Sample Code

# UIKit Catalog: Creating and customizing views and controls

Customize your app's user interface with views and controls.

Download

iOS 14.1+ | iPadOS 14.1+ | Mac Catalyst 14.1+ | Xcode 13.1+

## Overview

This sample guides you through several types of customizations you can make in your iOS app. It is built with Mac Catalyst, which means the sample runs on both iOS and macOS. The sample uses a split-view controller architecture to navigate UIKit views and controls. The primary view controller (`OutlineViewController`) shows the available views and controls. When you select one, `OutlineViewController` shows the secondary view controller associated with it.

The name of each secondary view controller reflects its target item. For example, the `AlertControllerViewController` class shows how to use a `UIAlertController` object. The only exceptions to this rule are `UISearchBar` and `UIToolbar`; the sample demonstrates these APIs in multiple view controllers to explain how their controls function and how to customize them. To demonstrate how to manage the complexity of your storyboards, the app hosts all view controllers in a separate storyboard and loads each as it's when needed.

This sample demonstrates the following views and controls; several of which are referenced in the sections below:

- `UIActivityIndicatorView`
- `UIAlertController`
- `UIButton`
- `UIButton.PointerStyleProvider`

- `UIDatePicker`
- `UIPickerView`
- `UIColorPickerViewController`
- `UIColorWell`
- `UIFontPickerViewController`
- `UIImagePickerController`
- `UIImageView`
- UIImageView with SF Symbols
- `UIPageControl`
- `UIProgressView`
- `UISearchBar`
- `UISegmentedControl`
- `UISlider`
- `UIStackView`
- `UIStepper`
- `UISwitch`
- `UITextField`
- `UITextFormattingCoordinator`
- `UITextView`
- `UIToolbar`
- `UIVisualEffect`
- `WKWebView`

## Configure the sample code project

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

## Customize the look of buttons with button configurations

You can customize the appearance and behavior of a UIButton by using `UIButton.Configuration`. This sample uses a `filled()` configuration so that the button draws with a red background color:

```
var config = UIButton.Configuration.filled()
config.background.backgroundColor = .systemRed
button.configuration = config
```

## Display a custom alert

`AlertControllerViewController` demonstrates several techniques to display modal alerts and action sheets from an interface. The configuration process is similar for all alerts:

1. Determine the message to display in the alert.
2. Create and configure a `UIAlertController` object.
3. Add handlers for actions the user may take.
4. Present the alert controller.

The `showSimpleAlert` function uses the `NSLocalizedString` function to retrieve the alert messages in the user's preferred language. The `showSimpleAlert` function uses those strings to create and configure the `UIAlertController` object. Although the button in the alert has the title OK, the sample uses a cancel action to ensure that the alert controller applies the proper styling to the button:

```
func showSimpleAlert() {
    let title = NSLocalizedString("A Short Title is Best", comment: "")
    let message = NSLocalizedString("A message needs to be a short, complete sentenc
    let cancelButtonTitle = NSLocalizedString("OK", comment: "")

    let alertController = UIAlertController(title: title, message: message, preferre

    // Create the action.
    let cancelAction = UIAlertAction(title: cancelButtonTitle, style: .cancel) { _ i
        Swift.debugPrint("The simple alert's cancel action occurred.")
    }

    // Add the action.
    alertController.addAction(cancelAction)

    present(alertController, animated: true, completion: nil)
}
```

## Customize the appearance of sliders

This sample demonstrates different ways to display a `UISlider`, a control to select a single value from a continuous range of values. Customize the appearance of a slider by assigning stretchable images for left-side tracking, right-side tracking, and the thumb. In this example, the track image is stretchable and is one pixel wide. Make the track images wider to provide rounded corners, but be sure to set these images' `capInsets` property to allow for the corners.

The `configureCustomSlider` function sets up a custom slider:

```
@available(iOS 15.0, *)
func configureCustomSlider(slider: UISlider) {
    /** To keep the look the same betwen iOS and macOS:
        For setMinimumTrackImage, setMaximumTrackImage, setThumbImage to work in Mac
        Available in macOS 12 or later (Mac Catalyst 15.0 or later).
        Use this for controls that need to look the same between iOS and macOS.
    */
    if traitCollection.userInterfaceIdiom == .mac {
        slider.preferredBehavioralStyle = .pad
    }

    let leftTrackImage = UIImage(named: "slider_blue_track")
    slider.setMinimumTrackImage(leftTrackImage, for: .normal)

    let rightTrackImage = UIImage(named: "slider_green_track")
    slider.setMaximumTrackImage(rightTrackImage, for: .normal)

    // Set the sliding thumb image (normal and highlighted).
    //
    // For fun, choose a different image symbol configuraton for the thumb's image b
    var thumbImageConfig: UIImage.SymbolConfiguration
    if slider.traitCollection.userInterfaceIdiom == .mac {
        thumbImageConfig = UIImage.SymbolConfiguration(scale: .large)
    } else {
        thumbImageConfig = UIImage.SymbolConfiguration(pointSize: 30, weight: .heavy
    }
    let thumbImage = UIImage(systemName: "circle.fill", withConfiguration: thumbImag
    slider.setThumbImage(thumbImage, for: .normal)

    let thumbImageHighlighted = UIImage(systemName: "circle", withConfiguration: thu
    slider.setThumbImage(thumbImageHighlighted, for: .highlighted)
    // Set the rest of the slider's attributes.
    slider.minimumValue = 0
    slider.maximumValue = 100
    slider.isContinuous = false
    slider.value = 84

    slider.addTarget(self, action: #selector(SliderViewController.sliderValueDidChar
}
```

## Add a search bar to an interface

Use a `UISearchBar` to receive search-related information from the user. There are various ways to customize the look of the search bar:

- Add a cancel button.
- Add a bookmark button.
- Set the bookmark image, both normal and highlighted states.
- Change the tint color that applies to key elements in the search bar.
- Set the search bar’s background image.

The `configureSearchBar` function sets up a custom search bar:

```
func configureSearchBar() {
    searchBar.showsCancelButton = true
    searchBar.showsBookmarkButton = true

    searchBar.tintColor = UIColor.systemPurple

    searchBar.backgroundImage = UIImage(named: "search_bar_background")

    // Set the bookmark image for both normal and highlighted states.
    let bookImage = UIImage(systemName: "bookmark")
    searchBar.setImage(bookImage, for: .bookmark, state: .normal)

    let bookFillImage = UIImage(systemName: "bookmark.fill")
    searchBar.setImage(bookFillImage, for: .bookmark, state: .highlighted)
}
```

## Customize the appearance of toolbars

This sample shows how to customize a `UIToolbar`, a specialized view that displays one or more buttons along the bottom edge of an interface. Customize a toolbar by determining its bar style (black or default), translucency, tint color, and background color.

The following `viewDidLoad` function in `CustomToolbarViewController` sets up a tinted tool bar:

```
override func viewDidLoad() {
    super.viewDidLoad()

    // See the `UIBarStyle` enum for more styles, including `.Default`.
    toolbar.barStyle = .black
    toolbar.isTranslucent = false

    toolbar.tintColor = UIColor.systemGreen
    toolbar.backgroundColor = UIColor.systemBlue

    let toolbarButtonItems = [
        refreshBarButtonItem,
        flexibleSpaceBarButtonItem,
        actionBarButtonItem
    ]
    toolbar.setItems(toolbarButtonItems, animated: true)
}
```

`CustomToolbarViewController` demonstrates further customization by changing the toolbar's background image:

```
override func viewDidLoad() {
    super.viewDidLoad()

    let toolbarBackgroundImage = UIImage(named: "toolbar_background")
    toolbar.setBackgroundImage(toolbarBackgroundImage, forToolbarPosition: .bottom,

    let toolbarButtonItems = [
        customImageBarButtonItem,
        flexibleSpaceBarButtonItem,
        customBarButtonItem
    ]
    toolbar.setItems(toolbarButtonItems, animated: true)
}
```

## Add a page control interface

Use a `UIPageControl` to structure an app's user interface. A *page control* is a specialized control that displays a horizontal series of dots, each of which corresponds to a page in the app's document or other data-model entity. Customize a page control by setting its tint color for all the page-indicator dots, and for the current page-indicator dot.

The `configurePageControl` function sets up a customized page control:

```
func configurePageControl() {
    // The total number of available pages is based on the number of available color
    pageControl.numberOfPages = colors.count
    pageControl.currentPage = 2

    pageControl.pageIndicatorTintColor = UIColor.systemGreen
    pageControl.currentPageIndicatorTintColor = UIColor.systemPurple

    pageControl.addTarget(self, action: #selector(PageControlViewController.pageCont
}
```

## Add menus to controls

Attach menus to controls like `UIButton` and `UIBarButtonItem`. Create menus with the `UIAction` class, and attach a menu to each control by setting the `UIMenu` property.

Attach a menu to a `UIButton` as shown here:

```
button.menu = UIMenu(children: [
    UIAction(title: String(format: NSLocalizedString("ItemTitle", comment: ""), "1")
             identifier: UIAction.Identifier(ButtonMenuActionIdentifiers.item1.rawVa
             handler: menuHandler),
    UIAction(title: String(format: NSLocalizedString("ItemTitle", comment: ""), "2")
             identifier: UIAction.Identifier(ButtonMenuActionIdentifiers.item2.rawVa
             handler: menuHandler)
])

button.showsMenuAsPrimaryAction = true
```

Create a `UIBarButtonItem` with a menu attached as shown here:

```
var customTitleBarButtonItem: UIBarButtonItem {
    let buttonMenu = UIMenu(title: "",
                            children: (1...5).map {
                               UIAction(title: "Option \($0)", handler: menuHandler)
                            })
    return UIBarButtonItem(image: UIImage(systemName: "list.number"), menu: buttonMe
}
```

## Add accessibility support to your views

VoiceOver and other system accessibility technologies use the information provided by views and controls to help all users navigate content. UIKit views include default accessibility support. Improve user experience by providing custom accessibility information.

In this UIKitCatalog sample, several view controllers configure the `accessibilityType` and `accessibilityLabel` properties of their associated view. Picker view columns don't have labels, so the picker view asks its delegate for the corresponding accessibility information:

```
func pickerView(_ pickerView: UIPickerView, accessibilityLabelForComponent component

    switch ColorComponent(rawValue: component)! {
    case .red:
        return NSLocalizedString("Red color component value", comment: "")

    case .green:
        return NSLocalizedString("Green color component value", comment: "")

    case .blue:
        return NSLocalizedString("Blue color component value", comment: "")
    }
}
```

## Support Mac Catalyst

This sample app is built with Mac Catalyst, which means the sample runs on both iOS and Mac. This is achieved by selecting the Mac checkbox in Project Settings. For more about how Mac Catalyst works see Mac Catalyst.

When built for Mac Catalyst, this sample achieves:

- Interface Optimization for Mac. With Optimize Interface For the Mac project setting turned on, the app has full control of every pixel on the screen, and the app can adopt more controls specific to Mac. Building the sample for Mac Catalyst makes the app take advantage of the system features in macOS. The option Show Designed for iPad Run Destination allows this sample, as an iPad app, to run as-is on Apple silicon Macs. This requires macOS 11 and a Mac with Apple silicon.

- Navigation and title bar hiding. The sample app hides these to make the app appear more like a Mac app. It also changes other behaviors by using traitCollection's `userInterfaceIdiom`.
- Translucent background. By setting the split view controller's `primaryBackgroundStyle` to `.sidebar`, the primary view controller or side bar shows a blurred desktop behind its view. Setting this property has no effect when running on iOS.

## See Also

### User interface

{} Building and improving your app with Mac Catalyst

Improve your iPadOS app with Mac Catalyst by supporting native controls, multiple windows, sharing, printing, menus and keyboard shortcuts.

Displaying a checkbox in your Mac app built with Mac Catalyst

Present a switch control as a Mac-style checkbox when your app runs in the Mac user interface idiom.

Removing the title bar in your Mac app built with Mac Catalyst

Display content that fills the entire height of a window by removing the title bar.

Toolbar

Provide a space for controls under a window's title bar and above your custom content.

Touch Bar

Display interactive content and controls in the Touch Bar.