Sample Code

Managing model data in your app

Create connections between your app's data model and views.

Download

iOS 17.0+ | iPadOS 17.0+ | macOS 14.0+ | Xcode 15.0+

Overview

A SwiftUI app can display data that people can change using the app's user interface (UI). To manage that data, an app creates a data model, which is a custom type that represents the data. A data model provides separation between the data and the views that interact with the data. This separation promotes modularity, improves testability, and helps make it easier to reason about how the app works.

Keeping the model data (that is, an instance of a data model) in sync with what appears on the screen can be challenging, especially when the data appears in multiple views of the UI at the same time.

SwiftUI helps keep your app's UI up to date with changes made to the data thanks to Observation. With Observation, a view in SwiftUI can form dependencies on observable data models and update the UI when data changes.

> Note
>
> Observation support in SwiftUI is available starting with iOS 17, iPadOS 17, macOS 14, tvOS 17, and watchOS 10. For information about adopting Observation in existing apps, see Migrating from the Observable Object protocol to the Observable macro.

Make model data observable

To make data changes visible to SwiftUI, apply the `Observable()` macro to your data model. This macro generates code that adds observation support to your data model at compile time, keeping your data model code focused on the properties that store data. For example, the following code defines a data model for books:

```swift
@Observable class Book: Identifiable {
    var title = "Sample Book Title"
    var author = Author()
    var isAvailable = true
}
```

Important

The `Observable()` macro, in addition to adding observation functionality, also conforms your data model type to the `Observable` protocol, which serves as a signal to other APIs that your type supports observation. Don't apply the `Observable` protocol by itself to your data model type, since that alone doesn't add any observation functionality. Instead, always use the `Observable` macro when adding observation support to your type.

Observe model data in a view

In SwiftUI, a view forms a dependency on an observable data model object, such as an instance of `Book`, when the view's `body` property reads a property of the object. If `body` doesn't read any properties of an observable data model object, the view doesn't track any dependencies.

When a tracked property changes, SwiftUI updates the view. If other properties change that `body` doesn't read, the view is unaffected and avoids unnecessary updates. For example, the view in the following code updates only when a book's `title` changes but not when `author` or `isAvailable` changes:

```swift
struct BookView: View {
    var book: Book

    var body: some View {
        Text(book.title)
    }
}
```

SwiftUI establishes this dependency tracking even if the view doesn't store the observable type, such as when using a global property or singleton:

```swift
var globalBook: Book = Book()

struct BookView: View {
    var body: some View {
        Text(globalBook.title)
    }
}
```

Observation also supports tracking of computed properties when the computed property makes use of an observable property. For instance, the view in the following code updates when the number of available books changes:

```swift
@Observable class Library {
    var books: [Book] = [Book(), Book(), Book()]

    var availableBooksCount: Int {
        books.filter(\.isAvailable).count
    }
}

struct LibraryView: View {
    @Environment(Library.self) private var library

    var body: some View {
        NavigationStack {
            List(library.books) { book in
                // ...
            }
            .navigationTitle("Books available: \(library.availableBooksCount)")
        }
    }
}
```

When a view forms a dependency on a collection of objects, of any collection type, the view tracks changes made to the collection itself. For instance, the view in the following code forms a dependency on `books` because `body` reads it. As changes occur to `books`, such as inserting, deleting, moving, or replacing items in the collection, SwiftUI updates the view.

```swift
struct LibraryView: View {
    @State private var books = [Book(), Book(), Book()]
    var body: some View {
        List(books) { book in
            Text(book.title)
        }
    }
}
```

However, `LibraryView` doesn't form a dependency on the property `title` because the view's `body` doesn't read it directly. The view stores the `List` content closure as an `@escaping` closure that SwiftUI calls when lazily creating list items before they appear on the screen. This means that instead of `LibraryView` depending on a book's `title`, each `Text` item of the list depends on `title`. Any changes to a `title` updates only the individual `Text` representing the book and not the others.

> **Note**
>
> Observation tracks changes to any observable property that appears in the execution scope of a view's `body` property.

You can also share an observable model data object with another view. The receiving view forms a dependency if it reads any properties of the object in its `body`. For example, in the following code `LibraryView` shares an instance of `Book` with `BookView`, and `BookView` displays the book's `title`. If the book's `title` changes, SwiftUI updates only `BookView`, and not `LibraryView`, because only `BookView` reads the `title` property.

```swift
struct LibraryView: View {
    @State private var books = [Book(), Book(), Book()]

    var body: some View {
        List(books) { book in
            BookView(book: book)
        }
    }
}

struct BookView: View {
    var book: Book

    var body: some View {
        Text(book.title)
    }
}
```

If a view doesn’t have any dependencies, SwiftUI doesn’t update the view when data changes. This approach allows an observable model data object to pass through multiple layers of a view hierarchy without each intermediate view forming a dependency.

```swift
// Will not update when any property of `book` changes.
struct LibraryView: View {
    @State private var books = [Book(), Book(), Book()]

    var body: some View {
        LibraryItemView(book: book)
    }
}

// Will not update when any property of `book` changes.
struct LibraryItemView: View {
    var book: Book

    var body: some View {
        BookView(book: book)
    }
}

// Will update when `book.title` changes.
struct BookView: View {
    var book: Book

    var body: some View {
        Text(book.title)
    }
}
```

However, a view that stores a reference to the observable object updates if the reference changes. This happens because the stored reference is part of the view's value and not because the object is observable. For example, if the reference to book in the following code changes, SwiftUI updates the view:

```
struct BookView: View {
    var book: Book

    var body: some View {
        // ...
    }
}
```

A view can also form a dependency on an observable data model object accessed through another object. For example, the view in the following code updates when the author's `name` changes:

```
struct LibraryItemView: View {
    var book: Book

    var body: some View {
        VStack(alignment: .leading) {
            Text(book.title)
            Text("Written by: \(book.author.name)")
                .font(.caption)
        }
    }
}
```

Create the source of truth for model data

To create and store the source of truth for model data, declare a private variable and initialize it with a instance of an observable data model type. Then wrap it with a `State` property wrapper. For example, the following code stores an instance of the data model type `Book` in the state variable `book`:

```swift
struct BookView: View {
    @State private var book = Book()

    var body: some View {
        Text(book.title)
    }
}
```

By wrapping the book with `State`, you're telling SwiftUI to manage the storage of the instance. Each time SwiftUI re-creates `BookView`, it connects the `book` variable to the managed instance, providing the view a single source of truth for the model data.

You can also create a state object in your top-level `App` instance or in one of your app's `Scene` instances. For example, the following code creates an instance of `Library` in the app's top-level structure:

```swift
@main
struct BookReaderApp: App {
    @State private var library = Library()

    var body: some Scene {
        WindowGroup {
            LibraryView()
                .environment(library)
        }
    }
}
```

Share model data throughout a view hierarchy

If you have a data model object, like `Library`, that you want to share throughout your app, you can either:

- pass the data model object to each view in the view hierarchy; or
- add the data model object to the view's environment

Passing model data to each view is convenient when you have a shallow view hierarchy; for example, when a view doesn't share the object with its subviews. However, you usually don't know if a view needs to pass the object to subviews, and you may not know if a subview deep inside the layers of the hierarchy needs the model data.

To share model data throughout a view hierarchy without needing to pass it to each view, add the model data to the view's environment. You can add the data to the environment using either `environment(_:_:)` or the `environment(_:)` modifier, passing in the model data.

Before you can use the `environment(_:_:)` modifier, you need to create a custom `EnvironmentKey`. Then extend `EnvironmentValues` to include a custom environment property that gets and sets the value for the custom key. For instance, the following code creates an environment key and property for `library`:

```
extension EnvironmentValues {
    var library: Library {
        get { self[LibraryKey.self] }
        set { self[LibraryKey.self] = newValue }
    }
}

private struct LibraryKey: EnvironmentKey {
    static let defaultValue: Library = Library()
}
```

With the custom environment key and property in place, a view can add model data to its environment. For example, `LibraryView` adds the source of truth for a `Library` instance to its environment using the `environment(_:_:)` modifier:

```
@main
struct BookReaderApp: App {
    @State private var library = Library()

    var body: some Scene {
        WindowGroup {
            LibraryView()
                .environment(\.library, library)
        }
    }
}
```

To retrieve the `Library` instance from the environment, a view defines a local variable that stores a reference to the instance, and then wraps the variable with the `Environment` property wrapper, passing in the key path to the custom environment value.

```
struct LibraryView: View {
    @Environment(\.library) private var library

    var body: some View {
        // ...
    }
}
```

You can also store model data directly in the environment without defining a custom environment value by using the `environment(_:)` modifier. For instance, the following code adds a `Library` instance to the environment using this modifier:

```
@main
struct BookReaderApp: App {
    @State private var library = Library()

    var body: some Scene {
        WindowGroup {
            LibraryView()
                .environment(library)
        }
    }
}
```

To retrieve the instance from the environment, another view defines a local variable to store the instance and wraps it with the `Environment` property wrapper. But instead of providing a key path to the environment value, you can provide the model data type, as shown in the following code:

```
struct LibraryView: View {
    @Environment(Library.self) private var library

    var body: some View {
        // ...
    }
}
```

By default, reading an object from the environment returns a non-optional object when using the object type as the key. This default behavior assumes that a view in the current hierarchy previously stored a non-optional instance of the type using the `environment(_:)` modifier. If a view attempts to retrieve an object using its type and that object isn't in the environment, SwiftUI throws exception.

In cases where there is no guarantee that an object is in the environment, retrieve an optional version of the object as shown in the following code. If the object isn't available the environment, SwiftUI returns `nil` instead of throwing an exception.

```
@Environment(Library.self) private var library: Library?
```

Change model data in a view

In most apps, people can change data that the app presents. When data changes, any views that display the data should update to reflect the changed data. With Observation in SwiftUI, a view can support data changes without using property wrappers or bindings. For example, the following toggles the `isAvailable` property of a book in the action closure of a button:

```
struct BookView: View {
    var book: Book

    var body: some View {
        List {
            Text(book.title)
            HStack {
                Text(book.isAvailable ? "Available for checkout" : "Waiting for retu
                Spacer()
                Button(book.isAvailable ? "Check out" : "Return") {
                    book.isAvailable.toggle()
                }
            }
        }
    }
}
```

However, there may be times when a view expects a binding before it can change the value of a mutable property. To provide a binding, wrap the model data with the `Bindable` property wrapper. For example, the following code wraps the `book` variable with `@Bindable`. Then it uses a `TextField` to change the `title` property of a book, and a `Toggle` to change the `is Available` property, using the $ syntax to pass a binding to each property.

```
struct BookEditView: View {
    @Bindable var book: Book
    @Environment(\.dismiss) private var dismiss

    var body: some View {
        VStack() {
            HStack {
                Text("Title")
                TextField("Title", text: $book.title)
                    .textFieldStyle(.roundedBorder)
                    .onSubmit {
                        dismiss()
                    }
            }

            Toggle(isOn: $book.isAvailable) {
                Text("Book is available")
            }

            Button("Close") {
                dismiss()
            }
            .buttonStyle(.borderedProminent)
        }
        .padding()
    }
}
```

You can use the `Bindable` property wrapper on properties and variables to an `Observable` object. This includes global variables, properties that exist outside of SwiftUI types, or even local variables. For example, you can create a `@Bindable` variable within a view's `body`:

```
struct LibraryView: View {
    @State private var books = [Book(), Book(), Book()]

    var body: some View {
        List(books) { book in
            @Bindable var book = book
            TextField("Title", text: $book.title)
        }
    }
}
```

The `@Bindable` variable `book` provides a binding that connects `TextField` to the `title` property of a book so that a person can make changes directly to the model data.

See Also

Creating model data

{} Migrating from the Observable Object protocol to the Observable macro

Update your existing app to leverage the benefits of Observation in Swift.

```
@attached(member, names: named(_$observationRegistrar), named(access), named(withMutation), named(shouldNotifyObservers)) @attached(member Attribute) @attached(extension, conformances: Observable) macro Observable()
```

Defines and implements conformance of the Observable protocol.

{} Monitoring data changes in your app

Show changes to data in your app's user interface by using observable objects.

`struct StateObject`

A property wrapper type that instantiates an observable object.

`struct ObservedObject`

A property wrapper type that subscribes to an observable object and invalidates a view whenever the observable object changes.

`protocol ObservableObject : AnyObject`

A type of object with a publisher that emits before the object has changed.