Article

Configuring and Displaying Symbol Images in Your UI

Create scalable images that integrate with your app's text, and adjust the appearance of those images dynamically.

Technologies

AppKit

UIKit

On This Page

Overview

See Also

Overview

Symbol images give you a consistent set of icons to use in your app, and ensure that those icons adapt to different sizes and to app-specific content. A symbol image contains a vector-based shape that scales without losing its sharpness. You generate its final appearance by applying a tint color, or if you're using SF Symbols 2 or 3 you can apply multiple colors to add depth and emphasis to your symbol. You use symbol images in places where you display a simple shape or glyph, such as a bar button item.

Although symbols are images, they support many traits you associate with text. In fact, several system symbol images include letters, numbers, or symbolic characters in their content. For example, the system provides symbol images for mathematical operators for addition, subtraction, multiplication, and division. You can also apply text-related traits to a symbol image to make it look like the surrounding text:

- Apply a font text style to a symbol image so that it matches text with the same style. Font text styles also cause symbol images to scale to match the current Dynamic Type setting.
- Apply weights, such as thin, heavy, or bold, to a symbol image.
- Scale and style symbol images to match the font you use for text.
- Align a symbol image with neighboring text by using the image's baseline.

The system provides a collection of standard symbol images for you, including images for folders, the trash can, favorite items, and many more. Symbol images also adapt to the current trait environment, reducing the work required to support different sized interfaces. To browse the available symbol images, use the SF Symbols app, which you can download from Apple Design Resources.

You can also create symbol image files for your app's custom iconography, as described in Creating Custom Symbol Images for Your App.

Load a Symbol Image

When configuring an image view in a storyboard file, you browse the list of symbol image names in the Attribute inspector. When loading symbol images from your code, you look up the name for a symbol image using the SF Symbols app. For custom symbol images, create a Symbol Image Set asset in your asset catalog and load the asset by name.

Methods for loading symbol images are available across SwiftUI, UIKit, and AppKit. In each, you use certain methods when loading system images by name. When loading custom symbol images you use a different set of methods. Each method looks only for its designated image type, which avoids namespace collisions between your custom images and the system images.

The following examples show how to load a system symbol and a custom symbol across frameworks.

In SwiftUI, you use `Image(systemName:)` to load a system symbol image and `Image(_:)` to load your custom symbol, as the following code shows:

```
// Create a system symbol image.
Image(systemName: "multiply.circle.fill")

// Create a custom symbol image using an asset in an asset catalog in Xcode.
Image("custom.multiply.circle")
```

In UIKit, a `UIImage` object includes methods for loading the symbol image with specific traits or configuration options. When you load system symbol images, use `init(systemName:)`, `init(systemName:compatibleWith:)`, or `init(systemName:withConfiguration:)`. When you load your custom symbol images from an asset catalog, use `init(named:)`, `init(named:in:compatibleWith:)`, or `init(named:in:with:)`.

```
// Create a system symbol image.
let image = UIImage(systemName: "multiply.circle.fill")

// Create a custom symbol image using an asset in an asset catalog in Xcode.
let image = UIImage(named: "custom.multiply.circle")
```

When you use AppKit, use `init(systemSymbolName:accessibilityDescription:)` to load a system symbol image, and `init(named:)` to load your custom symbol image.

```
// Create a system symbol image with an accessibility description.
let image = NSImage(systemSymbolName: "multiply.circle.fill",
                    accessibilityDescription: "A multiply symbol inside a filled c

// Create a custom symbol image using an asset in an asset catalog in Xcode.
let image = NSImage(named: "custom.multiply.circle")
```

Apply a Specific Appearance to a Symbol Image

UIKit and AppKit methods return an image object with symbol image information. When you display that symbol image in an image view, the system applies default styling to it. An image with the default styling might appear out of place next to bold text or text that uses the headline text style.

To make the symbol image blend in with the rest of your content, you create a `UIImage.SymbolConfiguration` or `NSImage.SymbolConfiguration` object with information about how to style the symbol image. Configure the object with the text style you use for

neighboring labels and text views, or specify the font you use in those views. You can add weight information to give the symbol image a thinner or thicker appearance, and you can specify whether you want the image to appear slightly larger or smaller than the neighboring text.

In UIKit, you assign configuration data to the `preferredSymbolConfiguration` property of the `UIImageView` that contains your symbol image. Typically, you apply configuration data only to image views. For other types of system views, UIKit provides configuration data based on system requirements. For example, bars configure the symbol images in their bar button items to match the bar's configuration. The only other time you might use configuration data is when drawing the image directly. In that case, use the `applying(_:)` method to create a version of your image that includes the specified configuration data.

```
// Create a configuration object that's initialized with two palette colors.
var config = UIImage.SymbolConfiguration(paletteColors: [.systemTeal, .systemGray!

// Apply a configuration that scales to the system font point size of 42.
config = config.applying(UIImage.SymbolConfiguration(font: .systemFont(ofSize: 42

// Apply the configuration to an image view.
imageView.preferredSymbolConfiguration = config
```

If you're using SwiftUI, there are several modifiers available to configure a symbol image, as the following code shows:

```
Image(systemName: "multiply.circle.fill")
      .foregroundStyle(.teal, .gray)
      .font(.system(size: 42.0))
```

In AppKit, you create a configuration object and set `symbolConfiguration` on `NSImage View`.

```
var configuration = NSImage.SymbolConfiguration(paletteColors: [.systemTeal, .sys
configuration = config.applying(.init(textStyle: .title1))
imageView.symbolConfiguration = config
```

Update the Rendering Mode for a Symbol

SF Symbols contains four rendering modes: monochrome, palette, hierarchical, and multicolor. Symbols don't have an intrinsic color, so by default, the system uses the tint color to render them. For example, the following illustrates applying a tint color to an entire image:

```
// Create a system symbol image and apply a tint using SwiftUI.
Image(systemName: "multiply.circle.fill")
      .foregroundColor(.red)

// Create a symbol image with a tint using UIKit.
imageView.image = image?.withTintColor(.systemRed, renderingMode: .alwaysOriginal
```

In SwiftUI, you set the rendering mode using `symbolRenderingMode(_:)`, and apply colors using `foregroundStyle(_:)`. If a symbol doesn't support the rendering mode you choose, the system uses the monochrome version. Using `foregroundStyle` with multiple colors implies switching to palette rendering mode, so you can omit setting the rendering mode.

```
// Create a system symbol image in palette rendering mode.
Image(systemName: "multiply.circle.fill")
      .foregroundStyle(.teal, .gray)
```

In UIKit and AppKit, you use a symbol configuration object to modify a symbol's rendering mode. In AppKit, you apply a configuration using `withSymbolConfiguration(_:)`, whereas in UIKit, you apply a configuration by using `applyingSymbolConfiguration(_:)`, as the code below shows:

```
// Create an object configured for palette rendering mode.
let config = UIImage.SymbolConfiguration(paletteColors: [.systemTeal, .systemGray

// Create a new symbol image using the configuration object.
imageView.image = image.applyingSymbolConfiguration(config)
```

Update the Weight and Scale of a Symbol

There are nine symbol weights corresponding to a weight of the San Francisco system font, helping you achieve precise weight matching between symbols and adjacent text, while supporting flexibility for different sizes and context.

By specifying a scale, you adjust a symbol's emphasis compared to adjacent text without disrupting the weight matching with text that uses the same point size. See `imageScale` (SwiftUI), `UIImage.SymbolScale` (UIKit), and `NSImage.SymbolScale` (AppKit).

```
// Create a large scaled symbol image using SwiftUI.
Image(systemName: "multiply.circle.fill")
      .imageScale(.large)

// Create a large scaled symbol image using UIKit.
var config = UIImage.SymbolConfiguration(scale: .large)
imageView.image = image.withSymbolConfiguration(config)

// Create a large scaled symbol image using AppKit.
var config = NSImage.SymbolConfiguration(scale: .large)
imageView.image = image.withSymbolConfiguration(config)
```

Apply a Symbol Image Variant

SF Symbols defines design variants — such as outline, fill, slash, and enclosed — to help you communicate precise states and actions, while maintaining visual consistency and simplicity in your UI. For example, you might use the slash variant to show that an action is unavailable, or use the fill variant to indicate when the user selects something.

> **Note**
>
> If a variant doesn't exist for a symbol, the system uses the base symbol. For example, tab bars in iOS default to using a fill variant, so choosing a symbol without a fill variant uses the original symbol.

In SwiftUI, you use the modifier `symbolVariant(_:)` to apply a variant. Search the SF Symbols app to find the variants the multiply symbol supports, such as `circle`, `circle.fill`, `square`, and `square.fill`.

```
// Create a system symbol image that is enclosed with a filled circle variant.
Image(systemName: "multiply")
      .symbolVariant(.circle.fill)
```

Align Symbol Images with a Text Label by Using a Baseline

Because of the typographical nature of SF Symbols, when you position an image view containing a symbol image next to a label, you should align the views using their baselines. To align views in your storyboard, select the two views and add a first baseline constraint. Programmatically, you create this constraint by setting the `firstBaselineAnchor` of both views to be equal, as shown in the following code example:

```
NSLayoutConstraint.activate([
    imageView!.firstBaselineAnchor.constraint(equalTo: label!.firstBaselineAnchor
])
```

All system symbol images include baseline information, and `UIImage` exposes the baseline value as an offset from the bottom of the image. Typically, the baseline of a symbol image aligns with the bottom of any text that appears in the image, but even symbol images without text have a baseline. In AppKit, a symbol's baseline corresponds to the bottom of the `alignmentRect` property, and in UIKit you can add a baseline to any image by calling its `withBaselineOffset(fromBottom:)` method.

```
// Create a custom symbol image.
let image = UIImage(named: "custom.multiply.circle")

// Add an offset of 2.0 points from the baseline.
let baselineImage = image?.withBaselineOffset(fromBottom: 2.0)
```

In SwiftUI, you use `firstTextBaseline` to baseline align a symbol with text.

```
HStack(alignment: .firstTextBaseline) {
    Image(systemName: "menucard")
    Text("SF Symbols 3")
}
```

Use Fallback Assets for Deprecated Symbol Name Changes

Symbol names may change between new operating system versions and SF Symbols app releases, so you need to review your symbol usage when supporting new versions. Look for the deployment target when browsing the SF Symbols app and use the old name if you're backward deploying.

If a symbol doesn't render in your app, use the SF Symbols app to search for the symbol name, and look for availability updates by choosing View > Inspectors > Show Info Sidebar.

Depending on the operating systems you support, you may need to provide assets that you fall back to. For example, SF Symbols is only supported in iOS 13 or later. In your asset catalog you could have a symbol image named `gamecontroller`, and a fallback PNG asset named `gamecontroller` that you use for earlier operating system versions. Using Interface Builder, set an image view with your asset named `gamecontroller` and it loads the first available asset based on the platform.

Programmatically, you need to use `#available` to load assets appropriately.

```
if #available(iOS 13.0, *) {
    // Load an SF Symbol image.
} else {
    // Load a PNG asset.
}
```

See Also

Loading and Caching Images

- Providing Images for Different Appearances

  Supply image resources appropriate for light and dark appearances and for high-contrast environments.

- Creating Custom Symbol Images for Your App

  Create, organize, and annotate symbol images using SF Symbols.

`init?(named: String, in: Bundle?, compatibleWith: UITraitCollection?)`

Creates an image object using the named image asset that is compatible with the specified trait collection.

`init?(named: String, in: Bundle?, with: UIImage.Configuration?)`

Creates an image object using the named image asset that is compatible with the specified configuration.

`init?(named: String)`

Creates an image object from the specified named asset.

`init(imageLiteralResourceName: String)`

Returns the image object for the specified resource.

`init?(systemName: String, withConfiguration: UIImage.Configuration?)`

Creates an image object that contains a system symbol image with the specified configuration.

`init?(systemName: String, compatibleWith: UITraitCollection?)`

Creates an image object that contains a system symbol image appropriate for the specified traits.

`init?(systemName: String)`

Creates an image object that contains a system symbol image.

{} Building High-Performance Lists and Collection Views

Improve the performance of lists and collections in your app with prefetching and image preparation.