Autolayout for everyone

My (one way) journey into land of autolayout

Krzysztof Kucharewicz
krix@mobilejazz.cat
@mobiosis

Logroño, 18/09/2014

Welcome everyone.

My name is Krzysztof, I’m a mobile software developer at MobileJazz. I’m happy to be with you today and I will speak about auto layout in practice.

What I will talk about

- A short history of Autolayouts
- How to stop using Frames
- How to adapt to screen orientation and size
- Autolayout by code and in IB
- Useful libraries (Lyt, NibWrapper)
- Animating with constraints
- Debugging autolayout

Let me give you a short introduction first

I will tell you where auto layouts come from and how they evolved.

I will show you how to transform a Frame-based layout into a fully-automatic AutoLayout.

I will also give you an idea how you can create common rules for layouts that will work equally well in different screen sizes, specifically when rotating screen.

We will see some layouts created by code, others in Interface Builder.

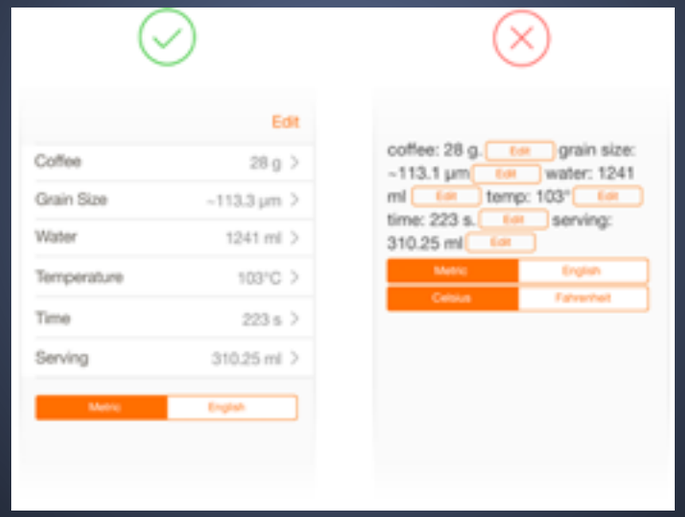
I will introduce two very helpful (almost required) libraries for day-to-day work with AutoLayouts

We will do some animations

And we will debug broken layouts

What I will not talk about

How to create a great UX



https://developer.apple.com/app-store/review/rejections/

I also wont talk about how to create a great User Experience in your app, because you should have learned it by now

What I will not talk about

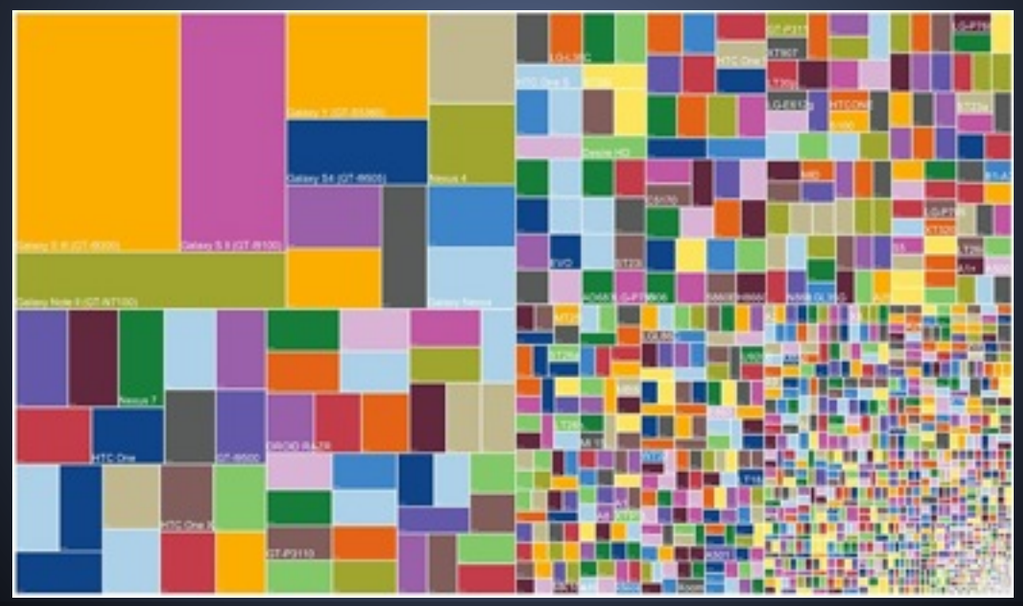
How to make yourselves rich

http://www.bet.com/news/national/2013/01/08/american-money-250k-a-year-rich-or-not-rich.html

I won’t tell you how to get rich on the App Store either

What I will not talk about

Other ecosystems' fragmentation



http://www.fastcodesign.com/1673162/the-disintegration-of-android-visualized

And I won't even mention how fragmented in terms of device sizes and variants other ecosystems are…

(pause 5s)

History of Autolayouts

- Autolayouts were introduced in iOS 6 (OS X 10.7)
- They are a way to move from **Imperative** to **Declarative** interfaces
- Support for Autolayouts in Xcode has gone through various iterations, where Xcode 4.6 was not the best one to start with
- They are still a **taboo** for many programmers

Autolaouts were only introduced only 3 years ago on Mac OS X

and even 1 year later on iOS

They fulfil the need to create Declarative interfaces where we say what relations UI elements have, instead of saying exactly where they are.

If you heard bad opinions of Autolayouts from someone using Xcode 4.6, you should give it a try again. It changed a whole lot since then.

If you are afraid to talk about it, you’re in the right room right now.

Why autolayouts?

1. No more setFrame
- autolayout uses declarative interfaces
- frame is calculated each time based on constraints
- invoke the layout by calling
  [view layoutIfNeeded]
  [view setNeedsLayout]
- you can combine the frame-based layout with autolayout (no need to switch to autolayout right away)

There are some great reasons why you should use Auto Layouts

Frames are in the past since we have multiple resolutions right now.

Of course Apple did their best to make “fixed” interfaces work, but for the moment their logic behind it creates only confusion

http://stackoverflow.com/questions/25755443/iphone-6-plus-resolution-confusion-xcode-or-apples-website

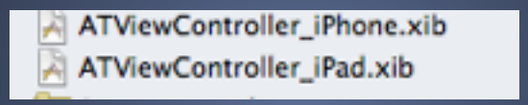
Thus it’s better to use AutoLayout and Constraints

Once you create all the constraints, you need to remember to let the system know it has to recalculate all the views

You don’t need to switch to AutoLayout right away though. Even if your Xib file doesn’t have the “Use Auto Layout” ticked, you can add sub-views using AutoLayout internally

Why autolayouts?

2. One layout fits all



3. Translations
4. New form factors
5. Because challenges are fun...

If your design is not very different between iPhone and iPad, you can have it all in one scalable layout file

AutoLayout adapts very well to different text lengths so you’re less likely to break it with too long strings.

Obviously it all comes down to the fact that there is not only one iPhone, so the logic behind the view positioning would get too complex if done manually.

And of course if you have a R&D spirit, you might like it for the sole fact of having to learn something new.

Creating autolayouts

Interface builder

- helps creating all the constraints for you
- will show error on insufficient layouts
- will show warning in case of ambiguity
- layouts created automatically can be pretty useless when not attended in detail

Interface builder become a very powerful layout modelling tool by now so it might likely be first thing you do when you jump into AutoLayout land.

It will help you debug basic relations and show ERRORS

Or WARNINGS when your layouts stop being consistent

But don’t use “Add missing constraints” to make your life easier, as it probably won’t do much better than setting the fixed rectangle for each view

Creating autolayouts

By code

- ASCII art
  - V: is an axis, (together with H:)
  - | is a bound
  - [item] is a view
  - (==100) is a fixed size constraint,
    might also be >= or <=
  - @123 is a priority
  - - is a spacer (recommended size by default)

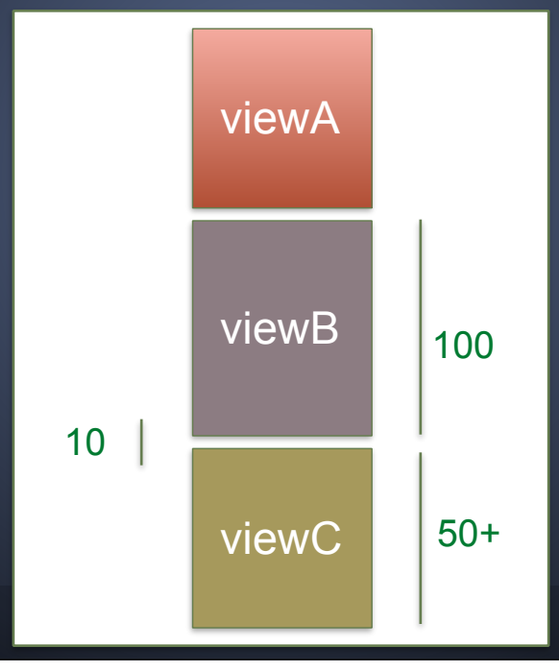
We can also do it all by code, and even though it loses all transparency of view relations and positioning, it might be sometimes required for things that you cannot easily do in IB or when you're animating views.

[START]

So creating auto-layouts can be done semi-visually with the "ascii art"

Creating autolayouts

i.e. "V:|[viewA]-[viewB(==100)]-(==10)-[viewC(>=50@900)]-|"



This relations means we want to order views vertically, starting from the edge, viewA without a size specifier, default spacer, viewB with height of 100, spacer of size 10, and viewC with height of 50 or more but can be shrunk if other views don't have enough space (priority 900)

Creating autolayouts

By code

- With a long method

```
NSLayoutConstraint *con1 = [NSLayoutConstraint
    constraintWithItem:self.imageLock
                 attribute:NSLayoutAttributeBottom
               relatedBy:NSLayoutRelationGreaterThanOrEqual
                  toItem:self.labelStatus
                 attribute:NSLayoutAttributeBaseline
                multiplier:1
                 constant:3];
con1.priority = 100
```

- Which represents the equation

  "view1.attr1 <relation> view2.attr2 * multiplier + constant"

We can also use this long method which seems pretty cumbersome, but is much more powerful when it comes to specifying complex relations.

Priorities

- Range from 0 (ignored?) to 1000 (required)
- Have some special meanings (see in IB)
- Content Hugging Priority (wraps around the content tightly when high)
- Compression Resistance (avoids shrinking/ shortening when high)

Priorities have values of 0 to 1000

They can have some intermediate values which are set by default for some parameters

We have a Hugging Priority which is making views compact

We have Compression Resistance which avoids shrinking views too early

Example

- Calculating layout with frames is cumbersome

```
CGRect screenRect = [UIScreen mainScreen].bounds;
float topMargin = 70;
float labelWidthBase = screenRect.size.width;

UILabel* label = [UILabel new];
label.textAlignment = NSTextAlignmentRight;
label.text = NSLocalizedString(@"Hello", @"");
CGSize fitSize = [label sizeThatFits:CGSizeMake(labelWidthBase, 15)];
label.frame = CGRectMake(8,topMargin, fitSize.width, fitSize.height);
label.backgroundColor = [UIColor lightGrayColor];
```

Manual layouts require quite a lot of logic

We need to take the device width, calculate how much would the label occupy to align it with other items.

Example

- Calculating layout with frames is cumbersome

```
float wrapperViewWidth = label.frame.origin.x + label3.frame.size.width + 8;
float viewAlignLeft = 8;
float viewAlignRight = screenRect.size.width - wrapperViewWidth;
float viewAlignCenter = screenRect.size.width/2-wrapperViewWidth/2;
wrapperView.frame = CGRectMake(viewAlignCenter, 0, wrapperViewWidth, 15);
[self.view addSubview:wrapperView];
```

It’s much less transparent.

And it’s easy to make a mistake.

Example

- Using Auto Layouts give much more transparency

```
UILabel* label3 = [UILabel new];
label3.translatesAutoresizingMaskIntoConstraints = NO;
label3.text = NSLocalizedString(@"@mobiosis", @"");
label3.backgroundColor = [UIColor redColor];
[view addSubview:label3];

[view addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"H:|-
 [label]-(8)-[label2]-(8)-[labelPlus]-(8)-[label3]-|" options:0 metrics:nil views:
 NSDictionaryOfVariableBindings(label, label2, labelPlus, label3)]];
```

When using auto layouts, we only care about content, and the relations with parent and sibilings

Example

- Even though it might require more lines of code

```
int margin = 80;
[view addConstraint:[NSLayoutConstraint constraintWithItem:label attribute:
 NSLayoutAttributeTop relatedBy:NSLayoutRelationEqual toItem:self.topLayoutGuide
 attribute:NSLayoutAttributeBottom multiplier:1 constant:margin]];
[view addConstraint:[NSLayoutConstraint constraintWithItem:label2 attribute:
 NSLayoutAttributeTop relatedBy:NSLayoutRelationEqual toItem:self.topLayoutGuide
 attribute:NSLayoutAttributeBottom multiplier:1 constant:margin]];
[view addConstraint:[NSLayoutConstraint constraintWithItem:label3 attribute:
 NSLayoutAttributeTop relatedBy:NSLayoutRelationEqual toItem:self.topLayoutGuide
 attribute:NSLayoutAttributeBottom multiplier:1 constant:margin]];
[view addConstraint:[NSLayoutConstraint constraintWithItem:labelPlus attribute:
 NSLayoutAttributeTop relatedBy:NSLayoutRelationEqual toItem:self.topLayoutGuide
 attribute:NSLayoutAttributeBottom multiplier:1 constant:margin]];
```

At times it might require a few extra lines of code though

We need to add these constraints separately to align each item vertically.

Demo

(3min)

Code example

- show multiple objects aligned in line
- rotate the screen
- mention the lack of alignment
- vertical displacement is broken
- autolayout recalculates the constraints when needed

New in iOS8 and Xcode 6

- View margins independent of the layout constraints

  UIView.layoutMargins
  UIView.preservesSuperviewLayoutMargins

- Activating constraints on-demand

  NSLayoutConstraint.active

We are able to set the view margins independently of the constraints of the items assigned to subviews

That works pretty much like in CSS or XML declarative interfaces

We also don’t need to remove constraints when not needed, but de-activate them that effectively is omitted when calculating layout.

Animations

- Animations are easy to achieve with constraints

```
UIView animateWithDuration:1.5
animations:^{
    //remove all, add new constraints here
    [view removeConstraint:self.topConstraint];
    [view addConstraint: [NSLayoutConstrait … ]];
    //when finished, request UI to lay it out
    [view layoutIfNeeded];
}
```

- You have to remember to call layoutIfNeeded after adding all constraints in the animation block

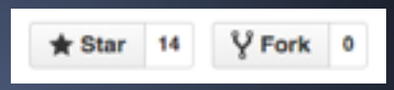
Animations are pretty straightforward

You only need to remove old layout and add a new layout in the animation block

Don’t forget to call “layoutIfNeeded” after adding all constraints in the animation block

Popular Auto Layout libraries

AutoLayoutKit



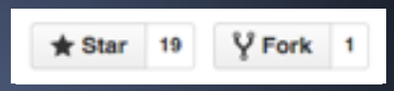

```
[ALKConstraints layout:childView do:^(ALKConstraints *c) {
  [c make:ALKCenterX equalTo:self s:ALKCenterX];
  [c make:ALKCenterY equalTo:self s:ALKCenterY];
  [c set:ALKWidth to:30.f];
  [c set:ALKHeight to:30.f];
}];
```

I’ve chosen the most popular Auto Layout libraries from CocoaPods and here are the examples

This one decided to create it’s own Enums which surely complicate the casual usage

Popular Auto Layout libraries

DRAutolayout

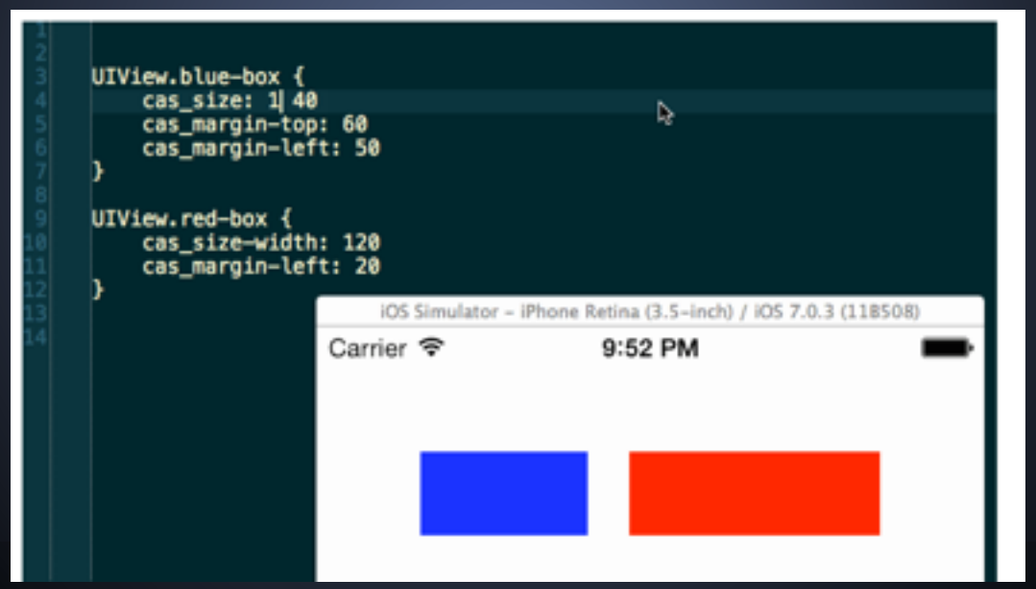



```
+(NSLayoutConstraint*)top:(id)view1 toTop:(id)view2 multiplier:(CGFloat)multiplier cons
+(NSLayoutConstraint*)bottom:(id)view1 toBottom:(id)view2 multiplier:(CGFloat)multiplie
+(NSLayoutConstraint*)left:(id)view1 toLeft:(id)view2 multiplier:(CGFloat)multiplier co
+(NSLayoutConstraint*)right:(id)view1 toRight:(id)view2 multiplier:(CGFloat)multiplier
+(NSLayoutConstraint*)width:(id)view1 toWidth:(id)view2 multiplier:(CGFloat)multiplier
+(NSLayoutConstraint*)height:(id)view1 toHeight:(id)view2 multiplier:(CGFloat)multiplie
+(NSLayoutConstraint*)height:(id)view1 toWidth:(id)view2 multiplier:(CGFloat)multiplier
+(NSLayoutConstraint*)width:(id)view1 toHeight:(id)view2 multiplier:(CGFloat)multiplier
```

This one decided to replace very long methods with just long methods. Not too much improvement, huh?

Popular Auto Layout libraries

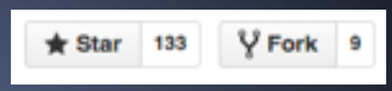
ClassyLiveLayout



This is an interesting concept, especially for prototyping, but not sure if the right choice for production apps.

Popular Auto Layout libraries

Lyt

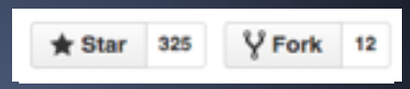



```
55      [view lyt_centerInParent
M                NSArray * lyt_centerInParent
M                NSArray * lyt_centerInParentWithMargin:(CGFloat)
M                NSArray * lyt_centerWithView:(UIView *)
M                NSArray * lyt_centerWithView:(UIView *) margin:(CGFloat)
M    NSLayoutConstraint * lyt_centerXInParent
M    NSLayoutConstraint * lyt_centerXInParentWithMargin:(CGFloat)
M    NSLayoutConstraint * lyt_centerXWithView:(UIView *)
M    NSLayoutConstraint * lyt_centerXWithView:(UIView *) margin:(CGFloat)
57
```

Lyt uses concise and compact method names for various configurations of constraints

Popular Auto Layout libraries

CompactConstraint




```
[self.view addCompactConstraints:@[
    @"emailLabel.left >= emailField.left",
    @"spinner.left = logInButton.right + 10",
    @"preview.height = preview.width / 1.6"
] metrics:metrics views:views];
```

This one is very popular, but I don’t see much point in replacing rather controversial one visual language with the other

Useful libraries

Lyt https://github.com/robotmedia/Lyt

- Shortens NSLayoutConstraint call to a concise and self-explanatory method
- Always finds a common parent for correct constraint placement
- Good for frequent use with simple relations
- Not so good for chaining multiple views
- Does not allow setting priorities
- You should avoid using it to just convert frames to constraints with lyt_setFrame:(CGRect)frame;

So my choice goes to Lyt

It can be very useful

…

but it also has it’s weak points

Useful libraries

NibWrapper https://github.com/mobilejazz/NibWrapper

For easier reusing of the UI elements

- Design a view layout in the IB (xib file)
- Create a custom controller class
- Encapsulate it with NibWrapper
- Add to your layout like if it was another UI control provided with the SDK

NibWrapper helps you reuse your custom views almost like if it was part of the Interface Builder

(3min)

We have used the Lyt library to help us create constraints quicker.

- added view
- added constraints
- animating to another position

NibWrapper

- Show ATUserProfile.xib
- Show how it is used in the Storyboard
- Rotate and explain how the image gets resized to 1/3 in ATUserProfile.m

Debugging the layouts

- Execute in the console

```
[view hasAmbiguousLayout];
[view exerciseAmbiguityInLayout];
[view constraintsAffectingLayoutForAxis:0]; //1
```

- Not available yet in iOS (only Mac OS)

```
[[UIWindow keyWindow] autoLayoutTrace];
[window visualizeConstraints];
```

We have a bunch of tools we can use when things go wrong in the layout system

Following methods are very useful

Debug the layouts

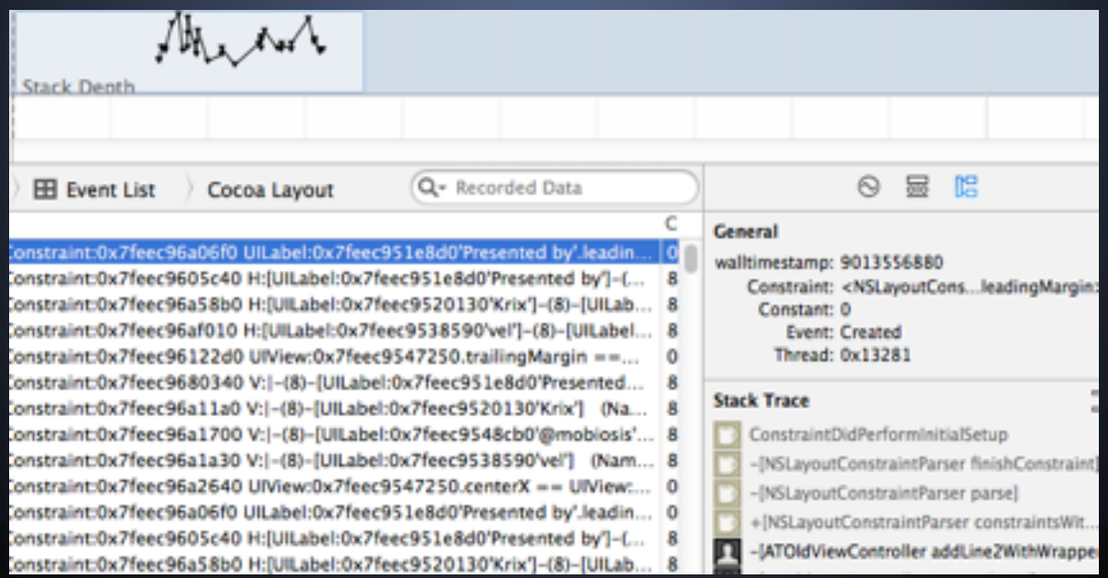
- break on objc_exception_throw
- in Scheme > Settings add to see long strings
  NSDoubleLocalizedStrings = YES;
- and other less useful variables
  AppleTextDirection = YES;
  NSForceRightToLeftWritingDirection = YES;
  NSShowViewAlignments = YES;
  UIShowViewAlignments = YES;

Additionally we can help us prevent issues by

- watching for exceptions
- setting strings to long

Debug the layouts

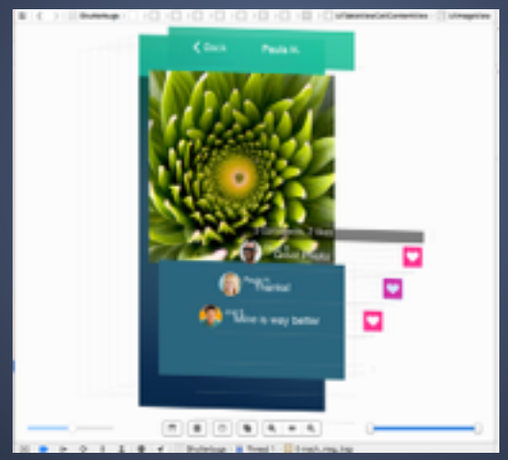
- Xcode Instruments has a useful tool **Cocoa Layout** for recording layout changes



Cocoa Layout allows us to record all layout changes (adding/removing constraints) and we can track each of them to the code that called it

Debug the layouts

- There's also new layout inspector tool in Xcode 6



- 3rd party apps **RevealApp** or **SparkInspector** provide useful insight into view during runtime

(3min)

- change the language

[START]

- break on “view did layout subviews”
- [self.line2 hasAmbiguousLayout]
- remove the “wrapper view”

[START]

- rotate and show that there is still some ambiguity
- add a “exercise the layout” button

[START]

- exercise the layout to show different behaviour on each recalculation

Run INSTRUMENTS > Cocoa Layout

- show how to record and track button added

Pro tips

- When creating UIView by code, always set
  `[view setTranslatesAutoResizingMaskIntoConstraints:NO];`
- Remove duplicate constraints before adding new
  ```
  [view removeConstraint:self.someConstraint];
  self.someConstraint = [NSLayoutConstraint ...
  [view addConstraint:self.someConstraint];
  ```
- Compression Resistance and Hugging Priority are keys to handling complex layouts

Where to find more

Coding Together

- Lecture 8 - https://itunes.apple.com/us/course/coding-together-developing/id593208016

WWDC 2012

- 202 - Introduction to Auto Layout
- 228 - Best practices for mastering Auto Layout
- 232 - Auto Layout by example

WWDC 2013

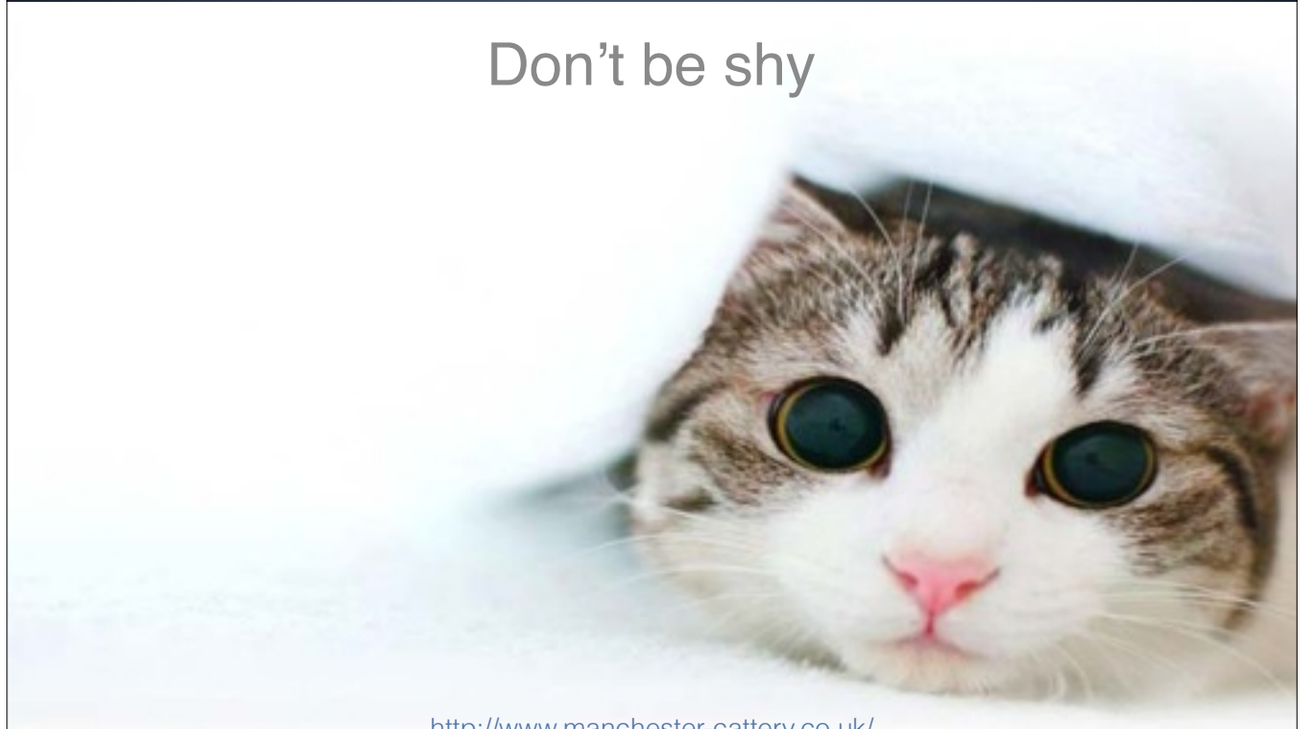
- 406 - Taking control of Autolayout in XCode 5 (...instead Autolayout taking control of XCode 4.6)

WWDC 2014

- 411 - What's new in Interface Builder

Questions?

Don’t be shy



http://www.manchester-cattery.co.uk/

Follow me

http://github.com/mobiosis

@mobiosis