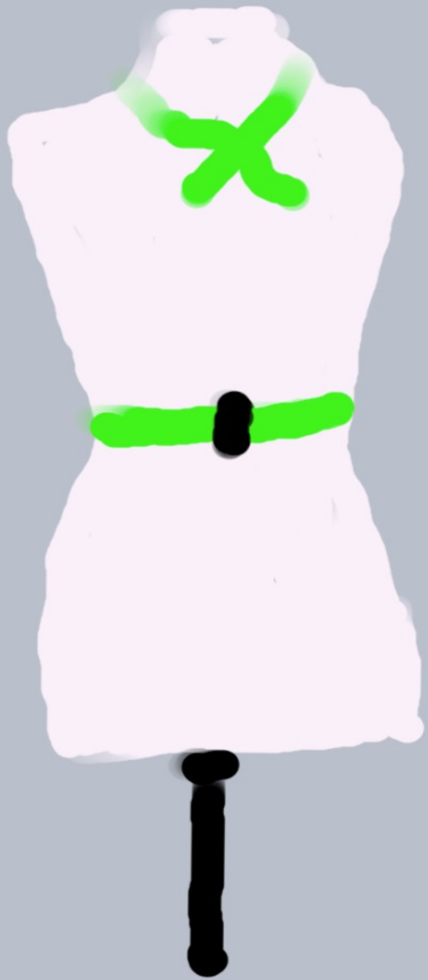
The Mystery of the Mutating Mannequin

An Exploration of Data Flow in SwiftUI



by Daniel H Steinberg

The Mystery Of The Mutating Mannequin

An Exploration Of

Data Flow In SwiftUI

by Daniel H Steinberg

Editors Cut

This is an excerpt from "The Mystery of the Mutating Mannequin."

Copyright

ISBN-13: 978-1-944994-05-1

Book Version

This is version 1.0 for Swift 5.9, Xcode 15.2, macOS Sonoma 14.3, and iOS 17.2 released February 2024. All code has been tested on Apple Silicon.

Code Download

Visit https://github.com/editorscut/ec014dataflow for all of the code for this book.

Run it in Xcode 15.2 or higher. All code is written in Swift.

To avoid long lines and code that wraps, I've split some lines in code listings in ways that you might not in an IDE. Please feel free to not break the lines where I have.

Recommended Settings

The ePub is best viewed in scrolling mode on an iPad. On smaller devices I also choose landscape. For some reason that I don't understand, scrolling mode is supported by Apple's Books app on the iPad but not on the Mac. If you view this book in Apple's Books app, choose "Let lines break naturally" in Preferences > General. Finally, I've gone to great pains to make this look good in light and dark mode but Apple has foiled me yet again. I'm told that not all of the syntax coloring works in dark mode.

Submit Errata

Submit your errata here for the book or for the source code by selecting New Issue. Please provide the book version listed above, chapter, section, and page number in your issue so that I can find it and, if possible, resolve it quickly.

Official Links

Please check http://developer.apple.com for additional resources including videos, sample code, documentation, and forums. You'll also find information on what is required to take advantage of these resources.

Apple has posted videos, slides, and sample code from the Worldwide Developers Conference.

Legal

Table Of Contents

CHAPTER 3

Sharing State

Sections:

We're at the part of the trek where the kids in the back seat are whining, "are we there yet?"

We tell them that the journey is just as important as the destination. Look out the window at the lovely cows.

Cows?

Well, in our case different ways of sharing state between two views so that changes initiated in one view can be seen in another.

"But you promised us SwiftData," the kids cry.

Sure. We're getting there. In fact, the topics of this chapter are setting up what we'll see in SwiftData.

But actually, there's a lot of intrinsic value in understanding bindings, `ObservableObject`s, the environment, `AsyncSequence`s, and `Observable` objects.

The kids roll their eyes, but they stop complaining. They sit back and look out the window.

You think you hear one of them say, "actually, this stuff is pretty cool."

In this chapter we look at various techniques for sharing state. Sometimes we need multiple views to reflect the same state and sometimes we want one view to mutate the state that others display. We start with the simplest case using bindings and then move on to more complicated setups where the relationships can't be passed from one view to another.

We build this using `ObservableObject` and `@Published`, then rebuild it using `AsyncSequence` and `AsyncStream`. We then use `Observable` and `Bindable` and simplify our code and communication by combining this approach with `@Binding`. I love how we come full circle and see that it isn't that `@Binding` is being replaced - it is more that each mechanism is used where it most makes sense.

Bindable

In the previous section we used `Observable` to communicate from the model to the views.

Now we'd like to allow the `TransportationTypePicker`'s `Picker` to bind to the `journey.transportationType` so that the changes in `Picker` selection change `journey.transportationType` directly without needing to implement `onChange()`.

To do this we use `@Bindable`, a property wrapper that is part of the `Observation` framework.

One of the interesting issues with `@Bindable` is that we need to apply it to the instance of the `Observable` object. In other words, we apply `@Bindable` to `journey` in `TransportationTypePicker` and not `transportationType` even though the `Picker` will be bound to `journey.transportationType` (or more accurately `$journey.transportationType`).

This presents us with a challenge as we have applied `@Environment` to `journey` to retrieve it from the environment and a property can't accommodate both `@Environment` and `@Bindable`.

You'll see three ways of addressing this limitation. We'll recast `journey` as `@Bindable` in `TransportationTypePicker`, we'll pull `journey` from the environment in `JourneyEditor` and pass it in to a `@Bindable` property, and we'll remove the use of the environment completely.

Continue with our current project or start with the project in *Chapter03/07/*.

Introducing Bindable

It's time for more programming by intention.

What I wish I could do is ignore the local `transportationType` that we use in `TransportationTypePicker` and bind to `journey`'s `transportationType`.

If we could do this then we wouldn't need `onChange()` because the `Picker` would be changing `journey.transportationType` using its bindings. We also wouldn't need `onAppear()` as `journey.transportationType` has a value when this view appears.

Something like this:

*RoadTrip/Views/Transportation Type Views/***TransportationTypePicker.swift**

```
struct TransportationTypePicker {
  ~~@State private var transportationType = TransportationType.bike~~
  @Environment(Journey.self) private var journey
}

extension TransportationTypePicker: View {
  var body: some View {
    let _ = Self._printChanges()
    Picker("Transportation Type",
           selection: $journey.transportationType) {
      ForEach(types) {type in
        Image(systemName: type.iconName)
      }
    }
    .pickerStyle(.segmented)
    .padding(.horizontal)
    ~~.onChange(of: transportationType) { oldValue, newValue in~~
      ~~journey.transportationType = newValue~~
    ~~}~~
    ~~.onAppear {~~
      ~~transportationType = journey.transportationType~~
    ~~}~~
  }
}
```

The correct property wrapper to use for `journey` is `@Bindable`,

BUT

as I mentioned, a property can't be both `@Environment` and `@Bindable`. This is gotcha number one.

Let's explore three strategies for addressing this. Although recommended in Apple documentation, the following is my least favorite.

Recasting as Bindable

The first way to address our problem is to create a local variable that is bindable inside `body`.

*RoadTrip/Views/Transportation Type Views/**TransportationTypePicker.swift***

```
extension TransportationTypePicker: View {
  var body: some View {
    @Bindable var journey = self.journey
    let _ = Self._printChanges()
    Picker("Transportation Type",
           selection: $journey.transportationType) {
      ForEach(types) {type in
        Image(systemName: type.iconName)
      }
    }
    .pickerStyle(.segmented)
    .padding(.horizontal)
  }
}
```

This builds and runs correctly but it just doesn't feel right to me. I can't explain why. It's essentially the same method we use for taking an argument of a method and recasting it as a `var` inside the method body.

We can, instead, create the `Bindable` version of `journey` at its point of use in the picker like this:

*RoadTrip/Views/Transportation Type Views/**TransportationTypePicker.swift***

```
extension TransportationTypePicker: View {
  var body: some View {
    ~~@Bindable var journey = self.journey~~
    let _ = Self._printChanges()
    Picker("Transportation Type",
           selection: Bindable(journey).transportationType) {
      ForEach(types) {type in
        Image(systemName: type.iconName)
      }
    }
    .pickerStyle(.segmented)
    .padding(.horizontal)
  }
}
```

Again, this builds and runs perfectly but it still feels awkward to me.

Once you've seen that this method works, let's undo it so we can try a second method.

In other words, remove `Bindable` and restore the `$`.

*RoadTrip/Views/Transportation Type Views/**TransportationTypePicker.swift***

```
extension TransportationTypePicker: View {
  var body: some View {
    let _ = Self._printChanges()
    Picker("Transportation Type",
           selection: $journey.transportationType) {
      ForEach(types) {type in
        Image(systemName: type.iconName)
      }
    }
    .pickerStyle(.segmented)
    .padding(.horizontal)
  }
}
```

The second approach is to move `@Environment` up a level.

Retrieving Journey in JourneyEditor

Instead of pulling `Journey` from the environment in `TransportationTypePicker`, we'll do that in `JourneyEditor` and pass it to `TransportationTypePicker`.

This won't compile yet as we haven't made the corresponding changes to `TransportationTypePicker`, but here's `JourneyEditor`.

*RoadTrip/Views/Journey Views/**JourneyEditor.swift***

```
struct JourneyEditor {
  @Environment(Journey.self) private var journey
}

extension JourneyEditor: View {
  var body: some View {
    let _ = Self._printChanges()
    TransportationTypePicker(journey: journey)
  }
}
```

Now wrap `journey` with `@Bindable` in `TransportationTypePicker`.

*RoadTrip/Views/Transportation Type Views/**TransportationTypePicker.swift***

```
struct TransportationTypePicker {
  @Bindable var journey: Journey
}
```

Unlike `@Binding`, the preview just uses an ordinary instance of `Journey`.

*RoadTrip/Views/Transportation Type Views/**TransportationTypePicker.swift***

```
#Preview {
  TransportationTypePicker(journey: Journey())
}
```

Run the app. It works perfectly. Despite the quirkiness of the restrictions on `@Bindable` it is a very powerful construct. We are able to change the value of a far-away property that is seen by other views.

Of course, there's another way to avoid the conflict between `@Environment` and `@Bindable`.

Don't use the environment

Our third strategy for addressing the issue that `journey` needs to be `@Bindable` and pulled from the environment and you can't do both at the same time, is to not use the environment at all.

At this point both `JourneyView` and `JourneyEditor` retrieve `Journey` from the environment. We can eliminate our use of the environment by creating the common instance of `Journey` in the view that contains these two: `EditableJourneyView`.

Remove `journey` from `RoadTripApp`.

*RoadTrip/Launch/**RoadTripApp.swift***

```
@main
struct RoadTripApp {
  ~~@State private var journey = Journey()~~
}

extension RoadTripApp: App {
  var body: some Scene {
    WindowGroup {
      ContentView()
    }
    ~~.environment(journey)~~
  }
}
```

Add `journey` to `EditableJourneyView`. Even though they aren't ready to receive it yet, pass `journey` on to `JourneyView` and `JourneyEditor`.

*RoadTrip/Views/Journey Views/**EditableJourneyView.swift***

```
struct EditableJourneyView {
  @State private var journey = Journey()
}

extension EditableJourneyView: View {
  var body: some View {
    let _ = Self._printChanges()
    VStack {
      JourneyView(journey: journey)
      JourneyEditor(journey: journey)
    }
  }
}
```

Remove the `@Environment` in `JourneyView` and make `journey` a simple `let`.

*RoadTrip/Views/Journey Views/**JourneyView.swift***

```
struct JourneyView {
  let journey: Journey
}
```

Do the same for `JourneyEditor`.

*RoadTrip/Views/Journey Views/**JourneyEditor.swift***

```
struct JourneyEditor {
  let journey: Journey
}
```

Run the app. Everything works perfectly. Isn't this cool?

Actually, now that we aren't using the environment, we can refine this by combining `@Bindable` and `@Binding`. Let's explore that in the next section.