Building custom SwiftUI property wrappers

Donny Wals - 2022

Donny Wals

iOS Engineer, Speaker, Writer

https://practicalcombine.com/

DW
donnywals.com

https://practicalcoredata.com/

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

```
@State var isActive = false
```

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

Is it magic?

Can you build it yourself?

Wouldn’t it be neat to have a property wrapper like this?

```
@FeatureFlag(\.newFeed) var shouldShowNewFeed
```

Or maybe like this?

```
@RemoteData(.posts) var posts = []
```

Let’s go and build it!

The anatomy of a property wrapper

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

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

The anatomy of a property wrapper

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

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

The anatomy of a property wrapper

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

    func printSample() {
        print(example) // "Hello, world"
        print(_example) // StringSample
        print($example) // "Projected: Hello, world"
    }
}
```

The anatomy of a property wrapper

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

    func printSample() {
        print(example) // "Hello, world"
        print(_example) // StringSample
        print($example) // "Projected: Hello, world"
    }
}
```

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

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

The anatomy of a property wrapper

```
struct PostsView: View {
    @State var isActive: Bool

    init(isActive: Bool) {
        _isActive = State(wrappedValue: isActive)
    }

    var body: some View {
        if isActive {
            Text("Yes, we're active")
        } else {
            Button("Become active") {
                isActive = true
            }
        }
    }
}
```

The anatomy of a property wrapper

```
struct PostsView: View {
    @State var isActive: Bool

    init(isActive: Bool) {
        _isActive = State(wrappedValue: isActive)
    }

    var body: some View {
        if isActive {
            Text("Yes, we're active")
        } else {
            Button("Become active") {
                isActive = true
            }
        }
    }
}
```

So how do we implement this property wrapper for SwiftUI?

```
@RemoteData(.posts) var posts = []
```

Let’s start with the basics

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

    var wrappedValue: [Post] = []

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

And let's add some data fetching capabilities

```
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")!
    }
}
```

So 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

- DynamicValue 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 prepare the environment

```
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() {
        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()
    }
}
```

Let’s set up 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()
    }
}
```

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()
    }
}
```

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…

Spot the difference!

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

    @State 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()
    }
}
```

Spot the difference!

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

    @State 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()
    }
}
```

Run the app and...

Here’s what happens now

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

So... about update()

update()

Updates the underlying value of the stored value.

Discussion

SwiftUI calls this function before rendering a view’s body to ensure the view has the most recent value.

https://developer.apple.com/documentation/swiftui/dynamicproperty/update()-29hjo

update()

Updates the underlying value of the stored value.

Discussion

SwiftUI calls this function before rendering a view's body to ensure the view has the most recent value.

https://developer.apple.com/documentation/swiftui/dynamicproperty/update()-29hjo

Here's what we have now

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

    @State 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 fetch data on every call to update()

Here’s what happens now

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
            }

            print("updating wrapped value")
            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
            }

            print("updating wrapped value")
            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
            }

            print("updating wrapped value")
            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
            }

            print("updating wrapped value")
            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
            }

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

But...

```
@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
            }

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

Cannot assign to property: 'self' is immutable

Cannot assign to property: 'self' is immutable

So we could do this...

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

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

But...

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

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

⚠ Modifying state during view update, this will cause undefined behavior.

update()

Updates the underlying value of the stored value.

Discussion

SwiftUI calls this function before rendering a view's body to ensure the view has the most recent value.

https://developer.apple.com/documentation/swiftui/dynamicproperty/update()-29hjo

Let’s separate the underlying value from the wrappedValue

First, let's build a DataLoader

```
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()
    }
}
```

First, let's build a DataLoader

```
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()
    }
}
```

First, let's build a DataLoader

```
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()
    }
}
```

First, let's build a DataLoader

```
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()
    }
}
```

First, let's build a DataLoader

```
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 drive our SwiftUI views through the DynamicProperty...

And now we update RemoteData

```
@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()
    }
}
```

And now we update RemoteData

```
@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()
    }
}
```

And now we update RemoteData

```
@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()
    }
}
```

And now we update RemoteData

```
@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()
    }
}
```

Here's what happens now

That app still works 👍🏼

And this is 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

- 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 should be a struct
- The update() method is called for every body evaluation; you'll want to check whether there's work to be done.

So how do we test this thing?

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 view to test with

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

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

Setting up our test

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

And now we... uhm...

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)
    }
}
```

It’s not hard… it’s just tricky

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