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

"The Mystery of the Mutating Mannequin", by Daniel H Steinberg

Copyright © 2024 Dim Sum Thinking, Inc. All rights reserved.

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

Every precaution was taken in the preparation of this book. The publisher and author assume no responsibility for errors and omissions, or for damages resulting from the use of the information contained herein and in the accompanying code downloads.

The sample code is intended to be used to illustrate points made in the text. It is not intended to be used in production code.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks or service marks. Where those designations appear in this book, and Dim Sum Thinking, Inc. was aware of the trademark claim, the designations have been printed with initial capital letters or in all capitals.

This book uses terms that are registered trademarks of Apple Inc. for which the terms of use don't permit rendering them in all caps or initial caps. You can view a complete list of the trademarks and registered trademarks of Apple Inc at http://www.apple.com/legal/trademark/appletmlist.html.

The Editor's Cut name and logo are registered trademarks of Dim Sum Thinking, Inc.

Table Of Contents

Copyright and Legal

Copyright
Book Version
Code Download
Recommended Settings
Submit Errata
Official Links
Legal

Chapter 5: SwiftData

SwiftData Relationships

CHAPTER 5

SwiftData

Sections:

Stubbing out the Model
The Model Macro
SwiftData Relationships
The SwiftData Stack
Queries
>Adding and Deleting
Updating
Severing Relationships

Year after year, many of us make our predictions ahead of WWDC (Apple's Worldwide Developers Conference).

Year after year, many of us were wrong when we predicted SwiftData.

It seemed so obvious.

Before SwiftUI, we could write our UI in code or use Interface Builder whether we used Storyboards or not.

Storyboards were kind of a schema for the UI. We still needed to connect to it from our code and sometimes we had to write code to

perform tasks that couldn't be performed in the Storyboard alone. Oh, and Storyboards were really just XML underneath.

What if the UI could be expressed and persisted as Swift code and instead of us visually designing the UI, we would create it in code and get immediate feedback in previews?

The same situation existed at the model end. We could write, manage, and persist our object graph manually or using Core Data.

We still needed to connect to the model we designed in Core Data as you saw when we explored the Core Data stack. Sometimes we had to write code to perform tasks that Core Data couldn't do on its own - you saw that too in the previous chapter. Oh, and as you saw, the Core Data model was really just XML underneath.

What if the model could be expressed and persisted as Swift code?

SwiftData does that and more. There are a few restrictions because SwiftData sits on top of and can co-exist with Core Data in the same way that SwiftUI had to live in a world where there was UIKit and AppKit.

In this chapter we begin with a new and more complicated model with four entities and relationships between pairs of them. We look at the `Model` macro and the code that it keeps us from having to write and we see old friends like `Observable` hidden inside.

In its initial incarnation, SwiftData is only suited to be used in SwiftUI apps. Once we use the SwiftData stack to add SwiftData to our SwiftUI app there are many features that depend on everything

we've learned already about when SwiftUI views are updated and rerendered.

You'll see that the standard CRUD operations are very straightforward in a SwiftUI app. We use `@Query` to read data, sort it, and filter it. We use a `ModelContext` connected to our `ModelContainer` to create, delete, and update.

This is a quick introduction to SwiftData. There are, of course, edge cases and subtleties that are beyond the scope of this introduction. You will, however, be surprised at how easy it is to get started and get quite far with SwiftData using model code that you'd almost write if you weren't using SwiftData.

SwiftData Relationships

"Show me again," said Edges.

"Show you what?" I asked.

"Show me how you add the person to the window and then the person knows which windows it is concerned with."

I did.

"Now, if you don't mind," said Edges, "can you bring up a window and show me all of the other windows it shares people with?"

"I think I can," I said, "but why would you want such a thing?"

"Perhaps," said Edges, "one of these windows must be delayed. This would allow me to see which other windows would be impacted."

"Ahhh," I nodded. "I'll have to think of how to do this. It's all a matter of setting up the right relationships between windows and people."

Edges said they would be away for a few days on a case, but they would be sure to check back.

Our classes currently each refer to the other types. These are already relationships. A `Presenter` has a relationship to `sessions` and

a `Session` has a relationship to `presenters` but there is nothing yet connecting `sessions` and `presenters`.

We can't yet assume that if we add a `Presenter` to a `Session`'s `presenters` that that `Session` will be added to all of its `presenters`' `sessions`.

In this section we formalize relationships between relationships by making them inverses of each other. We also begin to explore the delete rule for relationships.

Continue with our current project or start with the project in *Chapter05/02/*.

Thoughts on Relationships

Remember what we had to do to create a relationship in Core Data.

We started by choosing one of the entities. For example, we selected `Journey` and then explicitly added a relationship named `startingPoint` with destination `Location`. Recall that we immediately got a warning that `startingPoint` needed an inverse relationship.

We then created a relationship from `Location` to `Journey` and marked `startingPoint` as its inverse relationship. Also because the inverse relationship was a to-many relationship we had to set that in the Data Model Inspector.

Oh and then we had to navigate back to `startingPoint` and set its inverse relationship.

Things are different in SwiftData.

In SwiftData a property whose type is another model object automatically defines a relationship.

For example, consider the `presenters` property in `Session`.

*Conferences/Model/**Session.swift***
```
@Model
class Session {
  var title: String
  var presenters: [Presenter] = []
  var timeSlot: TimeSlot?

  init(title: String) {
    self.title = title
  }
}
```

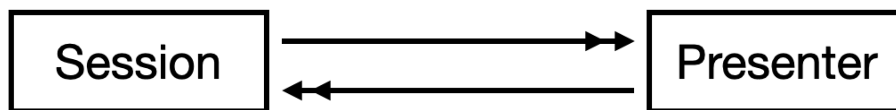
Not only is `presenters` a relationship but it's a to-many relationship because its type is a collection.

Similarly, consider the `sessions` property in `Presenter`. It too is a to-many relationship.

*Conferences/Model/**Presenter.swift***
```
@Model
class Presenter {
  let name: String
  var sessions: [Session] = []
  var links: [OnlineLink] = []

  init(name: String) {
    self.name = name
  }
}
```

So we have two relationships - one in each direction.



We got a lot for free. But if we want `sessions` and `presenters` to be inverses we need to do more. We can also specify deletion rules and specify constraints such as a maximum or minimum number of elements in the collection.

CloudKit

CloudKit provides a great way to sync among devices using the same iCloud account running a SwiftData or Core Data app.

The biggest challenge of getting to work is getting all the permissions right and setting things up in the web CloudKit interface.

We are not going to use CloudKit in this book.

BUT -

But, I wanted to bring up CloudKit and give it its own section to call attention to it to warn you that if you are going to use CloudKit all of your relationships in SwiftData must be `Optional`s.

"Why Daniel?" you ask.

Because Apple hates you and went out of its way to make your life miserable.

There. Feel better now?

Inverse Relationships

Let's set `presenters` to be the inverse of `sessions`.

Although it's not required, I added new lines on either side of the relationship code to indicate more clearly what `@Relationship` is modifying.

*Conferences/Model/**Session.swift***

```
@Model
class Session {
  var title: String

  @Relationship(inverse: \Presenter.sessions)
  var presenters: [Presenter] = []

  var timeSlot: TimeSlot?

  init(title: String) {
    self.title = title
  }
}
```

This is one of the rare cases where our experience with Core Data might lead us to do something incorrect.

We might be tempted to declare that `sessions` inverse is `presenters`.

*Conferences/Model/**Presenter.swift***

```
@Model
class Presenter {
  let name: String

  @Relationship(inverse: \Session.presenters) // error
  var sessions: [Session] = []

  var links: [OnlineLink] = []

  init(name: String) {
    self.name = name
  }
}
```

The error we get is essentially the compiler telling us, "you already told us that they are inverses."

```
Circular reference resolving attached macro 'Relationship'
```

We can fix this by removing the `inverse` parameter from the `@Relationship` macro.

*Conferences/Model/**Presenter.swift***

```
@Model
class Presenter {
  let name: String

  @Relationship(inverse: \Session.presenters)
  var sessions: [Session] = []

  var links: [OnlineLink] = []

  init(name: String) {
    self.name = name
  }
}
```

This is no longer incorrect but it is superfluous. We only use the `@Relationship` macro when we need to provide more information by setting one of its parameters. Here's the signature of the macro.

```
@Relationship(options: Schema.Relationship.Option...,
              deleteRule: Schema.Relationship.DeleteRule,
               minimumModelCount: Int?,
               maximumModelCount: Int?,
               originalName: String?,
               inverse: AnyKeyPath?,
              hashModifier: String?)
```

We aren't specifying any of these values so remove `@Relationship`.

*Conferences/Model/**Presenter.swift***

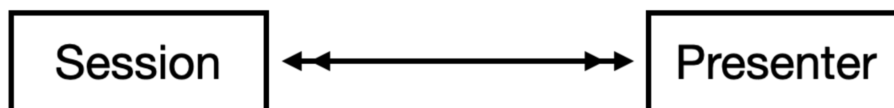
```
@Model
class Presenter {
  let name: String

  @Relationship
  var sessions: [Session] = []

  var links: [OnlineLink] = []

  init(name: String) {
    self.name = name
  }
}
```

Setting the inverse of `presenters` in `Session` was enough to join the two relationships as inverses of each other.



Before leaving `Session`, set the inverse of its `timeSlot` relationship to be `TimeSlot`'s `session` relationship.

*Conferences/Model/**Session.swift***

```
@Model
class Session {
  var title: String

  @Relationship(inverse: \Presenter.sessions)
  var presenters: [Presenter] = []

  @Relationship(inverse: \TimeSlot.session)
  var timeSlot: TimeSlot?

  init(title: String) {
    self.title = title
  }
}
```

We've got one more relationship pair to take care of but we also want to set the delete rule.

The Delete Rule

When you delete a `Session` what happens to the `Presenter`s reference in `presenters`?

You can't automatically delete them as well because they might be presenting other sessions.

But you have to be careful when deleting a `session` because its presenters have a reference back to it as one of it/their `sessions`. We need to remove the references back to the `session` to be deleted

.

This is the default delete rule `nullify`.

Our choices for the delete rule are:

- `cascade`: delete the objects referenced by the object to be deleted.
- `deny`: if the object to be deleted references and hence is referenced by other live objects don't allow it to be deleted
- `noAction`: just delete this object and don't worry about any other objects
- `nullify`: delete this object and nullify any references that other objects have backk to this object

As I mentioned, by default the delete rule is `nullify`. This is the safest. We only delete our object but we remove any references to it before doing so.

The other most useful deletion rule is `cascade` and we'll use that for `links`.

On the other hand, if you delete a `Presenter` you can delete all of their `links` as those `OnlineLink`s are assumed to be unique to that `Presenter`.

Here's how we do that:

*Conferences/Model/**Session.swift***

```
@Model
class Presenter {
  let name: String
  var sessions: [Session] = []

  @Relationship(deleteRule: .cascade,
                inverse: \OnlineLink.presenter)
  var links: [OnlineLink] = []

  init(name: String) {
    self.name = name
  }
}
```

I love how all of the information we need to know about relationships is right there in the code.

We have a class that represents a single `Session` but we need to manage a collection of `Session`s.

We need to be able to create, delete, and modify instances of `Session`.

SwiftData will do that for us. It will manage and persist our object graph.

These four `@Model` classes combine to give us our SwiftData model. You can think of it as our schema

Next, let's connect our model to our UI.