Sample Code

Migrating from the Observable Object protocol to the Observable macro

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

Download

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

Overview

Starting with iOS 17, iPadOS 17, macOS 14, tvOS 17, and watchOS 10, SwiftUI provides support for Observation, a Swift-specific implementation of the observer design pattern. Adopting Observation provides your app with the following benefits:

- Tracking optionals and collections of objects, which isn't possible when using `ObservableObject`.
- Using existing data flow primitives like `State` and `Environment` instead of object-based equivalents such as `StateObject` and `EnvironmentObject`.
- Updating views based on changes to the observable properties that a view's `body` reads instead of any property changes that occur to an observable object, which can help improve your app's performance.

To take advantage of these benefits in your app, you'll discover how to replace existing source code that relies on `ObservableObject` with code that leverages the `Observable()` macro.

> **Note**
>
> Download this sample to see the migrated version of the sample app. To see the premigrated version, download the sample available in Monitoring data changes in your app. You can also use the premigrated version to code along with this article.

Use the Observable macro

To adopt Observation in an existing app, begin by replacing `ObservableObject` in your data model type with the `Observable()` macro. The `Observable()` macro generates source code at compile time that adds observation support to the type.

```
// BEFORE
import SwiftUI

class Library: ObservableObject {
    // ...
}
```

```
// AFTER
import SwiftUI

@Observable class Library {
    // ...
}
```

Then remove the `Published` property wrapper from observable properties. Observation doesn't require a property wrapper to make a property observable. Instead, the accessibility of the property in relationship to an observer, such as a view, determines whether a property is observable.

```
// BEFORE
@Observable class Library {
    @Published var books: [Book] = [Book
}
```

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

If you have properties that are accessible to an observer that you don't want to track, apply the `ObservationIgnored()` macro to the property.

Migrate incrementally

You don't need to make a wholesale replacement of the `ObservableObject` protocol throughout your app. Instead, you can make changes incrementally. Start by changing one data model type to use the `Observable()` macro. Your app can mix data model types that use different observation systems. However, SwiftUI tracks changes differently based on the observation system that a data model type uses, `Observable` versus `ObservableObject`.

You may notice slight behavioral differences in your app based on the tracking method. For instance, when tracking as `Observable()`, SwiftUI updates a view only when an observable property changes and the view's `body` reads the property directly. The view doesn't update when observable properties not read by `body` changes. In contrast, a view updates when any published property of an `ObservableObject` instance changes, even if the view doesn't read the property that changes, when tracking as `ObservableObject`.

> **Note**
>
> To learn more about when SwiftUI updates views when observable properties change, see Managing model data in your app.

Migrate other source code

The only change made to the sample app so far is to apply the `Observable()` macro to `Library` and remove support for the `ObservableObject` protocol. The app still uses the `ObservableObject` data flow primitive like `StateObject` to manage an instance of `Library`. If you were to build and run the app, SwiftUI still updates the views as expected. That's because data flow property wrappers such as `StateObject` and `EnvironmentObject` support types that use the `Observable()` macro. SwiftUI provides this support so apps can make source code changes incrementally.

However, to fully adopt Observation, replace the use of `StateObject` with `State` after updating your data model type. For example, in the following code the main app structure creates an instance of `Library` and stores it as a `StateObject`. It also adds the `Library` instance to the environment using the `environmentObject(_:)` modifier.

```
// BEFORE
@main
struct BookReaderApp: App {
    @StateObject private var library = Library()

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

Now that `Library` no longer conforms to `ObservableObject`, the code can change to use `State` instead of `StateObject` and to add `library` to the environment using the `environment(_:)` modifier.

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

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

One more change must happen before `Library` fully adopts Observation. Previously the view `LibraryView` retrieved a `Library` instance from the environment using the `EnvironmentObject` property wrapper. The new code, however, uses the `Environment` property wrapper instead.

```
// BEFORE
struct LibraryView: View {
    @EnvironmentObject var library: Libr

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

```
// AFTER
struct LibraryView: View {
    @Environment(Library.self) private v

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

Remove the ObservedObject property wrapper

To wrap up the migration of the sample app, change the data model type `Book` to support Observation by removing `ObservableObject` from the type declaration and apply the `Observable()` macro. Then remove the `Published` property wrapper from observable properties.

```
// BEFORE
class Book: ObservableObject, Identifiab
    @Published var title = "Sample Book

    let id = UUID() // A unique identifi
}
```

```
// AFTER
@Observable class Book: Identifiable {
    var title = "Sample Book Title"

    let id = UUID() // A unique identifi
}
```

Next, remove the `ObservedObject` property wrapper from the `book` variable in the `BookView`. This property wrapper isn't needed when adopting Observation. That's because SwiftUI automatically tracks any observable properties that a view's `body` reads directly. For example, SwiftUI updates `BookView` when `book.title` changes.

```
// BEFORE
struct BookView: View {
    @ObservedObject var book: Book
    @State private var isEditorPresented

    var body: some View {
        HStack {
            Text(book.title)
            Spacer()
            Button("Edit") {
                isEditorPresented = true
            }
        }
        .sheet(isPresented: $isEditorPre
            BookEditView(book: book)
        }
    }
}
```

```
// AFTER
struct BookView: View {
    var book: Book
    @State private var isEditorPresented

    var body: some View {
        HStack {
            Text(book.title)
            Spacer()
            Button("Edit") {
                isEditorPresented = true
            }
        }
        .sheet(isPresented: $isEditorPre
            BookEditView(book: book)
        }
    }
}
```

However, if a view needs a binding to an observable type, replace `ObservedObject` with the `Bindable` property wrapper. This property wrapper provides binding support to an observable type so that views that expect a binding can change an observable property. For instance, in the following code `TextField` receives a binding to `book.title`:

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

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

```
    var body: some View {
        VStack() {
            TextField("Title", text: $bo
                .textFieldStyle(.rounded
                .onSubmit {
                    dismiss()
                }

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

```
    var body: some View {
        VStack() {
            TextField("Title", text: $bo
                .textFieldStyle(.rounded
                .onSubmit {
                    dismiss()
                }

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

See Also

Creating model data

{} Managing model data in your app

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

`@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.

{} 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.