“What I love most about development is design”

“What I love most about development is API design”

Crafting SwiftUI components.

The same way Apple does?

Do iOS 2025

Thomas Durand

Call me Dean

https://thomasdurand.fr

@deanatoire@mastodon.social

bsky.app/@thomasdurand.fr

Thomas Durand

Call me Dean

Backend architect at DiliTrust

https://thomasdurand.fr

@deanatoire@mastodon.social

bsky.app/@thomasdurand.fr

Thomas Durand

Call me Dean

Backend architect at DiliTrust

iOS indie dev

https://thomasdurand.fr

@deanatoire@mastodon.social

bsky.app/@thomasdurand.fr

Thomas Durand

Call me Dean

Backend architect at DiliTrust

iOS indie dev

Speaker, tech blog writer, enthusiast

https://thomasdurand.fr

@deanatoire@mastodon.social

bsky.app/@thomasdurand.fr

Thomas Durand

Call me Dean

Backend architect at DiliTrust

iOS indie dev

Speaker, tech blog writer, enthusiast

Building ButtonKit

https://thomasdurand.fr

@deanatoire@mastodon.social

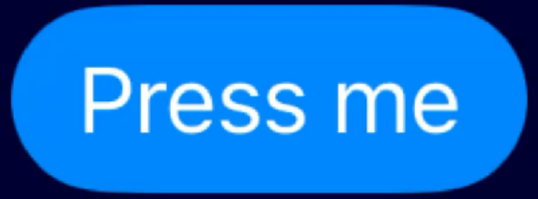
bsky.app/@thomasdurand.fr

ButtonKit

https://github.com/Dean151/ButtonKit

ButtonKit

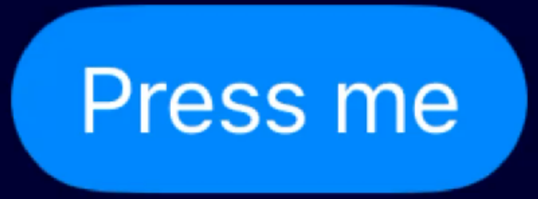
```
AsyncButton {
    await ...
} label: {
    Text("Press me")
}
.buttonStyle(.borderedProminent)
```



https://github.com/Dean151/ButtonKit

ButtonKit

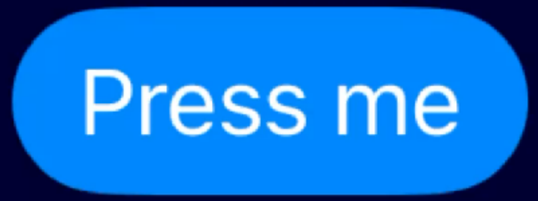
```
AsyncButton {
    await ...
} label: {
    Text("Press me")
}
.buttonStyle(.borderedProminent)
```



https://github.com/Dean151/ButtonKit

ButtonKit

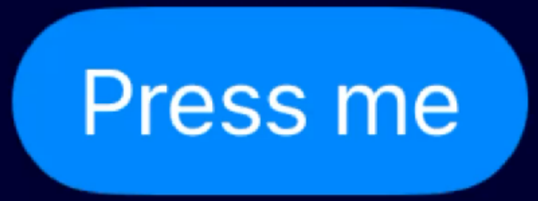
```
AsyncButton {
    try await ...
} label: {
    Text("Press me")
}
.buttonStyle(.borderedProminent)
```



https://github.com/Dean151/ButtonKit

ButtonKit

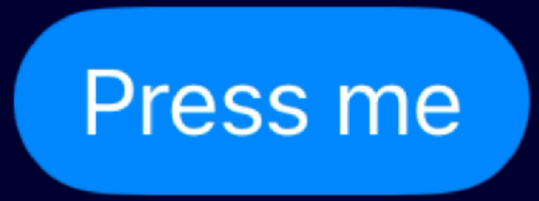
```
AsyncButton {
    try await ...
} label: {
    Text("Press me")
}
.buttonStyle(.borderedProminent)
```



https://github.com/Dean151/ButtonKit

ButtonKit

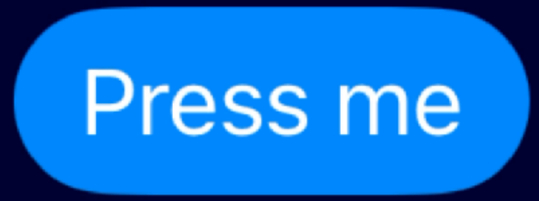
```
AsyncButton(
    progress: .estimated(for: .seconds(1))
) { progress in
   try await ...
} label: {
    Text("Press me")
}
.buttonStyle(.borderedProminent)
```



https://github.com/Dean151/ButtonKit

ButtonKit

```
AsyncButton(
    progress: .estimated(for: .seconds(1))
) { progress in
   try await ...
} label: {
    Text("Press me")
}
.buttonStyle(.borderedProminent)
```



https://github.com/Dean151/ButtonKit

The challenge?

The challenge?

Build an API that feels “SwiftUI”

What does SwiftUI feels like?

What does SwiftUI feels like?

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

What does SwiftUI feels like?

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

Image(.checkmark)
```

What does SwiftUI feels like?

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

Image(.checkmark)

Button("Press me", action: doSomething)
```

SwiftUI is declarative

SwiftUI is declarative

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

Image(.checkmark)

Button("Press me", action: doSomething)
```

SwiftUI is declarative

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

Image(.checkmark)

Button("Press me", action: doSomething)
```

SwiftUI is declarative

```
Text

Image

Button
```

SwiftUI is straight to the point

SwiftUI is straight to the point

Required parameters kept to minimal

SwiftUI is straight to the point

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

Image(.checkmark)

Button("Press me", action: doSomething)
```

SwiftUI is straight to the point

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

Image(.checkmark)

Button("Press me", action: doSomething)
```

```
Text(...)
Image(...)
Button(...)
```

```
VStack {
    Text(...)
    Image(...)
    Button(...)
}
```

SwiftUI is composable

SwiftUI is composable

```
VStack {
    Text(...)
    Image(...)
    Button(...)
}
```

SwiftUI is composable

```
HStack {
    Text(...)
    Image(...)
    Button(...)
}
```

SwiftUI is composable

```
ZStack {
    Text(...)
    Image(...)
    Button(...)
}
```

SwiftUI is composable

```
List {
    Text(...)
    Image(...)
    Button(...)
}
```

SwiftUI is composable

```
List {
    Text(...)
    Image(...)
    Button(...)
}
.font(.callout)
```

SwiftUI is composable

```
List {
    Text(...).font(.title)
    Image(...)
    Button(...)
}
.font(.callout)
```

SwiftUI is opinionated

SwiftUI is opinionated

“Best defaults” are chosen for you

SwiftUI is opinionated

```
VStack {
    Text(...)
    Image(...)
    Button(...)
}
```

SwiftUI is opinionated

```
VStack(alignment: .leading, spacing: 24) {
    Text(...)
    Image(...)
    Button(...)
}
```

SwiftUI is about "clean" call sites

Focus on your views call sites

Focus on your views inputs & outputs

Build a component

```
RatingStars(rating: 1.0)
```

```
RatingStars(rating: 0.4)
```

Required argument

```
RatingStars(rating: 1.0)
```

Optional input argument

```
RatingStars(rating: 1.0)
RatingStars(rating: 0.8, numberOfStars: 7)
```

Custom environment value

```
RatingStars(rating: 1.0)
RatingStars(rating: 0.8)
    .environment(\.numberOfStars, 7)
```

Custom environment value

@Entry macro

```
extension EnvironmentValues {
    @Entry var numberOfStars = 5
}
```

Custom environment value

@Entry macro

```
extension EnvironmentValues {
    @Entry var numberOfStars = 5
}
```

```
@Environment(\.numberOfStars) var numberOfStars
```

Custom environment value

```
RatingStars(rating: 1.0)
RatingStars(rating: 0.8)
    .environment(\.numberOfStars, 7)
```

Custom environment value

View extension

```
extension EnvironmentValues {
    @Entry var numberOfStars = 5
}
```

Custom environment value

View extension

```
extension EnvironmentValues {
    @Entry var numberOfStars = 5
}
extension View {
    func numberOfStars(_ count: Int) -> some View {
        environment(\.starCount, count)
    }
}
```

Custom environment value

View extension

```
extension EnvironmentValues {
    @Entry fileprivate(set) var numberOfStars = 5
}
extension View {
    func numberOfStars(_ count: Int) -> some View {
        environment(\.starCount, count)
    }
}
```

Custom environment value

```
RatingStars(rating: 1.0)
RatingStars(rating: 0.8)
    .environment(\.numberOfStars, 7)
```

Custom environment value

```
RatingStars(rating: 1.0)
RatingStars(rating: 0.8)
    .numberOfStars(7)
```

Custom environment value

```
Group {
    RatingStars(rating: 1.0)
    RatingStars(rating: 0.8)
}
.numberOfStars(7)
```

Don't hesitate to put optional behavior behind a view modifier

Build a control component

Binding argument

```
RatingStars(rating: $rating)
```

Built-in controls modifiers

```
RatingStars(rating: $rating)
    .controlSize(.large)
```

Built-in controls modifiers

```
RatingStars(rating: $rating)
    .controlSize(.large)
    .disabled(true)
```

Built-in controls modifiers

```
RatingStars(rating: $rating)
    .controlSize(.large)
    .disabled(true)
```

```
@Environment(\.controlSize) var controlSize
@Environment(\.isEnabled)   var isEnabled
```

```
SpeechButton(text: $text)
```

Read-only output?

```
enum SpeechRecognizerState {
    case idle
    case listening
    case processing
    case completed(reason: CompletionReason)

    enum SpeechCompletionReason { ... }
}
```

Read-only output?

```
SpeechButton(text: $text, state: $state)
```

Read-only output?

```
SpeechButton(text: $text, state: $state)
```

Read-only output

Callback argument

```
SpeechButton(text: $text, onStateChanged: { state in
    ...
})
```

Read-only output

Callback argument

```
SpeechButton(text: $text) { state in
    ...
}
```

Read-only output

Callback argument

```
SpeechButton(onStateChanged: { state in
    ...
}, text: $text)
```

Read-only output

Callback modifier 😌

```
SpeechButton(text: $text)
    .onStateChanged { state in
        ...
    }
```

Callback modifier

```
typealias StateChangedHandler = (SpeechRecognizerState) -> Void
```

Callback modifier

```
typealias StateChangedHandler = (SpeechRecognizerState) -> Void

extension View {
    func onStateChanged(
      _ handler: @escaping StateChangedHandler
    ) -> some View {
        ...
    }
}
```

Callback modifier

```
typealias StateChangedHandler = (SpeechRecognizerState) -> Void

extension View {
    func onStateChanged( ... ) -> some View {

    }
}
```

Callback modifier

```
typealias StateChangedHandler = (SpeechRecognizerState) -> Void

extension View {
    func onStateChanged( ... ) -> some View {
        environment(\.stateChanged, handler)
    }
}

extension EnvironmentValues {
    @Entry var stateChanged: StateChangedHandler? = nil
}
```

Callback modifier

```
SpeechButton(text: $text)
    .onStateChanged { print("First one", $0) }
    .onStateChanged { print("Second one", $0) }
```

Callback modifier

```
SpeechButton(text: $text)
    .onStateChanged { print("First one", $0) }
    .onStateChanged { print("Second one", $0) }
```

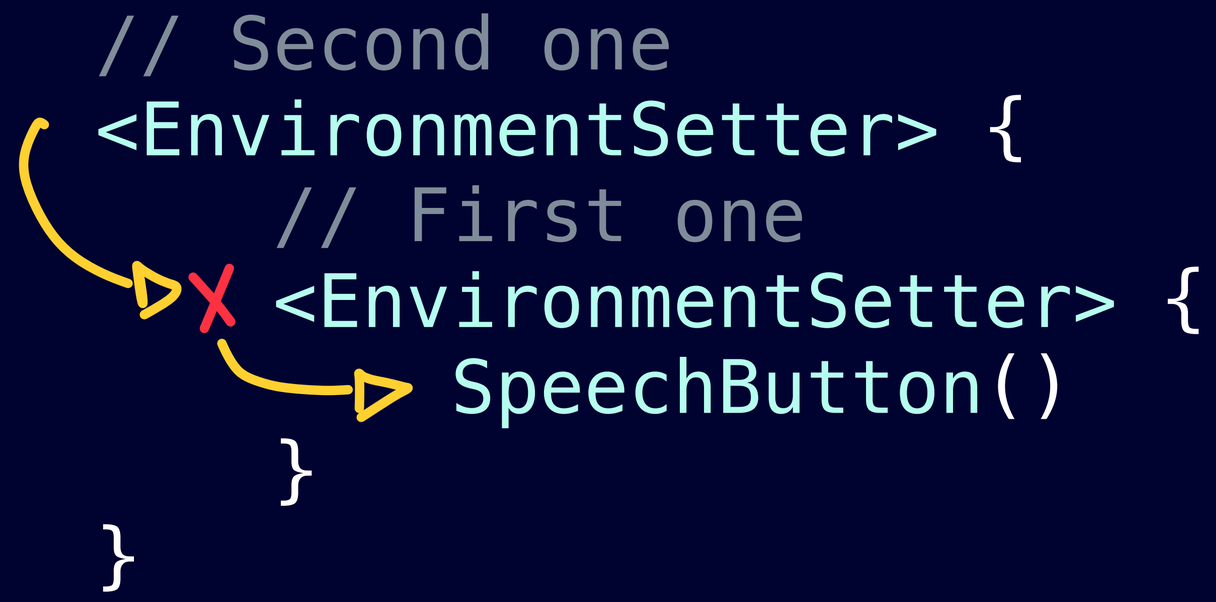
Callback modifier

Using Environment Values

```
// Second one
<EnvironmentSetter> {
    // First one
    <EnvironmentSetter> {
        SpeechButton()
    }
}
```

Callback modifier

Using Environment Values



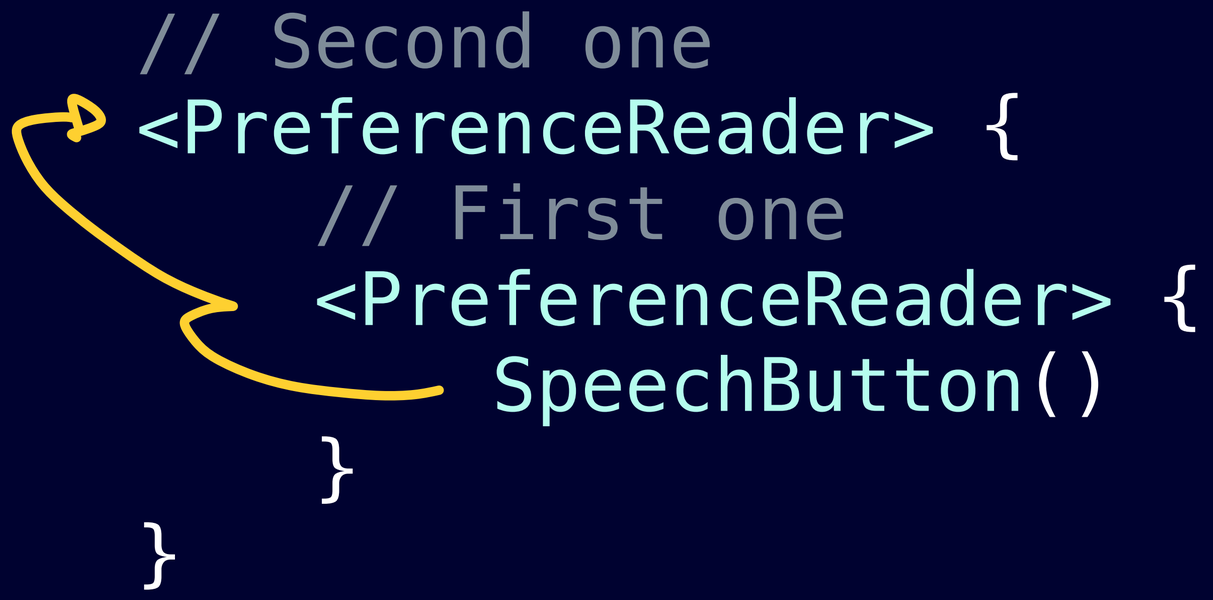
Callback modifier

Using Preference Values

```
// Second one
<PreferenceReader> {
    // First one
    <PreferenceReader> {
        SpeechButton()
    }
}
```

Callback modifier

Using Preference Values



Callback modifier

```
typealias StateChangedHandler = (SpeechRecognizerState) -> Void

extension View {
   func onStateChanged( ... ) -> some View {
      ...
   }
}
```

Callback modifier

```
extension View {
  func onStateChanged( ... ) -> some View {
    onPreferenceChange(LatestStatePreferenceKey.self) {
      if let value = $0 {
        handler(value)
      }
    }
  }
}

struct LatestStatePreferenceKey: PreferenceKey {
  ...
}
```

Callback modifier

```
extension View { ... }

struct LatestStatePreferenceKey: PreferenceKey {
   ...
}
```

Callback modifier

```
extension View { ... }

struct LatestStatePreferenceKey: PreferenceKey {
   static var defaultValue: SpeechRecognizerState? = nil
   static func reduce(
      value: inout SpeechRecognizerState?,
      nextValue: () -> SpeechRecognizerState?
   ) {
         // Prefer the most recent non-nil value
         value = nextValue() ?? value
   }
}
```

Callback modifier

```
extension View { ... }

struct LatestStatePreferenceKey: PreferenceKey { ... }
```

Callback modifier

```
extension View { ... }
struct LatestStatePreferenceKey: PreferenceKey { ... }
```

```
.preference(key: LatestStatePreferenceKey.self, value: state)
```

Callback modifier

```
SpeechButton(text: $text)
    .onStateChanged { print("First one", $0) }
    .onStateChanged { print("Second one", $0) }
```

Let's use it!

```
struct MyView: View {
    @State var text = ""

    var body: some View {
        SpeechButton(text: $text)
            .onStateChanged { print($0) }
    }
}
```

Let's use it!

```
struct MyView: View {
    @State var text = ""
    @State var state = SpeechRecognizerState.idle

    var body: some View {
        SpeechButton(text: $text)
            .onStateChanged { state = $0 }
    }
}
```

Let's use it!

```
struct MyView: View {
    @State var text = ""
    @State var state = SpeechRecognizerState.idle

    var body: some View {
        SpeechButton(text: $text)
            .onStateChanged { state = $0 }

        Text("The button is \(state)")
    }
}
```

We can do better!

```
struct MyView: View {
    @State var text = ""

    var body: some View {
        SpeechRecognitionReader { state in
            SpeechButton(text: $text)

            Text("The button is \(state)")
        }
    }
}
```

The provider

```
struct MyView: View {
    @State var text = ""

    var body: some View {
        SpeechRecognitionReader { state in
            SpeechButton(text: $text)

            Text("The button is \(state)")
        }
    }
}
```

The provider

```
struct SpeechRecognitionReader: View {
    var body: some View {
        ...
    }
}
```

The provider

```
struct SpeechRecognitionReader<Content: View>: View {
    let content = (SpeechRecognizerState) -> Content

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

The provider

```
struct SpeechRecognitionReader<Content: View>: View {
  let content = (SpeechRecognizerState) -> Content

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

  ...
}
```

The provider

```
struct SpeechRecognitionReader<Content: View>: View {
  let content = (SpeechRecognizerState) -> Content

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

  ...
}
```

The provider

```
struct SpeechRecognitionReader<Content: View>: View {
    let content = (SpeechRecognizerState) -> Content

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

The provider

```
struct SpeechRecognitionReader<Content: View>: View {
    @State var state = SpeechRecognizerState.idle
    let content = (SpeechRecognizerState) -> Content

    var body: some View {
        content(state)
    }
}
```

The provider

```
struct SpeechRecognitionReader<Content: View>: View {
    @State var state = SpeechRecognizerState.idle
    let content = (SpeechRecognizerState) -> Content

    var body: some View {
        content(state)
            .onStateChange { state = $0 }
    }
}
```

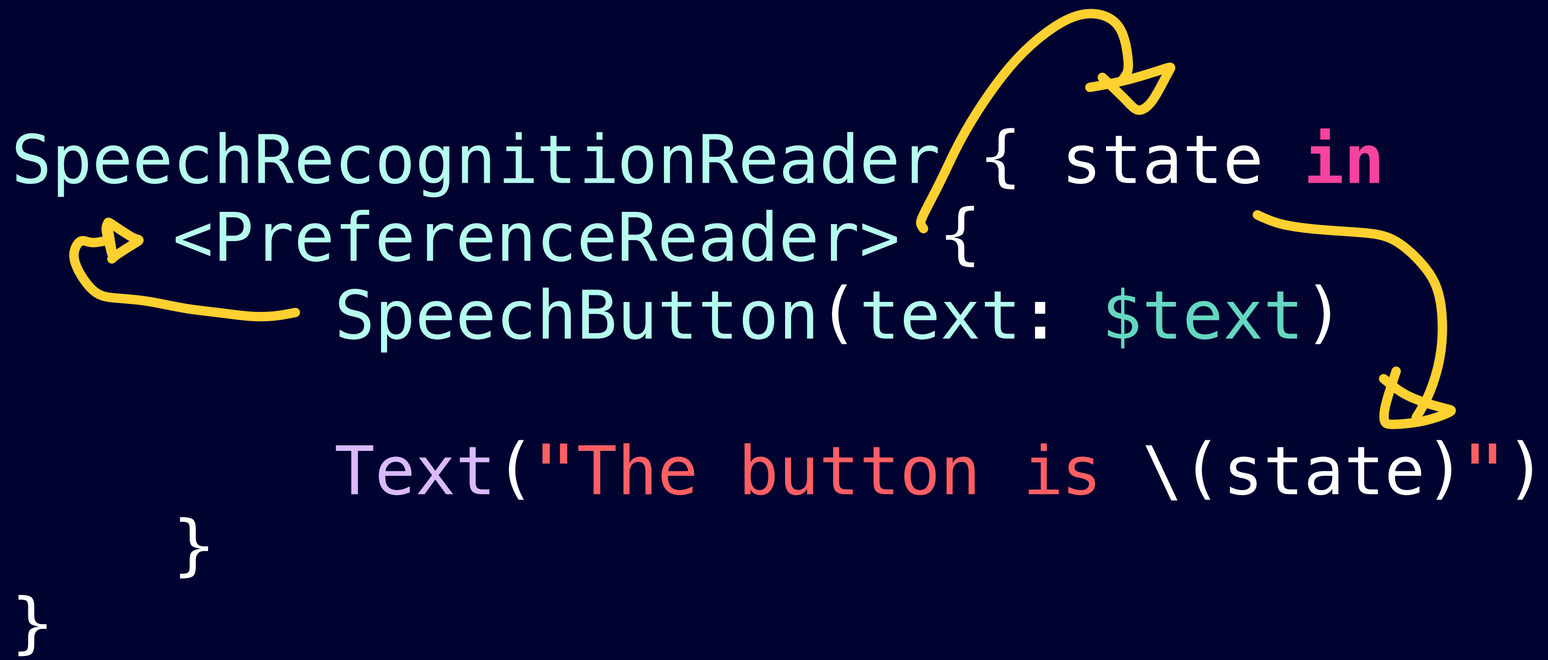
The provider

```
SpeechRecognitionReader { state in
    SpeechButton(text: $text)

    Text("The button is \(state)")
}
```

The provider

```
SpeechRecognitionReader { state in
    <PreferenceReader> {
        SpeechButton(text: $text)

        Text("The button is \(state)")
    }
}
```

The provider



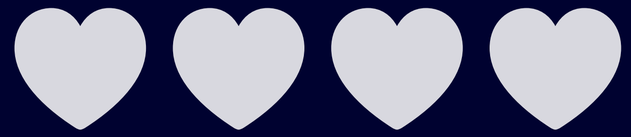
Style customization

```
RatingStars(rating: 0.8)
```

```
RatingDisplay(rating: 0.8)
```

```
RatingDisplay(rating: 0.8)
    .ratingStyle(.stars)
```

```
RatingDisplay(rating: 0.8)
    .ratingStyle(.hearts)
```



```
protocol RatingStyle {
    associatedtype Body: View

    func makeBody() -> Body
}
```

```
protocol RatingStyle {
    typealias Configuration = RatingStyleConfiguration
    associatedtype Body: View

    func makeBody(configuration: Configuration) -> Body
}

struct RatingStyleConfiguration {
    var rating: Double
    var symbolCount: Int
}
```

```
struct RatingDisplay: View {
    let rating: Double

    @Environment(\.ratingSymbolCount)
    private var symbolCount

    var body: some View {
        HStack(spacing: 4) {
            ...
        }
    }
}
```

```
struct StarsRatingStyle: RatingStyle {
    func makeBody(configuration: Configuration) -> some View {
        HStack(spacing: 4) {
            ...
        }
    }
}
```

```
struct StarsRatingStyle: RatingStyle {
    func makeBody(configuration: Configuration) -> some View {
        HStack(spacing: 4) {
            ...
        }
    }
}


extension RatingStyle where Self == StarsRatingStyle {
    static var stars: StarsRatingStyle {
        StarsRatingStyle()
    }
}
```

```
struct RatingDisplay: View {
    let rating: Double

    @Environment(\.ratingSymbolCount)
    private var symbolCount

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

```
struct RatingDisplay: View {
    let rating: Double

    @Environment(\.ratingSymbolCount)
    private var symbolCount

    var configuration: RatingStyleConfiguration {
        .init(rating: rating, symbolCount: symbolCount)
    }

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

```
struct RatingDisplay: View {
    let rating: Double

    @Environment(\.ratingSymbolCount)
    private var symbolCount

    var configuration: RatingStyleConfiguration { ... }

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

```
struct RatingDisplay<S: RatingStyle>: View {
    let rating: Double
    let style: S

    @Environment(\.ratingSymbolCount)
    private var symbolCount

    var configuration: RatingStyleConfiguration { ... }

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

```
struct RatingDisplay<S: RatingStyle>: View {
    let rating: Double
    let style: S

    @Environment(\.ratingSymbolCount)
    private var symbolCount

    var configuration: RatingStyleConfiguration { ... }

    var body: some View {
        style.makeBody(configuration: configuration)
    }
}
```

```
RatingDisplay(rating: 0.8, style: .stars)
```

```
struct RatingDisplay<S: RatingStyle>: View {
    ...
    init(rating: Double) where S == StarsRatingStyle {
        self.rating = rating
        self.style = .stars
    }
    init(rating: Double, style: S) {
        self.rating = rating
        self.style = style
    }
    ...
}
```

```
RatingDisplay(rating: 0.8)

RatingDisplay(rating: 0.8, style: .stars)
```

```
RatingDisplay(rating: 0.8)
    .ratingStyle(.stars)
```

```
RatingDisplay(rating: 0.8)
    .ratingStyle(.stars)
```

Type erasing ahead!

```
struct RatingDisplay<S: RatingStyle>: View {
    let rating: Double
    let style: S

    init(rating: Double) where S == StarsRatingStyle {
        self.rating = rating
        self.style = .stars
    }
    init(rating: Double, style: S) {
        self.rating = rating
        self.style = style
    }

    ...
}
```

```
struct RatingDisplay: View {
    let rating: Double

    @Environment(\.ratingStyle)
    private var style

    ...
}
```

```
struct RatingDisplay: View {
    let rating: Double

    @Environment(\.ratingStyle)
    private var style

    ...
}


extension EnvironmentValues {
    @Entry var ratingStyle: RatingStyle = .stars
}
```

```
extension EnvironmentValues {
    @Entry var ratingStyle: RatingStyle = .stars
}
```

```
extension EnvironmentValues {
    @Entry var ratingStyle: AnyRatingStyle = ...
}


struct AnyRatingStyle {

    ...

}
```

```
extension EnvironmentValues {
    @Entry var ratingStyle: AnyRatingStyle = ...
}


struct AnyRatingStyle: RatingStyle {

    func makeBody(configuration: Configuration) -> some View {
        ...
    }

}
```

```
extension EnvironmentValues {
    @Entry var ratingStyle: AnyRatingStyle = ...
}


struct AnyRatingStyle: RatingStyle {

    func makeBody(configuration: Configuration) -> AnyView {
        ...
    }

}
```

```swift
extension EnvironmentValues {
    @Entry var ratingStyle: AnyRatingStyle = AnyRatingStyle(.stars)
}


struct AnyRatingStyle: RatingStyle {

    init<S: RatingStyle>(_ style: S) {
        ...
    }

    func makeBody(configuration: Configuration) -> AnyView {
        ...
    }
}
```

```swift
extension EnvironmentValues {
    @Entry var ratingStyle: AnyRatingStyle = AnyRatingStyle(.stars)
}

struct AnyRatingStyle: RatingStyle { ... }

extension View {
    func ratingStyle<S: RatingStyle>(_ style: S) -> some View {
        environment(\.ratingStyle, AnyRatingStyle(style))
    }
}
```

```swift
extension EnvironmentValues {
    @Entry var ratingStyle: AnyRatingStyle = AnyRatingStyle(.stars)
}

struct AnyRatingStyle: RatingStyle { ... }

extension View {
    func ratingStyle<S: RatingStyle>(_ style: S) -> some View {
        environment(\.ratingStyle, AnyRatingStyle(style))
    }
}
```

```swift
@Environment(\.ratingStyle)
private var style
```

```
struct AnyRatingStyle: RatingStyle {

    init<S: RatingStyle>(_ style: S) {
        ...
    }

    func makeBody(configuration: Configuration) -> AnyView {
        ...
    }
}
```

```
struct AnyRatingStyle: RatingStyle {
    private var _makeBody: (Configuration) -> AnyView

    init<S: RatingStyle>(_ style: S) {
        ...
    }

    func makeBody(configuration: Configuration) -> AnyView {
        self._makeBody(configuration)
    }
}
```

```
struct AnyRatingStyle: RatingStyle {
    private var _makeBody: (Configuration) -> AnyView

    init<S: RatingStyle>(_ style: S) {
        self._makeBody = style.makeBodyTypeErased
    }

    func makeBody(configuration: Configuration) -> some View {
        self._makeBody(configuration)
    }
}


private extension RatingStyle {
    func makeBodyTypeErased(configuration: Configuration) -> AnyView {
        AnyView(self.makeBody(configuration: configuration))
    }
}
```

```
RatingDisplay(rating: 0.8)
    .ratingStyle(.stars)
```

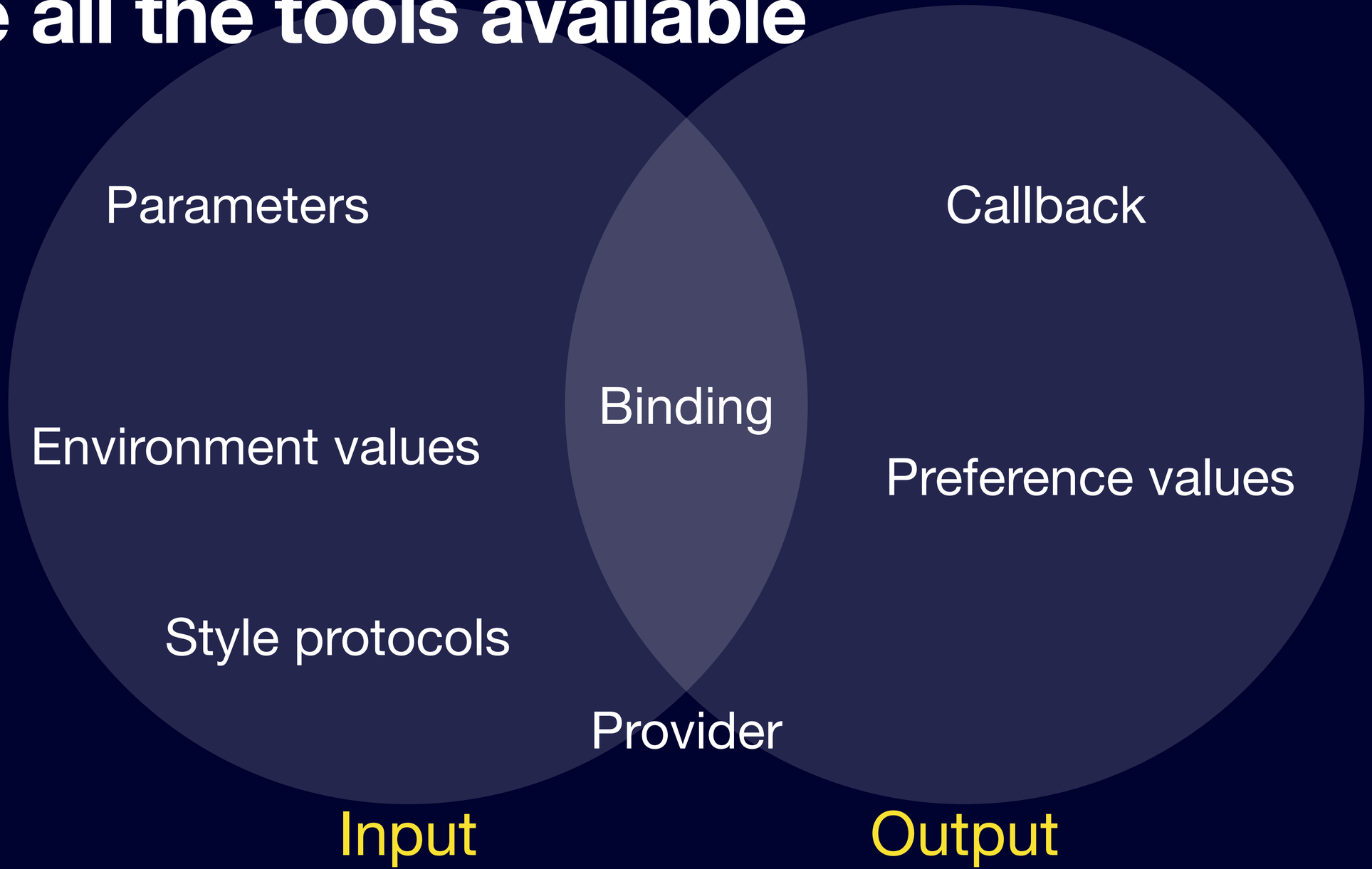
```
RatingDisplay(rating: 0.8)
    .ratingStyle(.stars)
```

 At the cost of type erasure

Examine the call sites of your views

Simplify the call sites of your views

Use all the tools available



Try to reproduce Apple SwiftUI patterns

Remember who you’re building for

A good API should be easy to use and hard to misuse.

It should be easy to do simple things; possible to do complex things; and impossible to do wrong things.

Thank you !

Slides & resources:

https://thomasdurand.fr/do-ios-2025

https://thomasdurand.fr

@deanatoire@mastodon.social

bsky.app/@thomasdurand.fr