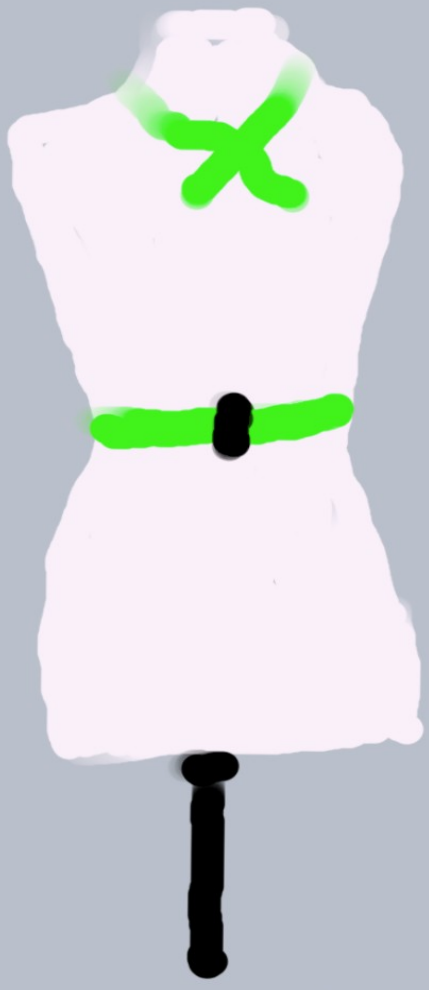
The Mystery of the Mutating Mannequin

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 2

View Identity

Sections:
- A Simple Model
- Stored Properties
- State and Memory
- State and Identity
- Changing State
- Adding a Conditional Identity

We're ready to add state to our views.

Although many of the topics sound familiar, you're going to see new aspects of SwiftUI by applying some of the tools we use in this chapter.

We begin with the simplest case in which a view has an immutable stored property and look at how that impacts the memory footprint of the view.

I'm not saying that the app we're building is that much fun, but if we don't account for mutable state this will be the worst app ever. So we use the `@State` property wrapper to allow mutating state that changes when a button is tapped.

By printing `self` we can see how the memory footprint changes when a property is wrapped with `@State`. In fact, the value is stored somewhere else and we also have an `identity` for the view so that the view can be reunited with its state in circumstances that we'll see at the very end of the chapter.

In addition to understanding the implications of `@State` on memory, we begin to investigate a view's identity and check out what happens when we have two presentations of the same view type in different branches of a conditional.

By the end of the chapter you'll have a pretty good idea of why view identity is so important and how it influences what we see on the screen.

State And Memory

"Swiftly, my friend," smiled Edges as we sat down to our afternoon coffee, "what have you discovered?"

"I've just received an email from Jean-Claude with the information," I said, "and the best selling color yesterday for the belt and matching hat was cerulean."

"Just as I expected," nodded Edges. "Try this. Each day ask your sales staff for the fourth-best selling color and change the display to use that color."

"Fourth best?", I asked.

"Well, there is no way to elevate the very bottom to the very top, but let's see if we can boost something in the middle."

I texted Jean-Claude and asked that each day he look at the sales and direct the window dresser to get the belt and ribbon in that color and change out the display.

Of course, in our code we will do essentially the same thing. The `TransportationTypeView` needs to be able to swap out its `TransportationType`. The `TransportationTypeImageView` and `TransportationTypeTitleView` needs to reflect these changes.

You've used `@State` before so the actual code won't be that new to you but we will learn a thing or two about what's going on under the hood and about view identity.

Continue with our current project or start with the `RoadTrip` project in *Chapter02/02/*.

Mutating transportationType

Introduce a method named `changeTransportationType` to `TransportationTypeView` that assigns a new random `TransportationType` to `transportationType`.

We are going to use this method as a button action. This restricts us in two ways. First, the signature of the method must be such that it does not accept or return any values and is marked neither `async` nor `throws`. Second, the method can not be mutating. The method cannot modify any stored properties that are value types.

That means, of course, that the following doesn't compile. Type it in anyway - we're going to make it compile.

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

```
extension TransportationTypeView {
  private func changeTransportationType() {
    // the following line does not compile
    transportationType = TransportationType.random()
  }
}
```

When you were new to Swift and SwiftUI you may have thought that the issue was that `transportationType` is a `let` and changed it to a `var`.

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

```
struct TransportationTypeView {
  private var transportationType = TransportationType.random()
}
```

Although it is true that `transportationType` must be a `var`, that change is not sufficient.

Unfortunately, `TransportationTypeView`, like all SwiftUI views, is a value type. Since `TransportationType` is also a value type, modifying `transportationType` modifies `self`. The only way to make this work is to mark `changeTransportationType()` as `mutating` but we're about to use `changeTransportationType()` as a button action and button actions can't be `mutating`.

That's a long-winded way of saying, we need to use the `@State` property wrapper for `transportationType`.

@State

Before introducing `@State`, take a moment to comment out `changeTransportationType()` and rerun the app, Take a look at the Console output for `TransportationTypeView`. It should be something like this:

```
TransportationTypeView(transportationType: Scooter)
```

This tells me that `TransportationTypeView` is storing the value `scooter` in the property `transportationType` as part of the memory for `TransportationTypeView`.

That's explains why a change to `transportationType` is a change to `TransportationTypeView`.

The way I think about `@State` is that it stores `transportationType` somewhere outside of this instance of `TransportationTypeView`. In a way it turns a value type into a reference type and stores a reference to where `TransportationType` is being persisted.

As you'll see later, there's more to `@State` than this, but this alone means that once we've used `@State` with `transportationType` we can modify the value of `transportationType` without modifying `self`.

Uncomment `changeTransportationType()`.

Go ahead and add `@State` to `transportationType`.

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

```
struct TransportationTypeView {
  @State private var transportationType = TransportationType.random()
}
```

Now our code compiles.

In fact, go ahead and run the app and check out the difference in the Console output. I've used ↲ to indicate where I've inserted a new line in this listing that you won't see in the Console just so it fits on the screen.

```
ContentView()
TransportationTypeView(_transportationType: ↵
    SwiftUI.State<RoadTrip.TransportationType>(↵
      _value: Scooter, ↵
      _location: ↵
        Optional(SwiftUI.StoredLocation<RoadTrip.TransportationType>)))
TransportationTypeTitleView(title: "Scooter")
TransportationTypeImageView(imageName: "scooter")
```

As a quick summary, `ContentView` has no stored properties, `TransportationTypeTitleView` and `TransportationTypeImageView` have immutable stored properties, and `TransportationTypeView` now contains a `@State` property.

Let's look at the differences more carefully.

How a View sees a property decorated with @State

Before you get to all the generics, look to see if you can spot a difference between the currently listed property name in `TransportationTypeView` and the original version.

Here again is the original before we added `@State`.

```
TransportationTypeView(transportationType: Scooter)
```

And here is the current version now that we added `@State`.

```
TransportationTypeView(_transportationType: /* ... */)
```

It's like one of those puzzles from when you were younger.

The property is `_transportationType` not `transportationType`. If you missed it, there's a leading underscore.

We aren't storing the value of `transportationType` in `TransportationTypeView`, we're storing a reference to it.

Next, note the type of `_transportationType`. It's not `TransportationType`. It's:

```
SwiftUI.State<RoadTrip.TransportationType>
```

Its type is `State` generic in `TransportationType`.

This `State` has two components. There's the actual value which is, in my case, `scooter`, and there's a location. The location points to the `StoredLocation` where this `TransportationType` is stored. The actual location isn't exposed to us but is maintained by the runtime.

You'll see the contents of `_transportationType` in your Console represented by something like this (for readability I've shortened `SwiftUI.StoredLocation` to `StoredLocation` and `RoadTrip.TransportationType` to `TransportationType`):

```
(_value: Scooter,
 _location: Optional(StoredLocation<TransportationType>))
```

This location allows for a bit of the magic I alluded to earlier. Once the view has been rendered, SwiftUI might decide to clean up some of the memory. If the value changes and the view needs to be re-rendered, this instance of the `TransportationTypeView` needs to be reconnected to the storage for its `transportationType` so the system needs to hold on to this location.

In this section we've looked at how wrapping an ordinary stored property using the `@State` property wrapper changes the containing view and how and where the values of this property is stored.

In the next section we back up a bit and investigate the difference that `@State` brings to the identity of `TransportationTypeView`. At that point we can introduce a button that uses `changeTransportationType()` as its action.