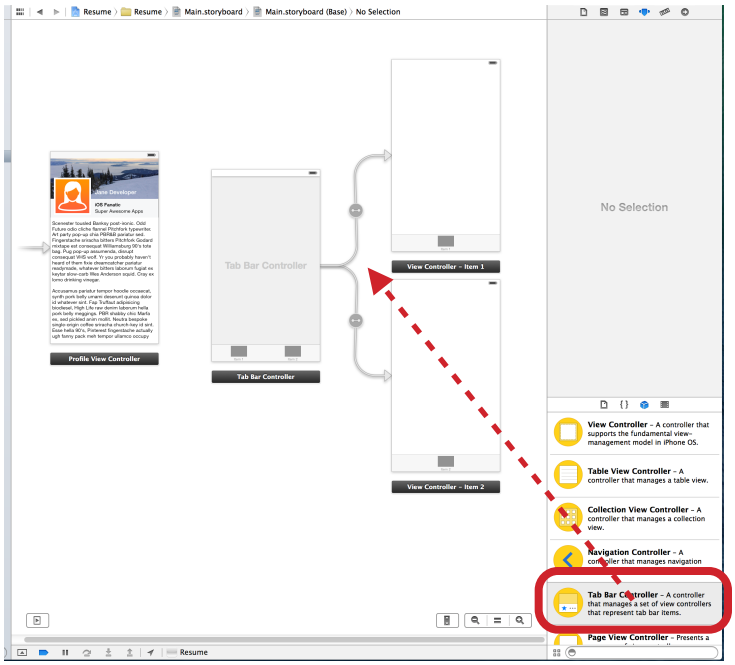
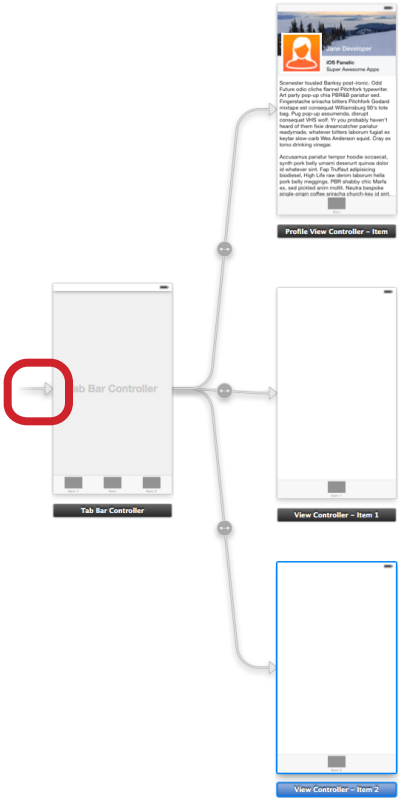
Now that we've polished the profile scene, it's time to move on to adding some navigation to our app so that it can be extended.

Recall that we want to provide some contact info (LinkedIn, GitHub, email, etc), and also a portfolio. This screams for tab based navigation, since we have only a few high level sections of the app.

So, open up the storyboard, and drag a **Tab Bar Controller** onto a blank area of the canvas. You should end up with a storyboard similar to the figure on the right.





Note that the profile scene is not yet wired into the tab view controller:

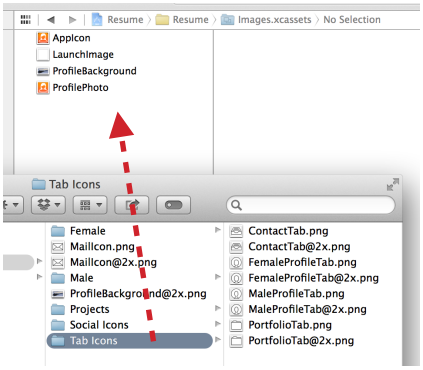
- **Right click** on the **tab view controller** to open its outlets panel. Drag from **view controllers** to the profile view controller to add the third tab.
- **Drag the arrow** that is pointing at the profile view controller **to the tab bar controller**. This indicates which view controller is loaded when the app first launches.

Try organizing your view controllers (try lining up each tab's view controller so that the flow makes sense). You should end up with a scene that looks something like the one on the left.

Run your app and click around! You should have two blank tabs, and your profile within a tab.

Tab Presentation

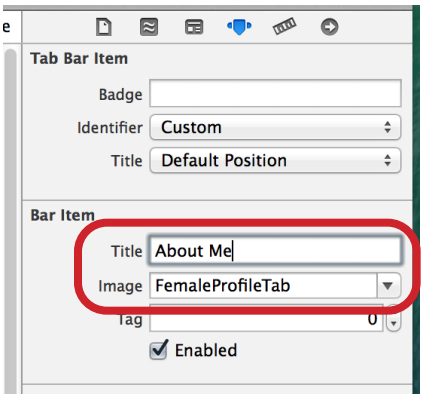
Blank tabs are just a smidge confusing, so let's give them icons and labels.



We need to load our tab images into the app via the asset catalog, just like the rest of our images. So, open **Images.xcassets** and get the project folder handy for dragging. You want to drag the **Tab Icons** folder from finder into the asset catalog.

This should result in a Tab Icons folder within your assets catalog, and 4 icons. Good to go!

Switch back to the storyboard, and zoom in to the profile view controller (double click on it). Select the placeholder **tab** at the bottom, and view its attributes.



Fill in the details for the Bar Item section. Give it a title, and choose the image you wish to use.

Go to the other two view controllers, and set their tabs. We want a "Contact" and "Portfolio" tab.

Chances are that your tabs are in an awkward order!

You can reorder your tabs by dragging them around within the tab bar controller. Whew.

Run your app, and make sure that all is well.

Re-arranging your view controllers into a different navigational structure can break some previous assumptions you might have had. There are a few minor things that we should tweak before we forget them:

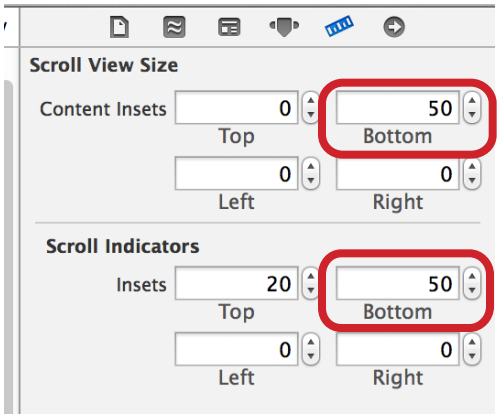
Content Under Navigation & Toolbars

Run the app, and scroll the profile view up and down. Notice how the scroll view is hidden behind the tab bar? Yuck!

We have two options: We can modify the scroll view's insets so that it applies extra padding. shift our profile view up so that it doesn't go under the tab bar at all. You're going to do both and go with which ever option you like most.

Option One: Edge Insets

This one's easy (and the translucent effect is nice). Select the **scroll view** of your profile view controller.



The height of a tab bar is 50px, so we need to set the **bottom content inset**, and the **bottom indicator inset** to **50px**.

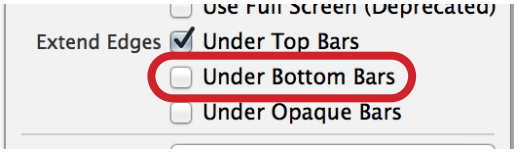
Note that scroll views will normally do this automatically, but because we are directly managing ours, we have to also supply insets.

Option Two: Content Shifting

For option one, open the attributes inspector of the profile view controller. **uncheck Under Bottom Bars** for the Extend Edges section.

Run the app and notice how the scrolling is fixed, but the tab bar is a bit darker. The darker tab bar is due to there being no view behind it (the default color is black).

We can fix this by making the tab bar opaque. Head to the **tab bar controller**'s attribute inspector. Switch the **Bottom Bar** mode to **Opaque Tab Bar**.



Application Tint

A large change in iOS 7 is that each app should pick a relatively unique color to be used as a tint for various controls used throughout (buttons, tabs, etc).

In order to set the tint, we need to drop down to code. The place to do it is inside the application delegate. Open **AppDelegate.m**.

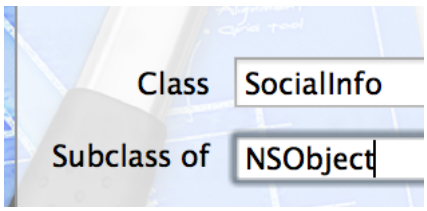
This file is responsible for all the lifecycle events of your app (just like a view controller is responsible for its views). In our case, we want to add a line of code to the application:didFinishLaunchingWithOptions: method:

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:
    (NSDictionary *)launchOptions
{
    self.window.tintColor = [UIColor colorWithRed:1.0 green:0.49 blue:0.098 alpha:1.0];
    return YES;
}
```

This gives us a nice orange tint. Note that you specify color channels via values that range from 0.0 to 1.0 (divide your normal color values by 255).

Our next goal is to provide a list of social networks (and links) that we want to advertise ourselves via. This means data, and data means models. Let's make a simple model that can house our social info.

Right click the **Models group** in the file view and choose **New File** (Create a Models group if you don't have one). Choose **Objective-C Class** and hit next. Name the class **SocialInfo**, and make it a subclass of **NSObject**. Hit next and create it.



Trivia: NS stands for NeXTSTEP, which was an operating system built by NeXT — the company that Steve Jobs founded after being ousted from Apple. Apple later bought NeXT, and much of NeXTSTEP's code became the foundation for OS X and iOS. Talk about legacy. This code has been around since the mid-80's.

Properties

Since models are focused on keeping track of your data, our first order of business is defining what data a SocialInfo model contains. Open the **SocialInfo.h** header. We want the following lines inside the @interface section of it:

```
@property (strong, nonatomic) NSString *title;
@property (strong, nonatomic) NSString *url;
@property (strong, nonatomic) UIImage  *icon;
```

Additionally, since we are going to be managing multiple SocialInfo objects, we will likely want a global list of them. This is as convenient a place as any. We will add an allSocialInfo method to the class that will return it:

```
@interface SocialInfo : NSObject

@property (strong, nonatomic) NSString *title;
@property (strong, nonatomic) NSString *url;
@property (strong, nonatomic) UIImage  *icon;

+ (NSArray *)allSocialInfo;

@end
```

Note that the + indicates that allSocialInfo is a class method (i.e. it is called via [SocialInfo allSocialInfo], rather than a method on an instance of SocialInfo).

Remember that .h files are headers, and only describe the *intent* of your class. We need to switch to the .m file to fill in its actual *behavior*.

@property definitions do not need any extra implementation, but our allSocialInfo method does. Note that it returns a NSArray (an array is a list of items). First, however, let's write a quick function to make constructing SocialInfo objects easier:

```
+ (SocialInfo *)socialInfoWithTitle:(NSString *)title iconNamed:(NSString *)iconName url:(NSString *)url
{
    SocialInfo *info = [self new];
    info.title = title;
    info.icon = [UIImage imageNamed:iconName];
    info.url = url;

    return info;
}
```

Now, we can implement allSocialInfo, filling in the social networks that we wish to display. Feel free to customize the list as you like:

```
+ (NSArray *)allSocialInfo
{
    static NSArray *all;
    if (!all)
    {
        all = @[
                [self socialInfoWithTitle:@"Bēhance"
                                 iconNamed:@"BehanceIcon"
                                       url:@"http://behance.net/janedev"],

                [self socialInfoWithTitle:@"DeviantArt"
                                 iconNamed:@"DeviantArtIcon"
                                       url:@"http://janedeveloper.deviantart.com/"],

                [self socialInfoWithTitle:@"Dribble"
                                 iconNamed:@"DribbleIcon"
                                       url:@"http://dribbble.com/janedev"],

                [self socialInfoWithTitle:@"Facebook"
                                 iconNamed:@"FacebookIcon"
                                       url:@"http://facebook.com/jane.dev.75"],

                [self socialInfoWithTitle:@"GitHub"
                                 iconNamed:@"GitHubIcon"
                                       url:@"http://github.com/janedev"],

                [self socialInfoWithTitle:@"Google+"
                                 iconNamed:@"GooglePlusIcon"
                                       url:@"http://plus.google.com/107153662642475014992"],

                [self socialInfoWithTitle:@"LinkedIn"
                                 iconNamed:@"LinkedInIcon"
                                       url:@"http://linkedin.com/pub/jane-developer/8b/343/b7a/"],

                [self socialInfoWithTitle:@"Twitter"
                                 iconNamed:@"TwitterIcon"
                                       url:@"http://twitter.com/janedevv"],
                ];
    }

    return all;
}
```

Note how the models are referencing an icon name, which we use to build a UIImage from. We don't have any images named DribbleIcon, TwitterIcon, etc, though! We'd better add them.

Open up **Images.xcassets** and the class resources folder. Drag the **Social Icons** folder into the asset catalog, just like before. This'll give you a few icons to work with.

Extra Credit

Those of you with a coding background might be cringing about hard coding our model list directly into the source. I don't blame you!

While it's certainly convenient for small sets of data like this, it breaks down. Have no fear, however — Apple provides a very nice persistence layer called Core Data (effectively a database).

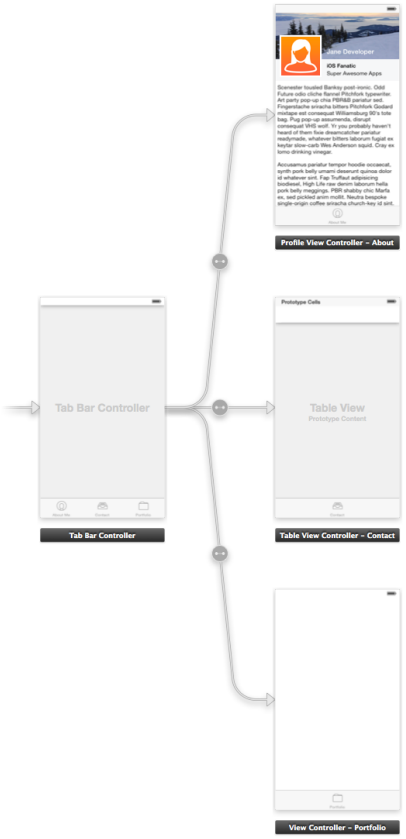
Spend some time reading through the **Core Data Programming Guide** (search the documentation for that).

Now that we have all our social info in place, it's time to display it to the user! Head back to the storyboard.

We want the contact view to be a table view. If it is not already, you will need to replace the blank contact view controller with a new table view controller.

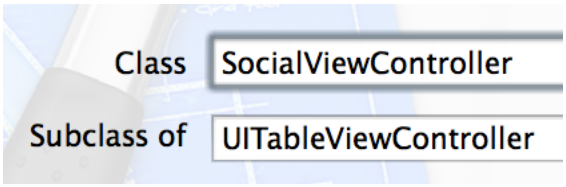
To do so, delete the blank contact view controller, and drag a table view controller out in its place. Then, you will need to right click the tab bar controller, and add the new table view controller as one of its view controllers. Don't forget to update the tab's label and icon.

Your storyboard should match the screenshot to the right.



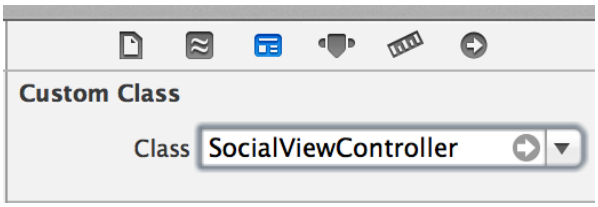
Wiring A Table View Controller

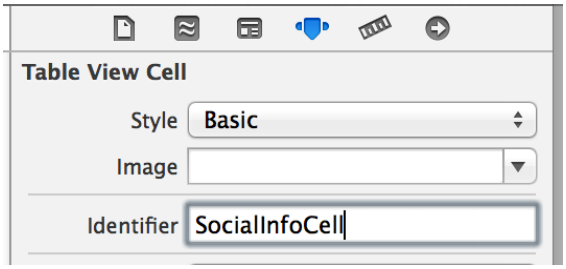
Table view controllers also act as data sources, but customizing them requires that we drop down to code.



Right click on the **View Controllers** group (creating it as necessary) in the file view, and **Add File**. As before, choose **Objective-C class**. In this case, we want to create a SocialViewController that inherits from UITableViewController. Create it.

Before we dive into the code, head back to the **storyboard**. Select the **table view controller** that we added, and switch to its identity inspector. Switch its class from UITableViewController to our custom **SocialViewController** class.





The last thing to do is to complete our table cell prototype view. Select on the table cell prototype and view its attributes. Set the **reuse identifier** to **SocialInfoCell**, and change its type to **Basic** (which gives us support for an icon and label).

To The Code!

Switch back to the **SocialViewController.m** implementation file. We need to fill in a few methods before our table view will behave as a proper data source.

Because we know we will be reading in data from our SocialInfo model, let's *import* it. Add

```
#import "SocialViewController.h"
#import "SocialInfo.h"
```

an import statement for its header to the top of the file:

As we don't want to partition our social links into sections, let's just hard code the section count to 1:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    // Return the number of sections.
    return 1;
}
```

Delete the warning while you're there too; it's just a reminder to fill in that method.

We also need to tell the table view controller how many rows it should render. We know that by looking at the size of the allSocialInfo array that we defined on our model:

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
    // Return the number of rows in the section.
    return [SocialInfo allSocialInfo].count;
}
```

The last method that we need to implement is tableView:cellForRowAtIndexPath: which is where we take a prototype cell and give it the data we wish to display:

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"SocialInfoCell";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier forIndexPath:indexPath];

    SocialInfo *info = [SocialInfo allSocialInfo][indexPath.row];
    cell.textLabel.text = info.title;
    cell.imageView.image = info.icon;

    return cell;
}
```

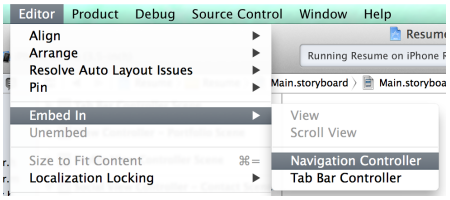
Note that we updated the CellIdentifier, too! Run it in the simulator, and you should see a list of social networks and their icons!

We'll fix the layout next :)

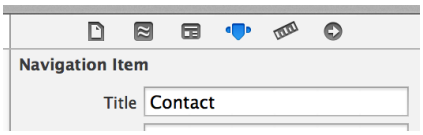
What we'd like to see happen when tapping a cell in the contact table is that a web view containing that social network should be pushed onto the screen.

This pushing behavior is managed by a *navigation controller*. Let's add one.

Head to the storyboard, and select the **social view controller**. Navigation controllers act as containers for all the view controllers they manage. So, go to the **Editor** menu, choose **Embed In** and finally **Navigation Controller**.



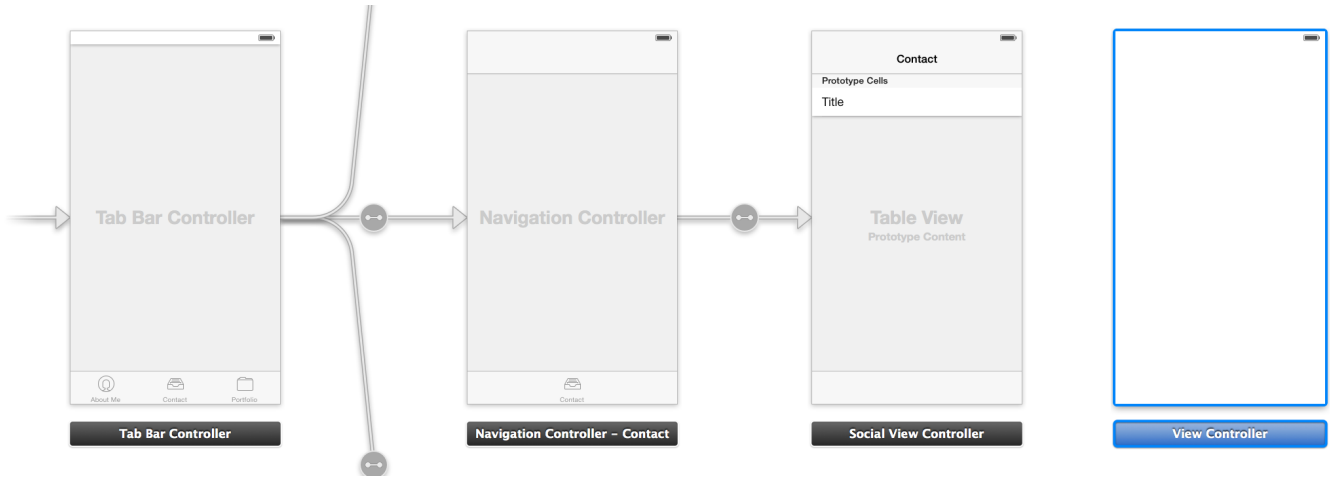
If you run the project and switch to the contact tab, you'll see that there is now a navigation bar above the table view. Nice.



Though, it's missing a title. Select the **navigation item** (the blank bar) in the **social view controller** again, and view its attributes. Set its title to **Contact**.

Web View Controller Placeholder

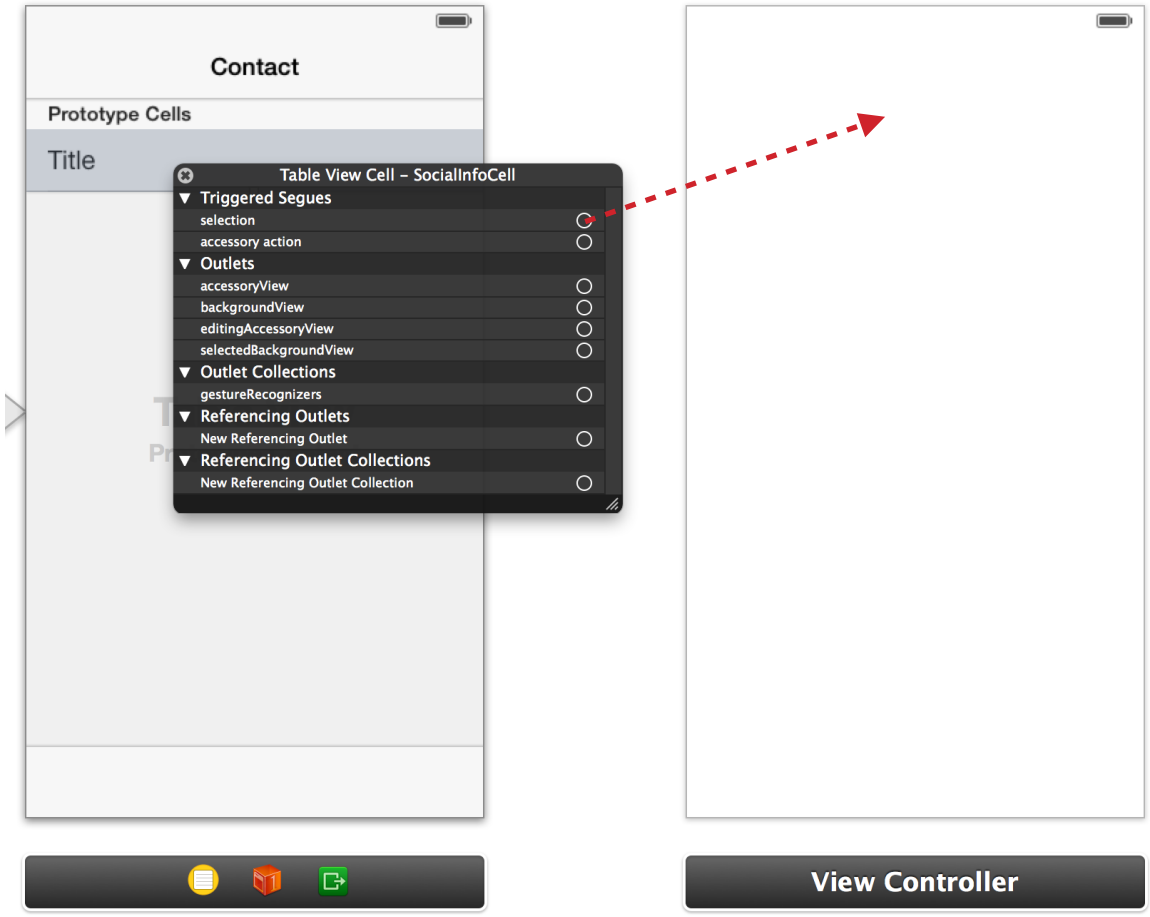
We need to create a new view controller that will display the appropriate web page for the social network we want to handle. Drag a normal **view controller** onto the storyboard. We'll wire it up as a web view later.



Makin' Things Happen

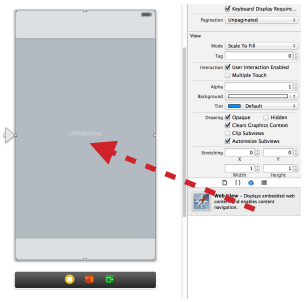
The arrows between view controllers on the storyboard are called **segues**. To make the social view controller match our desired behavior, we want a segue from our social view controller to our web view controller.

To add it, select our **placeholder cell** and **right click** on it to get the outlet editor. Drag from the **selection** segue to our placeholder web view controller to wire it up. You will be presented with another list, choose **push** to create a segue that pushes the view onto the navigation controller..



Run the project, and tap on a cell in the social view controller. It pushes on a blank view (our placeholder view controller). Magic!

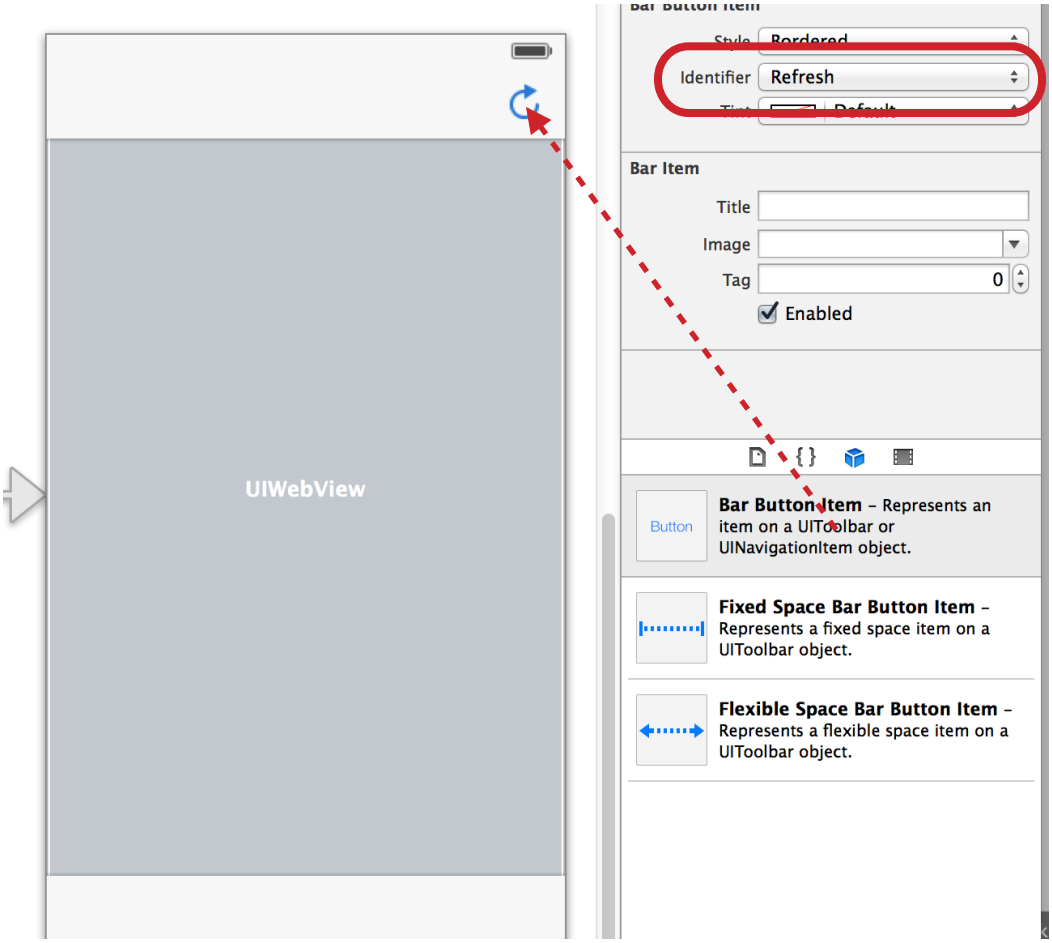
In order to make our web view controller display a web page, we need to wire it up, and dive into the code again.



First, however, go to the **web view controller** in the storyboard. Drag a **Web View** from the object library onto it. Make sure that the web view takes up the entire view. You'll also want to make sure that the web view autoresizes with the view.

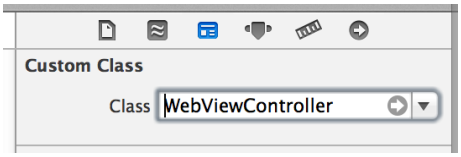
We should also give the user a toolbar button that refreshes the web view for them. Spotty connectivity, errors, and more can get in the way — it's good to give them a possible means to control it.

Drag a **Bar Button Item** onto the web view controller's toolbar. Also, change the **Identifier** for that item to **Refresh**. This will give us a standard refresh button.



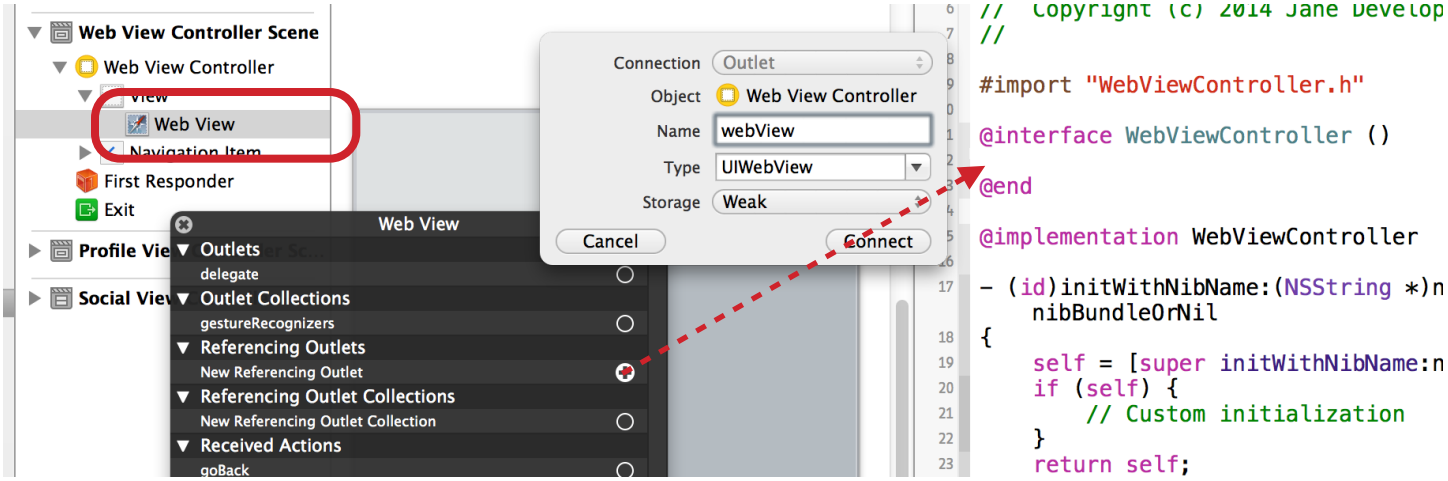
Now it's time to wire up that view with some custom code. Let's create a custom class for our web view controller.

Right click the View Controllers group in the file pane, and select **New File**. **Objective-C class**. Create a class called **WebViewController** that inherits from **UIViewController**.

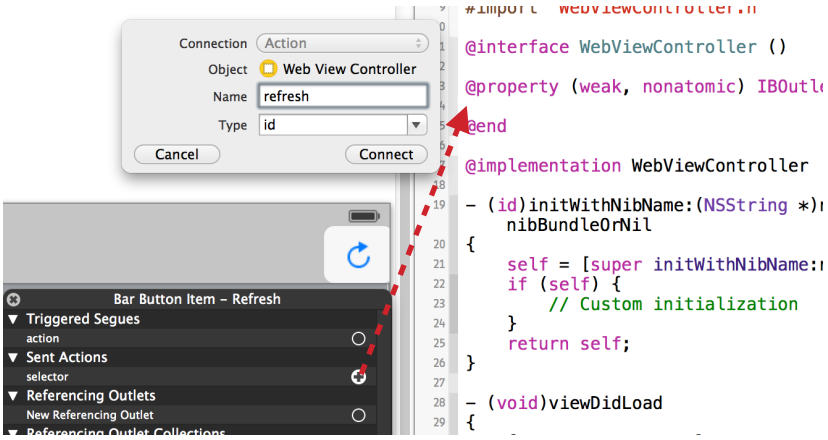


Let's wire it up and create a referencing outlet to the web view so that we can control it. Head back to the **storyboard** and select the web view controller. Set its class to match our newly created one.

Now, do the split pane thing so that the storyboard is in one pane, while **WebViewController.m** is open in the other. Drag a **referencing outlet** from the web view into the controller, and call the new property **webView**.



Additionally, we also need to connect the refresh button. Right click on the refresh button and drag an outlet from the **sent actions section (selector)** to the interface. Call the action **refresh**.



Loading A Page

Our web view controller lacks an important piece of data: the URL it should display. In **WebViewController.h**, let's add a URL property:

```
@interface WebViewController : UIViewController

@property (strong, nonatomic) NSString *url;

@end
```

Let's implement the refresh method. UIWebViews do have a reload method, but that's not the behavior we want. Rather than refreshing the current page, we want the web view to refresh back to its original page. This prevents the user from getting stuck if they clicked on a link inside the web view. Here's the code:

```
- (IBAction)refresh:(id)sender
{
    [self.webView loadRequest:[NSURLRequest requestWithURL:[NSURL URLWithString:self.url]]];
}
```

And now, we just need to load that URL when the web view is being displayed. Since refresh loads it, let's just call refresh, rather than repeating ourselves.

Also, so we can test our web view, let's hard code its URL:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view.
    self.url = @"https://google.com";
    [self refresh:nil];
}
```

Run it in the simulator. Tap on one of the social contact cells, and you should be greeted by Google's mobile website.

Test refresh, too: Search for something in Google (so that you navigate away), and then hit refresh. It should return you back to the Google homepage.

We're so close! Now that we know that the web view works, let's remove the hard coded URL:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view.
    [self refresh:nil];
}
```

In order to display the correct URL when the user taps a cell in our social controller, we need to hook into the segue as it is performed. View controllers give us an event designed to do precisely that.

Open **SocialViewController.m** and add an import statement for our web view controller, since we know that we'll need to interact with it (to set the URL):

```
#import "SocialViewController.h"
#import "SocialInfo.h"
#import "WebViewController.h"
```

Scroll down to the bottom of **SocialViewController.m**. By default, prepareForSegue:sender: is commented out (disabled). To uncomment it, delete the /* and */ that surround it (and it'll turn from green to regular source colors).

Because this method is called for any segue that occurs from the view controller, we need to be a bit careful and make sure that the segue points to a web view controller. Once we've verified that, we can figure out which cell was tapped, and set the appropriate values on the web view controller. Here's the code:

```
// In a story board-based application, you will often want to do a little preparation before navigation
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    // Segue to show a social network's page in a webview.
    if ([segue.destinationViewController isKindOfClass:[WebViewController class]] &&
        [sender isKindOfClass:[UITableViewCell class]]) {
        // Note that we are free to cast the sender and destination now that we know they are of the
        // appropriate classes.
        UITableViewCell *sourceCell = (UITableViewCell*)sender;
        WebViewController *destination = (WebViewController*)segue.destinationViewController;

        NSInteger row = [self.tableView indexPathForCell:sourceCell].row;
        SocialInfo *info = [SocialInfo allSocialInfo][row];
        destination.title = info.title;
        destination.url = info.url;
    }
}
```

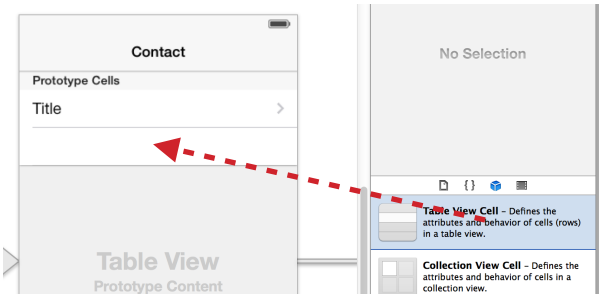
Run it in the simulator, and tap a row. The appropriate page should load for each social network!

The social networks are nice and all, but it'd also be really nice if the user could send you a quick email. Ideally, even if they don't have an email app set up on their phone. This is where Sendgrid comes in. Well, almost.

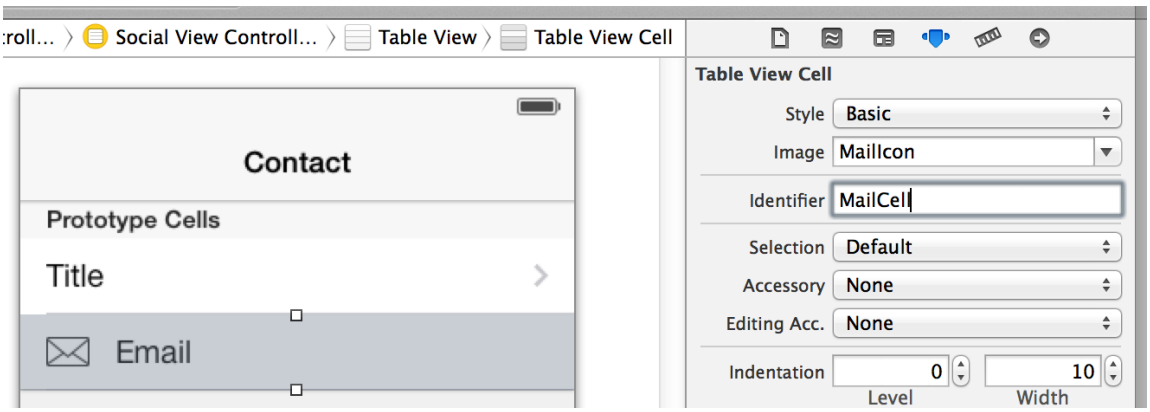
We're going to want an email icon. So drag **MailIcon@2x.png** into the image asset catalog to register it.

Let's add an extra prototype cell to the social view, that is specific to email.

Head to the storyboard, and focus on the contact **social view controller**. Drag a **Table View Cell** into the list of its prototype cells.



Select that cell, and let's tweak its attributes. Change the style to **Basic**, choose **MailIcon** for its image, and give it a **reuse identifier** of **MailCell**. Finally, double click the cell's label and relabel it to **"Email"**.



Displaying The Cell

We need to modify our social view controller's logic to display this email cell. Open **SocialViewController.m**.

Here's where sections come in handy! We can treat our social icons as one section, and the email button can be all on its own in another. So:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    // Return the number of sections.
    return 2;
}
```

We now need to update our counting logic:

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
    if (section == 0)
    {
        // Email
        return 1;
    } else {
        // Social links
        return [SocialInfo allSocialInfo].count;
    }
}
```

As well as our cell setup logic. Let's break it out into multiple functions for clarity:

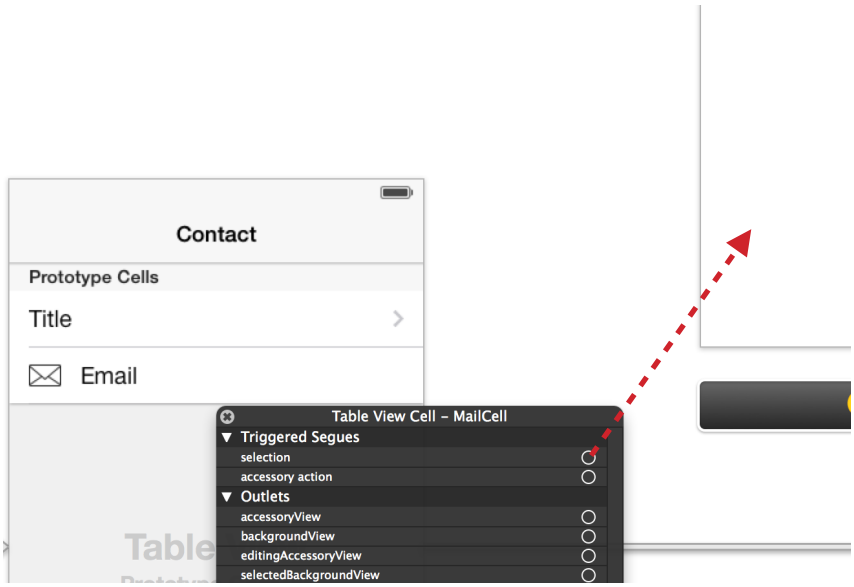
```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    if (indexPath.section == 0) {
        return [self tableView:tableView emailCellForRowAtIndexPath:indexPath];
    } else {
        return [self tableView:tableView socialNetworkCellForRowAtIndexPath:indexPath];
    }
}

- (UITableViewCell *)tableView:(UITableView*)tableView emailCellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"MailCell";
    return [tableView dequeueReusableCellWithIdentifier:CellIdentifier forIndexPath:indexPath];
}

- (UITableViewCell *)tableView:(UITableView *)tableView socialNetworkCellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"SocialInfoCell";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier forIndexPath:indexPath];

    SocialInfo *info = [SocialInfo allSocialInfo][indexPath.row];
    cell.textLabel.text = info.title;
    cell.imageView.image = info.icon;

    return cell;
}
```

Run it in the simulator, and there should now be an email cell above the rest.

Wiring Up The View

Alrighty. Head back to the storyboard.

Create a new **view controller** which will be our email sending view. Then, focus in on the **social view controller** and right click on the **email cell**. Create a segue for the **selection** event, connecting it to the new view controller. Choose **push** from the drop down that pops up at the end.

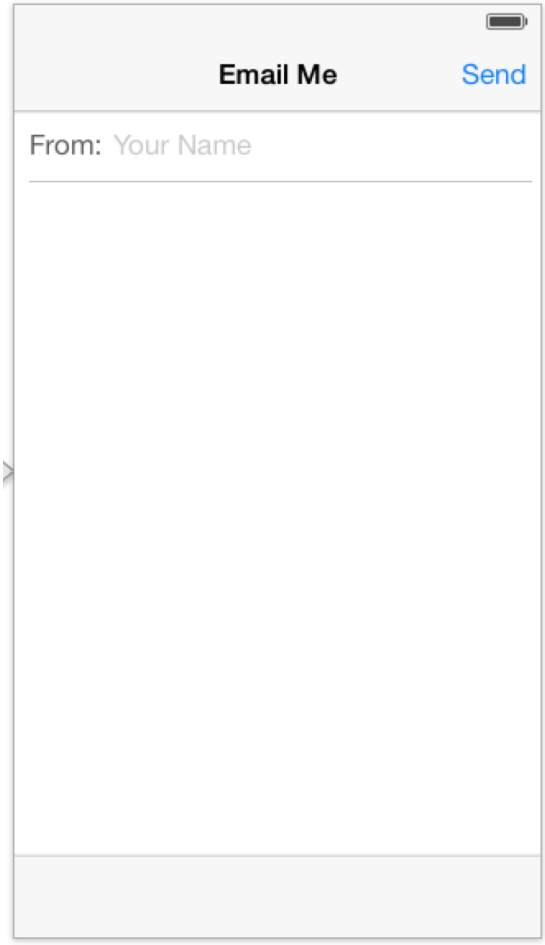


Filling In The View

Let's flesh out the email sending view. We just need two fields: the sender, and the body. You'll want to lay out a few views:

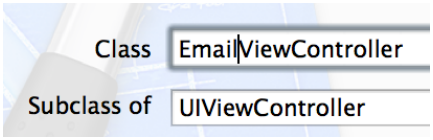
- A label ("From:")
- An input field for the from. "Your Name" as placeholder text.
- A text view for the body of the message
- A "Send" bar button item

When you have it all laid out, run it in the simulator. Your email view should show up when tapping on the email cell in the contact table.



Time to get back into the code!

Create a new **Objective-C class** for the email view controller. Let's call it **EmailViewController**, and it should subclass **UIViewController**.



Now, head back to the **storyboard**, and split pane it with **EmailViewController.m** so we can connect it.

- Set the email view controller's class to **EmailViewController**.
- Create an action called **send** that is connected to the **toolbar button**'s **triggered actions (selector)** action.
- Create a **referencing outlet** that connects the sender **text field** to a property called **senderTextField**.
- Create a **referencing outlet** that connects the body **text view** to a property called **bodyTextView**.

When you're done, you should have the following defined and connected in the interface:

```
11  @interface EmailViewController ()
12
13  @property (weak, nonatomic) IBOutlet UITextField *senderTextField;
14  @property (weak, nonatomic) IBOutlet UITextView *bodyTextView;
15
16  - (IBAction)send:(id)sender;
17
18  @end
```

Next

One behavior that we should implement is in handling the next button on the keyboard. When the user presses next while focused on the "From:" field, focus should shift to the body text view.

In order to implement this, we need to become a UITextFieldDelegate to listen to its events. Open **EmailViewController.h** and modify the @interface definition to indicate this:

```
@interface EmailViewController : UIViewController <UITextFieldDelegate>

@end
```

Then, switch over to **EmailViewController.m**. We need to register ourselves as the text field's delegate. Typically, you want to wire delegates up in **viewDidLoad**:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view.
    self.senderTextField.delegate = self;
}
```

And we need to implement the UITextFieldDelegate method

```
- (BOOL)textFieldShouldReturn:(UITextField *)textField
{
    [self.bodyTextView becomeFirstResponder];
    return NO;
}
```

Note that the first responder is just a fancy name for the view that has focus. Let's test it in the simulator.

Finally, let's be nice to the user and focus the field when our view is appearing:

```
- (void)viewWillLayoutSubviews
{
    [self.senderTextField becomeFirstResponder];
}
```

Resizing For The Keyboard

If you type a long body, notice that the text field is underneath the keyboard :(

Unfortunately, it's a bit painful to do this The Right Way: Whenever the keyboard shows or hides, a global notification is triggered. The app listens for those notifications, and then resizes its views as necessary based on where the keyboard is and will be.

So, let's listen for those notifications:

```
- (void)viewWillAppear:(BOOL)animated
{
    [[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(layoutWithKeyboardEvent:)
                                                 name:UIKeyboardWillShowNotification
                                               object:nil];
    [[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(layoutWithKeyboardEvent:)
                                                 name:UIKeyboardWillHideNotification
                                               object:nil];
}
```

With notifications, you have to be especially careful that you stop listening to them at some point (otherwise you have a memory leak!). We register in viewWillAppear, and unregister in viewWillDisappear to deal with this:

```
- (void)viewWillDisappear:(BOOL)animated
{
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}
```

Finally, we need to handle the actual notifications to resize the keyboard:

```
- (void)layoutWithKeyboardEvent:(NSNotification *)notification
{
    CGFloat animationDuration = [notification.userInfo[UIKeyboardAnimationDurationUserInfoKey]
        floatValue];
    CGRect keyboardEndFrame = [notification.userInfo[UIKeyboardFrameEndUserInfoKey] CGRectValue];
    CGFloat newBottom = MIN(keyboardEndFrame.origin.y, self.view.frame.size.height);

    [UIView animateWithDuration:animationDuration animations:^{
        // Animations work like keyframes: Any values set within the animations block will become
        // text destination values. So we set the height of the body text view so that it will be
        // snug with the keyboard while it shows/hides.
        CGRect bodyFrame = self.bodyTextView.frame;
        bodyFrame.size.height = newBottom - bodyFrame.origin.y;
        self.bodyTextView.frame = bodyFrame;
    }];
}
```

Whew. Run that in the simulator: you should now be able to scroll long body text.