The View Hierarchy as a Communication Channel

Emilio Peláez

iOS Developer since 2008

Mobile Engineering Director @ Modus Create

Motivation

Sending data and messages between components in an app is a common challenge

Whole architectures are created to solve it

We're going to explore the tools available in SwiftUI

And find interesting ways of using them!

Understanding the View Tree

```
VStack(alignment: .leading, spacing: 8) {
  Text("Emilio Peláez")
    .font(.headline)
    .foregroundStyle(.primary)
  VStack(alignment: .leading, spacing: 4) {
    Text("iOS Developer")
    Text("Modus Create")
  }
  .font(.callout)
  .foregroundStyle(.secondary)
}
```

Emilio Peláez

iOS Developer since 2008

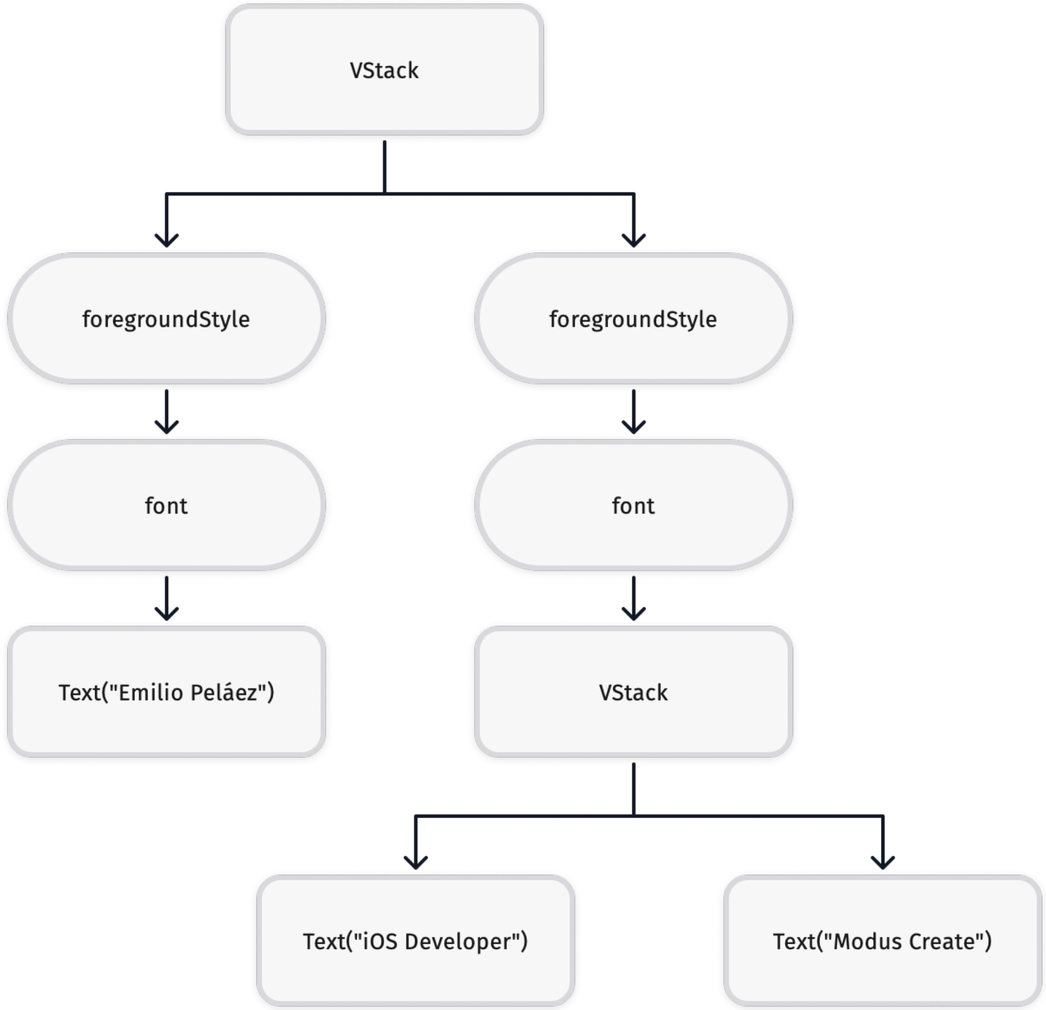
Modus Create

Understanding the View Tree

```
VStack(alignment: .leading, spacing: 8) {
  Text("Emilio Peláez")
    .font(.headline)
    .foregroundStyle(.primary)
  VStack(alignment: .leading, spacing: 4) {
    Text("iOS Developer")
    Text("Modus Create")
  }
  .font(.callout)
  .foregroundStyle(.secondary)
}
```

Understanding the View Tree

```
VStack(alignment: .leading, spacing: 8) {
  Text("Emilio Peláez")
    .font(.headline)
    .foregroundStyle(.primary)
  VStack(alignment: .leading, spacing: 4) {
    Text("iOS Developer")
    Text("Modus Create")
  }
  .font(.callout)
  .foregroundStyle(.secondary)
}
```



Communication Channels: The Basics

Init Parameters

- Single-level communication

Communication Channels: The Basics

Init Parameters

Single-level communication

Bindings

Wraps a value to allow a child to modify it

Communication Channels: The Basics

Init Parameters

Single-level communication

Bindings

Wraps a value to allow a child to modify it

Observable Objects

Encapsulates logic and values into a class

Communication Channels: The Basics

Init Parameters

Single-level communication

Bindings

Wraps a value to allow a child to modify it

Observable Objects

Encapsulates logic and values into a class

Closures

AKA Annonymous Functions

The Swift feature we all know and love

Communication Channels: The Environment

What is it?

A collection of data that gets propagated to every view in the view hierarchy

Communication Channels: The Environment

What is it?

- A collection of data that gets propagated to every view in the view hierarchy

Environment Objects

- Injects Observable Objects into the hierarchy
- Very easy to use and powerful
- Easy to overuse

Communication Channels: The Environment

What is it?

- A collection of data that gets propagated to every view in the view hierarchy

Environment Objects

- Injects Observable Objects into the hierarchy
- Very easy to use and powerful
- Easy to overuse

Environment Values

- Key-Value Store
- Defining them requires a default value

Communication Channels: The Environment

What is it?

- A collection of data that gets propagated to every view in the view hierarchy

Environment Objects

- Injects Observable Objects into the hierarchy
- Very easy to use and powerful
- Easy to overuse

Environment Values

- Key-Value Store
- Defining them requires a default value
- Closures*
- Bindings
- Not Objects

Communication Channels: Preferences

What are they?

"Reverse" Environment Values

Communication Channels: Preferences

What are they?

"Reverse" Environment Values

PreferenceKey Protocol

Only one public type conforms to it

But don't use it!

Almost a dozen modifiers

PreferredColorSchemeKey

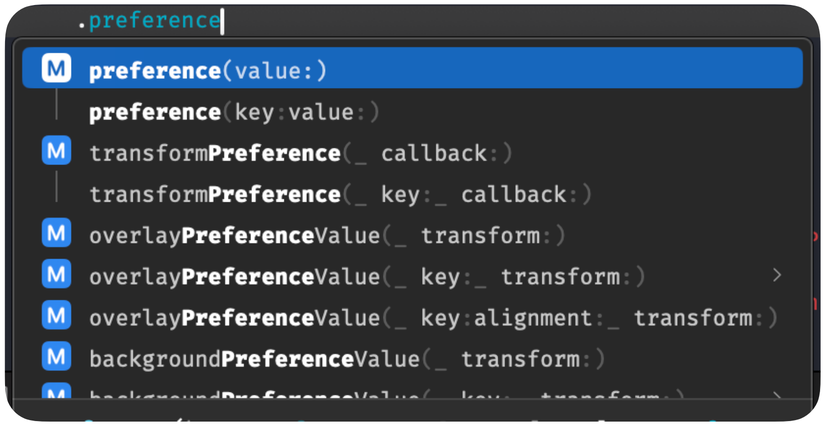
A key for specifying the preferred color scheme.

iOS 13.0+ | iPadOS 13.0+ | macOS 11.0+ | Mac Catalyst 13.0+ | tvOS 13.0+ | watchOS 6.0+ | visionOS 1.0+ Beta

```
struct PreferredColorSchemeKey
```

Overview

Don’t use this key directly. Instead, set a preferred color scheme for a view using the `preferredColorScheme(_:)` view modifier. Get the current color scheme for a view by accessing the `colorScheme` value.



Communication Channels: Preferences

What are they?

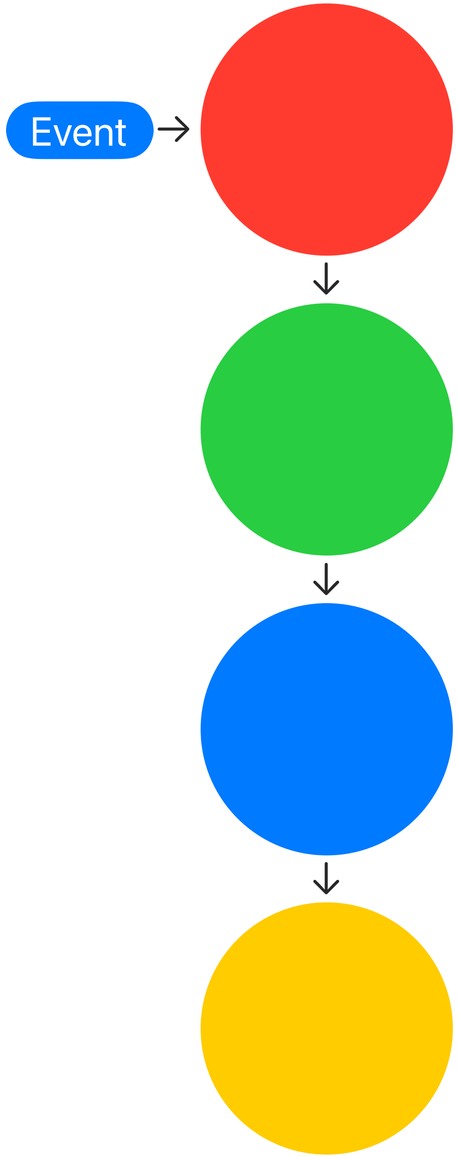
- "Reverse" Environment Values

PreferenceKey Protocol

- Only one public type conforms to it
- But don't use it!
- Almost a dozen modifiers

Responder Chains

- Chain of connected responders
- Responders may handle some arbitrary events
- If a responder can't handle an event, it's passed along



Responders in the Environment

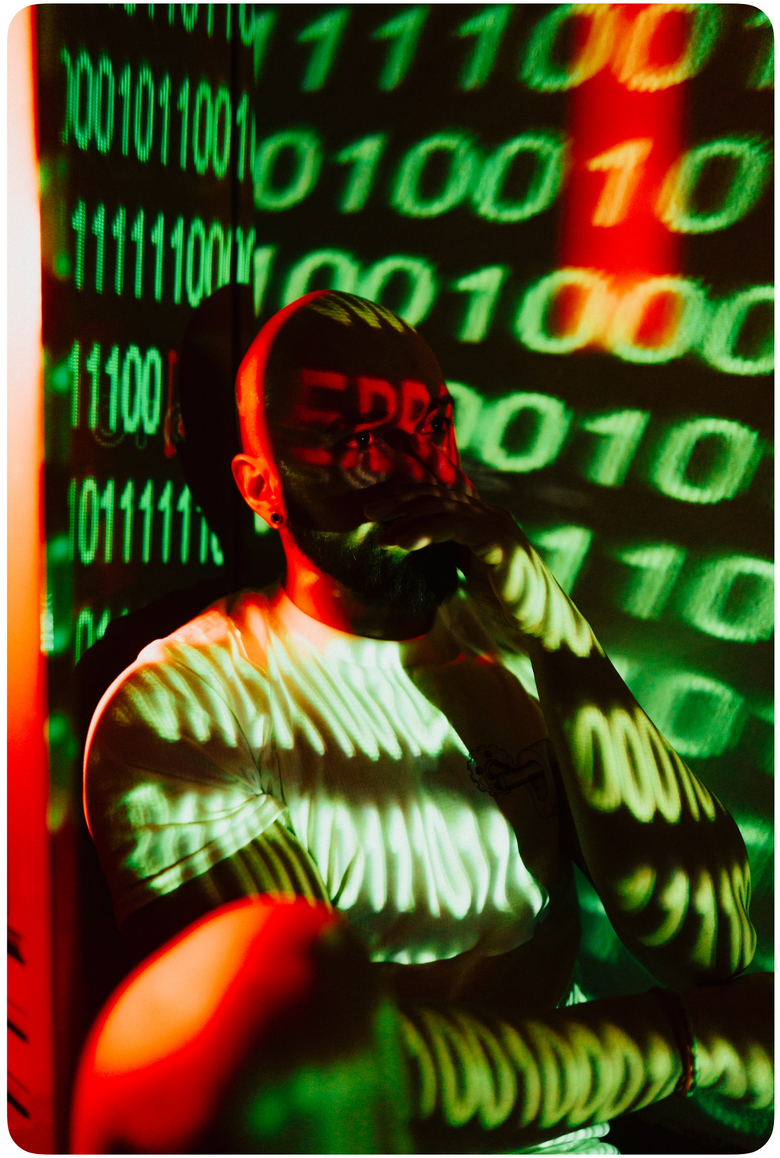
```
struct TriggerEvent {
  let execute: (Any) -> Void

  func callAsFunction(_ event: Any) {
    execute(event)
  }
}

// let trigger = TriggerEvent { _ in }
// trigger.execute("Hello")
// trigger("Hello")

struct TriggerEventKey: EnvironmentKey {
  static var defaultValue: TriggerEvent = .init { _ in
print("Unhandled Event") }
}

extension EnvironmentValues {
  var triggerEvent: TriggerEvent {
    get { self[TriggerEventKey.self] }
    set { self[TriggerEventKey.self] = newValue }
  }
}
```



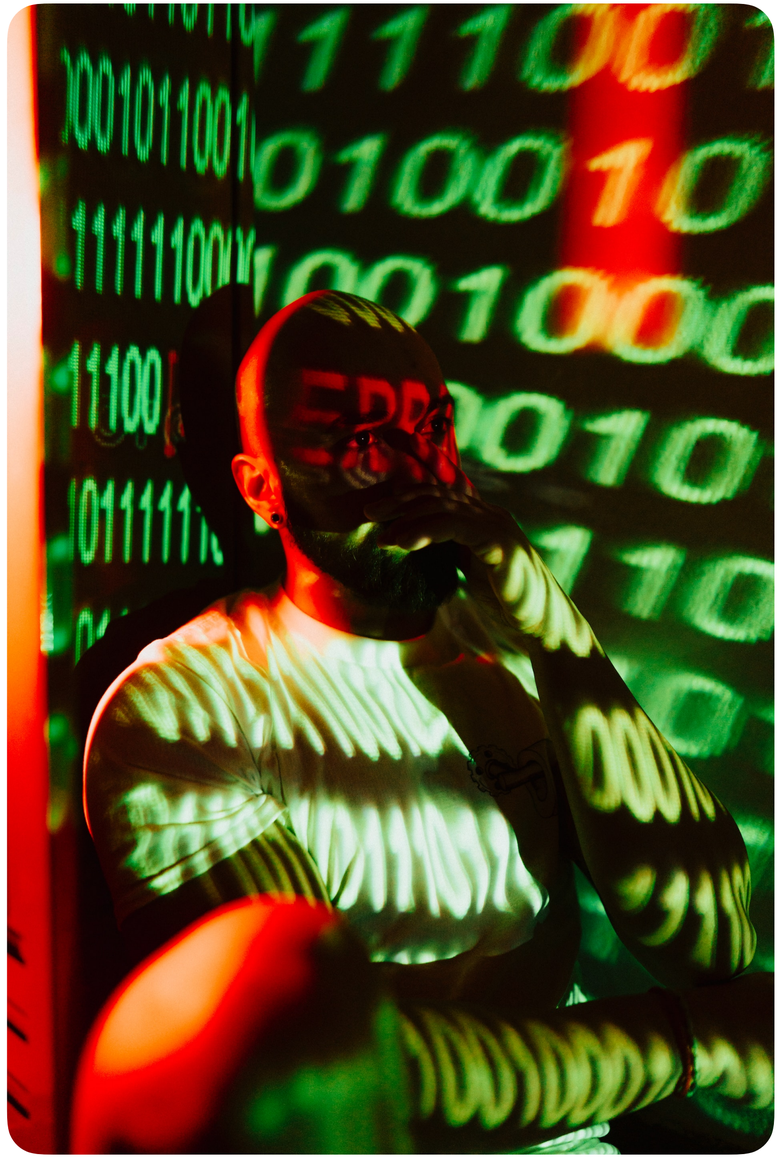
Responders in the Environment

```
struct EventHandlerModifier: ViewModifier {
  @Environment(\.triggerEvent) var triggerEvent

  let handler: (Any) -> Any?

  func body(content: Content) -> some View {
    content.environment(\.triggerEvent, TriggerEvent {
      if let event = handler($0) {
        triggerEvent(event)
      }
    })
  }
}

extension View {
  func receiveEvent(_ handler: @escaping (Any) -> Any?) -> some View {
    modifier(EventHandlerModifier(handler: handler))
  }
}
```



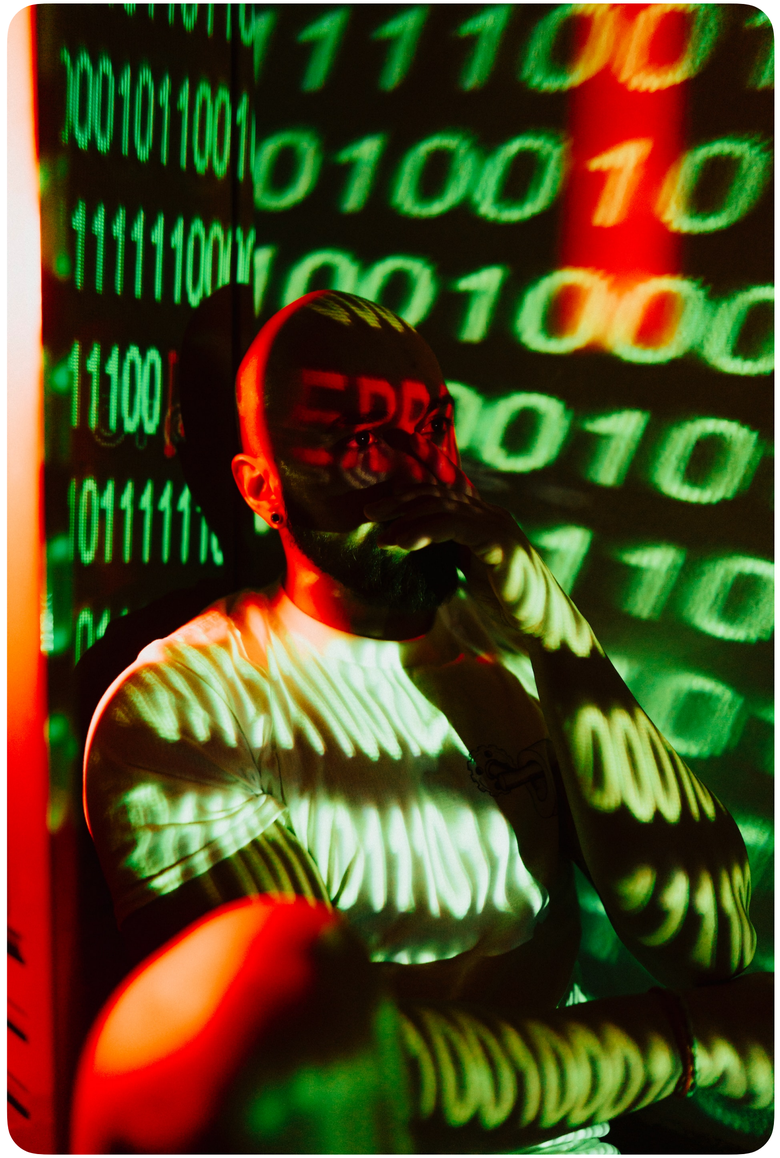
Responders in the Environment

```
struct CoolEvent {}

struct CoolView: View {
  @Environment(\.triggerEvent) var triggerEvent

  var body: some View {
    Button("😎🏂😎") {
      triggerEvent(CoolEvent())
    }
  }
}

struct CoolModifier: ViewModifier {
  func body(content: Content) -> some View {
    content
      .receiveEvent { event in
        if event is CoolEvent {
          print("😎🏂😎")
          return nil
        } else {
          return event
        }
      }
  }
}
```



Error Handling

```
struct MainView: View {
  @State var showAlert = false
  @State var error: AuthorizationError?

  var body: some View {
    Text("Welcome!")
      .task {
        do {
          try ContentService.loadContent()
        } catch is AuthorizationError {
          self.error = error
          showAlert = true
        } catch {
          print("Something went wrong")
        }
      }
      .alert(isPresented: $showAlert,
             error: error) {
        Button("Ok") {}
      }
  }
}
```

Error Handling

```
struct MyApp: App {
  var body: some Scene {
    WindowGroup {
      ContentView()
        .modifier(ErrorAlertModifier())
    }
  }
}

struct ContentView: View {
  @Environment(\.triggerEvent) var triggerEvent

  var body: some View {
    Text("Welcome")
      .task {
        do {
          try ContentService.loadContent()
        } catch {
          triggerEvent(error)
        }
      }
  }
}
```

```
struct ErrorAlertModifier: ViewModifier {
  @State var showAlert = false
  @State var errorMessage: String?

  func body(content: Content) -> some View {
    content
      .receiveEvent { event in
        guard let error = event as?
LocalizedError else {
          return event
        }
        showAlert = true
        errorMessage = error.localizedDescription
        return nil
      }
      .alert("Uh oh", isPresented: $showAlert) {
        Button("Ok") {}
      } message: {
        Text(errorMessage ?? "Unknown Error")
      }
  }
}
```

Error Handling

```
struct MyApp: App {
  var body: some Scene {
    WindowGroup {
      ContentView()
        .receiveEvent { event in
          if let error = event as? Error {
            AnalyticsService.log(error)
          }
          return event
        }
        .modifier(ErrorAlertModifier())
    }
  }
}
```

Error Handling

```
struct MyApp: App {
  var body: some Scene {
    WindowGroup {
      ContentView()
        .receiveEvent { event in
          if event is AuthorizationError {
            return ShowLoginEvent()
          } else {
            return event
          }
        }
        .modifier(LoginModifier())
        .receiveEvent { event in
          if let error = event as? Error {
            AnalyticsService.log(error)
          }
          return event
        }
        .modifier(ErrorAlertModifier())
    }
  }
}
```

```
struct ShowLoginEvent {}

struct LoginModifier: ViewModifier {
  @State var showLogin = false

  func body(content: Content) -> some View {
    content
      .receiveEvent { event in
        guard event is ShowLoginEvent else {
          return event
        }
        showLogin = true
        return nil
      }
      .sheet(isPresented: $showLogin) {
        LoginView()
      }
  }
}
```

Loading Remote Images

```
struct ImageRequest {
  let path: String
}

struct ImageResponder {
  let perform: (ImageRequest)
  async throws -> Image

  func callAsFunction(_ request: ImageRequest)
  async throws -> Image {
    try await perform(request)
  }
}

struct ResourceResponderKey: EnvironmentKey {
  static var defaultValue: ImageResponder =
    .init { _ in fatalError("Unhandled Request") }
}
```

```
struct ImageDisplay: View {
  @Environment(\.imageResponder) var imageResponder

  let path: String
  @State var image: Image?

  var body: some View {
    ZStack {
      Color.gray
      if let image {
        image
      }
    }
    .task {
      do {
        let request = ImageRequest(path: path)
        image = try await imageResponder(request)
      } catch {
        image = Image(systemName:
"exclamationmark.triangle")
      }
    }
  }
}
```

Loading Remote Images

```
ImageDisplay(path: "corgi")
  .imageResponder { request in
    let data = ResourceClient.fetch(request.path)
    guard let uiImage = UIImage(data: data) else {
      throw RemoteImageError.invalidData
    }
    return Image(uiImage: uiImage)
  }
```

```
struct ImageDisplay: View {
  @Environment(\.imageResponder) var imageResponder

  let path: String
  @State var image: Image?

  var body: some View {
    ZStack {
      Color.gray
      if let image {
        image
      }
    }
    .task {
      do {
        let request = ImageRequest(path: path)
        image = try await imageResponder(request)
      } catch {
        image = Image(systemName:
"exclamationmark.triangle")
      }
    }
  }
}
```

Loading Remote Images

```
ImageDisplay(path: "corgi")
  .imageResponder { request in
    guard let uiImage = UIImage(named:
request.path) else {
      return nil
    }
    return Image(uiImage: uiImage)
  }
  .imageResponder { request in
    let data = ResourceClient.fetch(request.path)
    guard let uiImage = UIImage(data: data) else {
      throw RemoteImageError.invalidData
    }
    return Image(uiImage: uiImage)
  }
```

```
struct ImageDisplay: View {
  @Environment(\.imageResponder) var imageResponder

  let path: String
  @State var image: Image?

  var body: some View {
    ZStack {
      Color.gray
      if let image {
        image
      }
    }
    .task {
      do {
        let request = ImageRequest(path: path)
        image = try await imageResponder(request)
      } catch {
        image = Image(systemName:
"exclamationmark.triangle")
      }
    }
  }
}
```

Wrapping Up

All views in a SwiftUI app are connected

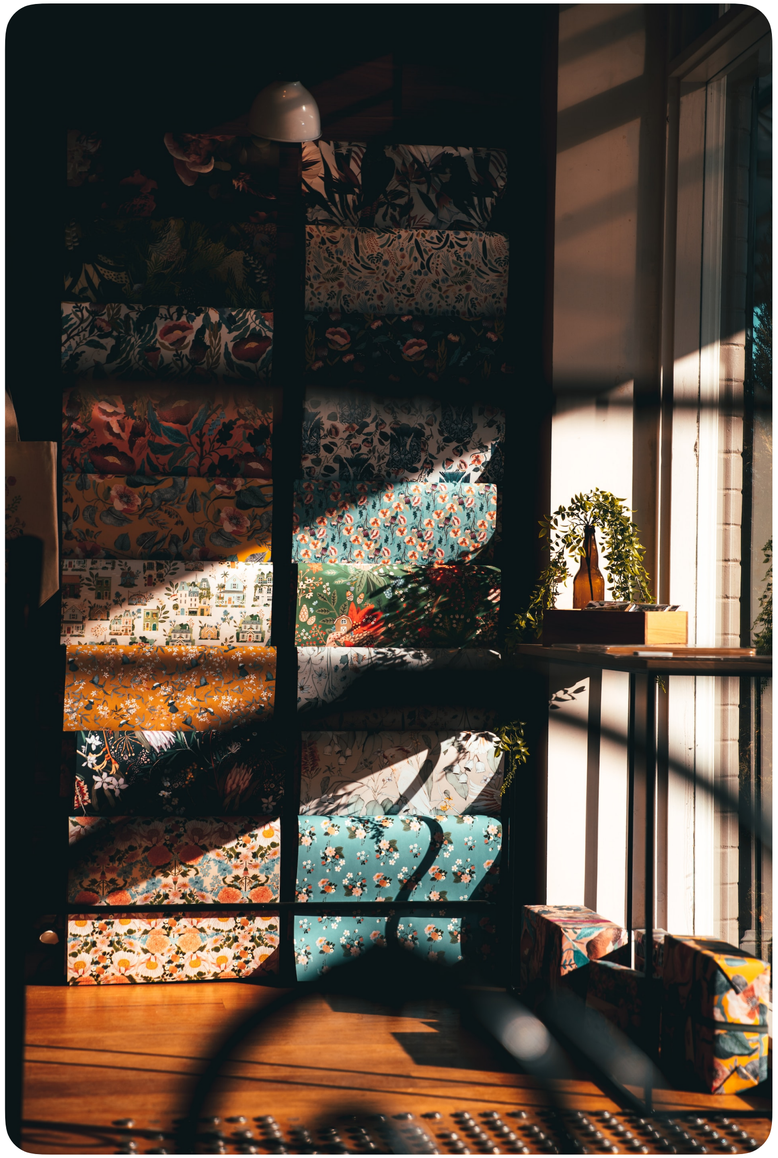
- Thanks to the View Hierarchy!

The View Hierarchy has multiple communication channels

- The Environment
- Preferences

The Environment and Preferences are very flexible

- At the cost of having no compile-time safety



Thank you!

Questions? Comments?

Follow me on Mastodon!

Built with SlideKit