SwiftUI Property Wrappers PoC

Preoteasa Ioan-Silviu

This PoC was created in order to showcase the functionality of @Published, @State and @Environment variables. As opposed to it's ancestor UIKit, SwiftUI does not use dataSources to get a handle on the data inputted to a view like it used to be done using UITableViews and UICollectionViews. This addition to the framework makes it easier to handle data, given the fact that we do not need to handle every state change manually, but it's handled for us in the background. In UIKit, when something in the dataSource was modified we would have to call reloadDataSource() in order to be sure that the current screen will be redrawn. Now, using @Published and @State variables in a view model.

A @Published variable is usually found in a ViewModel that inherits the @ObservableObject protocol. The views depending on these variables will be notified as soon as they are modified, leading the view to be redrawn (either partially or completely depending on how modular the developer has made the parent view). Behind the scenes, there are some features from the Combine framework that notify the view, most notably being the objectWillChange.send() function.

@State variables are used in more simple/primitive views, such views that would only benefit from having very little information given to them, those views that do not handle any business logic.

@Environment variables are used in certain situations where using dependency injection to change the state of a view is not ideal, a very good example being in the ColorPickerView from the PoC, where the @Environment(\.dismiss) variable is used to change the state of the view, such that we will not need to pass a binding or closure to the view, making the implementation quicker and cleaner.

ViewModels can be instantiated in one of 3 ways: @StateObject, @ObservedObject, @EnvironmentObject. In the PoC only @StateObject was used, as dependency injection would not be clean here. You instantiate viewModels using @StateObject when there is the first instance of the object. If a @StateObject is passed to a child view through an initializer the child will have to note the object as an @ObservedObject and it does not have to hit init(). Another way of sending a view model would be sending it to all children in the hierarchy using .environmentObject(viewModel). This way, every child in the hierarchy can make use of the view model, and it has to be marked as @EnvironmentObject by all children that make use of it.

Now to the PoC, I kept is a rudimentary as possible in order to have a clearer purpose. On the main screen we are presented with two choices: “Go to color picker” and “Open updateable sheet”. If we go to the color picker, we can see two @Published variables at work, color and opacity, it is a view with a big circle in the middle, where we can seamlessly change the color and opacity of it. What happens in the background is that when the two variables are changed, the view is immediately notified that a change has occurred and redraws itself. Now we can press the “Go Back” button in order to dismiss the view. This will trigger the @Environment(\.dismiss) variable.

Now that we went back to the main screen we can go to the updateable sheet which uses a @State and Binding<Bool> in order to be presented. There I made a sample list that is updated every second with a new random string, in order to showcase the fact that the view will be updated when the @Published variable is modified.