

**“What I love most about
development is design”**

**“What I love most about
development is API design”**



**“What I love most about
development is API design”**

Crafting SwiftUI components.

The same way Apple does?

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](mailto: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](mailto:deanatoire@mastodon.social)

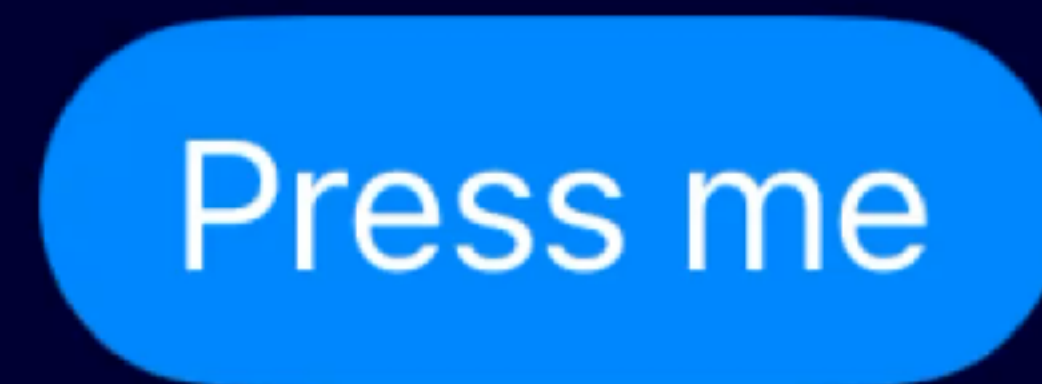
bsky.app/@thomasdurand.fr

ButtonKit

<https://github.com/Dean151/ButtonKit>

ButtonKit

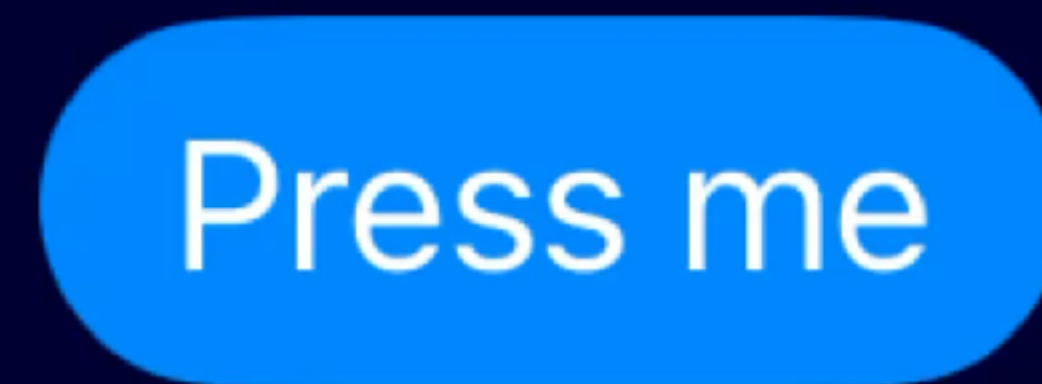
```
AsyncButton {  
    await ...  
} label: {  
    Text("Press me")  
}  
.buttonStyle(.borderedProminent)
```



<https://github.com/Dean151/ButtonKit>

ButtonKit

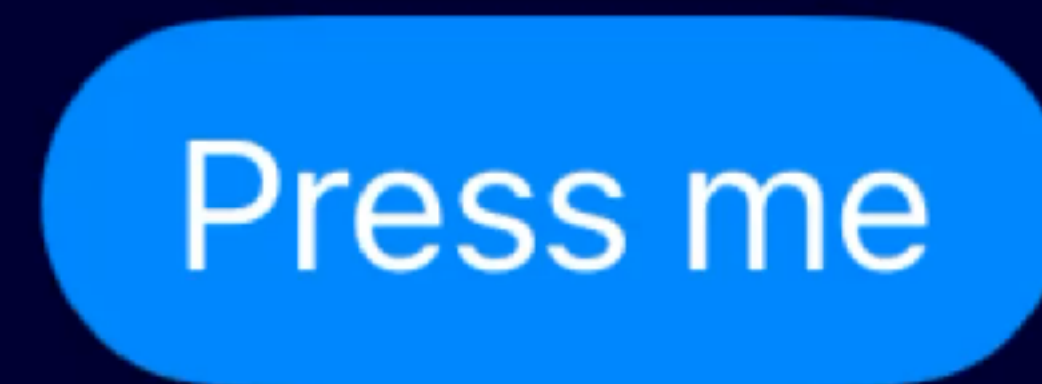
```
AsyncButton {  
    await ...  
} label: {  
    Text("Press me")  
}  
.buttonStyle(.borderedProminent)
```



<https://github.com/Dean151/ButtonKit>

ButtonKit

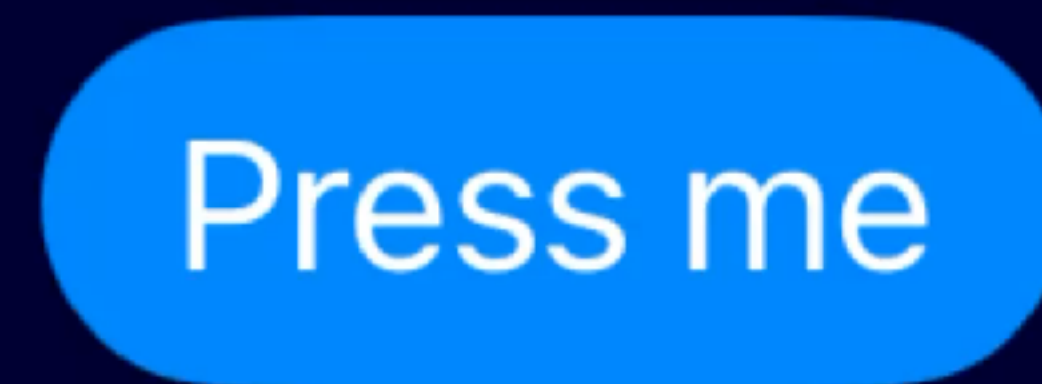
```
AsyncButton {  
    try await ...  
} label: {  
    Text("Press me")  
}  
.buttonStyle(.borderedProminent)
```



<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)
```

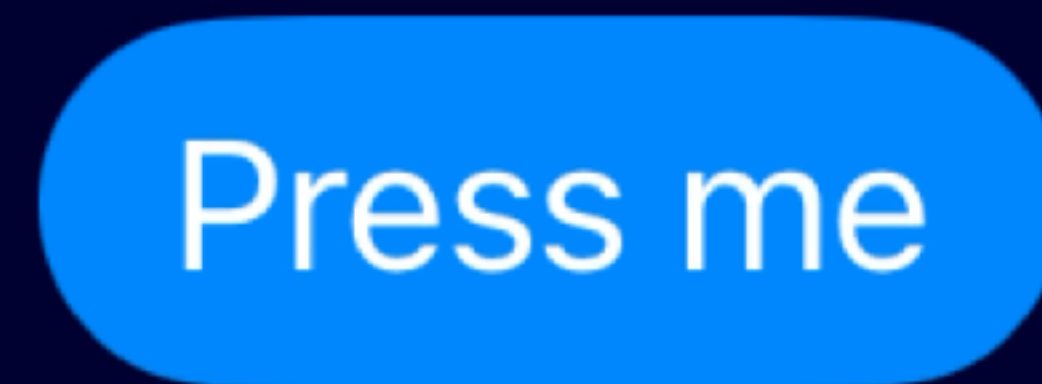


Press me

<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”

Two advices

Focus on the call site

Focus on the call site
Focus on input and outputs

RatingStars (rating: 1.0)



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**

**How to get data
out of your view?**

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

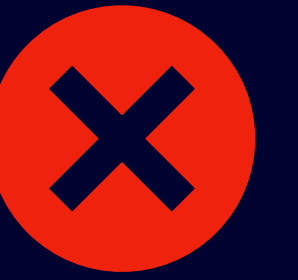
```
RatingStars(rating: $rating)  
    .controlSize(.large)  
    .disabled(true)
```

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

Read-only output?

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

Read-only output?



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

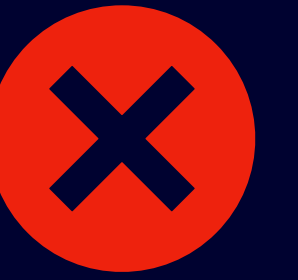
Read-only output

Callback argument

```
RatingStars(ratingChanged: { rating in  
    ...  
})
```

Read-only output

Callback argument



```
RatingStars { rating in  
    ...  
}
```

Read-only output

Callback modifier 🤔



```
RatingStars()  
  .onRatingChanged { rating in  
    ...  
  }
```

Callback modifier

```
 typealias  RatingChangedHandler = (Double) -> Void
```

Callback modifier

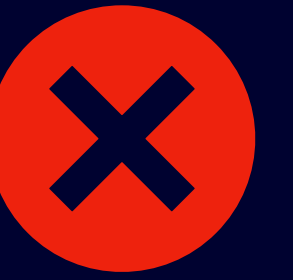
```
 typealias RatingChangedHandler = (Double) -> Void
```

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

Callback modifier

```
 typealias RatingChangedHandler = (Double) -> Void  
  
extension View {  
    func onRatingChanged( ... ) -> some View {  
    }  
}
```

Callback modifier

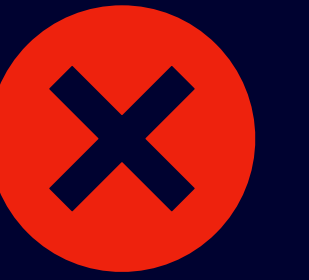


```
typealias RatingChangedHandler = (Double) -> Void
```

```
extension View {  
    func onRatingChanged( ... ) -> some View {  
        environment(\.ratingChanged, handler)  
    }  
}
```

```
extension EnvironmentValues {  
    @Entry var ratingChanged: RatingChangedHandler? = nil  
}
```

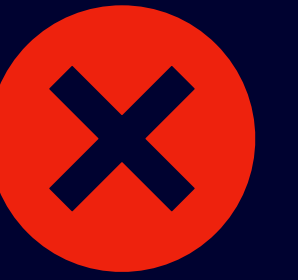
Callback modifier



```
RatingControl()  
  .onRatingChanged { print("First one", $0) }  
  .onRatingChanged { print("Second one", $0) }
```

Callback modifier

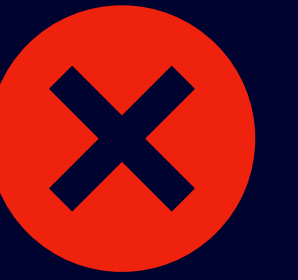
Using Environment Values



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

Callback modifier

Using Environment Values



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

The diagram shows a code snippet with annotations. A yellow arrow points from the opening curly brace of the first `<EnvironmentSetter>` block to a red 'X' mark placed on the opening curly brace of the second `<EnvironmentSetter>` block. Another yellow arrow points from the red 'X' to the `RatingControl()` line. A third yellow arrow points from the closing curly brace of the first `<EnvironmentSetter>` block to the closing curly brace of the second `<EnvironmentSetter>` block.

Callback modifier

Using Preference Values



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

Callback modifier

Using Preference Values



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

A yellow arrow originates from the inner <PreferenceReader> block and points to the outer <PreferenceReader> block, illustrating the relationship between the two nested preference readers.

Callback modifier

```
 typealias RatingChangedHandler = (Double) -> Void
```

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

Callback modifier

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

```
struct LatestRatingPreferenceKey: PreferenceKey {  
    ...  
}
```

Callback modifier

```
extension View { ... }
```

```
struct LatestRatingPreferenceKey: PreferenceKey {  
    ...  
}
```

Callback modifier

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

Callback modifier

```
extension View { ... }
```

```
struct LatestRatingPreferenceKey: PreferenceKey { ... }
```

Callback modifier

```
extension View { ... }
```

```
struct LatestRatingPreferenceKey: PreferenceKey { ... }
```

```
.preference(key: LatestRatingPreferenceKey.self, value: rating)
```

Callback modifier



```
RatingControl()  
  .onRatingChanged { print("First one", $0) }  
  .onRatingChanged { print("Second one", $0) }
```

Let's use it!

```
struct MyView: View {  
    var body: some View {  
        RatingStars()  
        .onRatingChanged { print($0) }  
    }  
}
```

Let's use it!

```
struct MyView: View {  
    @State var rating = 0.5  
  
    var body: some View {  
        RatingStars()  
        .onRatingChanged { rating = $0 }  
    }  
}
```

Let's use it!

```
struct MyView: View {
  @State var rating = 0.5

  var body: some View {
    RatingStars()
      .onRatingChanged { rating = $0 }

    Text("This talk is \((Int(rating * 100))% fun")
  }
}
```

We can do better!

```
struct MyView: View {
  var body: some View {
    RatingReader { rating in
      RatingStars()

      Text("This talk is \((Int(rating * 100))\)% fun")
    }
  }
}
```

The provider



```
struct MyView: View {
  var body: some View {
    RatingReader { rating in
      RatingStars()

      Text("This talk is \ (Int(rating * 100))% fun")
    }
  }
}
```

The provider

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

The provider

```
struct RatingReader<Content: View>: View {  
  let content = (Double) -> Content  
  var body: some View {  
    ...  
  }  
}
```

The provider

```
struct RatingReader<Content: View>: View {  
  @State var rating = 1.0  
  let content = (Double) -> Content  
  
  var body: some View {  
    content(rating)  
  }  
}
```

The provider

```
struct RatingReader<Content: View>: View {  
  @State var rating = 1.0  
  let content = (Double) -> Content  
  
  var body: some View {  
    content(rating)  
    .onRatingChange { rating = $0 }  
  }  
}
```

The provider

```
RatingReader { rating in  
    RatingStars()  
  
    Text("This talk is \ (Int(rating * 100))% fun"  
}
```

The provider

```
RatingReader { rating in  
    <PreferenceReader> {  
        RatingStars()  
  
        Text("This talk is \ (Int(rating * 100))% fun"  
    }  
}
```

The provider

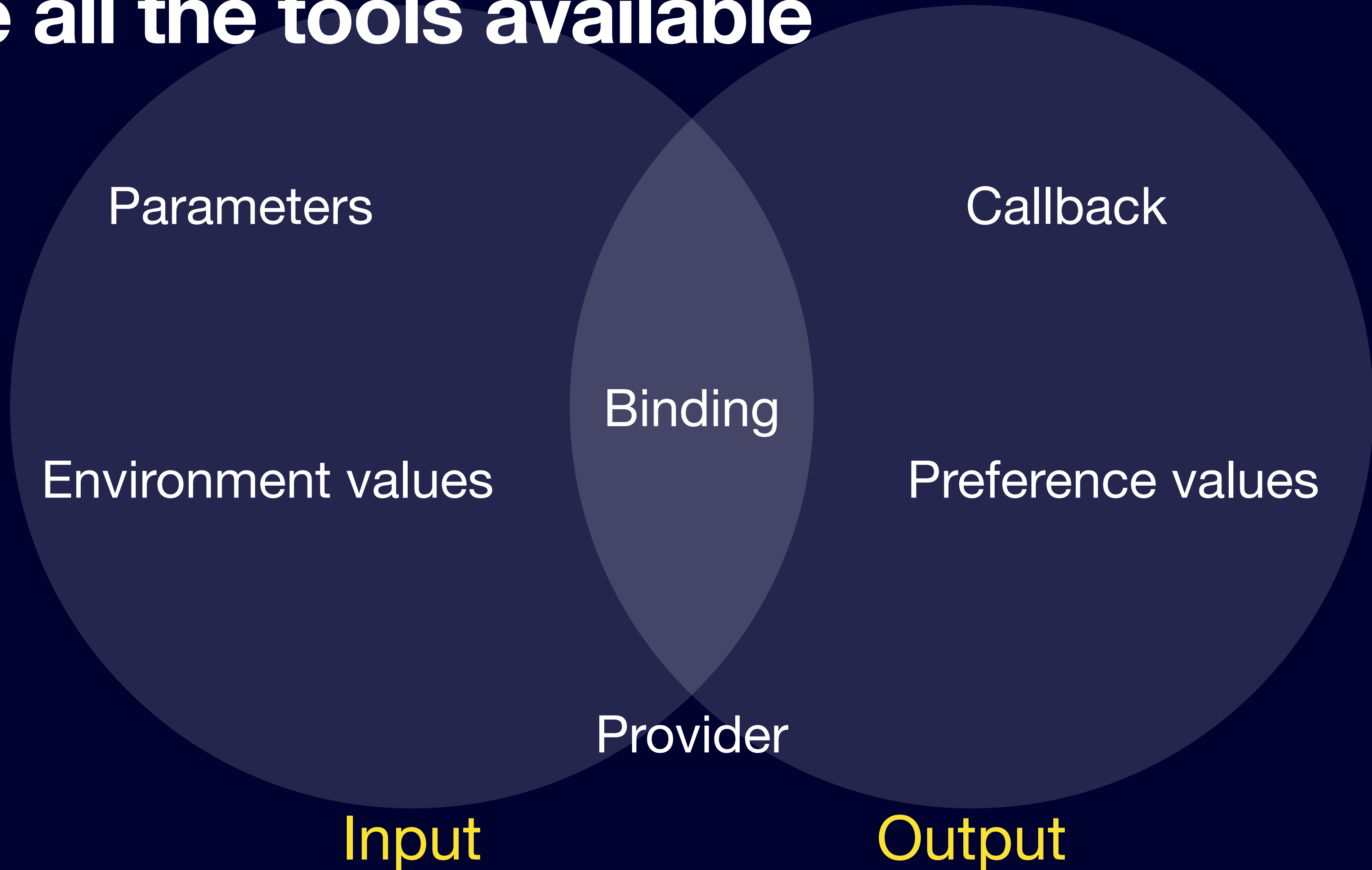
```
RatingReader { rating in  
  <PreferenceReader> {  
    RatingStars()  
  
    Text("This talk is \ (Int(rating * 100))% fun"  
  }  
}
```

The diagram illustrates the relationships between the code elements. A yellow arrow points from the `RatingReader` name to the `in` keyword, indicating that `RatingReader` is a function of `rating`. Another yellow arrow points from the `RatingStars()` call to the `rating` parameter, showing that `RatingStars` depends on `rating`. A third yellow arrow points from the `RatingStars()` call to the `Text` function call, indicating that `RatingStars` provides data to `Text`. A fourth yellow arrow points from the `RatingStars()` call to the `PreferenceReader` block, showing that `RatingStars` is implemented by `PreferenceReader`.

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

Merci !

Slides & resources:

<https://thomasdurand.fr/swift-connection-2025>

<https://thomasdurand.fr>
@deanatoire@mastodon.social
bsky.app/@thomasdurand.fr