

App Structure

Lecture 5

CIS 1951

<https://github.com/cis1951/lec5-code>



Previously, on CIS 1951...

SwiftUI State Management

@State

@Observable

@Binding

.onDisappear

.onAppear

.onChange

**So far, we've only made simple,
single-screen apps.**

That changes today.

This week

The tools you need to create larger apps

Navigation & modal presentations

MVVM

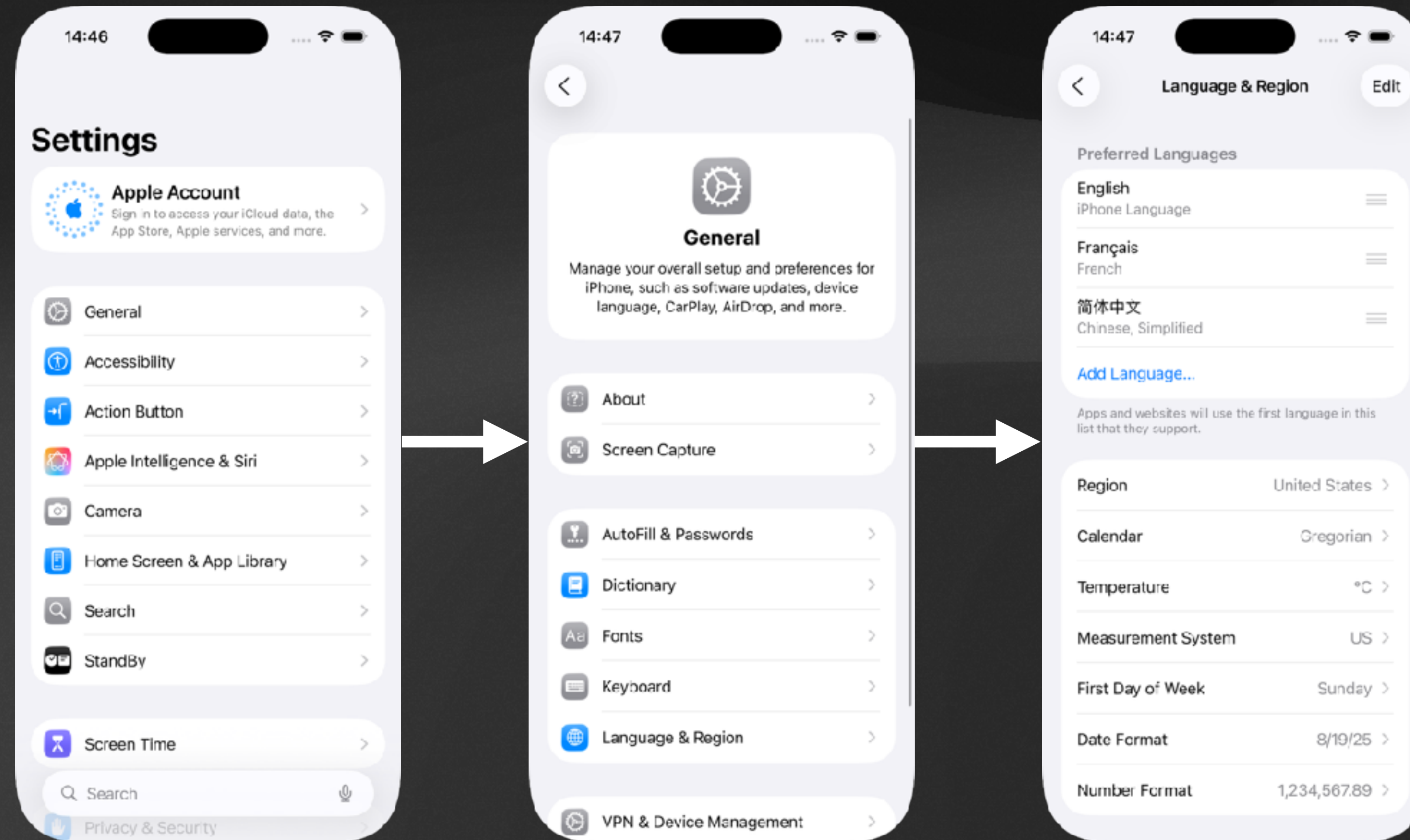
@Observable, @Bindable, @Environment

Navigation & Modal Presentations

How do we organize multiple
screens?

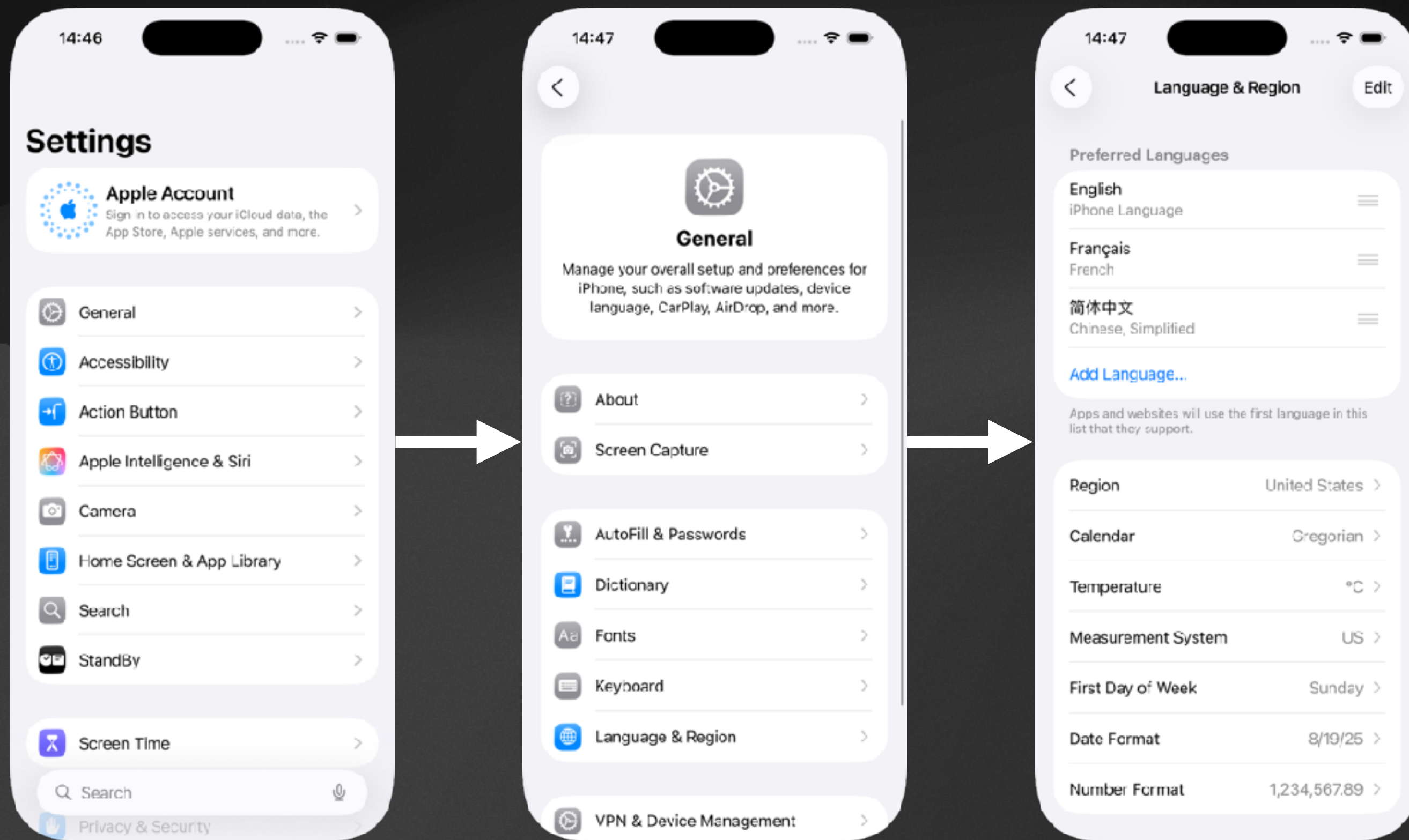
Hierarchical Navigation

Example: Settings App



Hierarchical Navigation

Example: Settings App



push

pop

Language & Region

General

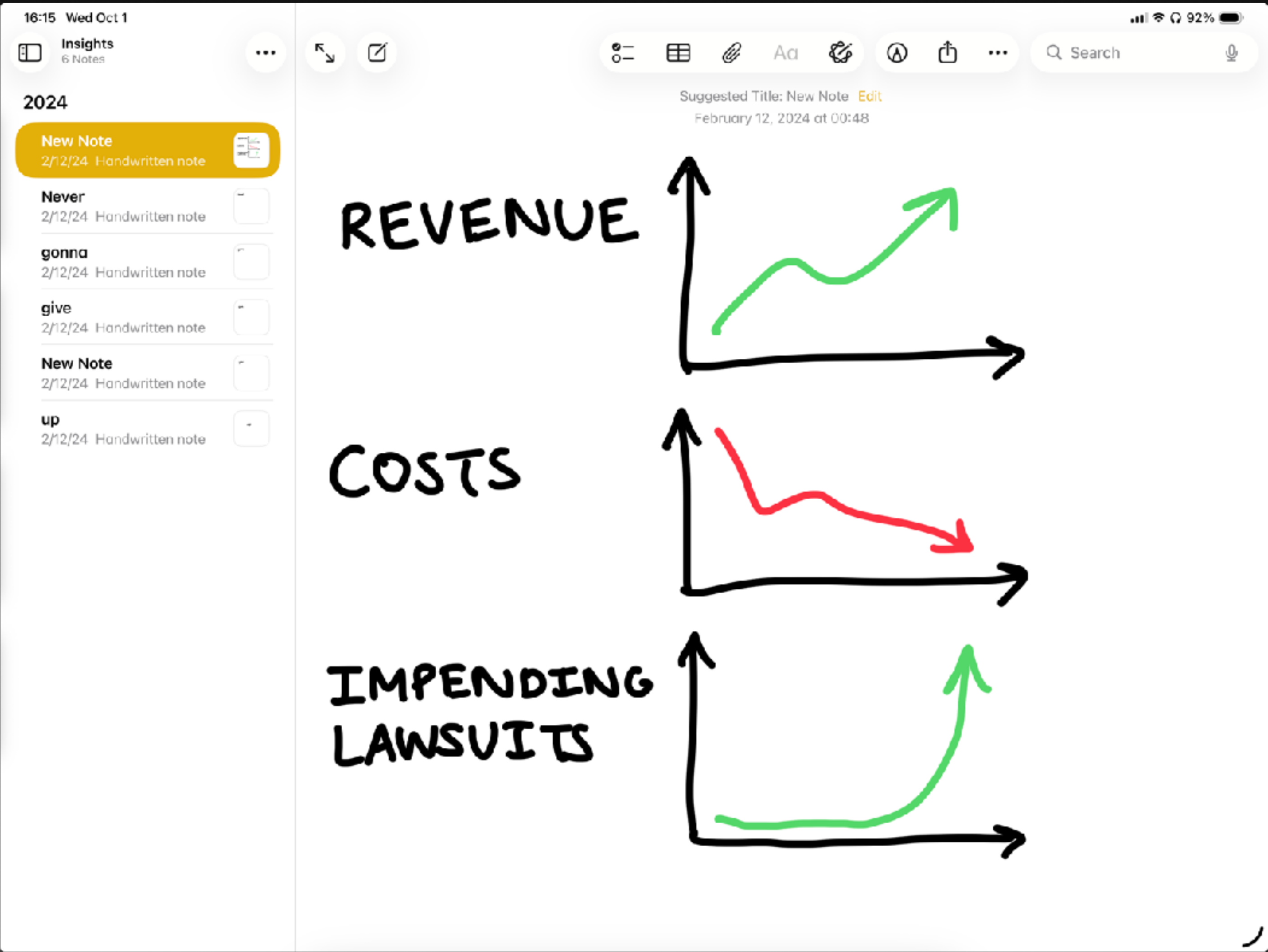
Settings

Master-Detail Navigation

Example: Notes App

Master

List of notes



Detail

Single note

Tab Bar Navigation

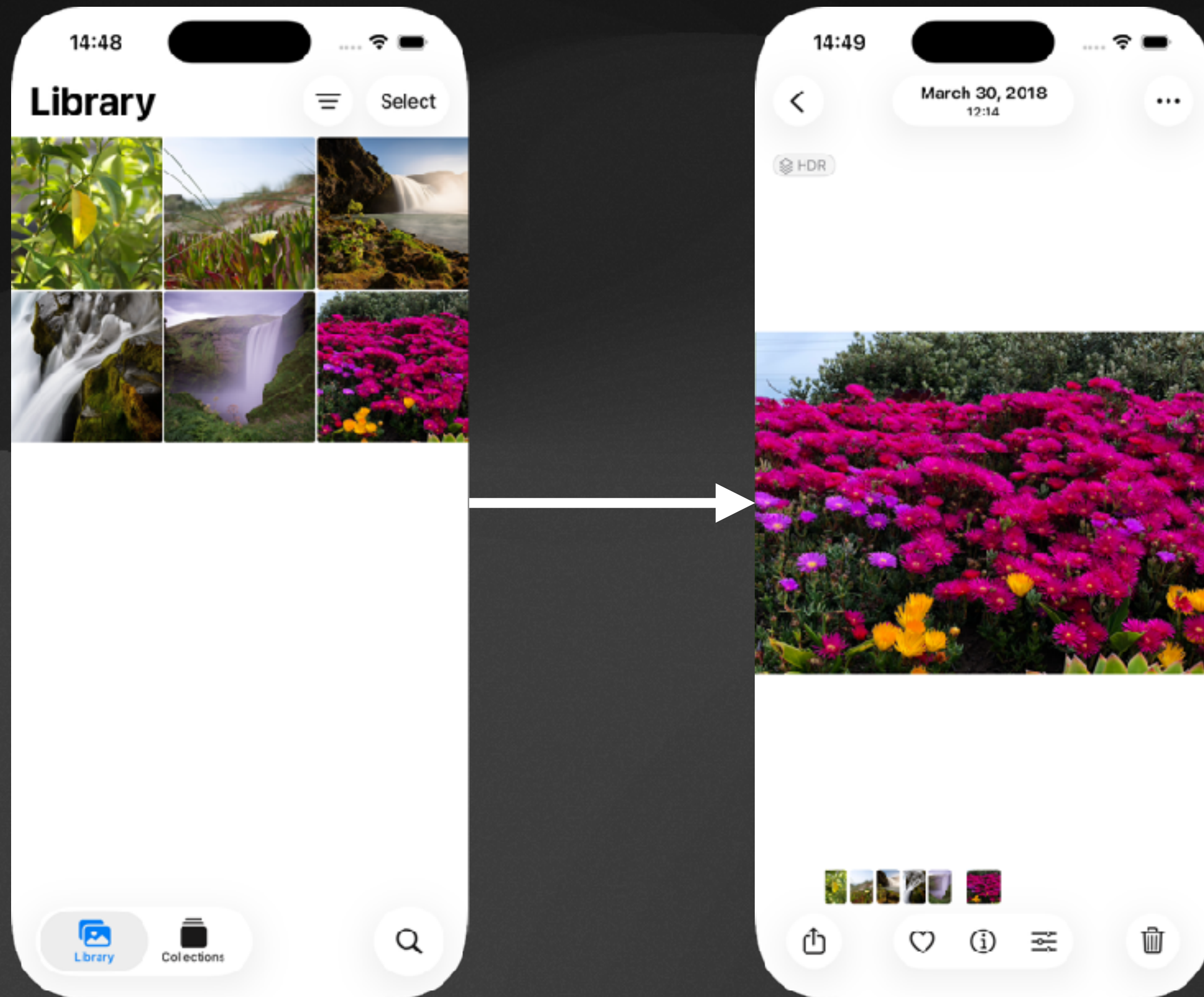
Example: Clock App



Bottom/top bar for quick navigation

Hybrid Navigation

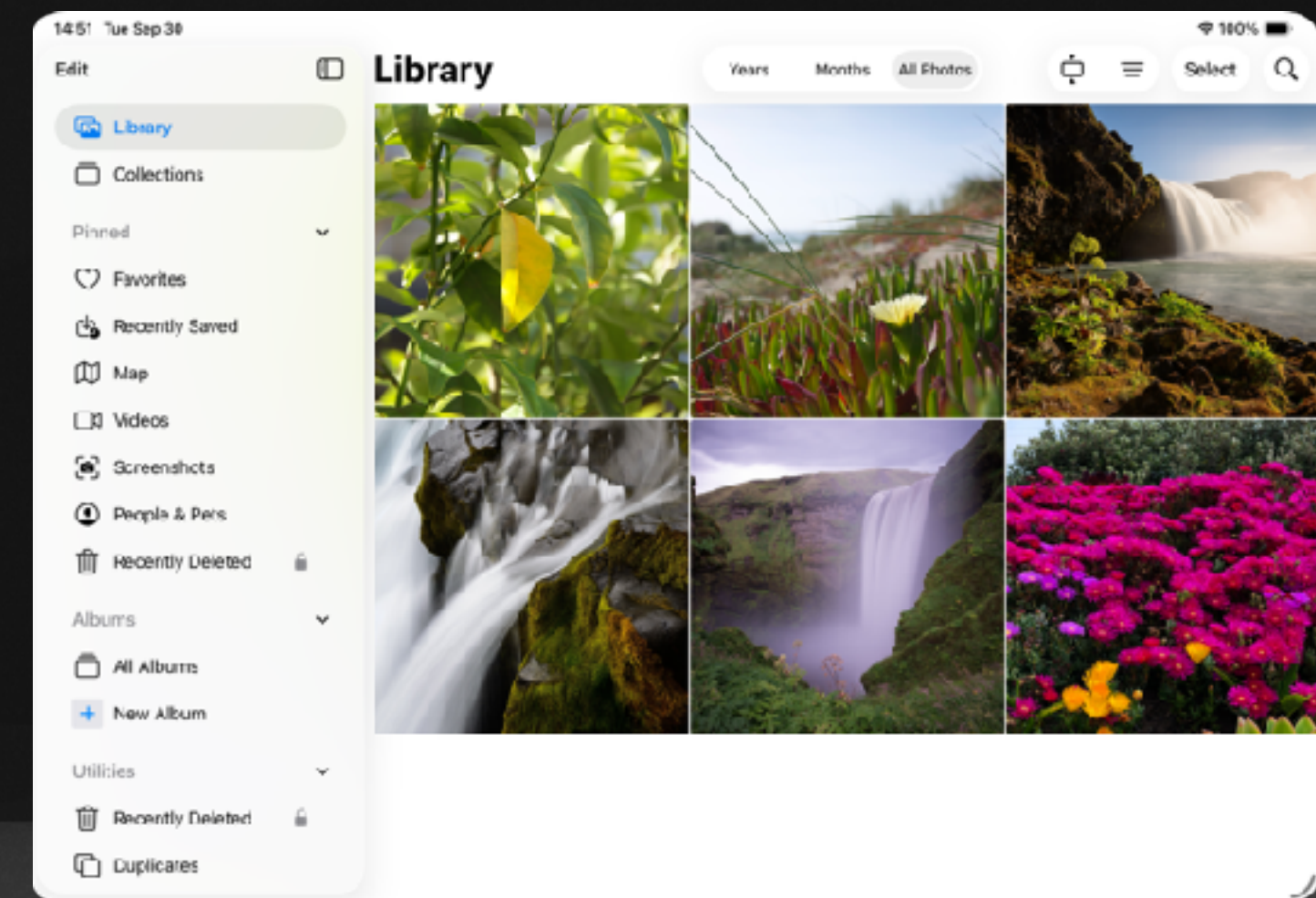
Example: Photos App



Tab bar

Hierarchical

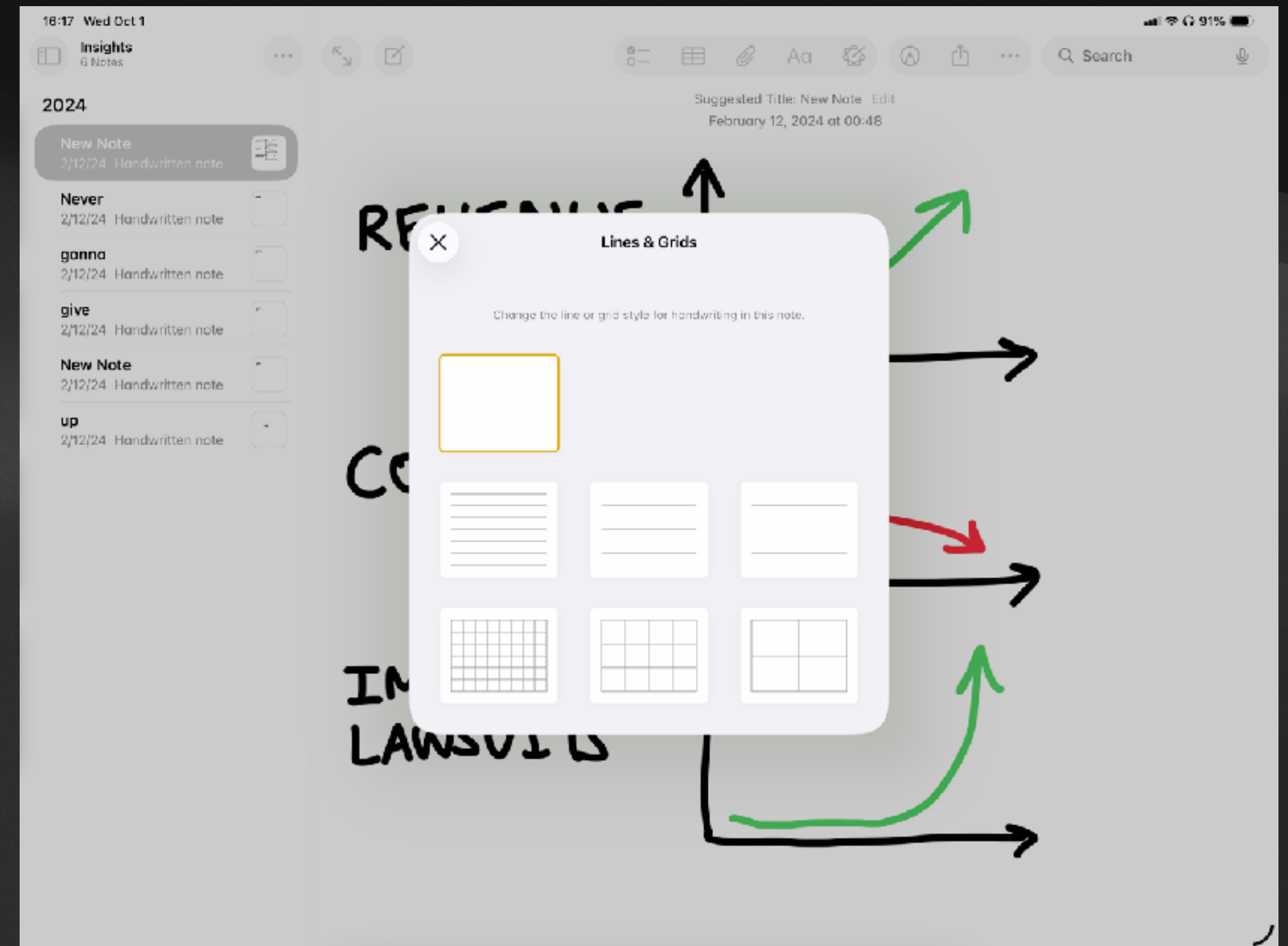
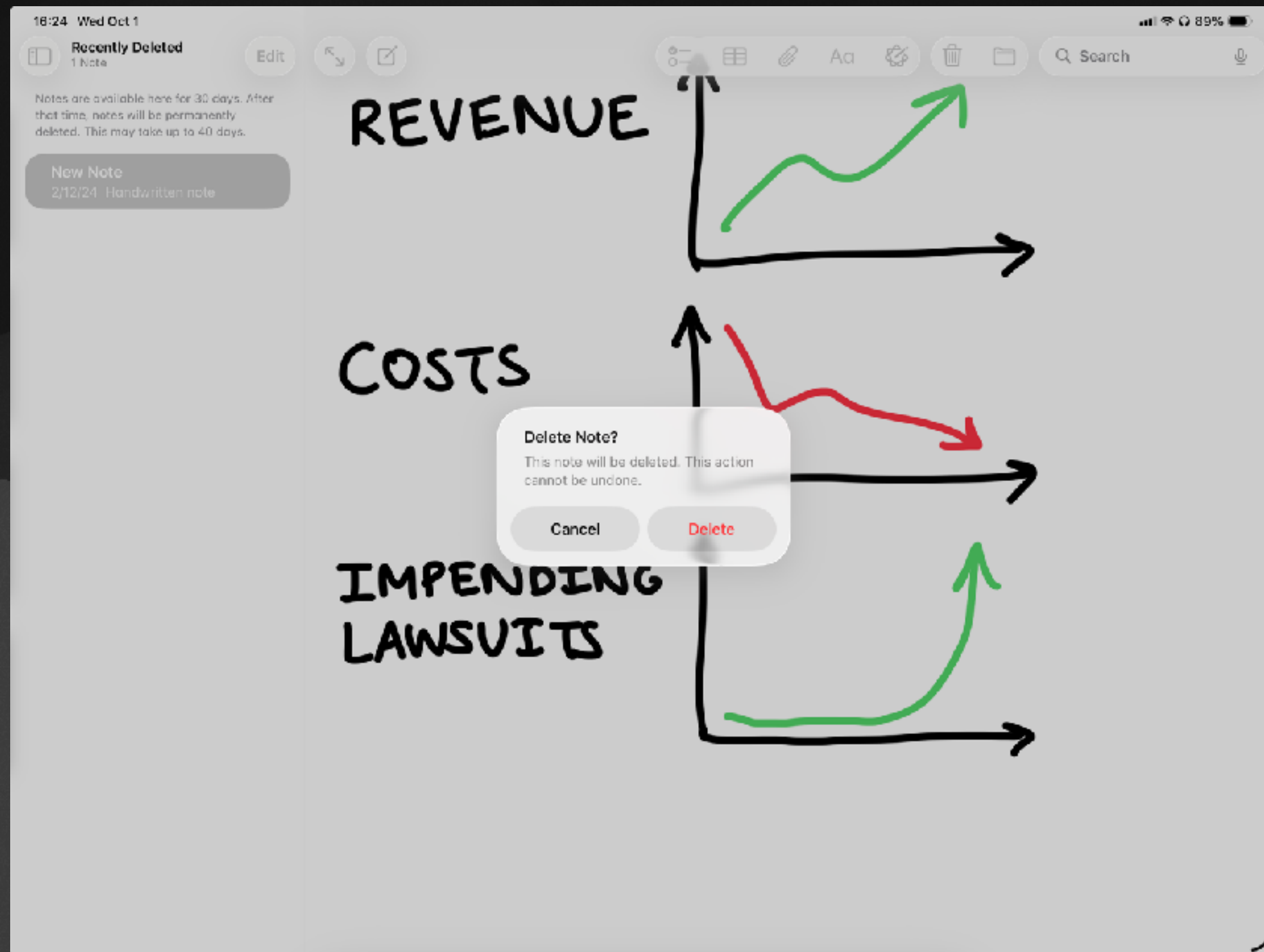
Master-detail



Hierarchical

Modals

Example: Notes App



Lightweight and focused interactions

Implementation

NavigationStack

Directly linking to views

```
NavigationStack {  
  List {  
    NavigationLink("Tap for analytics...") {  
      Text("[pretend we have useful content here]")  
        .navigationTitle("Analytics")  
        .navigationBarTitleDisplayMode(.inline)  
    }  
  }  
  .navigationTitle("Bootleg Penn Mobile")  
}
```



NavigationStack

Presenting based on data

```
NavigationStack {  
  List {  
    NavigationLink("Tap for analytics...", value: "Analytics")  
  }  
  .navigationTitle("Bootleg Penn Mobile")  
  .navigationDestination(for: String.self) { value in  
    Text("[pretend we have useful content here]")  
    .navigationTitle(value)  
    .navigationBarTitleDisplayMode(.inline)  
  }  
}
```



Allows you to modify path programmatically

value is passed into **.navigationDestination**

Can help make code cleaner

NavigationStack

Programmatically changing the path

```
@State var path = NavigationPath()
...
NavigationStack(path: $path) {
    List {
        Button(action: {
            path.append("Analytics")
        }) {
            Text("Tap for analytics...")
        }
    }
    .navigationTitle("Bootleg Penn Mobile")
    .navigationDestination(for: String.self) { value in
        Text("[pretend we have useful content here]")
        .navigationTitle(value)
        .navigationBarTitleDisplayMode(.inline)
    }
}
```

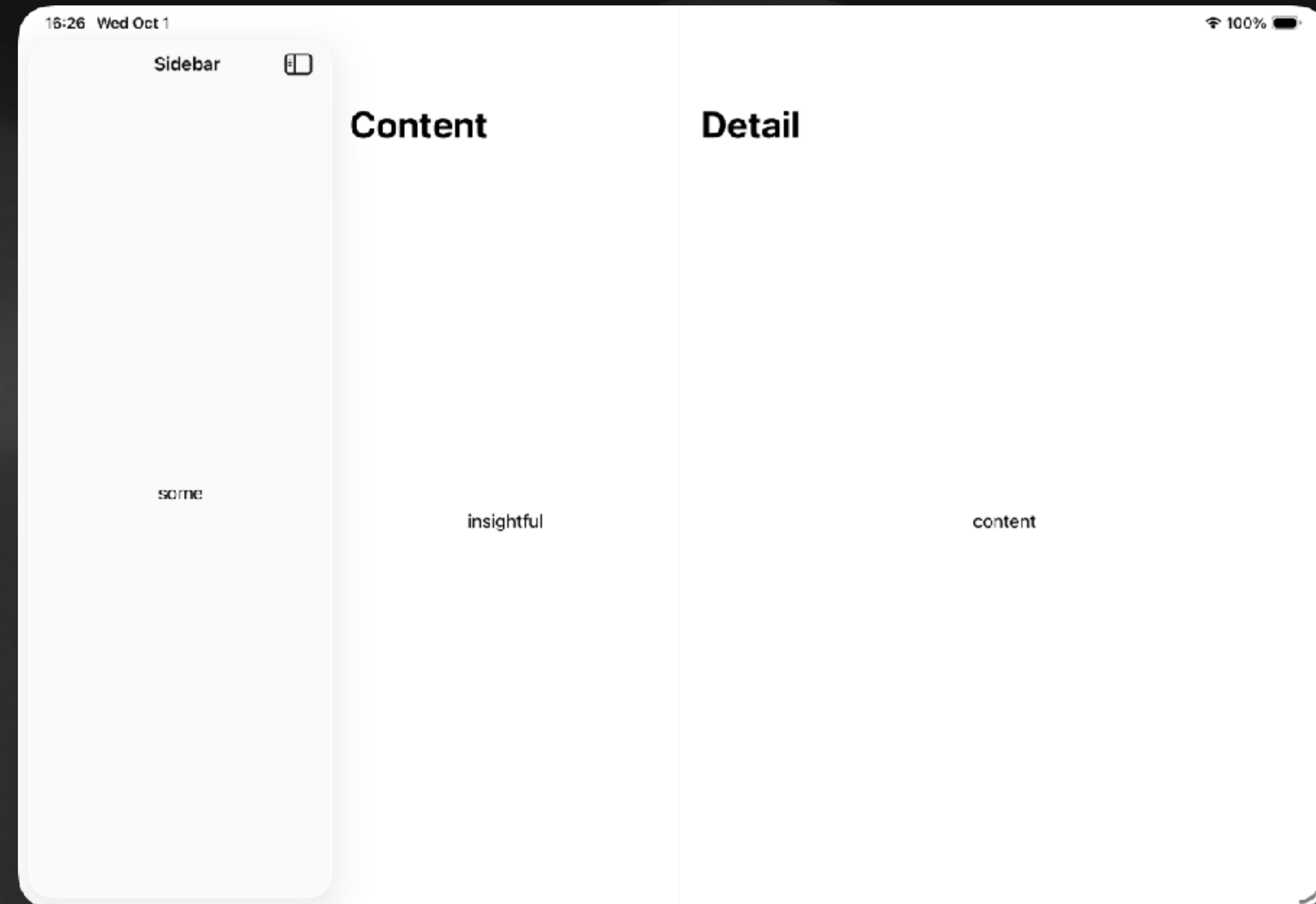


Allows you to modify path programmatically

NavigationSplitView

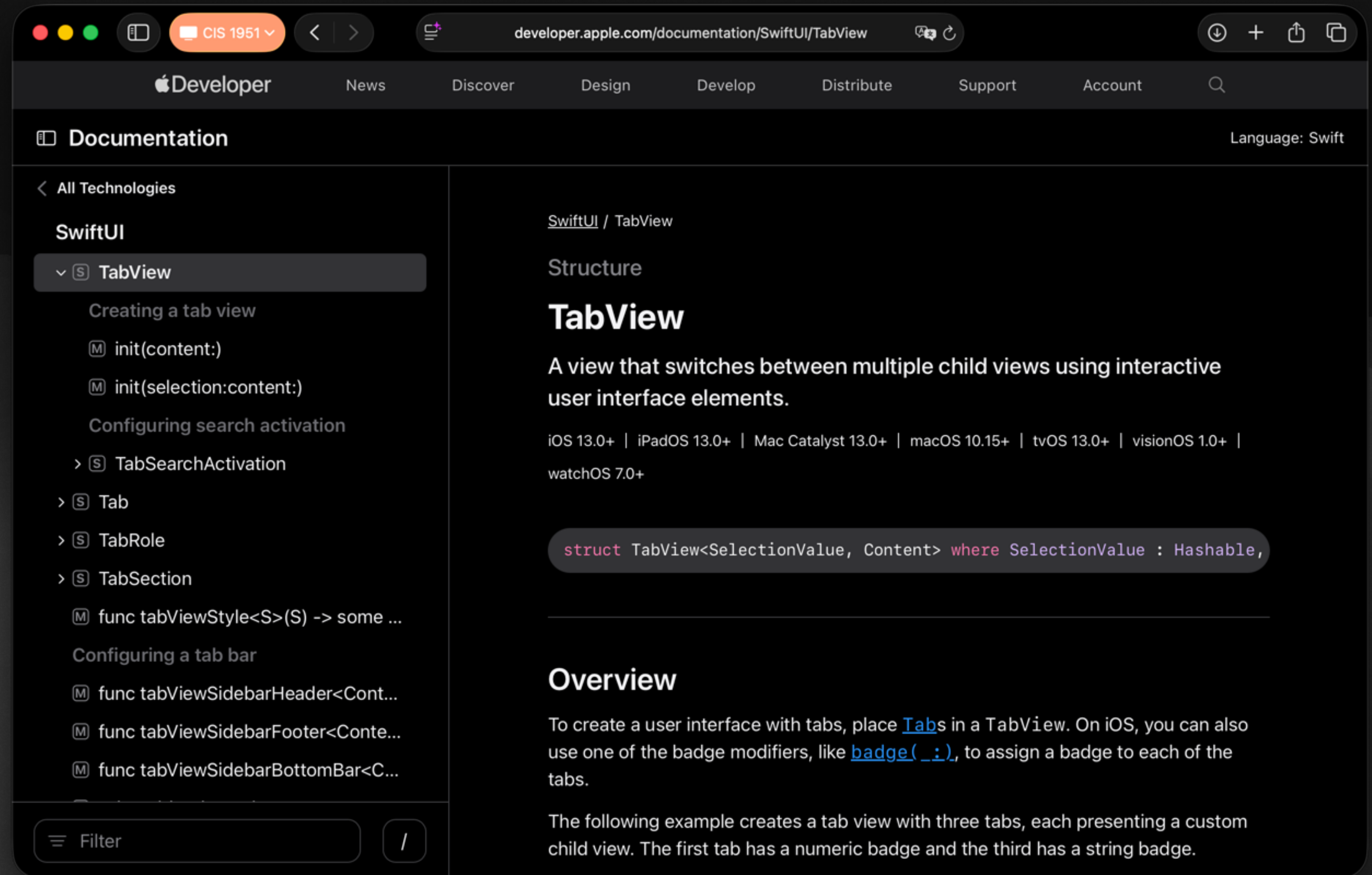
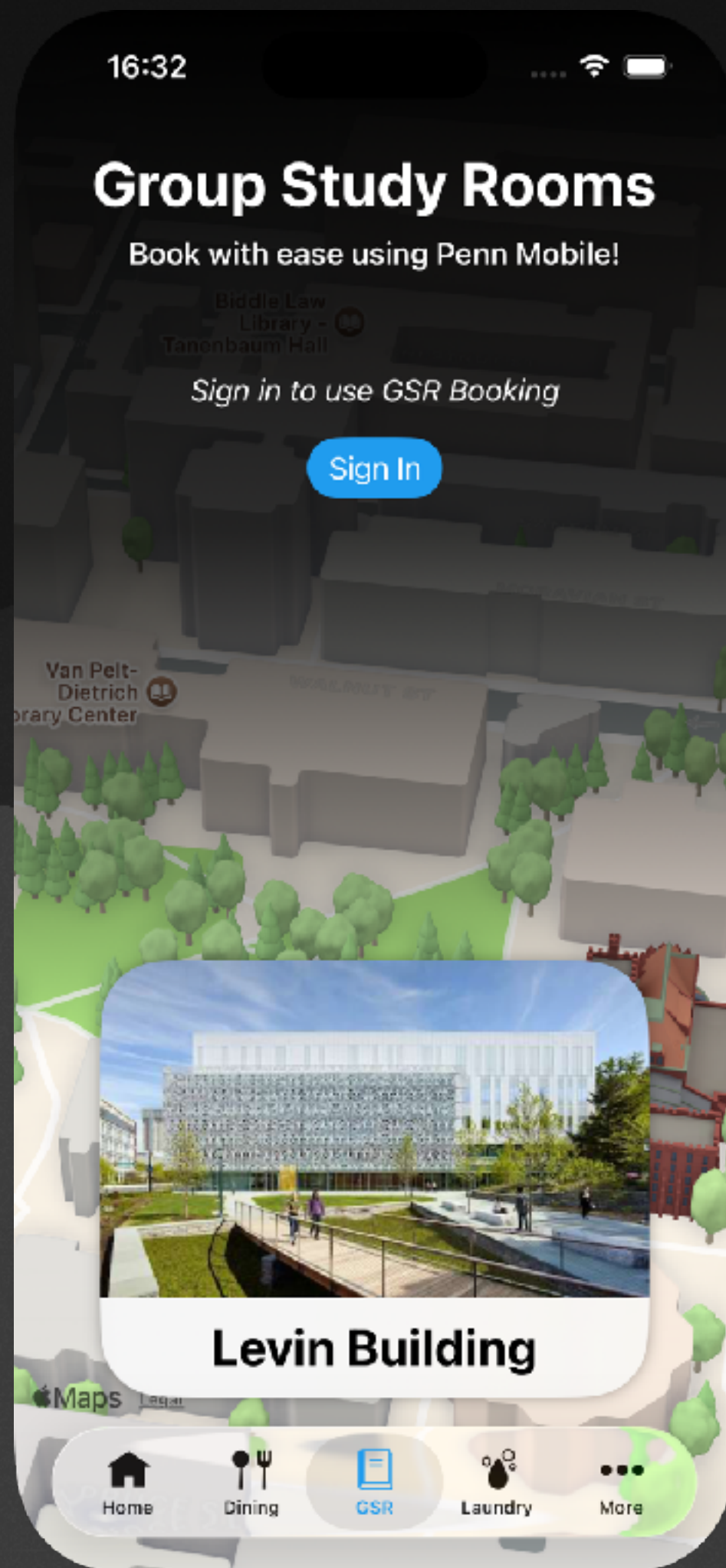
Multi-column layouts

```
NavigationSplitView {  
  Text("Sidebar")  
} content: {  
  Text("Content")  
} detail: {  
  Text("Detail")  
}
```



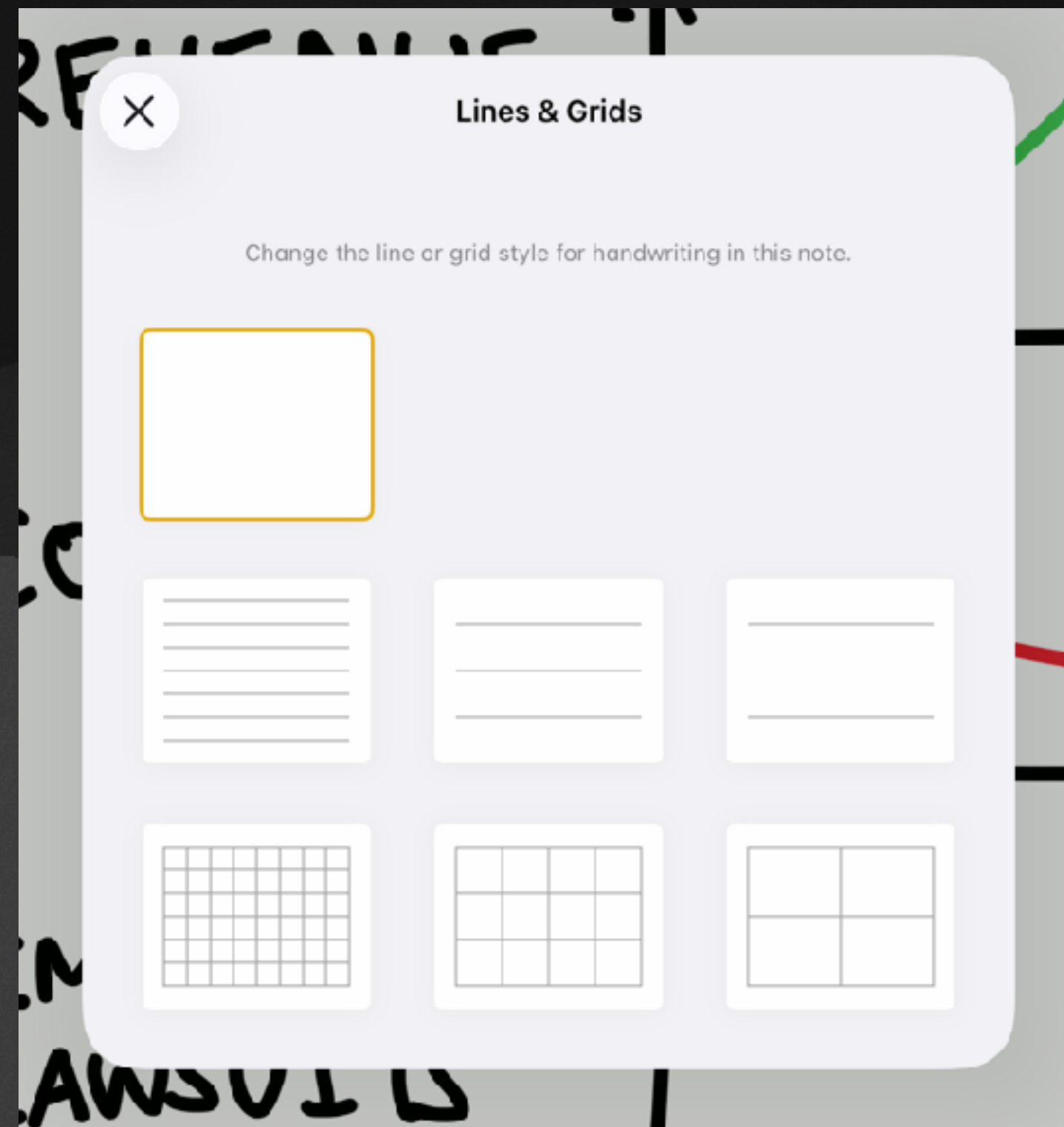
Appears as a NavigationStack on iPhone

TabView

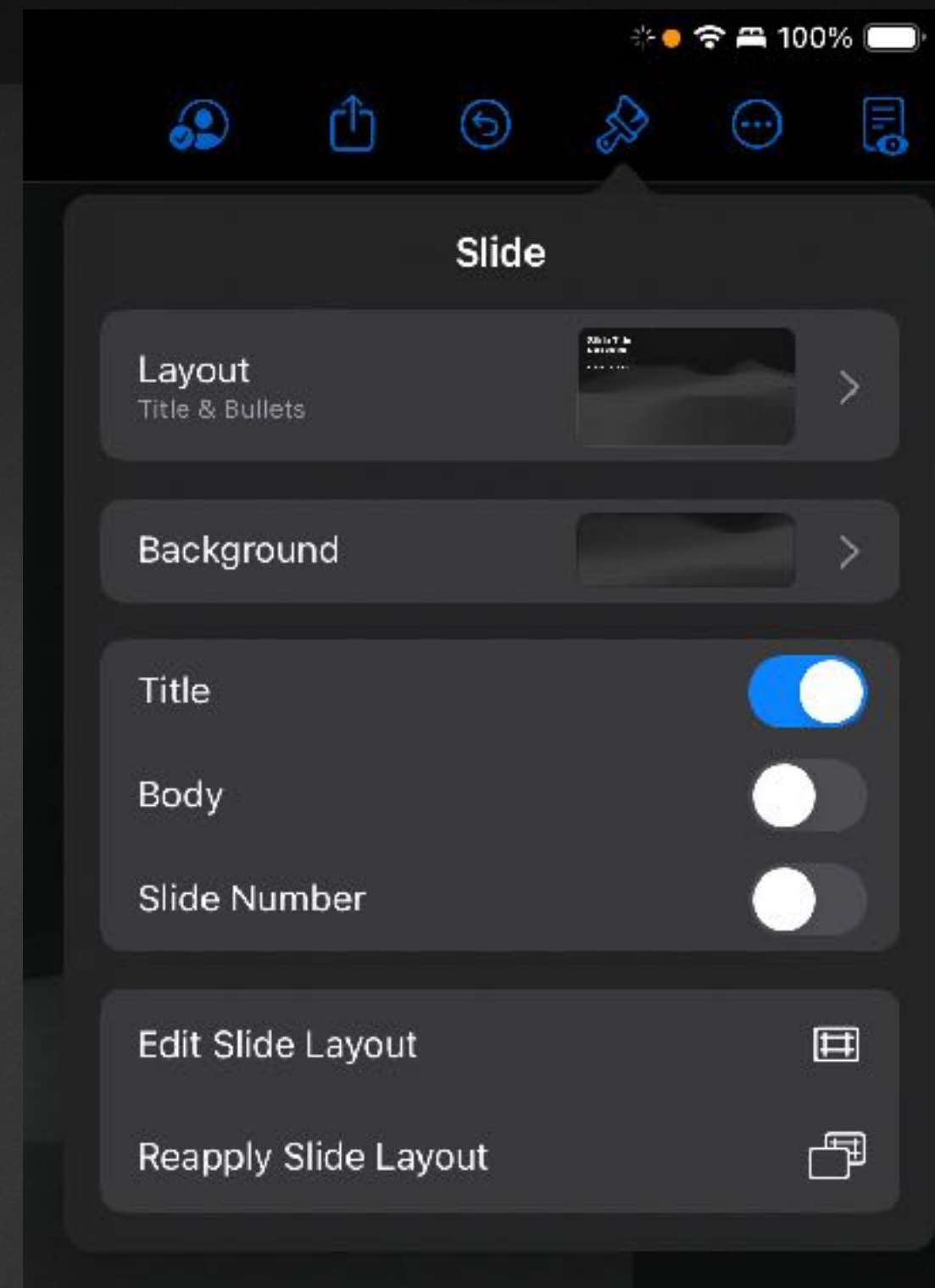


Modal presentations

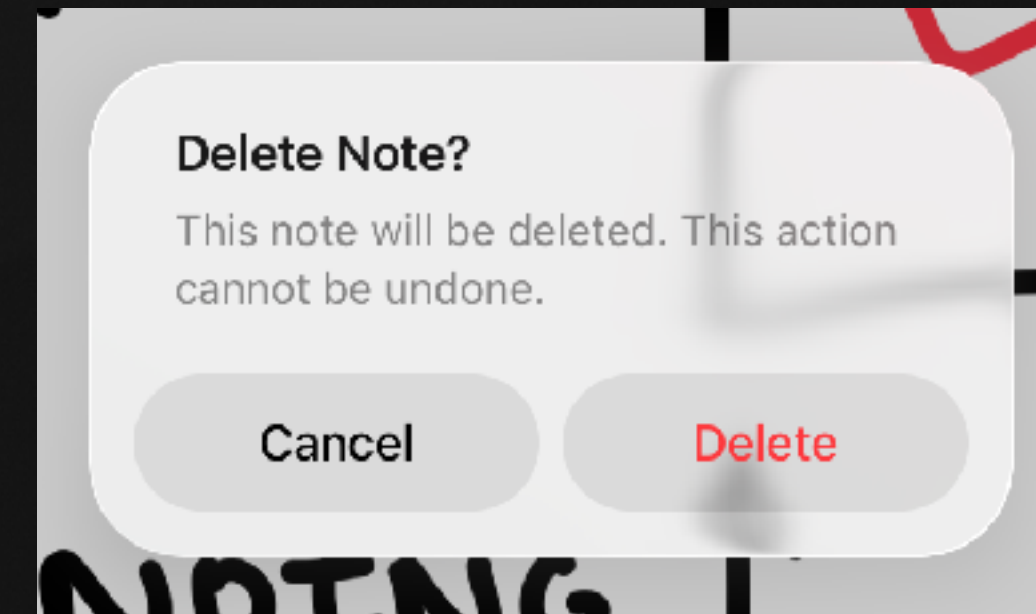
For lightweight, focused interactions



.sheet



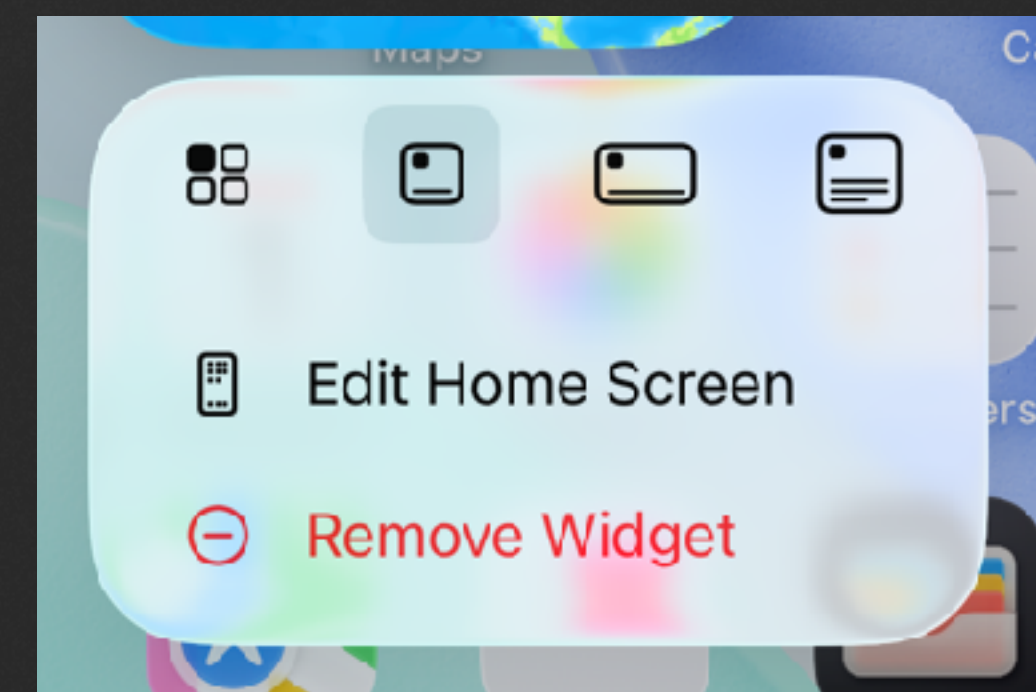
.popover



.alert



.confirmationDialog



Menu
— OR —
.contextMenu

.inspector

Files16:39Wed Oct 1

Investigating_New_Techniques_for_Feline_Simulation

⌕

✎

📄

📁

⋮

Investigating New Techniques for Feline Simulation

Abstract

In this paper, we investigate new techniques for feline simulation. We find that AI and blockchain technology can significantly improve the realism of feline simulations. In addition, we find that these technologies can also improve the accuracy of feline simulations. We also find that these technologies can be used to create more realistic and lifelike simulations of feline behavior. We were able to achieve a feline simulation score of 97.5% using these technologies, paving the way for future research in this area.

Introduction

With the advent of new technologies, there has been a recent surge in interest in investigating new techniques for feline simulation. In particular, there has been a lot of interest in using AI and blockchain technology to create more realistic and lifelike simulations of cats.

Currently, the primary approach to feline simulation is through the use of computer graphics. This involves creating a 3D model of a cat and then applying realistic movement to the model. While this can create a realistic-looking simulation, it falls short in two key areas. First, it is difficult to create a truly lifelike simulation with computer graphics. Second, even if a realistic simulation can be created, it is difficult to create one that is interactive and believable.

AI and blockchain-based solutions have the potential to address both of these limitations. First, AI can be used to create more realistic and lifelike simulations by learning from real-world data. Second, blockchain can be used to create a decentralized platform on which these simulations can run, making them more accessible and interactive. These new techniques have the potential to revolutionize the way we simulate cats, and could have a significant impact on the field of feline research.

Methods

The methods used in this study were AI, blockchain, and cat memes. These three techniques were used to generate lifelike simulations of cats. The AI was used to create the initial simulations, the blockchain was used to secure the data, and the cat memes were used to add realism to the simulations.

Python and TensorFlow were used to set up the AI for this project. The model was trained on a computer with a quad-core processor and an NVIDIA GTX 1080 Ti GPU. The architecture of the model is a deep convolutional neural network. The model consists of

- an input layer

📄

📁

🔍

Information

File Name	Investigating_New_Techniques_for_Feline_Simulation.pdf
Kind	PDF document
Size	91 KB
Created	May 31, 2022 at 23:03
Modified	May 31, 2022 at 23:03
Last Opened	October 1, 2025 at 16:39
Version	1.5
Pages	4
Page Size	21.59 × 27.94 cm
Security	None
Content Creator	LaTeX via pandoc
Encoding Software	pdfTeX-1.40.22

Ideal for showing
auxiliary content

CIS 1951

www.youtube.com/watch?v=6Rp71CvKIKs

YouTube

Search

Sign in

```
// NavigationLink variants

NavigationLink("Show detail") {
    DetailView()
}

NavigationLink("Apple Pie", value: applePieRecipe)
```

5:15 / 26:06

App structure

WWDC22: The SwiftUI cookbook for navigation | Apple

Apple Developer

239K subscribers

Subscribe

204

Share

Save

9,541 views Sep 24, 2024

The recipe for a great app begins with a clear and robust navigation structure. Join the SwiftUI team in our proverbial coding kitchen and learn how you can cook up a great experience for your app. We'll introduce you to SwiftUI's navigation stack and split view features, show you how you can link to specific areas of your app, and explore how you can quickly and easily restore navigational state.

Explore related documentation, sample code, and more:
List: <https://developer.apple.com/documenta...>
NavigationSplitView: <https://developer.apple.com/documenta...>
NavigationStack: <https://developer.apple.com/documenta...>
Bringing robust navigation structure to your SwiftUI app: <https://developer.apple.com/documenta...>
Migrating to new navigation types: <https://developer.apple.com/documenta...>
What's new in SwiftUI: <https://developer.apple.com/videos/pl...>
SwiftUI on iPad: Organize your interface: <https://developer.apple.com/videos/pl...>

CIS 1951

www.youtube.com/watch?v=3MugGCtm26A

YouTube

wwdc25 swiftui navigation

Sign in

App structure

NavigationSplitView
TabView
NavigationStack
Sheets
Inspectors
& more!

Every one of these members is refined for the new design.

3:24 / 22:16

App structure

WWDC25: Build a SwiftUI app with the new design | Apple

Apple Developer

239K subscribers

Subscribe

898

Share

Save

43,272 views Jun 9, 2025

Explore the ways Liquid Glass transforms the look and feel of your app. Discover how this stunning new material enhances toolbars, controls, and app structures across platforms, providing delightful interactions and seamlessly integrating your app with the system. Learn how to adopt new APIs that can help you make the most of Liquid Glass.

Explore related documentation, sample code, and more:
Applying Liquid Glass to custom views: <https://developer.apple.com/documenta...>
Adopting Liquid Glass: <https://developer.apple.com/documenta...>
Landmarks: Building an app with Liquid Glass: <https://developer.apple.com/documenta...>
Get to know the new design system: <https://developer.apple.com/videos/pl...>

00:00 - Introduction
02:07 - App structure

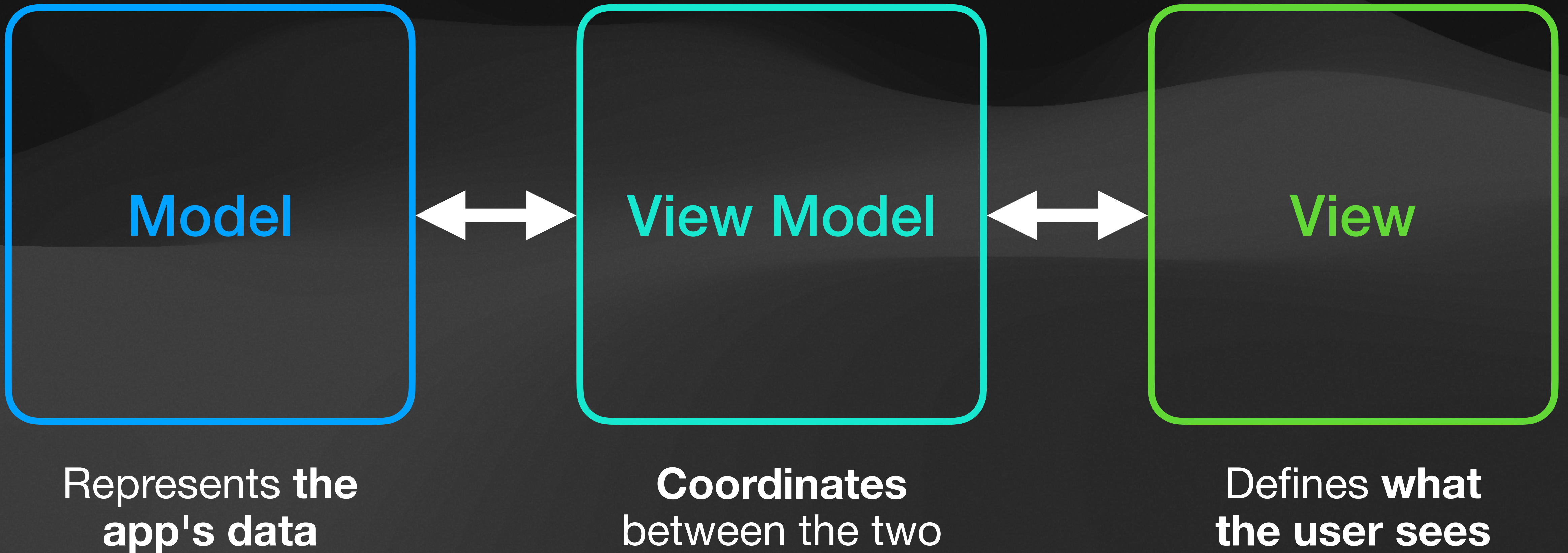
MVVM

Separation of Concerns

- Split code into **modular components**
- Each component only **handles one thing** (a "concern")
- Why? More testable, reusable, maintainable code

MVVM

Model-View-View Model



MVVM

The View Model

View Model

Lets the view **bind to data** and **send commands**

Notifies the view of any changes

Converts data to and from what the view wants

Isolates the view from its underlying data

Usually **a class**

Coordinates
between the two

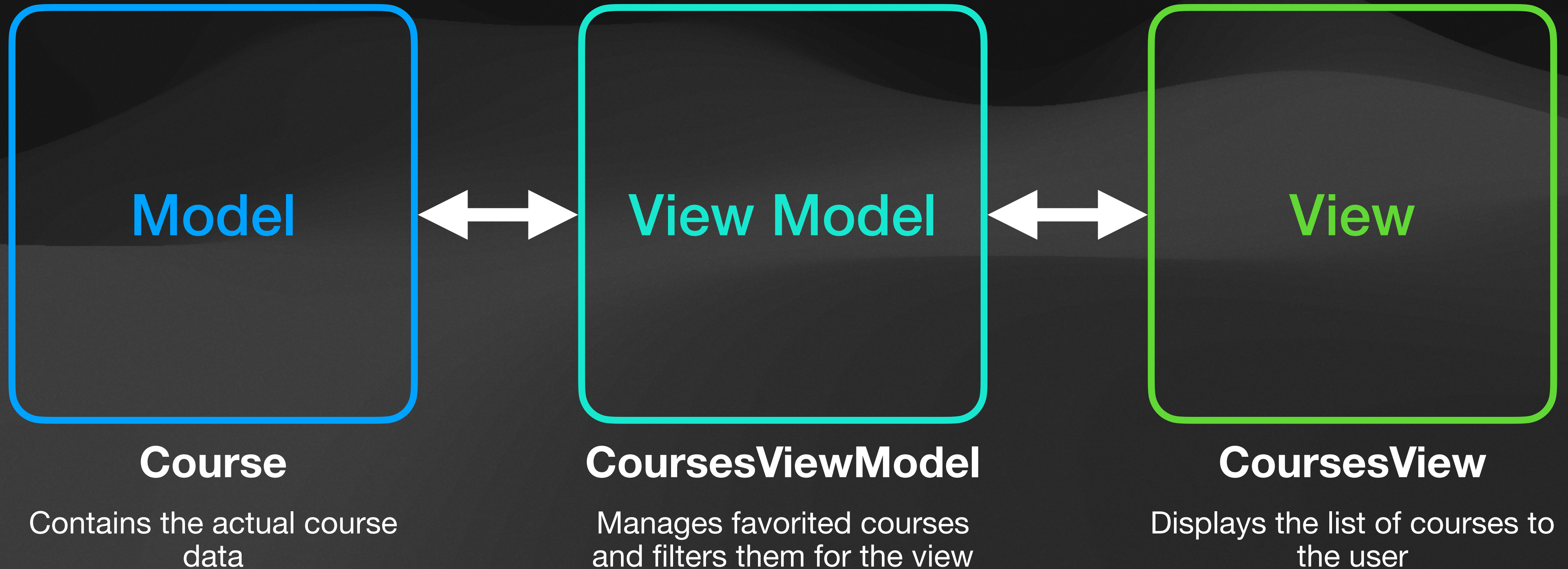
MVVM

How we used it last week



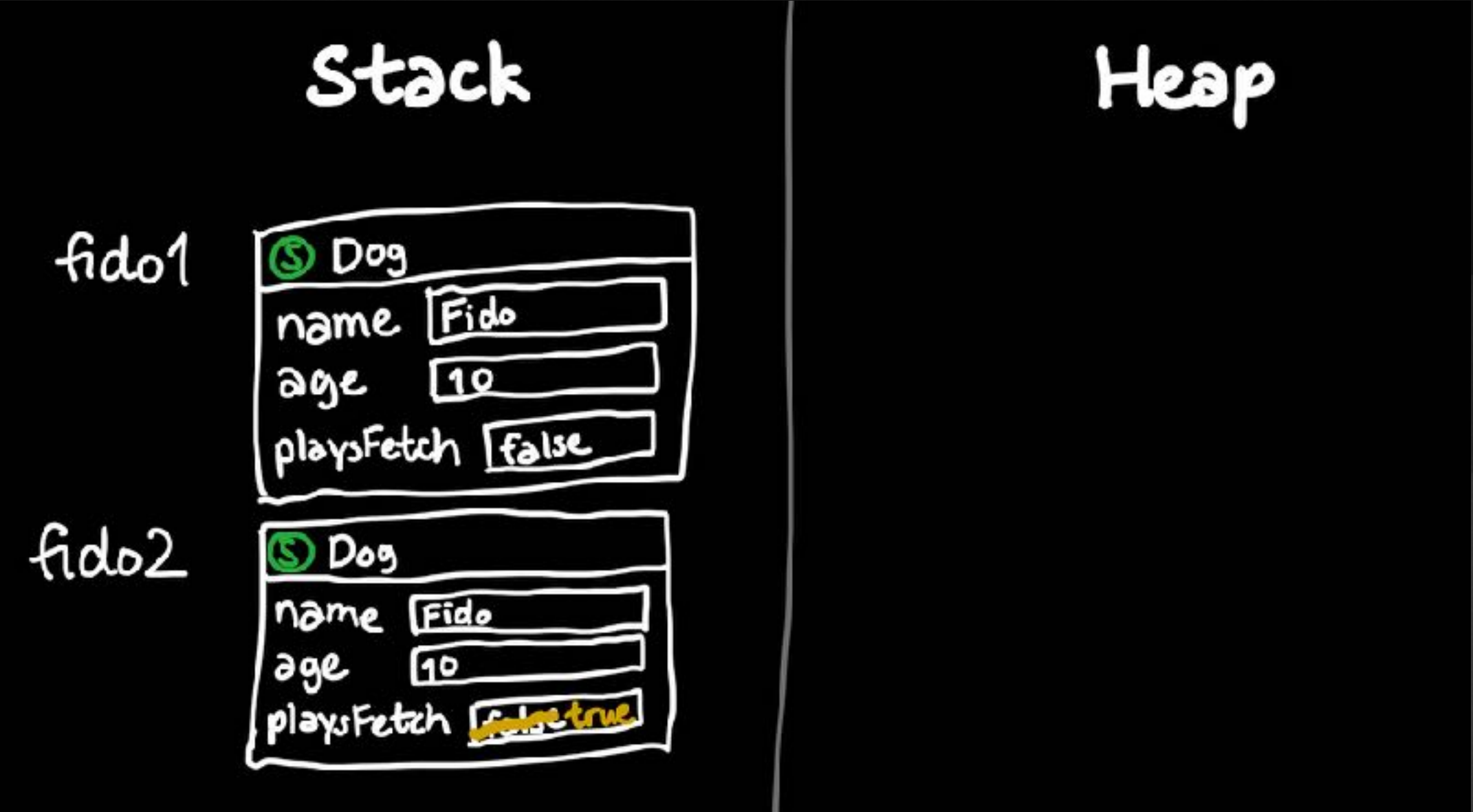
MVVM

How we'll use it

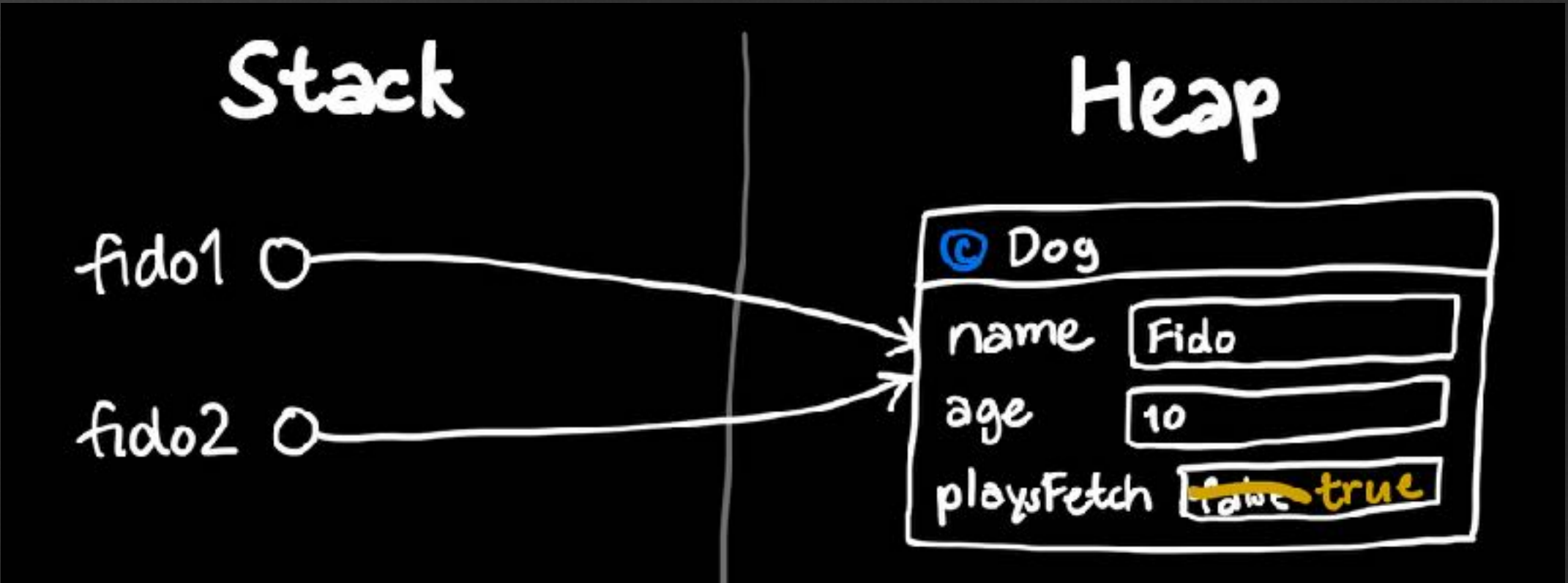


Refresher

Structs



Classes



@Observable

When we want to use a class's property as state, we need to tag the class as **@Observable**

```
@Observable class MyViewModel
```

@Bindable

Bindings for Observables

- Similar to @Binding, we can pass a class marked with @Observable to a child view.
- Remember the relationship between @State (owns) and @Binding (allowed to change). The same relationship exists here.

ew

```
@Observable
class MySettings: Identifiable {
    var color: Color = .red
    let internships = 0
}

struct PropDrilling: View {
    @State var settings = MySettings()

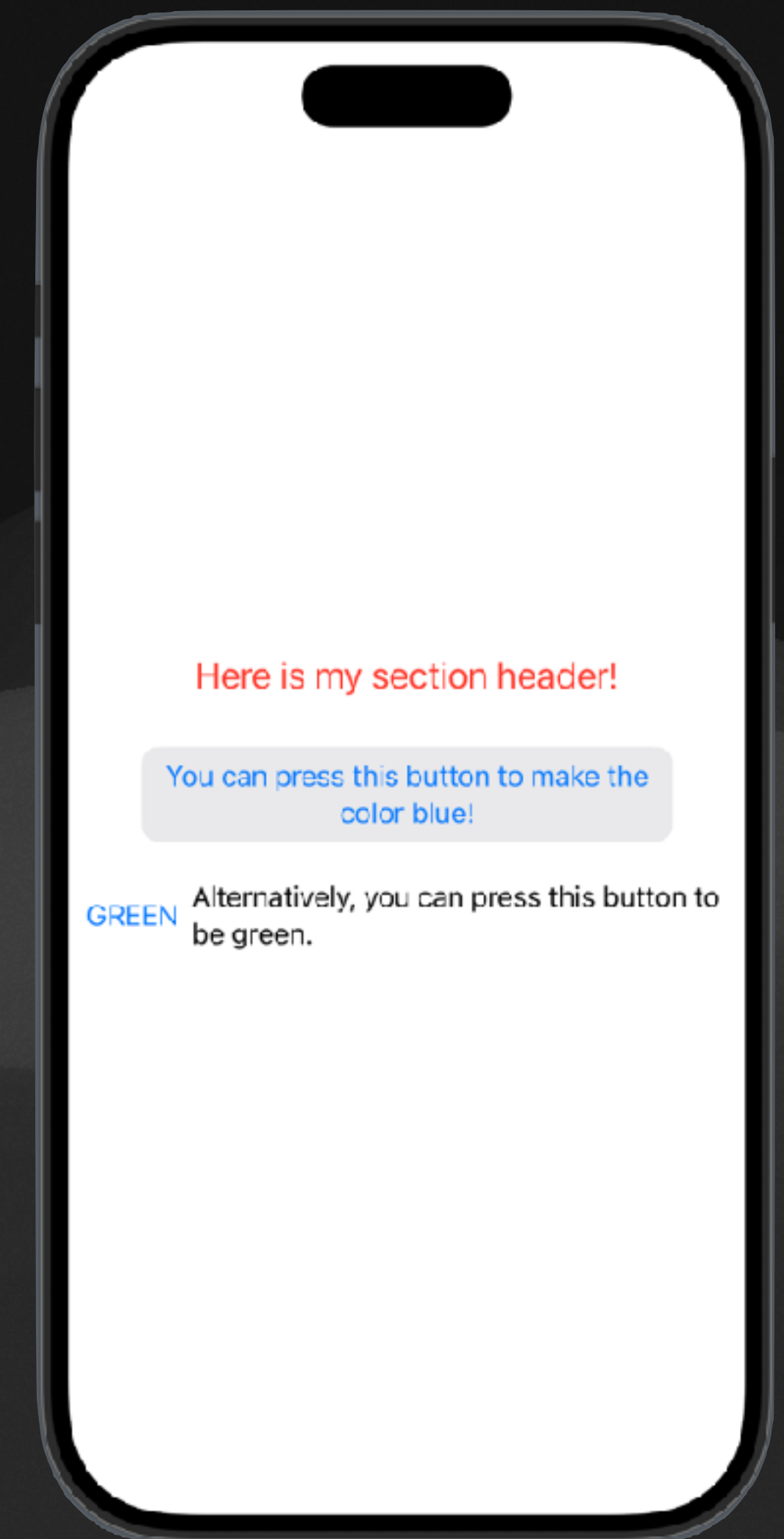
    var body: some View {
        SectionView(settings: settings)
    }
}
```

```
struct SectionView: View {
    @Bindable var settings: MySettings

    var body: some View {
        VStack {
            Text("Here is my section header!")
                .font(.title2)
                .foregroundColor(settings.color)
            Button {
                settings.color = .blue
            } label: {
                Text("You can press this button to make the color blue!")
            }
                .buttonStyle(.bordered)
                .padding()
            SwitchView(settings: settings)
        }
    }
}

struct SwitchView: View {
    @Bindable var settings: MySettings

    var body: some View {
        HStack {
            Button {
                settings.color = .green
            } label: {
                Text("GREEN")
            }
            Text("Alternatively, you can press this button to be green.")
        }
    }
}
```



"prop drilling"

@Environment

Pass an object/property to **all** subviews

```
@Observable
class MySettings: Identifiable {
    var color: Color = .red
    let internships = 0
}

struct PropDrilling: View {
    @State var settings = MySettings()

    var body: some View {
        SectionView()
            .environment(settings)
    }
}
```

```
struct SectionView: View {
    // Will crash if not present
    @Environment(MySettings.self) var settings

    var body: some View {
        VStack {
            Text("Here is my section header!")
                .font(.title2)
                .foregroundColor(settings.color)
            Button {
                settings.color = .blue
            } label: {
                Text("You can press this button to make the color blue!")
            }
                .buttonStyle(.bordered)
                .padding()
            SwitchView()
        }
    }
}
```

```
struct SwitchView: View {
    @Environment(MySettings.self) var settings

    var body: some View {
        HStack {
            Button {
                settings.color = .green
            } label: {
                Text("GREEN")
            }
            Text("Alternatively, you can press this button to be green.")
        }
    }
}
```

Note: using @Environment is a design decision. If you find yourself passing the same property/object to 2+ views (to be modified), consider using environment.

Brief aside

@ObservableObject

iOS 13-16

The screenshot shows the Apple Developer documentation page titled "Use the Observable macro". The page is in the "Documentation" section, with the language set to "Swift" and API changes set to "None". The left sidebar shows the navigation menu with "SwiftUI" expanded, and "Essentials" selected. The main content area is titled "Use the Observable macro" and explains how to adopt [Observation](#) in an existing app by replacing [ObservableObject](#) with the [Observable\(\)](#) macro. It provides two code examples: "BEFORE" and "AFTER".

Use the Observable macro

To adopt [Observation](#) in an existing app, begin by replacing [ObservableObject](#) in your data model type with the [Observable\(\)](#) macro. The [Observable\(\)](#) macro generates source code at compile time that adds observation support to the type.

```
// BEFORE
import SwiftUI

class Library: ObservableObject {
    // ...
}

// AFTER
import SwiftUI

@Observable class Library {
    // ...
}
```

Then remove the [Published](#) property wrapper from observable properties. [Observation](#) doesn't require a property wrapper to make a property observable. Instead, the accessibility of the property in relationship to an observer, such as a view, determines whether a property is observable.

```
// BEFORE
@Observable class Library {
    @Published var books: [Book] =
}

// AFTER
@Observable class Library {
    var books: [Book] = [Book(), B...
```

If you have properties that are accessible to an observer that you don't want to track, apply the [ObservationIgnored\(\)](#) macro to the property.

Migrate incrementally

Recap

- **Navigation and modal presentation views** let us organize multiple screens
- **Model-view-view model** enables separation of concerns
- **@Observable** lets us manage state in classes

Homework 2

Trivia Game

- Will be released **Wednesday, 2/19**
- Due on **Wednesday, 3/19**
- Focuses on **lectures 3-5**
- [details pending]

