MATHEW SANDERS / SATURDAY SEPTEMBER 20, 2014

Adaptive Layouts for iPhone 6

Apple has been politely suggesting that we use adaptive layouts since iOS 6, but until now I feel that people have been avoiding the topic, preferring to think mostly about fixed layouts.

With the iPhone 6 it's about to get a lot harder to avoid using adaptive layouts [1] *. With four screen sizes (five if you're supporting the iPad), three resolutions and orientations to account for it just seems easier (and smarter) to start thinking about adaptive layouts from the start of your design process.*

By the end of this post you should be comfortable with using storyboards, constraints *and* size class traits *– three tools that Apple provides in Xcode for exploring and crafting adaptive layouts like this:*

So download Xcode 6, get yourself a drink, set aside an hour and let's start!

TOPICS

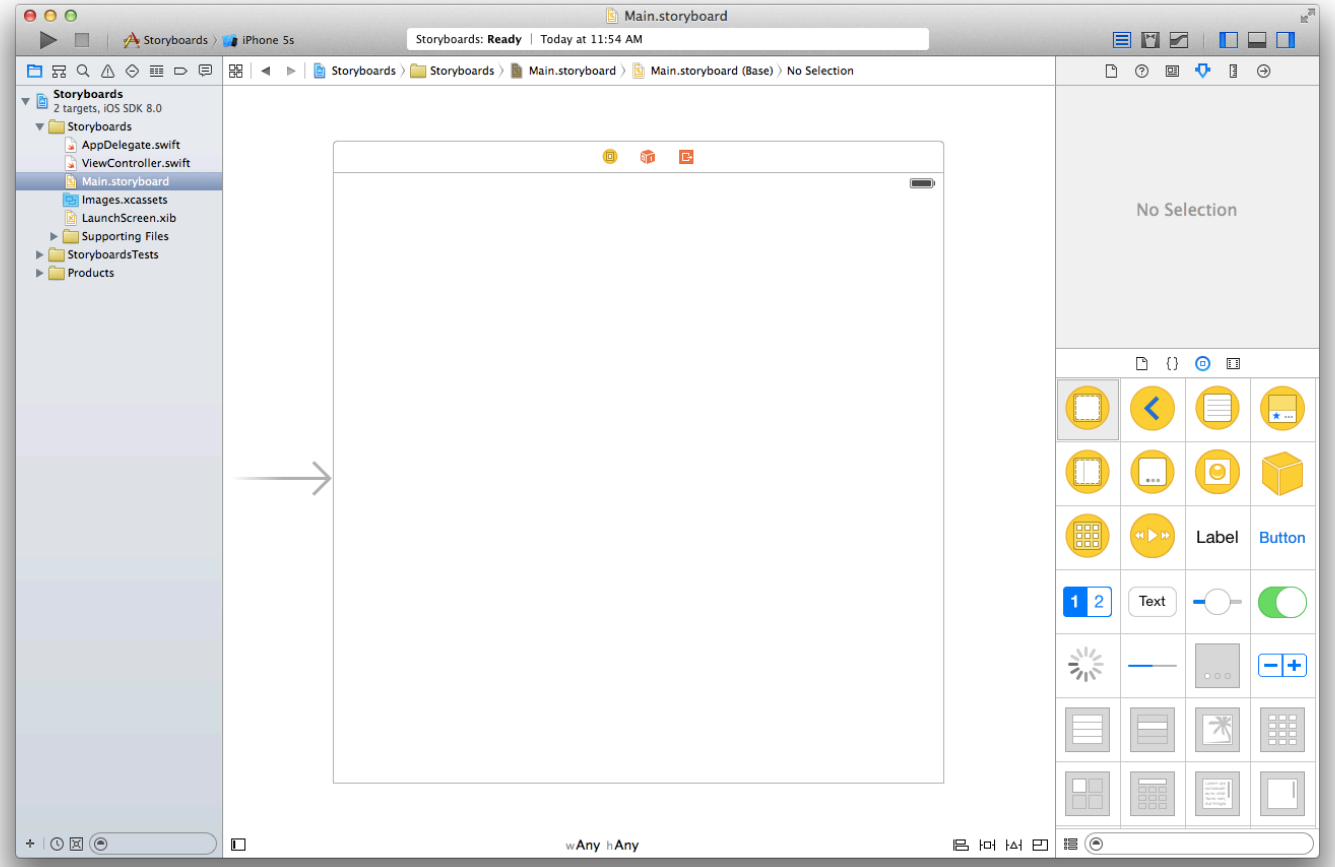
Storyboards

In Xcode, storyboards are a feature that allow you to layout app screens [2] by dragging and dropping native objects (like buttons, images, textfields and labels) onto a screen, and defining how those screens are connected with each other.

In Xcode terminology, user interface items that people can see, touch, or otherwise interact with (buttons, images, textfields, labels etc...) are called *views*.

The screens that contain and manage all those views are called *view controllers* [3] . I tend to use these terms interchangeably.

When we add a view controller to a storyboard it's aspect ratio doesn't match any particular device. In fact it's a perfect square 600pt wide and 600pt high:

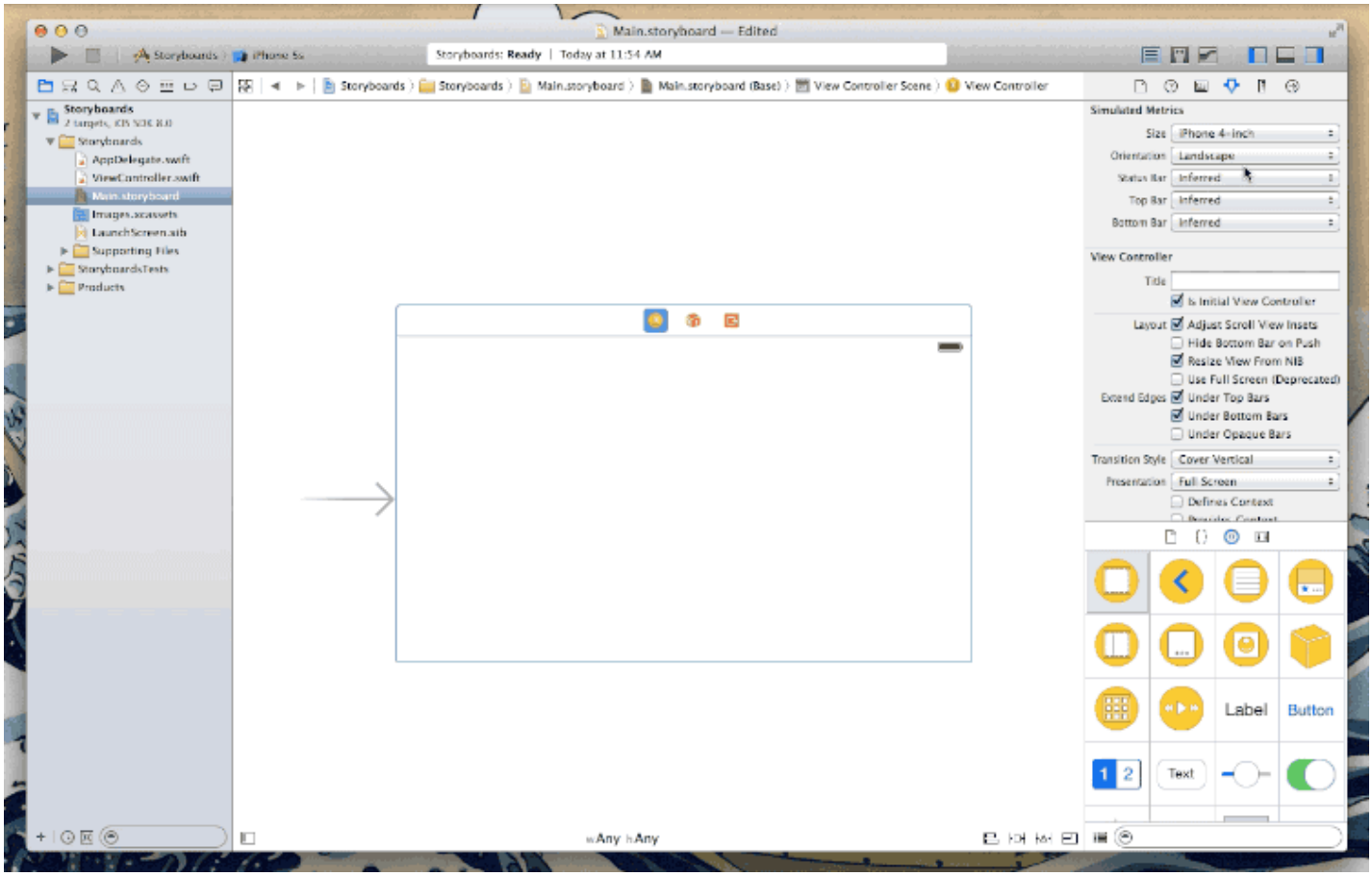


This is to remind us that a storyboard screen doesn't represent a screen on specific hardware, but instead it's an abstract representation of screens for *any* iOS device.

Simulated Sizes

You might be comfortable working with the 600×600pt canvas, but it will probably help to work with an actual size.

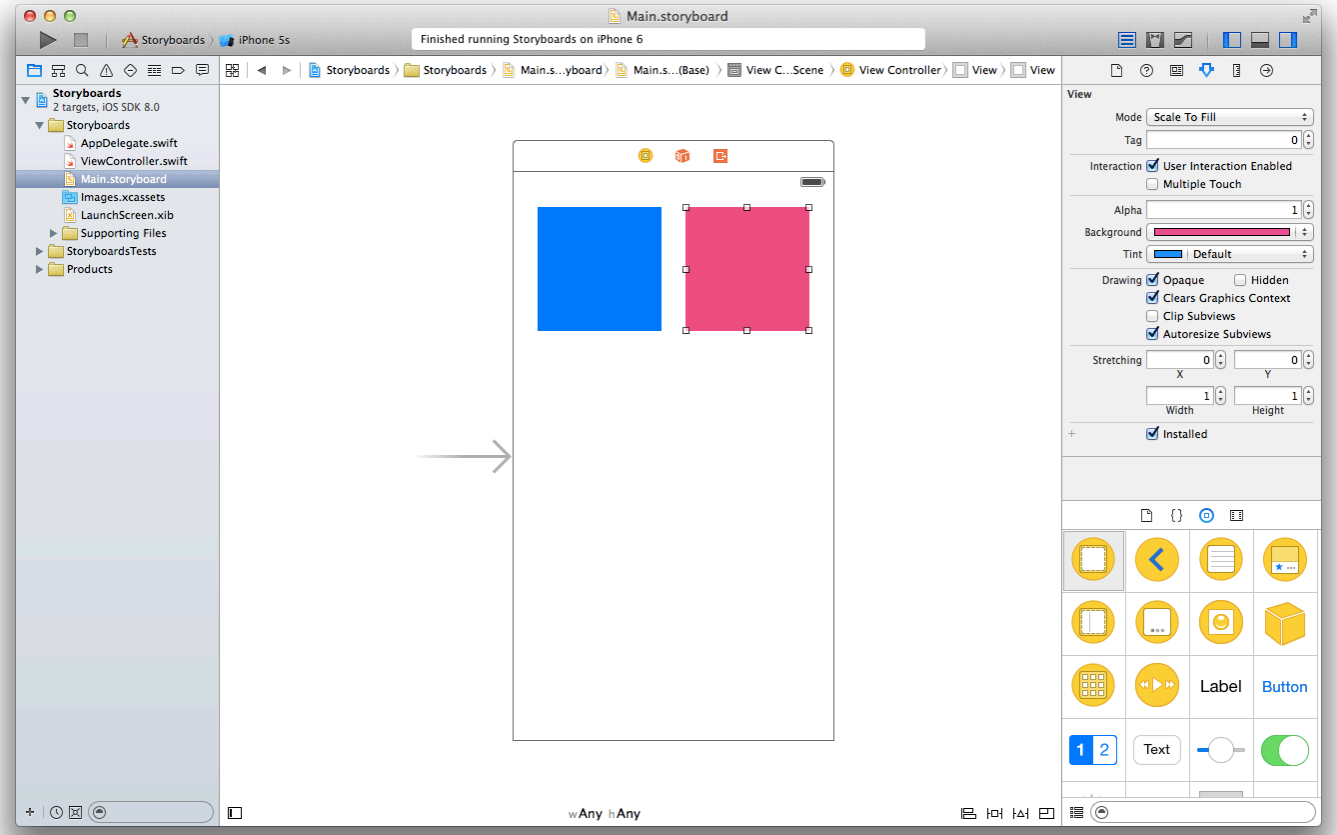
We can easily change this in Xcode by changing the view controllers simulated size and orientation [4] :



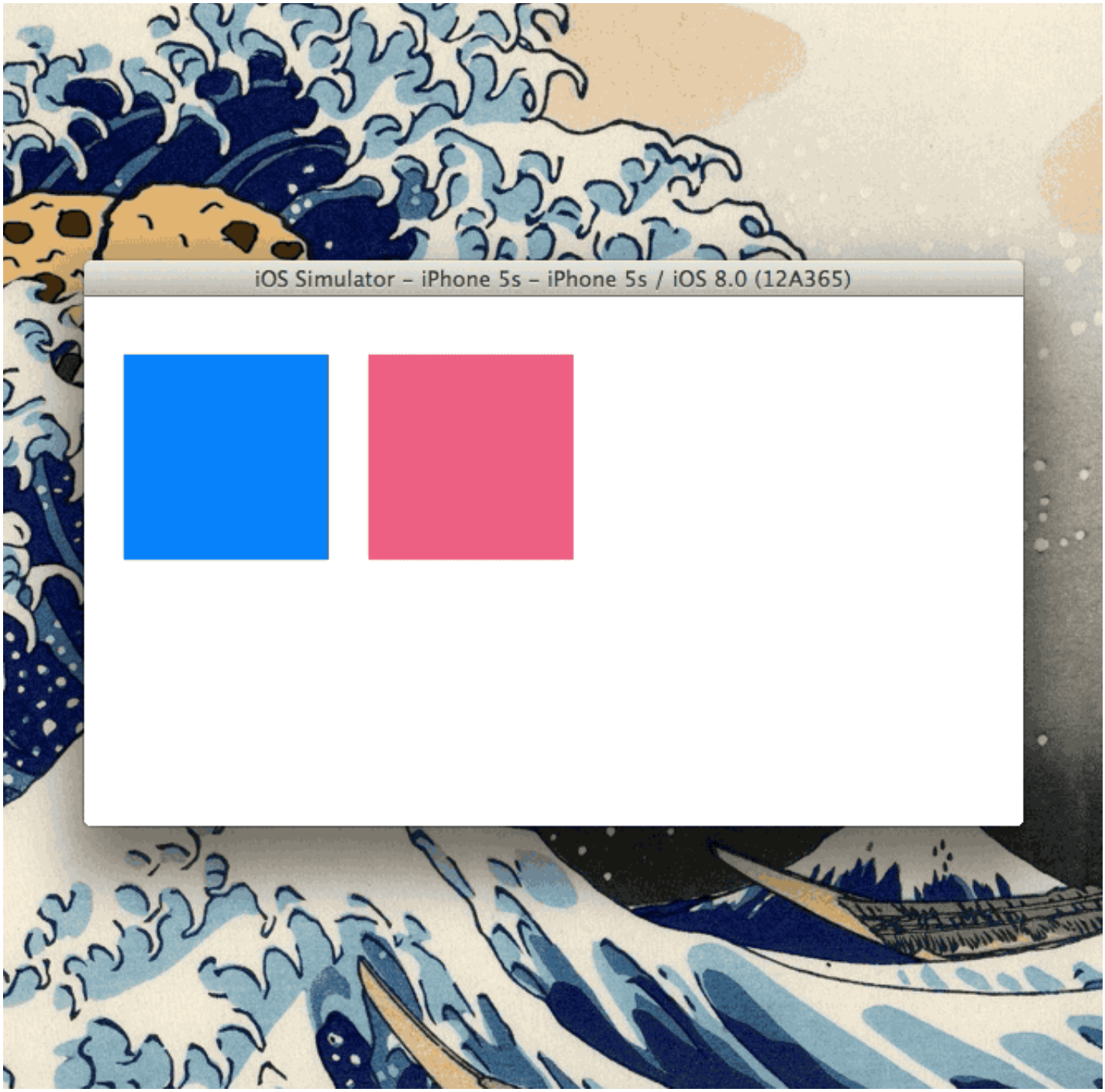
Constraints

The easiest way to introduce constraints is to show you them in action.

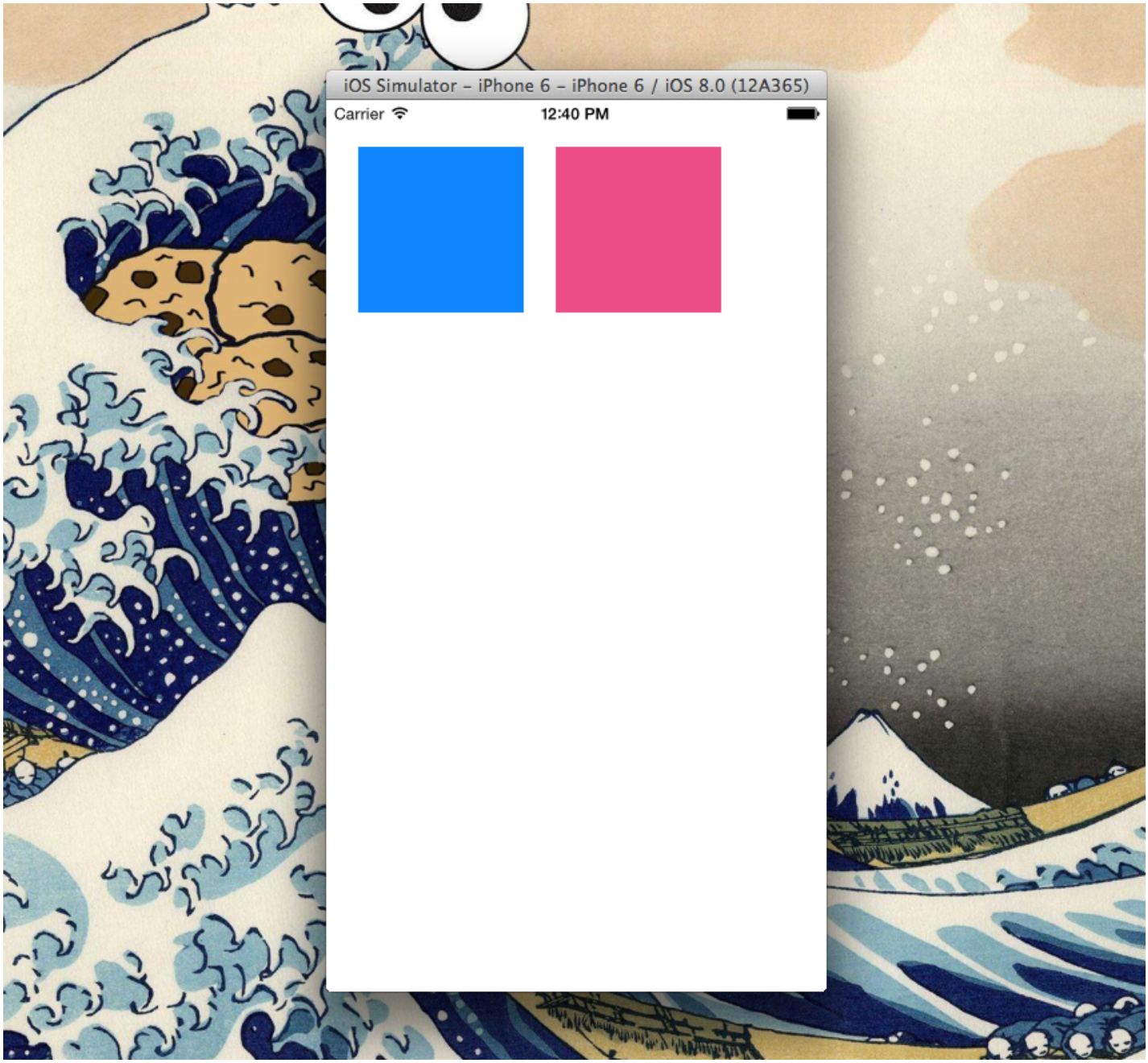
Start with a simple storyboard that contains one screen (simulated to the size of an 4-inch iPhone) and add two squares 5 positioned next to each other.



When we run the app the squares are laid out exactly where we positioned them in portrait. When we switch the orientation to landscape they remain loyal to their x,y coordinates:



And when we run the project with the iPhone 6 simulator, we see again that our layout hasn't adapted to these different screen sizes and extra space is visible on the right side of the screen.

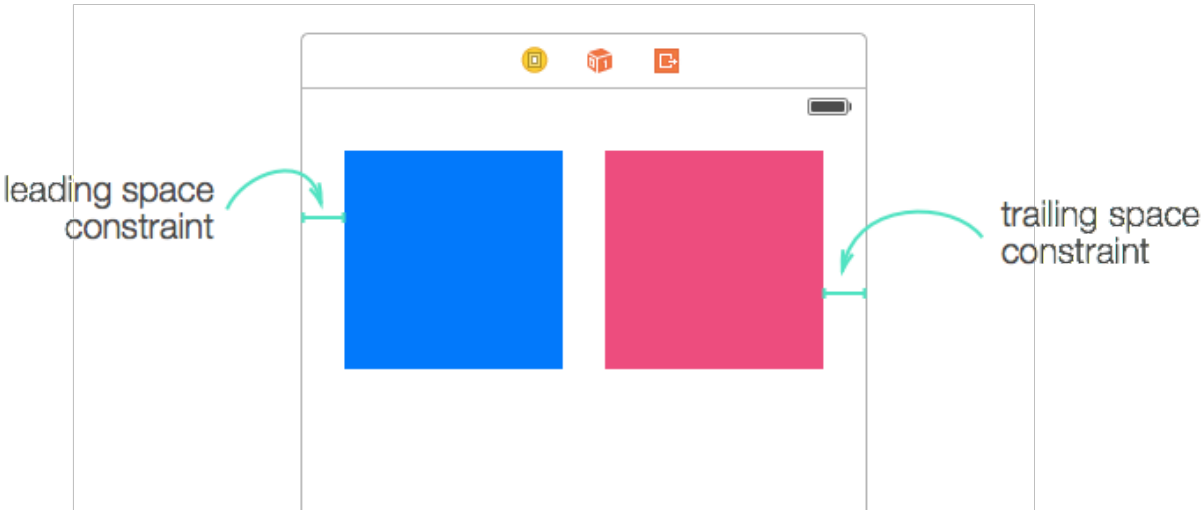


That's because without constraints, the objects remain positioned at the x,y coordinates and dimensions that we laid them out with on our storyboard.

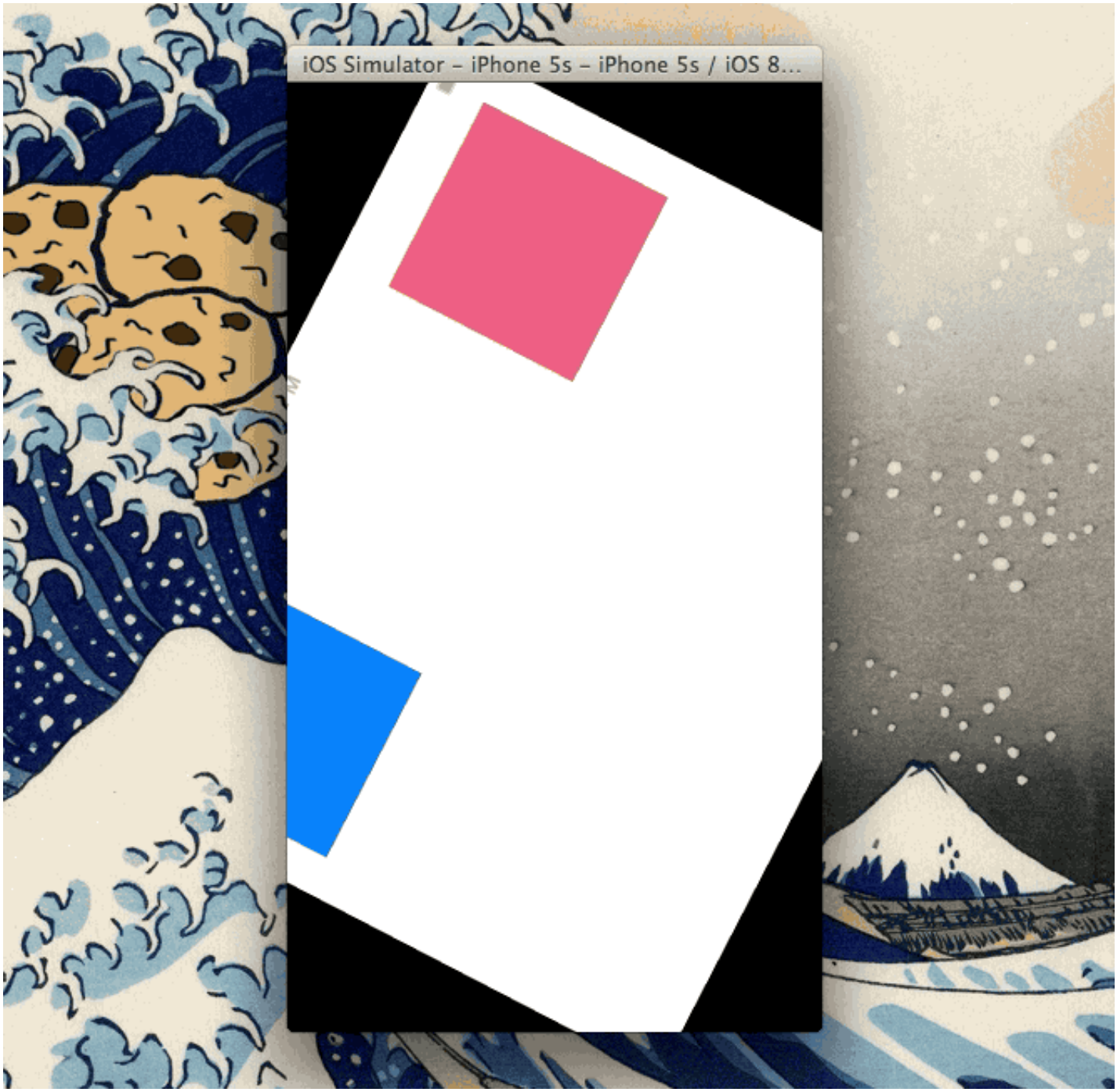
So let's start adding constants and see the effect they have.

Leading and trailing space

Add a *leading space constraint* of 10pt to the image and a *trailing space constraint* of 10pt to the red square.

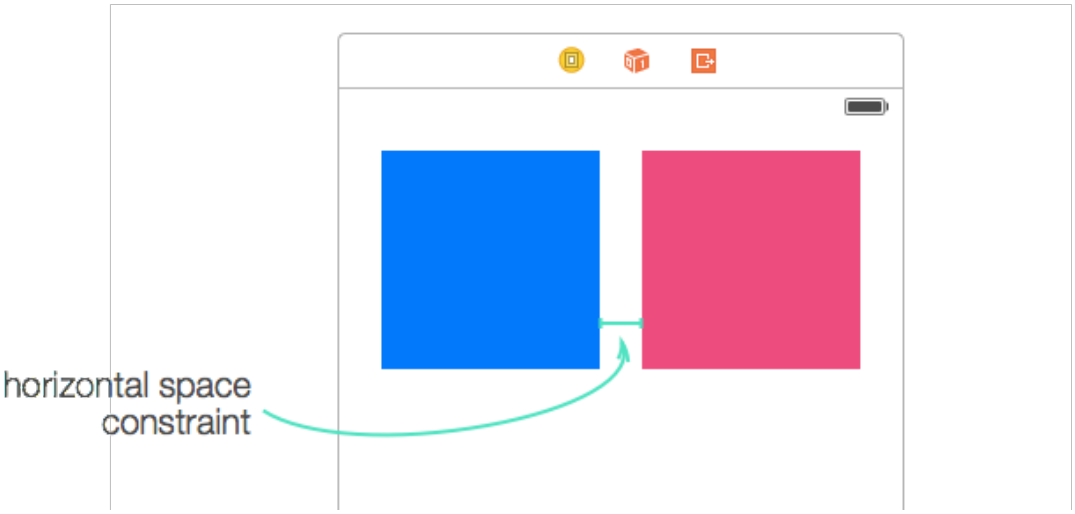


When we run the project now, the constants fix the blue square to the left edge of the screen and the red square to the right.



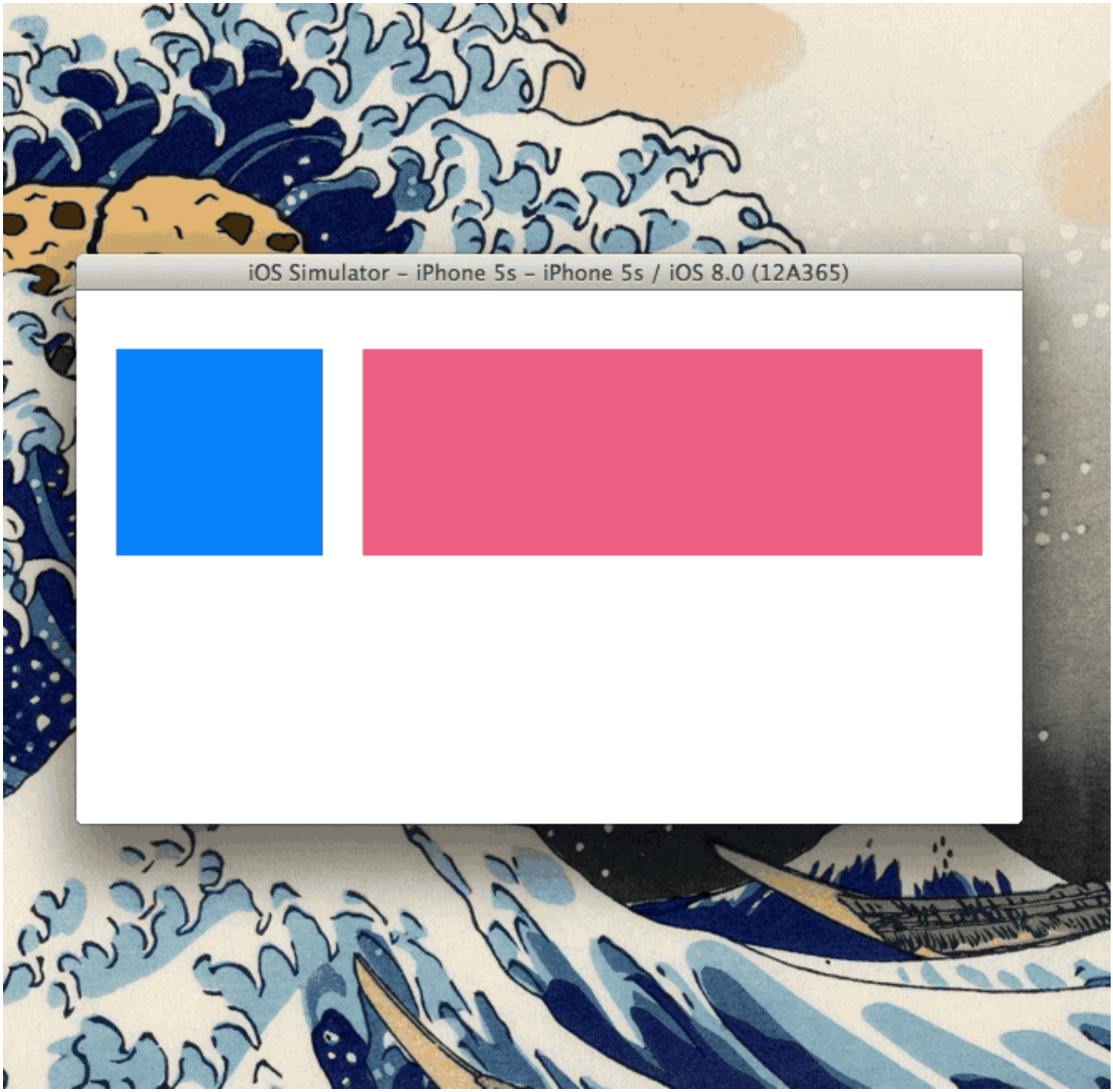
Horizontal space constraint

Add *horizontal gap constraint* of 10pt between the squares. Like the previous constraints this ensures that there will always be a fixed space between the objects.



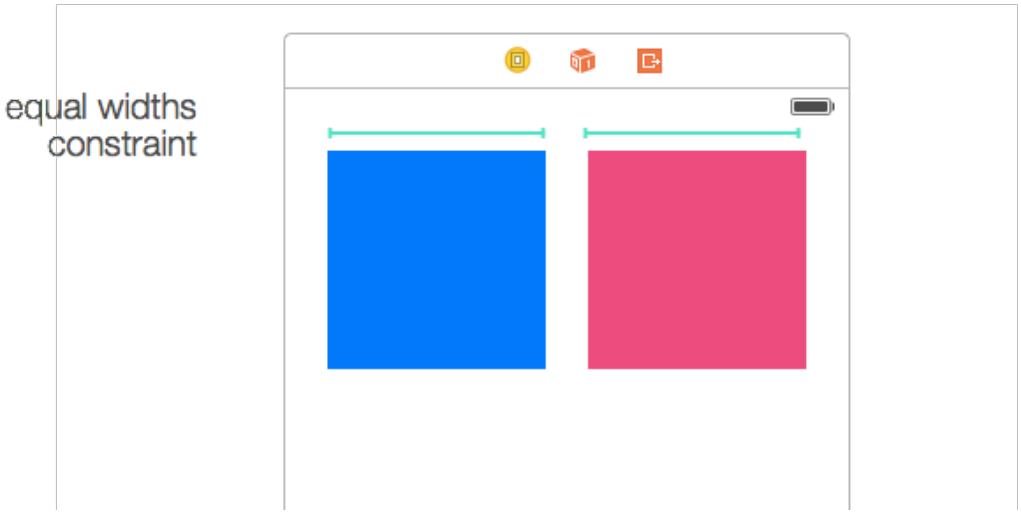
When we run the project we see a slightly unexpected result: All of our constraints are satisfied, but to do so one of the squares has been stretched to satisfy the constraint.

It seems that unless told otherwise, iOS will try and keep the ‘natural’ size of as many objects as possible.

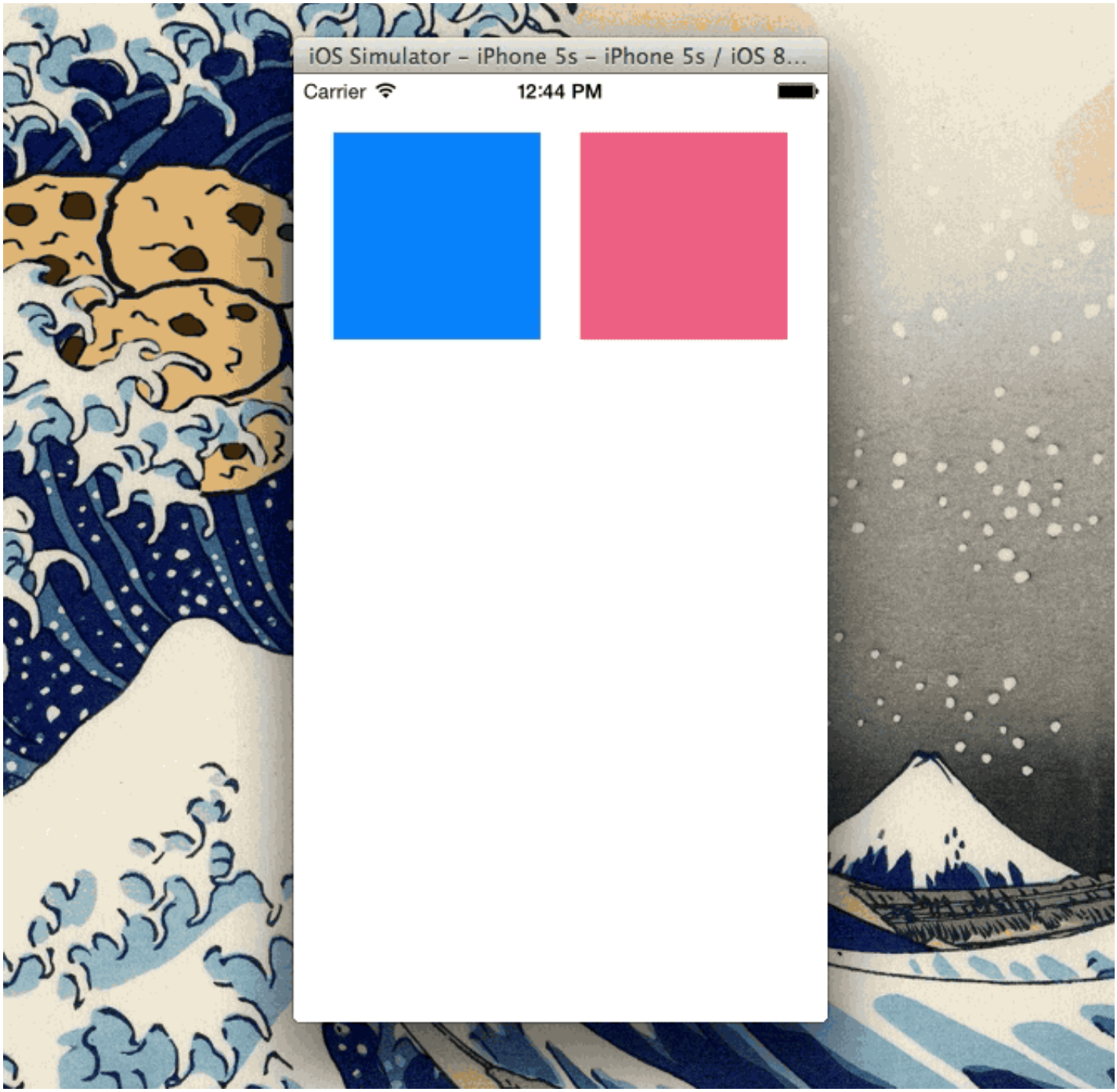


Equal widths constraint

Override that behavior by adding an *equal widths constraint*.

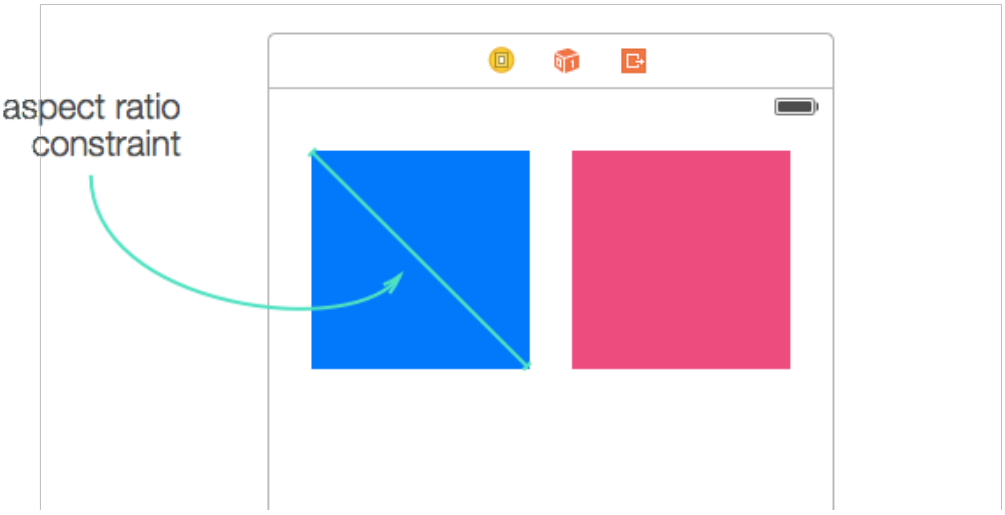


Running the project now we see that the widths of both squares are increased equally with the additional space.



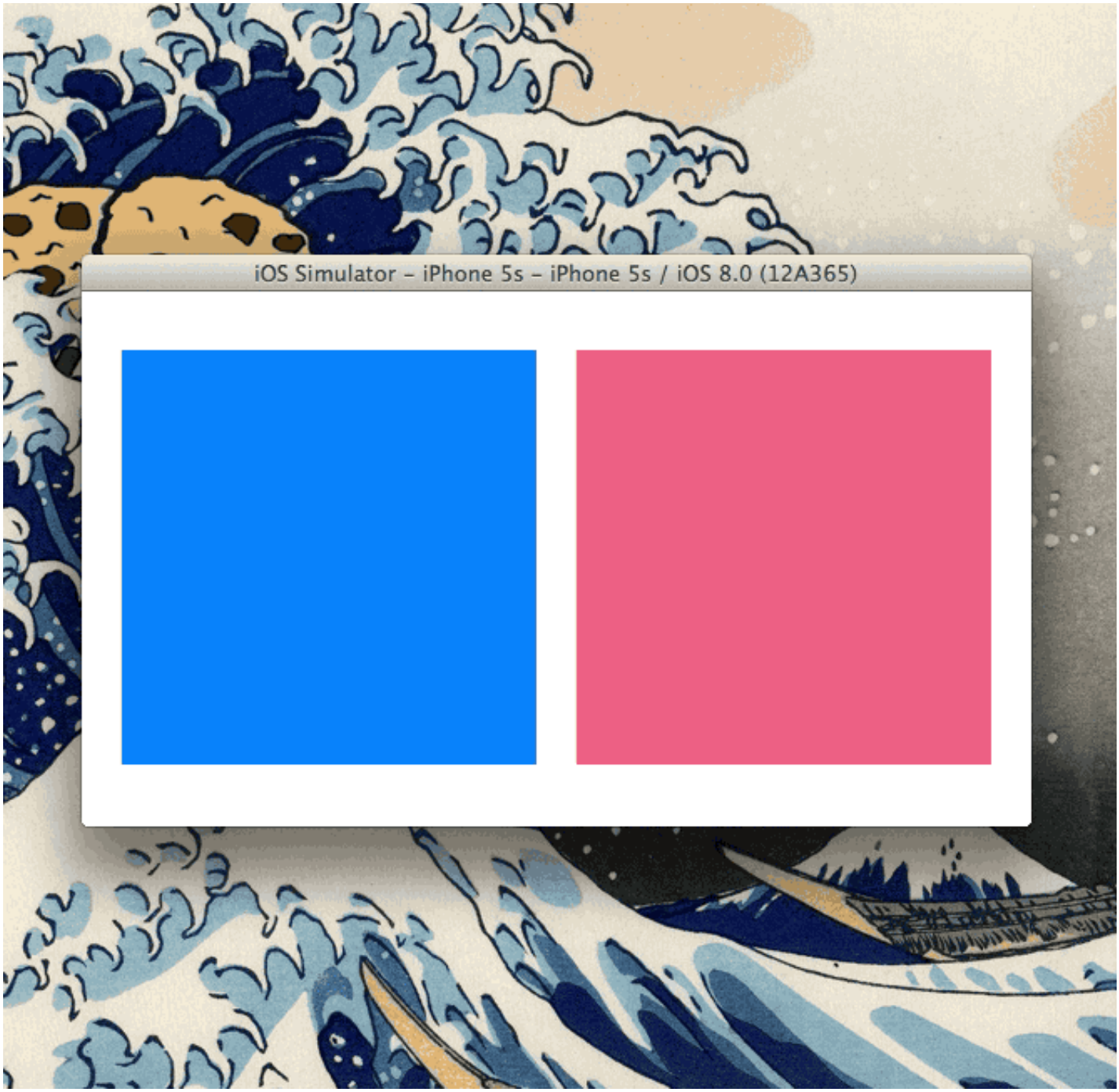
Aspect ratio constraint

Finally, add an *aspect ratio constraint* to both of the squares.

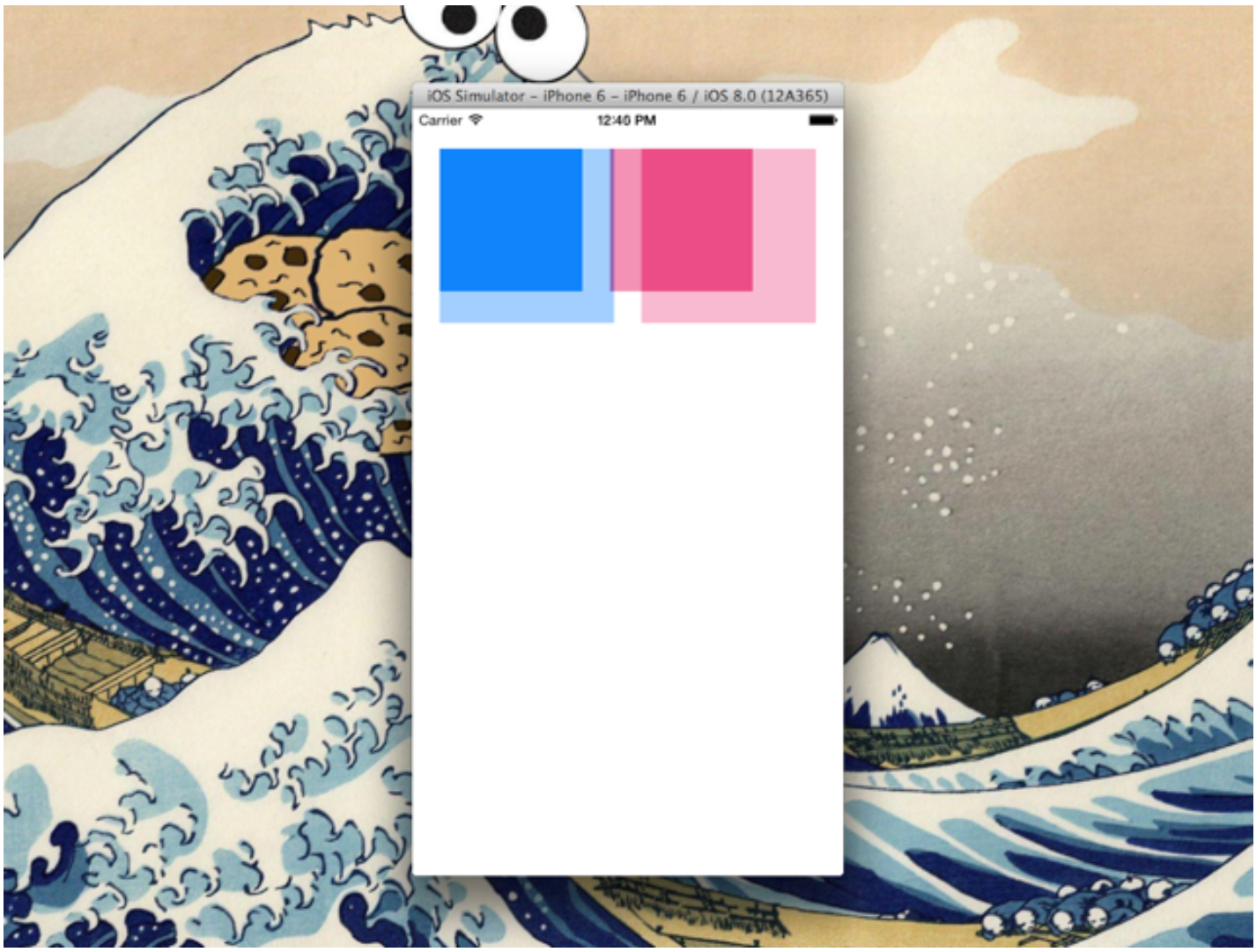


This constraint ensures that the ratio of width to height always remains the same.

Running the app now we see that as well as the width of each square increasing, the height also increases proportionality because of our aspect ratio constraint.



Running the project with the iPhone 6 simulator, and overlaying the result from before constraints we added, you can also see how the size of the squares have scaled up to take advantage of the extra space that's available.



Managing constraints

Now you've seen some of the ways that constraints can be used, here's how to actually add them in to your storyboard layouts.

Tip: Try to use a few constraints as possible

A good approach when adding constraints is to try and challenge yourself to use as few constraints as possible to achieve an effect.

I don't think there is a noticeable performance impact in iOS for the number of constraints that are entered, but the fewer constraints you have the easier your layout will be to maintain.

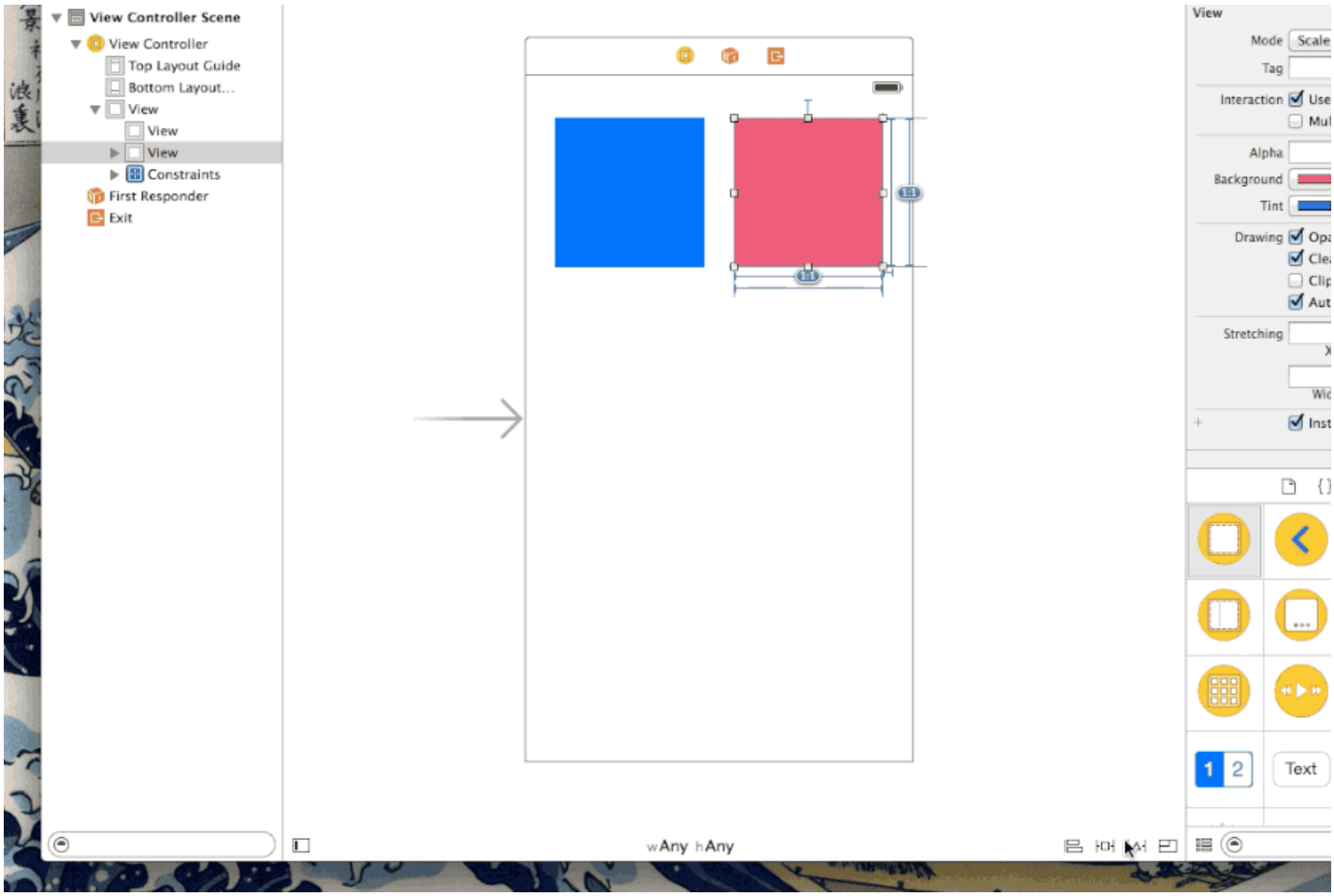
Adding constraints

There are three ways to add constraints to a storyboard [6] :

1. CONSTRAINT POPOVERS

Probably the easiest way to get started is to select one or more objects on the screen and then add a constraint from one of the popovers. I think these are a good way to start since they are the clearest way to see the range of constraints that exist.

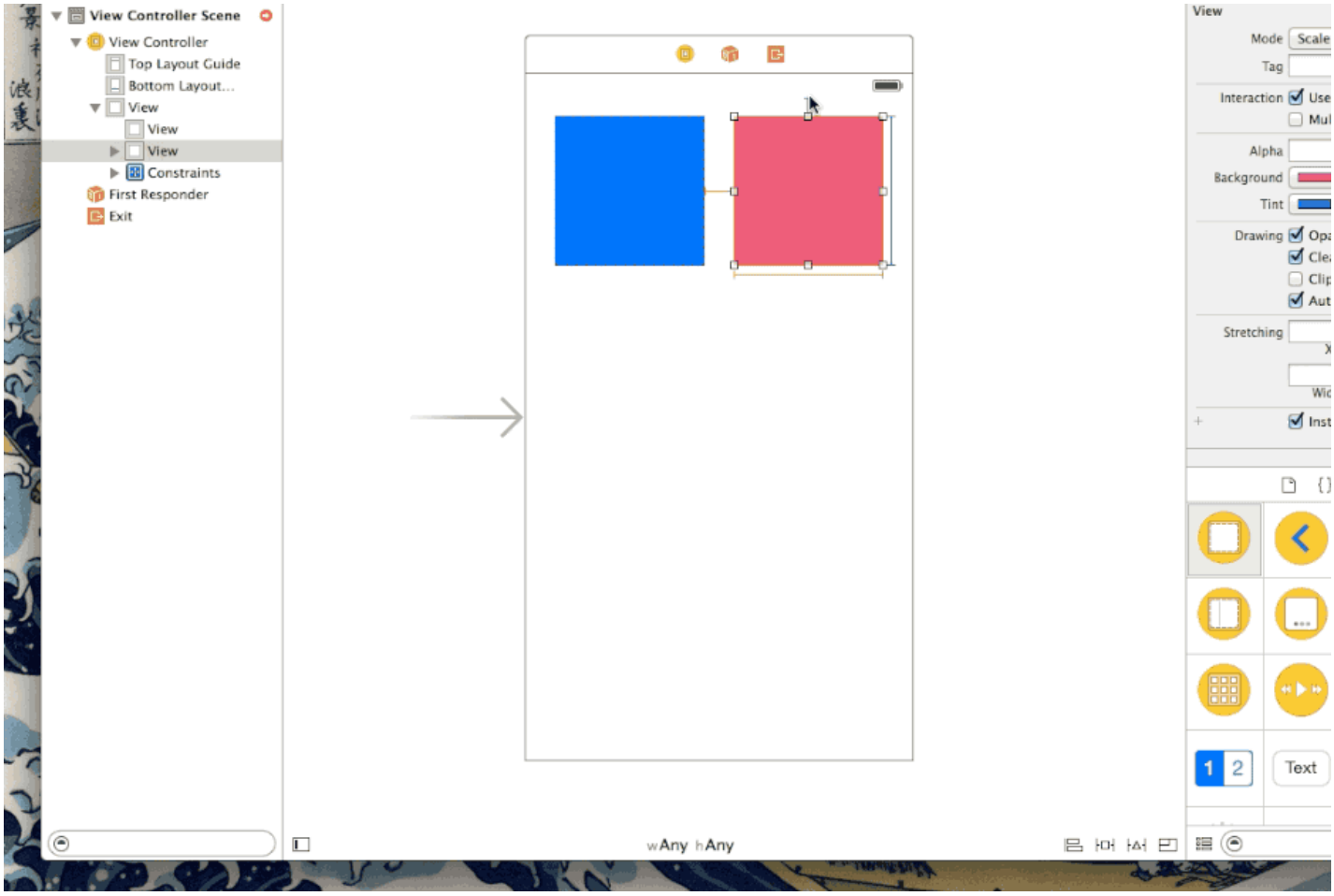
If a constraint is disabled it likely means that you need to have two or more objects selected for the constraint to make sense.



2. CONTROL-DRAG

You can hold down the `control` key and drag onto an object (either a neighboring object, the parent object, or onto itself) a pop up will show you possible constraint that can be added.

This is my preferred way to add constraints since it's faster than moving your mouse down to the bottom of the storyboard and back to the object.



3. MENU / KEYBINDINGS

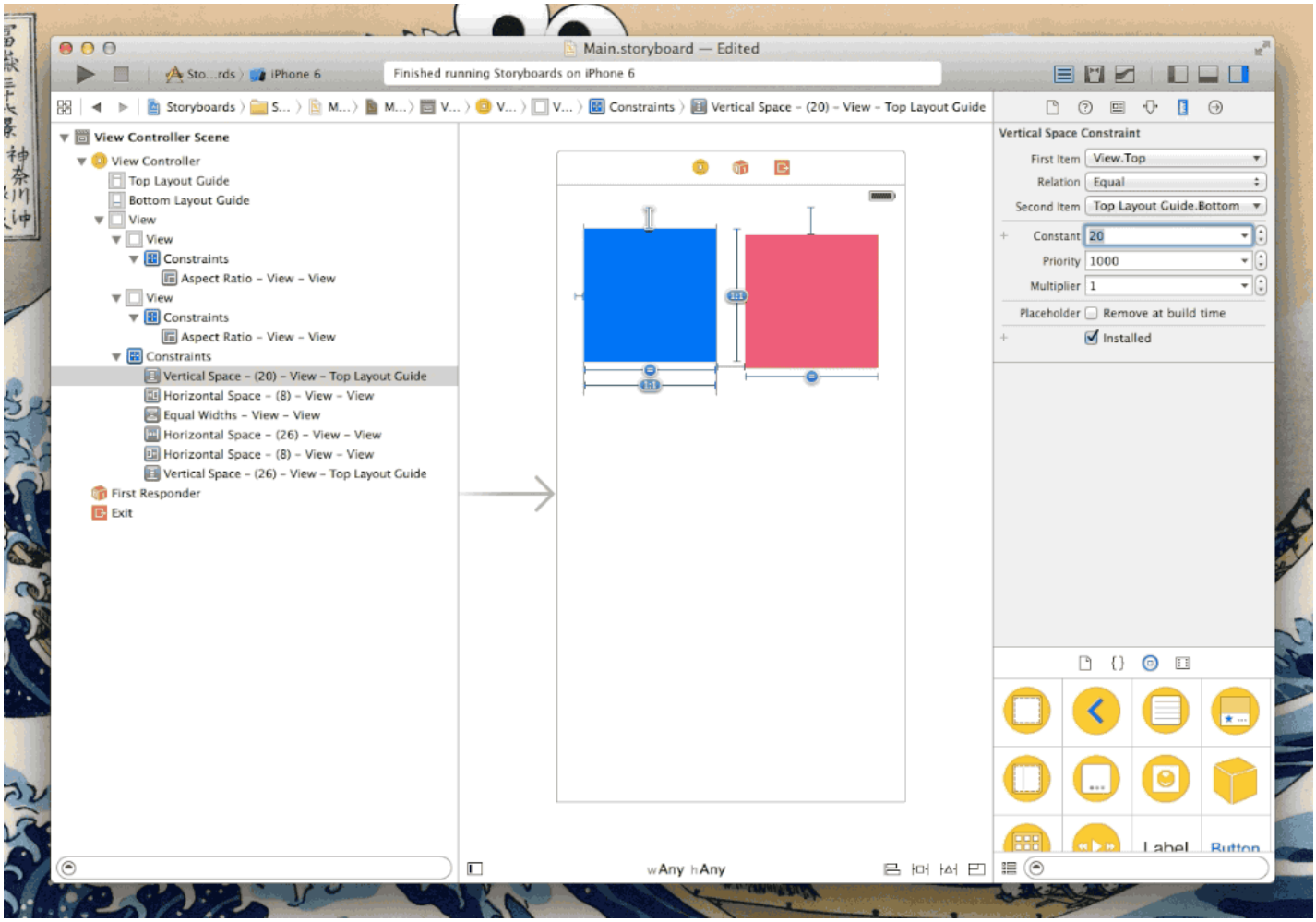
You can also add constraints by selecting one or more objects and choosing a constraint from the `editor -> pin` menu. This is the least efficient way to add constraints, but if you're finding yourself repeating a constraint over and over again it might make sense to add a keybinding to one or more options.

Reviewing & editing existing constraints

The easiest way to review constraints that are applied to a view is to select the view and toggle into the Size inspector panel.

Or toggle the constants open in the document outline hierarchy panel.

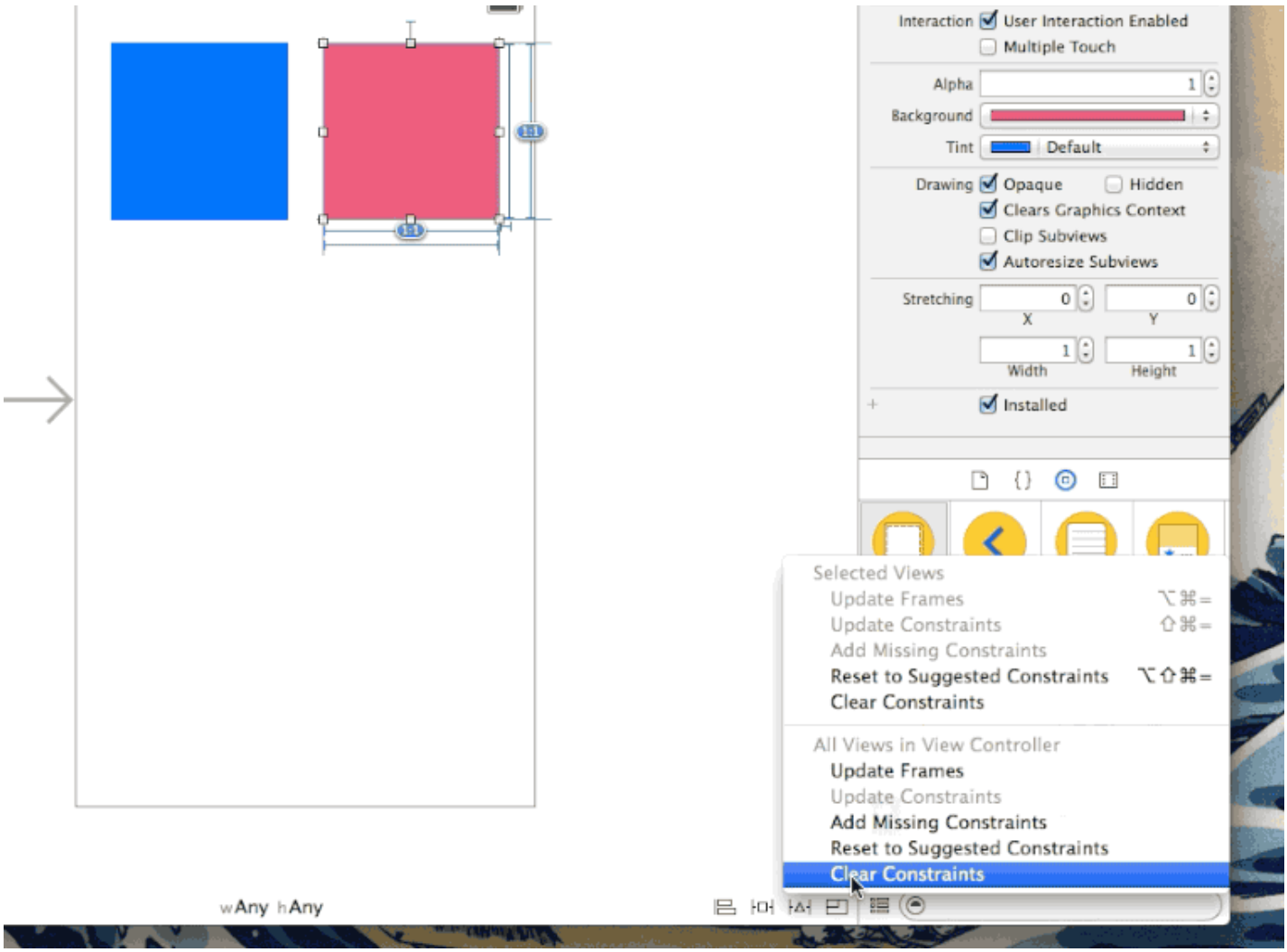
From either method, selecting a constraint allows you to directly update the constraint.



Removing constraints

Likewise, constraints can be deleted by highlighting and hitting the delete key.

To remove all constraints associated with a view, or all constraints in the screen a shortcut is available in the layout issues popover:

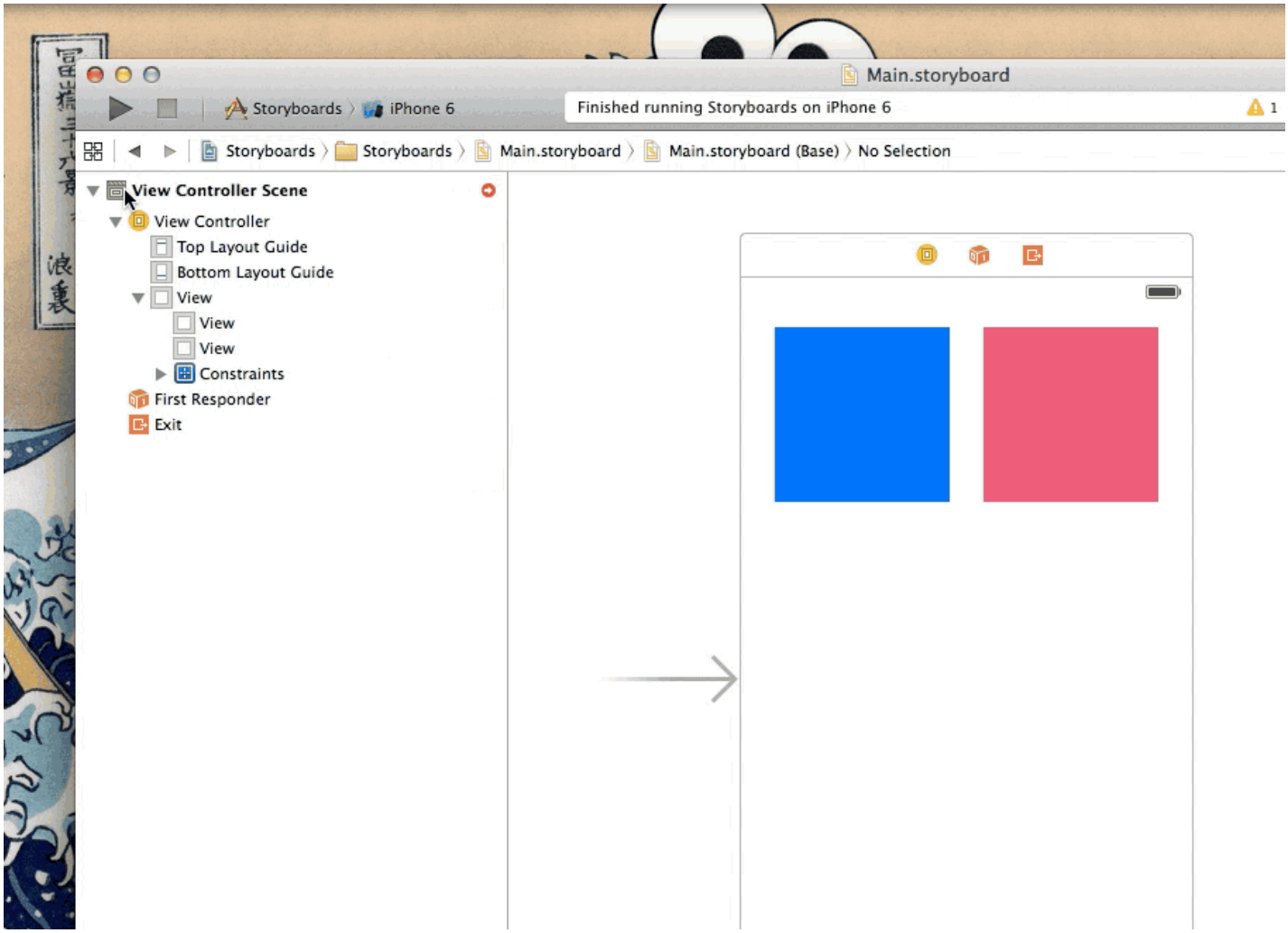


Layout issues & conflicts

As we add constraints to a storyboard Xcode will often give warnings and errors about the constraints that have been added.

Sometimes the issues are innocent, but more often they hint at a problem that needs to be resolved.

Constraint warnings and errors are best viewed in the document outline panel. Each view controller will have an red or yellow icon to indicate if there are any layout issues:

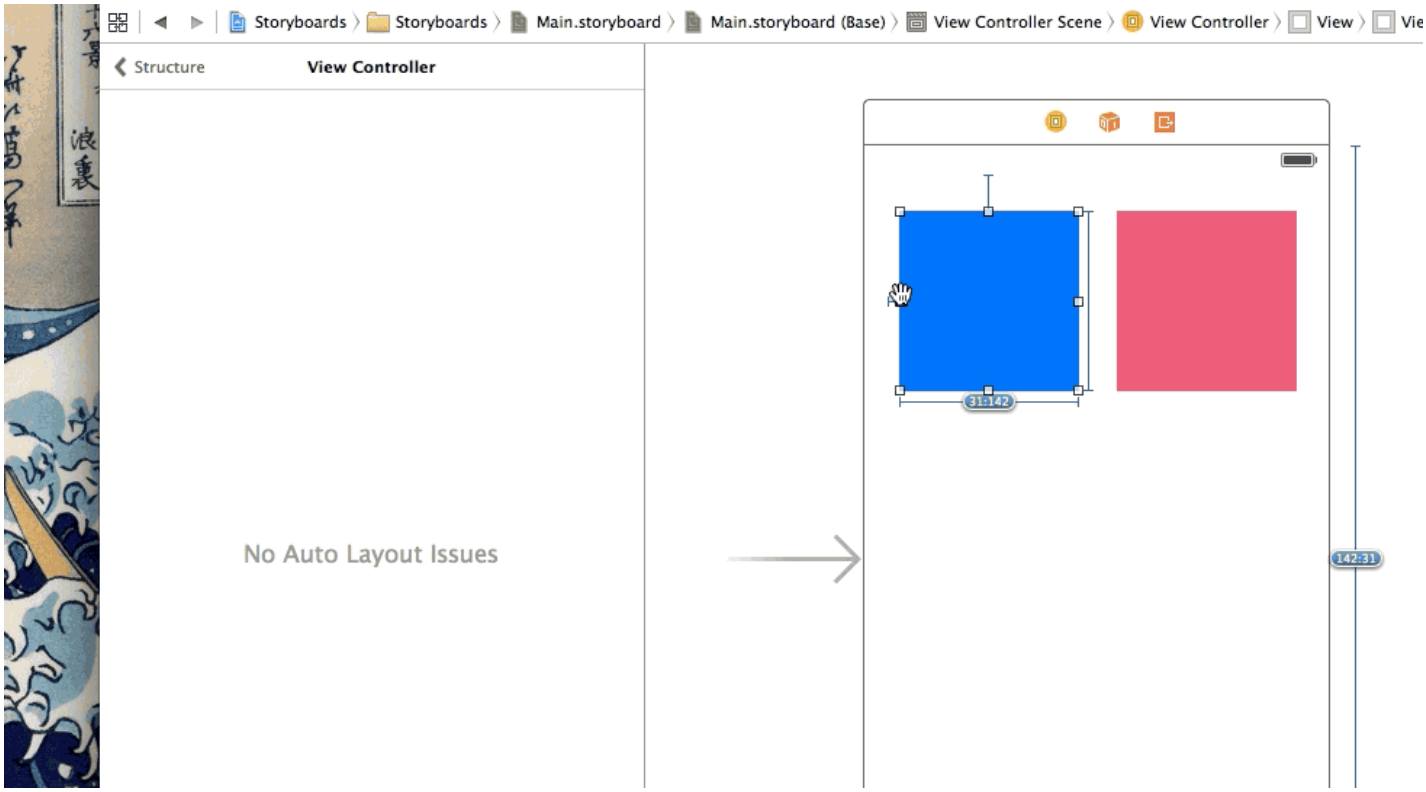


Misplaced views warning

A common warning is the *misplaced view* warning which will occur anytime there's a mismatch between where constraints say a view should be, and where it is currently shown.

These happen all the time when you switch between simulated sizes, or just accidentally moving a view with your mouse.

Clicking on an object with a misplaced view shows an dotted outline on the screen where the constraints say the view should be positioned.

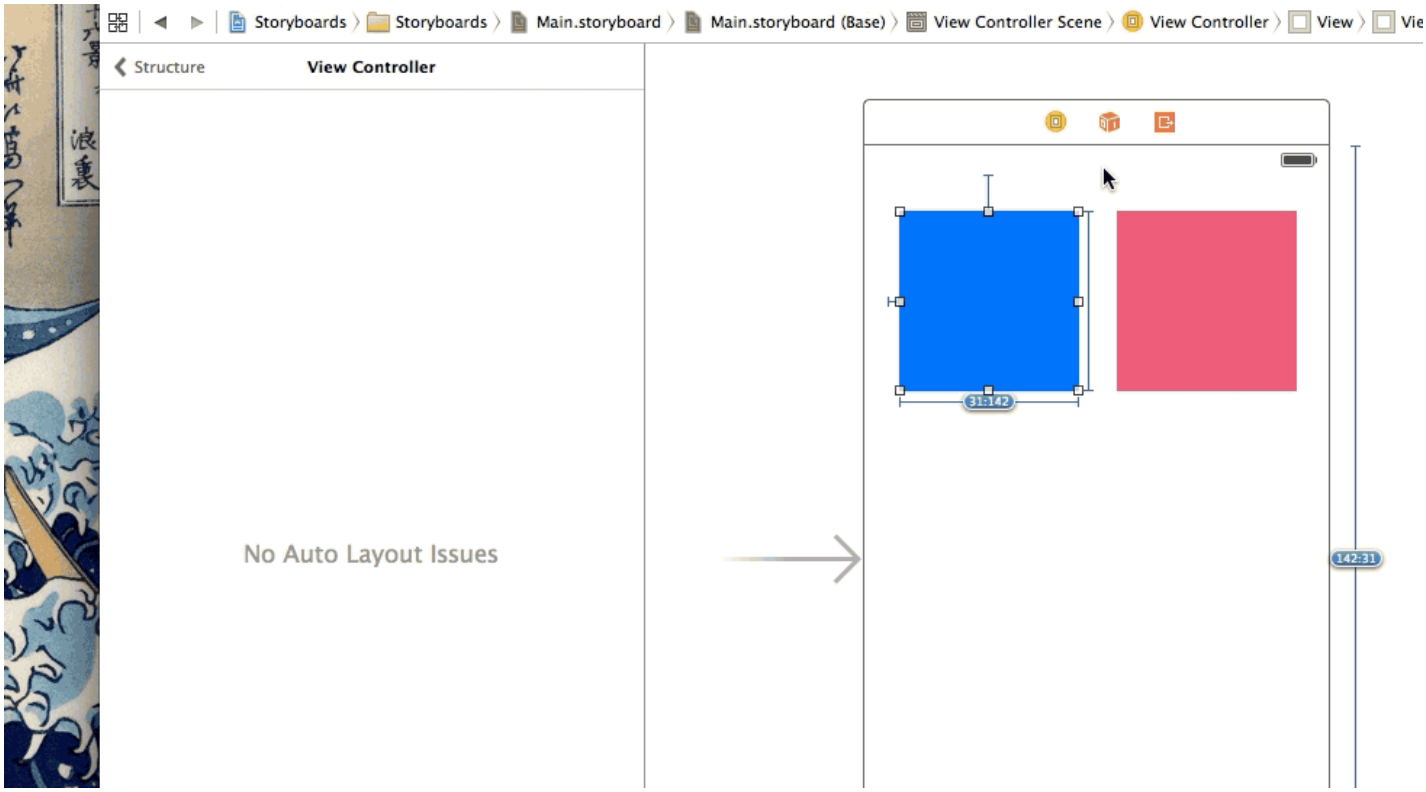


Selecting the warning presents a popover with a few options. You can either ask Xcode to update the constraints to match what's shown in the storyboard, or (most likely) you can ask Xcode to *Update Frame* which will move or resize views as needed to pop them back into the state that their constraints expect them to be.

Missing constraint error

Let's go back to our earlier example with the red and blue squares. We never added a constraint that specified the the Y-position of the squares. When we ran the app it was okay because iOS just assumed the y-location of where we placed the objects on the storyboard, but to avoid unexpected behavior it would have been better for us to specify exactly what we want.

Adding a *top space to top layout guide* constraint to each of the squares removes ambiguity of the constraints and removes the error [7] .



Size Classes

To take advantage of constraints on anything more than the most basic layouts you'll need to pair them with size classes.

Size classes are a type of trait that give information about the physical device width and height [8] .

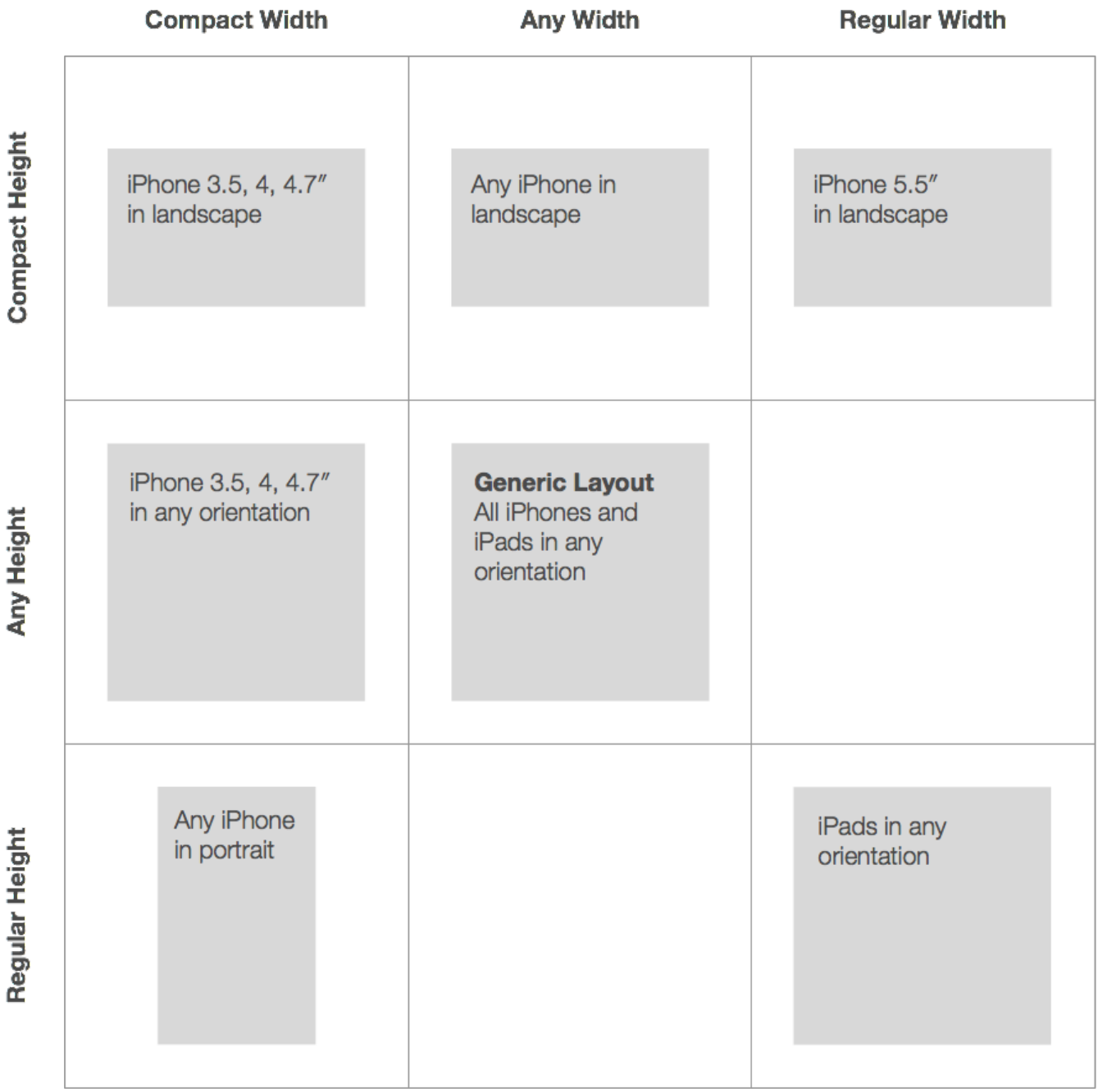
When we add an object or constraint to a screen on our storyboard, that object will only be shown, or the constraint will only be applied for devices that meet the current with and height traits.

This is an extremely powerful approach, and allows us to define variations in layouts for different devices and orientations [9] .

Width & height traits

Using a mixture of *compact*, *any*, and *regular* for *width* and *height* allows us to target a range of device layouts.

Not every variation is covered [10] (for instance there's no way to target the iPhone 6 Plus in portrait, or to distinguish orientation in iPads) but a good range of options are covered [11] .

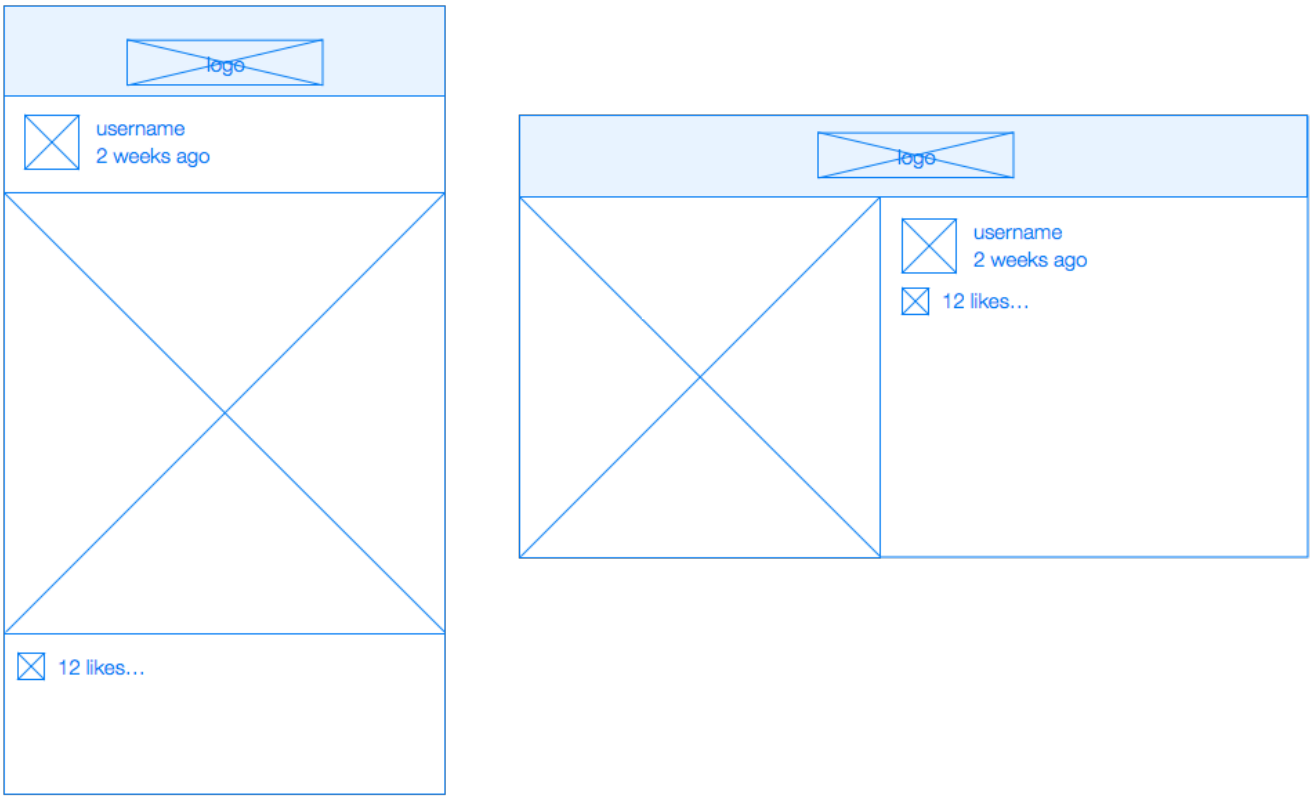


Again, the easiest way to introduce size classes is to show them in action so let's look at a real life example.

Our goal...

As an example, we'll experiment with making a simplified version of the Instagram detail screen adapt to larger screens, and both orientations.

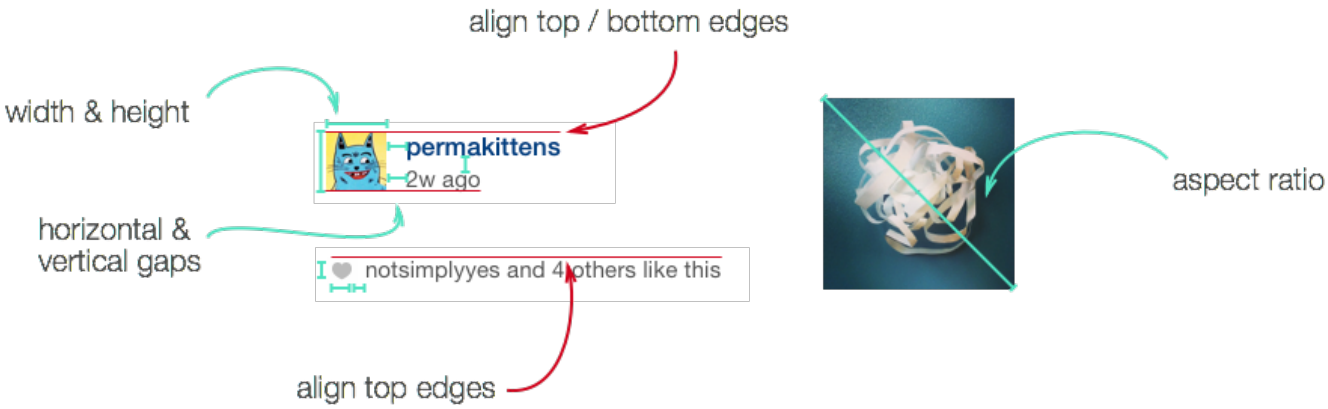
Our aim is to create a layout that scales sensibly to the larger iPhone 6 device sizes (by sensibly I mean only the photo scaling, and other elements like the profile photo remaining a constant size), and an landscape layout where the image and metadata are presented size-by-side rather than stacked on top of each other:



Add constraints for generic size class

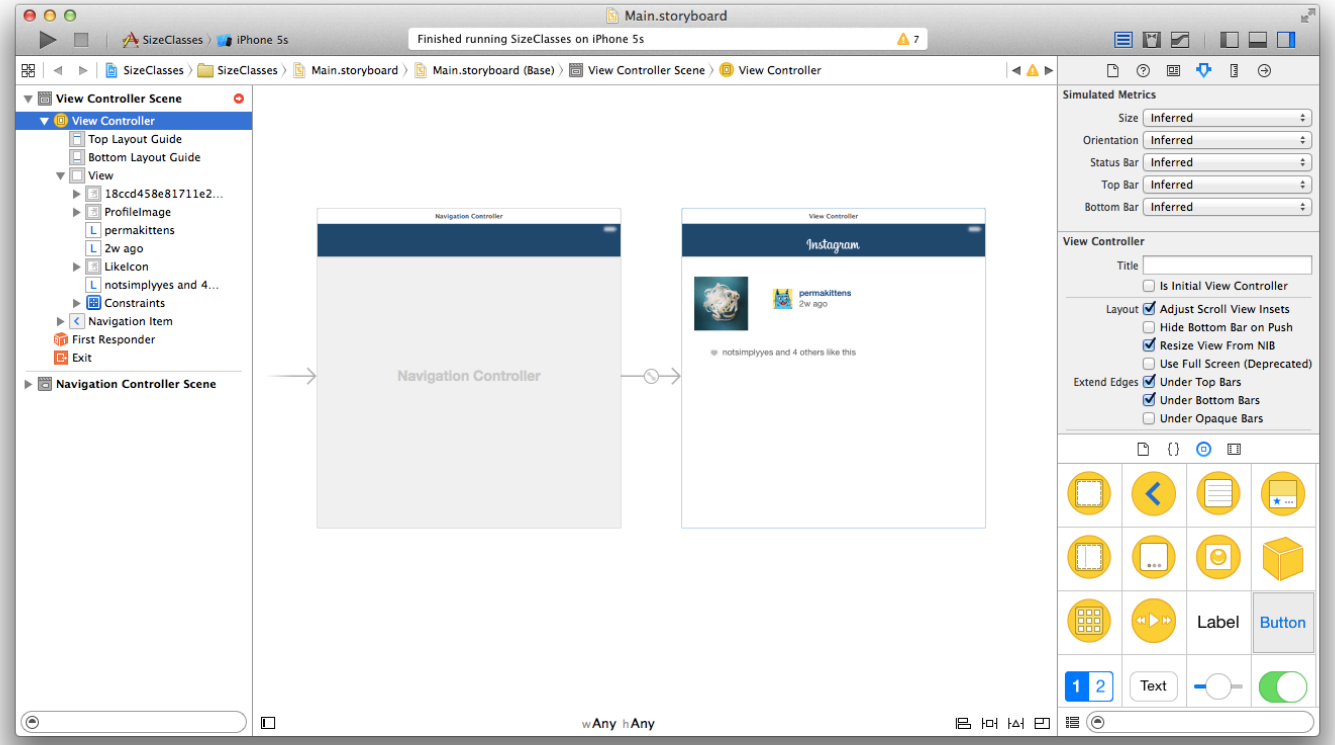
This time let's not use the simulated size metrics like we did in earlier examples, but instead use the abstract square layout of the generic size class to remind us that any views and constraints added for this size class should only be those that apply for any screen size and both orientations.

This includes these constraints:



As well as the spacing, gap, and aspect ratio constraints that you saw earlier, some of these objects are also positioned by aligning their top or bottom edges.

Add those views and constraints to your storyboard. Here's an example of what my storyboard looks like after this step:

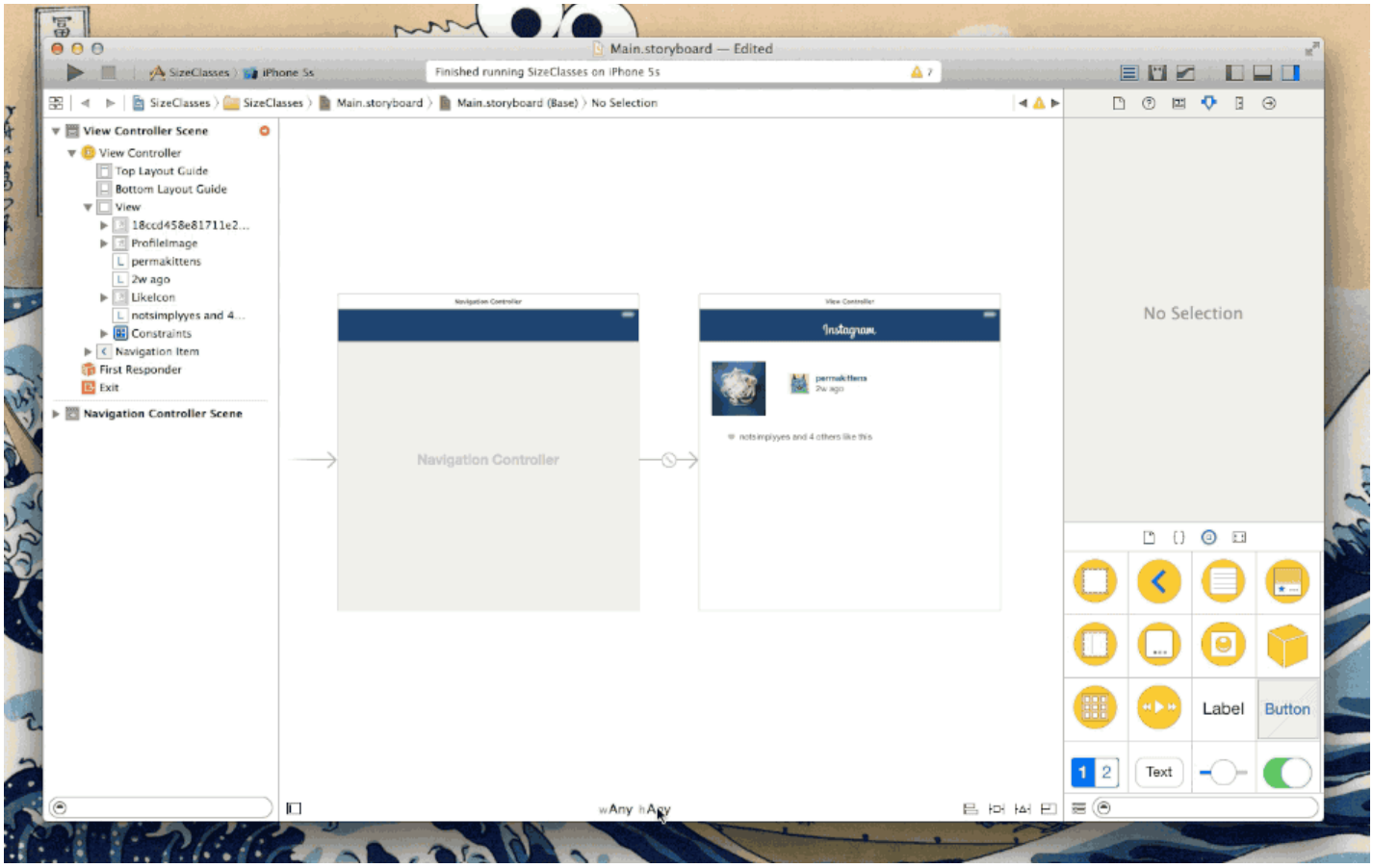


Note that it's got all the key items needed for both orientations, but I've not bothered with trying to layout this generic layout to look like either of the orientations. That comes next!

Add constraints for iPhone portrait layout

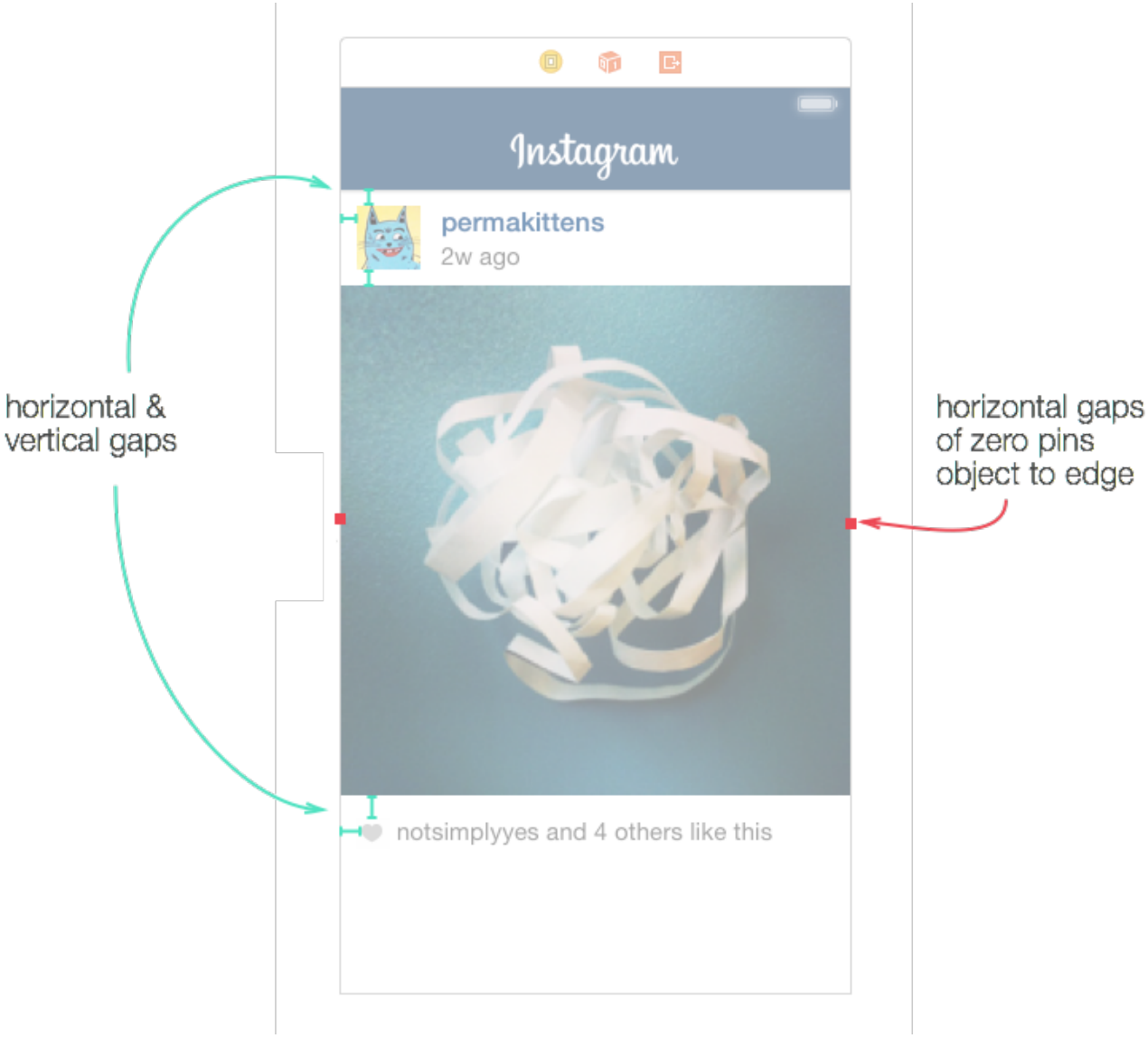
Now switch size classes to *compact width* and *regular height*. This combination of size traits apply to any iPhone device when in portrait.

To switch size classes click on the size class label at the center-bottom of your storyboard and select the combination you want to view.

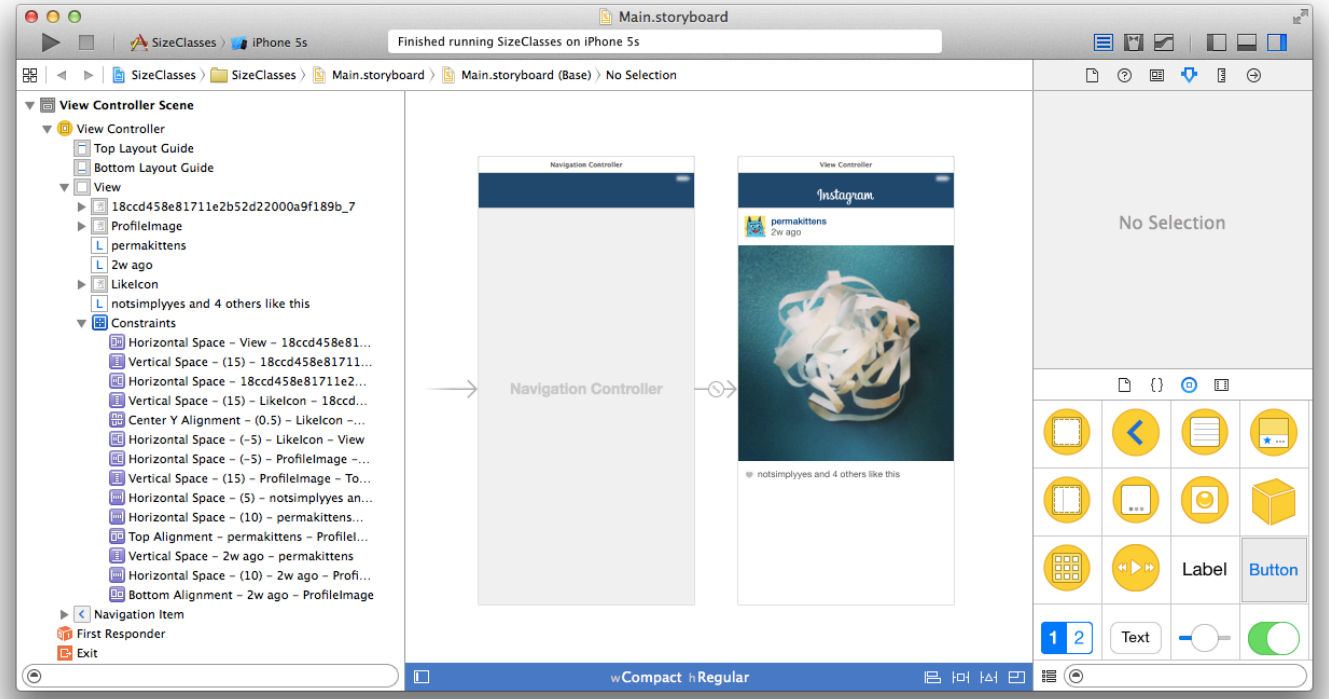


Notice that when you switch size class the size of the view controller updates to reflect the new abstract layout.

Finally, now you can reposition and resize the views and add constraints for a portrait layout. Here are the constraints that I added to the portrait layout:

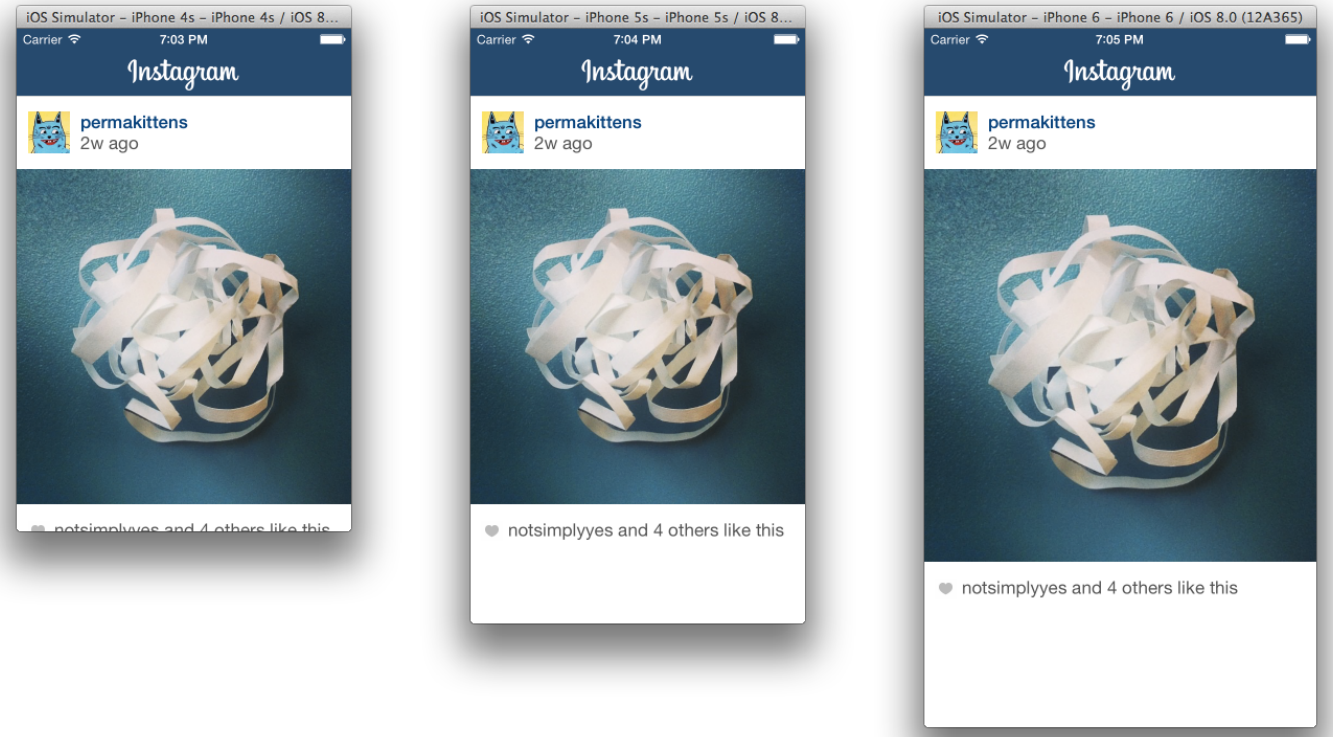


Here's what my storyboard looks like after this step:



At this stage you should be able to run the project with any of the iPhone simulators (3.5, 4, 4.7 or 5.5 inch screens) and see a layout that makes sense in portrait, although switching over to landscape will still be a mess.

Notice that we never defined the width of the image, instead we simply pinned the left and right edge of the image to the sides of the screen. As the screen size increases on the iPhone 6 the image width will be stretched out, and because we have a constraint that fixes the aspect ratio of the image width and height, the height also increases.



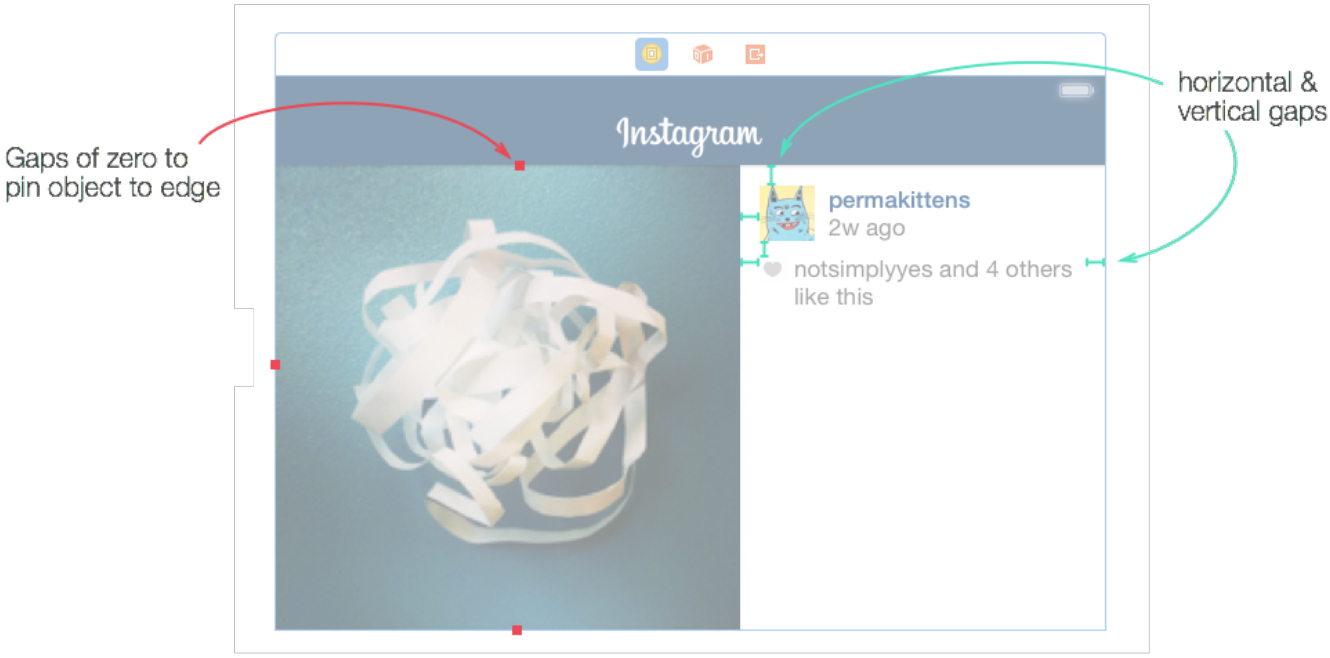
Add constraints for iPhone landscape layout

Switch size classes to *any width* and *compact height*. This combination of size traits apply to any iPhone device when in landscape and update the layout to match our goal for the landscape view.

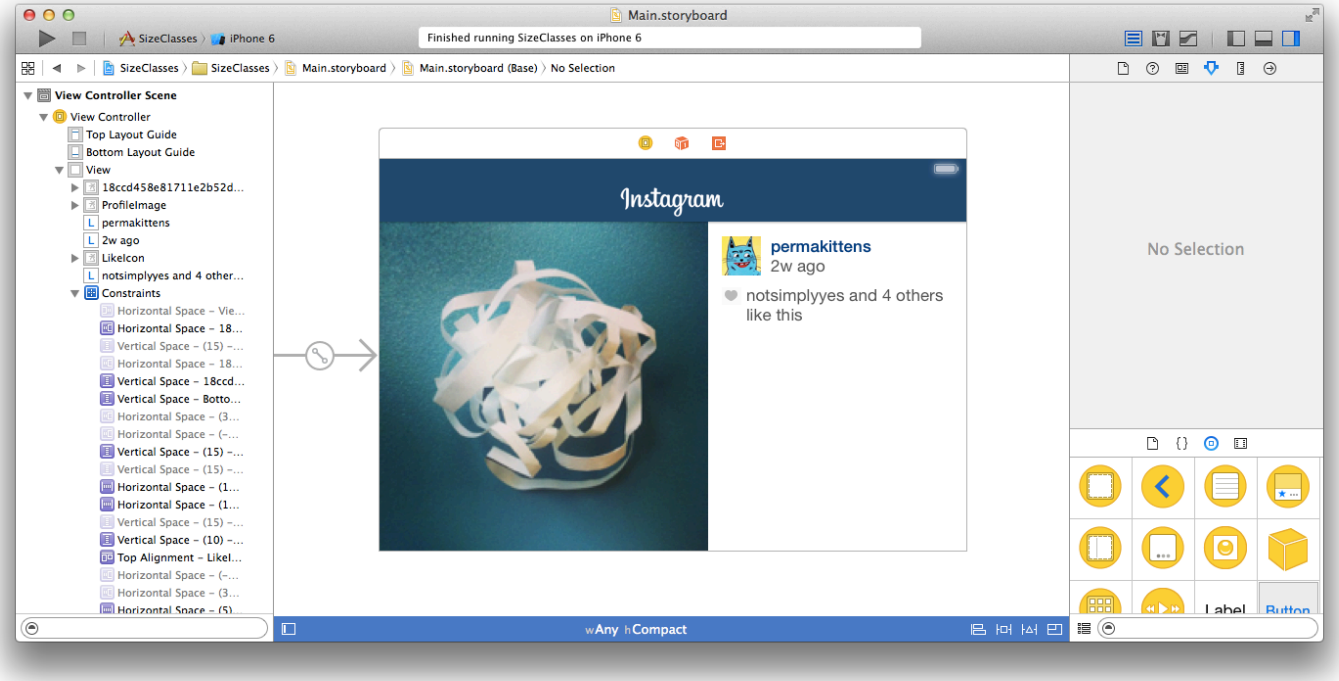
This time the top, right and bottom edges of the image are all pinned to the edge of the parent view.

The only other interesting part of this layout is that *trailing space constraint* added to the comments label.

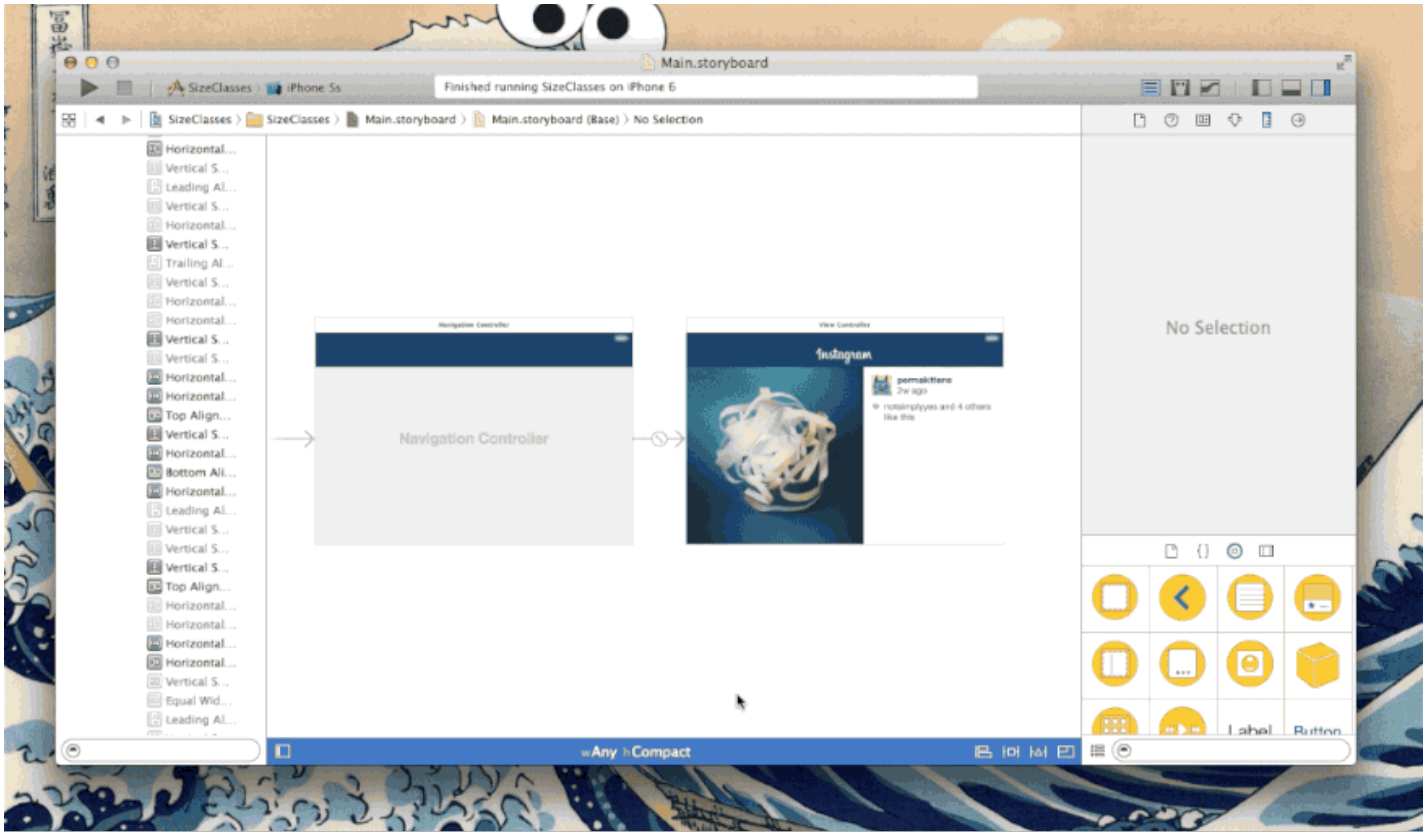
While the constraints on the image are scaling the image up, the constraints on this label are squeezing it to become narrower. To accommodate the text in this view, the height has to increase to show two lines [12] - which it does!



Here's what my storyboard looks like after this step. Notice that in the document outline some constraints are slightly faded out. These are the constraints that we added for the portrait view. They still exist, but with the current width and height traits selected, they no longer apply.



Also notice that when you switch between width and height traits the storyboard updates to show the constraints that you've added for those traits. Neat!



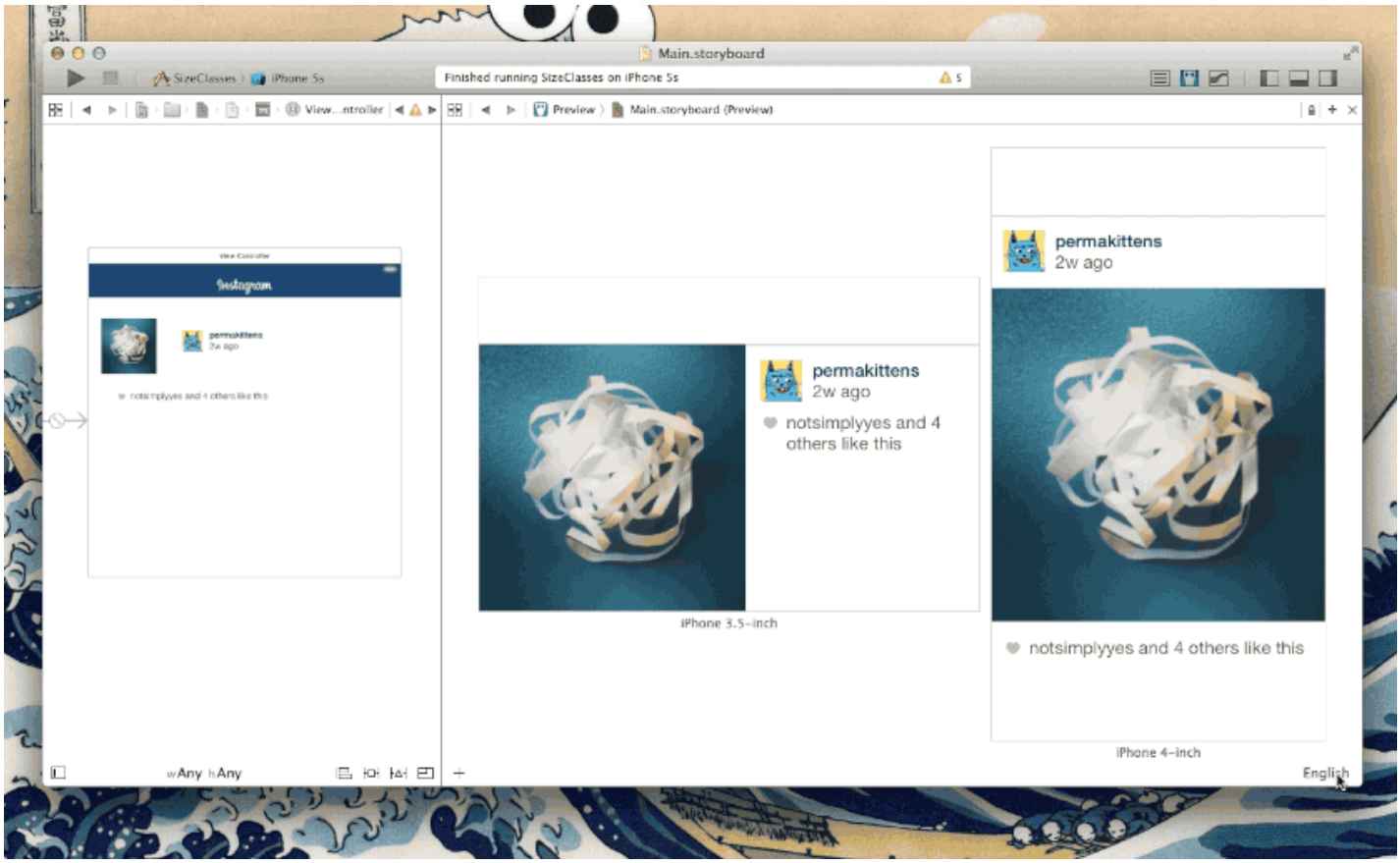
Running the project now, finally we'll see a layout that works for both portrait and landscape, and a nice transition between the two when the orientation changes.

I've slowed this transition down so you can see the details. Notice that the layering of views in the hierarchy is maintained in the transition. Because of this, we need to start thinking about how views are layered even for views that don't overlap in their normal layouts…

Assistant Editor: Device Preview

Another nice addition in Xcode 6 is the device preview assistant. Instead of running your project multiple times to check if your layouts are correct, you can use the assistant editor storyboard preview and choose as many devices and orientations as you want.

It's not perfect (for instance the navigation bar color is lost on the preview in this example) but it was a great option to switch from your language to a *Double Length Pseudolanguage* which repeats any text in your storyboard.



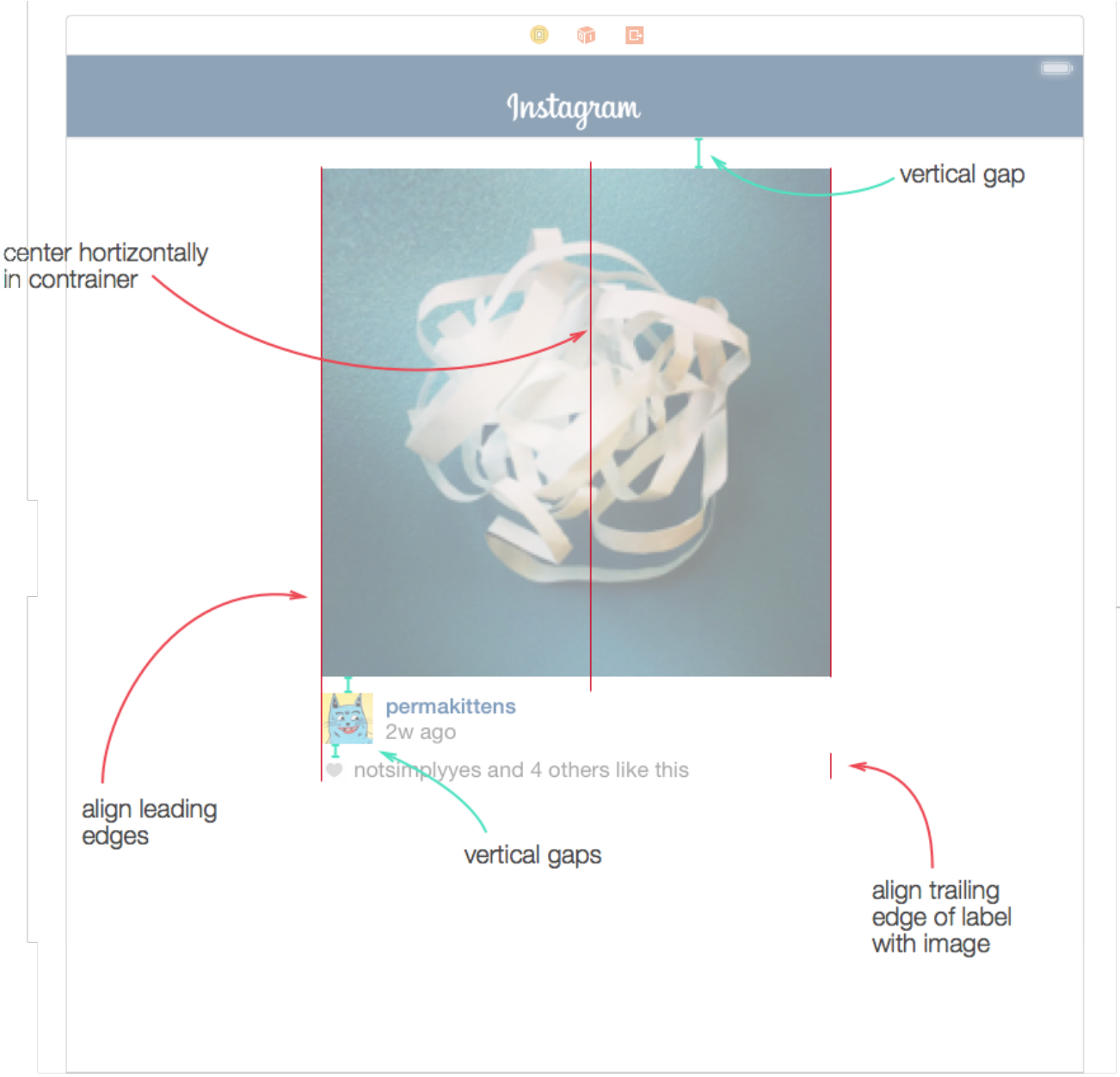
Here's the assistant preview for my storyboard showing the 3.5 inch iPhone in landscape and the 4 inch iPhone in portrait. You can see from my storyboard that I've still got some work to do to accommodate longer labels… -_-

Add constraints for iPad layout

Now switch size classes to *regular width* and *regular height*. These are the traits for an iPad in both landscape and portrait.

As before, reposition and resize the views in the screen. For the iPad layout I'm deciding to go with a fixed size image (so it won't change size between landscape and portrait like on the iPhone) and the metadata all underneath the image.

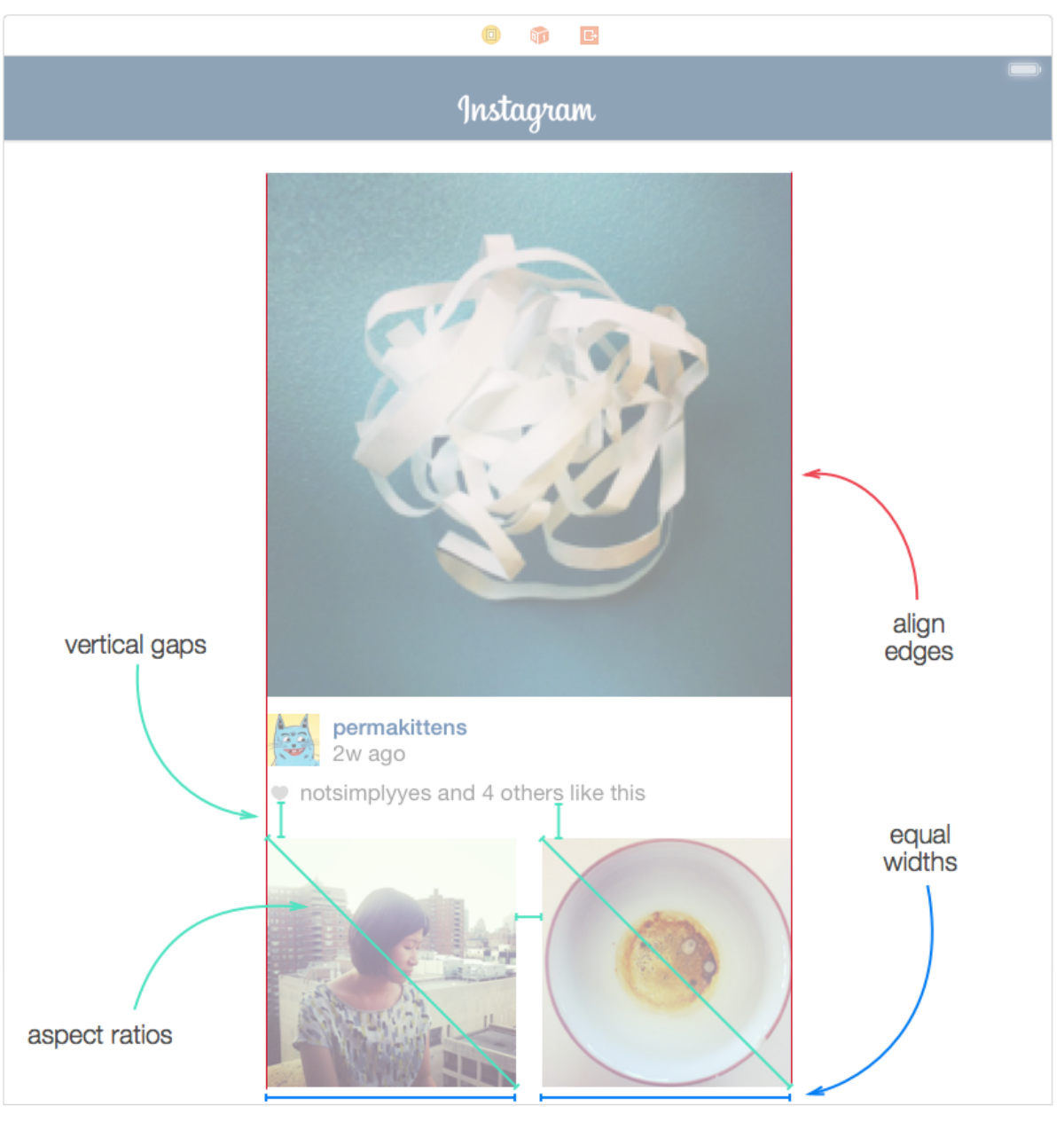
Here's my storyboard after rearranging the views and adding constraints:



So far we've added constraints for a particular set of size class traits, but now we're going to do something different.

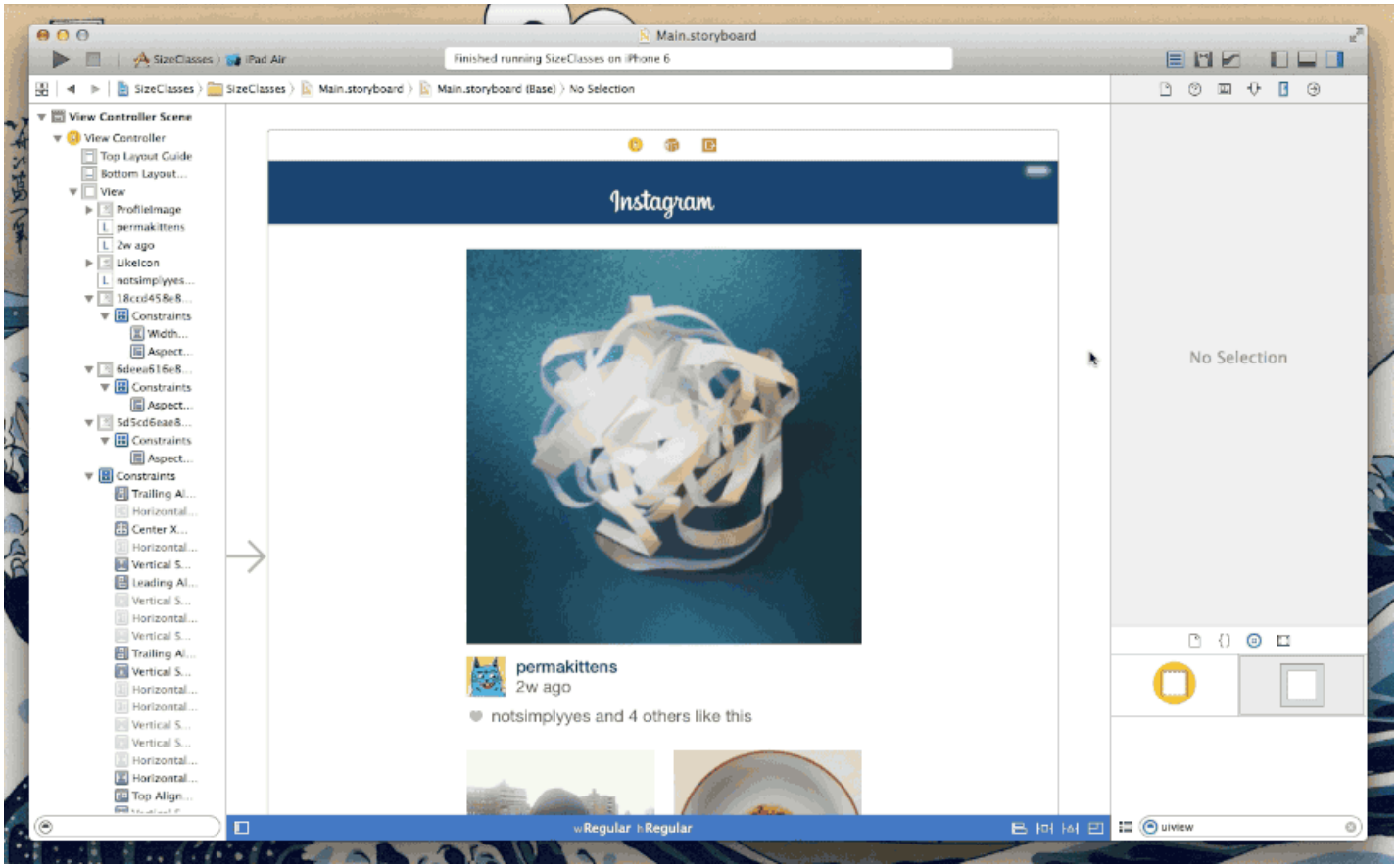
With the extra space on the iPad, let's not just rearrange views, but also add extra content. For the purpose of this example, add two more image views (perhaps to represent the images taken directly before and after the main image).

Here's my storyboard after these updates:



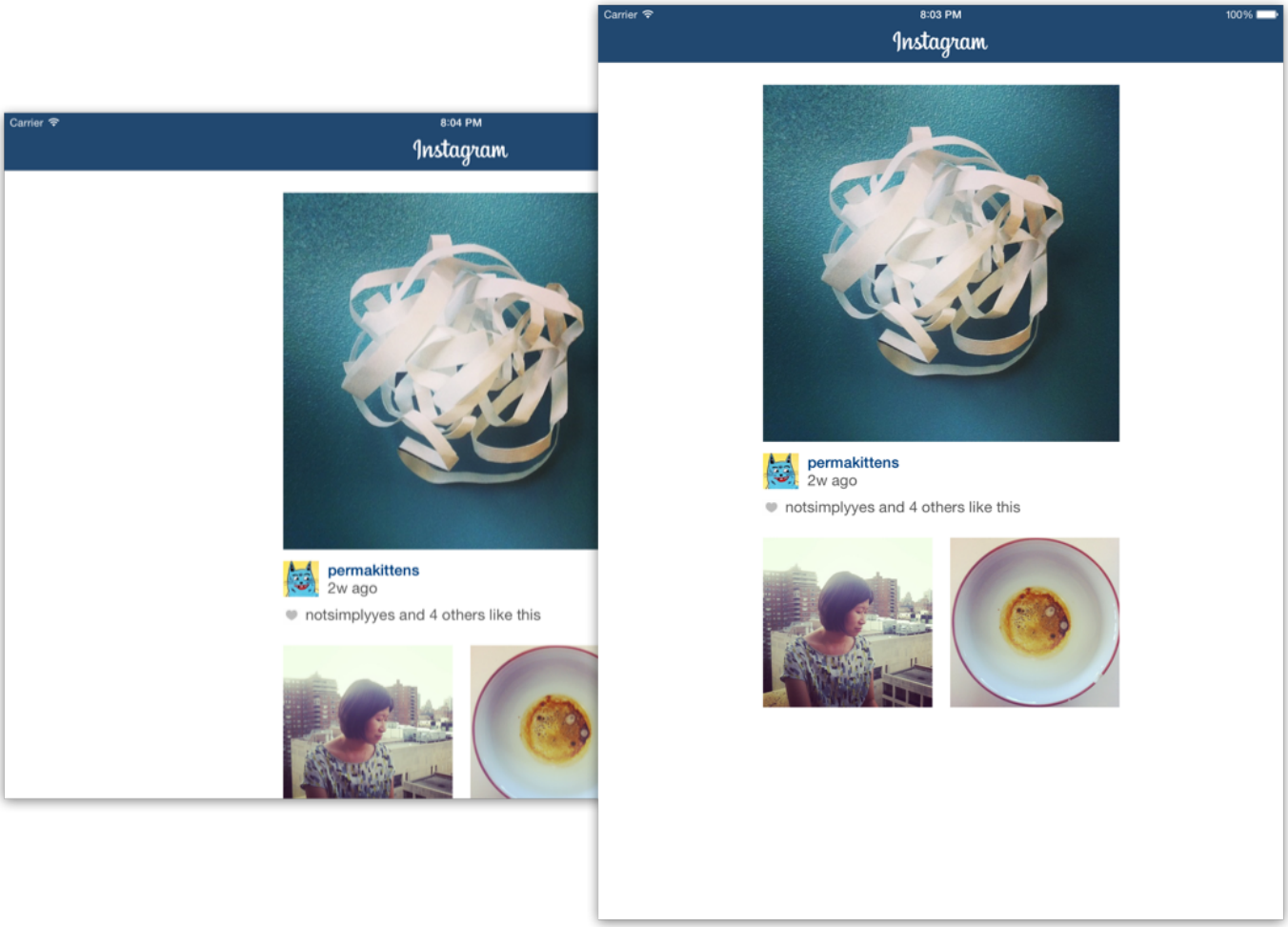
I could have chosen to enter the width and height of these secondary images manually but instead defined their size relative to the main image [13].

The nice benefit of this is that if we decide to change the size of the main image, the rest of the layout updates automatically without us having to manually calculate and update the width and height of the secondary images.



Here's a screen capture of me changing the value of the main image width constraint and seeing the waterfall of updates cascade onto the other views.

I don't have the screen resolution to capture the project running in an iPad simulator, but here are some screenshots of the resulting layout in landscape and portrait:



Using layout & spacer views

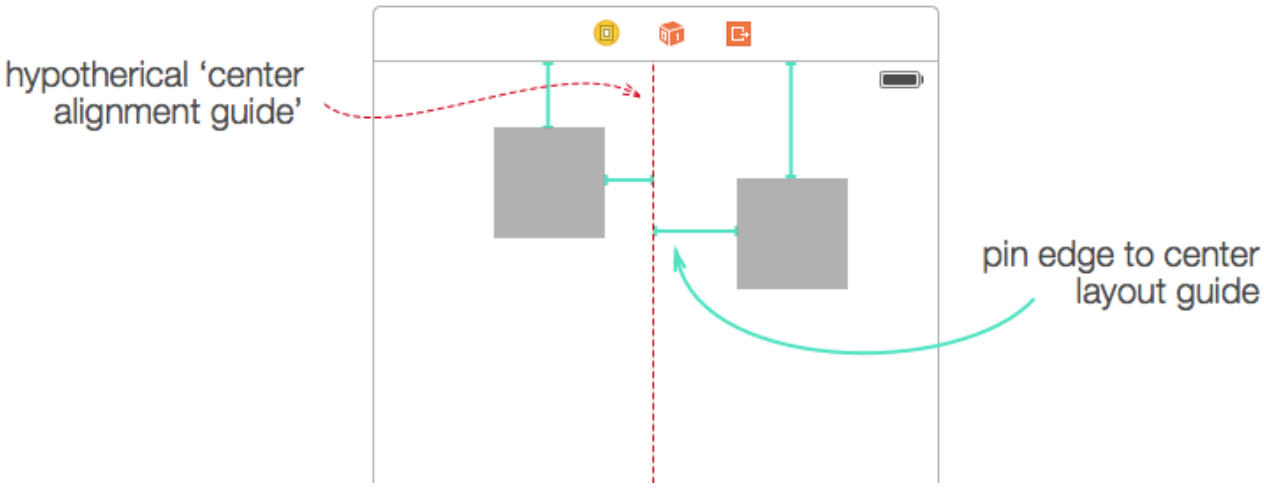
With the types of constraints that are currently available in Xcode there are some common layout scenarios that need a little help to achieve [14].

Layout view example

It's easy to align the edge of a view with the edge of the parent view, or to align a view exactly to the center of it's parent view, ~~but I've not found a simple way to pin a view to a fixed distance from the center~~.

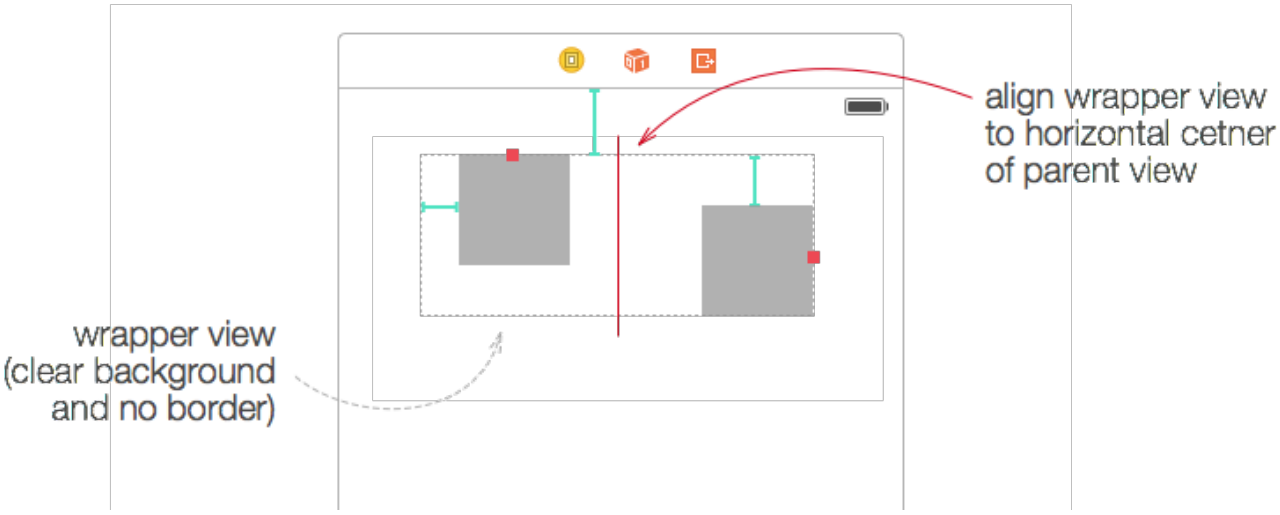
> *Thanks to Jeff Nouwen for pointing out that after pinning a view to the center, you can change it's constant value from the default of 0 to offset it by any amount you want .*

Conceptually, what we need is a hypothetical *'center alignment guide'* that would allow us to pin the edges of any view to the horizontal or vertical center of the parent view:



To achieve this with the constraints that are available to us today we could embed our views in a wrapper view — a view with a clear background and only exists to help achieve our layout — which can then be aligned to the center of the parent view.

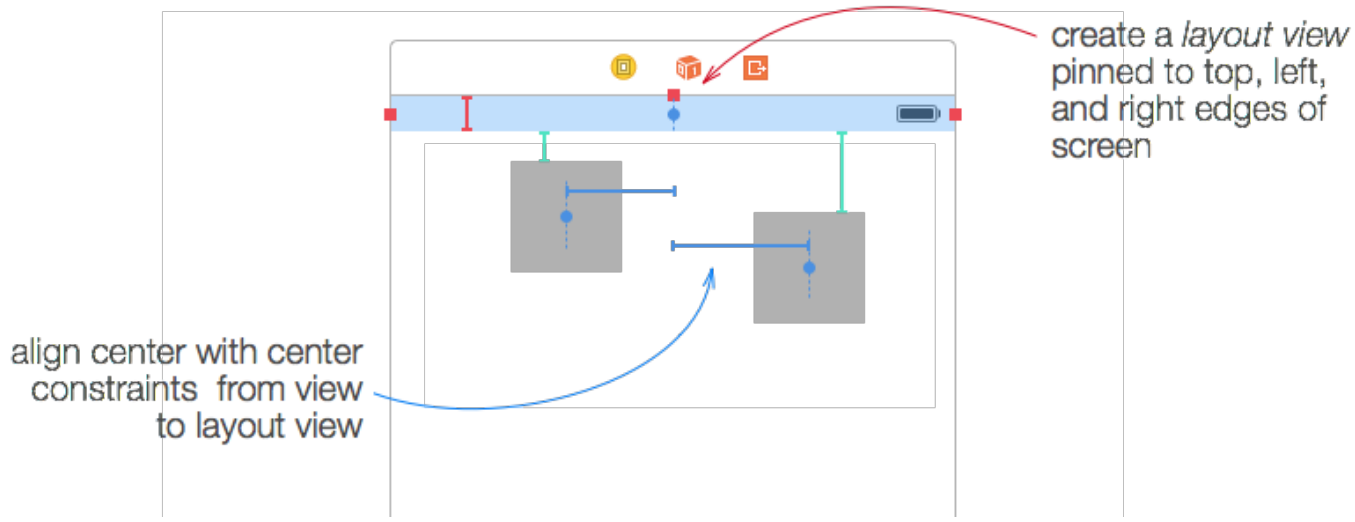
With this approach your constraints might look something like this:



Right now I prefer to keep my layout hierarchy as flat as possible. Another approach that avoids the use of a wrapper view is to use what I call a *layout view*.

For this approach I like to add a view to the storyboard that has a bold color but then set it's `hidden` property to `true`. You'll still see it in the storyboard layout (although slightly faded) but it won't appear when you run the simulator.

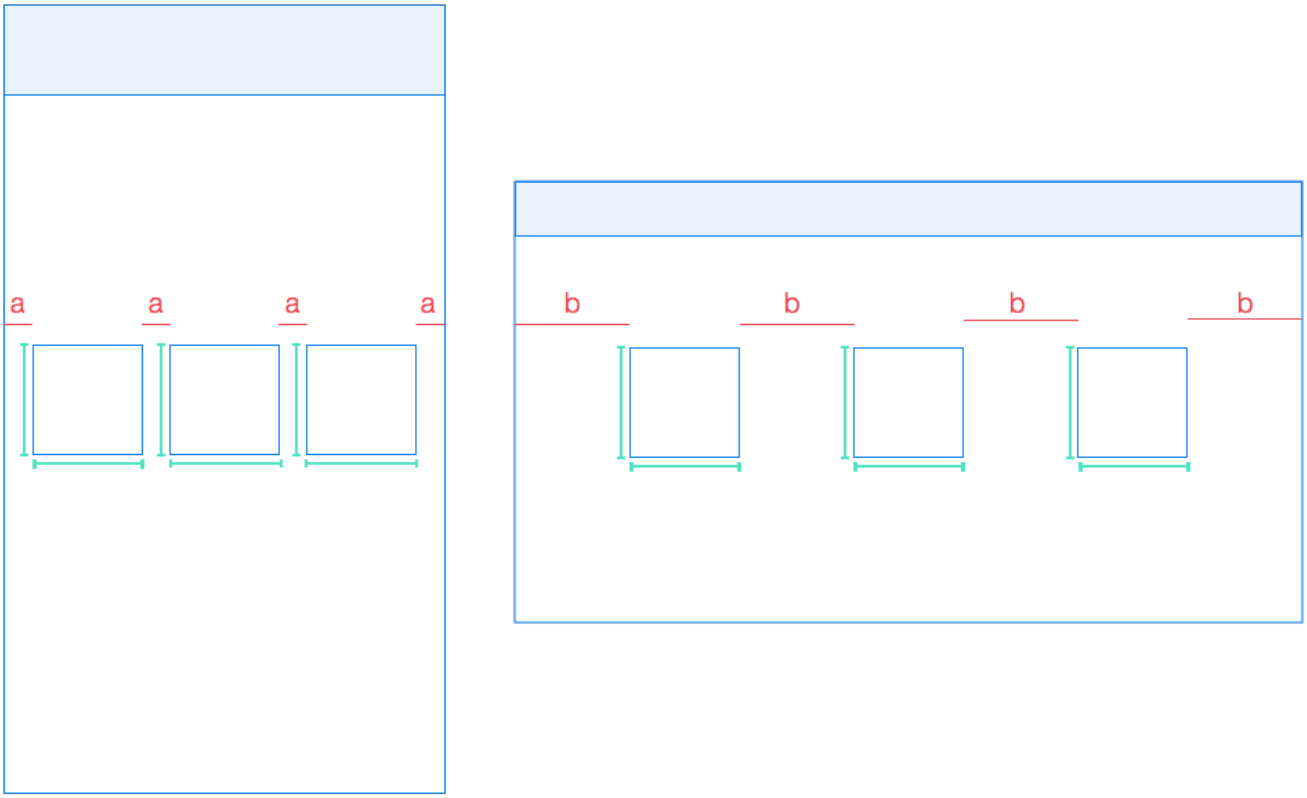
Now you can add *align center to center* constraints between your views and the layout view.



Spacer view example

Earlier we looked at how we can pin the gap from the edge of a view to its parent view so that when the parent view scales up the gap constraints pull on the view to increase it's size.

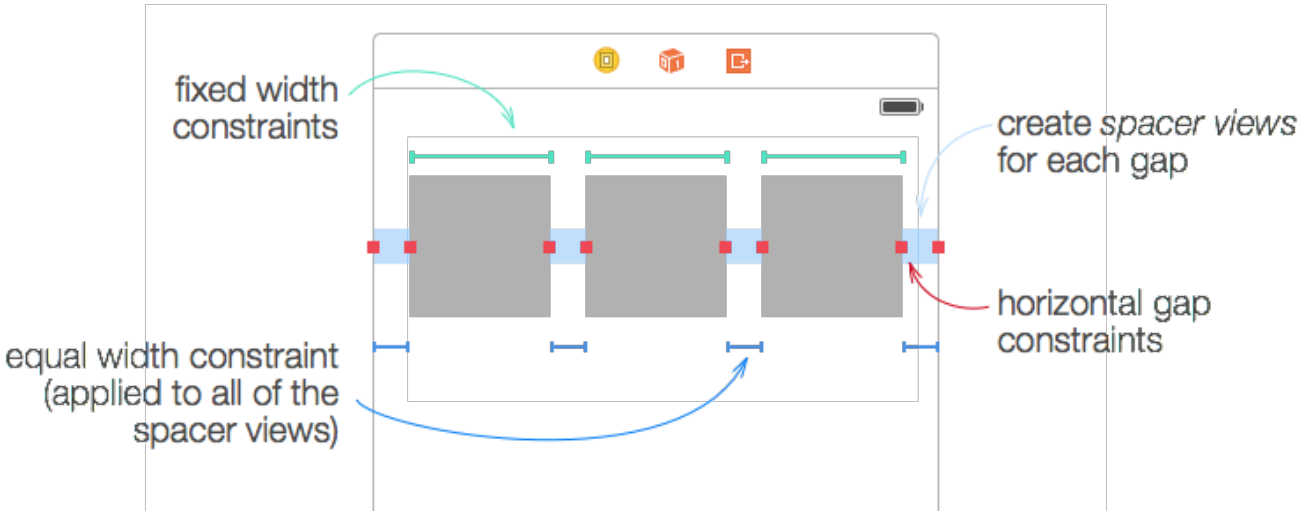
But if we want the size of the views to remain the same, and intend have the extra screen space used by increasing the size of the gaps between the views?



Conceptually, we want something like the *equal widths constraint* that we apply to views, but to be able to apply it to a constraint [15].

Instead we need to add *spacer views* which again are views that we'll add to the storyboard to help with our layout, but because we don't want to see them in the running app we'll set their `hidden` property to `true`.

The general approach with this technique is to add a spacer view for every gap you want to grow proportionally, pin all the edges of the views with your spacer views, and then add an *equal widths constraint* to all the spacer views.



Next steps

Hopefully by now you're confidant enough to start exploring layouts defined with constraints.

Don't expect instant success. Xcode is a complex tool with a lot of features that extend beyond storyboards and constraints so parts at first may seem mysterious and confusing. Like any complex tool it will take a while before you get familiar with it's quirks, but I'd recommend that you start getting familiar with it sooner rather than later as I predict (or at least I hope) that the next generation of designers will spend more time using tools like Xcode than Photoshop.

If you're a designer who creates design specs for iOS developers to follow build from you should start thinking about the best way to communicate what you want to your design team. Maybe you will create a visual language that describes the constraints you want applied.

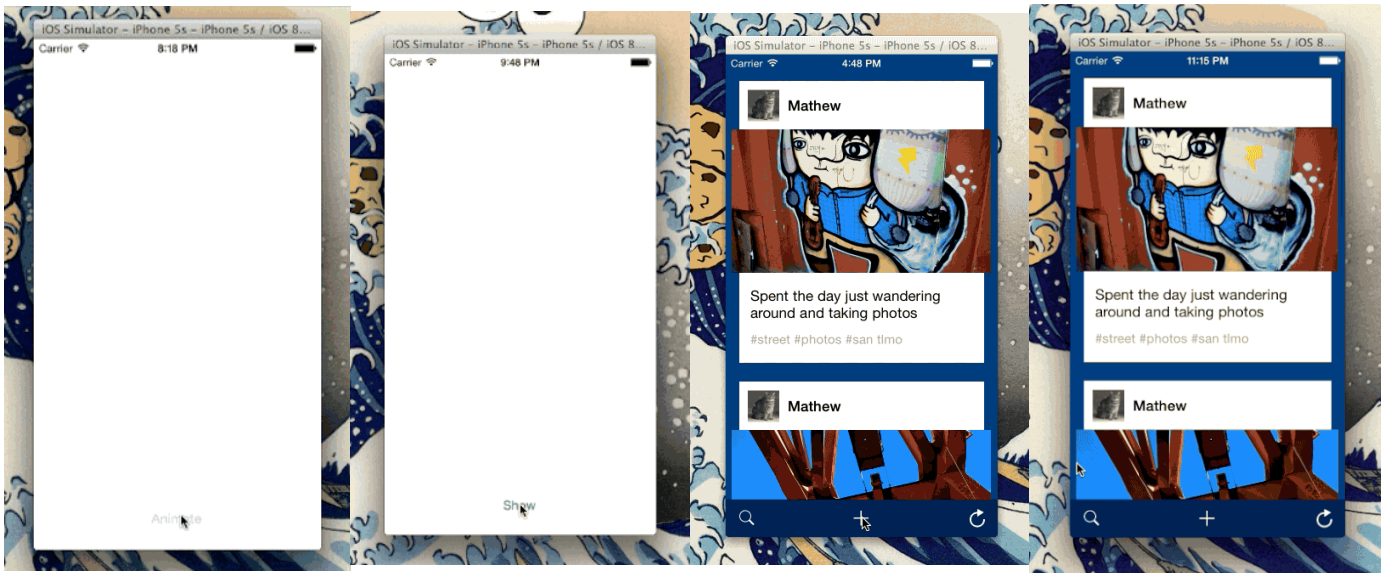
Personally I'm excited about the idea of designers taking more responsibility for the creation of their work and instead of having a front-end developer recreate the designers intention, the designer simply manages the storyboard file themselves.

If you found any part of this tutorial difficult to understand, or if there's something I completely missing please let me know and I'll try my best to make an update.

Related…

If you found this tutorial interesting you might also like my series on posts looking at creating animations in Swift.

Swift Animation Posts…



Animations Part 1 | *Transitions Part 1* | *Transitions Part 2* | *Transitions Part 3*

NOTES

1. By adaptive layouts I mean layouts that don't rely on knowing the specific dimensions of the screen, and can instead adapt to different widths and heights. ↩

2. Starting with Xcode 6, you can also use storyboards to layout screens for OS X apps. ↩

3. These definitions aren't specific to iOS but instead reflect a more general philosophy in computer science around organizing and structuring code called the Model-View-Controller (or MVC) pattern. In this pattern, software developers organize their code into three main parts:

    - The *Model(s)* which manages an abstract representation of the information in your app (often the model will manage a database that stores things like your username, your photos, your comments etc…);

    - The *View(s)* manage the presentation of the interface and capture any user input from people; and

- the *Controller(s)* that act like glue keeping the views and the model in sync with each other.

Technically, iOS uses a variant of this pattern that's called MVVC *Model-View-ViewController* but the general idea is the same.

Anyone can (and most apps probably do) create their own custom views but Apple gives us a bunch of views for 'free'. That's the difference between native components and custom components. Similarly, Apple also gives us a range of view controllers that are well suited for specific tasks (e.g. for managing a fixed view, or a tableview or a grid-based collection view layout). ↩

4. You can also add a navigation bar, toolbar or tab bar with these settings, but remember that any changes you make here are just to provide a simulation of available space in Xcode. E.g. If you're simulating a toolbar, it won't appear when running the app because you're only simulating the space it would take up if it did exist.

   Another size option is *freeform* which when selected you jump over to the size inspector panel and choose any specific width and height for the screen. This is useful for custom view controllers that don't take up the full size of the screen. ↩

5. A UIView is the most basic type of view. Basically it's just a colored area with a position and dimensions. ↩

6. Of course you can also add constraints in code which allows you do so somethings that aren't possible in storyboards. Luckily, the majority of things that we want to do are possible using storyboards alone. ↩

7. You can also have Xcode automatically add missing constraints to attempt to remove constraint errors but I've found that this can behave unexpectedly and I prefer to add constraints manually. ↩

8. There are also traits for resolution (e.g. @1× @2× @3×) and device idiom (e.g. iPad, iPhone). ↩

9. Prior to Xcode 6, developers needed to maintain a separate storyboard file for both an iPhone interface and an iPad interface. There also wasn't great support in storyboards for designing for both portrait and landscape orientations and so developers tended to support only the portrait version. ↩

10. I was excited in beta versions of Xcode 6 to see a feature hinting at smaller screen sizes that might of hinted towards storyboard support for Watch apps. But it's more likely that like the TV apps for this platform will not have storyboard support. ↩

11. It is possible to target layouts that aren't covered in these options, but to do this you'll need to create constraints in code which is outside the scope of this post. ↩

12. For the label height to grow to fit the content we need to set it's `lines` property to `0` which is

a code for ‘unlimited’ lines. Setting a non-zero value will force the label to to have that many lines. If we’d kept `lines` at the default value of `1` then instead of increasing it’s height, the label would just truncate any text that doesn’t fit. ↩

13. Again there are multiple ways to achieve this. The approach I took was pinning the outer edges of the secondary images with the main image, pinning the gap between the secondary images, adding an equal widths and aspect ratio constraints to both images. ↩

14. Some people would compare them to CSS hacks or the use of single-pixel images to layout HTML in the 90s. Personally I don’t think it’s that bad, but I do hope that future versions of Xcode will bring a wider range of constraints to use so these approaches will become obsolete. ↩

15. It would be nice if you could add constraints to constraints, then you could add a horizontal gap between these objects and add a constraint that they grow proportionally. But it could also quickly become a constraint-sphagetti nightmare… ↩

LATEST POSTS

Designing in the browser: data-driven prototypes & living style guides
Hello, computer: conversational interfaces
The search for the perfect process
Simulated realities, distorted realities
The curse of the local maxima / thinking tools for design
More…