SwiftUI MVVM Architecture:

In contrast to UIKit which was an imperative event-driven framework, SwiftUI is declarative and state-driven. In SwiftUI where our views are structs, we can think of them as functions, where we provide an input (state) and it draws the output, and the only way to change the output is to change the input. In the simplest form, where our view does not rely on an external state, its local @State variables take the role of the ViewModel, which provides the subscription method (Binding) for refreshing the UI when the state changes. In more complex cases where our view references an external (ObservableObject) which is called the ViewModel.

The reason that MVVM architecture is so popular for SwiftUI is that it gives us three main points, Separation of concerns, the ability to mock data to review UI, and an architecture that allows you to easily unit test and UI test to ensure the correctness of our software. In SwiftUI the view layer contains all of our view code, and we try to keep them as small as possible to be able to reuse them on different screens. Constructing the view itself is easy using the modifiers in swift but the real challenge is where to put the states of our app. The principle of a single source of truth is what helps us here.

Property Wrappers:

- @State private var: we use @State when the state won't leave our view or the sub-views, and is typically used for information that is UI relevant and has no impact on the logic of the app, and is not persisted.
- @Binding var: is used for sub-views when referencing the state of the parent view, the view does not own the state but is updated or updates the value owned by the parent view.
- @ObservedObject var: used when the state is not UI relevant, it is part of the logic of the app, but it is scoped for a specific part of the application. The observed object is typically not owned by the View

itself but it is received from its parent view, Observable object is recreated every time the parent view is recreated.

- @StateObject private var: similar to @ObservedObject in the way that they are intended for states that are UI relevant and scoped to the view and its subviews but the **ObservableObject** is not created every time the view is displayed. So when choosing between the two we must compromise between keeping the data in the memory or recreating it every time.
- @EnvironmentObject var: these are ObservableObjects injected into a view hierarchy using .environmentObject() modifier. These are intended for states that are not only UI relevant and not scoped for a specific part of the application. When choosing between @StateObject or @ObservedObject and @EnvironmentObject we are basically choosing if we want the state to be more public but easier to access through the hierarchy of the application or targeting it for a specific part of the view. Meaning we have to pass the other two through the initializer of the view.