Sample Code

Monitoring data changes in your app

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

Download

iOS 16.2+ | iPadOS 16.2+ | macOS 13.1+ | Xcode 14.3+

Overview

You typically store and process data in your app using a data model that's separate from your app's user interface (UI) and other logic. The separation promotes modularity, improves testability, and makes it easier to reason about how your app works.

Traditionally, you use a view controller to move data back and forth between the model and the UI, but SwiftUI handles most of this synchronization for you. To update views when data changes, you make your data model classes observable objects, publish their properties, and declare instances of them using property wrappers. To ensure user-driven data changes flow back into the model, you bind UI controls to model properties. Working together, these features help you to maintain a single source of truth for your data.

Important

This article explains how to observe model data changes in SwiftUI apps that have a minimum deployment target that is prior to iOS 17, iPadOS 17, macOS 14, tvOS 17, or watchOS 10. To prepare your app for a newer version of one of those operating systems, see Migrating from the Observable Object protocol to the Observable macro. If your app's minimum deployment target is iOS 17, iPadOS 17, macOS 14, tvOS 17, or watchOS 10 or later, use the observation approach described in Managing model data in your app.

Make model data observable

To make the data changes in your model visible to SwiftUI, adopt the `ObservableObject` protocol for model classes. For example, you can create a `Book` class that's an observable object:

```
class Book: ObservableObject {
}
```

The system automatically infers the `ObjectWillChangePublisher` associated type for the class and synthesizes the required doc://com.apple.documentation/documentation/Combine/ObservableObject/objectWillChange-2oa5v method that emits the changed values of published properties. To publish a property, add the `Published` property wrapper to the property's declaration:

```
class Book: ObservableObject {
    @Published var title = "Sample Book Title"
}
```

Avoid the overhead of a published property when you don't need it. Only publish properties that both can change and that matter to the UI. For example, the `Book` class might have an `identifier` property that never changes after initialization:

```
class Book: ObservableObject {
    @Published var title = "Sample Book Title"

    let identifier = UUID() // A unique identifier that never changes.
}
```

You can still display the identifier in your user interface, but because it isn't published, SwiftUI doesn't watch that particular property for changes.

Monitor changes in observable objects

To tell SwiftUI to monitor an observable object, add the `ObservedObject` property wrapper to the property's declaration:

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

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

You can pass individual properties of an observed object to child views, as shown above. When the data changes, like when you load new data from disk, SwiftUI updates all the affected views. You can also pass an entire observable object to a child view and share model objects across levels of a view hierarchy:

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

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

struct BookEditView: View {
    @ObservedObject var book: Book

    // ...
}
```

Instantiate a model object in a view

SwiftUI might create or recreate a view at any time, so it's important that initializing a view with a given set of inputs always results in the same view. As a result, it's unsafe to create an observed object inside a view. Instead, SwiftUI provides the `StateObject` property wrapper, which creates a single source of truth for a reference type that you store in a view hierarchy. You can safely create a `Book` instance inside a view this way:

```swift
struct LibraryView: View {
    @StateObject private var book = Book()

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

A state object behaves like an observed object, except that SwiftUI creates and manages a single object instance for a given view instance, regardless of how many times it recreates the view. You can use the object locally, or pass the state object into another view's observed object property, as shown in the above example.

While SwiftUI doesn't recreate the state object within a view, it does create a distinct object instance for each view instance. For example, each `LibraryView` in the following code gets a unique `Book` instance:

```
VStack {
    LibraryView()
    LibraryView()
}
```

You can also create a state object in your top level `App` instance, or in one of your app's `Scene` instances. For example, if you define an observable object called `Library` to hold a collection of books for a book reader app, you could create a single library instance in the app's top level structure:

```
@main
struct BookReader: App {
    @StateObject private var library = Library()

    // ...
}
```

Share an object throughout your app

If you have a data model object that you want to use throughout your app, but don't want to pass it through many layers of hierarchy, you can use the `environmentObject(_:)` view modifier to put the object into the environment instead:

```
@main
struct BookReader: App {
    @StateObject private var library = Library()

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

Any descendant view of the view to which you apply the modifier can then access the data model instance by declaring a property with the `EnvironmentObject` property wrapper:

```
struct LibraryView: View {
    @EnvironmentObject var library: Library

    // ...
}
```

If you use an environment object, you might add it to the view at the top of your app's hierarchy, as shown above. Alternatively, you might add it to the root view of a subtree in your view hierarchy. Either way, remember to also add it to the preview provider of any view that uses the object, or that has a descendant that uses the object:

```
struct LibraryView_Previews: PreviewProvider {
    static var previews: some View {
        LibraryView()
            .environmentObject(Library())
    }
}
```

Create a two-way connection using bindings

When you allow a person to change the data in the UI, use a binding to the corresponding property. This ensures that updates flow back into the data model automatically. You can get a binding to an observed object, state object, or environment object property by prefixing the name of the object with the dollar sign ($). For example, if you let someone edit the title of a book by adding a `TextField` to the `BookEditView`, give the text field a binding to the book's `title` property:

```
struct BookEditView: View {
    @ObservedObject var book: Book

    var body: some View {
        TextField("Title", text: $book.title)
    }
}
```

The binding connects the view element to the underlying model so that a person makes changes directly to the model data.

See Also

Creating model data

{} Managing model data in your app

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

{} 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(memberAttribute) @attached(extension, conformances: Observable) macro Observable()`

Defines and implements conformance of the Observable protocol.

`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.