Persistence and concurrency

Persisting data

Users can now create and edit scrums, and because you use state and bindings to pass data between views, SwiftUI automatically keeps the app's user interface up to date. However, quitting and relaunching Scrumdinger resets all data back to its initial state.

In this tutorial, you'll update Scrumdinger to support persistence, an essential feature of most apps. You'll add `Codable` conformance for the app's models and write methods to load and save scrums.

Download the starter project and follow along with this tutorial, or open the finished project and explore the code on your own.

30min
Estimated Time

Project files

Xcode 14

Section 1

Add codable conformance

In this section, you'll add `Codable` conformance to Scrumdinger's models.

`Codable` is a type alias that combines the `Encodable` and `Decodable` protocols. When you implement these protocols on your types, you can use the `Codable` API to easily serialize data to and from JSON.

Many types in the standard library and Foundation, like `UUID`, `Date`, and `Int`, are already `Codable`. You can adopt `Codable` in your own custom type by using types that are already `Codable` for all of its stored properties and declaring the type `Codable`.

```
import Foundation

struct DailyScrum: Identifiable, Codable {
    let id: UUID
    var title: String
    var attendees: [Attendee]
    var lengthInMinutes: Int
    var theme: Theme
    var history: [History] = []

    init(id: UUID = UUID(), title: String, attendees: [String],
         lengthInMinutes: Int, theme: Theme) {
        self.id = id
        self.title = title
        self.attendees = attendees.map { Attendee(name: $0) }
        self.lengthInMinutes = lengthInMinutes
        self.theme = theme
    }
}
```

Step 1

Open Models > `Theme.swift`, and declare conformance to the `Codable` protocol.

You gain automatic conformance because `Theme` stores raw `String` values, which are already `Codable`.

Step 2

Open Models > `History.swift`, and declare conformance to the `Codable` protocol.

Note

Xcode displays a compiler error because `attendees` isn't codable. You'll fix this error in the next step.

Step 3

Open Models > `DailyScrum.swift`, and add `Codable` conformance to the `Attendee` inner structure.

Now that all of its properties are codable, you can make `DailyScrum` codable with only a declaration.

Step 4

Add conformance to the `Codable` protocol.

```
    init(id: UUID = UUID(), title: String, attendees: [
        self.id = id
        self.title = title
        self.attendees = attendees.map { Attendee(name:
        self.lengthInMinutes = lengthInMinutes
        self.theme = theme
    }
}

extension DailyScrum {
```

Section 2

Create a data store

You'll create a new class named `ScrumStore` to man the scrum data for your app. `ScrumStore` has a sing property that holds the array of `DailyScrum` structu that the app displays. Because the property is markec `@Published`, views can bind to the value. As you lea in Passing data with bindings, using a binding ensures SwiftUI updates the view to reflect the latest value.

```
    struct Attendee: Identifiable, Codable {
        let id: UUID
        var name: String

        init(id: UUID = UUID(), name: String) {
            self.id = id
            self.name = name
        }
    }

    static var emptyScrum: DailyScrum {
        DailyScrum(title: "", attendees: [], lengthInMi
    }
}

extension DailyScrum {
    static let sampleData: [DailyScrum] =
    [
        DailyScrum(title: "Design",
                   attendees: ["Cathy", "Daisy", "Simor
                   lengthInMinutes: 10,
                   theme: .yellow),
        DailyScrum(title: "App Dev",
                   attendees: ["Katie", "Gray", "Euna",
                   lengthInMinutes: 5,
                   theme: .orange),
        DailyScrum(title: "Web Dev",
                   attendees: ["Chella", "Chris", "Chri
                   lengthInMinutes: 5,
                   theme: .poppy)
    ]
}
```

Step 1

In the Models group, create a new Swift file named `ScrumStore.swift`.

Step 2

Import SwiftUI, and then create a `ScrumStore` class that conforms to `ObservableObject`.

`ObservableObject` is a class-constrained protocol for connecting external model data to SwiftUI views.

Step 3

Add a `@Published scrums` property of type `[DailyScrum]`.

An `ObservableObject` includes an `objectWillChange` publisher that emits when one of its `@Published` properties is about to change. Any view observing an instance of

`ScrumStore` will render again when the `scrums` value changes.

Scrumdinger will load and save scrums to a file in the user's Documents folder. You'll add a function that makes accessing that file more convenient.

Step 4

Add a private static throwing function named `fileURL` that returns a URL.

Note

Xcode displays a compiler error until you return a URL in the next step.

Step 5

Call `url(for:in:appropriateFor:create:)` on the default file manager.

You use the shared instance of the `FileManager` class to get the location of the Documents directory for the current user.

Step 6

Call `appendingPathComponent(_:)` to return the URL of a file named `scrums.data`.

ScrumStore.swift

No Preview

```
1  import SwiftUI
2
3  class ScrumStore: ObservableObject {
4      @Published var scrums: [DailyScrum] = []
5
6      private static func fileURL() throws -> URL {
7          try FileManager.default.url(for: .documentDirec
8                                      in: .userDomainMasl
9                                      appropriateFor: nil
10                                     create: false)
11         .appendingPathComponent("scrums.data")
12     }
13 }
```

Section 3

Add a method to load data

In this section, you'll add a method to read JSON data from the `scrums.data` file and decode the data to a array of daily scrums.

Reading from the file system can be slow. To keep the interface responsive, you'll write an asynchronous fur to load data from the file system. Making the function asynchronous lets the system efficiently prioritize updating the user interface instead of sitting idle and waiting while the file system reads data.

Step 1

Declare an asynchronous function named `load`.

If an asynchronous function also throws an error, the `async` keyword comes before the `throws` keyword.

Step 2

Create a `Task`.

You store the task in a `let` constant so that later you can access values returned or catch errors thrown from the task.

Step 3

Add generic parameters to the task.

The parameters tell the compiler that your closure returns `[DailyScrum]` and can throw

an `Error`.

Note

Xcode displays a compiler error because of the missing return.

Step 4

Inside the task closure, create a local constant for the file URL.

Step 5

Use a `guard` statement to optionally load the file data.

Because the `scrums.data` file doesn't exist when the app opens for the first time, you return an empty array if there's an error reading the file.

Step 6

Decode the data into a local constant named `dailyScrums`.

Step 7

Return the decoded array of scrums.

The value returned from the task closure is available when the task completes.

Note

Returning the array fixes the compiler error.

Step 8

Use `try await` to wait for the task to finish, and assign the value to a constant named

`scrums`.

If the `JSONDecoder` throws an error inside the task, the error will be propagated when you try to access the `value` property.

Step 9

Add a `@MainActor` annotation to `ScrumStore`.

The class must be marked as `@MainActor` before it is safe to update the published `scrums` property from the asynchronous `load()` method.

Step 10

Assign the scrums to the `scrums` property.

The type of `task.value` is the type that you defined in the task initializer: `[DailyScrum]`.

ScrumStore.swift

No Preview

```
import SwiftUI

class ScrumStore: ObservableObject {
    @Published var scrums: [DailyScrum] = []

    private static func fileURL() throws -> URL {
        try FileManager.default.url(for: .documentDirec
                                    in: .userDomainMasl
                                    appropriateFor: nil
                                    create: false)
        .appendingPathComponent("scrums.data")
    }

    func load() async throws {
        let task = Task<[DailyScrum], Error> {
            let fileURL = try Self.fileURL()
            guard let data = try? Data(contentsOf: file
                return []
            }
            let dailyScrums = try JSONDecoder().decode(
        }
    }
}

```

Section 4

Add a method to save data

You'll write another method to save the user's scrums the file system. The pattern you'll use to save scrums be similar to the pattern you used to load scrums. The difference is that you won't need to handle a return value from the write operation, but you'll want to handle errors from saving.

Step 1

Add a `save` method at the bottom of the file.

Encoding scrums can fail, and you'll need to handle any errors that occur.

Step 2

Create a `Task`.

Step 3

Encode the scrums data.

Step 4

Create a constant for the file URL.

ScrumStore.swift

No Preview

```
import SwiftUI

class ScrumStore: ObservableObject {
    @Published var scrums: [DailyScrum] = []

    private static func fileURL() throws -> URL {
        try FileManager.default.url(for: .documentDirec
                                    in: .userDomainMasl
                                    appropriateFor: nil
                                    create: false)
        .appendingPathComponent("scrums.data")
    }

    func load() async throws {
        let task = Task<[DailyScrum], Error> {
            let fileURL = try Self.fileURL()
            guard let data = try? Data(contentsOf: file
                return []
            }
            let dailyScrums = try JSONDecoder().decode(
            return dailyScrums
        }
        let scrums = try await task.value
        self.scrums = scrums
    }

    func save(scrums: [DailyScrum]) async throws {
        let task = Task {
            let data = try JSONEncoder().encode(scrums)
            let outfile = try Self.fileURL()
            try data.write(to: outfile)
        }
        _ = try await task.value
    }
}
```

Step 5

Write the encoded data to the file.

Step 6

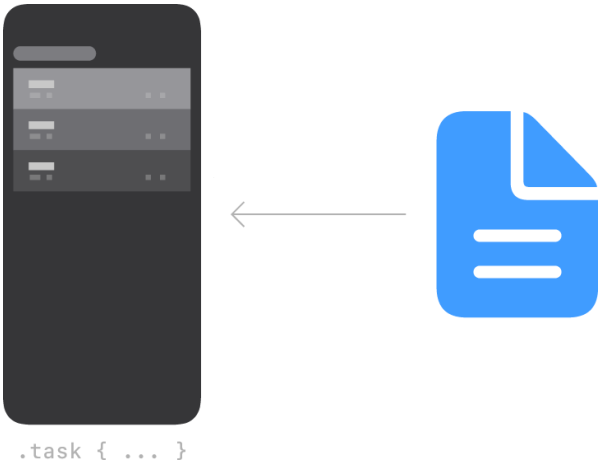
Wait for the task to complete.

Waiting for the task ensures that any error thrown inside the task will be reported to the caller. The underscore character indicates that you aren't interested in the result of `task.value`.

Section 5

Load data on app launch

In this section, you'll use the `load()` method that you wrote in a previous section to load data when the app's root view appears onscreen.



Step 1

Open `ScrumdingerApp.swift`, and replace the `@State` property named `scrums` with a `@StateObject` property named `store`. Set the value of `store` to `ScrumStore()`.

The `@StateObject` property wrapper creates a single instance of an observable object for each instance of the structure that declares it.

ScrumdingerApp.swift

No Preview

```
import SwiftUI

@main
struct ScrumdingerApp: App {
    @StateObject private var store = ScrumStore()

    var body: some Scene {
        WindowGroup {
            ScrumsView(scrums: $store.scrums)
                .task {
                    do {
                        try await store.load()
                    } catch {
                        fatalError(error.localizedDesc
                    }
                }
        }
    }
}
```

Note

Xcode will display a compiler error until you update the `ScrumsView` initializer in the next step.

Step 2

Pass `ScrumsView` a binding to `store.scrums`.

Next, you'll load the user's scrums when the app's root view appears onscreen.

Step 3

Add a `task` modifier to `ScrumsView`.

Recall that the task modifier allows asynchronous function calls.

Step 4

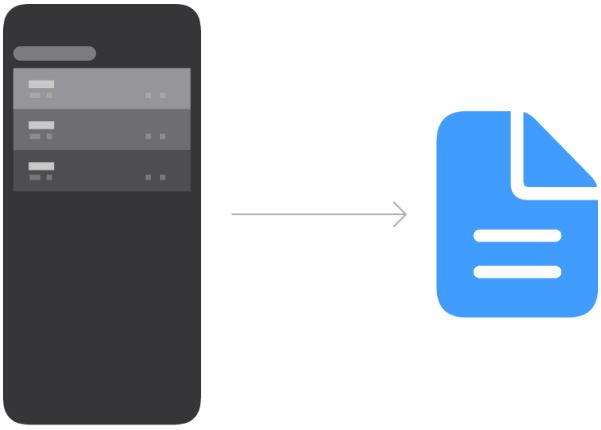
Use a `do-catch` statement to load the saved scrum or halt execution if `load()` throws an error.

You'll present error information to the user in a later tutorial.

Section 6

Save data in inactive state

You'll complete this tutorial by writing code to monitor the operational state of the app, and save user data when the value changes to `inactive`.



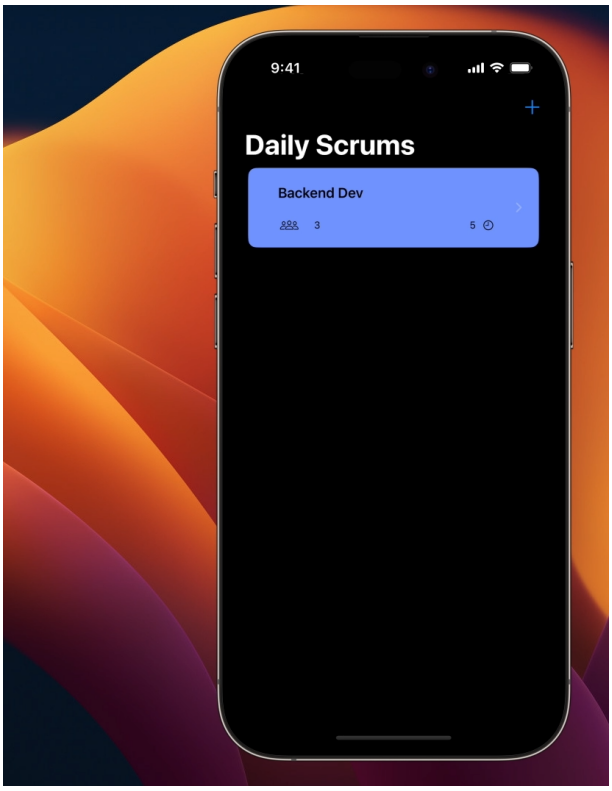
`.onChange (scenePhase) { ... }`

SwiftUI indicates the current operational state of your app's `Scene` instances with a `scenePhase` Environment value.

Step 1

In `ScrumsView.swift`, add an `@Environment` property for the `scenePhase` value.

You'll observe this value and save user data when it becomes inactive.



Replay

Step 2

Add a `saveAction` property, and pass an empty action in the preview.

You'll provide the `saveAction` closure when instantiating `ScrumsView`.

Note

Xcode will display a compiler error until you update the `ScrumsView` initializer in the `App` structure.

Step 3

Add an `onChange` modifier observing the `scenePhase` value.

You can use `onChange(of:perform:)` to trigger actions when a specified value changes.

Step 4

Call `saveAction()` if the scene is moving to the `inactive` phase.

A scene in the `inactive` phase no longer receives events and may be unavailable to the user.

Tip

Refer to the `ScenePhase` documentation for descriptions of each phase and instructions for triggering actions when the phase changes.

Now, you'll update `ScrumdingerApp` to pass a `saveAction` closure to the list view.

Step 5

In `ScrumdingerApp.swift`, add a trailing closure to the `ScrumsView` initializer, and create an empty `Task` inside.

Note

Updating the initializer fixes the compiler error.

Step 6

Add a `do-catch` block to save the scrum store or halt execution if `save()` throws an error.

Currently, the app terminates if it encounters an error writing to the file system. You'll add

more robust error handling in a later tutorial.

The scrums list is empty the first time a user launches the app.

Step 7

Run the app in Simulator, and add some scrums. Then, return to the Home Screen and quit the app.

The changes you made are visible when you launch the app again.

Experiment

With Scrumdinger running in Simulator, open Terminal and type `xcrun simctl get_app_container booted com.example.apple-samplecode.Scrumdinger data`. You can now view the path of the app's data folder. Try to locate the `scrums.data` file.

Scrumdinger now persists data between launches. In a later tutorial, you'll implement more robust error handling.

Check Your Understanding

Question 1 of 3

How can you update `Batch` so that views observing an instance of it render again when `temperature` is mutated?

```
class Batch {
    var name: String
    var temperature: Double
    var specificGravity: Double

    init(name: String, temp: Double, gravity: Double) {
        self.name = name
        self.temperature = temp
        self.specificGravity = gravity
    }
}
```

```
class Batch: StateObject {
   var name: String
   @Published var temperature: Double
   var specificGravity: Double

   init(name: String, temp: Double, gravity: Double) {
      self.name = name
      self.temperature = temp
      self.specificGravity = gravity
    }
}
```

```
class Batch: ObservedObject {
   var name: String
   var temperature: Double
   var specificGravity: Double

   init(name: String, temp: Double, gravity: Double) {
      self.name = name
      self.temperature = temp
      self.specificGravity = gravity
    }
}
```

```
class Batch: ObservableObject {
   var name: String
   @Published var temperature: Double
   var specificGravity: Double

   init(name: String, temp: Double, gravity: Double) {
      self.name = name
      self.temperature = temp
      self.specificGravity = gravity
    }
}
```

Submit Next Question

Next

Adopting new API features

Each SDK release includes new technologies, frameworks, and language features. In this article, you'll learn how to adopt new APIs while maintaining compatibility with older versions of operating systems.

Read article