Sample Code

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.

Download

macOS 10.13+ | Xcode 12.0+

Overview

The *toolbar* appears in the space immediately below or next to a window's title bar and above the app's content. You use an `NSToolbar` object to manage the items that appear in a toolbar, which are `NSToolbarItem` objects created in Interface Builder or source code.

This sample shows you how to add a toolbar to a window and add Touch Bar support that works in conjunction with the toolbar.

Add the Toolbar

The sample defines the custom class `WindowController`, which derives from `NSWindowController` and conforms to the `NSToolbarDelegate` protocol. In Interface Builder, the sample creates an instance of `WindowController` in the Window Controller Scene of `Main.storyboard`, and adds an `NSToolbar` object to the `window` referenced by the controller. Then the sample connects the toolbar's `delegate` to the `WindowController` instance.

The window controller also defines an outlet variable `toolbar`, making it possible to reference the toolbar instance in the source code of `WindowController.swift`.

```
@IBOutlet weak var toolbar: NSToolbar!
```

Create Toolbar Item Identifiers

Each `NSToolbarItem` object has a unique identifier of type `NSToolbarItem.Identifier`. AppKit provides identifiers for standard toolbar items such as cloud sharing, printing, and showing the font and color palette. For custom toolbar items, the app provides the identifiers. For example, the sample provides two identifiers for its custom toolbar items: one for setting the font size and one for setting the font style of an `NSTextView`.

```
private extension NSToolbarItem.Identifier {
    static let fontSize: NSToolbarItem.Identifier = NSToolbarItem.Identifier(rawValu
    static let fontStyle: NSToolbarItem.Identifier = NSToolbarItem.Identifier(rawVa
}
```

Specify Allowed Toolbar Items

To tell the toolbar which items are available, the sample's toolbar delegate implements the `toolbarAllowedItemIdentifiers(_:)` method, which returns an array of item identifiers. The available items appear in the toolbar's customization palette for customizing the toolbar when running the sample app.

```
func toolbarAllowedItemIdentifiers(_ toolbar: NSToolbar) -> [NSToolbarItem.Identifie
    return [ NSToolbarItem.Identifier.fontStyle,
             NSToolbarItem.Identifier.fontSize,
             NSToolbarItem.Identifier.space,
             NSToolbarItem.Identifier.flexibleSpace,
             NSToolbarItem.Identifier.print ]
}
```

Specify Default Toolbar Items

When the sample app launches for the first time, a default set of items appear in the toolbar. The sample provides these items by implementing the `toolbarDefaultItemIdentifiers(_:)` delegate method, which returns an array containing the font style and font size item identifiers.

```
func toolbarDefaultItemIdentifiers(_ toolbar: NSToolbar) -> [NSToolbarItem.Identifie
    return [.fontStyle, .fontSize]
}
```

Create a Toolbar Item from an Identifier

The toolbar asks its delegate to create a toolbar item by calling the `toolbar(_:itemForItemIdentifier:willBeInsertedIntoToolbar:)` method. It calls this method when adding an item to the toolbar but also when adding items to the toolbar's customization palette.

The sample app's implementation of this method creates items for the two identifiers created in the app, font style and font size.

```
func toolbar(
    _ toolbar: NSToolbar,
    itemForItemIdentifier itemIdentifier: NSToolbarItem.Identifier,
    willBeInsertedIntoToolbar flag: Bool) -> NSToolbarItem? {

    var toolbarItem: NSToolbarItem?

    /** Create a new NSToolbarItem instance and set its attributes based on
        the provided item identifier.
     */

    if itemIdentifier == NSToolbarItem.Identifier.fontStyle {
        // 1) Font style toolbar item.
        toolbarItem =
            customToolbarItem(itemForItemIdentifier: NSToolbarItem.Identifier.fontSt
                              label: NSLocalizedString("Font Style", comment: ""),
                              paletteLabel: NSLocalizedString("Font Style", comment:
                              toolTip: NSLocalizedString("tool tip font style", comm
                              itemContent: styleSegmentView)!
    } else if itemIdentifier == NSToolbarItem.Identifier.fontSize {
        // 2) Font size toolbar item.
        toolbarItem =
            customToolbarItem(itemForItemIdentifier: NSToolbarItem.Identifier.fontSi
                              label: NSLocalizedString("Font Size", comment: ""),
                              paletteLabel: NSLocalizedString("Font Size", comment:
                              toolTip: NSLocalizedString("tool tip font size", comme
                              itemContent: fontSizeView)!
    }

    return toolbarItem
}
```

The delegate isn't responsible for creating toolbar items for standard identifiers; AppKit creates those toolbar items.

Use a Custom View for a Toolbar Item

The font style and font size toolbar items display a custom view that the sample defines in `Main.storyboard` and connects with outlets in `WindowController`.

```
// Font style toolbar item.
@IBOutlet var styleSegmentView: NSView! // The font style changing view (ends up in

// Font size toolbar item.
@IBOutlet var fontSizeView: NSView! // The font size changing view (ends up in an NS
```

When the toolbar calls the delegate method `toolbar(_:itemForItemIdentifier:willBeInsertedIntoToolbar:)`, the sample creates the toolbar item by calling its helper function `customToolbarItem`, passing in the custom view for the requested toolbar item.

```
func customToolbarItem(
    itemForItemIdentifier itemIdentifier: String,
    label: String,
    paletteLabel: String,
    toolTip: String,
    itemContent: AnyObject) -> NSToolbarItem? {

    let toolbarItem = NSToolbarItem(itemIdentifier: NSToolbarItem.Identifier(rawValu

    toolbarItem.label = label
    toolbarItem.paletteLabel = paletteLabel
    toolbarItem.toolTip = toolTip
    toolbarItem.target = self

    // Set the right attribute, depending on if we were given an image or a view.
    if itemContent is NSImage {
        if let image = itemContent as? NSImage {
            toolbarItem.image = image
        }
    } else if itemContent is NSView {
        if let view = itemContent as? NSView {
            toolbarItem.view = view
        }
    } else {
        assertionFailure("Invalid itemContent: object")
    }
```

```
    // We actually need an NSMenuItem here, so we construct one.
    let menuItem: NSMenuItem = NSMenuItem()
    menuItem.submenu = nil
    menuItem.title = label
    toolbarItem.menuFormRepresentation = menuItem

    return toolbarItem
}
```

Add More Attributes to a Toolbar Item

The sample's toolbar delegate also implements the `toolbarWillAddItem(_:)` method to know when the toolbar is about to add an item. This gives the delegate the opportunity to add or change state information for the item. For example, the sample sets the `toolTip` property of a toolbar item with the `print` identifier.

```
func toolbarWillAddItem(_ notification: Notification) {
    let userInfo = notification.userInfo!
    if let addedItem = userInfo["item"] as? NSToolbarItem {
        let itemIdentifier = addedItem.itemIdentifier
        if itemIdentifier == .print {
            addedItem.toolTip = NSLocalizedString("print string", comment: "")
            addedItem.target = self
        }
    }
}
```

Provide Toolbar Customization to Users

The toolbar displays the default items in the same order as they appear in the array that `toolbarDefaultItemIdentifiers(_:)` returns. But users can rearrange the items, add and remove items, and reset the toolbar to its default items by selecting View > Customize Toolbar, which displays the toolbar's configuration palette. The sample app also saves the changes and reapplies them the next time the user launches the app.

The sample app provides these behaviors by setting the toolbar's `allowsUserCustomization` and `autosavesConfiguration` properties to `true`.

```
/*  If you pass false here, you turn off the customization palette. The
    NSWindow method -runToolbarCustomizationPalette: handles the display
    of the palette, which you can see in Interface Builder is connected
```

```
    to the"Customize Toolbar" menu item.
 */
toolbar.allowsUserCustomization = true

/*  Tell the toolbar that it should save any toolbar configuration changes
    to user defaults, that is, persist any mode changes or item reordering.
    The app writes the configuration to the app domain using the toolbar
    identifier as the key.
 */
toolbar.autosavesConfiguration = true
```

Add Touch Bar Support

The sample app provides Touch Bar support that works in conjunction with the toolbar. Like the toolbar, the app provides two items on the Touch Bar: font style and font size.

Before the sample app can add items to the Touch Bar, it needs to create an `NSTouchBar` object. The sample does this by overriding the `makeTouchBar()` method, where it creates and returns a new Touch Bar object.

Then, similar to adding toolbar items, the sample implements the `NSTouchBarDelegate` method `touchBar( _:makeItemForIdentifier:)`, where it creates and returns Touch Bar items for the specified identifier.

> **Note**
>
> It's possible to simulate a Touch Bar from Xcode by choosing Window > Show Touch Bar.

```
override func makeTouchBar() -> NSTouchBar? {
    let touchBar = NSTouchBar()
    touchBar.delegate = self
    touchBar.customizationIdentifier = .touchBar
    touchBar.defaultItemIdentifiers = [.fontStyle, .popover, NSTouchBarItem.Identifi
    touchBar.customizationAllowedItemIdentifiers = [.fontStyle, .popover]

    return touchBar
}
```

Provide Touch Bar Customization to Users

As with the toolbar, the sample app lets users customize items in the Touch Bar by choosing View > Customize Touch Bar. The app provides this feature by setting the `isAutomaticCustomizeTouchBarMenuItemEnabled` property to `true`.

```
func applicationDidFinishLaunching(_ aNotification: Notification) {

    // Allow users to customize the app's Touch Bar items.
    NSApplication.shared.isAutomaticCustomizeTouchBarMenuItemEnabled = true

}
```

See Also

Essentials

{} Creating and Customizing the Touch Bar

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

`class` NSTouchBar

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

`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.