Hello, iOS

Evolve Fundamentals Track, Chapter 1

Xamarin Inc.

Overview

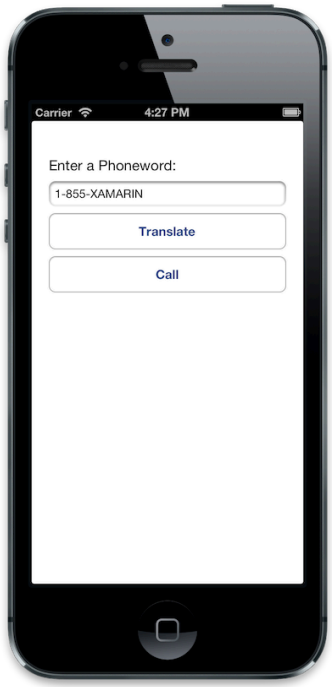
Using Xamarin.iOS, you can create native iOS applications by using the same UI controls that you would use if you were writing the application in *Objective-C* and *Xcode*, except now you have the flexibility and elegance of a modern language (*C#*), the power of the *.NET Base Class Library* (*BCL*),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

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

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

- **Xamarin Studio** – Introduction to *Xamarin Studio* (Xamarin's IDE) and how to use it to create Xamarin.iOS applications.
- **Anatomy of a Xamarin.iOS Application** – The component parts of a Xamarin.iOS application.
- **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.

We're going to create a simple application that translates a phone number with letters in it, i.e. 1.855.XAMARIN, and then allows the user to call it:



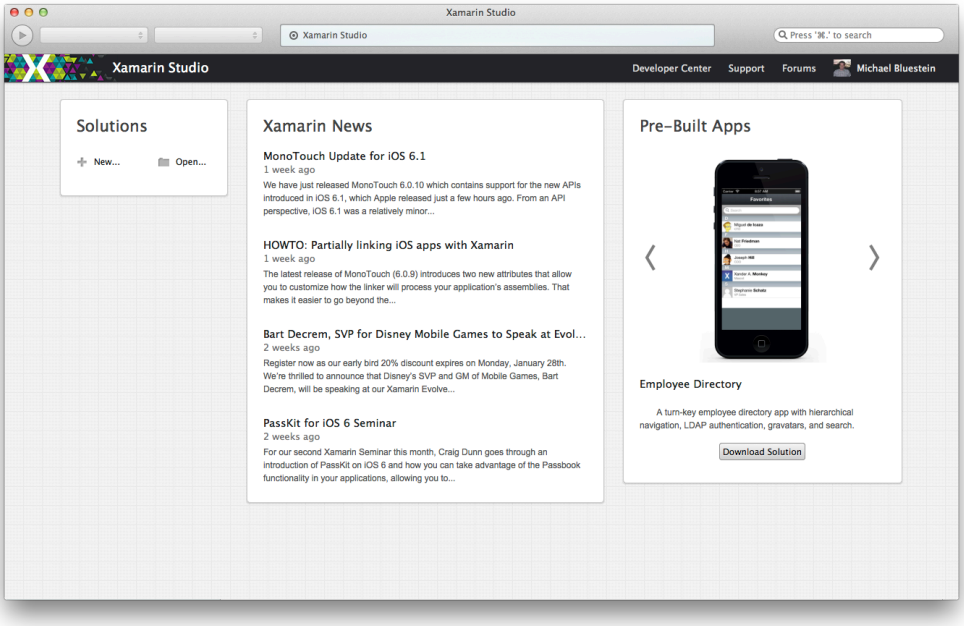
Before we start the tutorial, let's discuss the development environment first.

Introduction to Xamarin Studio

Nearly all Xamarin.iOS tutorials in this course assume that we're using Xamarin Studio as the Integrated Development Environment (IDE) of choice. Xamarin.iOS also supports development with Visual Studio, using Mac as a remote build machine, and we'll address that in a specific chapter.

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

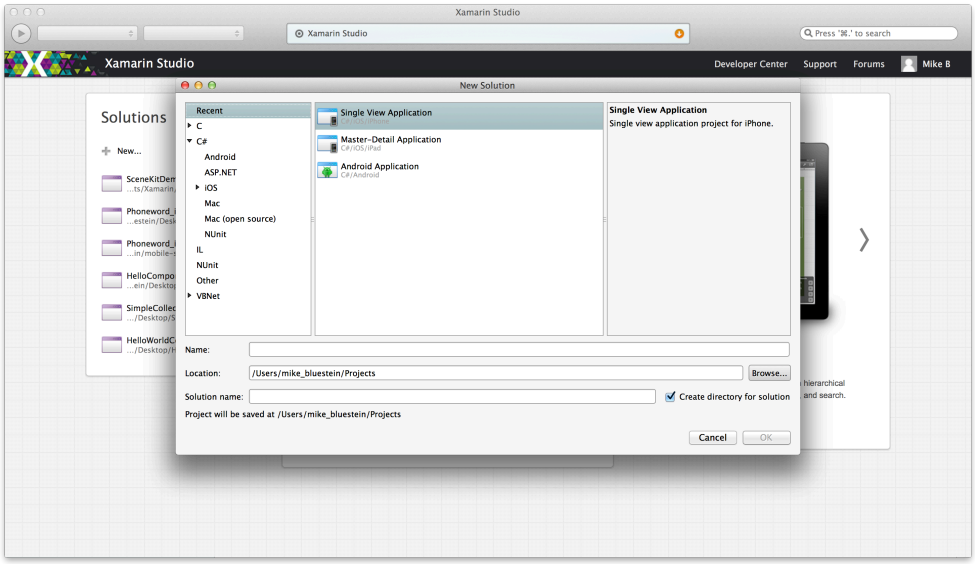
Xamarin Studio should open up and look something like the following:



Let's jump right in and begin our first Xamarin.iOS application.

Creating a New iPhone Project

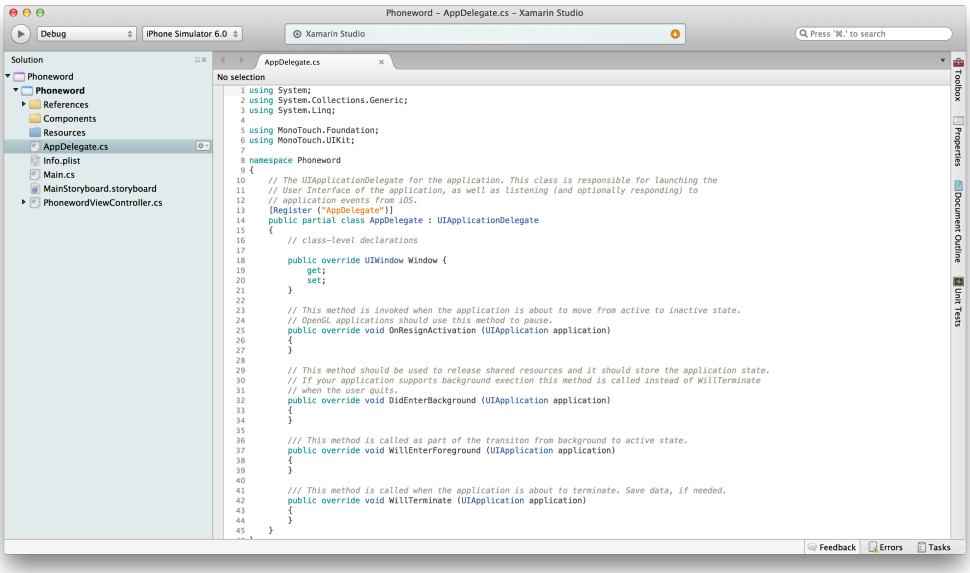
From the Xamarin Studio Welcome screen, click **New** (under the **Solutions** section) to create a new solution. This action opens the **New Solution** dialog shown below:



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

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

Xamarin Studio will create a new Xamarin.iOS application and solution that looks something like the following:



This should be 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.

Xamarin Studio uses *Solutions* and *Projects* in exactly the 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, Xamarin Studio 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

Our new project contains the following files and folders:

- 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.
- Phoneword_iOSViewController.cs – This contains the class that controls the lifecycle of our main screen.
- Phoneword_iOSViewController.designer.cs – This file contains plumbing code that helps you integrate with the main screen's user interface.
- MainStoryboard.storyboard – This file is used to design user interfaces with Xcode. Storyboards define views, controllers, and the navigation mechanism between screens.
- Info.plist – This file contains application properties such as the application name, icons, etc.
- Resources – This folder's content is automatically copied to the root of the application bundle at build time.

Let's take a quick look at each of these now. We'll explore them in more detail later, in other tutorials in this course.

Main.cs

The Main.cs file is very simple. It contains a static `Main` method that creates a new Xamarin.iOS 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 Phoneword_iOS
{
	public class Application
	{
		// This is the main entry point of the application.
		static void Main (string[] args)
		{
			// Delegate class from "AppDelegate"
			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. The following code shows the default implementation for `AppDelegate`:

```
using System;
using System.Collections.Generic;
using System.Linq;

using MonoTouch.Foundation;
using MonoTouch.UIKit;

namespace Phoneword_iOS
{
        // The UIApplicationDelegate for the application. This class is
responsible for launching the
        // User Interface of the application, as well as listening (and
optionally responding) to
        // application events from iOS.
        [Register ("AppDelegate")]
        public partial class AppDelegate : UIApplicationDelegate
        {
                // class-level declarations

                public override UIWindow Window {
                        get;
                        set;
                }

                // You have 17 seconds to return from this method, or iOS
will terminate your application.
                //
                public override bool FinishedLaunching (UIApplication
application, NSDictionary launchOptions)
                {
                        return true;
                }

                // This method is invoked when the application is about to
move from active to inactive state.
                // OpenGL applications should use this method to pause.
                public override void OnResignActivation (UIApplication
application)
                {
                }

                // This method should be used to release shared resources
and it should store the application state.
                // If your application supports background exection this
method is called instead of WillTerminate
                // when the user quits.
```

```
                public override void DidEnterBackground (UIApplication
application)
                {
                }

                /// This method is called as part of the transiton from
background to active state.
                public override void WillEnterForeground (UIApplication
application)
                {
                }

                /// This method is called when the application is about to
terminate. Save data, if needed.
                public override void WillTerminate (UIApplication
application)
                {
                }
        }
}
```

This code is probably unfamiliar unless you've built an iOS application before, but it's fairly simple. First, let's look at the following property declaration:

```
public override UIWindow Window {
        get;
        set;
}
```

The `Window` property represents the actual application window. When the application starts, it will query this property to retrieve a `UIWindow` instance. iOS applications only have one window, but your application can have many screens. Each screen is powered by a view, and it has a view controller that is responsible for the view's lifecycle, such as presenting (showing) the view.

This pattern is known as the Model View Controller (MVC) pattern. We'll examine this in more depth in the next chapter of the course, when we add multiple screens to the application.

The next section in the code is the `FinishedLaunching` method. This method runs after the application has been instantiated and the window has been created. It's important to note that this method has 17 seconds to return; otherwise, iOS will terminate the application. Therefore, this method should return as quickly as possible. Any long-running initializations should be performed on a background thread.

The remaining methods in the `AppDelegate` class deal with the lifecycle of the application, which, for example, supports cases where applications perform work in the background. We'll discuss iOS backgrounding in a later chapter.

Phoneword_iOSViewController.cs

The `Phoneword_iOSViewController` class is our main view controller. That means it's responsible for the lifecycle of the main view. 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 lifecycle events that are important, such as `ViewDidLoad`, `DidReceiveMemoryWarning`, and others.

Phoneword_iOSViewController.Designer.cs

At this point, the designer file for our `Phoneword_iOSViewController` class is empty, but Xamarin Studio will automatically populate it 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 Phoneword_iOS
{
        [Register ("Phoneword_iOSViewController")]
        partial class Phoneword_iOSViewController
        {
                void ReleaseDesignerOutlets ()
                {
                }
        }
}
```

Generally, we don't need to worry about Designer.cs files. They are simply internal plumbing that Xamarin.iOS keeps in synch with any screens designed in Xcode's Interface Builder, as we'll see next.

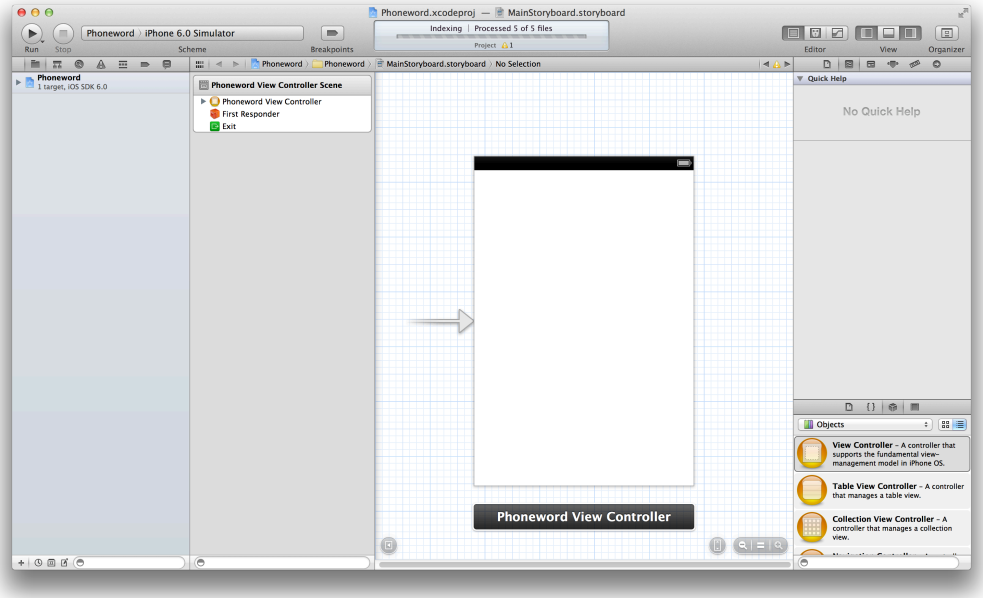
Now that we have created our Xamarin.iOS application and we have a basic understanding of its components, let's jump over to Xcode and create our UI.

Introduction to Xcode and Interface Builder

As part of Xcode, Apple includes a tool called Interface Builder (IB), which allows us to create our UI visually in a designer. Xamarin.iOS integrates fluently with IB, allowing you to create your UI with the same tools that are used by Objective-C programmers.

It's important to note that you don't have to use IB to create your UI; you can also build it programmatically (in code).

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

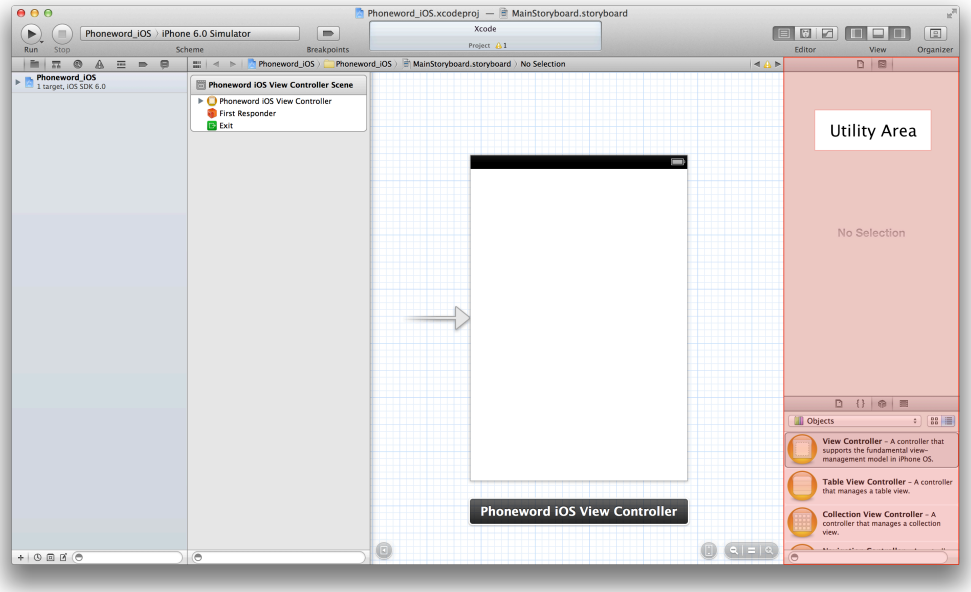


Let's quickly examine Xcode to orient ourselves.

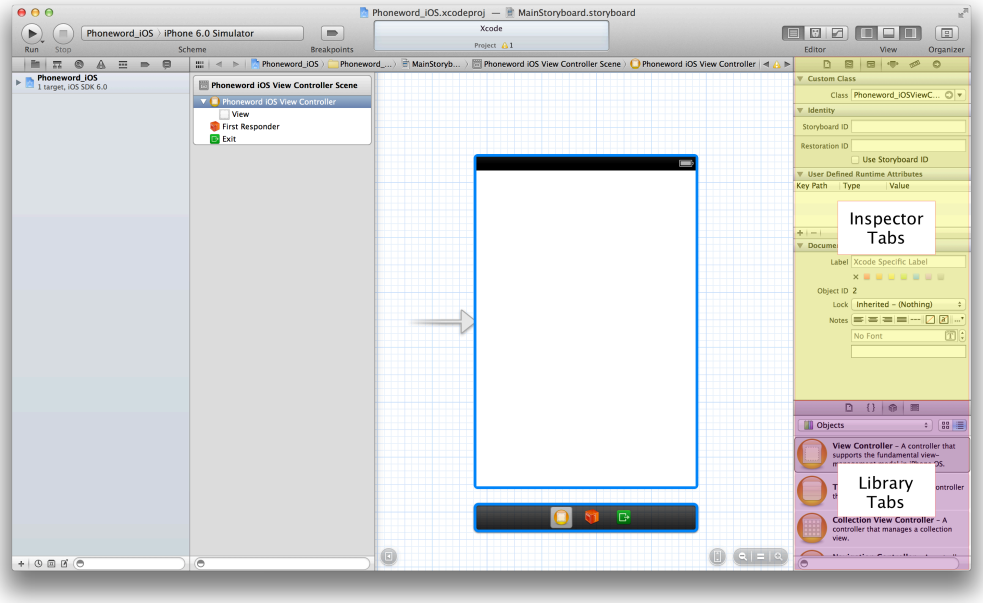
Components of Xcode

When you open a storyboard in Xcode from Xamarin Studio, Xcode opens with the *Navigator Area* on the left and the *Editor Area* on the right. When a .storyboard file is open, the Editor Area is actually IB.

The *Utility Area* should now display, exposing tools that will help us create our UI, as shown below:

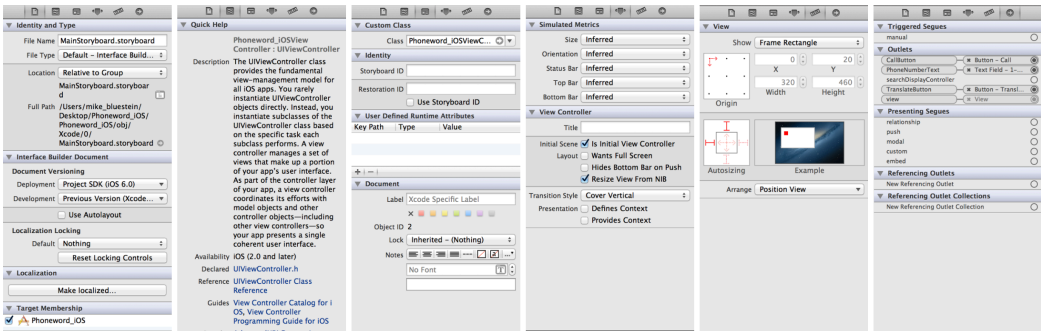


The Utility Area is mostly empty because nothing is selected; however, if you select the controller (or its view), it will populate. The Utility Area is further broken down into two sub-areas, *Inspector Tabs* and *Library Tabs*:



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

There are six different Inspector Tabs, as shown in the following screenshots:



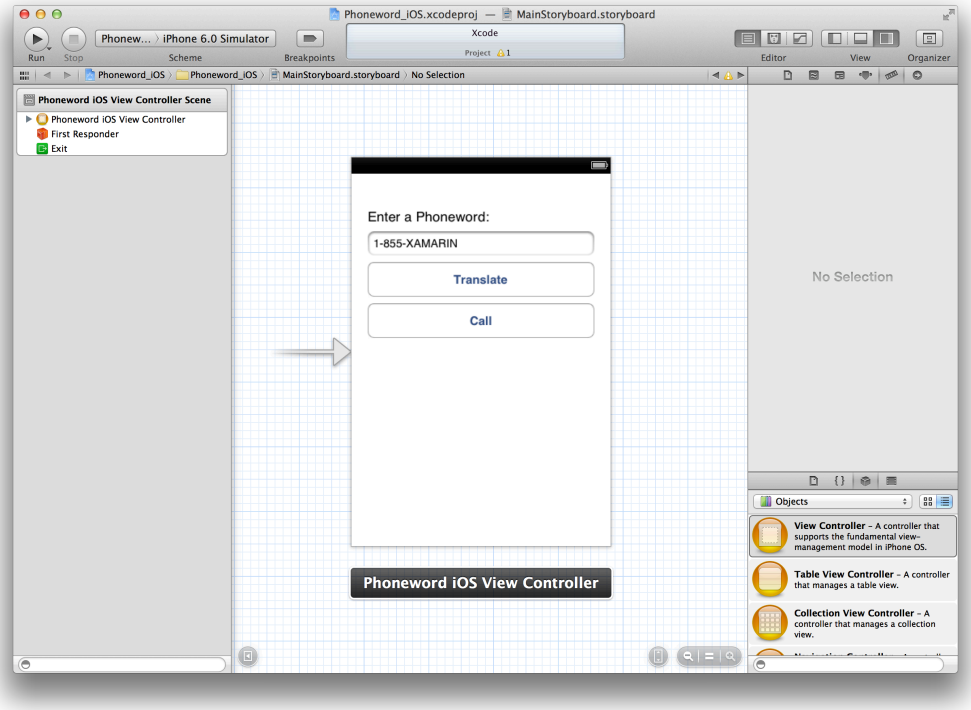
From left-to-right, these tabs are:

- **File Inspector** – Shows file information, such as the file name and location of the Xib file that is being edited.
- **Quick Help** – Provides contextual help, based on what is selected in Xcode.
- **Identity Inspector** – Provides information about the selected control/view.
- **Attributes Inspector** – Allows you to customize various attributes of the selected control/view.

- **Size Inspector** – Allows you to control the size and resizing behavior of the selected control/view.
- **Connections Inspector** – Shows the *Outlet* and *Action* connections of the selected controls. We'll examine these later in the chapter.

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:



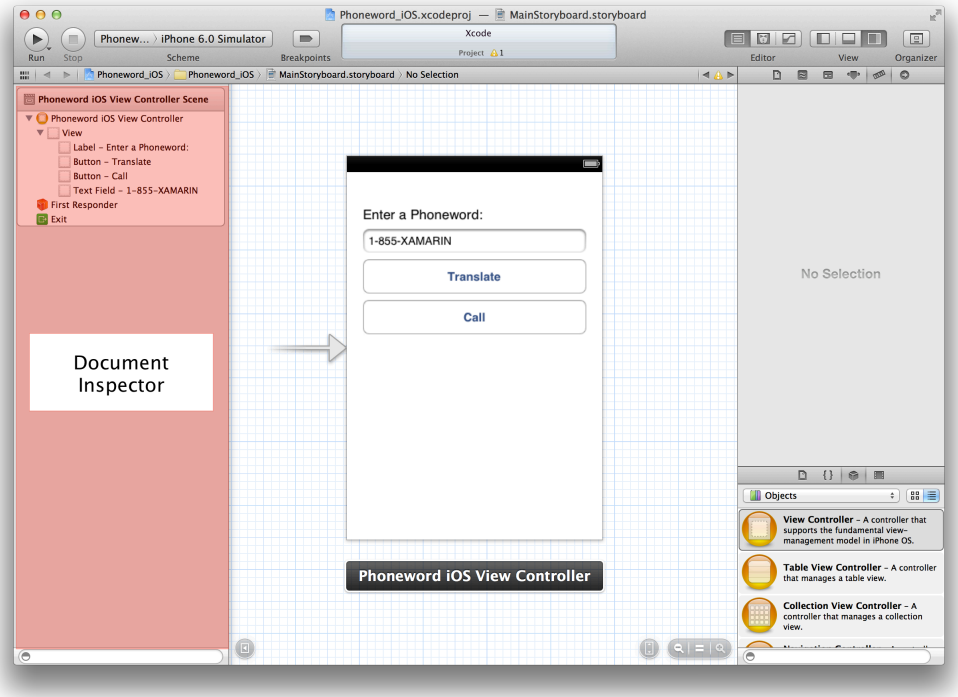
To create this UI:

1. Drag two buttons, a label, and a text field onto 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 the buttons to set their title text.
4. Similarly, double-click the label to set its text, and the text field to set its default value.

As you're resizing and moving controls around, notice that IB provides helpful snap hints that are based on Apple's Human Interface Guidelines (HIG). These guidelines help 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 all of the items in a tree and allows us to select from them:



If our UI is complicated, this can be a great alternative to using the designer window to select controls.

Now that we have created our UI, we need to add Outlets to the controls that need to be accessible via code.

Adding Outlets to the UI

In order to use the designer, Xamarin Studio created a file called `Phoneword_iOSViewController.h` as part of the Xcode project it generated. This header file is an Objective-C stub file that Xamarin Studio created to mirror the Designer.cs file. We'll use Xcode in this file to define our Outlets and Actions. Xamarin Studio 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, when a control is added in the UI, it is automatically exposed as a property. 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 we wire up a control to an Outlet, it's exposed to our code via a property, so we 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 our code. Actions are typically used when we want to wire up many controls to the same Action. However, Actions aren't very commonly used, so we'll skip them in this walkthrough.

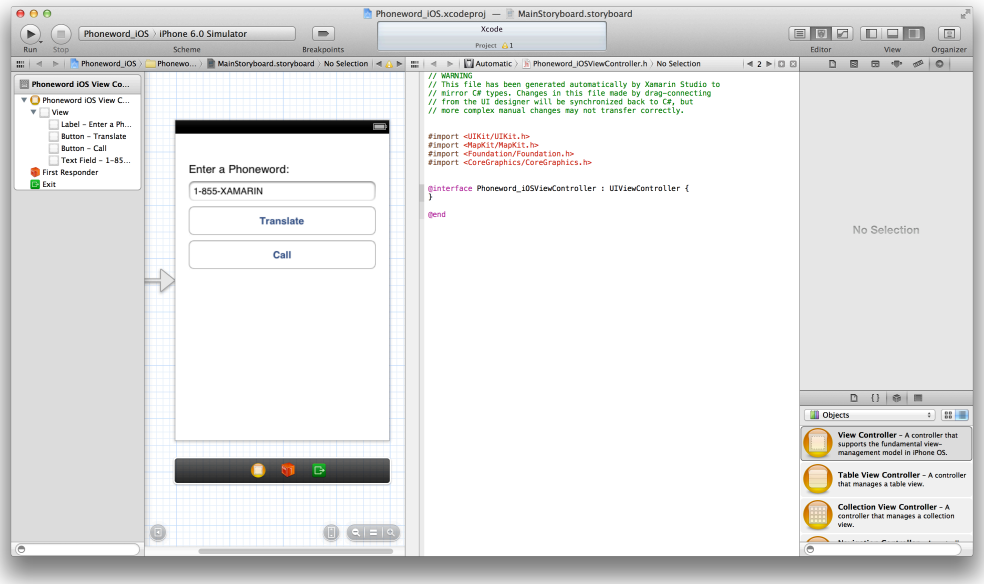
In Xcode, Apple allows us to create Outlets and Actions directly in code via *Control-dragging*. More specifically, this means that in order to create an Outlet or Action, we choose which control element we'd like to add an Outlet or Action to, hold down the **Control** button on the keyboard, and drag that control directly into our code.

For Xamarin.iOS developers, this means that we drag the control into the Objective-C stub file (the .h file) that corresponds to the C# file where we want to create the Outlet or Action.

In order to facilitate this, Xcode includes a split-view in the Editor Area called the *Assistant Editor* that allows two files to be visible at once (the .storyboard 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:

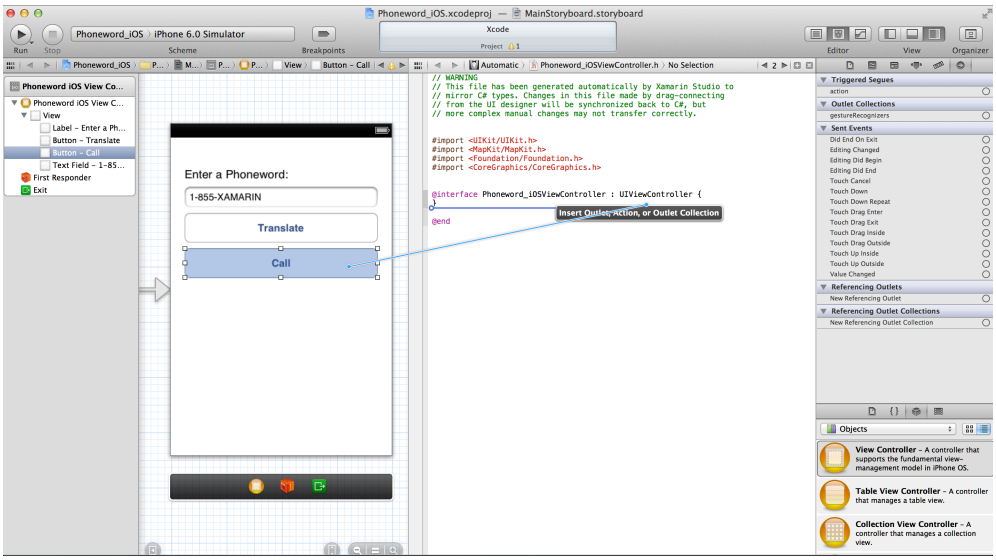


Now we can start wiring up our Outlets.

Adding an Outlet

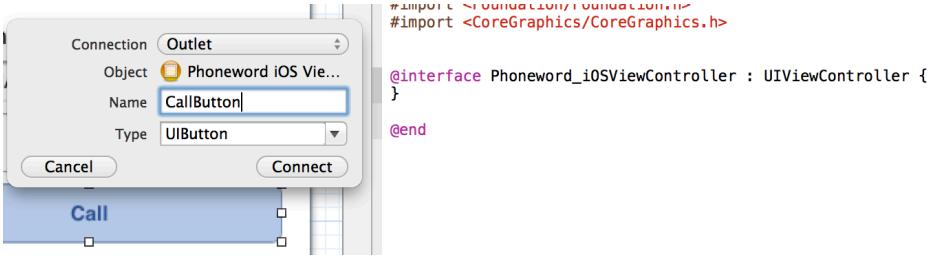
In order to create the Outlet, use the following procedure:

1. Determine for which control you want an Outlet. In our case, let's start with the **Call** button.
2. Hold down the Control key on the keyboard, and then drag *from* the control *to* an empty space in your code file (.h 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 `CallButton`:

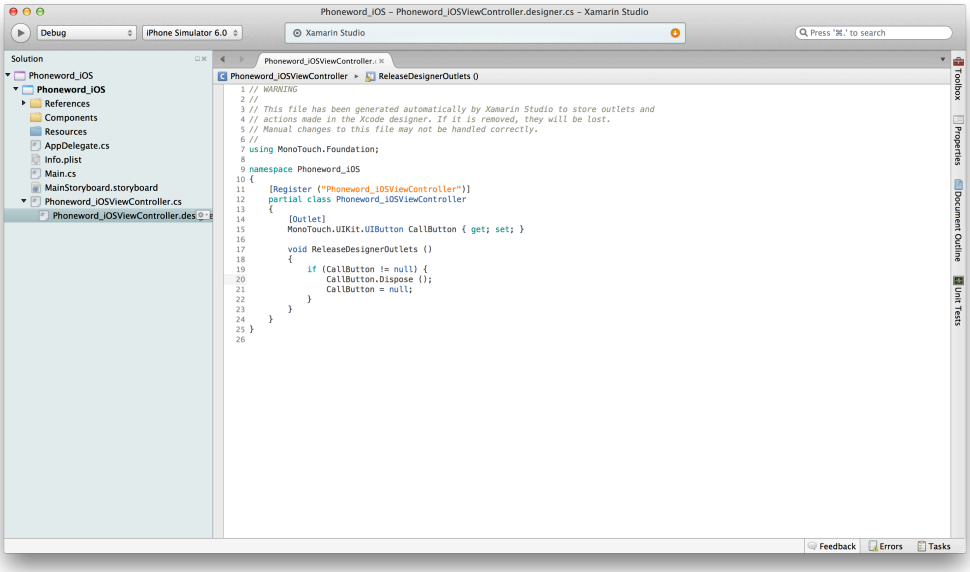


Click **Connect** after you've filled out the form, and Xcode will insert the appropriate code in the .h file:

```
@interface Phoneword_iOSViewController : UIViewController {
}
@property (retain, nonatomic) IBOutlet UIButton *CallButton;

@end
```

Save the file, and then go back to Xamarin Studio. 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:



If it's not there, switch back to IB and save.

As you can see, Xamarin Studio listens for changes to the header (.h) file, and then automatically synchronizes those changes in the respective .designer.cs file to expose them to your application via properties.

We need to create two more Outlets for our text field and translate button, so switch back over to Xcode. Create the Outlets the same way you did for the `CallButton's` Outlet, naming them `PhoneNumberText` and `TranslateButton` respectively. The .h file should have the following Outlets now defined:

```
@property (nonatomic, retain) IBOutlet UIButton *CallButton;
@property (retain, nonatomic) IBOutlet UIButton *TranslateButton;
@property (retain, nonatomic) IBOutlet UITextField *PhoneNumberText;
```

We're going to use these in our Xamarin.iOS application next.

Writing the Code

Now that we're back in Xamarin Studio and we have a UI, let's double-check our **Phoneword_iOSViewController.designer.cs** file:

```
using MonoTouch.Foundation;

namespace Phoneword_iOS
{
	[Register ("Phoneword_iOSViewController")]
	partial class Phoneword_iOSViewController
	{
		[Outlet]
		MonoTouch.UIKit.UIButton CallButton { get; set; }

		[Outlet]
		MonoTouch.UIKit.UIButton TranslateButton { get; set; }
```

```
                [Outlet]
                MonoTouch.UIKit.UITextField PhoneNumberText { get; set; }

                void ReleaseDesignerOutlets ()
                {
                        if (CallButton != null) {
                                CallButton.Dispose ();
                                CallButton = null;
                        }

                        if (TranslateButton != null) {
                                TranslateButton.Dispose ();
                                TranslateButton = null;
                        }

                        if (PhoneNumberText != null) {
                                PhoneNumberText.Dispose ();
                                PhoneNumberText = null;
                        }
                }
        }
}
```

As you can see, our Outlets have been synchronized. Let's write some code that uses them.

Wiring the Outlets

In our application, when the translate button is clicked, we're going to convert the text to the numeric phone number. In order to accomplish this, we need to handle the button's `TouchUpInside` event.

In the `Phoneword_iOSViewController` class, add the following code to the `ViewDidLoad` method:

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

        TranslateButton.TouchUpInside += (object sender, EventArgs e) => {
        };
}
```

As the name suggests, the `TouchUpInside` event is raised when the user touches a button and lifts up inside the button's bounds. It's important to use `TouchUpInside`, as opposed to *TouchDown*, because it provides an opportunity for users to slide their finger off the control to cancel the event. We wire up the handler for this event in the `ViewDidLoad` method, which is called after the view is loaded but before it is displayed on the screen.

This event is possible because the `TranslateButton` is exposed via an Outlet, so we can access it as we would a normal control that is exposed as a property.

Add an event handler for the `CallButton` just below the one for the `TranslateButton`, as shown below:

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

    string phoneNumber = "";

    // On "Translate" button-up, request translation from model
    TranslateButton.TouchUpInside += (object sender, EventArgs e) => {};

    // On "Call" button-up, try to dial a phone number
    CallButton.TouchUpInside += (object sender, EventArgs e) => {};
}
```

Next, let's add some behavior to these event handlers.

Adding Implementation Code

The code to translate text into a numeric phone number uses a helper function called `Translate`, which is included with the sample code that accompanies this section.

Let's add the following code to the `TranslateButton's` handler in `ViewDidLoad`:

```
public override void ViewDidLoad ()
{
    base.ViewDidLoad ();
    string phoneNumber = "";

    //On "Translate" button-up, request translation from model
    TranslateButton.TouchUpInside += (object sender, EventArgs e) => {
        PhoneNumberText.ResignFirstResponder ();
        phoneNumber = PhoneTranslator.Translate (PhoneNumberText.Text);
        CallButton.SetTitle ("Call " + phoneNumber,
               UIControlState.Normal);
        CallButton.Enabled = phoneNumber != "";
    };

    //On "Call" button-up, try to dial a phone number
    CallButton.TouchUpInside += (object sender, EventArgs e) => {};
}
```

We dismiss the keyboard in the `PhoneNumberText` field with a call to `ResignFirstResponder()`. Then, we retrieve the phone word that the user entered using the `PhoneNumberText` Outlet we created for the text field and pass that alphanumeric value to `PhoneTranslator.Translate()`, which converts it to a purely numeric string. We set the title of the `CallButton` outlet appropriately and, if the new phone number is not empty, we enable the button. When a button's `Enabled` property is set to `true`, it will allow clicks and raise touch events. When set to `false`, it will be disabled.

Next, we need to add code to dial the phone number when the user touches the call button. Add the following code to the `CallButton's` handler:

```
CallButton.TouchUpInside += (object sender, EventArgs e) => {
       var url = new NSUrl ("tel:" + phoneNumber);

       if (!UIApplication.SharedApplication.OpenUrl (url)) {
           var av = new UIAlertView ("Not supported"
              , "Scheme 'tel:' is not supported on this device"
              , null
```

```
                    , "OK"
                    , null);
                av.Show ();
            }
        };
```

iOS supports url schemes that trigger certain applications to open for a particular URL. For example, the `tel:` scheme is used to open the phone application for the telephone number in the `NSUrl` instance passed to the `UIApplication.SharedApplication.OpenUrl` method. For iOS devices that do not include telephone support, this method will return `false`. We handle that situation by raising a dialog that explains the situation to the user.

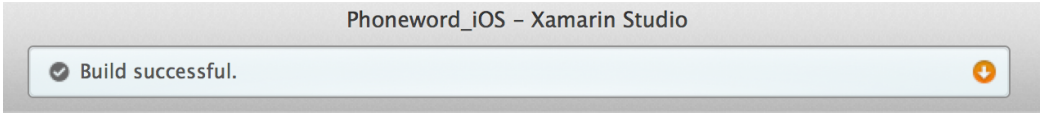
That's it! We've created our first Xamarin.iOS application, so now it's time to run it and test it!

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 toolbar, make sure that **Debug** and **iPhone Simulator 6.0** are 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 area of Xamarin Studio, as show below:



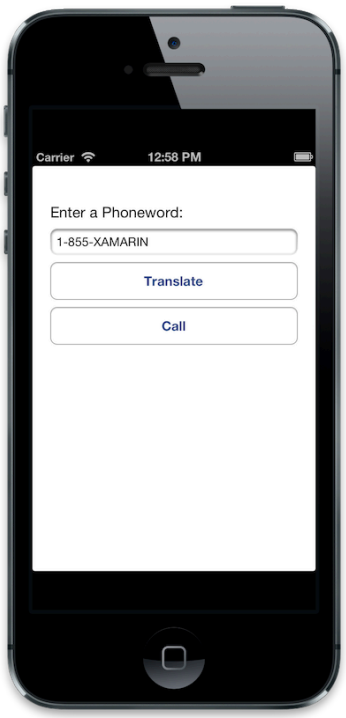
If there are errors, review your code and make sure that you've followed the steps correctly. Start by confirming that your code (both in Xcode and in Xamarin Studio) matches the code in the tutorial.

Deploying to the Device Simulator

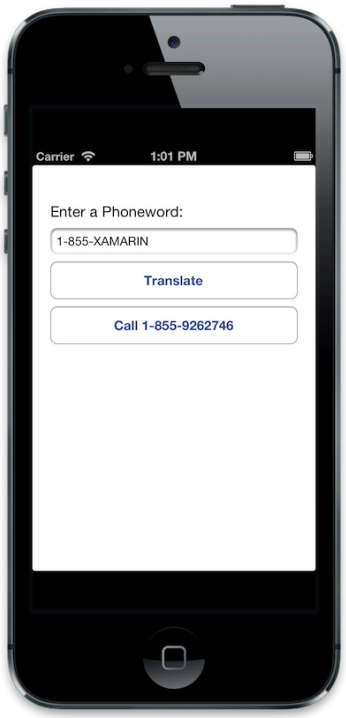
To run the application in the Device Simulator, you have three options:

- Press ⌘+Enter
- From the **Run** menu, choose Start **Debug**ging
- Click the **Run** button with the arrow from the toolbar.

The simulator should launch and the application will run, after which you should see something like following:



Clicking on the **Translate** button should give you something similar to the following, where the translated number is displayed in the Call button's title:



Congratulations, you've now built and run your very first Xamarin.iOS application for the iPhone!

Choosing Which Device to Simulate

By default, the Device Simulator will simulate the iPhone Retina 4-inch display of the iPhone 5. However, you can also simulate an older, smaller iPhone without the Retina Display, as well as the iPad.

To change the simulator, choose one of the following from the iOS Simulator's **Hardware > Device** menu:

- **iPad** – First and second generation iPad (1024 x 768)
- **iPad (Retina)** – iPad with Retina display (2048 x 1536)
- **iPhone** – Resolution matches iPhone 3GS (480 x 320)
- **iPhone (Retina 3.5 inch)** – Resolution matches the iPhone 4/4s (960 x 640)
- **iPhone (Retina 4.0 inch)** – Resolution matches the iPhone 5 (1136 x 640)

The simulator selection will be used the next time the application is run from Xamarin Studio.

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 Follow the instructions in the Device Provisioning document in the Xamarin documentation 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 target in the Xamarin Studio toolbar to **iOS Device** and deploy via run, as you did before for the simulator.

Debugging

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

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.

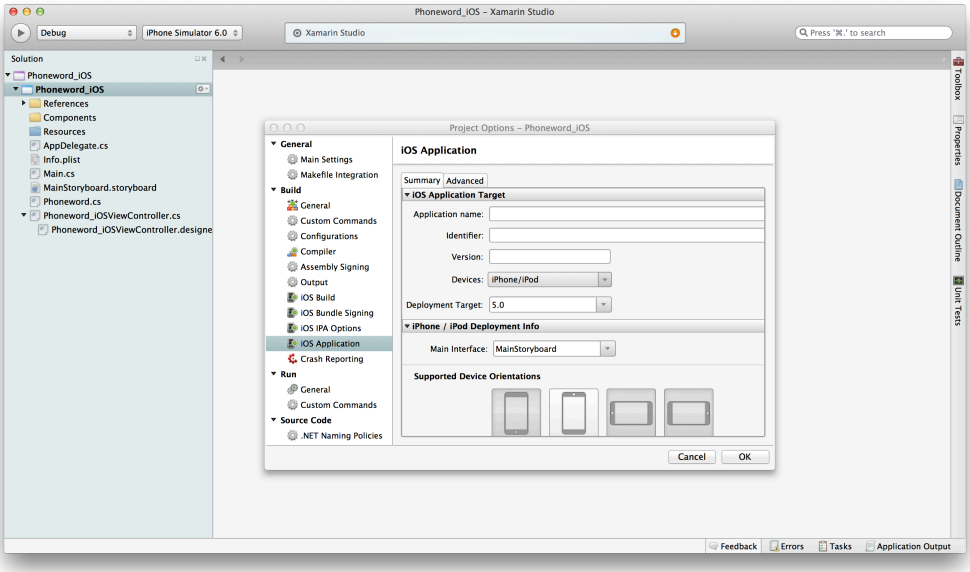
Let's spruce up our application by specifying its name, icons, and provide a splash screen image that will be shown while the application is loading. In addition, we'll see how to use the default screen to prevent the application from being letterboxed on the larger iPhone 5-screen size.

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 Xamarin Studio. This editor can be invoked either directly by double-clicking on the Info.plist file itself, or by double-clicking on the project and choosing **iOS Application**:



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

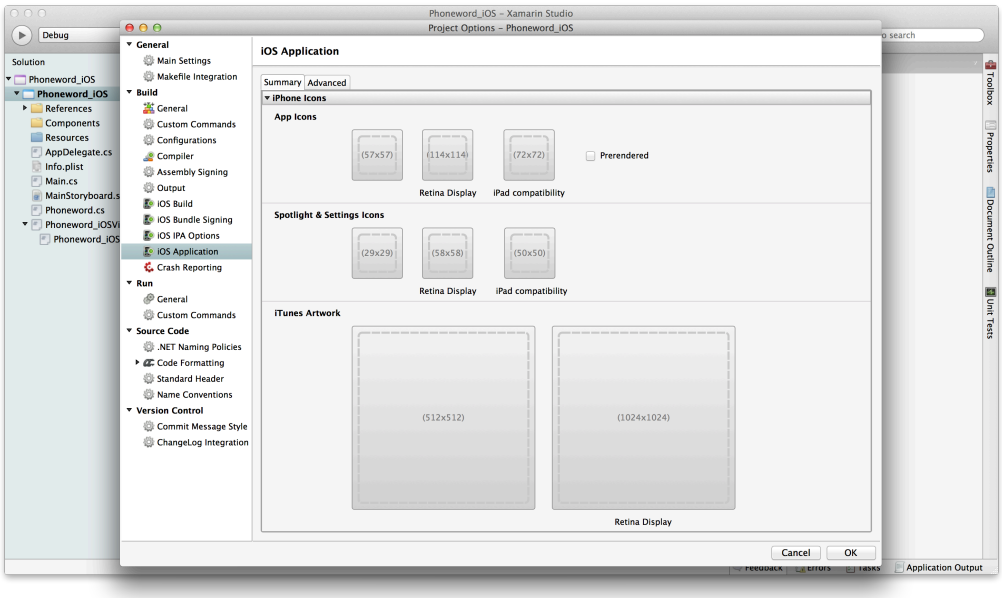
Application Name Setting

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

Now let's 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:

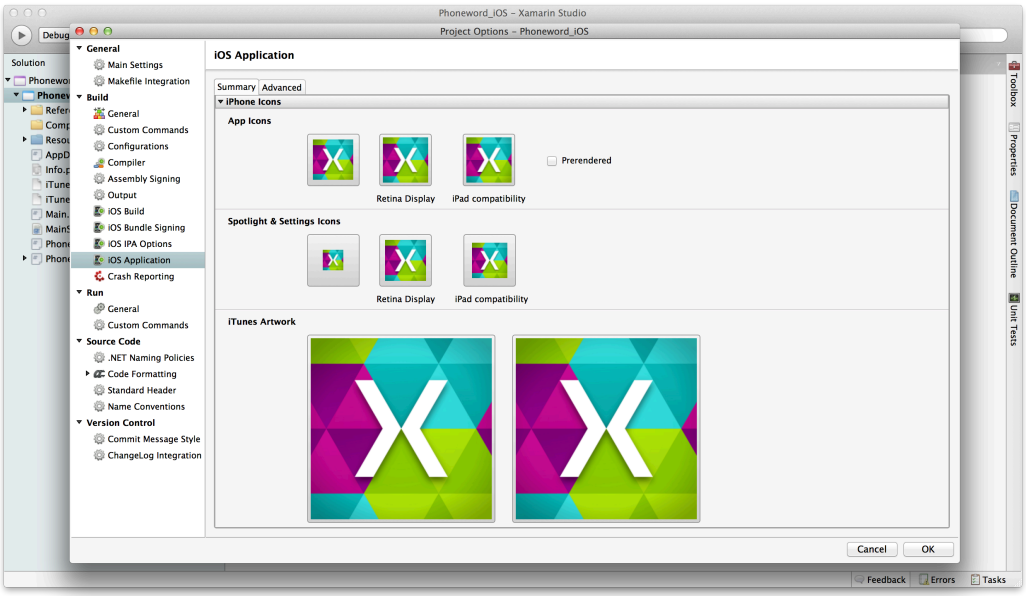


If you hover over each of the icon blocks, a tooltip will show you what size is needed for each icon and a description of what the icon is used for. You can also refer to the following table:

Image	Description	iPhone	iPhone Retina Display	iPad	iPad Retina Display	iPhone 5
Application Icon	Icon displayed on the device's home screen, used to launch an application.	57x57	114x114	72x72	144x444	114x114

Settings Icon	Icon used to identify an application in the iOS *Settings Application*.	29x29	58x58	29x29	58x58	58x58
Spotlight Search (uses same image as Setttings on iPhone)	Icon used to identify an application in the *Spotlight Search* results.	29x29	58x58	50x50	100x100	58x58
iTunes Image	Large image used by iTunes during ad hoc distribution.	512x512	1024x1024	512x512	1024x1024	1024x1024
Launch Image	Image used as a placeholder screen while the application loads.	320x480	640x960	Portrait: 768x1004 Landscape: 1024x748	Portrait: 1536x2008 Landscape: 2048x1496	640x1136

To configure an icon, click on one of the icon blocks and browse to the icon location. You can find some samples in the App Icons and Splash Screens folder in the curriculum folder. 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. However, if you don't want iOS to apply corners and the glassy effect, you can prevent it by checking the **Prerendered** check box.

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

Launch Image

You can provide an image that iOS will display while your application is launching by adding an image file with the appropriate naming convention to the

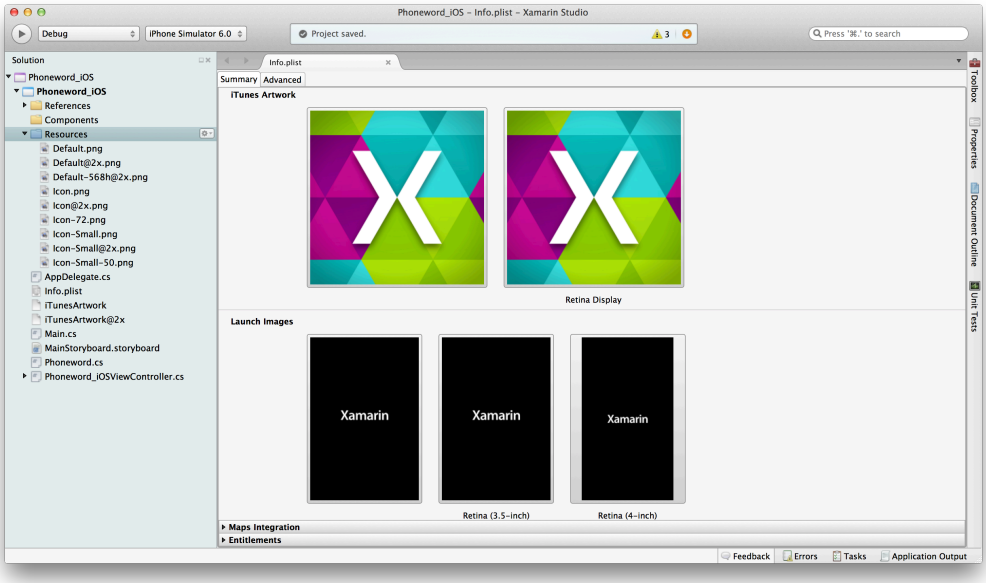
Resources of the project. The following table describes the size and name parameters of the file for regular iPhones (3Gs and older), iPhones with the Retina Display (4G and newer), and for the iPhone 5:

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.

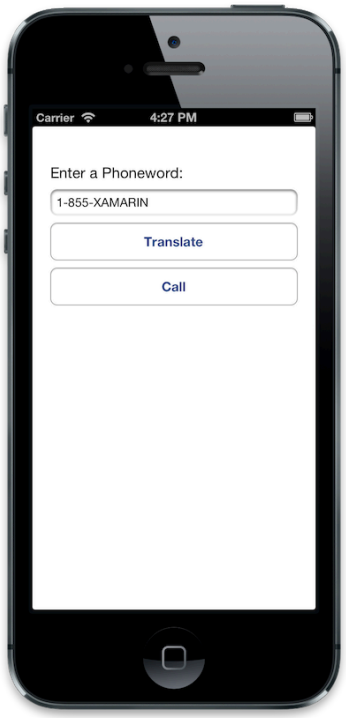
Note that the device has a case-sensitive file system, but the iOS simulator by default does not (unless you've formatted your hard drive as case-sensitive). This means that if the file case is incorrect (note the leading capital), the image will work correctly on the simulator, but not on the device.

To specify the loading screen images, simply click on the image block in Xamarin Studio and browse to the image, just as we did earlier for the icons. After selecting the images, the selected images will be copied into the project **Resources** folder and displayed in the Info.plist editor, as shown below:



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

By simply including a **Default-568h@2x.png** image, the application will now run full screen on an iPhone 5, rather than letterboxed, as shown below:



Summary

Congratulations! We covered a lot of ground here, but if you followed along from start to finish, you should now have a solid understanding of the components of a Xamarin.iOS application, as well as the tools used to create them.

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

In the next chapter, we're going to address the single screen shortcoming, and look at the *Model, View, Controller* (*MVC*) pattern in iOS and how it's used to create multi-screened applications. We don't cover creating universal iPad/iPhone apps in this course, so if you're interested, check out the iPad + Universal (iPhone + iPad) Applications guide.