Sample Code

Restoring your app's state with SwiftUI

Provide app continuity for users by preserving their current activities.

Download

iOS 14.0+ | iPadOS 14.0+ | Xcode 12.0+

Overview

This SwiftUI sample project demonstrates how to preserve your app's state information and restore the app to that previous state on subsequent launches. During a subsequent launch, restoring your interface to the previous interaction point provides continuity for the user, and lets them finish active tasks quickly.

When using your app, the user performs actions that affect the user interface. For example, the user might view a specific page of information, and after the user leaves the app, the operating system might terminate it to free up the resources it holds. The user can return to where they left off — and UI state restoration is a core part of making that experience seamless.

This sample app demonstrates the use of state preservation and restoration for scenarios where the system interrupts the app. The sample project manages a set of products. Each product has a title, an image, and other metadata you can view and edit. The project shows how to preserve and restore a product in its `DetailView`.

Configure the sample code project

In Xcode, select your development team on the iOS target's Signing and Capabilities tab.

Enable state preservation and restoration

This sample code project uses SwiftUI's `Scene` to manage the app's user interface with its life cycle managed by the system. On iOS, state restoration is especially important at the window or scene level, because windows come and go frequently. It's necessary to save and restore state associated with each one. On the iPad, it's especially important because an app in the switcher is not necessarily running. Scene-level state restoration preserves the illusion they are running.

To support state preservation and restoration, this sample uses `NSUserActivity` objects. For each user activity, the app must supply an activity type defined in its `Info.plist`.

Use scene storage

SwiftUI has the concept of "storing scene data" or `SceneStorage`. Operating similar to `State`, scene storage is a property wrapper type that consists of a key/value pair. The key makes it possible for the system to save and restore the value correctly. The value is required to be of a `plist` type, so the system can save and restore it correctly. iOS ingests this scene storage using the key/value and then reads and writes to persisted, per-scene storage. The OS manages saving and restoring scene storage on the user's behalf. The underlying data that backs scene storage is not directly available, so the app must access it via `@SceneStorage` property wrapper. The OS makes no guarantees as to when and how often the data will be persisted. The data in scene storage is not necessarily equivalent to an application's data model. Scene storage is intended to be used *with* the data model. Ultimately, consider scene storage a "state scoped to a scene". Don't use scene storage with sensitive data.

Each view that needs its own state preservation implements a `@SceneStorage` property wrapper. For example `ContentView` uses one to restore the selected product:

```
@SceneStorage("ContentView.selectedProduct") private var selectedProduct: String?
```

`DetailView` uses one to restore its current selected tab:

```
@SceneStorage("DetailView.selectedTab") private var selectedTab = Tabs.detail
```

> **Note**
>
> Each scene storage key must be unique, and properly scoped to the area or use within the app. Because this scene storage is local to the app, it's not necessary to prefix it with the app's bundle identifier. Use some disambiguating prefix where needed to ensure its uniqueness.

Restore the app state with an activity object

An `NSUserActivity` object captures the app's state at the current moment in time. For example, include information about the data the app is currently displaying. The system saves the provided object and returns it to the app the next time it launches. The sample creates a new `NSUserActivity` object when the user closes the app or the app enters the background.

Each SwiftUI view that wants to advertise an `NSUserActivity` for handoff, Spotlight, etc. must specify a `userActivity(_:isActive:_:)` view modifier to advertise the `NSUserActivity`. The `activityType` parameter is the user activity's type, the `isActive` parameter indicates whether a user activity of the specified type is advertised (this parameter defaults to `true`), and whether it uses the specified handler to fill in the user-activity contents. The scope of the user activity applies only to the scene or window in which the view is. Multiple views can advertise the same activity type, and the handlers can all contribute to the contents of the user activity. Note that handlers are only called for `userActivity` view modifiers where the `isActive` parameter is `true`. If none of the `userActivity` view modifiers specify `isActive` as `true`, the user activity will not be advertised by iOS.

Each SwiftUI view that wants to handle incoming `NSUserActivities` must specify a `onContinueUserActivity(_:perform:)` view modifier. This takes the `NSUserActivity` type and a handler to invoke when the view receives the specified activity type for the scene or window in which the view is.

```
.onContinueUserActivity(DetailView.productUserActivityType) { userActivity in
    if let product = try? userActivity.typedPayload(Product.self) {
        selectedProduct = product.id.uuidString
    }
}
```

Test state restoration

This sample restores the following user interface:

- Detail View Controller — Tap a product in the collection view to open its detail information. The app restores the selected product and selected tab.
- Detail View Controller's Edit State — In the detail view, tap Edit. The app restores the edit view and its content.
- Secondary Window — (iPad only) Drag a product from the collection view over to the left or right of the device screen to create a second scene window. The app restores that scene and its product.

State restoration can be tested both on the device and Simulator. When debugging the sample project, the system automatically deletes its preserved state when the user force quits the app.

Deleting the preserved state information is a safety precaution. In addition, the system also deletes the preserved state if this app crashes at launch time.

To test the sample app's ability to restore the sample's state, don't use the app switcher to force quit it during debugging. Instead, use Xcode to stop the app or stop the app programmatically. Another technique is to suspend the sample app using the Home button, and then stop the debugger in Xcode. Launch the sample app again using Xcode, and SwiftUI initiates the state restoration process.

To use Spotlight with Handoff, follow these steps:

1. In Xcode set a breakpoint in `DetailView.swift` at `onContinueUserActivity` closure.
2. Run the sample project.
3. Tap a product ("Cherries") in the collection view to navigate to its detail information.
4. Pull down the system sheet from the top of the screen (to force Spotlight to update its index and request the activity). Note that the `DetailView` `userActivity` closure is called by iOS.
5. Go back to the app, and go back to the collection view.
6. Tap a product other than Cherries (i.e. Mango).
7. Suspend the app by tapping the Home button.
8. At the Home screen, swipe downwards to open the Spotlight window.
9. In the Spotlight search field, type "Cherries". The search results will show "Show Cherries Product".
0. Tap it. Note that `DetailView` `onContinueUserActivity` closure is called. The `Detail View` will show the Cherries product.

See Also

Saving state across app launches

`func defaultAppStorage(UserDefaults) -> some View`

The default store used by `AppStorage` contained within the view.

`struct AppStorage`

A property wrapper type that reflects a value from `UserDefaults` and invalidates a view on a change in value in that user default.

`struct SceneStorage`

A property wrapper type that reads and writes to persisted, per-scene storage.