Class

UISplitViewController

A container view controller that implements a hierarchical interface.

iOS 3.2+ | iPadOS 3.2+ | Mac Catalyst 13.1+ | tvOS | visionOS 1.0+

```
@MainActor
class UISplitViewController
```

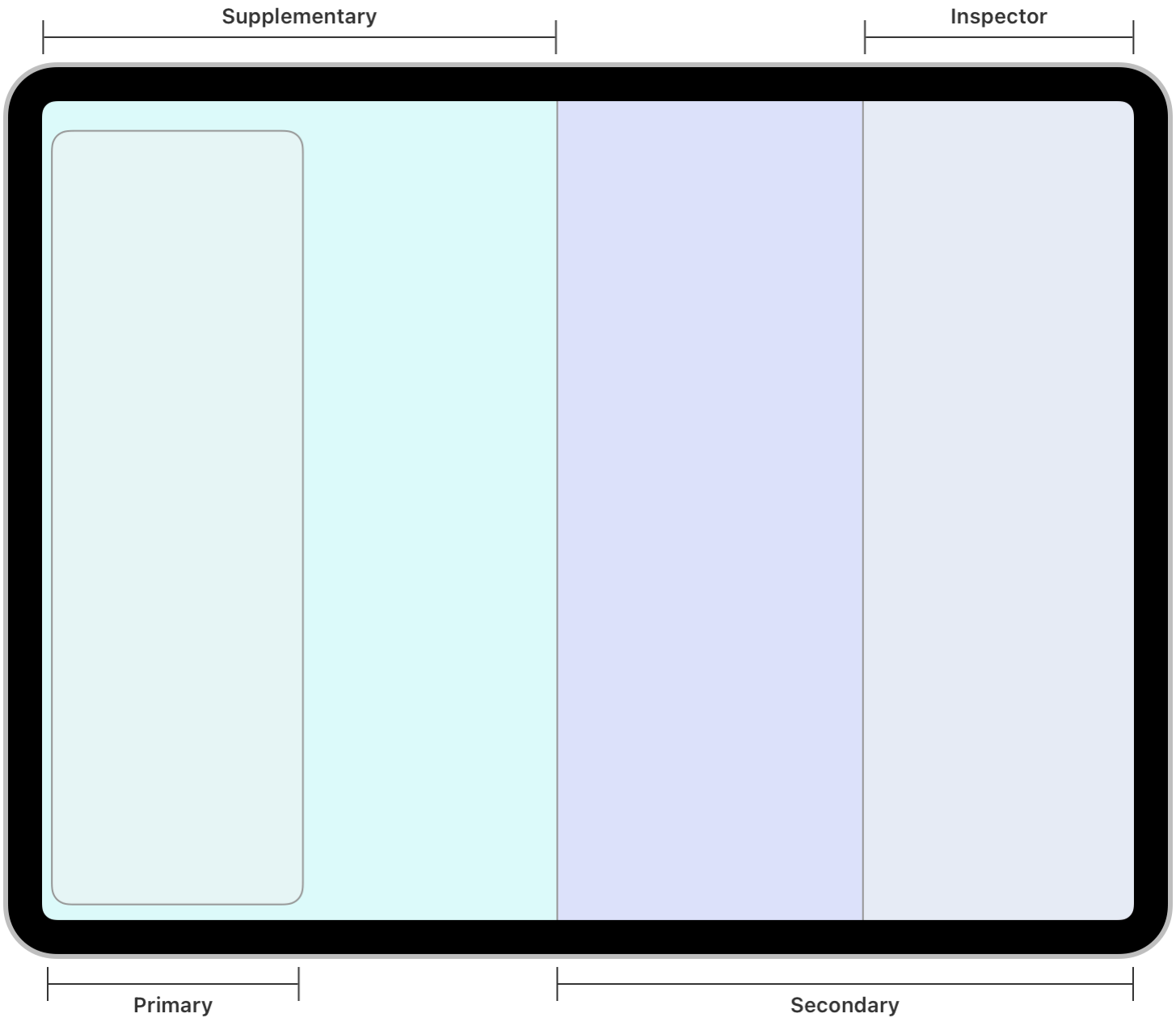
Mentioned in

- Managing content in your app's windows
- Creating a custom container view controller
- Customizing the behavior of segue-based presentations

Overview

A split view controller is a container view controller that manages child view controllers in a hierarchical interface. In this type of interface, changes in one view controller drive changes in the content of another.

Split view interfaces are most suitable for filterable content or navigating content hierarchies, like traversing the folders and notes within the Notes app to view each note. In the Notes app, selecting a folder in the primary sidebar shows the list of notes in that folder, and selecting a note from the list shows the contents of that specific note in the secondary view.



When you build your app's user interface, the split view controller is typically the root view controller of your app's window. The split view controller has no significant appearance of its own. Most of its appearance is defined by the child view controllers you install.

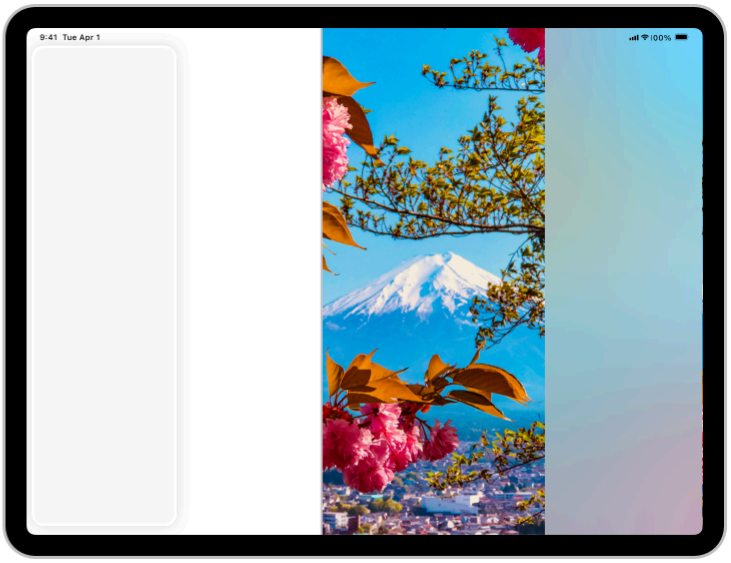
> **Note**
>
> You can't push a split view controller onto a navigation stack. Although it's possible to install a split view controller as a child in some other container view controllers, doing so isn't recommended in most cases. For design guidance, see Split views.

Split view styles

In iOS 14 and later, `UISplitViewController` supports column-style layouts. A column-style split view controller lets you create an interface with two or three columns by using `init(style:)` with the appropriate `style`:

- Use the `UISplitViewController.Style.doubleColumn` style to create a split view interface with a two-column layout. This style of split view controller manages two child view controllers, placed in the primary and secondary columns.
- Use the `UISplitViewController.Style.tripleColumn` style to create a split view interface with a three-column layout. This style of split view controller manages three child view controllers, placed in the primary, supplementary, and secondary columns.

Double column



Triple column

In either the two-column or three-column layout, `UISplitViewController` supports an inspector column on the trailing edge of the view. Use the inspector column to provide auxiliary information related to the secondary column, or for controls that affect content in the secondary column.

Before iOS 14, `UISplitViewController` supported just one split view interface style with a primary view controller and a secondary view controller. This classic interface style applies to split view controllers created using any other approach than `init(style:)`. Split view controllers with the classic interface have a `style` of `UISplitViewController.Style.unspecified` and they don't respond to any of the column-style APIs introduced in iOS 14 and later.

Child view controllers

In a column-style split view interface, use the `setViewController(_:for:)` and `viewController(for:)` methods to set and get view controllers for each column. The split view controller wraps all of its child view controllers in navigation controllers. If you set a child view controller that's not a navigation controller, the split view controller creates a navigation controller for it. The split view controller returns your original view controller through `viewController(for:)`, but its `children` property contains the navigation controller it used to wrap your view controller. After you assign view controllers to specific columns, you can show and hide those columns using `show(_:)` or `hide(_:)`.

In a classic split view interface, you can configure the child view controllers using Interface Builder or programmatically by assigning the view controllers to the `viewControllers` property. In cases where you need to change either the primary or secondary view controller, it's recommended that you do so using the `show(_:sender:)` and `showDetailViewController(_:sender:)` methods. Using these methods (instead of modifying the `viewControllers` property directly) lets the split view controller present the specified view controller in a way that's most appropriate for the current display mode and size class.

Interface transitions

The split view controller performs collapse and expand transitions in response to certain changes in its interface. For example, transitions occur when the interface's size class toggles between horizontally regular and horizontally compact, when a user interaction hides or shows a column, or when you hide or show columns programmatically. The split view controller works with its `delegate` object to perform collapse and expand transitions. The delegate is an object you provide that adopts the `UISplitViewControllerDelegate` protocol.

In a column-style split view interface, when the interface is collapsed, you can show a different view controller than your primary, supplementary, or secondary. Set the desired view controller for the `UISplitViewController.Column.compact` column using `setViewController(_:for:)`. If you want to further customize transitions for collapsing and expanding the interface, see Column-style split views.

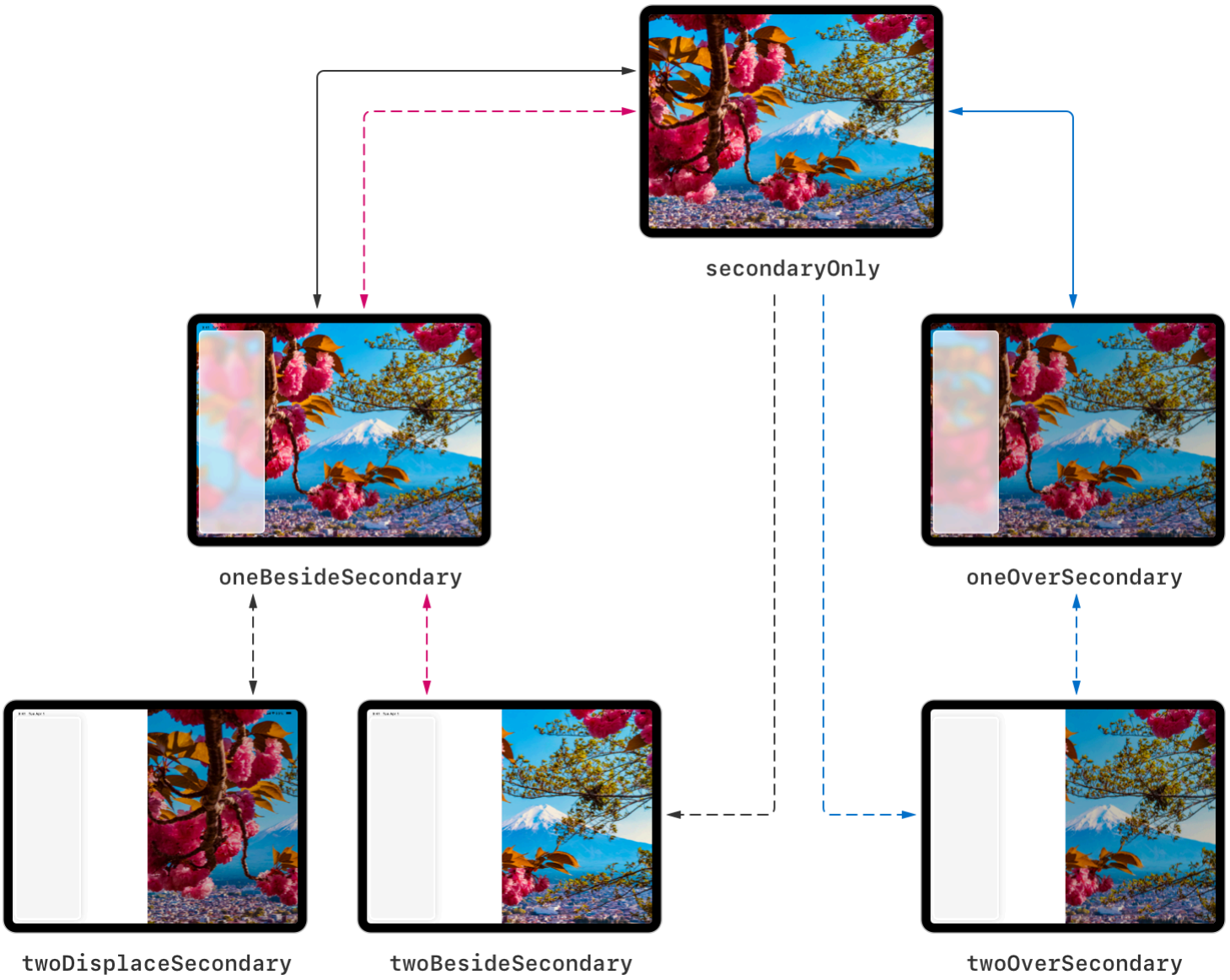
Configure your own custom views and interactions to show or hide the inspector column. When the interface is collapsed, the split view controller displays the inspector as a sheet over the secondary column.

For information about managing transitions in classic split view interfaces, see Classic split views.

Display mode

A split view controller's current display mode represents the visual arrangement of its child view controllers. It determines how many of its child view controllers are shown, and how they're positioned in relation to each other. For example, you can arrange the child view controllers so that they appear side-by-side, so that only one at a time is visible, or so that one is partially obscured by the others.

You don't set the display mode directly; instead, you set a preferred display mode by using the `preferredDisplayMode` property. The split view controller makes every effort to respect the display mode you specify, but it may not be able to accommodate that mode visually because of space constraints. For example, the split view controller can't display its child view controllers side-by-side in a horizontally compact environment. For possible configurations, see `UISplitViewController.DisplayMode`.



After you set the preferred display mode, the split view controller updates itself and reflects the actual display mode in the `displayMode` property. If you just want to change which columns are shown, try using `show(_:)` or `hide(_:)`. The split view controller will determine how to update the display mode to display the desired columns.

Gesture and button support

There are several ways for user interaction to change the current display mode.

The split view controller installs a built-in gesture recognizer that lets the user change the display mode using a swipe. You can suppress this gesture recognizer by setting the `presentsWithGesture` property to `false`. For example, you might set this property to `false` if you want your primary view controller to always be visible.

If `presentsWithGesture` is `true`, the split view controller also presents a special bar button item for changing the display mode. The split view controller manages the behavior, appearance, and positioning of this item. It appears as a sidebar toggle icon for `UISplitViewController.SplitBehavior.tile` and as a back-chevron icon for `UISplitViewController.SplitBehavior.overlay` and `UISplitViewController.SplitBehavior.displace`. Tapping

this button transitions to a new display mode based on the current display mode and split behavior.

For three-column split view interfaces—those with a `style` of `UISplitViewController.Style.tripleColumn`—another property that affects display mode is `showsSecondaryOnlyButton`. When this property is `true`, the split view controller presents another bar button item for toggling the display mode to and from `UISplitViewController.DisplayMode.secondaryOnly`. The split view controller manages the behavior, appearance, and positioning of this item. It appears as a double-arrow icon. When a user taps this button, it toggles the display mode to or from `UISplitViewController.DisplayMode.secondaryOnly`.

Split behavior

A split view controller's split behavior controls how its secondary view controller appears in relation to the others. You can configure this behavior so that the secondary view controller always appears side-by-side with the others, so that it's partially obscured by the others, or so that it's displaced offscreen opposite the others to make space for them.

You don't set the split behavior directly; instead, you set a preferred split behavior by using the `preferredSplitBehavior` property. This change takes effect after the next layout occurs. The split view controller reflects the actual split behavior in the `splitBehavior` property. The value of the `splitBehavior` property affects which display modes are available for the split view controller. For possible configurations, see `UISplitViewController.SplitBehavior`.

Column-width customization

You can specify custom widths for the primary, supplementary, secondary, and inspector columns of the split view interface by setting their respective minimum, maximum, and preferred width properties listed in Managing column dimensions. If you don't specify values for these properties, they default to `automaticDimension`.

Message forwarding

A split view controller interposes itself between the app's window and its child view controllers. As a result, all messages to the child view controllers must flow through the split view controller. Messages are forwarded as appropriate. For example, view appearance and disappearance messages are sent only when the corresponding child view controller actually appears onscreen.

Topics

Creating a split view controller

`init(style: UISplitViewController.Style)`

Creates a split view controller with the specified column style.

`init(nibName: String?, bundle: Bundle?)`

Creates a split view controller with the nib file in the specified bundle.

`init?(coder: NSCoder)`

Creates a split view controller from data in an unarchiver.

Getting the split view style

`var style: UISplitViewController.Style`

The style that determines the number of columns that the split view interface displays.

`enum Style`

Constants that describe the number of columns the split view interface displays.

Customizing the split view transitions

`var delegate: (any UISplitViewControllerDelegate)?`

The delegate you use to manage changes to a split view interface.

`protocol UISplitViewControllerDelegate`

The methods adopted by the object you use to manage changes to a split view interface.

Managing the child view controllers

`enum Column`

Constants that describe the columns within the split view interface.

`func setViewController(UIViewController?, for: UISplitViewController.Column)`

Presents the provided view controller in the specified column of the split view interface.

`func viewController(for: UISplitViewController.Column) -> UIViewController?`

Returns the view controller associated with the specified column of the split view interface.

`var viewControllers: [UIViewController]`

The array of view controllers the split view controller manages.

Displaying the child view controllers

`func show(UISplitViewController.Column)`

Presents the view controller in the specified column of the split view interface.

`func hide(UISplitViewController.Column)`

Dismisses the view controller in the specified column of the split view interface.

`func isShowing(UISplitViewController.Column) -> Bool`

A Boolean value that indicates whether the split view interface is showing the specified column.

`func show(UIViewController, sender: Any?)`

Presents the specified view controller as the primary view controller in the split view interface.

`func showDetailViewController(UIViewController, sender: Any?)`

Presents the specified view controller as the secondary view controller of the split view interface.

Managing the display mode

`var preferredDisplayMode: UISplitViewController.DisplayMode`

The preferred arrangement of the split view interface.

`var displayMode: UISplitViewController.DisplayMode`

The current arrangement of the split view interface.

`var displayModeButtonItem: UIBarButtonItem`

A button that changes the display mode of the split view controller.

`var presentsWithGesture: Bool`

Specifies whether a hidden view controller can be presented and dismissed using a swipe gesture.

`var showsSecondaryOnlyButton: Bool`

Specifies whether the secondary view controller shows a button to toggle to and from the secondary-only display mode.

`enum` `DisplayMode`

Constants that describe the possible arrangements for a split view interface.

`var` `displayModeButtonVisibility`: `UISplitViewController.DisplayModeButtonVisibility`

A setting that determines whether the display mode button is visible in the interface.

`enum` `DisplayModeButtonVisibility`

Constants that determine the visibility of the display mode button.

Managing the split behavior

`var` `preferredSplitBehavior`: `UISplitViewController.SplitBehavior`

The preferred behavior that determines how the child view controllers appear in relation to each other.

`var` `splitBehavior`: `UISplitViewController.SplitBehavior`

The current behavior that determines how the child view controllers appear in relation to each other.

`enum` `SplitBehavior`

Constants that describe the possible ways that the child view controllers appear in relation to each other.

Managing column dimensions

`var` `isCollapsed`: `Bool`

A Boolean value that indicates whether only one of the child view controllers displays.

`var` `preferredPrimaryColumnWidthFraction`: `CGFloat`

The relative width of the primary view controller’s content.

`var` `preferredPrimaryColumnWidth`: `CGFloat`

The preferred width, in points, of the primary view controller’s content.

`var` `primaryColumnWidth`: `CGFloat`

The width, in points, of the primary view controller’s content.

`var` `minimumPrimaryColumnWidth`: `CGFloat`

The minimum width, in points, for the primary view controller’s content.

`var maximumPrimaryColumnWidth: CGFloat`

The maximum width, in points, for the primary view controller's content.

`var preferredSupplementaryColumnWidthFraction: CGFloat`

The relative width of the supplementary view controller's content.

`var preferredSupplementaryColumnWidth: CGFloat`

The preferred width, in points, of the supplementary view controller's content.

`var supplementaryColumnWidth: CGFloat`

The width, in points, of the supplementary view controller's content.

`var minimumSupplementaryColumnWidth: CGFloat`

The minimum width, in points, for the supplementary view controller's content.

`var maximumSupplementaryColumnWidth: CGFloat`

The maximum width, in points, for the supplementary view controller's content.

`var preferredSecondaryColumnWidth: CGFloat`

The preferred width, in points, for the secondary view controller's content.

`var preferredSecondaryColumnWidthFraction: CGFloat`

The relative width of the secondary view controller's content.

`var minimumSecondaryColumnWidth: CGFloat`

The minimum width, in points, for the secondary view controller's content.

`var preferredInspectorColumnWidth: CGFloat`

The preferred width, in points, for the inspector view controller's content.

`var preferredInspectorColumnWidthFraction: CGFloat`

The relative width of the inspector view controller's content.

`var maximumInspectorColumnWidth: CGFloat`

The maximum width, in points, for the inspector view controller's content.

`var minimumInspectorColumnWidth: CGFloat`

The minimum width, in points, for the inspector view controller's content.

`class let automaticDimension: CGFloat`

The default value to apply to a dimension.

Inspecting the layout environment

`enum LayoutEnvironment`

Constants that indicate the current layout of the containing split view controller.

Positioning the primary view controller

`var primaryEdge: UISplitViewController.PrimaryEdge`

The side on which the primary view controller sits.

`enum PrimaryEdge`

Constants that indicate the side on which the primary view controller sits.

Managing the background style

`var primaryBackgroundStyle: UISplitViewController.BackgroundStyle`

The background style of the primary view controller.

`enum BackgroundStyle`

Styles that apply a visual effect to the background of a primary view controller.

Relationships

Inherits From

`UIViewController`

Conforms To

`CVarArg`
`CustomDebugStringConvertible`
`CustomStringConvertible`
`Equatable`
`Hashable`
`NSCoding`
`NSExtensionRequestHandling`

NSObjectProtocol
NSTouchBarProvider
Sendable
SendableMetatype
UIActivityItemsConfigurationProviding
UIAppearanceContainer
UIContentContainer
UIFocusEnvironment
UIPasteConfigurationSupporting
UIResponderStandardEditActions
UIStateRestoring
UITraitChangeObservable
UITraitEnvironment
UIUserActivityRestoring

See Also

Container view controllers

Creating a custom container view controller

Create a composite interface by combining content from one or more view controllers with other custom views.

`class` UINavigationController

A container view controller that defines a stack-based scheme for navigating hierarchical content.

`class` UINavigationBar

Navigational controls that display in a bar along the top of the screen, usually in conjunction with a navigation controller.

`class` UINavigationItem

The items that a navigation bar displays when the associated view controller is visible.

`class` UITabBarController

A container view controller that manages a multiselection interface, where the selection determines which child view controller to display.

`class` UITabBar

A control that displays one or more buttons in a tab bar for selecting between different subtasks, views, or modes in an app.

`class` `UITabBarItem`

An object that describes an item in a tab bar.

`class` `UITab`

An object that manages a tab in a tab bar.

`class` `UITabAccessory`

`class` `UISearchTab`

A tab subclass that represents the system's search tab.

`class` `UITabGroup`

An object that manages a collection of tab objects.

`class` `UIPageViewController`

A container view controller that manages navigation between pages of content, where a subview controller manages each page.