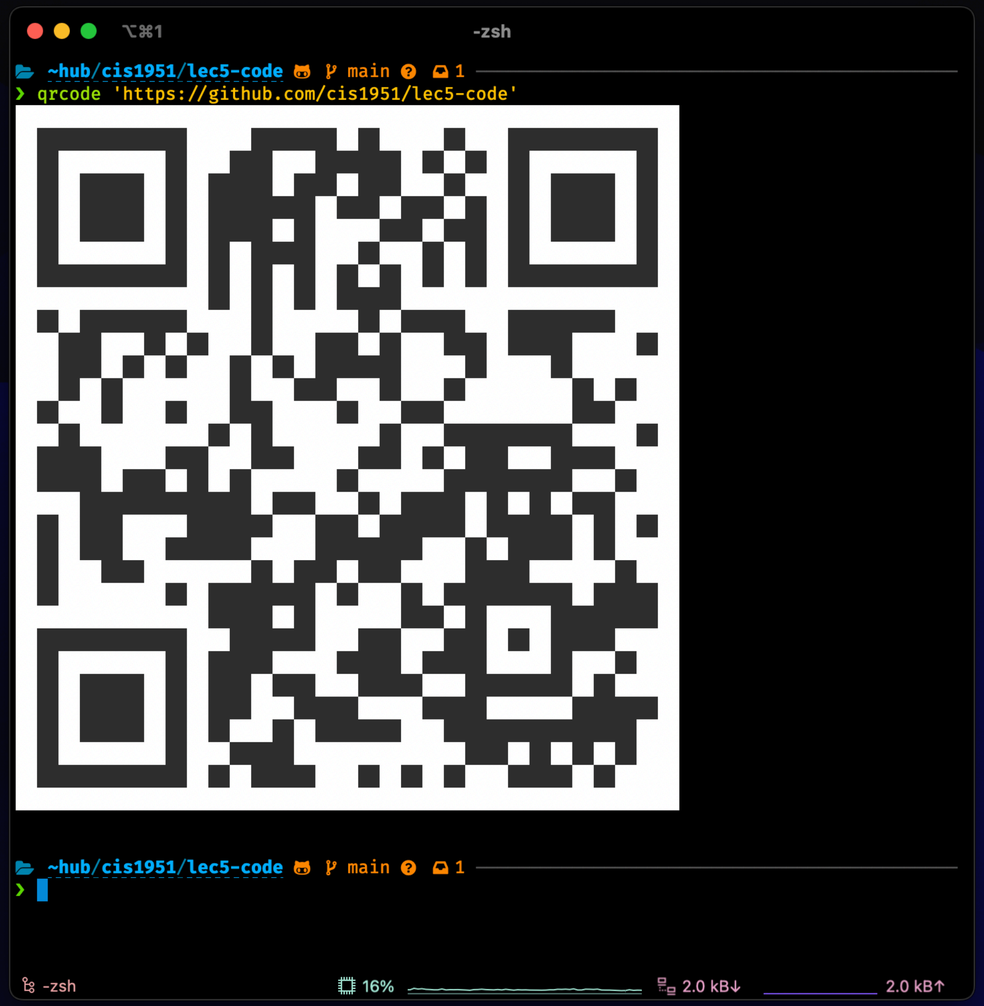
App Structure

Lecture 5

CIS 1951

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



Previously, on CIS 1951...

SwiftUI State Management

@State

@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

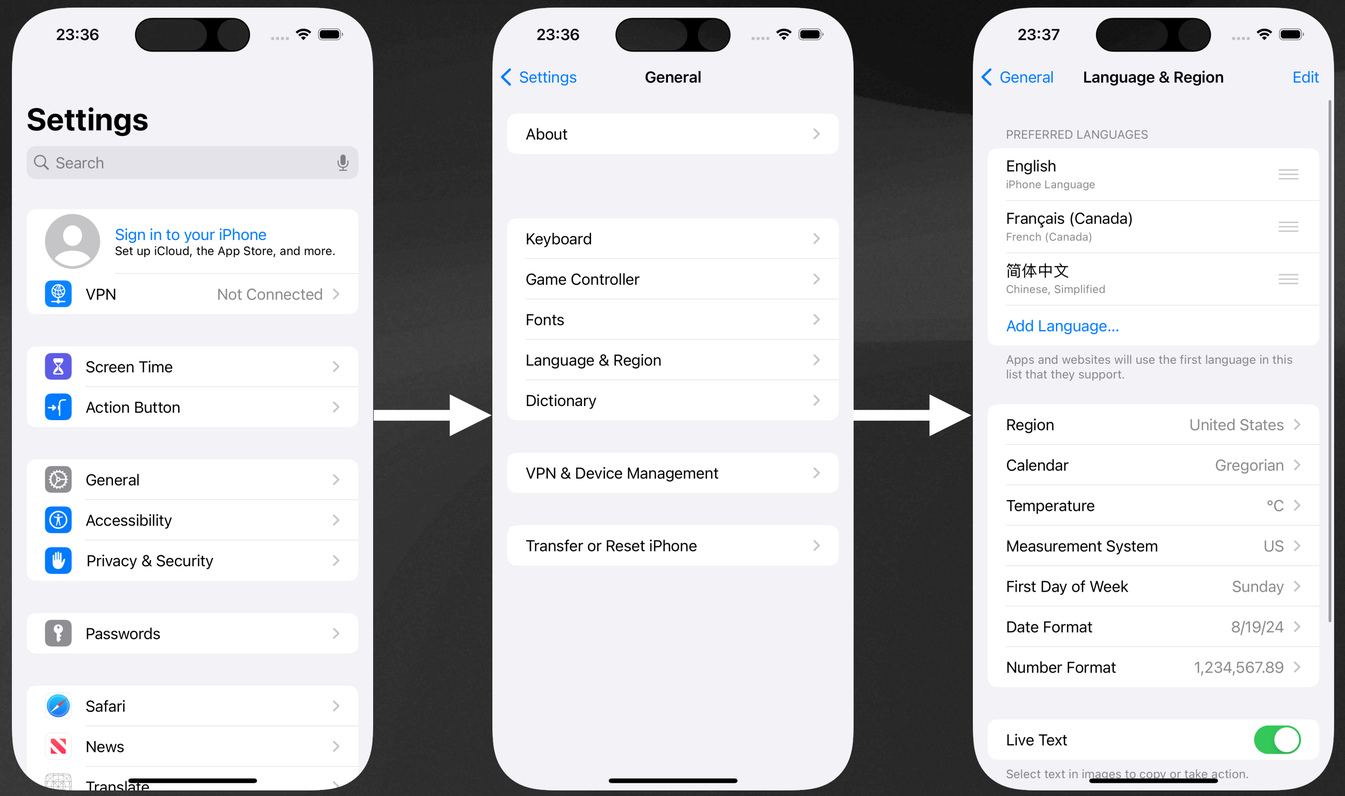
@Observable, @Bindable, @Environment

Navigation & Modal Presentations

How do we organize multiple screens?

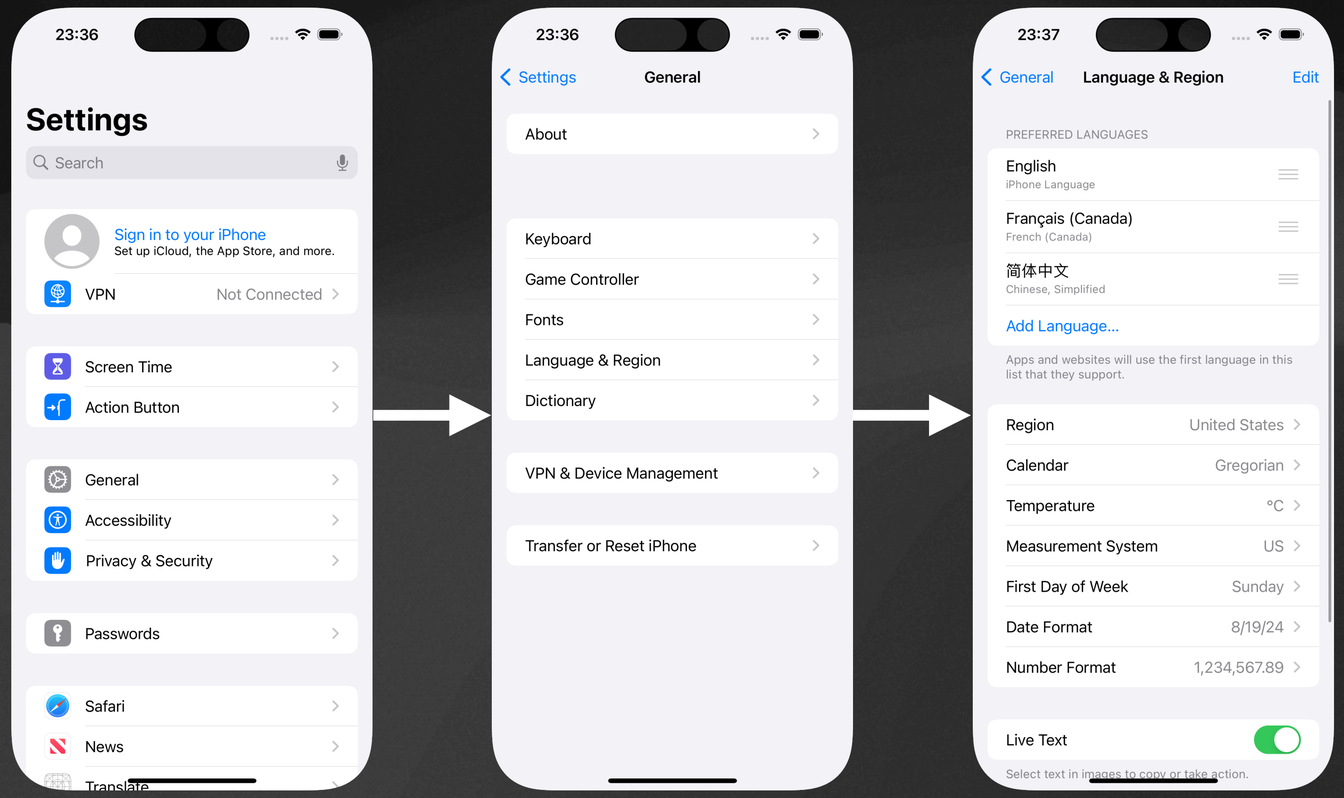
Hierarchical Navigation

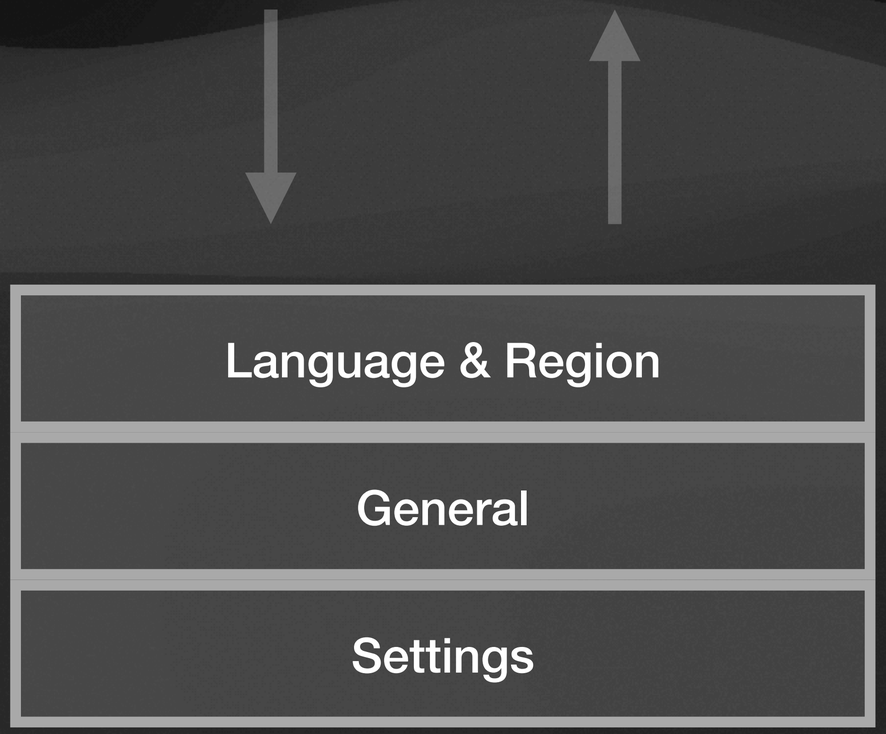
Example: Settings App



Hierarchical Navigation

Example: Settings App



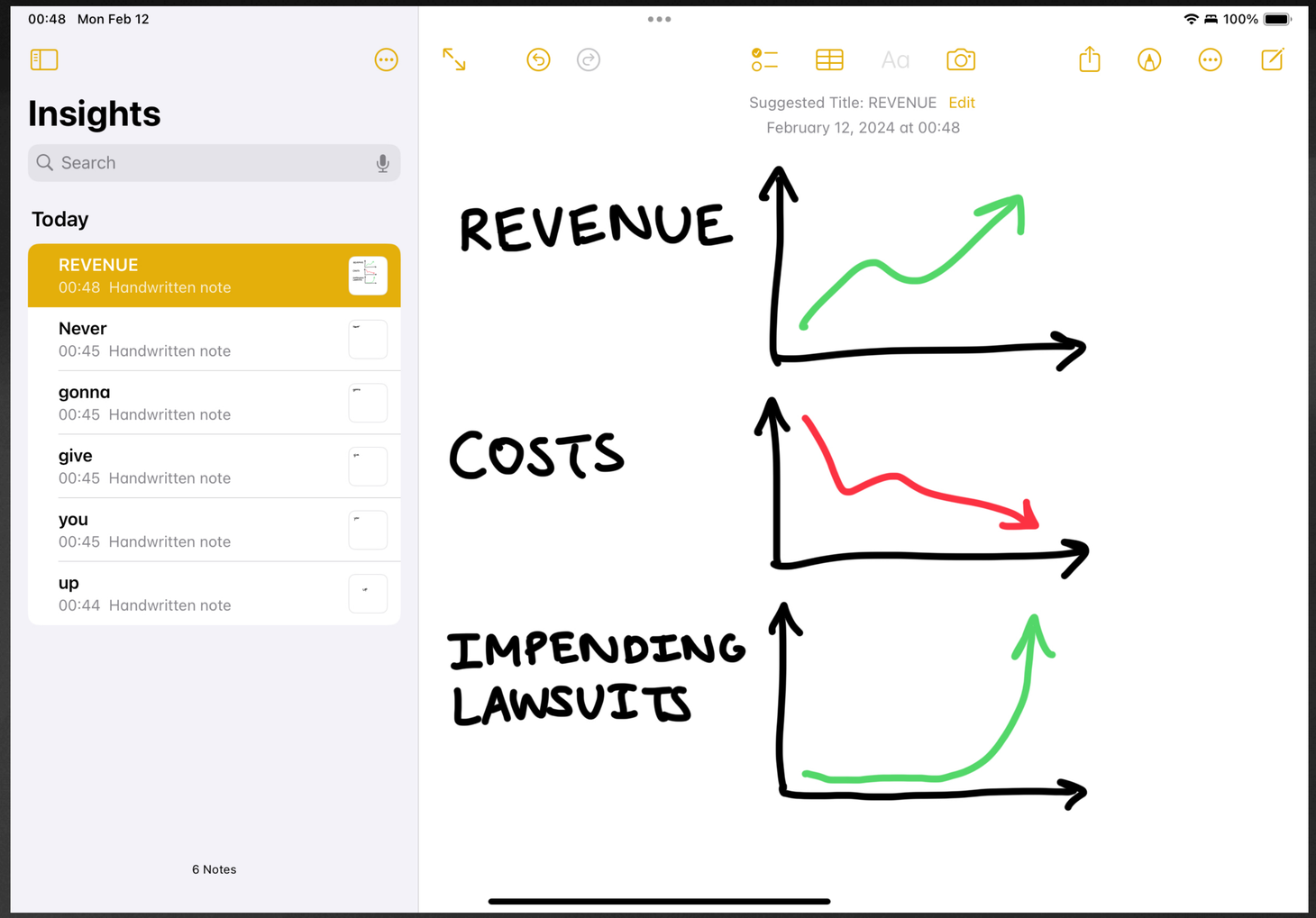


Master-Detail Navigation

Example: Notes App

Master

List of notes

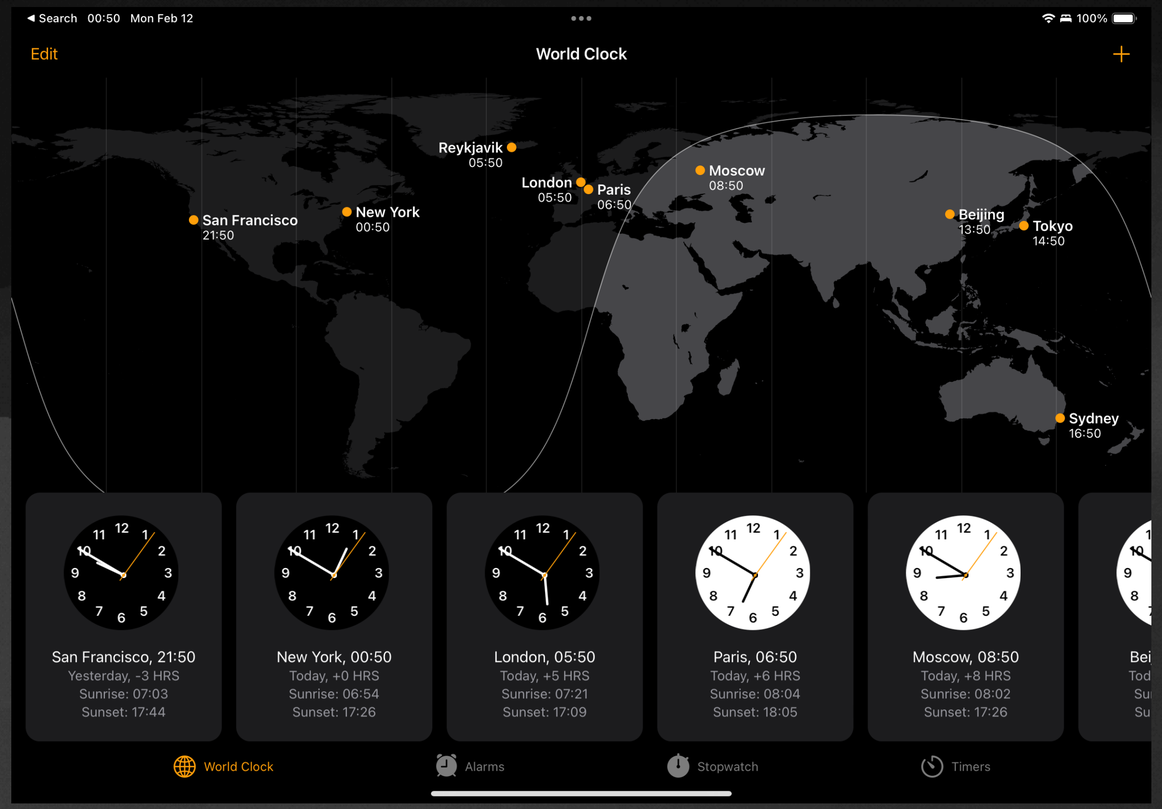


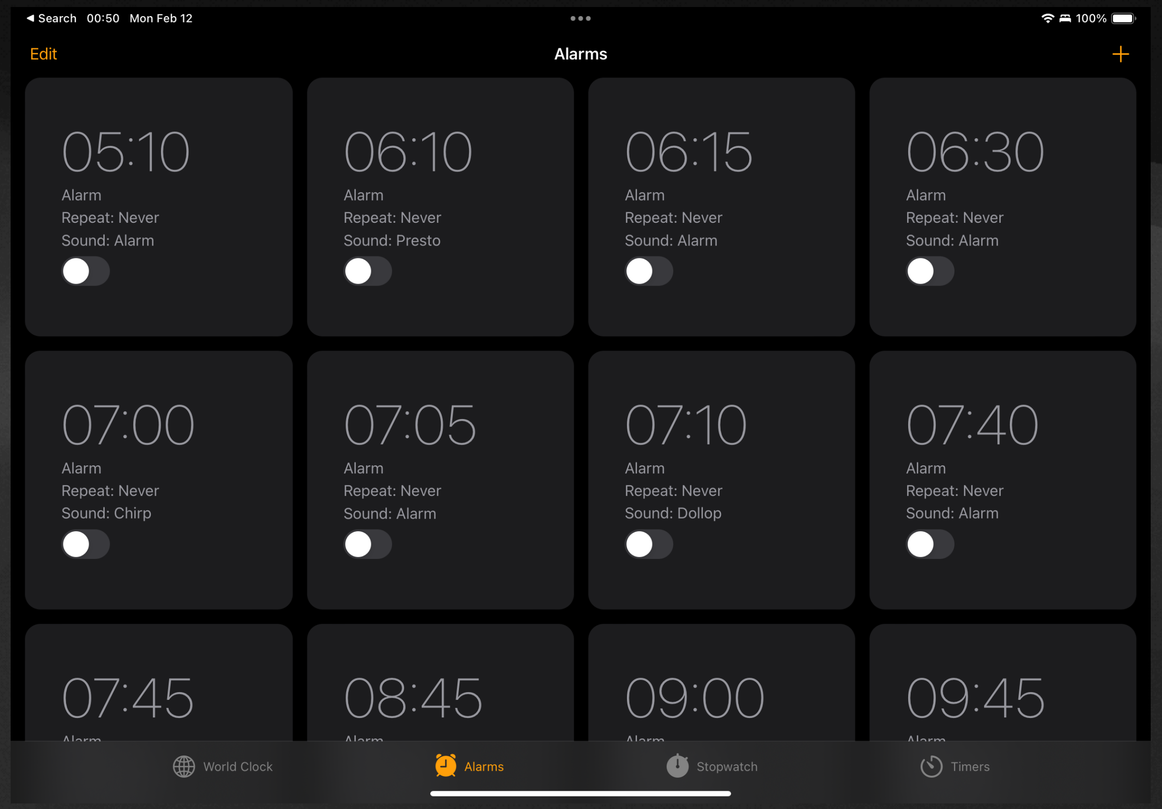
Detail

Single note

Tab Bar Navigation

Example: Clock App

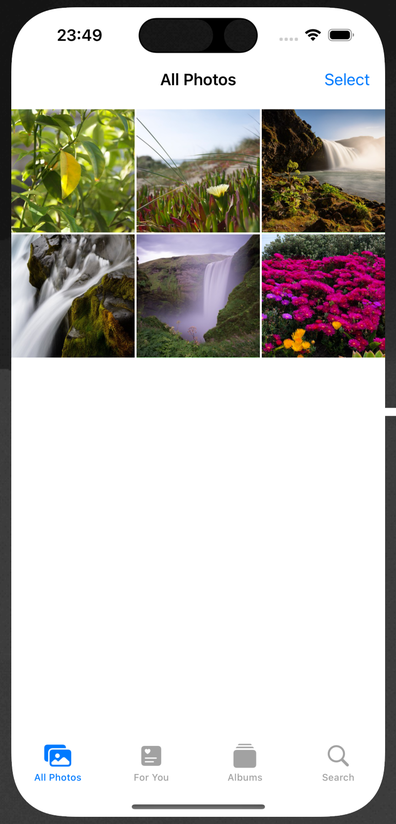




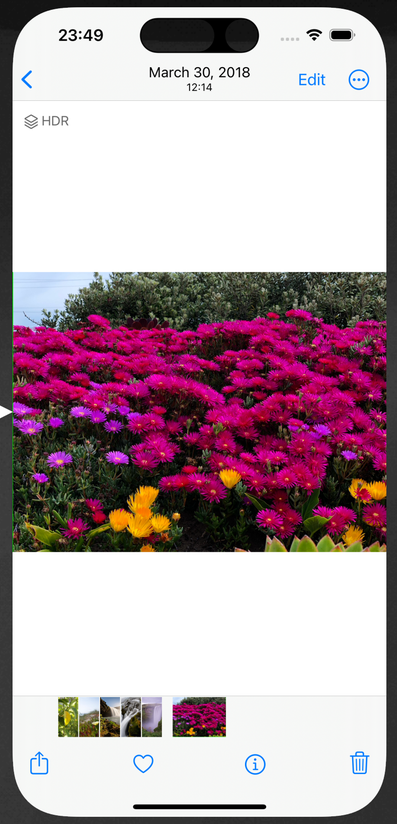
Bottom bar for **quick navigation**

Hybrid Navigation

Example: Photos App

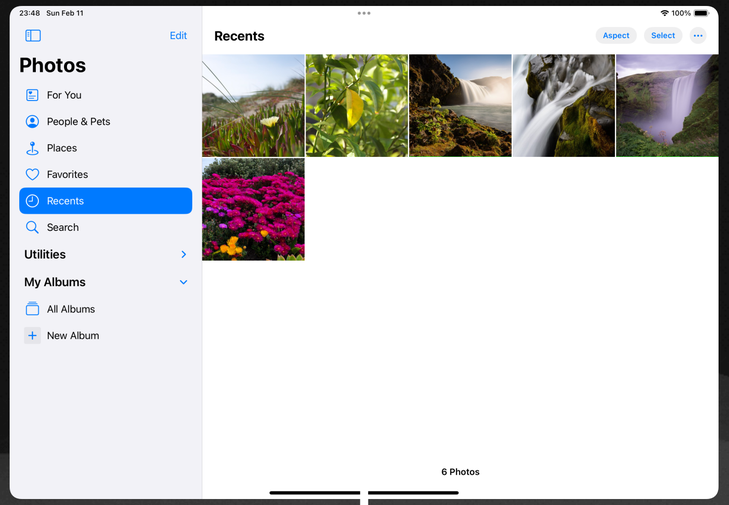


Tab bar

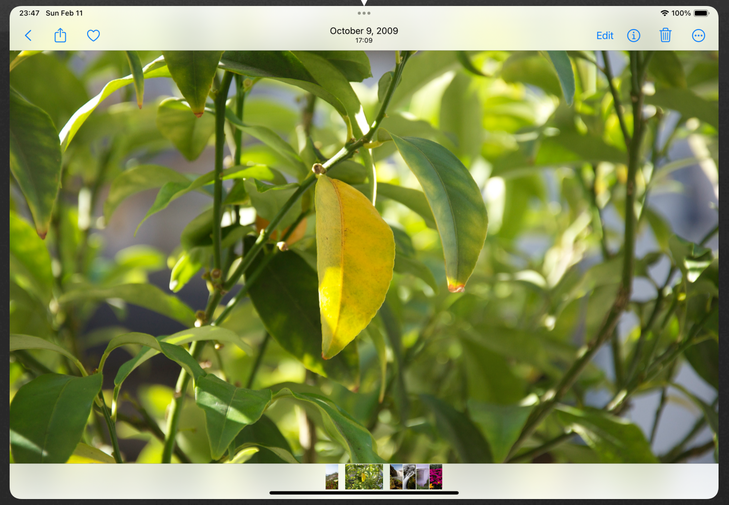


Hierarchical

Master-detail

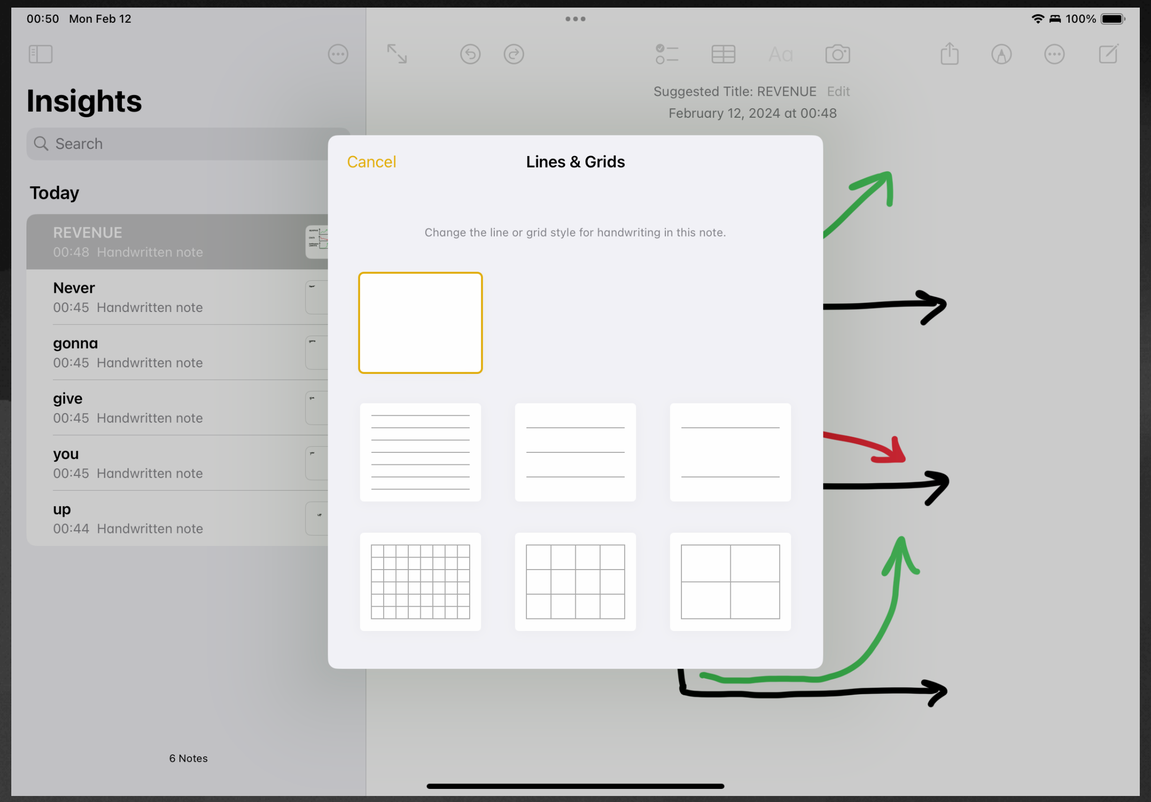


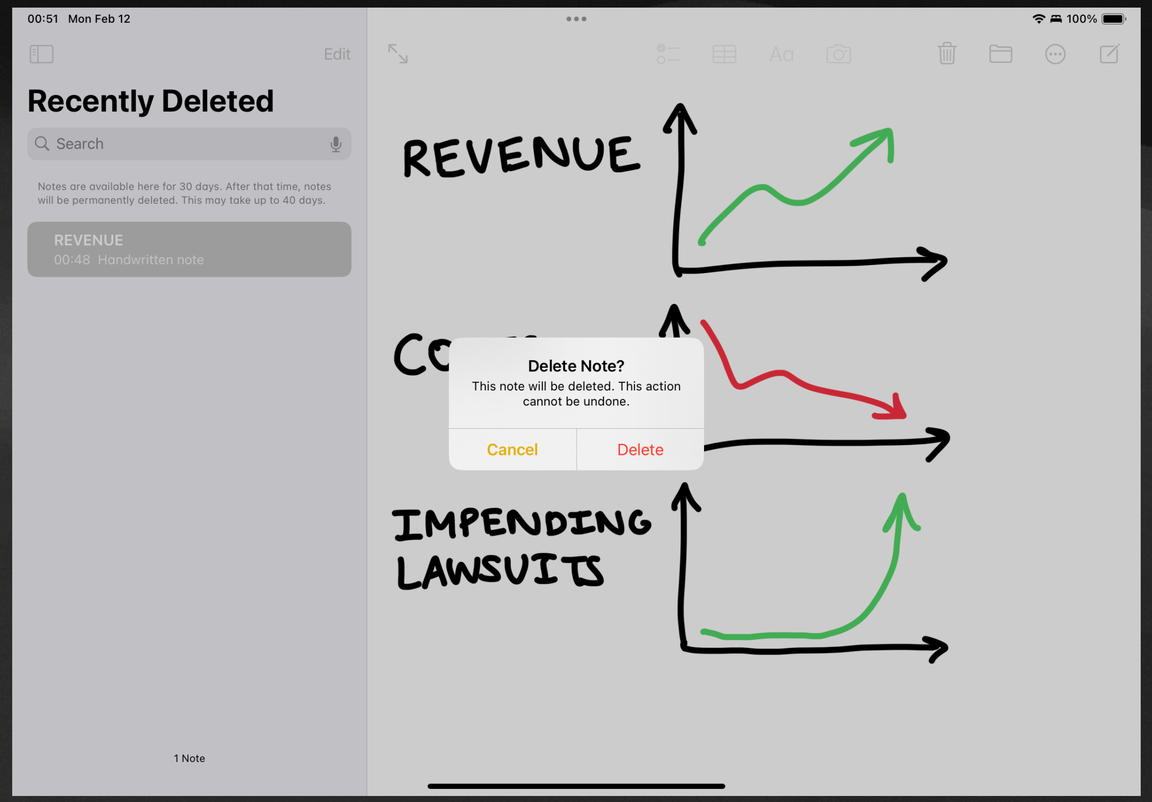
Hierarchical



Modals

Example: Notes App





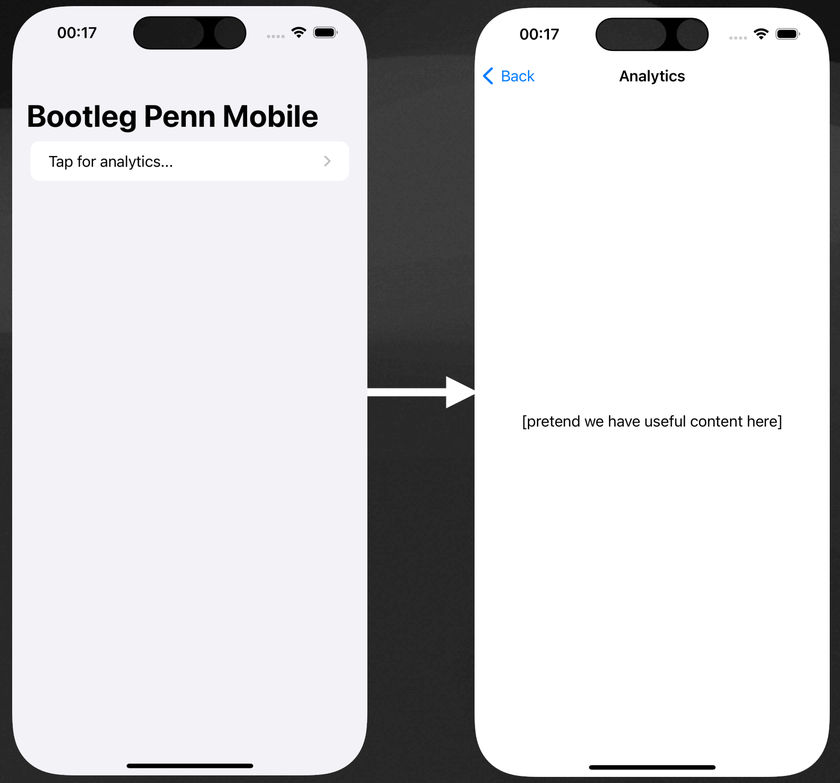
Lightweight and **focused** interactions

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(path: $path) {
    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

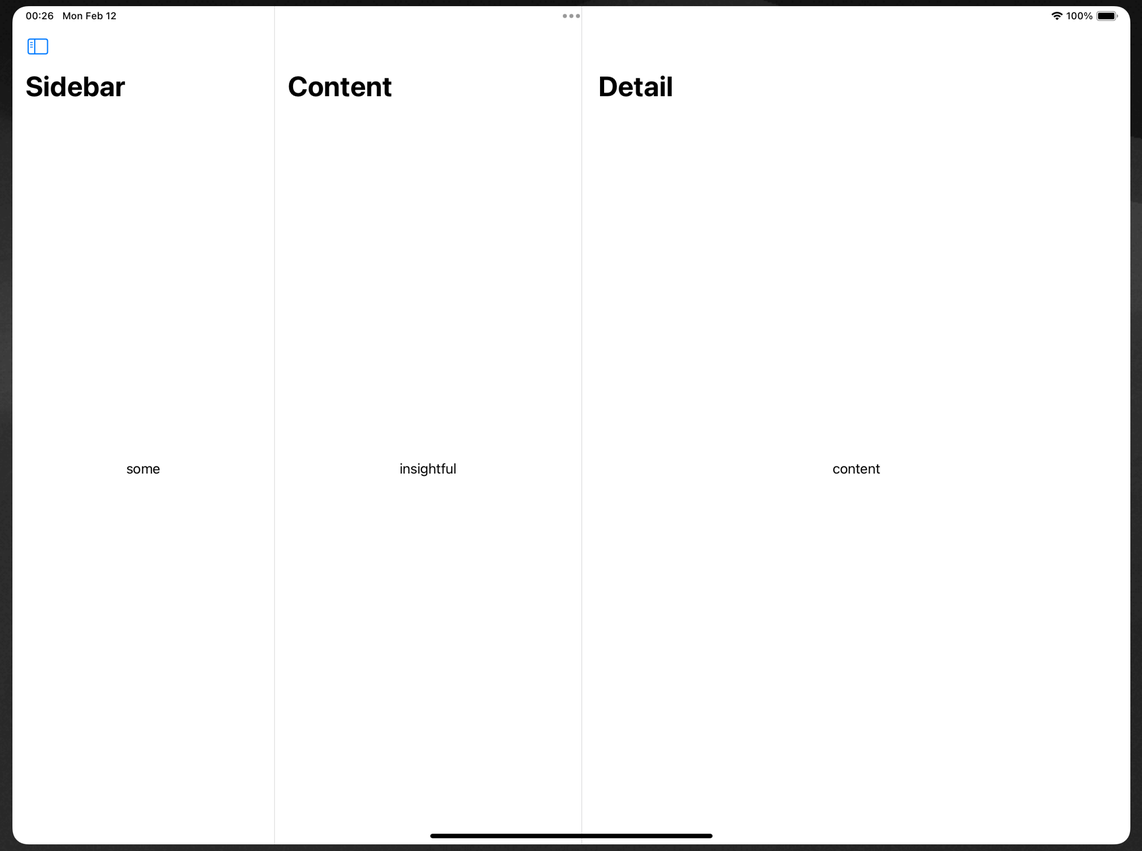
```
@State var path = NavigationPath()
...

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

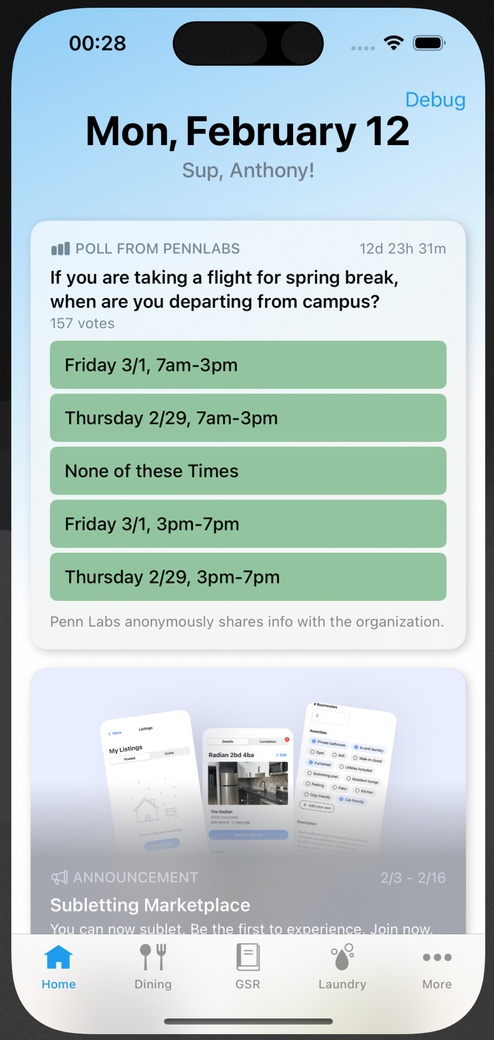
Multi-column layouts

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



Appears as a NavigationStack on iPhone

TabView



developer.apple.com/documentation/swiftui/tabview

Developer | News | Discover | Design | Develop | Distribute | Support | Account

SwiftUI

- Presenting views in tabs
- TabView
  - Creating a tab view
    - init(content: () -> Content)
    - init(selection: Binding<SelectionValue>?, c...
  - func tabViewStyle<S>(S) -> some View
  - func tabItem<V>(() -> V) -> some View
  - Displaying views in multiple panes
  - HSplitView
  - VSplitView
  - Deprecated Types
  - NavigationView Deprecated
- Modal presentations
- Toolbars
- Search
- App extensions
- Data and storage
- Model data
- Environment values
- Preferences

Filter /

Documentation / ··· / Navigation / TabView

Language: Swift API Changes: None

Structure

TabView

A view that switches between multiple child views using interactive user interface elements.

iOS 13.0+ | iPadOS 13.0+ | macOS 10.15+ | Mac Catalyst 13.0+ | tvOS 13.0+ | watchOS 7.0+ | visionOS 1.0+

```
struct TabView<SelectionValue, Content> where SelectionValue : Hashable, Conten
```

Overview

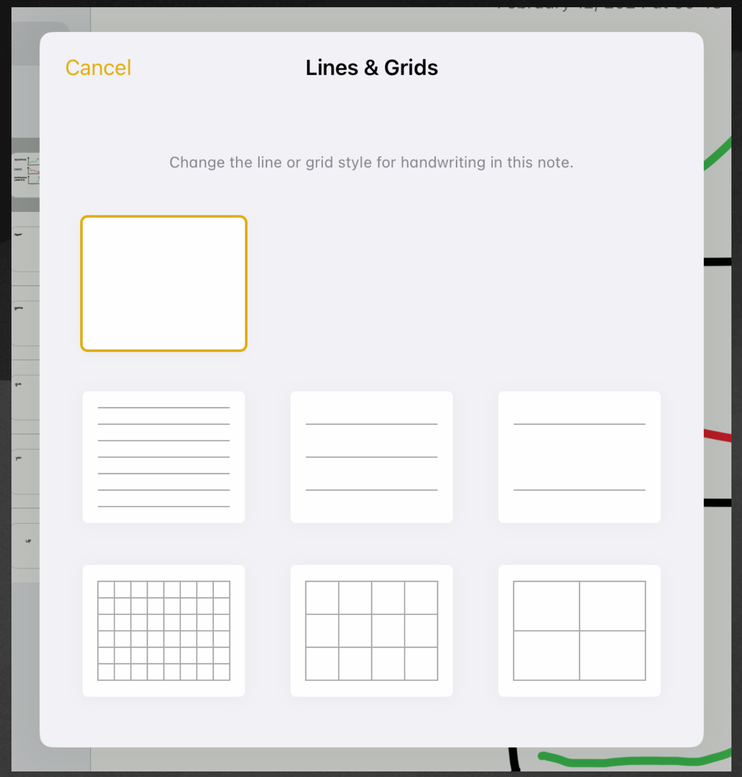
To create a user interface with tabs, place views in a `TabView` and apply the `tabItem(_:)` modifier to the contents of each tab. On iOS, you can also use one of the badge modifiers, like `badge(_:)`, to assign a badge to each of the tabs.

The following example creates a tab view with three tabs, each presenting a custom child view. The first tab has a numeric badge and the third has a string badge.

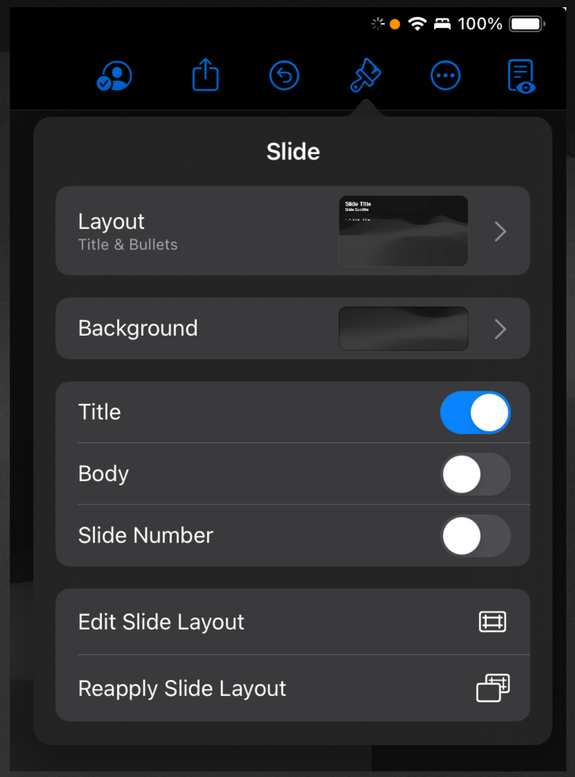
```
TabView {
    ReceivedView()
        .badge(2)
```

Modal presentations

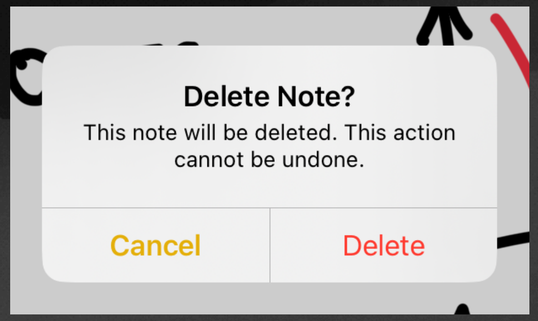
For lightweight, focused interactions



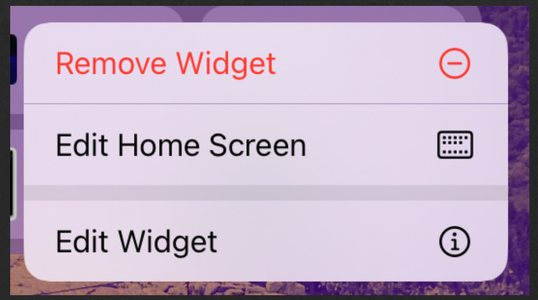
.sheet



.popover



.alert



Menu
— OR —
.contextMenu

developer.apple.com/videos/play/wwdc2022/10054/

Developer
Open in the Developer app

Open

Videos

Collections Topics All Videos About

The SwiftUI cookbook for navigation

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.

Resources

- Bringing robust navigation structure to your SwiftUI app
- Have a question? Ask with tag wwdc2022-10054
- List
- Migrating to new navigation types
- NavigationSplitView
- NavigationStack
- Search the forums for tag wwdc2022-10054
- HD Video | SD Video

Related Videos

Tech Talks

- What's new for enterprise developers

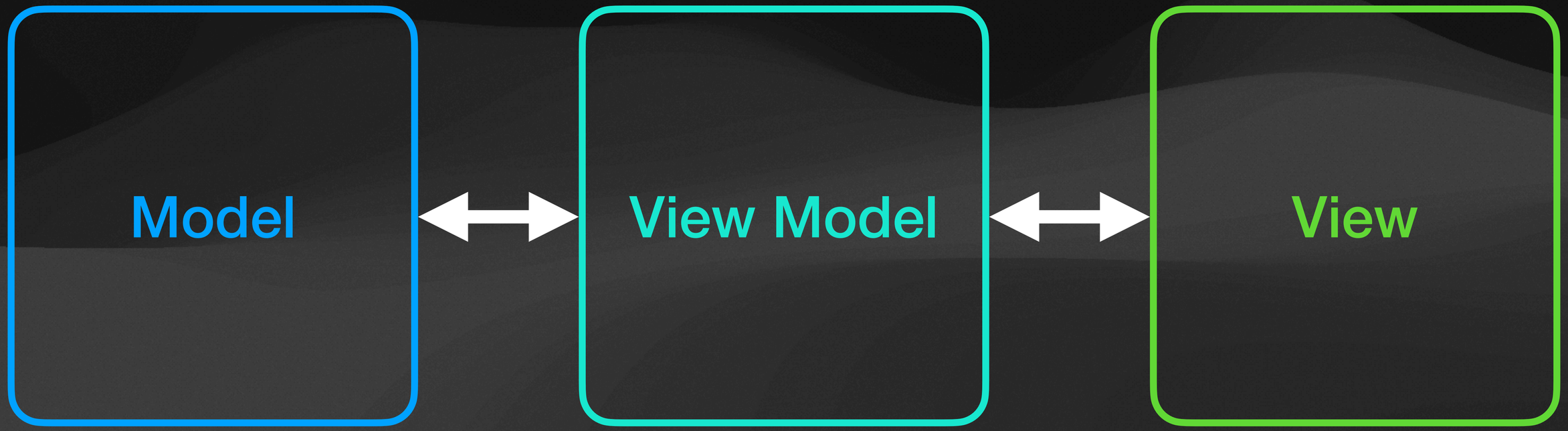
MVVM

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



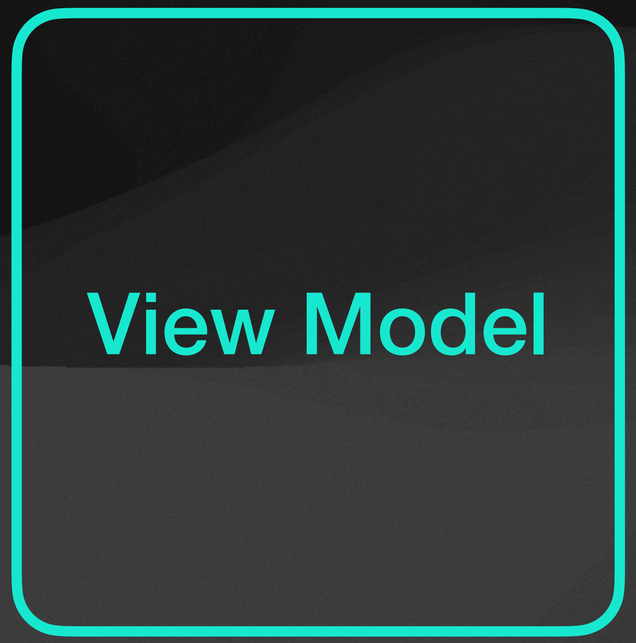
Represents **the app's data**

Coordinates between the two

Defines **what the user sees**

MVVM

The View Model



Coordinates
between the two

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

Notifies the view of any changes

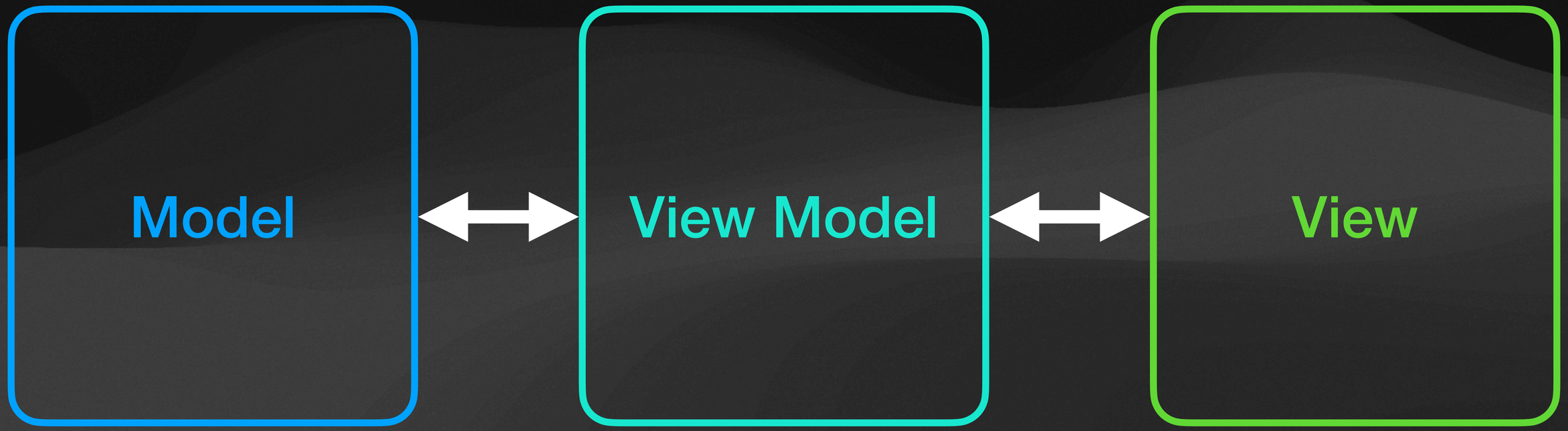
Converts data to and from what the view wants

Isolates the view from its underlying data

Usually **a class**

MVVM

How we'll use it



Course

Contains the actual course data

CoursesViewModel

Manages favorited courses and filters them for the view

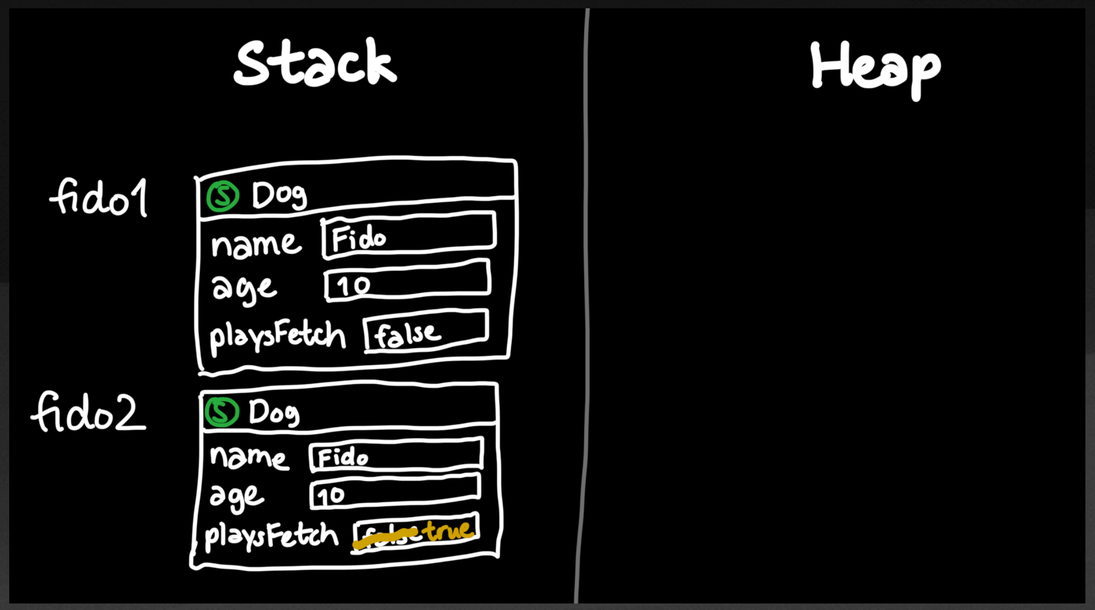
CoursesView

Displays the list of courses to the user

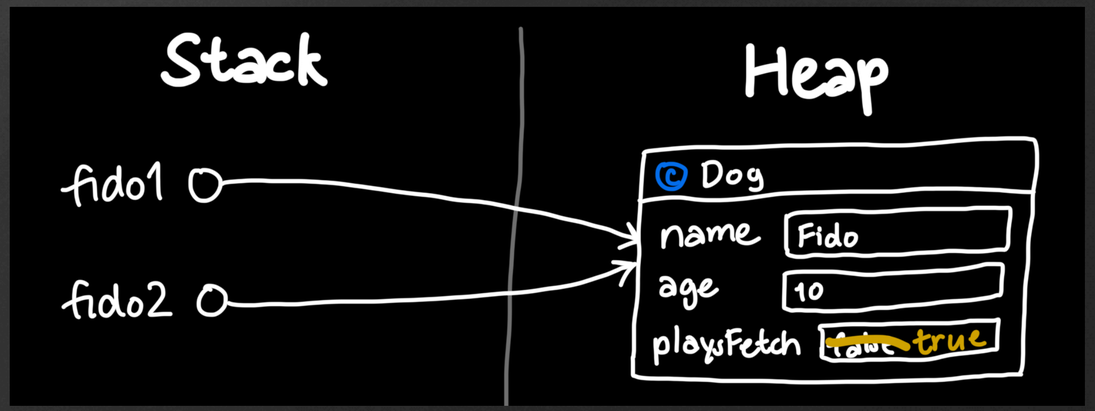
@Observable

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

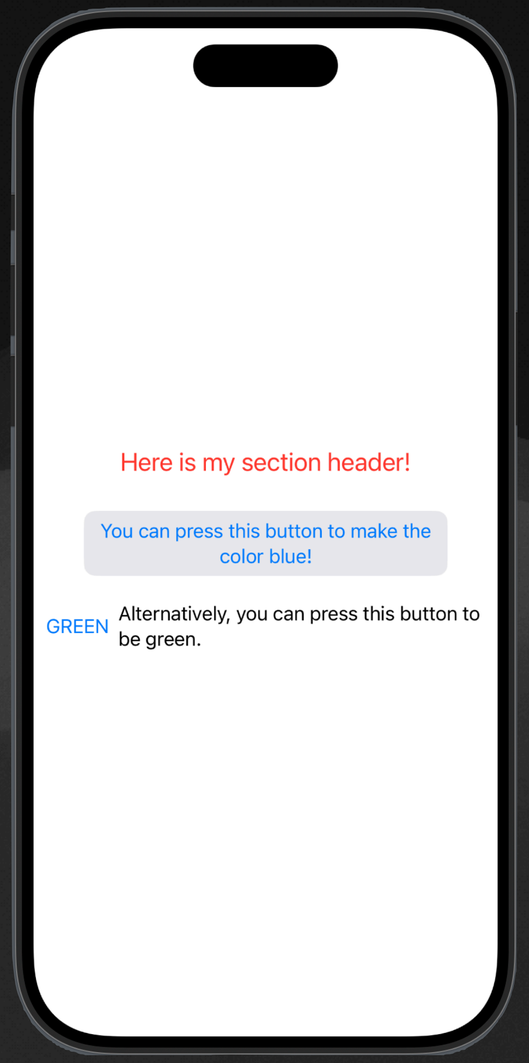
Bindings for Observables

- 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)
                .foregroundStyle(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)
                .foregroundStyle(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

developer.apple.com/documentation/swiftui/migrating-from-the-ob

Documentation

Language: Swift API changes: None

All Technologies

SwiftUI

Essentials

- Introducing SwiftUI
- Learning SwiftUI
- Exploring SwiftUI Sample Apps
- SwiftUI updates

App structure

- App organization
- Scenes
- Windows
- Immersive spaces
- Documents
- Navigation
- Modal presentations
- Toolbars
- Search
- App extensions

Data and storage

- Model data

Filter /

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]