Module II: Navigation

In the first module, we discussed the single view app, wherein we had various controls (typically buttons and text fields and labels) affecting each other within the confines of a single view. This allowed us to conceptualize and create a calculator app, which is essentially a bunch of buttons and a display.

In this module, we will take it one step further and look at the various ways an iOS app lets the user navigate across multiple views (or, in storyboard parlance, *scenes*).

We remind ourselves of the M-V-C paradigm, wherein the model, or data, is separate from the view, or user interface, and separate from the controller. The controller effectively acts as the middle man between the model and the view, hence we frequently talk about passing data around to update the view.

This is particularly manifested in our single view app, where we had one scene in the storyboard, corresponding to one view controller class. Remember, the ViewController.h and .m represent a *subclass* of the generic UIViewController. By creating this sub-class, we are able to imbibe our custom view controller the special processing we need to do.

Also remember that this view controller is physically separate from the scene in the storyboard. They can only act in tandem once the proper connections are made. Hence, if we need access to a particular object in the scene, we need to connect this via an IBOutlet to the view controller. The IBOutlet acts as a reference or proxy of sorts to the on-screen object. They are both of the same class, and any changes made to the IBOutlet affects the on-screen manifestation.

Similarly, we can create target-action pairs by connecting specific events, such as the Touch Up Inside event of a UIButton, to a custom method tagged as an IBAction in our view controller. This lets the view controller respond to these events that happen in the user interface. In response to these events, the controller can fetch data or ask the scene to update its display (through, you guessed it, IBOutlets!).

Do please recall how the IBOutlets and IBActions are done, as these are important aspects of iOS programming that will be used throughout the course.

The Application

As we discussed in Module 1, when the application is started, several things happen:

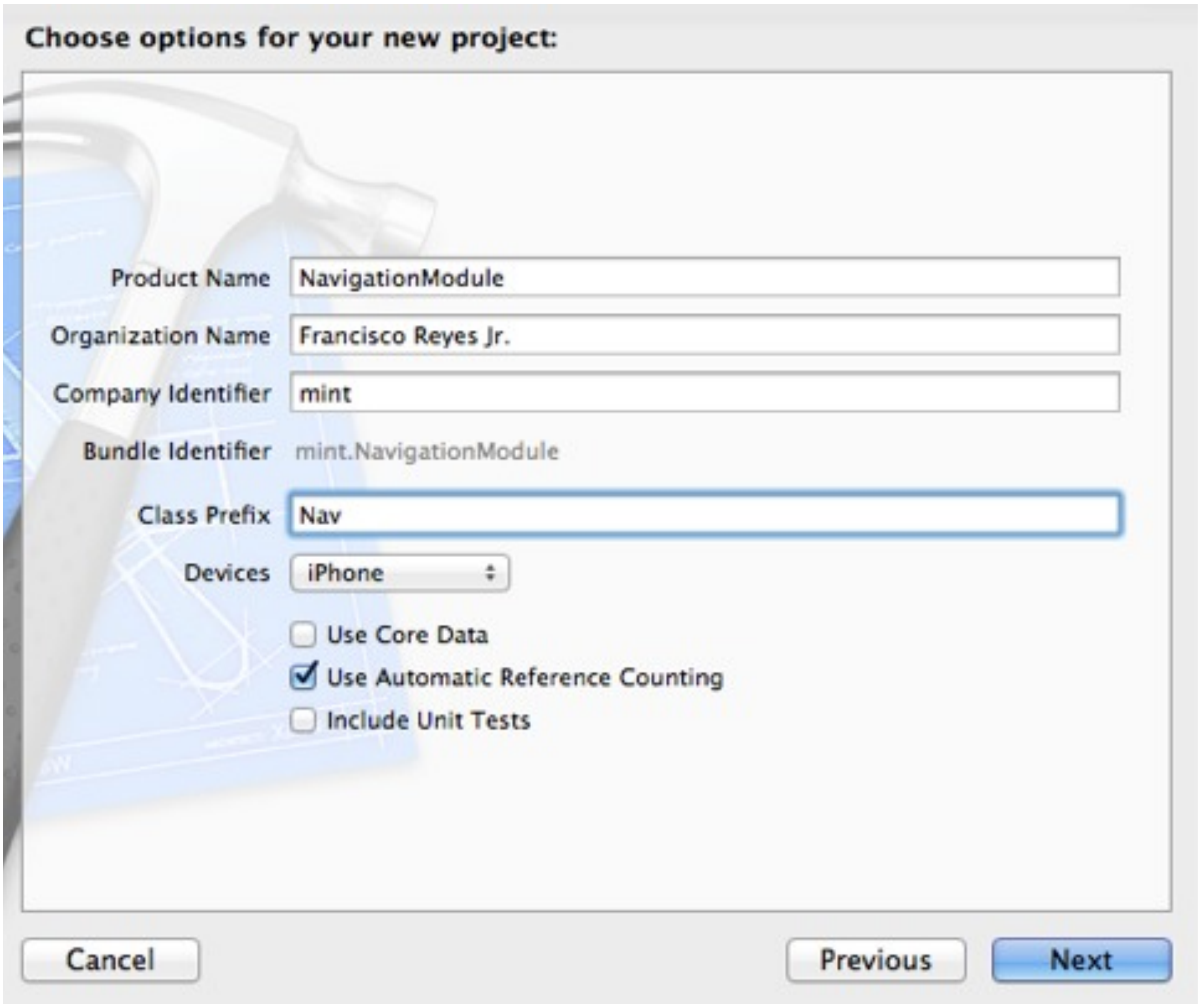
1) A UIApplication object is created, which starts a *run loop* or event loop. This simply waits for things to happen and calls the appropriate objects as needed.
2) A UIApplicationDelegate instance is created. This waits for certain events in the application lifecycle, such as applicationDidFinishLoading: or applicationWillEnterBackground: and so on. As a reminder, a *delegate* is something that does things for another.
3) The third thing that happens on application launch that we didn't quite discuss as much is that the application creates a *window*. Into this window, it creates an instance of the initial view controller or the *rootViewController*. In our previous encounter with the single view app, this root view controller is the one and only scene in the storyboard. This is

indicated by the inward pointing arrow. The behavior to load this as the rootViewController is automatic in a storyboard-enabled project.

NOTE: Pre-storyboard, i.e. iOS versions before 5.0, the main window was initialized in the applicationDidFinishLaunching:withOptions: method of the app delegate, after which an instance of the rootViewController was added as a sub-view of the main window. You'll also find a MainWindow.xib in these older projects. In theory, you can paint directly into the MainWindow.xib, but it's not recommended as you lose the flexibility of UIView changes. Storyboard allows some degree of ease for the programmer, but also has some limitations.

Let's Create an Empty Application

Under File->New->Project... select Empty Application in the iOS Application templates and click Next. For this exercise, we'll call it "NavigationModule" as below.

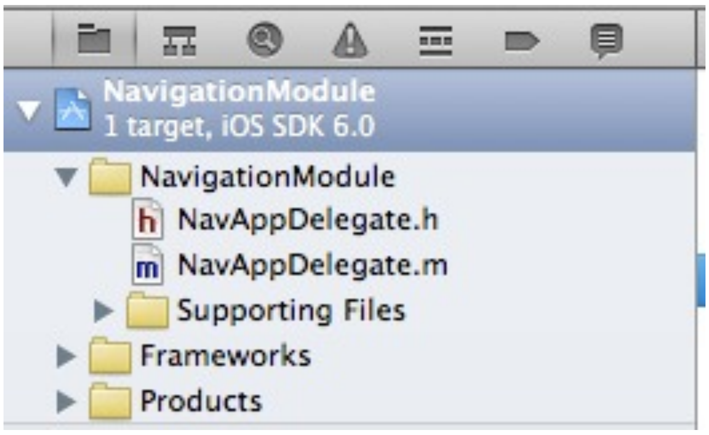

Choose options for your new project:

Product Name: NavigationModule
Organization Name: Francisco Reyes Jr.
Company Identifier: mint
Bundle Identifier: mint.NavigationModule
Class Prefix: Nav
Devices: iPhone

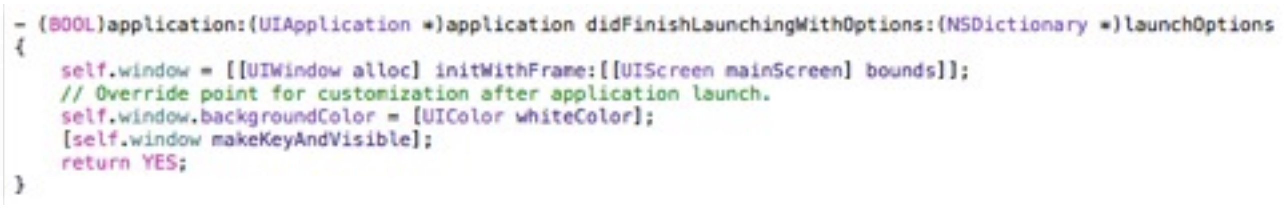
- [ ] Use Core Data
- [✓] Use Automatic Reference Counting
- [ ] Include Unit Tests

Click Next and allow Xcode to create your project files.

Notice in the navigation pane that you only get an AppDelegate:



Try clicking NavAppDeleate.m, and have a look at the applicationDidFinishLaunching:withOptions method:

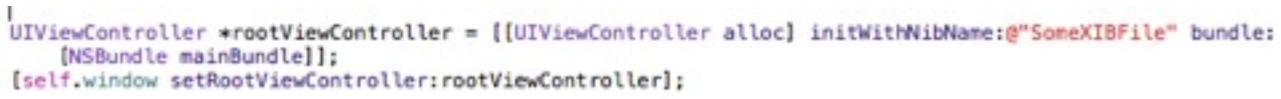


```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.
    self.window.backgroundColor = [UIColor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

You can compare this with the single view application you've been creating earlier, and you'll find that there are several new lines here.

The reason is because the Empty Application template does not have a storyboard. Instead, it manually creates an instance of window.

In case you come across older sample codes without storyboards, you'll find additional lines here which loads a view controller from a "nib" or an xib file, and calls setRootViewController, as in the below example:



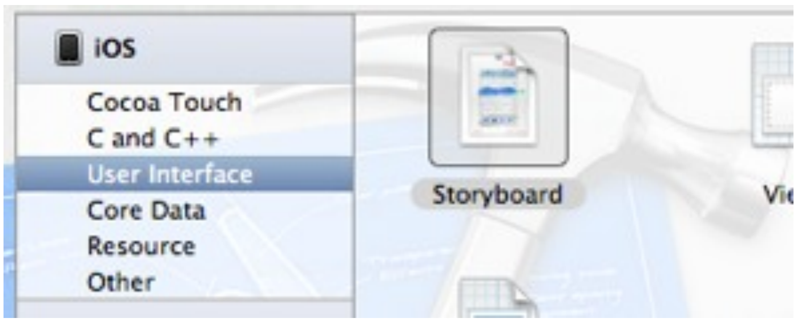
```
UIViewController *rootViewController = [[UIViewController alloc] initWithNibName:@"SomeXIBFile" bundle:
    [NSBundle mainBundle]];
[self.window setRootViewController:rootViewController];
```

However, with storyboards, none of these is necessary since it's all done automatically by the storyboard system.

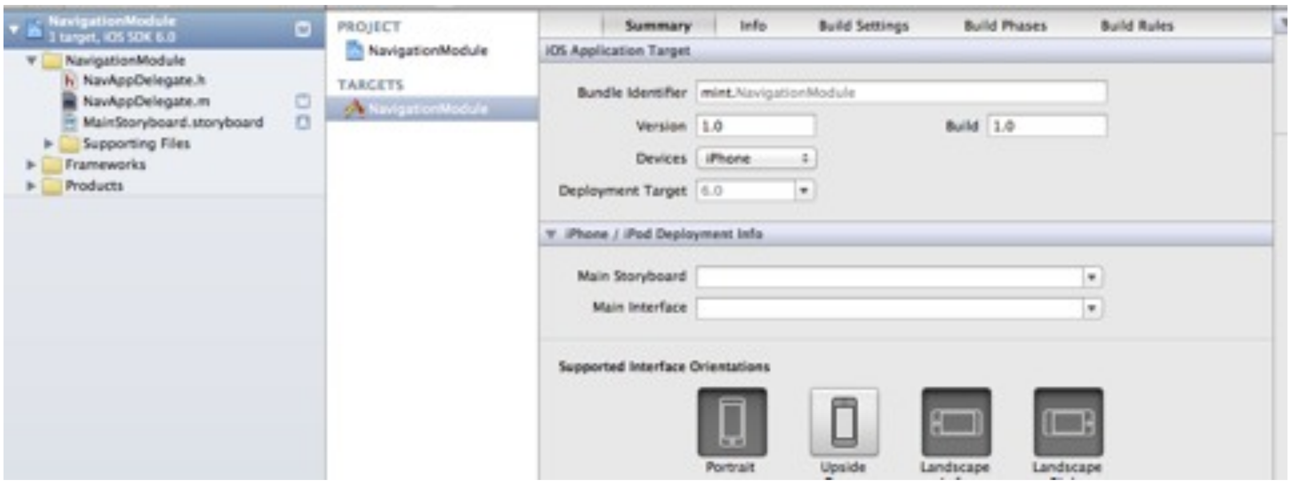
Convert to Storyboard

We can convert our empty application into a storyboard project by:

1) Under File->New->File..., choose User Interface under iOS and select Storyboard.



2) Click Next and choose iPhone under Device.
3) Give your storyboard a name, say MainStoryboard (the default is Storyboard).
4) You will now be taken to the storyboard editor (which will be empty) and you will find MainStoryboard.storyboard in the navigation pane.
5) Click on the project name to go to the project summary.



Bundle Identifier: mint.NavigationModule
Version: 1.0
Build: 1.0
Devices: iPhone
Deployment Target: 6.0
Main Storyboard:
Main Interface:

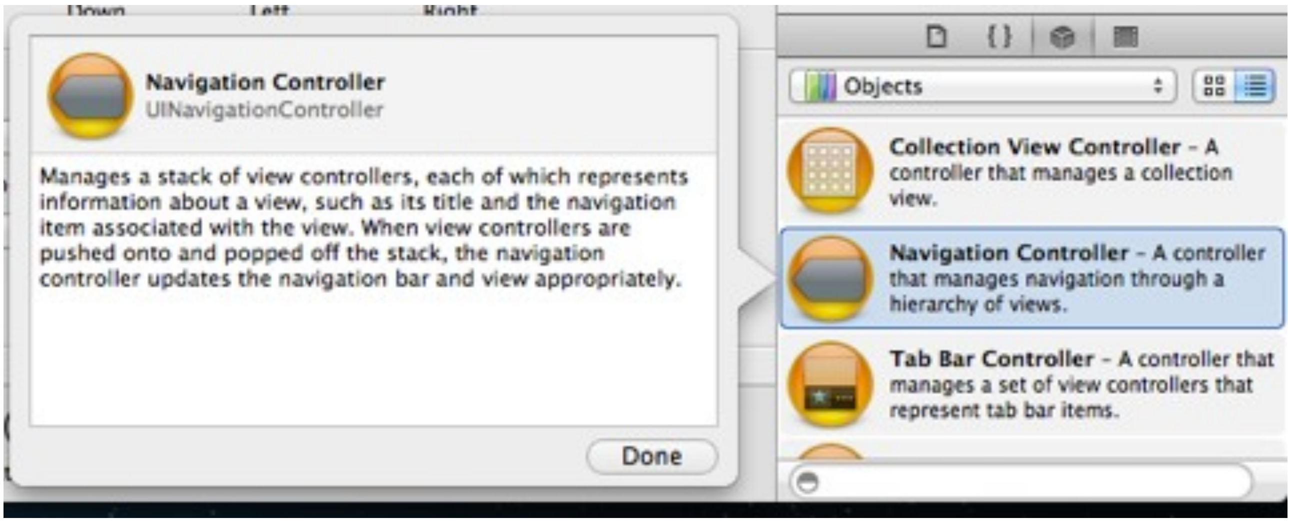
Here you will find the item "Main Storyboard", which you can drop down and select MainStoryboard.storyboard. This causes your project to use the selected storyboard as its main storyboard.

6) Comment out everything in applicationDidFinishLaunching:withOptions: except return YES; As we said, none of this is necessary with the storyboard system. If you fail to comment out or remove this code, the storyboard will not show up.

Note that you can create multiple storyboards, and you can change the active storyboard in your code. Further note, that storyboards are somewhat unwieldy when working on team projects. XIBs are easier to manage for team projects because each UI piece is separate. However, for purposes of this class, we will generally use storyboards, since this is the direction being taken by Apple, but we will spend some time before the end to talk about XIBs as well.

Once you've done this, your project is ready for storyboards.

The Navigation Controller


Navigation Controller
UINavigationController

Manages a stack of view controllers, each of which represents information about a view, such as its title and the navigation item associated with the view. When view controllers are pushed onto and popped off the stack, the navigation controller updates the navigation bar and view appropriately.

Done

Objects

Collection View Controller - A controller that manages a collection view.

Navigation Controller - A controller that manages navigation through a hierarchy of views.

Tab Bar Controller - A controller that manages a set of view controllers that represent tab bar items.

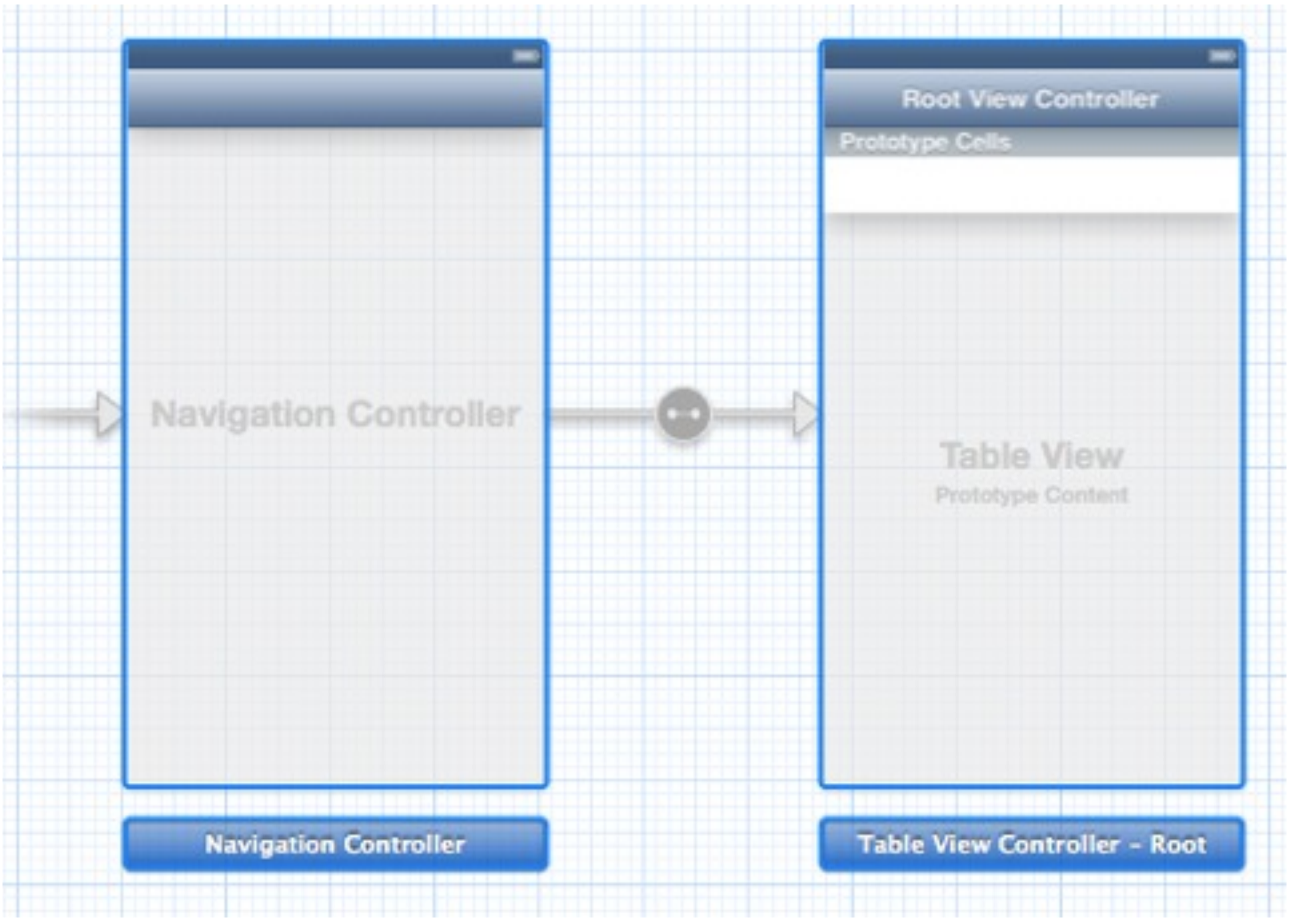
The navigation controller manages a stack of view controllers. That is, it allows a view controller to call up another view controller which gets pushed onto the top of the stack. Remember that only one view is visible at any given time, so the top-most view is what is visible and interacting with the user.

This second view controller can then call up another view (which gets pushed on top of the stack), or the navigation controller can go back to the previous view.

Note that the current (top most) view *and all previous views* are usually retained in memory (assuming there is no low-memory condition). Generally, you can pass data around among these view controllers. In case of a low-memory condition, some of these view controllers may receive a warning asking them to reduce their memory footprint, which is to say, they should destroy any objects (i.e., properties) that they can re-create later.

For our example, we will be using a Navigation Controller as our application's rootViewController. That is, when our app launches, it will create an instance of a Navigation Controller to display in its window.

We can easily do this in our project by going to our storyboard and dragging over a Navigation Controller onto the storyboard. This is what we will get:



Notice that we get a Navigation Controller, with the inward pointing arrow, indicating it as the initial view controller (effectively the app's rootViewController).

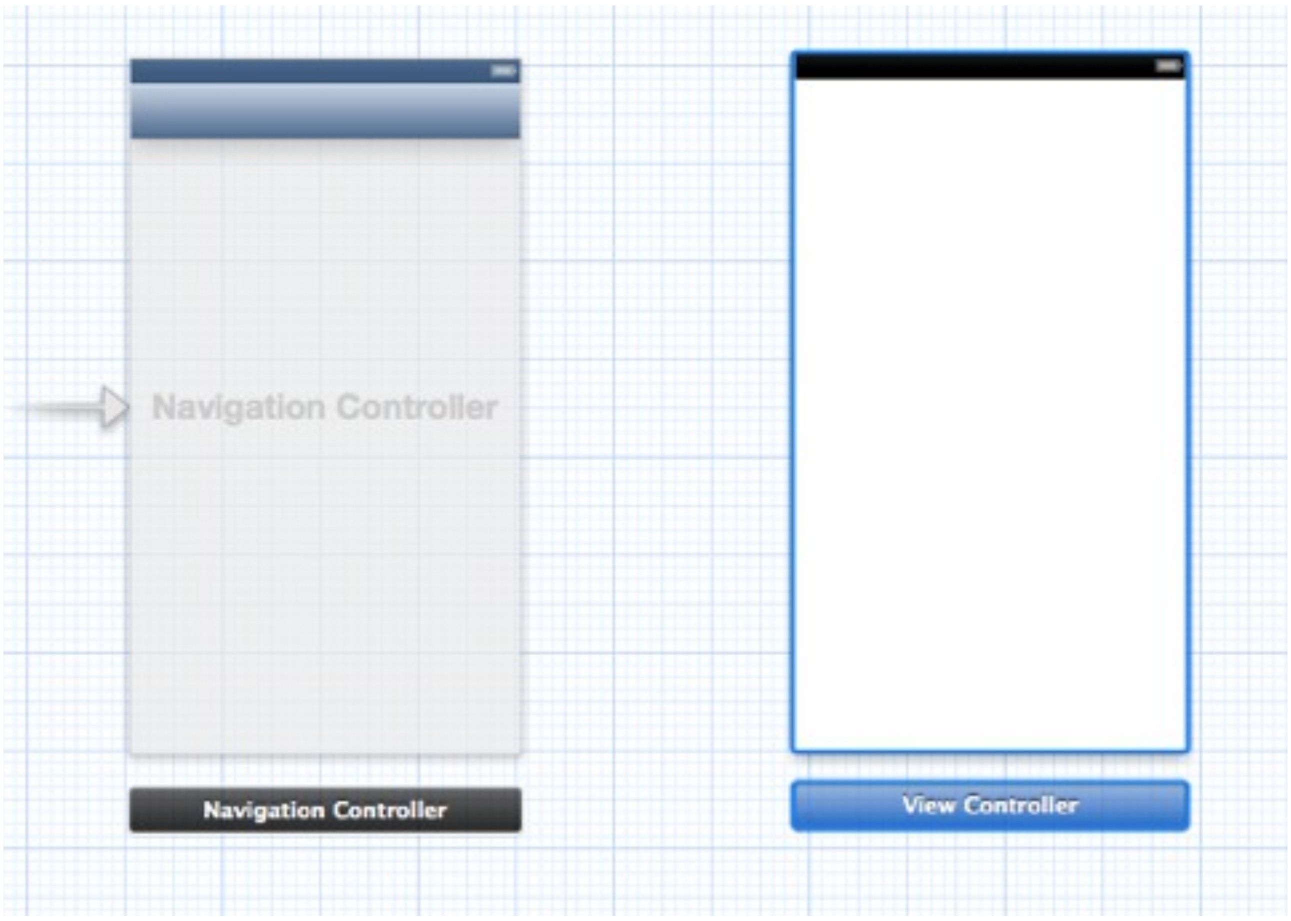
Notice, too, that it automatically creates a Table View Controller which, confusingly enough, is labelled Root View Controller.

Here's the thing, let's go ahead and delete that Table View Controller by clicking on the status bar, and hitting delete. Make sure you delete everything, so you're left with only the Navigation Controller. We'll get back to Table View Controllers later.

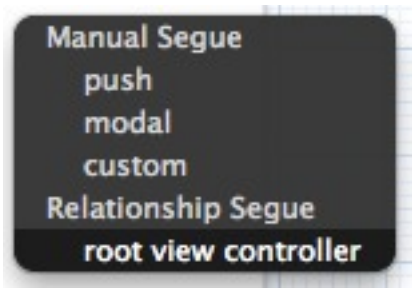
So now we're left with a basic Navigation Controller.

The thing with Navigation Controllers is that they have no view per se. Instead it manages *other* view controllers, and displays these view controllers. So, just like our application, it needs a rootViewController, which is the view controller it initially displays. The Table View Controller was its default root view controller, but since we deleted that we now need to create a new view controller.

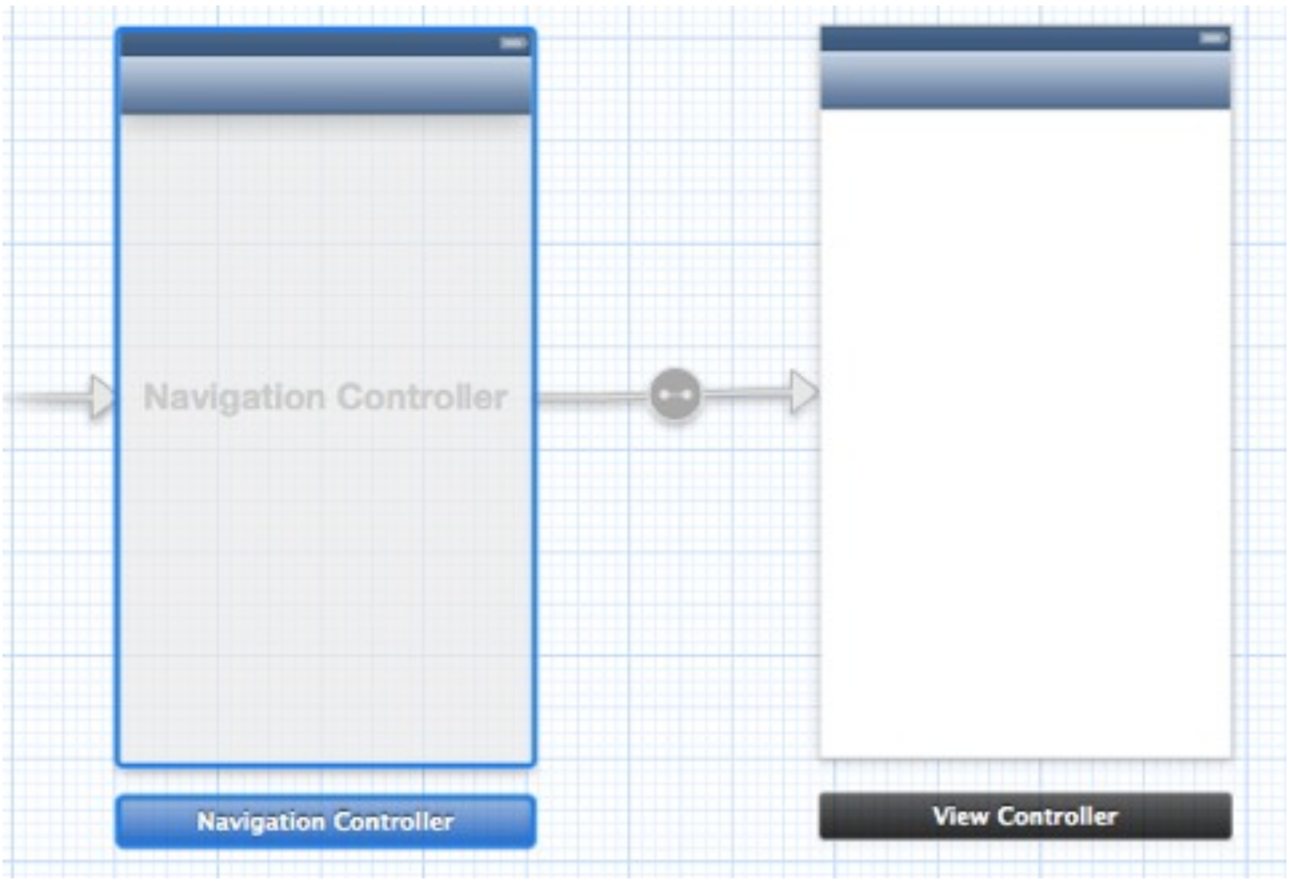
Go ahead and drag over a View Controller onto the storyboard. You should get:



Now control drag from the Navigation Controller to the View Controller, and you get:



Segues, as we've very briefly discussed earlier, and will discuss in greater detail later, are the storyboard's way of transitioning from one scene to another. In this instance, we're more interested in the Relationship Segue, which establishes the root view controller relationship. By selecting that, we now assign the View Controller to be the root view controller of our Navigation Controller.



You will notice that the View Controller changes to accommodate a navigation bar on top. This effectively shrinks the view of the view controller. We mentioned that the size of this nav bar may change depending on certain things.

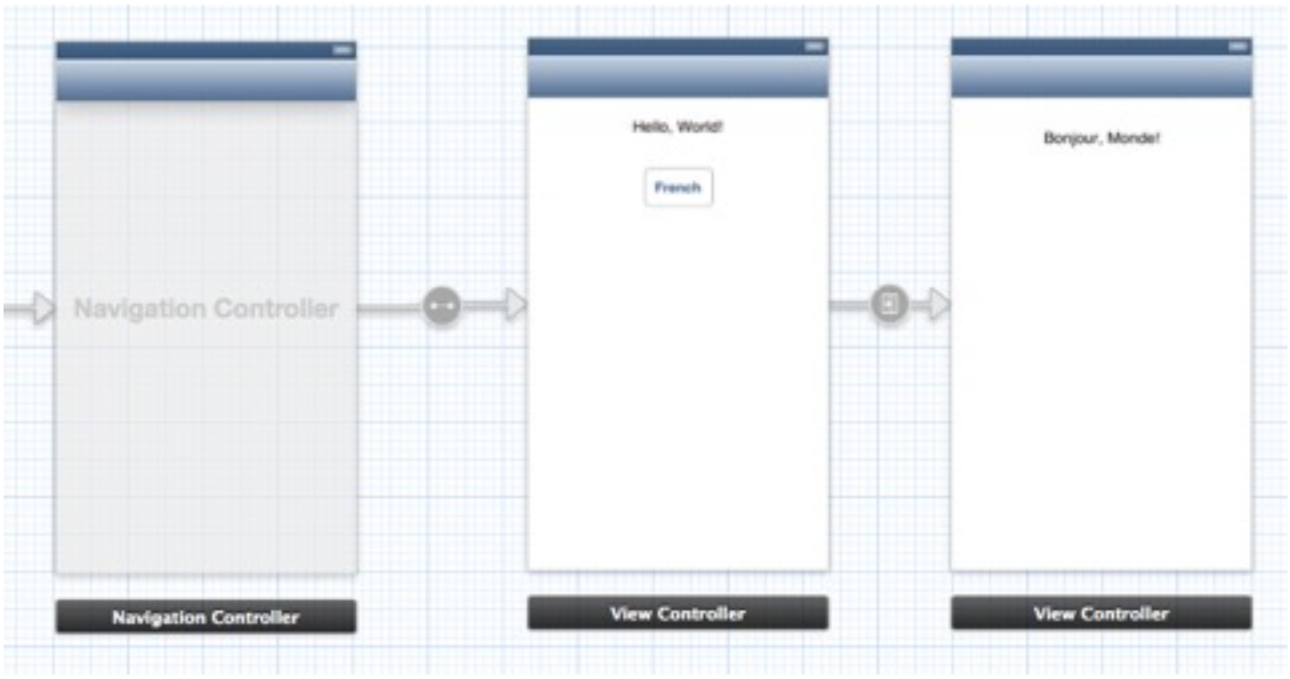
Go ahead and drag a label over to the view controller and change the text to Hello World. Run it, and you should see Hello World show up. Nothing too different from our single view app, so let's make the Navigation Controller do some work for us.

Go back to the storyboard and drag over a button. Label it French.

Now drag over another view controller to the storyboard. Control drag from the French button and connect up a "push" segue. Remember, we couldn't use "push" before because it needed a Navigation Controller. Now that we have the navigation controller, we can use push.

Drag over a label to the second view controller, and change the text to Bonjour, Monde!

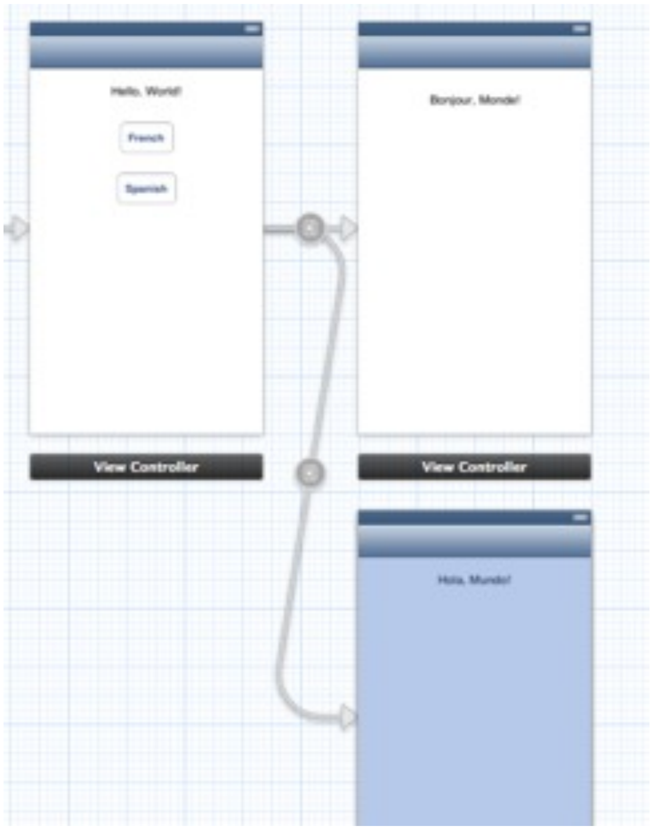
You should have something like this:



Run it, and marvel at the ease in which you've created a translation app. LOL!

But seriously, play around with this by clicking on the nav bar of the first view controller. This will allow you to set things like the Title, and Back button. The title is what's shown in the nav bar. When you transition over to another view controller, the back button normally changes to the title. Alternatively, you can also set an explicit Back button label as well.

Now let's make it more flexible by adding another button, say Spanish, and another view controller with a label saying Hola, Mundo!. Something like:



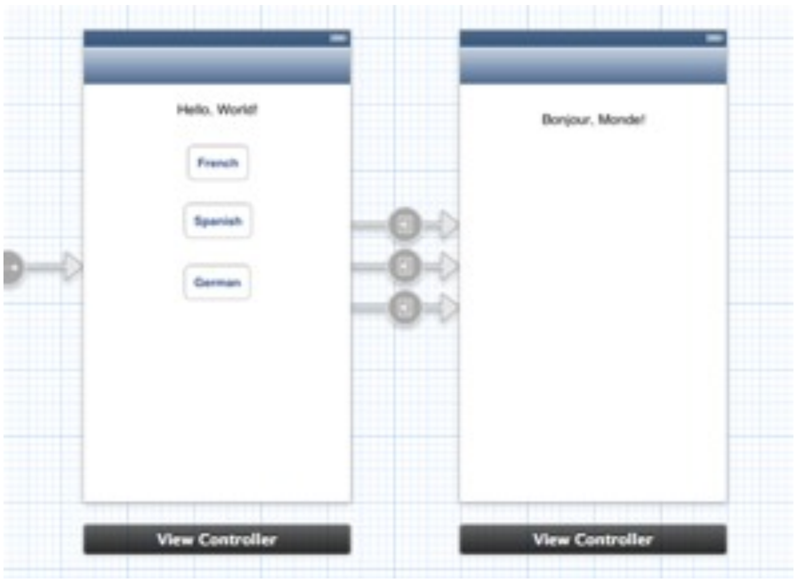
So we can just keep on adding buttons and view controllers translating all the languages of the world!

Instead of doing this, let's put some smarts into the whole thing.

Notice that up until now we've been relying on the automated behavior of storyboards, and the only piece of code we have is the App Delegate.

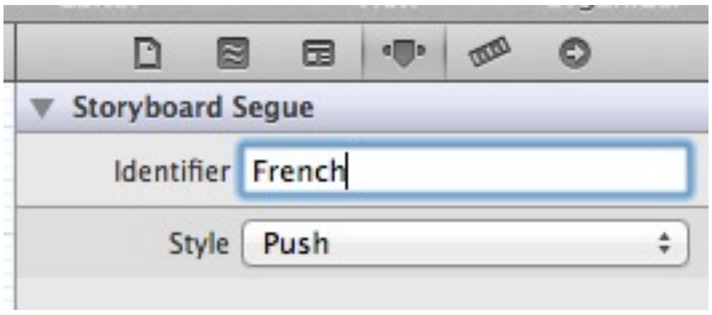
What we want to do is to put a custom view controller behind each of these scenes so that we can avoid having so many view controllers.

Start by removing all but one of the "translation" views. Now make all of your "language" buttons segue into this one view. You should get something like this:



Note that you should have one segue arrow for each button. Make sure all the segues are push.

Now, we need to be able to distinguish each of these segues, so click on the circle, go to the attributes inspector (fourth icon on the right pane) and set the identifier of the segue to the same as the language:



Storyboard Segue

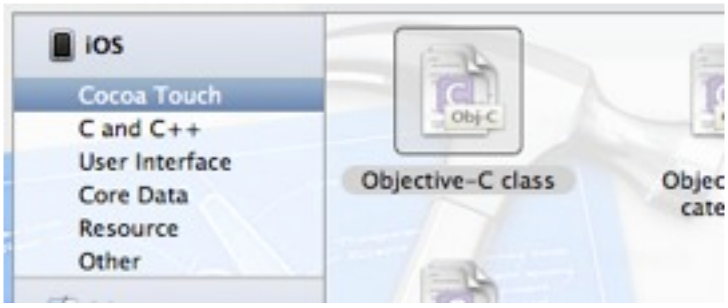
Identifier: French

Style: Push

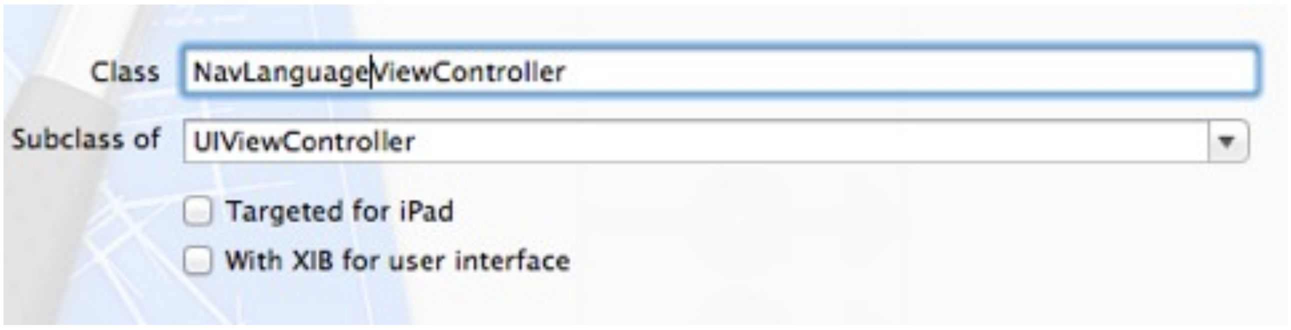
Now that our storyboard is ready, we can create our view controller classes.

Note that what we've been painting on the storyboard is simply the visual representation of the view controller, or the "V" in MVC. What we need to provide now is the "C". To do this, we create a sub-class of UIViewController.

Go to File->New->File... and select Objective-C Class under iOS Cocoa Touch. (Cocoa Touch is the overall UI framework for building apps on iOS).



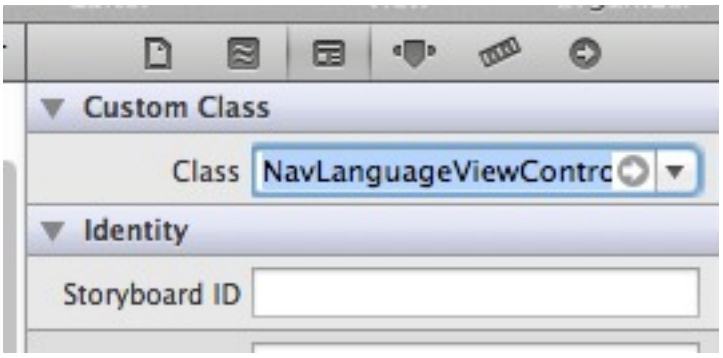
Name your class NavLanguageViewController and make sure it is a subclass of UIViewController:



Class: NavLanguageViewController
Subclass of: UIViewController
Targeted for iPad
With XIB for user interface

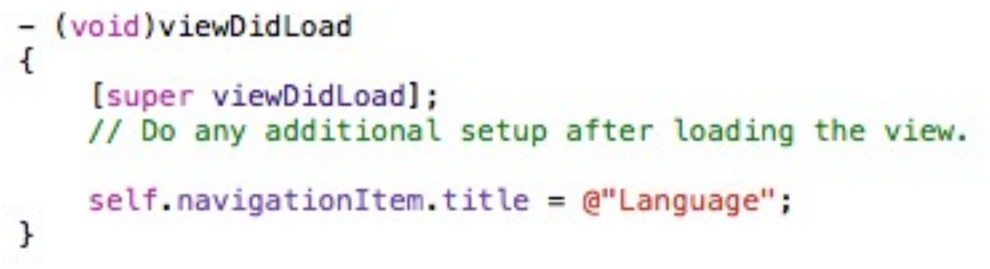
Create the file and note that it creates a .h and .m. So to repeat, what we have just done is to start defining a new class called NavLanguageViewController, which is a sub-class of UIViewController. For now, its behavior is exactly the same as a UIViewController, but we now have a place to write in custom behavior for our view controller.

First thing to do is to change the custom class of our scene to the newly created class. By default, the scene behaves as a normal UIViewController. To change it to NavLanguageViewController, select the language view controller (the one with the buttons), then identity inspector in the right pane, and change Class from UIViewController to NavLanguageViewController



Custom Class
Class: NavLanguageViewContrc
Identity
Storyboard ID:

You can test that it’s working properly by editing (or, in some cases creating) the viewDidLoad method:



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

    self.navigationItem.title = @"Language";
}
```

You should see the title change to Language when you run it.

Next, type in the following new method in NavLanguageViewController.m

```
- (void) prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    if ([[segue identifier] isEqualToString:@"French"]) {
        // translate to French
    }
    if ([[segue identifier] isEqualToString:@"Spanish"]) {
        // translate to Spanish
    }
    if ([[segue identifier] isEqualToString:@"German"]) {
        // translate to German
    }
}
```

Let’s break it down:

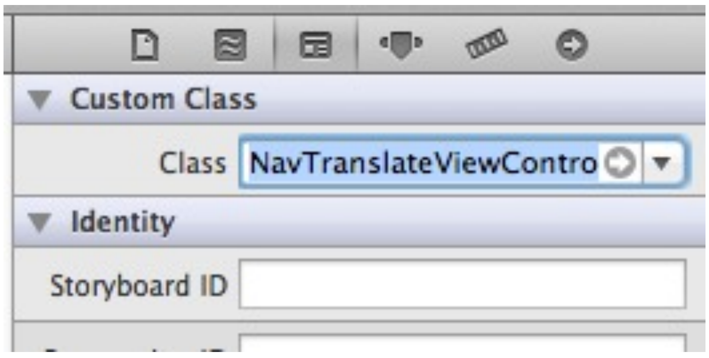
1) The prepareForSegue:sender: method is called just prior to a segue occurring. This gives your custom class the chance to do something. In this case, what we will do is to send over the translation to the next view controller.
2) The three “if” statements determine which segue is going to occur and reacts accordingly. Remember we tagged each segue with an identifier... this is a string identifier, French, Spanish or German.
3) [segue identifier] is equivalent to segue.identifier and returns the string object containing the identifier of the segue that’s about to occur.
4) isEqualToString is the string comparison method. The message is sent to the segue identifier to query if it is equal to one of French, Spanish, or German.
5) BONUS: notice that it also takes the sender (id) which in this case will contain a reference to the button that was clicked... we can actually use this to determine what to do as well. :)

Before continuing, we need to prepare our next view controller to receive the translation.

We do this by, again, defining a custom UIViewController subclass with the appropriate properties and methods.

Start by doing File -> New -> File..., select Objective-C class (under iOS Cocoa Touch), name it NavTranslateViewController, subclass of UIViewController, and create it.

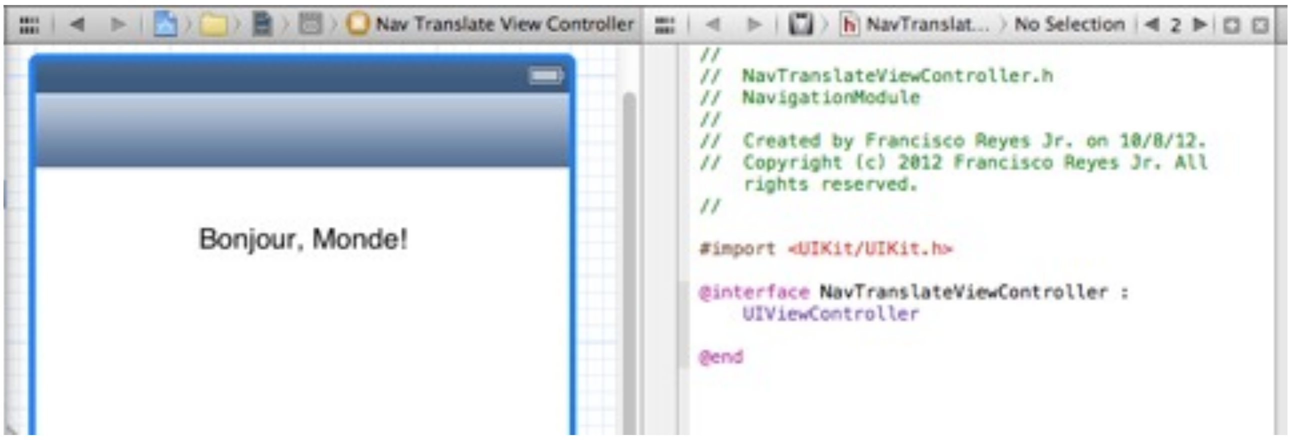
Now go to storyboard, select the second view controller, and change the custom class of the second view controller to NavTranslateViewController.


Custom Class
Class NavTranslateViewContro
Identity
Storyboard ID

You can now endow this view controller with custom behavior.

First thing, create an IBOutlet to the label by:

1) Calling up the assistant editor. Make sure it's showing NavTranslateViewController.h



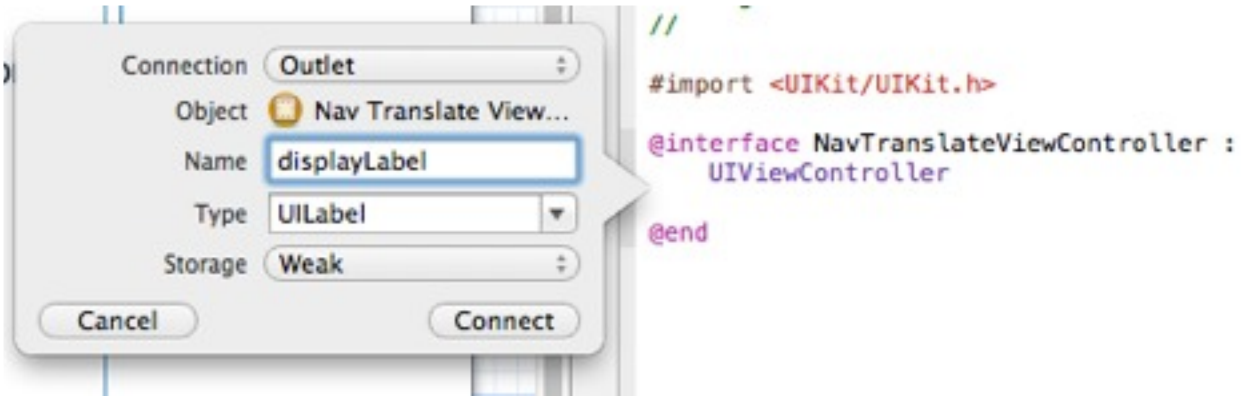
```
//
//  NavTranslateViewController.h
//  NavigationModule
//
//  Created by Francisco Reyes Jr. on 10/8/12.
//  Copyright (c) 2012 Francisco Reyes Jr. All
    rights reserved.
//

#import <UIKit/UIKit.h>

@interface NavTranslateViewController :
    UIViewController

@end
```

2) Control drag from label to the .h, and create an IBOutlet called displayLabel



Connection: Outlet
Object: Nav Translate View...
Name: displayLabel
Type: UILabel
Storage: Weak

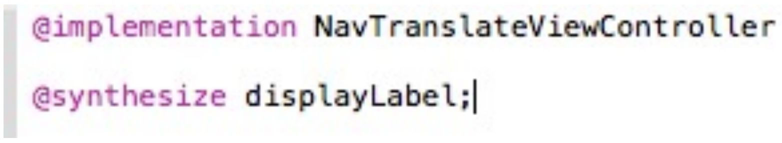
```
//

#import <UIKit/UIKit.h>

@interface NavTranslateViewController :
    UIViewController

@end
```

3) For Xcode prior to 4.5, go to the .m and type in the @synthesize displayLabel right after the @implementation



```
@implementation NavTranslateViewController

@synthesize displayLabel;
```

4) Go back to the NavTranslateViewController.h and create a property called translatedText:

```
@property (strong, nonatomic) NSString *translatedText;
```

5) For Xcode prior to 4.5, goto the .m and type in the @synthesize translatedText right after the previous @synthesize:

```
@implementation NavTranslateViewController

@synthesize displayLabel;
@synthesize translatedText;
```

You are now ready to pass data to this view controller.

Go to NavLanguageViewController.m, and #import the NavTranslateViewController.h header file. This will allow NavLanguageViewController to know the interface of the NavTranslateViewController class:

```
#import "NavLanguageViewController.h"
#import "NavTranslateViewController.h"
```

Now go to the prepareForSegue method and type thus:

```
- (void) prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    NavTranslateViewController *vc = [segue destinationViewController];
    if ([[segue identifier] isEqualToString:@"French"]) {
        // translate to French
        vc.translatedText = @"Bonjour, Monde!";
    }
    if ([[segue identifier] isEqualToString:@"Spanish"]) {
        // translate to Spanish
        vc.translatedText = @"Hola, Mundo!";
    }
    if ([[segue identifier] isEqualToString:@"German"]) {
        // translate to German
        vc.translatedText = @"Hallo, Welt!";
    }
}
```

First, we get a reference to the destination view controller. We need this reference so that we can pass data to it by setting certain properties, in this case, the translatedText property.

Next, depending on the segue identifier, we set the translatedText property to the appropriate value.

Last, we need the NavTranslateViewController to display the translated text.

We do this by going to NavTranslateViewController.m, go to viewDidLoad (or create viewDidLoad method), thus:

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

Note that although the displayLabel itself is a property, it is not recommended to touch IBOutlets from outside. The whole point of view controllers is for it to manage changes to the display by itself based on things that happen. By passing the data through the property, we allow the view controller to do any additional formatting or processing to the information prior to display.