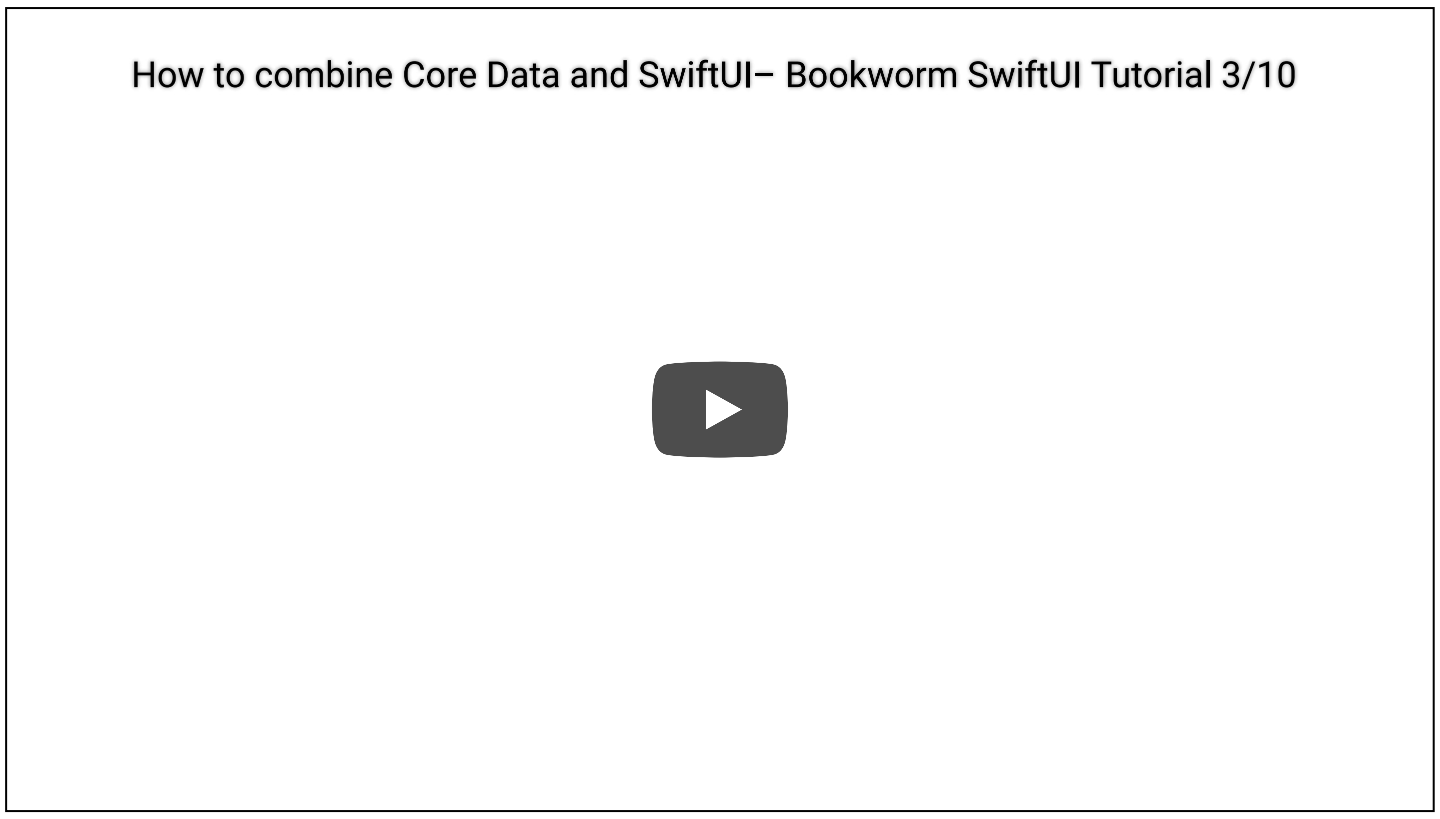
< Accepting multi-line text input with TextEditor

Creating books with Core Data >

# How to combine Core Data and SwiftUI

Paul Hudson @twostraws May 12th 2022


How to combine Core Data and SwiftUI– Bookworm SwiftUI Tutorial 3/10

SwiftUI and Core Data were introduced almost exactly a decade apart – SwiftUI with iOS 13, and Core Data with iPhoneOS 3; so long ago it wasn't even called iOS because the iPad wasn't released yet. Despite their distance in time, Apple put in a ton of work to make sure these two powerhouse technologies work beautifully alongside each other, meaning that Core Data integrates into SwiftUI as if it were always designed that way.

First, the basics: Core Data is an object graph and persistence framework, which is a fancy way of saying it lets us define objects and properties of those objects, then lets us read and write them from permanent storage.

On the surface this sounds like using `Codable` and `UserDefaults`, but it's much more advanced than that: Core Data is capable of sorting and filtering of our data, and can work with much larger data – there's effectively no limit to how much data it can store. Even better, Core Data implements all sorts of more advanced functionality for when you really need to lean on it: data validation, lazy loading of data, undo and redo, and much more.

In this project we're going to be using only a small amount of Core Data's power, but that will expand soon enough – I just want to give you a taste of it at first.

When you created your Xcode project I asked you to *not* check the Use Core Data box, because although it gets some of the boring set up code out of the way it also adds a whole bunch of extra example code that is just pointless and just needs to be deleted.

So, instead you're going to learn how to set up Core Data by hand. It takes three steps, starting with us defining the data we want to use in our app.

Previously we described data like this:

```
struct Student {
    var id: UUID
```

```
}
```

However, Core Data doesn't work like that. You see, Core Data needs to know ahead of time what all our data types look like, what it contains, and how it relates to each other.

This is all contained in a new file type called Data Model, which has the file extension "xcdatamodeld". Let's create one now: press Cmd+N to make a new file, select Data Model from the list of templates, then name your model Bookworm.xcdatamodeld.

When you press Create, Xcode will open the new file in its data model editor. Here we define our types as "entities", then create properties in there as "attributes" – Core Data is responsible for converting that into an actual database layout it can work with at runtime.

For trial purposes, please press the Add Entity button to create a new entity, then double click on its name to rename it "Student". Next, click the + button directly below the Attributes table to add two attributes: "id" as a UUID and "name" as a string.

That tells Core Data everything we need to know to create students and save them, so we can proceed to the second step of setting up Core Data: writing a little Swift code to load that model and prepare it for us to use.

We're going to write this in a few small pieces, so I can explain what's happening in detail. First, create a new Swift file called DataController.swift, and add this just above its `import Foundation` line:

```
import CoreData
```

We're going to start by creating a new class called `DataController`, making it conform to `ObservableObject` so that we can use it with the `@StateObject` property wrapper – we want to create one of these when our app launches, then keep it alive for as long as our app runs.

Inside this class we'll add a single property of the type `NSPersistentContainer`, which is the Core Data type responsible for loading a data model and giving us access to the data inside. From a modern point of view this sounds strange, but the "NS" part is short for "NeXTSTEP", which was a huge operating system that Apple acquired when they brought Steve Jobs back into the fold in 1997 – Core Data has some really old foundations!

Anyway, start by adding this to your file:

```
class DataController: ObservableObject {
    let container = NSPersistentContainer(name: "Bookworm")
}
```

That tells Core Data we want to use the Bookworm data model. It doesn't actually *load* it – we'll do that in a moment – but it *does* prepare Core Data to load it. Data models don't contain our actual data, just the definitions of properties and attributes like we defined a moment ago.

To actually load the data model we need to call `loadPersistentStores()` on our container, which tells Core Data to access our saved data according to the data model in Bookworm.xcdatamodeld. This doesn't load all the data into memory at the same time, because that would be wasteful, but at least Core Data can see all the information we have.

It's entirely possible that loading the saved data might go wrong, maybe the data is

the only meaningful thing you can do at this point is show an error message to the user, and hope that relaunching the app clears up the problem.

Anyway, we're going to write a small initializer for `DataController` that loads our stored data immediately. If things go wrong – unlikely, but not impossible – we'll print a message to the Xcode debug log.

Add this initializer to `DataController` now:

```
init() {
    container.loadPersistentStores { description, error in
        if let error = error {
            print("Core Data failed to load: \(error.localizedDe
        }
    }
}
```

That completes `DataController`, so the final step is to create an instance of `DataController` and send it into SwiftUI's environment. You've already met `@Environment` when it came to asking SwiftUI to dismiss our view, but it also stores other useful data such as our time zone, user interface appearance, and more.

This is relevant to Core Data because most apps work with only one Core Data store at a time, so rather than every view trying to create their own store individually we instead create it once when our app starts, then store it inside the SwiftUI environment so everywhere else in our app can use it.

To do this, open BookwormApp.swift, and add this property this to the struct:

```
@StateObject private var dataController = DataController()
```

That creates our data controller, and now we can place it into SwiftUI's environment by adding a new modifier to the `ContentView()` line:

```
WindowGroup {
    ContentView()
        .environment(\.managedObjectContext, dataController.cont
}
```

**Tip:** If you're using Xcode's SwiftUI previews, you should also inject a managed object context into your preview struct for `ContentView`.

You've already met data models, which store definitions of the entities and attributes we want to use, and `NSPersistentStoreContainer`, which handles loading the actual data we have saved to the user's device. Well, you just met the third piece of the Core Data puzzle: managed object contexts. These are effectively the "live" version of your data – when you load objects and change them, those changes only exist in memory until you specifically save them back to the persistent store.

So, the job of the view context is to let us work with all our data in memory, which is much faster than constantly reading and writing data to disk. When we're ready we still do need to write changes out to persistent store if we want them to be there when our app runs next, but you can also choose to discard changes if you don't want them.

At this point we've created our Core Data model, we've loaded it, and we've prepared it for

writing it too.

Retrieving information from Core Data is done using a *fetch request* – we describe what we want, how it should sorted, and whether any filters should be used, and Core Data sends back all the matching data. We need to make sure that this fetch request stays up to date over time, so that as students are created or removed our UI stays synchronized.

SwiftUI has a solution for this, and – you guessed it – it's another property wrapper. This time it's called `@FetchRequest` and it takes at least one parameter describing how we want the results to be sorted. It has quite a specific format, so let's start by adding a fetch request for our students – please add this property to `ContentView` now:

```
@FetchRequest(sortDescriptors: []) var students: FetchedResults<
```

Broken down, that creates a fetch request with no sorting, and places it into a property called `students` that has the the type `FetchedResults<Student>`.

From there, we can start using `students` like a regular Swift array, but there's one catch as you'll see. First, some code that puts the array into a `List`:

```
VStack {
    List(students) { student in
        Text(student.name ?? "Unknown")
    }
}
```

Did you spot the catch? Yes, `student.name` is an optional – it might have a value or it might not. This is one area of Core Data that will annoy you greatly: it has the concept of optional data, but it's an entirely different concept to Swift's optionals. If we say to Core Data "this thing can't be optional" (which you can do inside the model editor), it will *still* generate optional Swift properties, because all Core Data cares about is that the properties have values when they are saved – they can be nil at other times.

You can run the code if you want to, but there isn't really much point – the list will be empty because we haven't added any data yet, so our database is empty. To fix that we're going to create a button below our list that adds a new random student every time it's tapped, but first we need a new property to access the managed object context we created earlier.

Let me back up a little, because this matters. When we defined the "Student" entity, what actually happened was that Core Data created a class for us that inherits from one of its own classes: `NSManagedObject`. We can't see this class in our code, because it's generated automatically when we build our project, just like Core ML's models. These objects are called *managed* because Core Data is looking after them: it loads them from the persistent container and writes their changes back too.

All our managed objects live inside a *managed object context*, one of which we created earlier. Placing it into the SwiftUI environment meant that it was automatically used for the `@FetchRequest` property wrapper – it uses whatever managed object context is available in the environment.

Anyway, when it comes to adding and saving objects, we need access to the managed object context that it is in SwiftUI's environment. This is another use for the `@Environment` property wrapper – we can ask it for the current managed object context, and assign it to a property for our use.

So, add this property to `ContentView` now:

```
@Environment(\.managedObjectContext) var moc
```

With that in place, the next step is add a button that generates random students and saves them in the managed object context. To help the students stand out, we'll assign random names by creating `firstNames` and `lastNames` arrays, then using `randomElement()` to pick one of each.

Start by adding this button just below the `List`:

```
Button("Add") {
    let firstNames = ["Ginny", "Harry", "Hermione", "Luna", "Ron
    let lastNames = ["Granger", "Lovegood", "Potter", "Weasley"]

    let chosenFirstName = firstNames.randomElement()!
    let chosenLastName = lastNames.randomElement()!

    // more code to come
}
```

**Note:** Inevitably there are people that will complain about me force unwrapping those calls to `randomElement()`, but we literally just hand-created the arrays to have values – it will always succeed. If you desperately hate force unwraps, perhaps replace them with nil coalescing and default values.

Now for the interesting part: we're going to create a `Student` object, using the class Core Data generated for us. This needs to be attached to a managed object context, so the object knows where it should be stored. We can then assign values to it just like we normally would for a struct.

So, add these three lines to the button's action closure now:

```
let student = Student(context: moc)
student.id = UUID()
student.name = "\(chosenFirstName) \(chosenLastName)"
```

Finally we need to ask our managed object context to save itself, which means it will write its changes to the persistent store. This is a throwing function call, because in theory it might fail. In practice, nothing about what we've done has any chance of failing, so we can call this using `try?` – we don't care about catching errors.

So, add this final line to the button's action:

```
try? moc.save()
```

At last, you should now be able to run the app and try it out – click the Add button a few times to generate some random students, and you should see them slide somewhere into our list. Even better, if you relaunch the app you'll find your students are still there, because Core Data saved them.

Now, you might think this was an awful lot of learning for not a lot of result, but you now know what persistent stores and managed object contexts are, what entities and attributes

too. We'll be looking at Core Data more later on in this project, as well in the future, but for now you've come far.

This was the last part of the overview for this project, but this time I don't want you to reset your project fully. Yes, put ContentView.swift back to its original state, then delete the Student entity from Bookworm.xcdatamodeld, **but please leave BookwormApp.swift and DataController.swift alone** – we'll be using them in the real project!

< Accepting multi-line text input with TextEditor

Creating books with Core Data >

Table of Contents