PRIVATE WORKSHOP

Paul Hudson

Swift, Swift Data, and SwiftData (Part Two)

Paul Hudson

Version: 2023-10-04

Contents

Chapter 1
Advanced topics

Dynamic sorting

When it comes to sorting our data, SwiftData has two approaches: the trivial version that works great in a WWDC video and a handful of small projects, and a more complex version that is much more indicative of the kinds of apps you'll be building in real life.

We've already seen the simple version, because it's where we can put our sort order directly into the **@Query** macro, like this:

```
@Query(sort: \.name)
```

Or this:

```
@Query(sort: \.name, order: .reverse)
```

And again, if you need to sort by multiple properties you can use a **SortDescriptor** array like this:

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

So, that's the simple way of sorting, when you get to decide up front how you want your data to be sorted.

In practice, however, that doesn't happen much – usually users want to be able to set the sort order dynamically, which is not actually supported by **@Query** right now. Hopefully that will change before release, but don't be afraid to file feedback with Apple!

In the meantime, to get dynamic sorting working you need to move your **@Query** properties down a view in SwiftUI's hierarchy – you need to put it into a subview where you can provide a sort value using dependency injection.

This takes a few steps just to get set up, most of which aren't unique to SwiftUI:

1. Making a new SwiftUI view to show the results of your **@Query**.
2. Adding **import SwiftData** to the top of its file.
3. Moving your **@Query** property into there.
4. Moving view code that uses your query results, for example your **ForEach** or your **List**.
5. Moving any methods that work with your query results into there too, such as methods that handle deletion.
6. Copying the **@Environment** property for **modelContext** into there. Note: This is copying rather than moving, because we need this in both places.
7. Updating your original **ContentView** to use the new view.

For example, we might create a **UserListingView** like this one:

```
import SwiftData
import SwiftUI

struct UserListingView: View {
  @Query var users: [User]
  @Environment(\.modelContext) var modelContext

  var body: some View {
    List {
      ForEach(users) { user in
        NavigationLink(value: user) {
          Text("\(user.name) is \(user.age) years old")
        }
      }
      .onDelete(perform: deleteUser)
    }
  }

  func deleteUser(_ indexSet: IndexSet) {
    for item in indexSet {
```

```
            let object = users[item]
            modelContext.delete(object)
        }
    }
}
```

And then back in **ContentView** we could create that inside a list such as this:

```
NavigationStack {
    UserListingView()
}
// etc
```

This change doesn't actually handle sorting – this is just the setup required to make sorting possible. However, because we now have a subview we're able to send values into there to control the **@Query** property wrapper.

This takes five steps in in total:

1. Telling the **UserListingView** that it needs to be created with some kind of sort order.
2. Updating its preview to pass in an example sort order.
3. Making some storage to hold whatever is the currently active sort order when your program is running.
4. Creating some UI to adjust that sort order based on the user's settings.
5. Passing that into **UserListingView** when it's created.

The only one of those steps that's new is the first one, where we need to create the query dynamically inside the subview's initializer. Here we're trying to change the **@Query** property wrapper itself rather than the array inside it, so as a result we need to access the underscored property name like this:

```
init(sort: SortDescriptor<User>) {
    _users = Query(sort: [sort])
```

```
}
```

Yes, it's quite a bit of hassle. It was initially hard with Core Data too, but they smoothed it out by exposing the sort options as a property we could adjust freely – hopefully that arrives in SwiftData soon.

Over to you

Please write code to enable dynamic sorting in your Gusto project, including adding some UI that lets the user choose to sort by name, pricing, quality, and speed.

Filtering data

As you've seen, sorting can be easy, or can be… well, a little *less* than easy, to put it mildly. But if you thought *that* was hard, let me introduce you to *filtering*, because this is where things get thoroughly tricky indeed.

Filtering is done with predicates: a test that can be applied to decide whether objects should appear in the resulting array or not. Back in Core Data this was done with a specialized class called **NSPredicate**, which allowed us to create filters using strings. It was pretty ugly, and also the kind of thing that was easy to screw up, but it worked.

When working with simple queries, SwiftData replaces **NSPredicate** with a new **#Predicate** macro that does something quite remarkable: it takes Swift code we write and converts it into SQL that the underlying database can understand. That process doesn't happen in one jump: behind the scenes our code gets converted to a series of predicate expressions that match what we're trying to do, which SwiftData then converts to SQL when the predicate actually runs.

This means, for the first time, that our predicates are *type-safe*: Swift checks our code at compile time to make sure our predicate actually makes sense, which wasn't possible with Core Data.

If you've ever worked with Core Data predicates you might be thinking this is almost too good to be true, because they were pretty notoriously hard to use and easy to get wrong. However, sadly it *is* a bit too good to be true, as you'll see.

First up, let's look at predicates in action. Let's say we wanted to show users that were children, which means they have an age less than 18. Here's how that looks:

```
init(sort: SortDescriptor<User>) {
  _users = Query(filter: #Predicate {
    $0.age < 18
  }, sort: [sort])
}
```

So, we give **#Predicate** a closure that takes each object in our database, and we can then apply our test to it. In this case, does the object have an age lower than 18?

Tip: Sometimes Swift will have a hard time typechecking your predicates, so you might need to use **#Predicate<User>** or similar.

Our code looks simple enough, but that **#Predicate** macro is doing a ton of work behind the scenes for us and later on I encourage you to right-click on it and choose "Expand Macro" to see exactly what happens.

What you'll see is that your Swift code is read by the macro, and converted into something else entirely. This is a monumental piece of work from the Swift team, but it's not magic – it can't take *any* Swift code and turn into into static predicates, because that simple wouldn't be possible.

In fact, the limit is lower than you might expect: **#Predicate** can deal with only one expression, no more. Even then, you can only use the subset of expressions that are supported by Swift: you can't use arbitrary function calls, for example; they just aren't supported. At this time there is no way to combine predicates together, which means if your code relies on **NSCompoundPredicate** I'm afraid you're out of luck – it just won't work.

Instead, you need to be able to collapse your entire filter down to a single expression, which might mean extensive use of **&&** and || to combine different things together, although because **if** can now be used as an expression you can at least nest conditions.

You'll also find some other common operations just aren't supported yet either, including things like **map()**, **endsWith()**, and the regular expression variants of string methods. Some of this will change over time, but it will still be far from perfect – there *are* limits to what even Apple can pull off here.

Over to you

I showed you how to send a sort order in – can you upgrade your app to handle a dynamic predicate so users can search for a restaurant by name? To do this means starting with the

searchable() modifier in **ContentView**, but be careful: make sure your predicate handles the case when the search string is empty!

Working with relationships

So far we've had a simple data model containing just one type of object, but SwiftData supports relationships too.

At its simplest level, we can define relationships simply by adding extra properties to our classes. For example, we might say that every user can have children, which means adding a single property to the **User** class:

```
var children: [User]
```

Then updating its initializer to accept the extra data:

```
init(name: String, age: Int, children: [User] = []) {
  self.name = name
  self.age = age
  self.children = children
}
```

…that's literally all it takes: SwiftData takes care of the rest for us.

When inserting linked data, SwiftData will automatically insert any relationships for you. For example, if you created a new parent user with two children, you'd only need to insert the parent, like this:

```
let sophie = User(name: "Sophie", age: 13, children: [])
let charlotte = User(name: "Charlotte", age: 10, children: [])
let paul = User(name: "Paul", 43: 26, children: [sophie,
charlotte])
modelContext.insert(example)
```

This works just as well if we want to link different types of data together. For example, we might want to define a **Pet** class and say that each user can have several pets. This means creating a new class to store the pet data:

```
@Model class Pet {
  var name: String
  var age: Int

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

Then adding a new property to the **User** class:

```
var pets: [Pet]
```

And finally updating the initializer to accept the extra data:

```
init(name: String, age: Int, children: [User] = [], pets: [Pet]
= []) {
  self.name = name
  self.age = age
  self.children = children
  self.pets = pets
}
```

This creates what's called a "one-to-many relationship", because one **User** object can have many pets. When SwiftData performs its queries, the **children** and **pets** properties are automatically available for us to use.

Tip: Right at the beginning you saw how we use **.modelContainer(for: User.self)** to initialize all SwiftData and prepare it to store users. Well, SwiftData is smart enough to track relationships here too – you can write **modelContainer(for: [User.self, Pet.self])** if you want, but you can also just leave it as **User.self** and SwiftData will automatically initialize the **Pet**

data too because of the relationship.

If you want, you can also add an inverse relationship – you can add a property to **Pet** that stores its owner, so we can read the relationship in both directions.

This requires another macro, **@Relationship**, and looks like this:

```
@Relationship(inverse: \User.pets) var owner: User
```

You would then add it to the initializer, like this:

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

SwiftData tracks this inverse for us automatically, so if we create a new **Pet** object with an owner, SwiftData will update the **pets** property of that **User** to include it.

Important: The **@Relationship** macro must be applied on only one side of the relationship – attempting to place it on both sides will cause a circular reference, and your code won't build.

This same approach works just as well with other relationships:

- For a one-to-one relationship, for example every user must have exactly one pet, and every pet must have exactly one owner, we'd use **var pet: Pet** in the **User** class, and **var owner: User** in the **User** class. You'd need to make one of them optional otherwise it would be impossible to create one without first creating the other.
- For a many-to-many relationship, for example every user can have several children, and every child can have several parents, we'd specify arrays of **Pet** and **User** respectively.

Before you get on to adding a relationship to your Gusto project, there's one more thing to learn: handling relationships when objects are deleted.

If you add some users then add a bunch of pets for each user, what happens when you *delete* a user? SwiftData does exactly the same thing as Core Data here by default: it removes the user, but leaves all the pets intact.

You *might* want that, but you might also want what's called a *cascade deletion*: removing a user, should also remove all their pets. This is specified in the **@Relationship** macro, but you need to apply it to the parent object – the **User**, in our case.

So, we'd delete the **@Relationship** macro from the **Pet** class (it can only appear on one side of the relationship, remember!), then modify the **User** class so its **pets** property looks like this::

```
@Relationship(deleteRule: .cascade, inverse: \Pet.owner)
var pets: [Pet]
```

And now we get the desired result: deleting a **User** object also deletes all pets that belonged to it.

Over to you

It's time for you to put relationships into practice in Gusto: upgrade your app so that users can add their favorite dishes to each restaurant, specifying both the dish name and a short review.

Tip: Just give your **Dish** model the properties **name** and **review** – there's no need to declare the inverse relationship.

If you want to deepen your understanding, here's a small diversion…

Note: This requires a little bit of knowledge of SQLite, which is the underlying data store used by SwiftData.

Earlier we used an **@Model** class to track pets belonging to users, but we can also use structs because SwiftData is capable of handling any kind of **Codable** data automatically, including enums.

So, we can either create a **Pet** struct like this:

```
struct Pet: Codable {
  var name: String
}
```

Or a **Pet** class like this:

```
@Model class Pet {
  var name: String

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

Important: Those two code samples might look similar, but they yield very different results.

Regardless of which you choose, we would then modify the **User** class to add an array of pets, like this:

```
var pets: [Pet]
```

How they differ is quite brilliant, and really reflects the Swifty nature of SwiftData:

- Structs are value types, so SwiftData converts our struct into JSON data, then writes it directly into the SQLite database row for the user, alongside their name and age.
- Classes are reference types, so SwiftData creates a separate SQLite table for pets, then links data from there back to users.

If you're working with individual values – if for example you said that each user has exactly one pet rather than an array – then SwiftData collapses the structs down so they are literally just attributes of the user directly.

So:

- If you have an array of **Pet** structs, that array is stored as JSON alongside the **User** data.
- If you have an array of **Pet** classes, that array is stored separately and linked back to the **User**.
- If you have a single **Pet** struct, the properties of that struct get written directly alongside the properties of the **User**.
- If you have a single **Pet** class, it's still stored separately and linked back to the **User**.

This setup exactly reflects how structs and classes work in Swift: structs have unique owners so copying them directly into the owning **User** makes sense, whereas classes can be referenced in multiple places, so having them stored separately allows them to be linked to multiple owners.

When you're experimenting with structs and classes, it's useful to be able to see exactly how SwiftData stashes away our information.

I want to show you a really important debugging technique for just this purpose: looking at SwiftData's internal storage. Remember, SwiftData is just sitting on top of a big SQLite database, so if at any point you're wondering if there's a bug in your code or an error in your

data, you can just go and look – you can open the database and poke around in there.

To try this out:

1. Go to the Product menu.
2. Hold down the Option key.
3. Click “Run…”
4. Go to the Arguments tab.
5. Click + under Arguments Passed on Launch.
6. Add this in the text field: -com.apple.CoreData.SQLDebug 1
7. Now click Run to launch your app.

Note: That launch argument is in place from now on, and because it’s a little hard to remember you should just uncheck the box next to it to turn it off without removing it entirely.

When you run the app you’ll see a huge amount of logging information appear in Xcode’s console, all of which will say “CoreData” at the start. So, if you had any lingering questions on whether SwiftData is an overlay over Core Data, hopefully they are now fully dispelled!

What these logs are doing is telling us exactly what SwiftData is doing behind the scenes – creating databases, reading data, creating data, handling autosaves, and more.

Inside there you’ll see other things, including:

- It has loaded the underlying database storage file where our data is actually stored.
- It enables “wal" mode (write-ahead logging) so that our data is written safely
- It has registered for remote change notifications so that it’s able to watch for changes from iCloud.

Tip: A lot of messages will only appear the first time you run the app, so if you want to see the full set of logs make sure you erase the simulator device first.

A lot of these log entries will be SQL (Structured Query Language), which is the language used by many databases to add, remove, and search for data. SQL is a large and complex

language in its own right, but you don't need to know much to be able to use it well enough.

Like I said, a great way to see what's really happening is to inspect the data directly. To do that, scroll to the very top of Xcode's console, and look for the line that starts with "CoreData: annotation: Connecting to sqlite database file at…". That's where your database file lives.

I'd like you to copy the entire path to your clipboard, including quotes: "/users/yourusername/Library/etc/etc/etc/default.store". Now open the Terminal on your Mac, enter "sqlite3" then a space, then press paste. When you hit return, macOS will launch the SQLite 3 terminal app, which allows us to interact directly with database files created by both SwiftData and Core Data.

Once you're in there, type ".sch" and press return. SQLite will (correctly) interpret that as "show me the schema for my data," which means "show me the structure of all the data".

You'll see a fair amount of output, because SwiftData does a lot of work for us behind the scenes. But inside there you'll see code like this:

CREATE TABLE ZRESTAURANT (Z_PK INTEGER PRIMARY KEY, Z_ENT INTEGER, Z_OPT INTEGER, ZPRICERATING INTEGER, ZQUALITYRATING INTEGER, ZRATING INTEGER, ZSPEEDRATING INTEGER);

That's our **Restaurant** class, converted to SQL. You can see the name, price rating, quality rating, and more.

If you'd like to see the data inside the table, run this command: **SELECT * FROM ZRESTAURANT;** – including the semicolon at the end. Although you can in theory create, update, and delete data right here in SQLite, I wouldn't recommend it; leave that to SwiftData.

Handling migration

A few times so far we have cheated, or at least taken a huge shortcut: when we've made significant changes to our data, such as making a **User** name property be unique, I've told you to avoid problems by simply erasing all the contents and settings of your simulator. This blanks SwiftData's storage, and so means our code carries in working.

Clearly this isn't a workable solution beyond a tutorial, so we need a way to be able to change our data over time. SwiftData provides such a way called *migrations*, and these are similar if somewhat easier than the Core Data equivalent.

To demonstrate migrations, let's return to our earlier problem: making the name of some data unique. If you remember, in our **User** example this worked fine if all our names were already unique, but if they weren't – if there were two users or two restaurants with the same name and we tried to make the **name** property unique – then SwiftData would simply refuse to load the container.

Fixing this requires four steps:

1. We need to define multiple versions of our data model.
2. We wrap each of those versions inside an enum that conforms to the **VersionedSchema** protocol. (It's an enum only because we won't actually be instantiating these directly.)
3. We create another enum that conforms to the **SchemaMigrationPlan** protocol, which is where we handle the migrations between each model version.
4. We then create a custom **ModelContainer** configuration that knows to use the migration plan as needed.

If we strip the **User** example right back to its original, simpler form, then place it inside a **VersionedSchema** enum, we get this:

```
enum UsersSchemaV1: VersionedSchema {
  static var versionIdentifier = Schema.Version(1, 0, 0)
```

```
  static var models: [any PersistentModel.Type] {
    [User.self]
  }

  @Model
  class User {
    var name: String
    var age: Int

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

Notice how that places the **User** class inside the schema, which means it's neatly namespaced. There are also two important static properties there:

- The exact version of this schema, which for this simple version is 1.0.0. Treating this as semver is a good idea.
- The array of model types included in this migration, which for us is just a single value: **User.self**.

We'd then make a second versioned schema for v2, with the uniqueness constraint applied:

```
enum UsersSchemaV2: VersionedSchema {
  static var versionIdentifier = Schema.Version(2, 0, 0)

  static var models: [any PersistentModel.Type] {
    [User.self]
  }
```

```
  @Model
  class User {
    @Attribute(.unique) var name: String
    var age: Int

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

That's basically identical, with the only changes being the 2.0.0 version and **@Attribute(.unique)**.

With those two in place, we can delete the original, un-namespaced `User` class, leaving only the two nested classes in place.

Obviously we don't want to have to constantly change our code to refer to **UsersSchemaV2.User** and so on, so you should add a typealias so that **User** always points to whatever is your latest version:

```
typealias User = UsersSchemaV2.User
```

That completes the first two steps of our process, so the third step is to create a migration plan. This is defined as at least three properties:

1. An array of all the versioned schemas you want to use.
2. One or more **MigrationStage** instances, defining how to move from one specific version to another.
3. A **stages** array that lists all the migration stages you have defined.

Once you've done that, SwiftData will take care of all possible migrations for you – if your

user has v4 installed and needs to migrate straight to v5, or if they have v1 installed and need to migrate through v2, v3, v4, and then onto v5.

In our case, we'd start out with an enum like this:

```
enum UsersMigrationPlan: SchemaMigrationPlan {
}
```

We'd add to it an array of the two versioned schema we defined earlier:

```
static var schemas: [any VersionedSchema.Type] {
  [UsersSchemaV1.self, UsersSchemaV2.self]
}
```

Then we'd add one migration stage. This can be one of two types:

- A lightweight migration is something SwiftData can do on your behalf, e.g. deleting a property from a model class. I'll show you an example of this after we complete this uniqueness example.
- A custom migration, otherwise known as a *heavyweight* migration, where you need to handle the conversion yourself somehow.

We already saw that SwiftData can't handle the migration automatically when making a property unique after it has duplicates, so we need a custom migration here. That means adding a property such as this one to the enum:

```
static let migrateV1toV2 = MigrationStage.custom(
  fromVersion: UsersSchemaV1.self,
  toVersion: UsersSchemaV2.self,
  willMigrate: { context in
    // remove duplicates then save
  }, didMigrate: nil
)
```

We'll replace that comment in a moment, but first we need to finish the enum by listing all the migration stages we've defined. That's just one here, but in a real app you're likely to have several, probably mixing lightweight and custom stages as needed.

Add this to the enum now:

```
static var stages: [MigrationStage] {
  [migrateV1toV2]
}
```

What remains is filling in the // **remove duplicates then save** comment with some actual code. In this situation we'll be given the current model context filled with data, and it's our job to clean it up somehow before triggering a save manually.

In this instance, our goal is to make sure that the **name** property of our users is unique, which means deleting keeping one instance of each name and deleting the rest, like this:

```
let users = try
context.fetch(FetchDescriptor<UsersSchemaV1.User>())

var usedNames = Set<String>()

for user in users {
  if usedNames.contains(user.name) {
    context.delete(user)
  }

  usedNames.insert(user.name)
}

try context.save()
```

Now that we have a migration plan in place, the final step is to create a custom **ModelContainer** configuration that knows to use the migration plan as needed.

This means going to your main **App** struct and giving it a new property to store a custom-configured model container:

```
let container: ModelContainer
```

We need to create this by hand, telling it to load the latest version of our **User** model (thanks to our type alias!), and also specifying the migration plan so it knows how to upgrade data. So, we need to give our **App** struct an initializer such as this one:

```
init() {
  do {
    container = try ModelContainer(
      for: User.self,
      migrationPlan: UsersMigrationPlan.self
    )
  } catch {
    fatalError("Failed to initialize model container.")
  }
}
```

Last but not least, we need to adjust the **modelContainer()** modifier so that we pass in the **container** property we just configured rather than asking it to set up all the data from scratch:

```
.modelContainer(container)
```

I know it's a lot of work, but I hope you can see the importance of getting migration right – we're being very clear on exactly how SwiftData should move between various versions of our data, thus hopefully ensuring user data never gets lost.

On the flip side, *lightweight* migrations are trivial. Like I said, SwiftData can make a variety of

small changes automatically – if you delete a property from a model, or if you add a new relationship that has an empty array by default, it will just work.

You can also rename properties if you need to, although here you should make sure and tell SwiftData the old name so it can migrate smoothly. For example, if we renamed **age** to **currentAge** we'd use this:

```
@Attribute(originalName: "age") var currentAge: Int
```

Over to you…

Please try making the restaurant name property unique. Again, this isn't compatible with iCloud, but it will work just fine with local data – as long as you create the right migration path!

Undoing changes

SwiftData comes with baked-in support for both undo and redo, and it works in exactly the same way as regular SwiftUI undo and redo.

However, it's disabled by default, so the first thing to do is adjust the wya you you create your model container to enable undo.

For example, I'd write this:

```
.modelContainer(for: [User.self], isUndoEnabled: true)
```

Then ext step is to get access to SwiftUI's undo manager from the environment, like this:

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

And now you can trigger undo and redo whenever you want:

```
undoManager?.undo()
undoManager?.redo()
```

Important: I've found that undo works perfectly every time, correctly undoing property changes or removing objects and relations that were inserted. However, I've found *redo* more flaky: it seems happy to redo simple property changes (changing someone's name, for example), but struggles when redoing changes that involved objects with relations being deleted – I've had it crash several times. Tread carefully!

Over to you

Add undo support to your app, including undo/redo icons while the user is editing.