiration in
                InspirationCard(inspiration)
                    .scrollTransition { effect, phase in
                        effect
                            .opacity(phase.isIdentity ? 1 : 0)
                            .scaleEffect(phase.isIdentity ? 1 : 0.8)
                    }
            }
        }
        .contentMargins(10)
    }
}
```



From the definition of the scrollTransition modifier, you can see that you have a lot of parameter customisations available:

```
scrollTransition(_ configuration: ScrollTransitionConfiguration = .interactive,
                axis: Axis? = nil,
                transition: (EmptyVisualEffect, ScrollTransitionPhase)
                    -> some VisualEffect)
```

Changing the Timing of the Animation

The configuration of the scroll transition allows you to change the animation timing. The options for ScrollTransitionConfiguration are:

- **Identity:** this will use the identity of the view and thus not show an animation
- **Animated:** animates the appearance of the view when it leaves the scroll view. You can choose the timing of the animation like `.animated(.bouncy)`
- **Interactive:** synchronizes the animations with the user interaction during scrolling. This is the default setting.

For example, a scroll transition where the views are animated with a bouncy timing curve:

```
.scrollTransition(animated(.bouncy) { effect, phase in
    effect.opacity(phase.isIdentity ? 1 : 0)
})
```

You can also set a transition where the top leading views will have an animation and the bottom trailing views use identity :

```
.scrollTransition(topLeading: .animated(.bouncy),
                  bottomTrailing: .identity) { effect, phase in
    effect.opacity(phase.isIdentity ? 1 : 0)
}
```

Setting the Transition

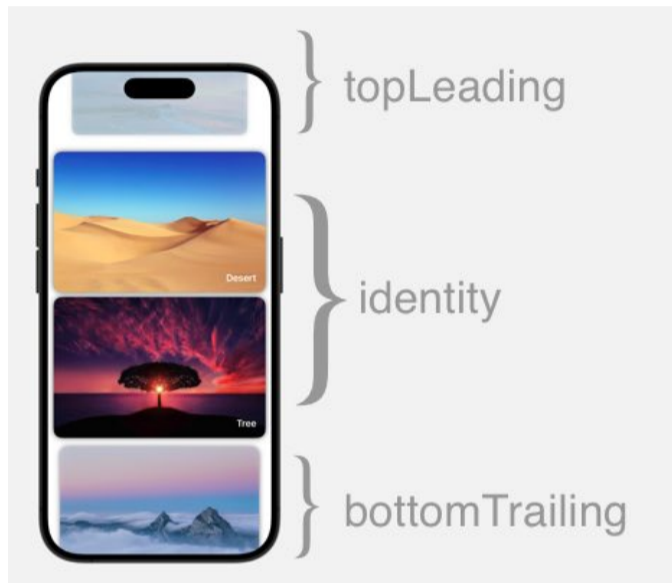
The transition closure gives you access to the EmptyVisualEffect and the ScrollTransitionPhase. The **EmptyVisualEffect represents the view that you are modifying**. It's similar to other modifiers you may have used before. However, not all modifiers are allowed within a scroll view. For example, you cannot use modifiers that change the layout of the view, such as padding or frames. These modifiers are not allowed because they would affect the layout of the other views in the scroll view, causing potential issues with scrolling.

Instead, you can use effects like scale effect, rotation effect, blur, brightness, or opacity. These effects don't change the layout or sizing of the view but rather alter the visual appearance. For example, you can scale down a view as it scrolls out, or fade its opacity.

Inside the scrollTransition closure, you have access to the ScrollTransitionPhase. This gives you information on where each view is inside the scroll view.

The **scroll transition phase** is an enum with three cases:

- top leading: views that are moving in/out in regards to the top leading edge of the scroll View
- identity: The identity case represents the view in the middle, where no animation is applied.
- bottom trailing: views that are moving in/out in regards to the bottom trailing area of the scroll View



For example, if you want to calculate an offset that moves the top leading views to the left and the bottom trailing views to the right:



```
func offset(phase: ScrollTransitionPhase) -> CGFloat {
    switch phase {
        case .topLeading:
            return -50
        case .identity:
```

```

        return 0
    case .bottomTrailing:
        return 50
    }
}

.scrollTransition { effect, phase in
    effect
        .offset(x: offset(phase: phase), y: 0)
}

```

ScrollTransitionPhase has a convenient property, which is **value**:

- top leading: value is equal -1
- identity: value is 0
- bottom trailing: value is equal to 1

Instead of a switch case statement, you can simplify the above function to:

```

func offset(phase: ScrollTransitionPhase) -> CGFloat {
    return phase.value * 50
}

```

In the above example, I set the offset differently for the top and bottom transition views. But oftentimes you want to use the same transition. For example, with a fade in/ out animation. In this case it is easier by using the **isIdentity** property like:

```

.scrollTransition { effect, phase in
    effect.opacity(phase.isIdentity ? 1 : 0)
}

```

Examples: Horizontal ScrollView with ScrollTransition

Now, let's say you want to create a rotation effect. You can define a function that returns an angle based on the phase:

```

func angle(for phase: ScrollTransitionPhase) -> Angle {
    Angle(degrees: (phase.isIdentity ? 0 : -15))
}

```

This will rotate the views as they enter and exit, creating a dynamic visual effect.

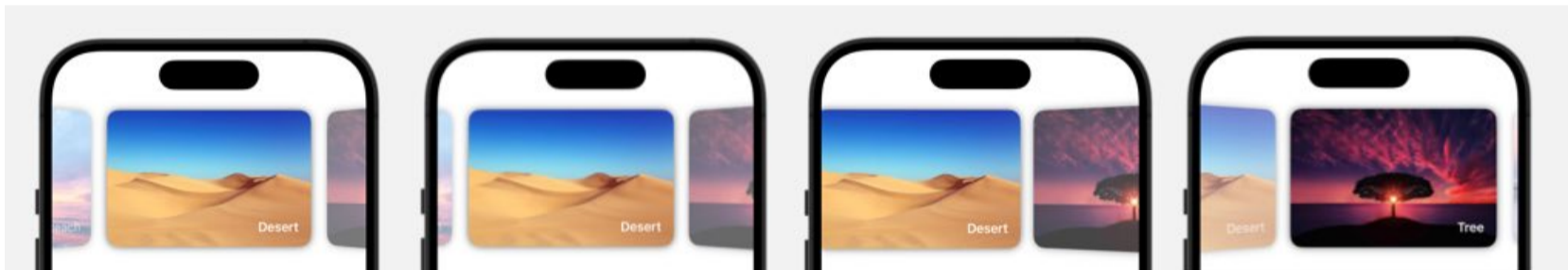


```

ScrollView(.horizontal) {
    LazyHStack {
        ForEach(inspirations) { inspiration in
            InspirationCard(inspiration: inspiration,
                frameAspectRatio: 0.5)
            .scrollTransition { effect, phase in
                effect
                .scaleEffect(phase.isIdentity ? 1 : 0.9)
                .opacity(phase.isIdentity ? 1 : 0)
                .rotation3DEffect(angle(for: phase),
                    axis: (x: 1, y: 0, z: 0),
                    anchor: .top)
            }
        }
        .containerRelativeFrame(.horizontal,
            count: 11,
            span: 3,
            spacing: 10.0)
    }
}
.contentMargins(10)

```

You can also generate a cinema transition effect like:



I am animating the opacity, scale and rotation of the views:

```

ScrollView(.horizontal) {
    LazyHStack(spacing: 0) {
        ForEach(inspirations) { inspiration in
            InspirationCard(inspiration: inspiration,
                frameAspectRatio: 1.5)
            .id(inspiration.id)
            .scrollTransition { effect, phase in
                effect
                .opacity(phase.isIdentity ? 1 : 0.5)
                .scaleEffect(phase.isIdentity ? 0.95 : 1.05)
                .rotation3DEffect(Angle(degrees: phase.value * -15),
                    axis: (x: 0, y: 1, z: 0),
                    anchor: .center)
            }
        }
        .containerRelativeFrame(.horizontal,
            count: 11,
            span: 8,
            spacing: 10.0)
    }
}
.contentMargins(10)
.scrollPosition(id: $scrollID, anchor: .center)

```

```
.onAppear(perform: {  
    scrollID = inspirations[1].id  
})
```

This effect looks best when one view is cantered in the scroll view. Therefore I need to set the scroll position away from the leading edge. In this case, I use again the scroll position modifier to programmatically scroll to the second image in the array:

```
.scrollPosition(id: $scrollID, anchor: .center)  
.onAppear(perform: {  
    scrollID = inspirations[1].id  
})
```

As you experiment with `scrollTransition`, remember to keep the animations subtle. A gentle fade, a slight scale down, or a soft rotation can add a touch of elegance without overwhelming the user. The goal is to enhance the scrolling experience, not detract from it.

In the next section, we'll explore the `visualEffects` modifier, which opens up even more possibilities for animating views within the scroll view. Get ready to take your animations to the next level!

9.11 ANIMATIONS WITH VISUALEFFECT

In this section, we're going to dive into some advanced animation techniques using the `VisualEffect` modifier. This approach is akin to how I utilize the `GeometryReader` in the background to animate views within a `ScrollView`. With `VisualEffect`, you're provided with a `GeometryProxy`, which is crucial for knowing the precise position of each view within a `ScrollView`. It requires **iOS 17 or higher**.

This isn't limited to just the views at the top or bottom; it includes all the views in between, allowing for smooth animations as they move through your `ScrollView`. This technique can also be leveraged to create a parallax effect. But before we jump into these fancier examples, let's set up a basic `ScrollView` to understand how the `VisualEffect` modifier is used.

I'll start with a `ScrollView` and, for demonstration purposes, use a range of 0 to 100 to display a row with an index number. To add some style, I'll wrap each row in a `GroupBox` and apply a custom `GroupBox` style we created earlier, the `OrangeGroupBoxStyle`. `VisualEffects` gives you the view itself "content" and the `geomtryProxy`:

```
ScrollView {
    ForEach(0..<100) { index in
        GroupBox {
            Text("Row \(index)")
                .frame(maxWidth: .infinity)
        }
        .groupBoxStyle(OrangeGroupBoxStyle())
        .visualEffect { content, geometryProxy in
            content
        }
    }
}
```



Example: Animating Views with Offset

Initially, I'll set up a basic animation that offsets the row by half the width of the GeometryProxy:

```
.visualEffect { content, geometryProxy in
    content
        .offset(x: offset(for: geometryProxy), y: 0)
}
```

I use the following function to calculate the offset. This function will take the GeometryProxy as an argument and return a CGFloat, representing the offset in the x-direction (since we're scrolling vertically):

```
func offset(for proxy: GeometryProxy) -> CGFloat {
    let scrollHeight = proxy.bounds(of: .scrollView)?.height ?? 100
    let cellOffset = proxy.frame(in: .scrollView).midY

    return cellOffset - scrollHeight / 2
}
```

To extract useful properties from the proxy, I can determine the cell height and the ScrollView's height. Then, I calculate the cell offset, which is the distance from the top of the ScrollView to the middle of the cell. This gives me the information I need to create animations for individual views.

For instance, if a cell is exactly in the middle of the ScrollView, I don't want it to offset. I can calculate this by checking if the cell offset is half the ScrollView's height. Using this logic, I can apply an x-offset to move the cells horizontally, creating a dynamic effect as they scroll.

Example: Animating Views with Opacity

Next, let's calculate an opacity value. The cell in the middle of the ScrollView should be the brightest, while others become more opaque.

```
.visualEffect { content, geometryProxy in
    content
        .opacity(opacity(for: geometryProxy))
}
```

This requires similar calculations to those used for the offset, but now I'm interested in the distance from the center in terms of percentage. I'll use this percentage to adjust the opacity of each cell.

```
func opacity(for proxy: GeometryProxy) -> Double {
    let scrollHeight = proxy.bounds(of: .scrollView)?.height ?? 100
    let cellOffset = proxy.frame(in: .scrollView).midY
    let distanceFromCenter = abs(scrollHeight / 2 - cellOffset)

    return 1 - Double(distanceFromCenter / scrollHeight * 2)
}
```

Example: Animating Views with ScaleEffect

Additionally, we can apply a scale effect:

```
.visualEffect { content, geometryProxy in
    content
        .scaleEffect(scaleAmount(for: geometryProxy))
}
```

Similar to the example with the opacity, I want to make the views smaller the further away they are from the center of the scroll view. The calculations are thus very similar and I create a reusable function that calculates the distance from the center. The returned value is in a range from 0 to 1. I can amplify the effect by increasing an additional scale factor:

```
func value(for proxy: GeometryProxy, scale: Double) -> Double {
    let scrollHeight = proxy.bounds(of: .scrollView)?.height ?? 100
    let cellOffset = proxy.frame(in: .scrollView).midY
    let distanceFromCenter = abs(scrollHeight / 2 - cellOffset)

    return 1 - Double(distanceFromCenter / scrollHeight * 2) * scale
}

func scaleAmount(for proxy: GeometryProxy) -> CGSize {
    let value = value(for: proxy, scale: 0.3)
    return CGSize(width: value, height: value)
}
```

By creating a function to calculate the scale amount based on the cell's distance from the center, we can make cells appear larger or smaller as they move through the ScrollView.

Example: 2-Dimensional Animations

Finally, let's experiment with a dynamic data example using a two-dimensional array for a LazyVGrid. By applying the VisualEffect modifier to each emoji view, we can create a shimmering effect as they scroll.

```
struct AnimatedLazyVGridExampleView: View {
    let emojis = Emoji.examples()

    var body: some View {
        ScrollView {
            LazyVGrid(columns: [GridItem(.adaptive(minimum: 80), spacing: 10)],
                alignment: .center,
                spacing: 10, content: {
                ForEach(emojis) {
                    EmojiView(emoji: $0)
                        .visualEffect { content, geometryProxy in
                            content
                                .opacity(opacity(for: geometryProxy))
                        }
                }
            })
        }
    }
}
```

```

        .padding()
    }
    .background(Color.black)
}

func value(for proxy: GeometryProxy, scale: Double) -> Double {
    let scrollHeight = proxy.bounds(of: .scrollView)?.height ?? 100
    let cellOffsetY = proxy.frame(in: .scrollView).midY
    let distanceFromCenterY = abs(scrollHeight / 2 - cellOffsetY)

    let scrollWidth = proxy.bounds(of: .scrollView)?.width ?? 100
    let cellOffsetX = proxy.frame(in: .scrollView).midX
    let distanceFromCenterX = abs(scrollWidth / 2 - cellOffsetX)

    return 1 - Double(distanceFromCenterX / scrollWidth) -
        Double(distanceFromCenterY / scrollHeight)
}

func opacity(for proxy: GeometryProxy) -> Double {
    return value(for: proxy, scale: 0.6)
}
}

```



Remember, the key to these animations is understanding the position of each cell. The GeometryProxy gives you access to a wealth of properties, including the bounds and frame of the ScrollView and the size and position of each cell. With this information, you can create intricate animations that respond to the scrolling behavior, enhancing the user experience.

In the next lesson, we'll continue to explore the power of the VisualEffects by using it to create advanced animations for parallax effects.

9.12 Parallax Example

In this section, I'm going to show you how to implement some captivating parallax effects. We'll work through two examples to get a good grasp of the concept. You'll notice in the first example, there are cards with text and a description layered above. As you scroll, not only does the text move, but the images also have a subtle motion, creating a dynamic visual experience.



Begin by setting up a horizontal ScrollView with an HStack. For each piece of inspiration data, display an image with the correct aspect ratio. I am using `containerRelativeFrame` in the horizontal axis to ensure that each of the sections is the same width as the scroll view:

```
struct ParallaxEffectView: View {
    let inspirations = NatureInspiration.examples()

    var body: some View {
        ScrollView(.horizontal) {
            HStack(spacing: 0) {
                ForEach(inspirations) { inspiration in
                    VStack(spacing: 20) {
                        VStack {
                            Text(inspiration.name)
                                .font(.title)
                            Text(inspiration.description)
                                .font(.caption)
                                .multilineTextAlignment(.center)
                                .lineLimit(2)
                        }

                        ImageAspectView(imageName: inspiration.imageName,
                                       frameAspectRatio: 1.5,
                                       cornerRadius: 15)
                            .shadow(radius: 5)
                    }
                    .padding()
                    .containerRelativeFrame(.horizontal)
                }
            }
        }
        .scrollTargetBehavior(.paging)
    }
}
```

Adding Text with Parallax

To create the parallax effect, we want the text to move at a different pace than the scroll view. To achieve this, you'll need to apply an offset to the `VStack` that contains the text. Use a visual effects modifier to attach the desired effect to the text views:

```
ScrollView(.horizontal) {
    HStack(spacing: 0) {
        ForEach(inspirations) { inspiration in
            VStack(spacing: 20) {
                VStack {
                    ...
                }
                .visualEffect { content, geometryProxy in
                    content.offset(x: textOffset(for: geometryProxy),
                                   y: 0)
                }
                // image
            }
        }
    }
}

func textOffset(for proxy: GeometryProxy) -> CGFloat {
    let scrollWidth = proxy.bounds(of: .scrollView)?.width ?? 100
    let cellOffset = proxy.frame(in: .scrollView).midX

    return (cellOffset - scrollWidth / 2) * 0.6
}
```

Implementing the Image Parallax

The image parallax is a bit trickier. You want the image to move within its frame, but the frame itself should remain static. After setting the image in place, use a `clipShape` with a `RoundedRectangle` to define the visible area.

```
ImageAspectView(imageName: inspiration.imageName,
                 frameAspectRatio: 2,
                 cornerRadius: 0)
.padding(.horizontal, -70)
.visualEffect { content, geometryProxy in
    content.offset(x: imageOffset(for: geometryProxy),
                  y: 0)
}
.clipShape(RoundedRectangle(cornerRadius: 15))
.shadow(radius: 5)

func imageOffset(for proxy: GeometryProxy) -> CGFloat {
    let scrollWidth = proxy.bounds(of: .scrollView)?.width ?? 100
    let cellOffset = proxy.frame(in: .scrollView).midX

    return (cellOffset - scrollWidth / 2) * 0.15
}
```

To ensure the image has enough area to move within the clipped frame, you might need to apply negative padding. This is a quick trick to make the image larger without altering its actual size. Adjust the padding and offset values until the image moves subtly within the frame, creating that gentle parallax effect.

To illustrate how the image is moving within the frame, I added a red border which represents the clipShape:



The image area (red shape) is moving across the image. This gives the impression that the image is moving within its card view.

In this example, I've demonstrated how to use visual effects modifiers to animate different elements separately. By calculating the offsets based on the scroll view's position, you can create a stunning parallax effect for both text and images. With this approach, you have complete control over the animation, allowing you to create a truly interactive and engaging user interface.

Remember, the key to a successful parallax effect is subtlety and precision. Take your time to adjust the movements until everything works together harmoniously.

9.13 Background Parallax Effect

In this section, I'm going to show you how to create a parallax effect within a ScrollView using visual effects. Imagine you have a set of nature-inspired data, including titles and text, and you want to add a background image that moves subtly as you scroll horizontally. Let's dive into implementing this animation.



First, create a new file named `BackgroundParallaxEffectView`. You'll use the same inspiration data as before. Inside, place a horizontal ScrollView that contains your titles and text. It's crucial to maintain a clean structure, so avoid excessive nesting and ensure your cells are appropriately sized. Utilize the `containerRelativeFrame` in the horizontal direction and set the scroll target behavior to paging for a smooth scrolling experience.

```
struct BackgroundParallaxEffectView: View {
    let inspirations = NatureInspiration.examples()

    var body: some View {
        ScrollView(.horizontal) {
            HStack(spacing: 0) {
                ForEach(inspirations) { inspiration in
                    VStack {
                        Text(inspiration.name)
                            .font(.title)
                        Text(inspiration.description)
                            .multilineTextAlignment(.center)
                    }
                    .padding()
                    .containerRelativeFrame(.horizontal)
                }
            }
        }
        .scrollTargetBehavior(.paging)
    }
}
```

Now, you want to add an image that will serve as the background of your ScrollView. You might think to add it outside the ScrollView, but that won't give you the desired effect. Instead, insert an image within the ScrollView and set it to be resizable and fill the available space.


```

struct BackgroundParallaxEffectView: View {
    let inspirations = NatureInspiration.examples()

    var body: some View {

        ScrollView(.horizontal) {
            HStack(spacing: 0) {
                ForEach(inspirations) { inspiration in
                    ""
                }
            }
            .background(alignment: .leading) {
                Image(.leaves)
                    .resizable()
                    .scaledToFill()
                    .containerRelativeFrame(.horizontal, { length, axis in
                        length * 2
                    })
                    .visualEffect { content, geometryProxy in
                        content.offset(x: imageOffset(for: geometryProxy),
                                      y: 0)
                    }
            }
        }
        .scrollTargetBehavior(.paging)
    }

    func imageOffset(for proxy: GeometryProxy) -> CGFloat {
        let scrollPosition = proxy.bounds(of: .scrollView)?.minX ?? 0
        let offset = scrollPosition / CGFloat(inspirations.count - 1)
        return scrollPosition - offset
    }
}

```

To achieve the parallax effect, you need to calculate the image offset for the background image based on the ScrollView's current position.

In this example, I choose to align the background image to the leading edge:

```

.background(alignment: .leading) {
    ""
    .visualEffect { content, geometryProxy in
        content.offset(x: imageOffset(for: geometryProxy),
                      y: 0)
    }
}

```

If I then offset it by the ScrollView's minimum X value. This makes it appear as if the background isn't moving when, in fact, it's moving in sync with the ScrollView.

```

func imageOffset(for proxy: GeometryProxy) -> CGFloat {
    let scrollPosition = proxy.bounds(of: .scrollView)?.minX ?? 0
    return scrollPosition
}

```

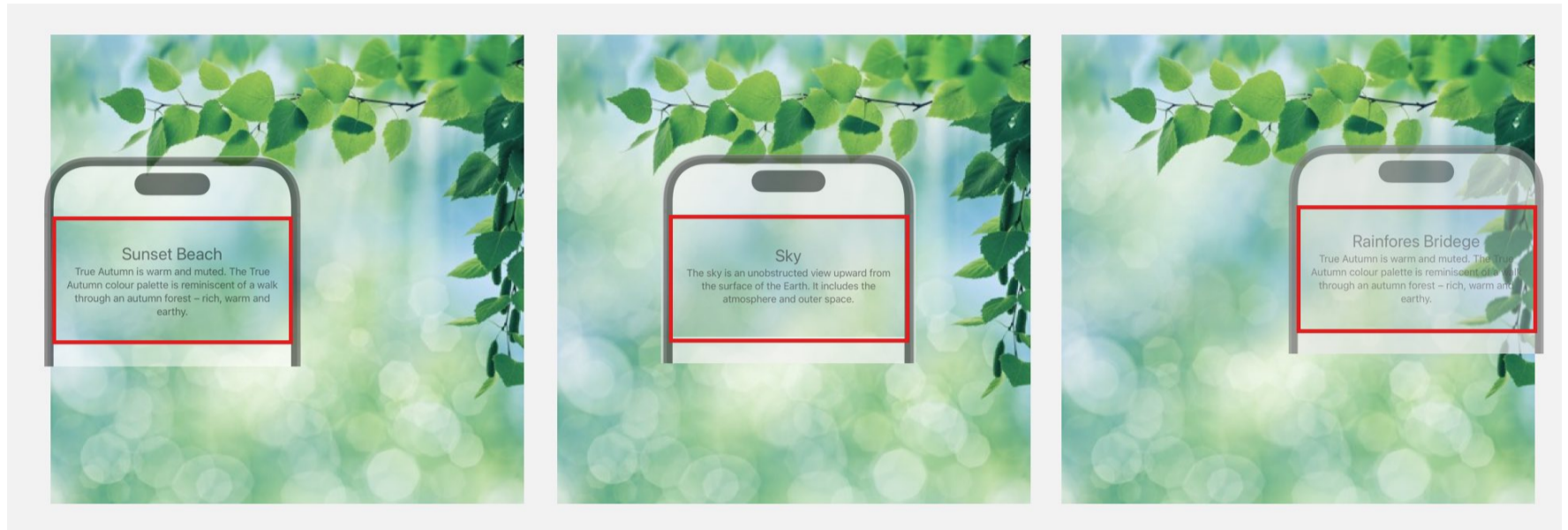
To add a small additional offset that moves the background image, I add an additional offset value:

```

func imageOffset(for proxy: GeometryProxy) -> CGFloat {
    let scrollPosition = proxy.bounds(of: .scrollView)?.minX ?? 0
    let offset = scrollPosition / CGFloat(inspirations.count - 1)
    return scrollPosition - offset
}

```

This increases the total offset to 2 times the width of the background image width.



Make sure the background image is large enough to accommodate the movement. You can use the `containerRelativeFrame` modifier to set the image's width to be twice as large as the visible area.

```

.background(alignment: .leading) {
    Image(.leaves)
        .resizable()
        .scaledToFill()
        .containerRelativeFrame(.horizontal, { length, axis in
            length * 2
        })
        .visualEffect { content, geometryProxy in
            content.offset(x: imageOffset(for: geometryProxy),
                y: 0)
        }
}

```

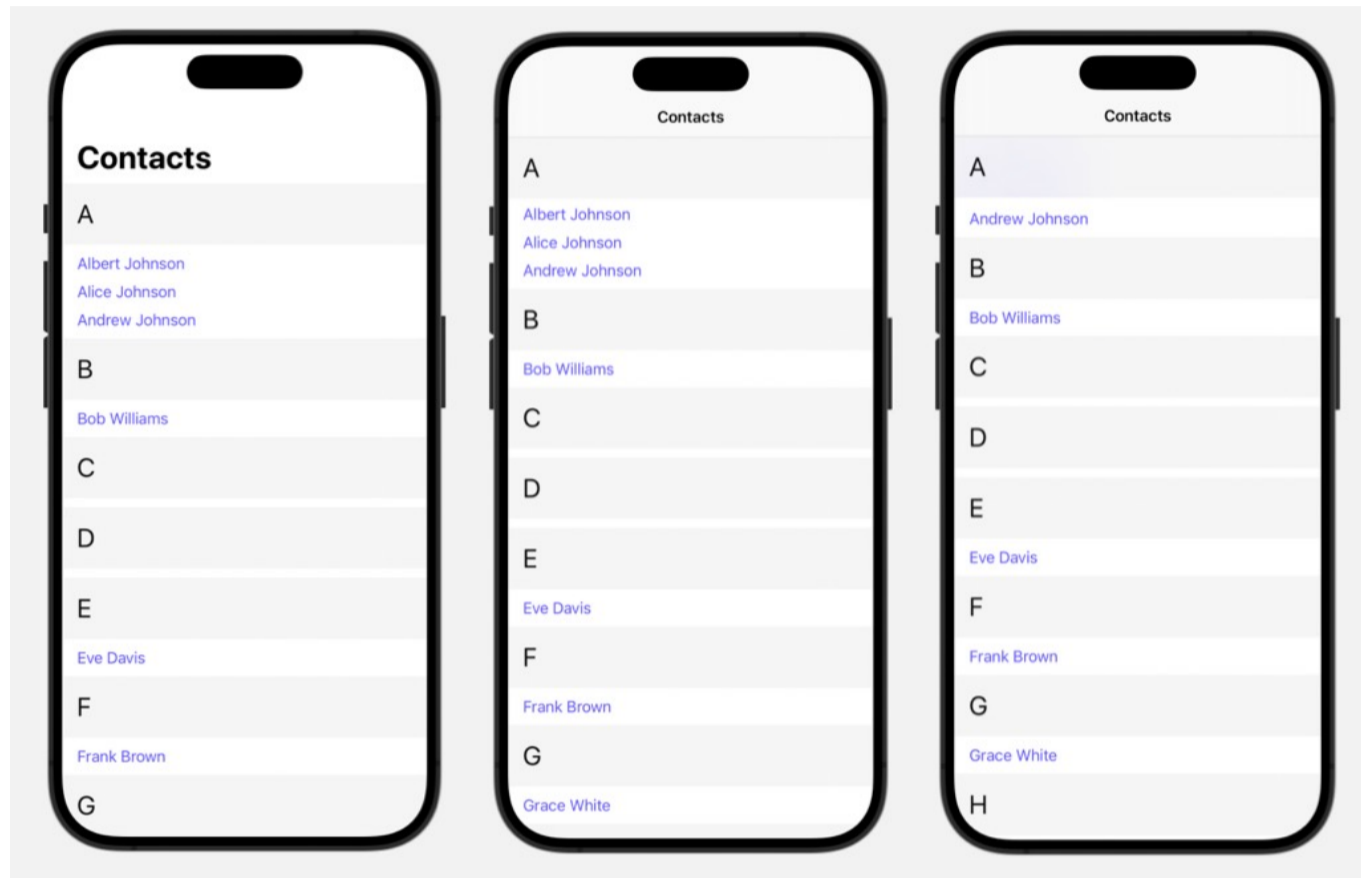
After setting up the offset, you'll see the background image moving slightly as you scroll through the images.

By adding a resizable image to the background and calculating the offset based on the `ScrollView`'s position, you create a beautiful parallax effect. Remember to keep the background image within the `ScrollView` to get the correct geometry proxy information for the visual effects.

This visual effects modifier is quite handy, though it requires precise calculations to use effectively. With this example, you've seen how to apply a parallax effect to a background image.

9.14 PINNED VIEWS

In this section, I'll show you how to create pinned views within a ScrollView that stick at the top as you scroll, similar to what you might have seen in the Contacts app on your iPhone. This feature is particularly useful when you want to categorize items and maintain the category header visible while scrolling through the list.



Creating a Contact Model

First, let's set up some data. Here's a simple Contact struct:

```
struct Contact: Identifiable {  
  
    let id: UUID = UUID()  
    var name: String  
    var phoneNumber: String  
  
    static func examples() -> [Contact] {  
        [  
            Contact(name: "John Doe", phoneNumber: "123-456-7890"),  
            Contact(name: "Jane Smith", phoneNumber: "987-654-3210"),  
            Contact(name: "Alice Johnson", phoneNumber: "555-123-4567"),  
            Contact(name: "Albert Johnson", phoneNumber: "555-123-4567"),  
            Contact(name: "Andrew Johnson", phoneNumber: "555-123-4567"),  
            Contact(name: "Bob Williams", phoneNumber: "111-222-3333"),  
            Contact(name: "Eve Davis", phoneNumber: "999-888-7777"),  
            Contact(name: "Frank Brown", phoneNumber: "444-333-2222"),  
            Contact(name: "Grace White", phoneNumber: "888-777-6666"),  
            Contact(name: "Hank Taylor", phoneNumber: "777-666-5555"),  
            Contact(name: "Ivy Lee", phoneNumber: "666-555-4444"),  
            Contact(name: "Kate Young", phoneNumber: "555-444-3333"),  
            Contact(name: "Louis Miller", phoneNumber: "222-333-4444"),  
            Contact(name: "Mary Adams", phoneNumber: "333-444-5555"),  
        ]  
    }  
}
```

```

    }
}

extension Contact: Comparable {
    static func < (lhs: Contact, rhs: Contact) -> Bool {
        lhs.name < rhs.name
    }
}

```

Implementing Pinned Headers

Now, let's get to the exciting part. In your ScrollView, you'll want to use a LazyVStack or LazyHStack because they allow you to pin header views. Here's how you can set up your alphabetized list:

```

struct PinnedHeadersScrollView: View {

    let contacts = Contact.examples()
    let alphabet: Set<Character> = Set("ABCDEFGHIJKLMNOPQRSTUVWXYZ")

    var body: some View {
        NavigationStack {
            ScrollView {
                LazyVStack(alignment: .leading,
                           spacing: 10,
                           pinnedViews: [.sectionHeaders]) {

                    ForEach(alphabet.sorted(), id: \.self) { letter in
                        Section {
                            ForEach(contacts(for: letter)) { contact in
                                NavigationLink(contact.name) {
                                    Text(contact.name)
                                }
                                .padding(.horizontal)
                            }
                        } header: {
                            Text(String(letter))
                                .font(.title)
                                .padding()
                                .frame(maxWidth: .infinity, alignment: .leading)
                                .background(.thinMaterial)
                        }
                    }
                }
            }
        }
        .navigationTitle("Contacts")
    }
}

func contacts(for letter: Character) -> [Contact] {
    contacts.filter({ $0.name.first == letter }).sorted()
}

```

Notice how I've used a Section to group the contacts under their respective letter headers. The LazyVStack initializer includes the pinnedViews parameter, where I've specified .sectionHeaders to pin the headers.

```

ScrollView {
    LazyVStack(alignment: .leading,
              spacing: 10,
              pinnedViews: [.sectionHeaders]) {
        ...
    }
}

```

Styling the Pinned Headers

To make the headers stand out and behave like those in the Contacts app, you'll need to add some styling. Here's how you can give the headers a background and ensure they're nicely formatted:

```

Section {
    ForEach(contacts(for: letter)) { contact in
        ...
    }
} header: {
    Text(String(letter))
        .font(.title)
        .padding()
        .frame(maxWidth: .infinity, alignment: .leading)
        .background(.thinMaterial)
}

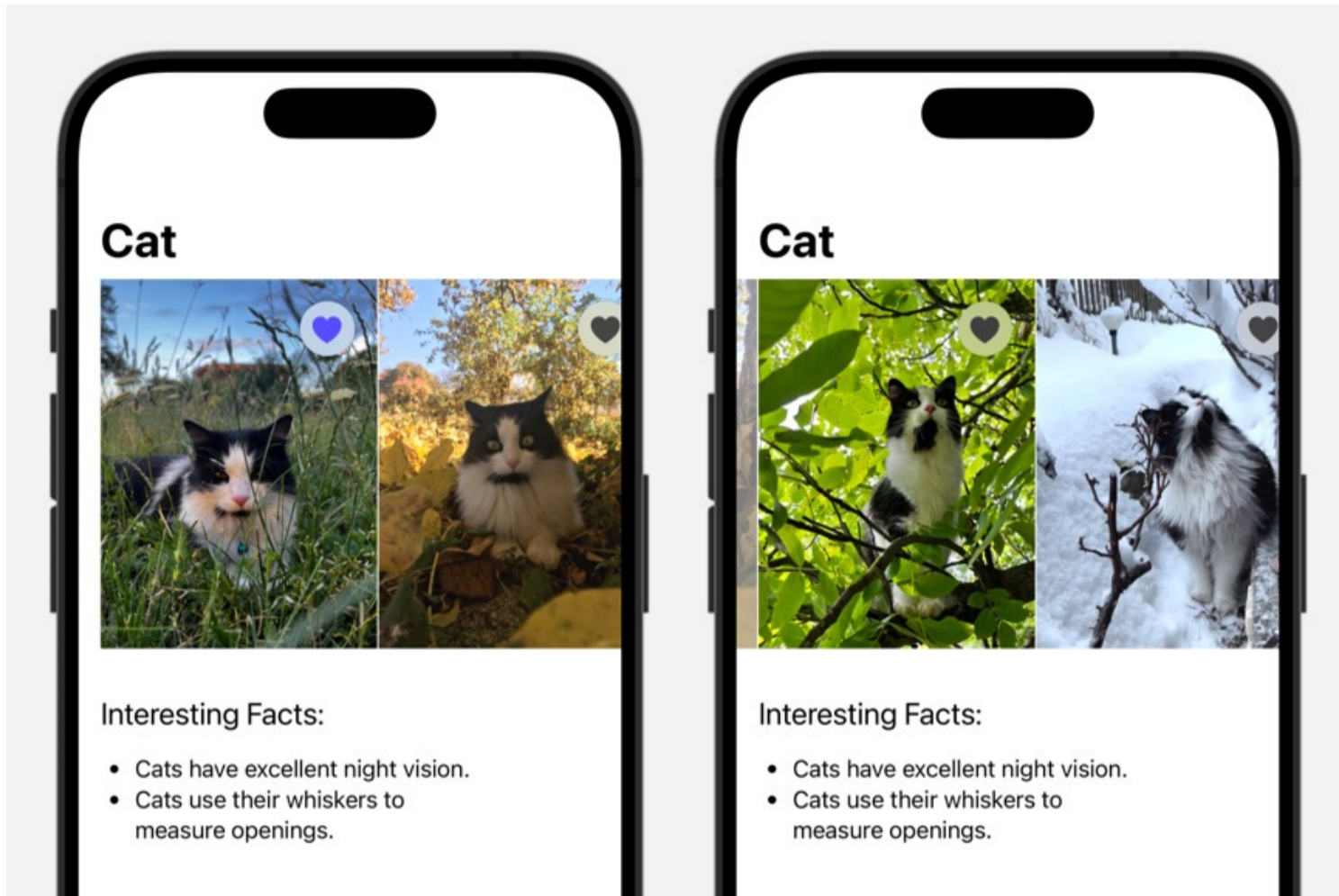
```

This will give your headers a translucent background, similar to the blur effect in iOS, and ensure they span the entire width of the screen.

Remember, you can only pin section headers or footers, and you need to use a `Section` to let the system know which views to pin. With a bit of styling, you can quickly create a professional-looking list with pinned views.

CHALLENGE 🖐️ SCROLLVIEW

You've been working on the Pet Pal app, particularly on the gallery view. However, there are a couple of screens that could use a bit more excitement, considering we're dealing with pets here. We want to add fun animations and improve the layout to make it more engaging.



When you tap on one of the cards in the gallery view, you're taken to the pet detail view. For the images, I added a horizontal ScrollView with a LazyHStack and a spacing of one, creating a subtle separation between the images. I included padding around the content to avoid the images being cut off at the edges, which I find unappealing.

```
struct PetDetailView: View {  
  
    let pet: Pet  
    @Binding var favoriteImages: Set<String>  
  
    var body: some View {  
        ScrollView {  
            VStack(alignment: .leading) {  
                ScrollView(.horizontal) {  
                    LazyHStack(spacing: 1) {  
                        ForEach(pet.images, id: \.self) { imageName in  
                            Image(imageName)  
                                .resizable()  
                                .scaledToFit()  
                                .overlay(alignment: .topTrailing) {  
                                    FavoritePetButton(imageName: imageName,  
                                                            favoriteImages: $favoriteImages)  
                                }  
                                .padding()  
                        }  
                    }  
                }  
            }  
        }  
        .scrollTransition { contentView, phase in  
            contentView.opacity(phase.isIdentity ? 1 : 0.5)  
        }  
    }  
}
```

```

        }
    }
    .scrollTargetLayout()
}
.scrollClipDisabled()
.scrollTargetBehavior(.viewAligned)
.contentMargins(-10, for: .scrollIndicators)
.containerRelativeFrame(.vertical, { length, axis in
    length * 0.4
})
.padding(.bottom)
.scrollIndicatorsFlash(onAppear: true)

Text("Interesting Facts:")
""
}
.padding([.horizontal, .bottom])
}
#if os(iOS)
.navigationBarTitle(pet.type.rawValue.capitalized)
#endif
}
}

```

I used the `petImages` array for the content, and since these are string values, I had to use `id: \.self`. Each image is displayed using a resizable view that is scaled to fit. I prefer this because it maintains the different aspect ratios, which is particularly nice for varied images like those of fish.

I also implemented **scroll target behavior with view alignment**. This ensures the images don't touch the edges too much, which I think looks better. I added some **scroll transition effects**, like opacity changes, to subtly highlight the images as they come into view.

I set the height of the image column to 40% of the available space, ensuring it scales nicely with different screen sizes. This excludes the toolbar area, so it's 40% from the top of the view to the bottom.

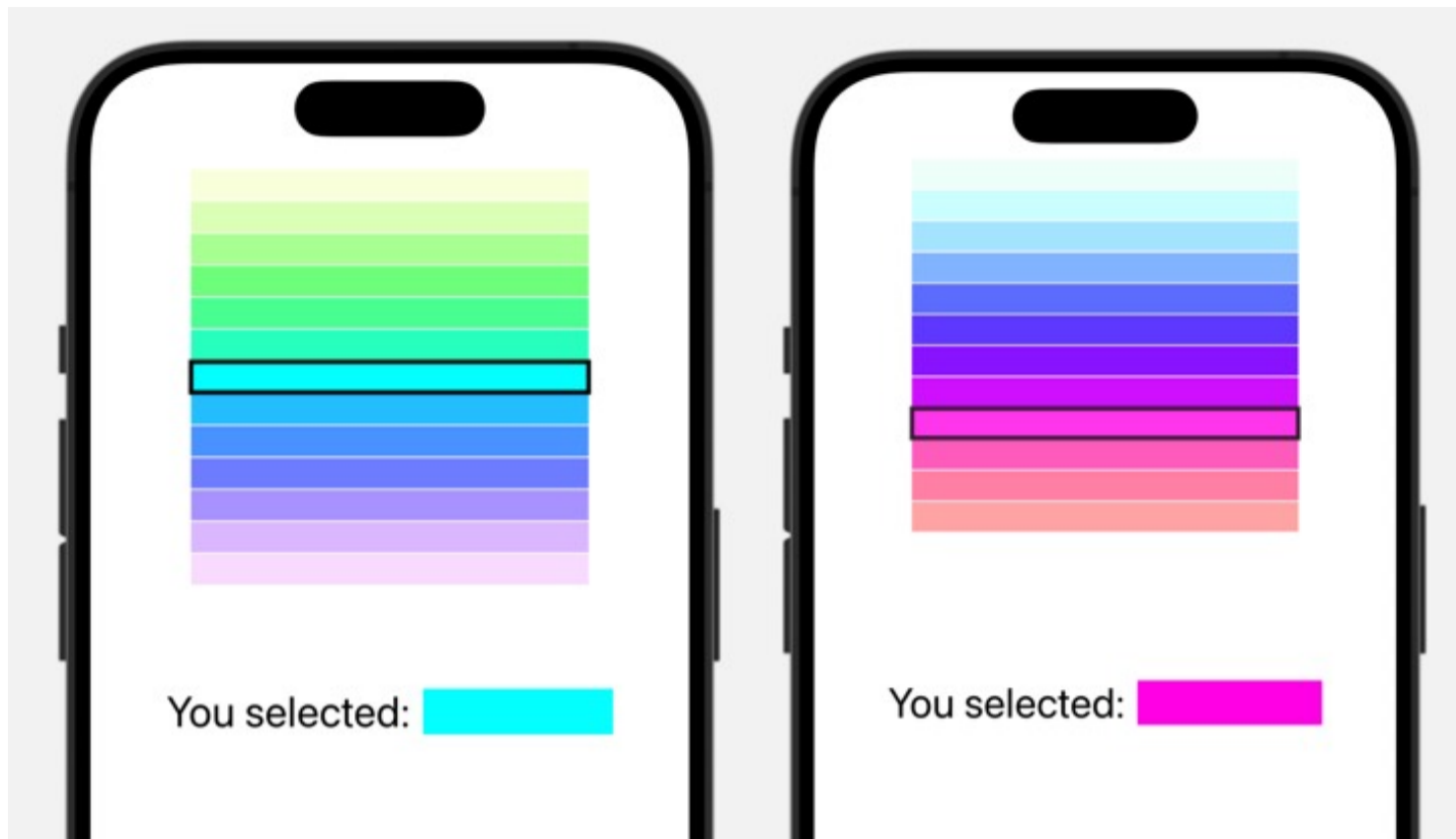
```

.containerRelativeFrame(.vertical, { length, axis in
    length * 0.4
})

```

CUSTOM PICKER VIEW CHALLENGE

ScrollView is very powerful and with the addition of scrollPosition more advanced use cases are possible. You can for example create the following color picker:



The basic is a vertical scroll view that shows a range of colors. You can use color from hue to easily iterate over different colors:

```
ScrollView {
  LazyVStack(spacing: 1) {
    ForEach(0..<21) { index in
      ColorRow(hueValue: index)
        .frame(height: 20)
    }
  }
}
.aspectRatio(1, contentMode: .fit)
```

Next, you can create a state property to track the selected color and bind it to the scroll view position:

```
@State private var selectedColorIndex: Int? = nil

ScrollView {
  LazyVStack(spacing: 1) {
    ForEach(0..<21) { index in
      ColorRow(hueValue: index)
        .frame(height: 20)
        .id(index)
        .overlay {
          if index == selectedColorIndex {
            Rectangle()
              .stroke(Color.black, lineWidth: 3)
          }
        }
    }
  }
}
```



```

    }

    }
    .scrollTargetLayout()
}
.scrollPosition(id: $selectedColorIndex, anchor: .center)

```

I need to add space before and after the ForEach to allow over-scrolling. The user can thus also select colors at the beginning and end of the scroll view. Since I use an aspect ratio of 1 for the scroll view and I need to add a white area with half the scroll view height, I can use white placeholders like:

```

ScrollView {
    LazyVStack(spacing: 1) {
        Color.white
            .aspectRatio(2, contentMode: .fit)

        ForEach(0..<21) { index in
            ""
        }

        Color.white
            .aspectRatio(2, contentMode: .fit)
    }
    .padding()
    .scrollTargetLayout()
}
.scrollPosition(id: $selectedColorIndex, anchor: .center)
.aspectRatio(1, contentMode: .fit)

```

I am also using visualEffect to make it look more like the default wheel picker in SwiftUI:

```

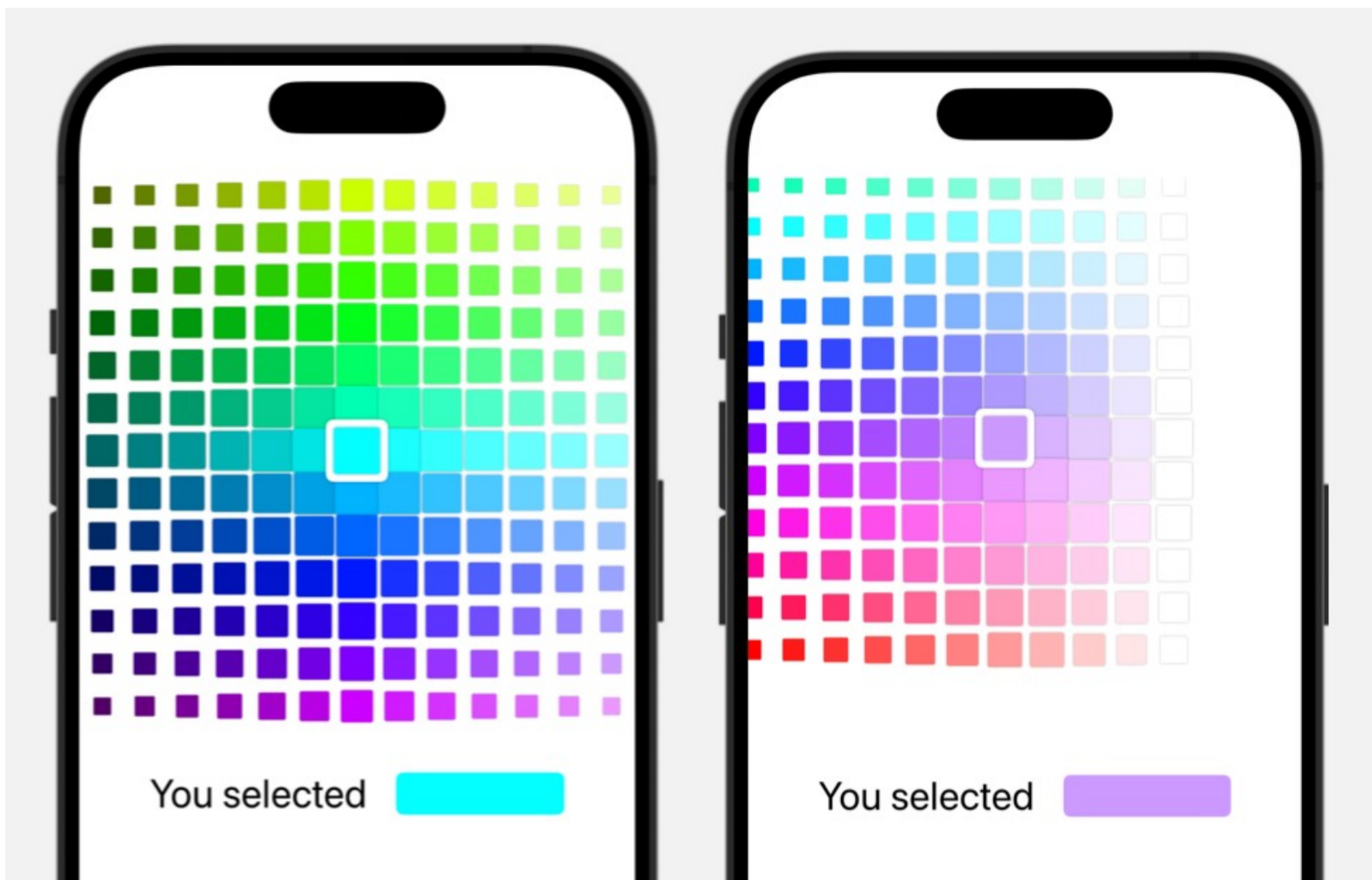
ForEach(0..<21) { index in
    ColorRow(hueValue: index)
        .frame(height: 20)
        .id(index)
        .overlay {
            if index == selectedColorIndex {
                Rectangle()
                    .stroke(Color.black, lineWidth: 3)
            }
        }
        .zIndex(index == selectedColorIndex ? 100 : 1)
        .visualEffect { content, proxy in
            content
                .opacity(opacity(for: proxy))
        }
}

func opacity(for proxy: GeometryProxy) -> Double {
    let scrollViewHeight = proxy.bounds(of: .scrollView)?.height ?? 100
    let rowCenterPosition = proxy.frame(in: .scrollView).midY
    let distanceFromScrollCenter = abs(scrollViewHeight / 2 - rowCenterPosition)

    return 1 - Double(distanceFromScrollCenter / scrollViewHeight) * 2
}

```

You can also extend this to a 2 dimension grid, where the user can scroll in vertical and horizontal direction. I am varying the hue in y-direction. In x-direction I am showing colors with different brightness and saturation:



The color model and sample data is:

```
struct CustomColor: Identifiable, Hashable {
    let hue: CGFloat
    let saturation: CGFloat
    let brightness: CGFloat
    let id = UUID()

    static func spectrum() -> [CustomColor] {
        let brightnessDistance = 0.1
        let hueDistance = 0.05
        let brightnessValues = Array(stride(from: 0.0,
                                           through: 1.0,
                                           by: brightnessDistance))

        let hueValues = Array(stride(from: 0.0,
                                     through: 1.0,
                                     by: hueDistance))

        var results = [CustomColor]()

        for hueValue in hueValues {
            let row = brightnessValues.map { CustomColor(hue: hueValue,
                                                         saturation: 1,
                                                         brightness: $0)}

            results.append(contentsOf: row)

            let rowEnd = brightnessValues.map { CustomColor(hue: hueValue,
                                                            saturation: $0,
```

```

1)}.reversed().dropFirst()
                                brightness:

        results.append(contentsOf: rowEnd)
    }
    return results
}

```

The custom color picker will use LazyVGrid and scroll position:

```

struct GridColorPickerView: View {

    @State private var selectedColor: CustomColor? = nil
    let colorOptions = CustomColor.spectrum()
    let colorSize: CGFloat = 30

    var body: some View {

        ScrollView([.horizontal, .vertical]) {
            LazyVGrid(columns: Array(repeating: GridItem(.fixed(colorSize),
                                                                    spacing: 0),
                                                                    count: 21),
                      spacing: 0) {

                ForEach(colorOptions) { color in
                    ZStack {
                        if color == selectedColor {
                            RoundedRectangle(cornerRadius: 5)
                                .fill(Color.white)
                                .padding(-5)
                        }

                        Color(hue: color.hue,
                              saturation: color.saturation,
                              brightness: color.brightness)
                            .cornerRadius(2)
                    }
                    .shadow(radius: 1)
                    .frame(height: colorSize)
                    .id(color)
                    .zIndex(color == selectedColor ? 1 : 0)
                    .visualEffect { content, proxy in
                        content
                            .scaleEffect(scaleAmount(for: proxy))
                    }
                }
            }
            .padding(170)
            .scrollTargetLayout()
        }
        .scrollPosition(id: $selectedColor, anchor: .center)
        .aspectRatio(1, contentMode: .fit)
        .padding(.top)
        .onAppear {
            selectedColor = colorOptions[colorOptions.count / 2]
        }
    }
}

func scaleAmount(for proxy: GeometryProxy) -> Double {
    let scrollViewHeight = proxy.bounds(of: .scrollView)?.height ?? 100

```

```
let rowCenterPositionY = proxy.frame(in: .scrollView).midY
let distanceFromScrollCenterY = abs(scrollViewHeight / 2 -
                                     rowCenterPositionY)

let scrollViewWidth = proxy.bounds(of: .scrollView)?.width ?? 100
let rowCenterPositionX = proxy.frame(in: .scrollView).midX
let distanceFromScrollCenterX = abs(scrollViewWidth / 2 - rowCenterPositionX)

return 1.1 - (Double(distanceFromScrollCenterX / scrollViewWidth) +
             Double(distanceFromScrollCenterY / scrollViewHeight)) * 0.75
}
}
```

10. ADAPTIVE LAYOUT

10.1 WHY YOU NEED ADAPTIVE LAYOUT

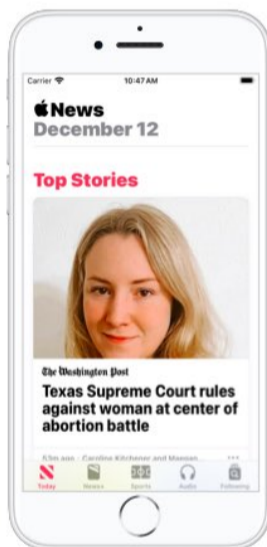
When developing apps with SwiftUI, it's crucial to consider how your views will appear across a variety of devices. From the compact iPhone SE to the expansive screens of the iPhone 15 Pro Max, and even cross-platform devices like the iPad Pro, WatchOS, and MacOS, each presents unique challenges due to their different screen sizes.

The Challenge of Diverse Devices

In recent years, the complexity of adaptive layout has increased. Apps are now packed with more features, screens, and additional functionality. Despite this complexity, we strive to maintain a clean and appealing design. This becomes even more challenging when you consider supporting landscape mode or dealing with split-screen views on iPads, where the available space can vary significantly.

Landscape Mode and Split Screen

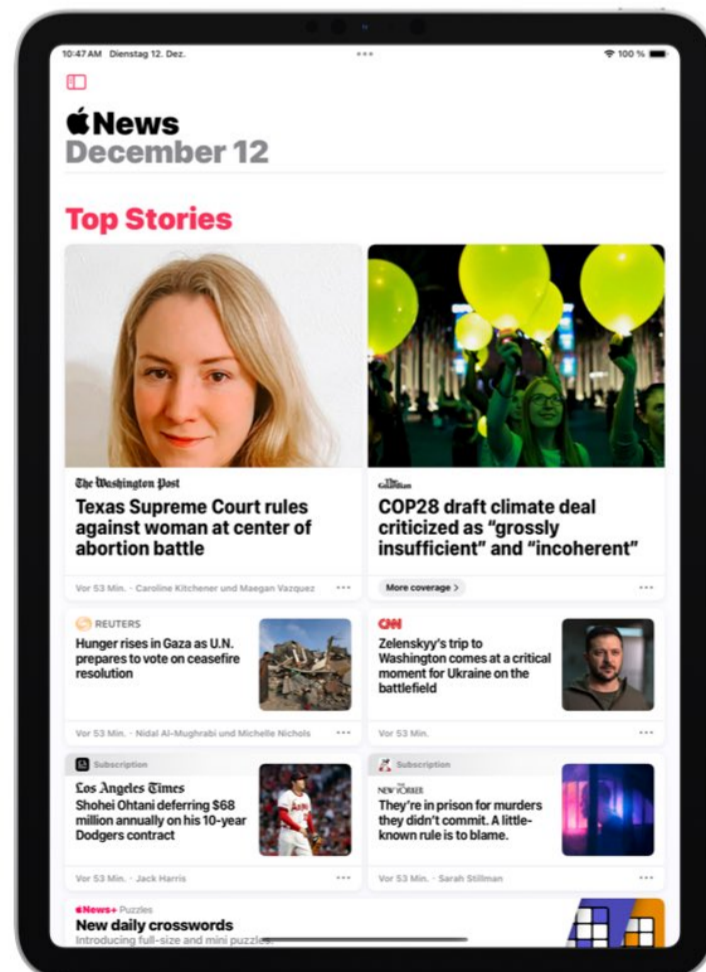
Some native Apple apps, like the News app, don't support landscape mode, as they're typically used in portrait orientation. Conversely, apps like YouTube benefit greatly from landscape support, enhancing the video-watching experience. The decision to support various orientations and split-screen modes will depend on the context of your app and the user's needs.



iPhone SE (2nd gen)



iPhone 15 Pro Max



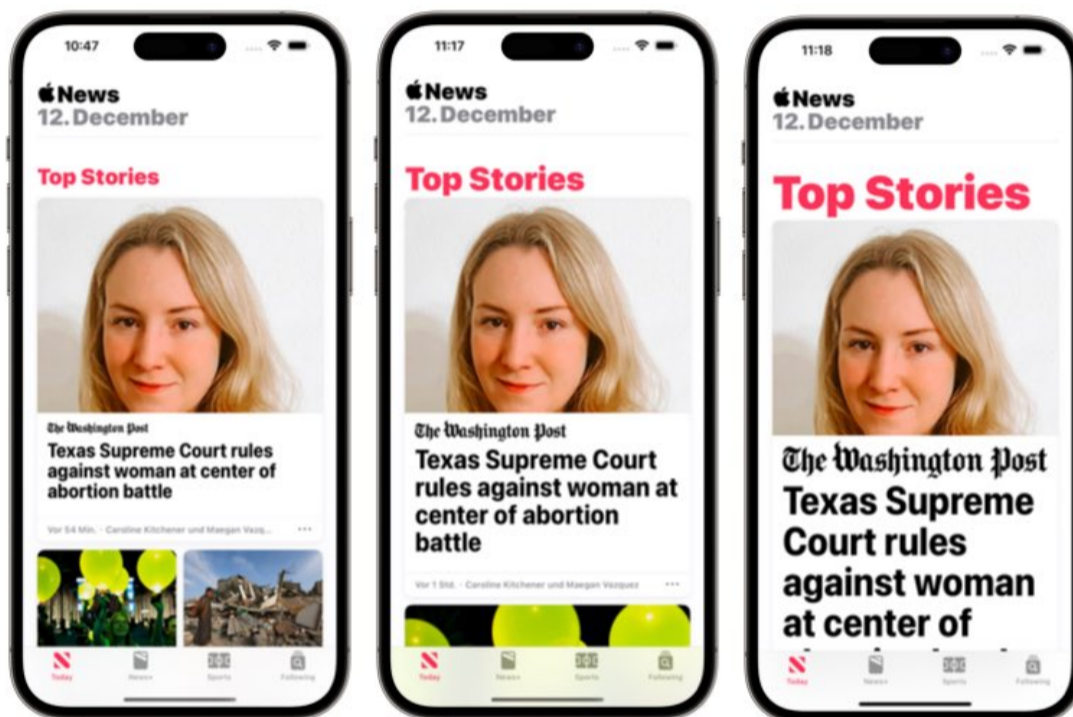
iPad Pro 11inch

Keyboard Considerations

Another aspect to consider is the keyboard. When it pops up, it can obscure content, especially on smaller devices like the iPhone. It's essential to ensure that text fields are not hidden behind the keyboard, allowing users to see what they're typing.

Dynamic Type and Accessibility

Dynamic type is another factor that can affect your layout. Users may adjust their preferred text size for better readability, and your app's design must adapt accordingly. Apple's News app, for example, handles this well by wrapping text to the next line and limiting the number of lines shown.



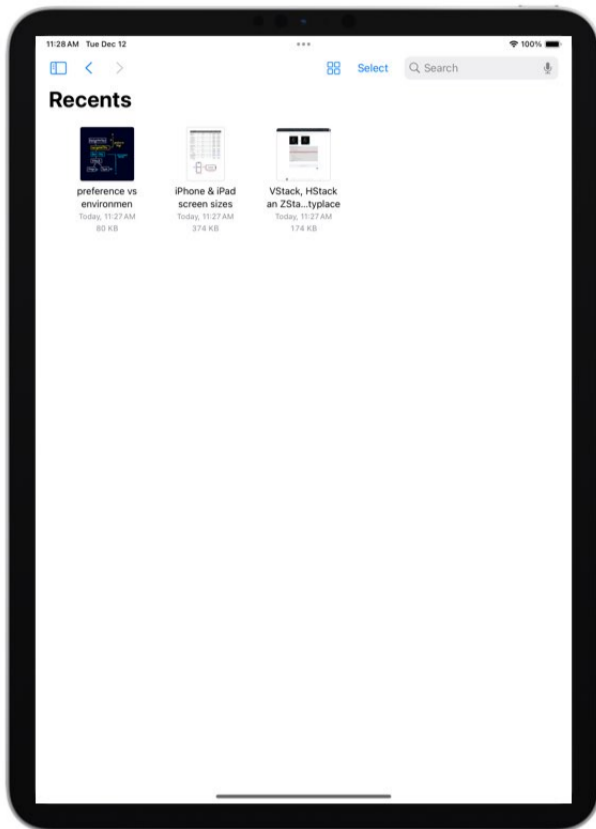
Design Strategies: Responsive vs. Adaptive

There are two main strategies for handling different screen sizes and scenarios: responsive design and adaptive design.

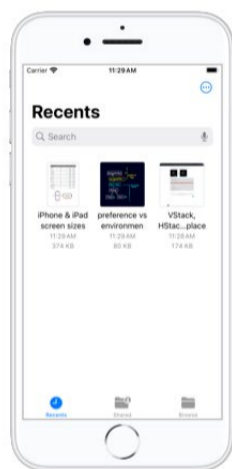
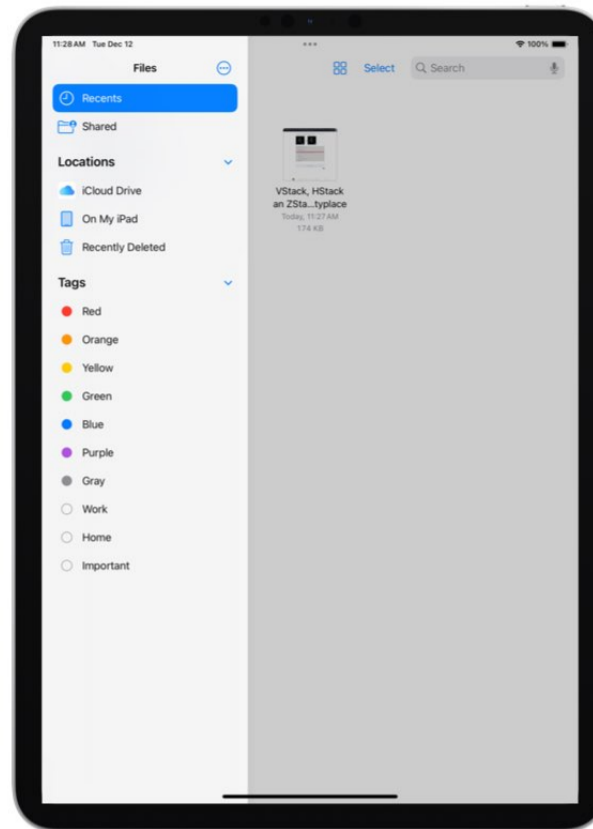
- **Responsive Design:** This approach involves minor modifications to the layout to accommodate different screen sizes. For instance, images may be resized while maintaining the same aspect ratio, and text may be reflowed to fit the available space. The News app screenshots above are a good example.
- **Adaptive Design:** This strategy involves more dramatic changes to the layout. Components may be repositioned or replaced entirely to create a layout that is better suited to the available space. For example, a VStack on a large screen might become a one-dimensional list on a smaller screen.

Navigation and Tab Views

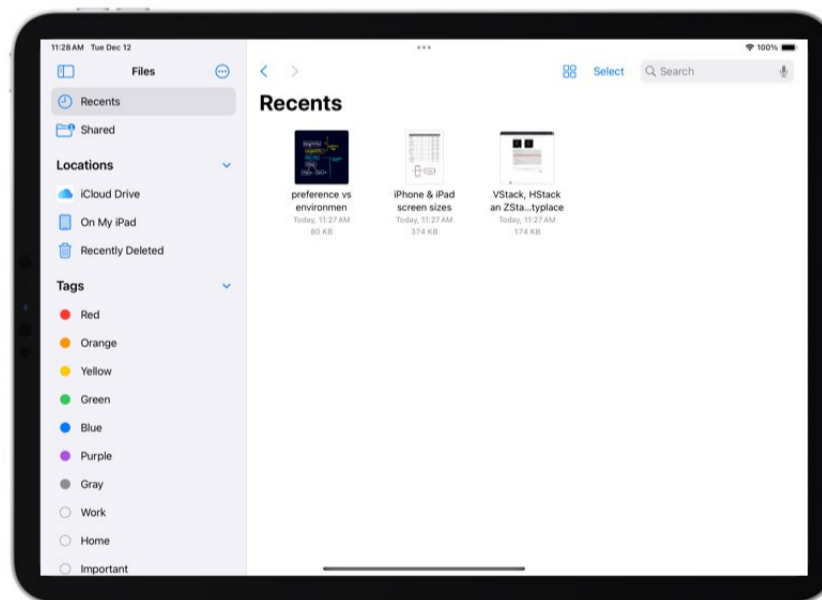
Apple’s approach to **adaptive design** can be seen in apps like Finder, where on an iPad in portrait mode a sidebar may slide in and out. On an iPhone, a NavigationSplitView might be presented as a NavigationStack, while on an iPad, it could appear as a sidebar in a NavigationSplitView.



iPad Pro 11inch



iPhone SE (2nd gen)

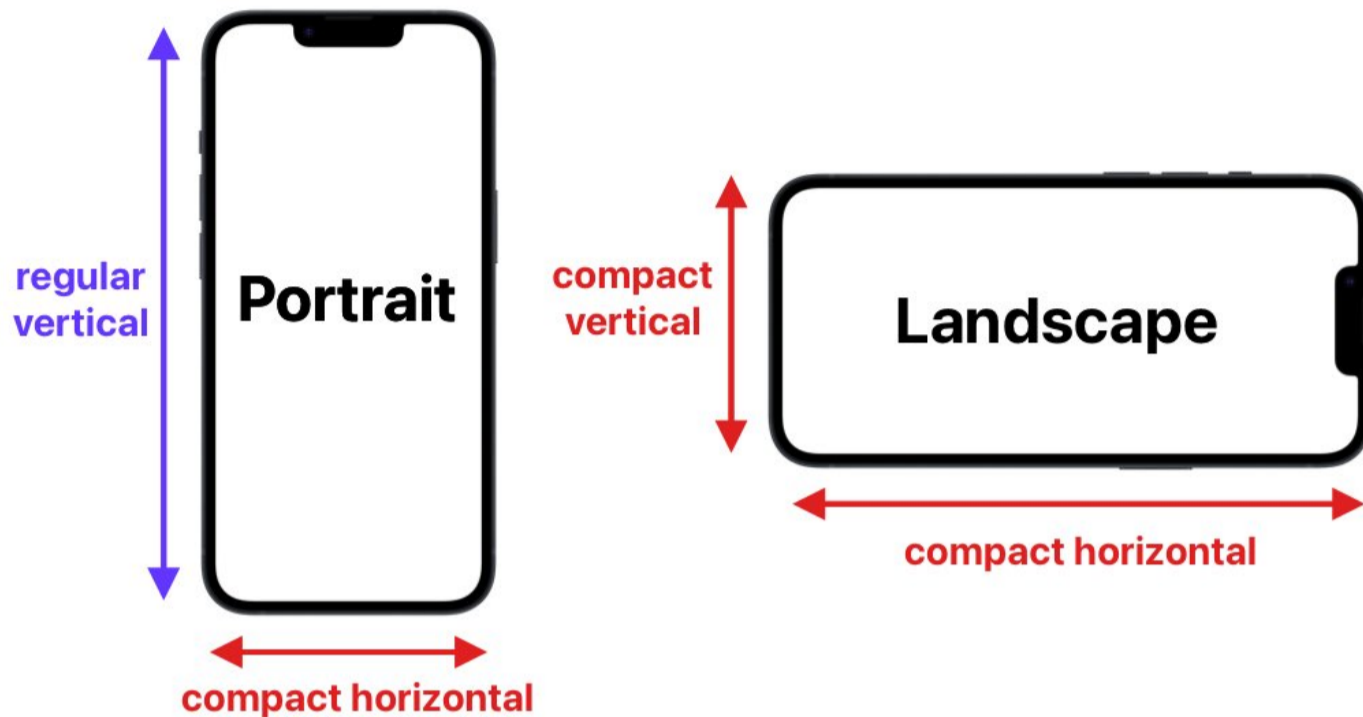


Throughout this section, I’ll review the principles of adaptive and responsive design, discuss the tools available in SwiftUI, and provide strategies for tackling the variations across different devices. Remember, the terms “adaptive” and “responsive” can sometimes be used interchangeably, and there’s a bit of a gray area between them. If you find my usage differs from your understanding, I ask for your understanding as these concepts can be nuanced.

In the upcoming sections, we’ll dive deeper into each of these topics, ensuring that you’re equipped to create SwiftUI layouts that look fantastic on any device.

10.2 WHAT IS THE AVAILABLE SPACE

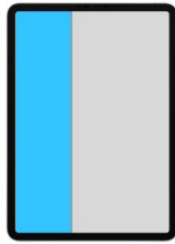
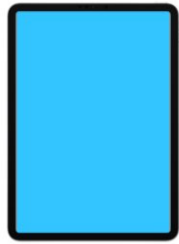
Let's take a look at the different device sizes we have. I have attached a table below that lists all the iPhones and their screen sizes, sorted by size in the vertical direction. The screen sizes are mentioned in inches, and the logical resolution is given in points.



iPhone Screen Sizes

The smallest iPhone that supports iOS 17 is the **iPhone SE** (2nd & 3rd) generation and the largest is the **iPhone 15 Pro Max**. When testing your designs, it is necessary to make sure that you consider the smallest and largest screen sizes. Thus, the layout also works for the other screen sizes.

iPhone	Screen Size (Inches)	Logical Resolution (Points)	Scale Factor	Supports up to	Portrait size classes	Landscape size classes
iPhone SE (1st generation)	4.0	320 x 568	2	iOS 16	H: Compact, V: Regular	H: Compact, V: Compact
iPhone 6/6s/7/8	4.7	375 x 667	2	iOS 16	H: Compact, V: Regular	H: Compact, V: Compact
iPhone SE (2nd & 3rd generation)	4.7	375 x 667	2	iOS 17	H: Compact, V: Regular	H: Compact, V: Compact
iPhone 6 Plus/6s Plus/7 Plus/8 Plus	5.5	414 x 736	3	iOS 16	H: Compact, V: Regular	H: Regular, V: Compact
iPhone 12 mini / iPhone 13 mini	5.4	375 x 812	3	iOS 17	H: Compact, V: Regular	H: Compact, V: Compact
iPhone X/XS/11 Pro	5.8	375 x 812	3	iOS 17	H: Compact, V: Regular	H: Compact, V: Compact
iPhone 12/12 Pro iPhone 13/13 Pro iPhone 14	6.1	390 x 844	3	iOS 17	H: Compact, V: Regular	H: Compact, V: Compact



H: **regular**
V: regular

H: **compact**
V: regular

H: **compact**
V: regular

iPad mini 5	width: 768 height: 1024	width: 320 height: 1024	width: 438 height: 1024
iPad 11inch	width: 834 height: 1194	width: 320 height: 1194	width: 504 height: 1194
iPad 12.9inch	width: 1024 height: 1366	width: 375 height: 1366	width: 639 height: 1366



H: **regular**
V: regular

H: **regular**
V: regular

H: **compact**
V: regular

H: **compact**
12.9: **regular**
V: regular

iPad mini 5	width: 1024 height: 768	width: 694 height: 768	width: 320 height: 768	width: 507 height: 768
iPad 11inch	width: 1194 height: 834	width: 809 height: 834	width: 375 height: 834	width: 592 height: 834
iPad 12.9inch	width: 981 height: 1024	width: 981 height: 1024	width: 375 height: 1024	width: 678 height: 1024

iPhone 14 Pro iPhone 15/15 Pro	6.1	393 x 852	3	iOS 17	H: Compact, V: Regular	H: Compact, V: Compact
iPhone XR/11	6.1	414 x 896	3	iOS 17	H: Compact, V: Regular	H: Regular , V: Compact
iPhone XS Max / 11 Pro Max	6.5	414 x 896	3	iOS 17	H: Compact, V: Regular	H: Regular , V: Compact
iPhone 12 Pro Max / iPhone 13 Pro Max / iPhone 14 Plus	6.7	428 x 926	3	iOS 17	H: Compact, V: Regular	H: Regular , V: Compact

iPhone 14 Pro Max / iPhone 15 Plus / 15 Pro Max	6.7	430 x 932	3	iOS 17	H: Compact, V: Regular	H: Regular, V: Compact
---	-----	-----------	---	--------	---------------------------	---------------------------

Size Classes

Size classes play a crucial role in determining the available space on a device. In the iPhone's portrait mode, the vertical size class is regular, indicating ample vertical space. However, the horizontal size class is compact, indicating limited horizontal space. These size classes are used by various UI elements, such as navigation split views, to determine their layout.

Considering iPad Sizes

In addition to iPhones, we also need to consider the different sizes of iPads. The smallest iPad is the iPad Mini, with a screen size of 7.9 inches, while the largest is the iPad Pro 12.9 inches.

iPad	Screen Size (Inches)	Logical Resolution (Points)	Scale Factor
iPad Mini 4	7.9	768 x 1024	2
iPad mini (6th generation)	8.3	744 x 1133	2
iPad Pro (9.7-inch)	9.7	768 x 1024	2
iPad Air 2	10.9	768 x 1024	2
iPad Pro 10.5-inch	10.5	834 x 1112	2
iPad (9th generation)	10.2	810 x 1080	2
iPad Pro (12.9-inch)	12.9	1024 x 1366	2

Below you can find information about the size classes for different configurations, such as fullscreen and split screen. These are used e.g. for NavigationSplitView configuration:

Determining the Device in SwiftUI

In SwiftUI, you can access device information using the `UIDevice` and `UIScreen` classes. By using `UIDevice.current`, you can determine the type of device your app is running on, such as iPhone, iPad, TV, Car, or Mac. This information can be useful in making layout decisions specific to each device.

```
struct DeviceInformationView: View {  
    let device = UIDevice.current.userInterfaceIdiom  
  
    var body: some View {  
        VStack {  
            switch device {  
            case .unspecified:  
                Text("unspecified")  
            case .phone:  
                Text("Running on iPhone")  
            case .pad:  
                Text("Running on iPad")  
            case .tv:  
                Text("Running on TV")  
            case .carPlay:  
                Text("Running on car")  
            case .mac:  
                Text("Running on mac")  
            case .vision:  
                Text("Running on Vision")  
            @unknown default:  
                Text("unknown")  
            }  
        }  
    }  
}
```

Understanding the Available Space

To determine the available space on a device, you can use the `UIScreen.main.bounds` property.

```
let screen = UIScreen.main.bounds
```

However, it is important to note that this property represents the screen size, not the app size. It may not update correctly when the device is rotated or when supporting landscape or split-screen modes. Instead, it is recommended to use the `GeometryReader` view to obtain accurate information about the available space.

```
struct ContentView: View {  
    var body: some View {  
        GeometryReader { geometry in  
            Text("GeometryReader: \(geometry.size.width) - \(geometry.size.height)")  
        }  
    }  
}
```

The GeometryReader view in SwiftUI provides a flexible approach to creating adaptive layouts. By using this view, you can access the safe area and obtain the updated dimensions of the available space. This approach is more reliable than relying solely on the UIScreen properties.

Understanding the available space on different devices is crucial for creating adaptive layouts in SwiftUI. By considering the device sizes, size classes, and using tools like GeometryReader, you can design apps that adapt seamlessly to various screen sizes and orientations.

10.3 INTERFACE SIZE CLASSES

In this lesson, we will explore interface size classes and how they are used in SwiftUI to adapt the layout based on device types and sizes. Apple defines internal size classes, namely horizontal and vertical size classes, to determine the layout adaptation. In UIKit, you can access these size classes from the trait collection. In SwiftUI, you can access them through the environment using the @Environment property wrapper.

Understanding Size Classes

To access the size classes, use the @Environment property wrapper and specify the desired property: \.horizontalSizeClass for the horizontal size class and \.verticalSizeClass for the vertical size class. These properties are enums with two cases: regular and compact. You can check the size classes and perform different actions based on their values.

```
struct SizeClassExampleView: View {  
  
    @Environment(\.horizontalSizeClass) var horizontalSizeClass  
    @Environment(\.verticalSizeClass) var verticalSizeClass  
  
    var body: some View {  
        if verticalSizeClass == .regular {  
            Text("iPhone in portrait mode")  
        } else {  
            Text("iPhone is in landscape mode")  
        }  
    }  
}
```

For example, in portrait mode on an iPhone, the horizontal size class is compact, while the vertical size class is regular. When the device is rotated to landscape mode, both size classes become compact. By checking the vertical size class, you can determine if the device is in landscape mode.

Adapting Layout with Size Classes

You can use size classes to adapt your layout dynamically. For example, you can switch between different stack layouts based on the size classes. By using the @Environment property wrapper and the compact size class, you can conditionally display different stack layouts for compact and regular size classes.

Here is a reusable layout component that adaptively chooses the layout container:

```
struct SizeClassStack<Content: View>: View {
    let content: () -> Content
    @Environment(\.verticalSizeClass) var verticalSizeClass

    init(@ViewBuilder content: @escaping () -> Content) {
        self.content = content
    }

    var body: some View {
        switch verticalSizeClass {
        case .compact:
            HStack(alignment: .center,
                  spacing: 10,
                  content: content)
        case .regular:
            VStack(alignment: .center,
                  spacing: 10,
                  content: content)

        case .none:
            EmptyView()
        case .some(_):
            EmptyView()
        }
    }
}
```

Advanced Example: Navigation Split View

To demonstrate how Apple internally handles adaptive layouts, we will create a navigation split view example. This example showcases how the layout adapts based on the size classes. By using the `NavigationView` or `SplitView`, we can create a sidebar and detail view layout. The layout automatically adjusts based on the size classes, providing a consistent user experience across different devices.

```
struct NavigationSplitViewExample: View {
    let inspirations = NatureInspiration.examples()
    @State private var selectedInspiration: NatureInspiration? = nil

    var body: some View {
        NavigationSplitView {
            List(inspirations,
                 selection: $selectedInspiration) { inspiration in
                InspirationRow(inspiration: inspiration)
                    .tag(inspiration)
            }

            .navigationTitle("Inspirations")
        } detail: {
            if let selectedInspiration {
                SizeInspirationDetailView(inspiration: selectedInspiration)
            } else {
                ContentUnavailableView("Please select an inspiration",
                                         systemImage: "photo")
            }
        }
    }
}
```

```

}

fileprivate struct SizeInspirationDetailView: View {

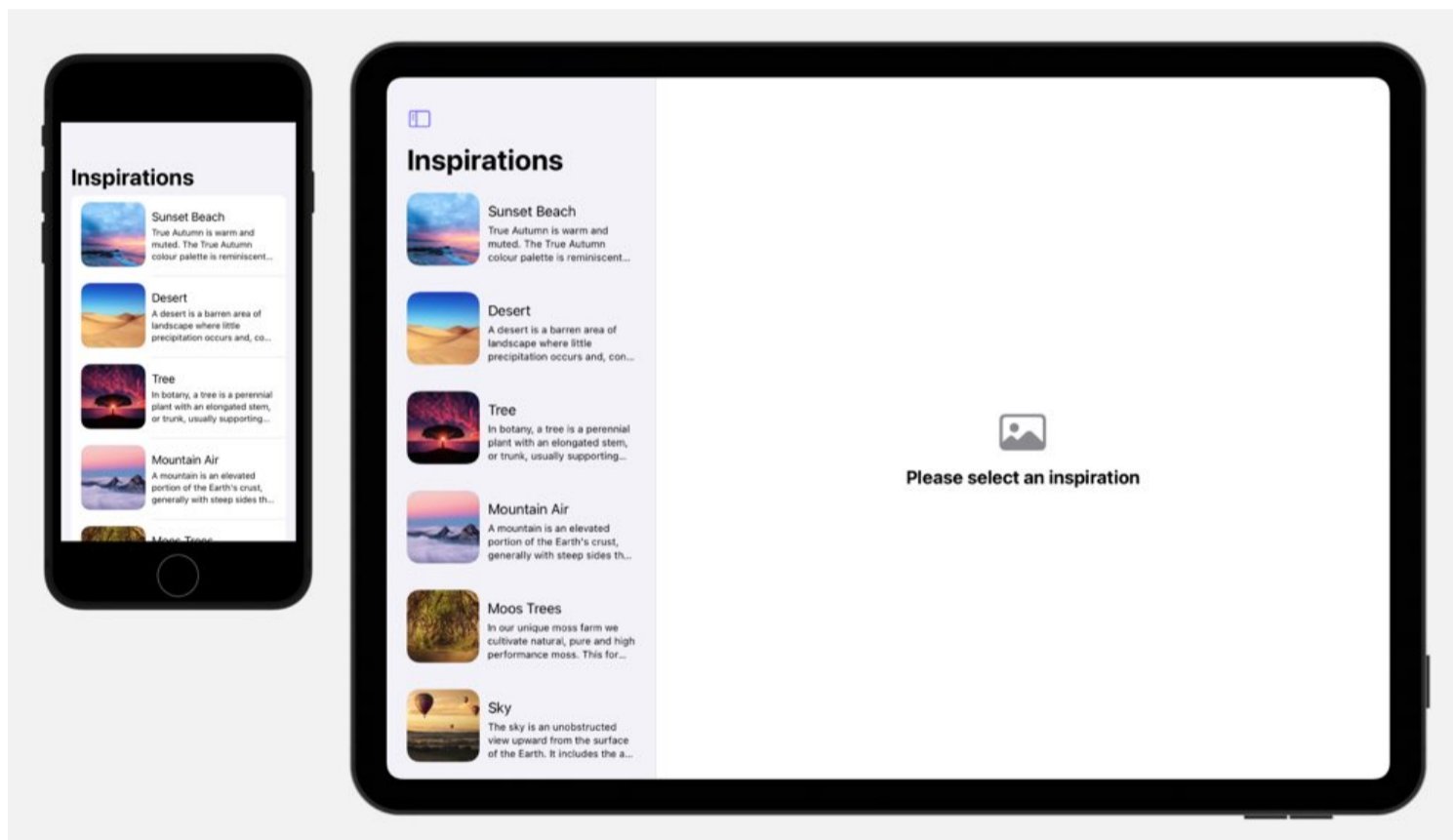
    let inspiration: NatureInspiration

    var body: some View {
        SizeClassStack {
            ImageAspectRatio(imageName: inspiration.imageName,
                             frameAspectRatio: 1.5)

            VStack(alignment: .leading, spacing: 5) {
                Text(inspection.name)
                    .font(.title)
                Text(inspection.description)
            }
                .padding(12)
        }
        .edgesIgnoringSafeArea([.bottom, .leading])
    }
}

```

On an iPhone in portrait mode, the vertical size class is regular and the horizontal size class is compact. Thus the NavigationView collapses into a stack navigation. Whereas on the iPad in landscape mode, the size classes are regular and the NavigationView uses a split navigation style:



Understanding and utilizing interface size classes in SwiftUI allows you to create adaptive layouts that seamlessly adjust to different device types and sizes. By leveraging size classes, you can use the same technique that Apple internally uses for NavigationView/NavigationSplitView and use the same cases. This helps to have a rough layout adjustment into distinguishable categories. A more fine-tuned adjustment can be done by e.g. resizing images and using multiline text views.

In the next section, we will explore another powerful feature in iOS 16 called ViewThatFits,” which offers an alternative approach to size classes.

10.4 ENVIRONMENT VALUES

In SwiftUI, the environment provides valuable information about the state of your app. You've already seen how to access information about size classes, but there's more to it. You can also retrieve information about color schemes, fonts, accessibility, and more. This is incredibly useful for creating adaptive layouts.

In the following example, I'm using the `@Environment` property wrapper to access the `colorScheme` environment property. This property tells us whether the app is currently in light or dark mode. Depending on the color scheme, we can adjust certain visual elements in our view. For instance, if we're in light mode, we might want to add a rounded rectangle border to the Text view. We can achieve this by using the background modifier and conditionally applying the border based on the color scheme.

```
struct EnvironmentListView: View {  
  
    @Environment(\.colorScheme) var colorScheme  
  
    var body: some View {  
        Text("Hello, World!")  
            .padding()  
            .background {  
                if colorScheme == .light {  
                    RoundedRectangle(cornerRadius: 5)  
                        .stroke(Color.cyan, lineWidth: 1.0)  
                }  
            }  
    }  
}
```

By using the environment value, we can dynamically adapt our layout based on the color scheme. This is just one example of how environment values can be utilized.

Here is a list of other environment properties:

- **layoutDirection**: Determines the layout direction, such as left-to-right or right-to-left.
- **accessibilityEnabled**: Indicates whether accessibility features are enabled.
- **calendar**: Gives access to the current calendar.
- **controlSize**: Allows you to override the size of control views, like buttons.
- **defaultMinimumListHeaderHeight** and **defaultMinimumListRowHeight**: Control the default heights for list headers and rows.
- **dismiss**: Used to dismiss a sheet or popover.
- **openWindow** and **dismissWindow**: Used to open and dismiss windows.
- **openURL**: Opens a URL in the default browser or another app.
- **undoManager**: Provides access to the default undo manager.

- **managedObjectContext**: Gives access to the managed object context in Core Data.
- **font**: Specifies the default font for views.
- **locale**: Provides information about the current locale.
- **imageScale**: Gives information about the scale of images.
- **dynamicType**: Allows you to adapt to dynamic type sizes.

These environment properties allow you to customize various aspects of your app's layout and behavior. By leveraging them, you can create adaptive and dynamic interfaces that respond to changes in the environment.

10.5 ENVIRONMENT VS PREFERENCEKEYS

In SwiftUI, understanding how data flows through your app's view hierarchy is crucial for creating adaptive and responsive layouts. Two key concepts in this data flow are the environment and preference keys. Let's dive into how these work internally.

NavigationView Example

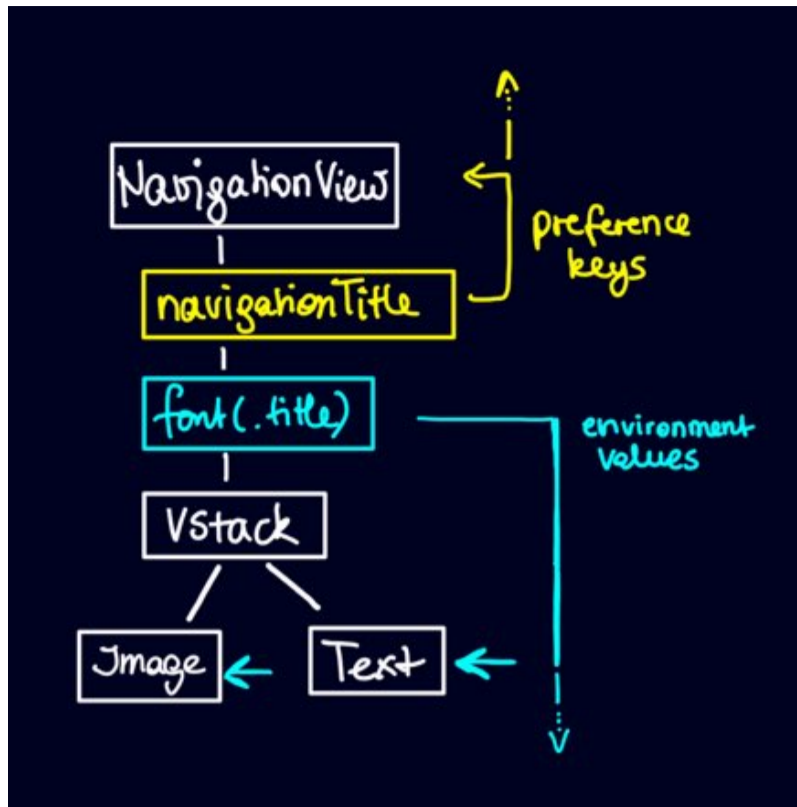
Let's consider the following example with NavigationView:

```
struct EnvironmentPreferenceExampleView: View {
    var body: some View {
        NavigationView {
            VStack {
                Image(systemName: "globe")
                Text("Hello World")
            }
            .font(.title)
            .navigationTitle("Title")
        }
    }
}
```

This title is passed using preference keys within a navigation view. You can attach a navigation title modifier to any child view within the navigation view, and the navigation view will use the innermost title it finds.

View Hierarchy and Modifiers

In the view hierarchy, view modifiers wrap around their respective views. For instance, a font modifier wraps around a VStack, and a navigation title modifier wraps around the font, which in turn wraps around the VStack. This nesting of views is crucial to understand how preference keys travel up the hierarchy.



Preference Keys

Preference keys travel upwards. When you add a navigation title modifier to a view, it changes a preference and overrides the navigation title, allowing the navigation view to display it. This upward movement means you need to attach the navigation title inside the view that requires this property.

Environment Values

On the other hand, environment values propagate downwards. When you attach a font modifier to a VStack, it internally changes the environment value for font. Instead of the font modifier you can also write:

```

VStack {
  Image(systemName: "globe")
  Text("Hello World")
}
.environment(\.font, .title)
  
```

The font view modifier is a convenient and easier-to-read way in comparison to using the environment.

Environment vs Preference Keys

In summary, preference keys allow child views to communicate with parent views by passing values up the view hierarchy. Environment values, conversely, enable parent views to pass down values to their children. This distinction has significant implications on where you attach modifiers that use either preference keys or environment values.

Creating Custom Environment Values

You can also define custom environment values. As an example, I will create a custom container that uses the environment to pass the container title.

To define a custom environment property, you extend the `EnvironmentValues` struct with a new property. For instance, if you want to pass a container title, you create a computed property within the extension and define a default value.

```
extension EnvironmentValues {  
  
    struct ContainerTitleKey: EnvironmentKey {  
        static var defaultValue: String? = nil  
    }  
  
    var containerTitle: String? {  
        get { self[ContainerTitleKey.self] }  
        set { self[ContainerTitleKey.self] = newValue }  
    }  
}
```

You need to create a type that conforms to `EnvironmentKey` and specify a default value. This type is used as an identifier in the environment dictionary, allowing you to access and set the value.

Once you've added your custom environment property, you can use it within your views:

```
struct EnvironmentContainerView<Content> : View where Content : View {  
  
    @Environment(\.containerTitle) var title  
  
    let content: () -> Content  
  
    init(@ViewBuilder content: @escaping () -> Content) {  
        self.content = content  
    }  
  
    var body: some View {  
        ZStack {  
            RoundedRectangle(cornerRadius: 25.0)  
                .fill(Color.cyan.gradient)  
  
            VStack(alignment: .leading,  
                spacing: 10) {  
  
                if let title {  
                    Text(title)  
                        .font(.title)  
                        .bold()  
                }  
  
                content()  
  
            }  
                .padding()  
        }  
        .aspectRatio(1, contentMode: .fit)  
    }  
}
```

You can either set it directly using the environment modifier or create a custom view modifier for a cleaner syntax:

```
extension View {
    func containerTitle(_ title: String) -> some View {
        self
            .environment(\.containerTitle, title)
    }
}
```

The following code example shows how to use the custom container:

```
VStack {
    EnvironmentContainerView {
        Text("this is inside")
    }
}
.containerTitle("Title")
.padding()
```

Advanced Use Case: Color Scheme

A practical example of using environment values is handling color schemes. If you have a settings view where users can switch between light and dark modes, you might encounter a situation where you need to **propagate a value both up and down the view hierarchy**.

If you use the environment for colorScheme, only the views inside see the change. But for a settings view, the changes need to be passed to the whole app:

```
struct SettingsView: View {
    @State private var selectedColorScheme: ColorScheme = .light

    var body: some View {
        VStack {
            Text("Color Scheme")

            Button(action: {
                selectedColorScheme = .light
            }, label: {
                if colorScheme == .light {
                    Image(systemName: "checkmark")
                }
                Text("light")
            })

            Button(action: {
                selectedColorScheme = .dark
            }, label: {
                if colorScheme == .dark {
                    Image(systemName: "checkmark")
                }
                Text("dark")
            })
        }
    }
}
```

```

        }
        Text("dark")
    })

    Button("system") {
    }
}
.environment(\.colorScheme, selectedColorScheme)
}
}

```

SwiftUI provides a **preferredColorScheme** modifier that uses preference keys to **propagate the selected color scheme upwards**, affecting the entire app. This is a convenient way to handle settings that should apply globally.

```

struct SettingsView: View {
    @State private var selectedColorScheme: ColorScheme = .light

    var body: some View {
        VStack {
            ""
        }
        .preferredColorScheme(selectedColorScheme)
    }
}

```

PreferredColorScheme uses reference keys to pass the new value up to the main app. From there SwiftUI sets the new value for the color scheme in the environment, which will set the new color scheme for the whole app.

By now, you should have a deeper understanding of how preference keys and environment values work in SwiftUI. This system allows you to write concise and reusable code, as you only need to set values once and let them propagate as needed. While property wrappers like `@State`, `@Binding`, and view models are commonly used for data flow, don't overlook the power of environment values for managing **global settings and styles** throughout your app.

10.6 DYNAMIC TYPE SIZE

In this section, we will explore how to utilize the `dynamicTypeSize` environment value in SwiftUI to adapt your layout based on the user's accessibility settings for text font size.

The `sizeCategory` property is replacing `dynamicTypeSize` in iOS 15. `dynamicTypeSize` is an enum that reflects the user's accessibility settings.

To explore the available values in `dynamicTypeSize`, let's add a `VStack` and examine them using a `switch` statement. Here's an example:

```

struct DynamicTypeSizeView: View {

    @Environment(\.dynamicTypeSize) var typeSize
    @Environment(\.sizeCategory) var sizeCategory // soft deprecated

    var body: some View {

        switch typeSize {

            case .xSmall:
                Text("xSmall Type Size")
            case .small:
                Text("small Type Size")
            case .medium:
                Text("medium Type Size")
            case .large:
                Text("Large Type Size")
            case .xLarge:
                Text("xLarge Type Size")
            case .xxLarge:
                Text("xxLarge Type Size")
            case .xxxLarge:
                Text("xxxLarge Type Size")

            case .accessibility1:
                Text("accessibility1 Type Size")
            case .accessibility2:
                Text("accessibility2 Type Size")
            case .accessibility3:
                Text("accessibility3 Type Size")
            case .accessibility4:
                Text("accessibility4 Type Size")
            case .accessibility5:
                Text("accessibility5 Type Size")
            @unknown default:
                Text("unknown Type Size")
        }
    }
}

```

There are five accessibility settings and seven regular sizes, ranging from extra small to medium, large, and extra extra large (XXL).

Handling Accessibility Sizes

Dynamic type sizes can be categorized into regular and accessibility sizes. Accessibility sizes are associated with specific accessibility settings. `dynamicTypeSize` has a boolean property that tells you if accessibility is enabled:

```
typeSize.isAccessibilitySize
```

You could use this information to hide or show certain text elements based on accessibility settings.

Adapting Fonts with Dynamic Type Size

When using the `dynamicTypeSize` environment property, the font is automatically adjusted as long as you use one of the system fonts, such as the title font. However, if you want to specify a fixed size, it will prevent dynamic type adaptation:

```
Text("Hello, World!")  
    .font(.system(size: 14))
```

To allow scaling with dynamic type, use one of the system fonts like headline.

```
Text("Hello, World!")  
    .font(.headline)
```

Limiting Dynamic Type Sizes

To limit the scaling of text sizes, SwiftUI provides the `limitedDynamicTypeSize` modifier. This modifier takes a range of dynamic type sizes, allowing you to set minimum and maximum limits. By specifying a range, you can ensure that the text size stays within a certain range, even when accessibility settings change.

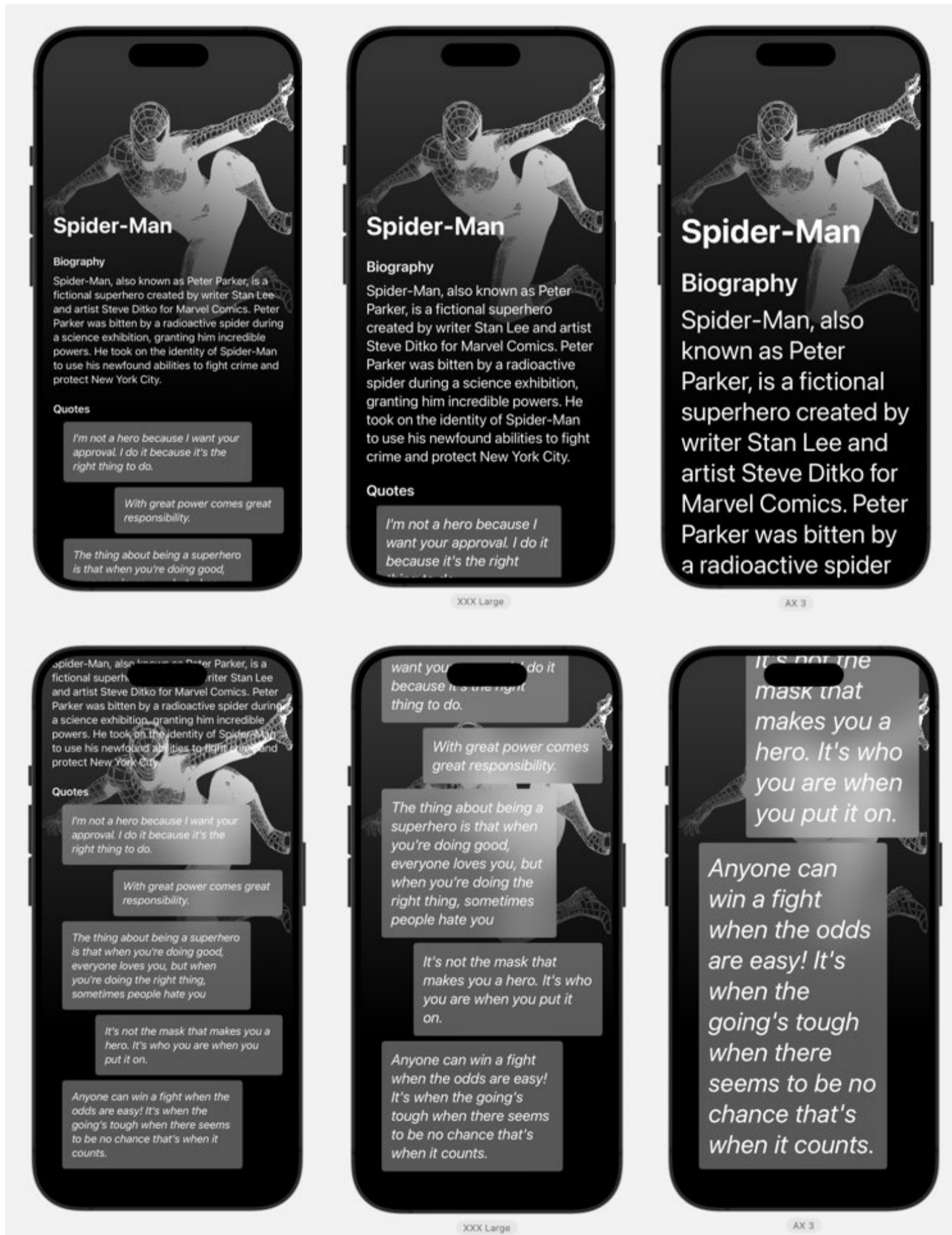
```
Text("this is using a fixed size")  
    .dynamicTypeSize(.xLarge)  
  
Text("Limited dynamic range")  
    // .dynamicTypeSize(...DynamicTypeSize.accessibility1) // set a maximum value  
    // .dynamicTypeSize(DynamicTypeSize.medium...) // set a minimum value  
    .dynamicTypeSize(DynamicTypeSize.medium...DynamicTypeSize.accessibility1)
```

It's worth mentioning that you should use these limitations judiciously. Apple intended for dynamic type size to adapt to the user's accessibility settings, so it's not recommended to fix your layouts by always neglecting higher accessibility settings.

You can limit larger elements like titles and sub-headlines because they are already large enough. However, smaller elements with e.g. caption font should scale according to accessibility settings.

Practical Example: SuperHeroDetailView

The superhero view from section 5 Sizing views has a complex layout, where I use an alternating alignment for the quotes list. If you check this layout for different dynamic font settings, you can see that for the accessibility settings, the layout does not look good. The text is so large that it stretches over many lines.



For the accessibility settings, I would prefer to not use the alternating alignment and remove some of the padding:



I want to adjust the layout based on dynamic type sizes. By conditionally applying view modifiers, we can remove unnecessary padding and alignment adjustments for larger text sizes, improving the overall layout.

```
struct MarvelView: View {
    let superHero: SuperHero
    @Environment(\.dynamicTypeSize) var dynamicTypeSize
    var body: some View {
        ZStack(alignment: .bottom) {
            // Image

            ScrollView {
                VStack(alignment: .leading, spacing: 10) {
                    // Texts

                    VStack(alignment: .leading) {
                        ForEach(Array(superHero.quotes.enumerated()), id: \.offset) {
                            (index, quote) in
                            Text(quote)
                                .applyIf(!dynamicTypeSize.isAccessibilitySize) {
                                    $0.padding(index.isOdd ? .leading : .trailing,
                                                50)
                                    .frame(maxWidth: .infinity,
                                           alignment:
                                               index.isOdd ? .trailing : .leading)
                                }
                        }
                    }
                }
            }
            .applyIf(!dynamicTypeSize.isAccessibilitySize) {
                $0.padding(.leading)
            }
        }
    }
}
```

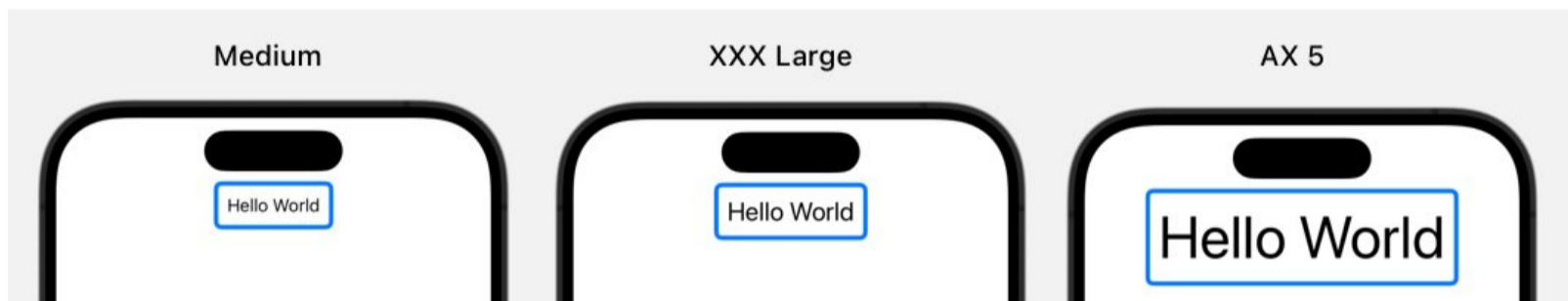

10.7 SCALED METRIC

When designing layouts that accommodate various font sizes, especially when supporting Dynamic Type, it's crucial to ensure that elements within your UI scale appropriately. You might have encountered situations where increasing the font size leads to a cramped interface because the padding and spacing remain constant. To address this, SwiftUI provides a powerful tool called `ScaledMetric`.

`ScaledMetric` is a property wrapper that automatically adjusts sizes based on the user's Dynamic Type settings. This means that as the text size changes, so can the other dimensions in your layout, such as padding, spacing, and even custom shapes.

Imagine you have a simple "Hello World" text view, and you want to add padding and a border with rounded corners. Here's how you might start:

```
Text("Hello World")
    .padding(10)
    .background(
        RoundedRectangle(cornerRadius: 5)
            .stroke(Color.blue, lineWidth: 4)
    )
```



When you test this with different font sizes, you'll notice that the padding and border don't scale with the text. The padding remains at a fixed size, which can make the text look too close to the border at larger sizes.

Using `ScaledMetric`

To ensure that the padding scales with the font size, you can use `ScaledMetric` which is a property wrapper:



```

struct ScaledMetricExampleView: View {
    @ScaledMetric var padding: CGFloat = 10
    @ScaledMetric var cornerRadius: CGFloat = 5
    @ScaledMetric var lineWidth: CGFloat = 4

    var body: some View {
        Text("Hello, World!")
            .padding(padding)
            .background {
                RoundedRectangle(cornerRadius: cornerRadius)
                    .stroke(Color.blue, lineWidth: lineWidth)
            }
    }
}

```

Now, when you change the font size, the padding adjusts accordingly, maintaining a balanced look. You can apply `ScaledMetric` to other properties as well, such as the corner radius and line width of your border.

`ScaledMetric` is an incredibly convenient tool for creating adaptive layouts in SwiftUI. It simplifies the process of ensuring that your UI elements scale correctly with text size, which is essential for accessibility. By extracting sizes into separate properties and making them `@ScaledMetric`, you can make your layout look good across all accessibility sizes with minimal effort

Custom Scaling with Dynamic Type Size

If you need more control over the scaling, you can use the `DynamicTypeSize` enumeration to manually adjust sizes. This approach allows you to fine-tune the scaling behavior for different Dynamic Type categories.

```

struct ScaledMetricExampleView: View {
    @Environment(\.dynamicTypeSize) var typeSize

    var customSize: CGFloat {
        switch typeSize {
            case .xSmall: return 3
            case .small: return 4
            case .medium: return 5
            case .large: return 6
            case .xLarge, .xxLarge, .xxxLarge: return 7
            case .accessibility1: return 8
            case .accessibility2: return 9
            case .accessibility3: return 10
            case .accessibility4: return 11
            case .accessibility5: return 12
            @unknown default:
                return 7
        }
    }

    var body: some View {
        Text("Hello, World!")
            .padding(customSize)
    }
}

```

10.8 CONDITIONAL VIEW MODIFIERS

Now that you have all the necessary information, such as color scheme and size categories, you can apply different layouts and styles based on these conditions. One approach to achieve this is by using conditional view modifiers. However, it's not as straightforward as using an if statement.

In this section, I'll show you how to conditionally apply changes using a practical example. Let's start by creating a boolean state variable called `hasBorder` and initializing it to `true`. We'll use this property to add a border to our `HelloWorldText` view. To toggle the border on and off, we'll add a toggle switch that binds to the `hasBorder` property.

```
struct ConditionalViewModifierExampleView: View {
    @State private var hasBorder: Bool = true

    var body: some View {
        VStack(spacing: 20) {
            Text("Hello, World!")
                .padding()
                .border(hasBorder ? Color.indigo : Color.clear)

            Toggle(hasBorder ? "show border" : "no border",
                isOn: $hasBorder.animation(.easeInOut(duration: 2)))
                .fixedSize()
        }
    }
}
```

By using a **ternary operator**, we can conditionally set the border color to indigo when `hasBorder` is true, and to clear when it's false. Toggling the switch will make the border appear and disappear accordingly.

Now, let's apply the same conditional logic to the padding modifier. Currently, we haven't specified any value for padding, so it uses the system default padding that scales nicely with the type sizes. However, if we want to toggle the padding, we need to specify a value, such as 0.

```
.padding(hasBorder ? 10 : 0)
```

In this case, since we want to preserve the system padding when `hasBorder` is false, it would be better to hide and show the entire padding modifier. To achieve this, we can define a custom view modifier that takes a condition and applies the modifiers accordingly.

```
extension View {
    @ViewBuilder
    func applyIf<M: View>(_ condition: Bool, transform: (Self) -> M) -> some View {
        if condition {
            transform(self)
        } else {
            self
        }
    }
}
```

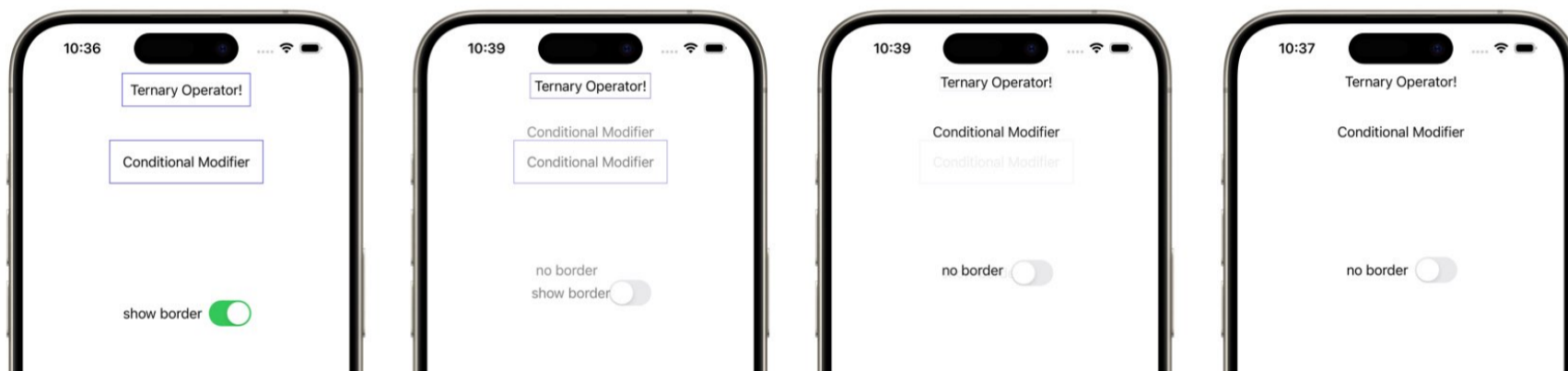
Now, instead of using the ternary operators, we can use the **applyIf** modifier to conditionally apply the border and padding modifiers.

```
Text("Hello, World!")
    .applyIf(hasBorder, transform: { view in
        view
            .padding()
            .border(Color.indigo)
    })
```

This approach allows us to handle the condition within the view modifier itself, making the code more readable and maintainable.

Problems with Conditional View Modifiers

However, it's important to note that conditional view modifiers have some limitations. For example, when it comes to animations, SwiftUI doesn't know how to animate between two different views. It defaults to a fade-in and fade-out animation. If you need to animate the changes, it's better to use ternary operators instead.



Additionally, if the views involved have their own states, such as `@State` or `@StateObject` properties, using conditional view modifiers may lead to the loss of state. Therefore, it's crucial to be cautious when using if statements and consider the impact on user data.

For example, using this subview that has a state property:

```
struct StateExampleView: View {
    @State private var text: String = ""
    var body: some View {
        TextField("Type something", text: $text)
    }
}
```

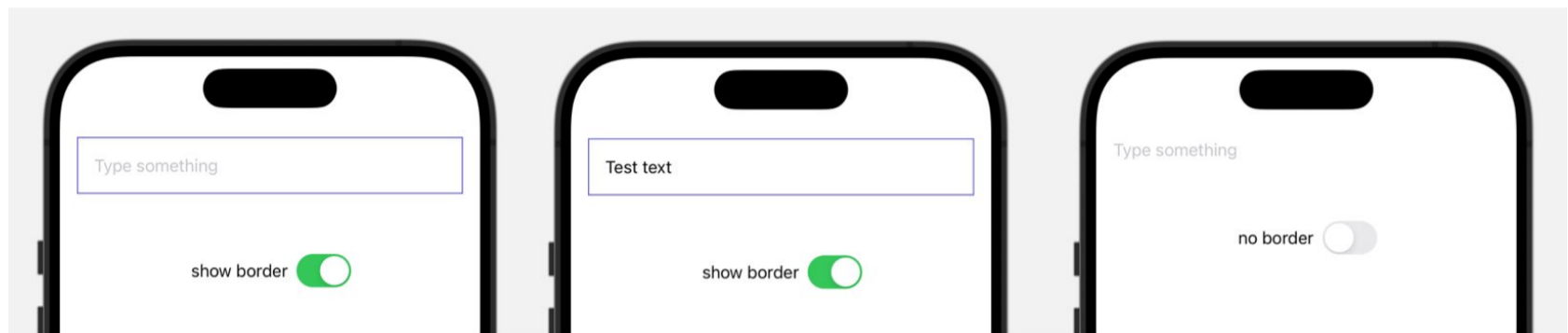
together with a conditional view modifier:

```

StateExampleView()
    .applyIf(hasBorder, transform: { view in
        view
            .padding()
            .border(Color.indigo)
    })
}

```

In the following screenshots, you see the initial empty TextField. Then I entered a test text. Last I toggled the hasBorder property. SwiftUI destroys the old view that holds the state with the text. It replaces the older view with a new view that has an empty state string.



This can lead to unexpected behavior, such as losing user input. While conditional view modifiers can be useful in certain cases, they should not be the default strategy. It's important to carefully consider where and when to use them. In many situations, using ternary operators provides a simpler and more reliable solution.

10.9 ANYLAYOUT - SWITCHING BETWEEN LAYOUT CONTAINERS

In this section, we will delve into the details of switching between different layout containers in SwiftUI. Let's start with a basic example where we have a state property called isHorizontal that we connect to a toggle. We can use this state to determine which layout container to display. If isHorizontal is true, we show an HStack; otherwise, we use a VStack.

```

structExampleView: View {
    @State private var isHorizontal: Bool = true

    var body: some View {
        VStack {
            if isHorizontal {
                HStack {
                    texts
                }
            } else {
                VStack {
                    texts
                }
            }
        }

        Toggle(horizontal ? "horizontal" : "vertical",

```

```

        isOn: $isHorizontal.animation(.easeInOut(duration: 2)))
    }
}

@ViewBuilder
var texts: some View {
    Text("First Text")
        .padding()
        .background(Color.yellow)
    Text("Second Text")
        .padding()
        .background(Color.cyan)
}
}

```

This approach seems fine at first glance, but there are a few problems with it. One issue is related to animations. If we add an animation to the state property, the default behavior is to fade out the old layout and fade in the new one. SwiftUI doesn't recognize that the texts in the HStack and VStack are the same views, so it doesn't know how to animate the transition smoothly. It uses the default animation behavior with is fade in/out:



Another problem arises when we have views within the layout containers that rely on state. If we switch between the layouts, we lose the state of those views. For example, if we have a text field within the layout, any input entered will be lost when switching between layouts. This is the same problem that we had in the [conditional view modifier section](#).

To address these issues, SwiftUI introduced a new view called `AnyLayout` in iOS 16. This view allows us to conditionally create a container based on the current state. **By using `AnyLayout`, we can preserve the identity of the views and maintain their state during layout transitions.**

If I use `AnyLayout` for the example from above:

```

struct AnyLayoutExampleView: View {
    @State private var isHorizontal: Bool = true

    var layout: AnyLayout {
        isHorizontal ? AnyLayout(HStackLayout()) : AnyLayout(VStackLayout())
    }

    var body: some View {
        VStack {
            layout {
                Text("First Text")
                    .padding()
                    .background(Color.yellow)
                Text("Second Text")
                    .padding()
            }
        }
    }
}

```

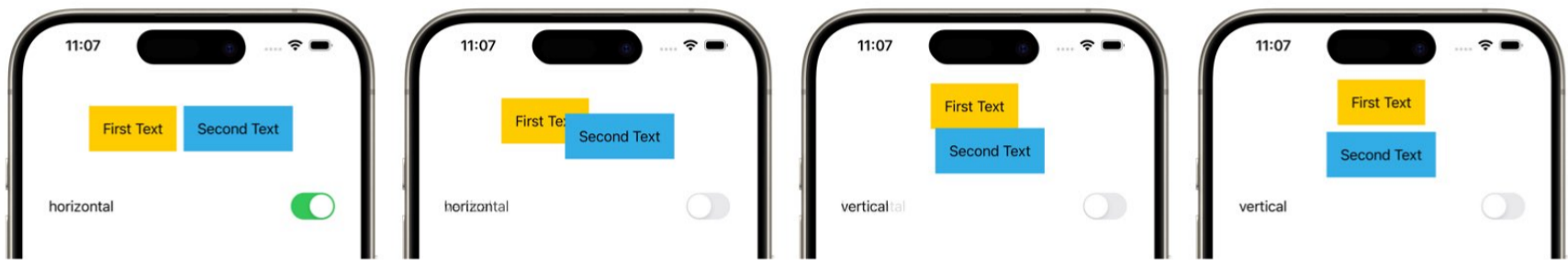
```

        .background(Color.cyan)
    }

    Toggle(isHorizontal ? "horizontal" : "vertical",
          isOn: $isHorizontal.animation(.easeInOut(duration: 2)))
        .padding()
    }
}
}

```

Now, when we switch between layouts, the views maintain their identity, allowing for smooth animations and preserving any state associated with them.



AnyLayout is a powerful tool that you should consider using whenever you need to switch between different layout containers without losing state. It was introduced in iOS 16 and can greatly improve the user experience of your app.

10.10 VIEWTHATFITS

In previous sections, we explored various properties like dynamic types and size classes to conditionally determine the layout. However, there are cases where we simply want to ensure that our content fits within the available space without worrying about different situations. That's where the new container, ViewThatFits introduced with iOS 16, comes into play.

The ViewThatFits container allows us to provide multiple variations of your layout. It automatically selects the variation that fits best within the available space, eliminating the need to consider all the different possibilities.

Let's consider an example. Imagine we have three texts: a longer text, a medium text, and a smaller text. Normally, when the accessibility settings are set to a larger size, the longer text may not fit on a single line and wrap to multiple lines. However, we want to ensure that the text always remains on a single line. If it doesn't fit, we want to switch to a smaller version. This is where View That Fits comes in handy:

```

ViewThatFits {
    Text("This is a long text with more information")
        .foregroundColor(.green)
    Text("This is a short text.")
        .foregroundColor(.red)
    Text("Very short text")
        .foregroundColor(.indigo)
}

```



However, using View That Fits can become more complicated when dealing with advanced examples. To understand its behavior better, let's dive into how it calculates the size.

According to the documentation, **ViewThatFits selects the first child whose ideal size on the constrained axis fits within the proposed size.** This may sound a bit complex, but it's essentially asking the child view its ideal size. If the size of the child view is smaller than the available size, the child view fits and is placed. If not ViewThatFits goes to the next child view and tests if it fits.

You can see the ideal size of a view by using the fixed size modifier. By applying the Fixed Size modifier, we can see the intrinsic size of the view, which is the size it prefers when given both vertical and horizontal directions unlimited space:

```
Text("This is a long text with more information")
    .fixedSize()
```

In the following image, you can see that the text view will always use its ideal size. For the larger accessibility setting the text does not fit on the screen:



ViewThatFits would compare the size of the child view to the available space. It would determine that for the accessibility settings the child does not fit and try the next child in the list.

Per default ViewThatFits will use the ideal size in both vertical and horizontal direction:

```
ViewThatFits(in: [.horizontal, .vertical]) {
    ...
}
```


You can restrict the child views to fit only in the vertical direction, the view will use its ideal size in the vertical dimension while still considering the proposed size in the horizontal dimension.

```
ViewThatFits(in: .vertical)
```

For example, the container would ask the child view “How do you fit if I give you unlimited space in the vertical direction and e.g. 300 points (the available space) in the horizontal direction. This is equivalent to using:

```
Text("This is a long text with more information")  
    .fixedSize(horizontal: false, vertical: true)
```

Similarly, if we restrict the view to fit only in the horizontal direction, it will use its ideal size in the horizontal dimension while considering the proposed size in the vertical dimension.

```
ViewThatFits(in: .horizontal) {  
    ...  
}
```

This is equivalent to using:

```
Text("This is a long text with more information")  
    .fixedSize(horizontal: true, vertical: false)
```

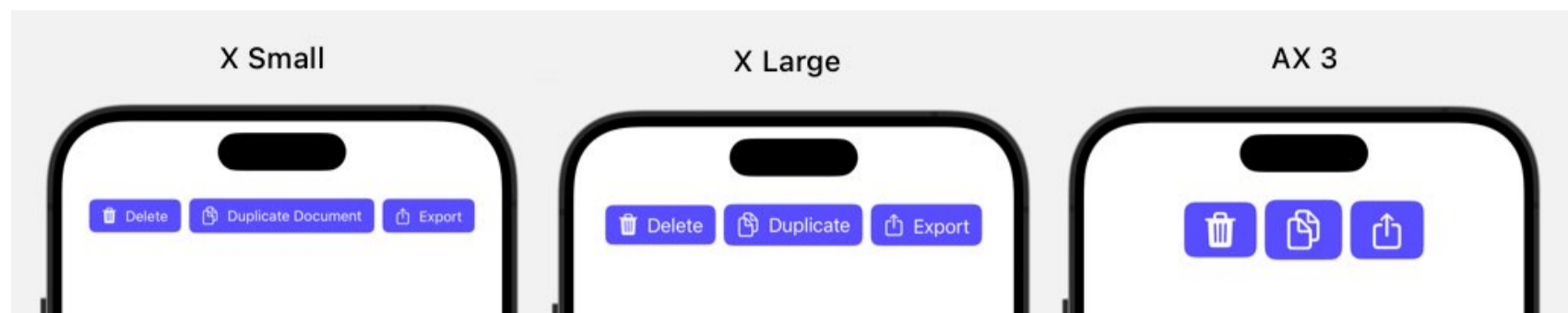
To test and visualize these scenarios, we can use the Fixed Size modifier with different restrictions. This allows us to see what size the layout actually considers and helps us understand the behavior of ViewThatFits. By using the Fixed Size modifier, you can gain better insights into how the view sizes itself and make informed decisions about our layout.

In the upcoming examples, we’ll explore different use cases where the behavior of ViewThatFits becomes clearer.

10.11 ViewThatFits Example 1

In this section, I’m going to show you an example of ViewThatFits where we display three buttons. Usually, buttons have a label, which consists of an icon and text. However, there may be cases where we want to show variations of the label based on the available space. For instance, if there is ample space, we can display both the icon and a larger text text. If the space is smaller, we can show the icon and a shorter text. And if space is very limited, we can simply show only the icon.

In the following screenshots, I tested this to adapt for different screen sizes and accessibility settings:



To achieve different variations of this button, we'll create a subview called `AdjustableButton`, which is a view. This subview will have an action closure that doesn't take any arguments or return any value (void). We'll also include a long title string, a short title string, and an icon name, all of which are of type string.

To switch between the different variations, we'll use the `ViewThatFits`. The largest variation will use the icon and large title property. Next, we'll have a smaller version with a short title, and finally, the shortest version will only display the system icon using an `Image` view with the system name.

```
struct AdjustableButton: View {  
  
    let longTitle: String  
    let shortTitle: String  
    let iconName: String  
    let action: () -> Void  
  
    init(longTitle: String,  
         shortTitle: String,  
         iconName: String,  
         action: @escaping () -> Void) {  
        self.longTitle = longTitle  
        self.shortTitle = shortTitle  
        self.iconName = iconName  
        self.action = action  
    }  
  
    var body: some View {  
        ViewThatFits {  
            Button(action: action) {  
                Label(longTitle, systemImage: iconName)  
            }  
  
            Button(action: action) {  
                Label(shortTitle, systemImage: iconName)  
            }  
  
            Button(action: action) {  
                Image(systemName: iconName)  
            }  
        }  
        .buttonStyle(.borderedProminent)  
    }  
}
```

You can then use this reusable component to create an HStack with 3 buttons like:

```
HStack {
  AdjustableButton(longTitle: "Delete Document",
    shortTitle: "Delete",
    iconName: "trash.fill") { }
  AdjustableButton(longTitle: "Duplicate Document",
    shortTitle: "Duplicate",
    iconName: "doc.on.doc") { }
  AdjustableButton(longTitle: "Export Document",
    shortTitle: "Export",
    iconName: "square.and.arrow.up") { }
}
```

This adaptive layout works well for different screen sizes as well. As a developer, you can start designing on the smallest screen, like the iPhone 8, and then test it on larger screens to ensure it looks good across devices.

In this example, we adjusted each button individually, but you can also have a variation where all three buttons are displayed as icons only or with both icons and text. In that case, the `ViewThatFits` modifier would be applied at a higher level.

10.12 ViewThatFits Example 2

Let's consider a scenario where you have a list of notes, each with different information such as a title and a timestamp indicating the creation date. Since the title length can vary due to user input, we need to be able to adjust the layout accordingly.

To tackle this challenge, we can utilize the `ViewThatFits` approach. It allows us to determine whether to display the title and timestamp on the same line or separate lines based on the available space. Even if we increase the text size, the layout scales gracefully, making efficient use of the available space.



To implement this adaptive layout, we'll create a `NoteListView` where we'll define a constant array of example notes. Each note will have properties like title, favorite status, creation date, color tag, and content.

```

struct SizeThatFitsListView: View {
    let notes = Note.examples()

    var body: some View {
        NavigationStack {
            List {
                ForEach(notes) { note in
                    NoteRow(note: note)
                }
            }
            .navigationTitle("Notes")
        }
    }
}

```

Next, I define a `NoteRow` which is used in the `NoteListView`. I am using the `title`, `isFavorite`, and `creationDate` properties of each note. To avoid code duplication, I extract the individual elements as computed properties:

```

struct NoteRow: View {
    note: Note

    var body: some View {
        ViewThatFits {
            HStack {
                favIcon
                titleView
                Spacer()
                dateView
            }

            HStack(alignment: .firstTextBaseline) {
                favIcon
                VStack(alignment: .leading) {
                    titleView
                    .lineLimit(2)
                    dateView
                }
            }
        }
    }

    var dateView: some View {
        Text(note.creationDate.formatted(date: .abbreviated,
                                         time: .shortened))
            .font(.caption)
            .foregroundColor(.gray)
    }

    var favIcon: some View {
        Image(systemName: "heart.fill")
            .foregroundColor(note.isFavorite ? .accent : .clear)
    }

    var titleView: some View {
        Text(note.title)
    }
}

```

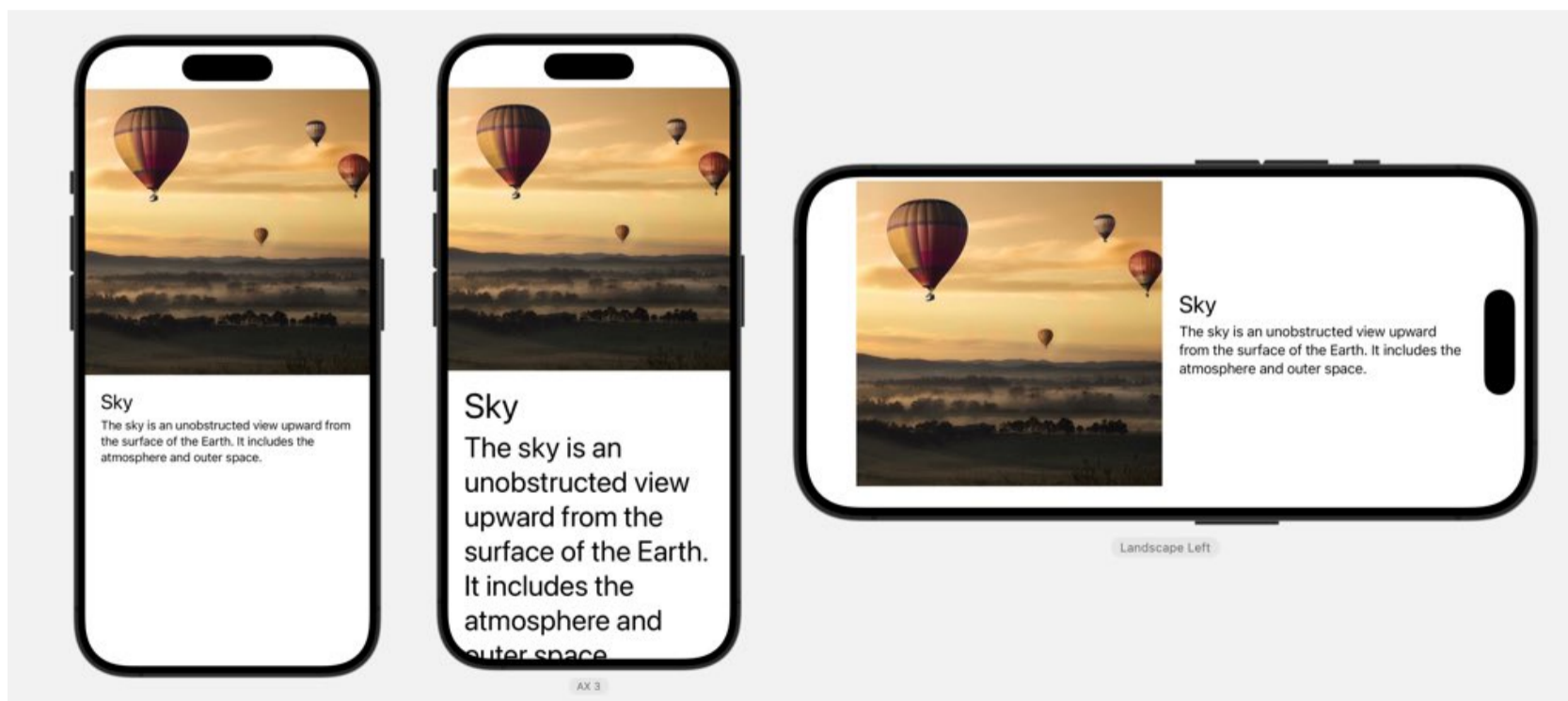
For the body of the NoteRow I use ViewThatFits with 2 different versions. I use an HStack first which is shown e.g. for smaller content and a VStack for larger content.

Additionally, you can also use the dynamic type size to **adjust the date formatting for accessibility settings**. In the following, I am showing a shorter date if accessibility settings are turned on:

```
struct NoteRow: View {  
  
    note: Note  
    @Environment(\.dynamicTypeSize) var typeSize  
  
    var body: some View {  
        ViewThatFits {  
            ...  
        }  
    }  
  
    var dateView: some View {  
        Text(note.creationDate.formatted(date: .abbreviated,  
                                         time:  
                                             typeSize.isAccessibilitySize ? .omitted : .shortened))  
            .font(.caption)  
            .foregroundColor(.gray)  
    }  
    ...  
}
```

10.13 ViewThatFits Example 3

In this section, we will focus on an example of an inspiration detail view that consists of an image, title, and text. Our goal is to create a layout that adapts to different orientations, screen sizes, and dynamic type variations. I already implemented a similar version of [InspirationDetailView with GeometryReader](#) and [InterfaceSizeClasses](#). Let's see how to use ViewThatFits for this use case.



To begin, let's create a separate view of the InspirationDetailView using the ViewThatFits container. We'll have one version with a Portrait layout and another with a Landscape layout. This will allow us to test and preview each version independently.

```
struct ViewThatFitsInspirationDetailView: View {
    let inspiration = NatureInspiration.example1()

    var body: some View {
        ViewThatFits(in: .horizontal) {
            LandscapeInspirationDetailView(inspiration: inspiration)
            PortraitInspirationDetailView(inspiration: inspiration)
        }
    }
}
```

To avoid code repetition, I am creating additional subviews for the image and text views:

```
struct TextsInspirationDetailView: View {
    let inspiration: NatureInspiration
    var body: some View {
        VStack(alignment: .leading, spacing: 5) {
            Text(inspiration.name)
                .font(.title)
            Text(inspiration.description)
        }
        .padding(12)
    }
}

struct ImageInspirationDetailView: View {
    let inspiration: NatureInspiration
    var body: some View {
        ImageAspectView(imageName: inspiration.imageName,
                        frameAspectRatio: 1)
    }
}
```

For the landscape version, I am using a VStack. However, we encounter a problem when using larger dynamic type sizes. The text becomes too big to fit within the screen, causing overflow. To address this, we can embed the VStack in a ScrollView to allow scrolling when necessary.

```
struct PortraitInspirationDetailView: View {
    let inspiration: NatureInspiration
    var body: some View {
        ScrollView {
            VStack {
                ImageInspirationDetailView(inspiration: inspiration)
                TextsInspirationDetailView(inspiration: inspiration)
            }
        }
    }
}
```

For the landscape version, I am using an HStack:

```
struct LandscapeInspirationDetailView: View {
    let inspiration: NatureInspiration
    var body: some View {
        GeometryReader(content: { geometry in
            ScrollView {
                HStack {
                    ImageInspirationDetailView(inspiration: inspiration)
                        .frame(minHeight: geometry.size.height)

                    TextsInspirationDetailView(inspiration: inspiration)
                }
            }
        })
        .frame(minWidth: 431)
    }
}
```

To see what layout `ViewThatFits` considers, I am adding previews for both the landscape and portrait version. I am adding the `fixedSize` modifier, which will apply the same sizing behavior as **`ViewThatFits(in: .horizontal)`**:

```
#Preview("Portrait", traits: .portrait, body: {
    PortraitInspirationDetailView(inspiration: NatureInspiration.example1())
})

#Preview("Landscape", traits: .landscapeLeft, body: {
    LandscapeInspirationDetailView(inspiration: NatureInspiration.example1())
        .fixedSize(horizontal: true, vertical: false)
})
```

By testing and previewing each version separately, we can gain a better understanding of how the layout behaves. We can use the `fixedSize` modifier with different variations to see how the views resize and fit within the available space. This allows us to fine-tune the layout and ensure it meets our requirements.

10.14 KEYBOARD LAYOUT ADJUSTMENTS

When you're working with SwiftUI, it's important to consider how the on-screen keyboard affects your layout. The following example includes a text, image, text field, and a save button:

```
struct EditCatView: View {
    @State private var text = ""

    var body: some View {
        VStack {
            Text("What is your cats name?")
                .font(.title)
            Image(.cat2)
                .resizable()
                .scaledToFit()
        }
    }
}
```

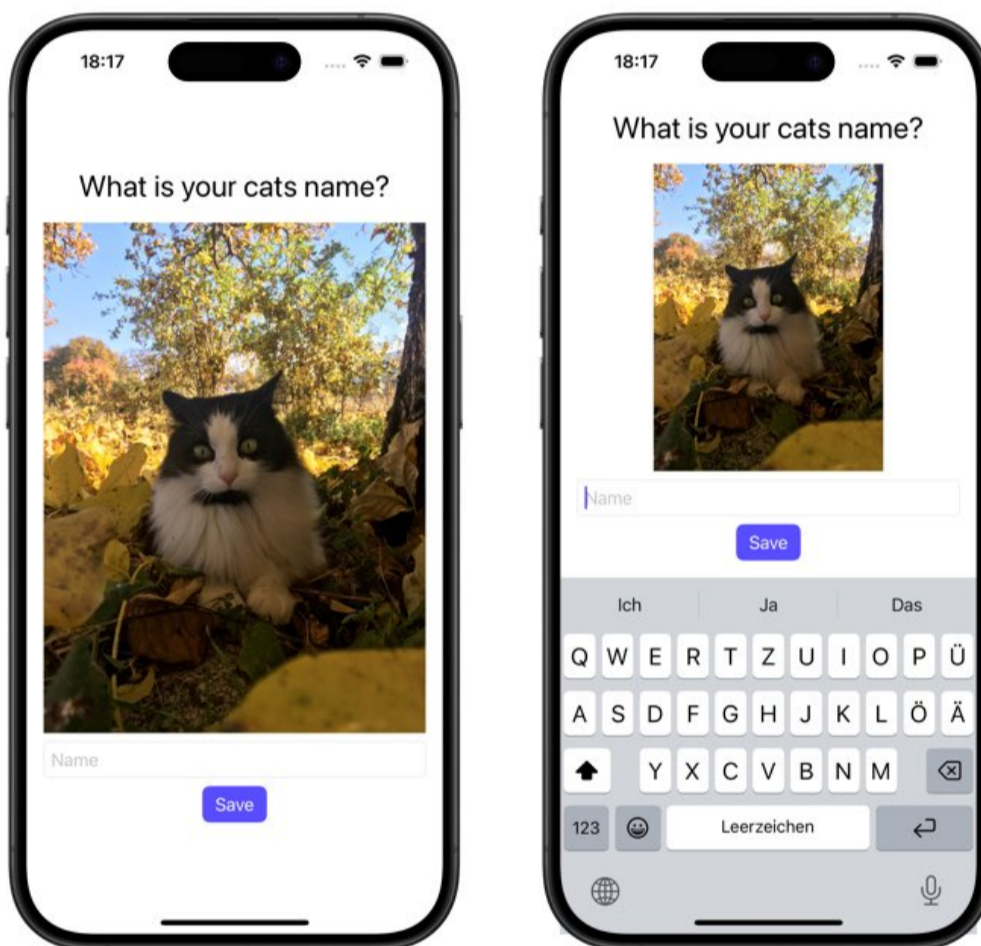
```

        TextField("Name", text: $text)
            .textFieldStyle(.roundedBorder)

        Button {
        } label: {
            Text("Save")
        }
            .buttonStyle(.borderedProminent)
    }
    .padding()
}
}
}

```

When you tap on a text field in your app, the keyboard slides in. If it doesn't appear in your simulator, it might be because you've disabled it. You can toggle the keyboard visibility with Command + K.



By default, SwiftUI tries to adapt to the presence of the keyboard. However, not all views are equally flexible. For instance, a text field or a button typically wants to maintain a height that fits one line of text. The title is usually designed to fit on a single line as well. The most adaptable element in our example is the image, which is why it scales down when the keyboard appears.

Handling Fixed Heights

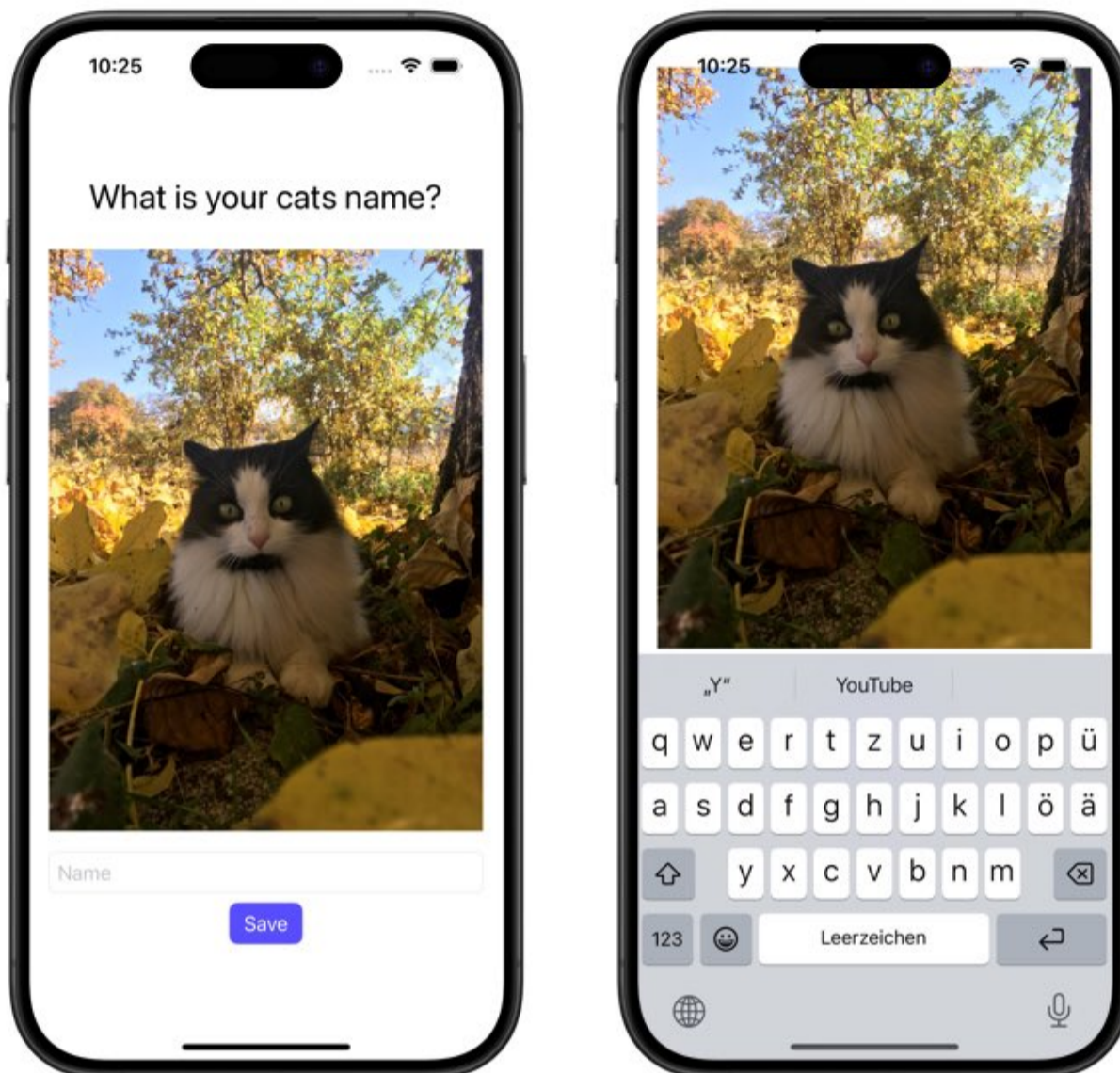
Let's consider what happens if you set a fixed height on the image:

```

Image(.cat2)
    .resizable()
    .scaledToFit()
    .frame(height: 500)

```


With a fixed height of 500 points, the image no longer resizes when the keyboard appears. This can cause layout issues, as the image may take up more space than is available, obscuring the text field and preventing you from seeing what you're typing. This is not the behavior you want.



Flexible Frames to the Rescue

To avoid such issues, using a flexible frame for the image is a good solution. It allows the image to adapt to the available space when the keyboard is displayed.

```
Image(.cat2)
    .resizable()
    .scaledToFit()
```

By making the image resizable and setting its aspect ratio to `.fit`, you ensure that it scales down appropriately when the keyboard is active, maintaining the integrity of your layout.

Remember, when designing your layouts, always consider how the keyboard will impact the user experience. By making your views flexible and responsive, you can create a seamless experience even when the keyboard is on screen.

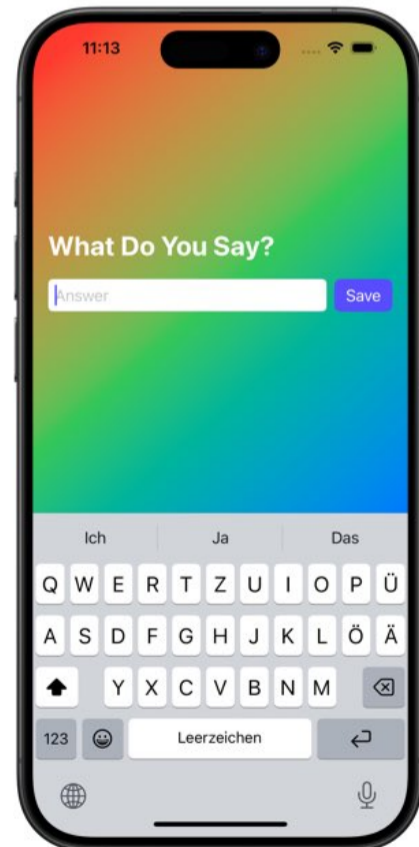
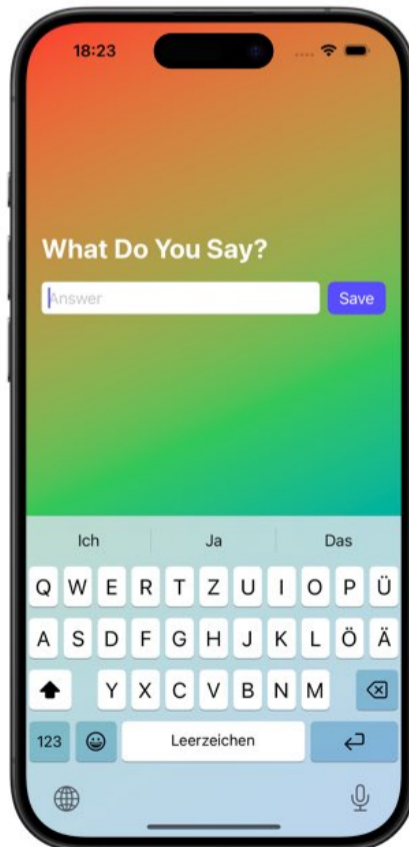
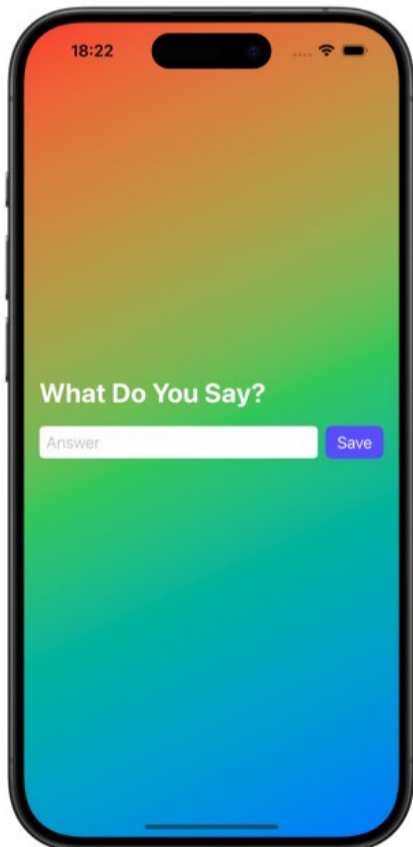
10.15 Keyboard & Background Image

In the following example, I have a gradient background. When the keyboard is shown, the whole layout adjusts including the gradient.

```
struct BackgroundGradientEditView: View {
    @State private var text = ""

    var body: some View {
        VStack(alignment: .leading) {
            Text("What Do You Say?")
                .font(.title)
                .bold()
                .foregroundColor(.white)
            HStack {
                TextField("Answer", text: $text)
                    .textFieldStyle(.roundedBorder)

                Button("Save") { }
                    .buttonStyle(.borderedProminent)
            }
        }
        .padding()
        .padding(.bottom, 50)
        .frame(maxWidth: .infinity, maxHeight: .infinity)
        .background(
            LinearGradient(colors: [Color.red, Color.green, Color.blue],
                startPoint: .topLeading,
                endPoint: .bottomTrailing)
        )
    }
}
```



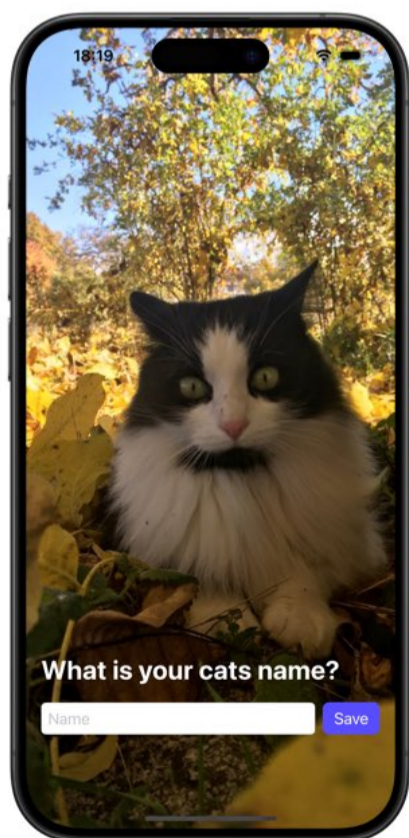
gradient adjusts when keyboard is shown

To prevent the layout adjustments, I can add an `ignoreSafeArea` modifier for keyboard. Attaching this to the gradient will leave the gradient unchanged when the keyboard appears. Additionally, I also ignore the container safe area insets, so that the gradient fills out the whole screen.

```
.background(  
  LinearGradient(...)  
  .ignoreSafeArea([.keyboard, .container])  
)
```

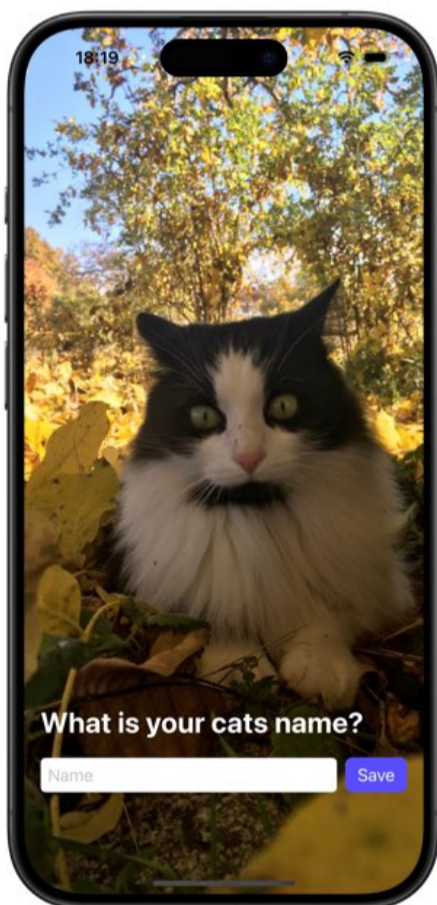
A similar example uses an image in the background. I am again using the `ignoreSafeArea` modifier to keep the image in the background unchanged:

```
struct BackgroundEditCatView: View {  
  @State private var text = ""  
  var body: some View {  
    VStack(alignment: .leading) {  
      ""  
      TextField("Name", text: $text)  
    }  
    .padding()  
    .padding(.bottom, 50)  
    .frame(maxWidth: .infinity, maxHeight: .infinity,  
          alignment: .bottom)  
    .background (   
      Color.gray.overlay {  
        Image(.cat2)  
          .resizable()  
          .scaledToFill()  
      }  
      .ignoreSafeArea(.all)  
    )  
  }  
}
```



Using the `ignoresSafeArea` modifier is crucial in scenarios like this. However, it's important to understand where to apply it to achieve the desired results. If you attach it after the background view, all the views within that hierarchy will ignore the safe area and the keyboard. This can lead to situations where the keyboard overlaps with important elements, such as text fields, making it difficult for users to see what they're typing.

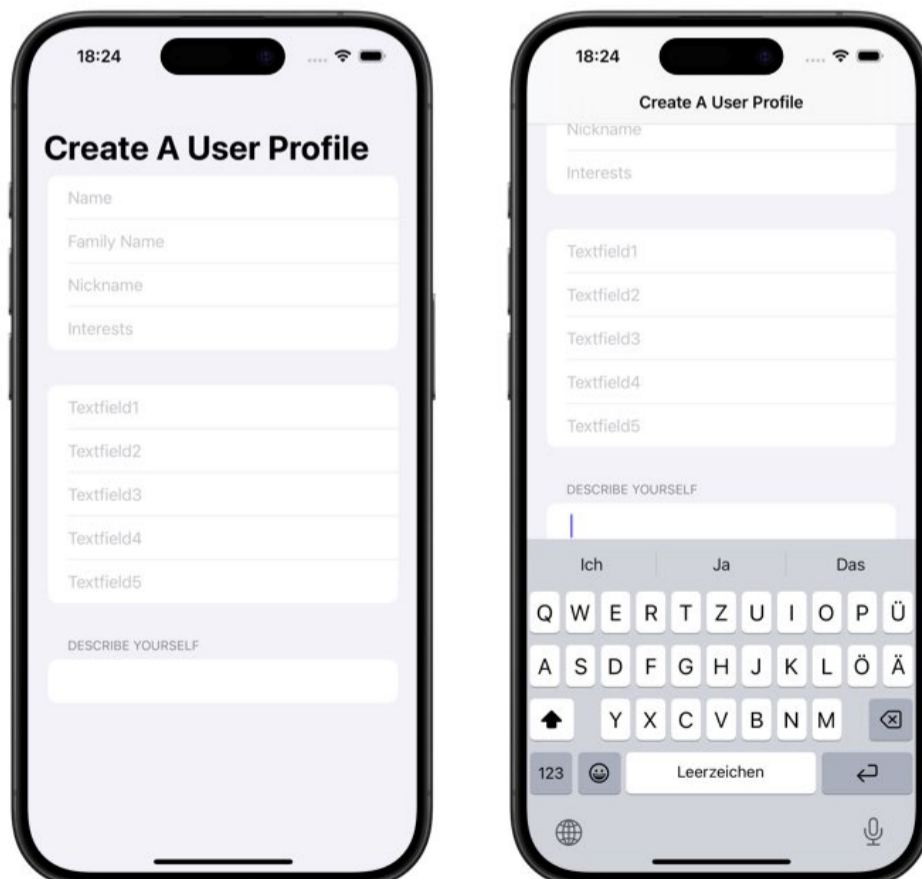
```
struct BackgroundEditCatView: View {
    @State private var text = ""
    var body: some View {
        VStack(alignment: .leading) {
            ""
            TextField("Name", text: $text)
        }
        ""
        .background(...)
        .ignoresSafeArea(.all)
    }
}
```



10.16 Keyboard & Forms

Let's dive into an example where we create a user profile view with several text fields for the name, family name, nickname, interests, and a few other random ones. Additionally, we'll include a multiline text editor.

```
struct CreateProfileView: View {
    @State var name: String = ""
    @State var familyName: String = ""
    @State var nickname: String = ""
    @State var interests: String = ""
    @State var profileDescription: String = ""
    @State var textfieldText: String = ""
    var body: some View {
        NavigationStack {
            List {
                TextField("Name", text: $name)
                TextField("Family Name", text: $familyName)
                TextField("Nickname", text: $nickname)
                TextField("Interests", text: $interests)
                Section {
                    TextField("Textfield1", text: $textfieldText)
                    TextField("Textfield2", text: $textfieldText)
                    TextField("Textfield3", text: $textfieldText)
                    TextField("Textfield4", text: $textfieldText)
                    TextField("Textfield5", text: $textfieldText)
                }
                Section("Describe Yourself") {
                    TextEditor(text: $profileDescription)
                }
            }
        }
        .navigationTitle("Create A User Profile")
    }
}
```



The key to handling the keyboard effectively is to use a Form. A Form in SwiftUI is similar to a List and comes with a styled scroll view, making it inherently scrollable. This is particularly useful when you have text fields within a scroll view because the view automatically adjusts to ensure the active text field is visible when the keyboard appears.

Here's what happens: when you tap on a text field, the form's scroll view moves so that the text field is not obscured by the keyboard. You can see the text field clearly, which enhances the user experience.

VStack vs. Form/List/ScrollView

To illustrate the difference, let's compare this to a layout that uses a VStack instead of a Form. If you place the same text fields and text editor within a VStack, you'll notice that when you tap on the multiline text editor at the bottom, it gets hidden behind the keyboard. This is not ideal because the user can't see what they're typing.

In contrast, using a Form, List or ScrollView ensures that when you tap into any text field, the list or scroll view moves the text field into view above the keyboard. This automatic adjustment has been a feature since iOS 14, and it's a significant improvement over iOS 13, where such adaptability was not provided by default.

Automatic Adjustments and Potential Hiccups

Be aware that there might be occasional delays or hiccups when the keyboard is showing, but generally, these adaptations work well.

In the example of our user profile form, you don't need to change anything as long as you use a List or ScrollView. Your users can enter information into the text fields, see what they're typing, and the keyboard is managed automatically.

10.17 Keyboard toolbar

There are times when you may want to add additional elements that stick to the edge of the keyboard and only appear when the keyboard is shown. To achieve this, you can use the toolbar and add a toolbar item with the placement of the keyboard.

Let's say you have a text field and you want to add some buttons similar to the suggestion text. You can simply add a toolbar item with a button inside. For example, you can add a button that says "Click me" and see where it appears.

```
struct KeyboardToolbarExampleView: View {
    @State private var name = "Taylor"
    var body: some View {
        TextField("Enter your name", text: $name)
            .textFieldStyle(.roundedBorder)
            .padding()
            .toolbar(content: {
                ToolbarItem(placement: .keyboard) {
                    Button("Click me") { }
                }
            })
    }
}
```

```
}  
}
```

When you open the keyboard, you'll notice an extra button appearing at the top. You can also add multiple elements like this. These buttons act as quick actions and will disappear when you remove the keyboard or press the return key.



Now, if you want to have something that sticks to the bottom edge instead of only showing when the keyboard is visible, you can use the **safe area insets**. In this case, you can use the safe area insets for the bottom and add a button there with some styling, such as padding and a background color.

```
struct SafeAreaInsetExampleView: View {  
    @State private var name = "Taylor"  
  
    var body: some View {  
        TextField("Enter your name", text: $name)  
            .textFieldStyle(.roundedBorder)  
            .padding()  
            .frame(maxWidth: .infinity, maxHeight: .infinity)  
            .background(  
                LinearGradient(colors: [Color.red, Color.green, Color.blue],  
                              startPoint: .topLeading,  
                              endPoint: .bottomTrailing)  
            )  
            .ignoresSafeArea([.keyboard, .container])  
        )  
        .safeAreaInset(edge: .bottom, content: {  
            Button("Click me") { }  
                .padding()  
                .frame(maxWidth: .infinity)  
                .background(.thinMaterial)  
        })  
    }  
}
```

To ensure the button appears above the background, you can attach it to the text field and add a background with a flexible frame. You can also make the background transparent or use materials like thin material, padding, and a frame to stretch it horizontally with a maximum width of infinity.



By doing this, the button will stick to the bottom edge and move up when the keyboard appears. However, it will always remain visible, unlike when using the toolbar placement for the keyboard, where it only shows when the toolbar is visible.

These are minor adjustments you can make when you want to place something close to the keyboard that adapts differently than your main content.

10.18 SUMMARY RESPONSIVE DESIGN

I already covered a lot of topics that are relevant to responsive design in the previous sections. Therefore I am not going to repeat it here. Instead, I will list the following main key takeaways:

- Use dividers and spacers which are flexible views
- Use flexible frames instead of fixed frames because they adapt automatically to the available space
- Know how to use alignment guides to position views
- Have a good understanding of how views like images and text are sized in SwiftUI
- Use layout priority to truncate text view behavior
- Make images resizable and respect the natural image aspect ratio
- embed your content in a ScrollView which will also hold potential larger content e.g. accessibility settings or smaller screen sizes

10.19 SUMMARY ADAPTIVE DESIGN

You can achieve more fine-tuned sizing with:

- Container relative frame which is especially useful for sizing content inside ScrollViews
- GeometryReader gives you full flexibility
- Write your own custom Layout container with the Layout Protocol
- ViewThatFits to give multiple view layout versions. This also works with custom layout containers
- LazyVGrid and LazyHGrid with adaptive Grid items have a lot of potential for adaptively adjusting layouts for e.g. device orientations

11. SPECIAL SYSTEM CONTAINERS

11.1 OVERVIEW OF SYSTEM CONTAINERS

In this section, I want to talk about system containers. They offer some advantages that can be quite beneficial for your app development. For instance, they come with specific styling that adapts well across platforms, providing a consistent look whether your app is on macOS or iOS. System containers include **List, Form, and Table**.

System containers in SwiftUI offer a range of benefits for your app development:

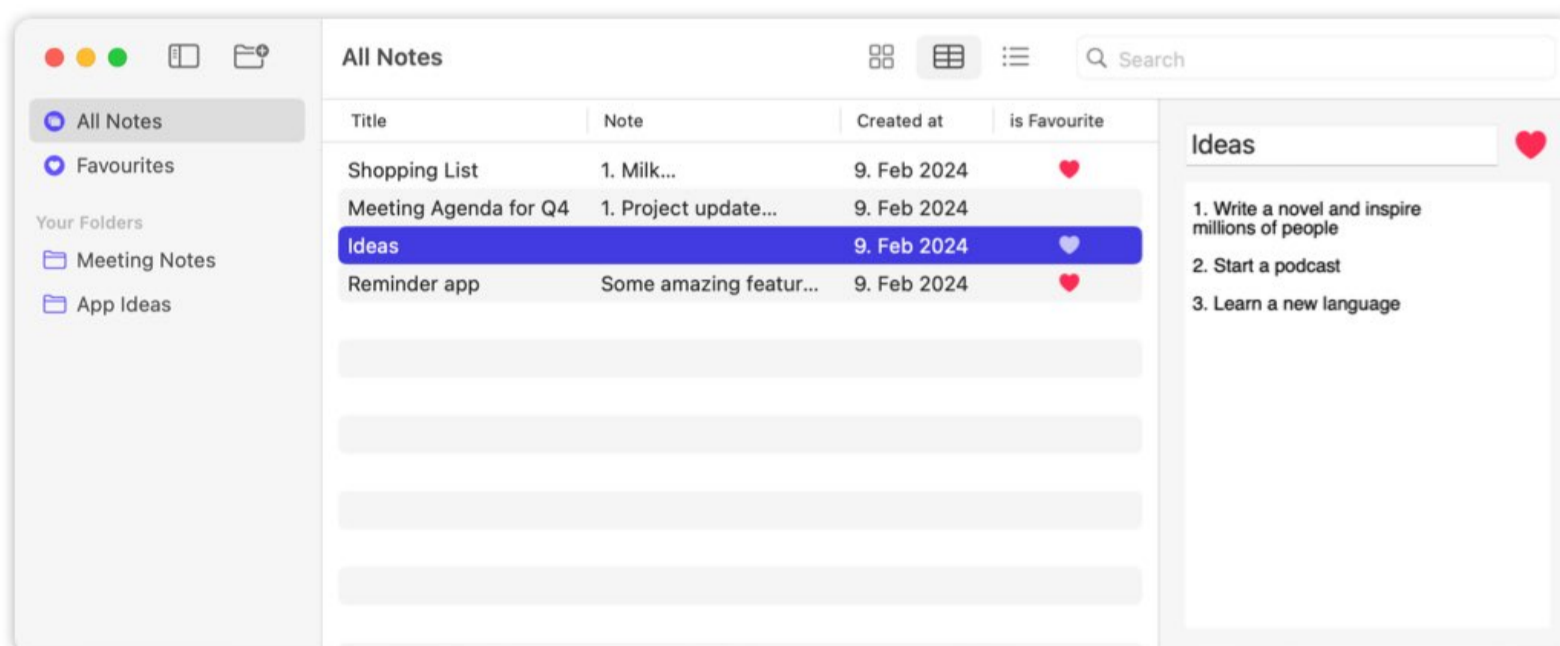
- **Native Styling and Behavior:** These components look like native Apple components e.g. Table has the same table-like layout as in the Finder app.
- **Adaptive Styling:** They come with specific styling that adapts seamlessly across platforms, such as macOS and iOS.
- **Built-in Interactions:** Containers like Table have features like tap-to-sort headers, and Lists offer swipe-to-delete actions.

When to Use System Containers

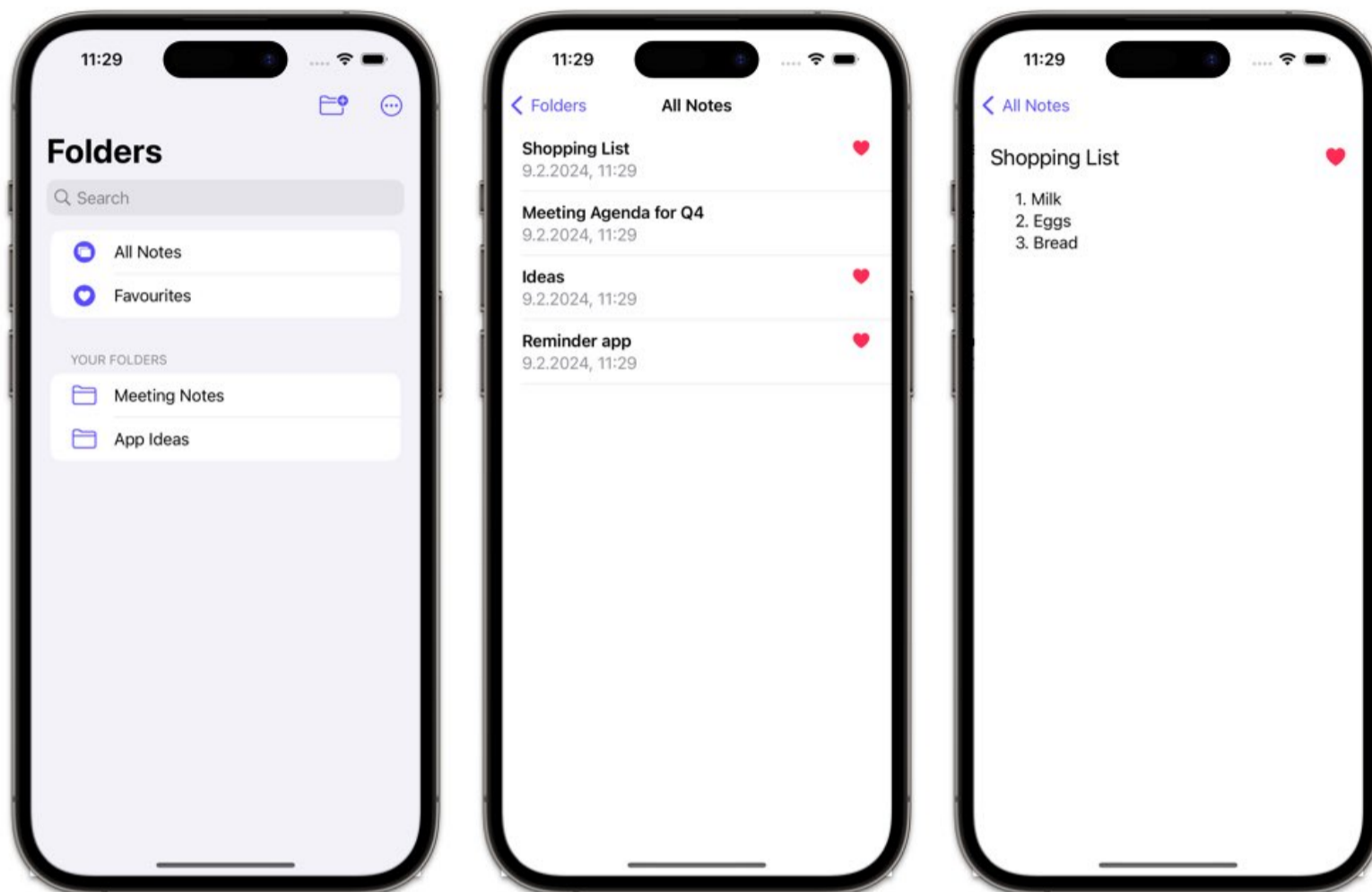
- **List vs. ScrollView:** If you're looking for predefined styling and behavior, a system container like List is a great choice. For more customization, consider using a ScrollView with a LazyVStack.
- **Lazy Loading:** Both Lists and Scroll Views with LazyVStacks load content lazily, which is efficient for performance.

Sample Project: Notes App

During this section, we'll work on a sample project—a little notes app. On the left side, there's a system List where you can select different categories like All Notes, Favorites, or Folders. I'll show you how to handle selections within a Navigation Split View, and we'll delve into the main area where a Table View allows for further interactions like sorting and filtering.



The same note-taking app on iOS shows the list in different styles. The first screen on the left uses the sidebar style, whereas the middle screen shows the list of notes in the inset list style.



11.2 ADDING MACOS TARGET

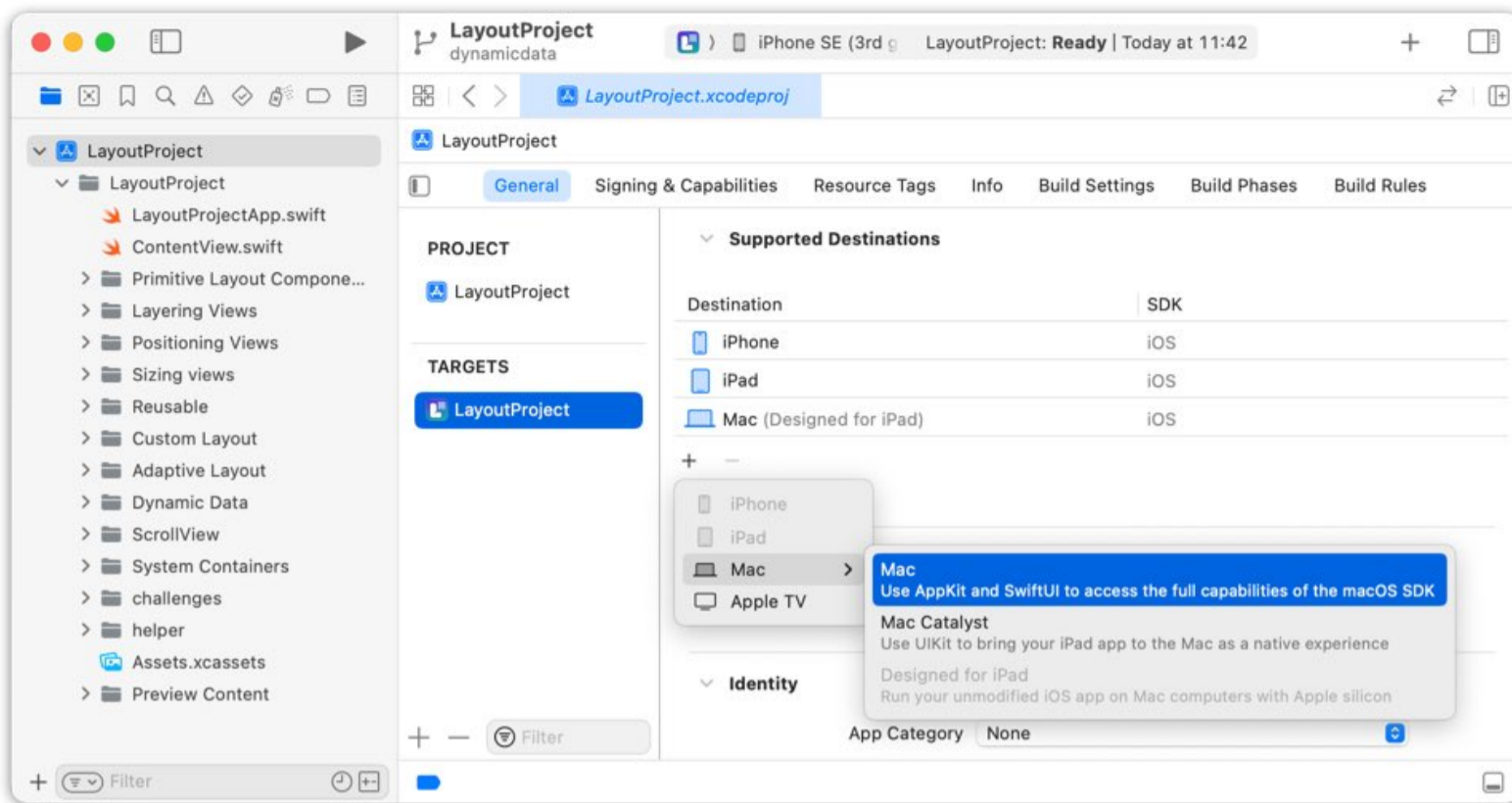
Before I dive into the details of system containers, I want to ensure that we can explore these views on macOS as well as iOS. Currently, my project is set up with a target for iOS, but I want to create a native macOS app. Let me walk you through the process of adding a macOS target to your existing SwiftUI project.

Step 1: Adjusting Project Settings

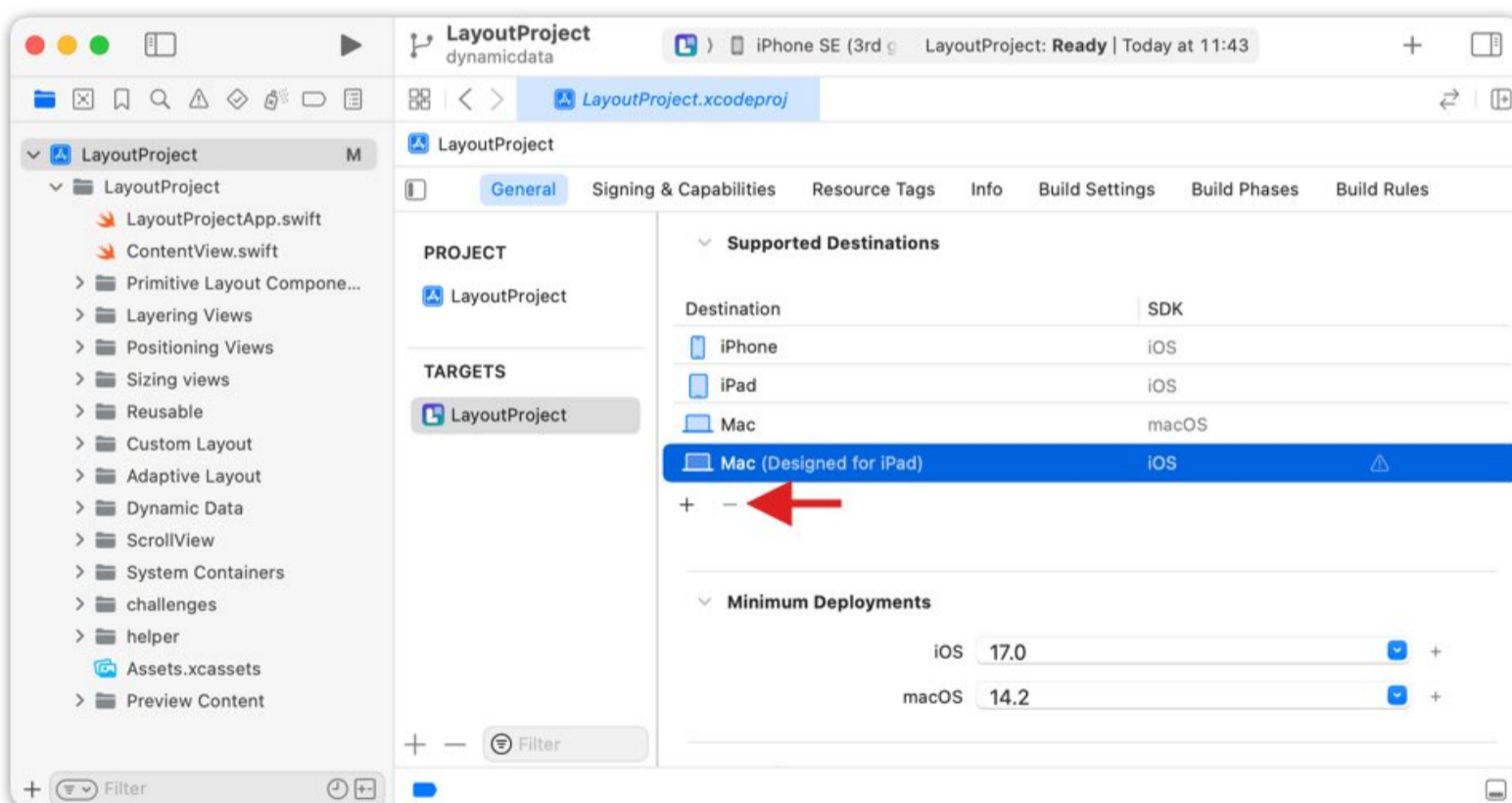
First, open your project settings and navigate to the target's general settings. You might notice that there's an option for "Mac" that's designed for iPad apps. This isn't what we're looking for; we want a native macOS app.

Step 2: Enabling macOS Support

To add macOS support, click on the plus button and select the option to enable it.



If you see two macOS targets, one being the “Designed for iPad” version, select it and press delete. We don’t need that one.



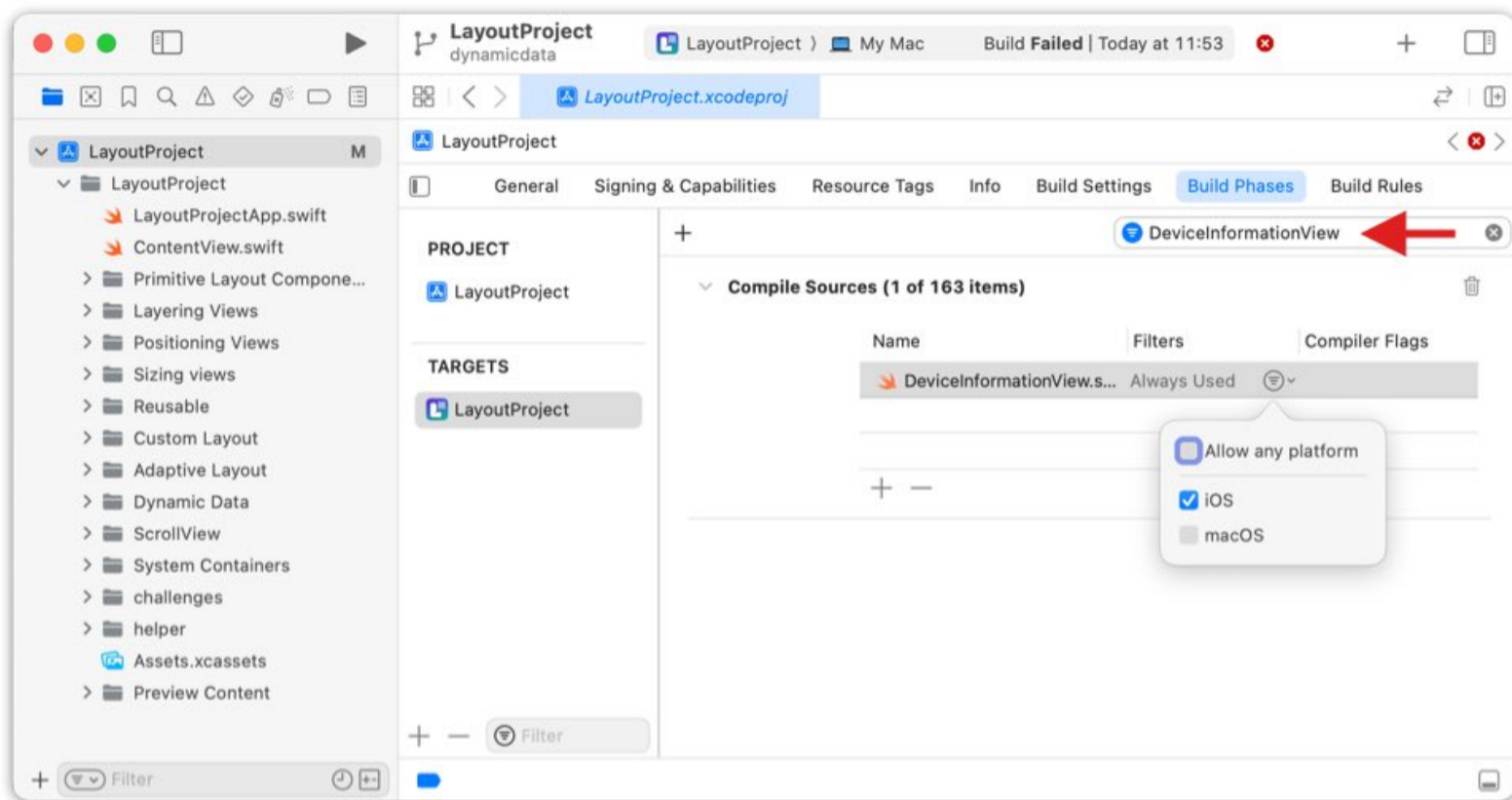
Step 3: Updating Deployment Info

I recommend setting the deployment target to macOS 14 and iOS 17. This allows us to use the latest features without worrying about availability checks.

Step 4: Resolving Compatibility Issues

After building for macOS, you might encounter some issues. For instance, `UIDevice` and `UIScreen` are part of `UIKit` and are not available on macOS. If you have files using these, you'll need to exclude them from the macOS build.

In your project settings, under the "Build Phases" tab, find the file that's causing the issue. Change its settings to make it available for iOS only.



Step 5: Updating Code for Cross-Platform Compatibility

In some cases, you'll need to update your code to use modifiers that are compatible with both platforms. For example, instead of using `navigationBarTitle`, which is iOS-specific, use the new modifier instead:

```
struct PetPalGalleryView: View {  
    ...  
    var body: some View {  
        NavigationStack {  
            ...  
            .navigationBarTitle("Pet Gallery")  
        }  
    }  
}
```

11.3 LIST

In this lesson, I'm going to introduce you to Lists in SwiftUI. We'll explore how to display a collection of data in a scrollable list. List view is loading its data lazily similar to LazyVStack. It is perfect for large data.

It's worth noting that on iOS, Lists use UICollectionView under the hood (iOS 16+), and on macOS, they use NSTableView. Be aware of performance considerations with NSTableView, especially when dealing with large datasets or custom cell heights.

Setting Up Example Data

The example data I'm using comes from a Note class and a Folder class. Each Note has properties like **title**, **isFavorite**, **creationDate**, **color**, and **content**. I've added a function to generate an array of sample notes to work with. You'll see this in action as we create our note list:

```
import SwiftUI
import Observation

@Observable class Note: Identifiable {

    var title: String
    var isFavorite: Bool
    let creationDate: Date
    var colorTag: Color
    var content: String

    static func examples() -> [Note] {
        [
            Note(title: "Shopping List",
                isFavorite: true,
                content: "1. Milk\n2. Eggs\n3. Bread",
                colorTag: .green),
            Note(title: "Meeting Agenda for Q4",
                isFavorite: false,
                content: "1. Project update\n2. Budget discussion\n3. Next steps",
                colorTag: .cyan),
            Note(title: "Ideas",
                isFavorite: true,
                content: "1. Write a novel and inspire millions of people \n2. Start
a podcast\n3. Learn a new language",
                colorTag: .orange),
        ]
    }
}
```

In addition, I have a Folder class that has a **notes** property that points to an array of Notes. This gives me the notes that belong to this folder:

```
import Foundation
import Observation

@Observable class Folder: Identifiable {
```

```

let id: UUID
let creationDate: Date
var name: String
var notes: [Note]

static func examples() -> [Folder] {
    [Folder(name: "Meeting Notes",
            notes: Note.examples()),
      Folder(name: "App Ideas",
            notes: [Note(title: "Reminder app",
                        isFavorite: true,
                        content: "Some amazing features for a reminder app",
                        colorTag: .blue)])])
}
}

```

Creating a List with Notes

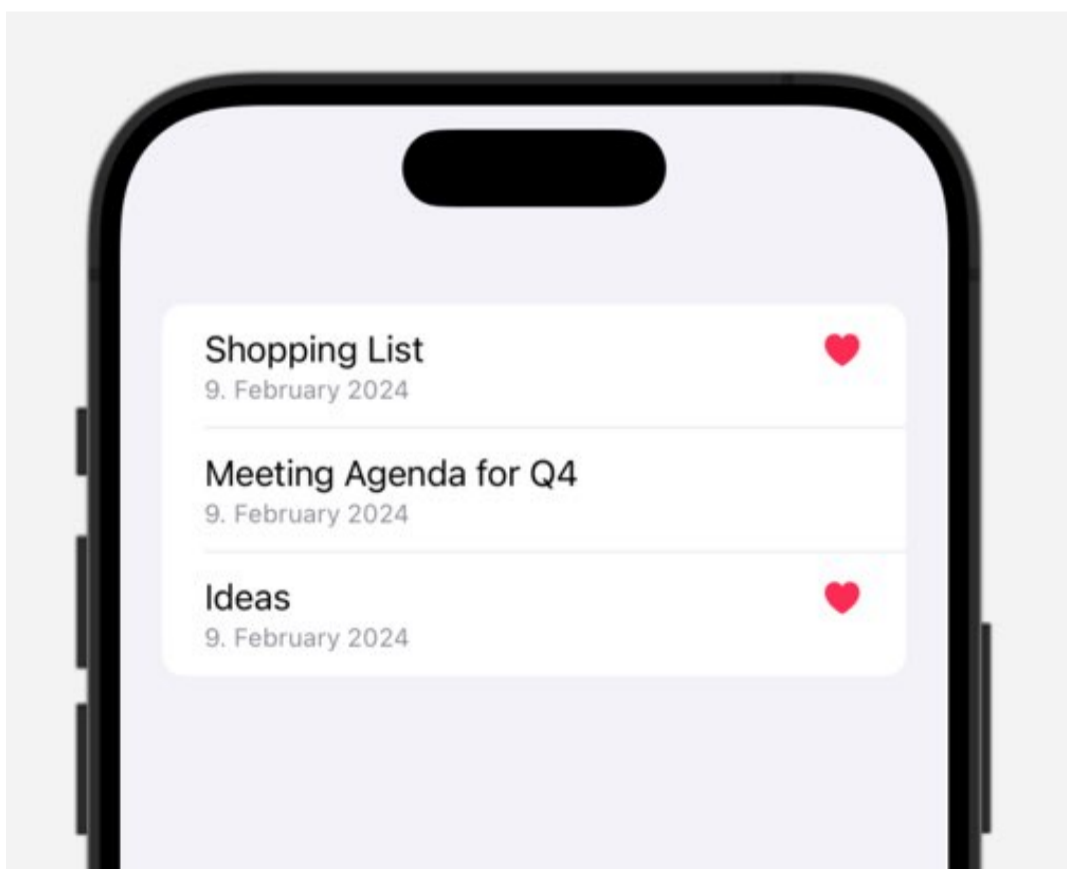
We can construct a basic List to display our notes. The Note data conforms to the Identifiable protocol, which makes it easier to work with List:

```

struct NoteListView: View {
    let notes = Note.examples()

    var body: some View {
        List(notes) { note in
            NoteRowView(note: note)
        }
    }
}

```



Customizing Note Rows

To make our list more informative, we'll create custom row view for each note. For instance, if you want to show the creation date below the note title, you can use a `VStack`. To add an icon for favorite notes, wrap the `VStack` in an `HStack` and use a `Spacer` to push the icon to the right.

```
struct NoteRowView: View {
    let note: Note

    var body: some View {
        HStack(alignment: .firstTextBaseline) {
            VStack(alignment: .leading) {
                Text(note.title)
                Text(note.creationDate, style: .date)
                    .font(.caption)
                    .foregroundColor(.gray)
            }
            Spacer()
            if note.isFavorite {
                Image(systemName: "heart.fill")
                    .foregroundColor(.pink)
            }
        }
    }
}
```

Adding Sections

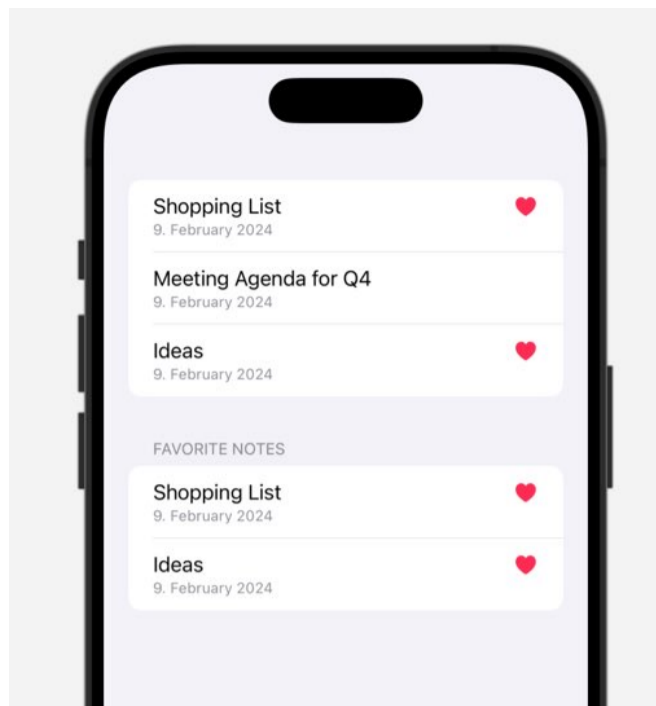
Lists in SwiftUI allow you to organize items into sections. In the following, I added a section that shows all the favorite notes:

```
struct NoteListView: View {
    @State private var notes: [Note] = Note.examples()

    var favoriteNotes: [Note] {
        notes.filter { $0.isFavorite }
    }

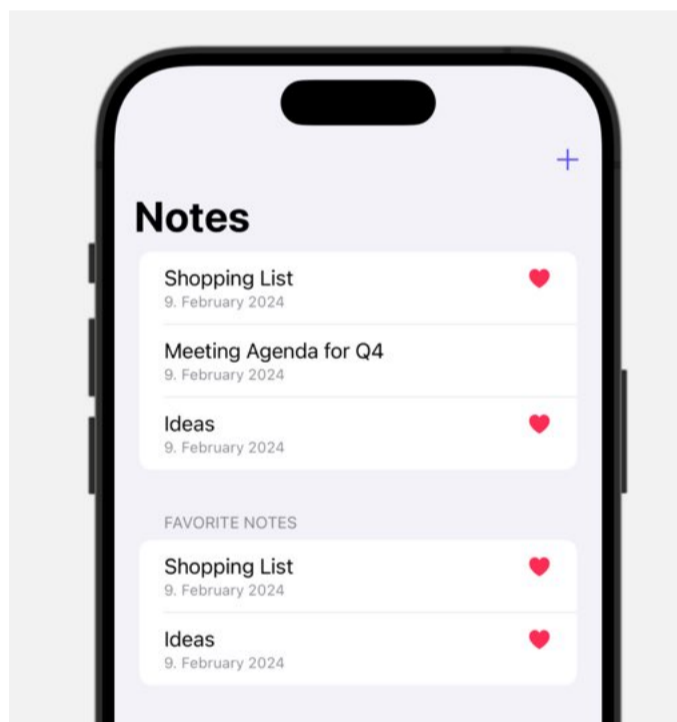
    var body: some View {
        List {
            ForEach(notes) { note in
                NoteRowView(note: note)
            }

            Section("Favorite Notes") {
                ForEach(favoriteNotes) { note in
                    NoteRowView(note: note)
                }
            }
        }
    }
}
```

Navigation and Toolbars

To add navigation and toolbars, we'll use a `NavigationStackView` and wrap our list within it. You can place action buttons in the navigation bar or the bottom toolbar for better accessibility and user experience.



```
NavigationStack {
  List {
    ForEach(notes) { note in
      NoteRowView(note: note)
    }

    Section("Favorite Notes") {
      ForEach(favoriteNotes) { note in
        NoteRowView(note: note)
      }
    }
  }
  .navigationTitle("Notes")
  .toolbar(content: {
```

```

        ToolbarItem(placement: .primaryAction) {
            Button(action: {
                notes.append(Note(title: "New Note"))
            }, label: {
                Label("Add New Note", systemImage: "plus")
            })
        }
    }
}

```

I added a 'plus' button in the toolbar, that adds a new note to the notes array.

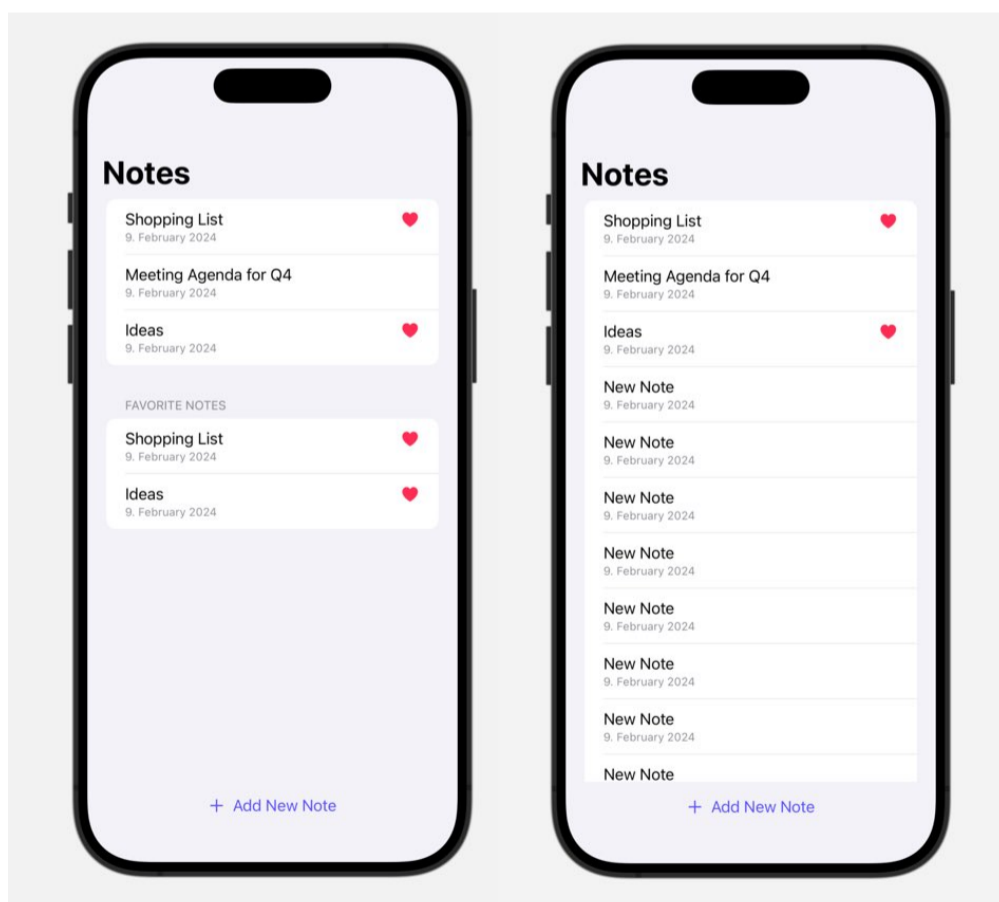
SafeAreaInsets

Alternatively, you can add buttons in the safeAreaInsets.

```

.safeAreaInset(edge: .bottom) {
    Button(action: {
        notes.append(Note(title: "New Note"))
    }, label: {
        Label("Add New Note", systemImage: "plus")
    })
    .padding()
    .frame(maxWidth: .infinity)
    #if os(iOS)
    .background(Color(.systemGroupedBackground))
    #endif
}

```



On iOS you could also use the toolbar with a bottom bar placement.

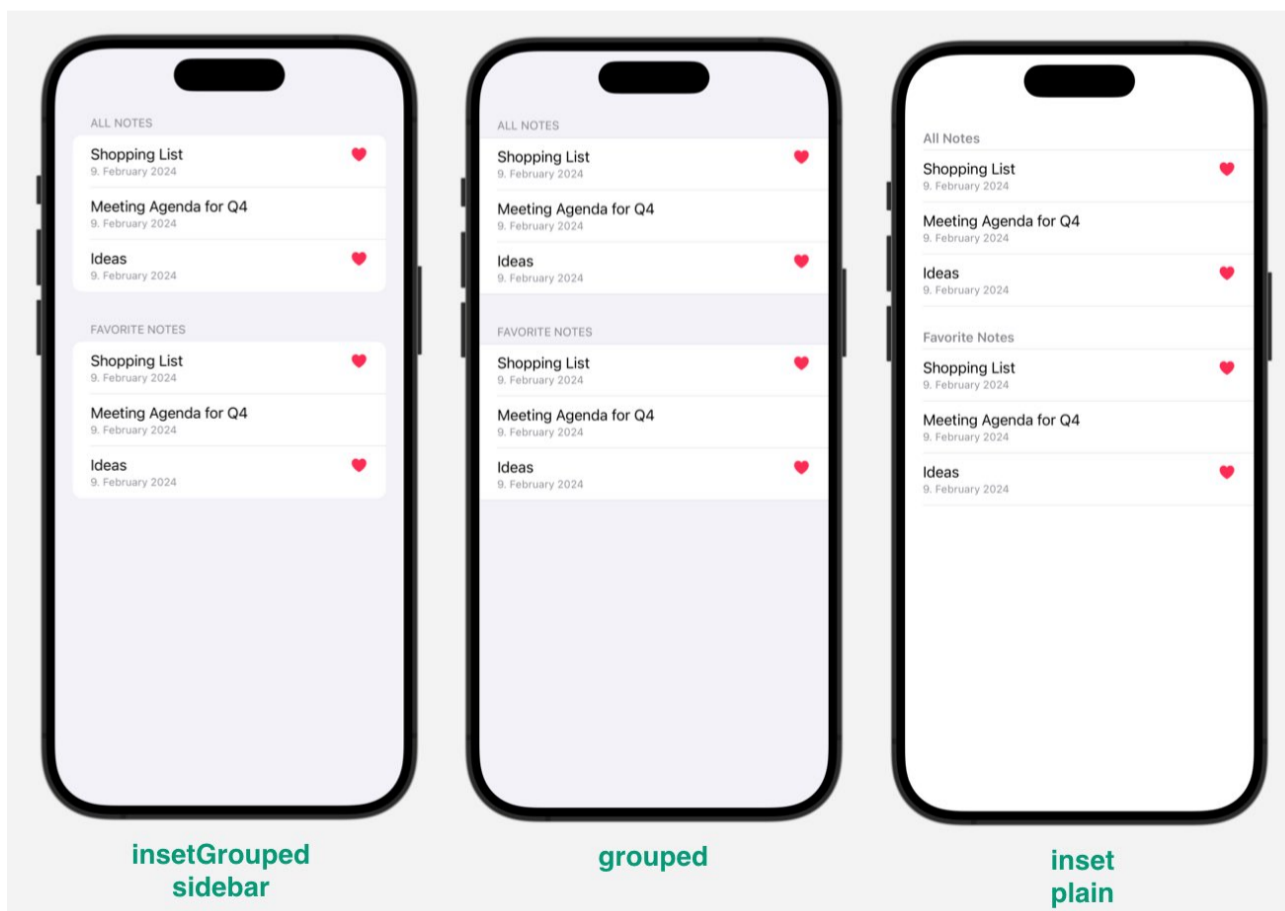
11.3.1 ListStyle

In this lesson, I'm going to demonstrate the default styling options available for SwiftUI lists. We'll use the Notes array and construct a list with sections to show how different styles affect the layout:

```
struct NoteListStyleView: View {
    let notes = Note.examples()
    var body: some View {
        List {
            Section("All Notes") {
                ForEach(notes) {
                    NoteRowView(note: $0)
                }
            }
            Section("Favorite Notes") {
                ForEach(notes) {
                    NoteRowView(note: $0)
                }
            }
        }
    }
}
```

By default, SwiftUI provides a set of predefined styles that you can apply to your lists. Here's how you can apply the default style:

```
List {
    ...
}.listStyle(.plain)
```



Now, let's talk about the different list styles you can apply:

- **Grouped Style:** This style adds some insets and groups the content in visually distinct sections.
- **Inset Grouped Style:** Similar to the grouped style but with additional insets, giving it a more pronounced sectioned appearance.
- **Plain Style:** If you prefer a minimalistic approach, the plain style removes most of the styling, presenting the list items without any separators or backgrounds.

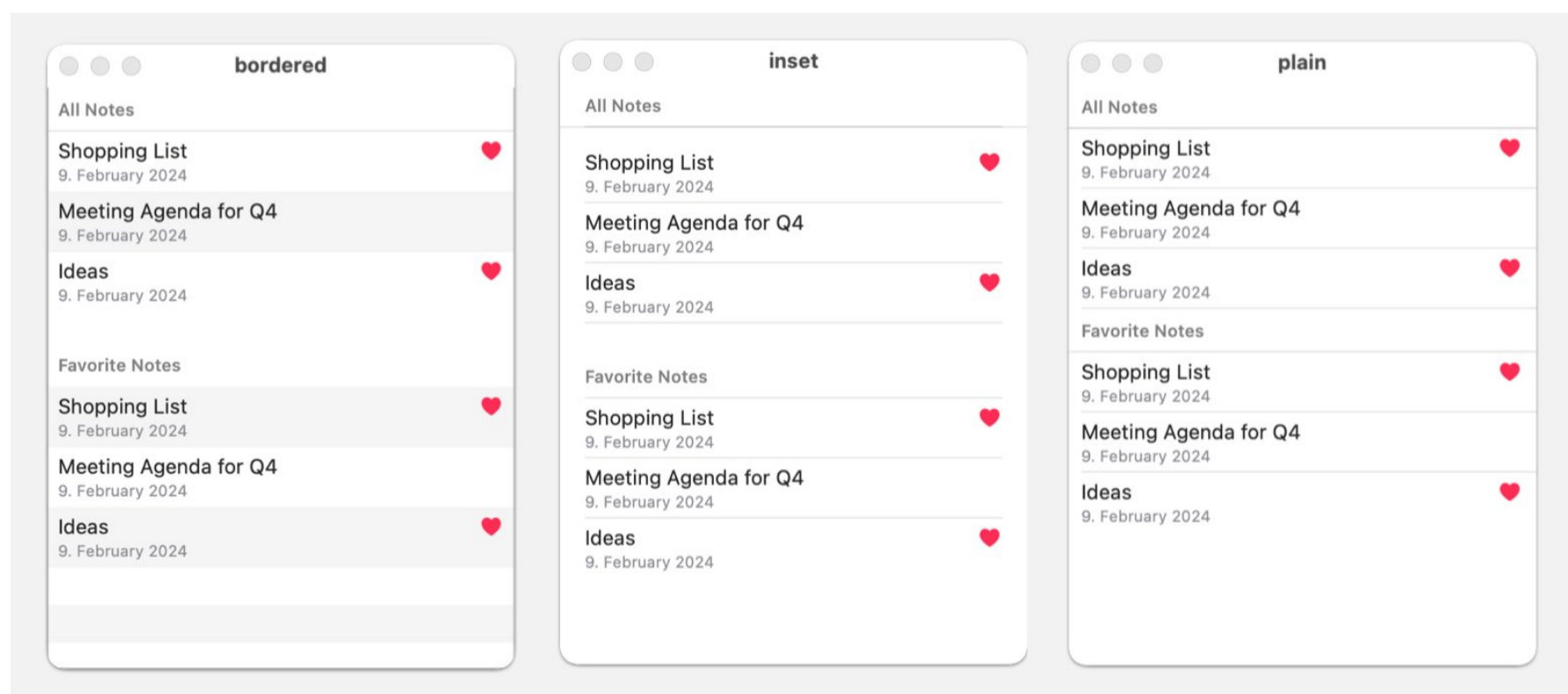
It's worth noting that the default style may vary depending on the environment. For instance, if you place your list inside a `NavigationView` and use it as a sidebar, the style will adapt to the sidebar appearance.

ListStyles for macOS

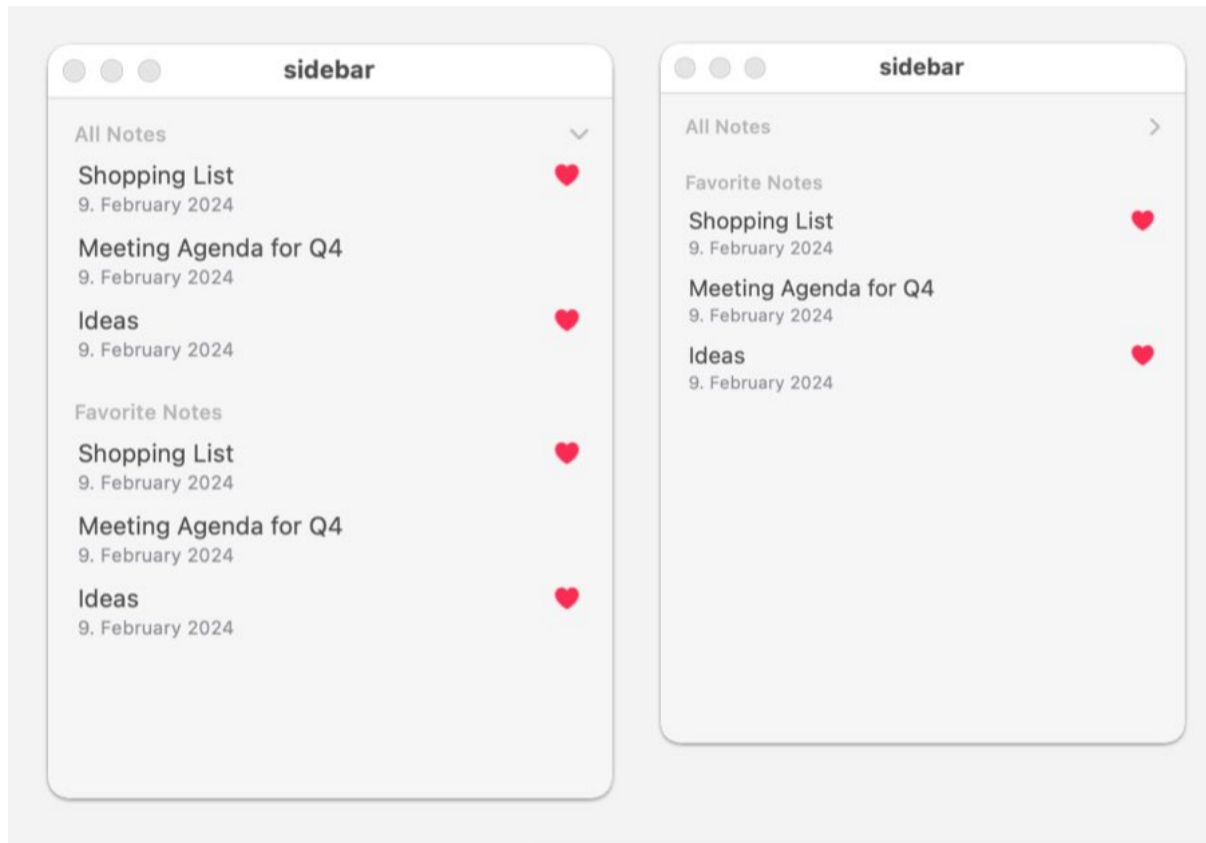
When you're developing for macOS, be aware that some styles, like `InsetGroupedListStyle` and `GroupedListStyle`, are not available.

On macOS, you have additional styles like `BorderedListStyle`, which is not available on iOS. You can also use the `.listRowBackground` modifier to disable the alternating background colors for list rows, which can add a nice touch to your macOS app.

```
List {  
    ...  
}  
.listStyle(.bordered)  
.alternatingRowBackgrounds(.disabled)
```



SidebarListStyle on macOS will add collapsible sections. If you hover with the mouse over the section title a small toggle will appear:



11.3.2 List Row Background

To customise individual list row backgrounds, you can use modifiers such as `.listRowBackground()`. You can attach a color directly to the `ForEach` loop to change the background color of all rows:

```
List {
    Section("All Notes") {
        ForEach(notes) {
            NoteRowView(note: $0)
        }
        .listRowBackground(Color.yellow)
    }
}
```



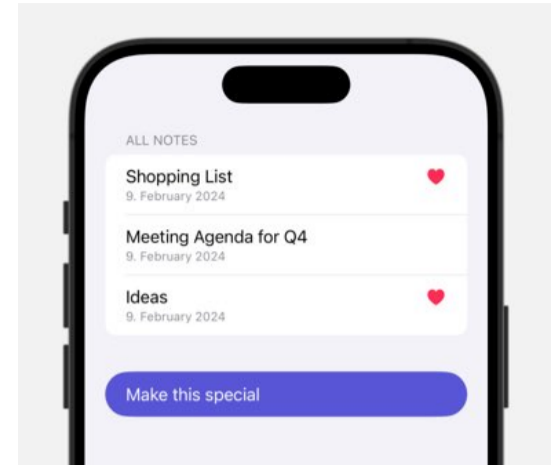
Now, suppose you've added a color property to your data model. In that case, you can use this to dynamically change the background color of each row:

```
List {
    Section("All Notes") {
        ForEach(notes) {
            NoteRowView(note: $0)
                .listRowBackground($0.colorTag)
        }
    }
}
```



Since the background modifier accepts a view, you could use different shapes like `RoundedRectangle`, `Ellipse`, or `Capsule`:

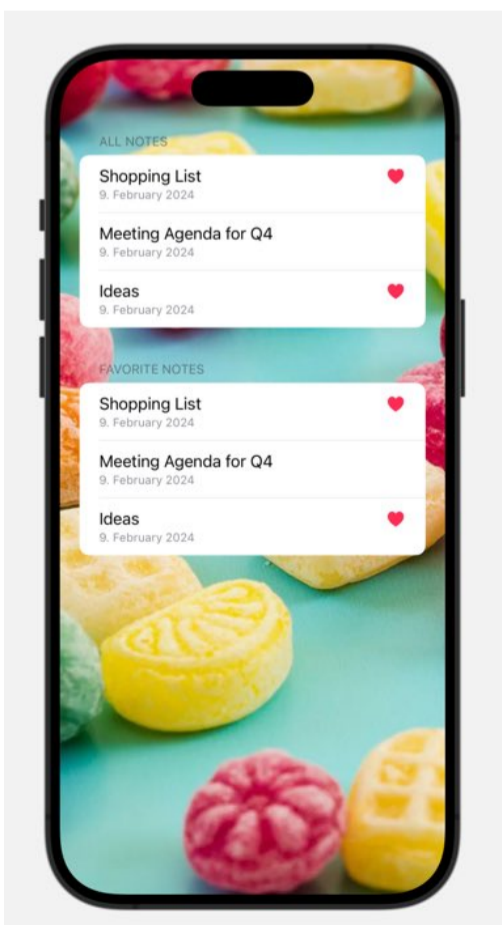
```
List {  
    ...  
    Text("Make this special")  
        .foregroundColor(Color.white)  
        .listRowBackground(  
            Capsule()  
                .fill(Color.indigo)  
        )  
}
```



Customizing the List Background

Now, let's change the background of the entire List. You might want to eliminate the default gray color and add something more vibrant. I previously mentioned using `.scrollContentBackground()` for a `ScrollView`, and it works similarly for a `List`:

```
List {  
    ...  
}  
.scrollContentBackground(.hidden)  
.background(  
    Image("candies")  
        .resizable()  
        .scaledToFill()  
        .ignoresSafeArea()  
)
```



11.3.3 List Row Insets and Separators

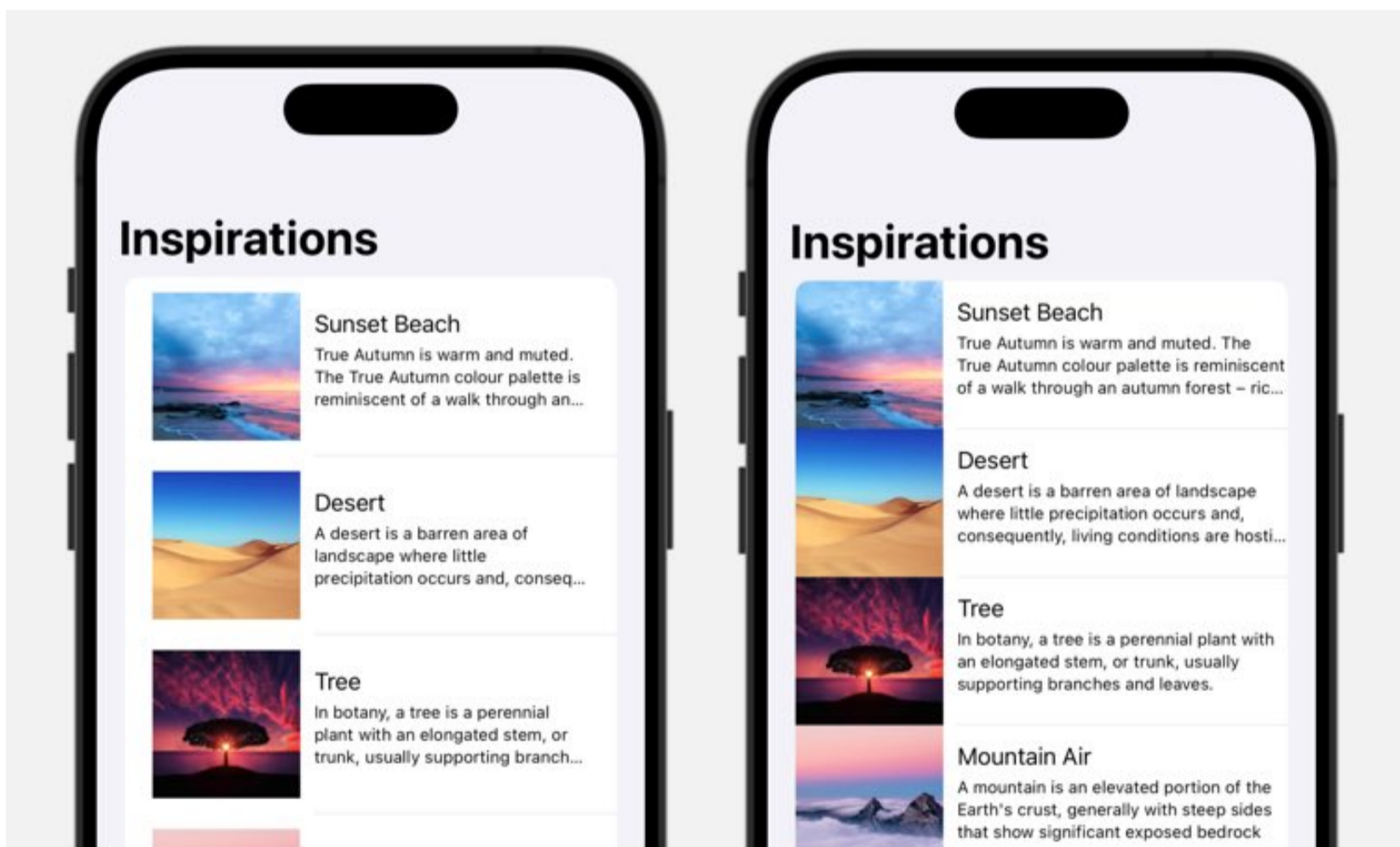
When you're working with SwiftUI's List, you can customise the look and feel of your list rows by adjusting insets and separators.

```
struct InspirationListView: View {
    let inspirations = NatureInspiration.examples()

    var body: some View {
        NavigationStack {
            List(inspirations) { inspiration in
                InspirationRow(inspiration: inspiration)
            }
            .navigationTitle("Inspirations")
        }
    }
}
```

If you want to change the insets of your list rows, you can use the `.listRowInsets` modifier. To demonstrate the effect of this modifier, I'll set the insets to zero:

```
List(inspirations) { inspiration in
    InspirationRow(inspiration: inspiration)
        .listRowInsets(.init(top: 0, leading: 0, bottom: 0, trailing: 0))
}
```



Customising Separators

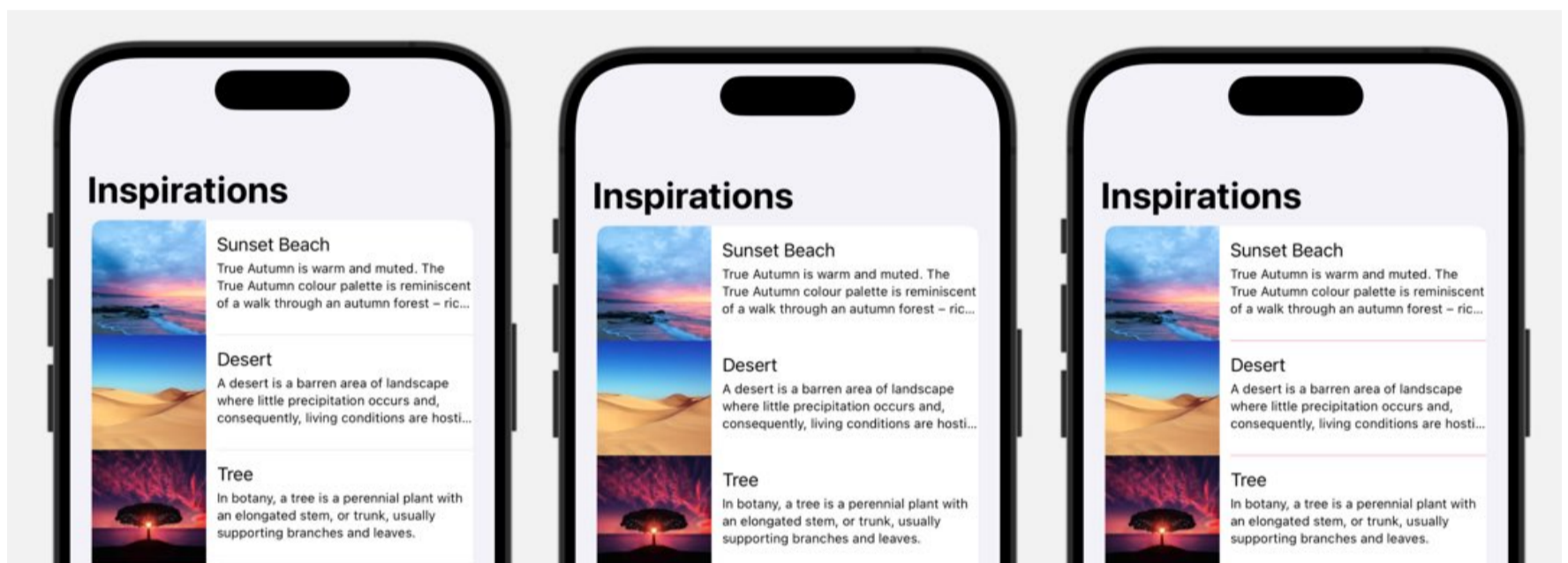
For separators, you have a few options. You can hide them completely:

```
.listRowSeparator(.hidden)
```

Or you can change their color and specify which edges to apply the color to:

```
.listRowSeparatorTint(.pink, edges: .bottom)
```

If you want to apply the color to all edges, you can omit the edges parameter.



Customizing Section Separators

In addition to individual row separators, you can also customize the separators between sections using `.listSectionSeparator`. For instance, if you want to hide the separator for a specific section:

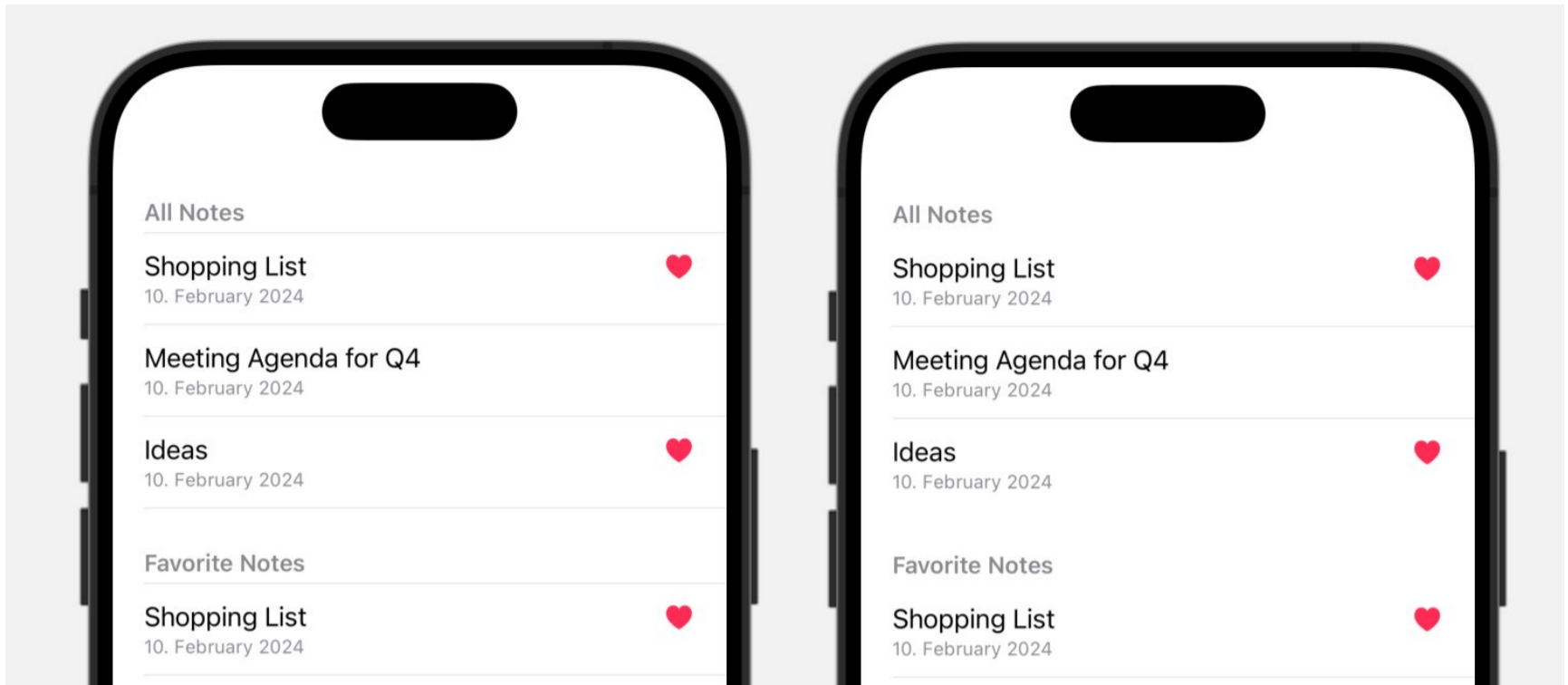
```
struct ContentView: View {
    let notes = Note.examples()

    var body: some View {
        List {
            Section("All Notes") {
                ForEach(notes) { note in
                    NoteRowView(note: note)
                }
            }
            .listSectionSeparator(.hidden)

            Section("Favorite Notes") {
                ForEach(notes) { note in
                    NoteRowView(note: note)
                }
            }
            .listSectionSeparator(.hidden)
        }
    }
}
```

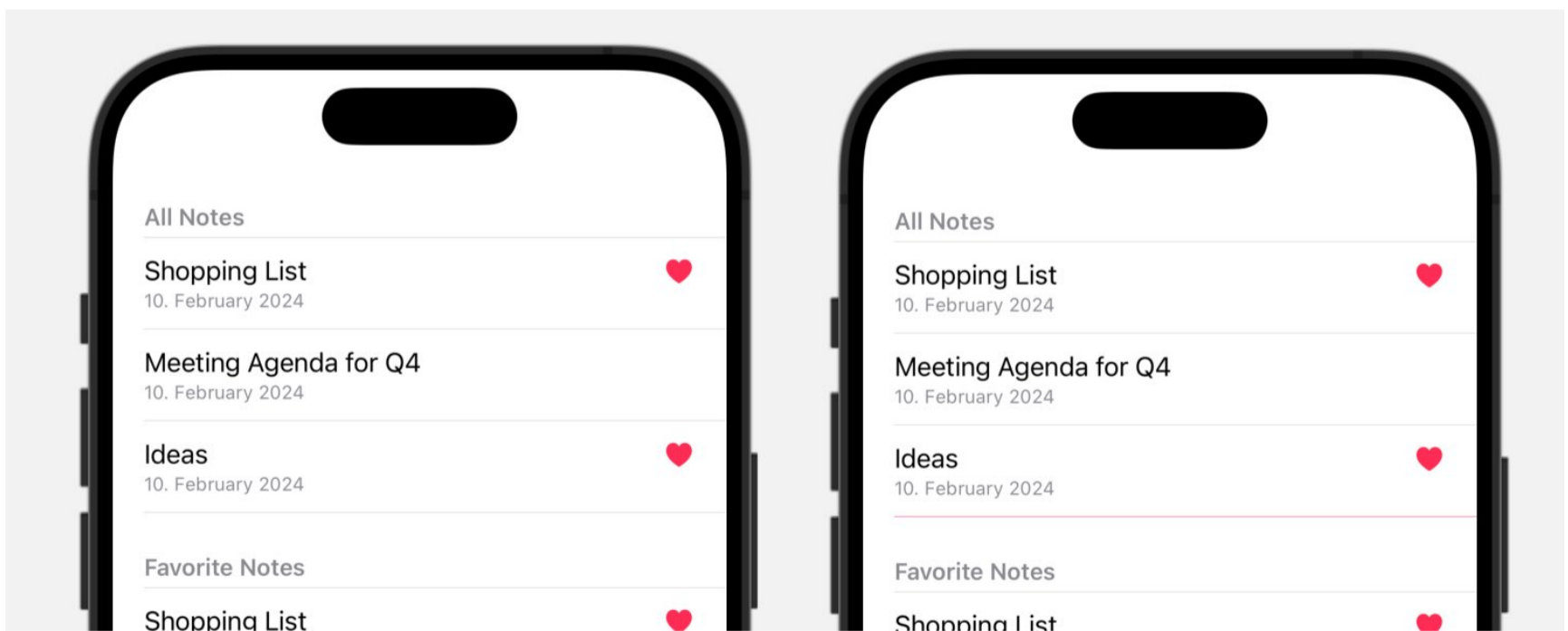


```
}  
    .listStyle(.plain)  
}  
}
```



Or if you want to change the color of a separator at the end of a section:

```
Section("All Notes") {  
    ...  
}  
.listSectionSeparatorTint(.pink)
```



With iOS 17 you can also change the spacing between sections with:

```
.listSectionSpacing(0)
```

11.3.4 Move and Delete

One of the best features of List in SwiftUI is the ability to interact with the items. Implementing functionalities like swipe to delete, moving items, and selection is surprisingly straightforward. Throughout the evolution of iOS, the implementation details have shifted, but I'll walk you through the most current and backward-compatible methods.

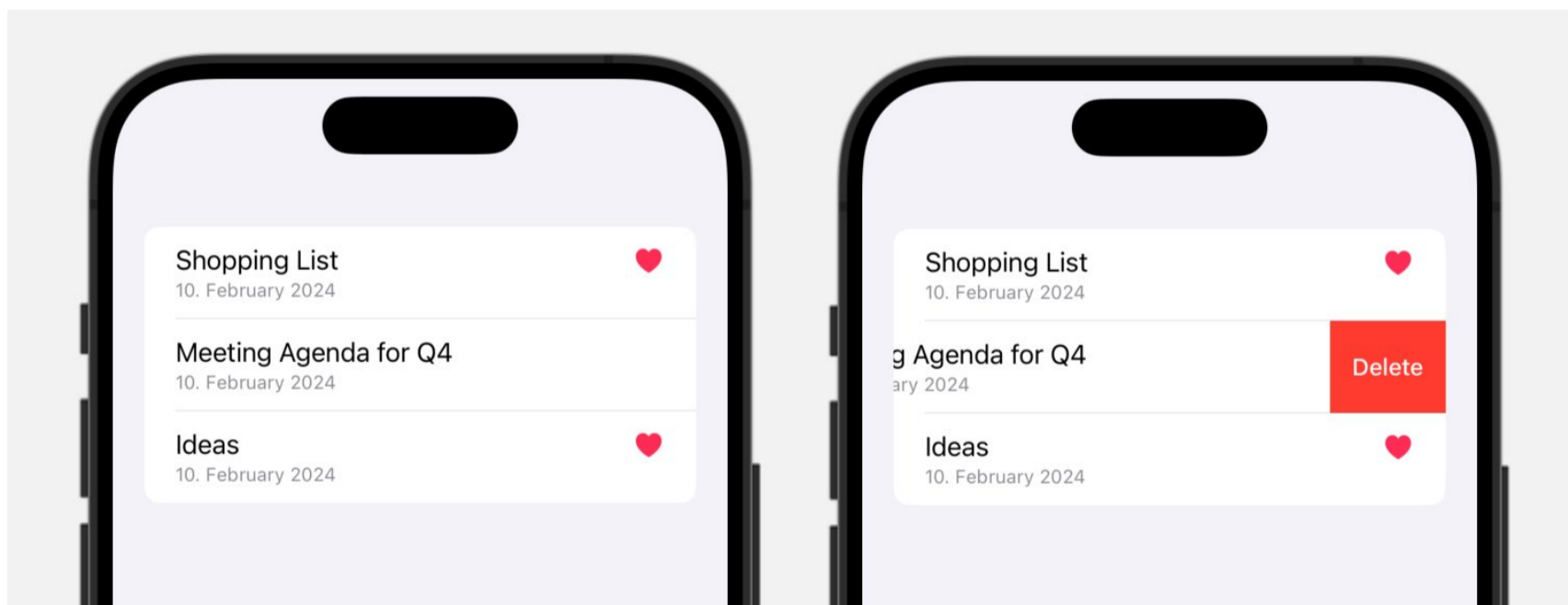
Swipe on Delete with onDelete

For the earliest iOS versions, you attach a modifier to ForEach. Here's how you can implement the deletion with a swipe:

```
struct NoteEditListView: View {
    @State private var notes: [Note] = Note.examples()
    var body: some View {
        List {
            ForEach(notes) { note in
                NoteRowView(note: note)
            }
            .onDelete(perform: delete)
        }
    }

    func delete(at offset: IndexSet) {
        notes.remove(atOffsets: offset)
    }
}
```

When you attach `.onDelete`, you receive an `IndexSet` which tells you which item to delete. I create a delete function that takes in the `IndexSet`: delete function that takes in the `IndexSet`.



Reordering Lists with onMove

The `.onMove` modifier allows users to reorder items in a list. You can use it similarly to `.onDelete`:

```
ForEach(notes) { note in
    ...
}
```

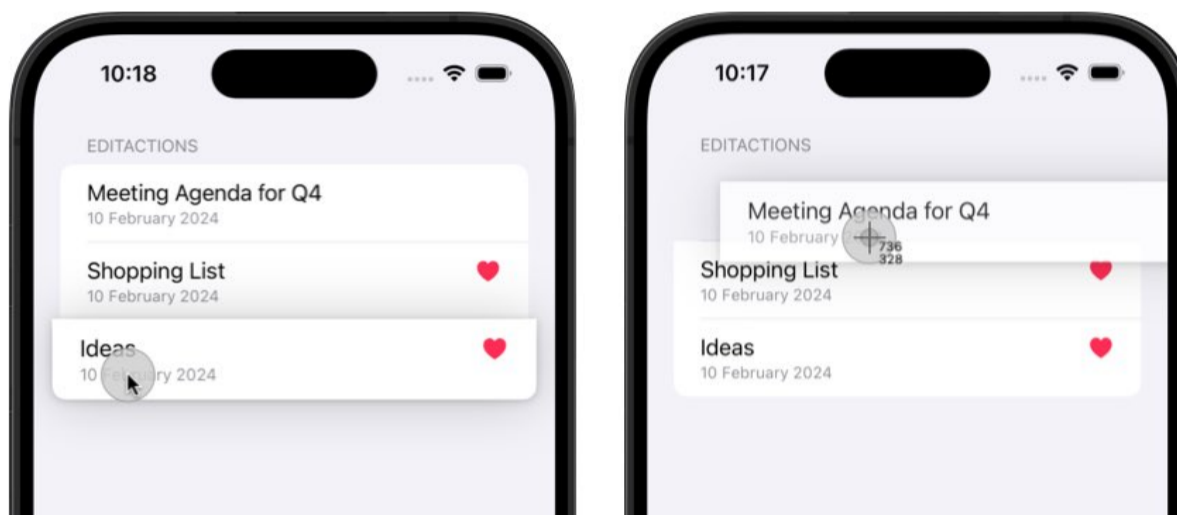
```

}
.onMove(perform: move)

func move(from source: IndexSet, to destination: Int) {
    notes.move(fromOffsets: source, toOffset: destination)
}

```

Since iOS 16, you can reorder items directly without entering edit mode.

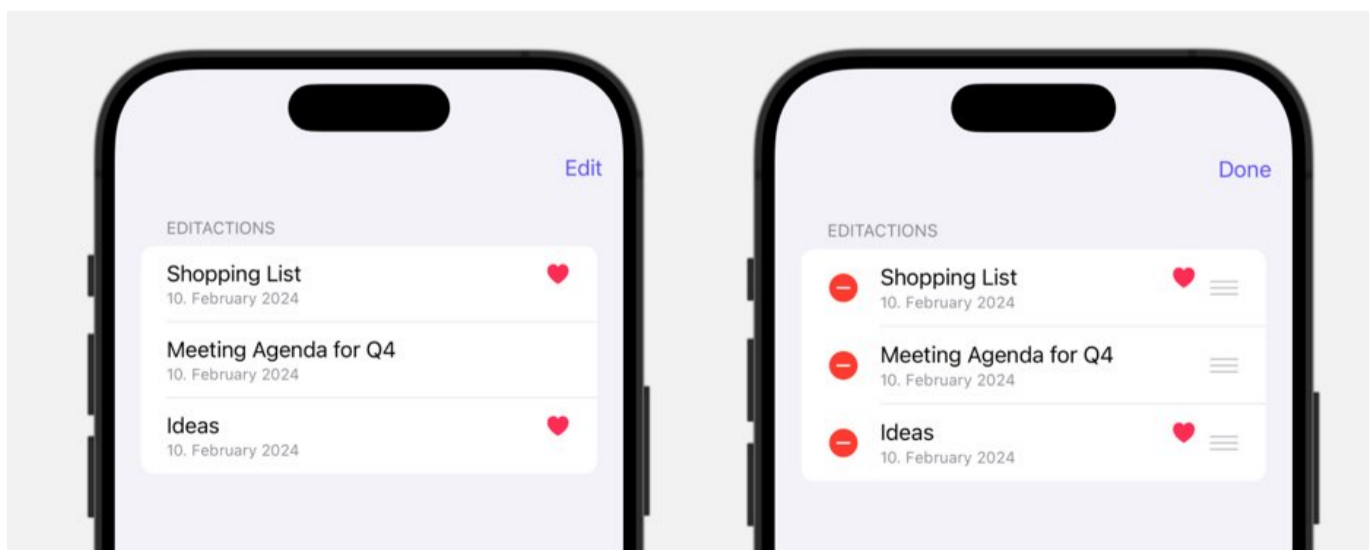


For iOS 15 and below, you need to use an edit button. In edit mode, the user can use drag indicators to reorder the list.

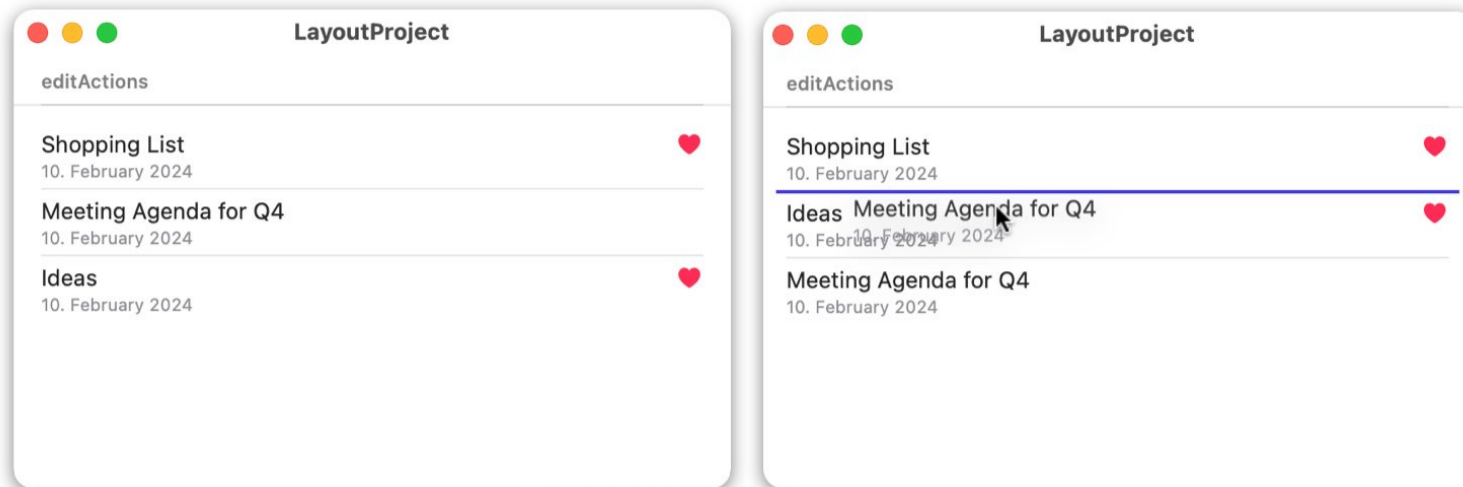
```

NavigationStack {
    List {
        Section("All Notes") {
            ForEach(notes) { note in
                NoteRowView(note: note)
            }
        }
        .onMove(perform: move)
        .onDelete(perform: delete)
    }
    .toolbar(content: {
        EditButton()
    })
}

```



On macOS, you can also use `onMove`, but swipe actions are not available:



EditActions

Since iOS 16 and macOS 13, you can replace `onDelete` and `onMove` with an easier solution. You can use the **`editActions`** argument of `ForEach` like so:

```
struct NoteEditListView: View {
    @State private var notes: [Note] = Note.examples()

    var body: some View {
        List {
            Section("Notes") {
                ForEach($notes, editActions: [.move, .delete]) { $note in
                    NoteRowView(note: note)
                }
            }
        }
    }
}
```

swipeAction

If you're targeting iOS 15 or macOS 12 and above, you have the option to use **`.swipeActions`**. You can add multiple custom buttons that are displayed on the **leading and trailing edges**.

In the following, a favorite and send button are shown when the user swipes from the leading edge:

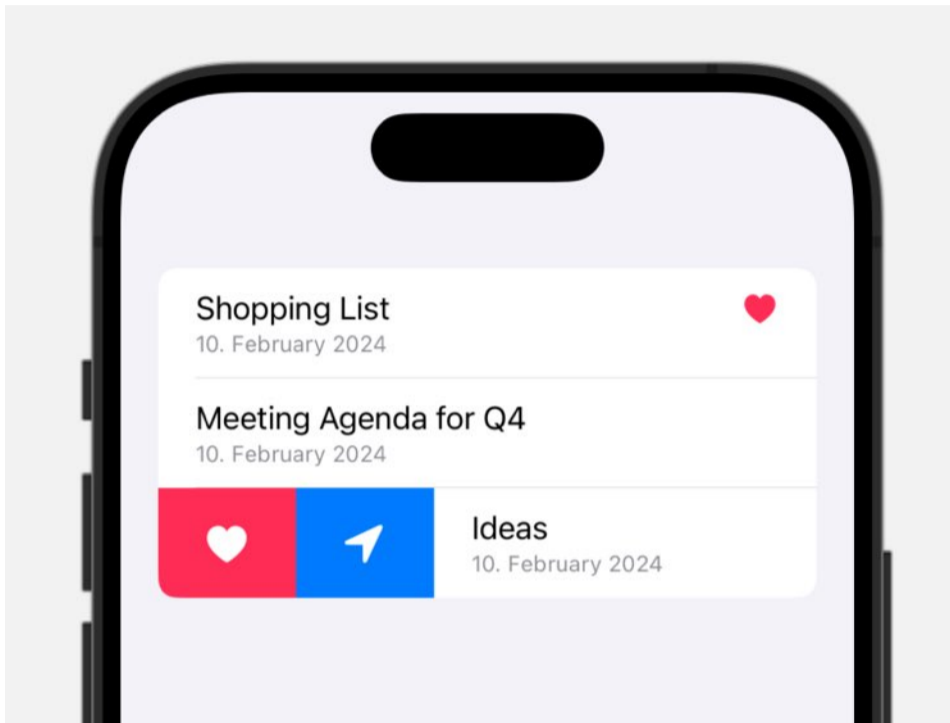
```
List {
    ForEach(notes) { note in
        NoteRowView(note: note)
        .swipeActions(edge: .leading) {
            Button(action: {
                note.isFavorite.toggle()
            }, label: {
                Label("Favourite", systemImage: "heart.fill")
            })
        }
    }
}
```

```

        .tint(.pink)

        Button(action: {
        }, label: {
            Label("Send", systemImage: "location.fill")
        })
        .tint(.blue)
    }
}
}

```



If you set `allowsFullSwipe` to `true`, a full swipe will automatically trigger the button action.

```

.swipeActions(edge: .leading, allowsFullSwipe: true) {
    ...
}

```

You could also add a custom delete button to the trailing edge if you want to override the default implementation.

Special Considerations for macOS

On macOS, swipe actions aren't the norm. Instead, context menus are used. Here's how you can add a context menu for deletion:

```

ForEach(notes) { note in
    NoteRowView(note: note)
        .contextMenu(ContextMenu(menuItems: {
            Button(action: {
                delete(note: note)
            }, label: {

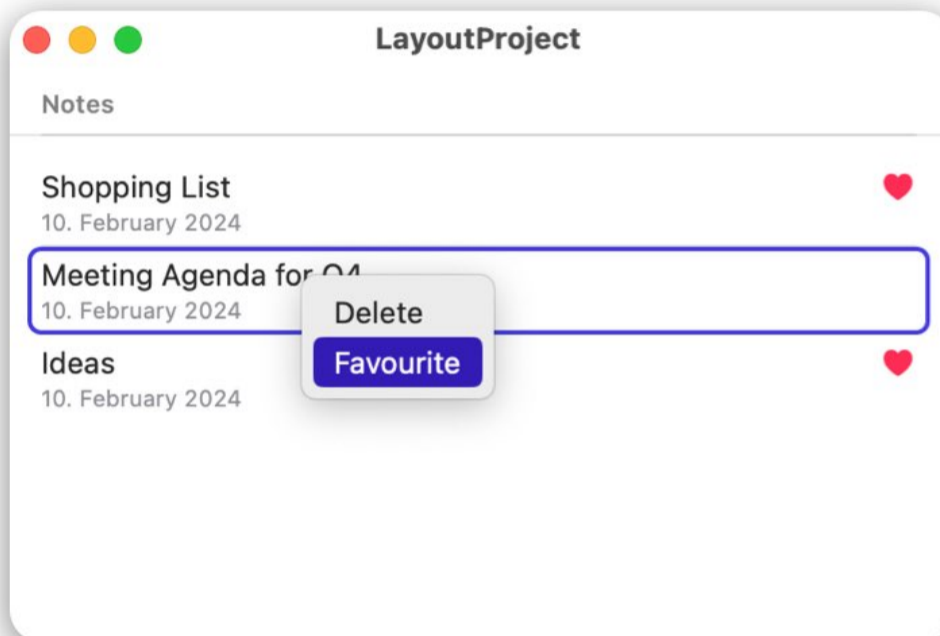
```

```

        Label("Delete", systemImage: "trash")
    })
    Button(action: {
        note.isFavorite.toggle()
    }, label: {
        Label("Favourite", systemImage: "heart.fill")
    })
    })
}

```

This way, right-clicking on an item will present a menu for the deletion or toggling the favorite state.



11.3.5 List Selection

In this section, I'm going to show you how to select items in a SwiftUI List. You'll learn how to handle single and multiple selections, and we'll delve into more complex data handling scenarios. This knowledge will be particularly useful when building an app with a sidebar for folder selection, which then displays a list of notes and a detail view for the selected note. But first, let's tackle the basics of selection.

Creating a Note Selection List View

Imagine you have a collection of notes. You want to be able to select a note from a list. To start simple, let's use some example data:

```

struct NoteSelectionListView: View {
    let notes: [Note]
    @State private var selectedNote: Note? = nil

    var body: some View {
        List(notes, selection: $selectedNote) { note in
            NoteRowView(note: note)
                .tag(note)
        }
    }
}

```

```
}  
}
```

Here, **selectedNote** is an optional Note because there might be situations where no note is selected, so we initialize it to nil.

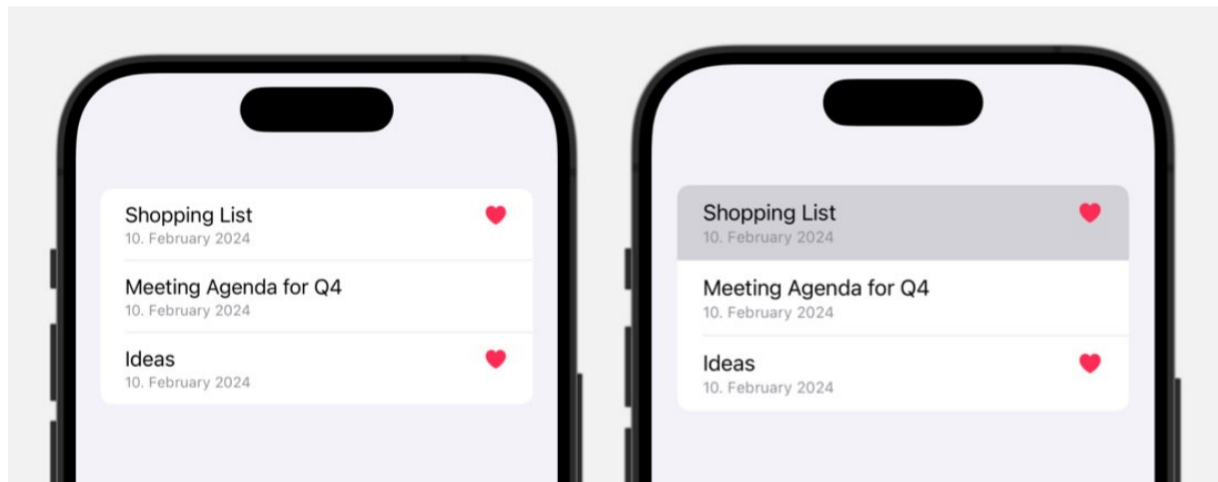
To create a list with selectable items, you use the List initializer that takes a selection parameter.

For this to work, your Note model must conform to Hashable. If you encounter an error, you'll need to override the hash(into:) function to use a unique property, like id.

Handling Selection Types

When you tap on an item in the list, you might notice that nothing happens. This is because the list uses the id property for selection by default, but we're passing a Note object to the selection binding. To resolve this, you can use the .tag modifier on each row to specify that the selection should be based on the Note object itself:

```
NoteRowView(note: note)  
    .tag(note)
```



Multiple Selections

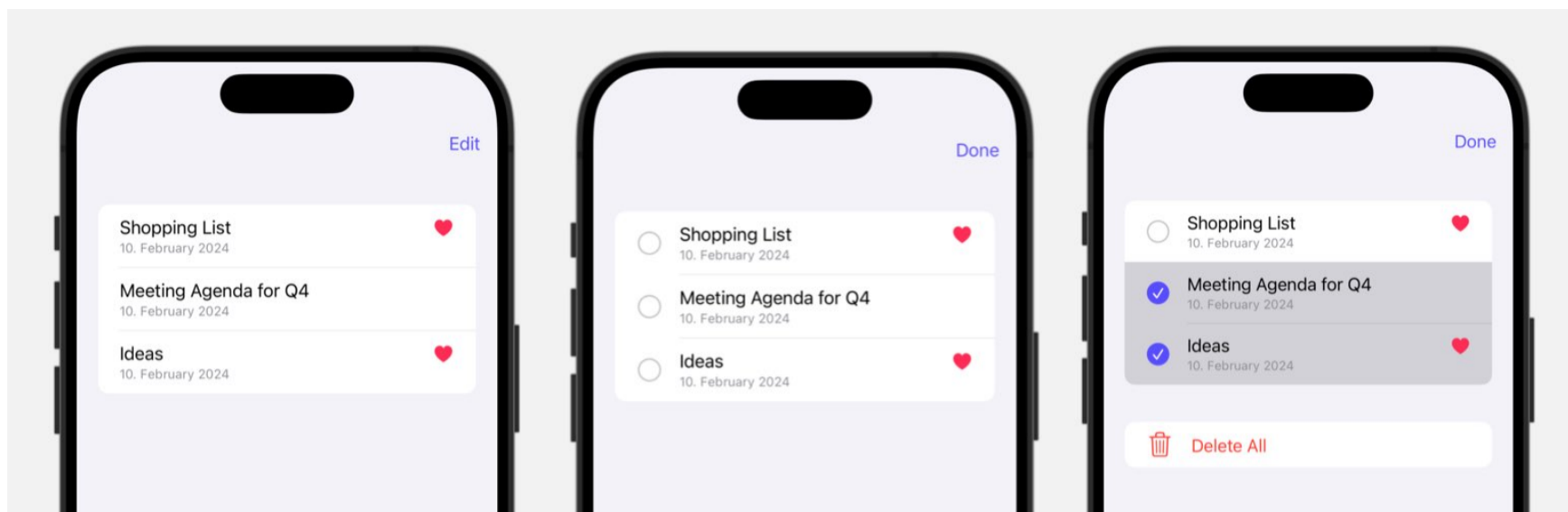
To handle multiple selections, you need to change your selection state to a Set:

```
@State private var selectedNote: Set<Note> = []
```

By default, a List allows only single selection. To enable multiple selections, you must enter edit mode. You can add an edit button to your navigation bar:

```
.toolbar(content: {  
    EditButton()  
})
```

In edit mode, the list will show checkboxes, allowing you to select multiple notes.



Additionally, I am showing a “Delete All” button if the user is in edit mode and has selected one or more notes. I am using the environment value for EditMode to check this conditionally:

```
List(selection: $selectedNote) {
    ForEach(notes){ note in
        NoteRowView(note: note)
            .tag(note)
    }
    #if os(iOS)
    EditListSubView(hasSelectedNotes: !selectedNote.isEmpty)
    #endif
}
```

I am using a subview to properly access the environment value inside the list:

```
struct EditListSubView: View {
    @Environment(\.editMode) var editMode
    let hasSelectedNotes: Bool

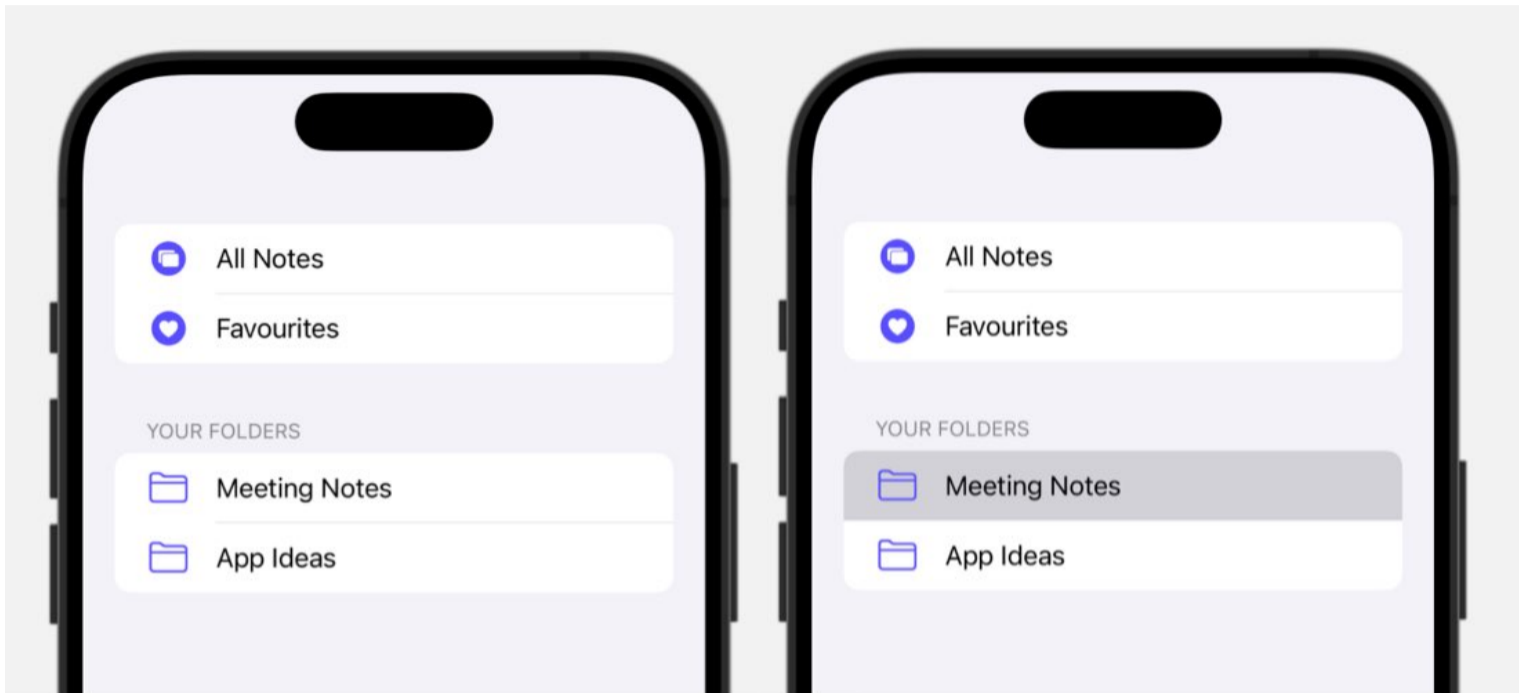
    var body: some View {

        if editMode?.wrappedValue.isEditing == true && hasSelectedNotes {
            Section {
                Button(role: .destructive,
                    action: {

                }, label: {
                    Label("Delete All", systemImage: "trash")
                })
                .foregroundColor(Color.red)
            }
        }
    }
}
```


Advanced Selection Handling

I am showing you another example where we want to select from a list of Folder objects. Additionally, I want to allow the user to select options like “All Notes” or “Favorites”. For this to work, you’ll need to rethink your data structure.



A common approach is to use an enumeration to encapsulate all selection types:

```
enum FolderSelectionType: Hashable {
    case all
    case favourites
    case folder(Folder)

    var displayName: String {
        switch self {
            case .all:
                return "All Notes"
            case .favourites:
                return "Favourite Notes"
            case .folder(let folder):
                return folder.name
        }
    }
}
```

With this enum, you can handle all selection cases with a single type. You then bind this enum to your list selection:

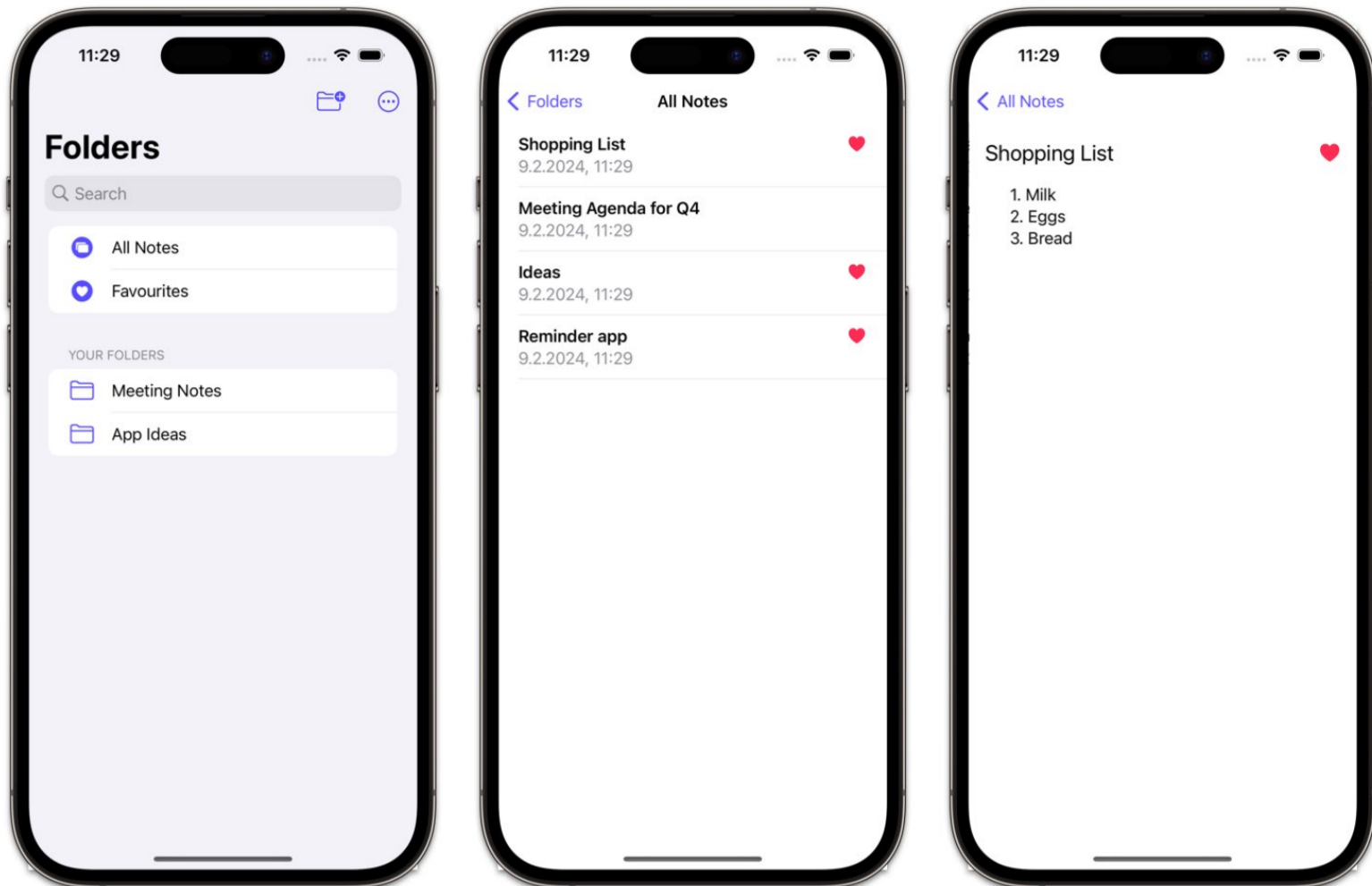
```
struct FolderSelectionListView: View {

    @State private var folders: [Folder] = Folder.examples()
    @State private var selection: FolderSelectionType? = nil

    var body: some View {
        List(selection: $selection) {
            Section {
                Label("All Notes",
```


You might have noticed that the two lists, the sidebar, and the content view, have different styles. On macOS, the sidebar uses a specific sidebar styling, while the content view uses an inset styling. The good news is that SwiftUI automatically adapts these styles for iOS as well.

On iOS, the sidebar styling looks slightly different, but the functionality remains the same. When you select a note, it appears in the content view, and you can navigate to the detail view from there.



Solution Overview

Let's take a look at the solution I have prepared for you. I have reused most of the views we previously worked on, such as the `FolderSelectionListView`. To make it work in this app, I needed to pass the necessary data between the top-level view "NavigationListView" and all subviews.

I added the folder list selection view to the sidebar. This allows us to select a folder. In the content view, I used the selected folder type to display the corresponding notes in a list view:

```
struct NavigationListView: View {

    @State private var folders = Folder.examples()
    @State private var folderSelection: FolderSelectionType? = nil
    @State private var selectedNote: Note? = nil

    var body: some View {
        NavigationSplitView {
            FolderSelectionListView(folders: $folders,
                                   selection: $folderSelection.animation())
                .navigationTitle("Folders")
        } content: {
            if let folderSelection {
                NoteSelectionListView(notes: notes(),
```

```

                selectedNote: $selectedNote)
            .navigationTitle(folderSelection.displayName)
        } else {
            ContentUnavailableView("Please select a folder",
                systemImage: "folder")
        }
    } detail: {
        if let selectedNote {
            NoteEditDetailView(note: selectedNote)
        } else {
            ContentUnavailableView("Please select a note",
                systemImage: "note")
        }
    }
}
}
}
}

```

The folder selection list view now takes two bindings: one for the folders and another for the folder selection type. This allows us to update the selected folder and the type of notes to display.

```

struct FolderSelectionListView: View {
    @Binding var folders: [Folder]
    @Binding var selection: FolderSelectionType?

    var body: some View {
        List(selection: $selection) {
            Section {
                ""
            }

            Section("Your Folders") {
                ""
            }
        }
    }
}

```

To make the preview work properly with the bindings, I made some changes. Inside the preview macro, I added another struct to hold the state for the folder and selection type. This way, you can test the selection in the preview.

```

#Preview {
    struct PreviewFolderSelectionListView: View {
        @State private var folders = Folder.examples()
        @State private var selectionType: FolderSelectionType? = nil

        var body: some View {
            FolderSelectionListView(folders: $folders,
                selection: $selectionType)
        }
    }

    return PreviewFolderSelectionListView()
}

```

I followed a similar approach for the node selection view. I created a selected note binding and passed it to the notes. Again, I used an extra struct in the preview to hold some state and example data.

```

struct NoteSelectionListView: View {
    let notes: [Note]
    @Binding var selectedNote: Note?

    var body: some View {
        List(notes, selection: $selectedNote) { note in
            NoteRowView(note: note)
                .tag(note)
        }
    }
}

#Preview("Single Selection") {
    struct PreviewNoteSelectionListView: View {
        let notes: [Note] = Note.examples()
        @State private var selectedNote: Note? = nil
        var body: some View {
            NoteSelectionListView(notes: notes,
                                  selectedNote: $selectedNote)
        }
    }

    return PreviewNoteSelectionListView()
}

```

Showing the Correct Notes in the ContentView

Depending on the selection in the sidebar, I need to show the corresponding notes in the content view of the NavigationSplitView:

```

NavigationSplitView {
    ...
} content: {
    if let folderSelection {
        NoteSelectionListView(notes: notes(),
                              selectedNote: $selectedNote)
    } else {
        ContentUnavailableView("Please select a folder", systemImage: "folder")
    }
} detail: {
    ...
}

```

To handle the different selection scenarios, I used a function that checks for the selection type. If it's set to "favorite," I display only the favorite notes. Otherwise, if a folder is selected, I pass the nodes from that folder to the **NoteSelectionListView**.

```

func notes() -> [Note] {
    switch folderSelection {
    case .all:
        return allNotes
    case .favourites:
        return favoriteNotes
    case .folder(let folder):
        return folder.notes
    }
}

```

```

        case nil:
            return []
    }
}

var allNotes: [Note] {
    folders.flatMap { folder in
        folder.notes
    }
}

var favoriteNotes: [Note] {
    folders.flatMap { folder in
        folder.notes.filter { $0.isFavorite }
    }
}

```

The list view doesn't care about the specific data it receives. This approach also applies to sorting. You can modify the notes function to include sorting based on specific properties.

Animations

To achieve smooth animations between different folder selections, you can add an animation to the property that triggers the changes. By specifying an animation, the views will animate nicely when items are added or removed.

```

NavigationView {
    FolderSelectionListView(folders: $folders,
                           selection: $folderSelection.animation())
}

```

It's important to note that using one list and passing different data is a better solution for animations. If you switch between completely different views, the default fade-in and out animation will be used.

Further Customisations

Although I didn't add much interaction in this example, you can extend the app by adding context menus on macOS or swipe gestures on iOS. Additionally, you can customise the styling of the lists to fit your preferences.

In an upcoming lesson, we will explore tables and replace the middle section with a table. We will also add a toggle to switch between grid and list layouts, allowing the user to choose their preferred layout.

Take inspiration from other apps to see how they utilize lists and explore more advanced filtering options. While our example keeps things simple, other apps may implement more complex filtering using Core Data or Swift Data.

11.4 STRUCTURING LISTS

In this lesson, I'll guide you through structuring your lists in SwiftUI by utilizing subviews such as sections and disclosure groups. You'll learn how to customize headers, add footers, and create collapsible sections. This will be particularly useful if you have large and complex data to display.

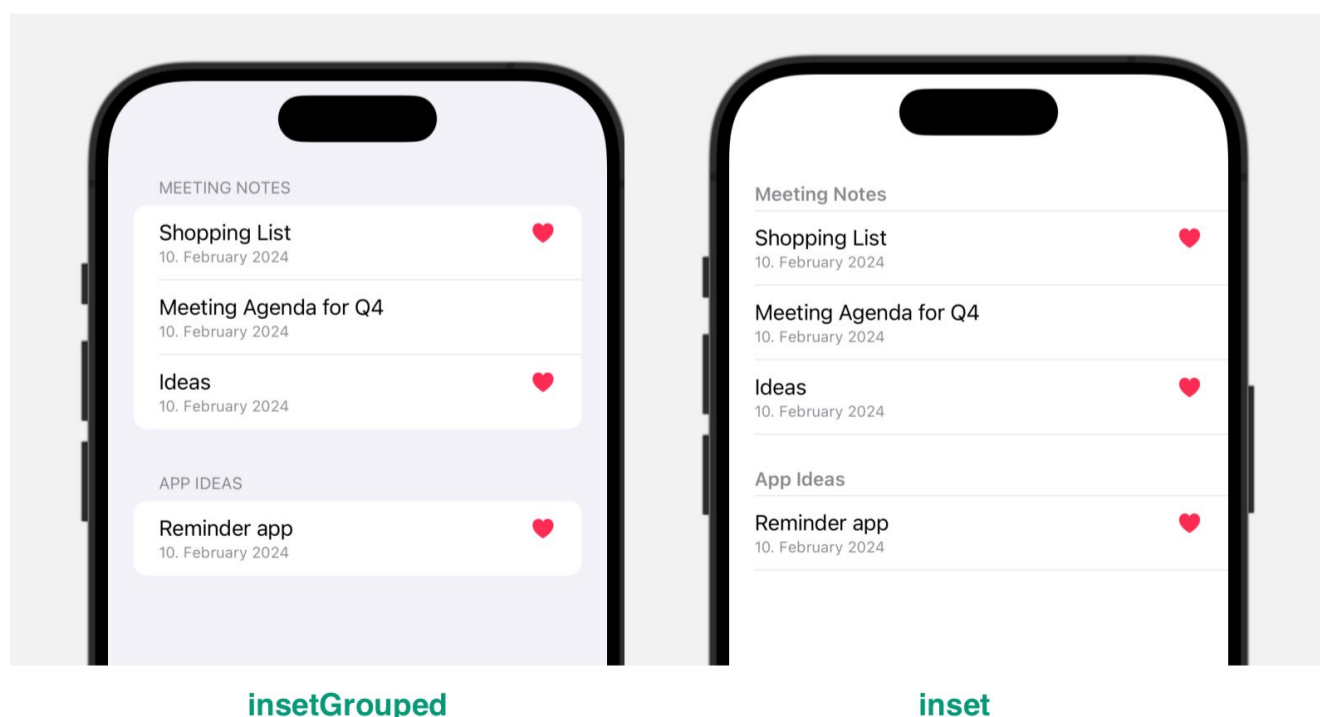
11.4.1 Sections

First, I'll demonstrate how to create a new SwiftUI view that structures a list with sections. Imagine you have a note-taking app with different folders, each containing a list of notes:

```
@Observable class Folder: Identifiable {
    let id: UUID
    let creationDate: Date
    var name: String
    var notes: [Note]
    ...
}
```

You can represent each folder with a section in your list.

```
struct SectionExampleView: View {
    let folders = Folder.examples()
    var body: some View {
        List {
            ForEach(folders) { folder in
                Section(folder.name) {
                    ForEach(folder.notes) { note in
                        NoteRowView(note: note)
                    }
                }
            }
        }
    }
}
```



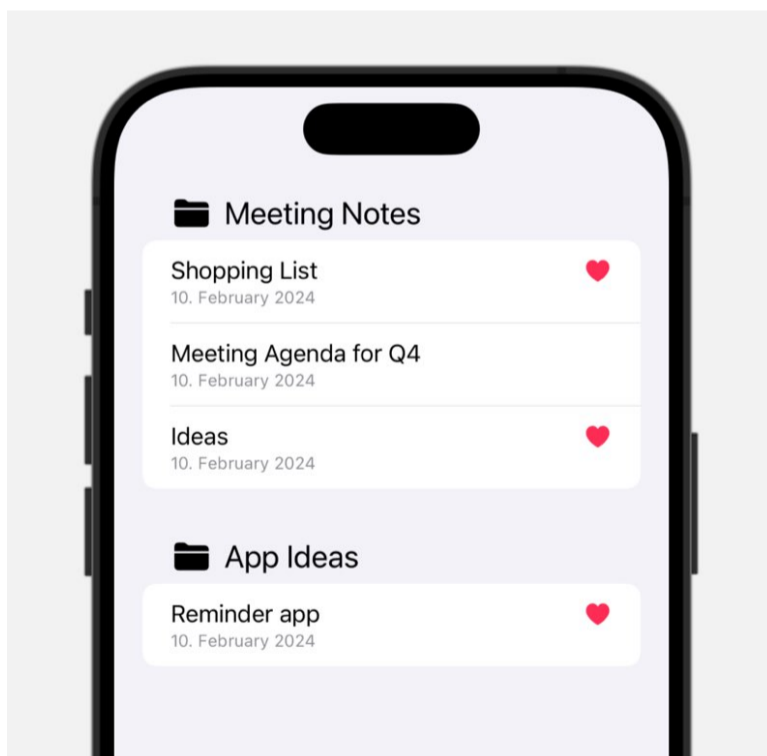
Customizing Section Headers and Footers

Now, let's dive into customization. Say you dislike the default capitalized section headers. You can change this by using a header closure instead of a simple string.

```
Section {
    ForEach(folder.notes) { note in
        NoteRowView(note: note)
    }
} header: {
    Text(folder.name)
        .textCase(.none)
}
```

You can also use other views than Text like Table. In the following, I changed to font size to large and used black as the foreground color:

```
Section {
    ...
} header: {
    Label(folder.name, systemImage: "folder.fill")
        .textCase(.none)
        .font(.title2)
        .foregroundColor(.black)
}
```



You can embed additional views in your section headers. For instance, adding a button within an HStack:

```
Section {
    ...
} header: {
    HStack {
```

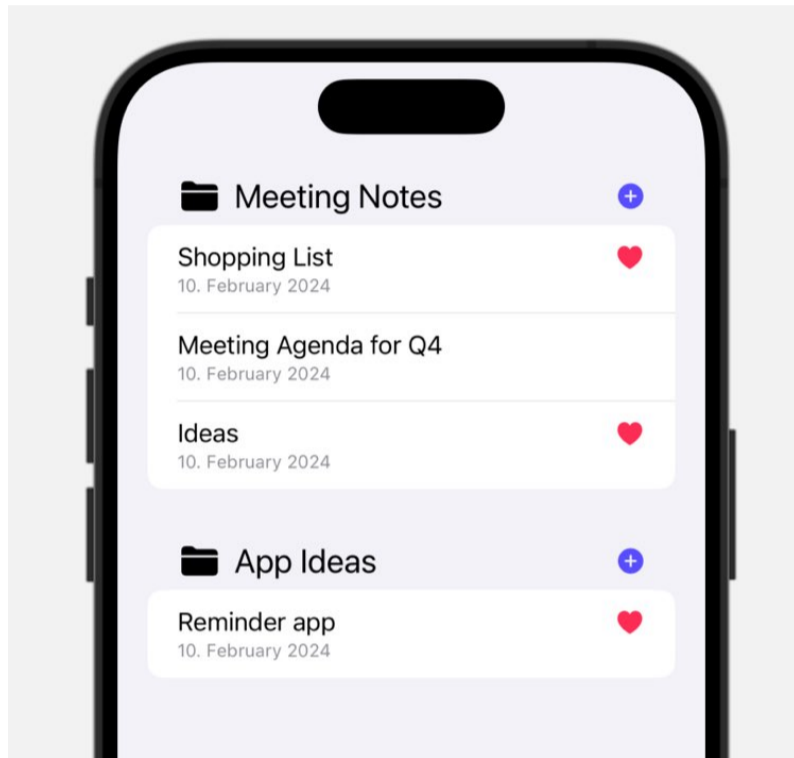


```

Label(folder.name, systemImage: "folder.fill")
    .textCase(.none)
    .font(.title2)
    .foregroundColor(.black)

Spacer()
Button {
    } label: {
        Label("Add New Note", systemImage: "plus.circle.fill")
    }
    .labelStyle(.iconOnly)
}
}

```

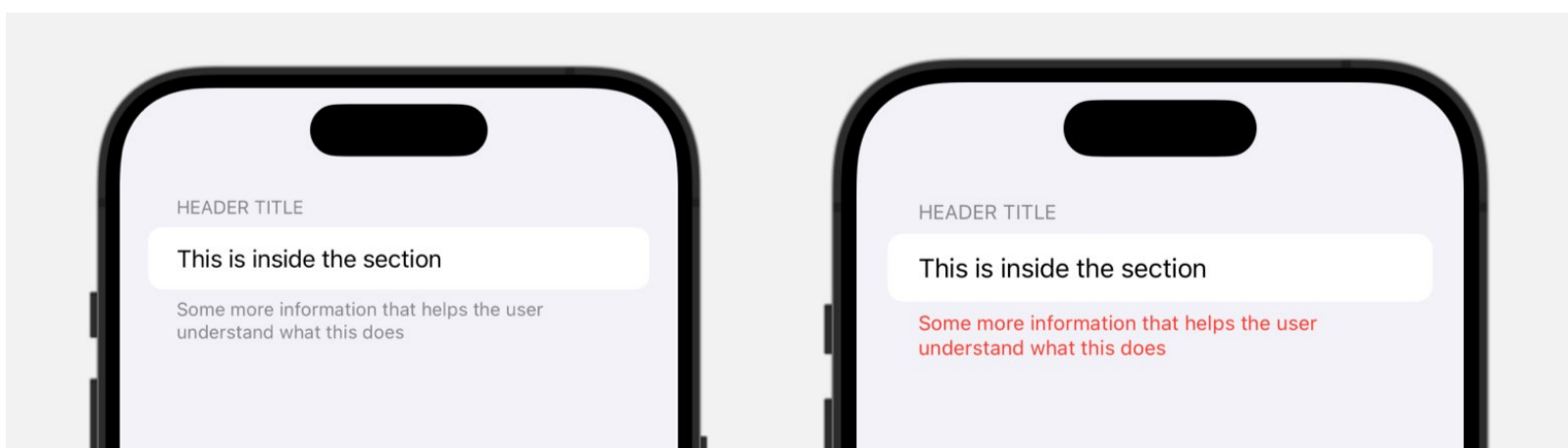


Moreover, you can include footers to provide additional information to the user:

```

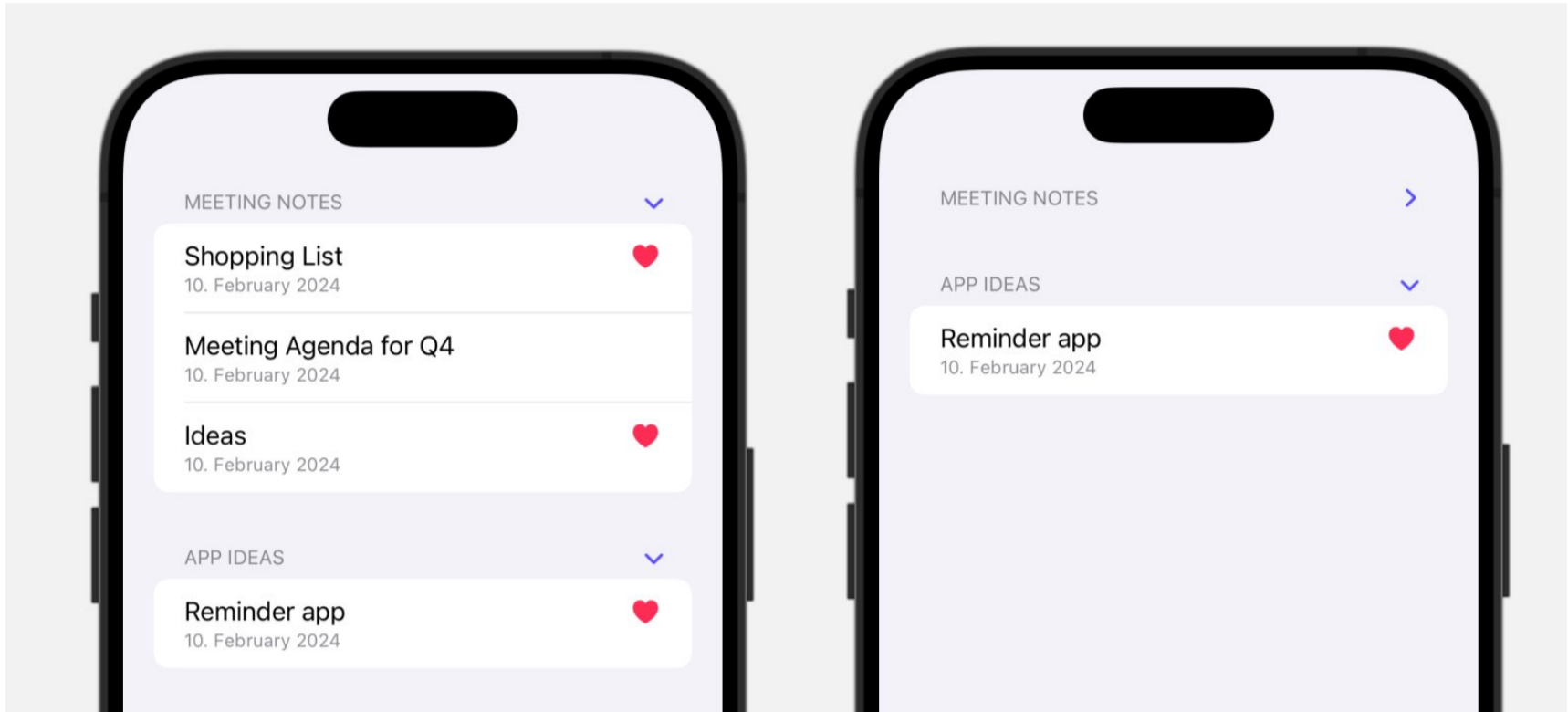
Section {
    Text("This is inside the section")
} header: {
    Text("Header Title")
} footer: {
    Text("Some more information that helps the user understand what this does")
    .foregroundColor(.red)
}

```



11.4.2 Collapsible Sections

In previous lessons, I showed you how sections display all items by default. However, when dealing with numerous sections, users might want to collapse some to manage screen real estate better. Luckily, SwiftUI provides straightforward methods to create collapsible sections. Let's walk through the process together.



I'll reuse the folder and note examples to keep things consistent. The key to creating collapsible sections is to use a specific initializer for **Section** that allows us to control the **expanded state**.

Because I need to keep this state for each section, I am creating a subview for each section that holds this state:

```
struct FolderSectionView: View {  
  
    let folder: Folder  
    @State private var isExpanded: Bool = true  
  
    var body: some View {  
        Section(folder.name,  
                isExpanded: $isExpanded) {  
            ForEach(folder.notes) { note in  
                NoteRowView(note: note)  
            }  
        }  
    }  
}
```

I can then use this subview for the List view:

```
List {  
    ForEach(folders) { folder in  
        FolderSectionView(folder: folder)  
    }  
}
```

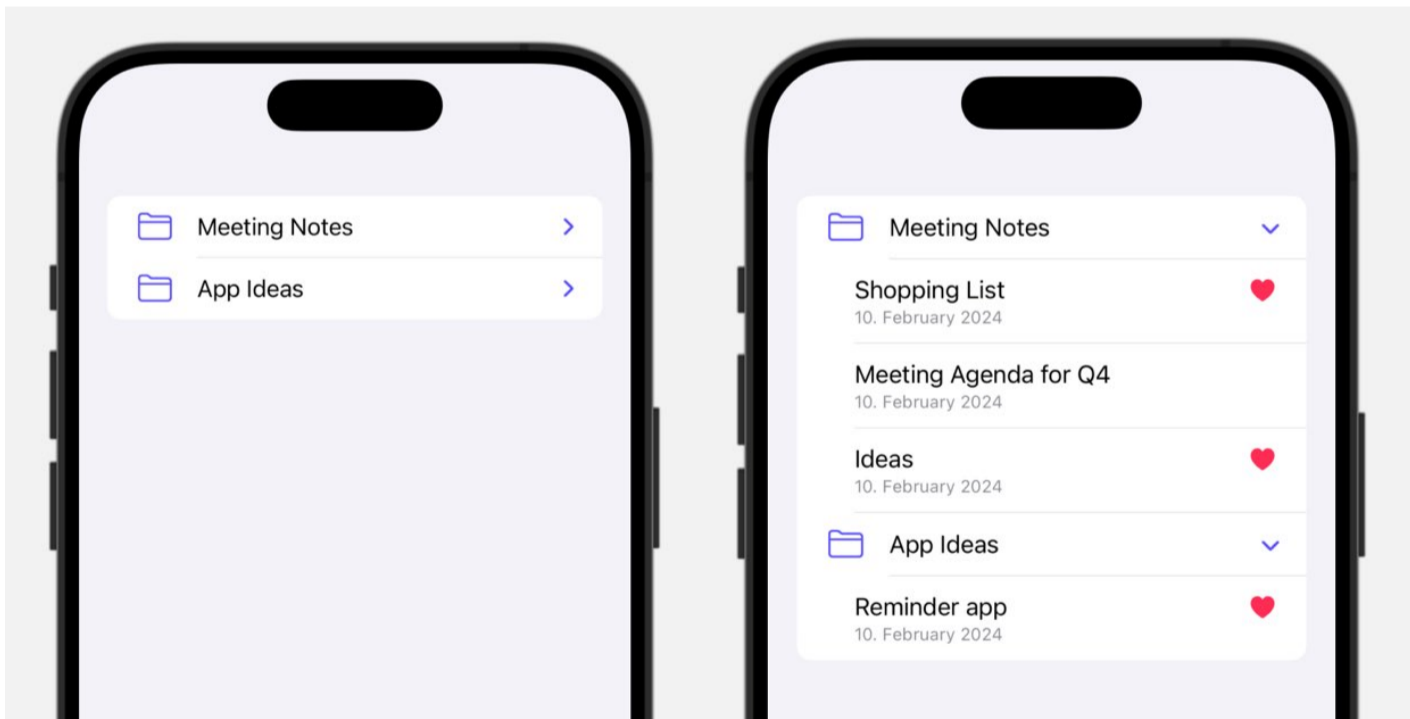
```
    }  
  }  
  .listStyle(.sidebar)
```

Note that you have to use sidebar list style to see the toggles next to the section headers. For all other styles collapsing is not available

Using DisclosureGroup for an Alternative Approach

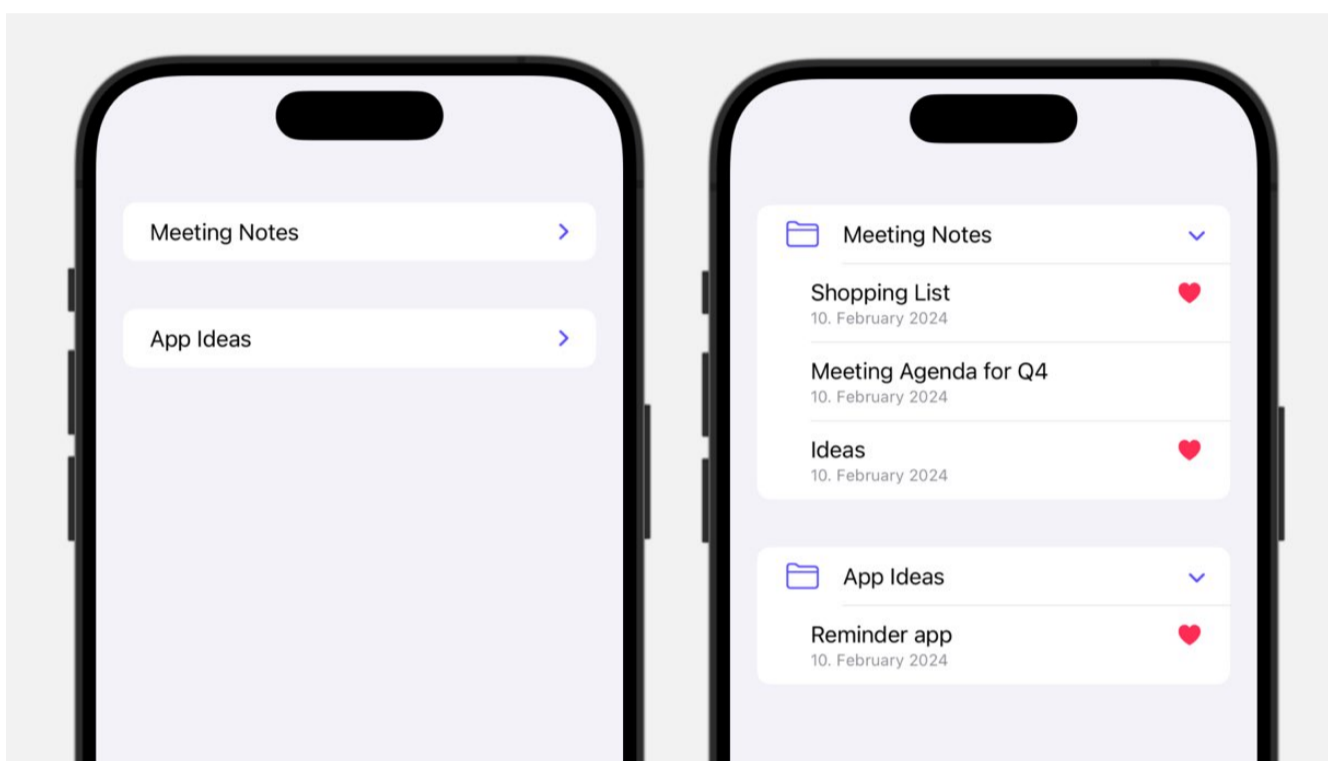
Alternatively, you can use DisclosureGroup to achieve a similar collapsible effect. This allows you to use multiple ForEach inside the list:

```
struct DisclosureListView: View {  
    let folders = Folder.examples()  
  
    var body: some View {  
        List {  
            ForEach(folders) { folder in  
                DisclosureGroup(folder.name) {  
                    ForEach(folder.notes) { note in  
                        NoteRowView(note: note)  
                    }  
                }  
            }  
        }  
    }  
}
```



Per default, all groups are shown in one section. But you can embed each DisclosureGroup in its section:

```
List {
  ForEach(folders) { folder in
    Section {
      DisclosureGroup(folder.name) {
        ForEach(folder.notes) { note in
          NoteRowView(note: note)
        }
      }
    }
  }
}
```



When the list appears all groups are initially collapsed. To control the initial state, DisclosureGroup also takes an expanded state property. Just like with the Section, you'll need to create separate subviews if you want to control the expanded state of each DisclosureGroup individually.

The following example will show the first section expanded:

```
struct DisclosureListView: View {
  let folders = Folder.examples()

  var body: some View {
    List {
      ForEach(folders) { folder in
        Section {
          FolderSectionView(folder: folder,
                           isInitiallyExpanded: folder == folders.first)
        }
      }
    }
  }
}
```

```

struct FolderSectionView: View {

    let folder: Folder
    let isInitiallyExpanded: Bool

    @State private var isExpanded: Bool = false

    var body: some View {
        DisclosureGroup(
            isExpanded: $isExpanded,
            content: {
                ForEach(folder.notes) { note in
                    NoteRowView(note: note)
                }
            },
            label: {
                Label(folder.name, systemImage: "folder")
            }
        )
        .onAppear(perform: {
            isExpanded = isInitiallyExpanded
        })
    }
}

```

By using either Section with the expanded initializer or DisclosureGroup, you can create collapsible sections within your SwiftUI list. Remember to manage the expanded state individually for each section to offer the best user experience. In the next lesson, we'll explore how to handle hierarchical data structures with multiple levels of nested sections.

11.4.3 Hierarchical Lists

When you're setting up your list, you'll notice that SwiftUI's List has an initializer that accepts a children parameter. This allows you to create deeply nested hierarchical lists.

Implementing the Data Model

Let's dive into the data model. You'll need a structure that has a property "children" to hold the nested items. All items in this array must be of the same type to enable the nesting:

```

import Foundation
import Observation

@Observable class FileItem: Identifiable {
    var title: String
    let isFolder: Bool
    var children: [FileItem]? = nil
    let id = UUID()
}

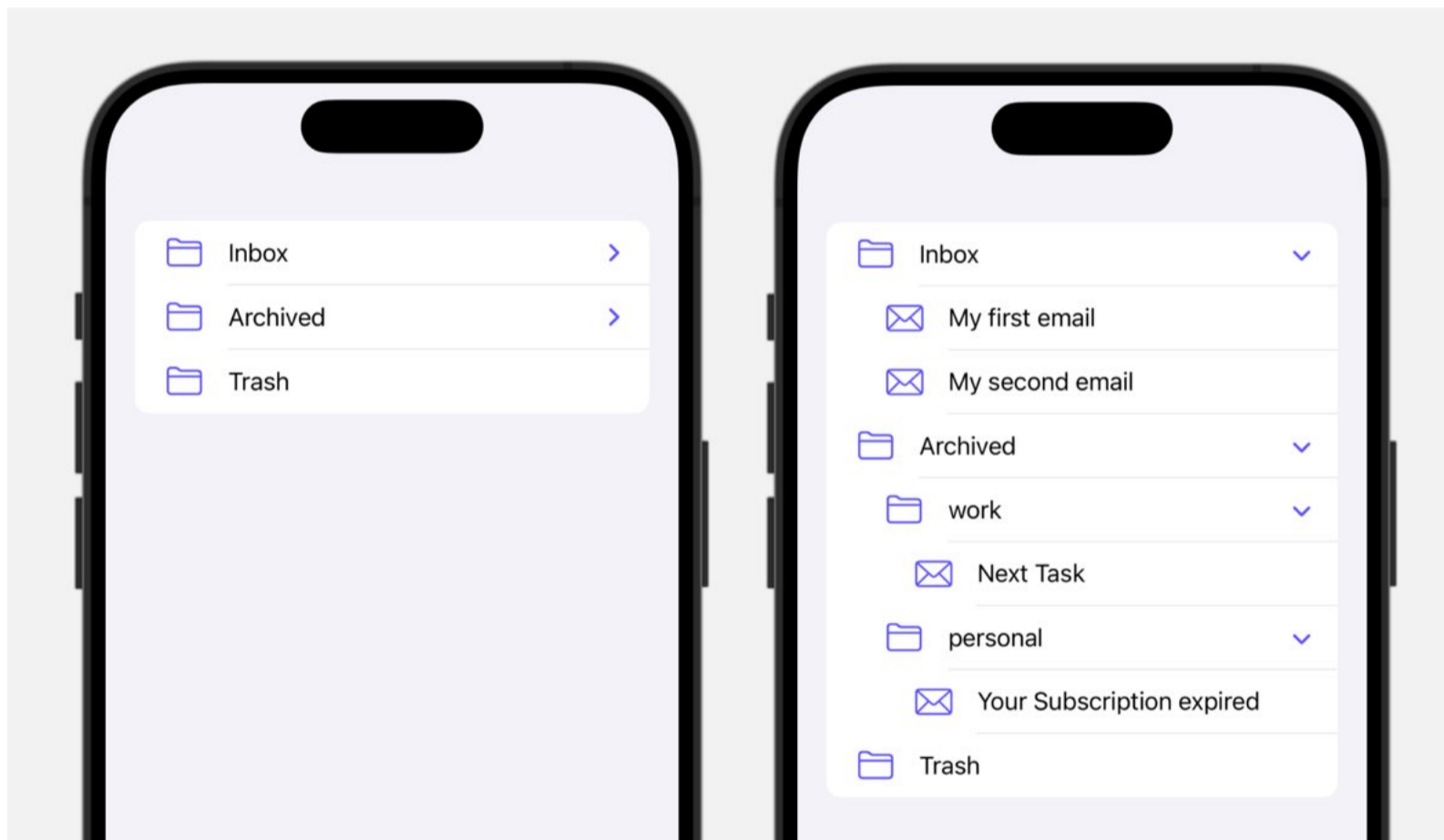
```

Creating the List

With the data model in place, you'll set up your list. You'll use the key path to point to the children property, which represents the nested items. Each of these children may, in turn, have their own children array, allowing for multiple levels of nesting.

You can also add a selection property to keep track of which item is selected. This will be an optional property that holds the selected item's ID.

```
struct HierarchicalListView: View {  
  
    @State private var fileItems = FileItem.preview()  
    @State private var selectedItem: FileItem.ID? = nil  
  
    var body: some View {  
        List(fileItems,  
            children: \.children,  
            selection: $selectedItem) { item in  
            Label(item.title,  
                systemImage: item.isFolder ? "folder" : "envelope")  
        }  
    }  
}
```



Adding Static Elements

If you need to include static elements or multiple ForEach in your list, you'll have to switch gears and use OutlineGroup instead. This component also supports children and can be used within a List to create complex hierarchical structures.

Here's an example of how you might set this up:

```
struct DisclosureHierarchicalListView: View {

    @State private var fileItems = FileItem.preview()
    @State private var selectedItem: FileItem.ID? = nil

    var body: some View {
        List(selection: $selectedItem) {
            OutlineGroup(fileItems, children: \.children) { item in
                Label(item.title, systemImage: item.isFolder ? "folder" : "envelope")
            }

            Section {
                Button("Do something") {
                    // do something
                }
            }
        }
    }
}
```

List with children and OutlineGroup will create the same-looking list UI.

By using SwiftUI's List with the children property or OutlineGroup, you can create deeply nested, hierarchical structures suitable for a variety of applications, including complex note-taking apps. Remember to ensure that all nested items are of the same type, and feel free to include static elements or multiple sections to create a rich and intuitive user interface.

11.5 FORM

When you're building most iOS apps, sticking with a List is typically sufficient. However, if you're venturing into macOS app development, taking a closer look at Form becomes more relevant.

On iOS, a Form essentially appears as a List with an `insetGrouped` style. So, if you're solely focused on iOS development, the Form may not seem as crucial. Yet, I'll walk you through some scenarios, such as user registration forms and settings views, where Form can be quite useful.

Creating a Form Example View

Forms are usually employed in scenarios where users need to input data, like in settings views. Let's start by creating a simple form with several input elements:

```
struct FormExampleView: View {  
  
    @State private var text = ""  
    @State private var number: CGFloat = 10  
    @State private var isTrue: Bool = false  
  
    var body: some View {  
        Form {  
            TextField("Enter Email", text: $text)  
            Slider(value: $number) {  
                Text("number \ \(Int(number))")  
            }  
            Picker("pick something", selection: $isTrue) {  
                Text("true").tag(true)  
                Text("false").tag(false)  
            }  
            Toggle("is True", isOn: $isTrue)  
        }  
        #if os(macOS)  
        .padding()  
        .frame(maxWidth: 350)  
        #endif  
    }  
}
```

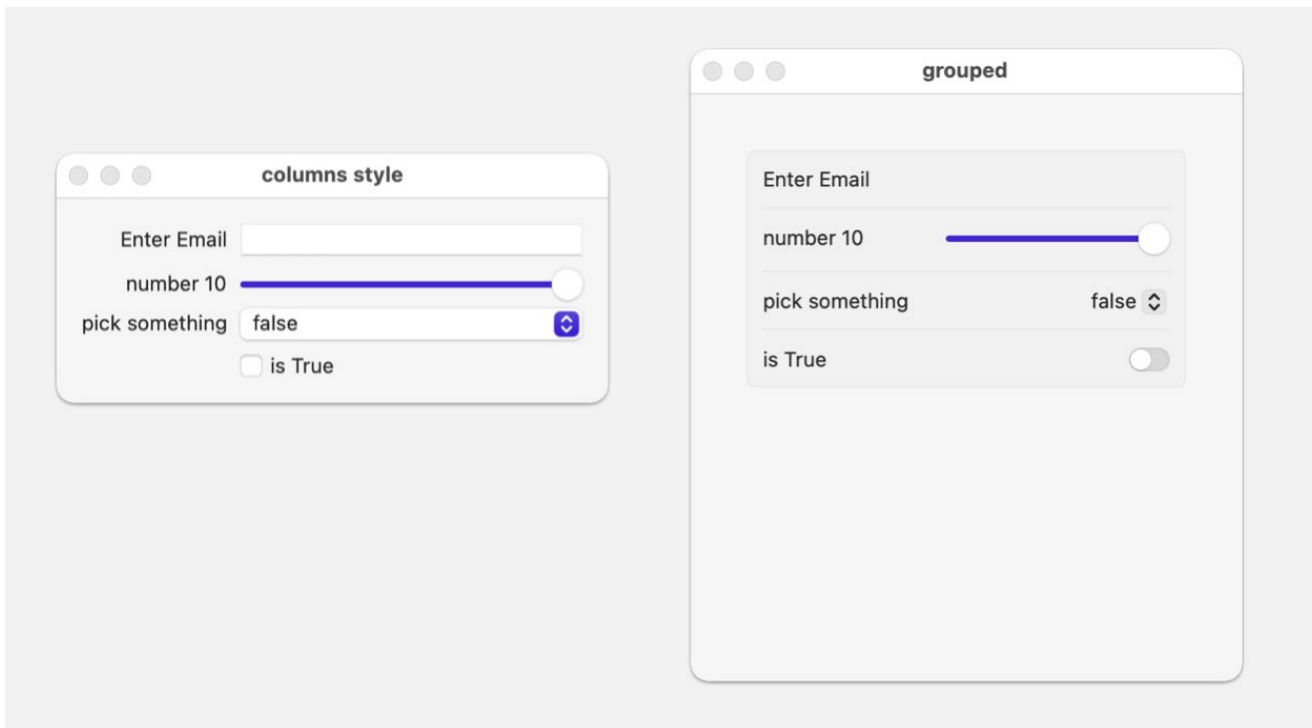
Form Styles

Form has two styles available:

- **Column Style:** Here, labels are in one column and inputs are in another. This is the default on macOS.
- **Grouped Style:** This style resembles a grouped List on iOS.

If you want to segment your form into sections, you can use the Section view. This works well for the grouped style but might not look as good with the column style.

```
Form {  
    ...  
}  
.formStyle(.grouped)
```

Form with Column Style that looks good

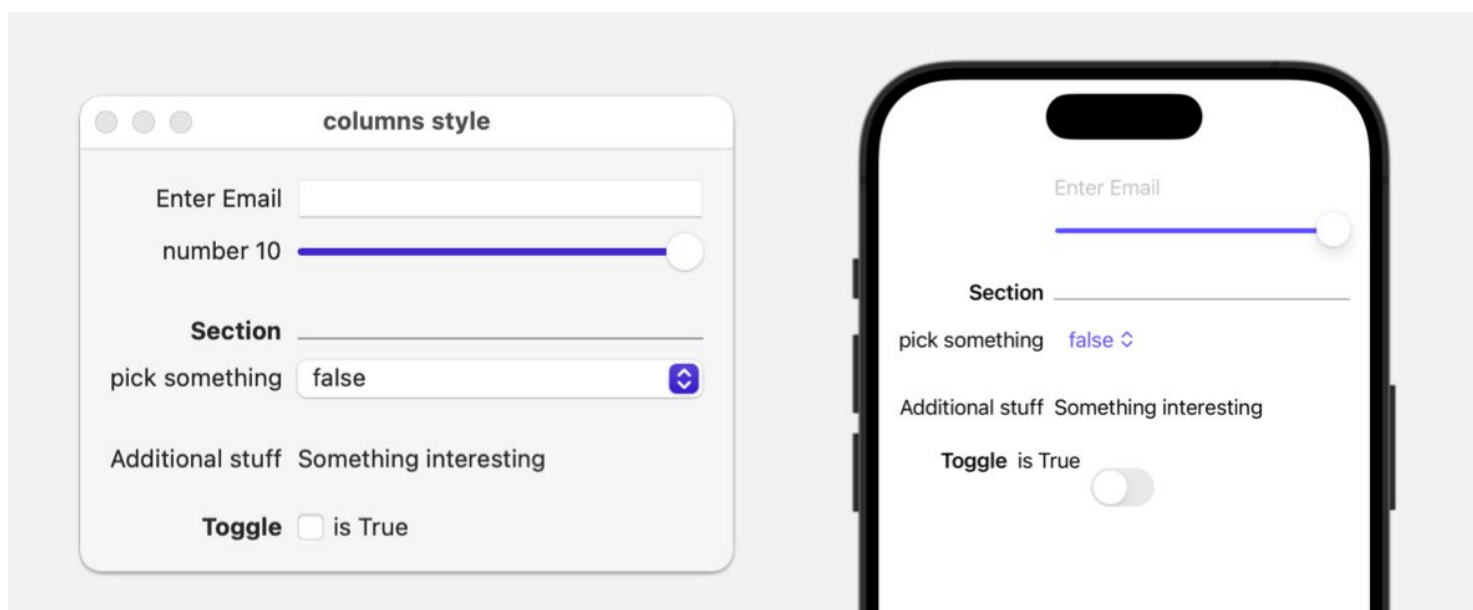
I want to show you now how to get a Form with a column style that looks well organised. For additional structure, you can use `LabeledContent` to place labels next to elements, or even insert custom separators like a thin line:

```

LabeledContent {
    Color.gray.frame(height: 1)
} label: {
    Text("Section")
    .bold()
    .padding(.top)
}

LabeledContent {
    Toggle("is True", isOn: $isTrue)
} label: {
    Text("Toggle")
    .bold()
}

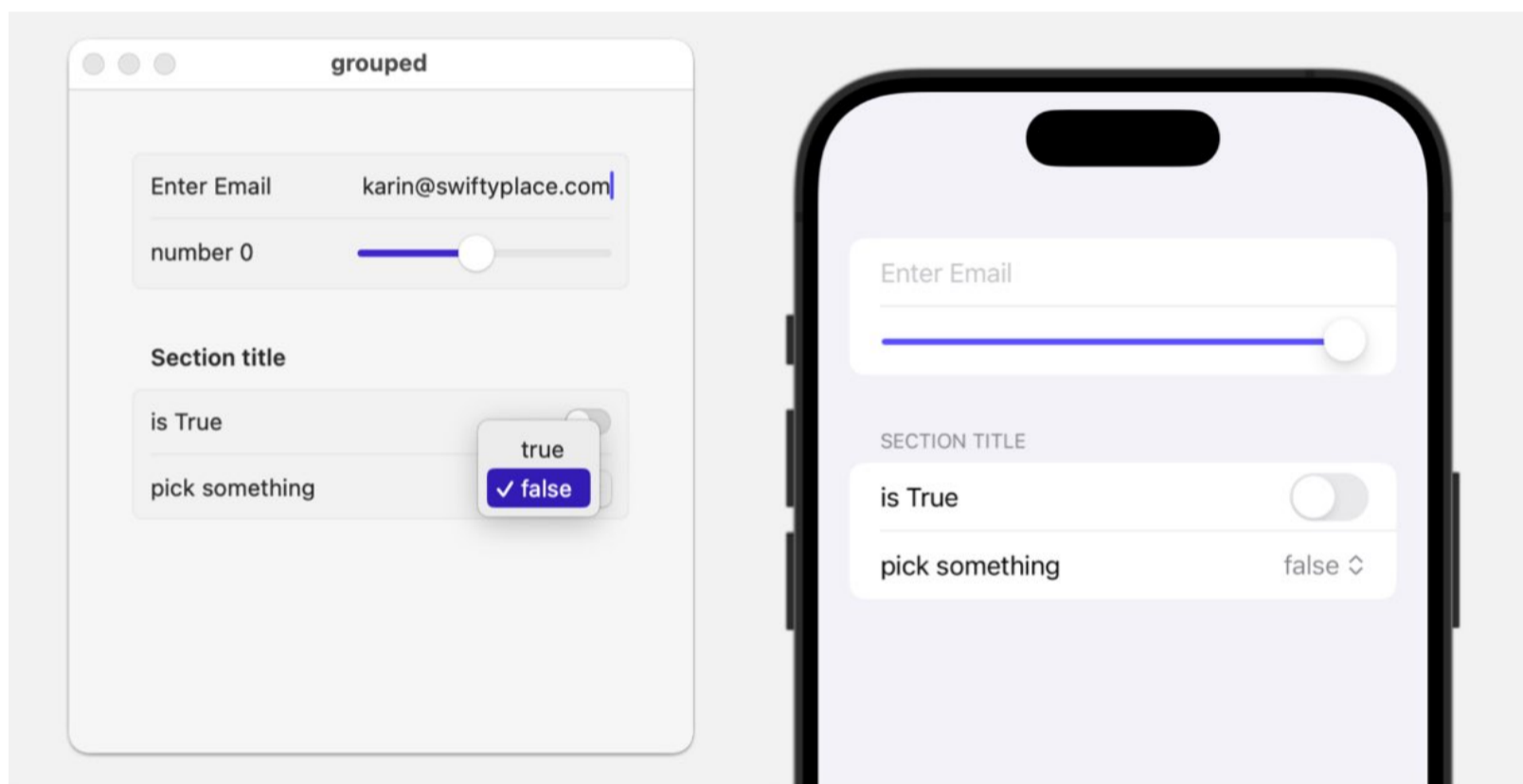
```



Form with Grouped Style

If you are using the grouped style for forms, you can also structure the content with sections:

```
Form {  
  TextField("Enter Email", text: $text)  
  Slider(value: $number) {  
    Text("number \$(Int(number))")  
  }  
  
  Section("Section title") {  
    Toggle("is True", isOn: $isTrue)  
  
    Picker("pick something", selection: $isTrue) {  
      Text("true").tag(true)  
      Text("false").tag(false)  
    }  
  }  
}
```



11.5.2 Example: Settings View

In this lesson, I want to show you how to create a settings view for a notes app using SwiftUI. We will be working on both macOS and iOS, so we'll explore different layout options for each platform.

macOS Settings View

On macOS, I'll use the default form style to create the settings view. This style is commonly used in the Settings app on macOS. Here's an example of how we can set up the form:

```
struct SettingMacOSView: View {

    @State private var licenseKey: String = ""
    @AppStorage("selectedAppearance") var selectedAppearance = 0

    @State private var fontSize: CGFloat = 15
    @State private var showLineNumbers = false
    @State private var showPreview = true

    var body: some View {
        Form {
            LabeledContent("macOS Version", value: "2.2.1")
                .bold()
            TextField("License Key", text: $licenseKey)

            Picker(selection: $selectedAppearance) {
                Text("Default System").tag(0)
                Text("Light").tag(1)
                Text("Dark").tag(2)
            } label: {
                Text("Color Scheme")
            }
            #if os(macOS)
                .pickerStyle(.radioGroup)
            #endif

            ...

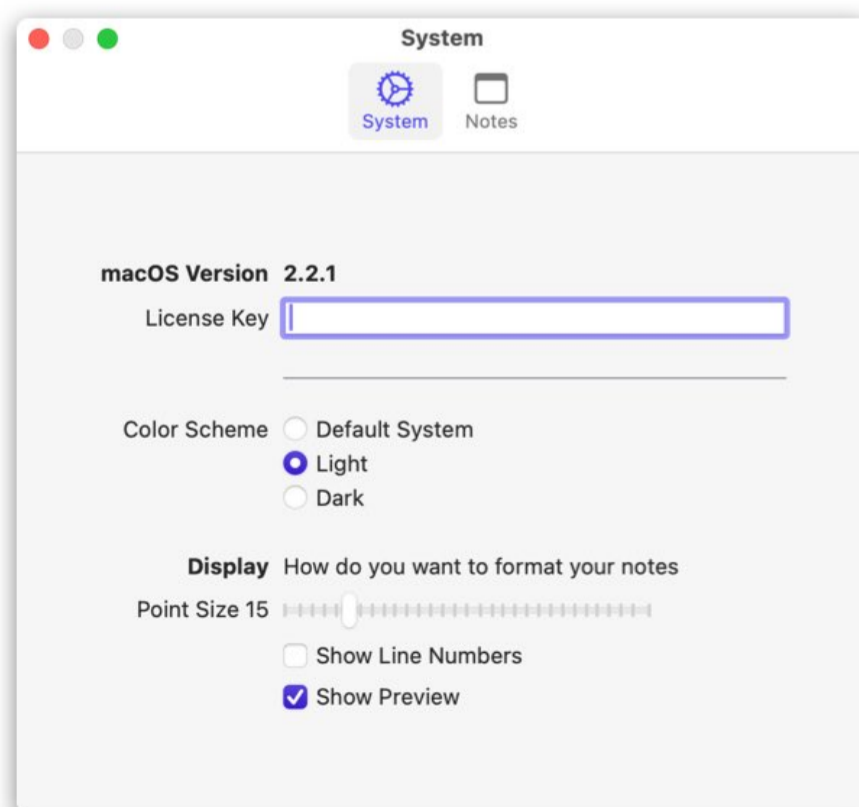
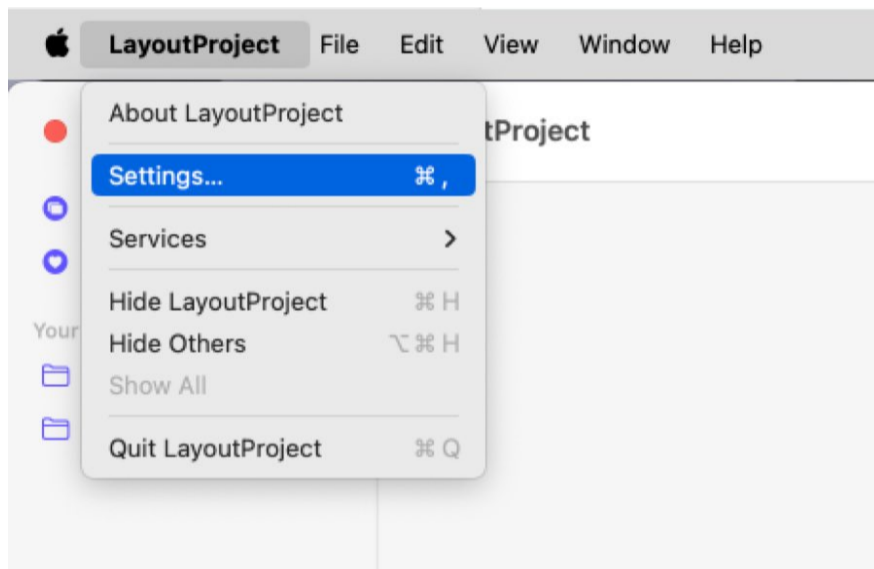
        }
        .frame(maxWidth: 400)
        .padding()
        .frame(maxWidth: .infinity, maxHeight: .infinity)
        .preferredColorScheme(colorScheme())
    }

    func colorScheme() -> ColorScheme? {
        if selectedAppearance == 1 {
            return ColorScheme.light
        } else if selectedAppearance == 2 {
            return ColorScheme.dark
        } else {
            return nil
        }
    }
}
```

To show this form in the settings window, I am using it for the Settings group in the main app file:

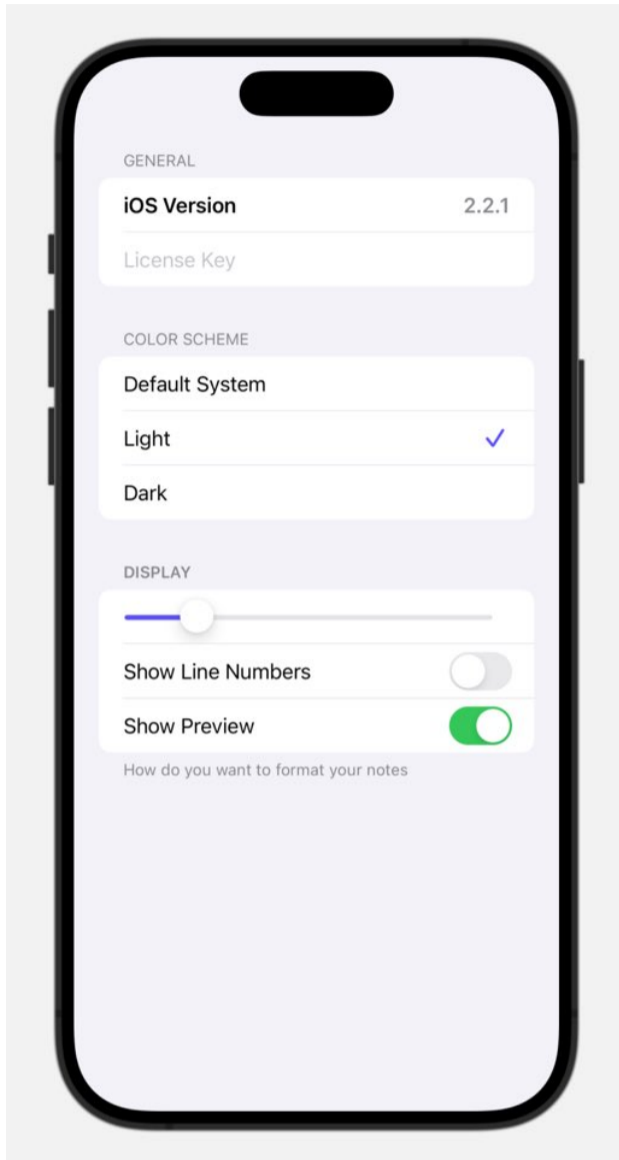
```
@main
struct LayoutProjectApp: App {
    var body: some Scene {
        WindowGroup {
            NavigationListView()
        }

        #if os(macOS)
        Settings {
            TabView {
                SettingMacOSView()
                .tabItem { Label("System", systemImage: "gear") }
                Text("Something else")
                .tabItem { Label("Notes", systemImage: "note") }
            }
        }
        #endif
    }
}
```



iOS Settings View

On iOS, we'll use sections to organize the settings. Here's an example of how we can set up the settings view for iOS:



```
struct SettingsIOSView: View {

    @State private var licenseKey: String = ""
    @AppStorage("selectedAppearance") var selectedAppearance = 0

    @State private var fontSize: CGFloat = 15
    @State private var showLineNumbers = false
    @State private var showPreview = true

    var body: some View {
        Form {
            Section("General") {
                LabeledContent("iOS Version", value: "2.2.1")
                    .bold()
                TextField("License Key", text: $licenseKey)
            }

            Picker(selection: $selectedAppearance) {
                ""
            } label: {
                Text("Color Scheme")
            }
            .pickerStyle(.inline)
        }
    }
}
```

```

        ...
    }
    .preferredColorScheme(colorScheme())
}

func colorScheme() -> ColorScheme? {
    ...
}
}

```

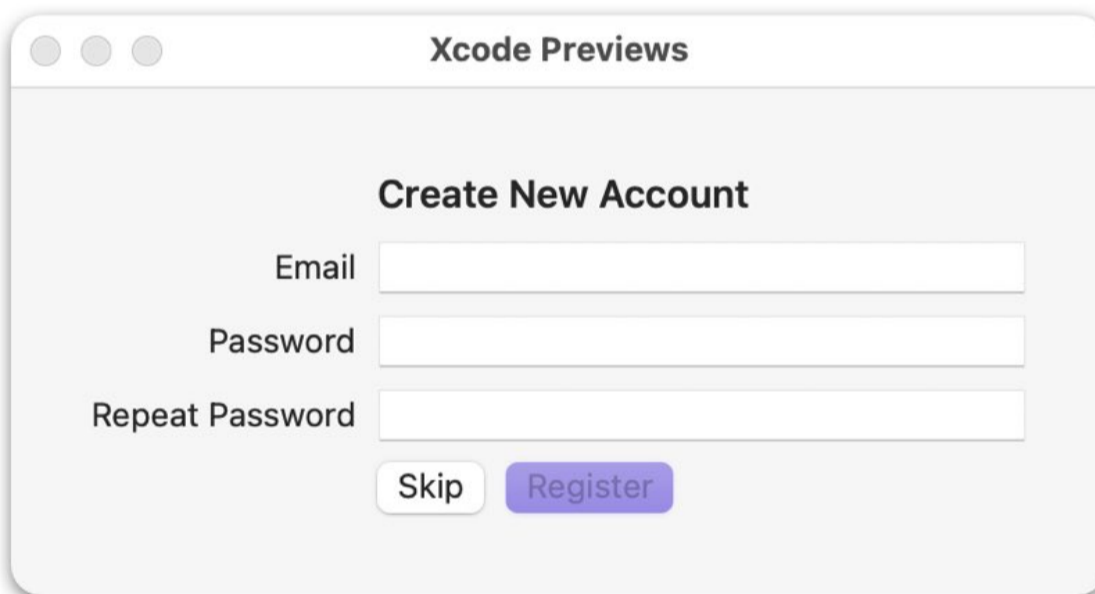
On iOS, you can navigate to the settings view using a navigation stack or any other navigation pattern that suits your app's structure.

11.5.3 Example: Registration Form

In most apps, one common view that you'll need is a user registration form. Let's take a look at how we can create a registration form for both macOS and iOS using SwiftUI.

macOS: Form with Column Style

When designing the registration form for macOS, we can utilize the Form container with a column form style. This style provides a clean and organized layout. Here's an example of what the form might look like:



```

struct RegistrationView: View {

    @State private var email = ""
    @State private var password = ""
    @State private var repeatPassword = ""

    var body: some View {
        Form {
            Text("Create New Account")
                .font(.title3)
                .bold()

```

```

        TextField("Email", text: $email)
        TextField("Password", text: $password)
        TextField("Repeat Password", text: $repeatPassword)

        HStack {
            Button {

                } label: {
                    Text("Skip")
                }
            Button {

                } label: {
                    Text("Register")
                }
                .buttonStyle(.borderedProminent)
                .disabled(email.isEmpty)
        }
    }
    .padding()
    .frame(maxWidth: 400)
    .frame(maxWidth: .infinity, maxHeight: .infinity)
}
}

```

iOS: Customized Registration Form

On iOS, we have the flexibility to create a more customized registration form. We can design the form with unique styles and layouts. Here's an example for iOS where I used ScrollView and VStack to layout the registration fields like email and password textfields:

```

struct RegistrationiOSView: View {

    @State private var email = ""
    @State private var password = ""
    @State private var repeatPassword = ""

    var body: some View {
        NavigationStack {
            ScrollView{
                VStack(alignment: .leading){
                    Text("Enter Your Email")
                    TextField("Email", text: $email)
                        .padding(.bottom, 30)

                    Text("Enter Your New Password")
                    TextField("Password", text: $password)
                    TextField("Repeat Password", text: $repeatPassword)
                        .padding(.bottom, 50)

                    HStack {
                        Button {

                            } label: {
                                Text("Skip")
                            }
                            .frame(maxWidth: .infinity)
                        }
                        .buttonStyle(.bordered)

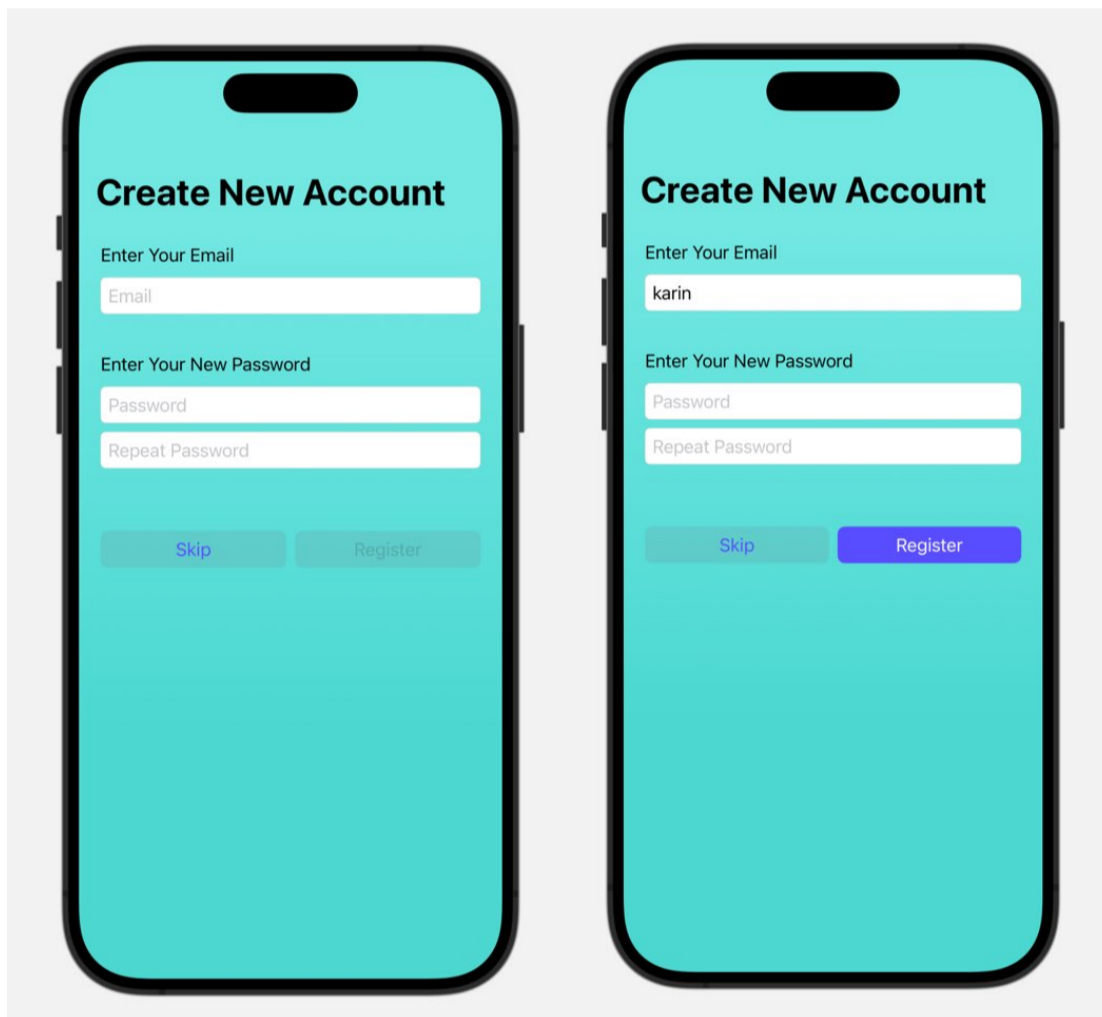
                        Button {

```

```

        } label: {
            Text("Register")
                .frame(maxWidth: .infinity)
        }
        .buttonStyle(.borderedProminent)
        .disabled(email.isEmpty)
    }
}
.textFieldStyle(.roundedBorder)
}
.contentMargins(20)
.navigationTitle("Create New Account")
.background(Color.mint.gradient.opacity(0.7))
}
}
}
}

```



Creating a user registration form is an essential part of many apps. Whether you prefer the clean and organized layout of a form with a column style on macOS or the customized and visually appealing design on iOS, SwiftUI provides the tools to create a user-friendly registration experience. Choose the style that best suits your app's needs and get started on building a functional and visually pleasing registration form.

11.5.4 Example: Inspector

In this lesson, we will explore how to create an inspector view using the Form container in SwiftUI. The inspector view allows you to display additional details or settings for a specific item, such as a note in a

note-taking app. We'll also discuss how to adapt the inspector for different presentation styles on iOS and macOS.

In the following, I have a view that shows the Note information. You can change the title and content of the note:

```
struct NoteEditView: View {
    @State private var note = Note.example()
    @State private var fontSize: CGFloat = 15
    @State private var isPresentingInspector = false

    var icon: String {
        #if os(macOS)
            "sidebar.trailing"
        #else
            "slider.horizontal.3"
        #endif
    }

    var body: some View {
        List {
            Section("title") {
                TextField("title", text: $note.title)
            }

            Section("Notes") {
                TextEditor(text: $note.content)
                    .font(.system(size: fontSize))
            }

            Toggle(isOn: $note.isFavorite) {
                Text("is Favourite")
            }
        }
        .inspector(isPresented: $isPresentingInspector) {
            EditorView(fontSize: $fontSize)
                .inspectorColumnWidth(min: 200, ideal: 400, max: 500)
        }
        .toolbar(content: {
            Toggle(isOn: $isPresentingInspector) {
                Label("Show Note Display Settings", systemImage: icon)
            }
        })
    }
}
```

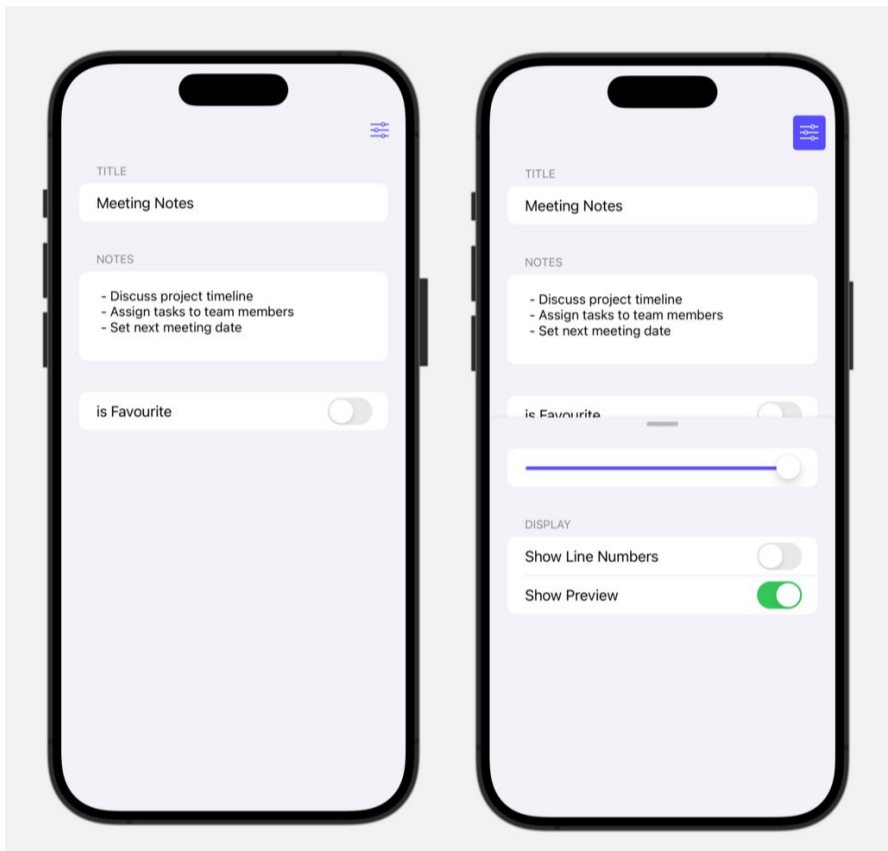
In the toolbar is a button to toggle the **isPresentingInspector** property which determines if the inspector is open or closed. The content of the inspector is the following EditorView. I am passing the `fontSize` with a Binding. This allows to change the `fontSize` in the inspector. In the above `NoteEditView` this property is then used to set the font size of the note content text.

```
struct EditorView: View {
    @Binding var fontSize: CGFloat
    @State private var showLineNumbers = false
    @State private var showPreview = true

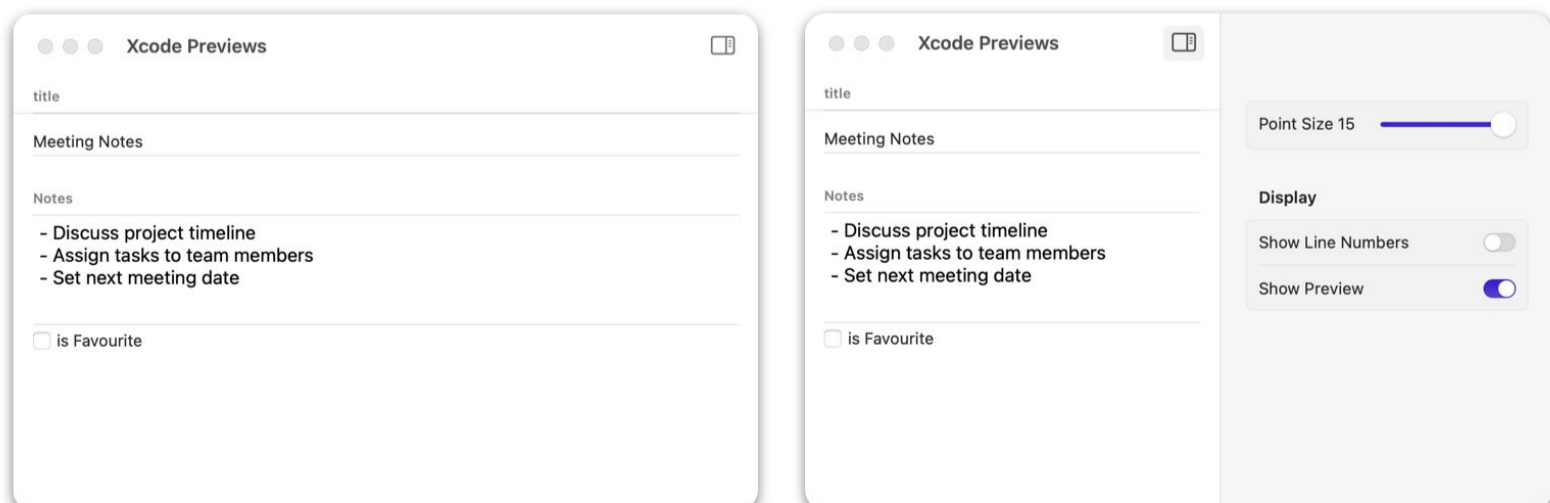
    var body: some View {
        Form {
            Slider(value: $fontSize) {
                Text("Point Size \((Int(fontSize))")
            }
        }
    }
}
```

```
}  
  
Section("Display") {  
    Toggle(isOn: $showLineNumbers, label: {  
        Text("Show Line Numbers")  
    })  
    Toggle(isOn: $showPreview, label: {  
        Text("Show Preview")  
    })  
}  
}  
}  
}
```

When you open the inspector on iOS, the content of the inspector is presented as a sheet:



On macOS, the inspector will open a column on the side of the main content. For the container in the inspector view, I am using Form, because this will give me a very specific look on macOS. You can see that Form with grouped form style is used:

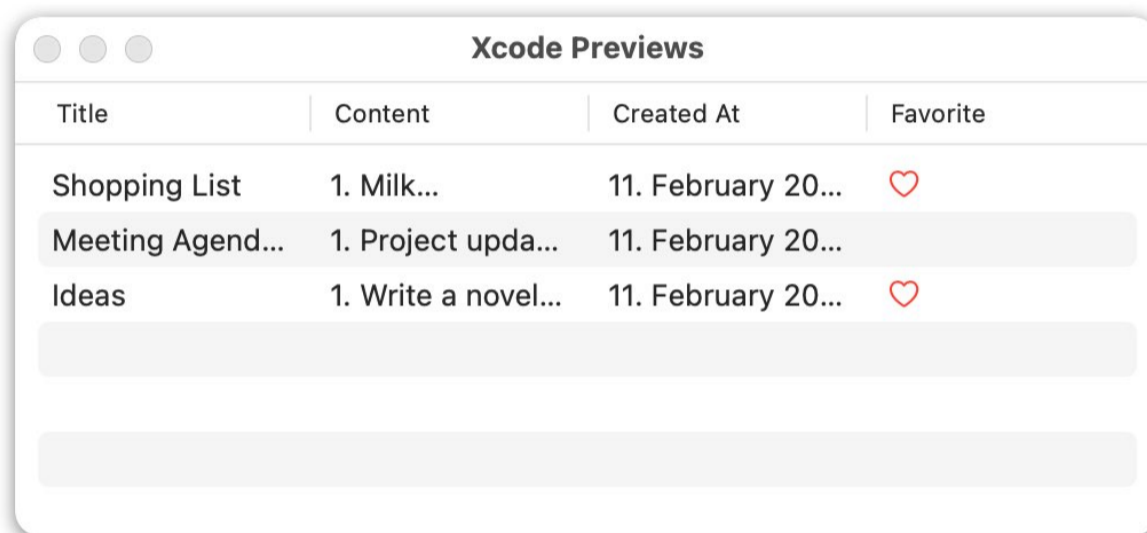


11.6 TABLE

The last system container I want to dive into is the Table. If you've worked with macOS, you might be familiar with tables, or `NSTableView` from `AppKit`, which you see in applications like Finder. These tables allow you to sort by different columns such as date, creation date, or file size. You can also adjust the width of these columns and make selections, even multiple at a time.

Adding context menus is another powerful feature that lets you perform actions like moving items to the bin, opening with different applications, or tagging. You can leverage this functionality with tables in SwiftUI, bringing a rich feature set to your app's UI.

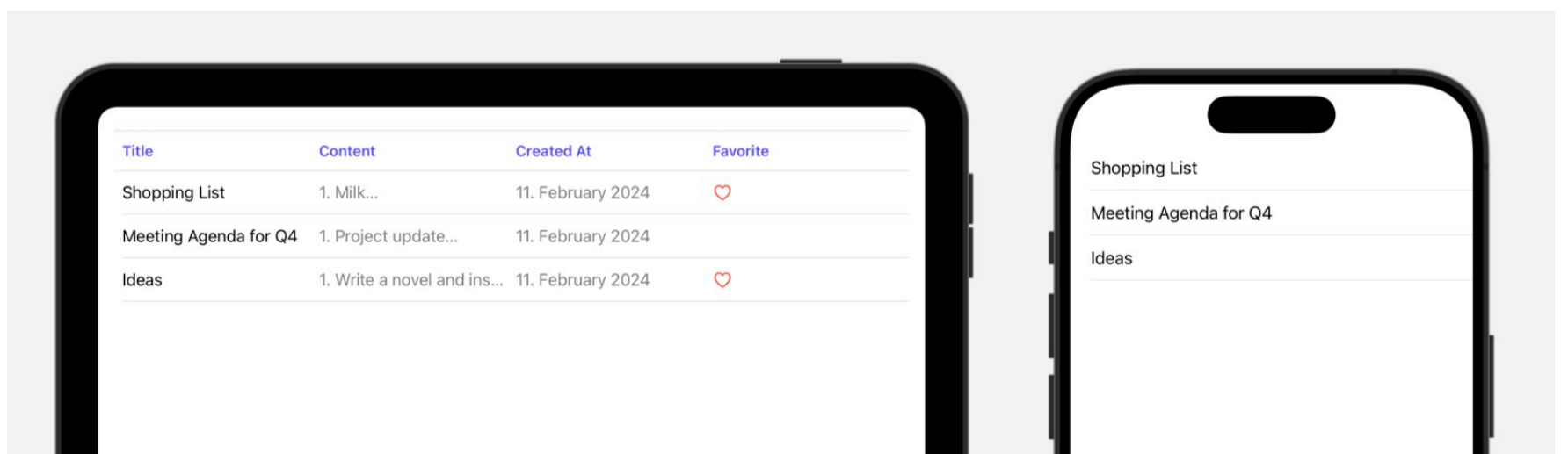
However, it's important to note that SwiftUI's Table doesn't offer all the same capabilities as the underlying `NSTableView`. For instance, reordering or hiding columns isn't something you can currently do in SwiftUI.



Title	Content	Created At	Favorite
Shopping List	1. Milk...	11. February 20...	♥
Meeting Agend...	1. Project upda...	11. February 20...	
Ideas	1. Write a novel...	11. February 20...	♥

Tables on iPad and iPhone

Tables aren't just for macOS; you can use them on iPad as well. However, when you bring tables to the iPhone, they will appear more like a list, showing only the first column. This is something to keep in mind when you're designing a multi-platform app—you'll need to adapt your UI accordingly.



11.6.1 Creating a Table with SwiftUI

I will again use the Note class as an example that I'm going to display in a Table. Here's how you can initialize a Table:

```
struct TableExampleView: View {
    let notes = Note.examples()

    var body: some View {
        Table(notes) {
            // Define columns here
        }
    }
}
```

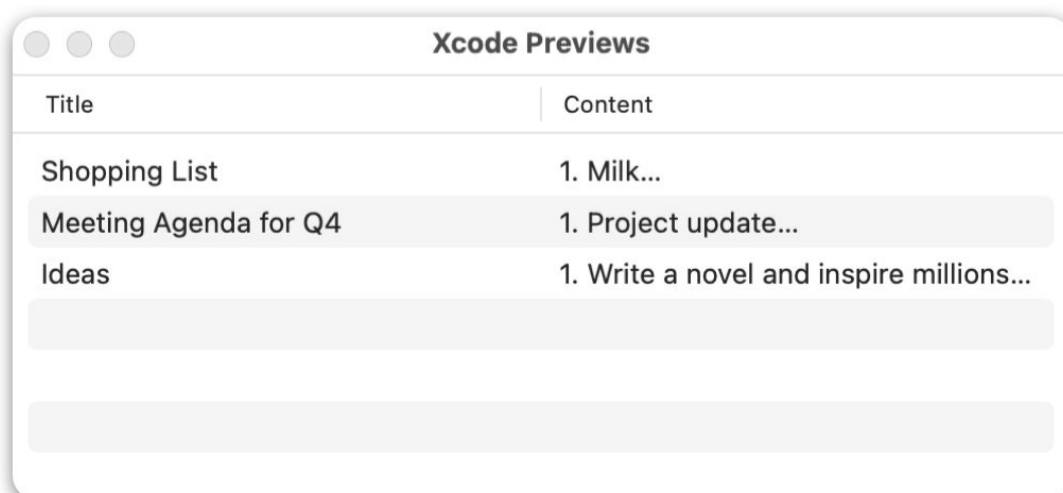
In the Table initializer, you pass in an array of data just like you would with a List. Then, you define your columns. With Table, you have specific types such as TableRow or TableColumn to work with. For example, you can create a column with TableColumn using the text and value initializers:

```
TableColumn("Title", value: \Note.title)
```

Here, the first parameter is the header text for the column, and the second is the key path to the property you want to display. This property needs to be of type String.

In the following, I am showing a table with 2 columns for the title and content of the Note data:

```
Table(notes) {
    TableColumn("Title", value: \Note.title)
    TableColumn("Content", value: \Note.content)
}
```



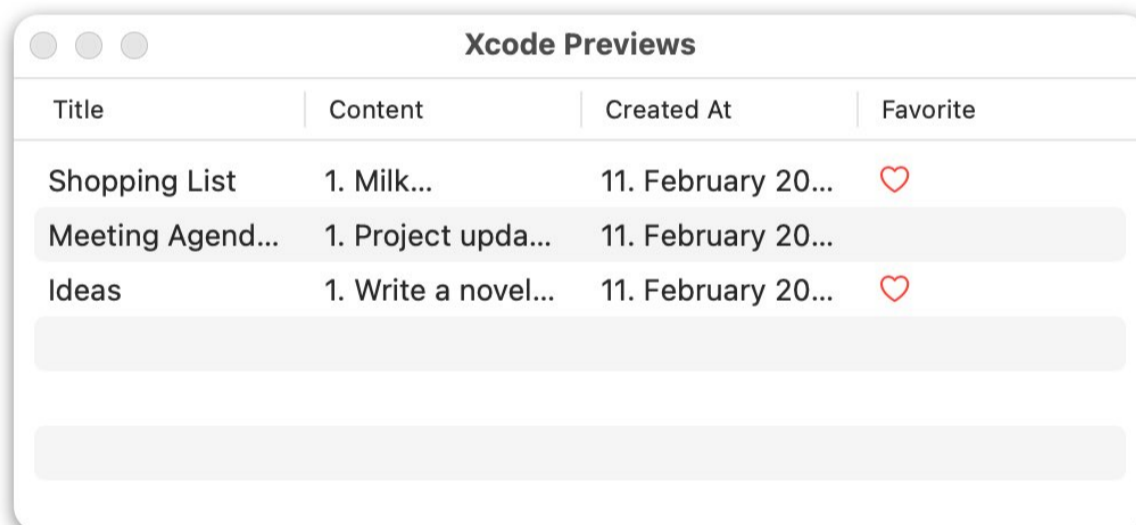
Title	Content
Shopping List	1. Milk...
Meeting Agenda for Q4	1. Project update...
Ideas	1. Write a novel and inspire millions...

Displaying Multiple Lines and Custom Content

If you want to display content that spans multiple lines, you might run into some limitations. The Table doesn't like to accommodate larger content, so you'll often be sticking with a single line for each cell.

However, you can still include different types of content. For instance, if you have a date property, you can use a TableRow initializer with content closure. This allows you to get the data instance and format it with a Text view. In the following, I am showing 2 columns for the creation date and isflvorite property of Note:

```
Table(notes) {  
    TableColumn("Title", value: \Note.title)  
    TableColumn("Content", value: \Note.content)  
  
    TableColumn("Created At") { note in  
        Text(note.creationDate, style: .date)  
    }  
  
    TableColumn("Favorite") { note in  
        Image(systemName: "heart")  
        .foregroundColor(note.isFavorite ? Color.red : Color.clear)  
    }  
}
```



Title	Content	Created At	Favorite
Shopping List	1. Milk...	11. February 20...	♥
Meeting Agend...	1. Project upda...	11. February 20...	
Ideas	1. Write a novel...	11. February 20...	♥

The order of the columns is determined by the order of the TableColumns that you specify.

Table uses TableRowContent, not standard View, so modifiers and views might differ from what you're used to with other SwiftUI views. For example, to set a column to a specific width, you can use the width modifier:

```
TableColumn("Content", value: \Note.content)  
    .width(200)
```

In the next lesson, I'll delve deeper into styling and customizing the Table to make it fit perfectly within your app's design. Stay tuned!

11.6.2 Table Styling

When you're working with tables in SwiftUI, you might find that the default look doesn't quite fit the aesthetic of your app. Luckily, you can tweak the appearance to a certain extent. Let's dive into some styling options that can help you customize your tables.

Table Styles

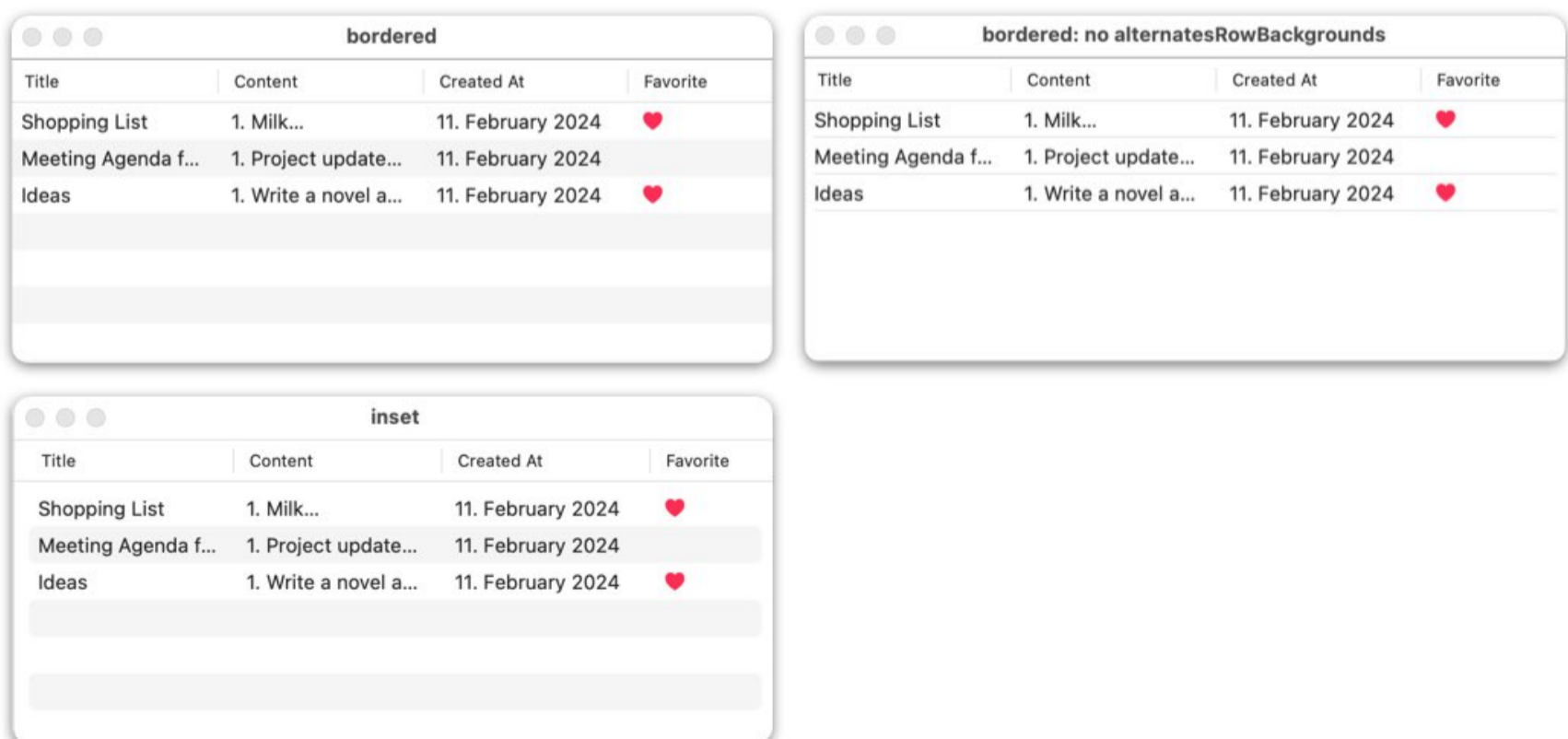
Moving on to table styles, you have a couple of options to choose from:

- **Bordered:** This style gives your table clear borders around each cell (macOS only)
- **Inset:** The inset style adds some spacing around the edges of the table, so the alternating row backgrounds don't touch the sides.

If you prefer to disable the alternating row background, the approach has changed a bit. Instead of a single modifier, you now use two:

```
Table(notes) {  
    ...  
}  
.tableStyle(.bordered)  
.alternatingRowBackgrounds(.disabled)
```

With these modifiers, your table will have a more uniform look without the alternating backgrounds. Unfortunately, there isn't a built-in way to remove the separator lines between rows.



Column Width Customization

First up, let's talk about column width. You can set a fixed column width with:

```
TableColumn("Content", value: \Note.content)
    .width(200)
```

Or allow the user to change the width by specifying columns with minimum, ideal, and maximum values. For example, if you're dealing with a column that displays favorite notes, you might want to set it like this:

```
TableColumn("Favorite") { note in
    ...
}
    .width(min: 20, ideal: 40, max: 80)
```

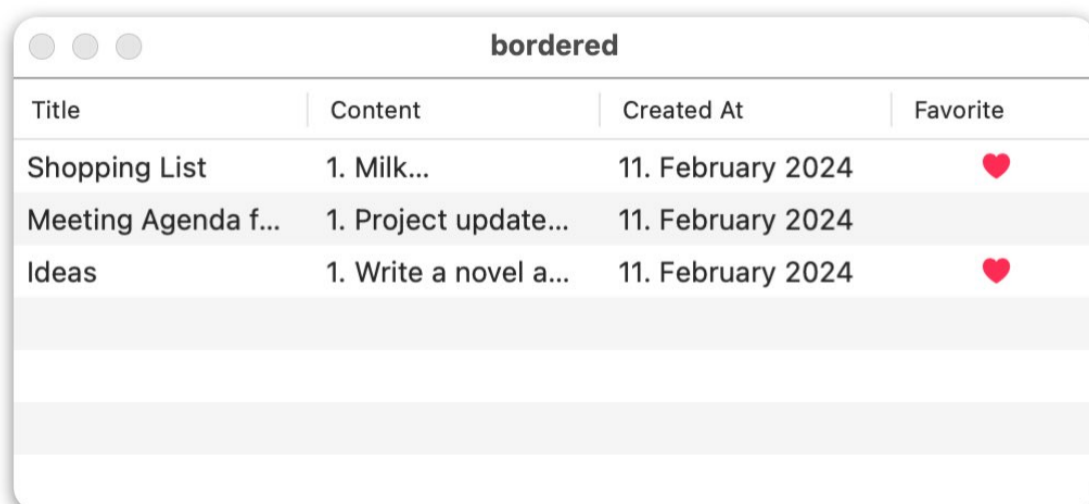
With these settings, the favorite column won't expand beyond 80 points in width, ensuring your table remains neatly organized.

Alignment for Table Columns

If you want to change the alignment of a view inside the column, you can use a flexible frame. For example, I can achieve a center alignment of the heart icon like so:

```
TableColumn("Favorite") { note in
    if note.isFavorite {
        Image(systemName: "heart.fill")
            .foregroundColor(.pink)
            .frame(maxWidth: .infinity, alignment: .center)
    }
}
```

This is more of a workaround. Note that the header remains leading aligned and a solution to change this behavior does not currently exist.



Title	Content	Created At	Favorite
Shopping List	1. Milk...	11. February 2024	♥
Meeting Agenda f...	1. Project update...	11. February 2024	
Ideas	1. Write a novel a...	11. February 2024	♥

Hiding the Table Header

Lastly, you might want to remove the table header for a cleaner look. Simply add the following modifier after your table definition:

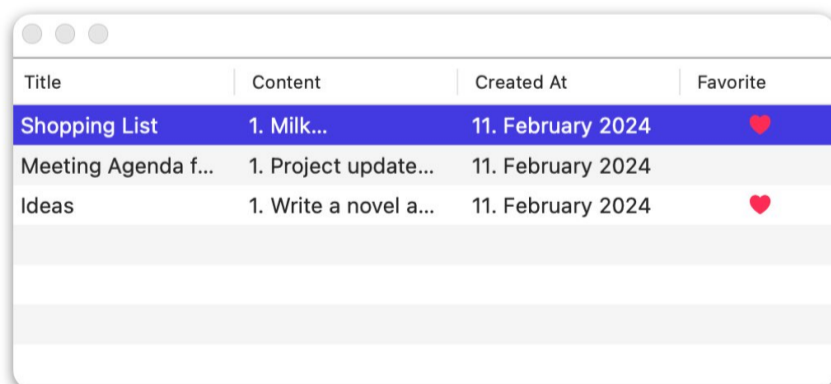
```
Table(notes) {  
    ...  
}  
.tableColumnHeaders(.hidden)
```



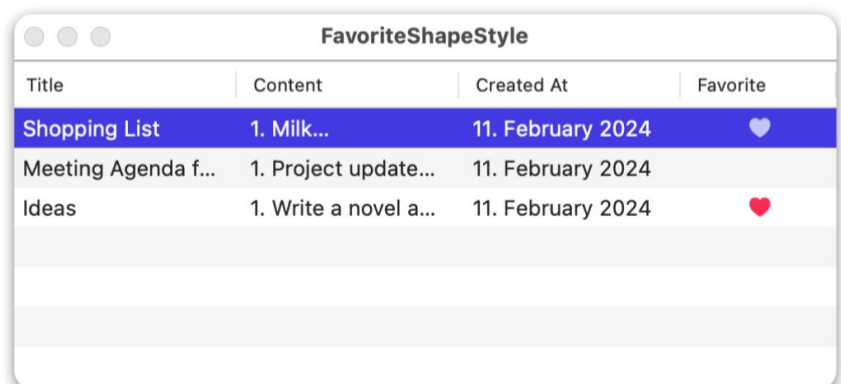
Title	Content	Created At	Favorite
Shopping List	1. Milk...	11. February 2024	♥
Meeting Agenda f...	1. Project update...	11. February 2024	
Ideas	1. Write a novel a...	11. February 2024	♥

Selection and Custom Shape Styles

If your table includes icons, you might face a situation where the default selection color clashes with your icons. To address this, you can define a custom shape style. In the below example on the left, you can see the pink heart icon in contrast to the blue selected row. On the right, I used a custom shape style. The heart icon for the selected row is dimmed down. All other icons like in the last row show the higher contact pink heart icon.



Title	Content	Created At	Favorite
Shopping List	1. Milk...	11. February 2024	♥
Meeting Agenda f...	1. Project update...	11. February 2024	
Ideas	1. Write a novel a...	11. February 2024	♥



Title	Content	Created At	Favorite
Shopping List	1. Milk...	11. February 2024	♥
Meeting Agenda f...	1. Project update...	11. February 2024	
Ideas	1. Write a novel a...	11. February 2024	♥

Here's how you can create a custom shape style for your favorite icons:

```
struct FavoriteShapeStyle: ShapeStyle {  
    func resolve(in environment: EnvironmentValues) -> some ShapeStyle {  
        if environment.backgroundProminence == .increased {  
            return AnyShapeStyle(.secondary)  
        } else {  

```



```

        }
        return AnyShapeStyle(.pink)
    }
}

```

Inside your custom shape style, you can check the `environment.backgroundProminence` and decide if you want to use a secondary style or your chosen color, like pink or red, for the icons.

To apply your custom style to a view, you would use:

```

struct TableStylingView: View {
    let notes = Note.examples()
    @State private var selectedNote: Note.ID? = nil

    var body: some View {
        Table(notes, selection: $selectedNote) {
            ...

            TableColumn("Favorite") { note in
                if note.isFavorite {
                    Image(systemName: "heart.fill")
                        .foregroundColor(FavoriteShapeStyle())
                }
            }
        }
    }
}

```

11.6.3 Edit Table Data

So far, you’ve learned how to display static data within tables. But what if you want to make changes to that data? Unlike lists, where you can pass a binding to interact with e.g. text fields, tables don’t offer that same flexibility directly. You might be wondering, “Can I edit the data in a table at all?” The answer is yes, but we’ll need to approach it differently.

Context Menus to the Rescue

One solution is to incorporate context menus into your table. You’ll use the same data model as in the previous examples, but this time, declare your data array using `@State private var` to ensure you can modify it.

Now, the tricky part is figuring out where to add your context menu. Thankfully, SwiftUI provides an initializer for tables that gives you access to the rows. It’s slightly different from what you’re used to because it requires specifying the type of data you’re working with. This is essential so that SwiftUI knows how to map your data to the columns.

```

struct TableEditView: View {
    @State private var notes = Note.examples()

    var body: some View {
        Table(of: Note.self) {
            TableColumn("Title", value: \Note.title)
            ...
        }
    }
}

```

```

    } rows: {
      ForEach($notes) { $note in
        TableRow(note)
      }
    }
  }
}

```

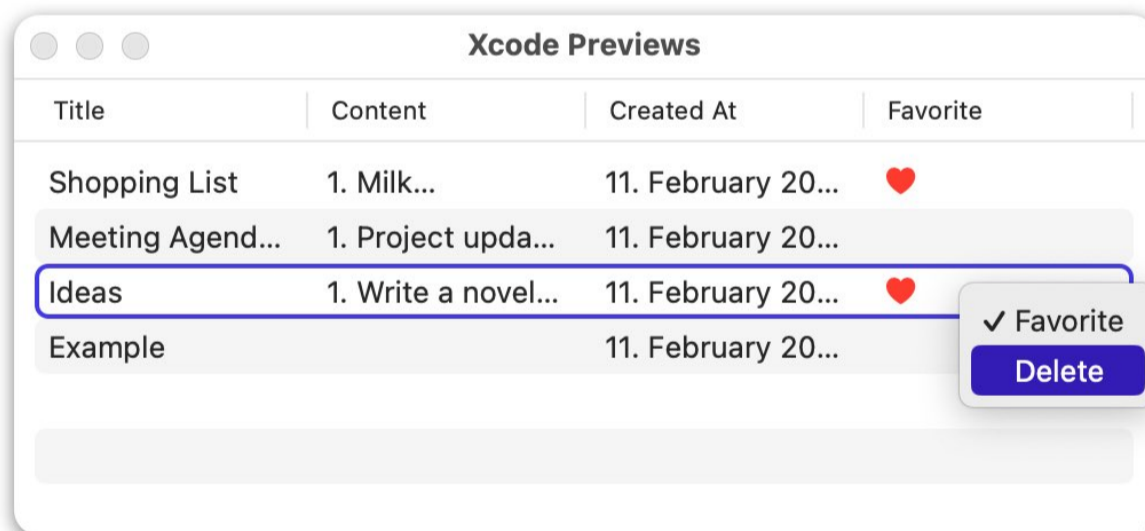
Now that you have access to the individual TableRows, you can apply a context menu. In the following, I added a Toggle for the favorite property and a delete button:

```

TableRow(note)
  .contextMenu {
    Toggle("Favorite", isOn: $note.isFavorite)
    Button(action: {
      if let index = notes.firstIndex(of: note) {
        notes.remove(at: index)
      }
    }, label: {
      Label("Delete", systemImage: "trash")
    })
  }
}

```

With this setup, a right-click on a row will reveal the context menu with “Delete” and “Favorite” options. The “Favorite” toggle will allow you to mark a note as a favorite, changing its state in real time.



Editing Notes with a Detail View

Another approach for editing data is to select a row and display its details in an inspector panel or present a sheet with editable fields. This method provides a more detailed view for making changes and is particularly useful for complex data.

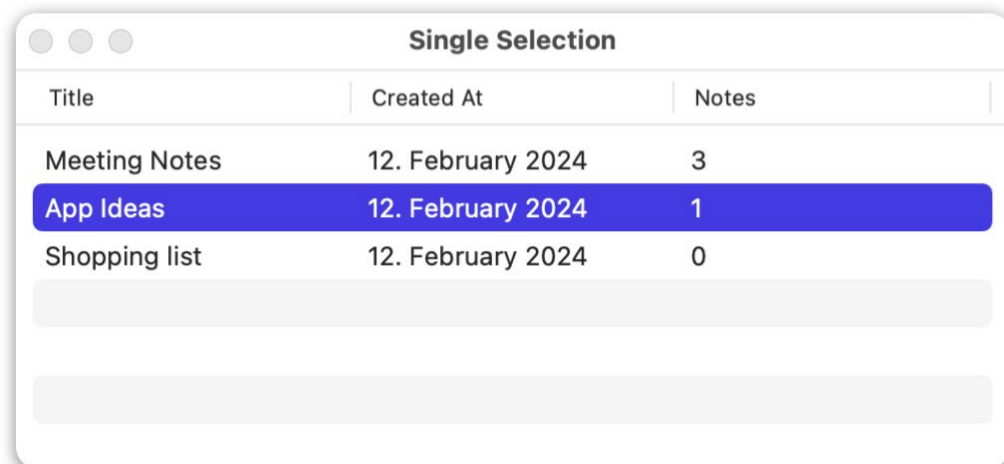
In the next lesson, I'll show you how to select rows in a table, you can then use these selected values to open a detail view.

11.6.4 Selecting Table Rows

Tables in SwiftUI allow you to select one or multiple rows, just like Lists. The mechanism for selection is very similar, utilizing the same selection and binding properties.

Single-Selection

Let's dive into an example. Imagine you're building an app with a table view that displays folders, and each folder contains notes.



Title	Created At	Notes
Meeting Notes	12. February 2024	3
App Ideas	12. February 2024	1
Shopping list	12. February 2024	0

You want to be able to select these folders within your table. In the following, I have a table showing an array of folders with 3 columns:

```
struct TableSelectionView: View {  
    @State private var folders = Folder.examples()  
  
    var body: some View {  
        Table(folders) {  
            TableColumn("Title", value: \Folder.name)  
            TableColumn("Created At") { folder in  
                Text(folder.creationDate, style: .date)  
            }  
            TableColumn("Notes") { folder in  
                Text(folder.notes.count, format: .number)  
            }  
        }  
    }  
}
```

Next, you'll need another state variable for the selected folder:

```
@State private var selectedFolderID: Folder.ID? = nil
```

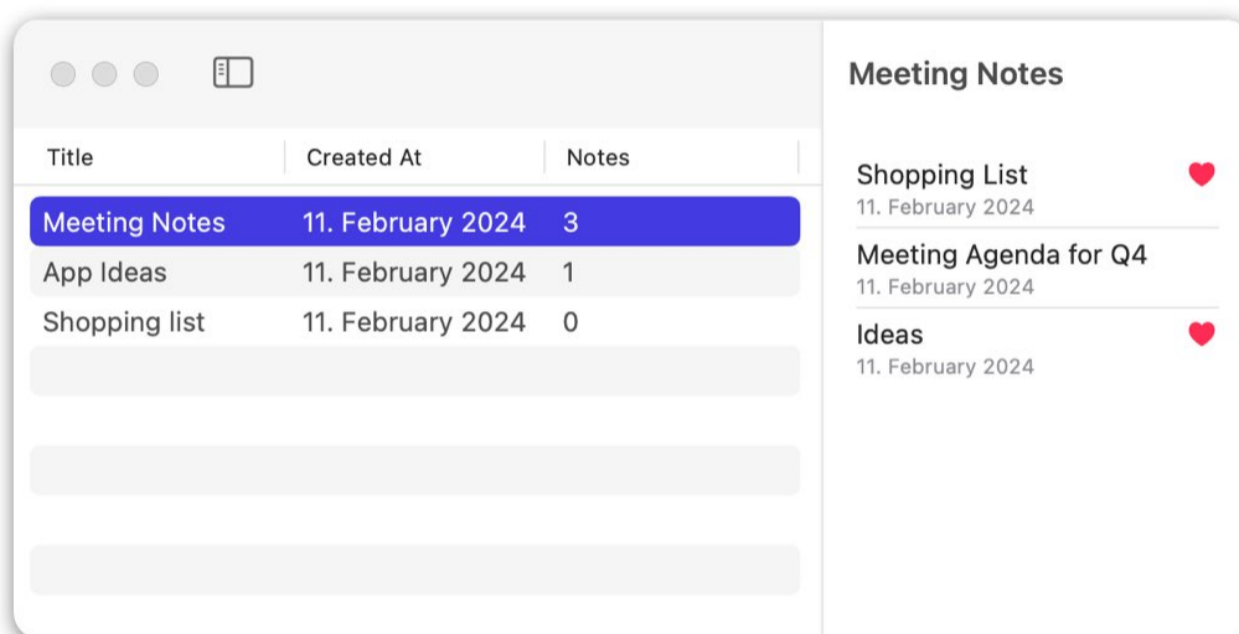
You must ensure that `selectedFolderID` is of the same type as the `id` property specified in the `Identifiable` protocol for your folder type. When you're working with a `Table`, it's important to use the correct type for

the ID. Unlike Lists, where you can use tags or the id property to specify selection, Tables manage everything automatically. When setting up your table, you'll bind it to the selectedFolderID:

```
Table(folders, selection: $selectedFolderID) {  
    ...  
}
```

Table with NavigationSplitView

With this setup, you can tap on a row to select a folder. This selection could then be used to display details in an inspector view, or perhaps in a navigation split view where the table is in the sidebar.



In the following, I am showing a table of folders in the sidebar:

```
struct TableSelectionView: View {  
  
    @State private var folders = Folder.examples()  
    @State private var selectedFolderID: Folder.ID? = nil  
  
    var body: some View {  
        NavigationSplitView {  
            Table(folders, selection: $selectedFolder) {  
                ...  
            }  
        } detail: {  
            // show notes for selected folder  
        }  
    }  
}
```

When the user selects a folder, I display a list of notes for this folder in a detail view:

```

if let selectedFolderID,
    let folder = folders.first(where: { $0.id == selectedFolderID }) {
    List(folder.notes) {
        NoteRowView(note: $0)
    }
    .navigationTitle(folder.name)
}

```

Multiple-Row Selection

If you want to enable multiple selections, you'll change the type of your selection state from a single Folder.ID to a Set<Folder.ID>:

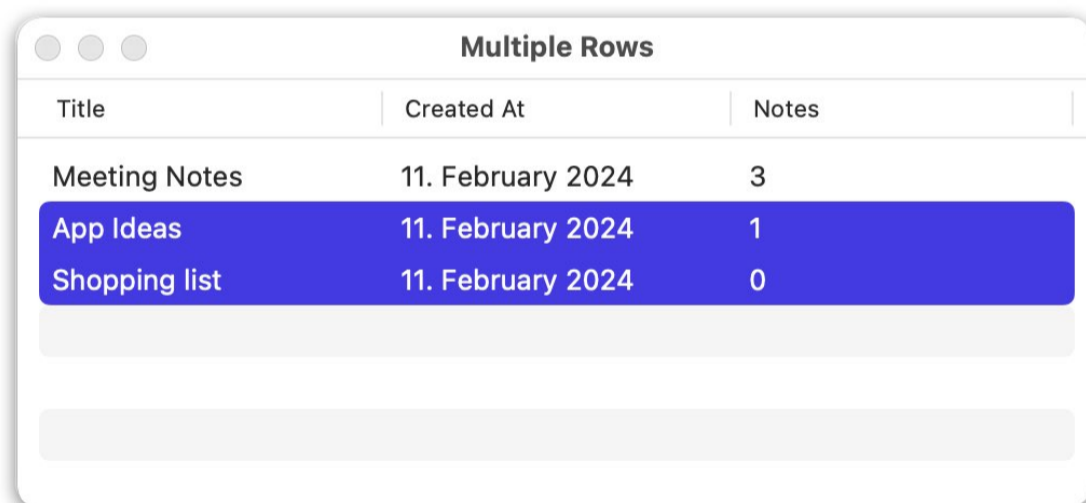
```

@State private var selectedFolders: Set<Folder.ID> = []

Table(folders, selection: $selectedFolderID) {
    ...
}

```

Then, you update your table to work with this new selection state. In the UI, you can select multiple folders by tapping on them while holding the shift key, for example. You could also add a context menu to perform actions with the selected items, like deleting multiple folders at once.



Title	Created At	Notes
Meeting Notes	11. February 2024	3
App Ideas	11. February 2024	1
Shopping list	11. February 2024	0

On iOS, you would need to use EditButton and editMode to allow multiple selections which is the same mechanism as for List.

11.6.5 Sorting and Filtering

When you dive into SwiftUI's Table, you'll quickly realize that sorting is one of its most impressive features. However, managing the data for sorting isn't as straightforward as you might hope, especially when you're not dealing with simple strings.

I am going to use the same Note data as in the previous sections:

```
struct TableSortingExampleView: View {
    @State private var notes = Note.examples()

    var body: some View {
        Table(notes) {
            TableColumn("Title", value: \Note.title)
            TableColumn("Content", value: \Note.content)

            TableColumn("Created At") { note in
                Text(note.creationDate, style: .date)
            }

            TableColumn("Favorite") { note in
                Image(systemName: "heart.fill")
                    .foregroundColor(note.isFavorite ? Color.red : Color.clear)
            }
        }
    }
}
```

First, you need to define a `@State private var sortOrder`s which is an array of `KeyPathComparator`. This array allows you to specify multiple sorting criteria. For instance, you might want to sort notes by whether they're marked as favorite and then by creation date or title if there's a tie.

```
@State private var sortOrder = [KeyPathComparator(\Note.title,
                                                    order: .reverse)]
```

In your Table, you bind the `sortOrder` to this state variable:

```
Table(note, sortOrder: $sortOrder) {
    ...
}
```

When you tap on the table's header, you'll see an arrow indicating that you can sort the items. However, the Table doesn't sort the data for you; it merely reflects the order of the data in your notes array. The `sortOrder`s binding captures the user's intent, but you need to sort the notes array yourself.

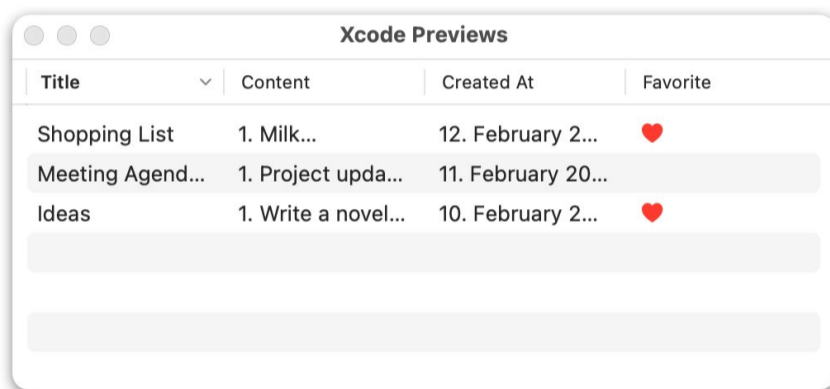
Handling User Interaction for Sorting

To react to sorting changes, you'll use the `.onChange` modifier on your `sortOrders`. When the user interacts with the table's header, the `sortOrders` value changes, and you can then sort your notes array accordingly.

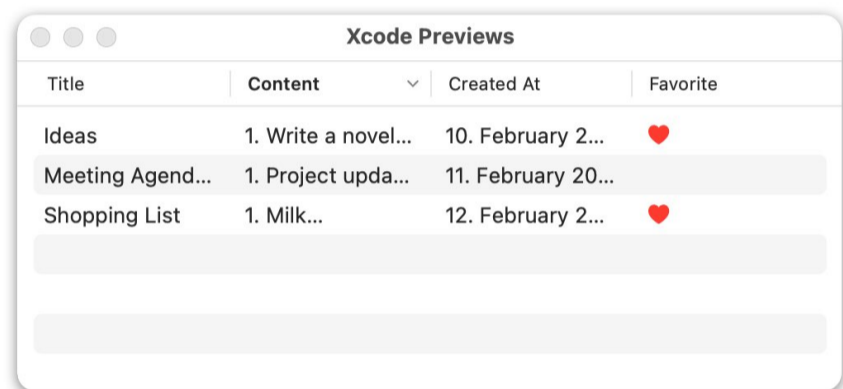
```
Table(note, sortOrder: $sortOrders) {  
  ""  
}  
.onChange(of: sortOrders) { oldValue, newValue in  
  notes.sort(using: newValue)  
}
```

Keep in mind that your notes array must be mutable, so you should declare it as `@State`.

When you run your app and tap on a header, your list will now change its sorting order, allowing you to sort ascending or descending based on the property you tapped.



Title	Content	Created At	Favorite
Shopping List	1. Milk...	12. February 2...	♥
Meeting Agend...	1. Project upda...	11. February 20...	
Ideas	1. Write a novel...	10. February 2...	♥



Title	Content	Created At	Favorite
Ideas	1. Write a novel...	10. February 2...	♥
Meeting Agend...	1. Project upda...	11. February 20...	
Shopping List	1. Milk...	12. February 2...	♥

Sorting by Date

When it comes to sorting notes in a table, one of the most common requirements is to sort by the creation date. You might want your users to easily view the most recent notes or perhaps the oldest ones first. Here's how you can implement sorting by creation date within your Table:

```
TableColumn("Created At",  
  value: \Note.creationDate) { note in  
  Text(note.creationDate, style: .date)  
}
```

The `TableColumn` initializer takes a **value parameter**, which is a key path to the property you want to sort by—in this case, `\Note.creationDate`.

When a user taps on the header of the creation date column, the `sortOrders` binding will update and you can update the notes array.

Title	Content	Created At	Favorite
Ideas	1. Write a novel...	10. February 2...	♥
Meeting Agend...	1. Project upda...	11. February 20...	
Shopping List	1. Milk...	12. February 2...	♥

Title	Content	Created At	Favorite
Shopping List	1. Milk...	12. February 2...	♥
Meeting Agend...	1. Project upda...	11. February 20...	
Ideas	1. Write a novel...	10. February 2...	♥

Sorting Non-String Values

When it comes to sorting non-string values in a Table, you might run into some challenges. SwiftUI's Table prefers to sort using strings because it relies on alphabetical order. However, what if you have Boolean values or custom types, like enums? Let's look at how you can work around these limitations.

For Boolean values, SwiftUI doesn't natively know how to sort them in a Table. You can't directly use a Boolean property in the value parameter of a Table column because SwiftUI won't know how to handle the sorting interaction. The trick is to convert these Boolean values into strings.

Here's how you can create a computed property to convert a Boolean into a string for sorting purposes:

```
@Observable class Note: Identifiable {

    var title: String
    var isFavorite: Bool
    let creationDate: Date
    var colorTag: Color
    var content: String

    var isFavoriteFormatted: String {
        isFavorite ? "favorite" : "not"
    }
}
```

And use it to create a TableColumn with a value property that handles the sorting selection:

```
TableColumn("Favorite",
            value: \Note.isFavoriteFormatted) { note in
    Image(systemName: "heart.fill")
        .foregroundColor(note.isFavorite ? Color.red : Color.clear)
}
```

This approach doesn't affect what's displayed in your Table—it's strictly for sorting. The actual values ("Favorite" and "Not Favorite") are not shown to the user but are used internally by SwiftUI to sort the notes based on their favorite status.

If you have an enum or a custom type, you can use a similar strategy by mapping these to string values. For enums, you can leverage the raw value if it's a string or use a computed property to return a string representation:

```
enum NoteCategory: String {
    case work
    case personal
    case ideas
}
```

```
@Observable class Note: Identifiable {
    ...
    var category: NoteCategory

    var categoryAsString: String {
        return category.rawValue
    }
}
```

and use this computed property for the TableColumn value parameter:

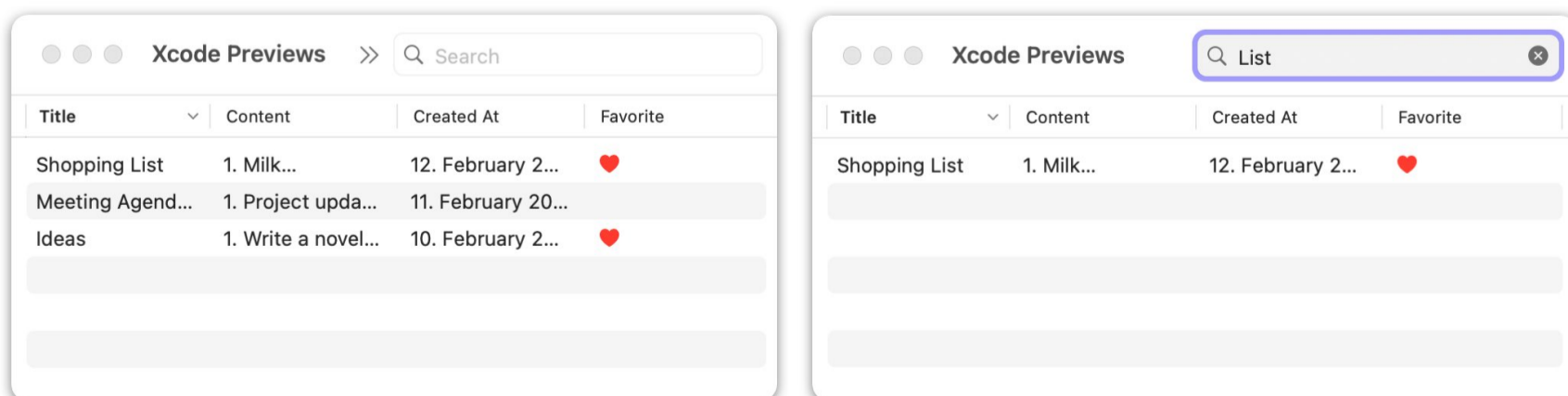
```
TableColumn("Category", value: \.categoryAsString)
```

By providing these string representations, you ensure that the sorting mechanism in SwiftUI's Table can understand and correctly sort your data. Remember to create these string representations in a way that reflects the intended sorting order, as they will be sorted alphabetically.

In conclusion, when dealing with non-string values for sorting in a Table, you'll often need to bridge the gap by providing string representations that SwiftUI can use for sorting. With these workarounds, you can sort practically any type of data in your Table.

Filtering With Table

Filtering is another feature that depends on your notes array. You could introduce a search text field and use the `.searchable` modifier to update the notes array whenever the search text changes.



First, I declare a state property for the search property and use the searchable modifier to show the search textfield in the toolbar:

```
@State private var searchText = ""

Table(notes, sortOrder: $sortOrders) {
    ...
}
.searchable(text: $searchText)
```

To manage both sorting and filtering, you might have allNotes and filteredAndSortedNotes arrays. You would filter and sort allNotes to get filteredAndSortedNotes, which is what you display in your table.

Here's a basic example of handling a search text change:

```
struct TableSortingExampleView: View {

    @State private var searchText = ""
    @State private var sortOrders = [KeyPathComparator(\Note.title, order: .reverse)]
    @State private var notes = Note.examples()

    private var filteredAndSortedNotes: [Note] {
        var result = [Note]()

        if !searchText.isEmpty {
            result = notes.filter { $0.title.contains(searchText) }
        } else {
            result = notes
        }

        result.sort(using: sortOrders)
        return result
    }

    var body: some View {
        Table(filteredAndSortedNotes, sortOrder: $sortOrders) {
            ...
        }
        .searchable(text: $searchText)
    }
}
```

If your data arrays are large, avoid doing these array operations for filtering, as it can slow down your app. Instead, consider using Swift's observation features or Core Data, which provide more efficient ways to handle large datasets and perform filtering and searching operations.

That covers sorting and filtering in a nutshell. If you're looking to implement more complex searching and filtering, like using search tokens or scopes, here are some additional resources:

- [How to use Search Scope in SwiftUI to improve search on iOS and macOS](#)
- [Search Tokens in SwiftUI: How to implement advanced search in iOS and macOS](#)

11.6.6 DisclosureTableRow

When you're dealing with complex data structures in your SwiftUI app, you might want to organize your data with more hierarchy. Tables in SwiftUI can be used for this purpose, especially when you want to create sections using disclosure groups (macOS 14, iOS 17+).

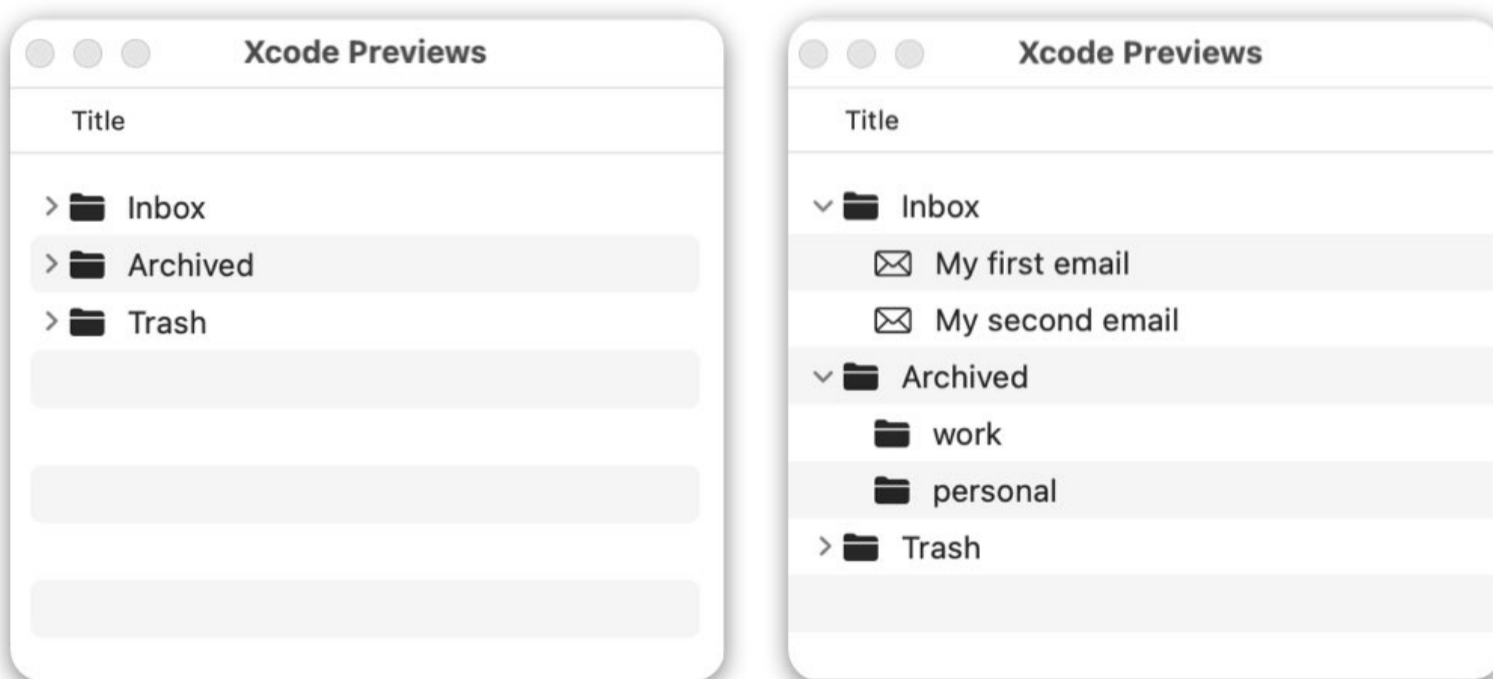
However, you need to be aware that tables are designed to work with consistent data types across all rows due to their columnar nature. I cannot for example use them for the Folder and Note data from the previous sections.

Working with Complex Data Types

Imagine you have a set of items, such as files and folders in a file system. You want to represent these in a table where you can expand and collapse sections to show or hide the contents of a folder.

```
@Observable class FileItem: Identifiable {  
  
    var title: String  
    let isFolder: Bool  
    var children: [FileItem]? = nil  
    let id = UUID()  
}
```

Here, FileItem is a data type that represents either a file or a folder, identifiable with an id, and has an optional children array for folders to hold their nested contents.



Implementing Disclosure Groups in Tables

To create a table with disclosure groups, you'll need to initialize the table with both columns and rows. You can define the columns based on the properties of your data type.

```
struct TableDisclosureView: View {
    let fileItems = FileItem.preview()
    var body: some View {
        Table(of: FileItem.self) {
            TableColumn("Title") { item in
                Label(item.title,
                    systemImage: item.isFolder ? "folder.fill" : "envelope")
            }
        } rows: {
            ForEach(fileItems) { item in
                DisclosureTableRow(item) {
                    ForEach(item.children ?? []) { subItem in
                        TableRow(subItem)
                    }
                }
            }
        }
    }
}
```

In the example above, you use a `ForEach` to iterate over the `fileItems`, and for each item, you create a `DisclosureGroup`. If the item has children, you iterate over them to create individual rows.

Limitations of Tables with Hierarchical Data

While tables can be useful for creating hierarchical views, they come with limitations. Since tables expect the same data type for all rows, you can't easily represent different types of data in different rows if they need to be displayed in a specific columnar format.

For example, if you're trying to represent a notes and folders structure similar to a note-taking app, it might not be the best fit for a table. The table expects each row to have the same columns, which can be restrictive if your notes and folders have different properties.

Example: Table of Tasks with Subtasks

A good example for `Table` with collapsible sections is a list of tasks with subtasks. The `Task` data has information about the name, priority and due date of the task. You could also split up a task into multiple subtasks:

```
struct Task: Identifiable {
    let id = UUID()
    var name: String
    var subTasks: [Task] = []
    var isDone: Bool = false
    var priority: Priority
    var dueDate: Date = Date()
}
```

name	is Done	Priority	Due Date
> This is import...	<input type="radio"/>	neutral	15. May 2024
> Publish App...	<input checked="" type="radio"/>	neutral	11. April 2024
> Grocery Shop...	<input type="radio"/>	urgent	12. February 2...

name	is Done	Priority	Due Date
∨ This is important	<input type="radio"/>	neutral	31. March 2024
Play with cats	<input type="radio"/>	backlog	12. February 2024
∨ Publish App Up...	<input checked="" type="radio"/>	neutral	18. May 2024
bug fix	<input checked="" type="radio"/>	backlog	12. February 2024
check in app...	<input checked="" type="radio"/>	urgent	12. February 2024
∨ Grocery Shoppi...	<input type="radio"/>	urgent	12. February 2024
Buy Bread	<input type="radio"/>	urgent	12. February 2024
Pet food	<input type="radio"/>	urgent	12. February 2024

This data can be a good fit to show in a Table, where we see all the different properties like due date in a dedicated column:

```

struct TaskTableSectionsView: View {

    @State private var tasks = Task.examples()
    @State private var sortOrders = [KeyPathComparator(\Task.name, order: .forward)]

    var body: some View {
        Table(of: Task.self, sortOrder: $sortOrders) {
            TableColumn("name", value: \.name)
            TableColumn("is Done") { task in
                Image(systemName: task.isDone ? "checkmark.circle.fill" : "circle")
                    .foregroundColor(task.isDone ? Color.accentColor : Color.gray)
            }

            TableColumn("Priority") { task in
                Text(task.priority.rawValue)
                    .bold()
                    .foregroundColor(task.priority.color)
            }

            TableColumn("Due Date", value: \Task.dueDate) { task in
                Text(task.dueDate, style: .date)
            }
        } rows: {
            ForEach(tasks) { task in
                DisclosureTableRow(task) {
                    ForEach(task.subTasks) { subTask in
                        TableRow(subTask)
                    }
                }
            }
        }
        .onChange(of: sortOrders) { oldValue, newValue in
            tasks.sort(using: newValue)
        }
    }
}

```

11.6.7 Cross-platform

In previous lessons, I've focused on showcasing tables within macOS and their distinctive look. However, it's time to expand our horizons and see how these tables appear on iOS and iPadOS. When you're working with an iPad in full screen, you can achieve a look that's quite similar to macOS. Let's dive in.

Setting Up a Cross-Platform Table View

For this example, we'll use a basic table example with the notes data:

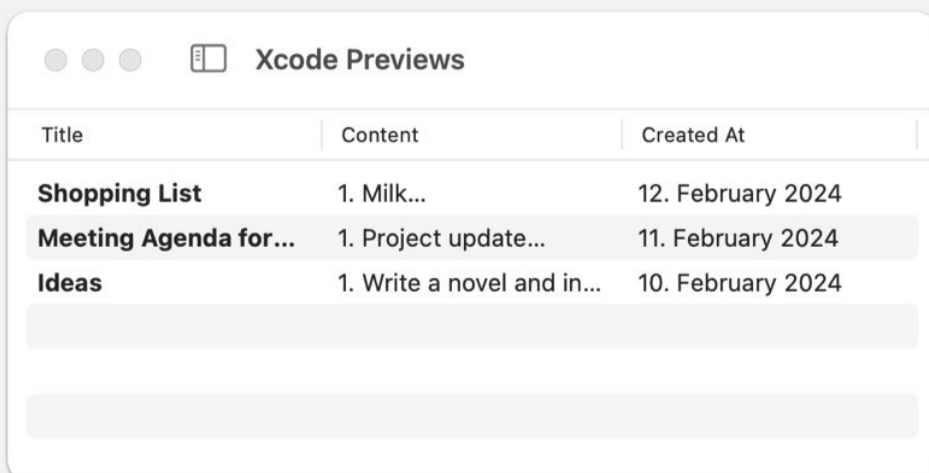
```
let notes = Note.examples()

Table(notes) {
    TableColumn("Title") { note in
        Text(note.title)
        .font(.headline)
    }

    TableColumn("Content", value: \Note.content)
    TableColumn("Created At") { note in
        Text(note.creationDate, style: .date)
    }
}
```

The appearance of the table on iPad depends on where it is used and in particular the environment value for horizontal size class. To demonstrate the difference I will place the same table inside the sidebar and content of a NavigationSplitView:

```
struct CrossplatformTableView: View {
    var body: some View {
        NavigationSplitView {
            AdaptedNoteTableView()
        } detail: {
            AdaptedNoteTableView()
        }
    }
}
```

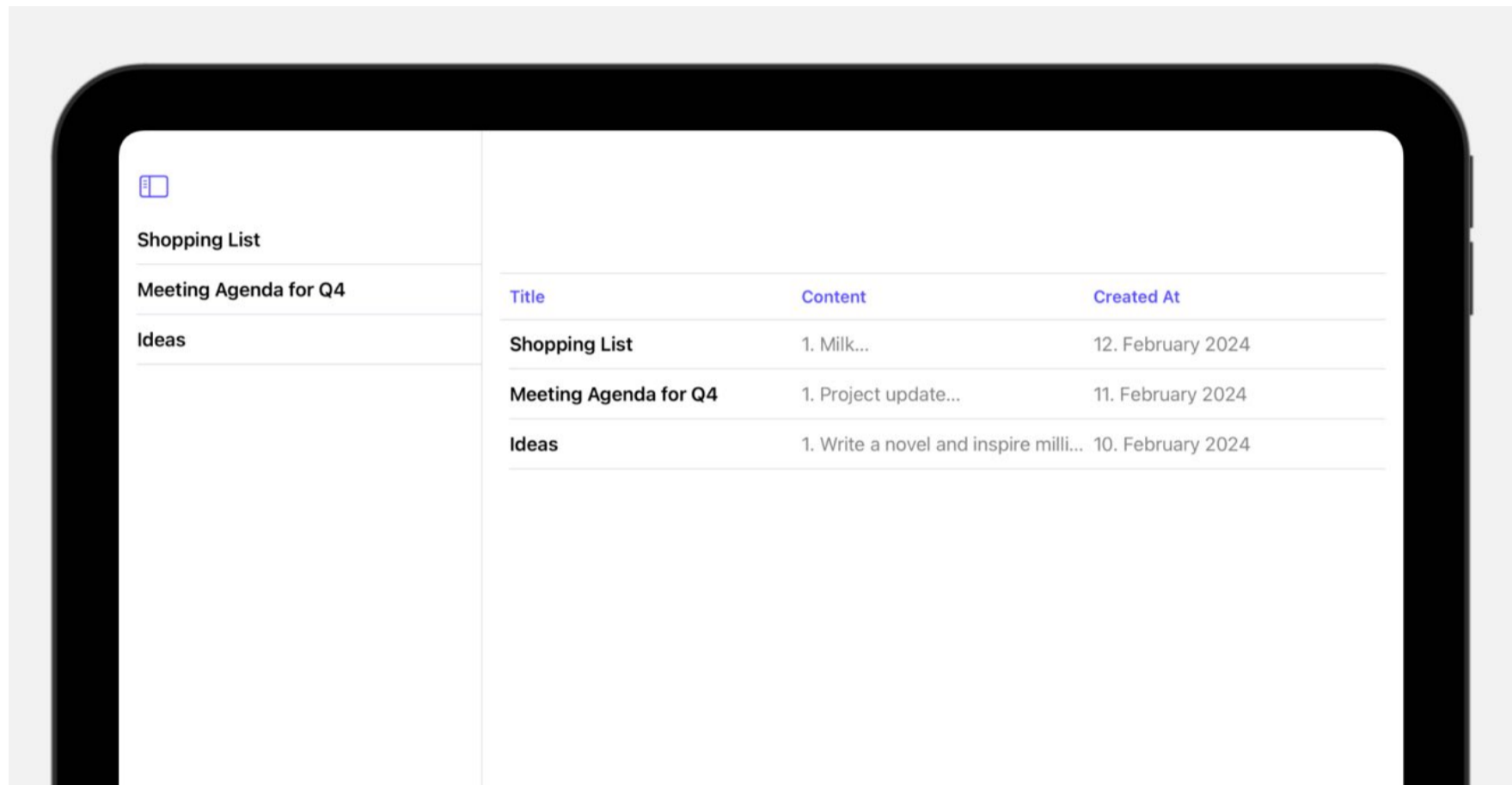


Title	Content	Created At
Shopping List	1. Milk...	12. February 2024
Meeting Agenda for...	1. Project update...	11. February 2024
Ideas	1. Write a novel and in...	10. February 2024



On the Mac, all table columns are shown. On the iPhone, which uses compact mode, the table collapses and only the first table column is shown. In the above screenshot, you can see the list of note titles. The table looks the same as if I would have used a SwiftUI List.

On the iPad in landscape mode, you can see the sidebar and content of the NavigationSplitView simultaneously. In the sidebar, SwiftUI uses a compact size class and the table is shown as a list. In the content, where there is ample space, the size class is regular and all columns are shown in a Table layout:



Adapting the Table for Compact Mode

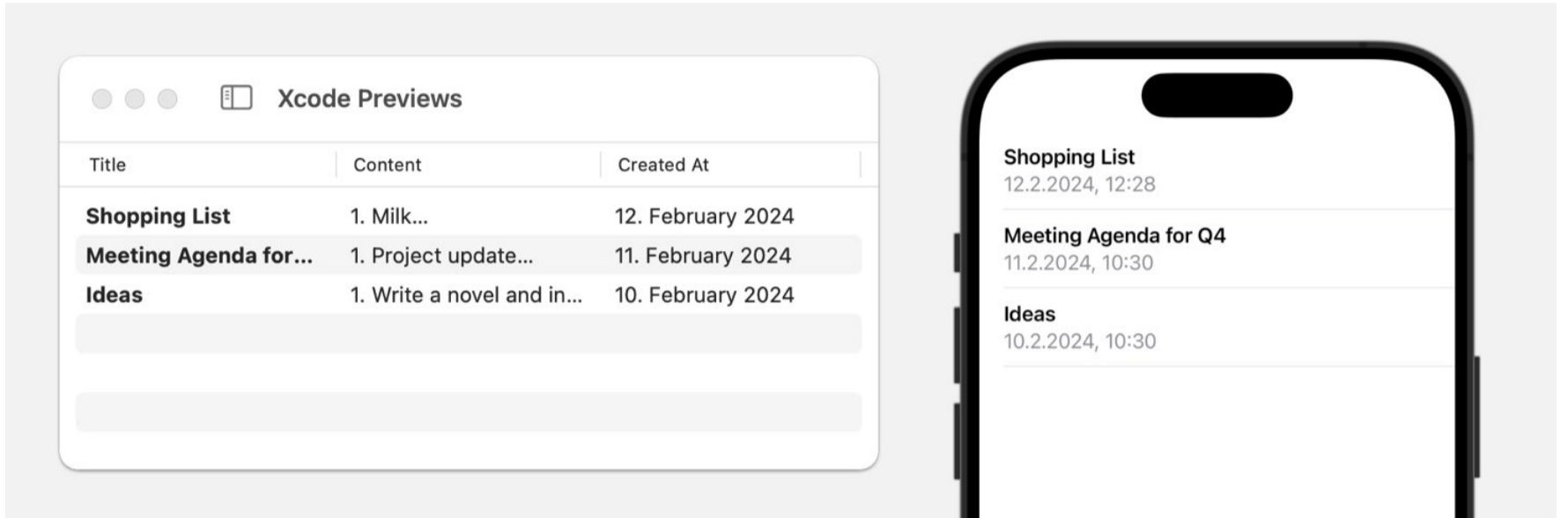
I want to adapt the table for the iPhone that means for compact mode. The key property to work with here is the `horizontalSizeClass`, which is available from the `@Environment`. By checking the `horizontalSizeClass`, you can tailor the content for the first column. If the size class is compact, you'll know the screen will only show this column. In this case, I am showing just the note's title and timestamp:

```
struct AdaptedNoteTableView: View {  
  
    let notes = Note.examples()  
    @Environment(\.horizontalSizeClass) var horizontalSizeClass  
  
    var body: some View {  
        Table(notes) {  
            TableColumn("Title") { note in  
                VStack(alignment: .leading) {  
                    Text(note.title)  
                        .font(.headline)  
                    if horizontalSizeClass == .compact {  
                        Text(note.creationDate, format: .dateTime)  
                            .foregroundColor(.gray)  
                    }  
                }  
            }  
        }  
    }  
}
```

```

    }
  }
  TableColumn("Content", value: \Note.content)
  TableColumn("Created At") { note in
    Text(note.creationDate, style: .date)
  }
}
}
}
}
}
}

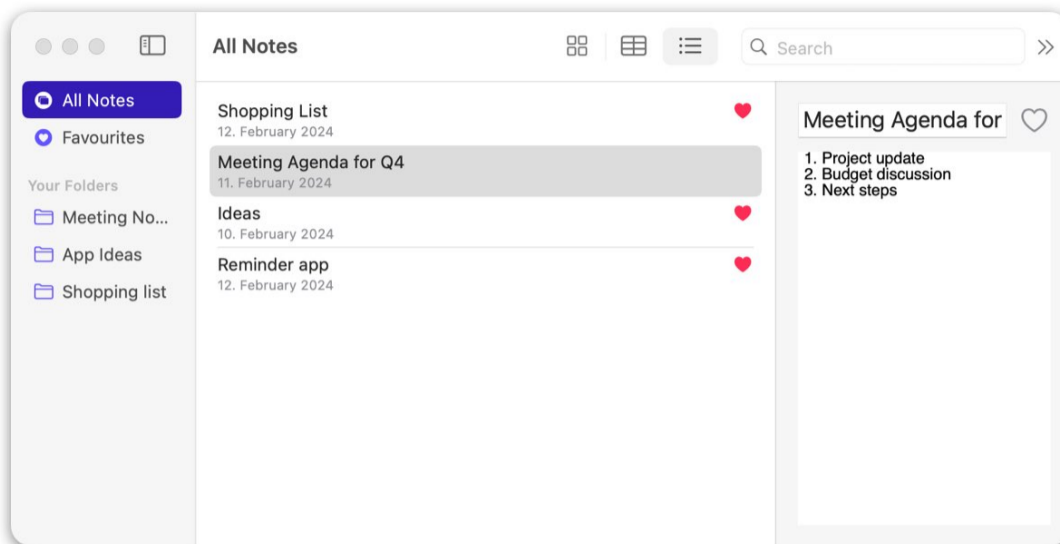
```



This approach is quite elegant because, on an iPhone, the table looks and feels like a list, which is a familiar interface element for iPhone users. By leveraging the `horizontalSizeClass`, you can create a single SwiftUI view that intelligently adapts to the device it's on, ensuring your app provides an optimal user experience whether on an iPhone, iPad, or Mac.

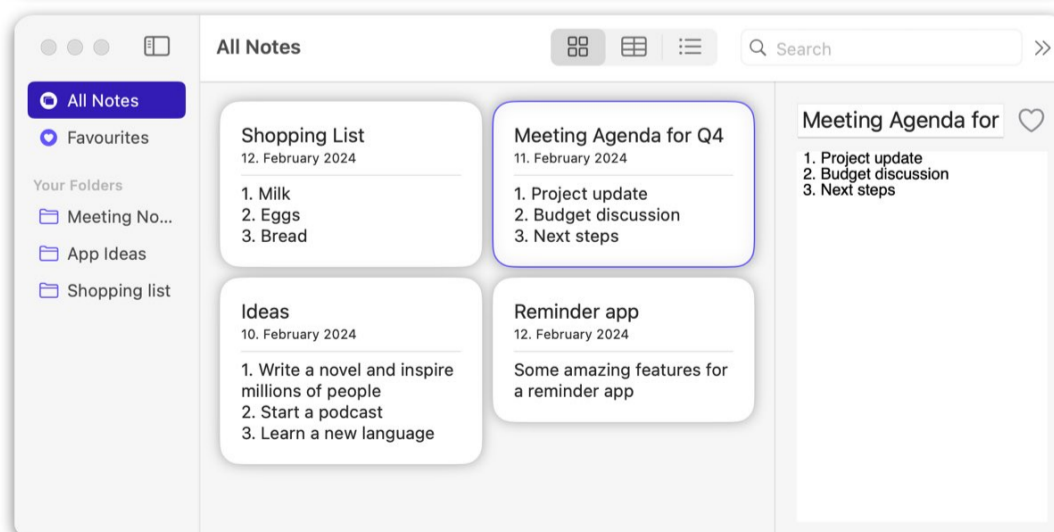
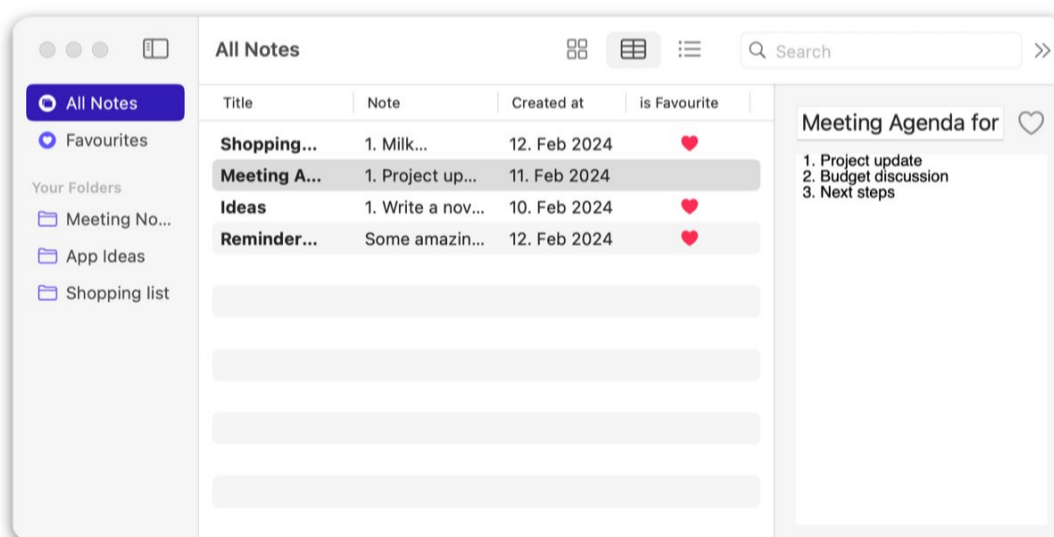
Challenge 🙌 Layout Variations

It's time for another challenge. This time, I want you to take the notes folder app you've already created to the next level. We've been using a list for the sidebar, but now I'd like you to introduce the ability for the user to switch between different layouts for the content area, where you display the notes.



Your task is to add a picker to the toolbar that allows switching between a list, a table, and a grid layout. When the user selects "Table" from the picker, they should see the notes arranged in a table format. Remember, your app should still handle selection properly across different layouts.

If you're a bit rusty on implementing lazy grids, particularly adaptive ones, this is an excellent opportunity to review that material. The main focus, however, is to implement the table view for the notes and add the toolbar picker to enable layout switching.



Solution Walkthrough

Okay, let me walk you through one possible solution. In the main view, I created a picker with an enum for the display styles: grid, table, and list. I used a computed property to determine which icon to show for each style.

```
enum DisplayStyle: CaseIterable, Identifiable {
    case grid
    case table
    case list

    var icon: String {
        switch self {
            case .grid: return "square.grid.2x2"
            case .table: return "tablecells"
            case .list: return "list.bullet"
        }
    }
    var id: Self { return self }
}
```

This property is then used in a @State property for the picker. The picker is added to the toolbar with a segmented display style.

```
struct NavigationListView: View {
    ...
    @State private var displayStyle = DisplayStyle.table

    var body: some View {
        NavigationSplitView {
            FolderSelectionListView(folders: $folders,
                                   selection: $folderSelection.animation())
                .toolbar(content: {
                    ToolbarItem(placement: .secondaryAction) {
                        Picker("", selection: $displayStyle) {
                            ForEach(DisplayStyle.allCases) {
                                Image(systemName: $0.icon)
                            }
                        }
                    }
                })
                .pickerStyle(.segmented)
        } content: {
            // conditionally check which layout to show from grid, table or list
        } detail: {
            ...
        }
    }
}
```

Within the content view, I used a switch case to determine which layout to display based on the selected display style. Regardless of the style, I always passed the correct data to the views and maintained the bindings to the selected note. This way, even if the user switches layouts, the selected note remains consistent.

```

NavigationSplitView {
    ...
} content: {
    if let folderSelection {
        Group {
            switch displayStyle {
                case .grid:
                    NoteSelectionGridView(notes: notes(),
                                           selectedNote: $selectedNote)
                case .table:
                    NoteSelectionTableView(notes: notes(),
                                           selectedNote: $selectedNote)
                case .list:
                    NoteSelectionListView(notes: notes(),
                                          selectedNote: $selectedNote)
            }
        }
    } else {
        ContentUnavailableView("Please select a folder",
                               systemImage: "folder")
    }
} detail: {
    ...
}

```

I created two separate files for the table and grid versions. I used a similar structure to the existing `NoteSelectionListView`. The view requires two parameters: an array of notes and a binding to the selected note:

```

struct NoteSelectionListView: View {
    let notes: [Note]
    @Binding var selectedNote: Note?

    var body: some View {
        ""
    }
}

```

```

struct NoteSelectionTableView: View {
    let notes: [Note]
    @Binding var selectedNote: Note?

    var body: some View {
        ""
    }
}

```

```

struct NoteSelectionGridView: View {
    let notes: [Note]
    @Binding var selectedNote: Note?

    var body: some View {
        ""
    }
}

```

For the table layout, I used 4 columns for the notes title, content, creation date and isFavorite. Because Table takes a binding to Note.ID for the section, but I want to handle this data with Note types, I am creating a custom binding in the body property. This converts from Note to Note.ID back and forward:

```

struct NoteSelectionTableView: View {
    let notes: [Note]
    @Binding var selectedNote: Note?

    var body: some View {
        let binding = Binding(
            get: { self.selectedNote?.id },
            set: { id in
                if let note = notes.first(where: { $0.id == id }) {
                    self.selectedNote = note
                } else {
                    self.selectedNote = nil
                }
            }
        )

        return Table(notes, selection: binding) {
            ""
        }
    }
}

```

For the grid layout, I used adaptive GridItems to create a responsive design, which works particularly well on macOS where window sizes can vary significantly. I wrapped each note in a VStack and adjusted the frames to get the backgrounds to display correctly. I also aligned the grid items to the top for consistency.

```

struct NoteSelectionGridView: View {

    let notes: [Note]
    @Binding var selectedNote: Note?

    let columns = [GridItem(.adaptive(minimum: 150, maximum: 300),
                             alignment: .top)]

    var body: some View {
        ScrollView {
            LazyVGrid(columns: columns, content: {
                ForEach(notes) { note in
                    VStack(alignment: .leading) {
                        Text(note.title)
                            .font(.title3)
                        Text(note.creationDate, style: .date)
                            .font(.caption)
                        Divider()
                        Text(note.content)
                    }
                    .frame(maxWidth: .infinity, alignment: .leading)
                    .padding()
                    .background(
                        RoundedRectangle(cornerRadius: 15)
                            .fill(Color.white)
                            .stroke(selectedNote == note ? Color.accentColor :
                                Color.clear,
                                    lineWidth: 1)
                            .shadow(radius: 5))
                }
            })
        }
    }
}

```

