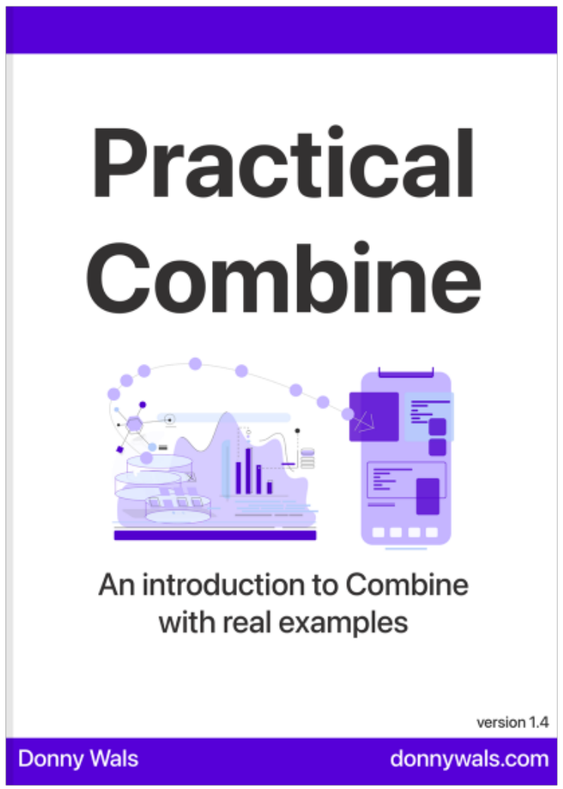
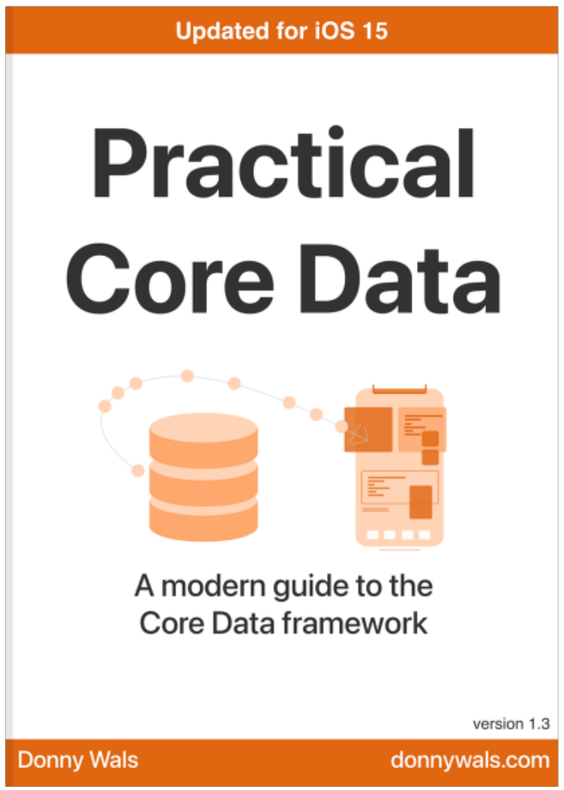
👋🏼 Hi, I'm Donny



https://practicalcombine.com/

donnywals.com



https://practicalcoredata.com/

Building custom SwiftUI property wrappers

What you’ll learn in this talk

What you’ll learn in this talk

- What property wrappers are, and how they work

What you’ll learn in this talk

- What property wrappers are, and how they work
- Why a regular property wrapper isn’t good enough for SwiftUI

What you’ll learn in this talk

- What property wrappers are, and how they work
- Why a regular property wrapper isn’t good enough for SwiftUI
- How to build a property wrapper that taps into SwiftUI’s View environment and lifecycle

What you’ll learn in this talk

- What property wrappers are, and how they work
- Why a regular property wrapper isn’t good enough for SwiftUI
- How to build a property wrapper that taps into SwiftUI’s View environment and lifecycle
- How you can test your custom property wrapper, and why it’s tricky

I’m sure code like this looks familiar

```
struct MyView: View {
    @State var isActive = false

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

I’m sure code like this looks familiar

```
struct MyView: View {
    @State var isActive = false

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

What does @State do?

What does @State do?

- It’s a container that holds a mutable value

What does @State do?

- It’s a container that holds a mutable value
- Whenever you mutate the contained value, the view updates

What does @State do?

- It’s a container that holds a mutable value
- Whenever you mutate the contained value, the view updates
- When your view’s initializer is called, your state isn’t reset

What does @State do?

- It’s a container that holds a mutable value
- Whenever you mutate the contained value, the view updates
- When your view’s initializer is called, your state isn’t reset
- You can obtain a binding to a state property

Or maybe you've seen this before

```
struct MyView: View {
    @FetchRequest(fetchRequest: Post.all) var posts

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

Or maybe you've seen this before

```
struct MyView: View {
    @FetchRequest(fetchRequest: Post.all) var posts

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

What does @FetchRequest do?

What does @FetchRequest do?

- Fetch data based on a Core Data entity or fetch request

What does @FetchRequest do?

- Fetch data based on a Core Data entity or fetch request
- Tell SwiftUI to redraw when the fetched data has changed

What does @FetchRequest do?

- Fetch data based on a Core Data entity or fetch request
- Tell SwiftUI to redraw when the fetched data has changed
- Leverages the SwiftUI environment for its dependencies

SwiftUI comes with many, many property wrappers...

If we follow what appear to be Apple's vision...

If we follow what appear to be Apple's vision...

- SwiftUI views own, or are passed state directly

If we follow what appear to be Apple's vision...

- SwiftUI views own, or are passed state directly
- SwiftUI views can leverage property wrappers to obtain data

If we follow what appear to be Apple's vision...

- SwiftUI views own, or are passed state directly
- SwiftUI views can leverage property wrappers to obtain data
- We *technically* don't need extra layers to fetch data

So wouldn't it be neat if we could write this?

```
struct PostsView: View {
    @RemoteData(endpoint: .feed) var feed: [Post]

    var body: some View {
        List(feed) { post in
            Text(post.title)
        }
    }
}
```

So wouldn't it be neat if we could write this?

```
struct PostsView: View {
    @RemoteData(endpoint: .feed) var feed: [Post]

    var body: some View {
        List(feed) { post in
            Text(post.title)
        }
    }
}
```

Let’s go and build it!

Quick property wrapper introduction

```
@propertyWrapper
class StringSample {
    let wrappedValue: String

    var projectedValue: String {
        "Projected: \(wrappedValue)"
    }
}
```

Quick property wrapper introduction

```
@propertyWrapper
class StringSample {
    let wrappedValue: String

    var projectedValue: String {
        "Projected: \(wrappedValue)"
    }
}
```

Quick property wrapper introduction

```
@propertyWrapper
class StringSample {
    let wrappedValue: String

    var projectedValue: String {
        "Projected: \(wrappedValue)"
    }
}
```

Quick property wrapper introduction

```
@propertyWrapper
class StringSample {
    let wrappedValue: String

    var projectedValue: String {
        "Projected: \(wrappedValue)"
    }
}
```

Example usage

```
struct SampleUsage {
    @StringSample var example = "Hello, world"

    func printSample() {
        print(example) // wrapped value: "Hello, world"
        print(_example) // property wrapper class: StringSample
        print($example) // projected value: "Projected: Hello, world"
    }
}
```

Example usage

```
struct SampleUsage {
    @StringSample var example = "Hello, world"

    func printSample() {
        print(example) // wrapped value: "Hello, world"
        print(_example) // property wrapper class: StringSample
        print($example) // projected value: "Projected: Hello, world"
    }
}
```

Example usage

```
struct SampleUsage {
    @StringSample var example = "Hello, world"

    func printSample() {
        print(example) // wrapped value: "Hello, world"
        print(_example) // property wrapper class: StringSample
        print($example) // projected value: "Projected: Hello, world"
    }
}
```

Reminder: this is our goal

```
struct PostsView: View {
    @RemoteData(endpoint: .feed) var feed: [Post]

    var body: some View {
        List(feed) { post in
            Text(post.title)
        }
    }
}
```

Let’s start with the basics

```
@propertyWrapper
class RemoteData {
    private let endpoint: Endpoint

    var wrappedValue: [Post] = []

    init(endpoint: Endpoint) {
        self.endpoint = endpoint
    }
}
```

Let’s start with the basics

```
@propertyWrapper
class RemoteData {
    private let endpoint: Endpoint

    var wrappedValue: [Post] = []

    init(endpoint: Endpoint) {
        self.endpoint = endpoint
    }
}
```

And let's add a simple fetch function

```
func fetchData() {
    let urlSession = URLSession.shared
    let url = URL.for(endpoint)
    urlSession.dataTask(with: url) { data, response, error in
        guard let data = data else { return }

        self.wrappedValue = try! JSONDecoder()
            .decode([Post].self, from: data)
    }.resume()
}
```

Uhmm... Donny...

And let's add a simple fetch function

```
func fetchData() {
    let urlSession = URLSession.shared
    let url = URL.for(endpoint)
    urlSession.dataTask(with: url) { data, response, error in
        guard let data = data else { return }

        self.wrappedValue = try! JSONDecoder().decode([Post].self, from: data)
    }.resume()
}
```

We'll get to that, I promise

And let's add a simple fetch function

```
func fetchData() {
    let urlSession = URLSession.shared
    let url = URL.for(endpoint)
    urlSession.dataTask(with: url) { data, response, error in
        guard let data = data else { return }

        self.wrappedValue = try! JSONDecoder().decode([Post].self, from: data)
    }.resume()
}
```

URL.for? Endpoint?

```
extension RemoteData {
    enum Endpoint {
        case feed
    }
}

extension URL {
    static func for(_ endpoint: RemoteData.Endpoint) -> URL {
        return URL(string: "https://donnywals.com")!
    }
}
```

About this...

```
func fetchData() {
    let urlSession = URLSession.shared
    let url = URL.for(endpoint)
    urlSession.dataTask(with: url) { data, response, error in
        guard let data = data else { return }

        self.wrappedValue = try! JSONDecoder().decode([Post].self, from: data)
    }.resume()
}
```

Where to grab our networking layer from?

💡

How about SwiftUI’s environment?

Introducing DynamicProperty

Introducing DynamicProperty

- DynamicProperty is a protocol with just one requirement: update()

Introducing DynamicProperty

- DynamicProperty is a protocol with just one requirement: update()
- SwiftUI will make environment values available to your property wrapper

Introducing DynamicProperty

- DynamicProperty is a protocol with just one requirement: update()
- SwiftUI will make environment values available to your property wrapper
- Should not internally store state

Introducing DynamicProperty

- DynamicProperty is a protocol with just one requirement: update()
- SwiftUI will make environment values available to your property wrapper
- Should not internally store state
- Can hold observable objects to tell SwiftUI something changed

First, we add a new environment key

```
private struct URLSessionKey: EnvironmentKey {
    static let defaultValue: URLSession = {
        fatalError("Attempt to read URLSession but it's not injected")
    }()
}

extension EnvironmentValues {
    var urlSession: URLSession {
        get { self[URLSessionKey.self] }
        set { self[URLSessionKey.self] = newValue }
    }
}
```

First, we add a new environment key

```
private struct URLSessionKey: EnvironmentKey {
    static let defaultValue: URLSession = {
        fatalError("Attempt to read URLSession but it's not injected")
    }()
}

extension EnvironmentValues {
    var urlSession: URLSession {
        get { self[URLSessionKey.self] }
        set { self[URLSessionKey.self] = newValue }
    }
}
```

And then we update the property wrapper

```
@propertyWrapper
class RemoteData: DynamicProperty {
    // ...
    @Environment(\.urlSession) var urlSession

    func update() {
        fetchData()
    }

    func fetchData() {
        // ...
    }
}
```

And then we update the property wrapper

```
@propertyWrapper
class RemoteData: DynamicProperty {
    // ...
    @Environment(\.urlSession) var urlSession

    func update() {
        fetchData()
    }

    func fetchData() {
        // ...
    }
}
```

And then we update the property wrapper

```
@propertyWrapper
class RemoteData: DynamicProperty {
    // ...
    @Environment(\.urlSession) var urlSession

    func update() {
        fetchData()
    }

    func fetchData() {
        // ...
    }
}
```

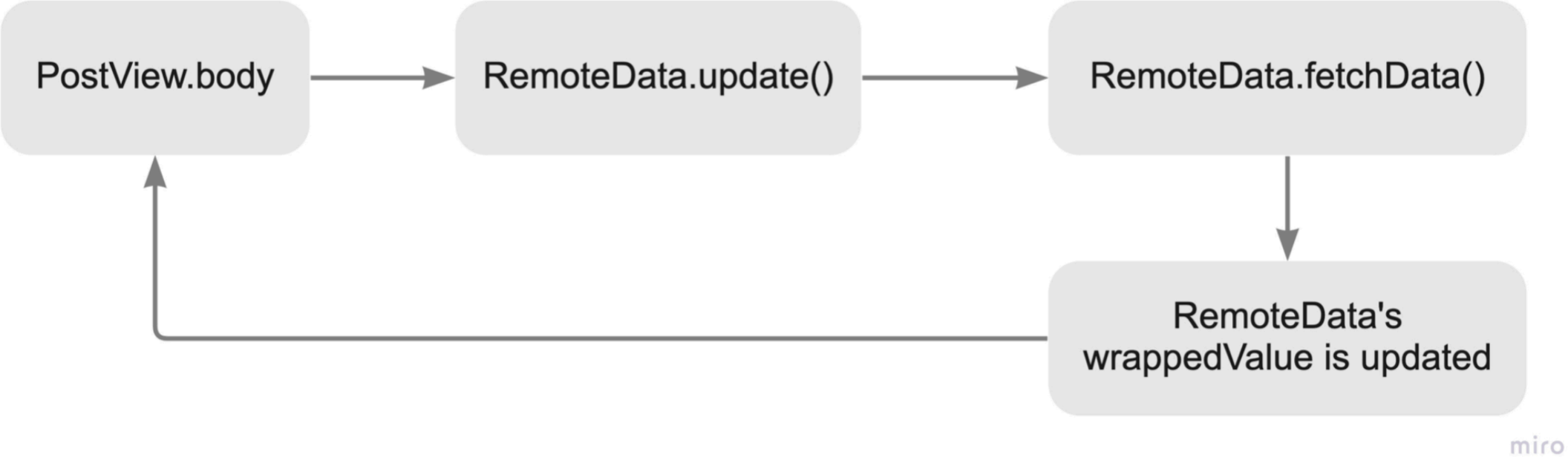
Let's configure the environment

```
@main
struct SwiftUIPropertyWrapperTalkApp: App {
    var body: some Scene {
        WindowGroup {
            PostsView()
                .environment(\.urlSession, URLSession.shared)
        }
    }
}
```

And we'll create a simple view

```
struct PostsView: View {
    @RemoteData(endpoint: .feed) var feed: [Post]

    var body: some View {
        List(feed) { post in
            Text(post.title)
        }
    }
}
```

Here’s what should happen



Run the app and…

What could be wrong?…

```
@propertyWrapper
class RemoteData: DynamicProperty {
    /* ... */

    var wrappedValue: [Post] = []
    @Environment(\.urlSession) var urlSession

    func update() {
        fetchData()
    }

    func fetchData() {
        let url = URL.for(endpoint)
        urlSession.dataTask(with: url) { data, response, error in
            guard let data = data else {
                /* ... */
                return
            }

            self.wrappedValue = try! JSONDecoder()
                .decode([Post].self, from: data)
        }.resume()
    }
}
```

```swift
@propertyWrapper
class RemoteData: DynamicProperty {
    /* ... */

    var wrappedValue: [Post] = []
    @Environment(\.urlSession) var urlSession

    func update() {
        fetchData()
    }

    func fetchData() {
        let url = URL.for(endpoint)
        urlSession.dataTask(with: url) { data, response, error in
            guard let data = data else {
                /* ... */
                return
            }

            self.wrappedValue = try! JSONDecoder()
                .decode([Post].self, from: data)
        }.resume()
    }
}
```

```swift
@propertyWrapper
class RemoteData: DynamicProperty {
    /* ... */

    var wrappedValue: [Post] = []
    @Environment(\.urlSession) var urlSession

    func update() {
        fetchData()
    }

    func fetchData() {
        let url = URL.for(endpoint)
        urlSession.dataTask(with: url) { data, response, error in
            guard let data = data else {
                /* ... */
                return
            }

            self.wrappedValue = try! JSONDecoder()
                .decode([Post].self, from: data)
        }.resume()
    }
}
```

```swift
@propertyWrapper
class RemoteData: DynamicProperty {
    /* ... */

    var wrappedValue: [Post] = []
    @Environment(\.urlSession) var urlSession

    func update() {
        fetchData()
    }

    func fetchData() {
        let url = URL.for(endpoint)
        urlSession.dataTask(with: url) { data, response, error in
            guard let data = data else {
                /* ... */
                return
            }

            self.wrappedValue = try! JSONDecoder()
                .decode([Post].self, from: data)
        }.resume()
    }
}
```

```
@propertyWrapper
class RemoteData: DynamicProperty {
    /* ... */

    var wrappedValue: [Post] = []
    @Environment(\.urlSession) var urlSession

    func update() {
        fetchData()
    }

    func fetchData() {
        let url = URL.for(endpoint)
        urlSession.dataTask(with: url) { data, response, error in
            guard let data = data else {
                /* ... */
                return
            }

            self.wrappedValue = try! JSONDecoder()
                .decode([Post].self, from: data)
        }.resume()
    }
}
```

We should tell SwiftUI new data was fetched…

We can use @State in our property wrapper

```
@propertyWrapper
class RemoteData: DynamicProperty {

    @State var wrappedValue: [Post] = []
    @Environment(\.urlSession) var urlSession

    /* ... */
}
```

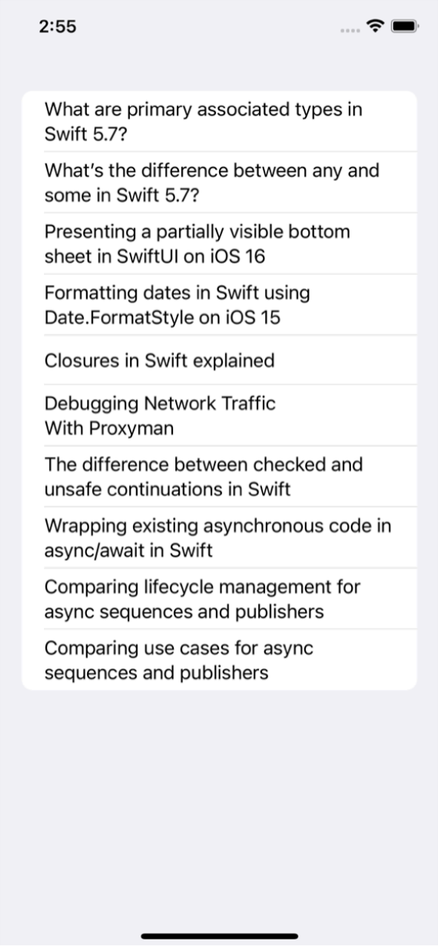
Run the app again…

There’s a hidden requirement for DynamicProperty...

```swift
@propertyWrapper
struct RemoteData: DynamicProperty {
    @State var wrappedValue: [Post] = []
    @Environment(\.urlSession) var urlSession

    func update() {
        fetchData()
    }

    func fetchData() {
        // unchanged...
    }
}
```

```swift
@propertyWrapper
struct RemoteData: DynamicProperty {
    @State var wrappedValue: [Post] = []
    @Environment(\.urlSession) var urlSession

    func update() {
        fetchData()
    }

    func fetchData() {
        // unchanged...
    }
}
```

One more try...


2:55
What are primary associated types in Swift 5.7?
What's the difference between any and some in Swift 5.7?
Presenting a partially visible bottom sheet in SwiftUI on iOS 16
Formatting dates in Swift using Date.FormatStyle on iOS 15
Closures in Swift explained
Debugging Network Traffic With Proxyman
The difference between checked and unsafe continuations in Swift
Wrapping existing asynchronous code in async/await in Swift
Comparing lifecycle management for async sequences and publishers
Comparing use cases for async sequences and publishers

So what do we have so far?

So what do we have so far?

- A struct called RemoteData that conforms to DynamicProperty

So what do we have so far?

- A struct called RemoteData that conforms to DynamicProperty
- It grabs a URLSession from the SwiftUI environment

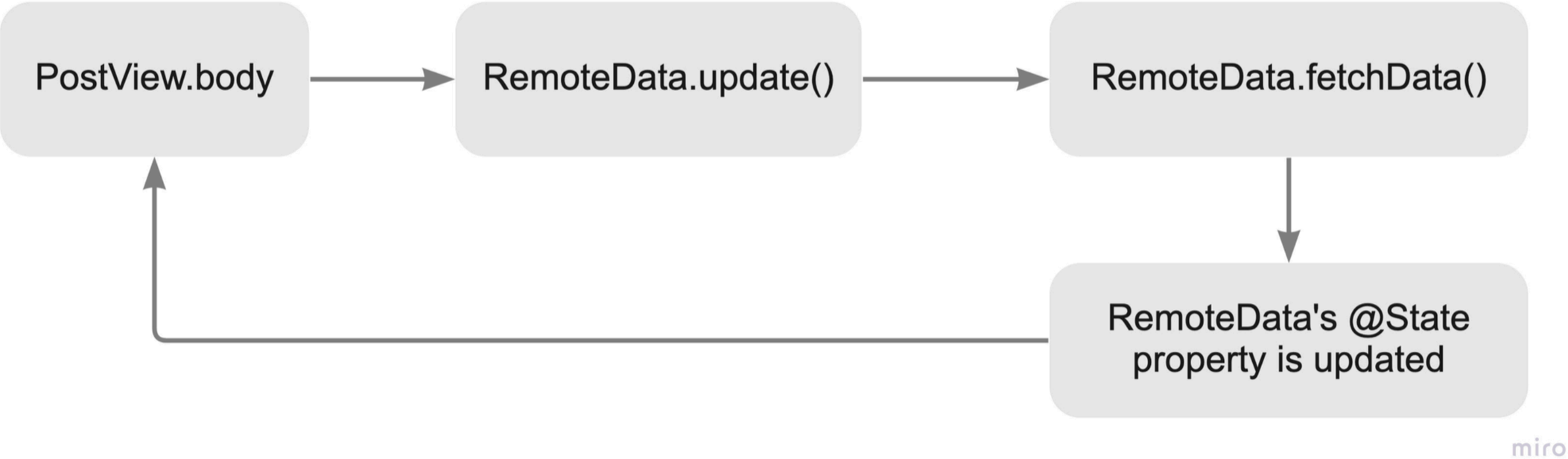
So what do we have so far?

- A struct called RemoteData that conforms to DynamicProperty
- It grabs a URLSession from the SwiftUI environment
- It uses @State on its wrappedValue to tell SwiftUI something changed

So what do we have so far?

- A struct called RemoteData that conforms to DynamicProperty
- It grabs a URLSession from the SwiftUI environment
- It uses @State on its wrappedValue to tell SwiftUI something changed
- Whenever SwiftUI calls update() we fetch data

Here’s what happens now



⚠️ We fetch data on every call to update()

We could try something like this

```
@propertyWrapper
struct RemoteData: DynamicProperty {
    private let endpoint: Endpoint

    @State var wrappedValue: [Post] = []
    var isLoading = false

    /* ... */

    func fetchData() {
        guard isLoading == false, wrappedValue.isEmpty else {
            return
        }

        isLoading = true
        let url = URL.for(endpoint)
        urlSession.dataTask(with: url) { data, response, error in
            guard let data = data else {
                return
            }

            self.isLoading = false
            self.wrappedValue = try! JSONDecoder().decode([Post].self, from: data)
        }.resume()
    }
}
```

We could try something like this

```
@propertyWrapper
struct RemoteData: DynamicProperty {
    private let endpoint: Endpoint

    @State var wrappedValue: [Post] = []
    var isLoading = false

    /* ... */

    func fetchData() {
        guard isLoading == false, wrappedValue.isEmpty else {
            return
        }

        isLoading = true
        let url = URL.for(endpoint)
        urlSession.dataTask(with: url) { data, response, error in
            guard let data = data else {
                return
            }

            self.isLoading = false
            self.wrappedValue = try! JSONDecoder().decode([Post].self, from: data)
        }.resume()
    }
}
```

We could try something like this

```
@propertyWrapper
struct RemoteData: DynamicProperty {
    private let endpoint: Endpoint

    @State var wrappedValue: [Post] = []
    var isLoading = false

    /* ... */

    func fetchData() {
        guard isLoading == false, wrappedValue.isEmpty else {
            return
        }

        isLoading = true
        let url = URL.for(endpoint)
        urlSession.dataTask(with: url) { data, response, error in
            guard let data = data else {
                return
            }

            self.isLoading = false
            self.wrappedValue = try! JSONDecoder().decode([Post].self, from: data)
        }.resume()
    }
}
```

We could try something like this

```
@propertyWrapper
struct RemoteData: DynamicProperty {
    private let endpoint: Endpoint

    @State var wrappedValue: [Post] = []
    var isLoading = false

    /* ... */

    func fetchData() {
        guard isLoading == false, wrappedValue.isEmpty else {
            return
        }

        isLoading = true
        let url = URL.for(endpoint)
        urlSession.dataTask(with: url) { data, response, error in
            guard let data = data else {
                return
            }

            self.isLoading = false
            self.wrappedValue = try! JSONDecoder().decode([Post].self, from: data)
        }.resume()
    }
}
```

We could try something like this

```
@propertyWrapper
struct RemoteData: DynamicProperty {
    private let endpoint: Endpoint

    @State var wrappedValue: [Post] = []
    var isLoading = false

    /* ... */

    func fetchData() {
        guard isLoading == false, wrappedValue.isEmpty else {
            return
        }

        isLoading = true // cannot assign to property: `self` is immutable
        let url = URL.for(endpoint)
        urlSession.dataTask(with: url) { data, response, error in
            guard let data = data else {
                return
            }

            self.isLoading = false // cannot assign to property: `self` is immutable
            self.wrappedValue = try! JSONDecoder().decode([Post].self, from: data)
        }.resume()
    }
}
```

We could add some @State

```
@propertyWrapper
struct RemoteData: DynamicProperty {
    private let endpoint: Endpoint

    @State var wrappedValue: [Post] = []
    @State var isLoading = false

    /* ... */

    func fetchData() {
        guard isLoading == false, wrappedValue.isEmpty else {
            return
        }

        isLoading = true // modifying state during view update, this will cause undefined behavior
        let url = URL.for(endpoint)
        urlSession.dataTask(with: url) { data, response, error in
            guard let data = data else {
                return
            }

            self.isLoading = false
            self.wrappedValue = try! JSONDecoder().decode([Post].self, from: data)
        }.resume()
    }
}
```

We could add some @State

```
@propertyWrapper
struct RemoteData: DynamicProperty {
    private let endpoint: Endpoint

    @State var wrappedValue: [Post] = []
    @State var isLoading = false

    /* ... */

    func fetchData() {
        guard isLoading == false, wrappedValue.isEmpty else {
            return
        }

        isLoading = true // modifying state during view update, this will cause undefined behavior
        let url = URL.for(endpoint)
        urlSession.dataTask(with: url) { data, response, error in
            guard let data = data else {
                return
            }

            self.isLoading = false
            self.wrappedValue = try! JSONDecoder().decode([Post].self, from: data)
        }.resume()
    }
}
```

We should seperate the underlying data from the wrapper

We'll extract our fetch logic into a Data Loader

```
class DataLoader: ObservableObject {
    @Published var loadedData: [Post] = []
    private var isLoadingData = false

    var urlSession: URLSession?
    var endpoint: RemoteData.Endpoint?

    init() { }

    func fetchDataIfNeeded() {
        guard let urlSession = urlSession, let endpoint = endpoint,
            !isLoadingData && loadedData.isEmpty else {
            return
        }

        isLoadingData = true
        let url = URL.for(endpoint)
        urlSession.dataTask(with: url) { data, response, error in
            guard let data = data else { return }

            DispatchQueue.main.async {
                self.loadedData = try! JSONDecoder().decode([Post].self, from: data)
            }
            self.isLoadingData = false
        }.resume()
    }
}
```

We'll extract our fetch logic into a Data Loader

```
class DataLoader: ObservableObject {
    @Published var loadedData: [Post] = []
    private var isLoadingData = false

    var urlSession: URLSession?
    var endpoint: RemoteData.Endpoint?

    init() { }

    func fetchDataIfNeeded() {
        guard let urlSession = urlSession, let endpoint = endpoint,
            !isLoadingData && loadedData.isEmpty else {
            return
        }

        isLoadingData = true
        let url = URL.for(endpoint)
        urlSession.dataTask(with: url) { data, response, error in
            guard let data = data else { return }

            DispatchQueue.main.async {
                self.loadedData = try! JSONDecoder().decode([Post].self, from: data)
            }
            self.isLoadingData = false
        }.resume()
    }
}
```

We'll extract our fetch logic into a Data Loader

```
class DataLoader: ObservableObject {
    @Published var loadedData: [Post] = []
    private var isLoadingData = false

    var urlSession: URLSession?
    var endpoint: RemoteData.Endpoint?

    init() { }

    func fetchDataIfNeeded() {
        guard let urlSession = urlSession, let endpoint = endpoint,
            !isLoadingData && loadedData.isEmpty else {
            return
        }

        isLoadingData = true
        let url = URL.for(endpoint)
        urlSession.dataTask(with: url) { data, response, error in
            guard let data = data else { return }

            DispatchQueue.main.async {
                self.loadedData = try! JSONDecoder().decode([Post].self, from: data)
            }
            self.isLoadingData = false
        }.resume()
    }
}
```

We'll extract our fetch logic into a Data Loader

```
class DataLoader: ObservableObject {
    @Published var loadedData: [Post] = []
    private var isLoadingData = false

    var urlSession: URLSession?
    var endpoint: RemoteData.Endpoint?

    init() { }

    func fetchDataIfNeeded() {
        guard let urlSession = urlSession, let endpoint = endpoint,
            !isLoadingData && loadedData.isEmpty else {
            return
        }

        isLoadingData = true
        let url = URL.for(endpoint)
        urlSession.dataTask(with: url) { data, response, error in
            guard let data = data else { return }

            DispatchQueue.main.async {
                self.loadedData = try! JSONDecoder().decode([Post].self, from: data)
            }
            self.isLoadingData = false
        }.resume()
    }
}
```

We'll extract our fetch logic into a Data Loader

```
class DataLoader: ObservableObject {
    @Published var loadedData: [Post] = []
    private var isLoadingData = false

    var urlSession: URLSession?
    var endpoint: RemoteData.Endpoint?

    init() { }

    func fetchDataIfNeeded() {
        guard let urlSession = urlSession, let endpoint = endpoint,
            !isLoadingData && loadedData.isEmpty else {
            return
        }

        isLoadingData = true
        let url = URL.for(endpoint)
        urlSession.dataTask(with: url) { data, response, error in
            guard let data = data else { return }

            DispatchQueue.main.async {
                self.loadedData = try! JSONDecoder().decode([Post].self, from: data)
            }
            self.isLoadingData = false
        }.resume()
    }
}
```

We'll extract our fetch logic into a Data Loader

```
class DataLoader: ObservableObject {
    @Published var loadedData: [Post] = []
    private var isLoadingData = false

    var urlSession: URLSession?
    var endpoint: RemoteData.Endpoint?

    init() { }

    func fetchDataIfNeeded() {
        guard let urlSession = urlSession, let endpoint = endpoint,
            !isLoadingData && loadedData.isEmpty else {
            return
        }

        isLoadingData = true
        let url = URL.for(endpoint)
        urlSession.dataTask(with: url) { data, response, error in
            guard let data = data else { return }

            DispatchQueue.main.async {
                self.loadedData = try! JSONDecoder().decode([Post].self, from: data)
            }
            self.isLoadingData = false
        }.resume()
    }
}
```

Because DataLoader is an ObservableObject, it can tell our property wrapper that it changed, which will redraw our view

Next step: rewrite the property wrapper

```
@propertyWrapper
struct RemoteData: DynamicProperty {
    @Environment(\.urlSession) var urlSession
    @StateObject private var dataLoader = DataLoader()
    private let endpoint: Endpoint

    var wrappedValue: [Post] {
        dataLoader.loadedData
    }

    init(endpoint: Endpoint) {
        self.endpoint = endpoint
    }

    func update() {
        if dataLoader.urlSession == nil || dataLoader.endpoint == nil {
            dataLoader.urlSession = urlSession
            dataLoader.endpoint = endpoint
        }

        dataLoader.fetchDataIfNeeded()
    }
}
```

Next step: rewrite the property wrapper

```
@propertyWrapper
struct RemoteData: DynamicProperty {
    @Environment(\.urlSession) var urlSession
    @StateObject private var dataLoader = DataLoader()
    private let endpoint: Endpoint

    var wrappedValue: [Post] {
        dataLoader.loadedData
    }

    init(endpoint: Endpoint) {
        self.endpoint = endpoint
    }

    func update() {
        if dataLoader.urlSession == nil || dataLoader.endpoint == nil {
            dataLoader.urlSession = urlSession
            dataLoader.endpoint = endpoint
        }

        dataLoader.fetchDataIfNeeded()
    }
}
```

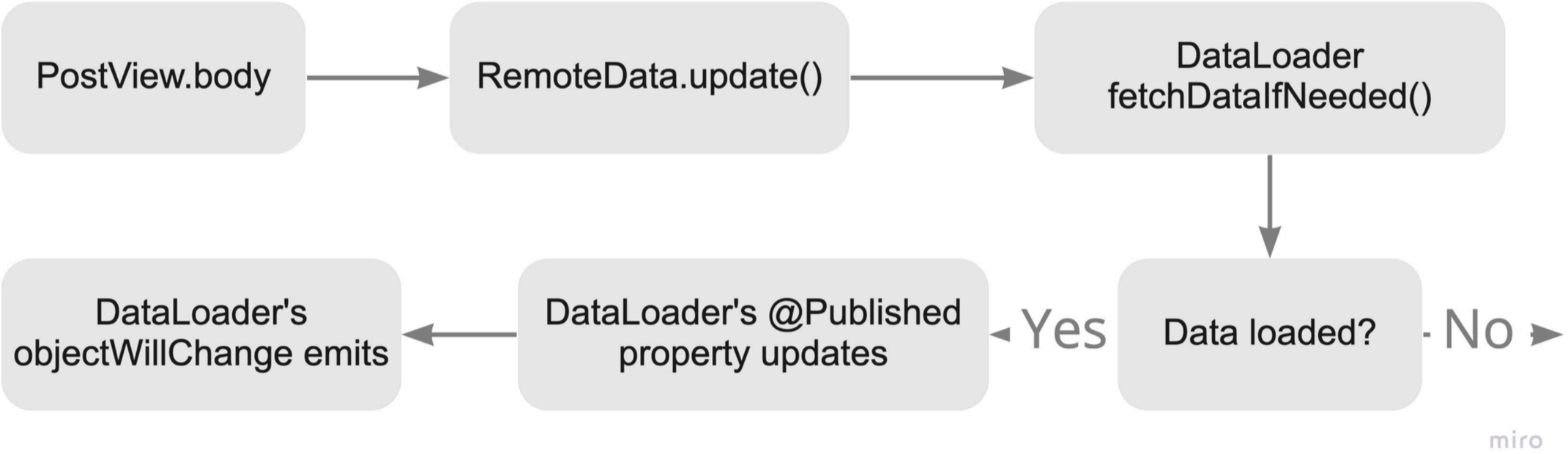
Next step: rewrite the property wrapper

```
@propertyWrapper
struct RemoteData: DynamicProperty {
    @Environment(\.urlSession) var urlSession
    @StateObject private var dataLoader = DataLoader()
    private let endpoint: Endpoint

    var wrappedValue: [Post] {
        dataLoader.loadedData
    }

    init(endpoint: Endpoint) {
        self.endpoint = endpoint
    }

    func update() {
        if dataLoader.urlSession == nil || dataLoader.endpoint == nil {
            dataLoader.urlSession = urlSession
            dataLoader.endpoint = endpoint
        }

        dataLoader.fetchDataIfNeeded()
    }
}
```

Next step: rewrite the property wrapper

```
@propertyWrapper
struct RemoteData: DynamicProperty {
    @Environment(\.urlSession) var urlSession
    @StateObject private var dataLoader = DataLoader()
    private let endpoint: Endpoint

    var wrappedValue: [Post] {
        dataLoader.loadedData
    }

    init(endpoint: Endpoint) {
        self.endpoint = endpoint
    }

    func update() {
        if dataLoader.urlSession == nil || dataLoader.endpoint == nil {
            dataLoader.urlSession = urlSession
            dataLoader.endpoint = endpoint
        }

        dataLoader.fetchDataIfNeeded()
    }
}
```

The final diagram



Phew... what an adventure...

Here's what it's all for

```
struct PostsView: View {
    @RemoteData(endpoint: .feed) var feed: [Post]

    var body: some View {
        List(feed) { post in
            Text(post.title)
        }
    }
}
```

Wrapping up DynamicProperties (pun intended)

Wrapping up DynamicProperties (pun intended)

- DynamicProperty allows us to build property wrappers that drive SwiftUI views.

Wrapping up DynamicProperties (pun intended)

- DynamicProperty allows us to build property wrappers that drive SwiftUI views.
- They can also tie into SwiftUI’s environment.

Wrapping up DynamicProperties (pun intended)

- DynamicProperty allows us to build property wrappers that drive SwiftUI views.
- They can also tie into SwiftUI’s environment.
- They can leverage SwiftUI’s state related property wrappers.

Wrapping up DynamicProperties (pun intended)

- DynamicProperty allows us to build property wrappers that drive SwiftUI views.
- They can also tie into SwiftUI’s environment.
- They can leverage SwiftUI’s state related property wrappers.
- A DynamicProperty property wrapper should (apparently) be a struct.

Wrapping up DynamicProperties (pun intended)

- DynamicProperty allows us to build property wrappers that drive SwiftUI views.
- They can also tie into SwiftUI’s environment.
- They can leverage SwiftUI’s state related property wrappers.
- A DynamicProperty property wrapper should (apparently) be a struct.
- The update() method is called for every body evaluation; you’ll want to check whether there’s work to be done.

Wrapping up DynamicProperties (pun intended)

- DynamicProperty allows us to build property wrappers that drive SwiftUI views.
- They can also tie into SwiftUI’s environment.
- They can leverage SwiftUI’s state related property wrappers.
- A DynamicProperty property wrapper should (apparently) be a struct.
- The update() method is called for every body evaluation; you’ll want to check whether there’s work to be done.
- BONUS: Your dynamic properties don’t have to be property wrappers

So how do we test this thing?

So how do we test this thing?

- We should create a hosting environment

So how do we test this thing?

- We should create a hosting environment
- We need a view to test

So how do we test this thing?

- We should create a hosting environment
- We need a view to test
- We’ll need to somehow receive updates whenever the view redraws

So how do we test this thing?

- We should create a hosting environment
- We need a view to test
- We’ll need to somehow receive updates whenever the view redraws
- Ideally we can extract the property wrapper’s current wrapped value

Creating a hosting environment

```
class CustomWrapperTest: XCTestCase {
    final func host<V: View>(_ view: V) {
        let app = UIViewController()
        let hosting = UIHostingController(rootView: view)
        hosting.view.translatesAutoresizingMaskIntoConstraints = false
        app.addChild(hosting)
        app.view.addSubview(hosting.view)
        NSLayoutConstraint.activate([
            hosting.view.leadingAnchor.constraint(equalTo: app.view.leadingAnchor),
            hosting.view.topAnchor.constraint(equalTo: app.view.topAnchor),
            hosting.view.trailingAnchor.constraint(equalTo: app.view.trailingAnchor),
            hosting.view.bottomAnchor.constraint(equalTo: app.view.bottomAnchor),
        ])
        app.view.layoutIfNeeded()
    }
}
```

Creating a hosting environment

```
class CustomWrapperTest: XCTestCase {
    final func host<V: View>(_ view: V) {
        let app = UIViewController()
        let hosting = UIHostingController(rootView: view)
        hosting.view.translatesAutoresizingMaskIntoConstraints = false
        app.addChild(hosting)
        app.view.addSubview(hosting.view)
        NSLayoutConstraint.activate([
            hosting.view.leadingAnchor.constraint(equalTo: app.view.leadingAnchor),
            hosting.view.topAnchor.constraint(equalTo: app.view.topAnchor),
            hosting.view.trailingAnchor.constraint(equalTo: app.view.trailingAnchor),
            hosting.view.bottomAnchor.constraint(equalTo: app.view.bottomAnchor),
        ])
        app.view.layoutIfNeeded()
    }
}
```

Creating a hosting environment

```
class CustomWrapperTest: XCTestCase {
    final func host<V: View>(_ view: V) {
        let app = UIViewController()
        let hosting = UIHostingController(rootView: view)
        hosting.view.translatesAutoresizingMaskIntoConstraints = false
        app.addChild(hosting)
        app.view.addSubview(hosting.view)
        NSLayoutConstraint.activate([
            hosting.view.leadingAnchor.constraint(equalTo: app.view.leadingAnchor),
            hosting.view.topAnchor.constraint(equalTo: app.view.topAnchor),
            hosting.view.trailingAnchor.constraint(equalTo: app.view.trailingAnchor),
            hosting.view.bottomAnchor.constraint(equalTo: app.view.bottomAnchor),
        ])
        app.view.layoutIfNeeded()
    }
}
```

Creating a hosting environment

```
class CustomWrapperTest: XCTestCase {
    final func host<V: View>(_ view: V) {
        let app = UIViewController()
        let hosting = UIHostingController(rootView: view)
        hosting.view.translatesAutoresizingMaskIntoConstraints = false
        app.addChild(hosting)
        app.view.addSubview(hosting.view)
        NSLayoutConstraint.activate([
            hosting.view.leadingAnchor.constraint(equalTo: app.view.leadingAnchor),
            hosting.view.topAnchor.constraint(equalTo: app.view.topAnchor),
            hosting.view.trailingAnchor.constraint(equalTo: app.view.trailingAnchor),
            hosting.view.bottomAnchor.constraint(equalTo: app.view.bottomAnchor),
        ])
        app.view.layoutIfNeeded()
    }
}
```

Creating a hosting environment

```
class CustomWrapperTest: XCTestCase {
    final func host<V: View>(_ view: V) {
        let app = UIViewController()
        let hosting = UIHostingController(rootView: view)
        hosting.view.translatesAutoresizingMaskIntoConstraints = false
        app.addChild(hosting)
        app.view.addSubview(hosting.view)
        NSLayoutConstraint.activate([
            hosting.view.leadingAnchor.constraint(equalTo: app.view.leadingAnchor),
            hosting.view.topAnchor.constraint(equalTo: app.view.topAnchor),
            hosting.view.trailingAnchor.constraint(equalTo: app.view.trailingAnchor),
            hosting.view.bottomAnchor.constraint(equalTo: app.view.bottomAnchor),
        ])
        app.view.layoutIfNeeded()
    }
}
```

Creating a hosting environment

```
class CustomWrapperTest: XCTestCase {
    final func host<V: View>(_ view: V) {
        let app = UIViewController()
        let hosting = UIHostingController(rootView: view)
        hosting.view.translatesAutoresizingMaskIntoConstraints = false
        app.addChild(hosting)
        app.view.addSubview(hosting.view)
        NSLayoutConstraint.activate([
            hosting.view.leadingAnchor.constraint(equalTo: app.view.leadingAnchor),
            hosting.view.topAnchor.constraint(equalTo: app.view.topAnchor),
            hosting.view.trailingAnchor.constraint(equalTo: app.view.trailingAnchor),
            hosting.view.bottomAnchor.constraint(equalTo: app.view.bottomAnchor),
        ])
        app.view.layoutIfNeeded()
    }
}
```

We can create a view to use in our test

```
struct SampleTestView: View {
    @RemoteData(endpoint: .feed) var feed: [Post]

    var body: some View {
        Text("This is just a test view...")
    }
}
```

Writing the XCTestCase

```
class SwiftUIPropertyWrapperTalkTests: CustomWrapperTest {
    func testRemoteDataIsEventuallyAvailable() throws {
        let view = SampleTestView()
            .environment(\.urlSession, URLSession.shared)
        host(view)
    }
}
```

And now we… uhm…

And now we… uhm…

And now we… uhm…

- When we apply the environmentObject view modifier to our SampleView, we no longer have an instance of SampleView.

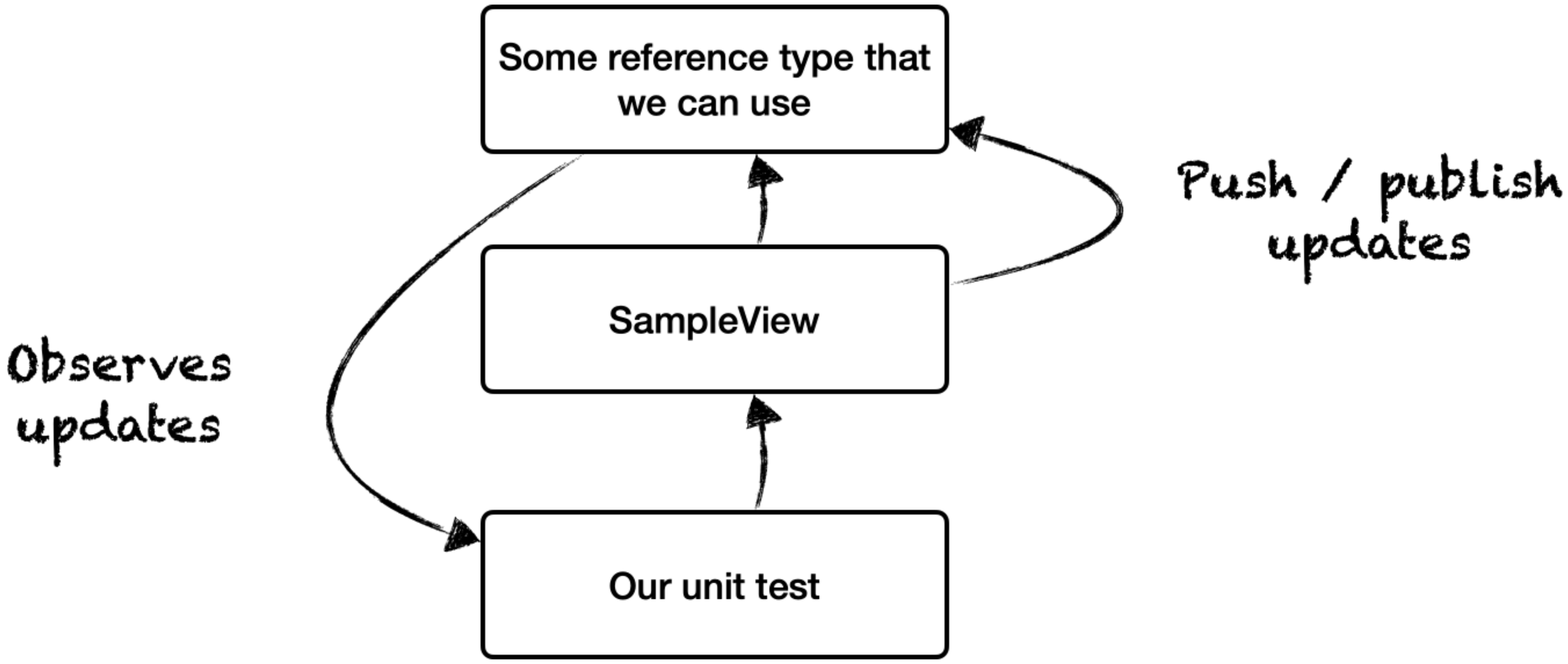
And now we… uhm…

- When we apply the environmentObject view modifier to our SampleView, we no longer have an instance of SampleView.
- We can’t make the view first, set up some observers, and then apply the environmentObject view modifier.

And now we… uhm…

- When we apply the environmentObject view modifier to our SampleView, we no longer have an instance of SampleView.
- We can’t make the view first, set up some observers, and then apply the environmentObject view modifier.
- We should add something to our sample view that has reference semantics.

Here's what we're looking for



This sounds like a job for Combine!

A view that tells us what we want to know

```
struct SampleTestView: View {
    @RemoteData(endpoint: .feed) var feed: [Post]

    let resultSubject = PassthroughSubject<[Post], Never>()

    var body: some View {
        Text("This is just a test view...")
            .onAppear {
                resultSubject.send(feed)
            }
            .onChange(of: feed) { newFeed in
                resultSubject.send(newFeed)
            }
    }
}
```

A view that tells us what we want to know

```
struct SampleTestView: View {
    @RemoteData(endpoint: .feed) var feed: [Post]

    let resultSubject = PassthroughSubject<[Post], Never>()

    var body: some View {
        Text("This is just a test view...")
            .onAppear {
                resultSubject.send(feed)
            }
            .onChange(of: feed) { newFeed in
                resultSubject.send(newFeed)
            }
    }
}
```

A view that tells us what we want to know

```
struct SampleTestView: View {
    @RemoteData(endpoint: .feed) var feed: [Post]

    let resultSubject = PassthroughSubject<[Post], Never>()

    var body: some View {
        Text("This is just a test view...")
            .onAppear {
                resultSubject.send(feed)
            }
            .onChange(of: feed) { newFeed in
                resultSubject.send(newFeed)
            }
    }
}
```

A view that tells us what we want to know

```
struct SampleTestView: View {
    @RemoteData(endpoint: .feed) var feed: [Post]

    let resultSubject = PassthroughSubject<[Post], Never>()

    var body: some View {
        Text("This is just a test view...")
            .onAppear {
                resultSubject.send(feed)
            }
            .onChange(of: feed) { newFeed in
                resultSubject.send(newFeed)
            }
    }
}
```

The full test

```
class SwiftUIPropertyWrapperTalkTests: CustomWrapperTest {

    var cancellables = Set<AnyCancellable>()

    func testRemoteDataIsEventuallyAvailable() throws {
        let view = SampleTestView()

        let expect = expectation(description: "expected data to be loaded")
        view.resultSubject.sink { posts in
            if !posts.isEmpty {
                expect.fulfill()
            }
        }.store(in: &cancellables)

        host(view.environment(\.urlSession, URLSession.shared))
        waitForExpectations(timeout: 1)
    }
}
```

The full test

```
class SwiftUIPropertyWrapperTalkTests: CustomWrapperTest {

    var cancellables = Set<AnyCancellable>()

    func testRemoteDataIsEventuallyAvailable() throws {
        let view = SampleTestView()

        let expect = expectation(description: "expected data to be loaded")
        view.resultSubject.sink { posts in
            if !posts.isEmpty {
                expect.fulfill()
            }
        }.store(in: &cancellables)

        host(view.environment(\.urlSession, URLSession.shared))
        waitForExpectations(timeout: 1)
    }
}
```

The full test

```
class SwiftUIPropertyWrapperTalkTests: CustomWrapperTest {

    var cancellables = Set<AnyCancellable>()

    func testRemoteDataIsEventuallyAvailable() throws {
        let view = SampleTestView()

        let expect = expectation(description: "expected data to be loaded")
        view.resultSubject.sink { posts in
            if !posts.isEmpty {
                expect.fulfill()
            }
        }.store(in: &cancellables)

        host(view.environment(\.urlSession, URLSession.shared))
        waitForExpectations(timeout: 1)
    }
}
```

The full test

```
class SwiftUIPropertyWrapperTalkTests: CustomWrapperTest {

    var cancellables = Set<AnyCancellable>()

    func testRemoteDataIsEventuallyAvailable() throws {
        let view = SampleTestView()

        let expect = expectation(description: "expected data to be loaded")
        view.resultSubject.sink { posts in
            if !posts.isEmpty {
                expect.fulfill()
            }
        }.store(in: &cancellables)

        host(view.environment(\.urlSession, URLSession.shared))
        waitForExpectations(timeout: 1)
    }
}
```

The full test

```
class SwiftUIPropertyWrapperTalkTests: CustomWrapperTest {

    var cancellables = Set<AnyCancellable>()

    func testRemoteDataIsEventuallyAvailable() throws {
        let view = SampleTestView()

        let expect = expectation(description: "expected data to be loaded")
        view.resultSubject.sink { posts in
            if !posts.isEmpty {
                expect.fulfill()
            }
        }.store(in: &cancellables)

        host(view.environment(\.urlSession, URLSession.shared))
        waitForExpectations(timeout: 1)
    }
}
```

The full test

```
class SwiftUIPropertyWrapperTalkTests: CustomWrapperTest {

    var cancellables = Set<AnyCancellable>()

    func testRemoteDataIsEventuallyAvailable() throws {
        let view = SampleTestView()

        let expect = expectation(description: "expected data to be loaded")
        view.resultSubject.sink { posts in
            if !posts.isEmpty {
                expect.fulfill()
            }
        }.store(in: &cancellables)

        host(view.environment(\.urlSession, URLSession.shared))
        waitForExpectations(timeout: 1)
    }
}
```

With some creativity, we came a long way!

In Summary

In Summary

- We can build custom properties with SwiftUI’s DynamicProperty

In Summary

- We can build custom properties with SwiftUI’s DynamicProperty
- You can tap into the environment which opens up a lot of possibilities

In Summary

- We can build custom properties with SwiftUI’s DynamicProperty
- You can tap into the environment which opens up a lot of possibilities
- To test a custom property wrapper you need a view environment

In Summary

- We can build custom properties with SwiftUI’s DynamicProperty
- You can tap into the environment which opens up a lot of possibilities
- To test a custom property wrapper you need a view environment
- A Combine subject is a nice way to get a reference type from the view that you can publish changes on. (Yes this can be made to work as an async sequence)

Thank you!

https://github.com/donnywals/SwiftUIPropertyWrapperTalk