SF Symbols for iOS: Getting Started

Learn to use SF Symbols, both existing and custom, to show data in an engaging way.

By Tom Elliott Apr 19 2021 · Article (30 mins) · Intermediate

In this tutorial, you'll learn all about SF Symbols as well as how to create your own custom symbols to help your app stand out from the crowd! :]

SF Symbols are a set of over 2,400 symbols, or icons, curated by Apple. They're designed to work well with the default system font on Apple devices called ***San Francisco***. They provide an easy way to add iconography to your projects, as they come in a wide variety of sizes and weights. With so many ready-made options, you're likely to find the perfect fit for your app's style. And, if you *can't* find exactly what you need, you can create it.

Here, you'll add new bling icons to an app showing the status of London Underground train lines, commonly known as the Tube. In the process, you'll learn how to:

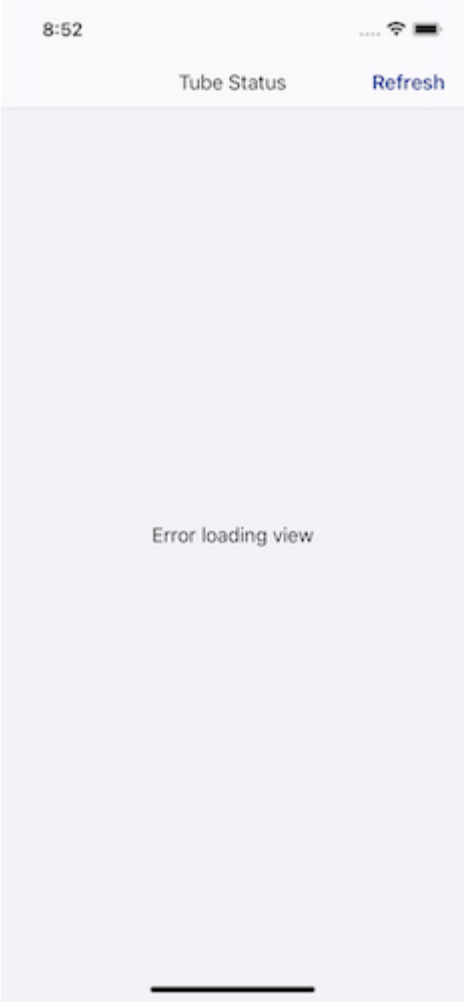
- Integrate SF Symbols into your app.
- Associate different SF Symbols with different statuses.
- Create your own, custom symbols for use in your app.

Getting Started

First, download the project materials using the ***Download Materials*** button at the top or bottom of this tutorial and open the starter project in Xcode.

The app displays the current status of Tube lines. You'll add to the project by associating an SF Symbol with each status and having it display in the app. This way, users can get the information they need with a glance.

Build and run the app.



Uh oh! It errors! This is because you need to set up the app to fetch data for the tube lines. This is what you'll do now after going through a walk-through of how the app works.

> ***Note***: If you aren't interested in learning about the inner workings of the app, you can skip ahead to ***Setting up TransportAPI***.

Getting Acquainted with the App

Open ***AppMain.swift***. The main view displays a view called `TubeStatusView`, which acts as the root view of the app.

Next, open ***TubeStatusView.swift***. `TubeStatusView` observes a model object of type `TubeStatusViewModel`. The `body` of the view displays a `ZStack` that sets the background color of the view and then displays a `Loadable` on top.

`Loadable` shows different content depending on the loading state:

A spinning activity indicator while waiting for data.

An error if loading the data fails.

The contents of the view builder once the data has loaded successfully.

Once data loading completes, `TubeStatusView` shows `LineStatusRow` for each line. These are contained in a `ScrollView`. It also displays `Text` showing when the data was last updated.

Notice how the view also contains `onAppear(perform:)`. This calls `loadData()` of the view when it first appears on the screen. This method asks the model to perform `fetchCurrentStatus` by calling `perform(action:)`.

Open ***LineStatusRow.swift*** and have a quick look around. This is a simple view that displays the status of a particular Tube line. You can look at the SwiftUI previews to see what these rows will look like.

Importing Models and Data

Next, open ***TubeStatusViewModel.swift***. The important parts of this class are marked under ***Actions*** and ***Action Handlers***.

You've already seen how `TubeStatusView` calls `perform(action:)` when it appears on the screen. By passing the `fetchCurrentStatus` enum as the action, this method calls `fetchCurrentStatus()`.

`fetchCurrentStatus()` calls `fetchStatus()` on an instance variable of type conforming to the `TubeLinesStatusFetcher` protocol. This handles fetching the data from the API. Once fetched, `tubeStatusState` is updated. As this is a published object, SwiftUI handles updating the UI automatically.

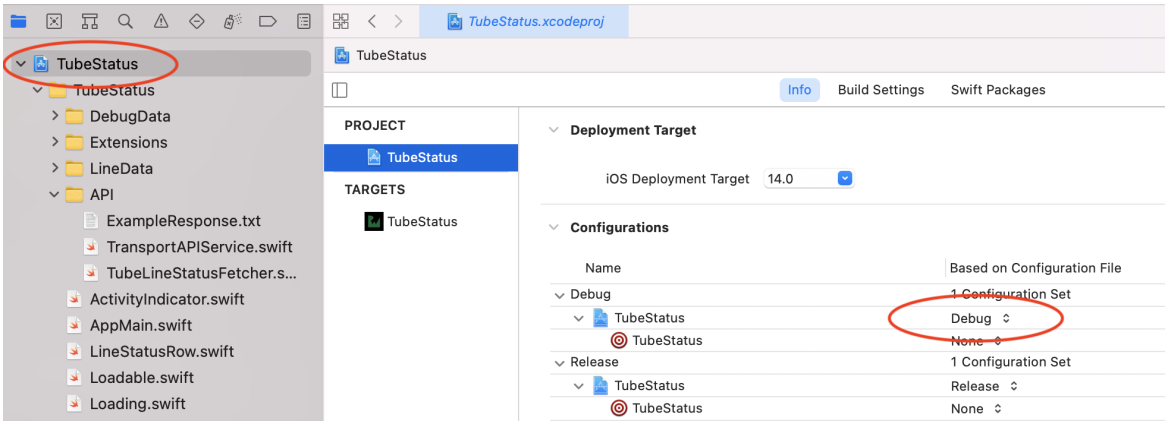
Adding TransportAPI Functionality

Open ***TransportAPIService.swift*** in the ***API*** group. This class handles fetching the JSON data from a network request in `fetchStatus()` and decodes the JSON into the `AllLinesStatus` struct.

The important part to note in this class is `appId` and `appKey` on lines 82 and 83. These values are loaded from the app's ***Info.plist***, specifically from the `TRANSPORT_API_SERVICE_APP_ID` and `TRANSPORT_API_SERVICE_APP_KEY` keys, respectively.

Open ***Info.plist***. Note how the values aren't set to any sort of valid ID, but instead set to `$(TRANSPORT_API_SERVICE_APP_ID)` and `$(TRANSPORT_API_SERVICE_APP_KEY)`. The `$(...)` syntax is called ***variable substitution***. It tells Xcode to substitute the value in the ***Info.plist*** with the value of the key from a configuration file when building the app. You'll set this up in just a moment.

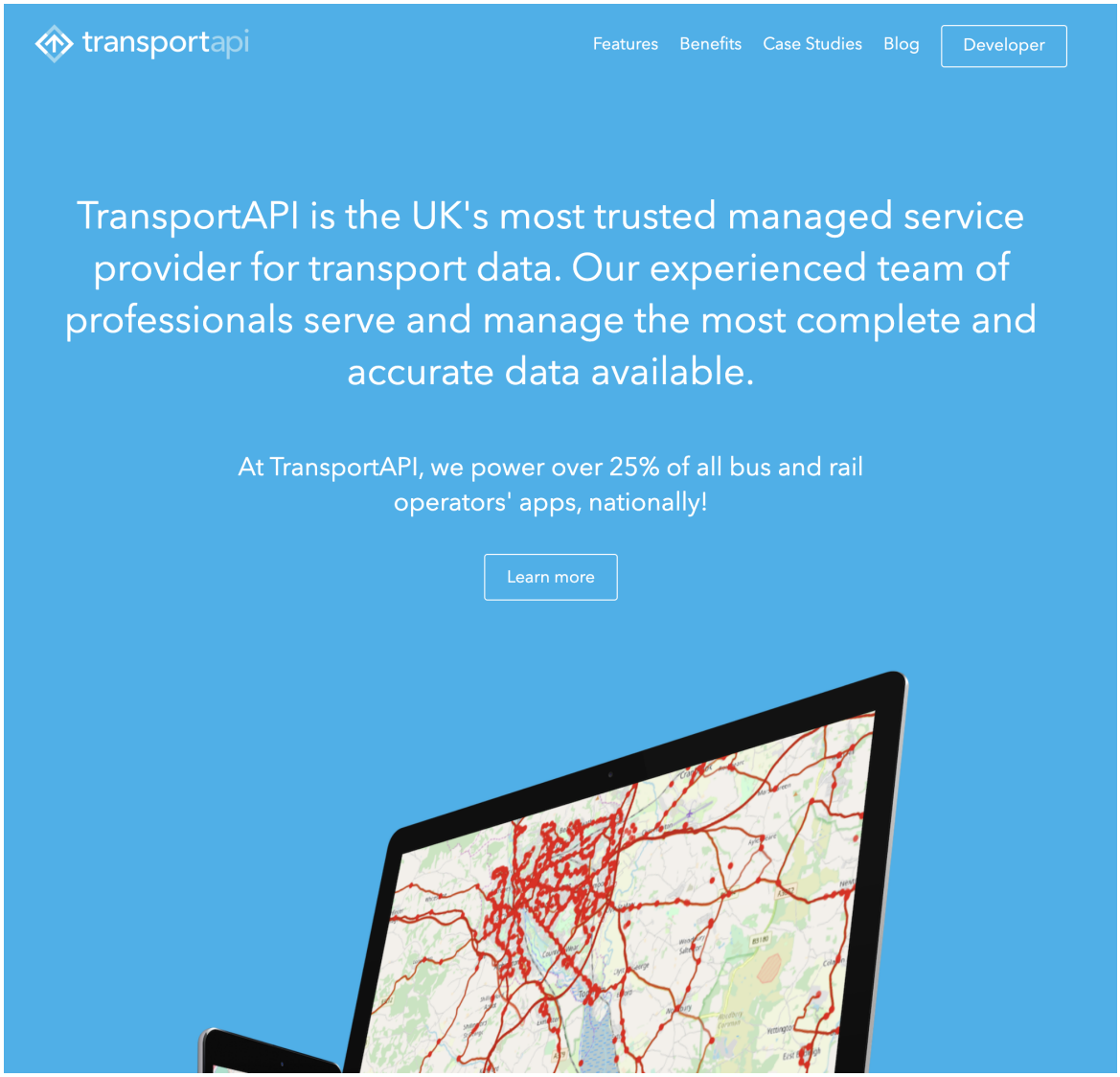
Open the ***Project Info*** panel by clicking the ***TubeStatus*** project in the ***Project navigator***. Then, click ***TubeStatus*** underneath the ***Project*** heading.



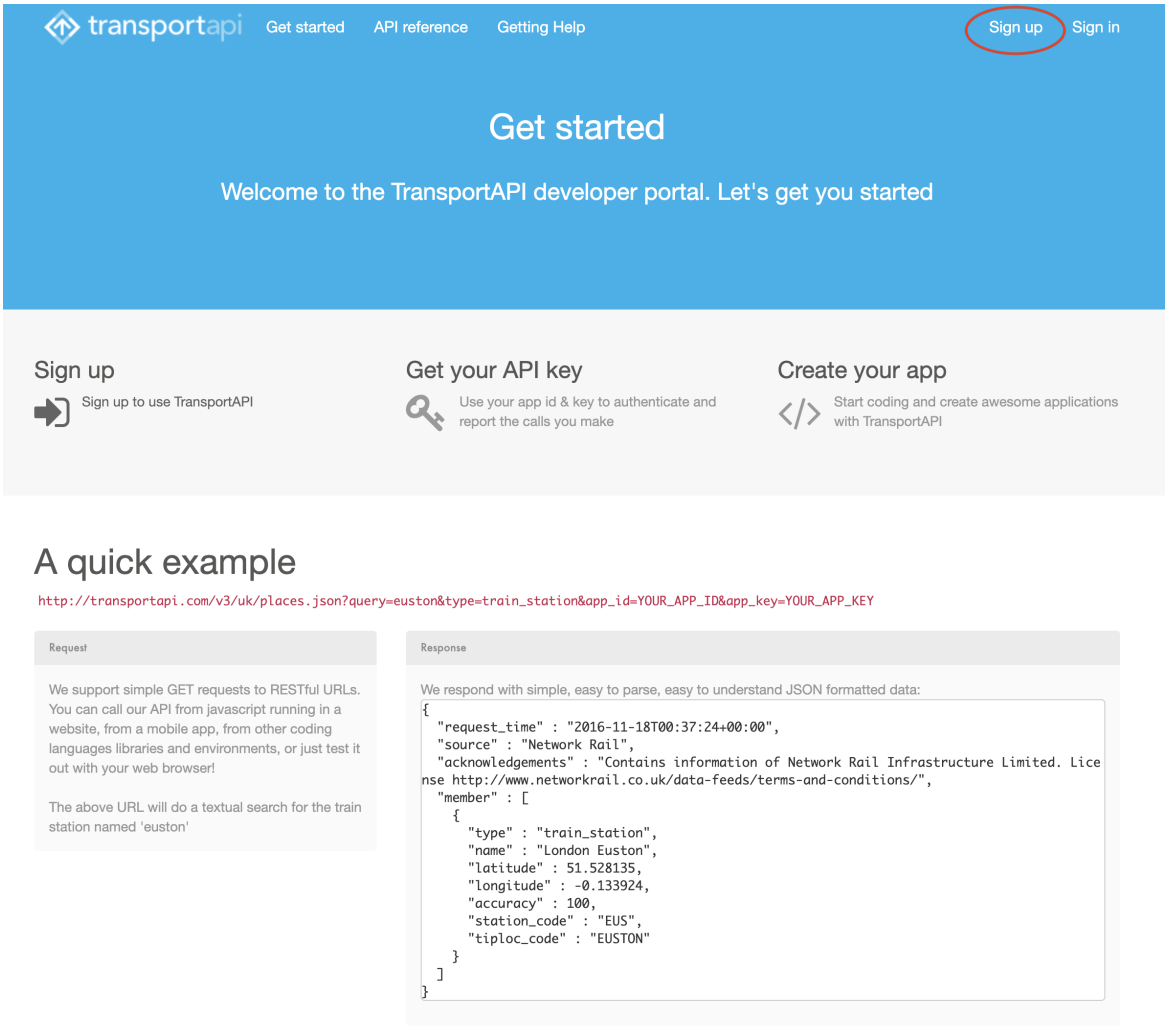
The ***Debug*** configuration loads a ***Debug*** configuration file with a similar setup for ***Release***.

Setting up TransportAPI

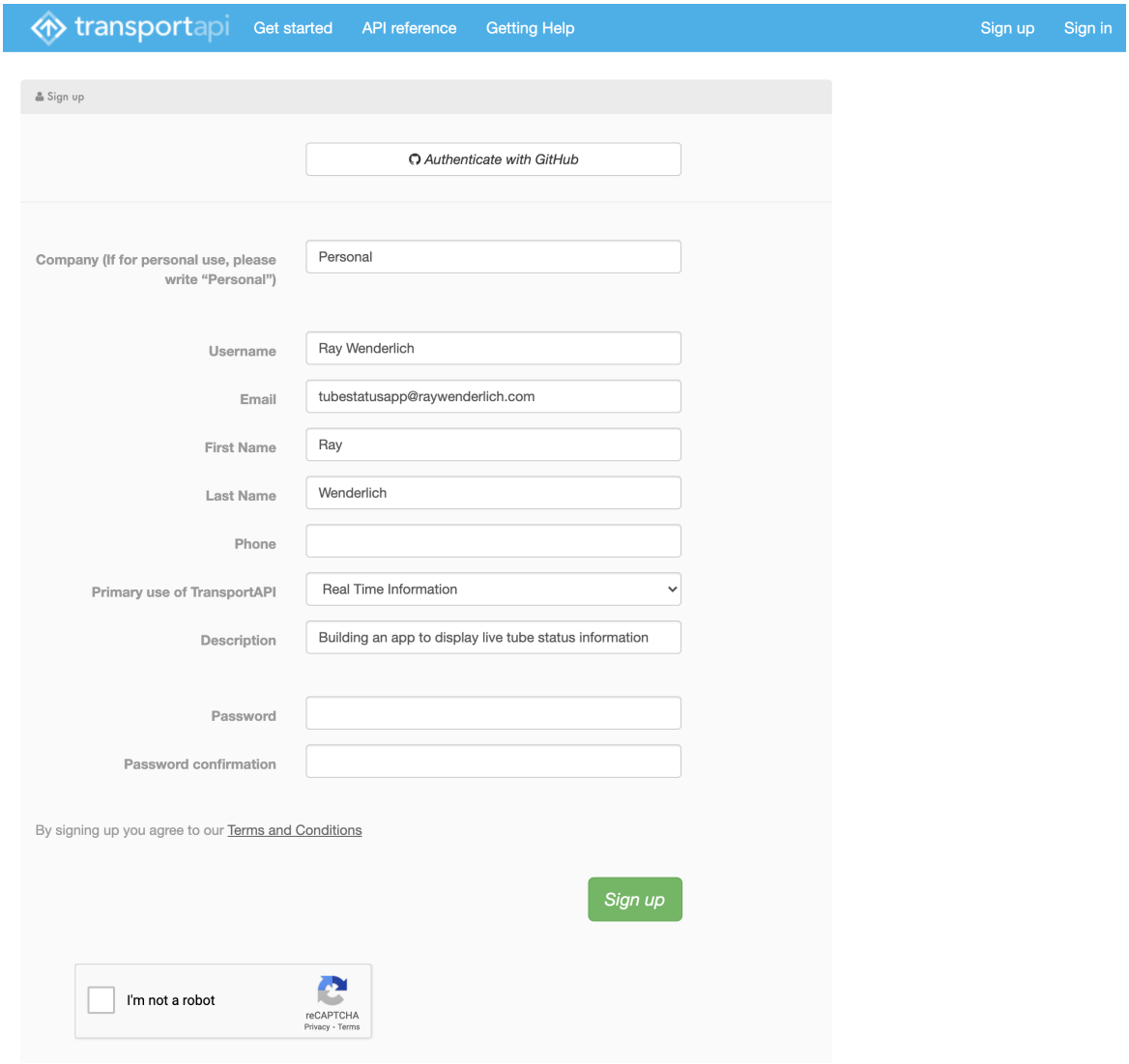
This app uses a free third-party API called ***TransportAPI*** to get up-to-date information about the various Tube lines. Before you begin, you need to register and create a new ***app***.



Go to the TransportAPI Developer Portal(https://developer.transportapi.com/) and click ***Sign up*** in the header.

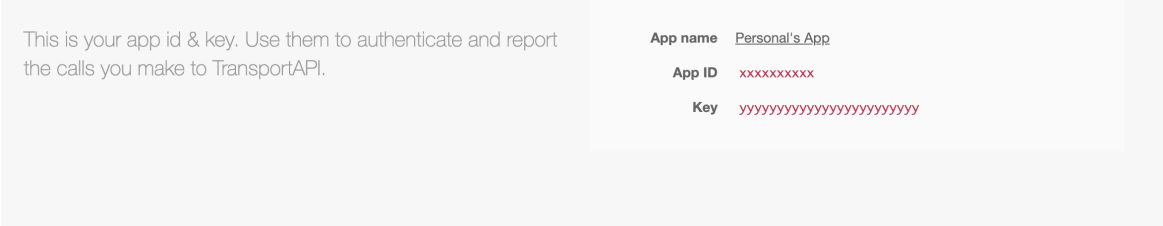


Fill out the form, check the reCAPTCHA checkbox, and click ***Sign up***. A confirmation screen will ask you to check your email.



You'll receive two emails. The first contains details on how to activate your account. The second confirms that you've successfully created a new ***Application Key***.

Follow the instructions in the first email to activate your account, then sign in to your new account in the Developer Portal(https://developer.transportapi.com/). The landing page shows you the credentials — ***App ID*** and ***App Key*** — that have just been created. Keep this tab open, as you'll use these values shortly.



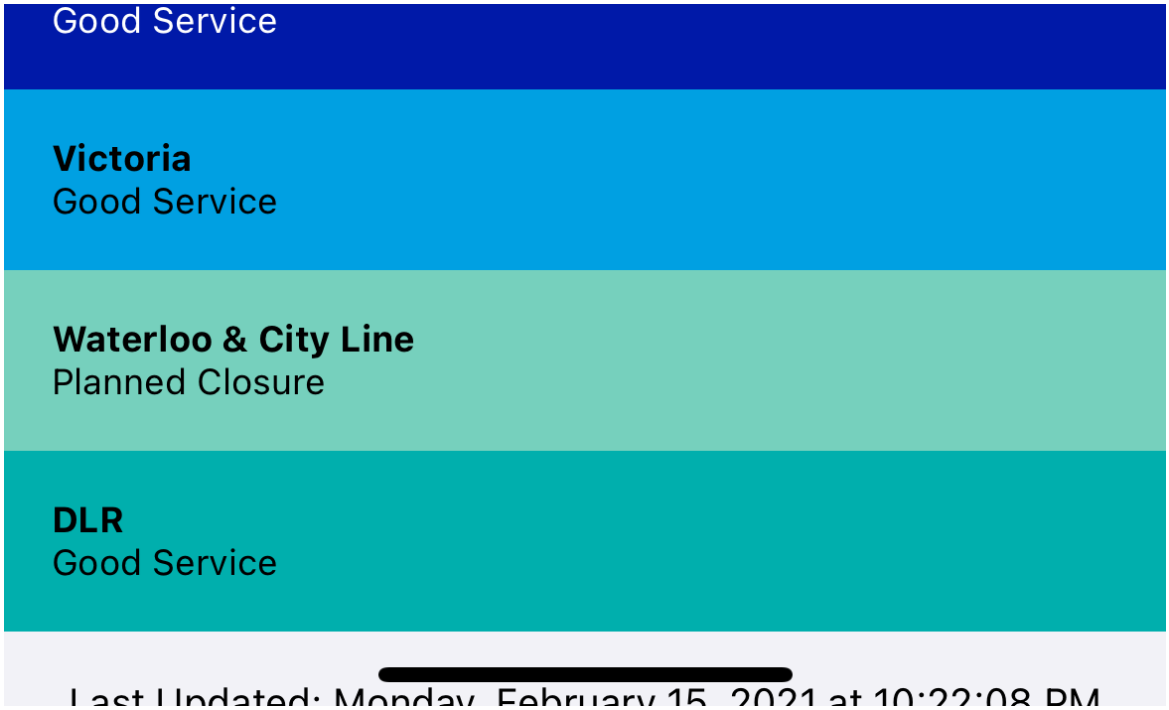
Connecting the API

Now, it's time to connect the Transport API to your app.

First, open ***Debug.xcconfig***. Here, you'll find `TRANSPORT_API_SERVICE_APP_ID` and `TRANSPORT_API_SERVICE_APP_KEY` with dummy values. Set each to the values provided by the Transport API Developer Portal.

Build and run your app — at last!


10:27
Tube Status
Refresh
Bakerloo
Part Suspended
Central
Good Service
Circle
Good Service
District
Good Service
Hammersmith & City
Minor Delays
Jubilee
Minor Delays
Metropolitan
Good Service
Northern
Good Service
Piccadilly



Note: Xcode Configuration (***.xcconfig***) files are great for providing build-specific configuration for your app. For example, you could use a staging server URL when running on the simulator and a different URL in production.

However, you shouldn't store your API key in the app like this, even if you use an .xcconfig. In a production app, you should proxy requests from your app through you own server, and have that server make the requests to the Transport API on your behalf. You can do this easily using a service like AWS Lambda(https://aws.amazon.com/lambda/).

Then, if you need to change API key, or if the Transport API introduces a breaking change, you only need to update your server rather than requiring your users to download a new version of your app!

Understanding SF Symbols

Now that the basic app is up and running, you'll spend the rest of this tutorial learning how to add some pizazz in the form of SF Symbols.

SF Symbols are currently available in three versions:

- Version 1.1 is available on iOS/iPadOS/tvOS 13 and watchOS 6.
- Version 2.0 is available on iOS/iPadOS/tvOS 14 and watchOS 7.0.
- Version 2.1 is available on iOS/iPadOS/tvOS 14.2 and watchOS 7.2.

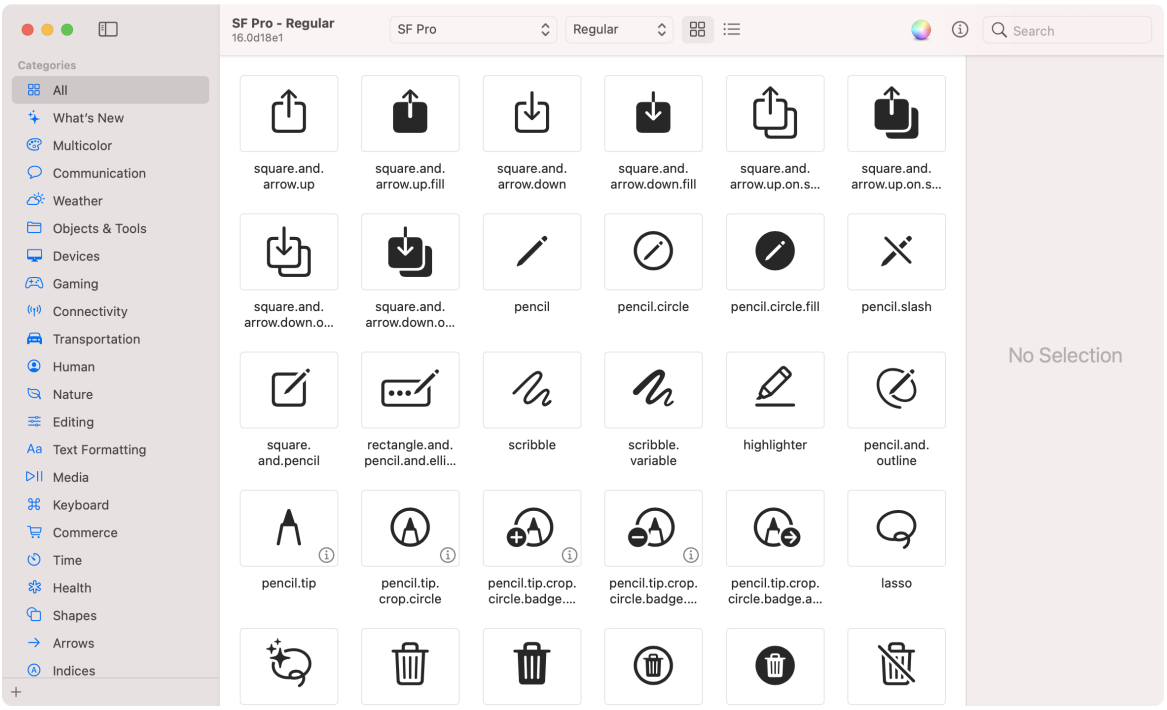
All versions are also available with macOS Big Sur.

As well as adding nearly 900 symbols, Version 2 of SF Symbols also introduced over 160 multicolor symbols, localized variants and improvements to how symbols can be aligned horizontally.

> ***Note***: Some symbols have changed their name between versions. While SF Symbols supports old names for backward compatibility, you should make sure that any symbols you use in your app work on all versions you intend to support.

Viewing Available Symbols

Apple has released an SF Symbols app for macOS(https://developer.apple.com/sf-symbols/) showcasing all the available symbols. Download the app and open it.



The left-hand panel acts as a filter, limiting which symbols are shown based on their category.

The top pane allows you to:

Alter the font and weight of the displayed symbols.

Switch the layout between grid or list.

Toggle multicolor preview.

Filter symbols by name.

When you click the ***i*** button on the top bar, a right-hand pane opens. This pane provides a detail view of any selected symbol, including which platforms it's available on and any restrictions for its use.

Finally, the main pane displays all the relevant symbols based on the options selected.

Using SF Symbols

It's finally time to bling up your app. In Xcode, open ***TFLLineStatus.swift*** in the ***LineData*** group. This file defines an `enum` containing all the line status values that the API supports. There are a lot of them!

At the end of the file, before the final closing brace, add the following code:

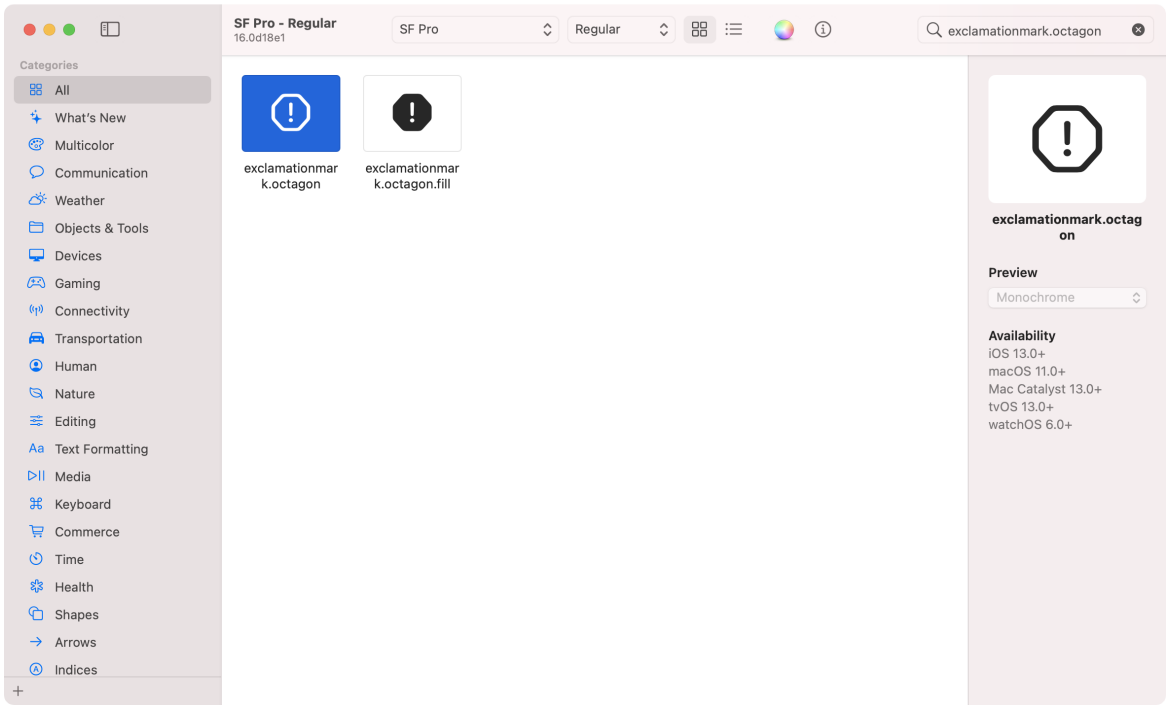
```
// 1
func image() -> Image {
  switch self {
  default:
    // 2
    return Image(systemName: "exclamationmark.octagon")
  }
}
```

In this code, you:

. Add a new method, `image()`, to `TFLLineStatus`.

. Use the new `init(systemName:)` on `Image` to create an image with the ***exclamationmark.octagon*** SF Symbol.

Search for ***exclamationmark.octagon*** in the SF Symbols app.



Next, you'll use this image when displaying the status for a line. Open ***LineStatusRow.swift***.

Add the following to `body` as the first child of `HStack`, before the `VStack`:

```
// 1
status.image()
  // 2
  .font(.title)
  .padding(.trailing)
  .foregroundColor(lineColor.contrastingTextColor)
```

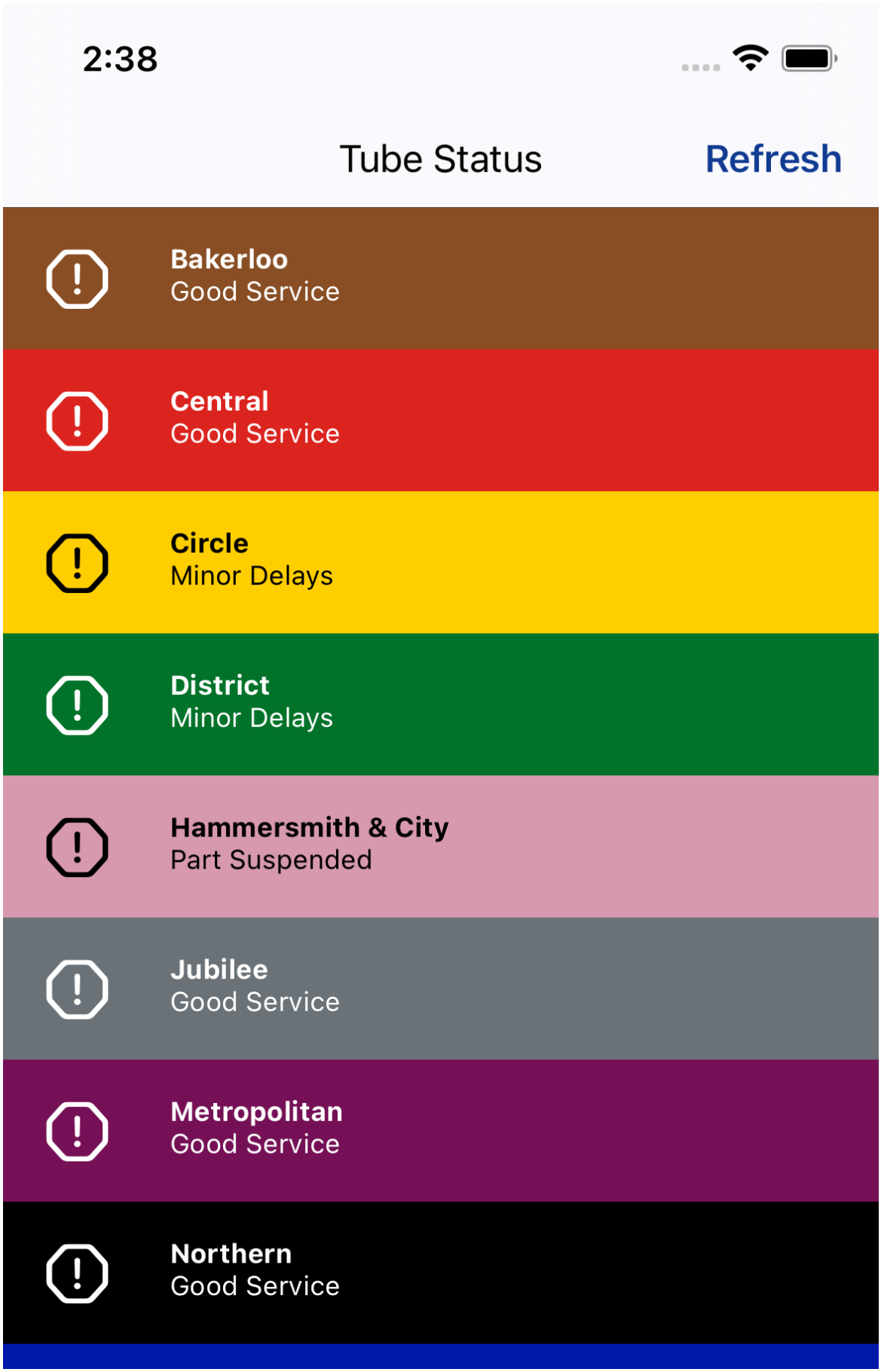
Here, you are:

. Calling `status.image()`, which you defined on `TFLLineStatus`, to insert the status image into the leading side of the `HStack`.

. Setting font style, padding and foreground color properties on the image using view modifiers. The foreground color is set such that it contrasts nicely with the row's background color.

Notice how you can call `font(_:)` on `Image`. Because SF Symbols are designed to work with the San Francisco font system, they automatically pick the right variant based on the font you provide. Neat!

Build and run the app.


2:38
Tube Status
Refresh
Bakerloo
Good Service
Central
Good Service
Circle
Minor Delays
District
Minor Delays
Hammersmith & City
Part Suspended
Jubilee
Good Service
Metropolitan
Good Service
Northern
Good Service

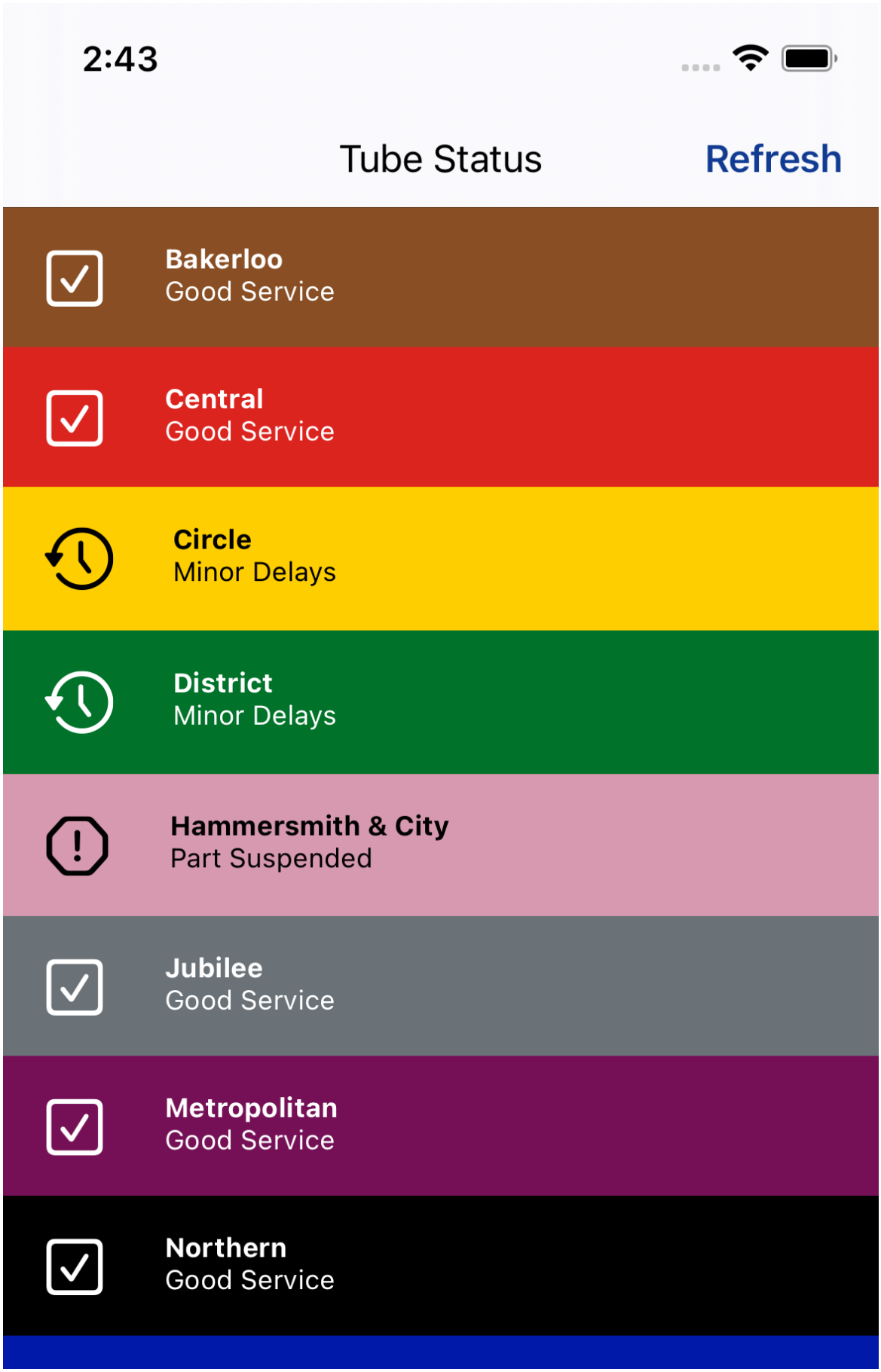
Voilà, you've just added your first SF Symbol into the app. Congratulations! :]

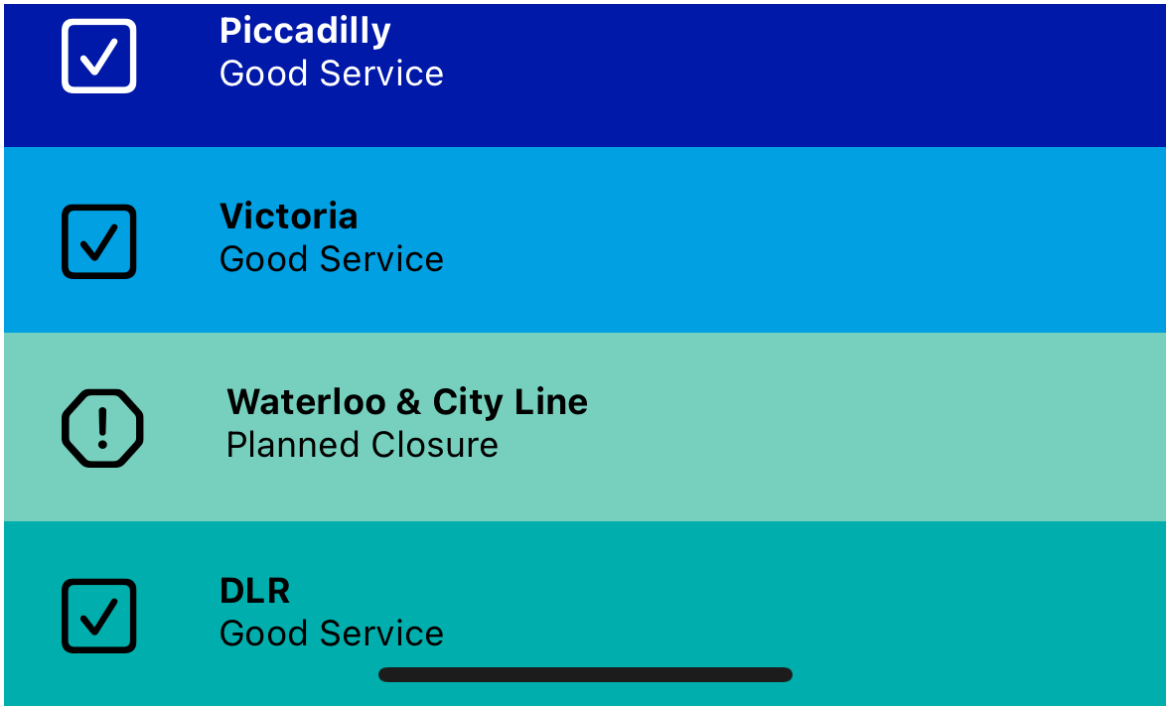
But, using the same symbol for every status code isn't too helpful for the user. To fix that, go back to ***TFLLineStatus.swift***. Place the following in the body of `switch` before `default`:

```
case .closed:
  return Image(systemName: "exclamationmark.octagon")
case .suspended:
  return Image(systemName: "nosign")
case .severeDelays:
  return Image(systemName: "exclamationmark.arrow.circlepath")
case .reducedService:
  return Image(systemName: "tortoise")
case .busService:
  return Image(systemName: "bus")
case .minorDelays:
  return Image(systemName: "clock.arrow.circlepath")
case .goodService:
  return Image(systemName: "checkmark.square")
case .changeOfFrequency:
  return Image(systemName: "clock.arrow.2.circlepath")
case .notRunning:
  return Image(systemName: "exclamationmark.octagon")
case .issuesReported:
  return Image(systemName: "exclamationmark.circle")
case .noIssues:
  return Image(systemName: "checkmark.square")
case .plannedClosure:
  return Image(systemName: "hammer")
case .serviceClosed:
  return Image(systemName: "exclamationmark.octagon")
case .unknown:
  return Image(systemName: "questionmark.circle")
```

In this code, you're picking out several common status codes and providing custom SF Symbols for each. Any codes not specified will continue to use the ***exclamationmark.octagon*** symbol from the switch's `default` case.

Build and run the app again. Your experience may vary from the image below depending on the state of the Tube system at the time you're running the app. But hopefully, you'll see many types of statuses displaying different images.


2:43
Tube Status
Refresh
Bakerloo
Good Service
Central
Good Service
Circle
Minor Delays
District
Minor Delays
Hammersmith & City
Part Suspended
Jubilee
Good Service
Metropolitan
Good Service
Northern
Good Service



Neat! Hopefully, you're starting to see how powerful SF symbols can be!

Testing with Mock Data

In the previous section, you chose different SF Symbols for different line statuses. However, you haven't yet been able to see how each of them looks, since your app only renders the current status of the lines. Now, you'll explore using mock data to test the full range of statuses.

Naïve Mock Data Approaches

You could wait around until each status occurs in real life, then quickly open the app. But you might be waiting a long time. :]

Another option is to add many ***Swift UI previews*** to `LineStatusRow`, setting the properties appropriately. This works, but it's clumsy.

Each preview displays on its own on a phone screen background. Interactivity isn't available, and worst of all, because `LineStatusRow` is a purely presentational view, you're only checking that the values you provide in the preview are rendered correctly.

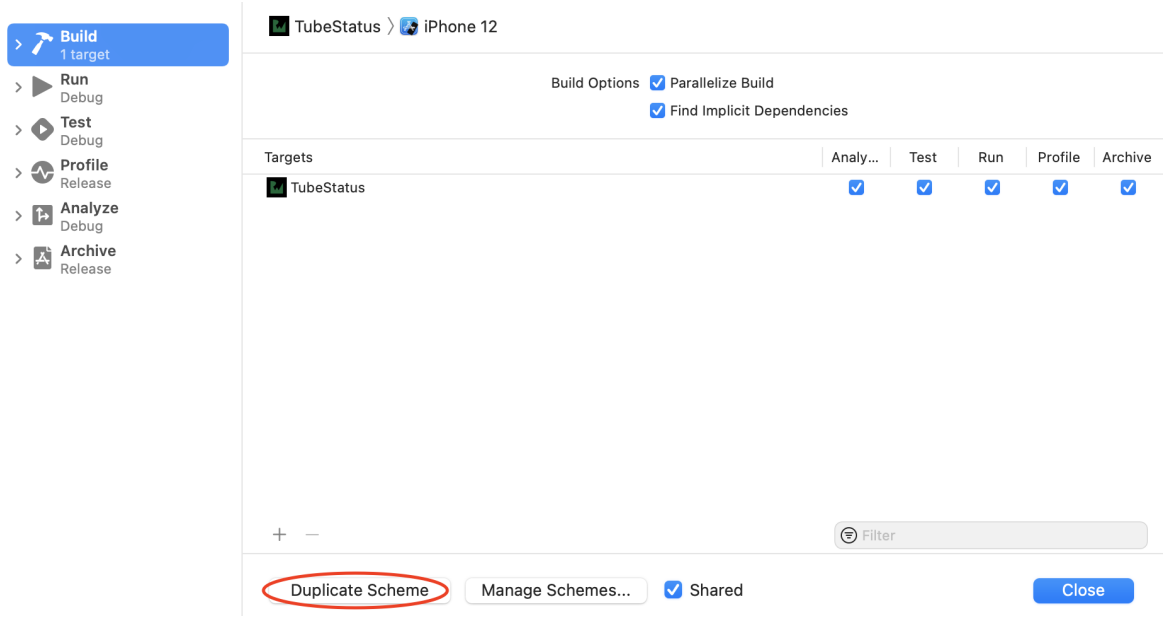
Another approach would be using unit tests and mock data. This is a pretty good approach but still lacks the interactivity element.

Using Mock Data with Environment Variables

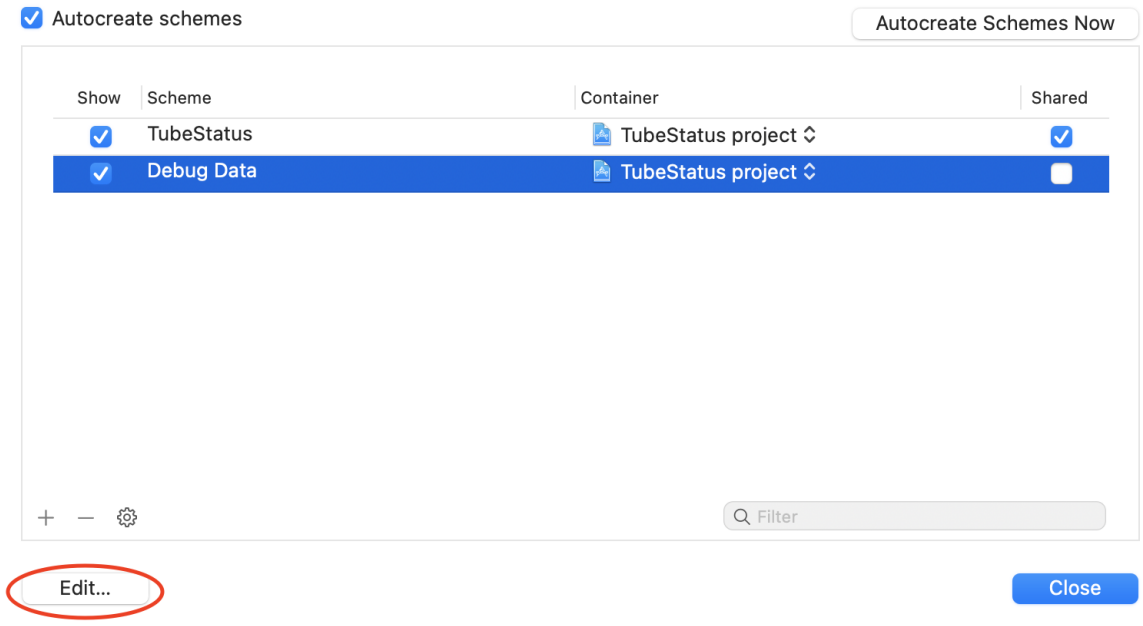
Another approach that may be more useful is to configure your app with mock data based on an environment variable. That way, you can choose to build your app with whatever data you wish and play

with the app on the simulator or your device as if it were the real thing.

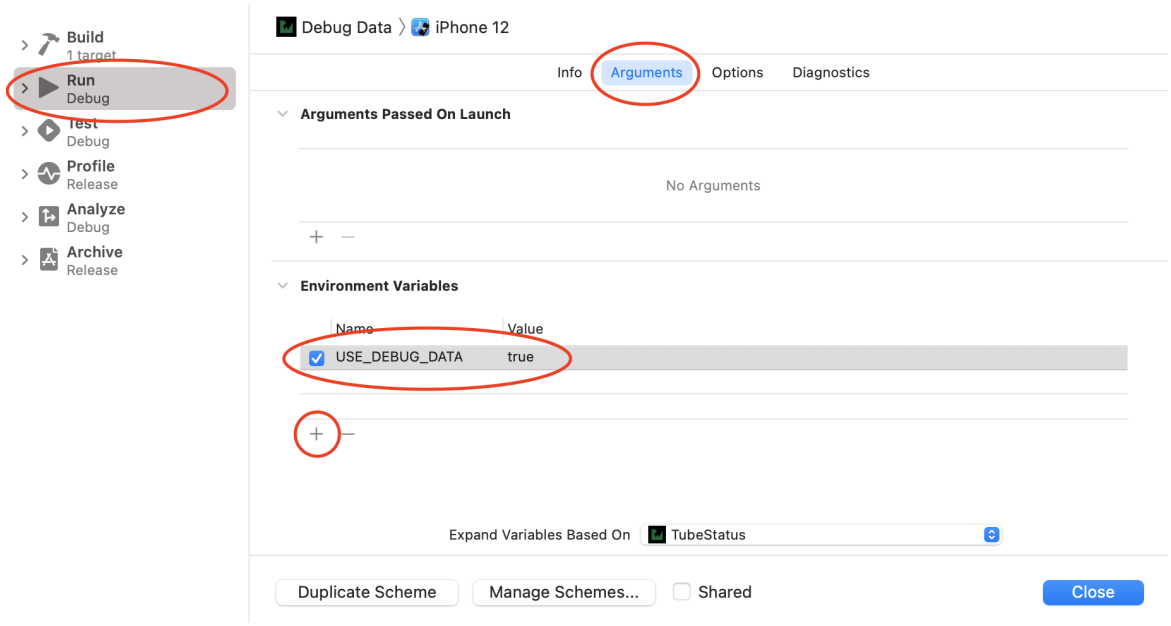
In Xcode, select ***Product ▸ Scheme ▸ Edit Schemes...*** and select ***Duplicate Scheme***.



Name the new scheme ***Debug Data*** and click ***Close***. Then, select ***Product ▸ Schemes ▸ Manage Schemes...***, select the ***Debug Data*** scheme and select ***Edit...***.



Select ***Run*** in the left-hand menu and then the ***Arguments*** tab. Click the + icon under ***Environment Variables*** and create a new environment variable called ***USE_DEBUG_DATA*** with a value of ***true***. Click ***Close***.



Your app now has two schemes, identical except that the ***DebugData*** scheme passes your new environment variable into the build environment.

Next, open ***DebugLineData.swift*** and add the following code immediately after the import declarations:

```
// 1
let bakerlooLineDebug = LineData(
  name: "BakerlooDebug",
  color: Color(red: 137 / 255, green: 78 / 255, blue: 36 / 255))
let centralLineDebug = LineData(
  name: "CentralDebug",
  color: Color(red: 220 / 255, green: 36 / 255, blue: 31 / 255))
let circleLineDebug = LineData(
  name: "CircleDebug",
  color: Color(red: 255 / 255, green: 206 / 255, blue: 0 / 255))
let districtLineDebug = LineData(
  name: "DistrictDebug",
  color: Color(red: 0 / 255, green: 114 / 255, blue: 41 / 255))
let hammersmithAndCityLineDebug = LineData(
  name: "Hammersmith & CityDebug",
  color: Color(red: 215 / 255, green: 153 / 255, blue: 175 / 255))
let jubileeLineDebug = LineData(
  name: "JubileeDebug",
  color: Color(red: 106 / 255, green: 114 / 255, blue: 120 / 255))
let metropolitanLineDebug = LineData(
  name: "MetropolitanDebug",
  color: Color(red: 117 / 255, green: 16 / 255, blue: 86 / 255))
let northernLineDebug = LineData(
  name: "NorthernDebug",
  color: Color(red: 0 / 255, green: 0 / 255, blue: 0 / 255))
let piccadillyLineDebug = LineData(
  name: "PiccadillyDebug",
  color: Color(red: 0 / 255, green: 25 / 255, blue: 168 / 255))
let victoriaLineDebug = LineData(
  name: "VictoriaDebug",
  color: Color(red: 0 / 255, green: 160 / 255, blue: 226 / 255))
```

Then, add the following between the square brackets of `lineStatus` inside the `debugData` constant declaration at the bottom:

```
// 2
LineStatus(line: bakerlooLine, status: .specialService),
LineStatus(line: centralLine, status: .closed),
LineStatus(line: circleLine, status: .suspended),
LineStatus(line: districtLine, status: .partSuspended),
LineStatus(line: hammersmithAndCityLine, status: .plannedClosure),
LineStatus(line: jubileeLine, status: .partClosure),
LineStatus(line: metropolitanLine, status: .severeDelays),
LineStatus(line: northernLine, status: .reducedService),
LineStatus(line: piccadillyLine, status: .busService),
LineStatus(line: victoriaLine, status: .minorDelays),
LineStatus(line: waterlooAndCityLine, status: .goodService),
LineStatus(line: dlr, status: .partClosed),
// 3
LineStatus(line: bakerlooLineDebug, status: .exitOnly),
LineStatus(line: centralLineDebug, status: .noStepFreeAccess),
LineStatus(line: circleLineDebug, status: .changeOfFrequency),
LineStatus(line: districtLineDebug, status: .diverted),
LineStatus(line: hammersmithAndCityLineDebug, status: .notRunning),
LineStatus(line: jubileeLineDebug, status: .issuesReported),
LineStatus(line: metropolitanLineDebug, status: .noIssues),
LineStatus(line: northernLineDebug, status: .information),
LineStatus(line: piccadillyLineDebug, status: .serviceClosed),
LineStatus(line: victoriaLineDebug, status: .unknown)
```

This code:

- Creates several “fake” tube lines. Your app has 21 status codes, but only 12 lines. So you created an additional 9 lines to make sure there are enough lines to display each code.
- Adds `LineStatus` items to `DebugData`‘s `lineStatus`. This first set adds a different status code to each of the “real” tube lines.
- The second set adds the remaining status codes to the fake tube lines you created.

Switching Between Mock Data and Actual Data

Now, look at ***DebugDataService.swift***. There are only a few lines of code here, but quite a lot going on! This is what the code in this file does:

- First, the file imports `Combine` to give it access to the `Future` class.
- Next, it defines `DebugDataService` as conforming to `TubeLinesStatusFetcher`.
- Then, it implements `fetchStatus` — the only method required by `TubeLinesStatusFetcher`.
- Finally, it returns `debugData` wrapped in a `Future`. This `debugData` is the data you added in the previous section.

To use the debug line status fetcher, create a Swift file called ***TubeLinesStatusFetcherFactory.swift*** and add the following code:

```
// 1
enum TubeLinesStatusFetcherFactory {
  // 2
  static func new() -> TubeLinesStatusFetcher {
    // 3
    #if DEBUG
    if ProcessInfo.processInfo.environment["USE_DEBUG_DATA"] == "true" {
      return DebugDataService()
    }
    #endif
    // 4
    return TransportAPIService()
  }
}
```

Here's what's happening:

. As you only want static methods on this entity, it's implemented as a ***caseless enum***. You could use a `struct` here instead, but it would be possible to needlessly instantiate a `struct`, so an `enum` is a better choice.

. Define a single static method, `new()`, which returns an object conforming to `TubeLineStatusFetcher`.

. If running in debug mode and `USE_DEBUG_DATA` is set to `true`, return an instance of `DebugDataService` created previously.

. Otherwise, return an instance of `TransportAPIService`, which fetches the real data from the Transport API.

Finally, open ***AppMain.swift*** and find this line:

```
model: TubeStatusViewModel(tubeLinesStatusFetcher: TransportAPIService())
```

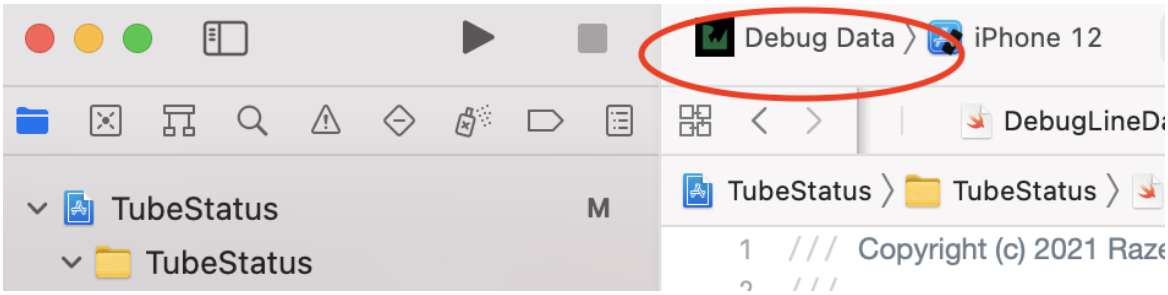
Replace it with the following:

```
model: TubeStatusViewModel(
  tubeLinesStatusFetcher: TubeLinesStatusFetcherFactory.new())
```

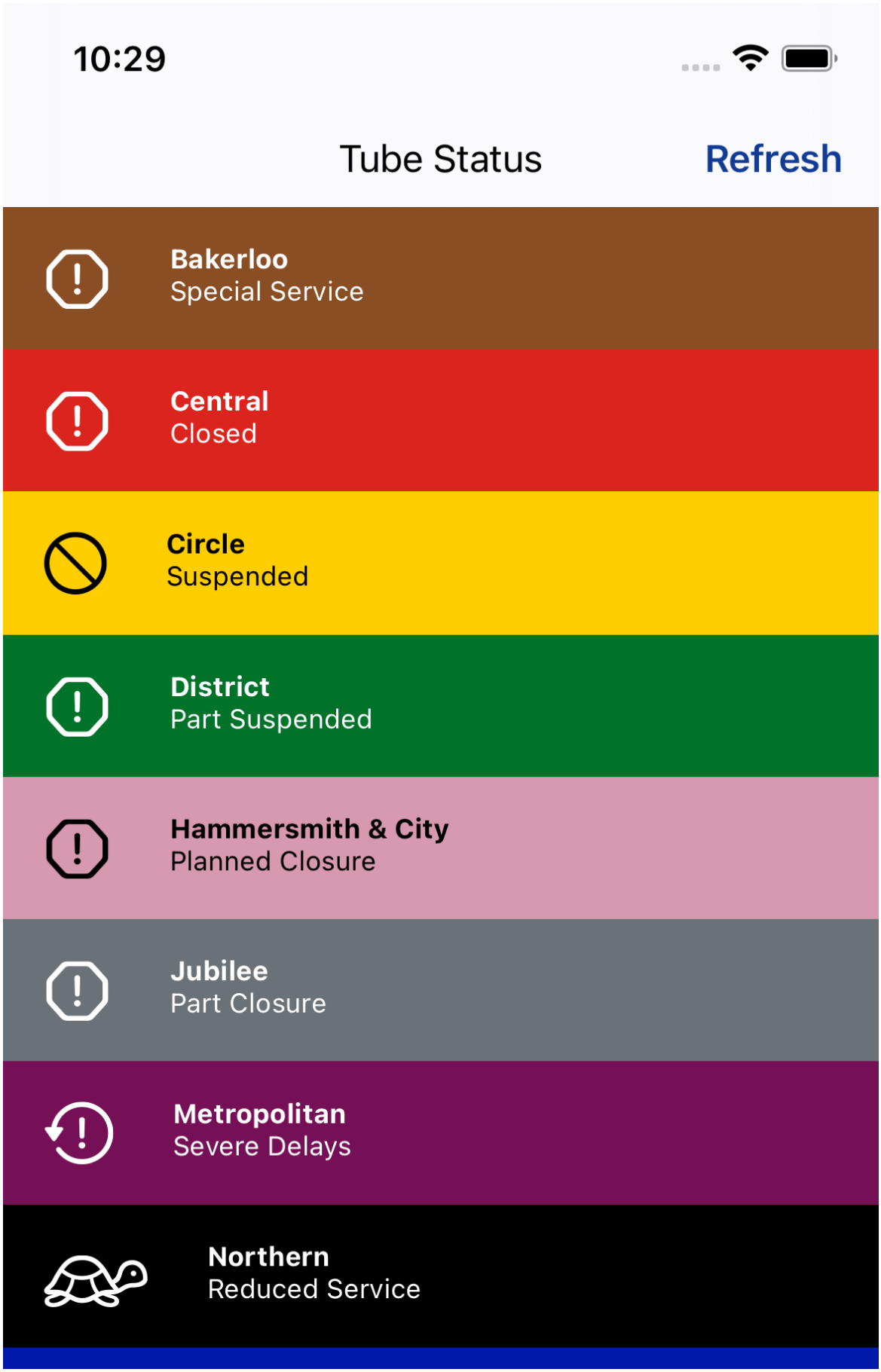
Here, you're adding another layer of indirection and initializing `TubeStatusViewModel` with whatever fetcher ***TubeLinesStatusFetcherFactory*** decides to provide for it rather than using `TransportAPIService` directly.

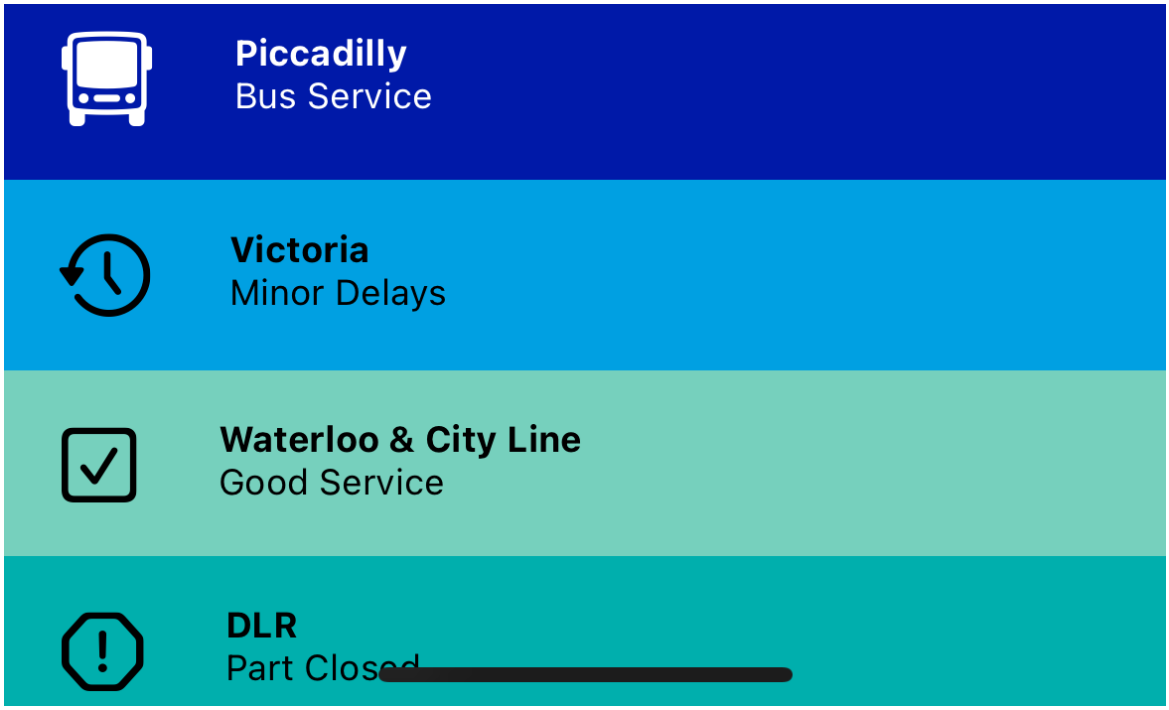
Using the DebugData Scheme

Now, time to test it out! Make sure to select the ***Debug Data*** scheme.



Now, build and run the app.


10:29
Tube Status
Refresh
Bakerloo
Special Service
Central
Closed
Circle
Suspended
District
Part Suspended
Hammersmith & City
Planned Closure
Jubilee
Part Closure
Metropolitan
Severe Delays
Northern
Reduced Service



This is a great example of the power and flexibility of both protocol-oriented programming (https://developer.apple.com/videos/play/wwdc2015/408/) and dependency injection.

By passing the data-fetching service as a dependency into `TubeStatusViewModel`, it was simple to replace how the data was fetched with a different implementation.

By only providing `TubeStatusViewModel` with the protocol that the data-fetching service uses to return the data — and not the implementation — the view model doesn't deal with how the data is fetched. It could be hard-coded or downloaded via a JSON API.

Understanding Restrictions on Using SF Symbols

Before you go crazy and add SF Symbols everywhere, be aware that restrictions exist on where and how you can use them.

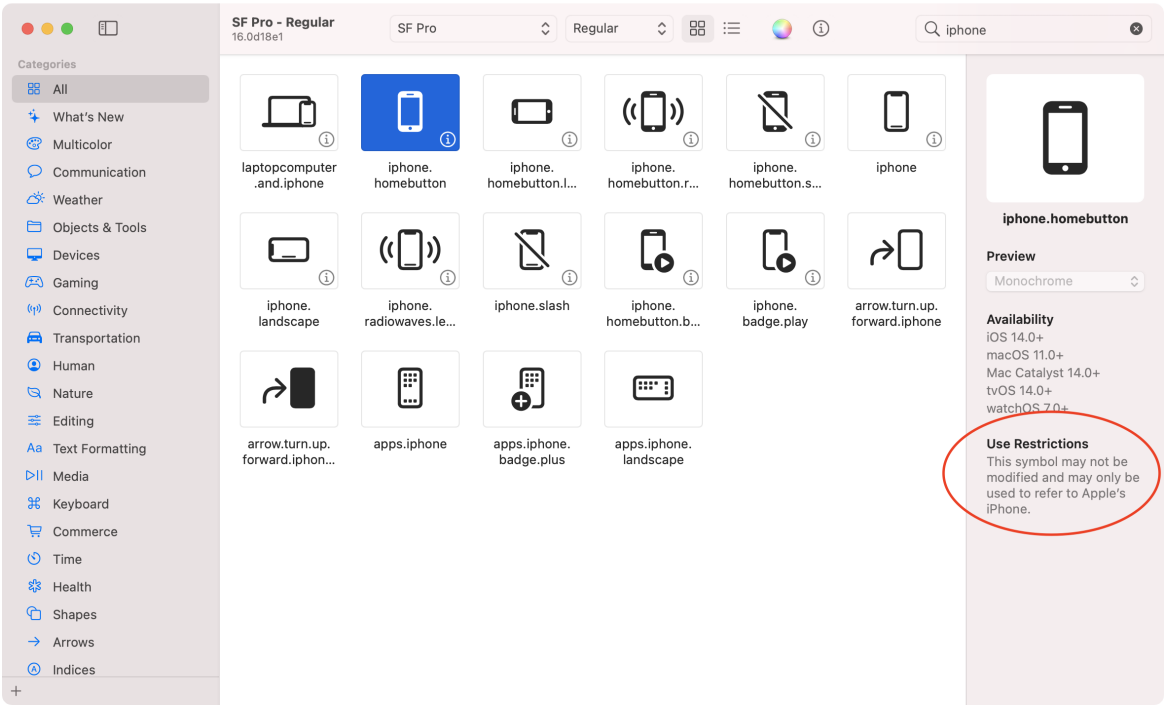
Quoted directly from Apple's Human Interface Guidelines (https://developer.apple.com/design/human-interface-guidelines/sf-symbols/overview/):

> "You may not use SF Symbols — or glyphs that are substantially or confusingly similar — in your app icons, logos, or any other trademark-related use. Apple reserves the right to review and, in its sole discretion, require modification or discontinuance of use of any Symbol used in violation of the foregoing restrictions, and you agree to promptly comply with any such request."

Additionally, SF Symbols are considered to be system-provided images and thus are covered by the Xcode and Apple SDK license agreements(https://developer.apple.com/terms/).

Furthermore, 124 SF Symbols aren't allowed to be exported, modified or used for any purpose other than Apple-specific technologies.

Apple publishes a full list (https://developer.apple.com/design/human-interface-guidelines/sf-symbols/overview/#symbols-for-use-as-is) of these more restricted icons. The right-hand pane in the SF Symbols app also details any extra restrictions when you select a symbol.



Creating Custom SF Symbols

Even though Apple has provided thousands of different symbols in the SF Symbol library, it's impossible to cover every conceivable image you may need in your app. What Apple has done instead is make it really easy for you to build your own custom symbols, when needed.

SF Symbols are built as vector graphics in an SVG file with a very specific formatting. At the top level, the file must contain three layers: ***Symbols***, ***Guides*** and ***Notes***. Each of these layers then contains sub-layers. For example, the ***Symbols*** layer contains 27 sub-layers — one for each of the variants available.

Furthermore, Apple makes it easy to create your own symbols by allowing you to export existing symbols from the SF Symbols app. That way, all the formatting is already present and you just need to change whatever you want to customize.

When building a custom symbol, it's a good idea to find a built-in symbol that's as close as possible to what you're trying to draw and then adapt it to your needs.

You're now going to see how to add a custom symbol into your app.

Making an "Information" Symbol

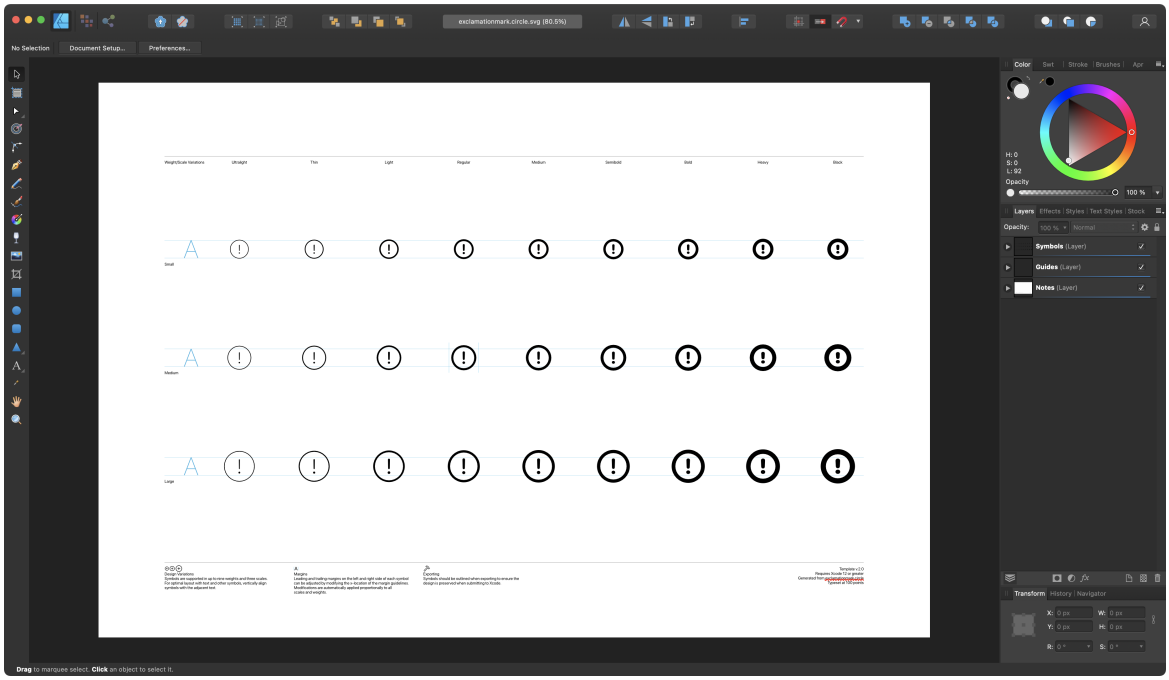
For this section, you need a vector art app that can edit SVG images.

Many different options are available, including paid products like Adobe Illustrator (https://www.adobe.com/products/illustrator.html), Sketch(https://www.sketch.com/), Figma(https://www.figma.com) and Affinity Designer(https://affinity.serif.com/). Most of these offer free trials. Open-source products are also available, like Inkscape(https://inkscape.org/) and OpenOffice Draw(https://www.openoffice.org/product/draw.html).

This tutorial uses Affinity Designer, but the process should be similar in other vector art apps.

Importing the Exclamation Mark Symbol

Open the SF Symbols app on your Mac and search for the ***exclamationmark.circle*** symbol. Select it, and choose ***File ▸ Export Custom Symbol Template...***. Save the symbol template on your machine and open it in your vector graphics app.

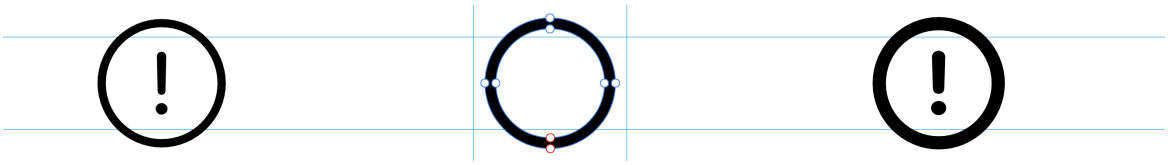


As you can see, the SVG file contains 27 separate images for each of the 9 font weights and 3 sizes.

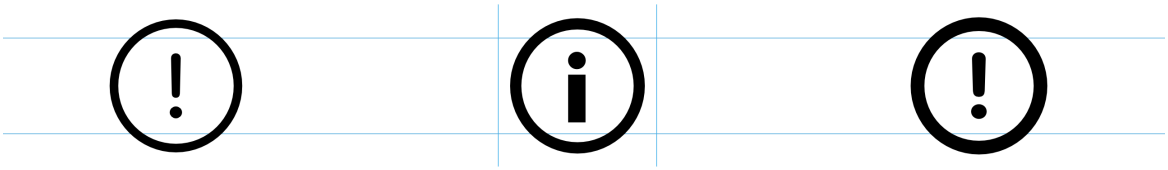
Updating all 27 images would take a long time, but fortunately you don't need to change any that you aren't going to use. The only variant you need here is ***Regular-M***, as that's all this app uses. Apple recommends you create Regular S/M/L and Semi-bold S/M/L, as many of the common UIKit controls use these variants.

Customizing the Exclamation Mark Symbol

Find the ***Regular-M*** variant near the center of the canvas. Each symbol is built as a group in the SVG. Delete the exclamation mark in the center of the circle. You may need to ungroup the layer first, depending on which vector art app you're using.



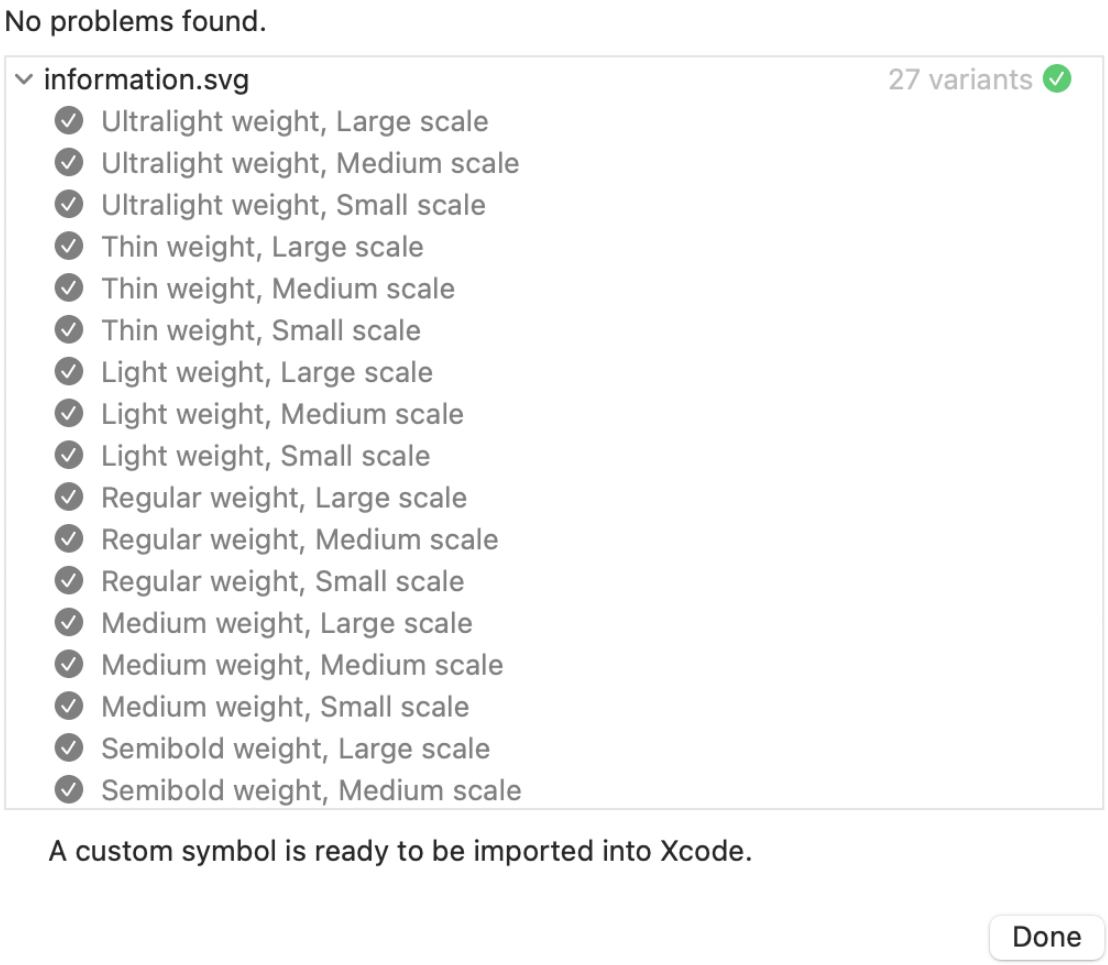
Next, add a text block to the layer and type a lowercase ***i***. Size it to ***64pt*** in ***SF Pro*** and weight ***Heavy***. Then place it at the center of the circle.



Each variant in a custom SF Symbol can only contain shapes/curves. They must not contain text, bitmaps or any other type of object. Convert your ***i*** text layer into a curve. In Affinity Designer, do this by selecting ***Layer ▸ Convert to Curves***.

Make sure your new layer is a sub-layer of the Regular-M group. Then, save or export your new symbol as an SVG called ***information.svg***.

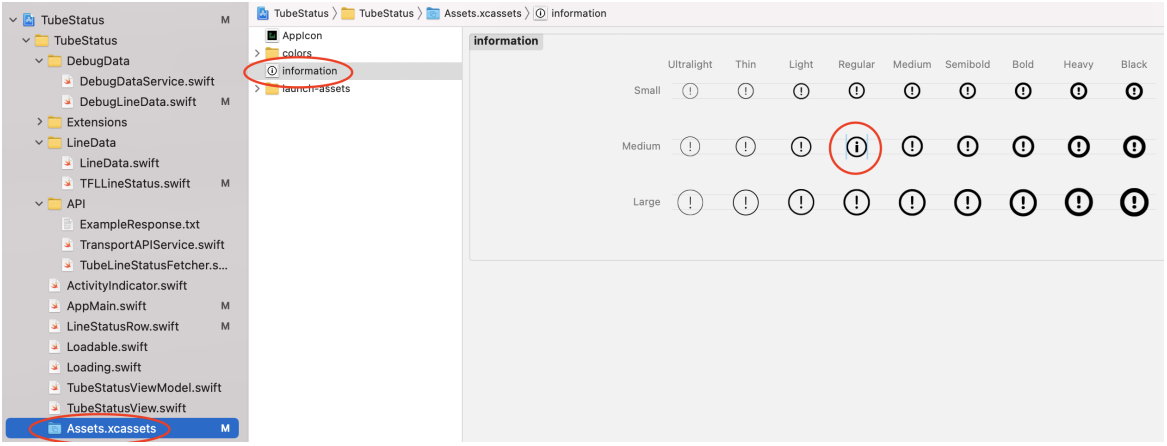
Switch to the SF Symbols app and select ***File ▸ Validate Custom Symbols…***, then select the file you just saved. If you've done everything correctly, your new symbol will be validated — yay!



Using Custom Symbols

Using a custom symbol in your app is incredibly easy. Switch to Xcode and select the main asset catalog, ***Assets.xcassets***, in the Project navigator on the left. Switch to ***Finder*** and drag your new SF Symbol into the asset catalog.

Confirm that the Regular-M variant is the letter “i” in a circle rather than an exclamation mark in a circle.



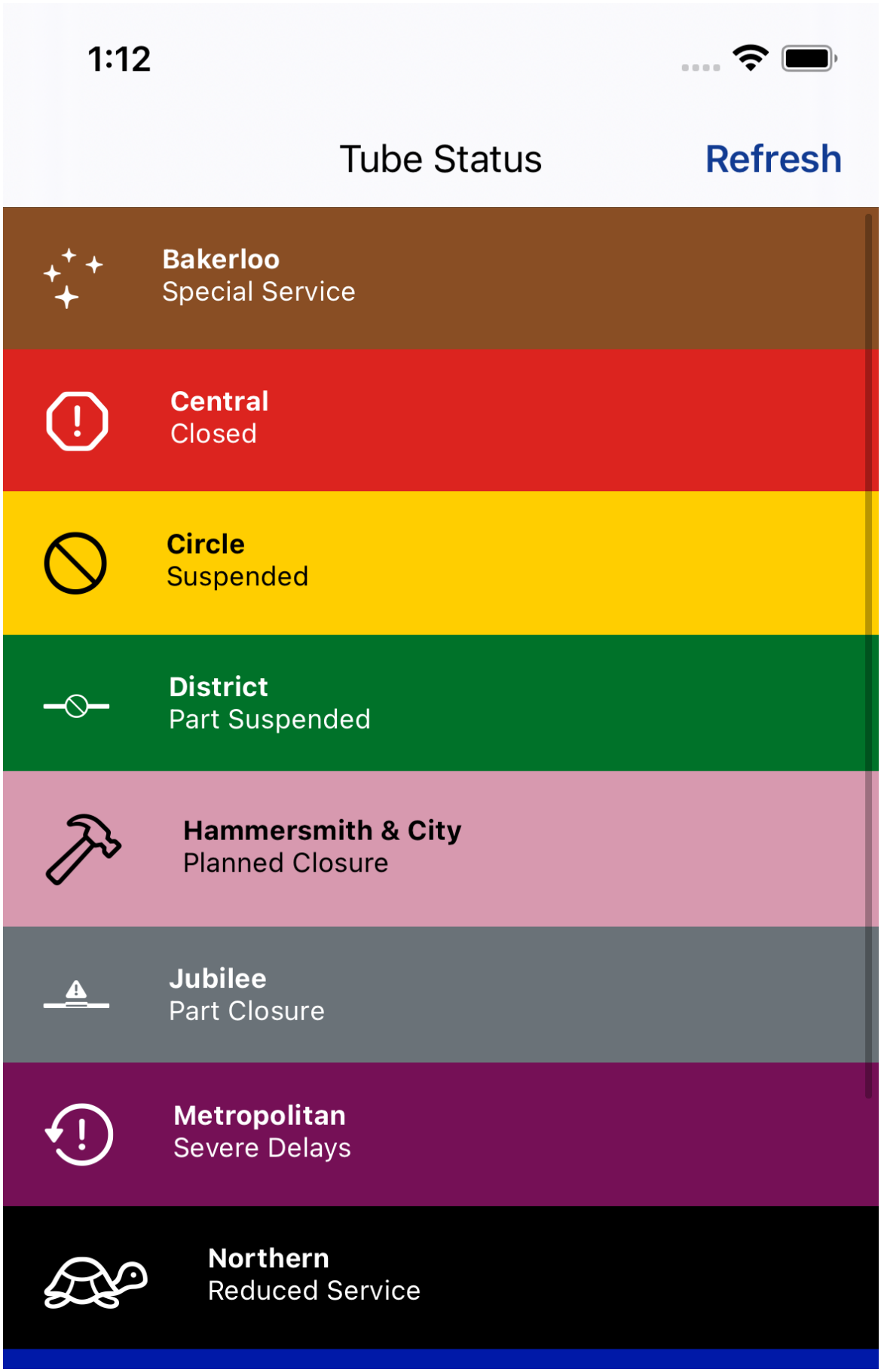
Next, switch to Finder and open the ***Custom Symbols*** directory in the materials you downloaded at the start of the tutorial. Drag all six custom symbols into the asset catalog as well. Alternatively, use your vector graphics app and newfound knowledge to create your own set of custom symbols. :]

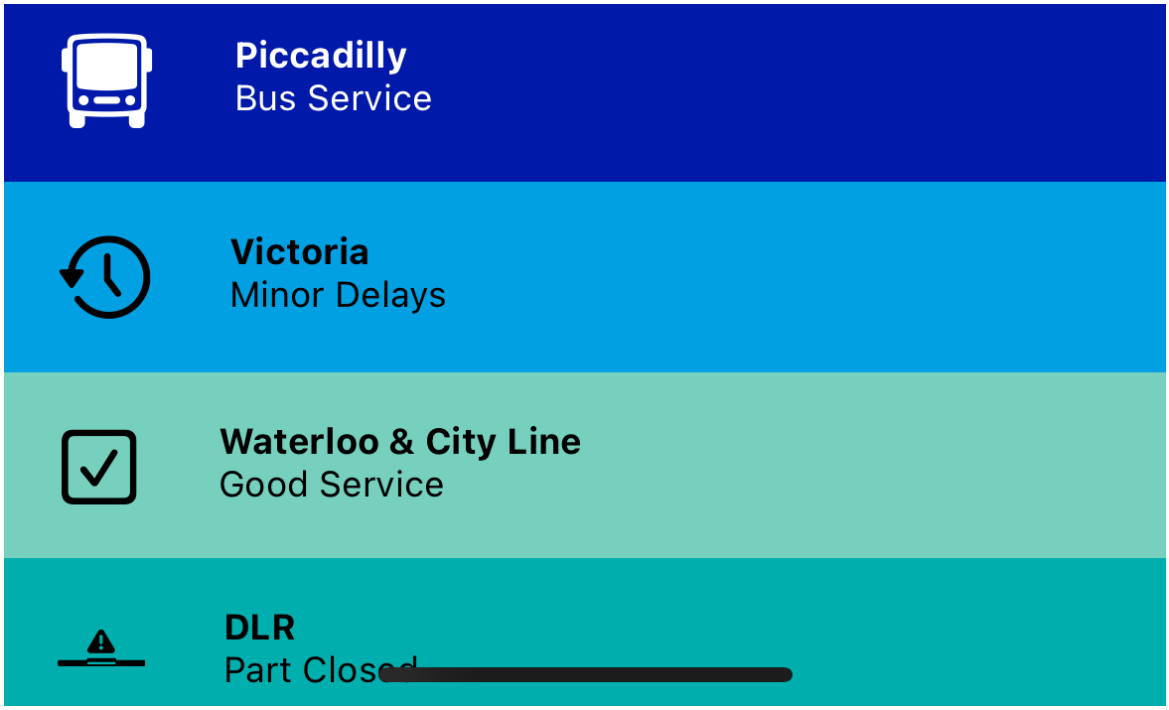
Loading a custom symbol uses a different API than the one used to load one of the provided SF Symbols. Rather than calling `init(systemName:)` on `Image`, you can just use the default initializer, `init()`.

Switch back to Xcode and open ***TFLLineStatus.swift***. At the bottom of `image()`, replace `default` with the following code:

```
case .specialService:
  return Image("special.service")
case .partSuspended:
  return Image("part.suspended")
case .partClosure:
  return Image("part.closure")
case .partClosed:
  return Image("part.closure")
case .exitOnly:
  return Image("exit.only")
case .noStepFreeAccess:
  return Image("no.step.free.access")
case .diverted:
  return Image("diverted")
case .information:
  return Image("information")
```

Ensure the ***Debug Data*** scheme is still selected. Then, build and run your app.


1:12
Tube Status
Refresh
Bakerloo
Special Service
Central
Closed
Circle
Suspended
District
Part Suspended
Hammersmith & City
Planned Closure
Jubilee
Part Closure
Metropolitan
Severe Delays
Northern
Reduced Service



Ta-da, you've successfully added custom symbols into your Tube status app! :]

Supporting Older Operating Systems

And finally, a quick tip!

If you need to support older operating system versions and still want to use SF Symbols in your app, here's a workaround to help. Simply export the SVG from the SF Symbols macOS app and open it in a vector graphics application like Sketch, Affinity Designer or Figma.

Export the layer you want to use as a PNG and add it as an asset in your app's asset catalog. Then, simply use the default initializer rather than using `init(systemName:)` in your `Image`.

You won't have access to the advanced features that SF Symbols provide, but you *can* still use the images.

Where to Go From Here?

You can download the completed project files by clicking ***Download Materials*** at the top or bottom of the tutorial.

As you can see, using both the built-in SF Symbols and providing your own custom symbols is incredibly easy with Swift.

If you'd like to see a video on how to build your own custom symbol in Affinity Designer, Caroline Begbie has put together a superb and very speedy demonstration (https://www.raywenderlich.com/6485147-drawing-in-ios-with-swiftui/lessons/14).

You should also check out the videos introducing SF Symbols from the 2019 (https://developer.apple.com/videos/play/wwdc2019/206/) and 2020 (https://developer.apple.com/videos/play/wwdc2020/10207/) WWDC events, as well as the documentation on Apple's Human Interface Guidelines(https://developer.apple.com/design/human-interface-guidelines/sf-symbols/overview/) about SF Symbols.

You can learn all the details from Apple on creating your custom symbols (https://developer.apple.com/documentation/uikit/uiimage/creating_custom_symbol_images_for_your_app), and finally, don't forget to read the official page for SF Symbols(https://developer.apple.com/sf-symbols/).

We hope you enjoyed this tutorial, and if you have any questions or comments, please join the forum discussion below!

raywenderlich.com Weekly

The raywenderlich.com newsletter is the easiest way to stay up-to-date on everything you need to know as a mobile developer.

stevewozniak@apple.cc

Get a weekly digest of our tutorials and courses, and receive a free in-depth email course as a bonus!

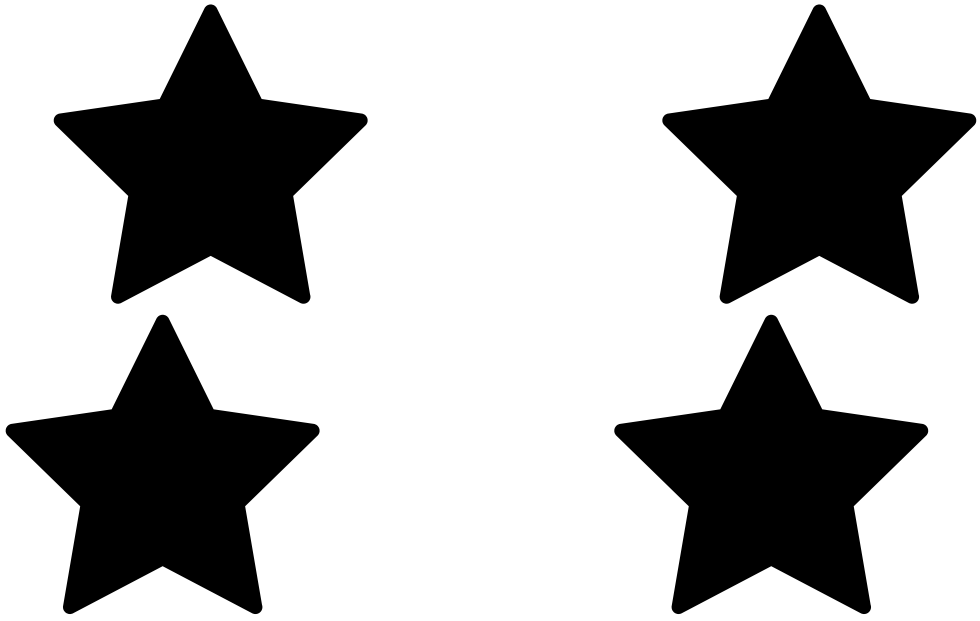
Reviews

We want to hear your thoughts!

Log into your account to leave a review for this Article.

All Reviews

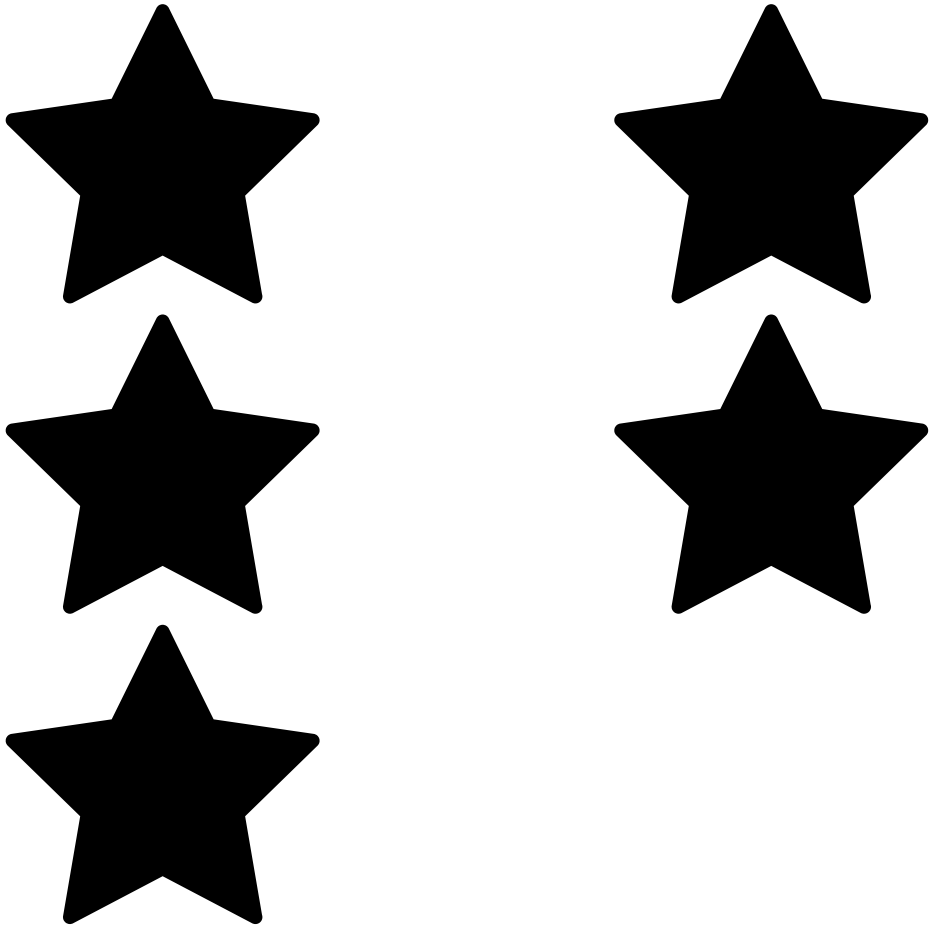
4.6



9 ratings · 1 review

carlpenn

February 11, 2022



There doesn't seem to be an Xcode project in the download materials.