HACKING WITH SWIFT

PRIVATE WORKSHOP

Paul Hudson

Swift, Swift Data, and SwiftData (Part One)

Paul Hudson

Version: 2023-10-04

Contents

Chapter 1
The basics

Defining a model

A great first step when building any app is to define your data model. Previously this might have been done using a struct, but with SwiftData we need to use a class so that our data can be shared easily across our app.

Reminder: Each step of the way I'll show you an example of what to do in isolation, and you need to adapt that into your own project. Below is the example that explains the current concept, and after that you'll see "Over to you" – that's the part you need to adapt into your own project.

For example, you might start out with a file called User.swift containing this code:

```
class User {
  var name: String
  var age: Int

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

Important: Again, you don't need to copy this code anywhere; this is just showing you an example. Once you've read through this explanation, follow the "Over to you" section to apply what you learned to your app.

For me to make my class work with SwiftData takes exactly four steps:

1. Add **import SwiftData** to the top of my User.swift file.
2. Add the **@Model** macro before **class User**.
3. Add **import SwiftData** to the top of my **App** struct's file
4. Add this modifier to my **WindowGroup**: **.modelContainer(for: User.self)**

That’s it: those four changes, all of which are trivial, give us a complete SwiftData stack.

The **@Model** macro wraps the **@Observable** macro, but also adds the ability to read and write **User** objects with SwiftData.

The only really new part of this code is the **modelContainer()** modifier, which creates and registers the *model container* for the app with SwiftUI. Every app that uses SwiftData needs to have at least one of these model containers, and its job is to handle data storage on the device – it’s responsible for creating and managing the actual database file used for all SwiftData’s needs.

Just placing that single modifier in our project tells SwiftData to create our database when the app launches, and also create storage for **User** objects.

Over to you

In your Gusto project, please create a new SwiftData model called **Restaurant** that can store the following data:

- Name (String)
- Price rating (Int)
- Quality rating (Int)
- Speed rating (Int)

Once you have that, connect it to SwiftUI using a **modelContainer()** modifier in your app struct.

Creating data

Now that we have a data layer up and running, it's time to add some values. SwiftData classes are much simpler than Core Data classes, and can be created directly using the initializer you give them rather than needing to be treated specially.

For example, an instance of my **User** class can be created like this:

```
let taylor = User(name: "Taylor Swift", age: 26)
```

That creates the object, but doesn't do anything with it – it's just a piece of memory right now. If we actually want to store it in SwiftData we need to insert it into our database.

To do *that* means learning an important concept in SwiftData called the *model context*. If you've used Core Data this is identical to the **NSManagedObjectContext**, just with a much shorter name!

If you remember, the **modelContainer()** modifier creates the *model container* for the app. Every app that uses SwiftData needs to have at least one of these model containers, and its job is to handle data storage on the device – it's responsible for creating and managing the actual database file used for all SwiftData's needs.

But not everything needs to be on disk at once. In fact, if we were always reading and writing every piece of data we were using, performance would be terrible. Instead, SwiftData prefers us to read data from storage into memory then use it from there. We can then make a whole bunch of changes in one pass, and have them saved out to storage in one go – it's a lot more efficient.

This temporary memory storage is what SwiftData calls the model context, and it has the job of tracking all objects that have been created, modified, and deleted in memory, so they can all be saved to the model container later.

I already said that each SwiftData app needs to have at least one model container, created like this:

```
.modelContainer(for: User.self)
```

That *also* creates a model context for us called the *main context*, and it places that context into SwiftUI's environment for us to use. This main context always runs on Swift's main actor, so it's safe to use from our user interface.

We can read this back out from the environment using SwiftUI's **@Environment** property wrapper by adding code like this to any view where we need the context:

```
@Environment(\.modelContext) var modelContext
```

And now we can insert SwiftData objects into the model context by calling **modelContext.insert()**:

```
modelContext.insert(taylor)
```

Note: We don't ask SwiftData to save. Instead, it will do it for us, and does so very aggressively. This autosave behavior is enabled by default: SwiftData batches all changes together from the current runloop (between screen redraws) and saves them all it once, meaning that for the most part we don't need to worry about saving our data explicitly.

Over to you

Add a **NavigationStack** to your app, then add a toolbar there that adds the following five pieces of sample data:

- Wok this Way
- Thyme Square
- Pasta la Vista
- Life of Pie
- Lord of the Wings

You'll need to think up some example ratings for each of those restaurants, but it's just sample data so make them whatever you like. Remember to call **insert()** on your new data so it’s actually saved!

Note: Beyond your toolbar, we haven’t actually written any SwiftUI code to display our data, so don’t be surprised when you don’t see anything on the screen.

Showing data in SwiftUI

Once you've set up SwiftData and created a model, it's time to actually show it somehow. This takes two steps, both of which are simple:

- Adding a SwiftData **import** wherever you want to make queries.
- Creating a query that looks for the data you want.
- Showing the results of that query using whatever SwiftUI code you'd like.

The simplest query you'll want is "give me all the objects you have," which is done by adding a property like this:

```
@Query var users: [User]
```

That automatically reads the main model context from our environment, runs the query, and also *watches* the data for changes – it will automatically refresh our UI if we add, delete, or edit user objects.

@Query is a macro that works similarly to **@FetchRequest** in older Core Data code, and it has various alternate forms. For example, rows aren't sorted by default, but you can add sorting by specifying a key path like this:

```
@Query(sort: \User.name) var users: [User]
```

Or if you want a reverse sort, add the **order** parameter:

```
@Query(sort: \User.name, order: .reverse) var users: [User]
```

If you want more advanced sorting – i.e., sorting by two or more properties – you need to use a sort descriptor array, like this:

```
@Query(sort: [SortDescriptor(\User.name),
SortDescriptor(\User.age, order: .reverse)]) var users: [User]
```

That asks for users sorted alphabetically by name, with oldest coming first in the case of two users having the same name.

You can have as many sort descriptors as you need in that array, and SwiftData will work through them one by one. If there's an ambiguity in what remains – if a sort has taken place and two items have the same sort order – then SwiftData will default to the order they were created.

There are also options for controlling *filtering*, but we'll cover those later.

Once you have your query in place, the data is made available to you as a simple array – **[User]**, in our code so far. SwiftData objects automatically conform to **Identifiable**, along with **Hashable** and **Observable**, which means we can use them inside SwiftUI really easily.

For example, I might want to add a simple **List** to show all my data:

```
List(users) { user in
  Text("\(user.name) is \(user.age) years old")
}
```

Remember, **@Query** knows to update itself automatically as your data changes, which means we can pretty much forget about it once it's created.

Over to you

Now that you've seen how to query SwiftData information, please add an **@Query** property to your app then display the results somehow. We already have test data in place, so by the time this step is complete you should see all five example restaurants in your layout.

What you choose to show and how is up to you – we have the restaurant name, plus three pieces of rating data. This would be a good time to add some SF Symbols and similar, but it's your app so take whatever route you think works best.

Describing our data more

Every SwiftData object has a built-in **id** property that is guaranteed to be unique, but otherwise the system will happily store duplicate data by default.

To see this in action, try tapping your sample data toolbar button a few times – you should see the same five objects being created each time, again and again.

Sometimes this isn't what you want. Instead, you want to be able to say "this data needs to be unique," and have SwiftData enforce that rule for you.

Making a property unique is done using another macro called **@Attribute**, and it gives us the ability to customize how various properties in our models work. In our case, we can try applying the **.unique** attribute, telling SwiftData that we never want the name of a user to be used more than once.

For example, I might want to say that users in my database must always have a unique name, like this:

```
@Attribute(.unique) var name: String
```

But if you try that with your restaurant **name** property, what you'll see is that we *definitely* have unique restaurant names now, because they've all gone! All the duplicates, but all the originals too; all gone.

What's happening here is that SwiftData detects a risky change has happened to our data, and it has taken the only option it can guarantee doesn't leave our data in an inconsistent state: it simply refuses to load the model container.

SwiftData is smart here: if all our names were already unique then there would be no problem, and nothing would be destroyed – it would simply enforce the rule going forward. Similarly, if we were to delete a property from our data then SwiftData can just quietly remove it without bothering us further.

We'll look at how to handle these kinds of problems in the future, but for now you'll need to go to the Device menu and choose Erase All Content And Settings to clear the old data. Once that finishes, run the app again with the new configuration, and you'll find you can only add your sample data once.

The **@Attribute** macro has a few other options that are going to be useful:

- The **.allowsCloudEncryption** attribute stores this property in an encrypted format, *but only on iCloud*. It has no effect on local data.
- The **externalStorage** attribute tells SwiftData this property might work best stored in an external file next to your model container's file. This is designed to work with large data blobs, such as images or movies, and SwiftData automatically takes care of loading the external data no different from regular data. Note: This only a suggestion – SwiftData will decide.
- The **.preserveValueOnDeletion** tells SwiftData that when you delete this object it should be kept around in long-term storage, which is helpful when you need to keep records.
- The **.ephemeral** attribute means you don't want SwiftData to store this data – it's just local data you use while the program runs.
- The **.spotlight** attribute isn't useful and won't work.

Note: Apple said there should also be an **.indexed** attribute to mark certain properties as being indexed for faster searches, but that isn't currently available.

SwiftData intelligently applies the ephemeral attribute when data can't be stored, such as when we're using a computed property. For example, we could add an extra property to store a greeting for each user:

```
var greeting: String {
  "Hi, my name is \(name) and I am \(age) years old."
}
```

That's a computed property, so SwiftData is smart enough to realize it's not something it can store.

You can see this for yourself, and in fact I'd encourage it so you can see all the work macros are doing here – right-click on **@Model** and expand the macro to see how it applies the **@_PersistedProperty** macro to all our stored properties, but *not* the new computed property. You'll also see a **persistentBackingData** property that does the actual work of talking to Core Data on our behalf, alongside a **schemaMetadata()** that describes how the various properties map to their string names, along with any other metadata – including our uniqueness request.

If you scroll a little further down you'll also see that the **@Model** macro makes **User** conform to two protocols:

- The **PersistentModel** protocol verifies that your model is safe for SwiftData to use, but also adds methods for reading and writing from the backing data.
- The **Observable** protocol is what we already looked at, and ensures that our data can be watched for changes.

I'd recommend you also expand the **@_PersistedProperty** macro so you can see what's inside there too. What you'll see will immediately remind of Swift's observation: it tracks when a value is read or written to using **get** and **set**, but then it calls down to the backing data to read or write the underlying value.

Important: You should be very careful when using **@Attribute(.unique)** with any properties, because it is not supported when you want to store your user's data in iCloud (which is most of the time!), and can otherwise cause problems as you're leaving SwiftData to enforce rules on your behalf. If you really want some data to be unique, it's a better idea to handle it yourself.

Over to you

I'd like you to add a new property to your data: a computed property that provides an overall rating using a mean average of the three other ratings.

I would strongly recommend against adding a uniqueness constraint for the restaurant name. We'll be looking at this in more detail later on.

Deleting data

At this point our little app can create some sample data and display it on the screen. If we think about this from a CRUD perspective – Create, Read, Update, and Delete – we've managed some of the C because we have some sample data but no actual user data, and we've implemented the R part because we're reading data just fine.

To make this app really function, it needs to be able to accept custom data: users need to be able to add custom objects, edit the ones they have, and also delete any they no longer care about.

We're going to work through those, from easiest to hardest, so we can start to build out our app.

First, deleting. This is done by passing any object to **modelContext.delete()**. For example, we could use this:

```
modelContext.delete(taylor)
```

You can also safely delete multiple items from an **@Query** array inside a **for** loop, because the array will only be refreshed at the end of the runloop. For example, you could delete an **IndexSet** of items like this:

```
func deleteUsers(_ indexSet: IndexSet) {
  for item in indexSet {
    let object = users[item]
    modelContext.delete(object)
  }
}
```

Between those two options, you can implement swipe actions, context menus, or even the **onDelete()** modifier in SwiftUI.

Note: Just like adding data, deleting data will trigger a refresh of any **@Query** properties, and will also autosave your data.

Over to you

Upgrade your app to allow users to delete individual restaurants. Which approach you take is down to you!

Editing data

Our next task will be the U in CRUD: Updating existing data. Yes, we're tackling this before adding, because adding is trivial once updating works well.

The first part is easy: make a new SwiftUI view, then give it a property to store whatever data is being edited. For example, we might add this:

```
var user: User
```

You then need to provide a value for that in your preview, which means creating an **EditUserView** and passing in an example piece of data it can work with. You might try doing this:

```
#Preview {
  let example = User(name: "Taylor", age: 26)
  return EditUserView(user: example)
}
```

But that won't work: we can't create a **User** instance unless there's a model context available – just trying to create one kicks off a whole chain of SwiftData work, where it looks up our model type in its storage to make sure our code matches what's in storage, or creates the database information if it's not already there.

You might then try adding the **modelContainer(for:)** modifier like this:

```
#Preview {
  let example = User(name: "Taylor", age: 26)
  return EditUserView(user: example)
    .modelContainer(for: User.self)
}
```

…and *that* won't work either: again, literally just calling the **User** initializer is enough for SwiftData to realize there's no model container currently available – us adding it to a view afterwards is too late.

Instead, the best thing to do is configure the container by hand before making any model data at all.

This takes several steps, starting with a model configuration. We've been using the default configuration so far, but now we're trying to preview something we're going to use a custom configuration that reads and writes data purely to memory – it's all temporary data, so nothing is ever written to our actual database.

So, start buy adding an import for SwiftData, then put this code in the preview, before the other code:

```
let config = ModelConfiguration(isStoredInMemoryOnly: true)
```

Next, we need to make a **ModelContainer** instance for the types we want to work with, passing in that container:

```
let container = try! ModelContainer(for: User.self,
configurations: config)
```

Note: This is just for previewing purposes only, so using **try!** is fine here.

And *now* we're safe to create the user and view:

```
let example = User(name: "Taylor", age: 26)
return EditUserView(user: example)
  .modelContainer(container)
```

At last that will compile cleanly and preview too.

Tip: Navigating to your editing view is easy because SwiftData objects automatically conform to **Hashable** – use **NavigationLink(value: user)** to create the link, then use **.navigationDestination(for: User.self)** to display your editing view.

To get actual editing to work, you might try binding various properties to SwiftUI controls, like this:

```
Form {
  TextField("Name of user", text: $user.name)
}
```

…but that brings more errors: you can't bind to **$user** because it's just a plain old variable, so there's nothing to bind.

We hit this problem before in our little QRMe test project, and I explained that SwiftUI has a new property wrapper that lets us get access to a binding from any object that conforms to the **Observable** protocol. It's called **@Bindable**, and we need to use it here:

```
@Bindable var user: User
```

Remember, **@Bindable** only works with objects that conform to the **Observable** protocol, which means it works great here.

And that's all it takes to get editing working! SwiftData will automatically take care of applying the changes to its data store and saving.

Over to you

Create a new view to handle editing restaurants, using a text field for the restaurant name, plus pickers for the price, quality, and speed ratings. Once that's done, make tapping a restaurant in your original list bring up the editing view.

Adding data

Now we have editing working, *adding* data is trivial: we can simply insert a new object into your model context, then bring it up for editing immediately.

In fact, this doesn't require any SwiftData code at all – it's all pure SwiftUI, because you just need to bind an array of **Restaurant** objects to the path of your **NavigationStack**, then add your new restaurant there.

So, without any further ado it's…

Over to you

Add a new toolbar item that creates a new **Restaurant** object, adds it to your SwiftData context, then immediately presents it for editing. Don't remove the Add Samples button just yet; it will still be useful.

Once you've written the code, please try it out with some other sample restaurants of your own choosing, such as Bun Voyage, Pita the Great, or Tequila Mockingbird. (I make no apologies for these terrible restaurant puns.)