Article

Creating custom symbol images for your app

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

Overview

SF Symbols 4 offers a set of over 4,000 consistent, highly configurable symbol images that you can use in your app. You can apply stylistic traits typically associated with text, such as applying colors, text style, weight, and scale. Symbols contain additional traits that allow them to integrate seamlessly with surrounding text, and adapt to platform features like Dynamic Text and Dark Mode.

You can create your own custom symbol images with the same capabilities that SF Symbols provides. To create your custom symbol:

1. Export an SVG file from the SF Symbols app.

2. Edit the SVG file in a vector-drawing app.

3. Export the file from your drawing app as an SVG file.

4. Validate the SVG file using the SF Symbols app.

5. Import the custom symbol into the SF Symbols app and organize it into a group.

6. Add annotations, if necessary.

7. Export a template file for distribution.

One way to begin creating your own symbol is by basing it on an existing symbol you find in the SF Symbols app. For example, the circle symbol can give you a great reference point to start working with.

Tip

You can drop an existing symbol — from a previous template version — onto the app, and it automatically converts the symbol to template version 4 and adds it to the current collection.

For design guidance, see Human Interface Guidelines > SF Symbols.

Export a custom symbol template file

Beginning with template version 3, the left-margin and right-margin guidelines have more explicit names that indicate the design variant they correspond to. For additional control over optical alignment, you can add margins to any variant in the template. The template is able to embed information about how a symbol should look in different rendering modes.

After you locate a symbol image to use as a base for your design, choose File > Duplicate as Custom Symbol. The app creates a new category at the bottom of the categories sidebar where you find your custom symbols. To create an SVG file for your custom symbol, export a symbol template file for design customization by selecting the symbol and choosing File > Export Template.

When exporting a template, you choose between static or variable. Use a static setup if you're targeting a particular weight and scale, or only plan to design one or two variants of your symbol. The setup contains 27 sets of paths and one set of explicit margins. A variable template setup contains three sets of paths and three sets of margins. If you plan on supporting all design variants, this gives you the minimum number of variants necessary for the system to generate the other 24.

Note

Beginning with template version 3, SF Symbols introduces vector interpolation for symbol variants. By using three sources — `Ultralight-S`, `Regular-S`, and `Black-S` — SF Symbol can dynamically generate the full range of weights and scales you don't specify.

After you export a template file, you use a vector-drawing app, such as Adobe Illustrator or Sketch, to begin modifying it.

Manage symbol image variants

The Symbols layer contains up to 27 sublayers, each representing a symbol image variant. Identifiers of symbol variants have the form `<weight>-<{S, M, L}>`, where *weight* corresponds to a weight of the San Francisco system font and *S*, *M*, or *L* matches the small, medium, or large symbol scale.

```
<g id="Symbols">
    <g id="Regular-M" transform="matrix(1 0 0 1 2855.62 1556)">
        <!-- Path and style details for the Regular-M image variant. -->
    </g>
</g>
```

SF Symbols treats a symbol as path-based, rather than stroke-based, if all the shapes within it have solid color fills, and don't have strokes or other graphical features. Interpolation allows the system to generate variants between compatible paths. A template is interpolatable when:

- It contains the interpolation sources `Ultralight-S`, `Regular-S`, and `Black-S`.
- The three interpolation sources are path-based.
- The three interpolation sources contain the same number of paths and the same number of control points.

It's not necessary for a template to contain all 27 variants. You can add as many variants to your template as you want, and if they're present, the system uses them instead of interpolation. Similar to working with template versions before version 3, you can delete the variants you don't need, so you can produce as many weights and scales as your app requires.

Organize the Guides layer

The system uses guides to align your custom symbol image with surrounding text. For example, it uses the provided baseline and cap height information for each of the three font scales to compute the symbol image's baseline offset and cap height.

The Guides layer contains an uppercase letter *A* in outline form for each scale in the San Francisco system font as a reference glyph. Use the reference glyphs in the template as guides for how a symbol image looks next to text.

Beginning with template version 3, each image variant of a symbol can have its own margin guides. This allows the margins to vary slightly by weight and scale instead of using a fixed margin for all variants. The explicit margin guides have the form `left-margin-<variant-specifier>` or `right-margin-<variant-specifier>`. The following example represents the left and right guides of the `Regular-S` symbol variant:

```
<g id="Guides">
    <line id="left-margin-Regular-S" style="fill:none;stroke:#00AEEF;stroke-width:0.
    <line id="right-margin-Regular-S" style="fill:none;stroke:#00AEEF;stroke-width:0
```

```
</g>
```

Symbols can contain negative margins to aid with horizontal alignment. If you don't specify explicit margin guides, the system uses the next available margins it finds — interpolated margins (if the template is interpolatable), `Regular-M`, `Regular-S`, and, finally, any available margins the system can use.

Create your custom symbol image

Start creating your custom symbol image by modifying the symbol in the template file you export. If you export a variable template, you need to modify all three symbol configurations so the system can generate the other variants.

When you finish your base variant, copy the existing drawing to the desired layer and adjust from there. This helps you keep the same number of paths across your design variants, which is a requirement if you want to produce a symbol with multicolor or hierarchical data.

Use the following scale factors when designing your variants:

	<weight>-S	**<weight>-M**	**<weight>-L**
Scale factor	0.783	1.0	1.29

The system automatically centers symbols vertically to San Francisco's cap-height in all the different scales and weights. It's important to position your custom symbol using the specified guides to make sure it appears correctly in text. When the symbol image appears in text, the system positions it vertically so that the bottom edge of the symbol image is the same distance, scaled by point size, that the symbol image is from the baseline guide in the template file.

Important

SF Symbols picks up all paths in a variant's layer — including invisible paths — as part of the symbol outlines. This may lead to unexpected results when working with layers in the SF Symbols app, so don't use hidden paths.

When you create a symbol, you work on the monochrome representation. To ensure that your symbol supports rendering modes other than monochrome:

- Convert any strokes to paths so the resulting shapes can take on colors or hierarchy groups. Paths make it easier to make minor optical adjustments when a stroke isn't precise enough.

- Use standard flat color fills with no additional effects like drop shadows. If these are present they override any multicolor or hierarchical data you create for your symbol.
- Check that all shapes in your design have a defined fill area with start and end points that connect.

Important

Annotation data requires the same number of paths across designs. To retain your annotation data when modifying the paths of an annotated symbol, you can add, remove, and adjust points, but removing or reordering whole paths makes your designs go out of sync. In these cases, you need to reannotate the symbol to account for its new path structure.

Image variants adapt automatically according to the user's device language, including right-to-left writing systems. If you're designing for left-to-right and right-to-left writing systems, consider the directionality and overall look of both localized variants. In some cases, some symbols don't have the intended look when you mirror them. For design guidance, see Human Interface Guidelines > Right to left.

Preserve annotations and meta information in the Notes layer

The Notes layer contains optional annotations and meta information about the template file. The `template-version` layer contains a required version string that indicates the template format version, so you must not remove it; otherwise, SF Symbols can't read the file. The `artboard` layer makes sure design tools display the template with a convenient canvas size and legible symbols.

The Notes layer includes annotations on the custom symbol template file that can help you understand its contents. These annotations are optional, but it's a good idea to keep them as-is for your reference.

Note

You don't need to modify the contents of the Notes layer. Lock the artboard layer in your vector-drawing app before making any modifications to the file to avoid interacting and moving it.

Export your custom symbol and preserve all names

When you finish designing your symbol, export it from your vector-drawing app as an SVG file with maximum precision. By default, Illustrator generates low-precision SVGs, so export your SVG by changing the default decimal place value to 7 or greater. Confirm that the SVG file preserves all of the identifiers for your symbol variants and guides.

Use any of the following methods to validate that your SVG file conforms to the requirements of SF Symbols:

- Use the SF Symbols app and choose File > Validate Templates.
- Add it to an asset catalog in your Xcode project. Xcode verifies the SVG file and displays error messages if it doesn't conform to the requirements.
- Inspect the SVG file to manually review the XML source code. Understanding the template layout helps you debug validation issues, so keep the original template you import into your vector-drawing app for a reference to compare against.

When you have a valid template file, you're ready to begin annotating it. Import your symbol back into the SF Symbols app by dropping it onto your custom symbol.

Annotate your custom symbol

If you want to control your symbol's appearance in rendering modes other than monochrome, you can annotate your symbol. SF Symbols applies annotations to individual shape objects in the form of a CSS style, using a class name that has the form `multicolor-<layer index>:<color name>` or `hierarchical-<layer index>:<hierarchy level>`, and starts the layer index at zero. You set the color name to either a system color, a named color from the app's asset catalog, or any constant that doesn't dynamically resolve. A shape can have both multicolor and hierarchical annotations, and they don't have to be in the same layer.

```
<style>
    .multicolor-0:systemBlueColor { fill:#007AFF; opacity:1.0 }
    .multicolor-1:white { fill:#FFFFFF; opacity:1.0 }
    .multicolor-2:tintColor { fill:#007AFF; opacity:1.0 }
    .hierarchical-0:tertiary { fill:#8E8E8E }
    .hierarchical-1:primary { fill:#212121 }
</style>

<g id="Symbols">
    <!-- A variant containing three shapes with multicolor and hierarchical annotati
    <g id="Regular-M" transform="matrix(1 0 0 1 2853.78 1556)">
        <!-- The shape is in the first multicolor layer, whose fill color is systemB
        <path class="multicolor-0:systemBlueColor hierarchical-1:primary" d="...">

        <!-- Two additional shapes. --><path class="multicolor-1:white hierarchical-
        <path class="multicolor-2:tintColor hierarchical-0:tertiary" d="...">
        </g>
    </g>
```

To annotate, you use the individual paths that make up your symbol as your building blocks. From there, you create a set of layers for each rendering mode. SF Symbols assigns a color to layers in multicolor mode, and assigns a hierarchy group to layers in hierarchical mode. Layers have an explicit Z-order where the layers on top block the layers below it.

In SF Symbols, select your symbol and enter the gallery view by choosing View > As Gallery. Open the color inspector by choosing View > Inspectors > Show Color Sidebar, and then select the rendering mode to start annotating. The center preview lets you interact with all the paths and assign them to layers. You can use a path in any number of layers. In hierarchical mode, you assign groups from primary to tertiary, and the system uses the same data in hierarchical and palette rendering mode.

Tip

Use system-provided colors wherever possible because they adapt to changes in the system's appearance — light, dark, and high-contrast modes — and in different vibrancy contexts.

A common problem with overlapping shapes is that you can see through where paths overlap. In the color inspector, each layer has a toggle to the right of it. If it's in a disabled state, the transparent layers blend with the layers below. If it's active, transparent layers clear what's behind them and render as if the other layers don't exist.

When you're done annotating your symbol, you export the symbol for distribution.

Annotate a variable symbol

System and custom symbols can dynamically apply colors by using a percentage value to convey the strength or progress over time. For example, a wireless signal strength symbol can reflect full signal strength at 100% or no signal at 0%. In SF Symbols, select your symbol and open the color inspector. Each layer allows you to activate variable color for it. The vector file contains the thresholds for your symbol.

```
<style>
    <!-- A symbol that contains three variable color layers -->
    .monochrome-0 {fill:#000000}
    .monochrome-1 {fill:#000000}
    .monochrome-2 {fill:#000000;-sfsymbols-variable-threshold:0.0}
    .monochrome-3 {fill:#000000;-sfsymbols-variable-threshold:0.34}
    .monochrome-4 {fill:#000000;-sfsymbols-variable-threshold:0.68}
</style>
```

Distribute your custom symbol

There are two options to consider when distributing a symbol. Choose File > Export Symbol to begin. At the bottom left of the screen, version 4 appears by default.

Template version 2 and later removes annotation data and explicit margins. Use this version if you plan to deploy to an older operating system, such as iOS 14. It only contains monochrome, so make sure your symbol makes sense in that mode.

Template version 3 and later embeds all of your multicolor and hierarchical data annotations, as well as any custom margins. It isn’t backward-compatible, so use this version when you’re supporting iOS 15 or later.

Template version 4 embeds your variable color annotations, so use this version when you’re supporting iOS 16 or later.

Important

None of the versions is a source artifact for editing. Current design tools may not be compatible with the embedded annotation data. If you need to make edits or share it with a colleague, import it back into the SF Symbols app.

If your minimum deployment target is iOS 15 or later, you only need the version 3 template. If your minimum deployment target is iOS 14, you need to export a version 2, 3, and 4 template, and use the appropriate asset depending on a version check. Use the latest template when sharing with a colleague because they can import it into their SF Symbols app to continue editing and annotating.

The following code example shows the latest template version file — preserving identifiers, but omitting pathing details — and includes additional notes for each area:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Generator: Apple Native CoreSVG 168-->
<!DOCTYPE svg
  PUBLIC "-//W3C//DTD SVG 1.1//EN"
         "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg version="1.1" xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org

<style>
<!--
Class names have the form <rendering mode style>-<layer index>:<color name>. Renderi
```

```
If the color name isn't a system color and isn't in the asset catalog, a style can h

Version 4 of the symbol template supports variable color thresholds by using `-sfsym
-->
  .multicolor-0:systemBlueColor {fill:#007AFF;-sfsymbols-variable-threshold:0.0}
  .multicolor-1:tertiaryLabelColor {fill:#BDBDBD;-sfsymbols-clear-behind:true}
  .multicolor-2:white {fill:#FFFFFF;opacity:0.4}
  .hierarchical-0:tertiary {fill:#8E8E8E}
  .hierarchical-1:primary {fill:#212121;-sfsymbols-clear-behind:true}
</style>

<g id="Notes">
<!--
The symbol template supports rendering mode annotations and path interpolation.
-->
  <text id="template-version">Template v.3.0</text>
</g>

<g id="Guides">
  <line id="Baseline-S" x1="..." x2="..." y1="..." y2="..."/>
  <line id="Capline-S" x1="..." x2="..." y1="..." y2="..."/>
  <line id="Baseline-M" x1="..." x2="..." y1="1126" y2="..."/>
  <line id="Capline-M" x1="..." x2="..." y1="1055.54" y2="..."/>
  <line id="Baseline-L" x1="..." x2="..." y1="1556" y2="..."/>
  <line id="Capline-L" x1="..." x2="..." y1="1485.54" y2="..."/>

  <!--
  The symbol template supports explicit margins for each variant.
  -->
  <line id="left-margin-Ultralight-S" x1="..." x2="..." y1="..." y2="..."/>
  <line id="right-margin-Ultralight-S" x1="..." x2="..." y1="..." y2="..."/>
  <line id="left-margin-Regular-S" x1="..." x2="..." y1="..." y2="..."/>
  <line id="right-margin-Regular-S" x1="..." x2="..." y1="..." y2="..."/>
  <line id="left-margin-Black-S" x1="..." x2="..." y1="..." y2="..."/>
  <line id="right-margin-Black-S" x1="..." x2="..." y1="..." y2="..."/>
</g>

<g id="Symbols">
<!--
The symbol template generates variants from the following source variants: Black-S,
-->
  <g id="Black-S">
  <!--
  The system concatenates together all shapes for each rendering mode layer. A shape
  -->
    <path class="multicolor-0:systemBlueColor hierarchical-0:tertiary" d="..."/>
    <path class="multicolor-1:tertiaryLabelColor hierarchical-1:primary" d="..."/>
    <path class="multicolor-0:systemBlueColor multicolor-1:tertiaryLabelColor hierar
    <path class="multicolor-2:white hierarchical-1:primary" d="..."/>
  </g>

  <g id="Regular-S">
    <path class="multicolor-0:systemBlueColor hierarchical-0:tertiary" d="..."/>
    <path class="multicolor-1:tertiaryLabelColor hierarchical-1:primary" d="..."/>
    <path class="multicolor-0:systemBlueColor multicolor-1:tertiaryLabelColor hierar
    <path class="multicolor-2:white hierarchical-1:primary" d="..."/>
  </g>

  <g id="Ultralight-S">
    <path class="multicolor-0:systemBlueColor hierarchical-0:tertiary" d="..."/>
    <path class="multicolor-1:tertiaryLabelColor hierarchical-1:primary" d="..."/>
    <path class="multicolor-0:systemBlueColor multicolor-1:tertiaryLabelColor hierar
    <path class="multicolor-2:white hierarchical-1:primary" d="..."/>
  </g>
</g>
```

Use your custom symbol images

Open your app’s Xcode project and select its asset catalog. In Xcode’s menu bar, select Editor > Add New Asset > Symbol Image Set, and drag your exported SVG file into the Symbol SVG section of the Symbol pane. Xcode validates the SVG file, and displays error messages if the file doesn’t fulfill the requirements. To use the symbol image in your app, follow Configuring and displaying symbol images in your UI.

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.

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.

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

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

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

Creates an image by using the named image asset that’s compatible with the configuration you specify.

`convenience init?(named: String, in: Bundle?, variableValue: Double, configuration: UIImage.Configuration?)`

Creates an image by using the name, configuration, and variable value you specify.

`init?(named: String)`

Creates an image object from the specified named asset.

`convenience 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.

`convenience init?(systemName: String, variableValue: Double, configuration: UIImage.Configuration?)`

Creates an image object that contains a system symbol image with the configuration and variable value you specify.

`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.

`convenience init(resource: ImageResource)`

{} Building high-performance lists and collection views

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