Hello, iPhone

A First MonoTouch Application
Getting Started – Tutorial 3

Xamarin Inc.

BRIEF

Once you've run through the MonoTouch Installation, you're ready to create your first iOS application. This article is the third in a series of Getting Started tutorials and walks through creating a basic MonoTouch application for iPhone. Along the way, we'll take a look at the MonoTouch IDE of choice, MonoDevelop (MD), then cover creating user interfaces with Xcode 4 and Interface Builder (IB). Next we'll look at connecting the work that we did in Interface Builder and making it available to your code in MonoTouch. Finally, you'll deploy your application and test it.

This article is aimed at developers who are familiar with C# and .NET, but who have never before built a MonoTouch iOS application.

Time to Complete:

2 Hours

Sample Code:

Hello, iPhone

Related Articles:

Getting Started 2 – MonoTouch Installation

Getting Started 4 – MVC in iOS

Debugging MonoTouch

Related Apple Documentation:

Xcode Quick Start Guide

Xcode User Guide

iOS Human Interface Guidelines

iOS Provisioning Portal

Overview

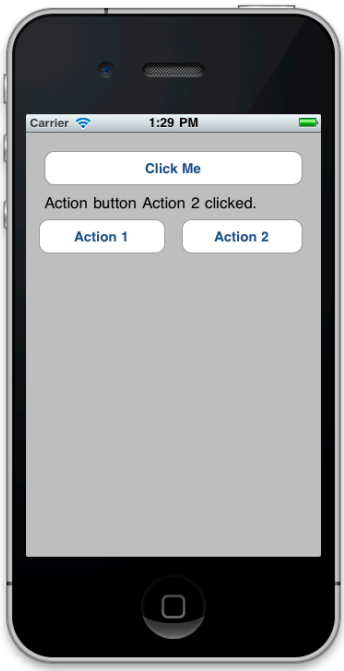
Using MonoTouch, you can create native iOS applications by using the same UI controls that you would if you were writing the application in *Objective-C* and *Xcode*, except you have the flexibility and elegance of a modern language (*C#*), the power of the *.NET Base Class Library* (*BCL*), and a first-class IDE (*MonoDevelop*) at your fingertips.

Additionally, MonoTouch integrates directly with Xcode so that you can use the integrated *Interface Builder* (*IB*) to create your application's user interface.

In this tutorial, we're going to walk through creating a simple application from start to finish. We'll cover the following items:

- **MonoDevelop (MD)** – Introduction to the Xamarin IDE (MonoDevelop) and how to create MonoTouch applications with it**.**
- **Anatomy of a MonoTouch Application** – What a MonoTouch application consists of.
- **Xcode's Interface Builder** – How to use Xcode's Interface Builder to define your application's user interface (UI).
- **Outlets and Actions** – How to use Outlets and Actions to wire up controls in the UI.
- **Deployment** – How to deploy your application to the iOS device simulator, as well as to a real device like an iPhone.

The following is a screenshot of the application that we're going to build, running in the iOS simulator:



Before we start the tutorial, let's review a few requirements first.

Requirements

In order to complete this tutorial, you'll need to have the latest iOS SDK with Xcode 4 installed, as well as the Mono Runtime, and MonoTouch. You can find instructions on how to get all this installed at the MonoTouch Installation tutorial.

In order to get the iOS SDK, you need to be a member of Apple's iOS Developer Program. For more information, see developer.apple.com.

Because you'll need the iOS SDK, you'll also need an Apple Mac computer running OS X Lion.

Additionally, if you want to deploy to a device, you'll need to make sure it's provisioned and have a non-evaluation version of MonoTouch installed. The evaluation version of MonoTouch does not allow deployment to devices. For more information about how to provision a device, see the Managing Devices and Digital Identities Apple Documentation.

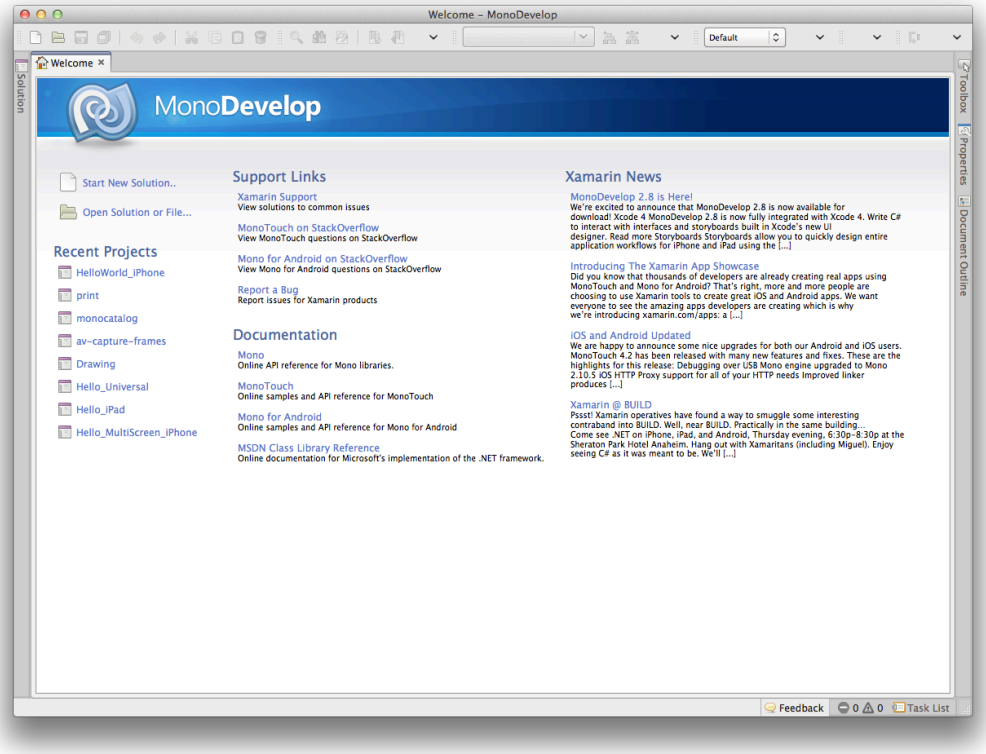
Introduction to MonoDevelop

Nearly all Xamarin MonoTouch tutorials are based on using MonoDevelop as the Integrated Development Environment (IDE) of choice. It's possible to develop for MonoTouch without using MonoDevelop, but MonoDevelop is integrated with MonoTouch in sophisticated ways that make development tremendously easier when you use both products together.

MonoDevelop is a free, open-source IDE, and is very similar to other IDEs such as Eclipse or Visual Studio.

Let's begin by launching MonoDevelop. You can find it in either your Applications directory, or via a Spotlight search for "MonoDevelop."

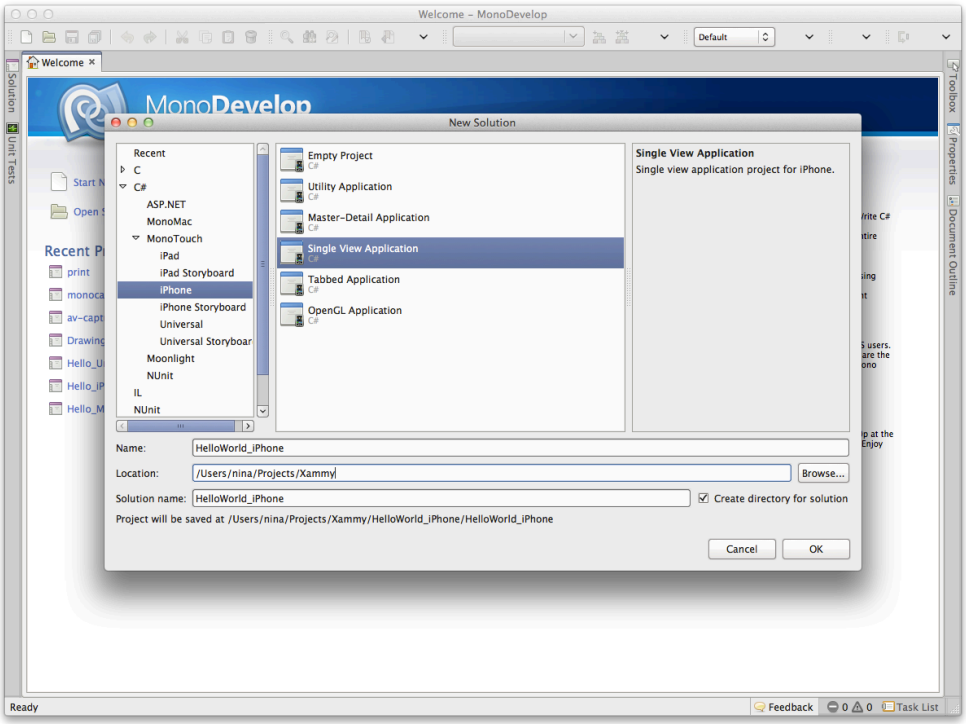
MonoDevelop should open up and look something like the following:



Let's jump right in and begin our first MonoTouch iPhone application.

Creating a New MonoTouch iPhone Project

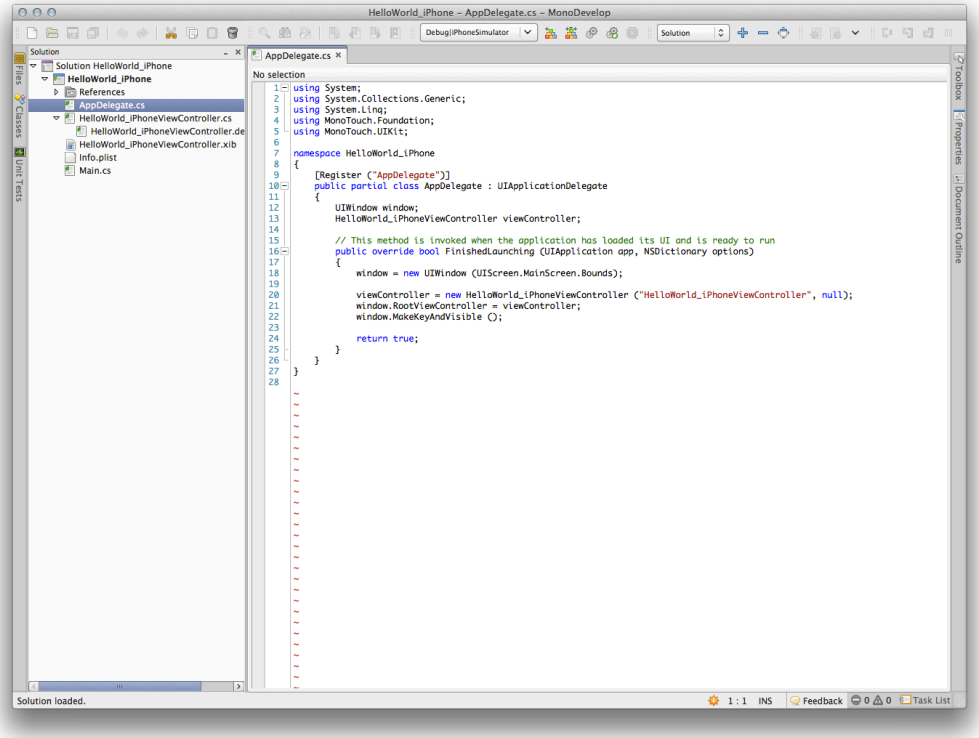
From the MonoDevelop Home screen, choose **Start a New Solution**:



In the left-hand pane, choose **C# > MonoTouch > iPhone**, and then, in the center pane, select **MonoTouch Single View Application** template. This will create a new MonoTouch iPhone application that has a single screen.

Let's name it `HelloWorld_iPhone`. Choose a location where you'd like the solution to reside, and click **OK**.

MonoDevelop will create a new MonoTouch application and solution that looks something like the following:



This should be pretty familiar territory if you've used an IDE before. There is a Solution Explorer "pad" that shows all the files in the solution, and a code editor pad that allows you to view and edit the selected file.

MonoDevelop uses *Solutions* and *Projects*, the exact same way that Visual Studio does. A solution is a container that can hold one or more projects; projects can include applications, supporting libraries, test applications, etc. In this case, MonoDevelop has created both a solution and an application project for you. If you wanted to, you could create code library projects that the application project uses, just as you would if you were building a standard .NET application.

The Project

Let's take a look at the files in the project:

- **Main.cs** – This contains the main entry point of the application.
- **AppDelegate.cs** – This file contains the main application class that is responsible for listening to events from the operating system.
- **HelloWorld_iPhoneViewController.cs** – This contains the class that controls the life cycle of our main screen.
- **HelloWorld_iPhoneViewController.designer.cs** – This file contains plumbing code that helps you integrate with the main screen's user interface.

➔ **HelloWorld_iPhoneViewController.xib** – This is the UI for the main screen. Xib files (also referred to as Nibs for legacy reasons) are XML files that define views.

➔ **Info.plist** – This file contains application properties such as the application name, icons, etc.

Let's take a quick look through some of these files. We'll explore them in more detail later, and in other tutorials, but it's a good idea to understand their basics now.

MAIN.CS

The Main.cs file is very simple. It contains a static `Main` method which creates a new MonoTouch application instance and passes the name of the class that will handle OS events, which in our case is the `AppDelegate` class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using MonoTouch.Foundation;
using MonoTouch.UIKit;

namespace HelloWorld_iPhone
{
        public class Application
        {
                static void Main (string[] args)
                {
                        UIApplication.Main (args, null, "AppDelegate");
                }
        }
}
```

APPDELEGATE.CS

The AppDelegate.cs file contains our `AppDelegate` class, which is responsible for creating our window and listening to OS events:

```
using System;
using System.Collections.Generic;
using System.Linq;
using MonoTouch.Foundation;
using MonoTouch.UIKit;

namespace HelloWorld_iPhone
{
        [Register ("AppDelegate")]
        public partial class AppDelegate : UIApplicationDelegate
        {
                UIWindow window;
                HelloWorld_iPhoneViewController viewController;

                // This method is invoked when the application has loaded
                // its UI and is ready to run
                public override bool FinishedLaunching (UIApplication app,
                        NSDictionary options)
                {
                        window = new UIWindow (UIScreen.MainScreen.Bounds);

                        viewController = new HelloWorld_iPhoneViewController
                                ("HelloWorld_iPhoneViewController", null);
```

```
                        window.RootViewController = viewController;
                        window.MakeKeyAndVisible ();

                        return true;
                }
        }
}
```

This code is probably unfamiliar unless you've built an iOS application before, but it's fairly simple. Let's examine the important lines.

First, let's take a look at the two class-level variable declarations:

```
UIWindow window;
HelloWorld_iPhoneViewController viewController;
```

The UIWindow declaration represents the actual application window. The second declaration is of type HelloWorld_iPhoneViewController, which is responsible for displaying a view in our window. In iOS applications, you only ever have one window, but you can have many screens. Each screen is actually a view, and it has a view controller that is responsible for the view's life cycle, such as showing the view.

This pattern is known as the Model View Controller (MVC) pattern. We'll examine this in more depth in the next tutorial, when we create an application with multiple screens.

Next, we have the `FinishedLaunching` method. This method runs after the application has been instantiated, and it's responsible for actually creating the application window and beginning the process of displaying the view in it. It's important to note that this method has 10 seconds to return; otherwise iOS will terminate the application. Therefore, you should keep this method as simple and spare as possible.

The first line of this method actually creates the window at the exact size of the screen:

```
window = new UIWindow (UIScreen.MainScreen.Bounds);
```

Next, the code creates a new view controller, and tells the window that it's the top most (root) view controller (and it is therefore the main screen):

```
viewController = new HelloWorld_iPhoneViewController
        ("HelloWorld_iPhoneViewController", null);
window.RootViewController = viewController;
```

Finally, we tell the iOS that this window should be visible and then return out of the method:

```
window.MakeKeyAndVisible ();
return true;
```

The `MakeKey` portion of this method is inherited from the OS X programming model in which you need to tell the OS which window should have focus in a multiple window environment.

HELLOWORLD_IPHONEVIEWCONTROLLER.CS

As we've already seen, the `HelloWorld_iPhoneViewController` class is our main view controller. That means it's responsible for the life cycle of the main view (screen). We're going to examine this in detail later, so we can skip any more detail about it for now, other than to point out that it has several view life cycle events that are important, including `ViewDidLoad`, `ViewDidUnload`, and others:

```
using MonoTouch.UIKit;
using System.Drawing;
using System;
using MonoTouch.Foundation;

namespace HelloWorld_iPhone
{
        public partial class HelloWorld_iPhoneViewController
```

```
                : UIViewController
        {
                public HelloWorld_iPhoneViewController (string nibName
                        , NSBundle bundle) : base (nibName, bundle)
                {
                }

                public override void DidReceiveMemoryWarning ()
                {
                        // Releases the view if it doesn't have a superview.
                        base.DidReceiveMemoryWarning ();

                        // Release any cached data, images, etc that aren't
                        // in use.
                }

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

                        //any additional setup after loading the view,
                        //typically from a nib.
                }

                // ViewDidUnload and ShouldAutorotateToInterfaceOrientation
    are deprecated in iOS 6.
                // The code below is no longer necessary and can be removed.
                // A future version of MonoDevelop will remove them from the
    template.
                /*public override void ViewDidUnload ()
                {
                        base.ViewDidUnload ();

                        // Release any retained subviews of the main view.
                        // e.g. this.myOutlet = null;
                }

                public override bool ShouldAutorotateToInterfaceOrientation
                        (UIInterfaceOrientation toInterfaceOrientation)
                {
                        // Return true for supported orientations
                        return (toInterfaceOrientation !=
                                UIInterfaceOrientation.PortraitUpsideDown);
                }*/
        }
}
```

HELLOWORLD_IPHONEVIEWCONTROLLER.DESIGNER.CS

The designer file for our `HelloWorld_iPhoneViewController` class is empty right now, but it will be automatically populated by MonoDevelop as we create our UI:

```
//
// This file has been generated automatically by MonoDevelop to store
outlets and
// actions made in the Xcode designer. If it is removed, they will be lost.
```

```
// Manual changes to this file may not be handled correctly.
//
using MonoTouch.Foundation;

namespace HelloWorld_iPhone
{
        [Register ("HelloWorld_iPhoneViewController")]
        partial class HelloWorld_iPhoneViewController
        {
        }
}
```

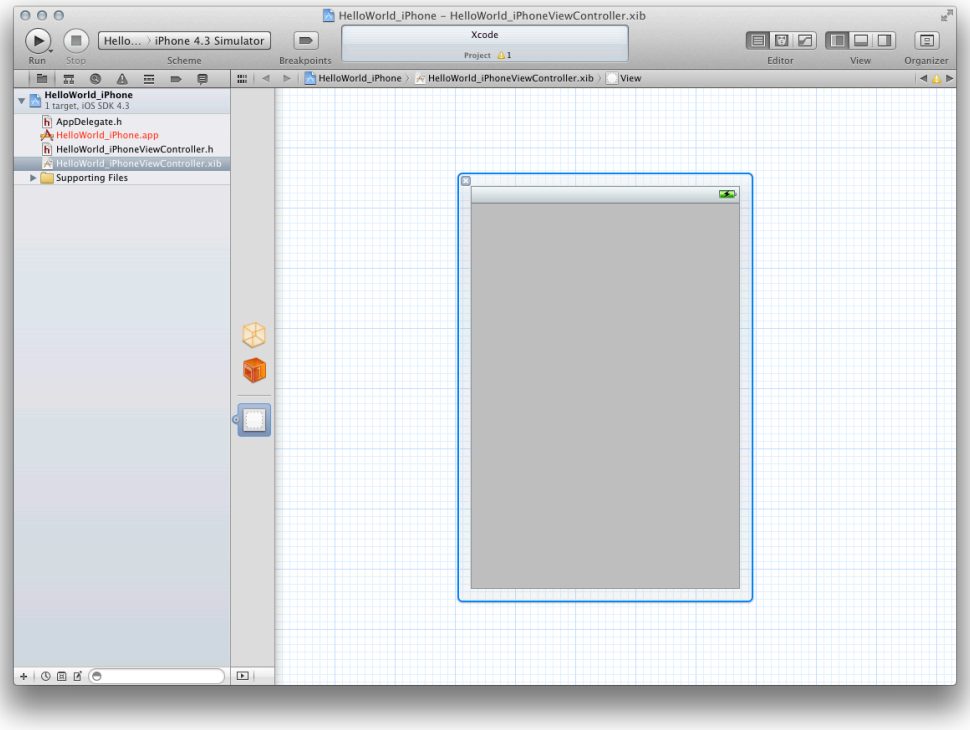
Now that we have created our MonoTouch application and we have a basic understanding of its components, let's jump over to Xcode and create our UI.

Introduction to Xcode 4 and Interface Builder

As part of Xcode, Apple has created a tool called Interface Builder (IB), which allows you to create your UI visually in a designer. MonoTouch integrates fluently with IB, allowing you to create your UI with the same tools that Objective-C users do.

It's important to note that you don't have to use IB to create your UI; you can also build it programmatically, which we'll explore in a later tutorial.

Let's go ahead and walk through how to use IB to define our UI. Double-click on the **HelloWorld_iPhoneViewController.xib** file in MonoDevelop. This should launch Xcode and look something like the following:

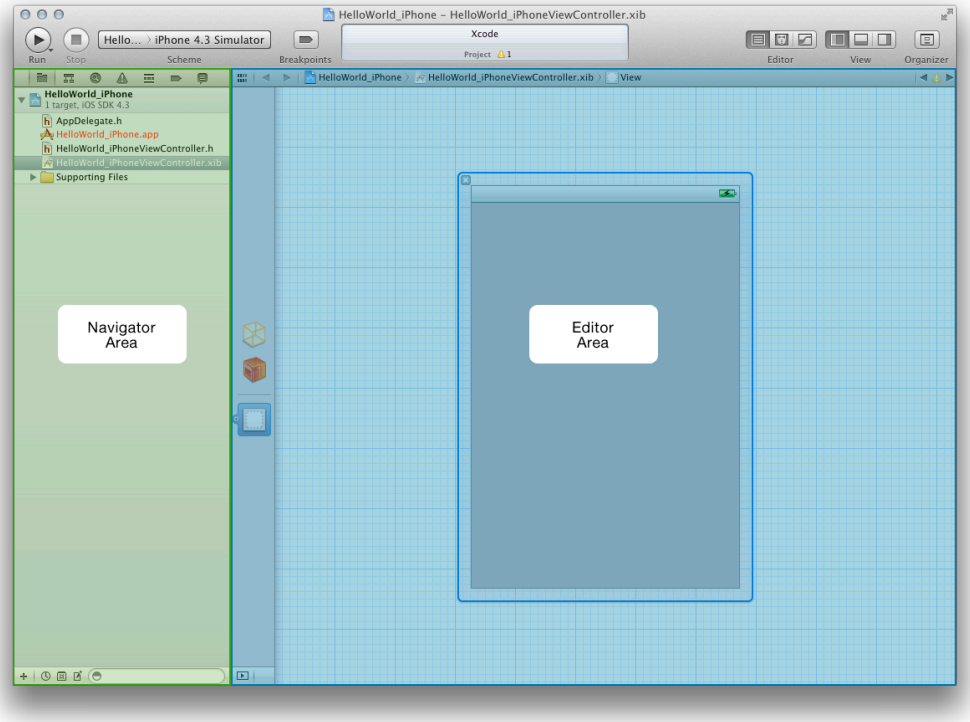


If Xcode doesn't open up, you can right-click on the file and choose **Open With : Xcode**.

Let's do a quick overview of Xcode to orient ourselves.

Components of Xcode

When you open a Xib in Xcode from MD, Xcode opens with the *Navigator Area* on the left and the *Editor Area* on the right:

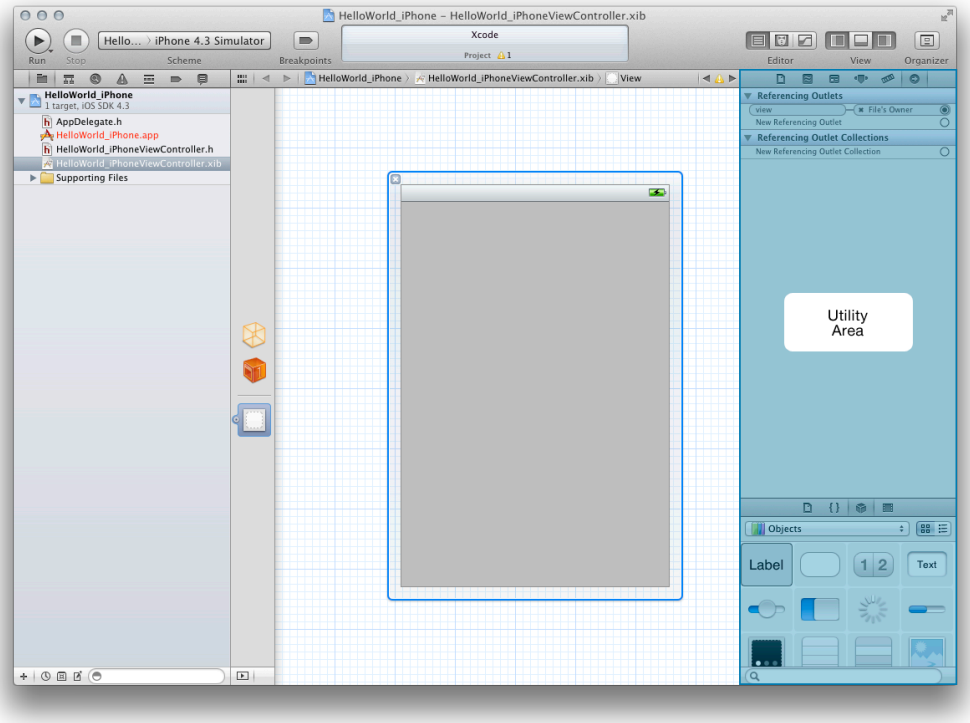


When a .xib file is open, the Editor Area is actually IB.

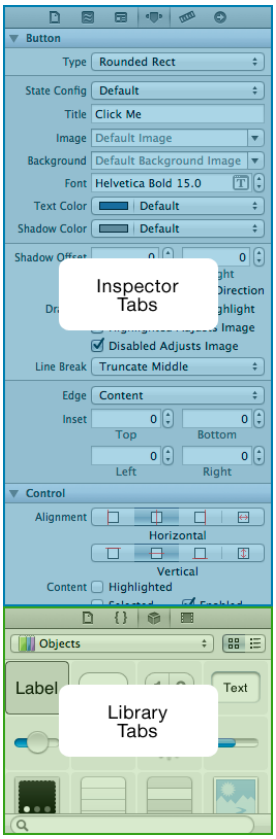
In previous versions of Xcode, IB was a separate application. Apple is still working out some of the integration issues with this new combined version.

Right now, IB isn't terribly useful until we show the *Utilities Area*. To display the Utilities Area, click the far-right button in the View section of the toolbar:

The Utilities Area should now display, exposing tools that will help us create our UI:

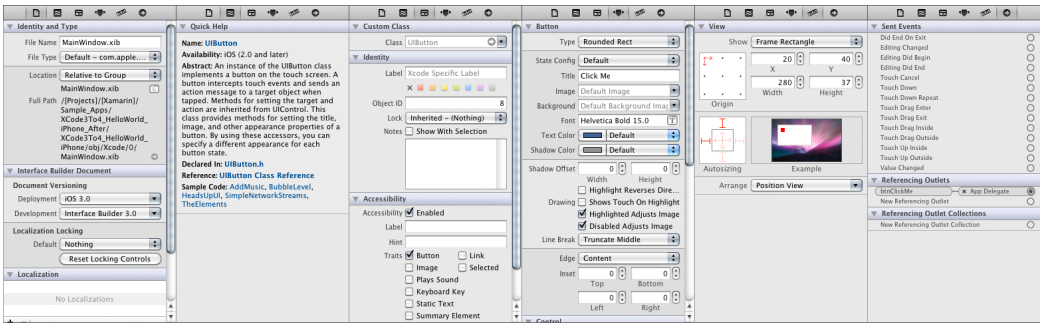


The Utility Area is mostly empty because nothing is selected; however, if you select a control (or the main view), it will populate. The Utility Area is further broken down into two sub-areas, Inspector Tabs and Library Tabs:



In the Library Tabs Area, you can find controls and objects to place into the designer. The Inspector Tabs are kind of like property pages, where you can examine and edit control instances in the designer.

There are 6 different Inspector Tabs, as shown in the following illustration:

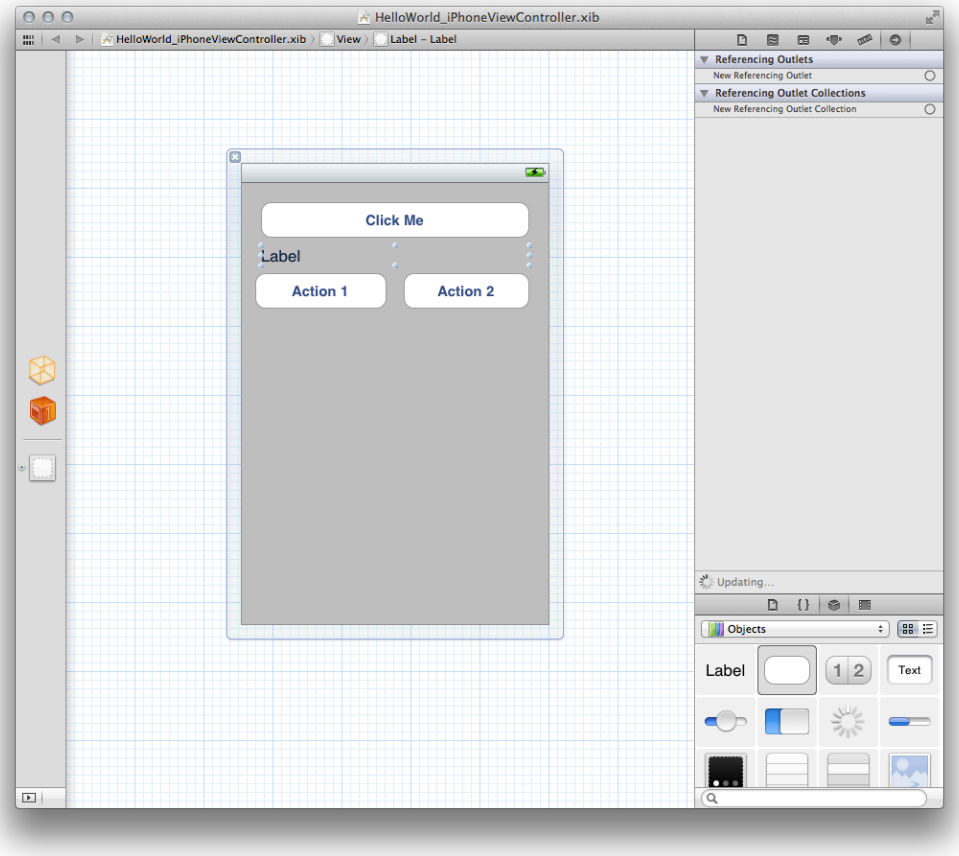


From left-to-right, these tabs are:

- **File Inspector** – New in Interface Builder 4, the File Inspector shows file information, such as the file name and location of the Xib file that is being edited.
- **Quick Help** – Also new in Interface Builder 4, the Quick Help tab is part of Xcode 4's redesigned help system. It provides contextual help based on what is selected in Xcode.
- **Identity Inspector** – The Identity Inspector provides information about the selected control/view.
- **Attributes Inspector** – The Attributes Inspector allows you to customize various attributes of the selected control/view.
- **Size Inspector** – The Size Inspector allows you to control the size and resizing behavior of the selected control/view.
- **Connections Inspector** – The Connections Inspector shows the Outlet and Action connections of the selected controls. We'll examine Outlets and Actions in just a moment.

Creating the Interface

Now that we're familiar with the Xcode IDE and IB, let's actually use IB to create the UI of our main view. We're going to use IB to create the following:



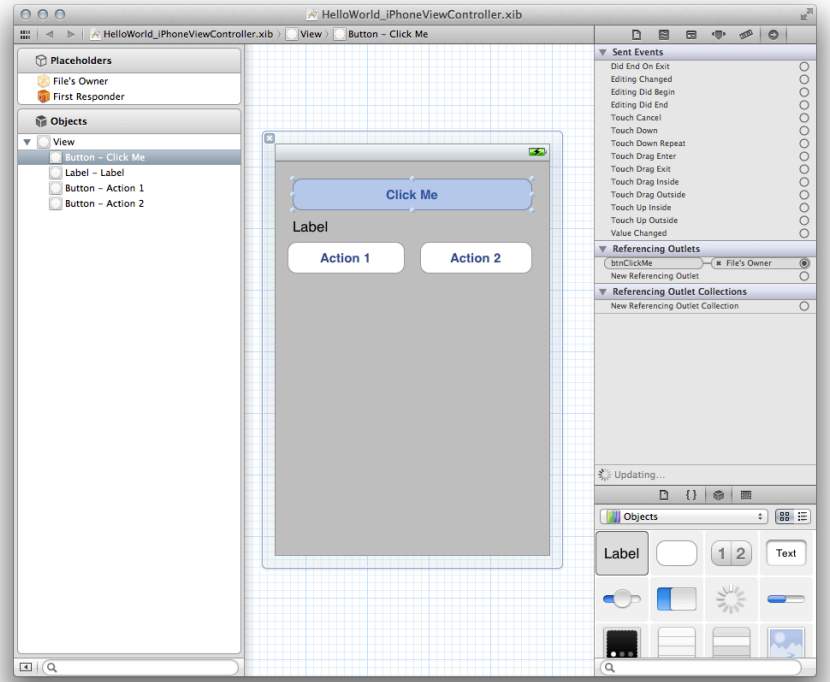
To create this UI:

1. Drag three buttons and a label to the designer from the third tab of the Library.
2. To resize the controls, select them and then pull on their resize handles.
3. Double-click on the buttons to set their title text.
4. Make the label nearly as wide as the view. This will let us update the label with text when the buttons are clicked.

As you're resizing and moving controls around, you'll notice that IB gives you helpful snap hints that are based on Apple's Human Interface Guidelines (HIG). These guidelines will help you create high quality applications that will have a familiar look and feel for iOS users.

While we have IB open, let's look at one other useful area, the Document Inspector. The Document Inspector is on the left and can be expanded by clicking the **>** arrow button at the bottom of the area:

The Document Inspector shows you all of the items in a tree and allows you to select from them:



If you have a complicated UI, this can be a great alternative to using the designer window.

OK, now that we have created our UI, we need to wire up our Outlets and Actions to code.

Adding Outlets and Actions to the UI

MonoDevelop created a file called **HelloWorld_iPhoneViewController.h** as part of the Xcode project it generated to use the designer. This .h file is a stub file that MonoDevelop created to mirror the Designer.cs file. This is where we'll use Xcode to define our Outlets and Actions. MonoDevelop will then synchronize the changes to this file with the designer file.

OUTLETS + ACTIONS DEFINED

So what are Outlets and Actions? In traditional .NET UI programming, a control in the UI is automatically exposed as a property when it's added. Things work differently in iOS (and in OS X programming, for that matter). Simply adding a control to a view doesn't make it accessible to code. In order to access our controls from code, Apple gives us two options:

- **Outlets** – Outlets are analogous to properties. If you wire up a control to an Outlet, it's exposed to your code via a property, so you can do things like attach event handlers, call methods on it, etc.
- **Actions** – Actions are analogous to the command pattern in WPF. For example, when an Action is performed on a control, say a button click, the control will automatically call a method in your code. Actions are powerful and convenient because you can wire up many controls to the same Action.

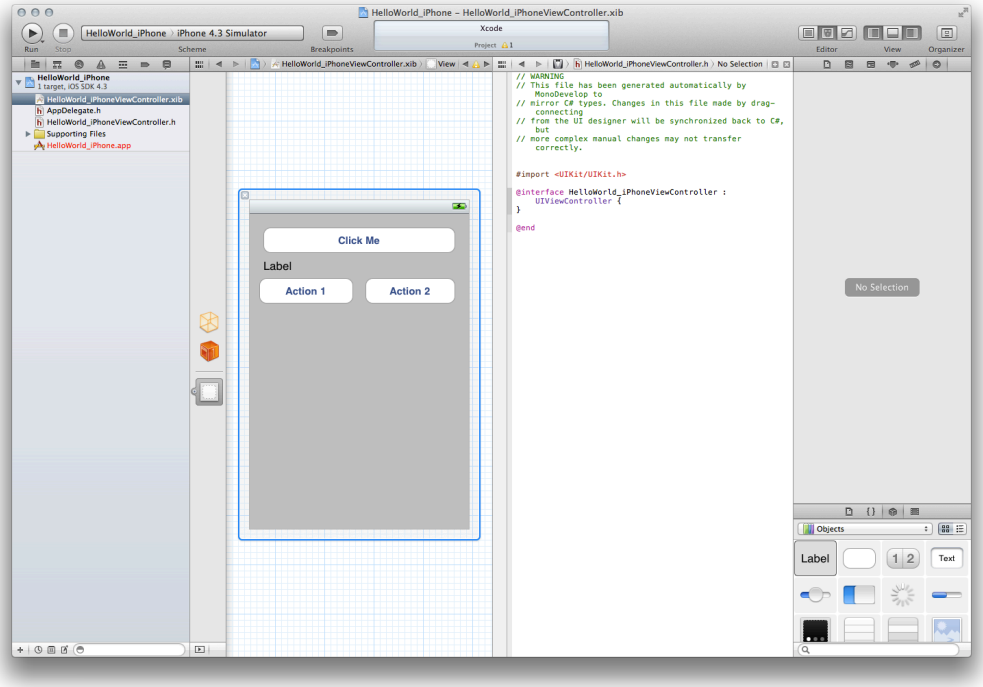
In Xcode 4, Apple has added a new way to create Outlets and Actions directly in code via Control-dragging. More specifically, this means that in order to create an Outlet or Action, you choose which control element you'd like to add an Outlet or Action, hold down the Control button on the keyboard, and drag that control directly into your code.

For MonoTouch developers, this means that you drag into the Objective-C stub files that correspond to the C# file where you want to create the Outlet or Action.

In order to facilitate this, Xcode 4 introduced a split-view in the Editor Area called the *Assistant Editor* that allows two files to be visible at once (the .xib file in the Interface Builder designer, and the code file in the code editor).

In order to view the Assistant Editor split screen, click the middle button of the **Editor** choice buttons in the toolbar:

Xcode will then show the designer and .h file at once:

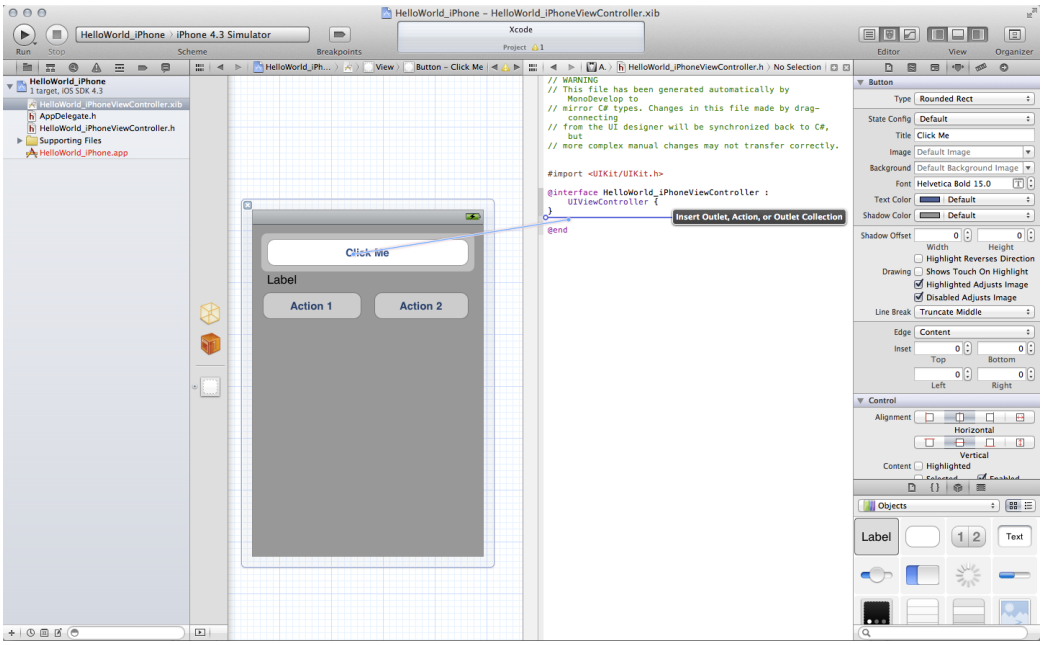


Now we can start wiring up our Outlets and Actions.

ADDING AN OUTLET

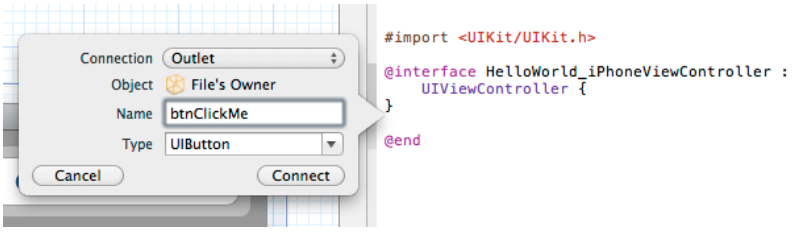
In order to create the Outlet, use the following procedure:

1. Determine for which control you want an Outlet.
2. Hold down the Control key on the keyboard, and then drag *from* the control *to* an empty space in your code file after the `@interface` definition.



You can drag from the Document Inspector as well, which can be helpful if you have a complicated UI with nested controls.

A popover will then show, giving you the option to choose either Outlet or Action. Choose **Outlet** and name it `btnClickMe`:



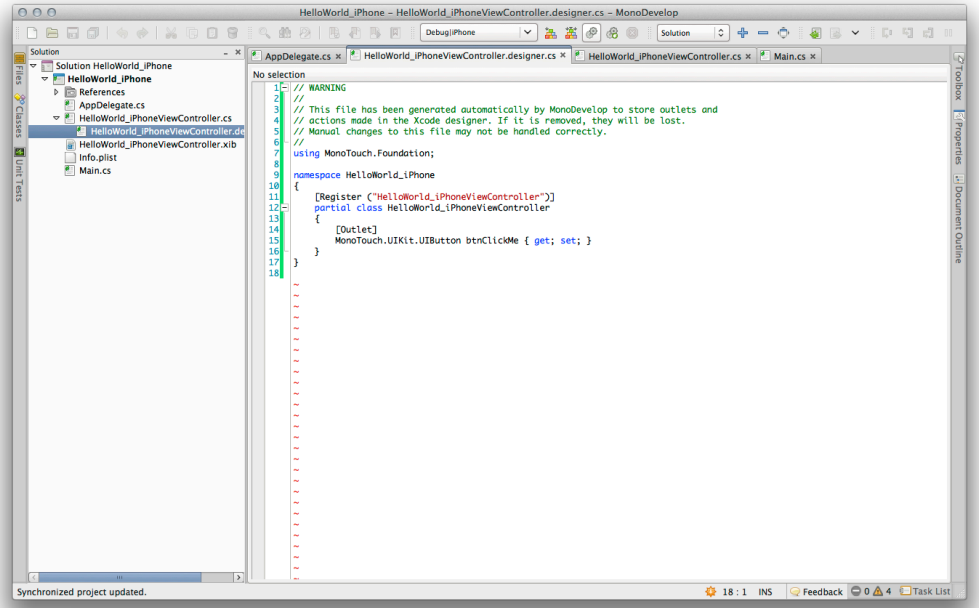
Click **Connect** after you've filled out the form, and Xcode will insert the appropriate code in the .h file:

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

@interface HelloWorld_iPhoneViewController :
    UIViewController {
    UIButton *btnClickMe;
}
@property (nonatomic, retain) IBOutlet UIButton *btnClickMe
    ;

@end
```

Save the file, and then go back to MonoDevelop. If you open the .designer.cs file that corresponds to the .h file that was just modified in Xcode, you'll see your new Outlet exposed as a property:



As you can see, MonoDevelop listens for changes to the .h file, and then automatically synchronizes those changes in the respective Designer.cs file to expose them to your application.

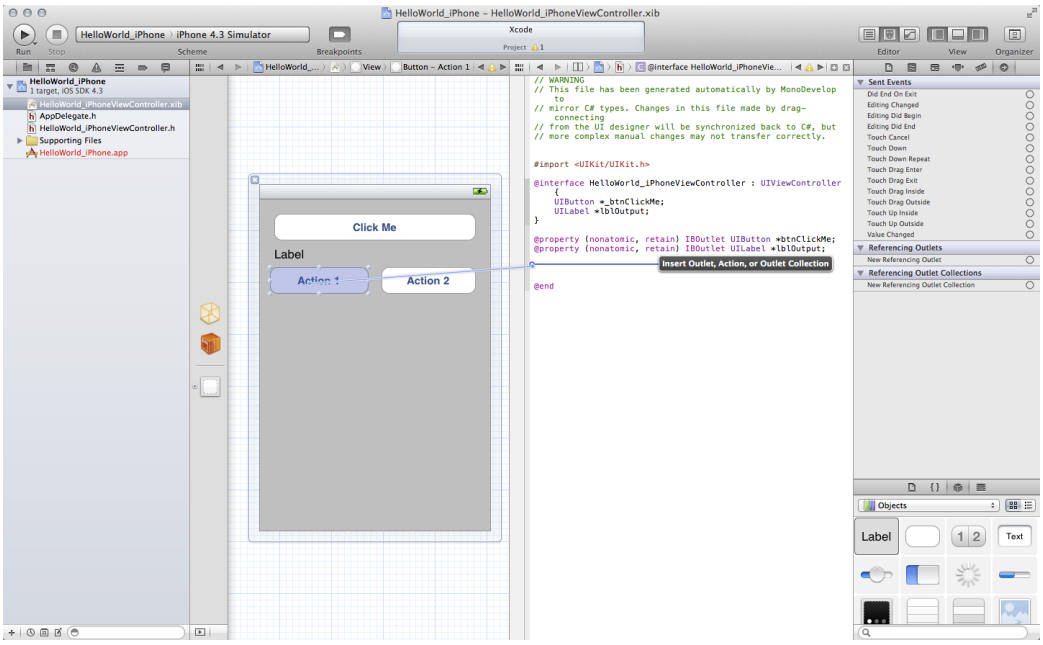
We need to create one more Outlet for our label, so switch back over to Xcode. Create the label's Outlet the same way you did the button's Outlet and name it lblOutput. The .h file should have the following Outlets now defined:

```
@property (nonatomic, retain) IBOutlet UIButton *btnClickMe;
@property (nonatomic, retain) IBOutlet UILabel *lblOutput;
```

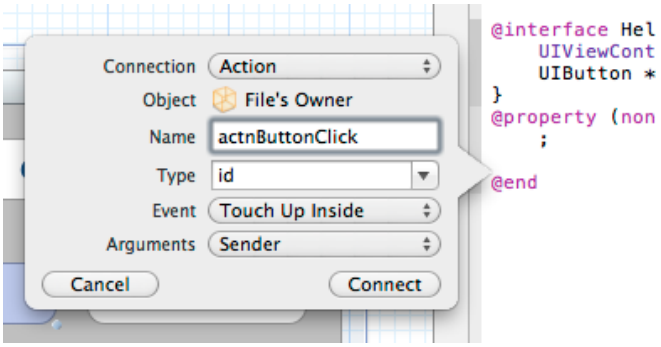
We're going to use these in our MonoTouch application in just a bit, but while we're in Xcode, let's wire up the two Action buttons to an Action.

ADDING AN ACTION

Creating an Action is done the same way as creating an Outlet. Control-drag from the control into the code, as you did before:



This time, though, choose **Action** as the Connection type in the popover menu:



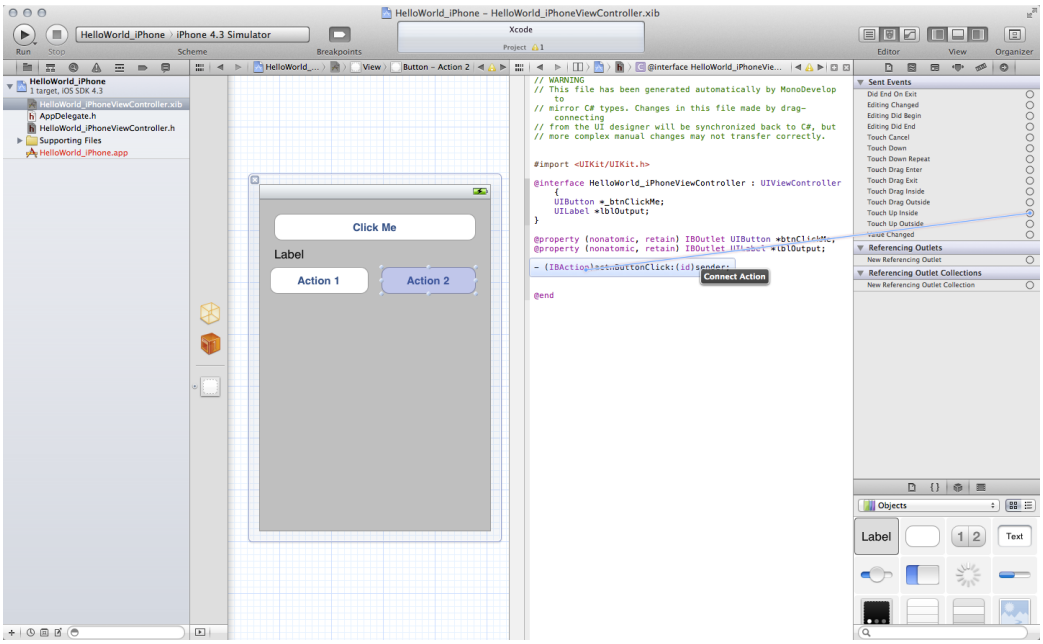
Name the Action and select on which event you'd like the Action to be called. In this case, we're going to use **Touch Up Inside**.

We use the TouchUpInside event because it doesn't get fired until the user releases the button with their finger. This allows the user to cancel an accidental touch by sliding their finger off the button before releasing it. At first it may seem more logical to use TouchDownInside; however, this will raise the event as soon as a button is touched, and doesn't allow for cancellation.

You can also strongly-type the type controls that can call this Action by changing the **Type** from `id` to a particular control type. If you don't strongly type it, you'll have to cast the sender object to its appropriate type when you want to use it, as we'll see in a moment. If you do strongly type this control, then only that one type of control can call this Action.

You may want to wire up multiple items to the same action (which is the entire point of using an Action, as opposed to an Outlet). In our UI, we have two Action buttons so that we can illustrate this at work.

To wire up a control to an existing Action, instead of Control-dragging to a blank space in the code, you Control-drag to an existing Action declaration in code:



In this case, we Control-Dragged from the Connections Inspector, which allowed us to select the event on which we wanted to call the Action, but we could have just as easily Control-dragged from the control in the designer or in the Document Inspector.

Once your Actions and Outlets are wired up, the final declarations in the .h file should be:

```
@property (nonatomic, retain) IBOutlet UIButton *btnClickMe;
@property (nonatomic, retain) IBOutlet UILabel *lblOutput;
- (IBAction)actnButtonClick:(id)sender;
```

Now that we have them wired up, let's save the files and switch back to MonoDevelop to actually do something interesting with them.

Note: It probably took a long time to create the UI, Outlets, and Actions for our first application, and it may seem like a lot of work, but we've introduced a lot of new concepts and we've spent a lot of time covering new ground. Once you've practiced awhile working with IB, this interface and all its Outlets and Actions can be created in just a couple of minutes.

Writing the Code

OK, now that we're back in MonoDevelop and our UI is all created, let's take a quick look at our Designer.cs file:

```
using MonoTouch.Foundation;

namespace HelloWorld_iPhone
{
        [Register ("HelloWorld_iPhoneViewController")]
        partial class HelloWorld_iPhoneViewController
        {
                [Outlet]
                MonoTouch.UIKit.UIButton btnClickMe { get; set; }

                [Outlet]
                MonoTouch.UIKit.UILabel lblOutput { get; set; }
```

```
                [Action ("actnButtonClick:")]
                partial void actnButtonClick (MonoTouch.Foundation.NSObject
sender);
        }
}
```

As you can see, both our Outlets and our Action (called by two different controls) have been synchronized. Let's write some code that uses our Outlets.

Wiring the Outlets

In our application, every time the first button is clicked, we're going to update our label to show how many times the button has been clicked. In order to accomplish this, we need to do two things. First, we need to create a class-level variable in our `HelloWorld_iPhoneViewController` class to track the number of clicks that have happened:

```
public partial class HelloWorld_iPhoneViewController : UIViewController
{
        protected int _numberOfTimesClicked = 0;
        …
}
```

Next, in our `ViewDidLoad` event, we're going to wire up the `btnClickMe` control's `TouchUpInside` event to update our label:

```
public override void ViewDidLoad ()
{
        base.ViewDidLoad ();
       //any additional setup after loading the view, typically from a nib.
        //---- wire up our click me button
        this.btnClickMe.TouchUpInside += (sender, e) => {
                this._numberOfTimesClicked++;
                this.lblOutput.Text = "Clicked [" +
                        this._numberOfTimesClicked.ToString() + "] times!";
        };
}
```

Here we increment our click count, and then set the `Text` property of the `lblOutput` control to be some text that specifies the number of clicks.

This is possible because both our `btnClickMe` button and our `lblOutput` label are exposed via Outlets and so we can access them as we would a normal control that is exposed as a property.

Next, let's configure our Action.

Wiring the Action

Our Action is actually already wired up as an empty partial method in the designer class. We simply need to add implementation to it.

If you type "partial" in the controller class, MonoDevelop will help us out with auto-complete and will fill out the entire method signature. In the method, we're going to update the label to show the title of the button that was clicked that called the Action:

```
/// <summary>
/// This is our common action handler. Two buttons call this via an action
method.
/// </summary>
partial void actnButtonClick (MonoTouch.Foundation.NSObject sender)
{
        this.lblOutput.Text = "Action button "
```

```
                + ((UIButton)sender).CurrentTitle + " clicked.";
        }
```

That's it! We've created our first MonoTouch application, so now it's time to run it and test it!

Testing the Application

It's time to build and run our application; to see it in action and make sure it runs as expected. We can build and run all in one step, or we can build it without running it.

Let's build it first, just to learn how to build without deploying. Whenever you build the application, you build for a particular target, such as the Device Simulator, or the Actual Device. So the first thing we're going to do is make sure that we're targeting the simulator. In the build toolbar, make sure that **Debug|iPhoneSimulator** is selected:

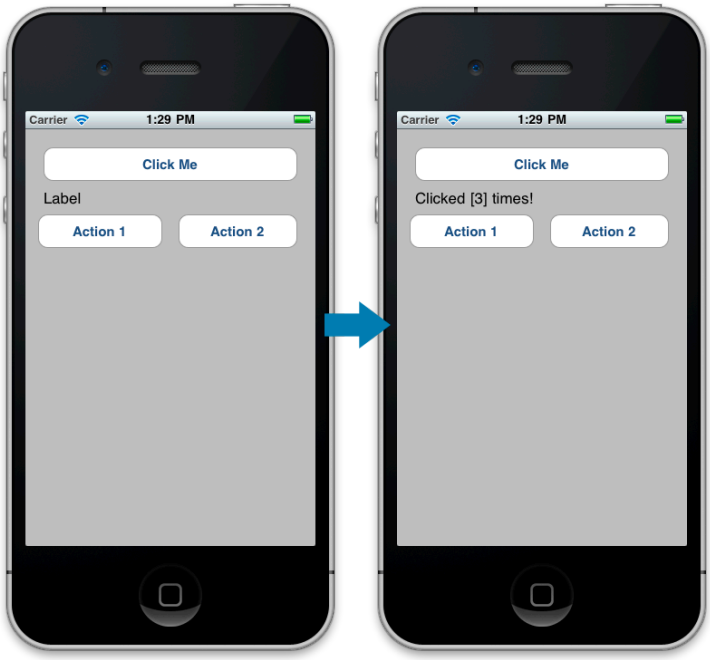
Then, either press **⌘+B**, or from the **Build** menu, choose **Build All**. If there are no errors, you'll see a Build Succeeded message in the status bar of MonoDevelop. If there are errors, review your procedure and make sure that you've followed the steps correctly. Start by confirming that your code (both in Xcode and in MonoDevelop) matches the code in the tutorial.

Deploying to the Device Simulator

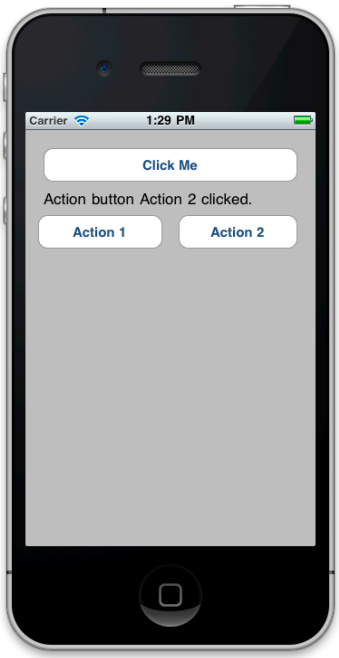
To run the application in the Device Simulator, you have three options:

- **Press ⌘+Enter**
- **From the** Run **menu, choose** Debug
- **Click the** Gear **icon with the green circle from the build toolbar.**

The simulator should launch and the application will run and if you click the first button a few times, you should see something like following:



Clicking one of the Action buttons should give you something similar to the following:



Congrats, you've now built and run your very first MonoTouch application for the iPhone!

CHOOSING WHICH DEVICE TO SIMULATE

By default, the Device Simulator will simulate the standard resolution iPhone (iPhone 3GS and older). However, you can also simulate an iPhone with the Retina Display (iPhone 4+), as well as the iPad.

To change your simulator target, choose one of the following from the **Project : iPhone Simulator Target** menu:

- **Default** – Default launches the iPhone simulator with the regular resolution (320x480) display. This resolution matches the iPhone 3GS and older phones.
- **iPhone Simulator 4.3** – Use this setting if you want to run the application using the iPhone simulator with the Retina Display resolution (640x960). Note: if you want to use the Retina Display iPhone Simulator, in addition to setting this target, you must first launch the simulator and then change it to the Retina Display by selecting Hardware > Device > iPhone (Retina) in the menu. This is, unfortunately, the way Apple designed it, and this same behavior exists even if you're using Xcode and writing in Objective-C.
- **iPad Simulator 4.3** – This setting will launch the iPhone simulator at the iPad resolution (1024x768).

Deploying to the Device

Deploying and testing your application in the Device Simulator is great, but the simulator is just that, a simulator. As such, it doesn't always behave the same way an actual device does. Because of this, it's critical to test your application on a real device early and often.

In order to be able to deploy to a device, you need a properly provisioned device and a non-evaluation version of MonoTouch (the evaluation version only allows deploying to the simulator). Follow the instructions at [point to apple provisioning doc] to make sure your device is provisioned correctly.

Once your device is provisioned, deploying to the device is as simple as deploying to the simulator. Make sure your device is plugged in, and then change the build target in the build toolbar to either **Debug|iPhone** or **Release|iPhone** and deploy via run, as you did before for the simulator.

Debugging

MonoTouch has sophisticated debugging capabilities in both the simulator and the device. For more information, check out the Debugging Tutorial.

Application Name, Icons, and Startup Image

So we now have our first iPhone application, but if we go to the Home screen on the simulator (or device), or view the Settings Application, we see that our application has taken its name from the project name, and the icon is blank:

Additionally, when we launch the app, before our first screen is shown, we see a blank, black screen.

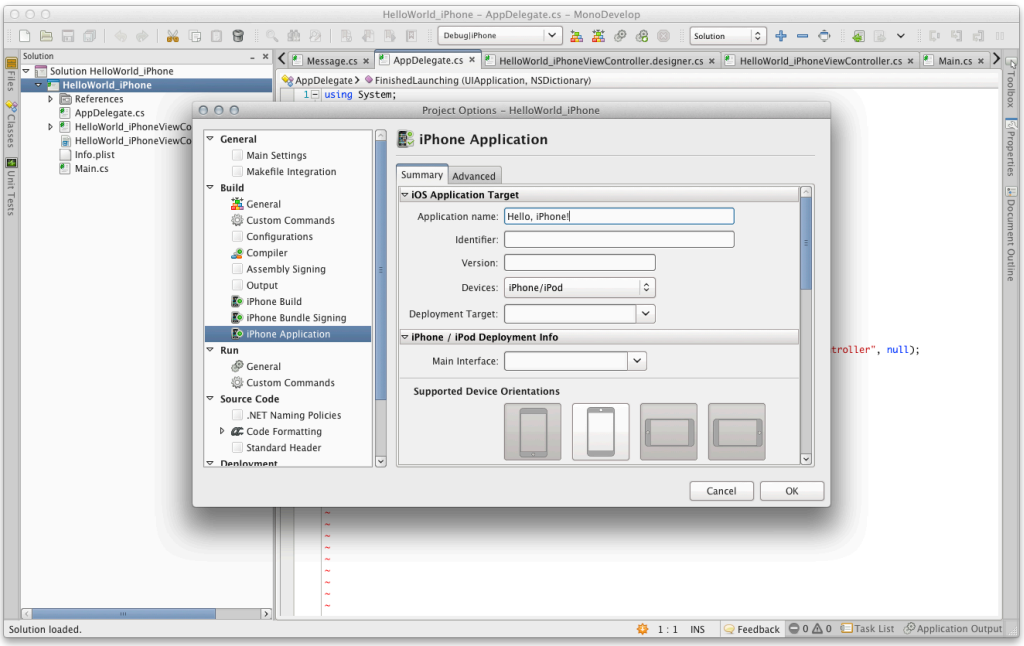
In this section, we're going to look at how to set our application name, its icons, and provide a splash screen image that will be shown while the application is loading.

Application Name

iOS applications use a special XML file called a Property List (.plist) file, named Info.plist (and found in the root of the project), that contains settings and descriptions about the application.

EDITING THE INFO.PLIST FILE

You can edit this file in several different ways. Because it's just an XML file, it can be edited by hand in a text editor, but the easiest way to edit it is to use the .plist editor that is built into MonoDevelop. This editor can either be invoked directly by double-clicking on the Info.plist file itself, or by double-clicking on the project.and choosing **iPhone Application**:



Whether you access the .plist editor directly or via the Project Options dialog, it looks the same.

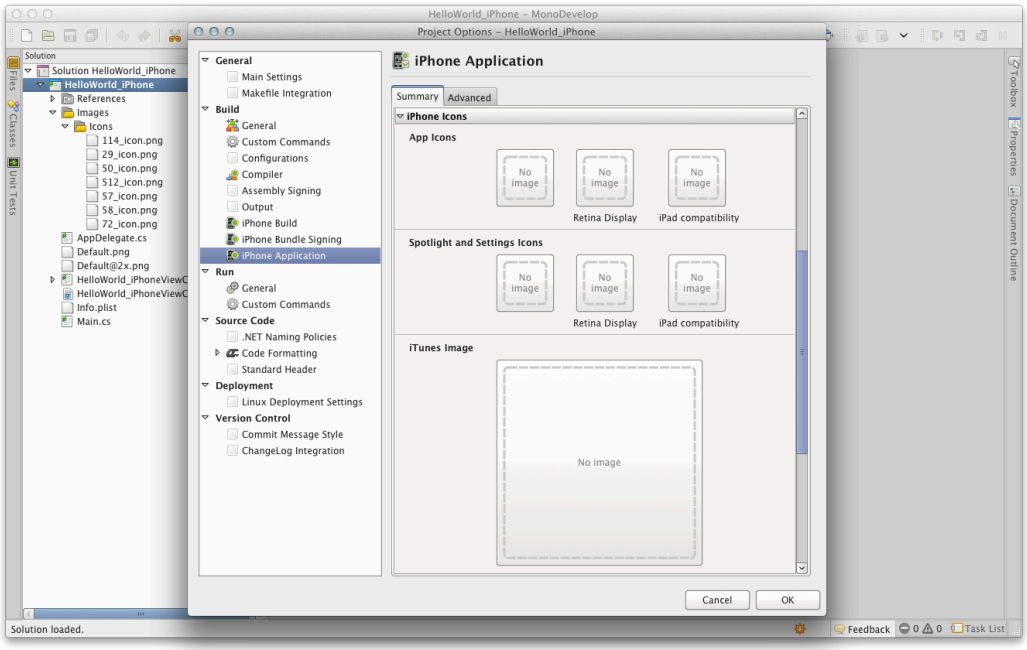
APPLICATION NAME SETTING

The first setting in the editor file is the Application Name. Let's set it to `Hello, iPhone`, and run our application. Now, when we click on the Home button, we see our properly named application:

Now let's take a look at setting the application icons.

Icons

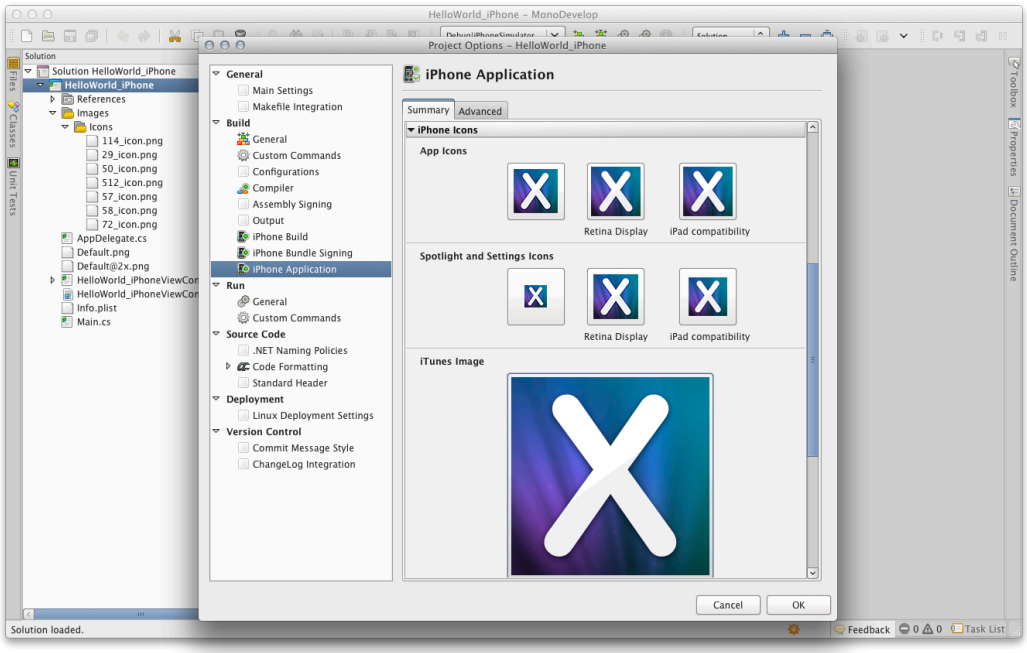
Application icons are also specified in the Info.plist file and can be configured in the same dialog in which we configured our application name:



If you hover over each of the icon blocks, a tooltip will show you what size is needed for each icon. You can also refer to the following table:

Icon Use	Size (in pixels)
Application Icon	Standard: 57x57 Retina Display: 114x114
Spotlight and Settings	Standard: 29x29 Retina Display: 58x58
iTunes Image	1024x1024

To configure an icon, click on one of the icon blocks and browse to the icon location. After you've chosen your icons, they should show up in the editor:



We don't have to worry about applying the glassy effect or rounding the corners of our icons, iOS does that for us.

When we run our application now, we will see our custom icons show up:

For more information about creating icons, see Apple's documentation at [Custom Icon and Image Creation Guidelines.]

Startup Image

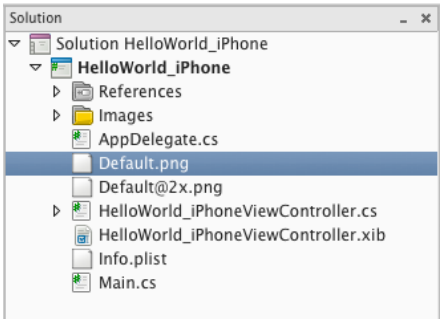
You can provide a splash screen image that iOS will display while your application is launching by adding that screen image file to the root of the project. The following table describes the size and name parameters of the file for regular iPhones (3Gs and older) and for iPhones with the Retina Display (4G and newer) and iPhone 5:

Device	Size (in pixels)	File Name
iPhone	320x480	Default.png
iPhone w/Retina Display	640x960	Default@2x.png
iPhone 5	640x1136	Default-568h@2x.png

The "@2x" suffix specifies that the image is twice the resolution density and iOS will automatically load images with the @2x suffix on Retina Display devices when an image without that suffix is specified.

Note that the device has a case-sensitive file system, but the iOS simulator by default does not (unless you've formatted your hard drive as case-sensitive). This means that if the file case is incorrect (note the leading capital), the image will work correctly on the simulator, but not on the device.

To specify a loading screen image, simply add the files to the root of the project by right-clicking on the project and choosing **Add : Add Files**. When you're done, they should show up in the Solution Navigator:



Now, when you launch the application, your loading screen will be displayed as the application is loading:

Summary

Congratulations! We covered a lot of ground here, but if you followed this tutorial from start to finish, you should now have a solid understanding of the components of a MonoTouch application as well as the tools used to create them. You're also half way through our Getting Started tutorial series!

The application we've built has a couple of limitations, however. For instance, it only has one screen and it's not optimized for the iPad.

In the next tutorial in this Getting Started series, we're going to address that shortcoming and look at the Model, View, Controller (MVC) pattern in iOS and how it's used to create multi-screened applications. In the following tutorial, we'll look at creating applications for the iPad.