
Design
SDK
DEVCON
SDK

RWDevCon 2018 Vault

By the raywenderlich.com Tutorial Team

Notice of Rights

Notice of Liability

Trademarks

Table of Contents: Overview

1: Living Style Guides 5

Living Style Guides: Demo 1 6

Living Style Guides: Demo 2 23

Living Style Guides: Demo 3 39

Table of Contents: Extended

1: Living Style Guides 5

Living Style Guides: Demo 1 6

1) Familiarize yourself with the sample app 6
2) Identify similar elements 8
3) Extract Colors to the Framework 9
4) Extract Fonts + Constrain Font Sizes 15
5) Creating A Label 19
6) That's it! 22

Living Style Guides: Demo 2 23

1) What changed? 23
2) The Style Guide App 24
3) Making Your Styles Viewable 28
4) Displaying Arbitrary StyleGuideViewable Types 31
5) Rendering Your Styles 33
6) Bringing It All Together With Labels and Buttons 35
7) That's it! 38

Living Style Guides: Demo 3 39

1) What's Changed? 39
2) Styling With Your Components 40
3) Changing Your Style Guide..Then Your App! 47
4) That's it! 53

1: Living Style Guides

In this session, you will learn how to make and manage a Living Style Guide for your application, which can show all of the building blocks for your application both in and out of context.

Building out a consistent user interface can be very difficult, but with a Living Style Guide, you always have a quick way to view (and review with designers) the building blocks of your application, the ability to build out new views quickly and consistently, and the power to make changes in one place which are reflected throughout your whole app.

1 Living Style Guides: Demo 1

By Ellen Shapiro

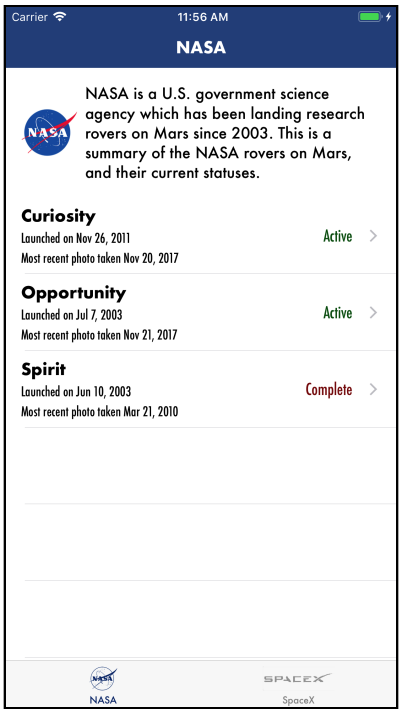
In this demo, you will learn how to extract UI logic from your Application into a UI framework.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Begin work with the starter project, **Spacetime**, in **Demo1\starter**.

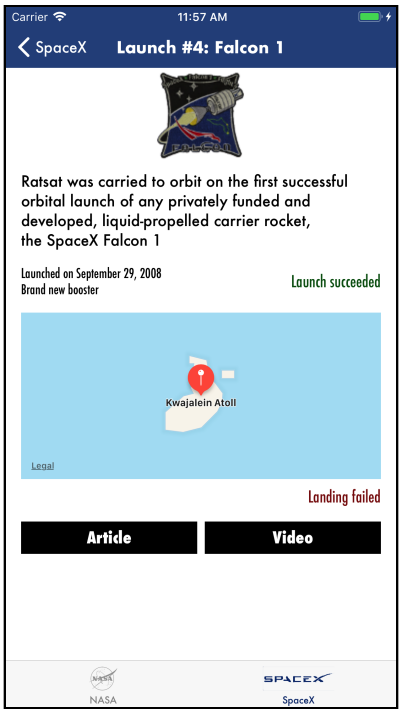
1) Familiarize yourself with the sample app

Open up `Spacetime.xcodeproj`, then build and run the application. You will see a two-tab interface which will initially display information about NASA rovers on Mars:



Tap the SpaceX tab, you will see information about SpaceX commercial launches.

Tap one of the rows, and you will see further details about that specific launch:



2) Identify similar elements

In all three places, you probably noticed a few things were the same, or at least very similar:

Colors

- The navigation bars and tab bars seem to be using a similar blue color
- Almost everything is using black as the text color.
- Anything positive has a dark green color, anything negative has a dark red color

Fonts

- Everything appears to be using some variation of the same font. (Spoiler: That font is **Futura**.)
- Titles on both sets of cells seem to be using the same level of boldness and size
- Regular text all seems to be around the same size
- Subtitles on both cells and on the detail view appear to be the same size and weight

Other

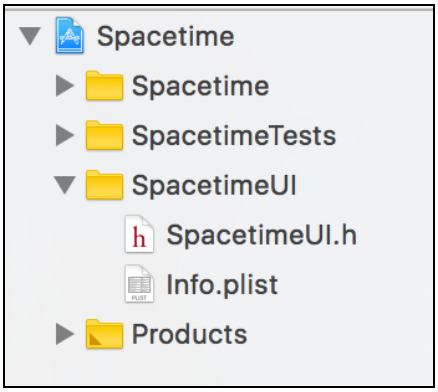
- Image views which load from the network appear to use the same placeholder image and loading mechanism

When you take the time to identify the things which are the same before you start writing code, the actual code where you pull the things together goes much faster!

3) Extract Colors to the Framework

To save some time, the framework where you're going to put the UI elements you're centralizing has already been created.

Toggle open the **SpacetimeUI** folder in the sidebar - you won't see a whole lot there:

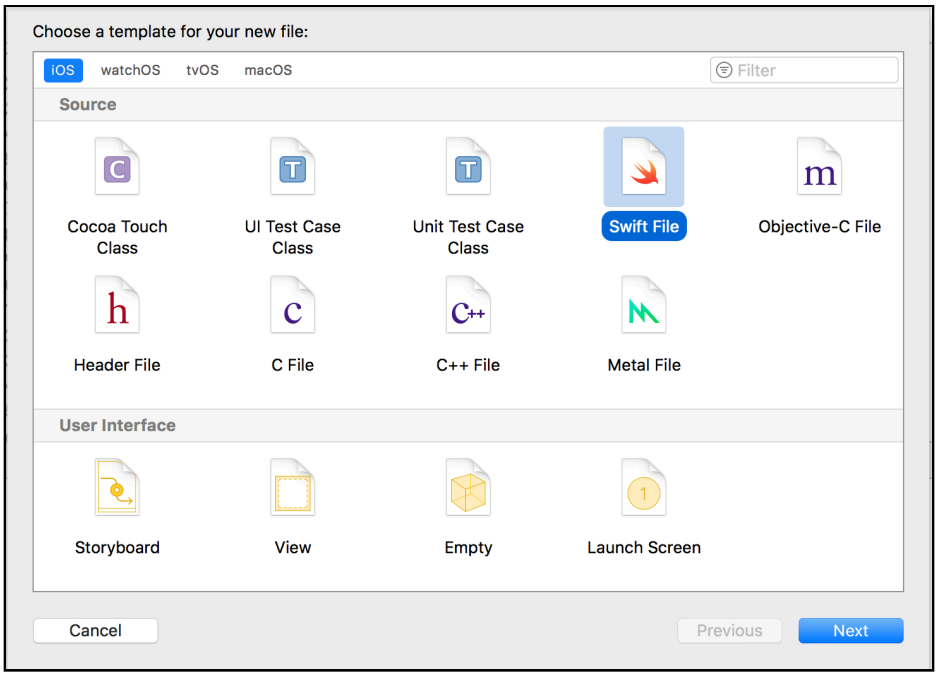


This is where you'll be adding files to the framework, so they can be shared between your app and your style guide.

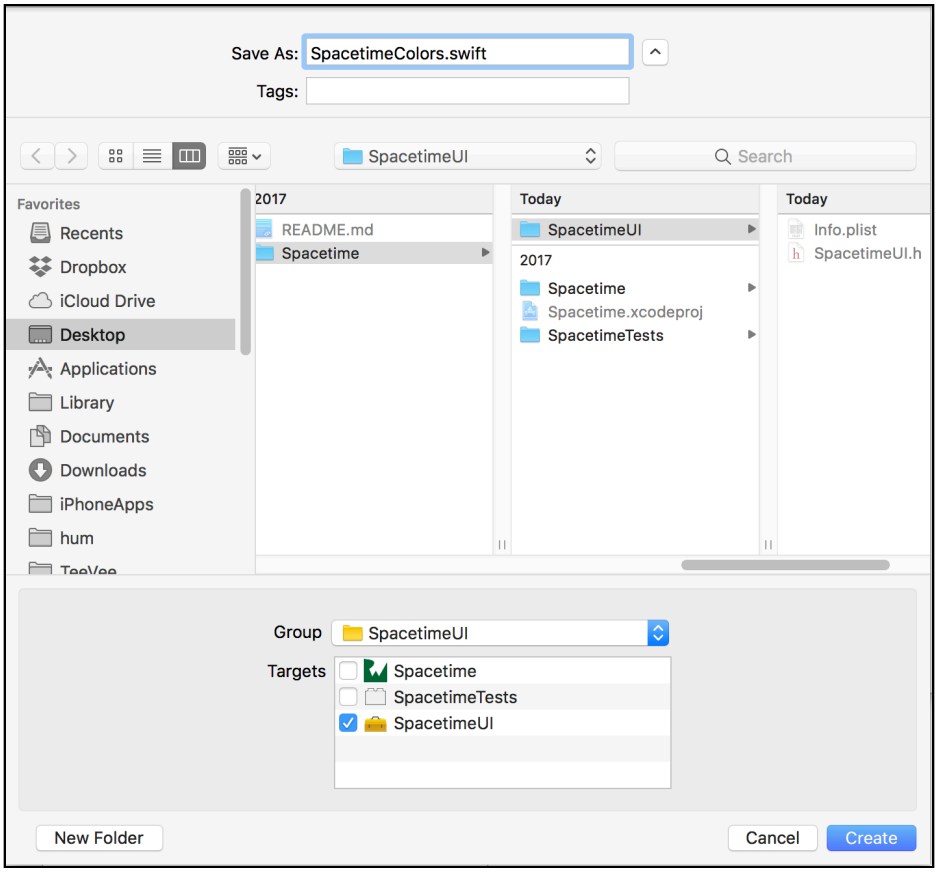
> **NOTE:** If you're interested in a step-by-step set of instructions on creating a framework from scratch, please watch out for the video of Mike Katz's talk, **Architecting Modules**, which is happening at the same time as this one.

You're going to start with the easiest thing to centralize: Colors. Colors are simple because there are only single values you have to extract for each one.

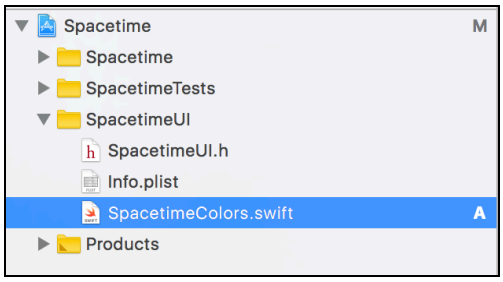
First, you'll want a place to put all the colors which you pull out. Make sure you've got the **SpacetimeUI** folder highlighted, so in the next step the file goes to the correct place. Go to **File\New\File...** and select an iOS **Swift File**:



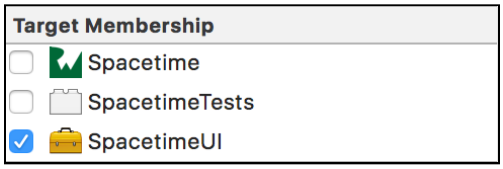
Name this file **SpacetimeColors.swift**, and click **Create**:



Your new file should now be added to the framework's folder:



And to the framework's target:



Select the file, and update the import since you're going to be using UIColors:

```
import UIKit
```

Next, since part of the idea behind using a Style Guide is that you want to be working with a defined set of known colors, go to the **SpacetimeColor.swift** file and create a **enum** to represent each color called `SpacetimeColor`:

```
enum SpacetimeColor {
}
```

Any piece of code in the framework you want to use in the main application will need to be marked `public` so it's visible to code which is importing your framework. Update the enum to be public:

```
public enum SpacetimeColor {
}
```

Now, you can use the work you did earlier to create a list of the colors you're going to need. Add the following inside the enum:

```
case
navigationBarBackground,
navigationBarContent,
tabBarContent,
success,
failure,
defaultText,
buttonBackground,
buttonText
```

A few pointers about these names:

- In this case since you don't have a designer you're working with, you'll use names which represent the colors' purpose rather than the actual underlying

color to make it easier to tell where they're used.

- If you **are** working with a designer, agree with the designer on a naming scheme for the colors, so that when they want to change a color, it's easier to figure out what exactly they want to change.
- Since `success` and `failure` can be used in multiple contexts, such as images, buttons, and text, the names of these colors are not differentiated by type.
- If something (like text) is the default value, it is helpful to prefix that case name with `default` to make it extra-obvious wherever it is used.

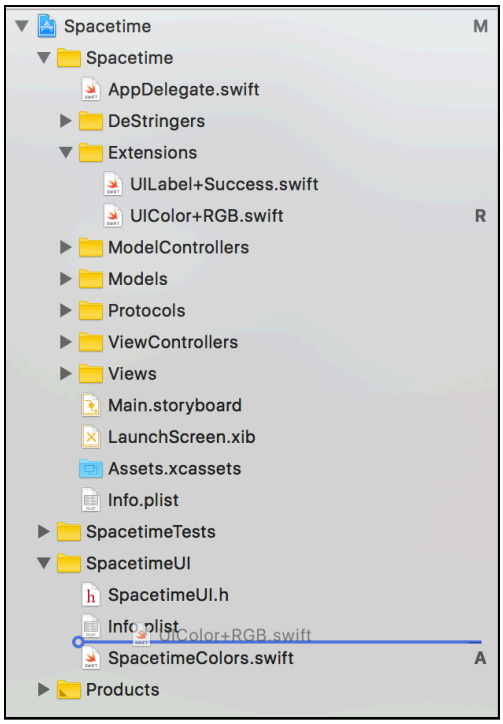
In this application, the vast majority of colors are set in code - these are the first things that you'll move over to the framework. To make them easy to calculate and independent of the enum itself, add a public computed variable to the `SpacetimeColor` enum which returns a `UIColor`:

```
public var color: UIColor {
}
```

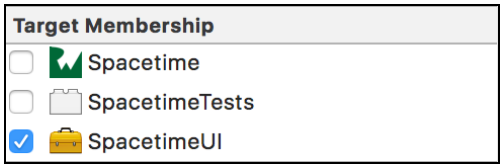
As part of this variable's definition, add a switch statement on `self`, so that it can eventually return the appropriate color per case. For now have it return a really out-of-place color by default so it compiles during the next stages:

```
switch self {
  // more code will go here
  default:
    return .orange
}
```

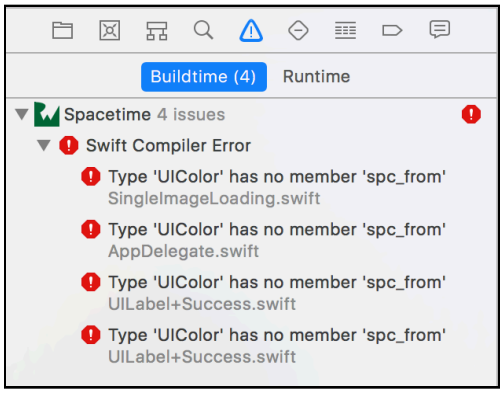
Next, move the **UIColor+RGB.swift** extension from the `Extensions` folder in the main application into the framework. To do this, just drag the file within Xcode into the **SpacetimeUI** folder:



When you do this with Xcode 9, it's finally smart enough to move the file in the underlying filesystem from the main app's folder to the framework's folder. It's also smart enough to update the target membership from the main app to the framework:



However, despite these niceties, Xcode doesn't fix everything for you. Build the app again - this should make a whole bunch of errors pop up in the main application:



Don't freak out - this is actually a really helpful way to find out what code needs to be moved to the framework!

Since the order the errors pop up in isn't always consistent, we'll move things file by file. First, click on the error which is appearing in **AppDelegate.swift** to be taken to the line of code causing the error.

```
  private func setupAppearanceProxies() {
    let blue = UIColor.spc_from(r: 9, g: 51, b: 119)
    UITabBar.appearance().tintColor = blue
    UITabBarItem.appearance().setTitleTextAttributes([
      .font: UIFont(name: "Futura", size: 10)!,
      ], for: .normal)

    UINavigationBar.appearance().barTintColor = blue
```

You can see from this code that the color is declared, then used twice: Once for the navigation bar's `barTintColor`, which is rendered as the navigation bar's background color, and once for the tab bar's `tintColor`, which is rendered as the color of the tab bar's contents.

Cut the `UIColor.spc_from` use out from the assignment to the variable `blue`, and return to **SpacetimeColor.swift**. Add it to the enum cases for the things it is related to:

```
case .navigationBarBackground,
       .tabBarContent:
  return UIColor.spc_from(r: 9, g: 51, b: 119)
```

> **NOTE:** You *can* add colors to an Asset Catalog in the framework's bundle instead of using the custom method shown here, but you have to make sure that they're loaded properly by using UIColor's `UIColor(named:in:compatibleWith)` constructor, and passing in the framework's bundle. This is a bit time-consuming to set up, so we're skipping it here.

Now, go back to **AppDelegate.swift**. At the top of the file, import the framework where the color has been moved:

```
import SpacetimeUI
```

Go back to the `setupAppearanceProxies` method. You can now delete the entire line assigning the `blue` variable. Update the beginning of the code to use the two colors you just set up:

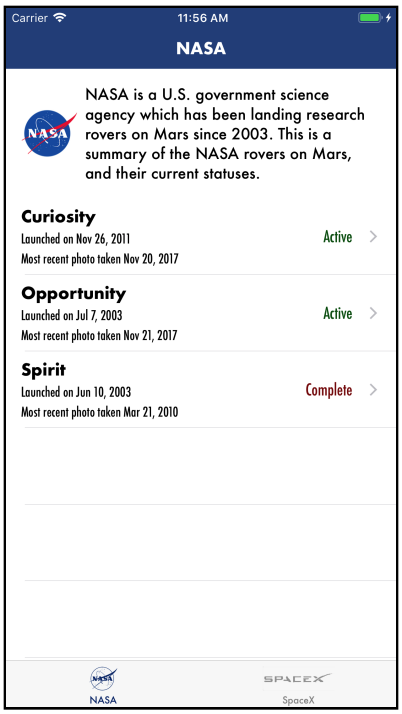
```
UITabBar.appearance().tintColor = SpacetimeColor.tabBarContent.color
UITabBarItem.appearance().setTitleTextAttributes([
  .font: UIFont.spc_standard(size: 10),
], for: .normal)

UINavigationBar.appearance().barTintColor =
    SpacetimeColor.navigationBarBackground.color
```

For now, we'll stop moving colors over there since this gets a little repetitive, and we only have so much time. But you can see how you can use compiler errors to determine some of the places where you need to update your code.

For now, to get the app compiling again go to **UIColor+RGB.swift**, and make the `spc_from(r:g:b:a:)` method public:

```
public static func spc_from...
```

Now, all the errors should be fixed! Yay! Build and run again and...



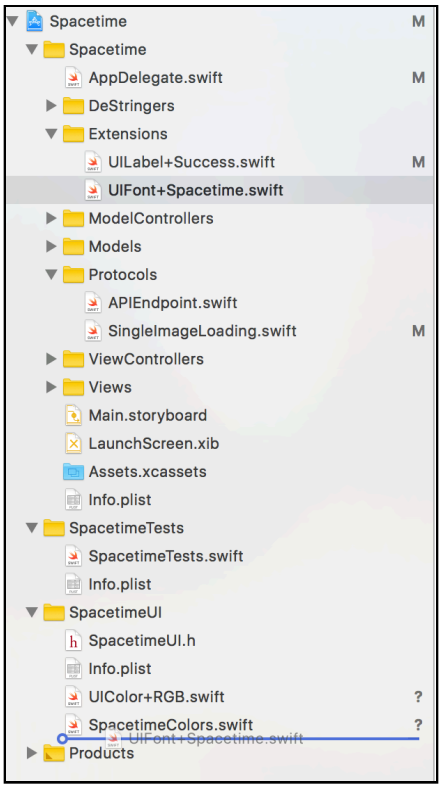
It looks the same! Hooray! You've now got the navigation bar's background and the tab bar's content colors in the code moved over to the framework, and nothing's changed.

Colors are simple, because they're only one piece: The color. Now it's time to look at something with two pieces: Fonts.

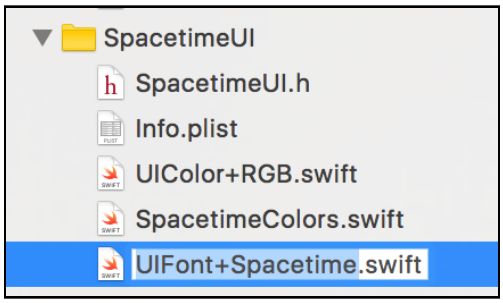
4) Extract Fonts + Constrain Font Sizes

In this case, a bit of work has already been done for you in the **UIFont+Spacetime.swift** file in the **Extensions** folder. It contains convenience methods for the types of font used in the application, which have the hard-coded strings needed to load the fonts.

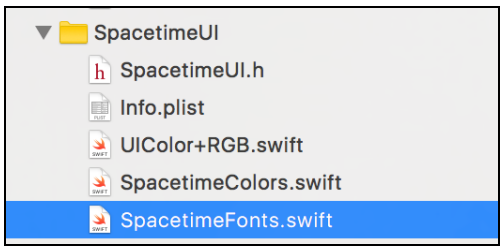
Drag this file down into the `SpacetimeUI` framework's folder:



Click, wait about half a second, then click again on the file name to bring it into renaming mode:



Rename it **SpacetimeFonts.swift**:



Build the project, and you'll notice that everything still builds: That's because all the methods in this file are already public!

Now one thing you may have noticed when looking at the font and the `AppDelegate`

is that the current setup requires you to hard-code sizes, which can make it really easy for the appearance of your app to become inconsistent.

You can agree with your designer (or yourself) on some consistent sizes and names for those sizes, and then they're easier to change at the drop of a hat.

Underneath the `UIFont` extension, add an enum to hold the font sizes that you want to use today:

```
public enum SpacetimeFontSize {
  case
  tiny,
  medium,
  normal
}
```

Then within that enum, add a computed property to return the font sizes you want to use:

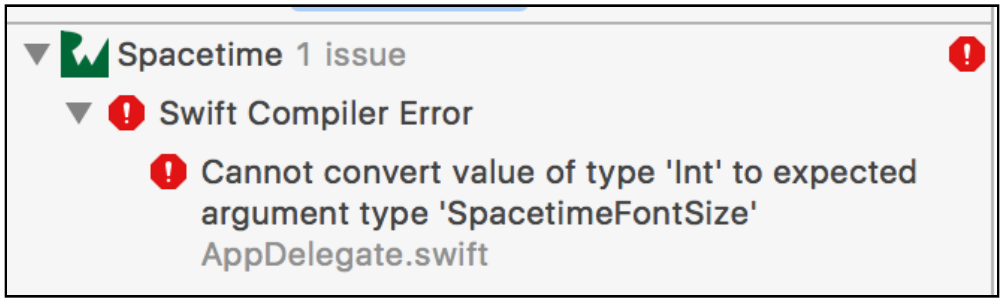
```
var value: CGFloat {
  switch self {
  case .tiny:
    return 10
  case .medium:
    return 16
  case .normal:
    return 17
  }
}
```

You'll note that this method is NOT public - that's because only the internal bits of this framework really care about the actual value.

Next, update the `size` parameter on `spc_bold` in the `UIFont` extension to take a SpacetimeFontSize, and use `size.value` instead of just `size`:

```
public static func spc_bold(size: SpacetimeFontSize) -> UIFont {
  return UIFont(name: "Futura-Bold", size: size.value)!
}
```

Build again, and you'll see that a new error has popped up in **AppDelegate.swift**:



Now that's what I call discouraging hard-coding! Update the hard-coded font size at

the point of the error to use a value from your new enum instead:

```
UIFont.spc_bold(size: .normal)
```

Build again, and your error will go away. Huzzah!

But you can do even better in enforcing style. Right now, you've constrained the size of the font - but what if you also want to constrain the type of font?

To do this, go to **SpacetimeFonts.swift** and at the top of the file, create a public `enum` with cases for the four types of fonts which are currently handled in the `UIFont` extension:

```
public enum SpacetimeFont {
  case
  standard,
  condensed,
  bold,
  condensedBold
}
```

Next, add a function to the enum to create the underlying UIFont when given a particular size. Again, use a really ridiculous font for the default value so you can easily see when you've done something wrong:

```
public func of(size: SpacetimeFontSize) -> UIFont {
  switch self {
  default:
     return UIFont(name: "Chalkduster", size: size.value)!
  }
}
```

Next, add a single case for `.bold`, and cut and paste the matching code from the `UIFont` extension into its case:

```
case .bold:
  return UIFont(name: "Futura-Bold", size: size.value)!
```

Finally, delete the `UIFont` extension method `spc_bold`, and build again. Everything which was using that extension method needs to be updated to use `SpacetimeFont`. In **AppDelegate.swift**, update the erroring line to use:

```
SpacetimeFont.bold.of(size: .normal)
```

Again, you'd want to replace all of these cases in the end, but in an in-person tutorial, we don't have a ton of time. So we're going to move on, since now that you have both colors and fonts in the framework, it's time to combine them into one single piece of UI: A label!

5) Creating A Label

Go to **File\New\File..** and create a new Swift file. Name it **SpacetimeLabels.swift**, and make sure it's going into the **SpacetimeUI** folder and framework:

Update the `import` statement since you'll be working with UIKit elements here:

```
import UIKit
```

Next, add an enum for label styles. You're only going to work with a single style right now - the style for cell title labels - but it's good to set it up as an enum to make adding more styles really easy:

```
enum SpacetimeLabelStyle {
  case
  cellTitle
}
```

Labels have two basic properties: the font, and the color of the text. First, add a property for the simpler one, the color of the text. Normally, you'd use `switch self` to set up return values for every case, but to save time here, we'll just return the value for the single case we just added:

```
var textColor: SpacetimeColor {
    return .defaultText
}
```

Next, add private properties for the font type and size, a non-private property to combine those two properties into a `UIFont`:

```
private var fontType: SpacetimeFont {
    return .bold
}

private var fontSize: SpacetimeFontSize {
    return .normal
}

var font: UIFont {
  return self.fontType.of(size: self.fontSize)
}
```

Note that for the font, you don't actually have to use an enum, since enums are handling both `fontType` and `fontSize` for you!

Now, in order to actually use all these things, you'll need to set up an actual instance of `UILabel`. Since you know there will be several different types, create a base class that configures the label when a label style is set.

Add the following to **SpacetimeLabels.swift** below the end of the

`SpacetimeLabelStyle` enum:

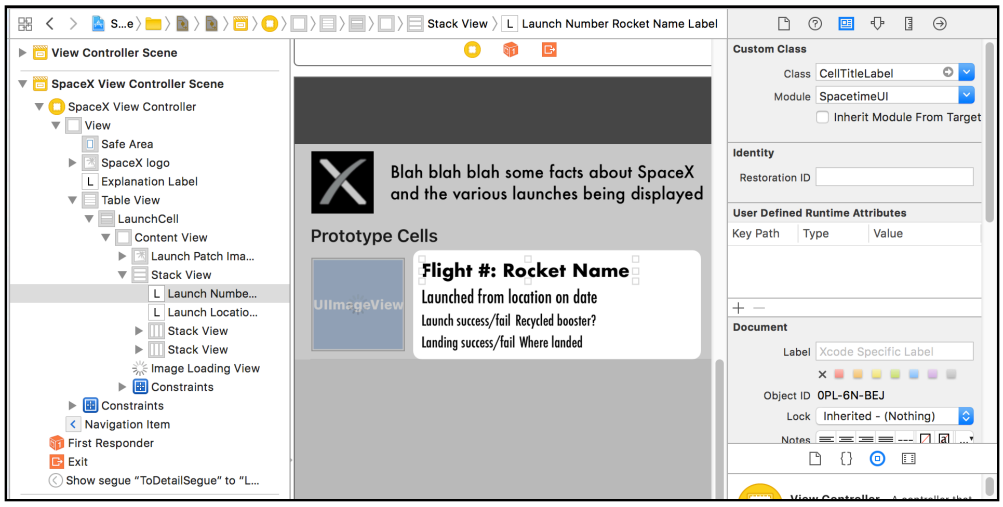
```
public class SpacetimeBaseLabel: UILabel {

  var labelStyle: SpacetimeLabelStyle! {
    didSet {
      self.font = labelStyle.font
      self.textColor = labelStyle.textColor.color
    }
  }
}
```

To save a little time, I'll note that the particular type of labels we're initially dealing with are all set up in Storyboards, so when their view controllers are loaded, `init(coder:)` will be called.

Below your declaration of **SpacetimeBaseLabel**, create a subclass of it called **CellTitleLabel** for cell titles that sets up the label style in the initializer:

```
public class CellTitleLabel: SpacetimeBaseLabel {

  public required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
    self.labelStyle = .cellTitle
  }
}
```

Open **Main.storyboard**, and in both **View Controller Scene** and **SpaceX View Controller Scene**, select the title label for the prototype cells, and update its type to **CellTitleLabel**, and make sure that the module is **SpacetimeUI**:



Note that you won't see the labels update in interface builder yet - you'll learn how to make that work in Demo 3.

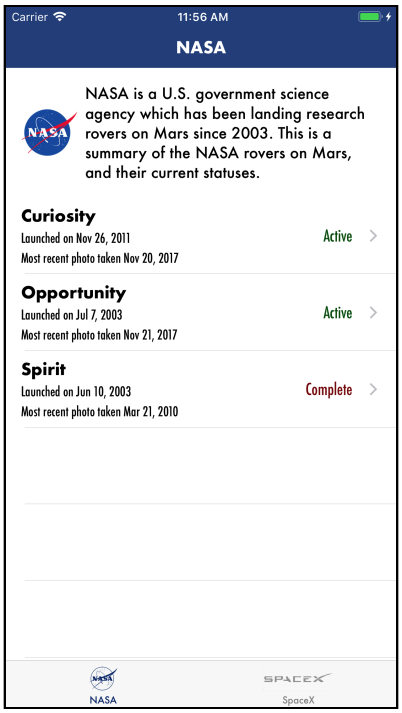
Build and run the application and...

Orange? Whoops! The good news is, the labels are definitely now being pulled from the framework, since they're showing the color you set up used as a default color. Time to fix that.

Go to **SpacetimeColors.swift** and add a case to the `color` enum which returns the proper default text color:

```
  case .defaultText:
     return .darkText
```

Build and run again, and finally, things are back to where they need to be:



6) That's it!

Congrats, at this time you should have a good understanding of extracting the building blocks of your UI into a framework, and beginning to combine them. It's time to move on to making them play nice with Interface builder, and creating the Style Guide App.

2 Living Style Guides: Demo 2

By Ellen Shapiro

In this demo, you will make an application which serves as your living style guide.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Make sure you begin work with the starter project in **Demo2\starter**.

1) What changed?

The **Spacetime** project you're starting with for this demo has been updated from the one at the end of Demo 1 in a few ways. These are things that normally you'd need to do yourself, but for the sake of time, we're going to skip over.

Here's what's changed:

- All colors have been added.
- All label and button styles have been defined.
- Custom label and button subclasses have been added for all styles.
- Label and button subclasses have been made `IBDesignable`, and updated so all initializers and `prepareForInterfaceBuilder` will cause the appropriate styling to be applied.
- All UI elements have been updated in the Storyboard to use the appropriate subclass from the framework instead of being designed directly in the storyboard.
- A few convenience extensions have been added.

Again, in the real world, you'd probably be doing most of these yourself rather than

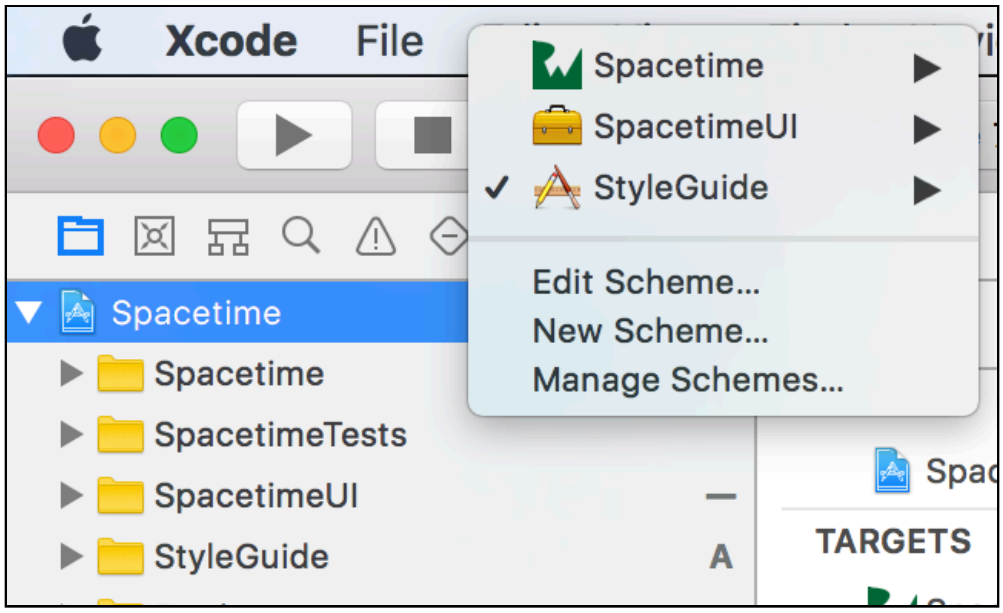
having Magical Ray Wenderlich Gnomes doing this for you. However, since the gnomes have been so kind as to help us out, it's time to move on to...

2) The Style Guide App

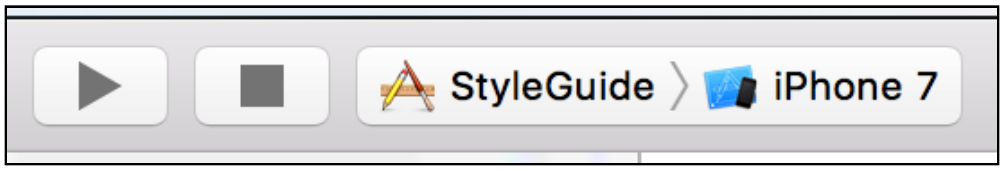
This is the application where your Living Style Guide will...well, live.

> **Note:** We've skipped a few steps here to save time, but if you want to create an application yourself: Add a new Target to the `.xcodeproj`, select a Single-View iOS App, and name it **StyleGuide**.

Your new lives in the same `.xcodeproj` as the rest of your code. You'll be able to see the files and a new build scheme, which was automatically created by Xcode:

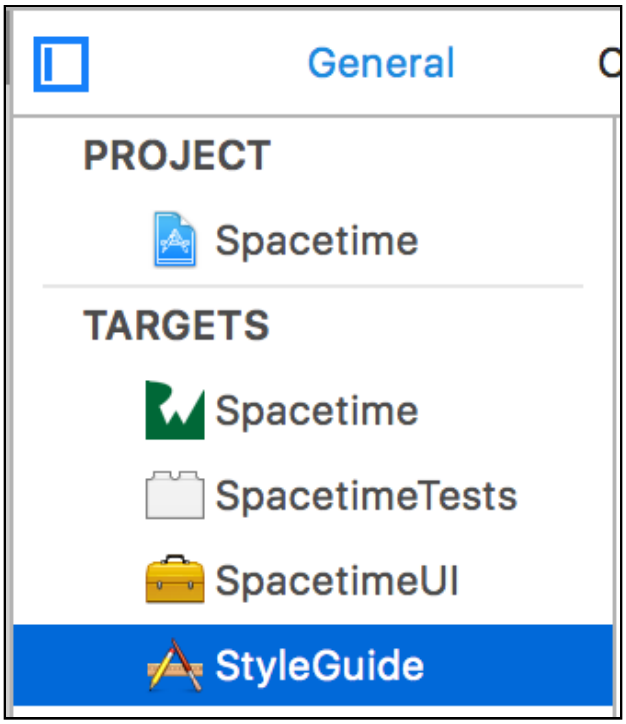


Make sure the build scheme for **StyleGuide** is selected so that for the next few steps when you build, the Style Guide app will be what builds:

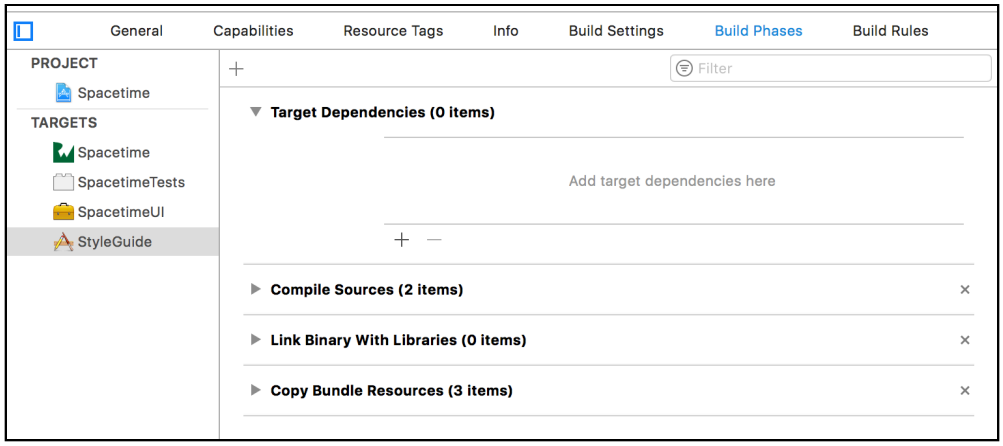


First, you want to make sure that when you build your **StyleGuide** app, any changes which have occurred in the **SpacetimeUI** library get applied to that build.

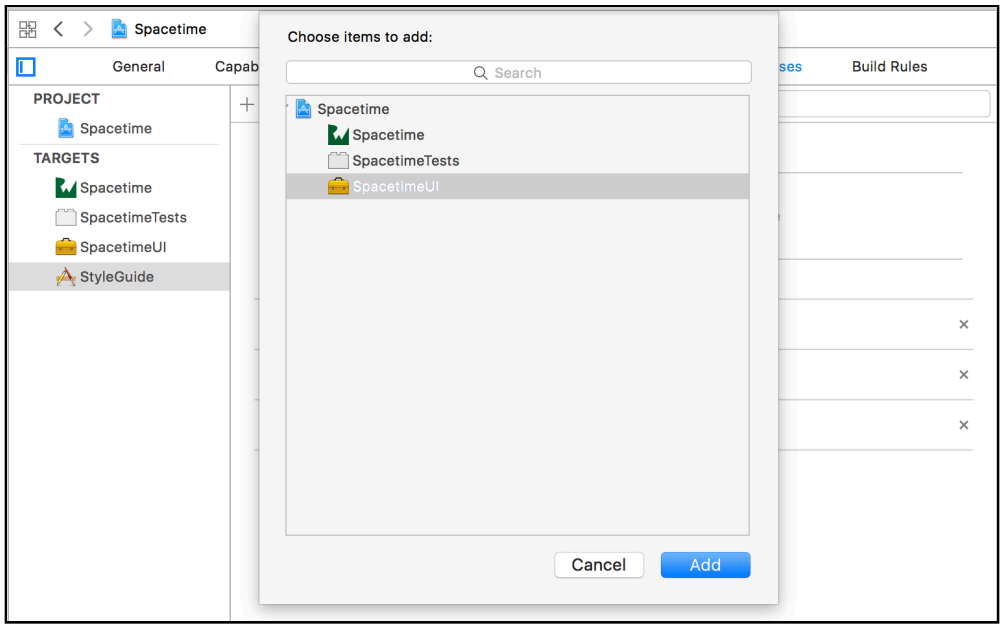
To do that, select the **StyleGuide** app in the list of targets:



Then select **Build Phases** and flip open the **Target Dependencies** section:

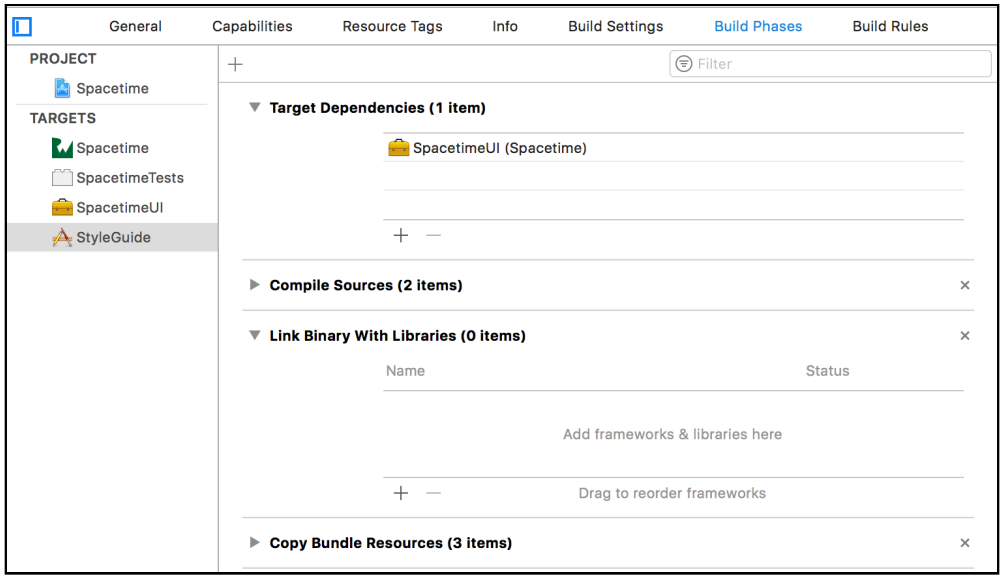


Click the plus button to add a target dependency, and select the **SpacetimeUI** framework as the dependency you want to add:

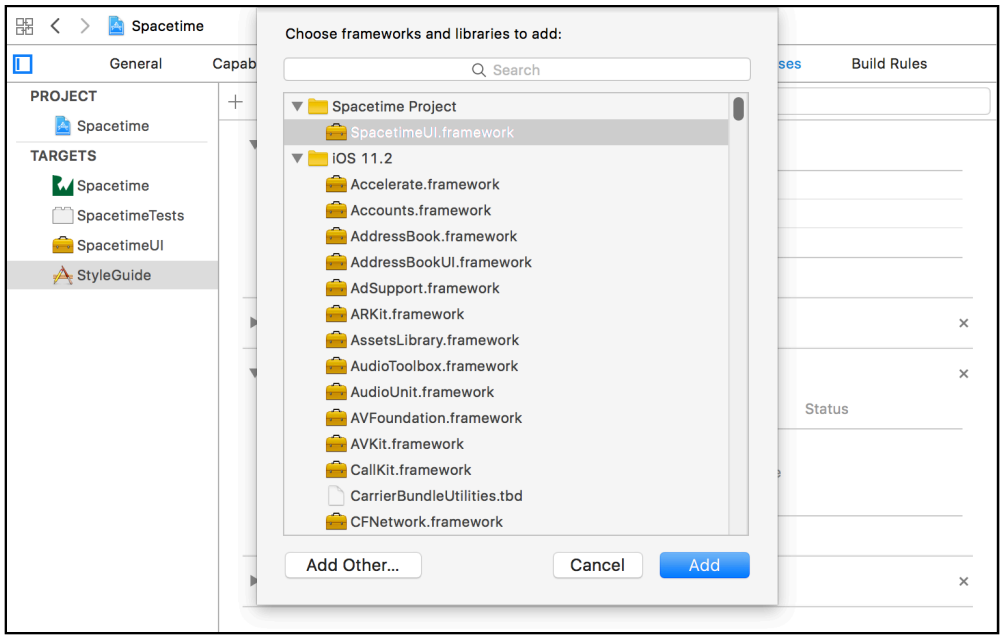


Now, anytime the **StyleGuide** app is built, its dependency **SpacetimeUI** will also get built!

Next, you need to make sure the framework is actually included with the binary of your style guide app. Flip open the **Link Binary With Libraries** section:



And then again, select **Spacetime UI** as the library you want to link to:



Next, it's time to do a quick experiment to make sure that the style guide app can actually see the framework properly.

In the **StyleGuide** app's folder, open **ViewController.swift**, a pretty standard, empty, `UITableViewController` and import the framework:

```
import SpacetimeUI
```

Make the background color of this view controller temporarily the background color of the navigation bar by adding the following to the end of `viewDidLoad()`:

```
self.view.backgroundColor = SpacetimeColor.navigationBarBackground.color
```

Build and run and:

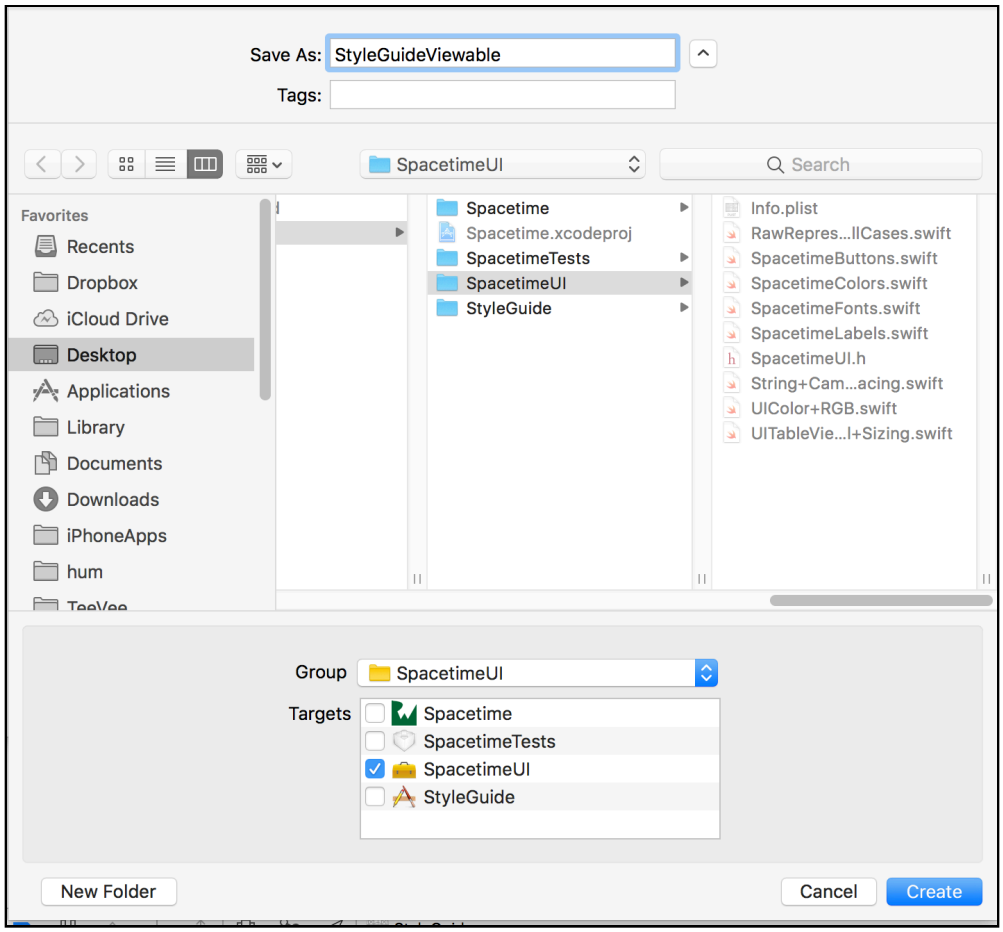
Woot! That nice dark blue confirms that the Style Guide app can see what's in the UI framework. Now it's time to make your style guide live!

3) Making Your Styles Viewable

In order to view the pieces of your style guide, you're going to want some kind of a way to set them up and display them.

Here, you'll use the power of protocols to take the enums you already made and make them easy to view.

In the **SpacetimeUI** framework, go to **File\New\File...** and create a new empty Swift file called **StyleGuideViewable.swift**. Make sure it's being added to the framework target:



Then click **Create**. Open up the file you just created, and within it, define the initial protocol:

```
public protocol StyleGuideViewable {
  // 1
  static var styleName: String { get }

  // 2
  var itemName: String { get }

  // 3
  var rawValue: String { get }
}
```

The things you're defining here:

1. Provide a display name for the style itself.
2. Provide a display name for each item in a given style.
3. This is a quick hint of a shortcut you're going to take: Turning all your style enums with no raw value into enums with a raw value of `String`. Once we do that, this requirement will automatically be fulfilled.

Next, add a default protocol implementation extension to grab the class name for the style name and the `rawValue` for an item name, and use an extension on `String` to make it a bit easier to read:

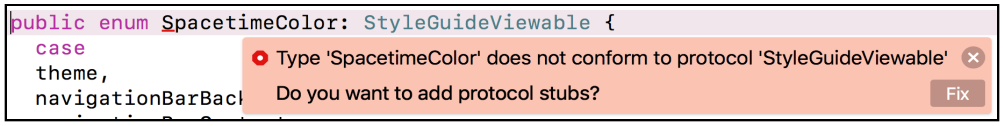
```
extension StyleGuideViewable {
  public static var styleName: String {
    return "\(Self.self)".camelCaseToSpacing()
  }

  public var itemName: String {
    return self.rawValue.camelCaseToSpacing()
  }
}
```

Now, anything which conforms to **StyleGuideViewable** will be able to display the style name and the item name in a nice, readable fashion.

Next, go to **SpacetimeColors.swift**. Tell the `SpacetimeColor` enum it's going to conform to `StyleGuideViewable`:

```
public enum SpacetimeColor: StyleGuideViewable {
```

Make sure the build scheme for **SpacetimeUI** is selected and do a quick build - you'll still see an error (Note: You may have to do a Clean to get it to show up):



Do not reach for that fix button! It's complaining that there's no `rawValue` of type `String` on the `SpacetimeColor` enum. But what's a really easy way to add that without actually having to add that property to the enum?

That's correct, all you have to do is update the `SpacetimeColor` enum to have a `rawValue` of type `String`. Do so in the enum's declaration:

```
public enum SpacetimeColor: String, StyleGuideViewable {
```

Rebuild, and the error should now be gone. Next, in **SpacetimeFont.swift** add the same enum type and conformance extensions to **SpacetimeFont**:

```
public enum SpacetimeFont: String, StyleGuideViewable {
```

and **SpacetimeFontSize**: (also in **SpacetimeFonts.swift**)

```
public enum SpacetimeFontSize: String, StyleGuideViewable  {
```

Next, you want to make something which can display all these fancy new **StyleGuideViewable** types.

4) Displaying Arbitrary StyleGuideViewable Types

To save us all a bunch of time and super-boring stock table view code code, the Code Gnomes left us a commented-out gift in the **StyleGuide** app: The **StyleGuideViewableViewController.swift** file.

To uncomment the whole file, hit ⌘-A to select all of its contents, and then ⌘-/ to uncomment at a single level. This view controller takes an array of `StyleGuideViewable` objects, and then displays them in a `UITableView`.

The only major difference from a stock table view setup is that this table view is showing a single cell per section - each section represents a single instance of a `StyleGuideViewable` object. In this case, a single case of an enum.

This is so you can easily use the section header which is automatically generated by the table view to show a label that indicates what style each view is showing.

Next, you're going to replace the contents of **ViewController.swift** with something a bit more useful. However, it's also quite boring to type, so you're going to do a little copy-pasta to save some time.

Back in Finder, you'll see a folder under **Demo2\starter** called **Shortcuts**. Open this folder, then open the **ViewController.swift** file within it. Select all of its contents by hitting ⌘-A, then cut them using ⌘-X.

Go back to Xcode, and look in the **StyleGuide** app for **ViewController.swift**. Use ⌘-A again to select all, then use ⌘-V to paste what you just cut out of the **Shortcuts** folder.

This updated view controller is another `UITableViewController`, but a more traditional one. The biggest thing is that it takes an array of arrays of the styles you'll be showing.

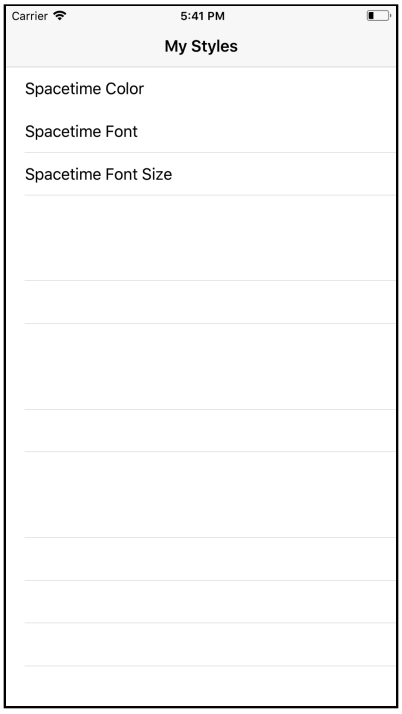
Each array of styles should contain all the cases of one enum conforming to `StyleGuideViewable`, so having an array of these arrays allows you to display and select which style you want to look at in the **StyleGuideViewableViewController**.

In **ViewController.swift**, update the `styles` array to pass in arrays of all the cases of each enum. Luckily for you, there's an `allCases` static extension method on enums conforming to `RawRepresentable`, so you only have to add the following to get access to these arrays:

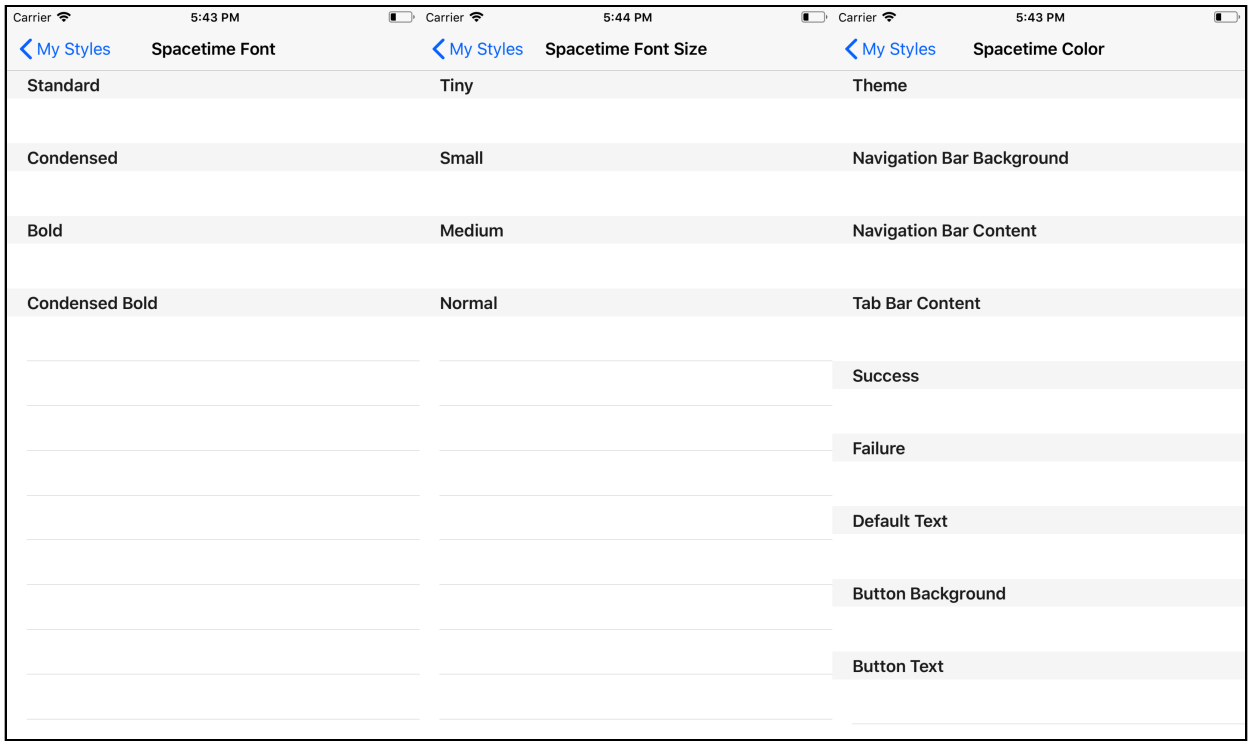
```
private lazy var styles: [[StyleGuideViewable]] = {
  return [
    SpacetimeColor.allCases,
    SpacetimeFont.allCases,
    SpacetimeFontSize.allCases,
  ]
```

```
}()
```

Build and run the **StyleGuide** app, and you'll see a list of all the styles you just created:



Tap one of the styles to go in, and you'll see the names of each style in section headers, but not much else:



Now it's time to give those names some meaning by adding a way to make each **StyleGuideViewable** item display some meaningful information.

5) Rendering Your Styles

To do that, you're going to add a computed property which will provide a view which can be displayed to the user. Go back to **StyleGuideViewable.swift** and define a `view` computed property:

```
var view: UIView { get }
```

With that addition, anything which conforms to `StyleGuideViewable` has to provide some sort of sample view indicating what it is or does.

Open up **SpacetimeColors.swift** add an implementation of this property which returns a view with the background color set to the current case's `color` property:

```
public var view: UIView {
  let view = UIView()
  view.backgroundColor = self.color
  return view
}
```

Next, open **SpacetimeFonts.swift** and for the `view` property, return a label which displays each font at a consistent size:

```
public var view: UIView {
  let label = UILabel()
  label.font = self.of(size: .normal)
  label.text = label.font.fontName
  return label
}
```

Next, scroll down to **SpacetimeFontSize**. Continuing to type up everything you'd want to happen for these extensions from here out would be extremely time-consuming and boring.

We also want to be a bit more proper about separating where code is part of a requirement for a protocol vs where it's part of an initial declaration. Start by deleting the `StyleGuideViewable` conformance declaration on the `SpacetimeFontSize` enum:

```
public enum SpacetimeFontSize: String {
```

To help you out, open up the **Demo2\Shortcuts** folder again, and then open the **VariousExtensions.swift** file, that has implementations for several extensions you're going to be creating.

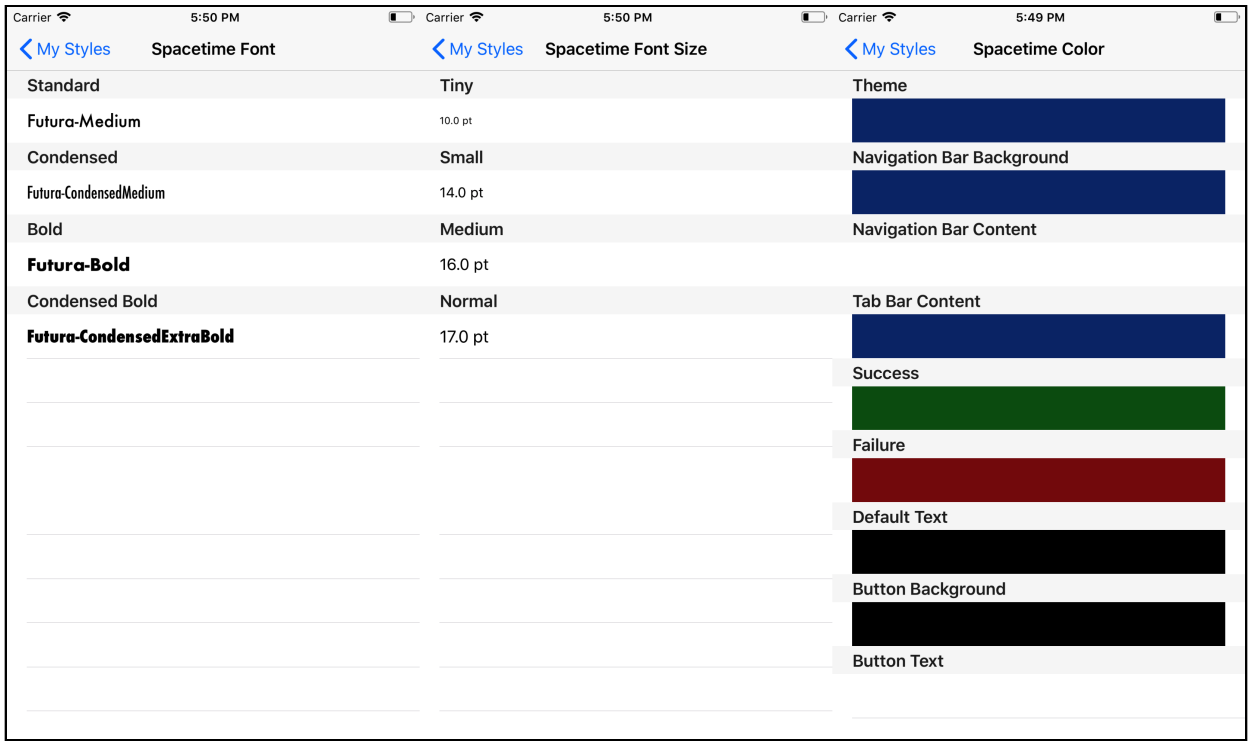
At the top, you'll see an extension for **SpacetimeFontSize** that implements **StyleGuideViewable**. Cut this first extension out of this file, and paste it into **SpacetimeFonts.swift**, right below the end of the font size enum.

Similarly to the **SpacetimeFont**, this extension will return a label which uses a consistent font, but which adjusts the size at which it is displayed.

Finally, open **StyleGuideViewableViewController.swift**, and replace the comment in `tableView(_:cellForRowAt:)` with some code to put the style's `view` into the cell:

```
cell.replaceContentViewSubviewsWith(style.view)
```

Build and run the **StyleGuide** app again. Now, when you tap into the styles you're displaying, you'll actually see some useful information at a glance:



Hooray! Now that you have the ability to see styles with one property, it's time to bring things together in the more composable Label and Button styles.

6) Bringing It All Together With Labels and Buttons

Open up **SpacetimeLabels.swift**. Make **SpacetimeLabelStyle** a `String` enum:

```
public enum SpacetimeLabelStyle: String {
```

Back in the **VariousExtensions.swift** file, which you still should have open, you should now see an extension for **SpacetimeLabelStyle** at the top. Cut this extension out of this file, and paste it into **SpacetimeLabels**, right below the end of the enum.

This sets up the the **SpacetimeLabelStyle** enum's `view` property to return a label showing information about what font, font size, and color are being used in the label.

Next, go to **SpacetimeButtons.swift** and make **SpacetimeButtonStyle** a String enum:

```
public enum SpacetimeButtonStyle: String {
```

Once again, you want to go to **VariousExtensions.swift**, and cut out the single remaining extension on **SpacetimeButtonStyle**, and paste it into **SpacetimeButtons.swift** right below the end of the button style enum.

With buttons, a nice thing to be able to see is all the different states the button can display at once. Fortunately, while you're returning a `view` as part of the `StyleGuideViewable` protocol, nobody ever said it couldn't have subviews! :]

This extension returns a Stack View with copies of each style of button, each set up to display a state - normal, highlighted, or selected.

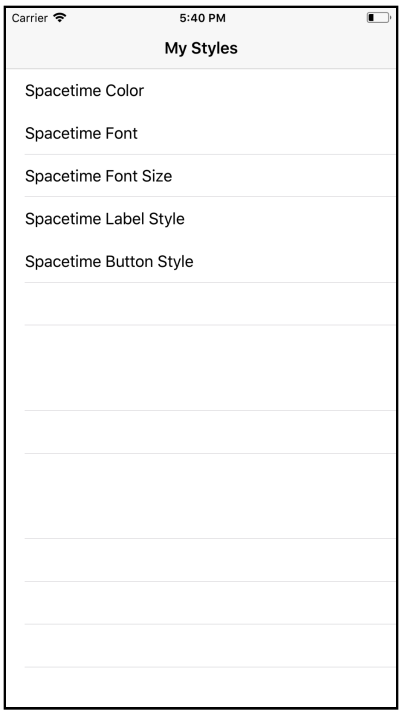
How is all that set up? Keep on scrolling down within **SpacetimeButtons.swift** to **SpacetimeBaseButton**.

Here, you'll see that a `buttonStyle` property does the hard work of determining which elements of the style need to be applied to the button, particularly including what title color to set up for each supported button state.

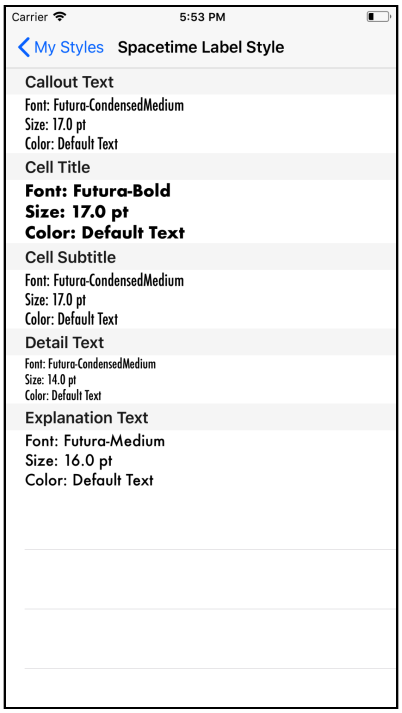
Finally, go back into the **StyleGuide** app's **ViewController**, and add the labels and buttons to the lazy `styles` array:

```
SpacetimeLabelStyle.allCases,
SpacetimeButtonStyle.allCases,
```

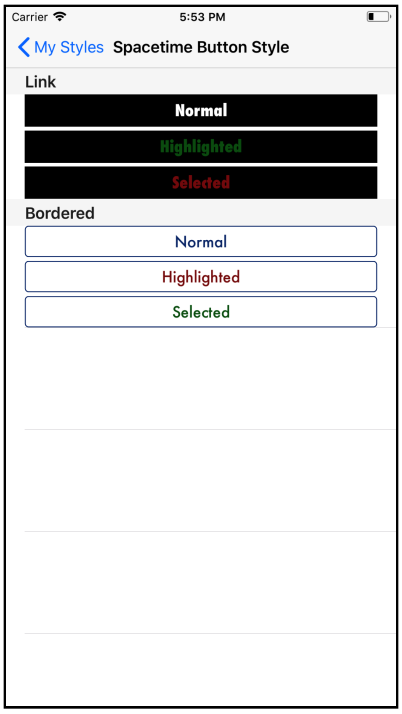
Build and run again, and you'll now see your two added sets of styles for buttons and labels:



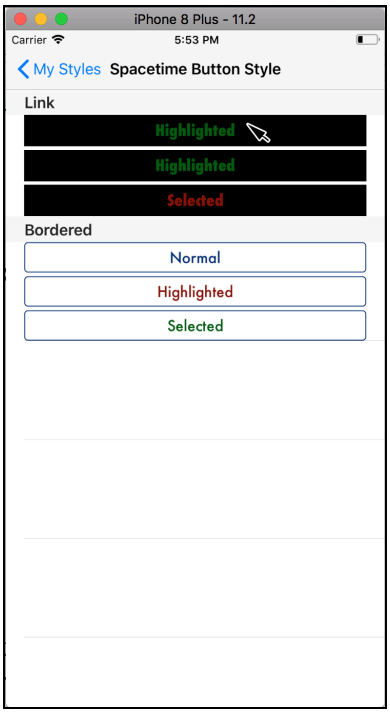
Tap into the labels, and you'll be able to easily see details about the labels in the exact font and color in which they will display:



Tap into the buttons, and you'll be able to see the button styles for each of the states you're supporting:



And when you tap on any of the buttons, the title and state will update based on the tap state:



7) That's it!

Congratulations - you've now built a living style guide application! Take a break, and then it's time to move on to using what you've built to add and adjust content in your main application quickly and easily.

3 Living Style Guides: Demo 3

By Ellen Shapiro

In this demo, you will use the UI framework and style guide you've built in the first two sessions in order to add new content to your app, and then test out adjusting styles.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

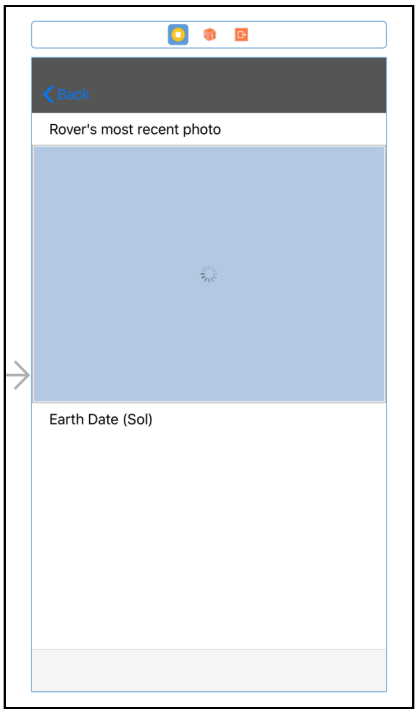
> **Note**: Begin work with the starter project in **Demo3\starter**.

1) What's Changed?

The magical Wenderlich Code Fairies have been at it again, and have made a few updates to the code to save us some time and typing in this demo:

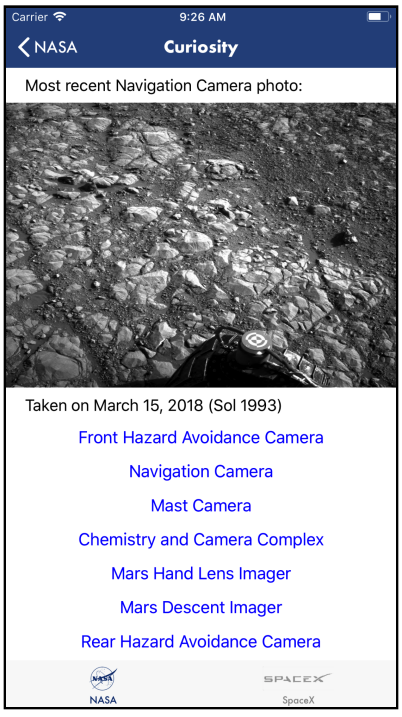
- A new **PhotoViewController** class has been added to the main application.
- This class is set up to load the most recent photo of a passed-in rover's selected camera. It automatically starts with the NAVCAM camera, since that exists on all three rovers, but you can select one of the other cameras with one of the buttons in a stack view.
- `@IBOutlets` and other UI elements are set up using basic `UIKit` classes, and are hooked up with the storyboard.
- A segue to go from selecting a rover to this screen has been added both in the storyboard and set up to be supported in code in **NASAViewController.swift** via a `prepareForSegue()` method.

Open up **Main.storyboard**, and to the right of the **NASAViewController**, you'll now see this view:



2) Styling With Your Components

Build and run the application, and select one of the Rovers (try Curiosity as it's the most recently launched rover). You'll see a pretty boring looking view - except for the picture of Mars:

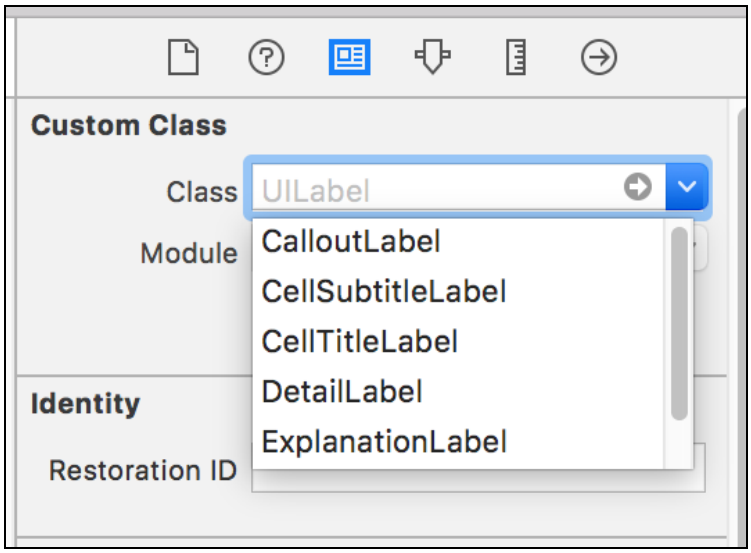


> **NOTE**: Your picture will probably be different if you're loading from live data, since active rovers are still taking new photos.

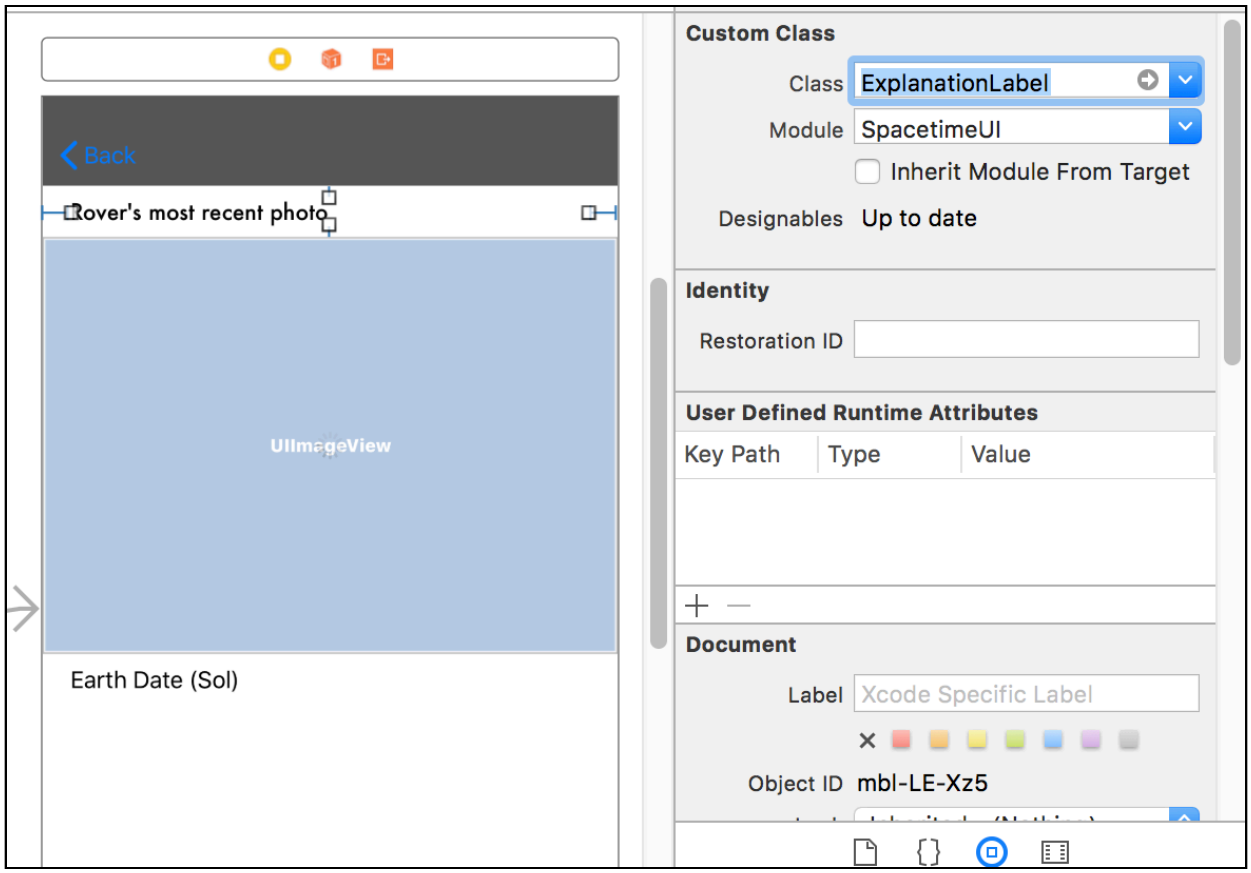
This works nicely, but it's not taking advantage of any of your custom classes. Time to fix that!

Go back to **Main.storyboard**, and select the `PhotoViewController`'s `description` label. Go to the right sidebar and select the identity inspector tab so you can change its class.

In the Custom Class section, click on the drop-down indicator next to the Class field. You'll see a list of all the various types of UILabel subclass which are available from the `SpacetimeUI` framework automatically:



Select the "Explanation Label", and you'll see the Storyboard update immediately to reflect the subclass's style:



What is this wizardry?! Let's take a look under the hood.

Go to **SpacetimeLabels.swift** in the **SpacetimeUI** framework. You'll notice that the base class for all labels, `SpacetimeBaseLabel` declares that it is `@IBDesignable`.

This tells Interface Builder that this class or any of its subclasses may provide custom rendering code so it can draw the view closer to what will appear on screen.

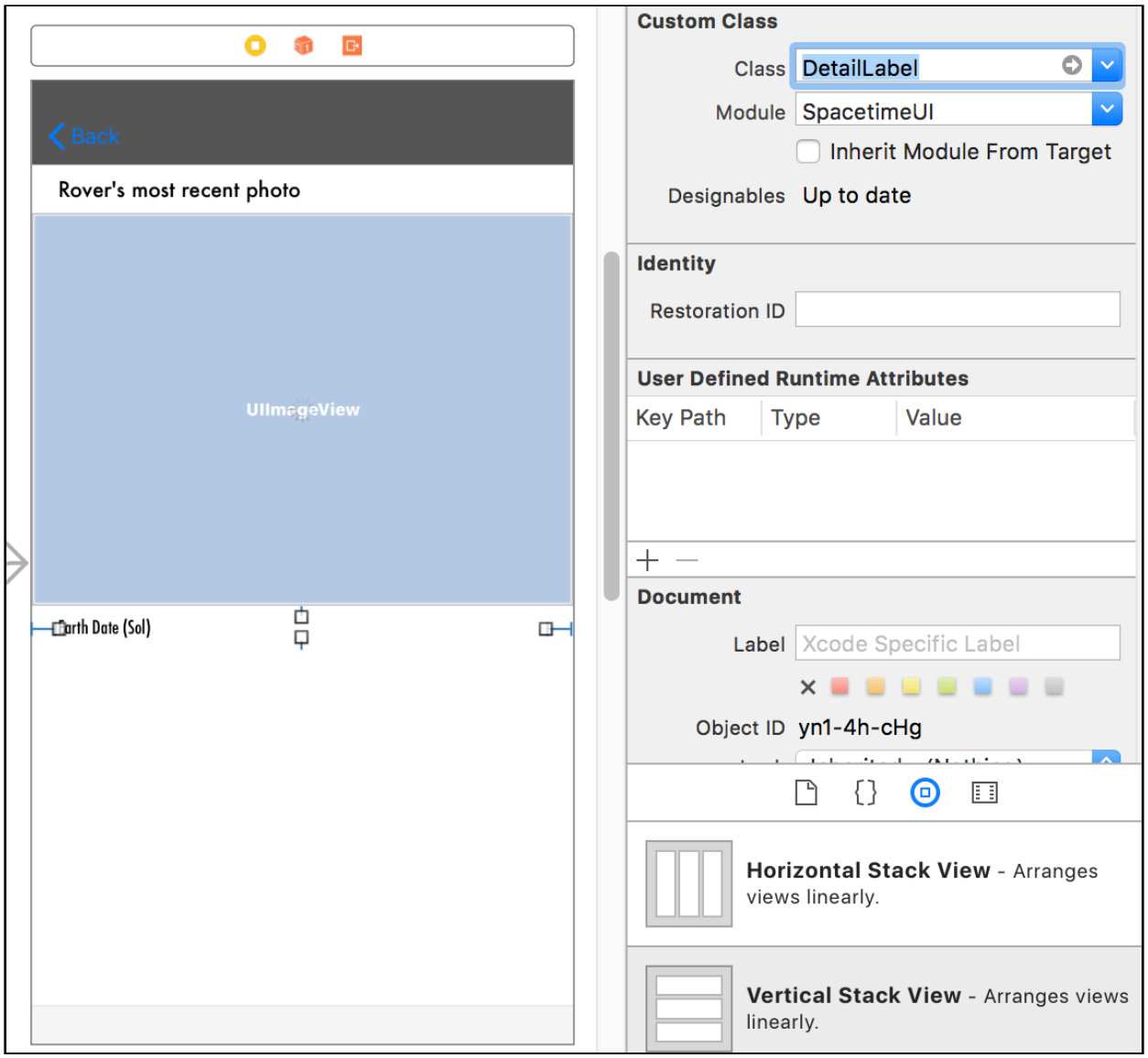
The code to actually do this rendering is in a method called `prepareForInterfaceBuilder()` - and that method calls into a `commonSetup()` method that all initializers call into, so that interface builder will set things up exactly as they would look when any other initializer is called.

Now, scroll down to the `ExplanationLabel` subclass. You'll notice it only has one overridden method - the very `commonSetup()` method which is called from all initializers and `prepareForInterfaceBuilder()`!

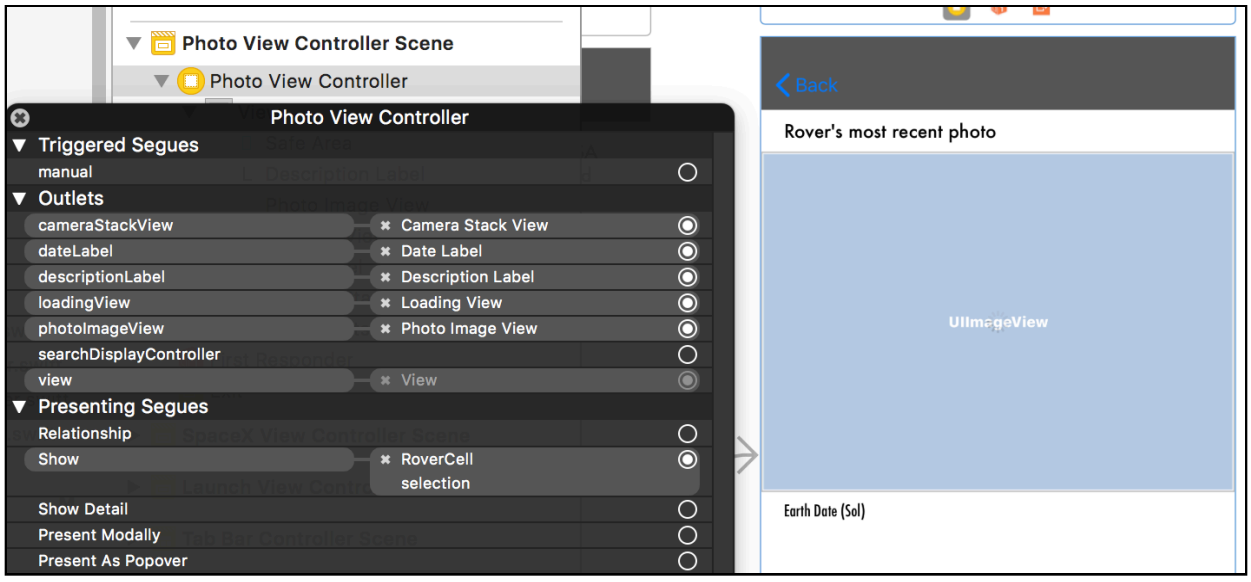
All the `commonSetup()` method does is set the style, which you saw when creating the style guide app causing all the proper colors and fonts to be set up for you.

But the lesson in this is that your style can power subclasses that are in turn powering Interface Builder - allowing your styles to be rendered in the storyboard just like they'll look on screen.

Now, head back to **Main.storyboard** and select the Date label. Change its type to `DetailLabel`, and it'll also update in the Storyboard:



Now, you've changed the two label types, so you have to hook them up again, right? Well, command click on the `PhotoViewController` in the left sidebar to show the list of outlets again, and you'll see that they're still hooked up:

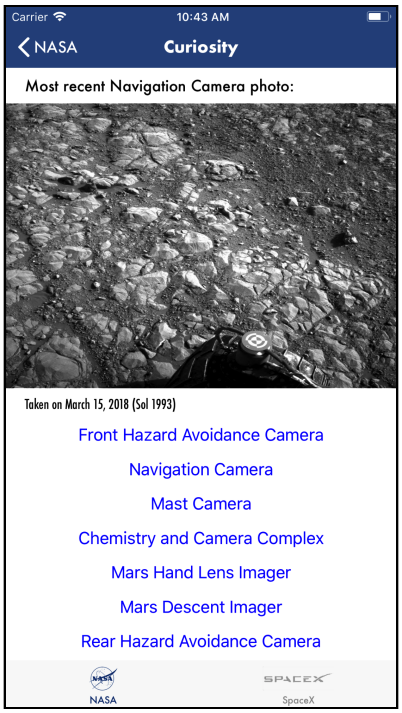


Why is that? Because your two custom classes have `UILabel` in their superclass hierarchy and you're not accessing any custom methods from these classes, you don't actually need to be more specific than that when declaring the `IBOutlet`.

This makes it really, really easy to change the class of a label in Interface builder. In fact, while you still have the outlets panel open, you can select any of the different classes for either label, and note that they remain hooked up. Neat!

After you mess around with this, make sure that the Description Label is back to type `ExplanationLabel` and the Date Label is back to type `DetailLabel`.

Build and run the application, and select one of the rovers again. This time, you'll see your styles in action for the labels!



Those buttons still look pretty boring, though. Why not use one of the custom styled buttons we already have?

Open **PhotoViewController.swift** and go to `viewDidLoad()`. Right now, there's a `for` loop which is creating buttons which are added to the stack view using these lines:

```
let button = UIButton()
button.setContentCompressionResistancePriority(.defaultLow,
  for: .vertical)
button.setTitleColor(.blue, for: .normal)
```

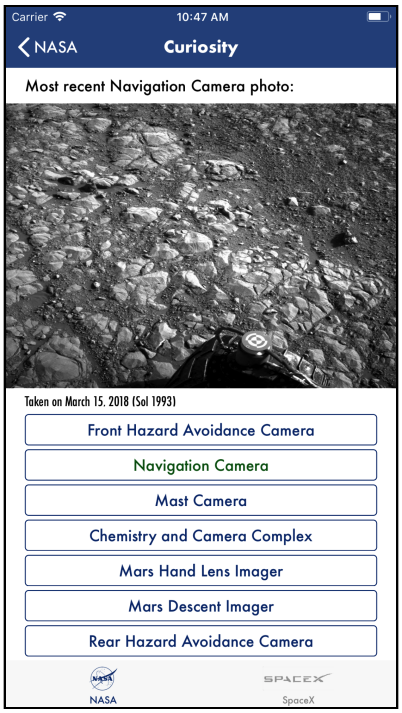
Remember that to access the `SpacetimeUI` framework from code, you need to import it. Add the following at the top of **PhotoViewController.swift** below the `UIKit` import:

```
import SpacetimeUI
```

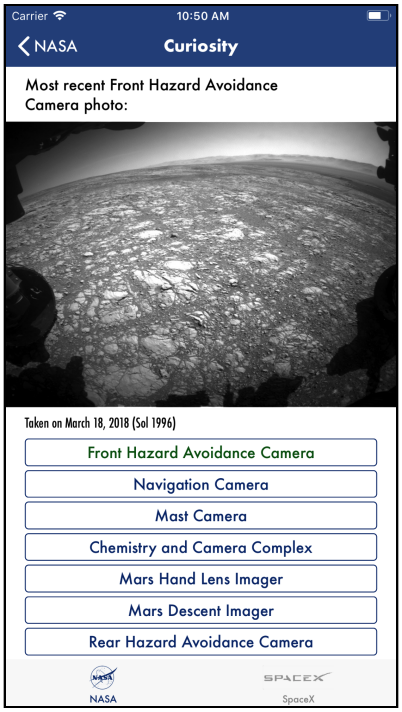
Replace the three lines setting up the vanilla buttons with a single line setting up one of the custom buttons you created:

```
let button = BorderedButton()
```

Since the buttons don't appear until the app is run, there's nothing to check in the storyboard - all you need to do is build and run. This time, when you tap into the detail, you'll see the fancy bordered buttons which were set up in the Style Guide:



You'll even see that the highlighting and selecting you tried in the style guide app work exactly the same way in the final application - when you tap on a new photo, that new button will become selected and reflect the proper color:



Whew! You've got your app fully styled with all your custom styles! Now it's time to see how powerful this technique is when it comes to wanting to make changes to your UI!

3) Changing Your Style Guide..Then Your App!

One of the things you probably noticed is that the text in the application is a bit monochromatic - it's essentially all using the `darkTextColor`. This can make an app feel really boring, or make it difficult to pick out what the most important portions of an app are.

You're going to add a new color for secondary text - something that is somewhat less emphasized than the main text of the application. In **SpacetimeColors.swift**, update `SpacetimeColor` to add a new case at the end:

```
...
buttonText,
secondaryText
```

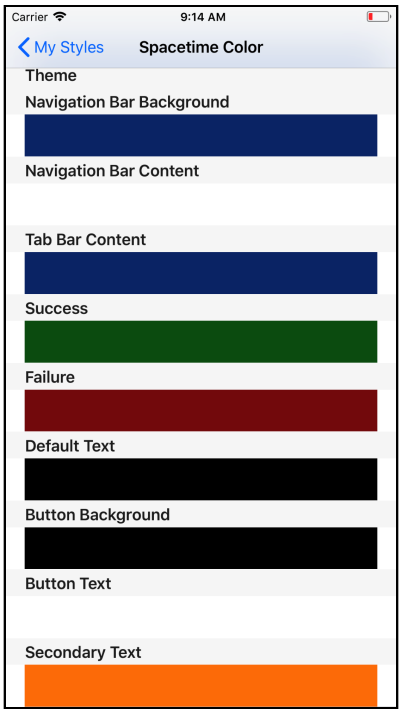
Then, in the computed `color` var, add handling for this new case - start with a ridiculous color so it's obvious that it worked:

```
case .secondaryText:
  return .orange
```

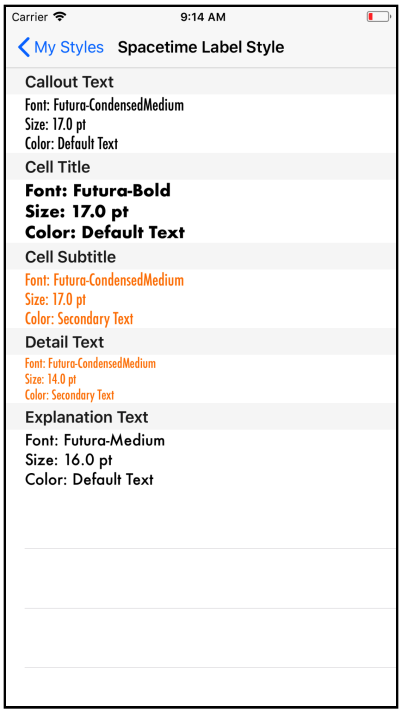
Next, in **SpacetimeLabels.swift**, update the computation of the `textColor` var so that the `cellSubtitle` and `detailText` labels have your new secondary text color by default:

```
var textColor: SpacetimeColor {
  switch self {
  case .calloutText,
      .cellTitle,
      .explanationText:
    return .defaultText
    case .cellSubtitle,
      .detailText:
    return .secondaryText
  }
}
```

Build and run the **StyleGuide** app. Tap on "Spacetime Color", and scroll down to the bottom of the list. You'll see the color you added right at the bottom:



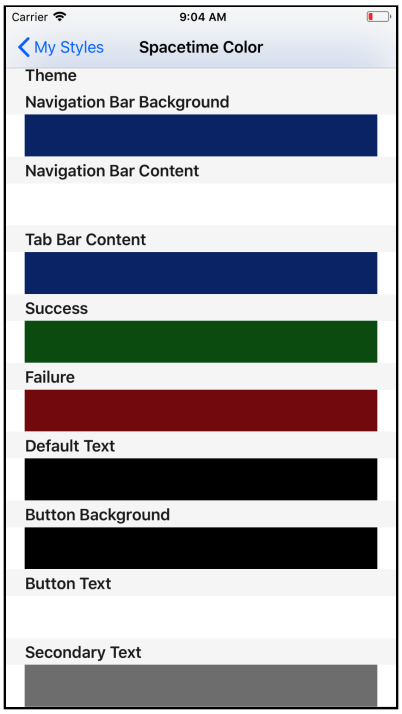
Go back to "My Styles", and tap on "Spacetime Label Style". You'll see the two label styles you updated with the new color:



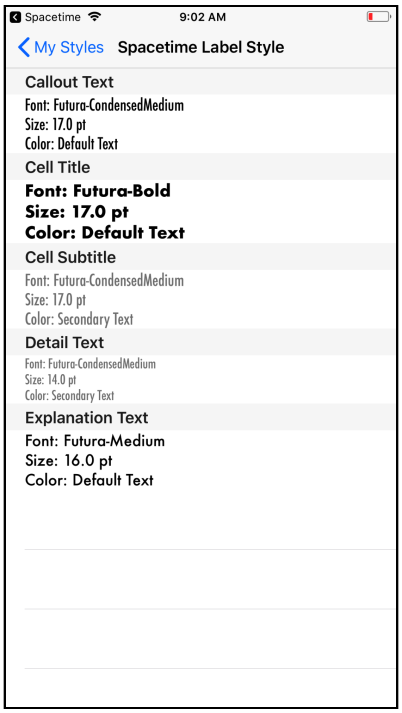
Looks pretty ridiculous, but it confirms that your changes are working. Time to try a color that might work a little better with your actual style. Perhaps something lighter than black but darker than white?

Go back to **SpacetimeColors.swift** and update the computed `color` variable for `secondaryText` to return `gray`:

```
case .secondaryText:
  return .gray
```



Looks pretty reasonable by itself, but how does it look when it's actually in use? Go back to "My Styles", and tap on "Spacetime Label Style". You'll see the two label styles you updated with the new color:

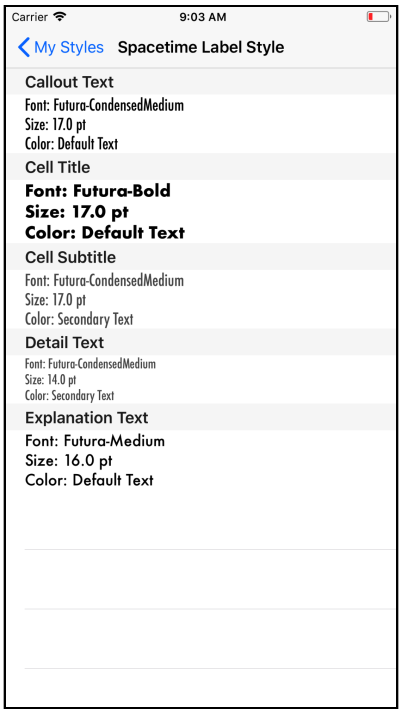


Neat, it applied! There's only one problem: It's a little bit hard to read because the color is so much lighter than the dark text color. Fortunately, there's a color for that.

Go back to **SpacetimeColor.swift** and update the computed `color` variable for `secondaryText` to return `darkGray`:

```
  case .secondaryText:
    return .darkGray
```

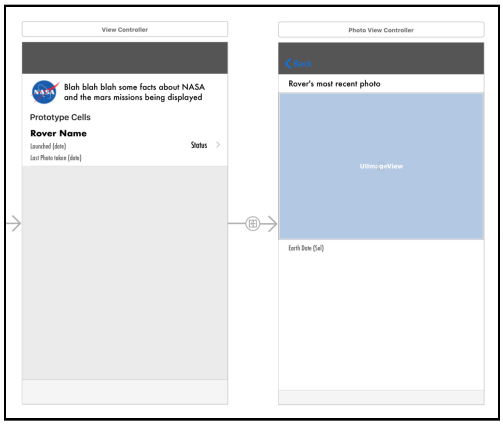
Build and run the **StyleGuide** app. You already know the color is updating, so you can skip checking the color in isolation and just tap the "Spacetime Label Style" row. Now, the text is much more readable:



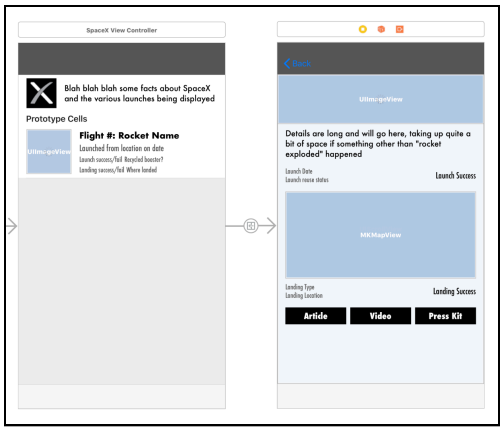
Sweet - you were able to make that comparison without having to launch the main app once! When you're working with a sprawling app, this can be an enormous time-saver.

Now that you've settled on a color, since you've made the changes in your framework and all your labels conform to `@IBDesignable`, all you have to do to see it in action is open up **Main.storyboard** and let it re-render.

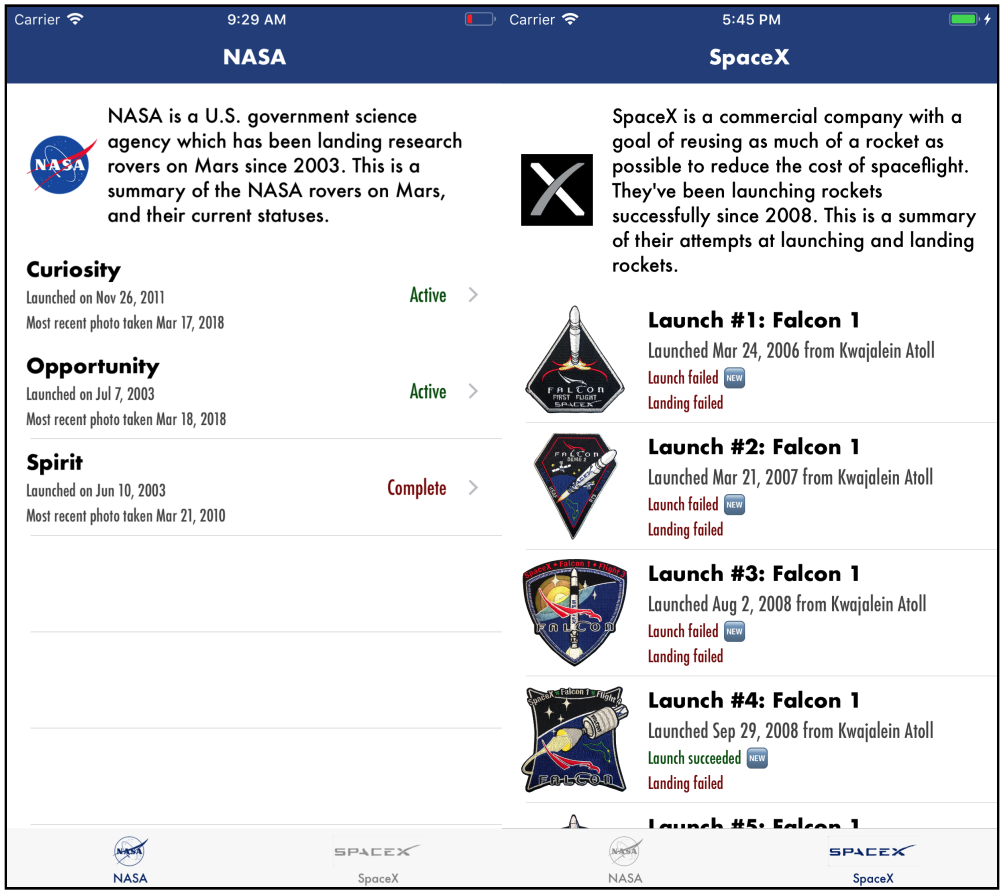
After the rendering completes (which may take a few seconds), you'll be able to see a sample of what it looks like directly for both your NASA view controllers:



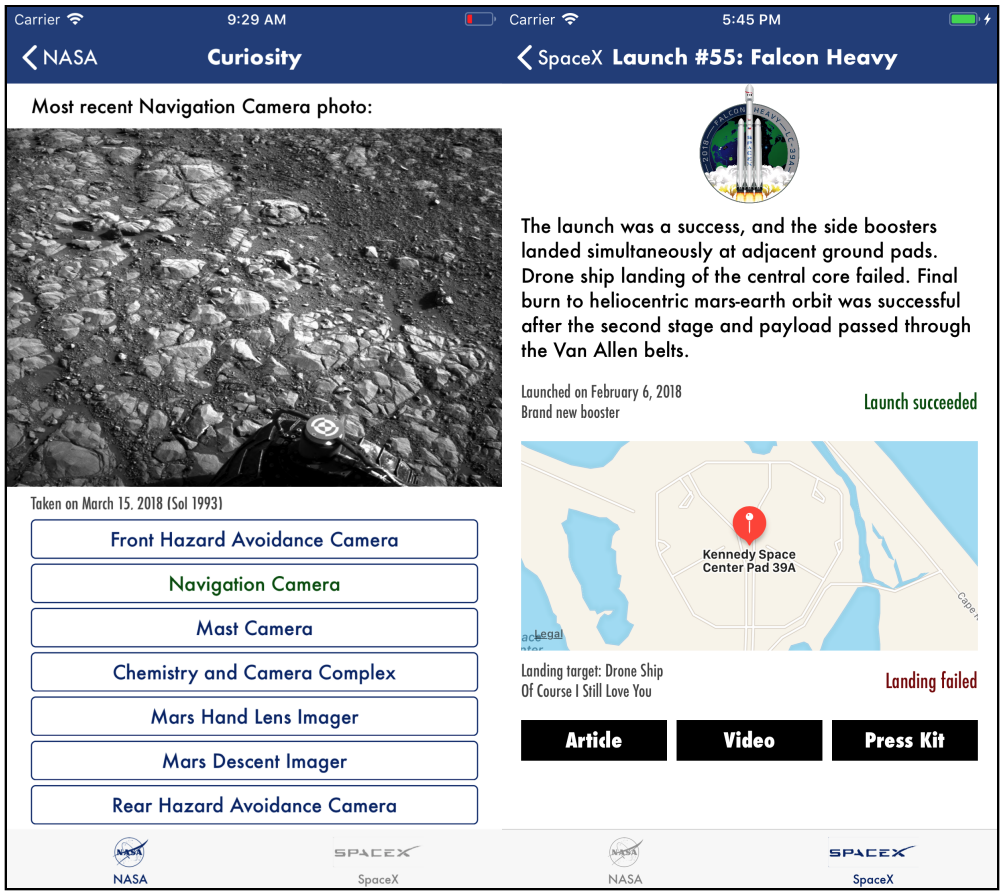
And your SpaceX view controllers:



But I know not everyone believes Interface Builder could ever possibly be right about what something will look like. Build and run the **Spacetime** app, and you'll immediately see that on both the top-level view controllers:



And the detail view controllers:



Your changes have been applied not just in the storyboard, but in the app. Awesome!

4) That's it!

You've now built a Living Style Guide application where you can see how your UI elements interact without changing them in the main application, and you've used that style guide to both add new elements to your app and change existing ones.