

Developers Log

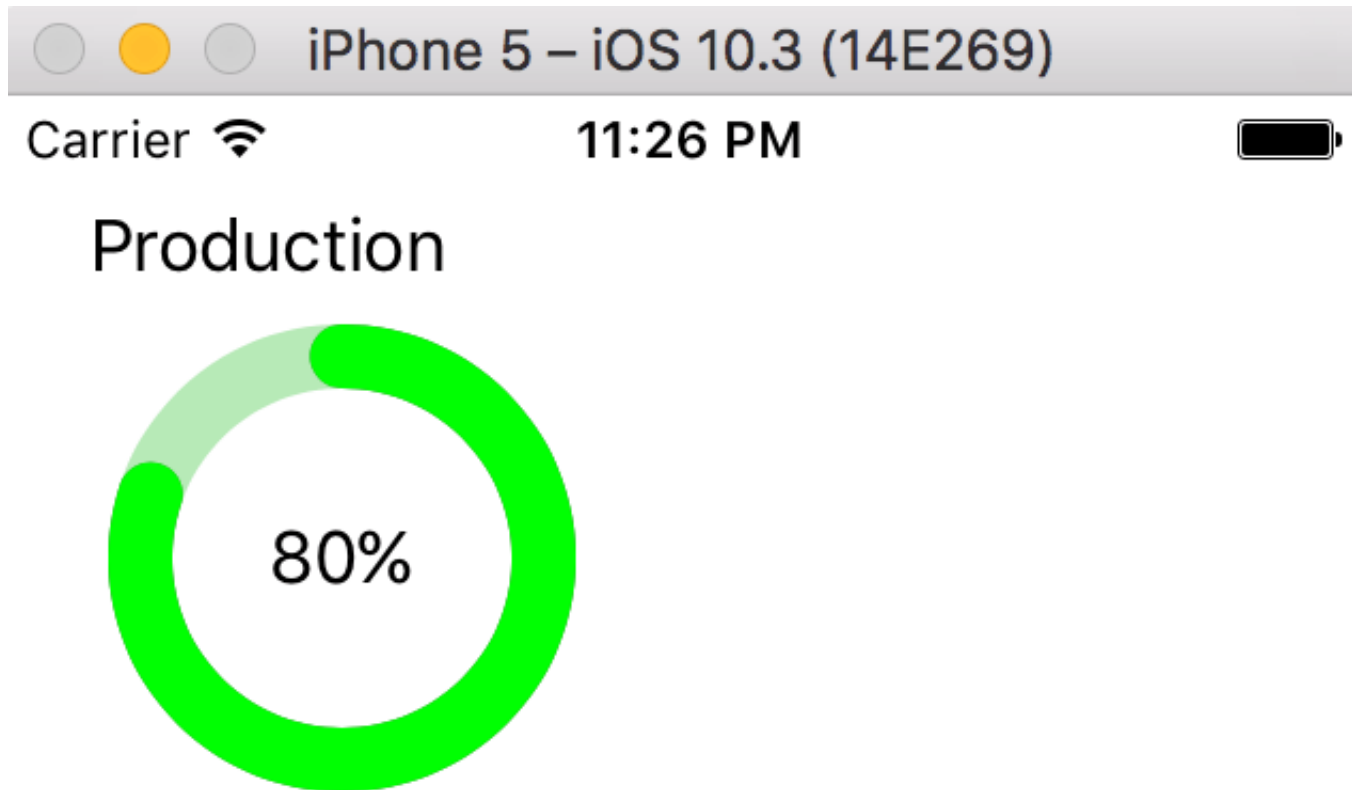
A blog about software developing and technology

Swift 3 : reusable components using .xib files and the Interface Builder

In this post we will see how to create reusable components using the Interface Builder: we will design our component (drag and drop of components , layout constraints, etc) as if we were working in a Storyboard, save the result as a .XIB file, and see how we can reuse our component in different Storyboards (or even in other .XIB files).

Our sample: a component with a gauge and a label

To illustrate the process of creating a component I will design a component like the one in the image below:



GaugeKit is an interesting library for creating ring like gauges. In my app I do not use gauges as isolated elements: they include a title indicating what kind of data I am representing, and a little label displaying a concrete number / percentage.

Since I use that same combination of elements in different screens, I 'd like to create a customizable component, and reuse it through the application

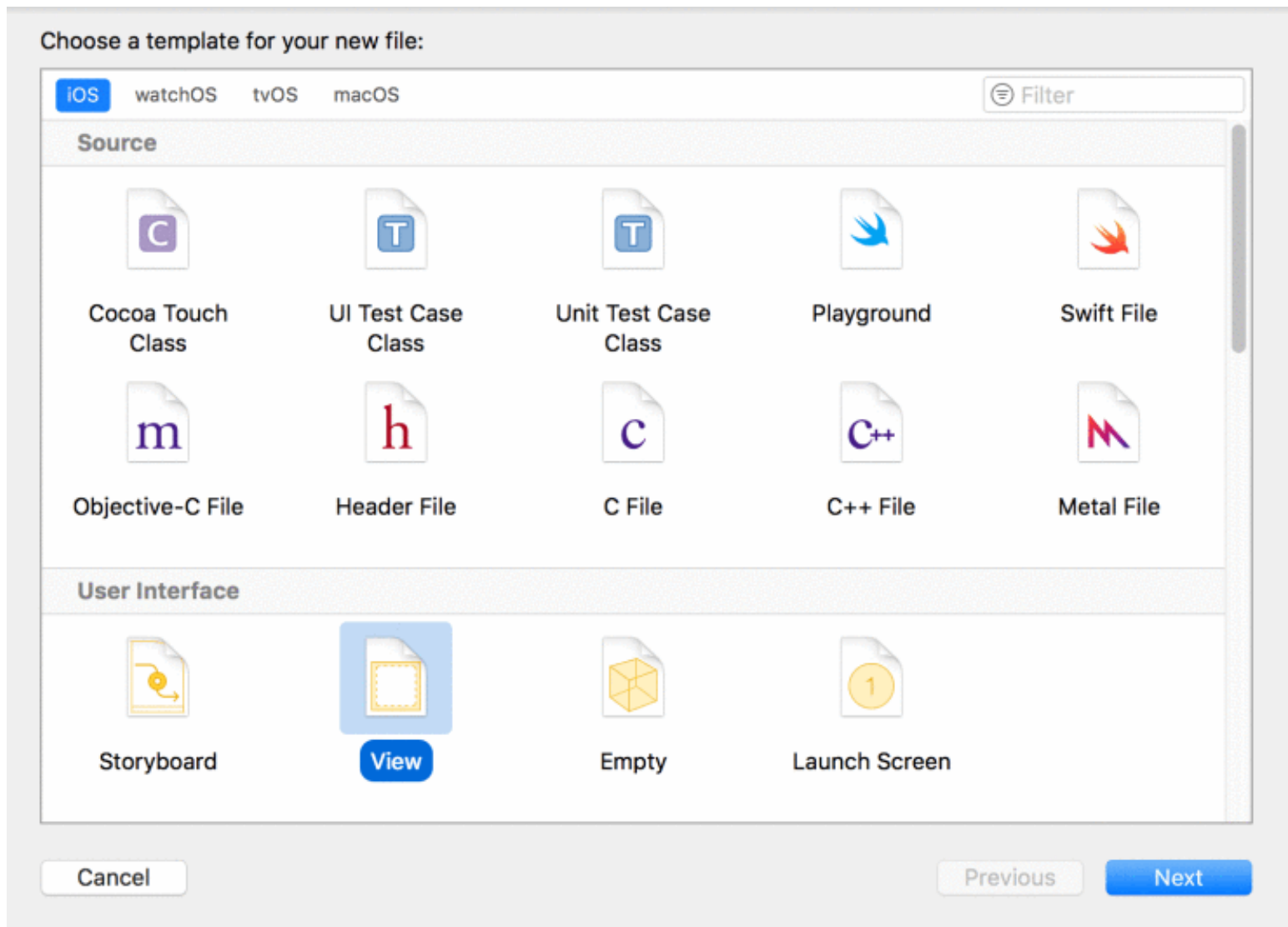
Create our UIView and Xib file associated

In order to create our custom component we will need to subclass a *UIView* component.

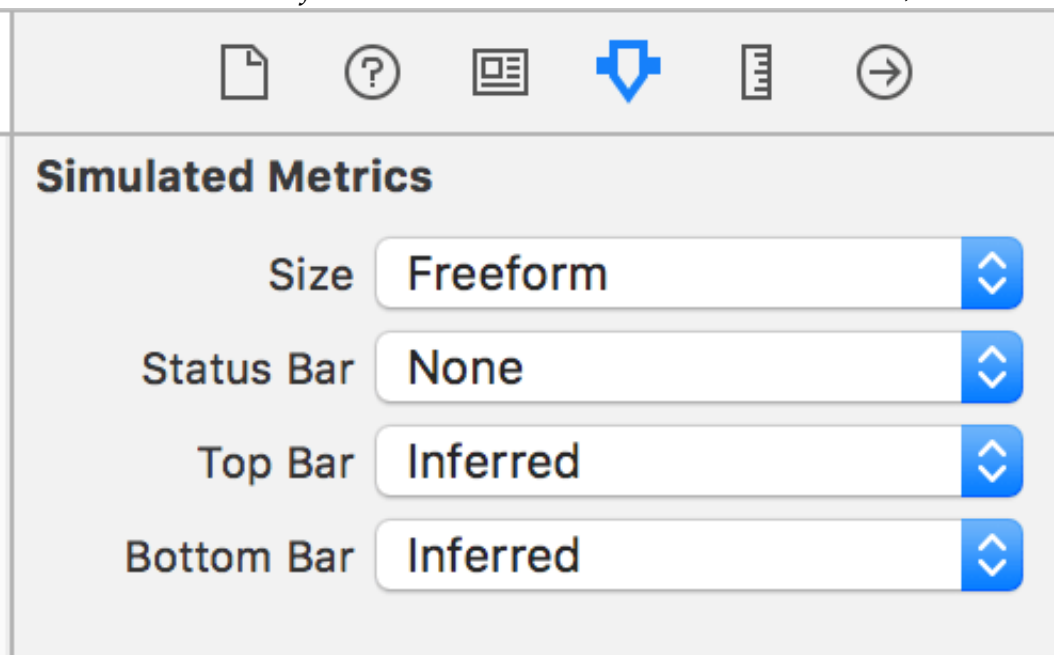
The *UIView* component offers us a rectangular area where we can draw, and add, other components (many common components like Labels, Buttons, etc are, in fact, extensions of the *UIView* component).

XCode offers to create automatically a *XIB* file, every time we create a new *CocoaTouch* Class extending from a *UITableViewCell* ... but for some reason that option is grayed out when you extend from an *UIView*. So we will have to create separately the *CocoaTouch* file, and the *XIB* file, and link them together manually.

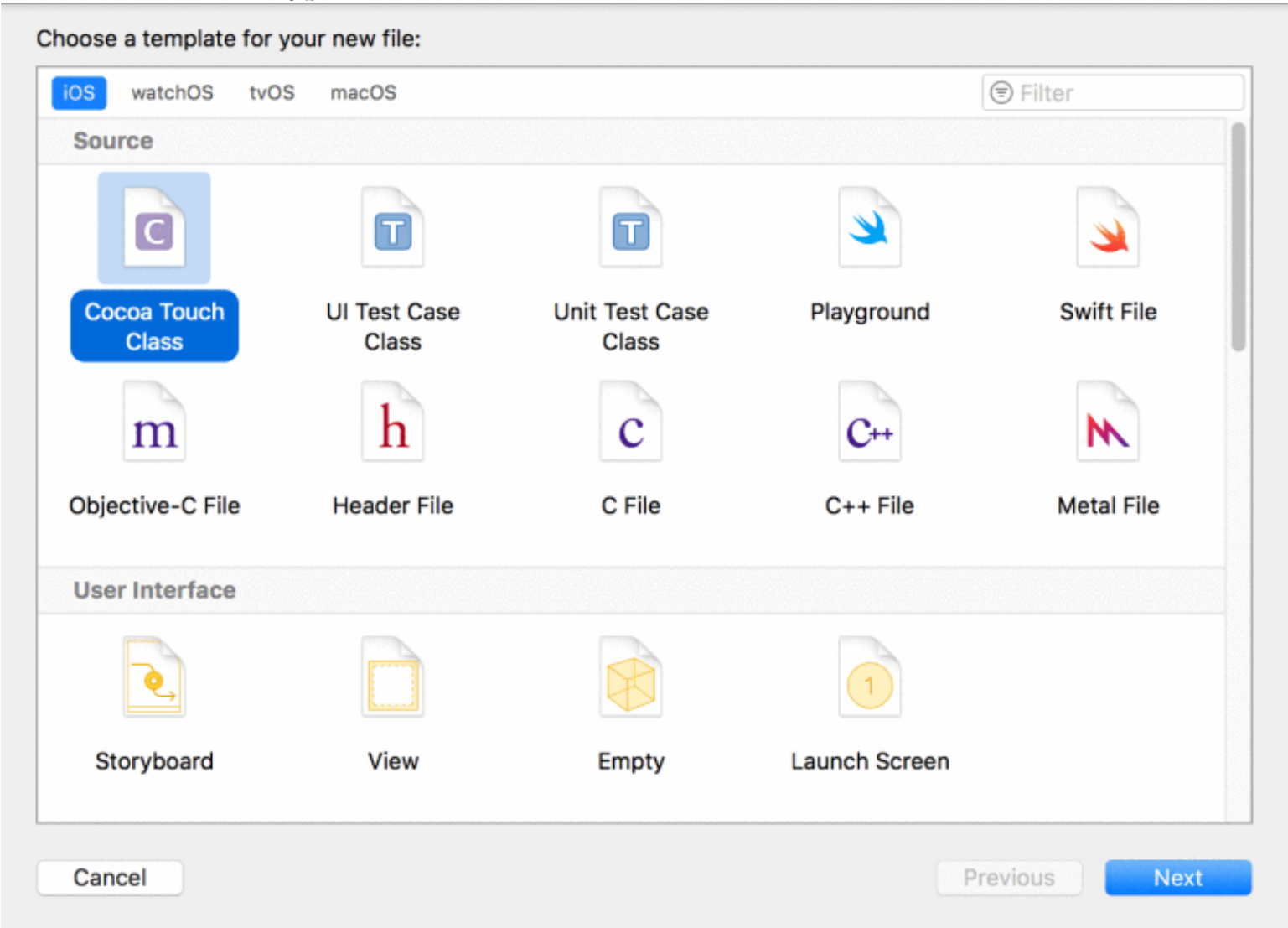
1. Create an *User Interface* file of type *View* and name it *SimpleGauge*



2. Set the size to *Freeform* to be able to resize the view at will , and set the *Status Bar* to *None*



3. Create a new file of type *Cocoa Touch Class*



4. Set the name to *SimpleGauge*, and make sure to select *UIView* as *Subclass*:

Choose options for your new file:

Class:

Subclass of:

☐ Also create XIB file

Language:

Cancel Previous Next

5. To connect the xib file to the SimpleGauge.swift file created in the step above, you need to select the File's Owner property of the XIB, and in the right panel, in the *Identity Inspector*, set the *Class* property in the *Custom Class* section to *SimpleGauge*:

Placeholders

File's Owner

First Responder

☐ View

Custom Class

Class:

Module:

☒ Inherit From Target

User Defined Runtime Attributes

Key Path	Type	Value
----------	------	-------

Document

Label:

Object ID: -1

Lock:

Notes:

Comment For Localizer

6. Next we need to add the code in our *SimpleGauge.swift* file to be able to initialize the *SimpleGauge* class from the xib file:

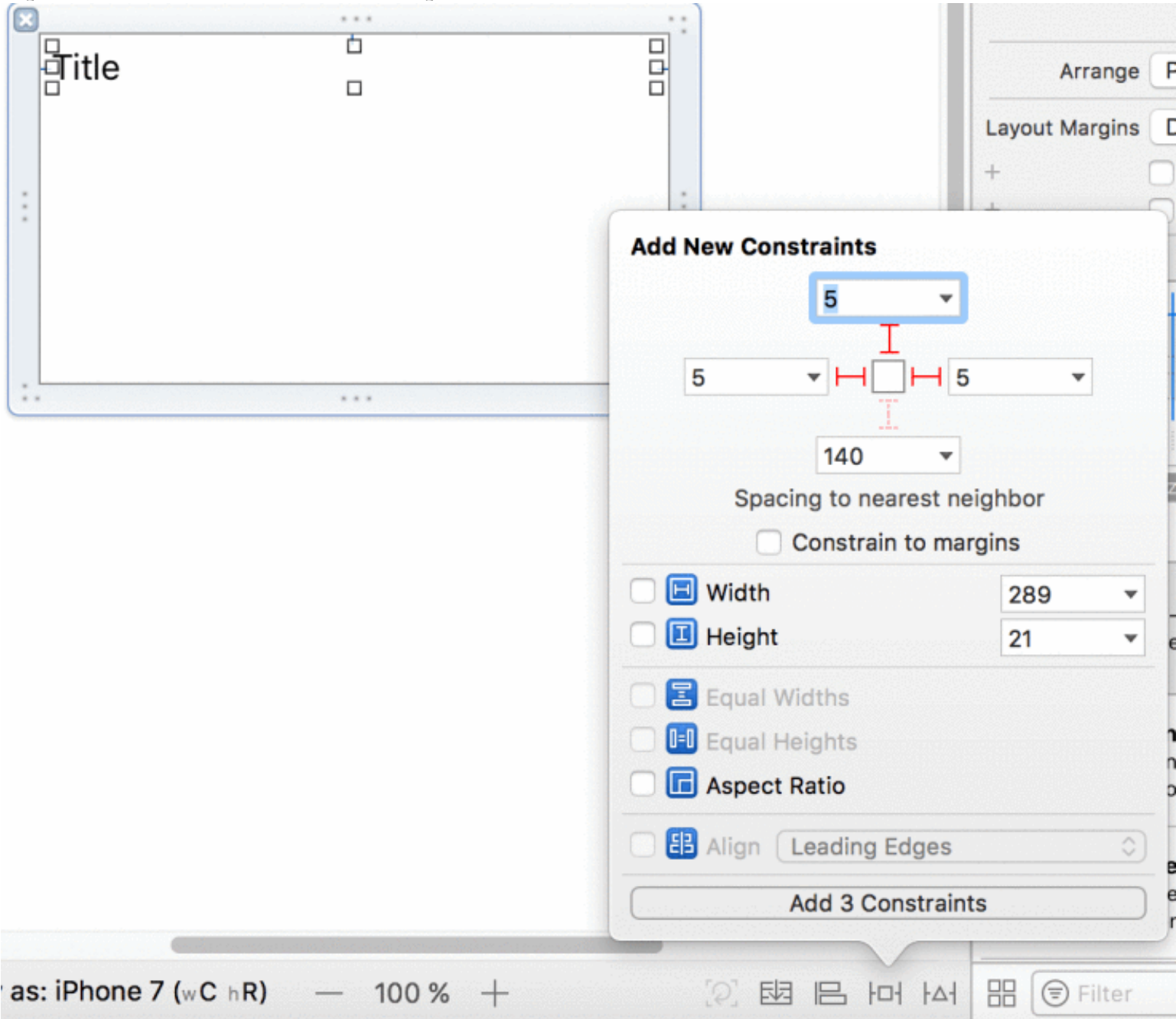
```
1 func loadViewFromNib() -> UIView {
2     let bundle = Bundle(for: type(of: self))
3     let nib = UINib(nibName: "SimpleGauge", bundle: bundle)
4     let view = nib.instantiate(withOwner: self, options: nil).first as! UIView
5
6     return view
7 }
```

And the whole class should look like this:

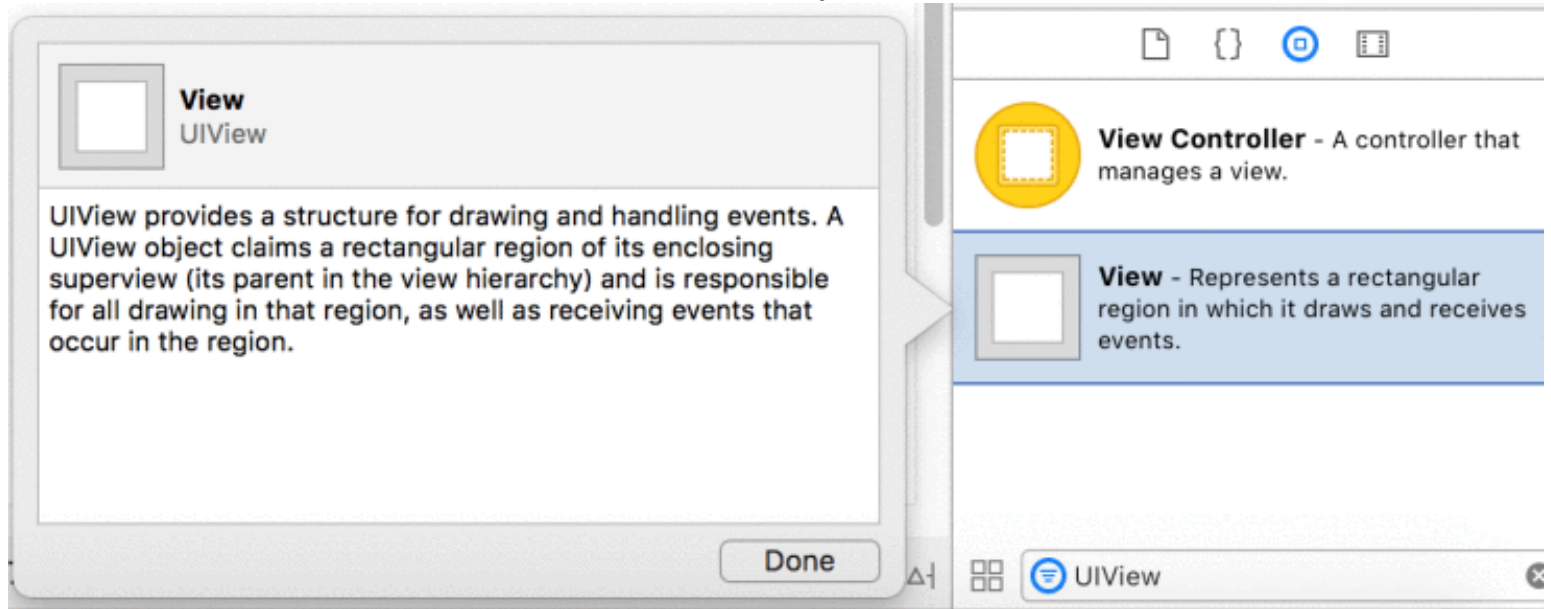
```
1 class SimpleGauge: UIView {
2
3     var view: UIView!
4
5     override init(frame: CGRect) {
6         super.init(frame: frame)
7         xibSetup()
8     }
9
10    required init?(coder aDecoder: NSCoder) {
11        super.init(coder: aDecoder)
12        xibSetup()
13    }
14
15    func xibSetup() {
16        view = loadViewFromNib()
17        view.frame = bounds
18        view.autoresizingMask = [.flexibleWidth, .flexibleHeight]
19
20        addSubview(view)
21    }
22
23    func loadViewFromNib() -> UIView {
24        let bundle = Bundle(for: type(of: self))
25        let nib = UINib(nibName: "SimpleGauge", bundle: bundle)
26        let view = nib.instantiate(withOwner: self, options: nil).first as! UIView
27
28        return view
29    }
30
31 }
```

7. Start designing the component by adding a *UILabel* component to act as the title of our gauge component.

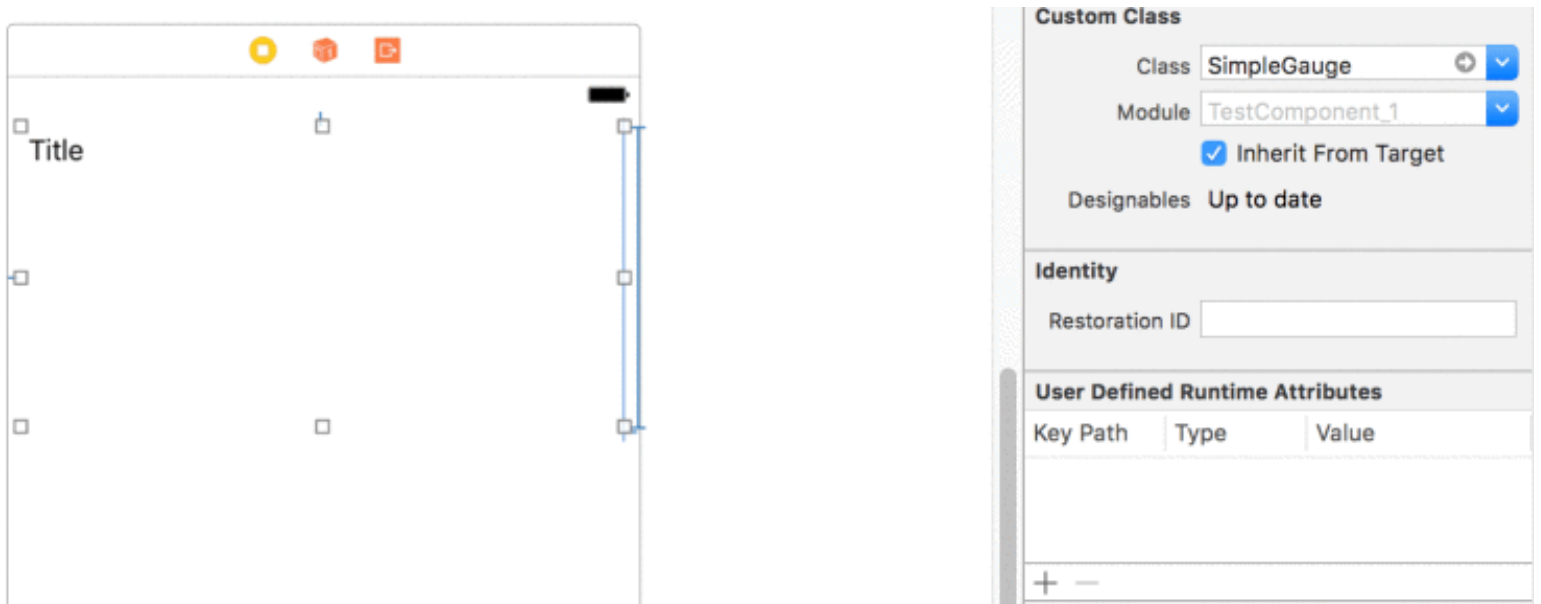
Add the top, leading, and trailing constraint to set it at the top part of the component, and let the title expand over all the width of the component.



8. Let's test what we have done so far: back in the storyboard let's add a *UIView* element:



Select the view, and in the *Identity Inspector* we will set the *Class* property to *SimpleGauge* to indicate that this *UIView* component it is actually our custom *SimpleGauge* component:



Notice that if you have been following the steps described here, your *UIView* will have a crucial difference with the image posted above: despite having indicated that the *UIView* is a *SimpleGauge* component, in your Storyboard it is still displaying as a blank *UIView* (no title label). In order to see how it looks like, you would need to run the application in the simulator.

Adding the attribute *@IBDesignable* to the *SimpleGauge* class will allow you to see the design while in the Interface Builder. Right like in the image above, so you should update our code like this:

```
1 | @IBDesignable class SimpleGauge: UIView {
2 | // ...
3 | }
```


At this point I would recommend to run the application to check that everything is working correctly, before we proceed to the next stage.

Adding the Gauge component and finishing the layout

Now that we have the *XIB* view file created, and connected to a swift class to manage the logic of the program, we can finish the design of the component. This will include adding the Gauge component, and adding the necessary constraints to set up the layout:

1. As described in the GaugeKit site, you can import the GaugeKit library to the project, either using CocoaPods, or Carthage. If we use CocoaPods we will update our *Pod* file like this:

```
1 | # Uncomment the next line to define a global platform for your project
2 | # platform :ios, '9.0'
3 |
4 | target 'mapper' do
5 |   # Comment the next line if you're not using Swift and don't want to use frameworks!
6 |   use_frameworks!
7 |
8 |   # Pods for mapper
9 |   pod 'GaugeKit'
10 | end
```

Carthage users need to update instead their *Cartfile*:

```
1 | github "skywinder/GaugeKit" >= 0.2
```

If you need more information on how to use CocoaPods, or Carthage you can check this post on how to add third party libraries to your Swift projects.

2. To add a gauge chart in our *XIB* we must add an *UIView*, and in the *Identity Inspector* we will set the *Class* property to *Gauge*; the module should automatically set itself to *GaugeKit*.

Custom Class

Class

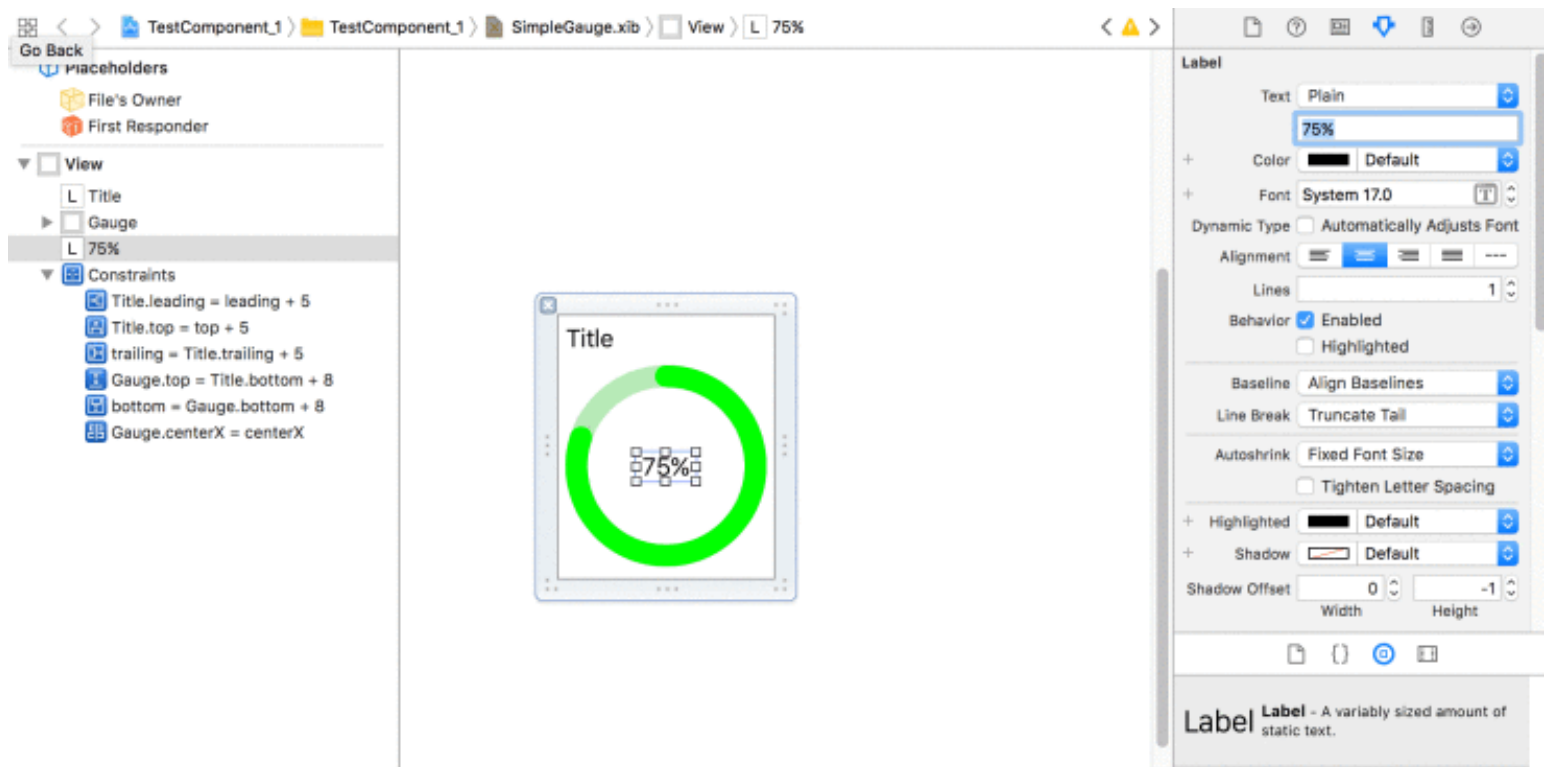
Module

☐ Inherit From Target

Designables Up to date

3. We add a new *UILabel* to represent the value at the center of the Gauge. And now we need to add the constraints to finish the layout.

Add the label:



I want the chart centered horizontally in the component, and I want it to adapt to the size of the parent container.

- Set top and bottom constraint to leave some space between the gauge, the title, and the bottom of the component
- Center the gauge horizontally respect the parent view
- Set upon the *Gauge* itself a *ratio* constraint and set it to 1:1 (this will keep the *Gauge* view a square: width: equals height)

Placeholders

File's Owner

First Responder

View

Title

Gauge

Constraints

Title.leading = leading + 5

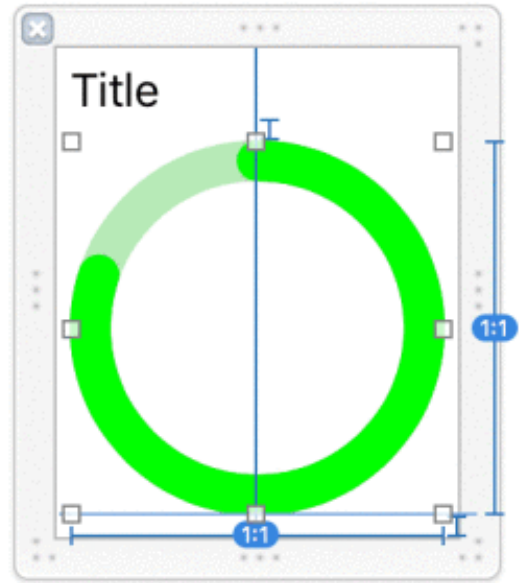
Title.top = top + 5

trailing = Title.trailing + 5

Gauge.top = Title.bottom + 8

bottom = Gauge.bottom + 8

Gauge.centerX = centerX



Center the value label, vertically, and horizontally respect the Gauge:

TestComponent_1 > TestComponent_1 > SimpleGauge.xib > View > L 75%

Placeholders

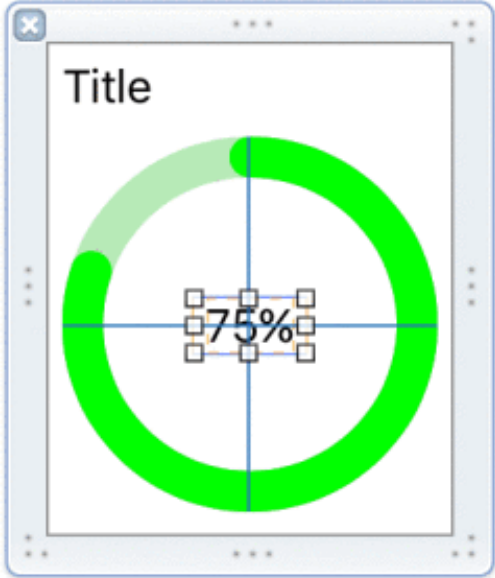
- File's Owner
- First Responder

View

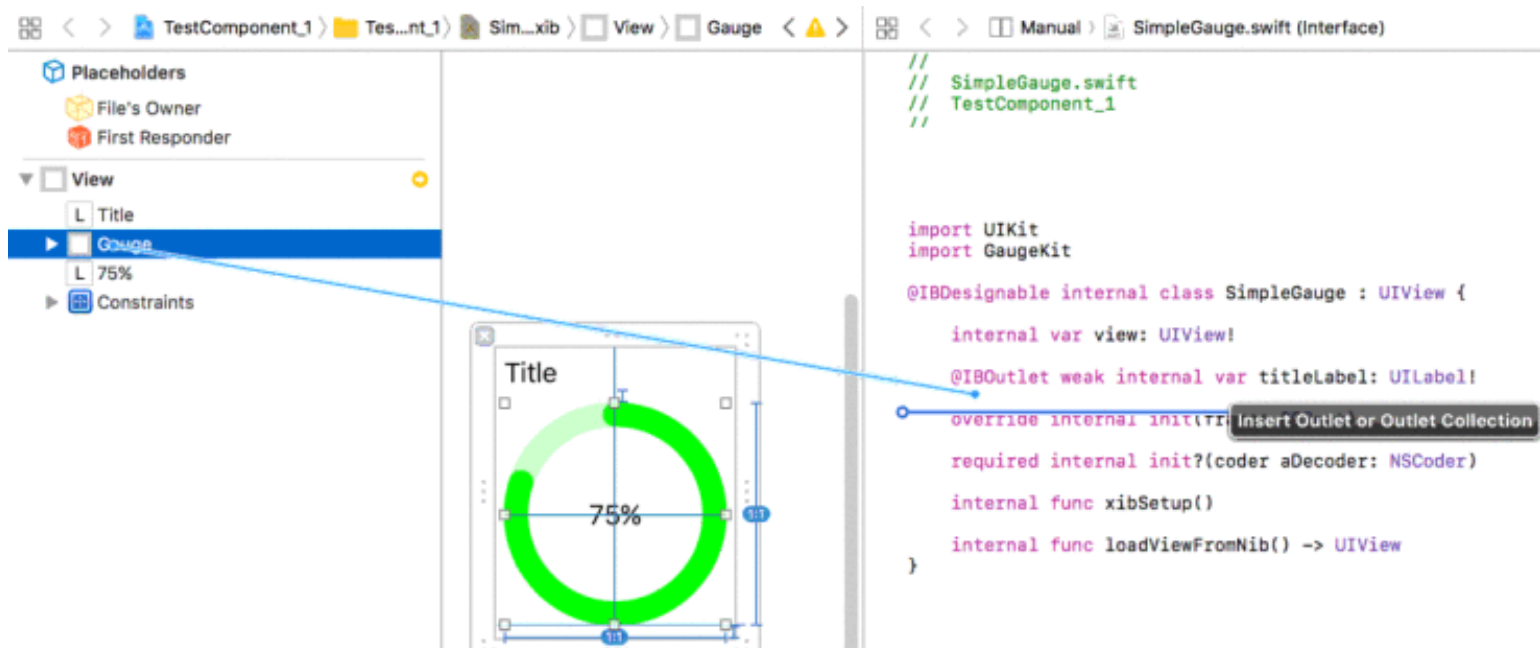
- Title
- Gauge
- 75%

Constraints

- Title.leading = leading + 5
- Title.top = top + 5
- trailing = Title.trailing + 5
- Gauge.top = Title.bottom + 8
- bottom = Gauge.bottom + 8
- Gauge.centerX = centerX
- 75%.centerX = Gauge.centerX
- 75%.centerY = Gauge.centerY



4. We add *IBOutlet*s to the *SimpleGauge* class for the gauge component, and both labels, in order to allow the title, value, and properties of the gauge (rate, max) to be accessible from code.



Finishing touches: @IBInspectable properties

At this point we already have a ready to use, reusable XIB component. There is still a little improvement we can add: we could make properties like the title, the text in the value label, etc customizable from the *Interface Builder*.

For that we need to use the *@IBInspectable* property:

```

1  @IBInspectable var valueLabel: String? {
2      get {
3          return valLabel.text
4      }
5      set(valueLabel) {
6          valLabel.text = valueLabel
7      }
8  }
9
10 @IBInspectable var titleGauge: String? {
11     get {
12         return titleLabel.text
13     }
14     set(titleGauge) {
15         titleLabel.text = titleGauge
16     }
17 }
```

Notice than now we can set the title of the component, and the text in the label at the center of the gauge from the *Interface Builder* like this:



Actually I am not really going to need the *Value Label* property, since, like the rate property, and the max value property of the gauge, is going to be set in runtime through the Outlets provided some steps before. But it can be useful when you are designing the whole screen in the storyboard, to set default values to get an idea of how the screen is going to look like.

And this is all. Using the principles described above you can make your own different custom components, combining different components to create more complex ones. I hope it was of help.

developerslogblog

□ May 14, 2017

Quick and Dirty Guide to ..., Software Development

FrontEnd, How to ..., mobile, Swift

BLOG AT WORDPRESS.COM.

UP ↑