Multi-Screened Applications

An Introduction to the MVC Pattern in iOS

Getting Started – Tutorial 4

Xamarin Inc.

BRIEF

After creating our first iPhone application using MonoTouch, it's time to take a look at the Model, View, Controller (MVC) pattern in iOS and learn how to utilize it to create multi-screen applications. This tutorial introduces the MVC pattern, examines how it is utilized in iOS, and then introduces the UINavigationController – a specialized control that helps to manage screens in a multi-screen application. Finally it walks through creating a multi-screen application utilizing the MVC pattern and the UINavigationController class.

Sample Code:

Hello, Multi-Screen App

Related Articles:

Getting Started Tutorial 3 - Hello, iPhone

Getting Started Tutorial 5 - iPad + Universal Apps

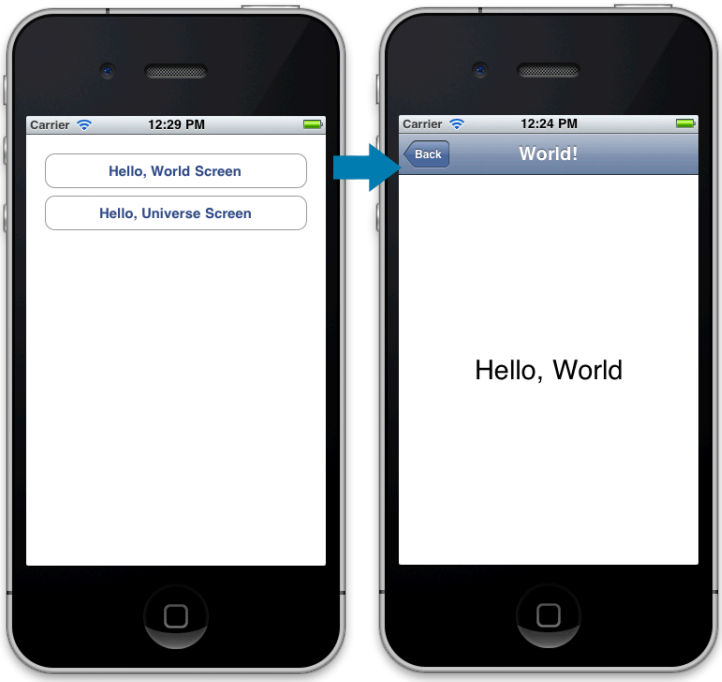
Overview

In the last tutorial we built our first MonoTouch application. We covered a lot of ground and introduced the tools that we use to build MonoTouch applications, as well as fundamental concepts such as Outlets and Actions. However, that application had only a single screen, which is great for a first application, but a rare occurrence in real-world applications.

In this tutorial we're going to take a look at the *Model, View, Controller (MVC)* pattern and see how it's used in iOS to create multi-screened applications.

Additionally, we're going to introduce the `UINavigationController` and learn how to use it to provide a familiar navigation experience in iOS.

Finally, we'll walk through the creation of an application that will include two buttons on the home screen, each of which will show a new screen and allow the user to navigate back home via a navigation controller:



Requirements

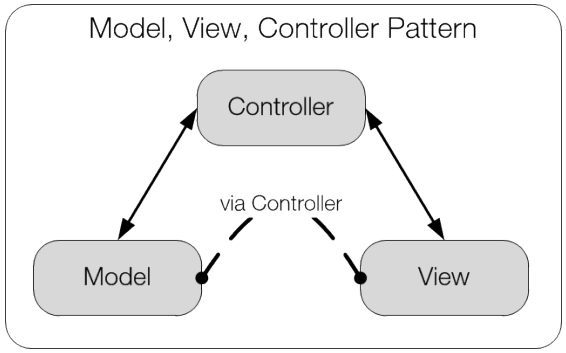
This tutorial builds directly on the skills and knowledge learned in the first getting started tutorial. Therefore, before starting this tutorial, it is necessary to have either completed the previous tutorial, or be familiar with the concepts introduced in it.

Model, View, Controller (MVC) Pattern

As we learned in the first tutorial, iOS applications have only one Window, but they can have lots of screens. This is accomplished via controllers and views. Let's take a look at how that actually works.

Before we dive too deeply into implementation, however, it's important to understand a pattern in iOS programming called the Model, View, Controller (MVC) Pattern.

The MVC pattern is a very old solution (it was created in 1979 at Xerox!) for building GUI applications, and as you can guess, it consists of three components, the *Model*, the *View*, and the *Controller*:



We're going to delve into each one of these components and their responsibilities in just a moment, but in the simplest terms, the MVC pattern is roughly analogous to the structure of ASP.NET pages, or WPF applications in which the View is the component that is actually responsible for describing the UI, corresponding to the ASPX (HTML) page in ASP.NET, or to XAML in a WPF application. The Controller is the component that is responsible for actually managing the View, which corresponds to the code-behind in ASP.NET or WPF.

Neither ASP.NET or WPF applications traditionally use a true MVC pattern, so the analogy is rough, but from a conceptual standpoint, it's a good start. Let's jump in and take a look at each of these components in more detail so that we have a solid understanding before we start using the MVC pattern.

Model

The Model portion of the MVC is typically an application-specific representation of data that is to be displayed and/or entered on the View. It's also very loosely defined in practical implementations of the MVC and is not absolutely necessary.

To make sense of this, let's examine a hypothetical real-world application. Let's imagine that we're creating a To-do list application. This application might have a list of `Todo` objects, each one representing a task item. We might have a screen that displays all the current, outstanding to-do items. The page itself might use a `List<Todo>` collection to represent them. In this case, that `List<Todo>` is the Model component of the MVC.

It's important to note that the MVC is completely agnostic of the data persistence and access of that Model. For example, you might store it in a SQL database, or persist it in some cloud storage mechanism. In terms of the MVC, only the data representation itself is included in the pattern.

Additionally, as aforementioned, the Model portion of the MVC is also an optional component. For example, let's say you have a screen that has instructional content, but no real domain data. In our

To-do application, it's conceivable that we might have a set of instructions screens. Those screens will still be created using Views and Controllers, but they would not have any real Model data.

In fact, in the application that we're going to create in this tutorial, we won't have any Model data. Instead, we'll only have navigable screens that are created using Views and Controllers.

View

The View portion of the MVC is the component that's responsible for describing how the actual screen/page/GUI is laid out. For example, in an ASP.NET page, the HTML portion of the ASPX page can be considered the View. In WPF, the XAML portion of the screen is the View. In iOS applications, the View can be described using XML (as in .xib files), or created programmatically.

In nearly all platforms that utilize the MVC, the View is actually a hierarchy of Views that—when taken as a whole—describe the UI. This means that the page/screen itself is often a View, and it contains subviews or children that describe the controls and elements on it.

In this regard, iOS is no different. Each screen is represented by a View, which then can contain additional controls and elements, themselves each being View objects.

In our To-do application, we might create views that represent different screens such as a page that lists to-do items, and a page that shows the details of a particular item. Additionally, within those screens, we might have specialized views/controls to handle various portions of the display, such as a table that displays the to-do items in a list, or a label that displays a particular item's description.

Controller

The Controller portion of the MVC pattern is the component that actually wires everything together. The Controller is responsible for listening for requests from the user and returning the appropriate View. It also then listens to requests from the View (say, for example, when a button is clicked), does the appropriate processing, View modification, and re-presentation of the View.

Controllers can also manage other controllers. For example, one controller might load another controller if it needs to display a different screen.

The Controller is also responsible for creating or retrieving the Model from whatever backing store exists in the application, and populating the View with its data.

Benefits of the MVC Pattern

The MVC pattern was developed to provide better separation between different parts of GUI applications. This architectural decoupling allows for easier reuse and testing of applications. For example, if you were building an application that targeted multiple devices, you could potentially have a single Controller managing a screen/page, but have different Views for different devices.

The MVC pattern also logically separates the code, making applications easier to understand and, therefore, easier to maintain.

Views and Controllers in iOS/CocoaTouch

In iOS, the application layer that is responsible for the UI is known as CocoaTouch (in OSX programming, it's just Cocoa). As part of CocoaTouch, the views and controllers that are available for building applications are known as the UIKit. The UIKit contains all the native controls that iOS users have come to expect in iOS applications.

In the UIKit, controllers are represented by the UIViewController class and views are represented by the UIView class.

When creating iOS applications, the topmost (first) controller added to the window is called the root controller. In our sample application, we'll set our root controller to be a Navigation Controller that will actually manage a stack of controllers (each representing a screen) and handle much of the user navigation in our application.

UINavigationController

In the sample application that we're going to build, we'll use the UINavigationController to help manage navigation between multiple screens. The UINavigationController is a very familiar control in iOS applications, as it's used all over the place in the stock applications that come with iOS. For example, navigating between screens in the Settings Application is done via the UINavigationController:

The UINavigationController does two things for us:

- **Provides Hooks for Forward Navigation** – The navigation controller uses a hierarchal navigation metaphor in which screens (represented by controllers) are "pushed" onto the navigation stack. The navigation controller provides a method that allows us to push controllers onto it. It also manages the navigation stack, including optionally animating the display of the new controller.
- **Provides a Title Bar with a Back Button** – The top portion of the navigation controller is known as the title bar and when you push a new item onto the navigation stack, it automatically displays a "back" button that allows the user to navigate backwards ("popping" the current controller off the navigation stack). Additionally, it provides a title area so we can display the name of the current screen.

It's important to note that there are other ways to handle navigation in your applications as well. For example, you might use a Tab Bar controller to split your application into different functional areas, or if you're building an iPad application, you might use the Split View controller, which allows you to create Master/Detail views.

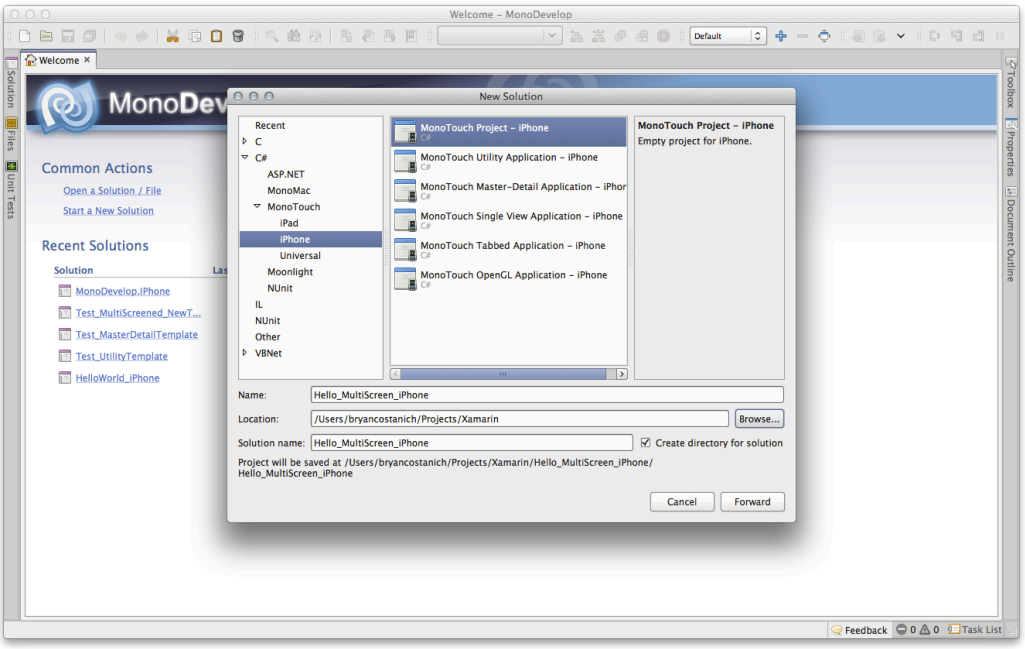
In this tutorial we're using the Navigation Controller because it's probably the single most commonly used way to provide multi-screen functionality in an application, and is often used with other navigation controls, such as the Tab Bar and the Split View Controller.

Creating our Application

OK, now that we have an understanding of the MVC pattern and the Navigation Controller, and how they're used in iOS programming, let's get started building our first multi-screened application.

First, start a new solution, but this time we're going to begin with the **MonoTouch Project – iPhone** template. This creates a mostly empty solution that we'll build from the ground up.

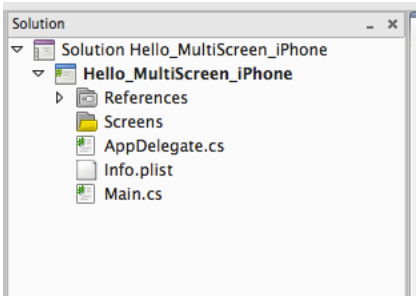
Let's name it `Hello_MultiScreen_iPhone`:



We're going to have three screens in our application, a home screen and two subscreens that can be launched via the home screen, so let's create those.

Creating the Screens

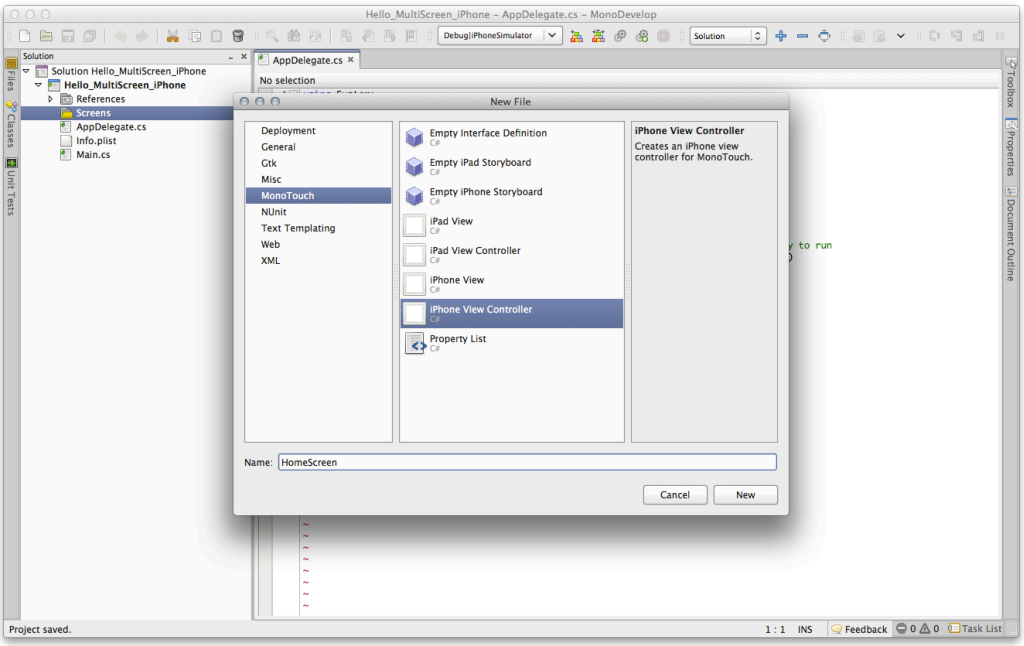
It's important to be organized, so we're going to first create a folder called `Screens` in our project where we'll create our screens. To create the folder, right-click on the project and choose **Add > New Folder**. Name it `Screens`, and the resulting project structure should now look like the following:



Next, let's create our three screens using the **iPhone View Controller** template in our **Screens** folder. This template will create three files for each screen:

- **Controller Class** – A C# class that derives from UIViewController.
- **XIB File** – The UIView, defined in a .xib file that can be edited in Xcode's Interface Builder (IB).
- **Designer File** – A .designer.cs file that MonoDevelop uses to expose the Outlets and Actions defined in our .xib file to our controller.

To create a new iPhone View Controller, right-click on the **Screens** folder and choose **Add > New File**, then choose **MonoTouch : iPhone View Controller**:



Do this three times, naming the files:

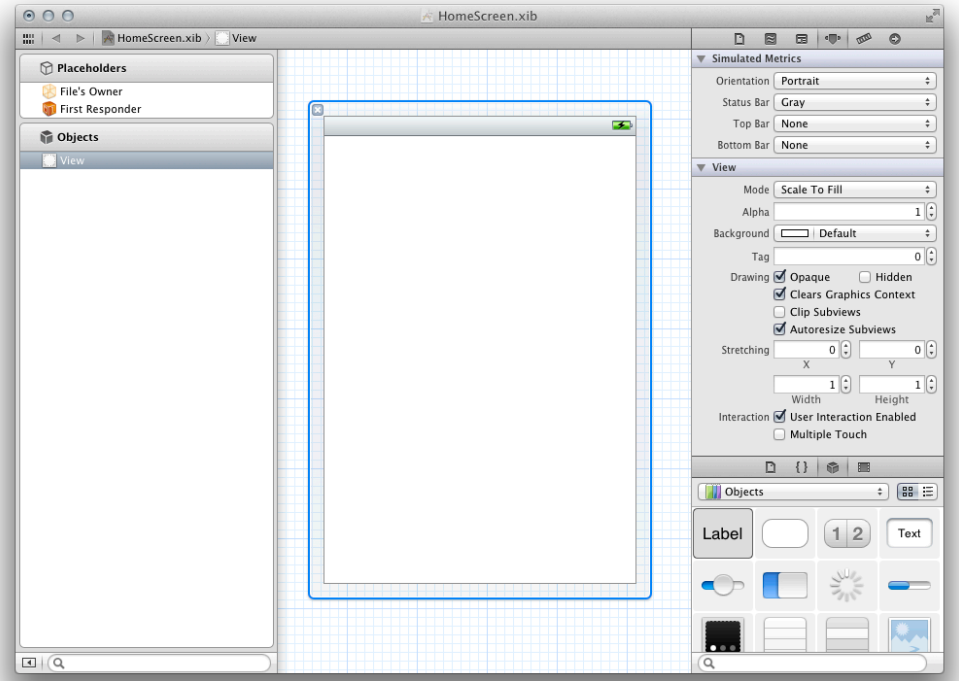
- ➔ **HomeScreen**
- ➔ **HelloWorldScreen**
- ➔ **HelloUniverseScreen**

As a naming convention, we'll use `[ScreenName]Screen` because a screen typically consists of a Controller and a View, which this template creates for us.

Next, we're going to design each of these screens using Xcode's Interface Builder.

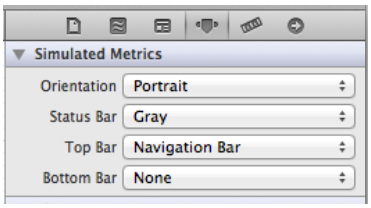
HOMESCREEN.XIB

To edit the HomeScreen.xib in Interface Builder, double-click on it from the Project Navigator in MonoDevelop. Xcode should then open up and load the .xib file in Interface Builder:

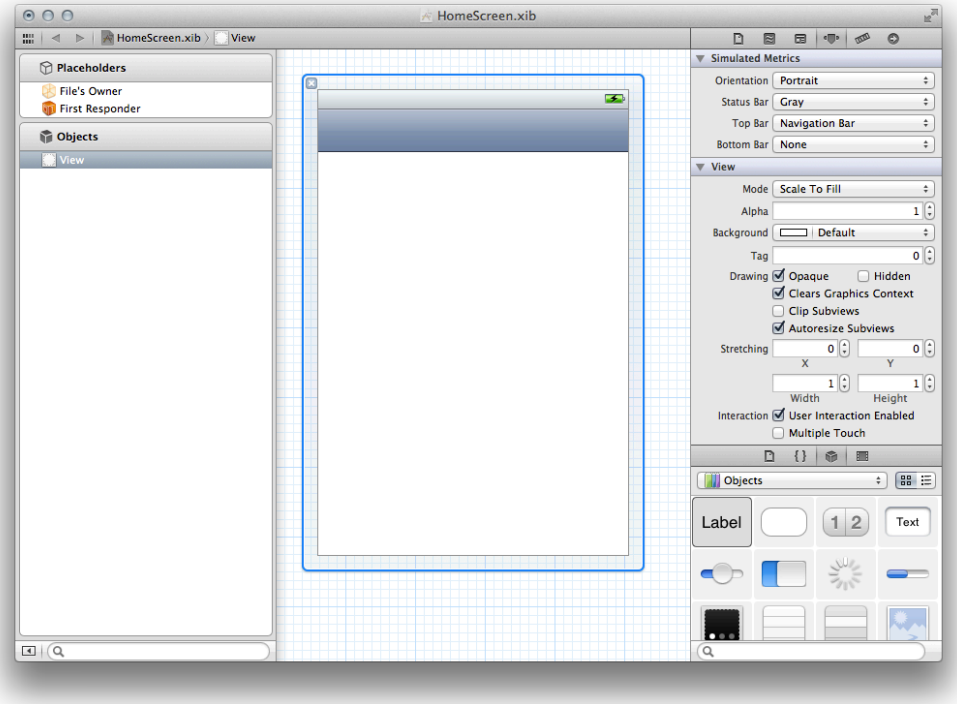


We're going to add a Navigation Controller, which will take up part of the screen, so let's have Xcode simulate what the screen would look like with an empty Navigation Bar at the top:

To view the simulated navigation bar, from the **Simulated Metrics** portion of the Attributes Inspector, choose **Top Bar : Navigation Bar**:

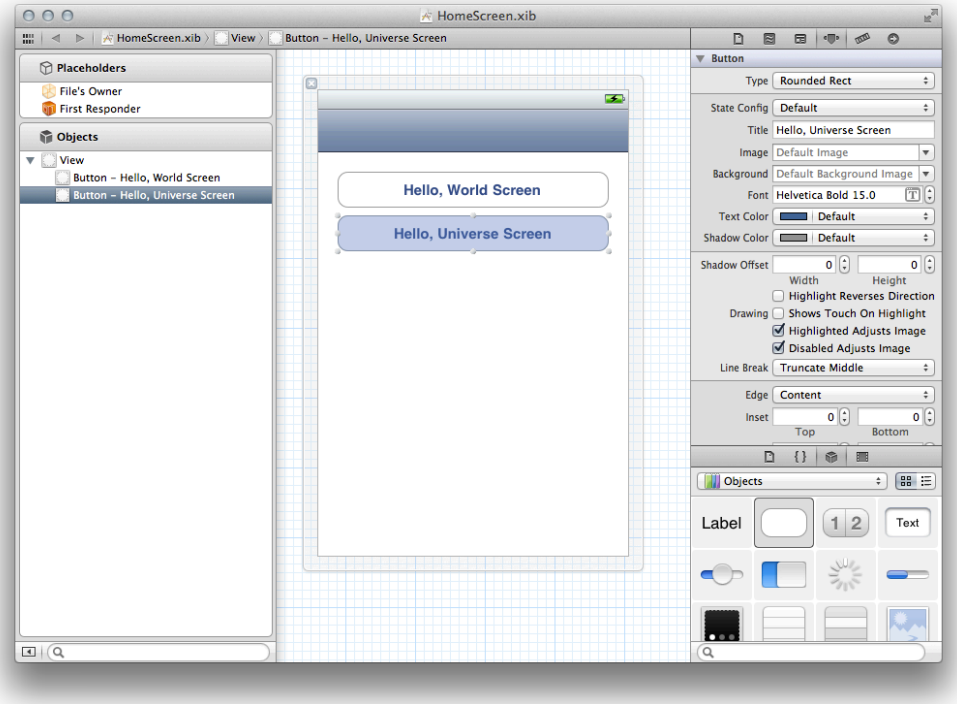


IB will now display a blue/gray navigation title bar at the top of the window.



This simulated element can help when designing screens because it simulates where an item would go, and shows how much space on the screen you actually might have.

Next, add the following two buttons to the home screen:



Once you've added the buttons, create an Outlet for each one:

➔ **btnHelloWorld**

➔ **btnHelloUniverse**

If you've forgotten how to add an Outlet, review the Hello, iPhone Getting Started tutorial's section on Outlets. The basic steps are: view the **Assistant Editor** and *Control-Drag* to the corresponding .h file.

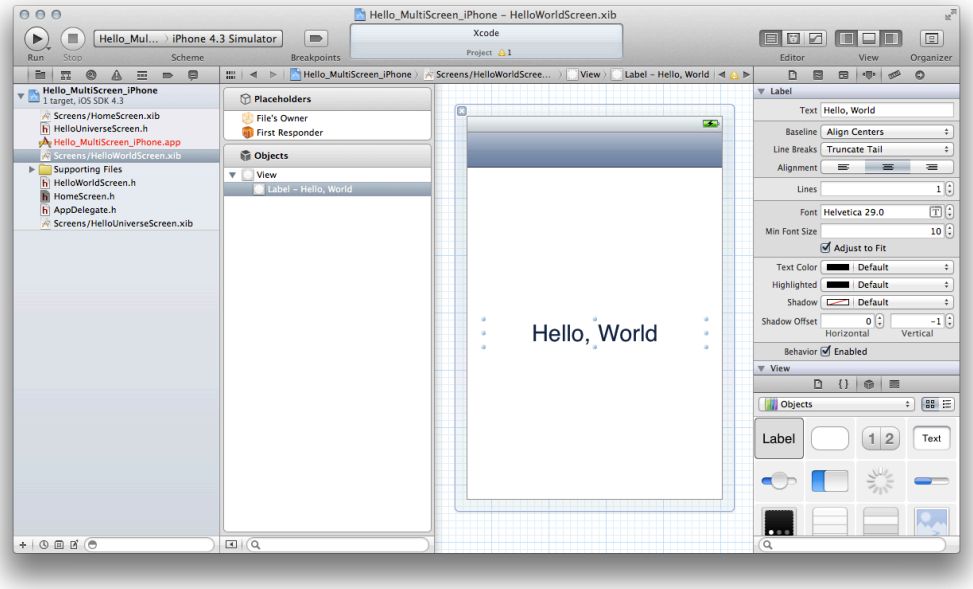
To confirm, once you've added the Outlets, your .h file should have the following code:

```
#import <UIKit/UIKit.h>

@interface HomeScreen : UIViewController {
    UIButton *btnHelloWorld;
    UIButton *btnHelloUniverse;
}
@property (nonatomic, retain) IBOutlet UIButton *btnHelloWorld;
@property (nonatomic, retain) IBOutlet UIButton *btnHelloUniverse;

@end
```

OK, let's create our other two screens now.

HELLOWORLDSCREEN.XIB

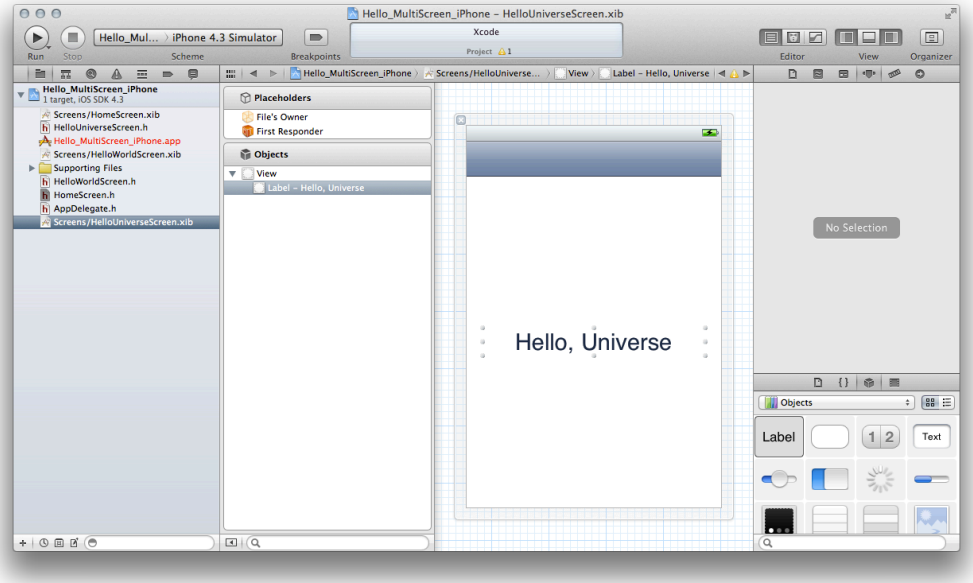
The HelloWorldScreen is very simple, just add a label that says "Hello, World:"



We don't need any Outlets on this screen.

HELLOUNIVERSESCREEN.XIB

Do the same thing with the HelloUniverse screen, but this time make the label say "Hello, Universe:"



Again, we don't need any Outlets here.

We're done in Interface Builder now, so save all your changes and pop back over to MonoDevelop.

Adding our UINavigationController

I mentioned that we're going to add a UINavigationController as our root controller on our window and then we'll add our screen controllers to that.

To accomplish this, first, we need to declare our `UINavigationController` as a class-level variable in our `AppDelegate` class (found in AppDelegate.cs):

```
public partial class AppDelegate : UIApplicationDelegate
{
        //---- declarations
        UIWindow window;
        UINavigationController rootNavigationController;
        ...
}
```

We have to declare this as a class-level variable because of a minor garbage-collection bug in MonoTouch. If we declare it in the `FinishedLaunching` method where we add it to the window, we might lose our navigation controller object as it might get cleaned up when the method returns. This shouldn't happen because a reference should be kept to it from the `Window` object, but in this particular case MonoTouch doesn't know that the reference is held there. This will be fixed in the upcoming release for iOS 5, but until then we have to live with this small problem.

In the `FinishedLaunching` method of the `AppDelegate` class, we need to do the following things:

1. **Instantiate the Window, UINavigationController, and HomeScreen**
2. **Add the Home Screen Controller to the UINavigationController**
3. **Add the Navigation Controller's View to the Window**
4. **Display the Window**

Some of this should already be familiar from the first tutorial, but there are some new bits:

```
public override bool FinishedLaunching (UIApplication app,
```

```
        NSDictionary options)
{
        this.window = new UIWindow (UIScreen.MainScreen.Bounds);

        //---- instantiate a new navigation controller
        this.rootNavigationController = new UINavigationController();
        //---- instantiate a new home screen
        HomeScreen homeScreen = new HomeScreen();
        //---- add the home screen to the navigation controller
        // (it'll be the top most screen)
        this.rootNavigationController.PushViewController(
                homeScreen, false);

        //---- set the root view controller on the window. the nav
        // controller will handle the rest
        this.window.RootViewController = this.rootNavigationController;

        this.window.MakeKeyAndVisible ();

        return true;
}
```

Most of this is pretty self-explanatory, but let's look at a couple of the trickier items:

```
this.rootNavigationController.PushViewController(homeScreen, false);
```

When you want to show a new screen using the navigation controller, you use the `PushViewController` method, which takes two parameters, the controller that you want to push, and a `bool` of whether or not to animate the transition. In this case, we're looking at the first screen on the navigation stack and we're pushing it before the window is even shown, so we pass `false`, since there's no need to animate something that's not even visible.

The second interesting line of code is this:

```
this.window.RootViewController = this.rootNavigationController;
```

In this case, our root (topmost) controller on the window is going to be our navigation controller, since it will manage all other controllers.

If we run our application now, we should see something like the following:

The navigation controller loads along with the first screen (as part of the navigation controller). However, nothing happens when we click on our buttons, so let's wire those up next.

Wiring up Navigation

When the buttons on the home screen are clicked, we want to load the appropriate screen onto the navigation stack. This is done the same way that we pushed the home page in `FinishedLaunching`, by using the `PushViewController` method on our navigation controller.

First, we need to declare references to the subscreens that we will load in our `HomeScreen` class (found in HomeScreen.cs):

```
public partial class HomeScreen : UIViewController
{
        HelloWorldScreen helloWorldScreen;
        HelloUniverseScreen helloUniverseScreen;
        …
}
```

Now let's wire up our buttons.

VIEWDIDLOAD

This is also a good time to introduce the most important event method of a UIView's life cycle – the *ViewDidLoad* method.

The ViewDidLoad method is called on the controller that manages that View after the View has been fully loaded (whether you've defined your View in a .xib as we did, or you have a custom View in code). This is a good time to add any additional subviews, wire up event handlers, or modify the View. In our case, we want to wire up the `TouchUpInside` events that our buttons raise on our HomeScreen controller.

So let's do just that. Open up the HomeScreen.cs file and go to the `public override void ViewDidLoad` method that is added by MonoDevelop.

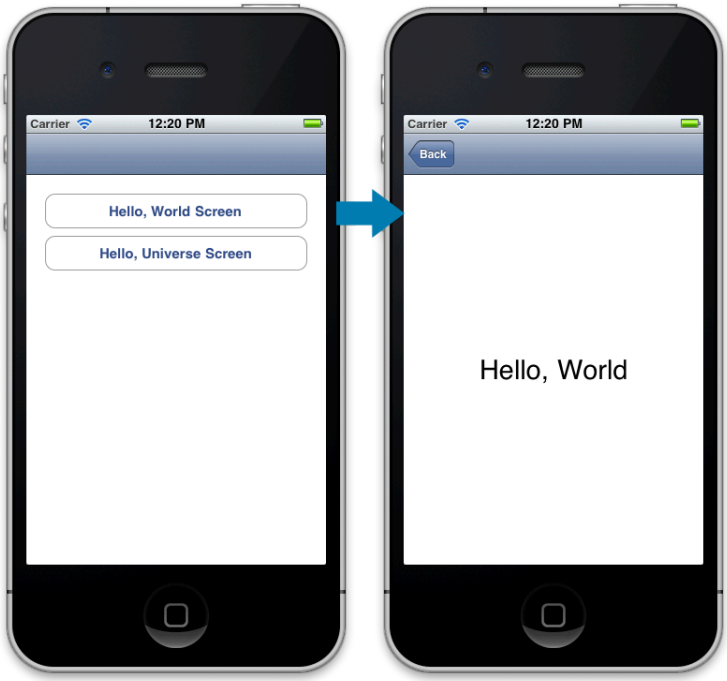
Let's wire up the `TouchUpInside` events on our buttons to push the appropriate screens onto our navigation controller in ViewDidLoad:

```
public override void ViewDidLoad ()
{
        base.ViewDidLoad ();

        //---- when the hello world button is clicked
        this.btnHelloWorld.TouchUpInside += (sender, e) => {
                //---- instantiate a new hello world screen, if it's null
                // (it may not be null if they've navigated backwards
                if(this.helloWorldScreen == null)
                { this.helloWorldScreen = new HelloWorldScreen(); }
                //---- push our hello world screen onto the navigation
                //controller and pass a true so it navigates
                this.NavigationController.PushViewController(
                        this.helloWorldScreen, true);
        };

        //---- same thing, but for the hello universe screen
        this.btnHelloUniverse.TouchUpInside += (sender, e) => {
                if(this.helloUniverseScreen == null)
                { this.helloUniverseScreen
                        = new HelloUniverseScreen(); }
                this.NavigationController.PushViewController(
                        this.helloUniverseScreen, true);
        };
}
```

I used Lambda expressions for brevity, but you could just as easily have added a delegate (see Consuming Events - MSDN Reference for more information).

We've not really introduced anything new here. We're using the same method we used in our `FinishedLaunching` method to push a controller onto the stack; the only difference here is that this time we're passing a `true` value for the `bool animate` parameter, telling the navigation controller to animate the transition.

Now when we run the application, we can click on the buttons and our subscreen will load and slide on with a nice cinematic animation:



Adding Titles to Pages

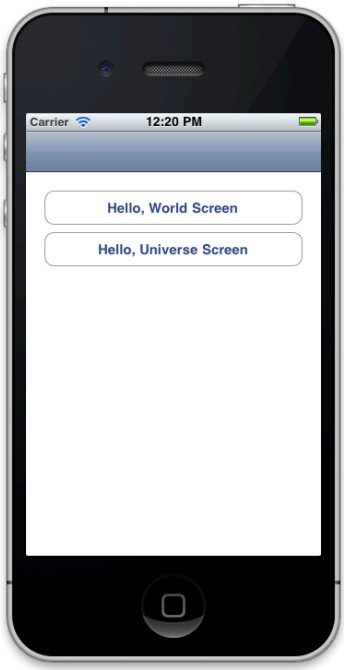
The application looks a little weird right now, however, because we don't have any titles on the navigation bar. These are easy enough to add, though. `UIViewController` objects have a `Title` property that the navigation controller will display if set. So to add titles to our pages, we simply open up our `HelloWorldScreen` and `HelloUniverseScreen` classes and set the `Title` in the constructor:

```
public HelloWorldScreen () : base ("HelloWorldScreen", null)
{
        this.Title = "World!";
}
```

Now our "Hello" screens will display a title in the navigation bar:

Hiding the Navigation Bar

Let's make one final change, and introduce a couple more View life cycle events. Right now, when you load the home page, you see something like this:



The navigation bar is somewhat unnecessary here, so let's hide it. However, we still want it to show up on the sub-pages, so that we can navigate back home.

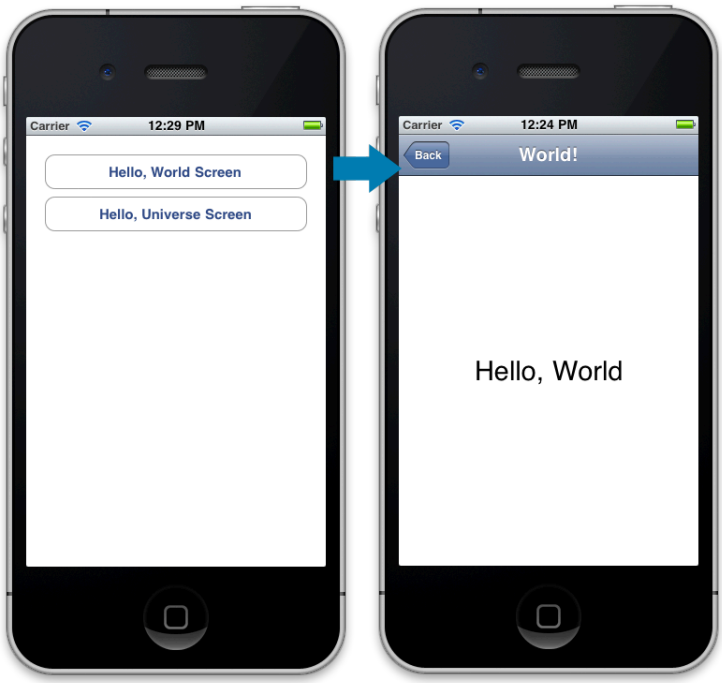
To accomplish this, we're going to hide the navigation bar when the home screen is shown, but make it visible when other screens are shown (more specifically, when the home screen disappears).

It just so happens that there are two View life cycle events, *ViewWillAppear* and *ViewWillDisappear*, that are called when this happens. Additionally, the UINavigationController exposes a method called `SetNavigationBarHidden` that lets us to show or hide the navigation bar.

Combining these, we can do the following in our `HomeScreen` class:

```
public override void ViewWillAppear (bool animated)
{
        base.ViewWillAppear (animated);
        this.NavigationController.SetNavigationBarHidden (true, animated);
}

public override void ViewWillDisappear (bool animated)
{
        base.ViewWillDisappear (animated);
        this.NavigationController.SetNavigationBarHidden (false, animated);
}
```

Now, when we launch the application, removing our unnecessary navigation bar gives us more screen space on the home screen, and when we show a subscreen, it animates on, along with the screen!



Summary

Congratulations, if you've been following these tutorials, you're now nearly through the Getting Started portion!

In this application we covered a number of new tools and practices, including the MVC pattern and how it's utilized in iOS to create multi-screened applications. We introduced the UINavigationController, and we touched on several important View life cycle events.

In our next Getting Started tutorial, we're going to jump over to the iPad and create our first MonoTouch iPad application, and then learn how to make applications that target both the iPhone and iPad!