Sample Code

Adding menus and shortcuts to the menu bar and user interface

Provide quick access to useful actions by adding menus and keyboard shortcuts to your Mac app built with Mac Catalyst.

Download

iOS 13.0+ | iPadOS 13.0+ | Mac Catalyst 13.0+ | Xcode 13.1+

Overview

This sample project demonstrates how to add menu commands and keyboard shortcuts to the menu bar. The sample app uses its `MenuController` object to insert a `UIMenuSystem` object that adds the following menus:

- New: Appears at the beginning of the File menu and contains the two operations New Date Item and New Text Item, which add items to the table view.
- Cities: Contains a group of UICommand and UIKeyCommand objects.
- Navigation: Contains a group of UIKeyCommand objects for command-key navigation.
- Style: Contains a group of UICommand objects that have checkmark states. This grouping looks like a font-style menu for text formatting.
- Tools: Contains a group of UICommand objects.

The sample project also shows you how to add contextual menus to views, and how to handle menu-command selection using `UIContextMenuInteractionDelegate`.

Configure the sample code project

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

Add menus to the menu bar

Create `UIMenu` objects and use them to construct the menus and submenus. This sample adds menus to the menu bar when it runs on macOS, and key command elements in those menus also appear in the discoverability heads-up display (HUD) on iPad when the user presses the command key.

The Mac version of an iPad app comes with a standard menu bar. Use the `UIMenuSystem` class, an object that represents the main or contextual menu system, to modify the menu bar.

The sample uses `UIMenuSystem` to add menus and key commands to the `main` system menu by implementing `buildMenu(with:)` in `AppDelegate`. This function receives a `UIMenuBuilder` object that the sample then uses to add and remove menus. Because menus can exist with no window or view hierarchy, the system only consults `UIApplication` and `UIApplicationDelegate` to build the app's menu bar.

Menu commands consist of `UICommand`, `UIKeyCommand`, and `UIAction` objects that are grouped in a `UIMenu` container.

Add menu commands to the file menu

This sample inserts a `UIKeyCommand` called Command-O into the File Menu and creates a corresponding keyboard shortcut:

```swift
class func openMenu() -> UIMenu {
    let openCommand =
        UIKeyCommand(title: NSLocalizedString("OpenTitle", comment: ""),
                     image: nil,
                     action: #selector(AppDelegate.openAction),
                     input: "o",
                     modifierFlags: .command)
    let openMenu =
        UIMenu(title: "",
               image: nil,
               identifier: .openMenu,
               options: .displayInline,
               children: [openCommand])
    return openMenu
}
```

Notice that the UIKeyCommand title is a localized string using the NSLocalizedString function, which can display the menu name in multiple languages.

This sample inserts the Open command into the middle of the menu bar's File menu:

```
builder.insertChild(MenuController.openMenu(), atStartOfMenu: .file)
```

Mac apps also typically include a Print menu command in the File menu. This sample includes both Print and Export as PDF menu commands in the File Menu. These menu commands are automatically inserted when the `Info.plist` contains the `UIApplicationSupportsPrintCommand` key whose value is set to `true`. The application responds to these print commands by implementing `UIResponder's` `printContent(_ sender: Any?)` function.

Contribute to the edit menu

Editing operations, such as cut, copy, paste, and delete, are commonly used in most apps. This sample app provides these operations through the Edit menu, where the user can edit the sample's left-side content or its primary table-view content. These operations represent the first-responder functions `cut(_ sender: Any?)`, `copy(_ sender: Any?)`, `paste(_ sender: Any?)`, `delete(_ sender: Any?)`.

This sample implements `canPerformAction:withSender:` to determine if these edit operations are supported.

```
override func canPerformAction(_ action: Selector, withSender sender: Any?) -> Bool
    if action == #selector(printContent) {
        // Allow for printing if a table view cell is selected.
        return tableView.indexPathForSelectedRow != nil
    } else if action == #selector(newAction(_:)) {
        // User wants to perform a New operation.
        return true
    } else {
        switch (tableView.indexPathForSelectedRow, action) {

        // These Edit commands are supported.
        case let (_?, action) where action == #selector(cut(_:)) ||
                                    action == #selector(copy(_:)) ||
                                    action == #selector(delete(_:)):
            return true
        case (_?, _):
            // Allow the nextResponder to make the determination.
            return super.canPerformAction(action, withSender: sender)
        // Paste is supported if the pasteboard has text.
        case (.none, action) where action == #selector(paste(_:)):
            return (UIPasteboard.general.string != nil) ? true :
                // Allow the nextResponder to make the determination.
                super.canPerformAction(action, withSender: sender)
        case (.none, _):
            return false
        }
    }
}
```

Add commands to control the user interface

In the sample, you can change the primary or left-side table view's selection by using `UIKeyCommand`. These key commands are connected to the up and down arrow keys and are added directly to the table view. The following example shows how to add the down arrow key as a `UIKeyCommand`:

```
let downArrowCommand =
    UIKeyCommand(input: UIKeyCommand.inputDownArrow,
                 modifierFlags: [],
                 action: #selector(PrimaryViewController.downArrowAction(_:)))
addKeyCommand(downArrowCommand)
```

The sample also demonstrates how to add menu commands as command-key equivalents. The following example shows how to create a menu with all four arrow keys as command keys:

```
class func navigationMenu() -> UIMenu {
    let keyCommands = [ UIKeyCommand.inputRightArrow,
                        UIKeyCommand.inputLeftArrow,
                        UIKeyCommand.inputUpArrow,
                        UIKeyCommand.inputDownArrow ]
    let arrows = Arrows.allCases

    let arrowKeyChildrenCommands = zip(keyCommands, arrows).map { (command, arrow) i
        UIKeyCommand(title: arrow.localizedString(),
                     image: nil,
                     action: #selector(AppDelegate.navigationMenuAction(_:)),
                     input: command,
                     modifierFlags: .command,
                        propertyList: [CommandPListKeys.ArrowsKeyIdentifier: arrow.raw\
    }

    let arrowKeysGroup = UIMenu(title: "",
                  image: nil,
                  identifier: .arrowsMenu,
                  options: .displayInline,
                  children: arrowKeyChildrenCommands)

    return UIMenu(title: NSLocalizedString("NavigationTitle", comment: ""),
                  image: nil,
                  identifier: .navMenu,
                  options: [],
                  children: [arrowKeysGroup])
}
```

Display contextual menus

The sample displays a contextual menu through the use of `UIContextMenuInteraction Delegate`, an interaction object used to display relevant actions for the content.

For macOS, the user control-clicks or right-clicks the `DetailViewController`. For iPadOS, the user taps and holds the `DetailViewController`. The sample uses `UIContextMenu InteractionDelegate` to display Cut, Copy, Paste, Delete, Rename, and Share commands. This kind of contextual menu is a grouping of `UIAction` objects. `UIAction` is a menu element that performs its action in a closure. In iOS, optionally customize this contextual menu's highlight preview by using `contextMenuInteraction(_:previewForHighlightingMenuWith Configuration:)`. This delegate function returns a primary view controller. Override this function to enable menu commands based on the table view state or the state of the pasteboard.

Adjust the style menu

The sample implements `validate(_:)`, where it can adjust the Style menu as the user selects between one or more font styles: plain, bold, italic, underline.

```
// The font style menu item check marks used in the Style menu.
var fontMenuStyleStates = Set<String>()

// Update the state of a given command by adjusting the Style menu.
// Note: Only command groups that are added will be called to validate.
override func validate(_ command: UICommand) {
    // Obtain the plist of the incoming command.
    if let fontStyleDict = command.propertyList as? [String: String] {
        // Check if the command comes from the Style menu.
        if let fontStyle = fontStyleDict[MenuController.CommandPListKeys.StylesIdent
            // Update the Style menu command state (checked or unchecked).
            command.state = fontMenuStyleStates.contains(fontStyle) ? .on : .off
        }
    } else {
        // Validate the disabled command. This keeps the menu item disabled.
        if let commandPlistString = command.propertyList as? String {
            if commandPlistString == MenuController.disabledCommand {
                command.attributes = .disabled
            }
        }
    }
}
```

Add a preferences menu

Mac apps typically display app-specific preferences using a preferences window. The sample adds a preferences window by adding a Settings bundle to the Xcode project's target. The window automatically becomes available to the user through the Preferences menu command in the Application menu. To learn more about preferences, see Displaying a Preferences window.

See Also

User interactions

{} Navigating an app's user interface using a keyboard

Navigate between user interface elements using a keyboard and focusable UI elements in iPad apps and apps built with Mac Catalyst.

Handling key presses made on a physical keyboard

Detect when someone presses and releases keys on a physical keyboard.

`class UIHoverGestureRecognizer`

A continuous gesture recognizer that interprets pointer movement over a view.