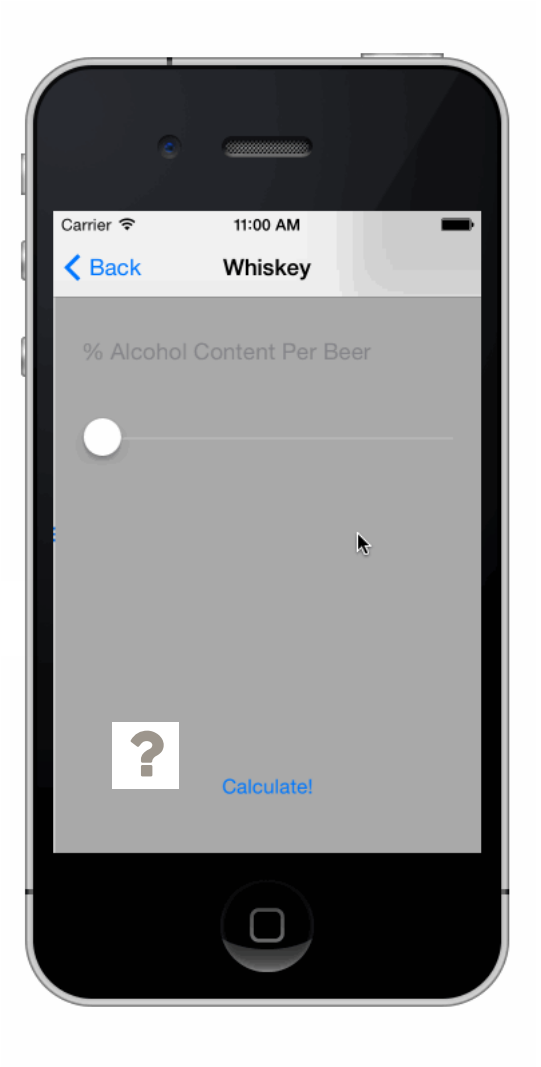
Basic Navigation



> *"A few turnings later and I was thoroughly lost. There is a school of thought which says that you should consult a map on these occasions, but to such people I merely say, 'Ha! What if you have no map to consult? What if you have a map but it's of the Dordogne?' My own strategy is to find a car, or the nearest equivalent, which looks as if it knows where it is going and follow it. I rarely end up where I was intending to go, but often I end up somewhere that I needed to be."*
>
> *- Douglas Adams, The Long Dark Tea-Time of the Soul*

Introduction

Since the first human learned to walk, navigation has been a field of study that has captured the minds of millions. Although perhaps less

Now that you know how to make a view controller, consider that you may make tens or hundreds of view controllers in the course of building a relatively complex app. **`UINavigationController`** is a special subclass of **`UIViewController`** that manages the navigation and organization of collections of view controllers.

A navigation controller's **`view`** has three subviews:

1. A Navigation Bar at the top.
2. A large content area, where your view controllers are displayed.
3. Optionally, a toolbar at the bottom.

Goals for this Checkpoint

You should now have two alcohol calculator view controllers: one for wine, and one for whiskey. We'll allow the user to toggle between the two.

Terms to Know

All the pieces of navigation can get confusing. Here are the most important terms:

Term	Technical Name	Definition
Navigation Controller	**`UINavigationController`**	A **`UIViewController`** subclass that manages navigation between view controllers.
Navigation Bar	**`UINavigationBar`**	A **`UIView`** subclass that shows the title of the current item, and some buttons.
Navigation Stack	**`viewControllers`** property on the **`UINavigationController`** instance	An ordered list of the view controllers (the currently visible view controller is the last one).
Push (verb)	n/a	Add a view controller to the Navigation Stack (often, this is animated).
Pop (verb)	n/a	Remove the top view controller from the Navigation Stack.
Navigation Item	**`UINavigationItem`**	A property on the content view controller that determines the title and buttons in the Navigation Bar.

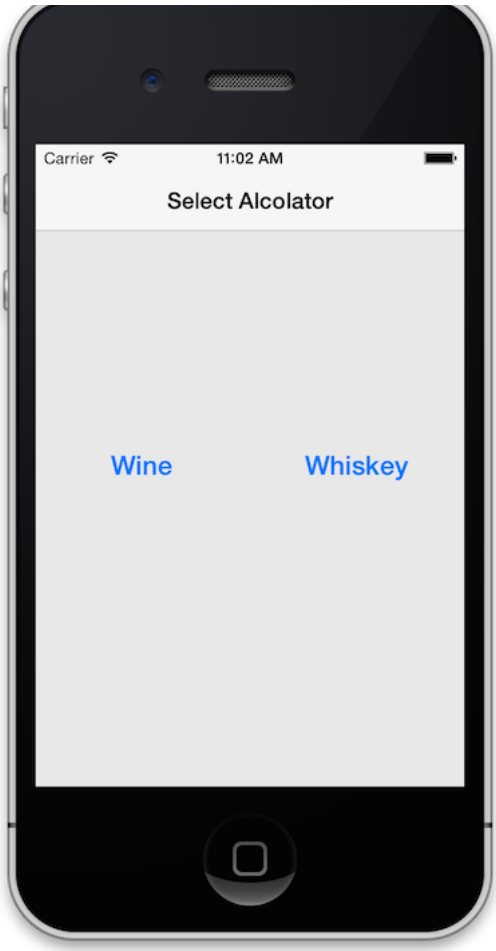
Don't worry if this is confusing; we'll use most of these for our calculator.

After completing the previous checkpoint, you should have two view controllers in your **`Alcolator`** project:

- **`BLCViewController`**, to compare beer and wine
- **`BLCWhiskeyViewController`**, a subclass that compares whiskey instead of wine.

We'll make a Navigation Controller so our users can pick which one they want, but first we'll need to make a menu view controller, for the use to pick from.

Making a “Menu” View Controller



Checkout a new branch:

Terminal

```
$ git checkout -b basic-nav
```

Make a new **`UIViewController`** subclass called **`BLCMainMenuViewController`** (follow the same steps you used to make **`BLCWhiskeyViewController`** in the prior checkpoint.)

At the top, **`#import`** both **`BLCViewController.h`** and **`BLCWhiskeyViewController.h`**:

BLCMainMenuViewController.m

```
#import "BLCMainMenuViewController.h"
+ #import "BLCViewController.h"
+ #import "BLCWhiskeyViewController.h"
```

Add two buttons programmatically, with the text "Wine" and "Whiskey":

Here's one implementation:

BLCMainMenuViewController.m

```
+
+ @property (nonatomic, strong) UIButton *wineButton;
+ @property (nonatomic, strong) UIButton *whiskeyButton;
+
+ @end
+
+ @implementation BLCMainMenuViewController
+
+ - (void) loadView {
+     self.wineButton = [UIButton buttonWithType:UIButtonTypeSystem];
+     self.whiskeyButton = [UIButton buttonWithType:UIButtonTypeSystem];
+
+     [self.wineButton setTitle:NSLocalizedString(@"Wine", @"Wine") forState:UIControlStateNormal];
+     [self.whiskeyButton setTitle:NSLocalizedString(@"Whiskey", @"Whiskey") forState:UIControlStateNormal];
+
+     [self.wineButton addTarget:self action:@selector(winePressed:) forControlEvents:UIControlEventTouchUpInside];
+     [self.whiskeyButton addTarget:self action:@selector(whiskeyPressed:) forControlEvents:UIControlEventTouchUpInside];
+
+     self.view = [[UIView alloc] init];
+
+     [self.view addSubview:self.wineButton];
+     [self.view addSubview:self.whiskeyButton];
+ }
+
+ - (void) viewDidLoad {
+     [super viewDidLoad];
+     self.view.backgroundColor = [UIColor colorWithWhite:0.91f alpha:1];
+
+     UIFont *bigFont = [UIFont boldSystemFontOfSize:20];
+
+     [self.wineButton.titleLabel setFont:bigFont];
+     [self.whiskeyButton.titleLabel setFont:bigFont];
+
+     self.title = NSLocalizedString(@"Select Alcolator", @"Select Alcolator");
+ }
+
+ - (void) viewWillLayoutSubviews {
+     [super viewWillLayoutSubviews];
+
+     CGRect wineButtonFrame, whiskeyButtonFrame;
+     CGRectDivide(self.view.bounds, &wineButtonFrame, &whiskeyButtonFrame, CGRectGetWidth(self.view.bounds) / 2, CGRectMinXEdge);
+
+     self.wineButton.frame = wineButtonFrame;
+     self.whiskeyButton.frame = whiskeyButtonFrame;
+ }
```

Add targets to the buttons, to call methods named **`winePressed:`** and **`whiskeyPressed:`**. Their implementation can look like this:

BLCMainMenuViewController.m

```
+  - (void) winePressed:(UIButton *) sender {
+     BLCViewController *wineVC = [[BLCViewController alloc] init];
+     [self.navigationController pushViewController:wineVC animated:YES];
+  }
+
+  - (void) whiskeyPressed:(UIButton *) sender {
+     BLCWhiskeyViewController *whiskeyVC = [[BLCWhiskeyViewController alloc] init];
+     [self.navigationController pushViewController:whiskeyVC animated:YES];
+  }
```

Let's discuss what's happening in **`[BLCMainMenuViewController -winePressed]`**:

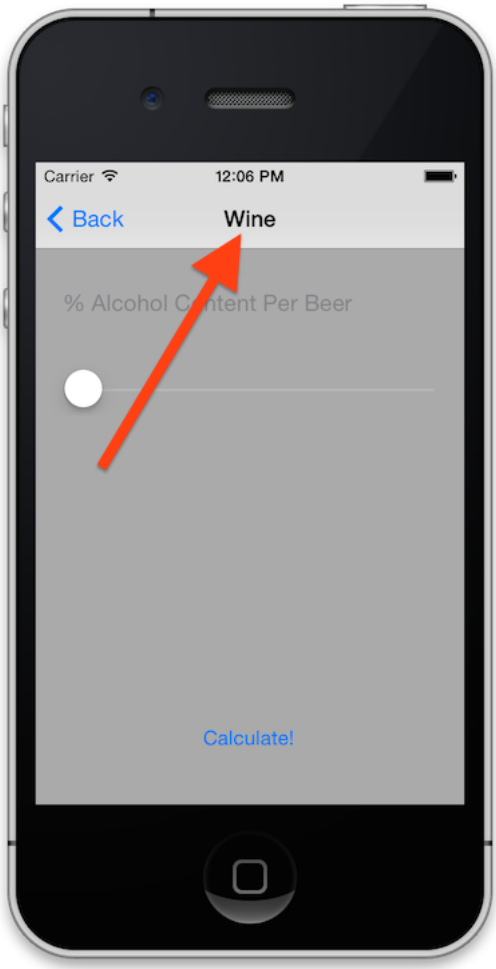
On the first line, in general language, we're creating a new **`BLCViewController`** called **`wineVC`**. In more technical terms, we:

1. Allocate memory for one new **`BLCViewController`**, and create it (**`alloc`**).
2. Run the view controller's initializer (**`init`**).
3. Create a variable called **`wineVC`**, which points to our new controller's memory address.

navigationController is a property of **UIViewController** that returns the nearest Navigation Controller.

Giving Your Controllers a Title

Every view controller has an **NSString** title property. This property is used by the navigation controller to populate the indicated label:



Without setting a title it would simply be blank. Let's set titles for both the **BLCViewController** and **BLCWhiskeyViewController**:

BLCViewController.m

```
- (void)viewDidLoad
{
    // SAME CODE AS BEFORE
+     self.title = NSLocalizedString(@"Wine", @"wine");
}
```

And we'll need to override **viewDidLoad** in **BLCWhiskeyViewController** in order to provide a distinct title for it:

BLCWhiskeyViewController.m

```
+ - (void) viewDidLoad {
+     [super viewDidLoad];
+     self.title = NSLocalizedString(@"Whiskey", @"whiskey");
+ }
```

Making the Navigation Controller

1. Open **`BLCAppDelegate.m`**.
2. At the top, add **`#import "BLCMainMenuViewController.h"`**.
3. Update your **`application:didFinishLaunchingWithOptions:`** method:

BLCAppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]] ;

    // Override point for customization after application launch.

-     BLCViewController *viewController = [[BLCViewController alloc] init];
-     self.window.rootViewController = viewController;
+     BLCMainMenuViewController *mainMenuViewController = [[BLCMainMenuViewController alloc] init];
+     UINavigationController *navigationController = [[UINavigationController alloc] initWithRootViewController:mainMenuViewControlle
+     self.window.rootViewController = navigationController;
    [self.window makeKeyAndVisible];
    return YES;
}
```

You'll notice that for the navigation controller, we're using **`initWithRootViewController:`** instead of **`init`**. Different classes have different initializers, and the best way to check is to review them in the class reference.

Run your app. You should be able to:

- See "Whiskey" and "Wine" buttons when the app launches
- Press either button to go to the appropriate view controller
- Press "Back" in the upper left corner to go back to the menu

Our current app has an issue: when you're in one of the calculator views, you can't tell which view you're in.

- Review the **`navigationItem`** and **`title`** properties on UIViewController, and use them to set the title view text appropriately. (It should say "Wine" or "Whiskey" in the Navigation Bar).
- Update the **`sliderValueDidChange:`** method to show the number of equivalent glasses/shots in the title view (so it says "Wine (123 glasses)", for example.

You can pair with your mentor on this assignment, or try it out yourself.

When you're satisfied with your project and the assignment, commit, merge and push your code to GitHub:

Terminal

```
$ git add .
$ git commit -m 'Toggle navigation between view controllers'
$ git checkout master
$ git merge basic-nav
$ git push
```

Submit this checkpoint's assignment with links for your repo and commit.

assignment completed

COURSES

- Full Stack Web Development
- Frontend Web Development
- UX Design
- Android Development
- iOS Development

ABOUT BLOC

- Our Team | Jobs
- Bloc Veterans Program
- Employer Sponsored
- FAQ
- Blog
- Engineering Blog
- Refer-a-Friend
- Privacy Policy
- Terms of Service

MADE BY BLOC

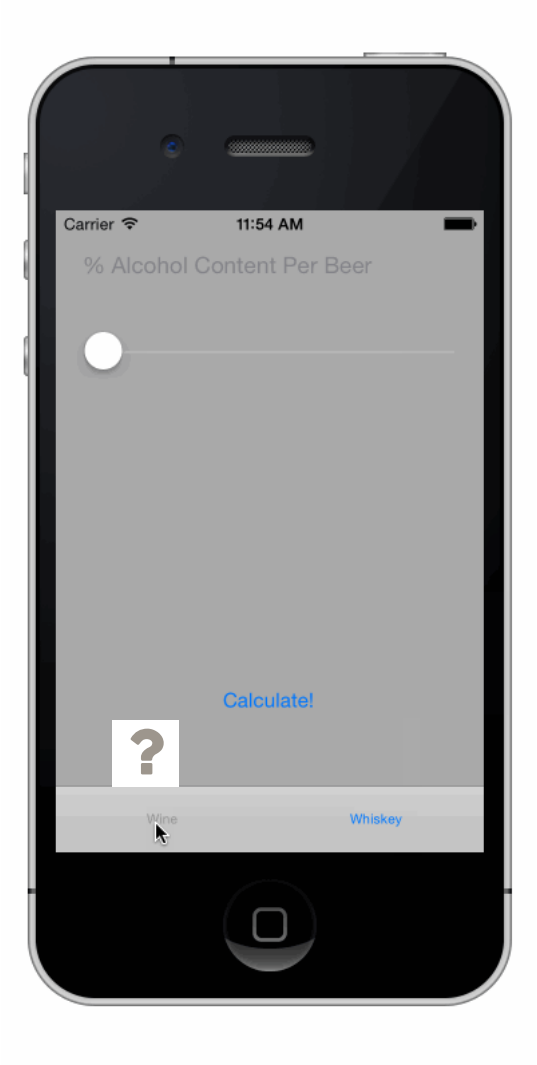
- Tech Talks & Resources
- Programming Bootcamp Comparison
- Jottly: A Beginner's Guide to HTML, CSS, Skeleton and Animate.css
- Swiftris: Build Your First iOS Game with Swift
- Webflow Tutorial: Design Responsive Sites with Webflow
- Ruby Warrior
- Bloc's Diversity Scholarship

SIGN UP FOR OUR MAILING LIST

Send

- hello@bloc.io
- Considering enrolling? (404) 480-2562
- Partnership / Corporate Inquiries? (650) 741-5682

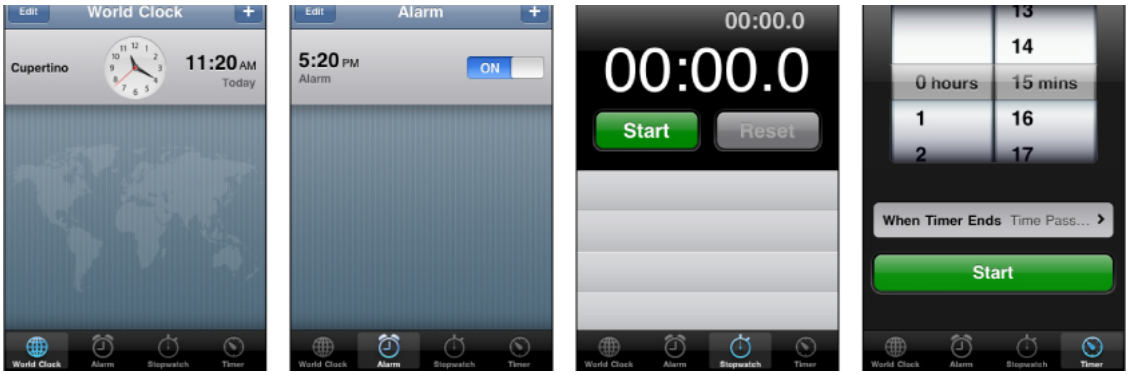
Tabbed Navigation



Introduction

In the previous checkpoint, we learned about **`UINavigationController`**. A view controller that displays other view controllers is commonly referred to as a *container view controller*. Another common container view controller is the **`UITabBarController`**.

The Clock app uses **`UITabBarController`**:



Using **`UITabBarController`**

Like **`UINavigationController`**, **`UITabBarController`** has a property called **`viewControllers`**. This property contains an array of view controllers.

If the array has	The tab bar will have
1 - 5 view controllers	1 tab for each view controller
6+ view controllers	1 tab for the first 4 view controllers, and a "More" tab to access the others (like in the Music app)

Let's get started. Create a new feature branch:

Terminal

```
$ git checkout -b tabbed-nav
```

To configure our app to use a tab bar controller, open **`BLCAppDelegate.m`** and begin by modifying the import section:

BLCAppDelegate.m

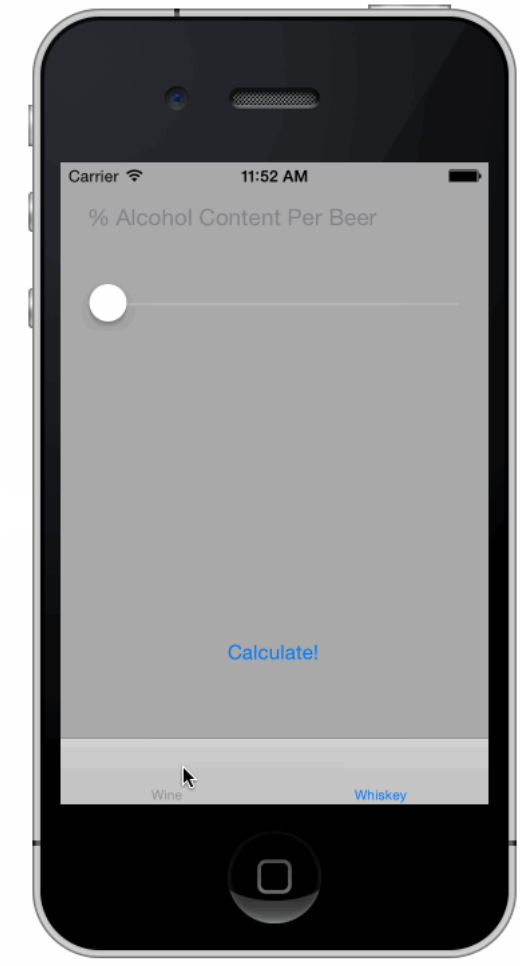
```
#import "BLCAppDelegate.h"
- #import "BLCMainMenuViewController.h"
+ #import "BLCViewController.h"
+ #import "BLCWhiskeyViewController.h"
```

Set **`UITabBarController`** as your root view controller:

BLCAppDelegate.m

```
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]] ;

    // Override point for customization after application launch.

-     BLCMainMenuViewController *mainMenuViewController = [[BLCMainMenuViewController alloc] init];
-     UINavigationController *navigationController = [[UINavigationController alloc] initWithRootViewController:mainMenuViewControlle
-     self.window.rootViewController = navigationController;

+     BLCViewController *wineVC = [[BLCViewController alloc] init];
+     BLCWhiskeyViewController *whiskeyVC = [[BLCWhiskeyViewController alloc] init];
+     UITabBarController *tabBarVC = [[UITabBarController alloc] init];
+     tabBarVC.viewControllers = @[wineVC, whiskeyVC];

+     self.window.rootViewController = tabBarVC;
    [self.window makeKeyAndVisible];
    return YES;
}
```

Now, build & run the app. You should see something similar to the following:



> *For lack of a kinder term, hideous.*

First of all, the "Whiskey" tab is blank until the tab is tapped for the first time. This is because the view hasn't loaded yet, and we're currently setting the title in **`viewDidLoad`**.

BLCViewController.m

```
+ - (instancetype) init {
+     self = [super init];
+
+     if (self) {
+         self.title = NSLocalizedString(@"Wine", @"wine");
+     }
+
+     return self;
+ }
+
```

Remove it from **`viewDidLoad`**:

BLCViewController.m

```
     //gets rid of the maximum number of lines on the label
     self.resultLabel.numberOfLines = 0;

-     self.title = NSLocalizedString(@"Wine", @"wine");
+     self.view.backgroundColor = [UIColor colorWithRed:0.741 green:0.925 blue:0.714 alpha:1]; /*#bdecb6*/
 }
```

Let's take a similar approach in **`BLCWhiskeyViewController.m`**:

BLCWhiskeyViewController.m

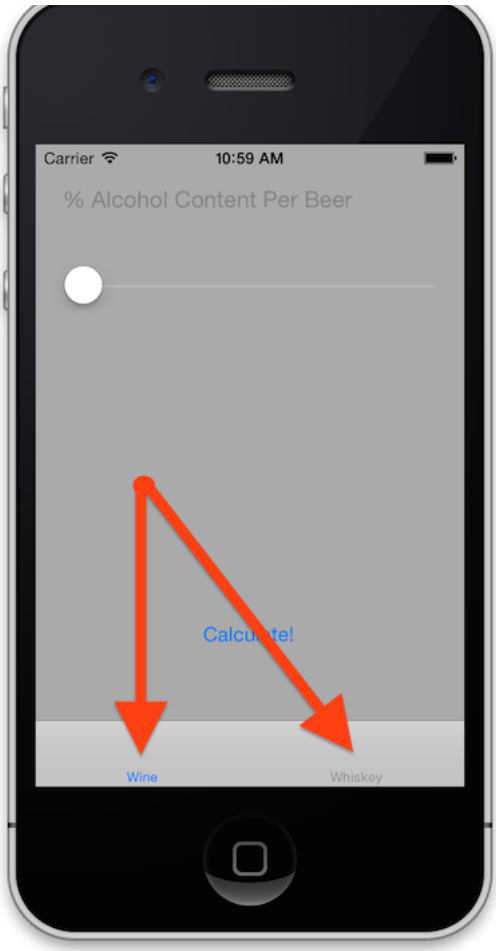
```
 @implementation BLCWhiskeyViewController

+ - (instancetype) init {
+     self = [super init];
+     if (self) {
+         self.title = NSLocalizedString(@"Whiskey", nil);
+     }
+     return self;
+ }
+
 - (void) viewDidLoad {
     [super viewDidLoad];

-     self.title = NSLocalizedString(@"Whiskey", @"whiskey");
+     self.view.backgroundColor = [UIColor colorWithRed:0.992 green:0.992 blue:0.588 alpha:1]; /*#fdfd96*/
 }

 - (void)buttonPressed:(UIButton *)sender {
```

Let's spruce up the tab bar by modifying **`tabBarItem`**.

Using **`tabBarItem`**

Here's another similarity with **`UINavigationController`**: Navigation Controllers populate info about the active view controller by accessing that view controller's **`navigationItem`** property. Similarly, Tab Bar Controllers get title and icon information from the view controller's **`tabBarItem`** property.

`tabBarItem` is an instance of **`UITabBarItem`** which provides details to the **`UITabBarController`**. It tells the tab controller how it to present the tab selector at the bottom of the screen for a given view controller:

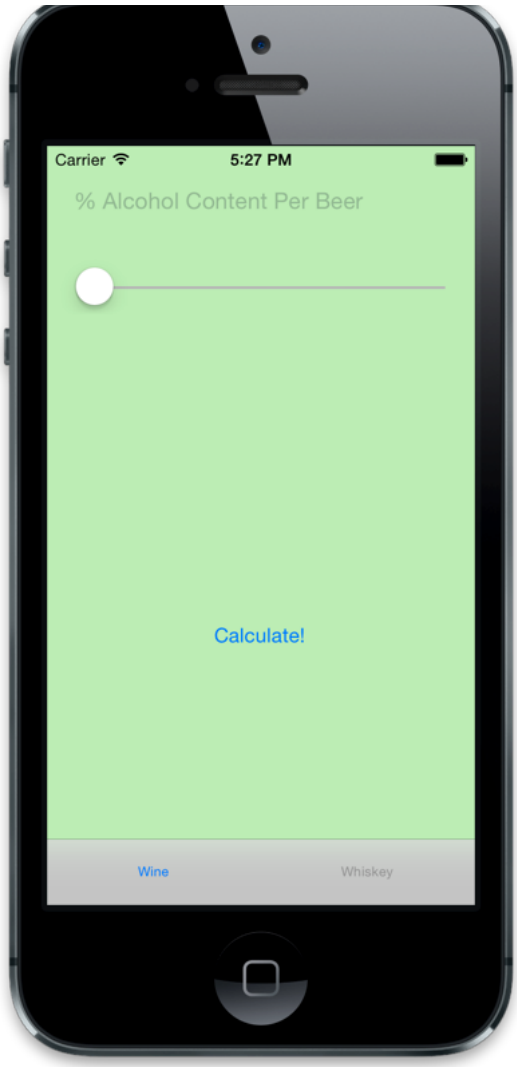


Both of your view controllers, **`BLCViewController`** and **`BLCWhiskeyViewController`** have their own **`tabBarItem`** property. The **`UITabBarItem`** has several properties which we can modify. Let's add one more line to **`init`**:

BLCViewController.m

```
    if (self) {
        self.title = NSLocalizedString(@"Wine", @"wine");
+
+         // Since we don't have icons, let's move the title to the middle of the tab bar
+         [self.tabBarItem setTitlePositionAdjustment:UIOffsetMake(0, -18)];
    }
```

Let's see how that affected our tab bar item:



Looks a bit nicer, no? Another neat thing we can do with **`UITabBarItem`** is assign it a badge value. A badge value is the little red circle found above iOS icons typically indicating notification count, unread emails, etc. This can be done with **`UITabBarItem`**. Despite not being a great design choice, let's update the badge with the number of beers indicated by **`beerCountSlider`**:

BLCViewController.m

```
- (void)sliderValueDidChange:(UISlider *)sender {
    NSLog(@"Slider value changed to %f", sender.value);
    [self.beerPercentTextField resignFirstResponder];
+     [self.tabBarItem setBadgeValue:[NSString stringWithFormat:@"%d", (int) sender.value]];
}
```

Now as you move the slider from **`0`** to **`10`**, you can see the badge value update.

Roadmap · **Previous Checkpoint** **Jump to Submission** · **Next Checkpoir**

Your assignment | Ask a question | Submit your work

- Edit **`BLCAppDelegate`** so it declares that it conforms to the **`UITabBarControllerDelegate`** protocol (review **iOS Design Patterns: Delegation & Protocols** if you don't remember how)
- Assign **`BLCAppDelegate`** as your **`UITabBarController`**'s **`delegate`** property (**`tabBarVC.delegate = self`**)

implementation should print the title of the newly-selected view controller into the console with a message:

> *New view controller selected: [title]*

If you don't know how to print a message to the console, refer to the Basic Objective-C Syntax and Strings checkpoints.

When you're satisfied with your project and the assignment, commit, merge and push your code to GitHub:

Terminal

```
$ git add .
$ git commit -m 'Implement tab bar navigation'
$ git checkout master
$ git merge tabbed-nav
$ git push
```

Submit this checkpoint's assignment with links for your repo and commit.

assignment completed

Ruby Warrior

Bloc's Diversity Scholarship

SIGN UP FOR OUR MAILING LIST

Send

- **hello@bloc.io**
- **Considering enrolling? (404) 480-2562**
- **Partnership / Corporate Inquiries? (650) 741-5682**

Roadmap · Previous Checkpoint
Jump to Submission · Next Checkpoir

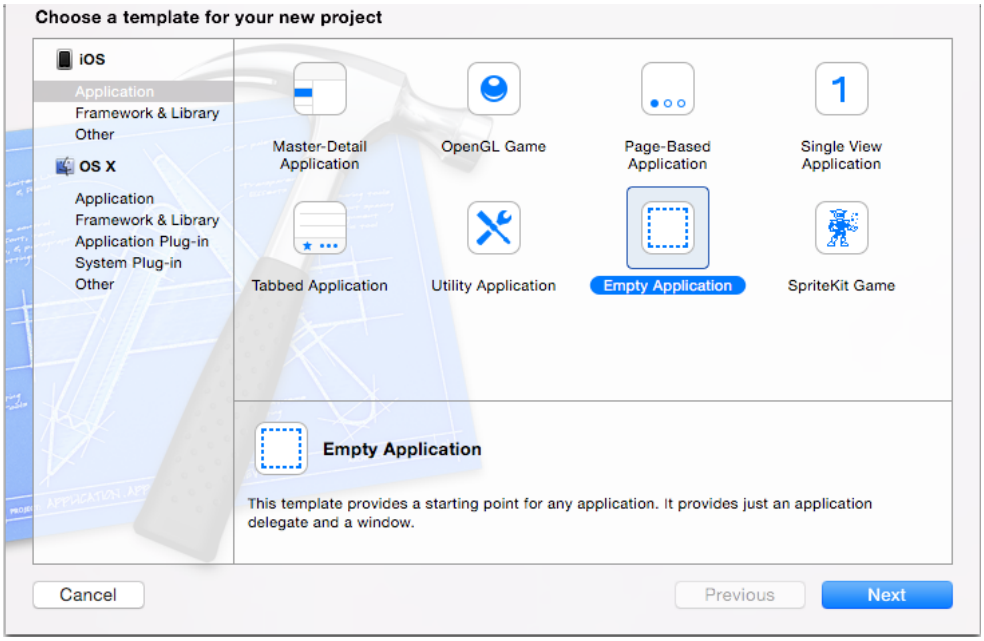
Building Bloc Browser

Let's combine everything we've learned so far into a new app: a web browser.

In the last project we started with a *Single View Application*. For this project we want more control so we'll create a new empty application.

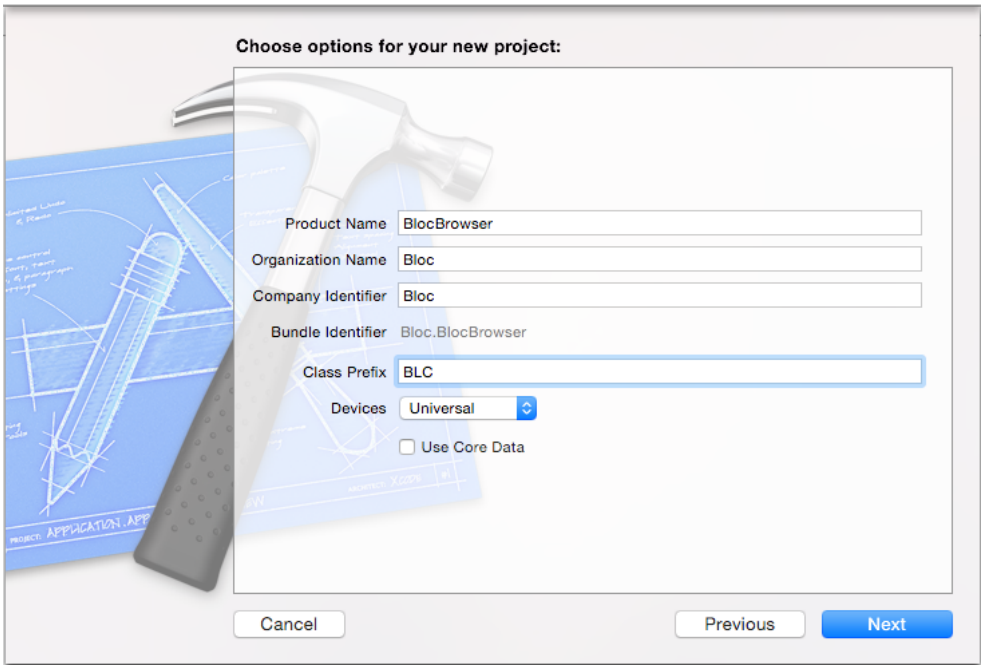
Create a New Empty Application

Let's create the project:



Enter the following details for the application:

- **Product Name**: BlocBrowser
- **Class Prefix**: BLC
- **Devices**: Universal



Check "Create git repository" when selecting a location.

Run the application immediately and Xcode will print a warning saying:

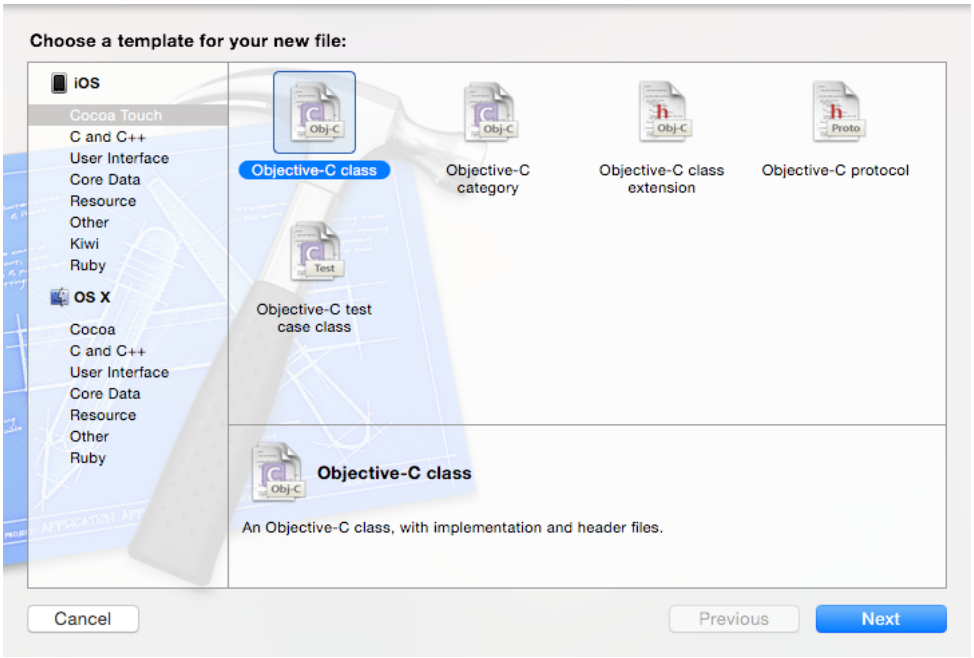
> *Application windows are expected to have a root view controller at the end of application launch.*

In iOS, **`UIViewController`**s exist in a hierarchy. When an application launches, the **`UIViewController`** that takes control is known as the **root view controller**.

We'll create the view controller and then set it as the root view controller when the application launches.

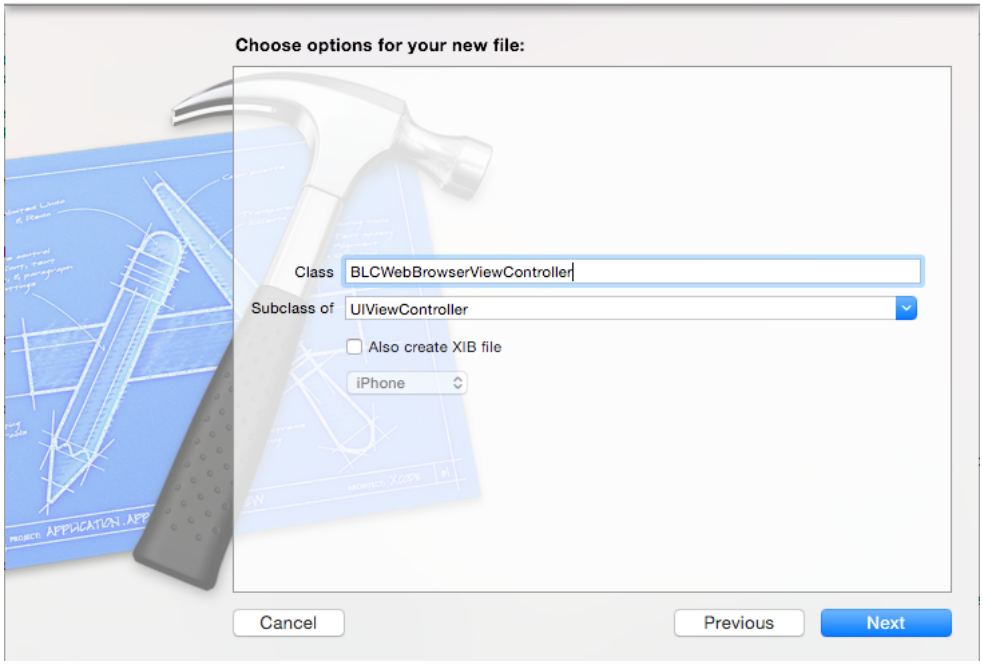
Creating the Main View Controller

Create a new file and select **Objective-C class** as the file template.

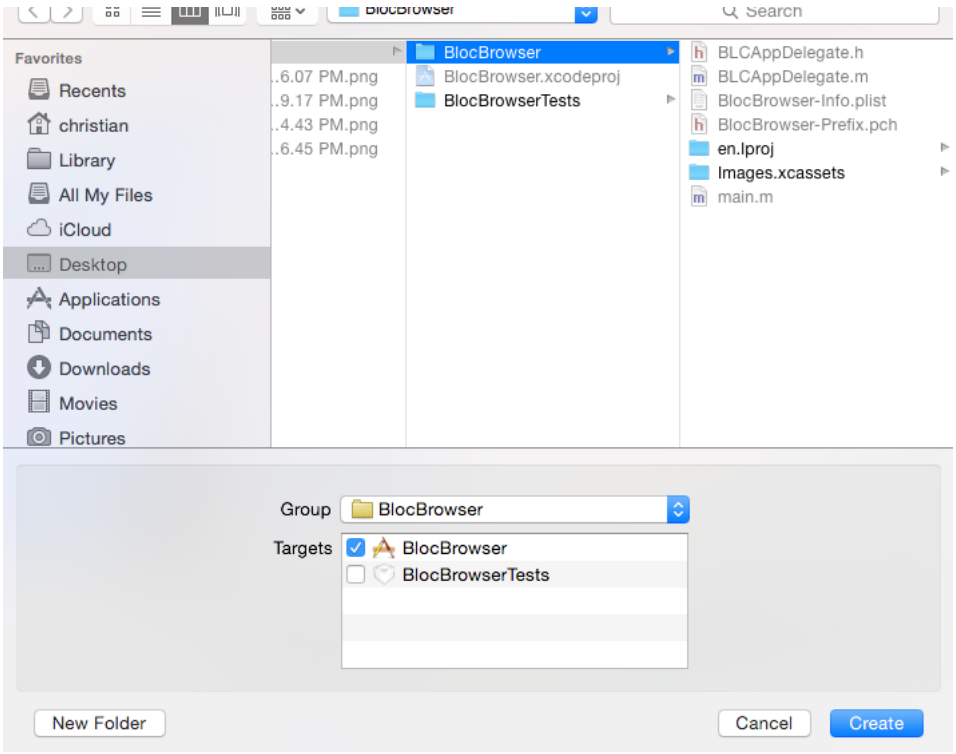


Enter these file options:

- **Class**: BLCWebBrowserViewController
- **Subclass Of**: UIViewController



Save the file in the BlocBrowser directory.



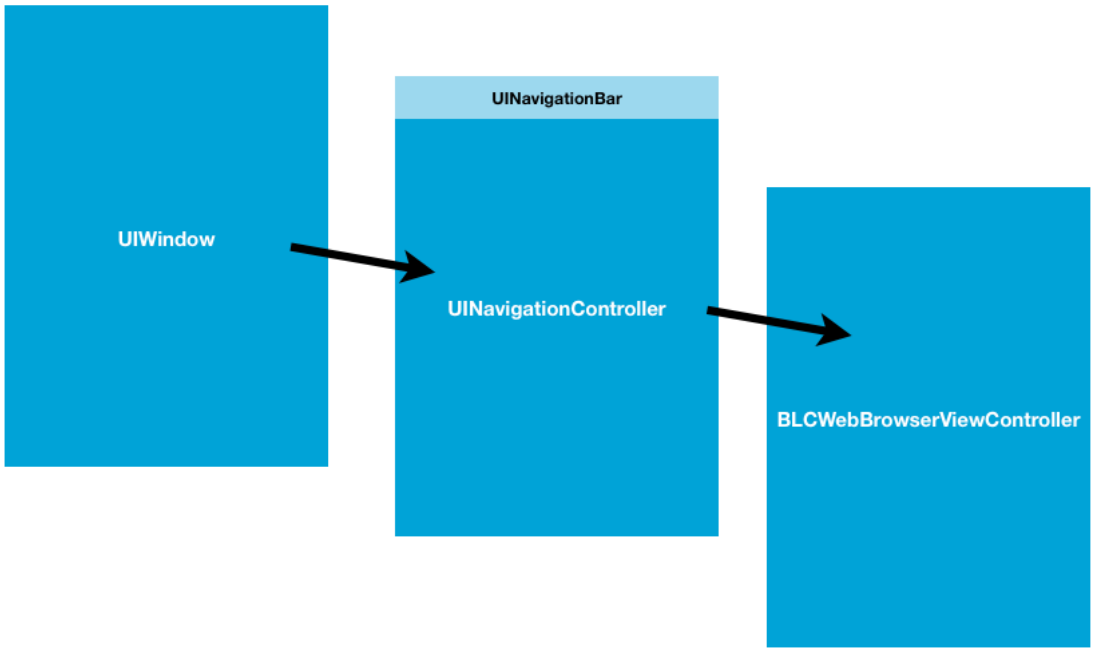
Import the file into **`BLCAppDelegate.m`**:

BLCAppDelegate.m

```
 #import "BLCAppDelegate.h"
+ #import "BLCWebBrowserViewController.h"

@implementation BLCAppDelegate

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```

For the **`rootViewController`** we'll use a **`UINavigationController`** since it will provide access to a **navigation bar** that we can use for web page titles.

Next we'll set **`BLCWebBrowserView`** as the **`rootViewController`** of the navigation controller, resulting in this hierarchy:



Add the following code:

BLCAppDelegate.m

```
#import "BLCAppDelegate.h"
#import "BLCWebBrowserViewController.h"

@implementation BLCAppDelegate

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.
    self.window.backgroundColor = [UIColor whiteColor];
+     self.window.rootViewController = [[UINavigationController alloc] initWithRootViewController:[[BLCWebBrowserViewController alloc]

    [self.window makeKeyAndVisible];
    return YES;
}
```

Deleting Unused Methods

To clean up the **`BLCWebBrowserViewController`** we'll remove some of the boilerplate code. Delete **`initWithNibName:bundle:`** and **`didReceiveMemoryWarning`** methods since we won't be using them:

BLCWebBrowserViewController.m

```
.

@interface BLCWebBrowserViewController ()

@end

@implementation BLCWebBrowserViewController

- - (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil
- {
-     self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
-     if (self) {
-         // Custom initialization
-     }
-     return self;
- }

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

- - (void)didReceiveMemoryWarning
- {
-     [super didReceiveMemoryWarning];
-     // Dispose of any resources that can be recreated.
- }
-
- /*
- #pragma mark - Navigation
-
- // In a storyboard-based application, you will often want to do a little preparation before navigation
- - (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
- {
-     // Get the new view controller using [segue destinationViewController].
-     // Pass the selected object to the new view controller.
- }
- */
```

Creating the Main View

We'll need a main container view in which we will place all our subviews. Create one by overriding the **`loadView`** method.

BLCWebBrowserViewController.m

```
@implementation BLCWebBrowserViewController

+ - (void)loadView {
+     UIView *mainView = [UIView new];
+     self.view = mainView;
+ }

- (void) viewDidLoad {
```

`UIWebView`

Apple makes loading web content easy with a class called **`UIWebView`**. **`UIWebView`** is a **`UIView`** subclass that is designed to display web content, including HTML, CSS, and JavaScript. It can also display other stuff that you see in web browsers and e-mail attachments like PDFs and Microsoft Word documents.

Here are some methods and properties we'll need for our browser:

- the **stopLoading** method tells the web view to stop loading
- the **reload** method tells the web view to reload the current content
- the **loading** property tells you if the web view is currently loading
- the **goBack** and **goForward** methods can be used as button actions to move through web history

Add **UIWebView** as a private property of **BLCWebBrowserViewController.m** and add it as a subview to **mainView**.

BLCWebBrowserViewController.m

```
@interface BLCWebBrowserViewController ()

+ @property (nonatomic, strong) UIWebView *webview;

@end

@implementation BLCWebBrowserViewController

- (void)loadView {
   UIView *mainView = [UIView new];

+    self.webview = [[UIWebView alloc] init];
+    self.webview.delegate = self;
+
+    [mainView addSubview:self.webview];
   self.view = mainView;
}
```

In order to use our view controller as the delegate for the **UIWebView** we need to declare that our controller conforms to the **UIWebViewDelegate** protocol:

BLCWebBrowserViewController.m

```
- @interface BLCWebBrowserViewController ()
+ @interface BLCWebBrowserViewController () <UIWebViewDelegate>

@property (nonatomic, strong) UIWebView *webview;

@end

@implementation BLCWebBrowserViewController

- (void)loadView {
    UIView *mainView = [UIView new];

    self.webview = [[UIWebView alloc] init];
    self.webview.delegate = self;

    [mainView addSubview:self.webview];
    self.view = mainView;
}
```

Resizing the Web View:

Before we can show anything in our web view we must first give it a size. For now we'll set its frame to be the same as the main view frame which will make it fill the main view.

BLCWebBrowserViewController.m

```
{
    [super viewDidLoad];
    // Do any additional setup after loading the view.
}

+ - (void) viewWillLayoutSubviews {
+     [super viewWillLayoutSubviews];
+
+     //make the webview fill the main view
+     self.webview.frame = self.view.frame;
+
+ }

@end
```

NOTE: we do this in **`viewWillLayoutSubviews`** because before that point the main view is not guaranteed to have adjusted to any rotation or resizing events.

Requesting web sites: **`NSURLRequest`** and **`NSURL`**

We need to tell our **`UIWebView`** instance which site to load. To do this, we'll need to construct an **`NSURLRequest`** object, which we haven't used yet. Here are the basics:

`NSURLRequest` objects represent a request to load a URL. The most common URLs look like **`http://www.bloc.io`**, but this object can be used to represent other protocols (like FTP) or other schemes (like **`file://`** or **`news://`**).

The URL itself is represented as an **`NSURL`** object, which will be covered in more detail in a later checkpoint.

Here's how a request is typically created:

```
NSString *blocURLString = @"http://bloc.io";
NSURL *blocURL = [NSURL URLWithString:blocURLString];
NSURLRequest *blocURLRequest = [NSURLRequest requestWithURL:blocURL];
```

As an experiment, let's make our web view load wikipedia.org when the view loads

BLCWebBrowserViewController.m

```
- (void)loadView {
    UIView *mainView = [UIView new];

    self.webview = [[UIWebView alloc] init];
    self.webview.delegate = self;

+     NSString *urlString = @"http://wikipedia.org";
+     NSURL *url = [NSURL URLWithString:urlString];
+     NSURLRequest *request = [NSURLRequest requestWithURL:url];
+     [self.webview loadRequest:request];

    [mainView addSubview:self.webview];
    self.view = mainView;
}
```

Run the app and it should load wikipedia's homepage.

Create URL Text Field Programmatically

Let's add a **`UITextField`** to serve as the URL bar.

First, add a property (you should notice a pattern here):

BLCWebBrowserViewController.m

```
@interface BLCWebBrowserViewController () <UIWebViewDelegate>

@property (nonatomic, strong) UIWebView *webview;
+ @property (nonatomic, strong) UITextField *textField;

@end
```

Declare that the view controller conforms to the **`UITextFieldDelegate`**:

BLCWebBrowserViewController.m

```
- @interface BLCWebBrowserViewController () <UIWebViewDelegate>
+ @interface BLCWebBrowserViewController () <UIWebViewDelegate, UITextFieldDelegate>

@property (nonatomic, strong) UIWebView *webview;
@property (nonatomic, strong) UITextField *textField;

@end
```

Finally build the text field and add it as a subview of the main view:

```
- (void)loadView {
    UIView *mainView = [UIView new];

    self.webview = [[UIWebView alloc] init];
    self.webview.delegate = self;

+     self.textField = [[UITextField alloc] init];
+     self.textField.keyboardType = UIKeyboardTypeURL;
+     self.textField.returnKeyType = UIReturnKeyDone;
+     self.textField.autocapitalizationType = UITextAutocapitalizationTypeNone;
+     self.textField.autocorrectionType = UITextAutocorrectionTypeNo;
+     self.textField.placeholder = NSLocalizedString(@"Website URL", @"Placeholder text for web browser URL field");
+     self.textField.backgroundColor = [UIColor colorWithWhite:220/255.0f alpha:1];
+     self.textField.delegate = self;

    NSString *urlString = @"http://wikipedia.org";
    NSURL *url = [NSURL URLWithString:urlString];
    NSURLRequest *request = [NSURLRequest requestWithURL:url];
    [self.webview loadRequest:request];

    [mainView addSubview:self.webview];
+     [mainView addSubview:self.textField];
    self.view = mainView;
}
```

> ***NOTE**: Many of the properties that we're configuring correspond to the options we had while storyboarding. It's helpful to know that every option that you have in Interface Builder is configurable from code. Take a moment to hold down the **option** key and click on some of the properties, such as `placeholder`, `returnKeyType`, and `autocapitalizationType`, to see what they do.*

Now we need to adjust our layout to show the URL field on the screen.

BLCWebBrowserViewController.m

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

-       //make the webview fill the main view
-       self.webview.frame = self.view.frame;

+       // First, calculate some dimensions.
+       static const CGFloat itemHeight = 50;
+       CGFloat width = CGRectGetWidth(self.view.bounds);
+       CGFloat browserHeight = CGRectGetHeight(self.view.bounds) - itemHeight;

+       // Now, assign the frames
+       self.textField.frame = CGRectMake(0, 0, width, itemHeight);
+       self.webview.frame = CGRectMake(0, CGRectGetMaxY(self.textField.frame), width, browserHeight);
}
```

Let's go through this line-by-line.

We want our URL bar to have a height of 50 so we create a variable for that.

```
static const CGFloat itemHeight = 50;
```

> `static` *keeps the value the same between invocations of the method.* `const` *tells the compiler that this value won't change, allowing for additional speed optimizations.*

Next, we calculate the width to be the same as the view width. We'll reuse this variable for both views.

```
CGFloat width = CGRectGetWidth(self.view.bounds);
```

We then calculate the height of the browser view to be the height of the entire main view, minus the height of the URL bar.

```
CGFloat browserHeight = CGRectGetHeight(self.view.bounds) - itemHeight;
```

Before, the browser could be the same height as the main view, but now it has to shrink to fit the URL bar.

Next, we make the frames for each view. The text field is simple enough: It's positioned at 0 x 0, and we'll use the **`width`** and **`itemHeight`** values we created for the size.

```
self.textField.frame = CGRectMake(0, 0, width, itemHeight);
```

The browser frame is similar, with the addition of a trick to figure out the value of its Y-origin.

```
self.webview.frame = CGRectMake(0, CGRectGetMaxY(self.textField.frame), width, browserHeight);
```

We use the function **`CGRectGetMaxY`** to figure out where the bottom of the text field will be. In this scenario we could have also used **`itemHeight`**, but the way we chose has the advantage of continuing to work even if we move the text field somewhere else later.

Finally, we need to set the **`edgesForExtendedLayout`** property on our view controller:

BLCWebBrowserViewController.m

```
- (void)viewDidLoad
{
      [super viewDidLoad];
+       self.edgesForExtendedLayout = UIRectEdgeNone;

      // Do any additional setup after loading the view.
}
```

Some apps scroll their content up under the navigation bar and behind the status bar. Setting **`edgesForExtendedLayout`** to **`UIRectEdgeNone`** opts out of this behavior.

Now we should have our URL field displaying at the top of our view

Make the URL Field work

Next we need to make **`UIWebView`** load the URLs we enter into the URL field.

Remove the hardcoded url that we use in the **`loadView`** method:

BLCWebBrowserViewController.m

```
- (void)loadView {
    UIView *mainView = [UIView new];

    self.webview = [[UIWebView alloc] init];
    self.webview.delegate = self;


    self.textField = [[UITextField alloc] init];
    self.textField.keyboardType = UIKeyboardTypeURL;
    self.textField.returnKeyType = UIReturnKeyDone;
    self.textField.autocapitalizationType = UITextAutocapitalizationTypeNone;
    self.textField.autocorrectionType = UITextAutocorrectionTypeNo;
    self.textField.placeholder = NSLocalizedString(@"Website URL", @"Placeholder text for web browser URL field");
    self.textField.backgroundColor = [UIColor colorWithWhite:220/255.0f alpha:1];
    self.textField.delegate = self;

-     NSString *urlString = @"http://wikipedia.org";
-     NSURL *url = [NSURL URLWithString:urlString];
-     NSURLRequest *request = [NSURLRequest requestWithURL:url];
-     [self.webview loadRequest:request];

    [mainView addSubview:self.webview];
    [mainView addSubview:self.textField];
    self.view = mainView;
}
```

BLCWebBrowserViewController.m

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

    // First, calculate some dimensions.
    static const CGFloat itemHeight = 50;
    CGFloat width = CGRectGetWidth(self.view.bounds);
    CGFloat browserHeight = CGRectGetHeight(self.view.bounds) - itemHeight;

    // Now, assign the frames
    self.textField.frame = CGRectMake(0, 0, width, itemHeight);
    self.webview.frame = CGRectMake(0, CGRectGetMaxY(self.textField.frame), width, browserHeight);

}

+ - (BOOL)textFieldShouldReturn:(UITextField *)textField {
+     [textField resignFirstResponder];
+
+     NSString *URLString = textField.text;
+
+     NSURL *URL = [NSURL URLWithString:URLString];
+
+     if (URL) {
+         NSURLRequest *request = [NSURLRequest requestWithURL:URL];
+         [self.webview loadRequest:request];
+     }
+
+     return NO;
+ }
```

Run the app again. Enter a complete URL like "**http://www.bloc.io**" and press enter. The requested page should load.

Correcting User Mistakes

What happens if a user forgets to add **`http://`** before a URL? A good application should try to interpret what they meant and correct the mistake. We can use this nifty trick:

BLCWebBrowserViewController.m

```
- (BOOL)textFieldShouldReturn:(UITextField *)textField {
    [textField resignFirstResponder];

    NSString *URLString = textField.text;

    NSURL *URL = [NSURL URLWithString:URLString];

+    if (!URL.scheme) {
+        // The user didn't type http: or https:
+        URL = [NSURL URLWithString:[NSString stringWithFormat:@"http://%@", URLString]];
+    }

    if (URL) {
        NSURLRequest *request = [NSURLRequest requestWithURL:URL];
        [self.webview loadRequest:request];
    }

    return NO;
}
```

If the URL doesn't have a scheme, we will assume they meant *http://* and add it for them.

Handling Failure

A web page could fail to load for any number of reasons. Whether it's a poor wifi signal, subway, or **a data center engulfed in flames**

To handle web page load failures we rely on a method from the **`UIWebViewDelegate`** protocol.

UIWebViewDelegate

Just like **`UITextField`** tells its **`delegate`** when text changes, **`UIWebView`** tells its **`delegate`** about web pages loading.

Let's take a look at the web view's delegate protocol to see what we can do.

Press ⌘⇧**O** to open the Xcode documentation, and search for **`UIWebViewDelegate`**.

There are four methods:

```
- webView:shouldStartLoadWithRequest:navigationType:
- webViewDidStartLoad:
- webViewDidFinishLoad:
- webView:didFailLoadWithError:
```

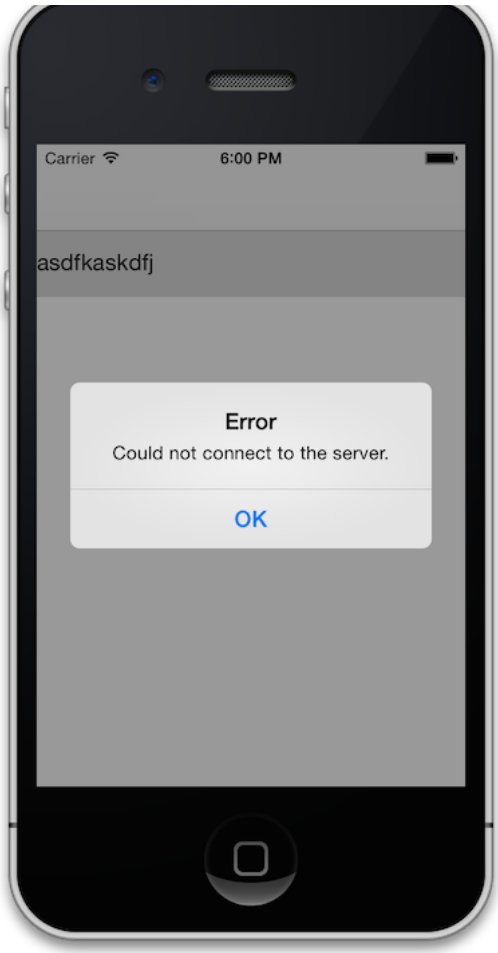
According to the documentation the **`webView:didFailLoadWithError:`** will be called if a web page fails to load. We'll handle this by showing an alert to the user.

BLCWebBrowserViewController.m

```
- (BOOL)textFieldShouldReturn:(UITextField *)textField {
    [textField resignFirstResponder];

    NSString *URLString = textField.text;

    NSURL *URL = [NSURL URLWithString:URLString];

    if (!URL.scheme) {
        // The user didn't type http: or https:
        URL = [NSURL URLWithString:[NSString stringWithFormat:@"http://%@", URLString]];
    }

    if (URL) {
        NSURLRequest *request = [NSURLRequest requestWithURL:URL];
        [self.webview loadRequest:request];
    }

    return NO;
}

+ - (void)webView:(UIWebView *)webView didFailLoadWithError:(NSError *)error {
+     UIAlertView *alert = [[UIAlertView alloc] initWithTitle:NSLocalizedString(@"Error", @"Error")
+                                                     message:[error localizedDescription]
+                                                    delegate:nil
+                                           cancelButtonTitle:NSLocalizedString(@"OK", nil)
+                                           otherButtonTitles:nil];
+     [alert show];
+ }
```

We'll use **`UIAlertView`** to show our error.

Run the app and enter a fake URL like "bloc.ios": (note that you don't need to type "http://" anymore)



Organizing our Code with **`#pragma mark`**

As the code in a **`UIViewController`** grows in length, inserting some code organization can add clarity and peace-of-mind.

For that we have the *Pragma Mark*.

A pragma mark is a special kind of comment that can be used to break code up into related chunks.

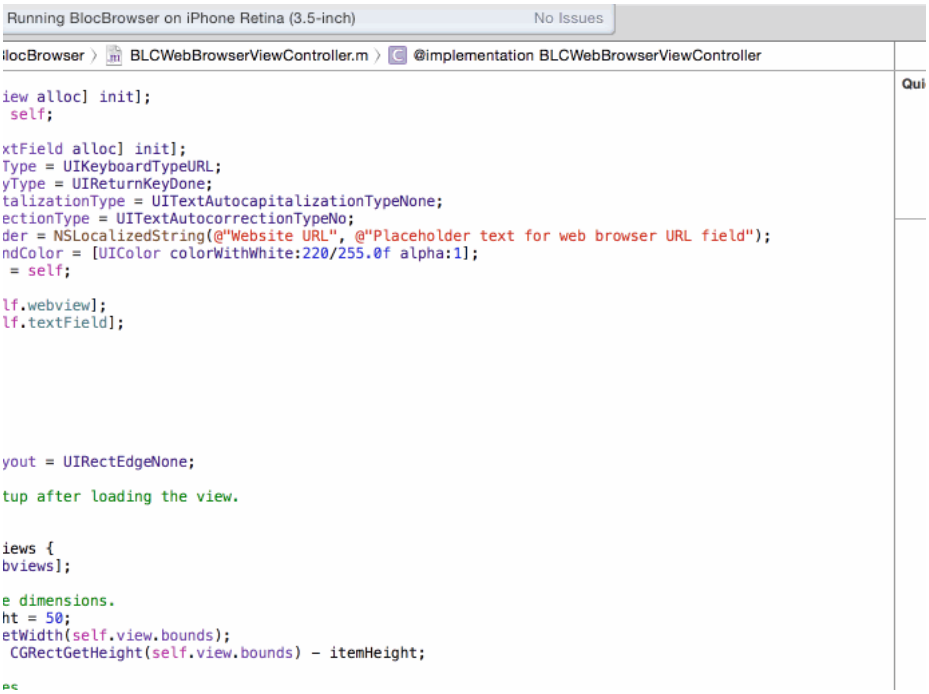
We'll break up our **`UIViewController`** based on different areas of responsibility.

BLCWebBrowserViewController.m

```
@implementation BLCWebBrowserViewController

 + #pragma mark - UIViewController

- (void)loadView {
    UIView *mainView = [UIView new];

    self.webview = [[UIWebView alloc] init];
    self.webview.delegate = self;

    self.textField = [[UITextField alloc] init];
    self.textField.keyboardType = UIKeyboardTypeURL;
    self.textField.returnKeyType = UIReturnKeyDone;
    self.textField.autocapitalizationType = UITextAutocapitalizationTypeNone;
    self.textField.autocorrectionType = UITextAutocorrectionTypeNo;
    self.textField.placeholder = NSLocalizedString(@"Website URL", @"Placeholder text for web browser URL field");
    self.textField.backgroundColor = [UIColor colorWithWhite:220/255.0f alpha:1];
    self.textField.delegate = self;
```

```
    self.view = mainView;
}


- (void)viewDidLoad
{
    [super viewDidLoad];
    self.edgesForExtendedLayout = UIRectEdgeNone;

    // Do any additional setup after loading the view.
}

- (void) viewWillLayoutSubviews {
    [super viewWillLayoutSubviews];

    // First, calculate some dimensions.
    static const CGFloat itemHeight = 50;
    CGFloat width = CGRectGetWidth(self.view.bounds);
    CGFloat browserHeight = CGRectGetHeight(self.view.bounds) - itemHeight;

    // Now, assign the frames
    self.textField.frame = CGRectMake(0, 0, width, itemHeight);
    self.webview.frame = CGRectMake(0, CGRectGetMaxY(self.textField.frame), width, browserHeight);

}

 + #pragma mark - UITextFieldDelegate

- (BOOL)textFieldShouldReturn:(UITextField *)textField {
    [textField resignFirstResponder];

    NSString *URLString = textField.text;

    NSURL *URL = [NSURL URLWithString:URLString];

    if (!URL.scheme) {
        // The user didn't type http: or https:
        URL = [NSURL URLWithString:[NSString stringWithFormat:@"http://%@", URLString]];
    }

    if (URL) {
        NSURLRequest *request = [NSURLRequest requestWithURL:URL];
        [self.webview loadRequest:request];
    }

    return NO;
}

 + #pragma mark - UIWebViewDelegate

- (void)webView:(UIWebView *)webView didFailLoadWithError:(NSError *)error {
    if (error.code != -999) {
        UIAlertView *alert = [[UIAlertView alloc] initWithTitle:NSLocalizedString(@"Error", @"Error")
                                                        message:[error localizedDescription]
                                                       delegate:nil
                                              cancelButtonTitle:NSLocalizedString(@"OK", nil)
                                              otherButtonTitles:nil];
        [alert show];
    }
}

@end
```

This makes a nice table of contents effect in the class navigator.



Building the Button Bar

The next step is to build the button bar that will hold the basic navigation controls for our browser.

Just like with the text field we will create properties for each button, configure them, and add them to our view.

BLCWebBrowserViewController.m

```
@property (nonatomic, strong) UIWebView *webview;
@property (nonatomic, strong) UITextField *textField;
+ @property (nonatomic, strong) UIButton *backButton;
+ @property (nonatomic, strong) UIButton *forwardButton;
+ @property (nonatomic, strong) UIButton *stopButton;
+ @property (nonatomic, strong) UIButton *reloadButton;

@end

@implementation BLCWebBrowserViewController

#pragma mark - UIViewController

- (void)loadView {
    UIView *mainView = [UIView new];

    self.webview = [[UIWebView alloc] init];
    self.webview.delegate = self;

    self.textField = [[UITextField alloc] init];
    self.textField.keyboardType = UIKeyboardTypeURL;
    self.textField.returnKeyType = UIReturnKeyDone;
    self.textField.autocapitalizationType = UITextAutocapitalizationTypeNone;
    self.textField.autocorrectionType = UITextAutocorrectionTypeNo;
    self.textField.placeholder = NSLocalizedString(@"Website URL", @"Placeholder text for web browser URL field");
    self.textField.backgroundColor = [UIColor colorWithWhite:220/255.0f alpha:1];
    self.textField.delegate = self;

+     self.backButton = [UIButton buttonWithType:UIButtonTypeSystem];
+     [self.backButton setEnabled:NO];

+     self.forwardButton = [UIButton buttonWithType:UIButtonTypeSystem];
+     [self.forwardButton setEnabled:NO];

+     self.stopButton = [UIButton buttonWithType:UIButtonTypeSystem];
+     [self.stopButton setEnabled:NO];

+     self.reloadButton = [UIButton buttonWithType:UIButtonTypeSystem];
+     [self.reloadButton setEnabled:NO];

+     [self.backButton setTitle:NSLocalizedString(@"Back", @"Back comnmand") forState:UIControlStateNormal];
+     [self.backButton addTarget:self.webview action:@selector(goBack) forControlEvents:UIControlEventTouchUpInside];

+     [self.forwardButton setTitle:NSLocalizedString(@"Forward", @"Forward comnmand") forState:UIControlStateNormal];
+     [self.forwardButton addTarget:self.webview action:@selector(goForward) forControlEvents:UIControlEventTouchUpInside];

+     [self.stopButton setTitle:NSLocalizedString(@"Stop", @"Stop comnmand") forState:UIControlStateNormal];
+     [self.stopButton addTarget:self.webview action:@selector(stopLoading) forControlEvents:UIControlEventTouchUpInside];

+     [self.reloadButton setTitle:NSLocalizedString(@"Refresh", @"Reload comnmand") forState:UIControlStateNormal];
+     [self.reloadButton addTarget:self.webview action:@selector(reload) forControlEvents:UIControlEventTouchUpInside];

    [mainView addSubview:self.webview];
    [mainView addSubview:self.textField];
+     [mainView addSubview:self.backButton];
+     [mainView addSubview:self.forwardButton];
+     [mainView addSubview:self.stopButton];
+     [mainView addSubview:self.reloadButton];

    self.view = mainView;
}
```

Before we go further we should refactor the repetitive code we're using to add each view to the main view.

We can make this code shorter if we use a loop.

BLCWebBrowserViewController.m

```
-     [mainView addSubview:self.textField];
-     [mainView addSubview:self.backButton];
-     [mainView addSubview:self.forwardButton];
-     [mainView addSubview:self.stopButton];
-     [mainView addSubview:self.reloadButton];
+     for (UIView *viewToAdd in @[self.webview, self.textField, self.backButton, self.forwardButton, self.stopButton, self.reloadButto
+         [mainView addSubview:viewToAdd];
+     }

    self.view = mainView;
```

Now let's adjust our layout code to accommodate the buttons. First we should make **`browserHeight`** smaller (to make room for the new buttons). We'll do that by subtracting an additional **`itemHeight`**'s worth of space from its height:

BLCWebBrowserViewController.m

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

    // First, calculate some dimensions.
    static const CGFloat itemHeight = 50;
    CGFloat width = CGRectGetWidth(self.view.bounds);
-     CGFloat browserHeight = CGRectGetHeight(self.view.bounds) - itemHeight;
+     CGFloat browserHeight = CGRectGetHeight(self.view.bounds) - itemHeight - itemHeight;

    // Now, assign the frames
    self.textField.frame = CGRectMake(0, 0, width, itemHeight);
    self.webview.frame = CGRectMake(0, CGRectGetMaxY(self.textField.frame), width, browserHeight);

}
```

Next we should create a variable to determine the width of each button.

BLCWebBrowserViewController.m

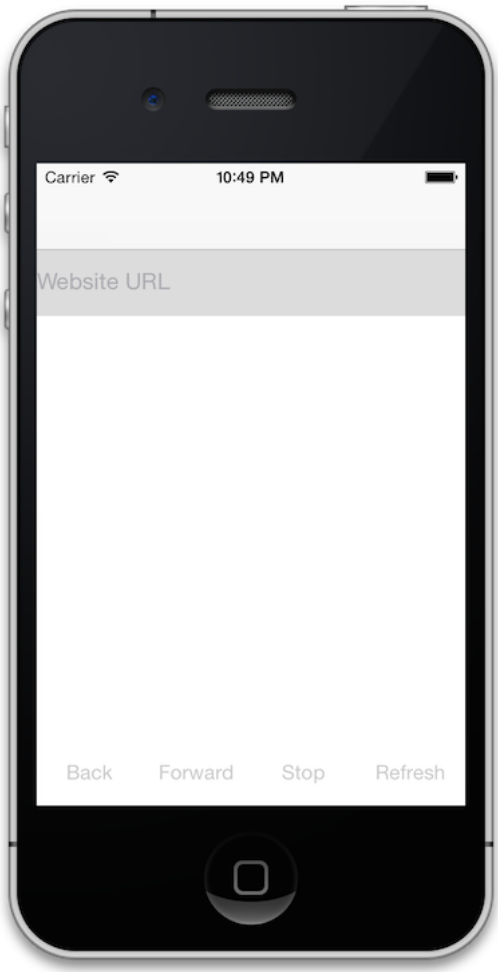
```
- (void) viewWillLayoutSubviews {
    [super viewWillLayoutSubviews];

    // First, calculate some dimensions.
    static const CGFloat itemHeight = 50;
    CGFloat width = CGRectGetWidth(self.view.bounds);
    CGFloat browserHeight = CGRectGetHeight(self.view.bounds) - itemHeight;
    CGFloat browserHeight = CGRectGetHeight(self.view.bounds) - itemHeight - itemHeight;
+     CGFloat buttonWidth = CGRectGetWidth(self.view.bounds) / 4;

    // Now, assign the frames
    self.textField.frame = CGRectMake(0, 0, width, itemHeight);
    self.webview.frame = CGRectMake(0, CGRectGetMaxY(self.textField.frame), width, browserHeight);

}
```

We'll then add another loop to handle the positioning of each button.

BLCWebBrowserViewController.m

```
    [super viewWillLayoutSubviews];

    // First, calculate some dimensions.
    static const CGFloat itemHeight = 50;
    CGFloat width = CGRectGetWidth(self.view.bounds);
    CGFloat browserHeight = CGRectGetHeight(self.view.bounds) - itemHeight;
    CGFloat browserHeight = CGRectGetHeight(self.view.bounds) - itemHeight - itemHeight;

    // Now, assign the frames
    self.textField.frame = CGRectMake(0, 0, width, itemHeight);
    self.webview.frame = CGRectMake(0, CGRectGetMaxY(self.textField.frame), width, browserHeight);

+   CGFloat currentButtonX = 0;
+
+   for (UIButton *thisButton in @[self.backButton, self.forwardButton, self.stopButton, self.reloadButton]) {
+       thisButton.frame = CGRectMake(currentButtonX, CGRectGetMaxY(self.webview.frame), buttonWidth, itemHeight);
+       currentButtonX += buttonWidth;
+   }
}
```

We should now have buttons.



Updating Buttons and Titles

There are several UI elements that we'll want to update depending on the changing state of the web view:

- When a web page loads, the title in the **`UINavigationBar`** should update to reflect the page title.
- The buttons in the button bar should become enabled or disabled depending on whether or not you can use them.

We'll write a single update method that will handle all these UI updates.

BLCWebBrowserViewController.m

```
- (void)webView:(UIWebView *)webView didFailLoadWithError:(NSError *)error {
    if (error.code != -999) {
        UIAlertView *alert = [[UIAlertView alloc] initWithTitle:NSLocalizedString(@"Error", @"Error")
                                                        message:[error localizedDescription]
                                                       delegate:nil
                                              cancelButtonTitle:NSLocalizedString(@"OK", nil)
                                              otherButtonTitles:nil];
        [alert show];
    }

+    [self updateButtonsAndTitle];
}

+ #pragma mark - Miscellaneous
+
+ - (void) updateButtonsAndTitle {
+
+ }
```

We need to call this method whenever a page starts or stops loading. We'll use the the **`UIWebViewDelegate`** methods **`webViewDidStartLoad`**, and **`webViewDidFinishLoad:`** to do this:

BLCWebBrowserViewController.m

```
#pragma mark - UIWebViewDelegate

+ - (void)webViewDidStartLoad:(UIWebView *)webView {
+    [self updateButtonsAndTitle];
+ }

+ - (void)webViewDidFinishLoad:(UIWebView *)webView {
+    [self updateButtonsAndTitle];
+ }

- (void)webView:(UIWebView *)webView didFailLoadWithError:(NSError *)error {
```

Next we'll make title in the **`UINavigationBar`** update to reflect whatever page is loaded in the web view. The best way to get the title of a webpage in a **`UIWebView`** is by running JavaScript directly inside of it.

BLCWebBrowserViewController.m

```
- (void) updateButtonsAndTitle {
+    NSString *webpageTitle = [self.webview stringByEvaluatingJavaScriptFromString:@"document.title"];

+    if (webpageTitle) {
+        self.title = webpageTitle;
+    } else {
+        self.title = self.webview.request.URL.absoluteString;
+    }
}
```

Next, the forward and back buttons:

BLCWebBrowserViewController.m

```
    NSString *webpageTitle = [self.webview stringByEvaluatingJavaScriptFromString:@"document.title"];

    if (webpageTitle) {
        self.title = webpageTitle;
    } else {
        self.title = self.webview.request.URL.absoluteString;
    }

+    self.backButton.enabled = [self.webview canGoBack];
+    self.forwardButton.enabled = [self.webview canGoForward];
}
```

The logic for these is pretty simple. The enabled state for the forward and back buttons is entirely dependent on whether or not the web view *can* go forward or back. As for the stop and refresh buttons, it gets a little trickier.

One option is to create a new **`BOOL`** property named **`isLoading`**:

BLCWebBrowserViewController.m

```
@interface BLCWebBrowserViewController () <UIWebViewDelegate, UITextFieldDelegate>

@property (nonatomic, strong) UIWebView *webview;
@property (nonatomic, strong) UITextField *textField;
@property (nonatomic, strong) UIButton *backButton;
@property (nonatomic, strong) UIButton *forwardButton;
@property (nonatomic, strong) UIButton *stopButton;
@property (nonatomic, strong) UIButton *reloadButton;

+ @property (nonatomic, assign) BOOL isLoading;

@end
```

Then when **`webViewDidStartLoad`** is called we set the **`isLoading`** property to **`YES`** and then set it back to **`NO`** when **`webViewDidFinishLoad`** is called.

BLCWebBrowserViewController.m

```
- (void)webViewDidStartLoad:(UIWebView *)webView {
+    self.isLoading = YES;
    [self updateButtonsAndTitle];
}

- (void)webViewDidFinishLoad:(UIWebView *)webView {
+    self.isLoading = NO;
    [self updateButtonsAndTitle];
}
```

Finally we change the buttons' enabled state based on the current value of **`isLoading`**:

BLCWebBrowserViewController.m

```
- (void) updateButtonsAndTitle {
    NSString *webpageTitle = [self.webview stringByEvaluatingJavaScriptFromString:@"document.title"];

    if (webpageTitle) {
        self.title = webpageTitle;
    } else {
        self.title = self.webview.request.URL.absoluteString;
    }

    self.backButton.enabled = [self.webview canGoBack];
    self.forwardButton.enabled = [self.webview canGoForward];
+    self.stopButton.enabled = self.isLoading;
+    self.reloadButton.enabled = !self.isLoading;
}
```

Adding the Activity Indicator Spinner

To indicate a page loading we'll use a simple class called **`UIActivityIndicator`**. We'll stick it in the **`rightToolbarItem`** position of the **`UINavigationBar`**.

Create a property for the activity indicator.

BLCWebBrowserViewController.m

```
@interface BLCWebBrowserViewController () <UIWebViewDelegate, UITextFieldDelegate>

@property (nonatomic, strong) UIWebView *webview;
@property (nonatomic, strong) UITextField *textField;
@property (nonatomic, strong) UIButton *backButton;
@property (nonatomic, strong) UIButton *forwardButton;
@property (nonatomic, strong) UIButton *stopButton;
@property (nonatomic, strong) UIButton *reloadButton;
+ @property (nonatomic, strong) UIActivityIndicatorView *activityIndicator;

@property (nonatomic, assign) BOOL isLoading;

@end
```

Then set it in the viewDidLoad method:

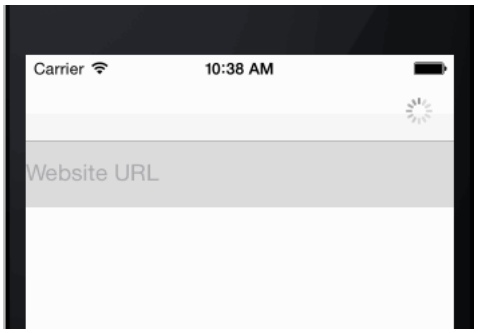
BLCWebBrowserViewController.m

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

    self.edgesForExtendedLayout = UIRectEdgeNone;

+     self.activityIndicator = [[UIActivityIndicatorView alloc] initWithActivityIndicatorStyle:UIActivityIndicatorViewStyleGray];
+     self.navigationItem.rightBarButtonItem = [[UIBarButtonItem alloc] initWithCustomView:self.activityIndicator];
}
```

As a test, try turning it on.

BLCWebBrowserViewController.m

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

    self.edgesForExtendedLayout = UIRectEdgeNone;

    self.activityIndicator = [[UIActivityIndicatorView alloc] initWithActivityIndicatorStyle:UIActivityIndicatorViewStyleGray];
    self.navigationItem.rightBarButtonItem = [[UIBarButtonItem alloc] initWithCustomView:self.activityIndicator];
+     [self.activityIndicator startAnimating];
}
```



Remove the **`startAnimating`** call we just added and add it to **`updateButtonsAndTitle`**.

BLCWebBrowserViewController.m

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

    self.edgesForExtendedLayout = UIRectEdgeNone;

    self.activityIndicator = [[UIActivityIndicatorView alloc] initWithActivityIndicatorStyle:UIActivityIndicatorViewStyleGray];
    self.navigationItem.rightBarButtonItem = [[UIBarButtonItem alloc] initWithCustomView:self.activityIndicator];
-     [self.activityIndicator startAnimating];
}
```

BLCWebBrowserViewController.m

```
#pragma mark - Miscellaneous

- (void) updateButtonsAndTitle {
    NSString *webpageTitle = [self.webview stringByEvaluatingJavaScriptFromString:@"document.title"];

    if (webpageTitle) {
        self.title = webpageTitle;
    } else {
        self.title = self.webview.request.URL.absoluteString;
    }

+     if (self.isLoading) {
+         [self.activityIndicator startAnimating];
+     } else {
+         [self.activityIndicator stopAnimating];
+     }

    self.backButton.enabled = [self.webview canGoBack];
    self.forwardButton.enabled = [self.webview canGoForward];
    self.stopButton.enabled = self.isLoading;
    self.reloadButton.enabled = !self.isLoading;
}
```

A Small Problem with Web Frames

Check out the **documentation for UIWebViewDelegate** and read what it says about **`webViewDidStartLoad:`** and **`webViewDidFinishLoad:`**

`webViewDidStartLoad:`

> *Sent after a web view starts loading a frame.*

`webViewDidFinishLoad:`

What does it mean by "frame"? A single webpage can sometimes be divided into several *subpages*. These are sometimes used to display navigation links or sidebars. More often, they are used for embedding media, such as YouTube videos.

This causes a problem with our code. It's possible that when we visit a page with multiple frames, one of them will finish loading (causing our **`webViewDidFinishLoad`** method to get triggered) and our activity indicator will stop spinning before the page has been entirely loaded.

This means that we can't simply use a **`BOOL`** to keep track of **`isLoading`**. What we need to do instead is keep a *tally* of the number of frames that are still loading. When a frame starts loading, we will increment this number and when a frame stops loading we will decrement this number. We'll know the page is entirely done loading when the number reaches zero.

Start by changing our **`BOOL isLoading`** property to an **`NSUInteger`** property called **`frameCount`**:

BLCWebBrowserViewController.m

```
@interface BLCWebBrowserViewController () <UIWebViewDelegate, UITextFieldDelegate>

@property (nonatomic, strong) UIWebView *webview;
@property (nonatomic, strong) UITextField *textField;
@property (nonatomic, strong) UIButton *backButton;
@property (nonatomic, strong) UIButton *forwardButton;
@property (nonatomic, strong) UIButton *stopButton;
@property (nonatomic, strong) UIButton *reloadButton;
@property (nonatomic, strong) UIActivityIndicatorView *activityIndicator;

- @property (nonatomic, assign) BOOL isLoading;
+ @property (nonatomic, assign) NSUInteger frameCount;

@end
```

Next, let's update the delegate methods methods to modify this number:

BLCWebBrowserViewController.m

```
- (void)webViewDidStartLoad:(UIWebView *)webView {
-     self.isLoading = YES;
+     self.frameCount++;
    [self updateButtonsAndTitle];
}

- (void)webViewDidFinishLoad:(UIWebView *)webView {
-     self.isLoading = NO;
+     self.frameCount--;
    [self updateButtonsAndTitle];
}
```

BLCWebBrowserViewController.m

```
- (void)webView:(UIWebView *)webView didFailLoadWithError:(NSError *)error {
    if (error.code != -999) {
        UIAlertView *alert = [[UIAlertView alloc] initWithTitle:NSLocalizedString(@"Error", @"Error")
                                                        message:[error localizedDescription]
                                                       delegate:nil
                                              cancelButtonTitle:NSLocalizedString(@"OK", nil)
                                              otherButtonTitles:nil];
        [alert show];
    }

    [self updateButtonsAndTitle];
+     self.frameCount--;

}
```

Finally update the references to **`isLoading`** in **`updateButtonsAndTitle`** to use the new **`frameCount`** property.

BLCWebBrowserViewController.m

```
    NSString *webpageTitle = [self.webview stringByEvaluatingJavaScriptFromString:@"document.title"];

    if (webpageTitle) {
        self.title = webpageTitle;
    } else {
        self.title = self.webview.request.URL.absoluteString;
    }

-     if (self.isLoading) {
+     if (self.frameCount > 0) {
        [self.activityIndicator startAnimating];
    } else {
        [self.activityIndicator stopAnimating];
    }

    self.backButton.enabled = [self.webview canGoBack];
    self.forwardButton.enabled = [self.webview canGoForward];
-     self.stopButton.enabled = self.isLoading;
-     self.reloadButton.enabled = !self.isLoading;
+     self.stopButton.enabled = self.frameCount > 0;
+     self.reloadButton.enabled = self.frameCount == 0;
}
```

We now have everything we need for a functional web browser.

Add built-in Google search using the following information:

- If the user's text includes a space, assume it's a search query instead of a URL
- If there's a search query, load a URL like this:

```
google.com/search?q=<search query>
```

- Replace the spaces in their search with + characters. Refer back to the Strings checkpoint if you need help doing this.

  For example, if the user inputs *charlie bit my finger*, the corresponding Google query should be **`http://www.google.com/search?q=charlie+bit+my+finger`**

Additionally, change the text field's placeholder text to indicate that the user can also search.

When you're satisfied with your project and the assignment, commit, merge and push your code to GitHub. Since this is a new project, remember to first create a new repo in Github. You can name it "blocbrowser" - as we do below - or any other name that's to your liking.

Terminal

```
$ git add .
$ git commit -m 'Functional web browser'
$ git remote add origin https://github.com/<username>/blocbrowser.git
$ git push -u origin master
```

Submit the assignment with the relevant commit and repo links. We covered a lot in this checkpoint, so be sure to discuss your questions and concerns with your mentor.

assignment completed

COURSES

- Full Stack Web Development
- Frontend Web Development
- UX Design
- Android Development
- iOS Development

ABOUT BLOC

Our Team | Jobs
Bloc Veterans Program
Employer Sponsored
FAQ
Blog
Engineering Blog
Refer-a-Friend
Privacy Policy
Terms of Service

MADE BY BLOC

Tech Talks & Resources
Programming Bootcamp Comparison
Jottly: A Beginner's Guide to HTML, CSS, Skeleton and Animate.css
Swiftris: Build Your First iOS Game with Swift
Webflow Tutorial: Design Responsive Sites with Webflow
Ruby Warrior
Bloc's Diversity Scholarship

SIGN UP FOR OUR MAILING LIST

Send

hello@bloc.io
Considering enrolling? (404) 480-2562
Partnership / Corporate Inquiries? (650) 741-5682

Roadmap · Previous Checkpoint
Jump to Submission · Next Checkpoir

Clearing Browser History

> *"I have never found a companion that was so companionable as solitude."*
>
> *- Henry David Thoreau, Walden*

Introduction

The web browser market in the App Store is pretty well-saturated, so let's enhance our app by targeting a specific subsection: the privacy-obsessed.

In this checkpoint, we'll update our web browser app to clear all web history when the app becomes inactive.

Clearing the Web History

Create a new branch:

Terminal

```
$ git checkout -b clear-history
```

First, let's add a method to **`BLCWebBrowserViewController`** in order to accomplish this. Declare this method in the **`@interface`** in the **`BLCWebBrowserViewController.h`** file:

BLCWebBrowserViewController.h

```
@interface BLCWebBrowserViewController : UIViewController

+ /**
+  Replaces the web view with a fresh one, erasing all history. Also updates the URL field and toolbar buttons appropriately.
+  */
+ - (void) resetWebView;
+
@end
```

And implement it in the .m file:

BLCWebBrowserViewController.m

```
+ - (void) resetWebView {
+     [self.webview removeFromSuperview];
+
+     UIWebView *newWebView = [[UIWebView alloc] init];
+     newWebView.delegate = self;
+     [self.view addSubview:newWebView];
+
+     self.webview = newWebView;
+
+     [self addButtonTargets];
+
+     self.textField.text = nil;
+     [self updateButtonsAndTitle];
+ }
```

This code does the following:

- removes the old web view from the view hierarchy
- creates a new, empty web view and adds it back in
- clears the URL field
- calls **`addButtonTargets`** to point the buttons to the new web view
- updates the buttons and navigation title to their proper state

You'll notice that Xcode displays a "No visible interface..." error for **`addButtonTargets`**. This is because we haven't implemented that method yet. Let's add **`addButtonTargets`** and resolve this error:

BLCWebBrowserViewController.m

```
+ - (void) addButtonTargets {
+     for (UIButton *button in @[self.backButton, self.forwardButton, self.stopButton, self.reloadButton]) {
+         [button removeTarget:nil action:NULL forControlEvents:UIControlEventTouchUpInside];
+     }
+
+     [self.backButton addTarget:self.webview action:@selector(goBack) forControlEvents:UIControlEventTouchUpInside];
+     [self.forwardButton addTarget:self.webview action:@selector(goForward) forControlEvents:UIControlEventTouchUpInside];
+     [self.stopButton addTarget:self.webview action:@selector(stopLoading) forControlEvents:UIControlEventTouchUpInside];
+     [self.reloadButton addTarget:self.webview action:@selector(reload) forControlEvents:UIControlEventTouchUpInside];
+ }
```

communicate with a web view that no longer exists, and cause a crash.

Here's what **addButtonTargets** does:

- the **for** loop at the beginning will loop through all four of our buttons and remove the reference to the old web view
- the web view is added as a target to each button, just like it is in **loadView**

To make our code cleaner, we should just have **loadView** call **addButtonTargets**. In **loadView**:

BLCWebBrowserViewController.m

```
    [self.backButton setTitle:NSLocalizedString(@"Back", @"Back command") forState:UIControlStateNormal];
-     [self.backButton addTarget:self.webview action:@selector(goBack) forControlEvents:UIControlEventTouchUpInside];
-
    [self.forwardButton setTitle:NSLocalizedString(@"Forward", @"Forward command") forState:UIControlStateNormal];
-     [self.forwardButton addTarget:self.webview action:@selector(goForward) forControlEvents:UIControlEventTouchUpInside];
-
    [self.stopButton setTitle:NSLocalizedString(@"Stop", @"Stop command") forState:UIControlStateNormal];
-     [self.stopButton addTarget:self.webview action:@selector(stopLoading) forControlEvents:UIControlEventTouchUpInside];
-
    [self.reloadButton setTitle:NSLocalizedString(@"Refresh", @"Reload command") forState:UIControlStateNormal];
-     [self.reloadButton addTarget:self.webview action:@selector(reload) forControlEvents:UIControlEventTouchUpInside];
+
+     [self addButtonTargets];
```

Now, if we ever want to change the button targets, we only have to do it in one place.

We have one more small change to make in our view controller. Since it's now possible for a web page to clear after one has appeared, we need to update our logic for enabling the reload button. In **updateButtonsAndTitle**:

BLCWebBrowserViewController.m

```
-     self.reloadButton.enabled = self.frameCount == 0;
+     self.reloadButton.enabled = self.webview.request.URL && self.frameCount == 0;
```

This change ensures that the web view has an **NSURLRequest** with accompanying **NSURL**. Otherwise, there'd be nothing to reload.

App-Wide Changes & Information

Before our next change you'll need to know some general information about how iOS switches between apps, and gives information to apps.

In addition to all of the objects that we create, there are lots of other objects created and managed by iOS. One common object used in almos every app is **UIApplication**. This object represents your current app to the OS, and allows you to do lots of stuff, including:

- enable or disable proximity sensing (closeness to the user's face)
- register for push notifications
- ask iOS for time to do some work in the background when your user leaves the app
- ask iOS if you can open a file in another app
- set the badge number on your app's home screen icon

Like many objects, **UIApplication** has a delegate, commonly referred to as an **app delegate**. An app delegate contains methods that iOS use to communicate information to your app. Every Xcode template creates one for you.

The application delegate method **applicationWillResignActive:** is called whenever the app goes to the background. This happens, for example, if the user presses the home button or receives a phone call.

> *"Background" is one of a few "application states" that your app can be in. See the **App Delegate Resource** to learn more about the application delegate and application states. This information will be helpful for this checkpoint's assignment.*

BLCAppDelegate.m

```
- (void)applicationWillResignActive:(UIApplication *)application
{
    // Sent when the application is about to move from active to inactive state. This can occur for certain types of temporary interru
    // Use this method to pause ongoing tasks, disable timers, and throttle down OpenGL ES frame rates. Games should use this method t
+       UINavigationController *navigationVC = (UINavigationController *)self.window.rootViewController;
+       BLCWebBrowserViewController *browserVC = [[navigationVC viewControllers] firstObject];
+       [browserVC resetWebView];
}
```

This code finds the navigation controller (we know it's the root view controller), looks at its first view controller (which is our **`BLCWebBrowserViewController`** instance), and sends it the **`resetWebView`** message.

Run the app. BlocBrowser should now clear when switching between apps.

Display a welcome message when the user launches the app. You can use code like this:

```
UIAlertView *alert = [[UIAlertView alloc] initWithTitle:NSLocalizedString(@"Welcome!", @"Welcome title")
                                                message:NSLocalizedString(@"Get excited to use the best web browser ever!", @"
                                               delegate:nil
                                      cancelButtonTitle:NSLocalizedString(@"OK, I'm excited!", @"Welcome button title") otherB
[alert show];
```

Send a message to your mentor describing how you think these four apps could reasonably behave in response to the application state changing:

- a banking app with sensitive financial data, like Mint
- a social media app, like Facebook
- a Pong-like game
- an app that lets you take pictures and apply filters to them, like Instagram

---

When you're satisfied with your project and the assignment, commit, merge and push your code to GitHub:

Terminal

```
$ git add .
$ git commit -m 'Automatically clear history when the app is closed'
$ git checkout master
$ git merge clear-history
$ git push
```

Submit this checkpoint's assignment with links for your repo and commit.

assignment completed

COURSES

- Full Stack Web Development
- Frontend Web Development
- UX Design
- Android Development
- iOS Development

ABOUT BLOC

- Our Team | Jobs
- Bloc Veterans Program
- Employer Sponsored
- FAQ
- Blog
- Engineering Blog
- Refer-a-Friend
- Privacy Policy
- Terms of Service

MADE BY BLOC

- Tech Talks & Resources
- Programming Bootcamp Comparison
- Jottly: A Beginner's Guide to HTML, CSS, Skeleton and Animate.css
- Swiftris: Build Your First iOS Game with Swift
- Webflow Tutorial: Design Responsive Sites with Webflow
- Ruby Warrior
- Bloc's Diversity Scholarship

SIGN UP FOR OUR MAILING LIST

Send

hello@bloc.io

Considering enrolling? (404) 480-2562

Partnership / Corporate Inquiries? (650) 741-5682

Roadmap · Previous Checkpoint Jump to Submission · Next Checkpoir

Adding a New Toolbar

> *"People are not disturbed by things, but by the view they take of them."*
>
> *- Epictetus, Stoic philosopher, banished from Rome in 93 AD*

Introduction

We've now worked with a number of types of views. In addition to **`UIView`** itself, we've worked with these subclasses:

- **`UIButton`**
- **`UILabel`**

- **`UISlider`**
- **`UIWebView`**

Great news: in addition to the library of built-in **`UIView`** subclasses, it's possible to make your own!

In this checkpoint, we'll make a new, fancier toolbar by subclassing **`UIView`**.

To summarize what we know so far:

- A view controller mediates between its views and your model (your knowledge).
- A view accepts input from the user and passes it to a view controller, usually using a delegate or target-action pattern.

In an earlier checkpoint, we subclassed **`UIViewController`** to customize it. Over the next three checkpoints, we'll build our own **`UIView`** subclass, which will be a floating, movable, and colorful toolbar.

We'll start with the interface.

Creating a **`UIView`** subclass interface

When you're creating a new class, the best place to start is defining its interface. The interface provides an outline of the functionality you need to implement.

Create a new branch:

Terminal

```
$ git checkout -b awesome-colorful-toolbar
```

Create a new Objective-C class file. Call it **`BLCAwesomeFloatingToolbar`** and make sure it's a subclass of **`UIView`**.

Let's start by defining our interface:

BLCAwesomeFloatingToolbar.h

```
+ #import <UIKit/UIKit.h>
+
+ @class BLCAwesomeFloatingToolbar;
+
+ @protocol BLCAwesomeFloatingToolbarDelegate <NSObject>
+
+ @optional
+
+ - (void) floatingToolbar:(BLCAwesomeFloatingToolbar *)toolbar didSelectButtonWithTitle:(NSString *)title;
+
+ @end
+
+ @interface BLCAwesomeFloatingToolbar : UIView
+
+ - (instancetype) initWithFourTitles:(NSArray *)titles;
+
+ - (void) setEnabled:(BOOL)enabled forButtonWithTitle:(NSString *)title;
+
+ @property (nonatomic, weak) id <BLCAwesomeFloatingToolbarDelegate> delegate;
+
+ @end
```

This interface communicates four things to relevant classes:

1. It should be initialized with four titles using the custom initializer, **`initWithFourTitles:`**.
2. It implements a delegate protocol, so classes can optionally be informed when one of the titles is pressed.
3. It allows other classes to enable and disable its buttons.
4. It is a **`UIView`** subclass, which will let us add it to another view, set its **`frame`**, etc.

```
@class BLCAwesomeFloatingToolbar;
```

We're about to declare a delegate protocol, which references the class **`BLCAwesomeFloatingToolbar`**. However, **`BLCAwesomeFloatingToolbar`** hasn't been defined yet, because the protocol definition is *before* the **`@interface`**.

We include this line as a promise to the compiler that it will learn what a **`BLCAwesomeFloatingToolbar`** is later.

```
@protocol BLCAwesomeFloatingToolbarDelegate <NSObject>
```

This line indicates that the definition of **`BLCAwesomeFloatingToolbarDelegate`** is beginning. The **`<NSObject>`** at the end indicates that this protocol inherits from the **`NSObject`** protocol.

> *While not required, it's a best practice for your custom protocols to require conformity to the* **`NSObject`** *protocol. If you're curious about this, you can read more in* ***Programming with Objective-C: Working with Protocols****.*

```
@optional

- (void) floatingToolbar:(BLCAwesomeFloatingToolbar *)toolbar didSelectButtonWithTitle:(NSString *)title;
```

One optional delegate method is declared. If the delegate implements it, it will be called when a user taps a button.

```
@end
```

The definition of the delegate protocol has ended.

```
@interface BLCAwesomeFloatingToolbar : UIView

- (instancetype) initWithFourTitles:(NSArray *)titles;

- (void) setEnabled:(BOOL)enabled forButtonWithTitle:(NSString *)title;

@property (nonatomic, weak) id <BLCAwesomeFloatingToolbarDelegate> delegate;

@end
```

This is the interface for the toolbar itself, which declares:

- a custom initializer to use, which takes an array of four titles as an argument
- a method that enables or disables a button based on the **`title`** passed in
- a **`delegate`** property to use if a delegate is desired

Writing the Implementation

Our implementation will need to fulfill the promises made in our interface:

- save the four titles
- create four labels
- align them in a 2x2 grid
- if the delegate is set and it has implemented the optional **`floatingToolbar:didSelectButtonWithTitle:`** method, call it when the user taps any button
- respond to any requests to enable or disable buttons

of which label the user is currently touching:

BLCAwesomeFloatingToolbar.m

```
+ #import "BLCAwesomeFloatingToolbar.h"
+
+ @interface BLCAwesomeFloatingToolbar ()
+
+ @property (nonatomic, strong) NSArray *currentTitles;
+ @property (nonatomic, strong) NSArray *colors;
+ @property (nonatomic, strong) NSArray *labels;
+ @property (nonatomic, weak) UILabel *currentLabel;
+
+ @end
```

Next, we'll write our initializer. Its job is to create the four labels and assign both colors and text to each. Then, it needs to add all of them to the view.

In the **`@implementation`** section:

BLCAwesomeFloatingToolbar.m

```
+ - (instancetype) initWithFourTitles:(NSArray *)titles {
+     // First, call the superclass (UIView)'s initializer, to make sure we do all that setup first.
+     self = [super init];
+
+     if (self) {
+
+         // Save the titles, and set the 4 colors
+         self.currentTitles = titles;
+         self.colors = @[[UIColor colorWithRed:199/255.0 green:158/255.0 blue:203/255.0 alpha:1],
+                         [UIColor colorWithRed:255/255.0 green:105/255.0 blue:97/255.0 alpha:1],
+                         [UIColor colorWithRed:222/255.0 green:165/255.0 blue:164/255.0 alpha:1],
+                         [UIColor colorWithRed:255/255.0 green:179/255.0 blue:71/255.0 alpha:1]];
+
+         NSMutableArray *labelsArray = [[NSMutableArray alloc] init];
+
+         // Make the 4 labels
+         for (NSString *currentTitle in self.currentTitles) {
+             UILabel *label = [[UILabel alloc] init];
+             label.userInteractionEnabled = NO;
+             label.alpha = 0.25;
+
+             NSUInteger currentTitleIndex = [self.currentTitles indexOfObject:currentTitle]; // 0 through 3
+             NSString *titleForThisLabel = [self.currentTitles objectAtIndex:currentTitleIndex];
+             UIColor *colorForThisLabel = [self.colors objectAtIndex:currentTitleIndex];
+
+             label.textAlignment = NSTextAlignmentCenter;
+             label.font = [UIFont systemFontOfSize:10];
+             label.text = titleForThisLabel;
+             label.backgroundColor = colorForThisLabel;
+             label.textColor = [UIColor whiteColor];
+
+             [labelsArray addObject:label];
+         }
+
+         self.labels = labelsArray;
+
+         for (UILabel *thisLabel in self.labels) {
+             [self addSubview:thisLabel];
+         }
+     }
+
+     return self;
+ }
```

Most of this should be familiar, but there's a few new things:

- **`userInteractionEnabled`** is a property that indicates whether a **`UIView`** (or **`UIView`** subclass) receives touch events
- **`alpha`** represents a view's opacity between **`0`** (transparent) and **`1`** (opaque)
- a label's **`textAlignment`** property represents its horizontal text alignment
- **`UIFont`** represents a font and its associated information; **`systemFontOfSize:`** returns the default system font in a given size

Finally, we'll lay out the 4 **`UILabel`**s in a 2x2 grid:

BLCAwesomeFloatingToolbar.m

```
+ - (void) layoutSubviews {
+     // set the frames for the 4 labels
+
+     for (UILabel *thisLabel in self.labels) {
+         NSUInteger currentLabelIndex = [self.labels indexOfObject:thisLabel];
+
+         CGFloat labelHeight = CGRectGetHeight(self.bounds) / 2;
+         CGFloat labelWidth = CGRectGetWidth(self.bounds) / 2;
+         CGFloat labelX = 0;
+         CGFloat labelY = 0;
+
+         // adjust labelX and labelY for each label
+         if (currentLabelIndex < 2) {
+             // 0 or 1, so on top
+             labelY = 0;
+         } else {
+             // 2 or 3, so on bottom
+             labelY = CGRectGetHeight(self.bounds) / 2;
+         }
+
+         if (currentLabelIndex % 2 == 0) { // is currentLabelIndex evenly divisible by 2?
+             // 0 or 2, so on the left
+             labelX = 0;
+         } else {
+             // 1 or 3, so on the right
+             labelX = CGRectGetWidth(self.bounds) / 2;
+         }
+
+         thisLabel.frame = CGRectMake(labelX, labelY, labelWidth, labelHeight);
+     }
+ }
```

`layoutSubviews` will get called any time our view's **`frame`** is changed.

This code simply loops through our array of labels (**`self.labels`**) and sets the correct origin point and size.

The loop goes left-to-right, top-to-bottom:

```
| 0 | 1 |
| 2 | 3 |
```

Handling Touch Events

We used four **`UILabel`**s to build our toolbar. **`UILabel`** doesn't handle touches by default, so we'll need to write some code to handle that.

How Touches Get Routed

iOS passes various **events** to your app. The most common of these are "touch events", when the user touches the screen.

> *Other events include "accelerometer events", like when a user shakes a device, and "remote control events", if the user presses a button on a remote control.*

superview until it finds a view that *does* want to respond to it.

Here's how Apple explains it, in the **Event Handling Guide for iOS**:

> *When iOS recognizes an event, it passes the event to the initial object that seems most relevant for handling that event, such as the view where a touch occurred. If the initial object cannot handle the event, iOS continues to pass the event to objects with greater scope until it finds an object with enough context to handle the event. This sequence of objects is known as a responder chain, and as iOS passes events along the chain, it also transfers the responsibility of responding to the event. This design pattern makes event handling cooperative and dynamic.*

How to Handle Touch Events

In our toolbar, the user will tap one of the **UILabel**s. The label won't want to respond to the touch, so the event will get passed to our toolbar object. Since we want to respond to the touch event, we'll need to indicate this by implementing these four methods:

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event;
```

The methods will get called when touches begin, move, end, and are cancelled. A few things can cause a touch event to be cancelled: the user sliding their finger entirely off the screen, or the user getting a phone call while touching the screen.

We're seeing a few classes for the first time:

- **NSSet**, like **NSArray**, represents a collection of other objects. However, unlike an array, a set isn't ordered, and can't contain duplicate objects. -- In these methods, each **NSSet** (called **touches**) will contain one **UITouch** object.
- **UITouch** represents one finger currently touching the screen.
- **UIEvent** represents a touch event, a motion event, or a remote-control event - in our case, a touch event.

Implementing Touch Handling

First, to avoid repeating code in all of the touch-handling methods, let's add this method, which will figure out which of the labels was touched

BLCAwesomeFloatingToolbar.m

```
+ #pragma mark - Touch Handling
+
+ - (UILabel *) labelFromTouches:(NSSet *)touches withEvent:(UIEvent *)event {
+     UITouch *touch = [touches anyObject];
+     CGPoint location = [touch locationInView:self];
+     UIView *subview = [self hitTest:location withEvent:event];
+     return (UILabel *)subview;
+ }
```

This method:

- takes a touch from the touch set
- determines the touch's coordinates on the screen
- finds the **UIView** at that location (**hitTest:withEvent:** only finds views with **userInteractionEnabled == YES**; we'll enable some buttons later)
- returns the label to whoever requested it

> *Casting*
>
> *Note the* **(UILabel *)** *in the return line. This is called casting. We know that all of our subviews are* **UILabel***s, so we can safely*

> *Without the cast, the compiler would warn us: "Incompatible pointer types returning 'UIView ' from a function with result type 'UILabel '" -- Telling the compiler that we're sure it's a* **`UILabel`** *silences this warning.*

We're now ready to implement the touch methods outlined above.

When a touch begins, we'll dim the label to make it look highlighted. We'll also keep track of which label was most recently touched:

BLCAwesomeFloatingToolbar.m

```
+ - (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
+     UILabel *label = [self labelFromTouches:touches withEvent:event];
+
+     self.currentLabel = label;
+     self.currentLabel.alpha = 0.5;
+ }
```

When a touch moves, we'll check if the user is still touching the same label. If the user drags off the label, we'll reset the alpha. If they drag back on, we'll dim it again:

BLCAwesomeFloatingToolbar.m

```
+ - (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
+     UILabel *label = [self labelFromTouches:touches withEvent:event];
+
+     if (self.currentLabel != label) {
+         // The label being touched is no longer the initial label
+         self.currentLabel.alpha = 1;
+     } else {
+         // The label being touched is the initial label
+         self.currentLabel.alpha = 0.5;
+     }
+ }
```

Now that a touch has begun, it must eventually either be ended or cancelled.

If the user lifts their finger, the touch is *ended*. In this case, check if their finger was lifted from the same label they started with. If so, print a log to the console and inform the delegate. Either way, reset everything to how it was before anything was touched:

BLCAwesomeFloatingToolbar.m

```
+ - (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
+     UILabel *label = [self labelFromTouches:touches withEvent:event];
+
+     if (self.currentLabel == label) {
+         NSLog(@"Label tapped: %@", self.currentLabel.text);
+
+         if ([self.delegate respondsToSelector:@selector(floatingToolbar:didSelectButtonWithTitle:)]) {
+             [self.delegate floatingToolbar:self didSelectButtonWithTitle:self.currentLabel.text];
+         }
+     }
+
+     self.currentLabel.alpha = 1;
+     self.currentLabel = nil;
+ }
```

On calling the delegate: first, we check if the delegate has implemented the method. This is **always required** for *optional* protocol methods. I you try to call a method without checking first - and it turns out the delegate doesn't implement it - the app will crash.

In space, no one can hear you scream. Similarly, in Objective-C, no one can hear your method invocations on a **`nil`** object. If no delegate has been set, **`self.delegate`** is **`nil`**, so nothing will happen. This is notable because in many other languages, similar behavior could cause a crash.

```
+ - (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
+     self.currentLabel.alpha = 1;
+     self.currentLabel = nil;
+ }
```

This wraps up our touch handling.

We only have two things left to do:

- implement **`setEnabled:forButtonWithTitle:`**
- use the toolbar in our view controller

Button enabling

Handling button enabling is simple:

BLCAwesomeFloatingToolbar.m

```
+ #pragma mark - Button Enabling
+
+ - (void) setEnabled:(BOOL)enabled forButtonWithTitle:(NSString *)title {
+     NSUInteger index = [self.currentTitles indexOfObject:title];
+
+     if (index != NSNotFound) {
+         UILabel *label = [self.labels objectAtIndex:index];
+         label.userInteractionEnabled = enabled;
+         label.alpha = enabled ? 1.0 : 0.25;
+     }
+ }
```

We try to find the index of the button using **`title`**. If we find an index, then we use it to get the corresponding **`UILabel`**, and set its **`userInteractionEnabled`** and **`alpha`** properties depending on the value for **`enabled`** that was passed in.

Adding the Toolbar to the View Controller

Our new **`UIView`** subclass is complete! Now all we need to do is tell our view controller about it.

First, **`#import`** it:

BLCWebBrowserViewController.m

```
+ #import "BLCAwesomeFloatingToolbar.h"
```

Before we get into the **`@interface`**, let's also add some **`#define`** directives:

BLCWebBrowserViewController.m

```
+ #define kBLCWebBrowserBackString NSLocalizedString(@"Back", @"Back command")
+ #define kBLCWebBrowserForwardString NSLocalizedString(@"Forward", @"Forward command")
+ #define kBLCWebBrowserStopString NSLocalizedString(@"Stop", @"Stop command")
+ #define kBLCWebBrowserRefreshString NSLocalizedString(@"Refresh", @"Reload command")
```

Our code might need these strings in a few places. **`#define`** lets us make up a word that is replaced with whatever follows it. After we add this we can now use **`kBLCWebBrowserStopString`** in our code, and it will automatically get replaced with **`NSLocalizedString(@"Stop", @"Stop command")`**.

In our interface, we need to declare that we conform to the new protocol we created:

```
- @interface BLCWebBrowserViewController () <UIWebViewDelegate, UITextFieldDelegate>
+ @interface BLCWebBrowserViewController () <UIWebViewDelegate, UITextFieldDelegate, BLCAwesomeFloatingToolbarDelegate>
```

Now on to our properties. We can delete all four button properties and add one property for our toolbar:

BLCWebBrowserViewController.m

```
@property (nonatomic, strong) UIWebView *webview;
@property (nonatomic, strong) UITextField *textField;
- @property (nonatomic, strong) UIButton *backButton;
- @property (nonatomic, strong) UIButton *forwardButton;
- @property (nonatomic, strong) UIButton *stopButton;
- @property (nonatomic, strong) UIButton *reloadButton;
@property (nonatomic, strong) UIActivityIndicatorView *activityIndicator;
+ @property (nonatomic, strong) BLCAwesomeFloatingToolbar *awesomeToolbar;
@property (nonatomic, assign) NSUInteger frameCount;
```

In **`loadView`**, we need to delete the references to the old buttons, and add our new toolbar.

When you delete the old button properties, you'll get a bunch of errors. Don't fret: they'll get fixed as we go through the code.

BLCWebBrowserViewController.m

```
-     self.backButton = [UIButton buttonWithType:UIButtonTypeSystem];
-     [self.backButton setEnabled:NO];
-
-     self.forwardButton = [UIButton buttonWithType:UIButtonTypeSystem];
-     [self.forwardButton setEnabled:NO];
-
-     self.stopButton = [UIButton buttonWithType:UIButtonTypeSystem];
-     [self.stopButton setEnabled:NO];
-
-     self.reloadButton = [UIButton buttonWithType:UIButtonTypeSystem];
-     [self.reloadButton setEnabled:NO];
+     self.awesomeToolbar = [[BLCAwesomeFloatingToolbar alloc] initWithFourTitles:@[kBLCWebBrowserBackString, kBLCWebBrowserForwardSt
+     self.awesomeToolbar.delegate = self;

-     [self.backButton setTitle:NSLocalizedString(@"Back", @"Back comnmand") forState:UIControlStateNormal];
-     [self.forwardButton setTitle:NSLocalizedString(@"Forward", @"Forward comnmand") forState:UIControlStateNormal];
-     [self.stopButton setTitle:NSLocalizedString(@"Stop", @"Stop comnmand") forState:UIControlStateNormal];
-     [self.reloadButton setTitle:NSLocalizedString(@"Refresh", @"Reload comnmand") forState:UIControlStateNormal];
-
-     [self addButtonTargets];
```

(We don't need to call **`addButtonTargets`** any more, since our new delegate method will take care of handling button taps for us.)

We also need to replace the buttons with our toolbar in our array of views to add:

BLCWebBrowserViewController.m

```
-     for (UIView *viewToAdd in @[self.webview, self.textField, self.backButton, self.forwardButton, self.stopButton, self.reloadButt
+     for (UIView *viewToAdd in @[self.webview, self.textField, self.awesomeToolbar]) {
```

In **`viewWillLayoutSubviews`** we no longer need to calculate where each button goes, but we do need to place our toolbar:

BLCWebBrowserViewController.m

```
+     CGFloat browserHeight = CGRectGetHeight(self.view.bounds) - itemHeight;
-     CGFloat buttonWidth = CGRectGetWidth(self.view.bounds) / 4;

    // Now, assign the frames
    self.textField.frame = CGRectMake(0, 0, width, itemHeight);
    self.webview.frame = CGRectMake(0, CGRectGetMaxY(self.textField.frame), width, browserHeight);

-     CGFloat currentButtonX = 0;
-
-     for (UIButton *thisButton in @[self.backButton, self.forwardButton, self.stopButton, self.reloadButton]) {
-         thisButton.frame = CGRectMake(currentButtonX, CGRectGetMaxY(self.webview.frame), buttonWidth, itemHeight);
-         currentButtonX += buttonWidth;
-     }
+     self.awesomeToolbar.frame = CGRectMake(20, 100, 280, 60);
```

Now, let's respond to our delegate method:

BLCWebBrowserViewController.m

```
+ #pragma mark - BLCAwesomeFloatingToolbarDelegate
+
+ - (void) floatingToolbar:(BLCAwesomeFloatingToolbar *)toolbar didSelectButtonWithTitle:(NSString *)title {
+     if ([title isEqual:NSLocalizedString(@"Back", @"Back command")]) {
+         [self.webview goBack];
+     } else if ([title isEqual:NSLocalizedString(@"Forward", @"Forward command")]) {
+         [self.webview goForward];
+     } else if ([title isEqual:NSLocalizedString(@"Stop", @"Stop command")]) {
+         [self.webview stopLoading];
+     } else if ([title isEqual:NSLocalizedString(@"Refresh", @"Reload command")]) {
+         [self.webview reload];
+     }
+ }
```

This simply checks **`title`**, and if we recognize it, calls the appropriate **`UIWebView`** method.

We need to make sure the button's enabled state is updated properly in **`updateButtonsAndTitle`**:

BLCWebBrowserViewController.m

```
-     self.backButton.enabled = [self.webview canGoBack];
-     self.forwardButton.enabled = [self.webview canGoForward];
-     self.stopButton.enabled = self.frameCount > 0;
-     self.reloadButton.enabled = self.webview.request.URL && self.frameCount == 0;
+     [self.awesomeToolbar setEnabled:[self.webview canGoBack] forButtonWithTitle:kBLCWebBrowserBackString];
+     [self.awesomeToolbar setEnabled:[self.webview canGoForward] forButtonWithTitle:kBLCWebBrowserForwardString];
+     [self.awesomeToolbar setEnabled:self.frameCount > 0 forButtonWithTitle:kBLCWebBrowserStopString];
+     [self.awesomeToolbar setEnabled:self.webview.request.URL && self.frameCount == 0 forButtonWithTitle:kBLCWebBrowserRefreshString
```

Since we don't need to add the button targets any more, let's remove the call to **`addButtonTargets`** from the **`resetWebView`** method:

BLCWebBrowserViewController.m

```
-     [self addButtonTargets];
```

And since the method isn't getting called any more, let's just delete the whole thing.

BLCWebBrowserViewController.m

```
-     for (UIButton *button in @[self.backButton, self.forwardButton, self.stopButton, self.reloadButton]) {
-         [button removeTarget:nil action:NULL forControlEvents:UIControlEventTouchUpInside];
-     }
-
-     [self.backButton addTarget:self.webview action:@selector(goBack) forControlEvents:UIControlEventTouchUpInside];
-     [self.forwardButton addTarget:self.webview action:@selector(goForward) forControlEvents:UIControlEventTouchUpInside];
-     [self.stopButton addTarget:self.webview action:@selector(stopLoading) forControlEvents:UIControlEventTouchUpInside];
-     [self.reloadButton addTarget:self.webview action:@selector(reload) forControlEvents:UIControlEventTouchUpInside];
- }
```

Don't worry about deleting code - since we're tracking our changes in Git, we can always go back and restore the code if needed.

We're all done! Run the app. You should have a beautiful, functional, colorful toolbar.

Assignment

Our **`floatingToolbar:didSelectButtonWithTitle:`** method isn't using the constants we defined (using **`#define`**) at the beginning. Update the code to use those.

Experiment with setting the **`frame`** of the toolbar to some different values. Try them out in both landscape and portrait.

When you're satisfied with your project and the assignment, commit, merge and push your code to GitHub:

Terminal

```
$ git add .
$ git commit -m 'Add a beautiful, functional, colorful toolbar'
$ git checkout master
$ git merge awesome-colorful-toolbar
$ git push
```

Submit this checkpoint's assignment with links for your repo and commit.

Our **`floatingToolbar:didSelectButtonWithTitle:`** method isn't using the constants we defined (using **`#define`**) at the beginning. Update the code to use those.

Experiment with setting the **`frame`** of the toolbar to some different values. Try them out in both landscape and portrait.

When you're satisfied with your project and the assignment, commit, merge and push your code to Github:

Terminal

```
$ git add .
$ git commit -m 'Add a beautiful, functional, colorful toolbar'
$ git checkout master
$ git merge awesome-colorful-toolbar
$ git push
```

Submit this checkpoint's assignment with links for your repo and commit.

COURSES

- Full Stack Web Development
- Frontend Web Development
- UX Design
- Android Development
- iOS Development

ABOUT BLOC

- Our Team | Jobs
- Bloc Veterans Program
- Employer Sponsored
- FAQ
- Blog
- Engineering Blog
- Refer-a-Friend
- Privacy Policy
- Terms of Service

MADE BY BLOC

- Tech Talks & Resources
- Programming Bootcamp Comparison
- Jottly: A Beginner's Guide to HTML, CSS, Skeleton and Animate.css
- Swiftris: Build Your First iOS Game with Swift
- Webflow Tutorial: Design Responsive Sites with Webflow
- Ruby Warrior
- Bloc's Diversity Scholarship

SIGN UP FOR OUR MAILING LIST

Send

- hello@bloc.io
- Considering enrolling? (404) 480-2562
- Partnership / Corporate Inquiries? (650) 741-5682

Roadmap · Previous Checkpoint
Jump to Submission · Next Checkpoir

Inappropriate Gestures

> *"When I disapprove of something, [spitting in their face is] the only thing I can think of to do. It's a rather rude gesture, but at least it's clear what you mean."*
>
> *- Katharine Hepburn, American actress*

Our goal in this checkpoint is to allow the user to drag the toolbar around the screen.

As you can imagine, handling several types of events within **`touchesBegan:…`**, **`touchesMoved:…`**, etc. would become tedious and difficult.

Fortunately, Apple makes common gesture handling even easier than how you learned in the previous checkpoint, with **Gesture Recognizers**
While Apple has yet to announce the **`UISpitInFaceGestureRecognizer`**, they do provide these helpful gesture recognizers:

Gesture Recognizer	Purpose
`UITapGestureRecognizer`	Detects any number of taps
`UIPinchGestureRecognizer`	Detects pinching in and out (usually for zooming)
`UIPanGestureRecognizer`	Detects a pan or drag gesture
`UISwipeGestureRecognizer`	Detects when the user swipes upwards, downwards, left, or right
`UIRotationGestureRecognizer`	Detects two touches moving in a circular motion
`UILongPressGestureRecognizer`	Detects a long press (press-and-hold)

Tap Gesture Recognizers

`UITapGestureRecognizer` will be used to replace all of our tap-detecting code. The tap recognizer will detect whether or not a tap occurred. A tap can be described as a finger touching down then lifting shortly after from a location in proximity to where it began.

Create a new branch:

Terminal

```
$ git checkout -b gesture-recognizers
```

Delete all of the **`touches...`** methods. We no longer need them:

BLCAwesomeFloatingToolbar.m

```
-     UILabel *label = [self labelFromTouches:touches withEvent:event];

-     self.currentLabel = label;
-     self.currentLabel.alpha = 0.5;
- }

- - (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
-     UILabel *label = [self labelFromTouches:touches withEvent:event];

-     if (self.currentLabel != label) {
-         self.currentLabel.alpha = 1;
-     } else {
-         self.currentLabel.alpha = 0.5;
-     }
- }

- - (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
-     UILabel *label = [self labelFromTouches:touches withEvent:event];

-     if (self.currentLabel == label) {
-         NSLog(@"Label tapped: %@", self.currentLabel.text);

-         if ([self.delegate respondsToSelector:@selector(floatingToolbar:didSelectButtonWithTitle:)]) {
-             [self.delegate floatingToolbar:self didSelectButtonWithTitle:self.currentLabel.text];
-         }
-     }

-     self.currentLabel.alpha = 1;
-     self.currentLabel = nil;
- }

- - (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
-     self.currentLabel.alpha = 1;
-     self.currentLabel = nil;
- }
```

Let's add a property to store the recognizer:

BLCAwesomeFloatingToolbar.m

```
+ @property (nonatomic, strong) UITapGestureRecognizer *tapGesture;
```

Create and initialize the recognizer at the end of **`initWithFourTitles:`**:

BLCAwesomeFloatingToolbar.m

```
    // First, call the superclass (UIView)'s initializer, to make sure we do all that setup first.
    self = [super init];
    if (self) {
        // Save the titles, and set the 4 colors
        self.currentTitles = titles;
        self.colors = @[[UIColor colorWithRed:199/255.0 green:158/255.0 blue:203/255.0 alpha:1],
                        [UIColor colorWithRed:255/255.0 green:105/255.0 blue:97/255.0 alpha:1],
                        [UIColor colorWithRed:222/255.0 green:165/255.0 blue:164/255.0 alpha:1],
                        [UIColor colorWithRed:255/255.0 green:179/255.0 blue:71/255.0 alpha:1]];

        NSMutableArray *labelsArray = [[NSMutableArray alloc] init];

        // Make the 4 labels
        for (NSString *currentTitle in self.currentTitles) {
            UILabel *label = [[UILabel alloc] init];
            label.userInteractionEnabled = NO;
            label.alpha = 0.25;

            NSUInteger currentTitleIndex = [self.currentTitles indexOfObject:currentTitle]; // 0 through 3
            NSString *titleForThisLabel = [self.currentTitles objectAtIndex:currentTitleIndex];
            UIColor *colorForThisLabel = [self.colors objectAtIndex:currentTitleIndex];

            label.textAlignment = NSTextAlignmentCenter;
            label.font = [UIFont systemFontOfSize:10];
            label.text = titleForThisLabel;
            label.backgroundColor = colorForThisLabel;
            label.textColor = [UIColor whiteColor];

            [labelsArray addObject:label];
        }

        self.labels = labelsArray;

        for (UILabel *thisLabel in self.labels) {
            [self addSubview:thisLabel];
        }
+       self.tapGesture = [[UITapGestureRecognizer alloc] initWithTarget:self action:@selector(tapFired:)];
+       [self addGestureRecognizer:self.tapGesture];
    }
    return self;
}
```

Our initialization tells the gesture recognizer which view to detect the tap for - **`self`** or that instance of **`BLCAwesomeFloatingToolbar`** - and which method to call when a tap is detected (**`tapFired:`**). Now, all we need to do is implement the **`tapFired:`** method:

BLCAwesomeFloatingToolbar.m

```
+ - (void) tapFired:(UITapGestureRecognizer *)recognizer {
+     if (recognizer.state == UIGestureRecognizerStateRecognized) {
+         CGPoint location = [recognizer locationInView:self];
+         UIView *tappedView = [self hitTest:location withEvent:nil];

+         if ([self.labels containsObject:tappedView]) {
+             if ([self.delegate respondsToSelector:@selector(floatingToolbar:didSelectButtonWithTitle:)]) {
+                 [self.delegate floatingToolbar:self didSelectButtonWithTitle:((UILabel *)tappedView).text];
+             }
+         }
+     }
+ }
```

Let's discuss what's happening in **`tapFired:`**. The first thing we do is check for the proper **`state`**. A gesture recognizer has several states it car be in, and **`UIGestureRecognizerStateRecognized`** is the state in which the type of gesture it recognizes has been detected. In our case, a tap has been completed and the recognizer's state was switched to **`UIGestureRecognizerStateRecognized`**. If the gesture recognizer is in any other state, the gesture hasn't been detected. You can read about other possible states in the **Apple documentation.**

The next line - **`CGPoint location = [recognizer locationInView:self];`** - gives us an x-y coordinate of where the gesture occurred with respect to our bounds. A tap detected in the top-left corner of the toolbar will register as **`(0,0)`**. Then we invoke **`hitTest:withEvent:`** to

toolbar labels and if so, we verify our delegate for compatibility before performing the appropriate method call.

Next, let's allow the toolbar to be moved around.

Pan Gesture Recognizers

A Pan Gesture Recognizer recognizes panning or dragging motion, like when you move around in the Maps application.

Begin by adding the gesture recognizer property:

BLCAwesomeFloatingToolbar.m

```
+ @property (nonatomic, strong) UIPanGestureRecognizer *panGesture;
```

Then create and initialize **panGesture** immediately after **tapGesture**:

BLCAwesomeFloatingToolbar.m

```
        self.tapGesture = [[UITapGestureRecognizer alloc] initWithTarget:self action:@selector(tapFired:)];
        [self addGestureRecognizer:self.tapGesture];
+         self.panGesture = [[UIPanGestureRecognizer alloc] initWithTarget:self action:@selector(panFired:)];
+         [self addGestureRecognizer:self.panGesture];
```

As a subview, **BLCAwesomeFloatingToolbar** shouldn't be trusted to move itself around in its superview. That would be a bad design practice. Changes made by objects should only affect themselves and objects they own. If the toolbar moved itself around, it might collide with other objects it doesn't know about.

Instead, let's add a new delegate method to our protocol which will indicate that **BLCAwesomeFloatingToolbar** wishes to be moved and the direction it wishes to be moved in. Its delegate (the view controller) can decide whether to actually move the toolbar or not.

BLCAwesomeFloatingToolbar.h

```
@protocol BLCAwesomeFloatingToolbarDelegate <NSObject>
@optional
- (void) floatingToolbar:(BLCAwesomeFloatingToolbar *)toolbar didSelectButtonWithTitle:(NSString *)title;
+ - (void) floatingToolbar:(BLCAwesomeFloatingToolbar *)toolbar didTryToPanWithOffset:(CGPoint)offset;
@end
```

Now we need to implement **panFired**; the method our gesture recognizer will invoke when a pan has been detected:

BLCAwesomeFloatingToolbar.m

```
+ - (void) panFired:(UIPanGestureRecognizer *)recognizer {
+     if (recognizer.state == UIGestureRecognizerStateChanged) {
+         CGPoint translation = [recognizer translationInView:self];

+         NSLog(@"New translation: %@", NSStringFromCGPoint(translation));

+         if ([self.delegate respondsToSelector:@selector(floatingToolbar:didTryToPanWithOffset:)]) {
+             [self.delegate floatingToolbar:self didTryToPanWithOffset:translation];
+         }

+         [recognizer setTranslation:CGPointZero inView:self];
+     }
+ }
```

As you can see, this gesture handler is very similar to the one we wrote earlier. The primary difference being that we no longer care *where* the gesture occurred. What's important now is which direction it travelled in. A pan gesture recognizer's **translation** is how far the user's finger has moved in each direction since the touch event began. This method is called often during a pan gesture because a "full" pan as we perceive it is actually a linear collection of small pans traveling a few pixels at a time.

is called.

Finally, we need to implement **floatingToolbar:didTryToPanWithOffset** in **BLCWebBrowserViewController** because that's where the **BLCAwesomeFloatingToolbar** will actually be assigned a new frame:

BLCWebBrowserViewController.m

```
+ - (void) floatingToolbar:(BLCAwesomeFloatingToolbar *)toolbar didTryToPanWithOffset:(CGPoint)offset {
+     CGPoint startingPoint = toolbar.frame.origin;
+     CGPoint newPoint = CGPointMake(startingPoint.x + offset.x, startingPoint.y + offset.y);

+     CGRect potentialNewFrame = CGRectMake(newPoint.x, newPoint.y, CGRectGetWidth(toolbar.frame), CGRectGetHeight(toolbar.frame));

+     if (CGRectContainsRect(self.view.bounds, potentialNewFrame)) {
+         toolbar.frame = potentialNewFrame;
+     }
+ }
```

We begin by getting the top-left corner of where the toolbar is currently located. **newPoint** is where the *future* top-left corner is stored by adding the difference in **x** and the difference in **y** to the original top-left coordinate. Then we create a new **CGRect** which represents the toolbars potential new frame.

We say *potential* because we want to make sure that we don't push the toolbar off the screen. **CGRectContainsRect(CGRect rect1, CGRect rect2)** will return **YES** if the **rect2**'s bounds are contained entirely by **rect1**, or **NO** otherwise. If the test passes, we set the toolbar's new fram and it's done.

Run the app. You should now be able to move the toolbar around the screen.

We've built a functioning mobile browser with privacy control and a cool toolbar. Our next project will be the biggest, and most fun yet: Blocstagram!

Roadmap · **Previous Checkpoint** **Jump to Submission** · **Next Checkpoir**

Your assignment | **Ask a question** | **Submit your work**

- Add a pinch gesture recognizer to allow the user to resize the toolbar.
- Add a long press gesture recognizer that rotates the background colors when fired.
- Because the **UITapGestureRecognizer** doesn't have *started* and *ended* states, the labels no longer dim and light up when you tap on them. Remove the labels and tap gesture recognizers, and replace the labels with **UIButton**s to resolve this.

When you're satisfied with your project and the assignment, commit, merge and push your code to GitHub:

Terminal

```
$ git add .
$ git commit -m 'Converted touch handlers to gesture recognizers'
$ git checkout master
$ git merge gesture-recognizers
$ git push
```

Submit this checkpoint's assignment with links for your repo and commit.

assignment completed

COURSES

- Full Stack Web Development
- Frontend Web Development
- UX Design
- Android Development
- iOS Development

ABOUT BLOC

- Our Team | Jobs
- Bloc Veterans Program
- Employer Sponsored
- FAQ
- Blog
- Engineering Blog
- Refer-a-Friend
- Privacy Policy
- Terms of Service

MADE BY BLOC

- Tech Talks & Resources
- Programming Bootcamp Comparison
- Jottly: A Beginner's Guide to HTML, CSS, Skeleton and Animate.css
- Swiftris: Build Your First iOS Game with Swift
- Webflow Tutorial: Design Responsive Sites with Webflow
- Ruby Warrior
- Bloc's Diversity Scholarship

SIGN UP FOR OUR MAILING LIST

Send

- hello@bloc.io
- Considering enrolling? (404) 480-2562
- Partnership / Corporate Inquiries? (650) 741-5682

Roadmap · Previous Checkpoint
Jump to Submission · Next Checkpoir

Building Blocstagram

> *"A table, a chair, a bowl of fruit and a violin; what else does a man need to be happy?"*
>
> *- Albert Einstein, theoretical physicist*

Displaying an Image Feed

We've used classes like **`UILabel`**, **`UIImageView`**, and **`UIAlertView`** to display small amounts of information to users. When there's a lot to share though, these classes just don't cut it.

basic app that lets us scroll through a list of images.

In order to accomplish this, we'll learn about three new classes:

- **`UITableView`**: a table view displays a scrolling list of views in a single column. It allows the items to be in a continuous list, or grouped into sections. It also provides some tools for editing the displayed data.
- **`UITableViewCell`**: a table cell controls the appearance and behavior of an item in the table view.
- **`UITableViewController`**: because **`UITableView`** has many delegate methods and configuration options, Apple provided this **`UIViewController`** subclass to implement common table behavior.

In addition to those classes, there are two classes - **`UIImage`** and **`UIImageView`** - that you haven't seen before. Their relationship is very similar to **`UILabel`** and **`NSString`**'s relationship, so they're not covered in detail here. If you'd like more information, check out **`UIImage` and `UIImageView`**.

Before we start, you'll need to obtain 10 placeholder photos to display. It's more fun to use your own, but if you don't have any, **you can use ours**. Name each of your photos numerically from **`1.jpg`** up to **`10.jpg`**, and put them aside for now.

Creating a Table View Controller

Begin by creating a new **Empty Application** project. Use the following specifics:

- **Product Name**: Blocstagram
- **Class Prefix**: BLC
- **Devices**: Universal

Make sure that you check **Create git repository**, and then click **Create**.

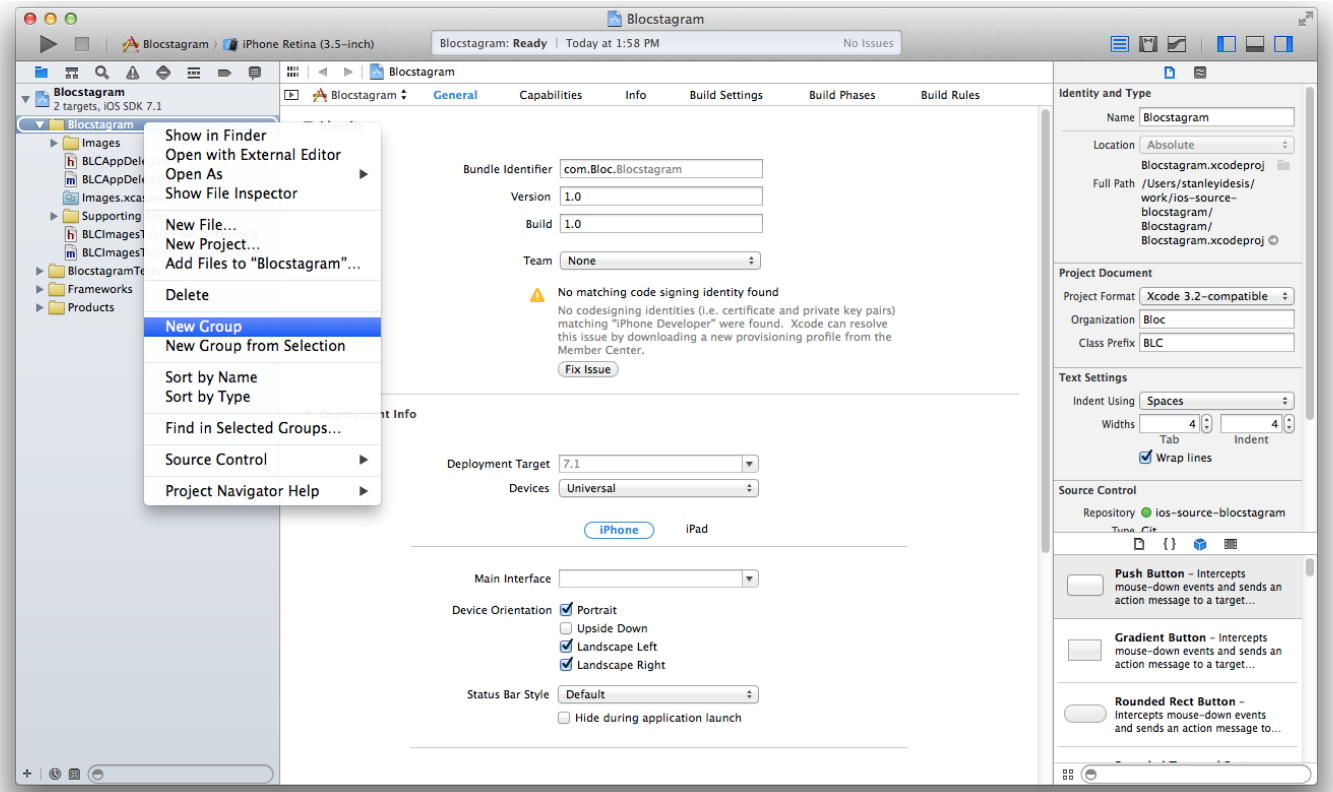
Open your terminal, **`cd`** into the Blocstagram directory, and commit your code to the **`master`** branch. Next, we'll check out a new branch to work in for this checkpoint:

Terminal

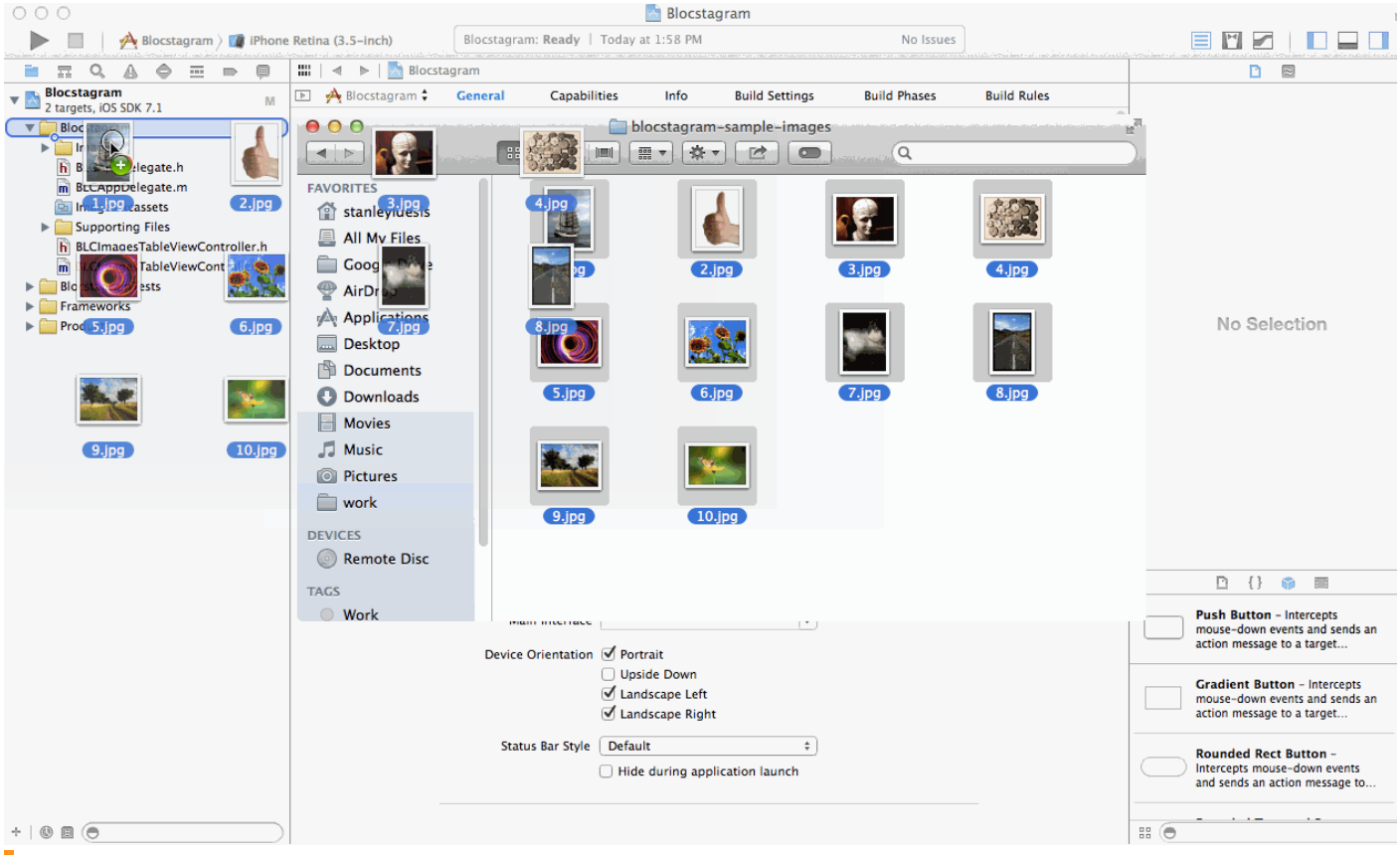
```
$ pwd #=> should be in the Blocstagram directory. If not, then cd into it
$ git status #=> check to make sure you see the new project files we just created in Xcode
$ git add .
$ git commit -m "Initial Blocstagram commit"
$ git checkout -b building-blocstagram
```

Adding Images to Xcode

We'll put our images into a project folder unremarkably titled, "Images." Xcode refers to these folders as **groups**. In Xcode, right–click the "Blocstagram" group from within Project Navigator and choose "New Group":



Label the group, "Images" and press return. Now, from wherever your images are located, highlight them all and drag them into the "Images" group like so:



> *When the dialog appears, make sure you have the "Copy items into destination group's folder (if needed)" box* ***checked****. This embeds the images in your project, so it will still work if you open the project on another computer*

Your images are now part of your project. Awesome!

Table View Controller Subclass

Create a new Objective-C class and name it **`BLCImagesTableViewController`**. Make sure that it is a subclass of **`UITableViewController`**. Onc the implementation file opens, let's take a quick detour to ensure this view controller is used. Open **`BLCAppDelegate.m`** and set up the root view controller.

BLCAppDelegate.m

```
 #import "BLCAppDelegate.h"
+ #import "BLCImagesTableViewController.h"

 @implementation BLCAppDelegate

 - (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
 {
     self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
+      self.window.rootViewController = [[UINavigationController alloc] initWithRootViewController:[[BLCImagesTableViewController allo
     // Override point for customization after application launch.
     self.window.backgroundColor = [UIColor whiteColor];
     [self.window makeKeyAndVisible];
     return YES;
 }
```

Now switch back to **`BLCImagesTableViewController.m`**. You may notice the following two methods:

BLCImagesTableViewController.m

```
#pragma mark - Table view data source

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
#warning Potentially incomplete method implementation.
    // Return the number of sections.
    return 0;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
#warning Incomplete method implementation.
    // Return the number of rows in the section.
    return 0;
}
```

Xcode automatically created these methods and attached **`#warning`** markers to them. The markers suggest that despite the fact that the methods have been generated for us, we must implement them properly for our table view controller to function as we expect. We may have to change **`numberOfSectionsInTableView:`** and we *certainly* have to alter **`tableView:numberOfRowsInSection:`**

An Array of Images

Nearly every table you've seen in an iOS application is backed by an array of data. Our data are going to be the 10 images we've prepared to display as placeholders for actual Instagram images. We'll store references to these images inside of an **`NSArray`** of **`UIImage`** objects:

BLCImagesTableViewController.h

```
 @interface BLCImagesTableViewController ()
+ @property (nonatomic, strong) NSMutableArray *images;
 @end
```

BLCImagesTableViewController.m

```
{
    self = [super initWithStyle:style];
    if (self) {
        // Custom initialization
+         self.images = [NSMutableArray array];

    }
    return self;
}
```

After initializing our array, let's populate it with **viewDidLoad**:

BLCImagesTableViewController.m

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

+     for (int i = 1; i <= 10; i++) {
+         NSString *imageName = [NSString stringWithFormat:@"%d.jpg", i];
+         UIImage *image = [UIImage imageNamed:imageName];
+         if (image) {
+             [self.images addObject:image];
+         }
+     }
}
```

UITableViewCell Class

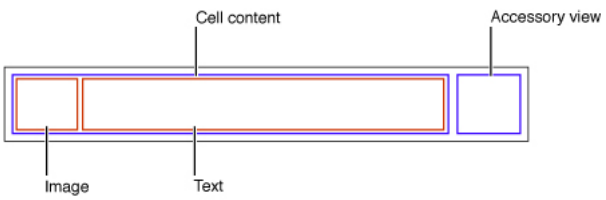
As we stated earlier, an instance of **UITableViewCell** represents a row that appears in a table view. Our table view is responsible for creating or reusing copies of the table view cells we register with it. We register a **UITableViewCell** using **registerClass:forCellReuseIdentifier**.

We may register as many types of table view cells as we like with our table, but we need at least one. For the purposes of Blocstagram we'll use the generic **UITableViewCell** class:

BLCImagesTableViewController.m

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

    for (int i = 1; i <= 10; i++) {
        NSString *imageName = [NSString stringWithFormat:@"%d.jpg", i];
        UIImage *image = [UIImage imageNamed:imageName];
        if (image) {
            [self.images addObject:image];
        }
    }

+     [self.tableView registerClass:[UITableViewCell class] forCellReuseIdentifier:@"imageCell"];
}
```

UITableViewCell is a subclass of **UIView** which provides behaviors and properties that help it represent an item in a list. The base **UITableViewCell** view is laid out like so:



The content, image and accessory views are all customizable during the **tableView:cellForRowAtIndexPath:** method call, which you'll learn more about shortly. The default cell is extendable and easy to modify, however, it's perfectly acceptable to subclass **UITableViewCell** to mak

For example, the table found inside of a social network's application - whose name rhymes with *SpaceHook* - might have a custom table cell fc text posts. One cell could hold images and another could hold obnoxious mobile advertisements. These cell types are likely represented by separate **`UITableViewCell`** subclasses.

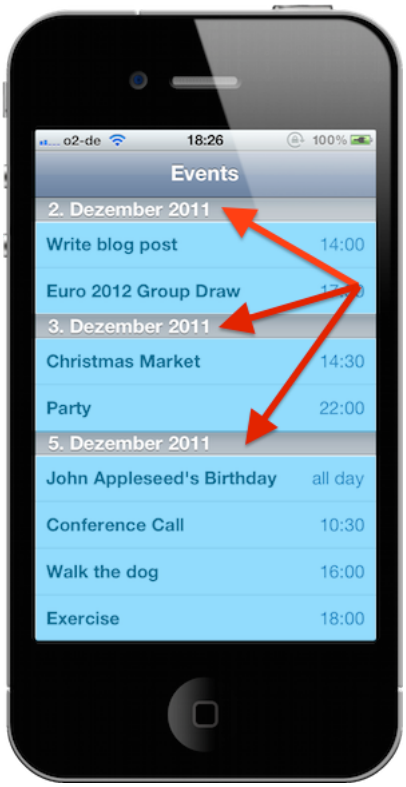
Responding to **`UITableViewDelegate`** and **`UITableViewDataSource`** methods

`UITableView`, like many classes we've seen before, offers a delegate protocol. However, **`UITableView`** requires an object to implement its *data source* protocol, **`UITableViewDataSource`**. As you've seen earlier, a delegate receives feedback from the object when a user performs an action.

Contrarily, a data source provides information *to the object*. In the case of **`UITableView`**, it needs to know things such as, "how many cells am I presenting?", "which cell goes at this spot?", etc. Some of these methods have been partially pre-written in our **`BLCImagesTableViewController`** class. Let's modify them to get the ball rolling.

Number of Rows and Sections

The first method is **`numberOfSectionsInTableView:`**. Sections are the partitioned groupings you occasionally see in a table view. Here's an example of some:



Each blue area represents a section of cells and the red arrows are pointing to the section headers. For this checkpoint and most tables, we'll only require one section. In this case, we can delete the **`numberOfSectionsInTableView:`** because **`UITableView`** will default to **`1`**:

BLCImagesTableViewController.m

```
- - (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
- {
- #warning Potentially incomplete method implementation.
-     // Return the number of sections.
-     return 0;
- }
```

The next method we need to consider is **`tableView:numberOfRowsInSection:`**. It's a required method. As its name implies, this data source method asks us how many rows belong to a given section. Since we only have one section we may disregard the **`section`** parameter entirely:

BLCImagesTableViewController.m

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
 - #warning Incomplete method implementation.
 -     // Return the number of rows in the section.
 -     return 0;
 +     return self.images.count;
}
```

It's a best practice to reference the size of our backing array directly. Unless you have a very good reason, do not hard-code the return value. In our case, it would be valid to replace the return statement with **`return 10;`**. However, that's acceptable only because we know the number of images will never change during the course of this checkpoint. In future checkpoints and perhaps all projects you work on, this will not likely be the case.

Creating Table View Cells

The only remaining required method is the most important: **`tableView:cellForRowAtIndexPath:`**. This is the method which returns a prepped and ready **`UITableViewCell`** instance to the table view to display at a given location. This method is called *every time* a new row is about to appear on screen whether scrolling up or down:

BLCImagesTableViewController.m

```
 - /*
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
 -     UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:<#@"reuseIdentifier"#> forIndexPath:indexPath];
 +     UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"imageCell" forIndexPath:indexPath];

    // Configure the cell...
 +     static NSInteger imageViewTag = 1234;
 +     UIImageView *imageView = (UIImageView*)[cell.contentView viewWithTag:imageViewTag];

 +     if (!imageView) {
 +         // This is a new cell, it doesn't have an image view yet
 +         imageView = [[UIImageView alloc] init];
 +         imageView.contentMode = UIViewContentModeScaleToFill;

 +         imageView.frame = cell.contentView.bounds;
 +         imageView.autoresizingMask = UIViewAutoresizingFlexibleHeight | UIViewAutoresizingFlexibleWidth;

 +         imageView.tag = imageViewTag;
 +         [cell.contentView addSubview:imageView];
 +     }

 +     UIImage *image = self.images[indexPath.row];
 +     imageView.image = image;

    return cell;
}
 - */
```

A lot is going on here, let's break it down piece by piece beginning with

```
UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"imageCell" forIndexPath:indexPath];
```

`dequeueReusableCellWithIdentifier:forIndexPath:` takes the identifier string we provide it and compares it with its roster of registered table view cells. Remember, we registered the **`UITableViewCell`** class in **`viewDidLoad`** with the identifier, **`imageCell`**.

Dequeue will either return:

- a *brand new* cell of the type we registered, or
- a *used* one that is no longer visible on screen

As we scroll, these recycled cells will leap-frog each other:

Since our current cell may be *used*, we have to make sure to update all of its information. Forgetting to do so will result in showing the user a cell populated with data from an earlier row. That'd be weird.

Let's briefly cover the **NSIndexPath** object. For the purposes of the table view, an **NSIndexPath** has two properties: **section** and **row**. The **section** property is exactly what you expect; it identifies the section this row belongs in. We've let our table default to a single section, so we can disregard this portion. The **row** property is equally self-explanatory; it is the location of the row whose cell we need to return. **0** for the firs image, **1** for the next image, **2** for the one below that, etc.

```
static NSInteger imageViewTag = 1234;
UIImageView *imageView = (UIImageView*)[cell.contentView viewWithTag:imageViewTag];
```

These two lines should be new to you as well. **imageViewTag** is an arbitrary number we've chosen. However, what matters is that it remains consistent. A numerical tag can be attached to any **UIView** and used later to recover it from its superview by invoking **viewWithTag:**. This is a quick and dirty way for us to recover the **UIImageView** which will host the image for this cell.

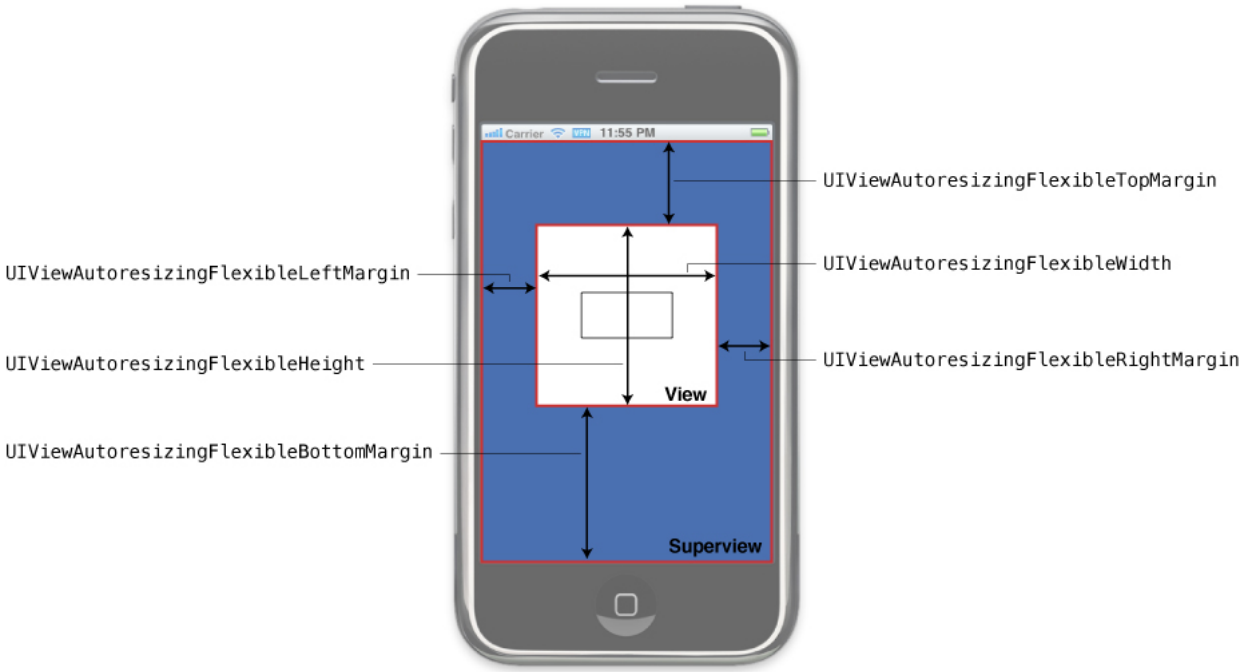
```
if (!imageView) {
    // This is a new cell, it doesn't have an image view yet
    imageView = [[UIImageView alloc] init];
    imageView.contentMode = UIViewContentModeScaleToFill;

    imageView.frame = cell.contentView.bounds;
    imageView.autoresizingMask = UIViewAutoresizingFlexibleHeight | UIViewAutoresizingFlexibleWidth;

    imageView.tag = imageViewTag;
    [cell.contentView addSubview:imageView];
}
```

If **viewWithTag:** fails to recover a **UIImageView**, that means it didn't have one and therefore it's a *brand new* cell. We know it's a new cell because we plan to add a **UIImageView** to each cell we come across. Therefore, on the second time around, it should already be there.

We proceed to allocate a new **UIImageView** object. Its **contentMode** is specified as **UIViewContentModeScaleToFill**, which means the image will be stretched both horizontally and vertically to fill the bounds of the **UIImageView**. Then we set its frame to be the same as the **UITableViewCell**'s **contentView** such that the image consumes the entirety of the cell.

Next we set its auto-resizing property. The **autoresizingMask** is associated with all **UIView** objects. This property lets its superview know how to resize it when the superview's width or height changes. These are called "bit-wise flags" and we set by OR-ing them together using **|**. The available options are as follows:

Resize Option	Effect
UIViewAutoresizingNone	The default option, no resizing will occur if the superview is resized.
UIViewAutoresizingFlexibleWidth	The width of the view will be proportionally stretched.
UIViewAutoresizingFlexibleHeight	The height of the view will be proportionally stretched.
UIViewAutoresizingFlexibleLeftMargin **UIViewAutoresizingFlexibleRightMargin** **UIViewAutoresizingFlexibleTopMargin** **UIViewAutoresizingFlexibleBottomMargin**	When not included, the margins between the view and its parent will remain constant, e.g. if a view is **20** points away from the left edge of its superview, as the superview grows it will remain **20** points away unless the **UIViewAutoresizingFlexibleLeftMargin** is set. When set, the distance from that respective edge will grow proportionally with the superview.



Finally, we set the **`tag`** of our new **`UIImageView`** before we add it to **`contentView`** as a subview. Remember, the next time we dequeue this cell, it will already have a **`UIImageView`** object inside of its **`contentView`** with a tag of **`1234`** and our call to **`viewWithTag:`** will return a reference to that **`UIImageView`**.

Run the app. Unfortunately, you see something resembling the following:

Let's fix that issue by implementing an **`@optional`** method of **`UITableViewDataSource`**.

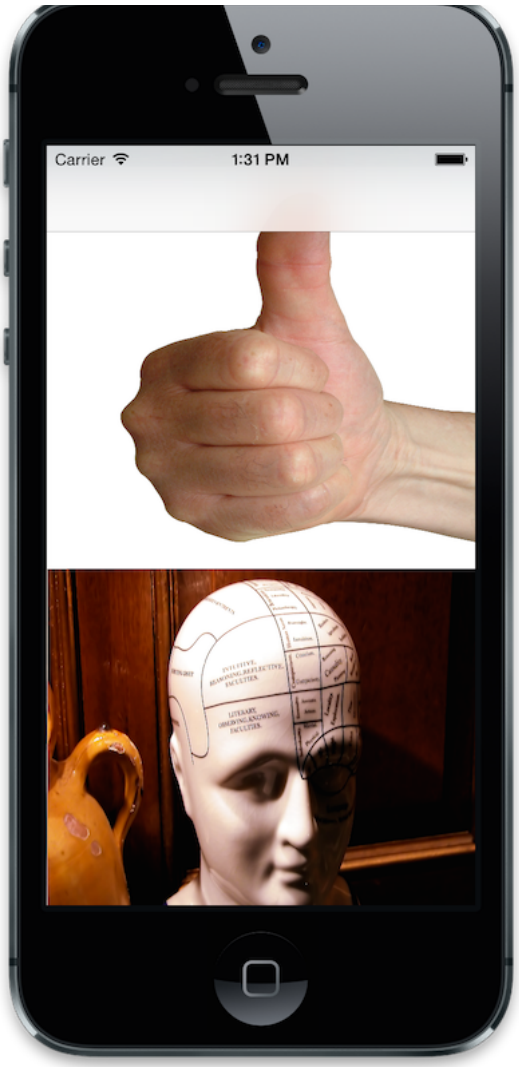
Providing the height

The default height of a **`UITableViewCell`** is 44 points which is precisely why your images have been flattened like pancakes. The **`tableView:heightForRowAtIndexPath:`** data source method is responsible for providing the correct height for each cell at a given index path. Let's see what happens when we increase the size to an arbitrary number; **`300`** for example:

BLCImagesTableViewController.m

```
+ - (CGFloat) tableView:(UITableView *)tableView heightForRowAtIndexPath:(NSIndexPath *)indexPath {
+     return 300;
+ }
```

Run the application and you will see the following:

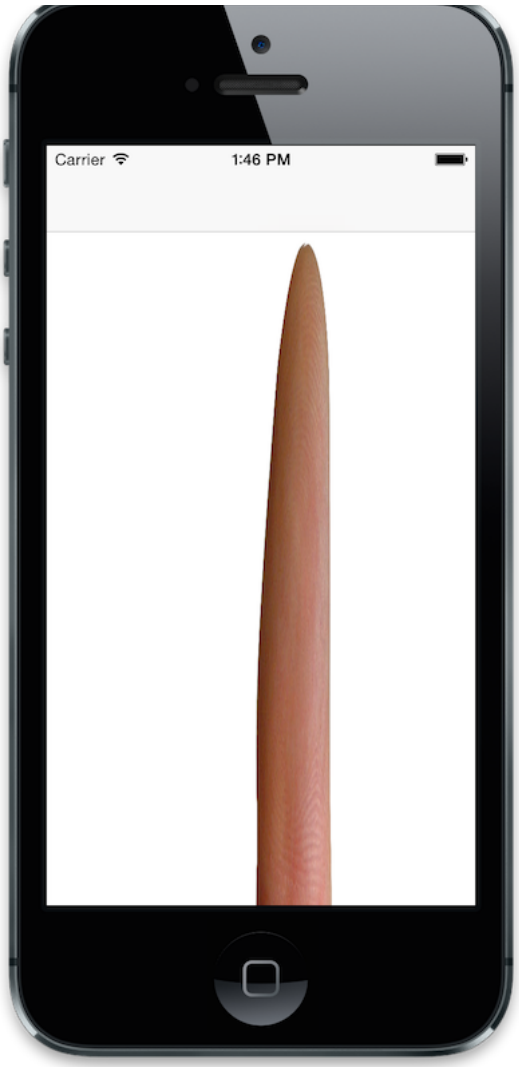


It certainly looks better but there's still an issue with the images here, can you spot it? If not, don't fret. It might become obvious if we try another approach. Let's return the actual pixel height of the image instead of a hard-coded value:

BLCImagesTableViewController.m

```
- (CGFloat) tableView:(UITableView *)tableView heightForRowAtIndexPath:(NSIndexPath *)indexPath {
-       return 300;
+       UIImage *image = self.images[indexPath.row];
+       return image.size.height;
}
```

And our latest changes now show the largest thumbs up this developer has ever seen:



The same issue exists in both versions of our **`tableView:heightForRowAtIndexPath:`**. What's happening is that we're not returning a height that matches the aspect ratio of our image. Remember that we're using the full width of the screen to display our images; **`320`** points. That means the height we choose has to be the same multiple that the width is with respect to the screen.

For example, let's consider an image which is **`500`** pixels tall and **`640`** pixels wide. When we squeeze this image to **`320`** points wide, we've cut its width in *half.* Therefore, when supplying the height, we must cut that in half as well, to **`250`** points. The math we did looks like this:

- (widthOfTheScreen ÷ widthOfThePicture) x heightOfThePicture = heightOfTheCell
- (320 ÷ 640) x 500 = heightOfTheCell
- (½) x 500 = heightOfTheCell
- 250 = heightOfTheCell

Let's mathematize that work into code!

BLCImagesTableViewController.m

```
- (CGFloat) tableView:(UITableView *)tableView heightForRowAtIndexPath:(NSIndexPath *)indexPath {
-     UIImage *image = self.images[indexPath.row];
-     return image.size.height;
+     UIImage *image = self.images[indexPath.row];
+     return (CGRectGetWidth(self.view.frame) / image.size.width) * image.size.height;
}
```

For even better performance, resize the image objects themselves to the exact size in which they'll be displayed. This is likely a reason why photos in Instagram (originally an iOS-only app) are always the same size: 612x612.

You should now have an iOS app that scrolls a bunch of images up and down!

Your assignment:

- Make your image rows *deletable* by implementing `tableView:canEditRowAtIndexPath:` and `tableView:commitEditingStyle:forRowAtIndexPath:`
  - The comments within the pre-written implementation should give you some hints as to how you should accomplish this
  - Remember to update your image array as well to actually delete it from the collection
  - Rows are editable by swiping left on them

Since this is a new project, remember to first create a new repo in GitHub. You can name it "blocstagram" - as we do below - or any other name that's to your liking.

When you're satisfied with your project and the assignment, commit, merge and push your code to GitHub:

Terminal

```
$ git add .
$ git commit -m 'Table View controller implemented'
$ git checkout master
$ git merge building-blocstagram
$ git remote add origin https://github.com/<username>/blocstagram.git
$ git push -u origin master
```

Submit this checkpoint's assignment with links for your repo and commit.

assignment completed