Class

NSTouchBar

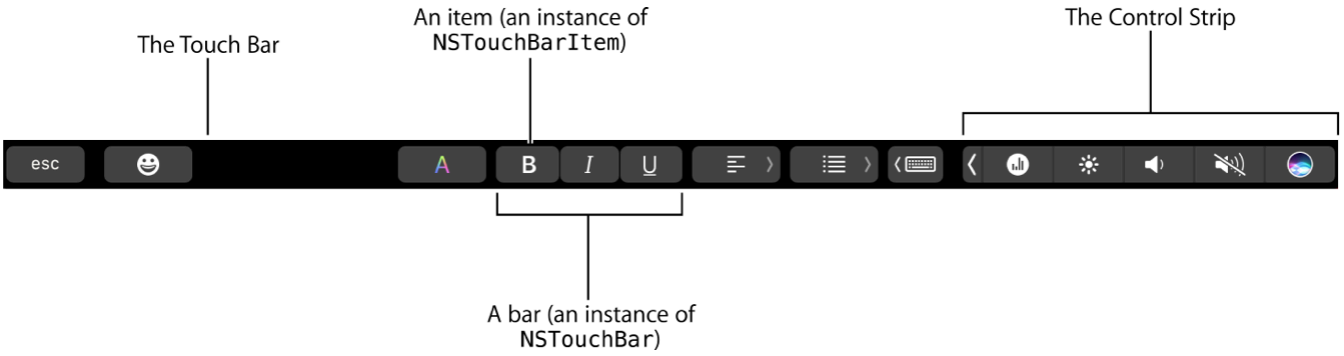
An object that provides dynamic contextual controls in the Touch Bar of supported models of MacBook Pro.

iOS 13.0+ | iPadOS 13.0+ | Mac Catalyst 13.1+ | macOS 10.12.2+

```
@MainActor
class NSTouchBar
```

Overview

On supported MacBook Pro models, the Touch Bar, above the keyboard, shows instances of the `NSTouchBar` class from the front-most app. Such an instance is called a *bar*. You define a bar to provide controls relevant to the user's context. Each such control is an instance of the `NSTouchBarItem` class, called an *item*.



You can provide many bars within your app, one for each responder instance; macOS frameworks can provide bars, as well, that can appear alongside your app's bars. The system determines which bars to show at any given time. For example, an app that uses standard AppKit objects, such as text fields (instances of the `NSTextField` class), obtains appropriate bars along with relevant items automatically.

Refer to the following sample code projects, which demonstrate how to use `NSTouchBar` and related classes, including the `NSScrubber` class, with its rich API that lets you build a highly customized picker control:

- Creating and Customizing the Touch Bar
- Integrating a Toolbar and Touch Bar into Your App

To use the Touch Bar, define bars in objects in your app's responder chain. At run time, the system traverses up the responder chain to discover, combine, and show bars from your app and from frameworks you link against.

You can configure a bar to support dynamic composition, in which the system shows the bar in an expanded form that contains items from bars lower in the responder chain (from closer to the first responder). Because of dynamic composition and placement of items shown on the Touch Bar, always ensure that your bars appear as you expect them to, testing on the versions of macOS that you support.

Instances of the `NSTouchBar` class employ gesture recognizers and take advantage of macOS 10.12.1 event enhancements. Because of the physical geometry of the Touch Bar, touch events passed to gesture recognizers have only a meaningful *x*, or horizontal, component.

There's no need, and no API, for your app to know whether or not there's a Touch Bar available. Whether your app is running on a machine that supports the Touch Bar or not, your app's onscreen user interface (UI) appears and behaves the same way.

The Touch Bar is a Retina display, like the screen of a MacBook Pro. To perform custom drawing or animation within the Touch Bar, follow the same best practices that you would on the screen.

> **Note**
>
> Although the Touch Bar is a display, it's first and foremost an input device. Don't use the Touch Bar to provide a display-only element.

On the right side of the Touch Bar, the system supplies the always-available *Control Strip*. The Control Strip gives the user access to standard controls for display brightness, sound volume, Siri, and so on. Your app's bars appear to the left of the Control Strip. The user can choose to hide the Control Strip, which gives the frontmost app the entire Touch Bar width.

To the right of the Control Strip is a Touch ID sensor. To use Touch ID on supported MacBook Pro models, use methods from the Local Authentication framework.

The Touch Bar dims automatically and wakes when the user touches it. Don't show alerts in the Touch Bar, and don't use the Touch Bar for widgets.

For Touch Bar design guidance, read Human Interface Guidelines.

Bar objects

You can think of an NSTouchBar object (or *bar*), with its array of NSTouchBarItem objects (or *items*), as analogous to a window toolbar with its toolbar items, or a menu with its menu items.

To provide a bar in your app, define it in an object that meets three requirements. The object that defines the bar must:

- Be a responder (an instance of an NSResponder subclass) that's present within a responder chain at runtime.
- Implement the makeTouchBar() delegate method from the NSTouchBarProvider protocol.

The built-in responder classes conform to the NSTouchBarProvider protocol and support key-value observing (KVO), both of which are used and required by the NSTouchBar infrastructure. In the context of Touch Bar support, a responder instance can also be called a *bar provider*.

The following code shows an example implementation of the makeTouchBar() delegate method. In this code snippet, you can see some statements related to bar customization.

```
override func makeTouchBar() -> NSTouchBar? {
    let mainBar = NSTouchBar()
    mainBar.delegate = self    mainBar.customizationIdentifier = .imageViewer
    mainBar.defaultItemIdentifiers = [.sharingPicker, .strokePopover, .strokeColorPi
    mainBar.customizationAllowedItemIdentifiers = [.strokeSlider, .strokePopover, .p
    mainBar.principalItemIdentifier = .photoPicker
    return mainBar
}
```

You can take advantage of built-in KVO support to keep track of bar state, such as which items are visible as the user customizes and interacts with the Touch Bar.

If you explicitly adopt the NSTouchBarProvider protocol in the app delegate or in a window delegate, you must also explicitly send the associated key-value observing notifications from within your implementations of NSTouchBar methods; this lets the system respond appropriately to changes in the bar. To avoid the need to manually support KVO, use the *app object* as a bar provider, instead of the app delegate, or use a *window controller* or *window* as a bar provider, instead of the associated window delegate.

To programmatically invalidate a bar associated with a bar provider, such as because you're changing the bar's state, set its touchBar property to a value of nil.

Item objects

A bar itself (an `NSTouchBar` object) has no visible representation in a MacBook Pro Touch Bar. A user instead sees the bar's items, each of which is an instance of the `NSTouchBarItem` class.

> **Note**
>
> This section explains how to use items in bars. For details on the various kinds of items and how to create and configure them, see `NSTouchBarItem`.

The items presented in a bar are the elements in a private array owned by the bar. To specify the items for a bar, you don't fill this array directly, but rather rely on the bar to manage its items based on various groups of items and item identifiers that you do specify directly.

In specifying items for a bar you have two options, giving you flexibility for optimizing resource use and efficiency in your app.

- The `templateItems` property is a set that you can directly populate with item instances for a bar. Use this option when your items are lightweight enough to stay in memory for the duration of your app's lifetime, and when they don't contain state that might change over time.
- The `NSTouchBarDelegate` protocol, and its `touchBar( :makeItemForIdentifier:)` delegate method, give your app a way to create items on-demand. Use this option when it makes more sense in terms of resource usage and reflecting dynamic state.

Whichever of these two approaches you employ, the system is in charge of populating a bar's private items array based on three things:

1. Your configuration of the bar's item-identifiers properties
2. Any nesting you have specified
3. Any customization that the user has specified for the bar

As your app runs, you can obtain the identifiers of the items eligible for presentation in a bar — specifically, those in its private items array — by accessing the read-only `itemIdentifiers` property. This property reflects the current state of the bar instance, including any customization that has been performed by the user and any dynamic composition that has been performed by the system.

Customization

AppKit provides a rich Touch Bar customization facility for users that appears, upon user request, on the main display. Make your bars customizable unless you have a specific UI need not to do so.

A customizable bar automatically obtains onscreen UI which lets the user:

- Change which items are part of the shown bar

- Rearrange items within the shown bar

A user invokes the onscreen customization UI by choosing a dedicated menu item.

To make an NSTouchBar object eligible for customization, assign it a globally-unique customizationIdentifier identifier. For the identifier string, use reverse-DNS style, such as "com.company-name.app-name.alphanumeric-ID".

Next, specify the bar's items and customization possibilities by populating its item identifier lists. Each such list is an array, each of whose elements is the identifier (of type NSTouchBarItem.Identifier) for an item (an NSTouchBarItem object). A bar's item identifier lists are:

Default item identifiers, specified in a bar's defaultItemIdentifiers property. Always specify this property for an NSTouchBar object, even if you elect to make the bar noncustomizable. The system:

- Shows this list's items by default when the system displays the bar.
- Includes a preconfigured bar representation, containing these items, in the associated customization UI (when you have designated the bar as customizable by assigning it a customizationIdentifier property value); the user can drag the default bar into the Touch Bar, should they want to return to the default configuration.

Additional item identifiers, specified in a bar's customizationAllowedItemIdentifiers property. Always configure this property for a customizable bar. The system uses this list by showing representations of its items individually in the customization UI, arranged in the same order as you specify in the property array. When there's available geometric space, a user can drag in to the active bar any of the items in this list. If there isn't enough space, a dragged item replaces the item or items under the spot the new item is dropped.

Required item identifiers, specified in a bar's customizationRequiredItemIdentifiers property. Configure this property at your discretion, depending on the design of your app. The user can't remove from the bar any of the items you specify in this list.

To provide textual labels in the customization UI, use the customizationLabel property on each NSTouchBarItem instance you include in a customizable bar. The accessibility system in macOS also makes use of these labels.

If your app design requires a noncustomizable NSTouchBar object:

- List all of the bar's items in the defaultItemIdentifiers property, and only in this property.
- Don't use the other properties described in this section and, in particular, don't assign the bar a customizationIdentifier property value.

Group item, popover item, and composed bar customization

AppKit lets you specify any bar as customizable or not. The customization configuration you provide for a bar remains associated with the bar and its items — even when those items are nested by the system into another bar higher in the responder chain.

The system also respects your bar customization configuration when you use group and popover items. Each of these item types itself contains one or more bars—which can, in turn contain group items and popover items, and so on. The rest of this section explains how customization works for the bars containing, and bars within, group items and popover items.

A group item (an instance of the `NSGroupTouchBarItem` class) has one bar, held in the object's `groupTouchBar` property. AppKit supports nesting of group items, in that you can configure a `groupTouchBar` bar to itself contain one or more group items (or, for that matter, items of any other type, guided by what works well in your app).Here are some examples of how customization for group items works in practice:

- If you configure a bar as customizable, and give it a group item whose bar you configure as *not* customizable, then the array of items in the `groupTouchBar` bar appears in the customization UI as an atomic unit. During customization, a user can manipulate the array of items, but strictly as a unit: If the (noncustomizable) `groupTouchBar` bar is visible in the Touch Bar, the user can remove it as a unit, or can rearrange it among the other items in the `groupTouchBar` bar; if the `groupTouchBar` bar is instead visible in the customization UI, the user can add it back to the Touch Bar, as a unit, placing it within the bar that owns the group item.
- If you configure a bar as *not* customizable, and give it a group item whose bar you configure as customizable, then the `groupTouchBar` bar's items appear in the customization UI as individual items. During customization, a user can manipulate each item separately: If an item from the `groupTouchBar` bar is visible in the Touch Bar, the user can remove it or can rearrange its position individually among the other items in the `groupTouchBar` bar; if an item from the `groupTouchBar` bar is instead visible in the customization UI, the user can add it back to the Touch Bar, individually, placing it anywhere within the `groupTouchBar` bar that owns it.

A popover item (an instance of the `NSPopoverTouchBarItem` class) has two bars: one bar you specify in its `popoverTouchBar` property and a second, optional bar you can specify in its `pressAndHoldTouchBar` property. Here are some examples of how customization for popover items works in practice:

- If you configure a bar as customizable, and give it a popover item whose `popoverTouchBar` bar you configure as *not* customizable, the `popoverTouchBar` bar never appears in the customization UI. If the user invokes the customization UI when the (noncustomizable) popover item itself (not the button's associated `popoverTouchBar` bar) is visible in the Touch Bar, the customization UI lets the user rearrange the position of the popover item relative to the other items in the containing bar. If, on the other hand, the user invokes the customization UI when the (noncustomizable) `popoverTouchBar` bar is visible in the Touch Bar, the system dismisses the popover bar and shows, in the customization UI, customization options for the bar that contains the popover item.

- If you configure a `popoverTouchBar` bar as customizable, the user can invoke the `popoverTouchBar` bar (by tapping the popover item that owns it, in the Touch Bar) and then use the customization UI to manipulate the items in the `popoverTouchBar` bar itself.

Customization menu item

A user invokes the customization UI for a particular `NSTouchBar` object, when it's visible in the Touch Bar, by choosing the bar customization menu item. To enable this menu item you must explicitly opt-in, which you can do in the following ways:

- If you want the system to automatically name, place, validate, and activate this menu item in your app's menus, set the `isAutomaticCustomizeTouchBarMenuItemEnabled` property of your app object (of type `NSApplication`) to `true`.
- To explicitly place the customization menu item in one of your app's menus, employ the `toggleTouchBarCustomizationPalette(_:)` method of your app object. When you do this, the system still names and validates the menu item, and hides it on systems that don't have a Touch Bar.

If you attempt to employ the customization menu item (using either of these two approaches), but do not provide a customization identifier property (`customizationIdentifier`) for a bar, the customization menu item appears when that bar is active — but the menu item, in this case, is disabled.

If your app attempts to use both automatic and explicit placement of the customization menu item, the system respects your explicit control and doesn't place the item automatically.

Layout

The user controls the width of the Control Strip and can choose to hide it, and the system is in charge of the nesting of `NSTouchBar` instances (for the bars you make eligible for composition). As a result, the available display width for your bars can vary. There's no API for you to obtain the current available display width.

In your layout design, don't depend on a particular Control Strip size. Do anticipate dynamic composition and nesting for your bars.

If you need more horizontal space than might be available, use a popover item, a scrubber, or a scroll view — as they fit your design needs, but in that, descending, order of preference.

In geometric-space-constrained scenarios, the system hides `NSTouchBarItem` instances according to their visibility priority.

If you need to center an item in the Touch Bar, designate it as a *principal item* by assigning it to its bar's `principalItemIdentifier` property. Don't hard-code spacing in an attempt to ensure

an item is centered. If you want a group of items to appear centered in the Touch Bar, designate the group item (of type `NSGroupTouchBarItem`) as the principal item.

Composition and nesting

You can configure a bar to support dynamic composition, in which the system shows the bar in an expanded form that contains items from bars lower in the responder chain (closer to the first responder).

To allow a bar to serve as a container for nesting, add the `otherItemsProxy` item identifier to the bar's `defaultItemIdentifiers` array. A bar that includes this identifier, and that's relatively higher in the responder chain, can then (at runtime) include the items from an eligible bar relatively lower in the responder chain.

The position that you specify for the other-items proxy, within a bar's `defaultItemIdentifiers` array, tells the system where you want nested items to be placed.

The system determines whether or not to compose bars in this way, based on system policy and available geometric space in the Touch Bar.

`NSTouchBar` object nesting can be chained, according to available geometric space in the Touch Bar. For example, a view and a text field within that view could each contribute their items to the bar defined for a parent window controller.

When the system nests one bar's items into another bar higher in the responder chain, the items appear to the user, in the Touch Bar, as fully incorporated into the higher bar. There's no visual boundary or additional spacing to distinguish the items as being nested.

If a bar doesn't employ the `otherItemsProxy` identifier, the system hides that bar when another bar, lower in the responder chain, is eligible for display.

When determining which items to show in the Touch Bar for the current first responder, the system traverses up the entire responder chain. This lets the system accommodate any proxy items in bars defined for objects higher in the chain, thereby respecting the fact that any bar, defined for an object at any position in the responder chain, might include the other-items-proxy identifier.

Customization for composed bars

The logical, geometric boundary for a nested `NSTouchBar` object isn't visible to the user in the Touch Bar. However, the boundary remains in effect in terms of customization. A user can't rearrange a nested bar's items outside of its boundary.

For example, say you have a bar, higher in the responder chain, configured like this:

```
[(1)(2)(other-items-proxy)(3)]
```

And say you also have a bar, lower in the responder chain, eligible for display in the Touch Bar according to the system and the current app state, configured like this:

```
[(A)(B)(C)]
```

The composed bar would correspond to this arrangement in the Touch Bar:

```
[(1)(2)(A)(B)(C)(3)]
```

With the customization UI, the user could then rearrange the items represented here by `(A)`, `(B)`, and `(C)`, but only as long as those items remained contiguous during the rearrangement, thereby respecting the logical boundary of the bar that defines them.

Item spacing for composed bars

When the system nests `NSTouchBar` objects that include spacing items, it merges any resulting adjacent spacing. Ensure that your bars appear as you expect them to, testing on the versions of macOS that you support. For more on spacing items, see `NSTouchBarItem`.

Bar discovery and the responder chain

At runtime, the system traverses up the responder chain, starting at the object with focus, to discover objects that conform to the `NSTouchBarProvider` protocol. Such objects are called *bar providers*. The system then populates the Touch Bar, potentially with multiple, nested bars, according to system policy and available geometric space.

Specifically, bar discovery by the system proceeds in the following order:

1. Key window's first responder
2. Key window
3. Key window's delegate
4. Key window's controller
5. Main window's first responder
6. Main window
7. Main window's delegate
8. Main window's controller
9. App object

0. App delegate

When the system encounters a bar provider that's an instance of an `NSResponder` subclass, the system then additionally searches up the responder chain anchored at that object.

For example, in a complicated but otherwise standard app, bar discovery might proceed in this order:

1. Key window's first responder
2. View controller of key window's first responder
3. Intermediate view controllers and views
4. View that's closest to root of window
5. View controller that's closest to root of window
6. Key window
7. Key window's controller
8. App object
9. App delegate

The Touch Bar can show one bar nested within another, as described in Composition and nesting.

Accessibility and the Touch Bar

AppKit views and controls adopt the `NSAccessibilityProtocol` protocol and automatically send appropriate accessibility notifications. Because the Touch Bar is designed to work with AppKit, it's fully accessible.

Be sure to use the `customizationLabel` property on every `NSTouchBarItem` instance that you designate as customizable. The accessibility system in macOS makes use of these labels.

To learn more about accessibility, read Accessibility for AppKit.

AppKit support for the Touch Bar

To support the Touch Bar feature, AppKit provides several enhancements, first available in macOS 10.12.1:

- **Scrubbers.** The `NSScrubber` class, along with related APIs, provide a way for you to add a flexible, horizontally-oriented picker to a custom item (an instance of the `NSCustomTouchBarItem` class).
- **Gesture recognizer support.**

- You can use the NSMagnificationGestureRecognizer class in bar items. To enable two-finger pinch gestures, set the recognizer's allowedTouchTypes mask property, on the gesture recognizer, to the NSTouch.TouchType.direct constant from the NSTouch.TouchTypeMask enumeration.
- The NSGestureRecognizer abstract class is enhanced with a set of methods that let you implement responses to touch events: touchesBegan(with:), touchesCancelled(with:), touchesEnded(with:), and touchesMoved(with:).
- The NSClickGestureRecognizer, NSPanGestureRecognizer, and NSPressGestureRecognizer concrete classes are each enhanced with a numberOfTouchesRequired property to let you specify the number of touches required for a gesture match.
- **Touch type changes.** To enable touch events in a custom view, you must set the value of a view's allowedTouchTypes property to a value of direct. (In macOS 10.12.1, the acceptsTouchEvents property is deprecated in favor of the new allowedTouchTypes property.)
- **Touch changes.** The NSTouch class has a new property and two new methods for supporting the Touch Bar: NSTouch.TouchType, location(in:), and previousLocation(in:).
- **Control appearance support.** The NSButton, NSSegmentedControl, and NSSlider classes are each enhanced with appearance support properties: bezelColor for buttons, selectedSegmentBezelColor for segmented controls, and trackFillColor for sliders. (With the Touch Bar, you employ sliders indirectly, as used by slider items.)
- **Convenience initializers.** Starting in macOS 10.12, you can use a rich set of convenience initializers for controls. These initializers simplify the definition of bar items and take care of appearance and sizing for the Touch Bar. In particular, the NSButton, NSSegmentedControl, and NSSlider classes now offer a variety of convenience initializers such as init(title:image:target:action:).
- **Text support.** Methods and properties in the NSSpellChecker, NSTextField, and NSTextView classes, and in the NSTextFieldDelegate and NSTextViewDelegate protocols, support using the Touch Bar for spell checking, predictive text suggestion, text completion, and automatic handling of trailing space. For example:
- When you use an NSTextView object, you gain automatic Touch Bar support for text styling and predictive text suggestions.
- When you use an NSCandidateListTouchBarItem object, you can use the requestCandidates(forSelectedRange:in:types:options:inSpellDocumentWithTag:completionHandler:) method. This method provides a completion handler that you can use to filter or otherwise manage the candidate text.
- **New template images.** AppKit adds many new template images for you to use in your NSTouchBarItem objects. A few examples of these images are: touchBarAddTemplateName, touchBarComposeTemplateName, touchBarGoBackTemplateName, touchBarGoForward

TemplateName, and NSImageNameTouchBarHomeTemplate. Always use templates for images in your items: they respond automatically to system white-point changes. Note that these images are exclusively for use in the Touch Bar and *not* in onscreen windows. For a complete list of these template images, see NSTouchBarItem.

- **Scroll views.** If your UI for a popover item needs more horizontal space, you can use a scroll view (an instance of the NSScrollView class). *Don't*, in this case, enable the popover item's press-and-hold option, because doing so interferes with scrolling.
- **Stack views.** You can group bar items by using an instance of the NSStackView class. However, doing so loses system support for spacing. When you instead place items into a group item (an instance of the NSGroupTouchBarItem class), the system:
- Manages inter-item spacing
- Supports user customization for the individual items

Development considerations for the Touch Bar

The Xcode Touch Bar simulator represents the Touch Bar onscreen and supports some user interaction. However, some interactions are unavailable in the simulator. For example, you can't perform two-finger gestures in the Touch Bar simulator.

If you're adopting Touch Bar support for your app, but running Xcode on a Mac without a Touch Bar, you can enable the Xcode Touch Bar simulator by choosing Window > Show Touch Bar in Xcode.

Interface Builder supports development for the Touch Bar with nib objects available in the object library. Drag and drop bars and items from the object library into your canvas, and then attach them to your app's responders as desired. For more information, see Xcode Help.

Performance considerations for the Touch Bar

The Touch Bar's display and the MacBook screen share resources, including the main CPU and GPU of the MacBook. To ensure that your Touch Bar controls perform well, follow the usual best practice of protecting your app's main thread from doing too much work. For example, don't perform rendering work for the main display on the main thread.

In addition, pay attention to the relative amounts of time your app spends on updates to the main display relative to updates to the Touch Bar. The optimum ratio can vary according to what the user is doing. For example:

- When a user is interacting with the Touch Bar *and* watching the Touch Bar, such as to control audio, ensure that your app gives priority to updating the Touch Bar.

- When a user is instead interacting with the Touch Bar but watching the main display, such as when using a scrubber to browse pages of content, balance your app's display update work between the Touch Bar and the main display.

Always test Touch Bar performance using the specific MacBook hardware you support. Specifically, don't rely on the Xcode Touch Bar simulator when tuning your app for Touch Bar performance.

Topics

Creating a bar

`init()`

Creates a Touch Bar object.

`init?(coder: NSCoder)`

Creates a Touch Bar object from a coder object provided by a storyboard or NIB file.

Providing bar items

`var delegate: (any NSTouchBarDelegate)?`

The delegate that provides items to the Touch Bar.

`var templateItems: Set<NSTouchBarItem>`

The primary source of items that the Touch Bar uses to fill its private items array, unless you provide items using a delegate.

`var defaultItemIdentifiers: [NSTouchBarItem.Identifier]`

A required list of identifiers for items that you want to appear in the Touch Bar after instantiating it.

`var principalItemIdentifier: NSTouchBarItem.Identifier?`

The identifier of an item you want the system to center in the Touch Bar.

`var escapeKeyReplacementItemIdentifier: NSTouchBarItem.Identifier?`

The identifier of an item that replaces the system-provided button in the Touch Bar.

Observing bar status

`var isVisible: Bool`

A Boolean value that Indicates whether the Touch Bar is eligible for display.

`var itemIdentifiers: [NSTouchBarItem.Identifier]`

The list of identifiers for the current items in the Touch Bar.

`func item(forIdentifier: NSTouchBarItem.Identifier) -> NSTouchBarItem?`

Returns the Touch Bar item that corresponds to a given identifier.

Configuring user customization

`var customizationIdentifier: NSTouchBar.CustomizationIdentifier?`

A globally unique string that makes the Touch Bar eligible for user customization.

`var customizationAllowedItemIdentifiers: [NSTouchBarItem.Identifier]`

A list of identifiers for items to show in the Touch Bar's customization UI.

`var customizationRequiredItemIdentifiers: [NSTouchBarItem.Identifier]`

An optional list of identifiers for items you want to always appear in the Touch Bar and which the user can't remove during customization.

`typealias CustomizationIdentifier`

The default type for a Touch Bar customization identifier.

`class var isAutomaticCustomizeTouchBarMenuItemEnabled: Bool`

A Boolean value indicating whether the main menu contains an item for customizing the contents of the Touch Bar.

Relationships

Inherits From

`NSObject`

Conforms To

`CVarArg`
`Copyable`
`CustomDebugStringConvertible`
`CustomStringConvertible`

`Equatable`
`Hashable`
`NSCoding`
`NSObjectProtocol`
`Sendable`

See Also

Essentials

{} Integrating a Toolbar and Touch Bar into Your App

Provide users quick access to your app's features from a toolbar and corresponding Touch Bar.

{} Creating and Customizing the Touch Bar

Adopt Touch Bar support by displaying interactive content and controls for your macOS apps.

`protocol NSTouchBarDelegate`

A protocol that allows you to provide the items for a bar dynamically.

`protocol NSTouchBarProvider`

A protocol that an object adopts to create a bar object in your app.