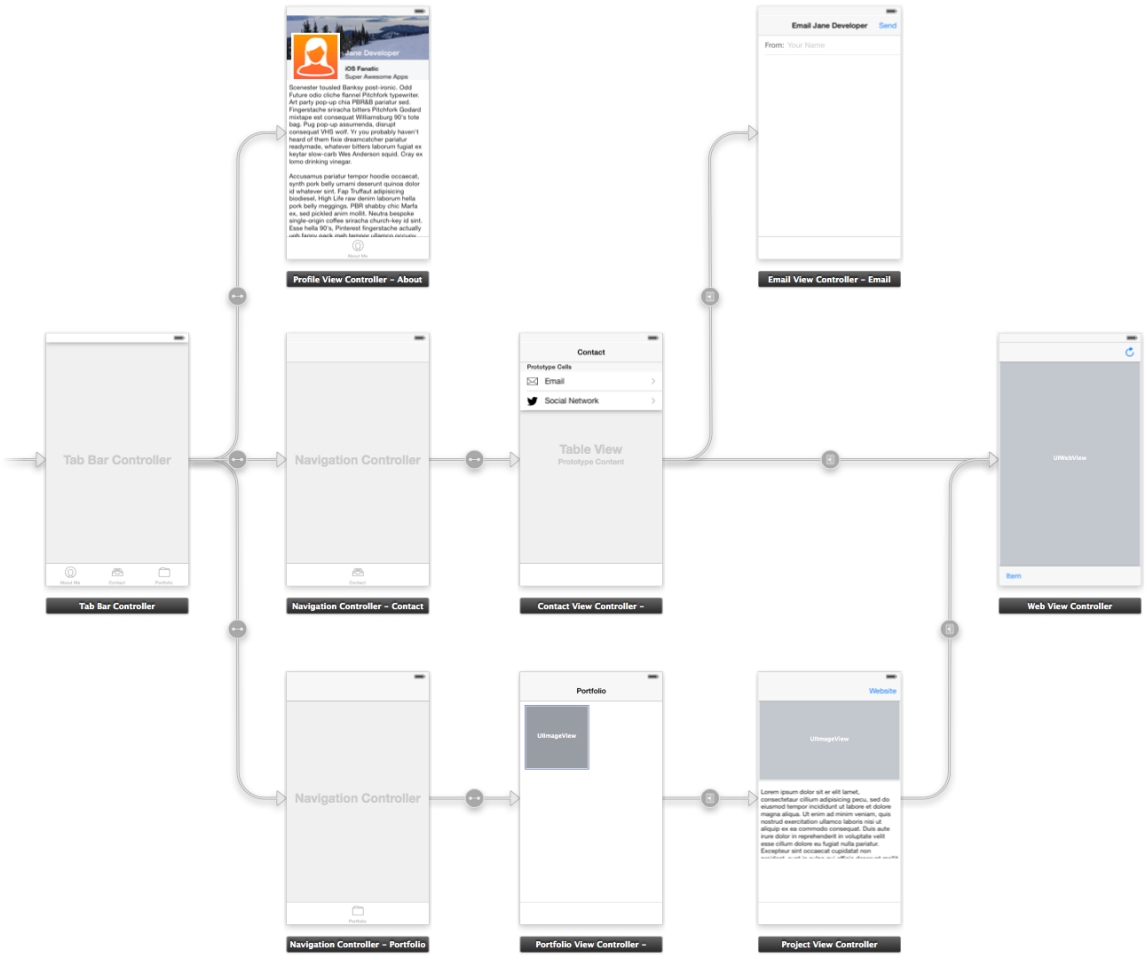
You check your iPhone multiple times a day, and so do over **575 million** other people. This tutorial will help you get started on your journey to becoming an app maker.

Over the duration of this course, you should become familiar with:

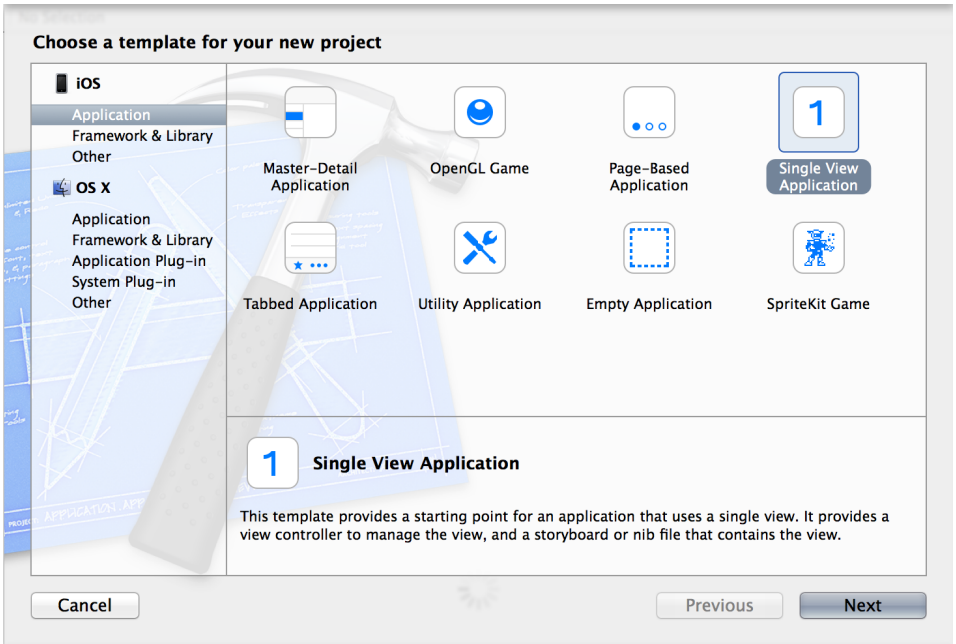
- **Xcode:** How to navigate an Xcode project, look up documentation, and much more.
- **iOS Fundamentals:** What it means to make an iOS app, and common design patterns.
- **Storyboards:** How to lay out your app's scenes, transitions and navigational flow.
- **View Layout:** The ins and outs of laying out views within a scene.
- **Coding With MVC:** The basics of building custom views, view controllers, and models.

To accomplish that, we will be building a resumé app to teach these fundamentals — and also to show off what you have learned!

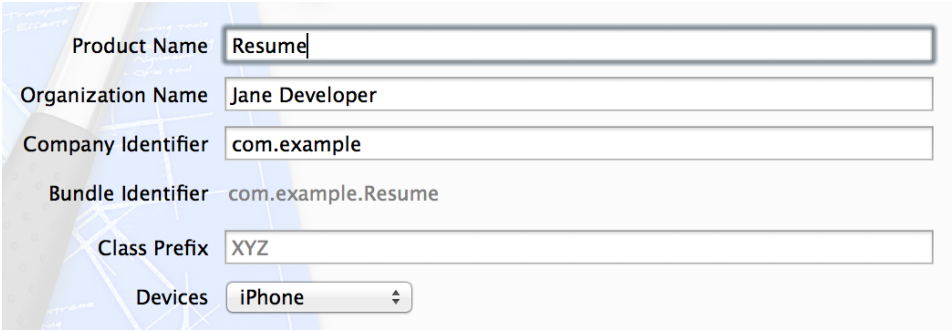
The app is broken up into three major sections: Your profile, Contact (email; social networks), and a portfolio.



Open up Xcode (you can install it via the Mac App Store), and create a new project (File > New Project). You'll be greeted by the project template chooser:

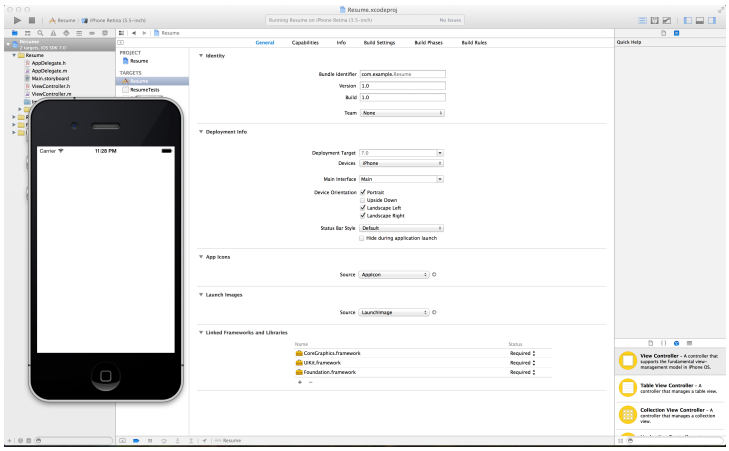


For this project, we will start with a **Single View Application** (click Next). You will then need to fill in basic project information:



- **Product Name:** Used throughout the project to identify and name your app.
- **Organization Name:** Company or developer name.
- **Company Identifier:** A reverse domain name used to uniquely identify your app in the app store when combined with the product name.
- **Class Prefix:** A short prefix to help avoid confusion with other project's code. You generally want to **leave this blank** for application projects.
- **Devices:** iPhone, iPad, or both (Universal). Choose **iPhone** for this project.

Hit **next**, and you will be greeted with your newly created project!

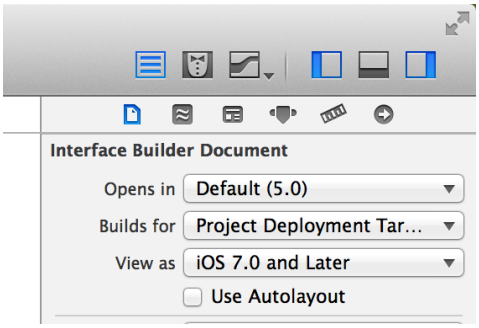


Bask in its glory for a moment, and then hit the play button (top left) to make sure that everything worked out.

The simulator will launch, and you will be greeted by a blank white iPhone app. Success!

Simplifying

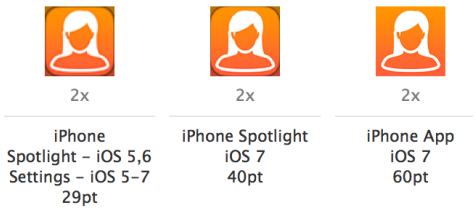
To avoid diving straight into the deep end, we are going to change a couple settings:



- **Uncheck both Landscape options** for the device orientation on the project view.
- Select **Main.storyboard** and **uncheck Use Autolayout** in the file inspector.

Beautifying

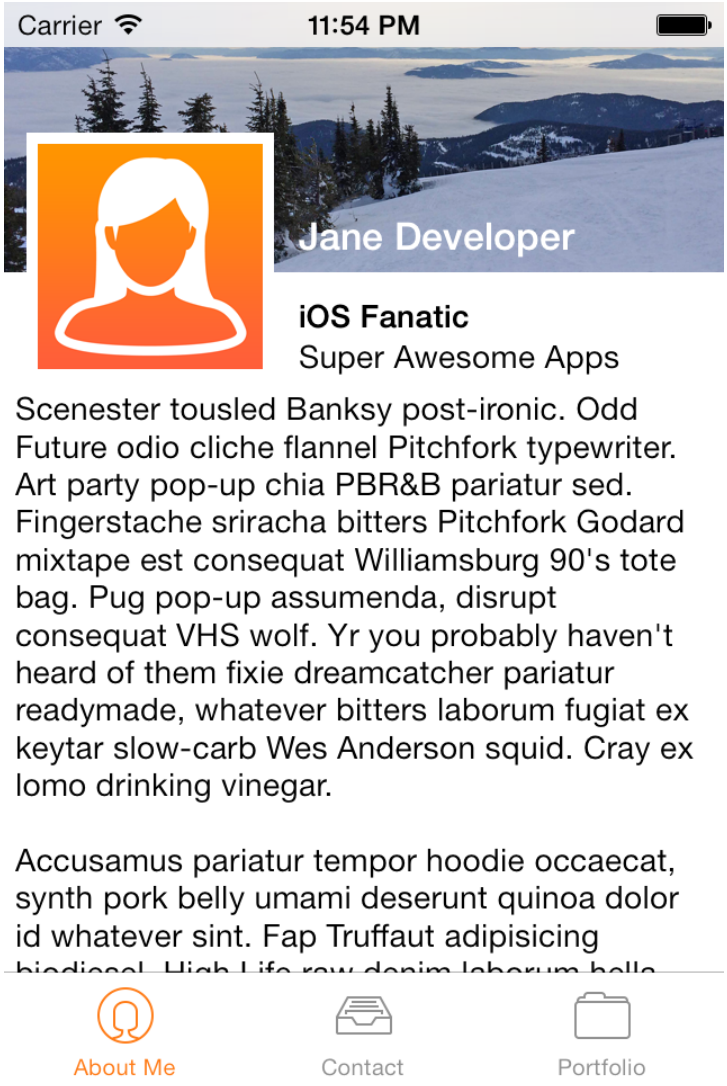
Let's add some icons! Open the **Images.xcassets** asset catalog (which will eventually contain all the images for your app), and select **AppIcon**.



Drag in icons of the appropriate sizes (note that a pt is 2px, so 58px, 80px, and 120px). You can find some pre-cut app icons in the Male and Female folders under **Prefab Assets**. Note that @2x indicates a retina image, and Xcode honors that convention.

Hit the **play** button again (or ⌘R) to rebuild and relaunch the app. Click the **home button** on the simulator (or ⇧⌘H), and your app should be using the new icon!

Now it's time for some real dev work :) We are going to build out your profile view into something like the screenshot on the right.



Assets

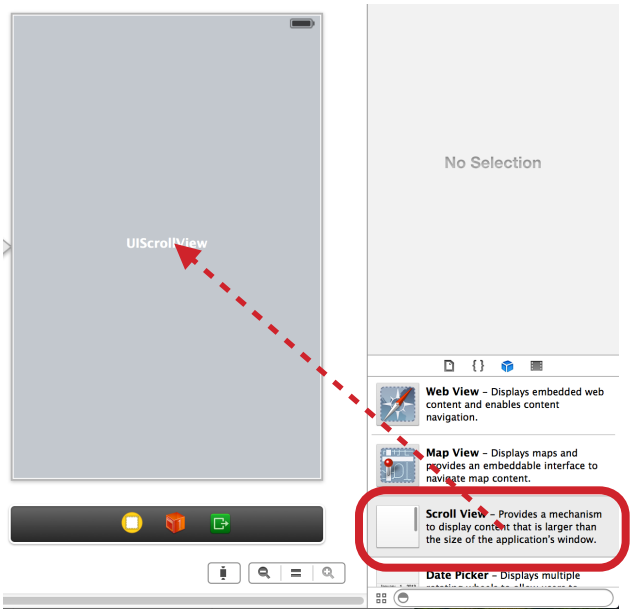
First, let's get our assets in order. Open **Images.xcassets** and drag in a background image, as well as a larger photo for your profile picture (200x200px is a good size).

You can find some prefab images: **ProfileBackground@2x.png** and the **ProfilePhoto.png+ProfilePhoto@2x.png** in the Male or Female folders.

Make sure that the images are named **ProfileBackground** and **ProfilePhoto** (you'll be referencing them by name).

View Layout

Open **Main.storyboard**. The storyboard is where you will manage most of the visual and navigational design of your app. You should see a single scene (view controller).



Ok, now we're really ready for some real dev work.

Find the **object catalog** (bottom right), and within it the image view. Drag a **scroll view** onto the blank view controller. Make sure that it exactly covers the available space.

Background Image

Next up: drag an **image view** onto the scroll view, and resize it so that it takes up part of the top of the scene. This is the background image.

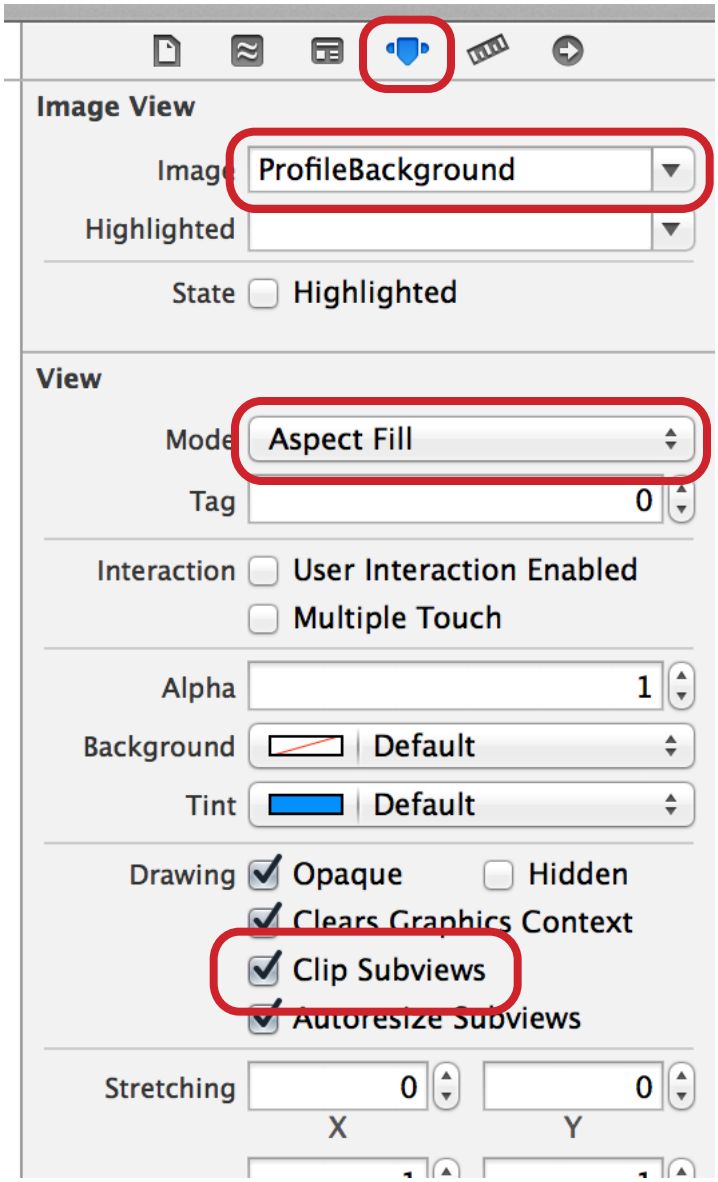
We have a few settings to set for the background image. Open the attributes inspector (a section of the utilities pane,

Set the image to **ProfileBackground**, and you should see your image filled in!

Additionally, change the mode to **Aspect Fill** so that the image is not stretched.

Thirdly, make sure that you check **Clips Subviews**. When image views are rendered, they resize themselves to fit the image directly.

The clip subviews setting disables that (even though - confusingly - an image view has no subviews).



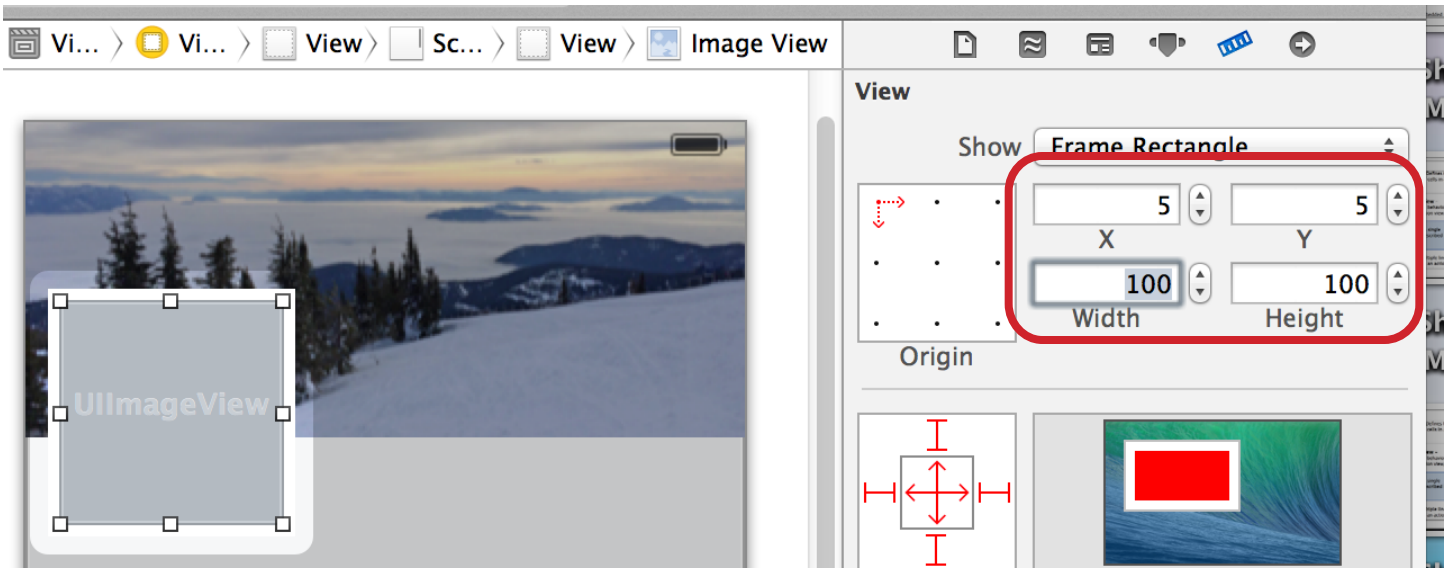
Profile Photo

Drag a simple **View** from the object library and place it where you wish your profile photo to be. This view will act as a white border around your photo.

Resize the border view to **110x110px**. That was pretty painful to drag to a precise size, wasn't it? :(

Now drag an **Image View** inside the border view. It will fit to the same size as the border view.

You could resize the image view by dragging the corners, but that's annoying! Select the image view, and switch the utilities pane to the size inspector:

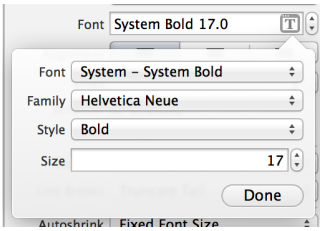


Set the size of the image to **100x100px**, and set its x and y values to **5px**. This gives us a nice 5px border around the image.

Finally, set the image to your profile photo, and you're one step closer to world domination.

Labels

The next type of view we will use is a label. Drag a **label view** out to make your name. Let's be fancy and place it over the background.



Ugh, black text looks awful! Open the attributes inspector, and change the color to white.

You probably also want a bold font, which can be switched via that strange T icon/button in the font field.

Let's fill in some details about yourself. Drag out a few more labels to fill the space below the background image, but to the right of the profile photo.

Perhaps something like this:

Extended Details

The last view we will add to the scene is a text view containing a longer summary about you. Or, if you're feeling really lazy, a treatise on Lorem Ipsum.

Drag a text view onto the scene, placing it below your labels, but tall enough to fill the remaining space. Like so:

You'll also want to disable the **editable** and **selectable** options so that the view is static text.

Not too shabby!

Extra Credit

- Read through the list of views (and other things) in the object library. It's good to know what you've got in your tool belt.
- Spice up the scene with some more views, images, and whatnot!
- Select your various views, and read through their available attributes.
- Rename all your views to something meaningful. It helps, trust me.
- Try switching the description text view to attributed text, and add some extra formatting to it.

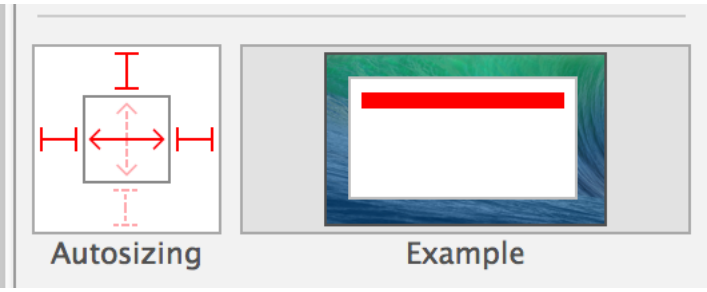
If you ran your project in the simulator, chances are that it looked awful: the background image probably shifted up. Agh!

By default, Xcode launches the simulator as a 3.5 inch iPhone (e.g. iPhone 4/4S). Now, you could just switch it to a 4 inch phone and call it a day, but that's cheating.

Let's make our scene resilient to varying screen sizes!

Background

Select the background image, and switch to the **sizing inspector**. The behavior we're going for is that the background should stay a fixed height, and not move.



This can be specified by changing the autoresizing settings so that the bottom and height values of the view are not pinned.

Testing It

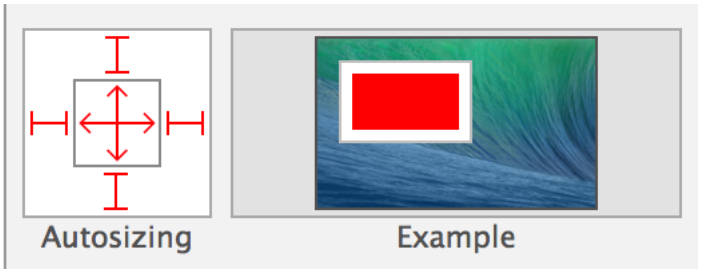
You could launch the simulator to see the fruit of your labor, but Xcode gives you an easier option.

Click on the form-factor toggle button located in the bottom-right-ish portion of the storyboard editor.

Yeah!

Description

Select the description text view, and toggle the size of the container. Notice how it gets cut off. Now, run it in the 3.5" simulator, and scroll the description text. See how it's cut off?



To fix this, we want the description view to resize completely w/ the screen.

Status Bar

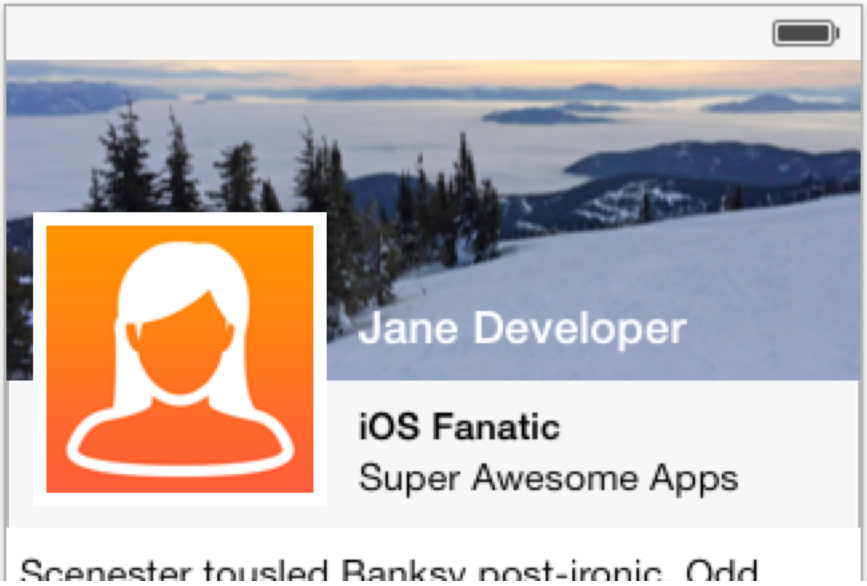
One tricky aspect of iOS 7 is that it forces you to use your app's content as a background to the status bar.

Chances are that the status bar doesn't look great on your chosen background. One way we can get around this is to place another view directly under the status bar.

Rather than using a generic view, though, it'd be really cool if we could get the translucent effect of a navigation or toolbar. Easy: use a toolbar without any buttons.

Note that using a toolbar like this is a bit of a hack, but it's the best option we have currently. Apple may be adding in status-bar specific background support in a later release.

Ok, drag a **toolbar view** onto the scene. Click on the item within it, and **delete that extra item** so we have a pristine toolbar. *You may need to double click slowly, or use the view hierarchy browser.*



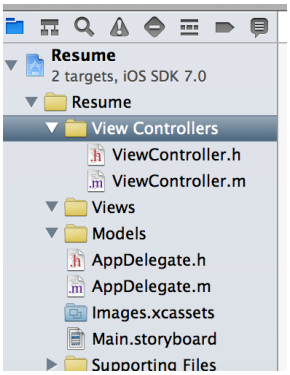
Finally, position the toolbar behind the status bar. Set the y value to -24 (status bars are 20px tall). You will also need to fix the autoresizing options so that it is pinned to the top of the scene, rather than the bottom.

Extra Credit

- Turn landscape mode back on for the project, and see how your views behave when rotating the simulator (⌘→, ⌘←). Fix any autoresizing settings that are broken!

From a design perspective, having our description text scroll independently isn't ideal. This will become especially problematic once we add tabs to the app.

So, for this section, our goal is to make the entire profile page scroll. To do this right, we need to pull up our sleeves and dive into some code. Finally!



Organization

Since we know that we will be writing a fair bit of code to support this app, let's organize our code into a few groups: **View Controllers**, **Views**, and **Models**. To add the groups, right click on the project group > New Group.

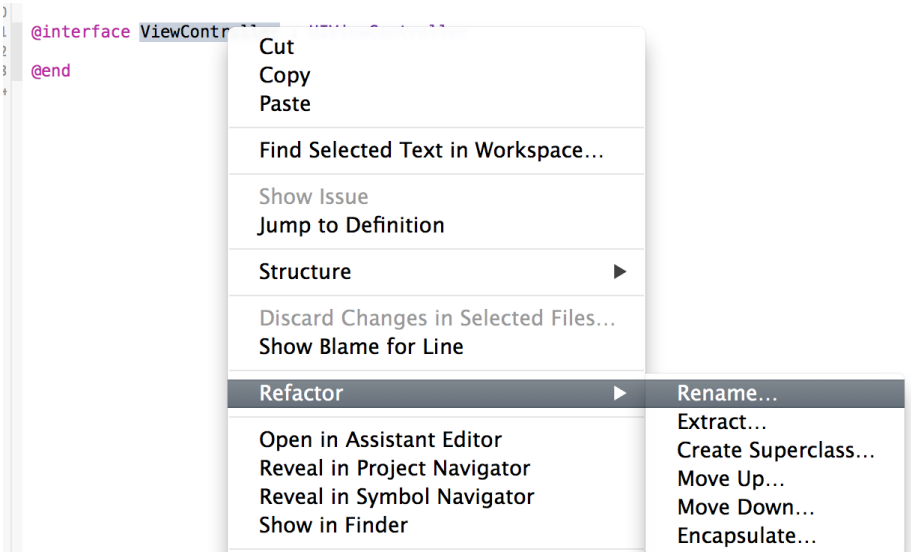
Move **ViewController.h** and **ViewController.m** into the **ViewControllers group**.

Renaming Things

Since we will have multiple view controllers, the name ViewController is a bit too generic. Let's rename it.

Wait! Don't just rename the file! There is also a class defined within it by the same name — and the storyboard also references it.

Xcode gives us a refactoring tool that makes renaming things en-masse easy. Open the **ViewController.h** header. Right click on the **ViewController** class > **Refactor** > **Rename**



Let's rename the class to **ProfileViewController**, and click **next**. Xcode will give you a summary of the changes it will make. Click through them, and when you're satisfied, click **Save**.

Also, this time only, Xcode will prompt about whether you want snapshots. Might as well! Click **Enable**.

It's a good habit to make sure everything works after a refactor: Load your app in the simulator and make sure all is well.

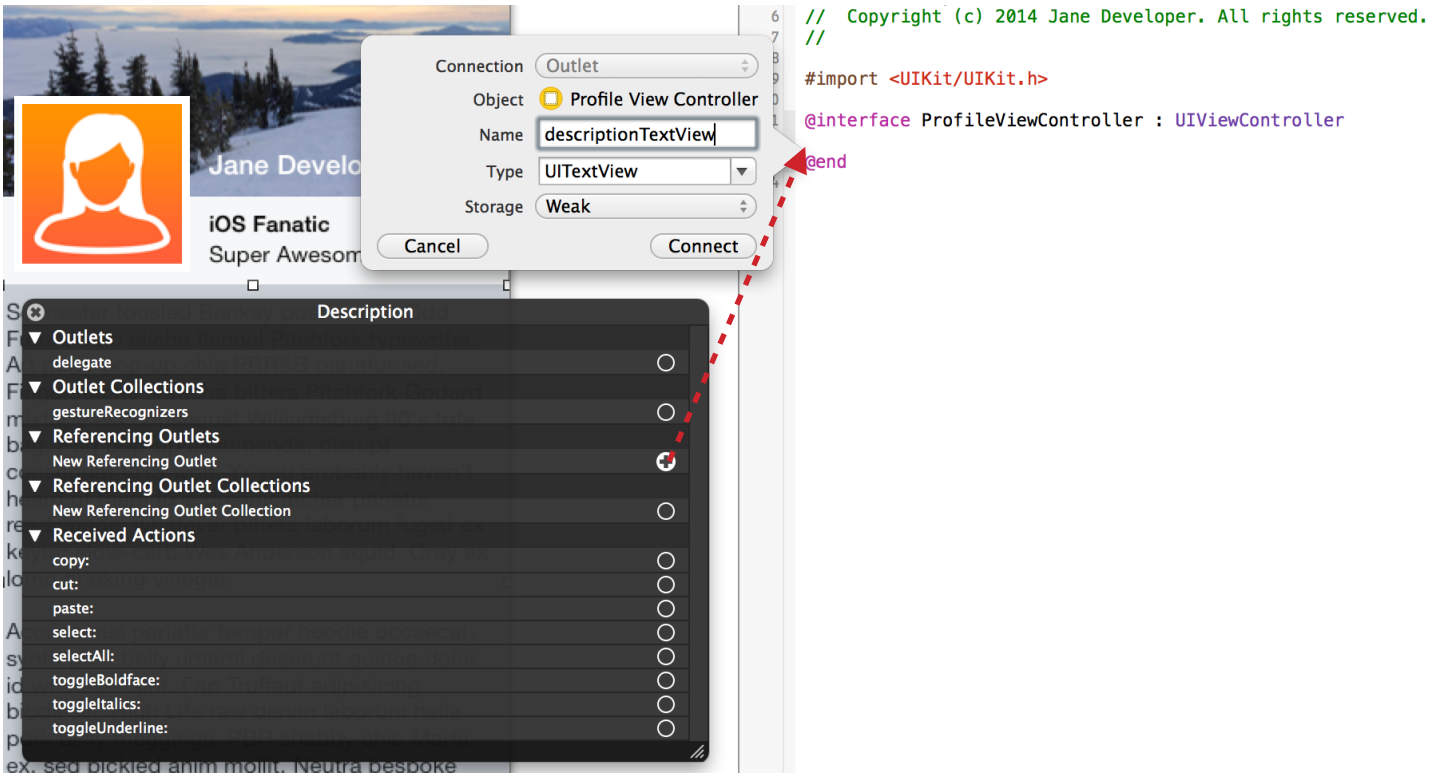
Tying Code To UI

Scroll views are fancy - if their content is smaller than their viewport, they do nothing. In order to make our top level scroll view behave, we need to stretch the description text view so that it is exactly as tall as its content (so it, also, will not scroll).

We need to make our **ProfileViewController** aware of the description view so that it can resize it properly on demand. We need to switch Xcode into assistant mode (split pane editing).

Click the assistant **mode toolbar button** (a tuxedo icon), and open **Main.storyboard**. Select our profile scene, on the right, ProfileViewController.m is probably open. If not, switch to **ProfileViewController.m** by selecting the file drop down in the header.

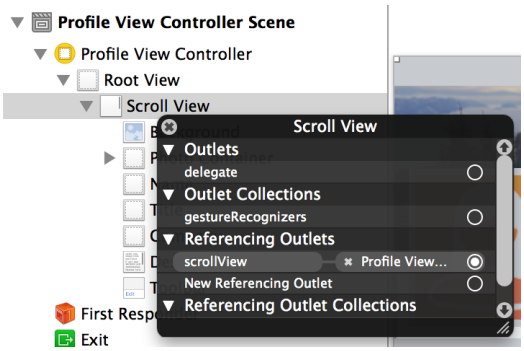
Now, **select** and then **right click the description text view** to bring up its outlet panel. Drag from the dot next to **New Referencing Outlet** into the other editor and insert it **between @interface and @end**.



Call the outlet **descriptionTextView**.

This did two things: It declared that ProfileViewController maintains a reference to a UITextView and that when the storyboard loads your UI, it should associate the text view you created with the descriptionTextView property on the controller.

Do the same thing for the scroll view. This time you'll have to right click the **Scroll View** in the view hierarchy, and drag a **New Referencing Outlet** into the ProfileViewController's class header. Call it **scrollView**.



While you're here, make sure to **uncheck Scrolling Enabled** for the scroll view (in the attributes panel).

The @interface block within ProfileViewController.m should now look like:

```
11  @interface ProfileViewController ()
12
13  @property (weak, nonatomic) IBOutlet UITextView *descriptionTextView;
14  @property (weak, nonatomic) IBOutlet UIScrollView *scrollView;
15
16  @end
```

Making The Scroll View Behave

Now that we can reference the scroll view and description view, we can get down to business.

Inside the **@implementation** section for ProfileViewController, we need to implement a method called **viewDidLayoutSubviews**. This is called whenever the view managed by the view controller rearranges its subviews. Typically, that is when the view is first loaded, and whenever it resizes (i.e. the device rotated).

```
- (void)viewDidLayoutSubviews
{
```

Our first order of business is resizing the **descriptionTextView's frame** to match the size of its content. Within viewDidLayoutSubviews, we want the following snippet:

```
    // Force the text view to resize to fit its content.
    CGRect descriptionFrame = self.descriptionTextView.frame;
    descriptionFrame.size = [self.descriptionTextView sizeThatFits:self.view.bounds.size];
    self.descriptionTextView.frame = descriptionFrame;
```

This reads the size of the text view, asks it what size best fits our layout and then updates it. Note that we use **self.view.bounds** instead of self.view.frame because we might be mid-rotation.

Now that all our views are properly laid out, we need to let the scroll view know just how much content it has.

Let's calculate the bottommost point of our views:

```
    // Calculate the scroll view's content size, since all our subviews are of
    // a known size and position.
    CGSize scrollContentSize = self.view.frame.size;
    // We walk through all subviews and find the bottommost point: That's our new height.
    for (UIView *subview in self.scrollView.subviews) {
        CGFloat viewBottom = CGRectGetMaxY(subview.frame);
        if (viewBottom > scrollContentSize.height) {
            scrollContentSize.height = viewBottom;
        }
    }
```

And finally, let's let the scroll view know:

```
    self.scrollView.contentSize = scrollContentSize;
```

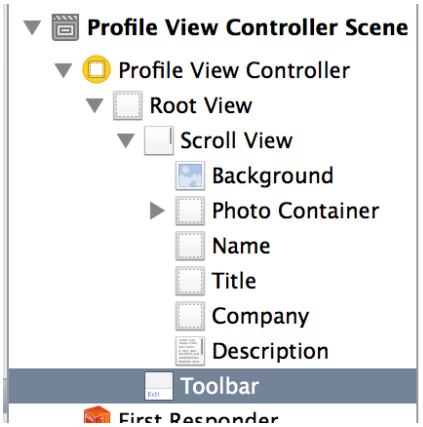
And just to make sure, the final function should be:

```
- (void)viewDidLayoutSubviews
{
    // Force the text view to resize to fit its content.
    CGRect descriptionFrame = self.descriptionTextView.frame;
    descriptionFrame.size = [self.descriptionTextView sizeThatFits:self.view.bounds.size];
    self.descriptionTextView.frame = descriptionFrame;

    // Calculate the scroll view's content size, since all our subviews are of
    // a known size and position.
    CGSize scrollContentSize = self.view.frame.size;
    // We walk through all subviews and find the bottommost point: That's our new height.
    for (UIView *subview in self.scrollView.subviews) {
        CGFloat viewBottom = CGRectGetMaxY(subview.frame);
        if (viewBottom > scrollContentSize.height) {
            scrollContentSize.height = viewBottom;
        }
    }

    self.scrollView.contentSize = scrollContentSize;
}
```

If all went well, you should be able to build and run the project. Hit the play button, and scroll that profile view. Magic!

Hmm, a couple things are off, though: The status bar background scrolls with the view, yuck! Also, tapping the status bar doesn't scroll to top :(

Pinning The Status Bar Background



We need to move the toolbar we're using as the status bar background *outside* of the scroll view.

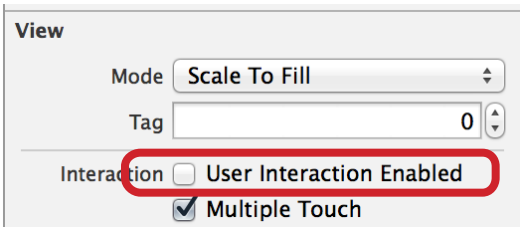
In the storyboard's view hierarchy, drag the toolbar to be a direct descendant of the top level view. Note that you want it to be **below** the scroll view in the list. Views that come later in the list are rendered on top of those before them.

Unfortunately, reparenting a view causes Xcode to drop it in the center of the screen, so we need to move it up. You'll be tempted to drag, but **do not drag** the toolbar! Xcode will attempt to place it back within the scroll view if you do.

Instead, use the size inspector and reset its y value back to **-24**.

Fixing Tap To Top

Tap to top has an annoying property on iPhones. If more than one scroll view is active on screen, it is completely disabled.

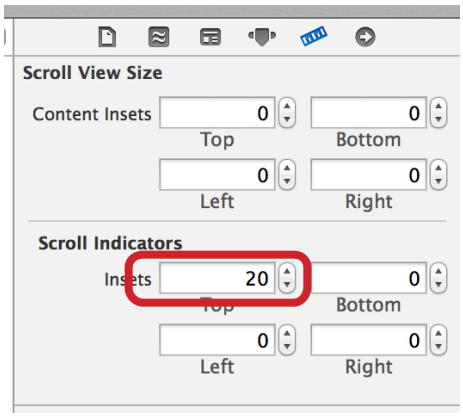


In our case, our custom view *and* the description text view have active scroll views (even if their content fits). The simplest way to deal with this is to disable all touch interactions for our description text view.

Run the app, and everything should be better! …almost.

Scroll Indicators

The last niggling issue is that the scroll indicators are now behind the status bar (and obscured by our background for it).



Thankfully, this is pretty easy to fix by specifying custom insets for the scroll view. On the size inspector, change the **scroll indicator top inset** to **20**.

Run the app again, and make sure it all checks out. Awesome. You're done with the profile view controller!

Extra Credit

Make the background image bounce with the scroll view when the user pulls down. In order to do this, you'll need a few things:

- You'll need to create an outlet for the background image so that it can be referenced by your view controller's code.
- The view controller will need to implement the UIScrollViewDelegate protocol. You can indicate this by changing the interface definition in the header to

```
@interface ProfileViewController : UIViewController <UIScrollViewDelegate>
```

- You will need to set the delegate of the scroll view to be the view controller. A good place to do this is in viewDidLoad:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.scrollView.delegate = self;
}
```

- And finally, you will want to implement the did scroll delegate method, which is called for every frame during a scroll.

```
- (void)scrollViewDidScroll:(UIScrollView *)scrollView
```

- Use self.scrollView.contentOffset.y to access the current scroll offset.