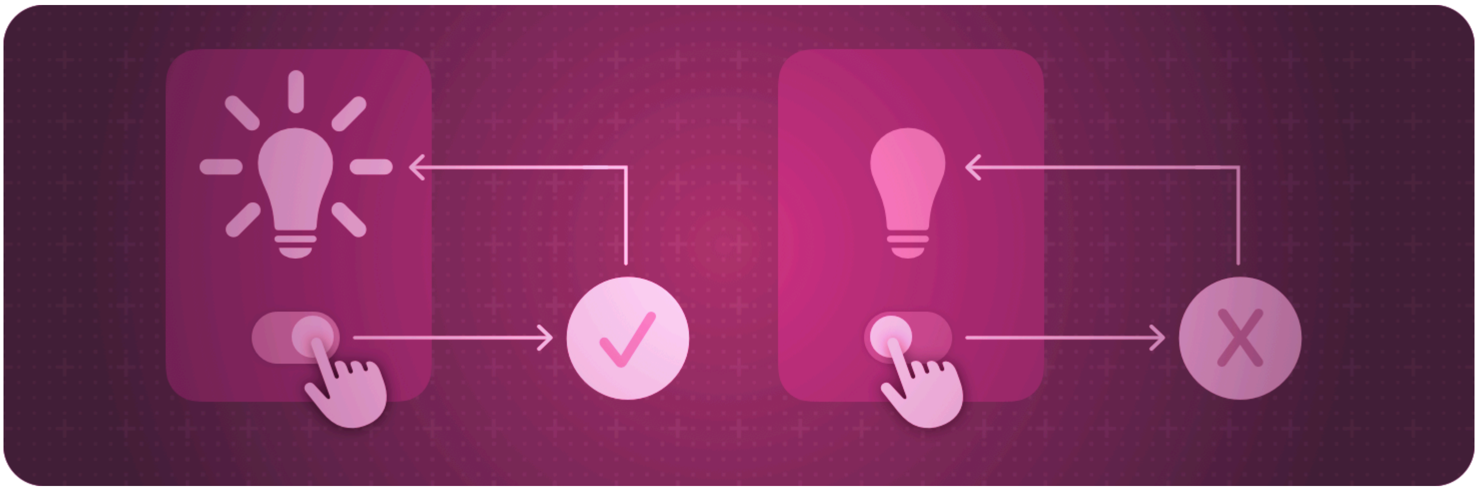
API Collection

Model data

Manage the data that your app uses to drive its interface.

Overview

SwiftUI offers a declarative approach to user interface design. As you compose a hierarchy of views, you also indicate data dependencies for the views. When the data changes, either due to an external event or because of an action that the user performs, SwiftUI automatically updates the affected parts of the interface. As a result, the framework automatically performs most of the work that view controllers traditionally do.



The framework provides tools, like state variables and bindings, for connecting your app's data to the user interface. These tools help you maintain a single source of truth for every piece of data in your app, in part by reducing the amount of glue logic you write. Select the tool that best suits the task you need to perform:

- Manage transient UI state locally within a view by wrapping value types as `State` properties.
- Share a reference to a source of truth, like local state, using the `Binding` property wrapper.

- Connect to and observe reference model data in views by applying the `Observable()` macro to the model data type. Instantiate an observable model data type directly in a view using a `State` property. Share the observable model data with other views in the hierarchy without passing a reference using the `Environment` property wrapper.

Leveraging property wrappers

SwiftUI implements many data management types, like `State` and `Binding`, as Swift property wrappers. Apply a property wrapper by adding an attribute with the wrapper's name to a property's declaration.

```
@State private var isVisible = true // Declares isVisible as a state variable.
```

The property gains the behavior that the wrapper specifies. The state and data flow property wrappers in SwiftUI watch for changes in your data, and automatically update affected views as necessary. When you refer directly to the property in your code, you access the wrapped value, which for the `isVisible` state property in the example above is the stored Boolean.

```
if isVisible == true {
    Text("Hello") // Only renders when isVisible is true.
}
```

Alternatively, you can access a property wrapper's projected value by prefixing the property name with the dollar sign ($). SwiftUI state and data flow property wrappers project a `Binding`, which is a two-way connection to the wrapped value, allowing another view to access and mutate a single source of truth.

```
Toggle("Visible", isOn: $isVisible) // The toggle can update the stored value.
```

For more information about property wrappers, see Property Wrappers in The Swift Programming Language.

Topics

Creating and sharing view state

Managing user interface state

Encapsulate view-specific data within your app's view hierarchy to make your views reusable.

`struct State`

A property wrapper type that can read and write a value managed by SwiftUI.

`struct Bindable`

A property wrapper type that supports creating bindings to the mutable properties of observable objects.

`struct Binding`

A property wrapper type that can read and write a value owned by a source of truth.

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(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.

Responding to data changes

`func onChange(of:initial:_:)`

Adds a modifier for this view that fires an action when a specific value changes.

`func onReceive<P>(P, perform: (P.Output) -> Void) -> some View`

Adds an action to perform when this view detects data emitted by the given publisher.

Distributing model data throughout your app

`func environmentObject<T>(T) -> some View`

Supplies an observable object to a view's hierarchy.

`func environmentObject<T>(T) -> some Scene`

Supplies an `ObservableObject` to a view subhierarchy.

`struct EnvironmentObject`

A property wrapper type for an observable object that a parent or ancestor view supplies.

Managing dynamic data

`protocol DynamicProperty`

An interface for a stored variable that updates an external property of a view.

See Also

Data and storage

Environment values

Share data throughout a view hierarchy using the environment.

Preferences

Indicate configuration preferences from views to their container views.

Persistent storage

Store data for use across sessions of your app.