Mobile App Development Crash Course
Natasha Murashev, @natashatherobot
natasha@natashatherobot.com

GENERAL ASSEMBLY

Welcome To iOS Development!

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.

We're going to start by building a Resume app. Imagine showing that to employers!

The Resume app will consist of four sections:

- **About Me**: Share your story.
- **Social Links**: Make it easy for others to follow you on LinkedIn, Twitter, Facebook, Dribbble, Github, and more.
- **Contact Me**: Make it easy for employers to contact you without exposing your email address.

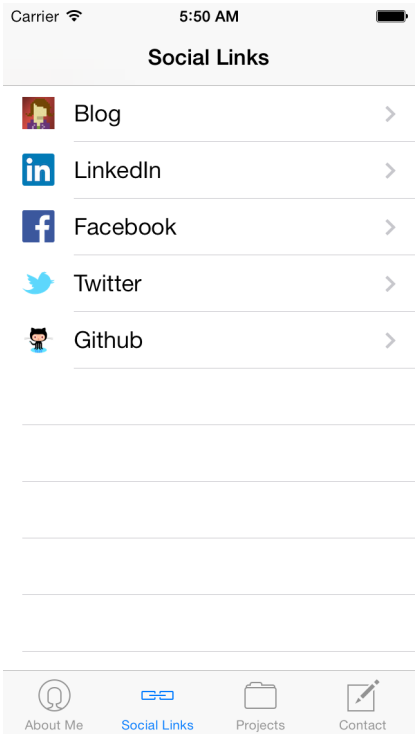
Now that we got that out of the way, let's get started!

GENERAL ASSEMBLY

Social Links

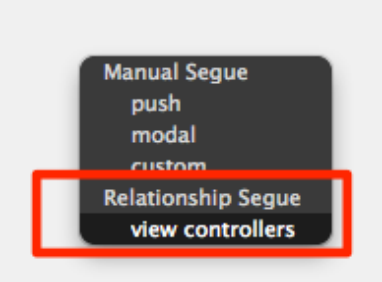
What's the use of a Resume App without easily accessible social links?! So this is what we're building for our second tab:
When a user clicks on one of your social profiles, they'll be taken to a WebView of your social profile. So, let's get started!

Since you're a pro at the StoryBoard by now, I will assume that you know what to do with very minimal instruction. Call me over if you're confused and need help!

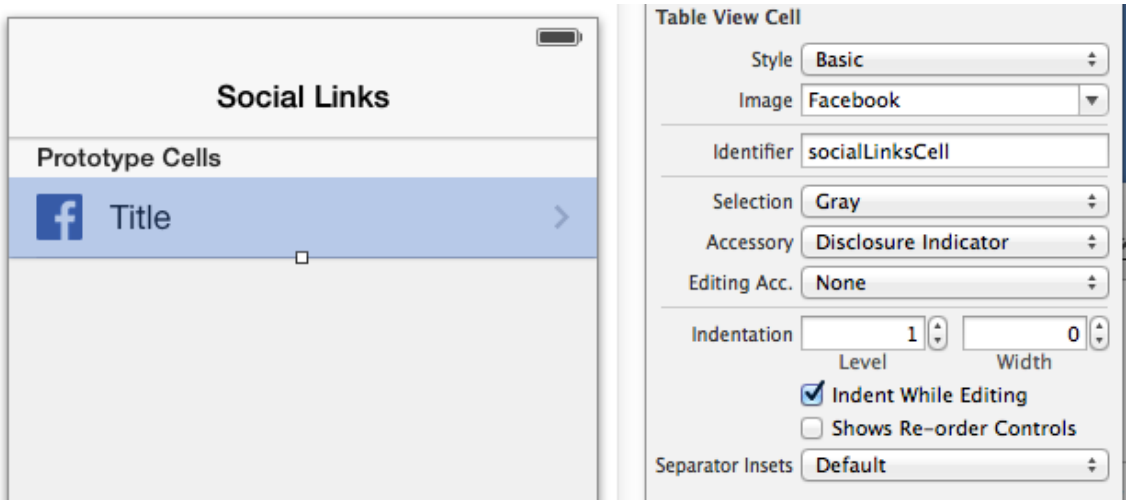


- Open your Storyboard file
- Find the **Navigation Controller** and drag it onto the storyboard under your AboutMeViewController. The Navigation Controller will automatically come with a **TableViewController**, which we'll be using. Feel free to re-arrange the views, so that the AboutMeViewController is above the Tab Bar Controller and the new Navigation Controller is next to it!
- A Navigation Controller is used for Navigating between different views - in our case we'll be Navigating from the main Social Links page to each social link's WebView and back. Look up **UINavigationController** in the Documentation to learn more!
- The **UITableViewController** is used specifically for list views (like the one you see above in my Social Links screenshot).

- **Control-Drag** from the **Tab Bar Controller** to the **Navigation Controller**. You'll get a few animation options for the relationship between the Tab Bar Controller and the Navigation Controller. Select the **View Controllers** option, since this Social Links Navigation Controller is one of the View Controllers on the Tab Bar.

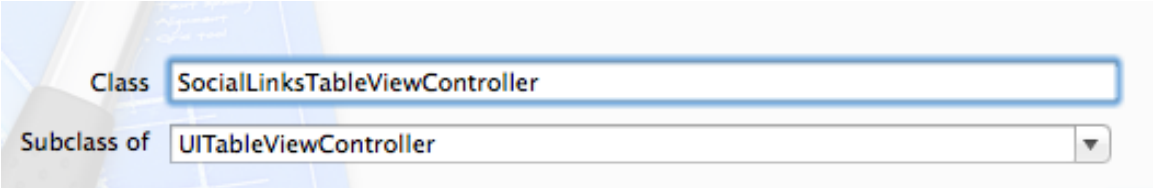


- You'll see the Navigation Controller now has a **Tab Bar Item**. Change the name of the Tab Bar Item to **Social Links**. In the icon's folder, find the socialLinks icons at normal and double size, add them to your Images.xcassets folder, and assign the image to the Tab Bar Item.
- In Images.xcassets, create a Social Links folder, and add the social icons you plan to use in that folder. The normal size of the icons will be 25x25, with 50x50 for retina display.
- Change the **Title** of your **Table View Controller** to **Social Links**.
- Select the **Prototype Cell**, and go to the Tool Belt Selection on the Right Hand Pane.
- Change it to the following settings:
  - **Style**: Basic - in the future, you can customize your cells, but a basic cell is perfect for what we need. It's just a title with an Image.
  - **Image**: Select one of your Social Links images (e.g. Facebook) as a placeholder.
  - **Identifier**: call it something specific like socialLinksCell.
  - **Selection**: This is the color for when the user clicks on the cell. The default is Blue, but I personally like Gray.
  - **Accessory**: Select Disclosure Indicator - that is the arrow at the end of the cell, indicating that when you click on it, it will go to the next screen (the WebView for our social profile).

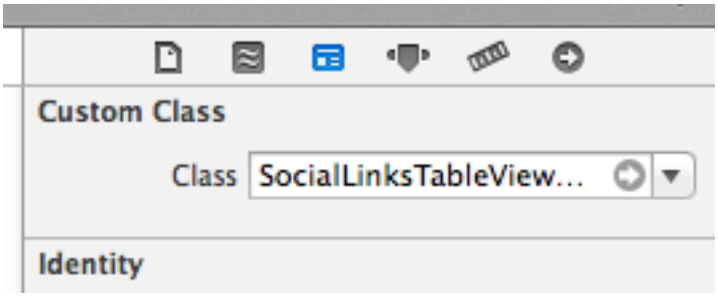


GENERAL ASSEMBLY

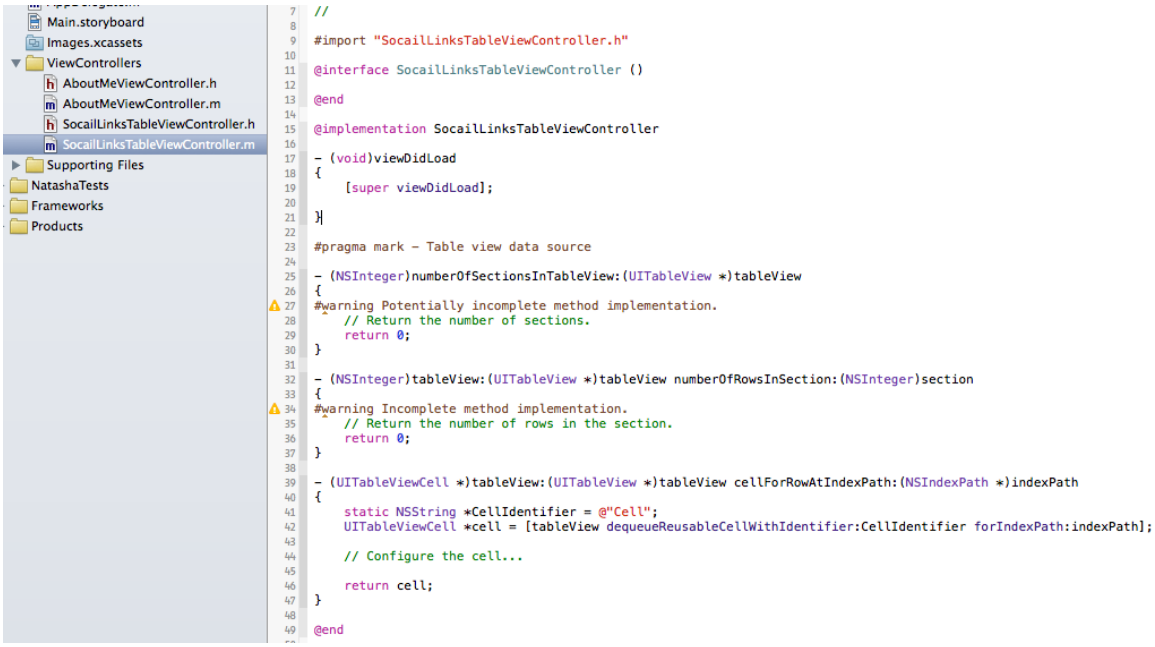
- Create a new ViewController (in your ViewControllers group) called **SocialLinksTableViewController**, which is a **subclass of UITableViewController**.



- Back in **Storyboard**, connect your UITableViewController to the SocialLinksTableViewController:



- In your new **SocialLinksTableViewController.m** file, remove all the comments, the didReceiveMemoryWarning method, and the initWithStyle: method, since we won't need these:



GENERAL ASSEMBLY

- Read the names of the tableView methods which are there. What do you think they do? What is their return value? What is being passed in?
- Look up **UITableViewDataSource** in the Documentation. What other methods can you add to enrich your TableView Data? Which methods are required? Which are optional?
- Look up **UITableViewDelegate** in the Documentation. What other customizations can you make to the TableView?
- In the **@interface** section of your SocialLinksViewController.m file, **create two NSArrays**, one for the **socialServices** you'll be including in your table view, and one for the **socialURLs** those services lead to. If you image names will not match the names of the social services, either change the image names or create a third array for your social service image names. The **socialServices array is your Table View Data Source**!

```
@interface SocialLinksTableViewController ()

@property (strong, nonatomic) NSArray *socialServices;
@property (strong, nonatomic) NSArray *socialURLs;

@end
```

- **Option Double-Click** on **NSArray** to bring up it's Documentation. What are some ways of creating or initializing an array with our Social Services? Here are a few ways:

```
// using the class method +arrayWithObjects:. Make sure to put a nil at the end to indicate a stopping point!
self.socialServices = [NSArray arrayWithObjects:@"Blog", @"LinkedIn", @"Facebook", @"Twitter", @"Github", nil];

// you can first allocate memory for the array, and then instantiate it using the -initWithObjects: instance method.
self.socialServices = [[NSArray alloc] initWithObjects:@"Blog", @"LinkedIn", @"Facebook", @"Twitter", @"Github", nil];

// not in the documentation, but this is literal syntax for creating an Array. No nil at the end necessary!
self.socialServices = @[@"Blog", @"LinkedIn", @"Facebook", @"Twitter", @"Github"];
```

- In the viewDidLoad method, populate the arrays with the correct information in any way you prefer (I personally love the literal syntax):

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    self.socialServices = @[@"Blog", @"LinkedIn", @"Facebook", @"Twitter", @"Github"];
    self.socialURLs = @[@"http://natashatherobot.com/",
                        @"http://www.linkedin.com/in/natashatherobot",
                        @"https://www.facebook.com/natasha.murashev",
                        @"https://twitter.com/NatashaTheRobot",
                        @"https://github.com/natashatherobot"];
}
```

- Now that we have the data sources settled, we will populate the table view!

GENERAL ASSEMBLY

- We only have one section in our Table View, so the **numberOfSectionsInTableView:** method is easy. **We just have to return 1**!

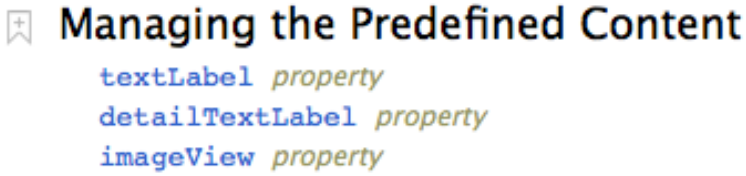
```
#pragma mark - Table view data source

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1;
}
```

- The numberOfRowsInSection should be the same as the number of socialServices we have. Open up the NSArray Documentation and look for a method that will tell us how many items there are in the array! Hint: It's in the Querying an Array Section:

- You got it, it's count! So in the **tableView:numberOfRowsInSection:** method just has to return the count of objects in the socialServices Array!

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
    return [self.socialServices count];
}
```

- Next up, in the **tableView:cellForRowAtIndexPath:**, change the **CellIdentifier** to the one you added in the storyboard (in my case, it was **socialLinksCell**). If you forgot, just go back to the Storyboard, select the Prototype Cell, and check the identifier you gave it in the Tool Belt Section.
- **Option Double-Click on UITableViewCell** to open up it's documentation. Since we're using a **Basic Style cell**, we can use the **predefined content properties**:



GENERAL ASSEMBLY

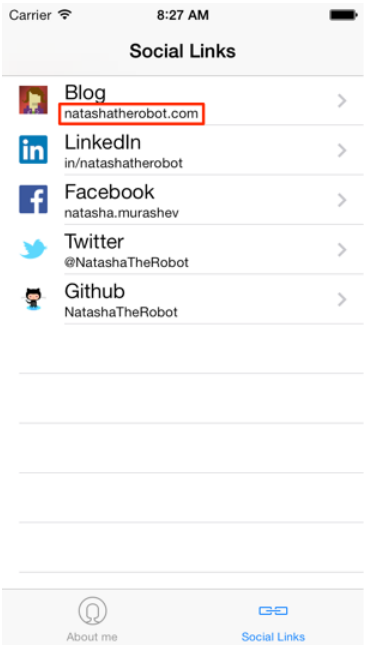
- The **NSIndexPath** that is passed in defines the position of the cell. So an IndexPath with Section 0 and Row 0, means the cell is in the first section, at first position. In our case, since we only have one section, we can tie the row to the position in our Array to find the correct social service:

```
NSString *socialService = [self.socialServices objectAtIndex:indexPath.row];
```

- We can now use the socialService to populate the textLabel and the Image (considering the socialService name is the same as the Image name):

```
43
44  - (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
45  {
46      static NSString *CellIdentifier = @"socialLinksCell";
47      UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier forIndexPath:indexPath];
48
49      NSString *socialService = [self.socialServices objectAtIndex:indexPath.row];
50      cell.textLabel.text = socialService;
51      cell.imageView.image = [UIImage imageNamed:socialService];
52
53      return cell;
54  }
```

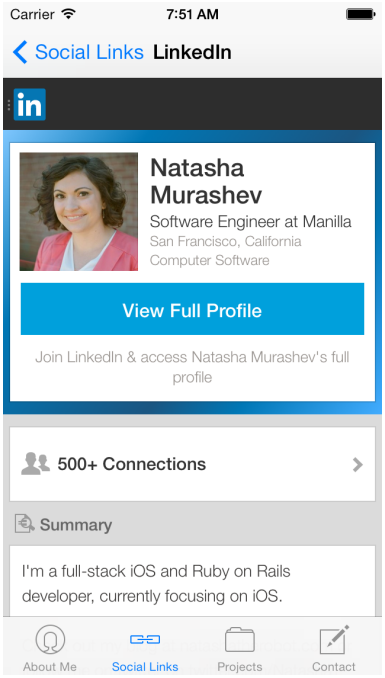
- Run the app to make sure all the socialServices are populated in the TableView!
- Stop the app. Now **put a break point near the end of the method**. Run the app again, select the Social Links tab, and **print out each cell's indexPath and matching socialService**.
- **Challenge**:
  - In the Storyboard, change the Style of the socialLinksCell to Subtitle.
  - In the SocialLinksTableViewController.m, add an NSArray of usernames for your socialServices.
  - User the detailsTextLabel cell property to set the correct username as the subtitle for each service.



GENERAL ASSEMBLY

Segue to WebView

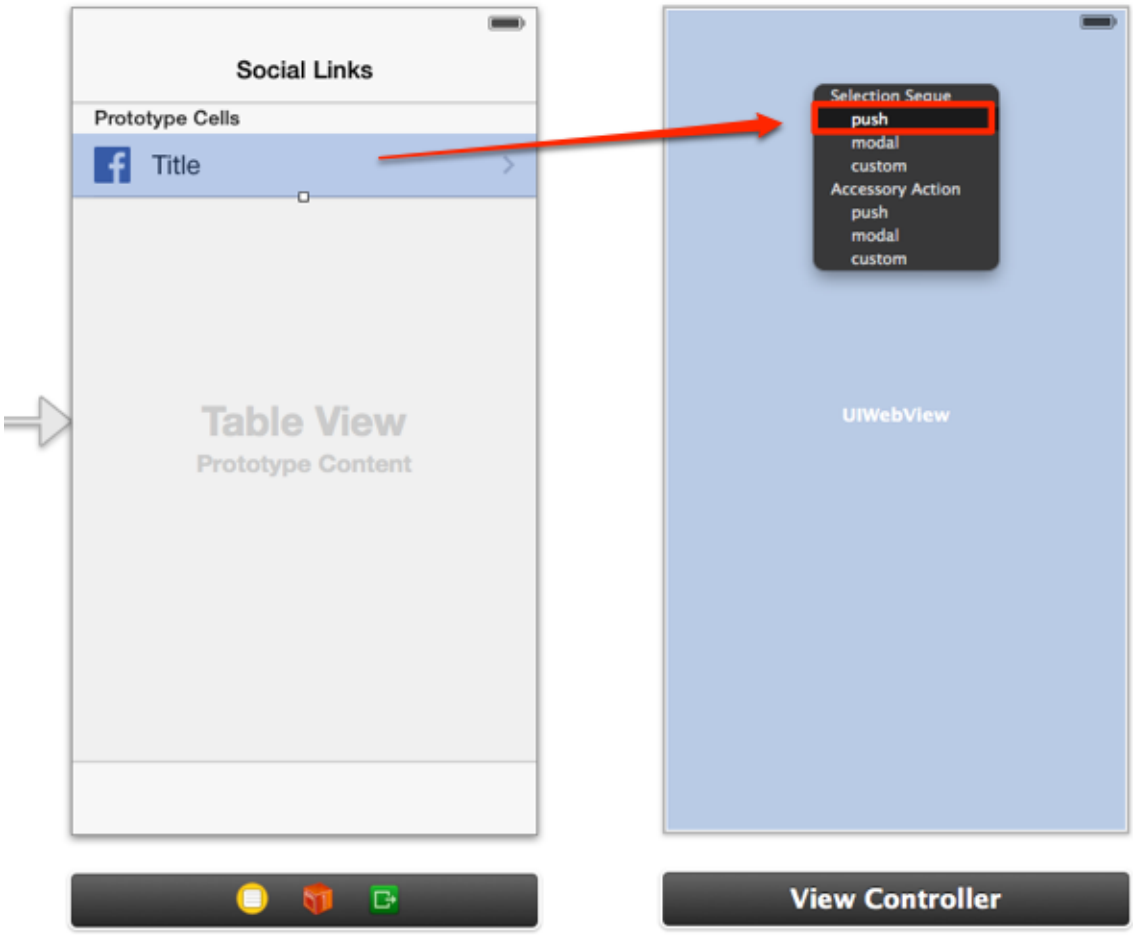
Next up, as the user clicks on the socialLinkCell, we want them to go to a WebView for that socialService. For example, here is the WebView for my LinkedIn profile:



- Go back to the Storyboard.
- Drag a **ViewController to the right of the SocialLinksTableViewController**.
- Drag a **Web View** inside the ViewController. The Web View should take up **the full width and height of the ViewController**.
- Select the **socialLinkCell** on the SocialLinksTableViewController and **Control-Drag from it to the new WebView ViewController**.

GENERAL ASSEMBLY

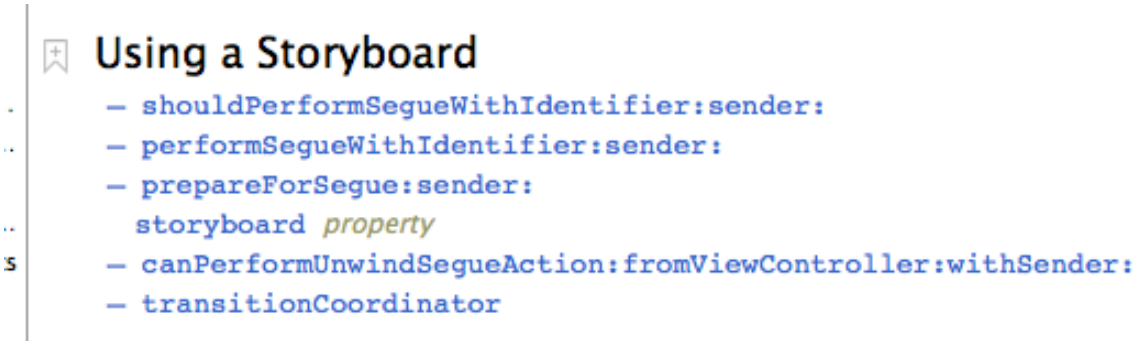
- Select the **push segue option**. Push is a built-in transition to navigate from one controller to the next. In this case, when the user clicks the socialLinkCell, they will transition to the WebView ViewController.



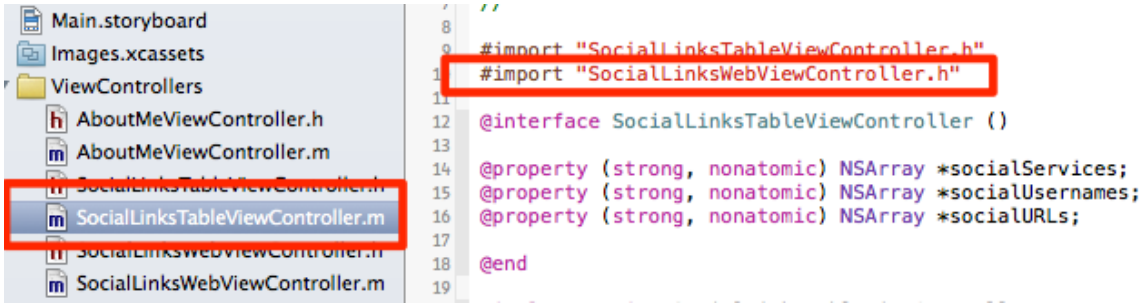
- Run the app. Click on the Social Links Tab. Click on any socialService, and you'll see the push transition!
- Of course, we'll need to first pass the information from the SocialLinksTableViewController to the new WebView ViewController.
- Create another ViewController called **SocialLinkWebViewController**, which is a **subclass of UIViewController**.
- In the Storyboard, identify the WebView ViewController as the SocialLinksWebViewController class.
- Create an **outlet property called webView in the SocialLinksWebViewController** to connect the UIWebView in the storyboard to the ViewController code. We need an outlet so we can tell the webView which URL to load!
- In **SocialLinksWebViewController.h**, create an **NSString property called socialURL**. The reason this is in the .h file is because we need this property to be accessible publicly by the SocialLinksTableViewController!

GENERAL ASSEMBLY

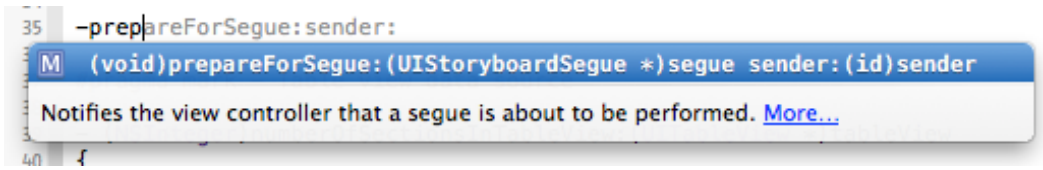
- Go back to the SocialLinksTableViewController. We're going to pass the serviceURL for the cell that was selected to the SocialLinksWebViewController.
- Look up **UIViewController in the documentation**. Remember, the UITableViewController, which SocialLinksTableViewController is a subclass of, is also a subclass of UIViewController, which means our SocialLinksTableViewController can user or override any methods mentioned in the UIViewController. We're going to override the prepareForSegue:sender: method mentioned in the **Using a Storyboard section of the documentation**.



- Since we dragged our segue in the Storyboard from the socialLinksCell to the SocialLinksWebViewController, the performSegueWithIdentifier:sender: method is automatically called when a user selects the socialLinksCell. The prepareForSegue:sender: method will be called right before the segue actually happens. This is where we need to set the socialURL property for the SocialLinksWebViewController.
- Import the **SocialLinksWebViewController.h** file into the **SocialLinksTableViewController.m** file:



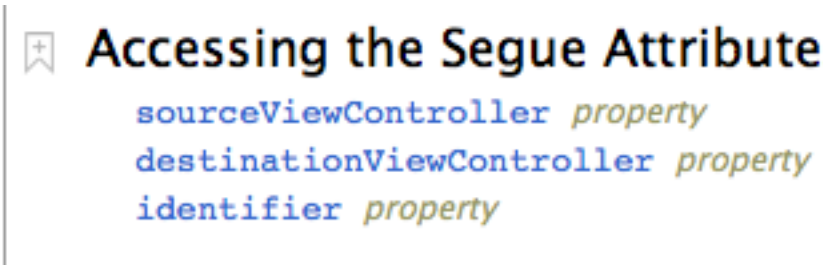
- Start typing "-prepare" and the prepareForSegue:sender method should be auto-suggested, since it already exists in the UIViewController, which SocialLinksTableView is a subclass of. Select the suggested option:



- Add the curly brackets underneath to make it a function:

```
-(void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{

}
```

- **Option Double-Click on UIStoryboardSegue** to bring up the documentation. We're going to use the **destinationViewController property** to access the SocialLinksWebViewController (since that is the destination view controller when a cell is selected):



- We know that the destinationViewController is the SocialLinksWebViewController, since we dragged the segue in the Storyboard from the socialLinksCell to the SocialLinksWebViewController, so we can declare this:

```
-(void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    SocialLinksWebViewController *socialLinksWebViewController = segue.destinationViewController;
}
```

- Now we need to get the NSIndexPath of the selected cell, so we can get the correct socialURL from our socialURLs array. The UITableView has a method for doing this:

```
-(void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    SocialLinksWebViewController *socialLinksWebViewController = segue.destinationViewController;
    NSIndexPath *indexPathForSelectedCell = [self.tableView indexPathForSelectedRow];
}
```

- We can now use the indexPathForSelectedCell to get the socialURL for the selected cell:

```
-(void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    SocialLinksWebViewController *socialLinksWebViewController = segue.destinationViewController;
    NSIndexPath *indexPathForSelectedCell = [self.tableView indexPathForSelectedRow];
    NSString *socialURL = [self.socialURLs objectAtIndex:indexPathForSelectedCell.row];
}
```

- Put a break point right after socialURL, and run the program. Make sure that when you select a socialLinkCell, the socialURL that we just generated is the correct one for the cell!
- Now, we can assign the socialURL from the selected cell to the destination SocialLinksWebViewController socialURL property:

```
-(void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    SocialLinksWebViewController *socialLinksWebViewController = segue.destinationViewController;
    NSIndexPath *indexPathForSelectedCell = [self.tableView indexPathForSelectedRow];
    NSString *socialURL = [self.socialURLs objectAtIndex:indexPathForSelectedCell.row];
    socialLinksWebViewController.socialURL = socialURL;
}
```

- That's it! This should pass the socialURL from the SocialLinksTableViewController to the SocialLinksWebViewController!
- Put a breakpoint in the viewDidLoad method in the SocialLinksWebViewController.m. Run the app, go to the Social Links tab, tap on one of the socialLinkCells, and make sure the socialURL in the SocialLinksWebViewController is set in viewDidLoad!
- As you can see, the SocialLinksWebViewController does not have a title. We want the title to be dynamic based on the service. So when the user taps the LinkedIn cell on the SocialLinksTableViewController, the title will be LinkedIn.
- To do this, create another NSString **public property** in the SocialLinksWebViewController called **serviceName**, and pass the serviceName from the SocialLinksTableViewController to the SocialLinksWebViewController in the prepareForSegue:sender: method!
- The UIViewController has a title property, which is how you set the title for the top navigation bar. Set the **title property to the serviceName in viewDidLoad**.

```
#import "SocialLinksWebViewController.h"

@interface SocialLinksWebViewController ()

@property (weak, nonatomic) IBOutlet UIWebView *webView;

@end

@implementation SocialLinksWebViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    self.title = self.serviceName;
}

@end
```

- Run the app to make sure the title is set correctly!
- Now we just need to get the WebView to load our socialURL.
- Look up the documentation for UIWebView. To Load our Content, we're going to use the **loadRequest:** method.

- The NSURLRequest consists of an NSURL object. So we're first going to create an NSURL using the socialURL string that we have:

```
NSURL *url = [NSURL URLWithString:self.socialURL];
```

- Next, we're going to use this NSURL to create the NSURLRequest:

```
NSURLRequest *request = [NSURLRequest requestWithURL:url];
```

- Finally, our webView (that we connected earlier from the storyboard), can load the request:

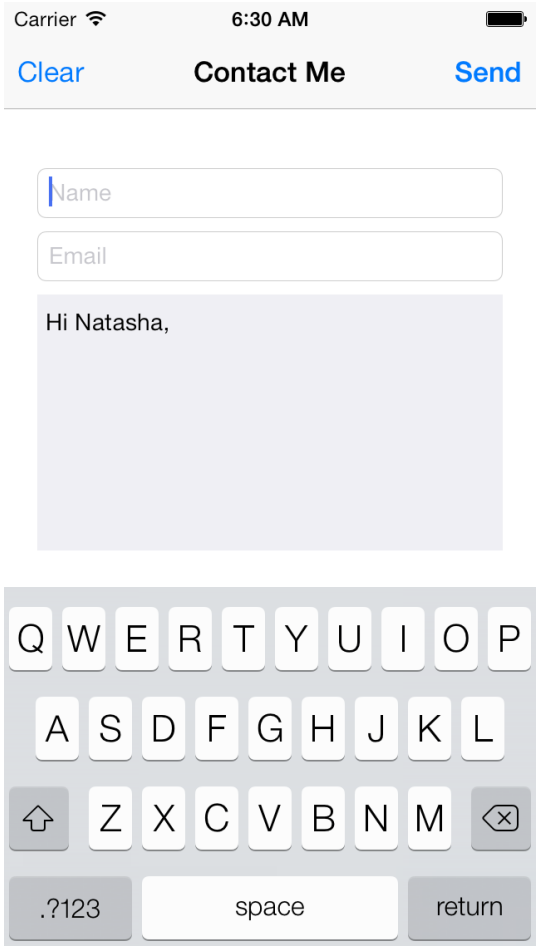
```
[self.webView loadRequest:request];
```

- Run the app. Your links should load!

GENERAL ASSEMBLY

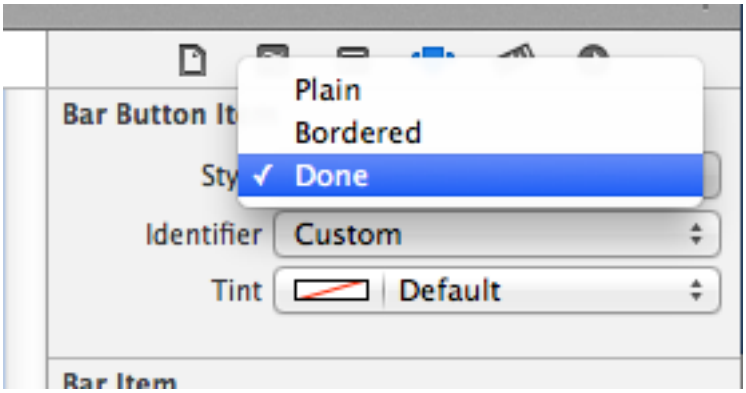
Contact me!

No Resume App is complete without an easy way for the user to contact you! Since it's not that great to reveal your actual email, we're going to use the Sendgrid API to send an email from the contact form you'll create:

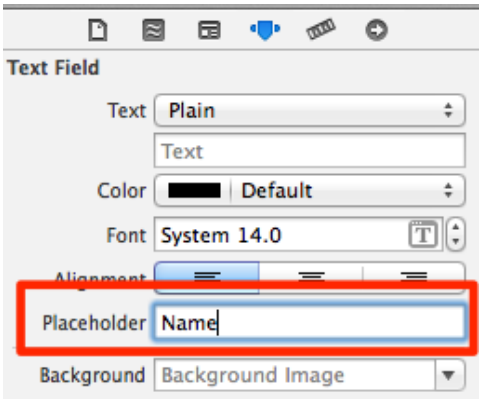


- If you haven't already, sign up for a Sendgrid account - they have a free account with 200 emails per day, which should be plenty. It might take a day or two for your profile to be provisioned, so you might not actually be able to send email right now (but we'll pretend it sends).
- Open up the Storyboard.
- Add a NavigationController on, and make it the third tab on your TabViewController.
- Configure the new tab item to be named "**Contact Me**" with a **composeIcon** as the image.
- Delete the UITableViewController that camp as part of the Navigation Controller.
- Drag on a UIViewController, and connect it to the Navigation Controller as the **Root View Controller**.
- Change the **title** of the UIViewController (which I'll be calling ContactMeViewController) to "**Contact Me**".

- Find a **Bar Button Item** and drag it on to the left of the Navigation Bar. Name this button **Clear**.
- Add another Bar Button Item on the right hand side of the navigation button called **Send**. Make sure that the Send Button has the **"Done" Style**. The Done style makes the Button text bold, so the user knows this is the main action for the page!



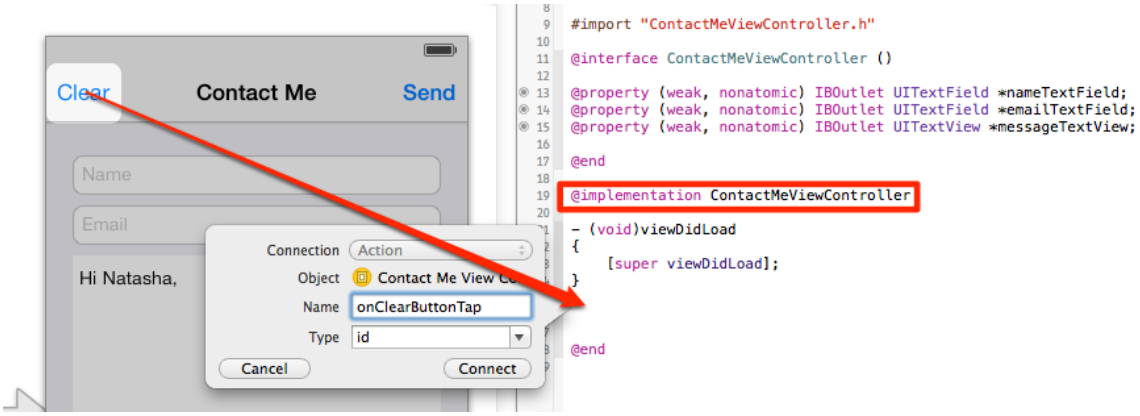
- Next, add two Text Fields on to the ContactMeViewController. One for Name and one for Email.
- Change the **Placeholder attribute** for these fields to Name and Email respectively:



- Find and drag on a **TextView** to the ContactMeViewController.
- Change the TextField's text to a greeting with your name on it (to begin the message someone will send to you). Mine is simply "Hi Natasha,".
- Change the **TextField's background color** to something visible (since white will blend in) - or, alternatively, you can change the background color of the main background view, so the TextField will look good white.
- Make sure all the your text fields and text views are **in the top half of ContactMeViewController**, to account for the space the keyboard will take up.
- Run the app. Go to the new Contact Me tab. Make sure everything looks good. Tap into one of the text fields to make the keyboard come up - make sure everything looks good when the keyboard is up.
- Now, it's time to write some code! Create a new ContactMeViewController class files.

GENERAL ASSEMBLY

- Connect the ContactMe UIViewController in the Storyboard to the ContactMeViewController class you just created in the identity section.
- Create **outlets in your ContactMeViewController.m** file for the **nameTextField**, **emailTextField**, **messageTextView**.
- If you want to change the appearance of a button, then you would create an Outlet. If you want something to happen when the button is pressed, you would create an Action. In our case, we want the Clear button to be associated with an action that clears out all the text on the page, and the Send button to be associated with an action of actually sending the email.
- To create an action, **Control Drag from the button into the @implemenation part of the ContactMeViewController**. Call the action **onClearButtonTap** to make it clear when this action is happening:



- An o**nClearButtonTap:** function should appear:

```
26  - (IBAction)onClearButtonTap:(id)sender
27  {
28
29  }
```

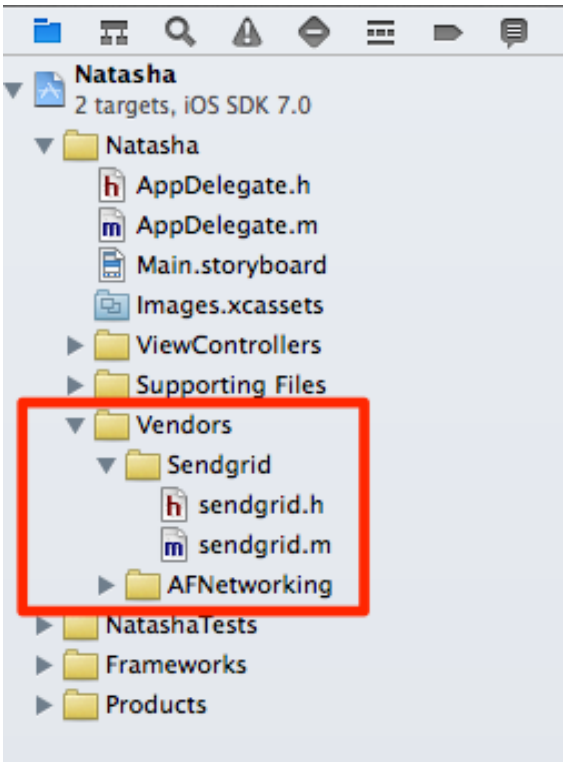
- Create an **onSendButtonTap:** action for the Send button.
- In the onClearButtonTap: function, we can just clear out all the text resetting the text to the defaults:

```
28  - (IBAction)onClearButtonTap:(id)sender
29  {
30      self.nameTextField.text = nil;
31      self.emailTextField.text = nil;
32      self.messageTextView.text = @"Hi Natasha, ";
33  }
```

- Run the app. Enter information into the Contact Me fields, and click the Clear button. The text should clear!

GENERAL ASSEMBLY

- For the onSendButtonTap Action, we're going to use the Sendgrid iOS SDK library, which requires the AFNetworking library to make API requests.
- **Future Challenge:** Read about CocoaPods, which is a way to manage external libraries. Follow the CocoaPod installation instruction for AFNetworking and Sendgrid to install these libraries as Pods.
- For now, we're going to keep it simple. Create a **Vendors group** in your file structure, and add the AFNetworking and Sendgrid libraries from the starter folder into it.



- In your **ContactMeViewController.m** file, **import the sendgrid.h file**.
- Follow the instructions in the Sendgrid documentation to send the message! Use the emailTextField text entered by the user to populate the msg.from field and the messageTextView text to populate the msg.text and msg.html fields.
- Incorporate the name of the sender in the Subject using the NSString stringWithFormat: method:

```
msg.subject = [NSString stringWithFormat:@"Message from %@", self.nameTextField.text];
```

GENERAL ASSEMBLY

- To send the message, we're going to use an undocumented method I added **sendWithWebUsingSuccessBlcok:faiureBlock:** so we can control what alert is displayed when the message succeeds or fails:

```
[msg sendWithWebUsingSuccessBlock:^(id responseObject) {

} failureBlock:^(NSError *error) {

}];
```

- In case of success, we want to clear out all the fields, and show a success message - feel free to modify it!

```
ContactMeViewController *weakSelf = self;
[msg sendWithWebUsingSuccessBlock:^(id responseObject) {
    [weakSelf onClearButtonTap:weakSelf];
    UIAlertView *messageSentAlertView = [[UIAlertView alloc] initWithTitle:@"Yay!"
                                                                   message:@"Glad to hear from you. I'll reply back as soon as I can."
                                                                  delegate:self
                                                         cancelButtonTitle:@"Ok"
                                                         otherButtonTitles:nil];

    [messageSentAlertView show];
} failureBlock:^(NSError *error) {

}];
```

- What should happen when there is an error?! It's up to you to fill that in!
- Run the app, and send a message! Was it successful?!