Project 6: Auto Layout

Overview

Brief: Get to grips with Auto Layout using practical examples and code.
Learn: Auto Layout

- Setting up
- Advanced Auto Layout
- Auto Layout in code: addConstraints
- Auto Layout metrics and priorities: constraintsWithVisualFormat
- Wrap up

Setting up

In this technique project you're going to learn more about Auto Layout, the powerful and expressive way iOS lets you design your layouts. We used it in Project 2 to make sure our flag buttons were positioned correctly, but that project has a problem: if you rotate your device, the flags don't fit on the screen!

So, we're first going to fix Project 2 so that it demonstrates more advanced Auto Layout techniques (while also making the flags stay on the screen correctly!), then take a look at ways you can use Auto Layout in code.

First: take a copy of Project 2, then open it in Xcode. All set? Then let's begin…

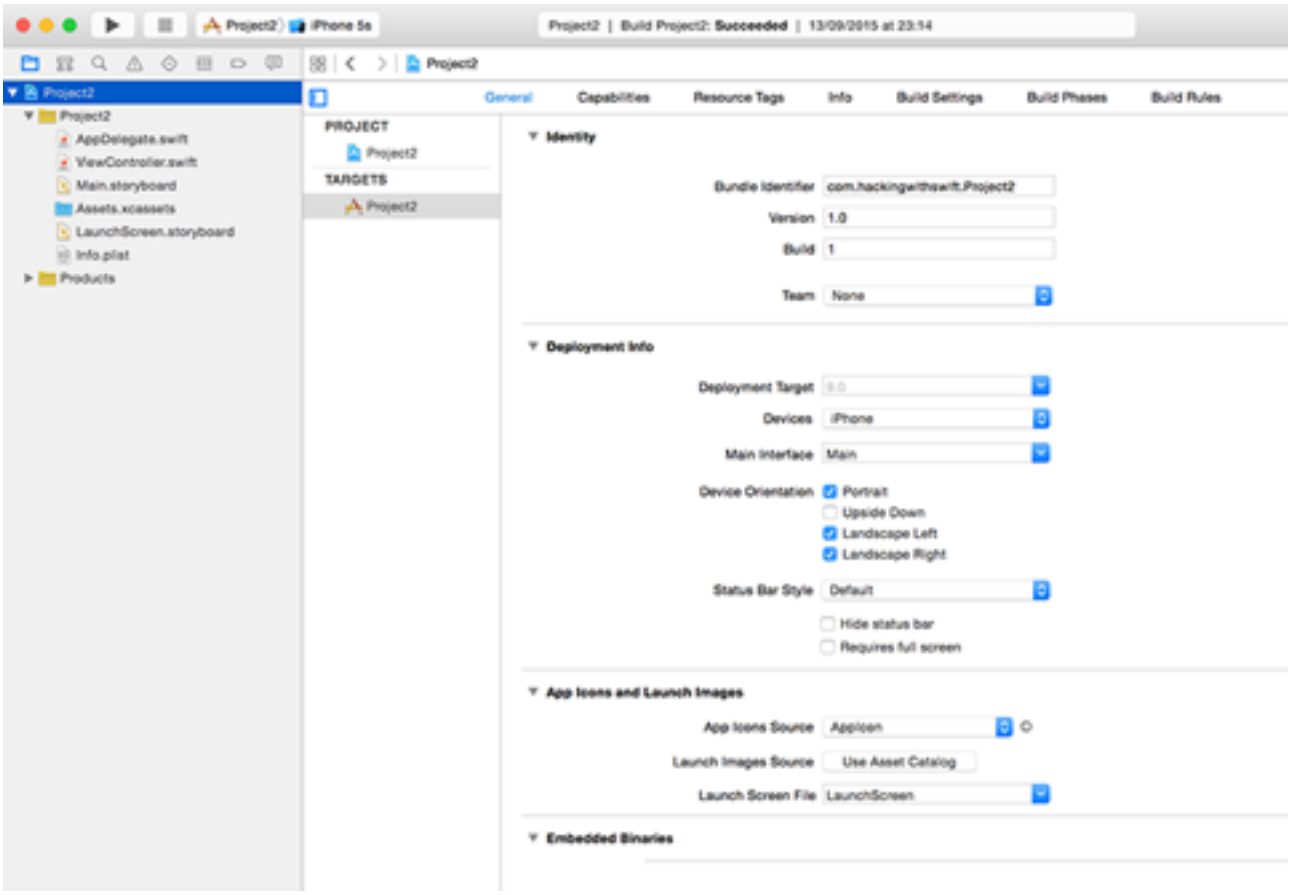
Advanced Auto Layout

When you run the project, it looks fine in portrait, but is unplayable on landscape because some of the buttons are hidden. You have two options: either disable landscape mode, or make your layout work across both orientations.

Disabling orientations isn't a great solution, but sometimes it's the right solution. Most games, for example, fix their orientation because it just doesn't make sense to support both. If you want to do this, press `Cmd + 1` to show the project navigator on the left of your `Xcode` window, select your project (it's the first item in the pane), then to the right of where you just clicked will appear another pane showing `“PROJECT”` and `“TARGETS”`, along with some more information in the center.

`Please note`: This project and targets list can be hidden by clicking the disclosure button in the top-left of the project editor (directly beneath the icon with four squares), and you may find yours is already hidden. I strongly recommend you show this list – hiding it will only make things harder to find, so please make sure it's visible!

In the picture below you can see the project editor, with the device orientations at the bottom. This is the collapsed view of projects and targets, so there's a dropdown arrow at the top that says `“Project2”` (just above where it says Identity in bold), and to the left of that is the button to show the projects and targets list.



This view is called the project editor, and contains a huge number of options that affect the way your app works. You'll be using this a lot in the future, so remember how to get here! Select Project 2 under `TARGETS`, then choose the `General` tab, and scroll down until you see four checkboxes called `Device Orientation`. You can select only the ones you want to support.

You'll need to support selective orientations in some later projects, but for now let's take the smart solution: add extra rules to `Auto Layout` so it can make the layout work great in landscape mode.

Open `Main.storyboard` in `Interface Builder`, select the bottom flag, then `Ctrl-drag` from the flag to the white space directly below the flag – in the view controller itself. The direction you drag is important, so please drag straight down.

When you release your mouse button, a popup will appear that includes the option “Bottom Space to Bottom Layout Guide” – please select that. This creates a new Auto Layout constraint that the bottom of the flag must be at least X points away from the bottom of the view controller, where X is equal to whatever space there is in there now.

Although this is a valid rule, it will screw up your layout because we now have a complete set of exact vertical rules: the top flag should be 36 points from the top, the second 30 from the first, the third 30 from the second, and the third X away from the bottom. It's 140 for me, but yours might be different.

Because we've told Auto Layout exactly how big all the spaces should be, it will add them up and divide the remaining space among the three flags however it thinks best. That is, the flags must now be stretched vertically in order to fill the space, which is almost certainly what we don't want.

Instead, we're going to tell Auto Layout where there is some flexibility, and that's in the new bottom rule we just created. The bottom flag doesn't need to be 140 points away from the bottom layout guide – it just needs to be some distance away, so that it doesn't touch the edge. If there is more space, great, Auto Layout should use it, but all we care about is the minimum.

Select the third flag to see its list of constraints drawn in blue, then (carefully!) select the bottom constraint we just added. In the utilities view on the right, choose the attributes inspector (Alt + Cmd + 4), and you should see Relation set to Equal and Constant set to 140 (or some other value, depending on your layout).

What you need to do is change Equal to be “Greater Than or Equal”, then change the Constant value to be 20. This sets the rule “make it at

least 20, but you can make it more to fill up space". The layout won't change visually while you're doing this, because the end result is the same. But at least now that Auto Layout knows it has some flexibility beyond just stretching the flags!

Our problem is still not fixed, though: in landscape, and iPhone 4s has just 320 points of space to work with, so Auto Layout is going to make our flags fit by squashing one or maybe even two of them. Squashed flags aren't good, and having uneven sizes of flags isn't good either, so we're going to add some more rules.

Select the second button, then Ctrl–drag to the first button. When given the list of options, choose Equal Heights. Now do the same from the third button to the second button. This rule ensures that at all times the three flags have the same height, so Auto Layout can no longer squash one button to make it all fit and instead has to squash all three equally.

That fixes part of the problem, but in some respects it has made things worse. Rather than having one squashed flag, we now have three! But with one more rule, we can stop the flags from being squashed ever. Select the first button, then Ctrl–drag a little bit upwards – but stay within the button! When you release your mouse button, you'll see the option "Aspect Ratio", so please choose it.

The Aspect Ratio constraint solves the squashing once and for all: it means that if Auto Layout is forced to reduce the height of the flag, it will reduce its width by the same proportion, meaning that the flag will always look correct. Add the Aspect Ratio constraint to the other two flags, and run your app again. It should work great in portrait and landscape, all thanks to Auto Layout!

Auto Layout in code: addConstraints

Create a new Single View Application project in Xcode, name it Project6b and set its target to be iPhone. We're going to create some views by hand, then position them using Auto Layout. Put this into your viewDidLoad() method:

```
override func viewDidLoad() {
    super.viewDidLoad()

    let label1 = UILabel()
    label1.translatesAutoresizingMaskIntoConstraints = false
    label1.backgroundColor = UIColor.redColor()
    label1.text = "THESE"

    let label2 = UILabel()
    label2.translatesAutoresizingMaskIntoConstraints = false
    label2.backgroundColor = UIColor.cyanColor()
    label2.text = "ARE"

    let label3 = UILabel()
    label3.translatesAutoresizingMaskIntoConstraints = false
    label3.backgroundColor = UIColor.yellowColor()
    label3.text = "SOME"

    let label4 = UILabel()
    label4.translatesAutoresizingMaskIntoConstraints = false
    label4.backgroundColor = UIColor.greenColor()
    label4.text = "AWESOME"

    let label5 = UILabel()
```

```
        label5.translatesAutoresizingMaskIntoConstraints = false
        label5.backgroundColor = UIColor.orangeColor()
        label5.text = "LABELS"

        view.addSubview(label1)
        view.addSubview(label2)
        view.addSubview(label3)
        view.addSubview(label4)
        view.addSubview(label5)
}
```

Put to one side what that code does for a moment, please add this method somewhere after `viewDidLoad()`:

```
override func prefersStatusBarHidden() -> Bool {
    return true
}
```

That tells `iOS` we don't want to show the `iOS` status bar on this view controller – that's the bit that tells you what time it is.

OK, back to `viewDidLoad()`: all that code creates five `UILabel` objects, each with unique text and a unique background color. All five views then get added to the view belonging to our view controller by using `view.addSubview()`. We also set the property `translatesAutoresizingMaskIntoConstraints` to be false on each label, because by default `iOS` generates `Auto Layout` constraints for you based on a view's size and position. We'll be doing it by hand, so we need to disable this feature.

If you run the app now, you'll see seem some colorful labels at the top, overlapping so it looks like it says "LABELS ME". That's because our labels are placed in their default position (at the top-left of the screen) and are all sized to fit their content.

We're going to do is add some constraints that say each label should start at the left edge of its superview, and end at the right edge. We're going to do this using a technique called Auto Layout Visual Format Language (VFL), which is kind of like a way of drawing the layout you want with a series of keyboard symbols. Before we do that, we need to create a dictionary of the views we want to lay out. This is a new data type, but it's quite easy to understand. You've already met arrays, which hold values that can be read using their order, for example myArray[0] reads the first item from the array. Rather than making you use numbers, dictionaries let you specify any object as the access method (known as the "key"), for example you could read out via a string: myDictionary["name"].

The reason this is needed for VFL will become clear shortly, but first here's the dictionary you need to add below the last call to addSubview():

```
let viewsDictionary = ["label1": label1, "label2": label2,
"label3": label3, "label4": label4, "label5": label5]
```

That creates a dictionary with strings (the keys) and our UILabels as its values (the values). So, to get access to label1, we can now use viewsDictionary["label1"]. This might seem redundant, but wait just a moment longer: it's time for some Visual Format Language!

Add these lines directly below the viewsDictionary that was just created:

```
view.addConstraints(NSLayoutConstraint.constraintsWithVisualF
ormat("H:|[label1]|", options: [], metrics: nil, views:
viewsDictionary))
view.addConstraints(NSLayoutConstraint.constraintsWithVisualF
ormat("H:|[label2]|", options: [], metrics: nil, views:
viewsDictionary))
view.addConstraints(NSLayoutConstraint.constraintsWithVisualF
ormat("H:|[label3]|", options: [], metrics: nil, views:
viewsDictionary))
view.addConstraints(NSLayoutConstraint.constraintsWithVisualF
ormat("H:|[label4]|", options: [], metrics: nil, views:
viewsDictionary))
view.addConstraints(NSLayoutConstraint.constraintsWithVisualF
ormat("H:|[label5]|", options: [], metrics: nil, views:
viewsDictionary))
```

That's a lot of code, but actually it's just the same thing five times over. As a result, we could easily rewrite those fix in a loop, like this:

```
for label in viewsDictionary.keys {

view.addConstraints(NSLayoutConstraint.constraintsWithVisualF
ormat("H:|[\(label)]|", options: [], metrics: nil, views:
viewsDictionary))
}
```

Note that we're using string interpolation to put the key (“`label1`”, etc) into the VFL.

Let's eliminate the easy stuff, then focus on what remains.

- `view.addConstraints()`: this adds an array of constraints to our view controller's view. This array is used rather than a single constraint because `VFL` can generate multiple constraints at a time.
- `NSLayoutConstraint.constraintsWithVisualFormat()` is the `Auto Layout` method that converts `VFL` into an array of constraints. It accepts lots of parameters, but the important ones are the first and last.
- We pass `[]` (an empty array) for the options parameter and `nil` for the metrics parameter. You can use these options to customise the meaning of the `VFL`, but for now we don't care.

That's the easy stuff. So, let's look at the `Visual Format Language` itself: "`H:|[label1]|`". As you can see it's a string, and that string describes how we want the layout to look. That `VFL` gets converted into `Auto Layout` constraints, then added to the view.

The `H:` parts means that we're defining a horizontal layout; we'll do a vertical layout soon. The pipe symbol, `|`, means "`the edge of the view`". We're adding these constraints to the main view inside our view controller, so this effectively means "`the edge of the view controller`". Finally, we have `[label1]`, which is a visual way of saying "`put label1 here`". Imagine the brackets, `[` and `]`, are the edges of the view.

So, "`H:|[label1]|`" means "`horizontally, I want my label1 to go edge to edge in my view`". But there's a hiccup: what is "`label1`"? Sure, we know what it is because it's the name of our variable, but variable names are just things for humans to read and write – the variable names aren't actually saved and used when the program runs.

This is where our `viewsDictionary` dictionary comes in: we used strings for the key and `UILabels` for the value, then set `"label1"` to be our label. This dictionary gets passed in along with the `VFL`, and gets used by `iOS` to look up the names from the `VFL`. So when it sees `[label1]`, it looks in our dictionary for the `"label1"` key and uses its value to generate the `Auto Layout` constraints.

That's the entire `VFL` line explained: each of our labels should stretch edge-to-edge in our view. If you run the program now, that's sort of what you'll see, although it highlights our second problem: we don't have a vertical layout in place, so although all the labels sit edge-to-edge in the view, they all overlap.

We're going to fix this with another set of constraints, but this time it's just one (long) line.

```
view.addConstraints(NSLayoutConstraint.constraintsWithVisualF
ormat("V:|[label1]-[label2]-[label3]-[label4]-[label5]",
options: [], metrics: nil, views: viewsDictionary))
```

That's identical to the previous five, except for the `VFL` part. This time we're specifying `V:`, meaning that these constraints are vertical. And we have multiple views inside the `VFL`, so lots of constraints will be generated. The new thing in the `VFL` this time is the - symbol, which means `"space"`. It's `10` points by default, but you can customise it.

Note that our vertical `VFL` doesn't have a pipe at the end, so we're not forcing the last label to stretch all the way to the edge of our view. This will leave whitespace after the last label, which is what we want right now.

If you run your program now, you'll see all five labels stretching edge-to-edge horizontally, then spaced neatly vertically. It would have taken quite a lot of `Ctrl-`

dragging in Interface Builder to make this same layout, so I hope you can appreciate how powerful VFL is!

Auto Layout metrics and priorities: constraintsWithVisualFormat

We have a working layout now, but it's quite basic: the labels aren't very high, and without a rule regarding the bottom of the last label it's possible the views might be pushed off the bottom edge.

To begin to fix this problem, we're going to add a constraint for the bottom edge saying that the bottom of our last label must be at least 10 points away from the bottom of the view controller's view. We're also going to tell Auto Layout that we want each of the five labels to be 88 points high. Replace the previous vertical constraints with this:

```
view.addConstraints(NSLayoutConstraint.constraintsWithVisualF
ormat("V:|[label1(==88)]-[label2(==88)]-[label3(==88)]-
[label4(==88)]-[label5(==88)]-(>=10)-|", options: [],
metrics: nil, views: viewsDictionary))
```

The difference here is that we now have numbers inside parentheses: (==88) for the labels, and (>=10) for the space to the bottom. Note that when specifying the size of a space, you need to use the - before and after the size: a simple space, -, becomes -(>=10)-.

We are specifying two kinds of size here: == and >=. The first means "exactly equal" and the second "greater than or equal to". So, our labels will be forced to be an exact size, and we ensure that there's some space at the bottom

while also making it flexible – it will definitely be at least `10` points, but could be `100` or more depending on the situation.

Actually, wait a minute. I didn't want `88` points for the label size, I meant `80` points. Go ahead and change all the labels to `80` points high.

Whoa there! It looks like you just received an email from your IT director: he thinks `80` points is a silly size for the labels; they need to be `64` points, because all good sizes are a power of `2`.

And now it looks like your designer and IT director are having a fight about the right size. A few punches later, they decide to split the difference and go for a number in the middle: `72`. So please go ahead and make the labels all `72` points high.

Bored yet? You ought to be. And yet this is the kind of pixel-pushing it's easy to fall into, particularly if your app is being designed by committee.

`Auto Layout` has a solution, and it's called metrics. All these calls to `constraintsWithVisualFormat()` have been sent nil for their metrics parameter, but that's about to change. You see, you can give `VFL` a set of sizes with names, then use those sizes in the `VFL` rather than hard-coding numbers. For example, we wanted our label height to be `88`, so we could create a metrics dictionary like this:

```
let metrics = ["labelHeight": 88]
```

Then, whenever we had previously written `==88`, we can now just write `labelHeight`. So, change your current vertical constraints to be this:

```
view.addConstraints(NSLayoutConstraint.constraintsWithVisualFormat("V:|[label1(labelHeight)]-[label2(labelHeight)]-[label3(labelHeight)]-[label4(labelHeight)]-[label5(labelHeight)]->=10-|", options: [], metrics: ["labelHeight": 88], views: viewsDictionary))
```

So when your designer / manager / inner-pedant decides that `88` points is wrong and you want some other number, you can change it in one place to have everything update.

Before we're done, we're going to make one more change that makes the whole user interface much better, because right now it's still imperfect. To be more specific, we're forcing all labels to be a particular height, then adding constraints to the top and bottom. This still works fine in portrait, but in landscape you're unlikely to have enough room to satisfy all the constraints.

With our current configuration, you'll see this message when the app is rotated to landscape: `“Unable to simultaneously satisfy constraints”`. This means your constraints simply don't work given how much screen space there is, and that's where priority comes in. You can give any layout constraint a priority, and `Auto Layout` will do its best to make it work.

Constraint priority is a value between `1` and `1000`, where `1000` means `“this is absolutely required”` and anything less is optional. By default, all constraints you have are priority `1000`, so `Auto Layout` will fail to find a solution in our current layout. But if we make the height optional – even as high as priority `999` – it means `Auto Layout` can find a solution to our layout: shrink the labels to make them fit.

It's important to understand that Auto Layout doesn't just discard rules it can't meet – it still does its best to meet them. So in our case, if we make our 88-point height optional, Auto Layout might make them 78 or some other number. That is, it will still do its best to make them as close to 88 as possible. TL;DR: constraints are evaluated from highest priority down to lowest, but all are taken into account.

So, we're going to make the label height have priority 999 (i.e., very important, but not required). But we're also going to make one other change, which is to tell Auto Layout that we want all the labels to have the same height. This is important, because if all of them have optional heights using labelHeight, Auto Layout might solve the layout by shrinking one label and making another 88.

From its point of view it has at least managed to make some of the labels 88, so it's probably quite pleased with itself, but it makes our user interface look uneven. So, we're going to make the first label use labelHeight at a priority of 999, then have the other labels adopt the same height as the first label. Here's the new VFL line:

```
“V:|[label1(labelHeight@999)]-[label2(label1)]-
[label3(label1)]-[label4(label1)]-[label5(label1)]->=10-|"
```

It's the @999 that assigns priority to a given constraint, and using (label1) for the sizes of the other labels is what tells Auto Layout to make them the same height.

That's it: your Auto Layout configuration is complete, and the app can now be run safely in portrait and landscape.

Wrap up

There are two types of iOS developer in the world: those who use Auto Layout, and fools. It has bit of a steep learning curve (and we didn't even use the hard way of adding constraints!), but it's an extremely expressive way of creating great layouts that adapt themselves automatically to whatever device they find themselves running on – now and in the future.

Most people recommend you do as much as you can inside Interface Builder, and with good reason – you can drag lines about until you're happy, you get an instant preview of how it all looks, and it will warn you if there's a problem (and help you fix it.) But, as you've seen, creating constraints in code is remarkably easy thanks to the Visual Format language, so you might find yourself mixing the two to get the best results.