SwiftUI / WindowGroup

Structure

WindowGroup

A scene that presents a group of identically structured windows.

iOS 14.0+ | iPadOS 14.0+ | Mac Catalyst 14.0+ | macOS 11.0+ | tvOS 14.0+ | visionOS 1.0+ | watchOS 7.0+

```
struct WindowGroup<Content> where Content : View
```

Mentioned in

Building and customizing the menu bar with SwiftUI

Overview

Use a `WindowGroup` as a container for a view hierarchy that your app presents. The hierarchy that you declare as the group's content serves as a template for each window that the app creates from that group:

```
@main
struct Mail: App {
    var body: some Scene {
        WindowGroup {
            MailViewer() // Define a view hierarchy for the window.
        }
    }
}
```

SwiftUI takes care of certain platform-specific behaviors. For example, on platforms that support it, like macOS and iPadOS, people can open more than one window from the group simultaneously.

In macOS, people can gather open windows together in a tabbed interface. Also in macOS, window groups automatically provide commands for standard window management.

Important

To enable an iPadOS app to simultaneously display multiple windows, be sure to include the `UIApplicationSupportsMultipleScenes` key with a value of `true` in the `UIApplicationSceneManifest` dictionary of your app's Information Property List.

Every window in the group maintains independent state. For example, the system allocates new storage for any `State` or `StateObject` variables instantiated by the scene's view hierarchy for each window that it creates.

For document-based apps, use `DocumentGroup` to define windows instead.

Open windows programmatically

If you initialize a window group with an identifier, a presentation type, or both, you can programmatically open a window from the group. For example, you can give the mail viewer scene from the previous example an identifier:

```
WindowGroup(id: "mail-viewer") { // Identify the window group.
    MailViewer()
}
```

Elsewhere in your code, you can use the `openWindow` action from the environment to create a new window from the group:

```
struct NewViewerButton: View {
    @Environment(\.openWindow) private var openWindow

    var body: some View {
        Button("Open new mail viewer") {
            openWindow(id: "mail-viewer") // Match the group's identifier.
        }
    }
}
```

Be sure to use unique strings for identifiers that you apply to window groups in your app.

Present data in a window

If you initialize a window group with a presentation type, you can pass data of that type to the window when you open it. For example, you can define a second window group for the Mail app that displays a specified message:

```
@main
struct Mail: App {
    var body: some Scene {
        WindowGroup {
            MailViewer(id: "mail-viewer")
        }

        // A window group that displays messages.
        WindowGroup(for: Message.ID.self) { $messageID in
            MessageDetail(messageID: messageID)
        }
    }
}
```

When you call the `openWindow` action with a value, SwiftUI finds the window group with the matching type and passes a binding to the value into the window group's content closure. For example, you can define a button that opens a message by passing the message's identifier:

```
struct NewMessageButton: View {
    var message: Message
    @Environment(\.openWindow) private var openWindow

    var body: some View {
        Button("Open message") {
            openWindow(value: message.id)
        }
    }
}
```

Be sure that the type you present conforms to both the `Hashable` and `Codable` protocols. Also, prefer lightweight data for the presentation value. For model values that conform to the `Identifiable` protocol, the value's identifier works well as a presentation type, as the above example demonstrates.

If a window with a binding to the same value that you pass to the `openWindow` action already appears in the user interface, the system brings the existing window to the front rather than opening a new window. If SwiftUI doesn't have a value to provide — for example, when someone opens a window by choosing File > New Window from the macOS menu bar — SwiftUI passes a binding to a `nil` value instead. To avoid receiving a `nil` value, you can optionally specify a default value in your window group initializer. For example, for the message viewer, you can present a new empty message:

```
WindowGroup(for: Message.ID.self) { $messageID in
    MessageDetail(messageID: messageID)
} defaultValue: {
    model.makeNewMessage().id // A new message that your model stores.
}
```

SwiftUI persists the value of the binding for the purposes of state restoration, and reapplies the same value when restoring the window. If the restoration process results in an error, SwiftUI sets the binding to the default value if you provide one, or `nil` otherwise.

Title your app's windows

To help people distinguish among windows from different groups, include a title as the first parameter in the group's initializer:

```
WindowGroup("Message", for: Message.ID.self) { $messageID in
    MessageDetail(messageID: messageID)
}
```

SwiftUI uses this title when referring to the window in:

- The list of new windows that someone can open using the File > New menu.
- The window's title bar.
- The list of open windows that the Window menu displays.

If you don't provide a title for a window, the system refers to the window using the app's name instead.

> **Note**
>
> You can override the title that SwiftUI uses for a window in the window's title bar and the menu's list of open windows by adding one of the `navigationTitle(_:)` modifiers to the window's content. This enables you to customize and dynamically update the title for each individual window instance.

Distinguish windows that present like data

To programmatically distinguish between windows that present the same type of data, like when you use a UUID as the identifier for more than one model type, add the id parameter to the group's initializer to provide a unique string identifier:

```
WindowGroup("Message", id: "message", for: UUID.self) { $uuid in
    MessageDetail(uuid: uuid)
}
WindowGroup("Account", id: "account-info", for: UUID.self) { $uuid in
    AccountDetail(uuid: uuid)
}
```

Then use both the identifer and a value to open the window:

```
struct ActionButtons: View {
    var messageID: UUID
    var accountID: UUID

    @Environment(\.openWindow) private var openWindow

    var body: some View {
        HStack {
            Button("Open message") {
                openWindow(id: "message", value: messageID)
            }
            Button("Edit account information") {
                openWindow(id: "account-info", value: accountID)
            }
        }
    }
}
```

Dismiss a window programmatically

The system provides people with platform-appropriate controls to dismiss a window. You can also dismiss windows programmatically by calling the dismiss action from within the window's view hierarchy. For example, you can include a button in the account detail view from the previous example that dismisses the view:

```
struct AccountDetail: View {
    var uuid: UUID?
    @Environment(\.dismiss) private var dismiss

    var body: some View {
        VStack {
            // ...

            Button("Dismiss") {
                dismiss()
            }
        }
    }
}
```

The dismiss action doesn't close the window if you call it from a modal — like a sheet or a popover — that you present from the window. In that case, the action dismisses the modal presentation instead.

Topics

Creating a window group

~~`init(content: () -> Content)`~~

Creates a window group.

Deprecated

~~`init(_:content:)`~~

Creates a window group with a text view title.

Deprecated

Identifying a window group

~~`init(id: String, content: () -> Content)`~~

Creates a window group with an identifier.

Deprecated

~~`init(_:id:content:)`~~

Creates a window group with a text view title and an identifier.

Deprecated

Creating a data-driven window group

`init<D, C>(for: D.Type, content: (Binding<D?>) -> C)`

Creates a data-presenting window group.

`init(_:for:content:)`

Creates a data-presenting window group with a text view title.

Providing default data to a window group

`init<D, C>(for: D.Type, content: (Binding<D>) -> C, defaultValue: () -> D)`

Creates a data-presenting window group with a default value.

`init(_:for:content:defaultValue:)`

Creates a data-presenting window group with a text view title and a default value.

Identifying a data-driven window group

`init<D, C>(id: String, for: D.Type, content: (Binding<D?>) -> C)`

Creates a data-presenting window group with an identifier.

`init(_:id:for:content:)`

Creates a data-presenting window group with a text view title and an identifier.

Identifying a window group that has default data

`init<D, C>(id: String, for: D.Type, content: (Binding<D>) -> C, defaultValue: () -> D)`

Creates a data-presenting window group with an identifier and a default value.

`init(_:id:for:content:defaultValue:)`

Creates a data-presenting window group with a text view title, an identifier, and a default value.

Supporting types

`struct PresentedWindowContent`

A view that represents the content of a presented window.

Initializers

`init(_:id:makeContent:)`

Creates a window group with a text view title and an identifier.

`init(_:makeContent:)`

Creates a window group with a text view title.

`init(id: String, makeContent: () -> Content)`

Creates a window group with an identifier.

`init(makeContent: () -> Content)`

Creates a window group.

Relationships

Conforms To

`Scene`

See Also

Creating windows

`struct Window`

A scene that presents its content in a single, unique window.

`struct UtilityWindow`

A specialized window scene that provides secondary utility to the content of the main scenes of an application.

`protocol WindowStyle`

A specification for the appearance and interaction of a window.

```
func windowStyle<S>(S) -> some Scene
```

Sets the style for windows created by this scene.