306: Animating Auto Layout Constraints

Part 3: Lab Instructions

306: Animating Auto Layout Constraints
Part 3: Lab Instructions

Copyright © 2014 Razeware LLC.

All rights reserved. No part of this book or corresponding materials (such as text, images, or source code) may be reproduced or distributed by any means without prior written per- mission of the copyright owner.

This book and all corresponding materials (such as source code) are provided on an "as is" basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose, and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in action of contract, tort or otherwise, arising from, out of or in connec- tion with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this book are the property of their respective owners.

Even More Animations

At this point the UI of the project looks pretty jumpy. This however is nothing compared to where you are about to bring it by the end of the lab.

Following the instructions in this document you will add the following features to the project:

1. **Reading animations**. As of now only the Speaking strip has a special animation applied to it. No fair! You will add a custom animation also to the Reading strip.
2. **Interface Builder integration**. You will learn how to connect an outlet to a constraint in Interface Builder and use it to access a constraint to animate it.

That's what you'll work on in this lab!

Adding a custom animation to the Reading strip

You'll start by custom Reading animations. In **ViewController.swift**, find and replace in `viewDidLoad()` this:

```
Selector("toggleView:")
```

with:

```
Selector("toggleReading:")
```

This will allow you to have a custom tap handler for the Reading strip. Add the initial version of that method:

```
func toggleReading(tap: UITapGestureRecognizer) {
  toggleView(tap)
  let isSelected = (selectedView==tap.view!)

  //custom animations

  UIView.animateWithDuration(0.5, delay: 0.0, options:
.CurveEaseOut, animations: {
    self.readingView.layoutIfNeeded()
  }, completion: nil)
}
```

Just as demoed during the first part of the tutorial for `toggleSpeaking(...)` you call `toggleView(...)` and determine whether the strip is selected or not and save the result in `isSelected`.

This code will take care to resize all three strips accordingly to the current selection. At the end of the method you call `layoutIfNeeded()` inside an animation block so all changes you do to the layout will be animated.

Now you can add your own custom animation code where the `//custom animations` comment is.

Just as before you will add a new method to run your custom animation, replace the comment with:

```
toggleReadingImageSize(readingImage, isSelected: isSelected)
```

And to remove the Xcode error add the inital version of `toggleReadingImageSize(...)`:

```
func toggleReadingImageSize(imageView: UIImageView, isSelected:
Bool) {

}
```

In a similar way to how the code finds all the `.Height` constraints in `adjustHeight(...)` you will need to find the `.Height` constraint of the `readingImage` image view.

The image view is already connected via an outlet to the image in the Reading strip and you pass it over as the `imageView` parameter of `toggleReadingImageSize(...)`.

Add the code to loop over the constraints affecting `imageView` and also insert an `if` to check whether you've reached the `.Height` constraint:

```
for constraint in imageView.superview!.constraints() as
[NSLayoutConstraint] {
  if constraint.firstItem as UIView == imageView &&
constraint.firstAttribute == .Height {

  }
}
```

Inside the `if` statement the first thing you will need to do is to remove the existing `.Height` constraint:

```
imageView.superview!.removeConstraint(constraint)
```

Then you will need to create the new constraint. If the strip is currently selected the image height will be 33% of the strip height, when it's not that will be 67%. You will shrink the image when the user taps the strip – this will create a comical effect.

Append just after the last code:

```
let newConstraint = NSLayoutConstraint(
  item: constraint.firstItem,
  attribute: .Height,
  relatedBy: .Equal,
  toItem: (constraint.firstItem as UIView).superview!,
  attribute: .Height,
  multiplier: isSelected ? 0.33 : 0.67,
  constant: 0.0)
imageView.superview!.addConstraint(newConstraint)
```

You add the new constraint between the image view and its parent and your code is complete.

To make a minimal optimisation you will also break the loop – there's only one `.Height` constraint to the image view. Append still inside the `if` statement:

```
break
```

It's time you test the code and see the custom animation. Run the app – the Reading strip should look as usual:

When you tap Reading the image will start to grow (because its original `.Height` constraint will force it to) but then the new `.Height` constraint will make it shrink. It looks as if the image grows to push away the two labels and as soon as it touches them shrinks in place. `WOOt`!

Interface Builder Integration

As you can imagine you can get sick of enumerating over all constraints pretty soon. Lucky for you you can significantly decrease the amount of code you need to write by integrating with Interface Builder.

A constraint is a UI element that you can create an outlet to just as with any UIView, UIButton, or UISwitch.

In this part of the lab you are going to create an outlet to the `.Trailing` constraint of the image in the Reading strip and animate it. By doing this you will learn how to:

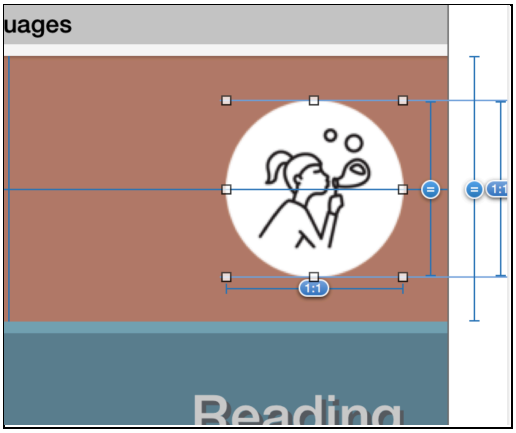
1. Animate an existing constraint without replacing it.

2. Create an IB outlet to a constraint.

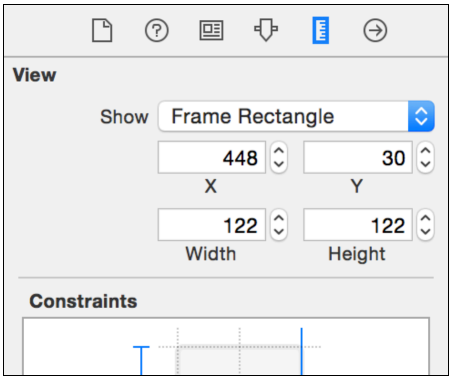
First add the outlet declaration inside the `ViewController` class. Add the line just under `class ViewController...`:

```
@IBOutlet var speakingTrailing: NSLayoutConstraint!
```

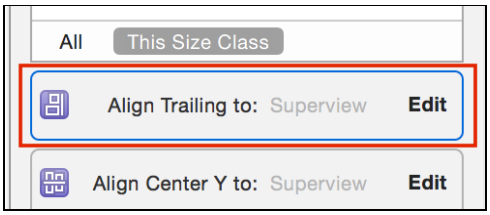
To connect the outlet open Interface Builder and select the Speaking image:



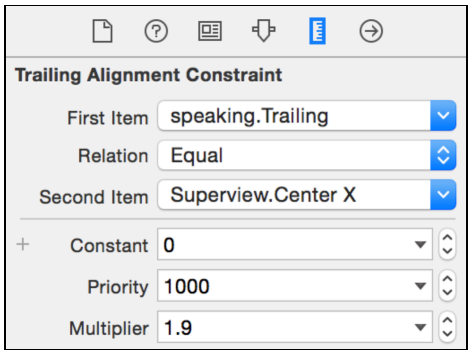
Open the Size Inspector (2nd tab from right to left):



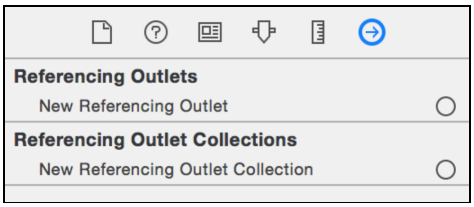
Double click the `.Trailing` constraint:



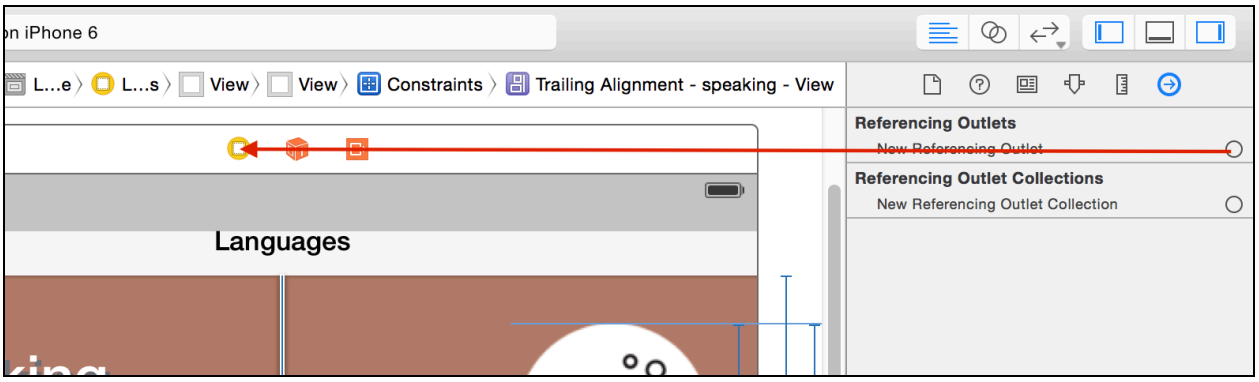
This will open the constraint for editing:



Select the Connections tab and you will see the UI allowing you to connect the constraint via an outlet to your code:



While holding the Ctrl button on your keyboard drag with the mouse from Referencing Outlets to your View Controller object in Interface Builder:



From the popup menu select **speakingTrailing** and you're done!

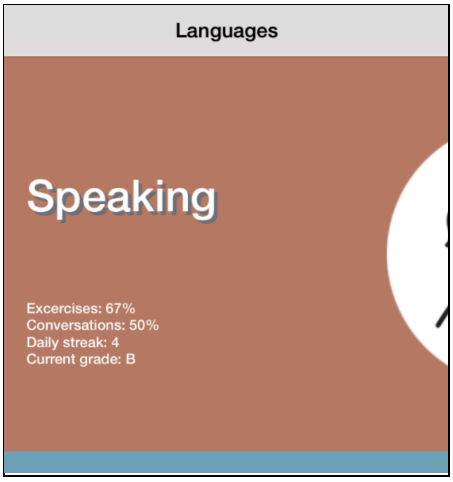
Now you can access the live constraint at run time just as any other outlet your have in `ViewController`. But what to do with it?

Since the only read-write property is the `constant` of the constraint – this is also the only property you can animate without replacing the constraint with a new one.

Let's offset the image by increasing the constant when the user taps the Speaking strip. Add in `toggleSpeaking(...)` inside the animation block like so (the code to insert is highlighted):

```
UIView.animateWithDuration(1.0, delay: 0.00,
usingSpringWithDamping: 0.4, initialSpringVelocity: 10.0, options:
.CurveEaseIn, animations: {
  self.speakingTrailing.constant = isSelected ?
self.speakingView.frame.size.width/2.0 : 0.0
  self.changeDetailsTo(isSelected ? kSelectedDetailsText :
kDeselectedDetailsText)
  self.view.layoutIfNeeded()
}, completion: nil)
```

This will move away the image to make space inside the strip. Run the app to see the effect:



The image moves away when you select Speaking but does not return if you tap another strip afterwards. You need to adjust the "deselect" handler. At the bottom of `toggleSpeaking(...)` add inside the `deselectCurrentView` closure:

```
deselectCurrentView = {
  self.speakingTrailing.constant = 0.0
  self.changeDetailsTo(kDeselectedDetailsText)
}
```

This will bring back the image to its original location each time the Speaking strip is deselected.

Congratulations, the UI looks amazing! You’re ready to continue on to the challenges, where you’ll rinse and repeat one more time to make sure your new skills are here to last.