Hello, iOS MVC

Evolve Fundamentals Track, Chapter 2

Xamarin Inc.

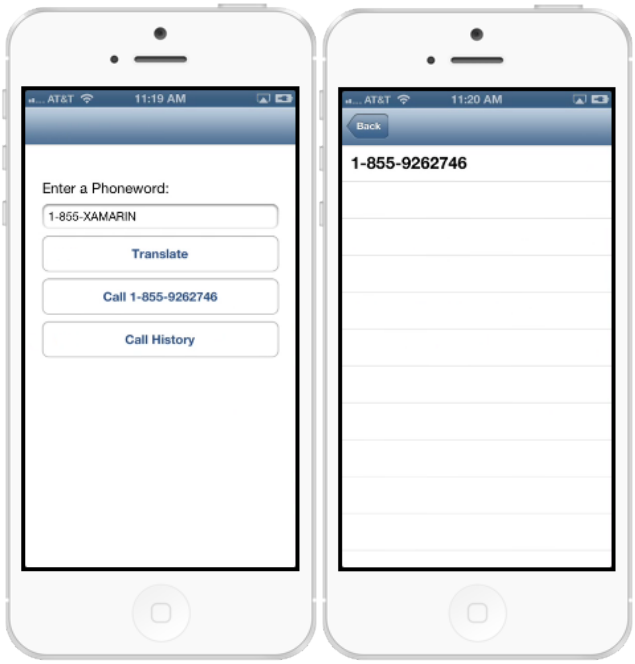
Overview

We built our first Xamarin.iOS application in the last chapter. In that chapter, we covered a lot of ground and introduced the tools that we use to build Xamarin.iOS applications, as well as fundamental concepts such as Outlets. However, the application we built had only one screen, which is great for a first app, but real-world applications are rarely this simple.

In this chapter, we're going to take a look at the *Model, View, Controller (MVC)* pattern and see how it is 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 look at how to include support for navigation between multiple screens by extending the PhoneWord application we created in Chapter One, adding a second screen that contains the call history as shown below:



Requirements

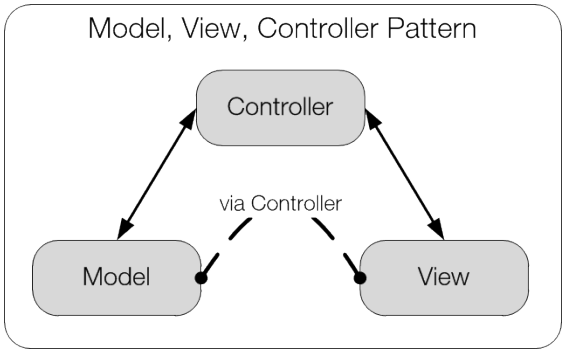
This tutorial builds directly on the skills and knowledge learned in the tutorial from the first chapter. Therefore, before starting this tutorial, you must have either completed the previous tutorial, or be familiar with the concepts introduced in it.

Model, View, Controller (MVC) Pattern

As we learned in the first chapter, 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.

However, before we dive too deeply into implementation, 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'll soon examine each one of these components and their responsibilities, 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. Continuing this comparison, the View corresponds to the ASPX (HTML) page in ASP.NET, or to XAML in a WPF application. The Controller is the component that is responsible for managing the View, which corresponds to the code-behind in ASP.NET or WPF.

Neither ASP.NET nor WPF applications traditionally use a true MVC pattern, so the analogy is rough, but it's a good start from a conceptual standpoint. 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 refers to either an application-specific representation of data that is to be displayed and/or entered on the view, such as a domain model, or it can represent the entire back-end logic of an application. Either way, it's 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. Or, some might include those classes, as well as any other supporting classes, such as those that persist the data to a database.

It's important to note that the MVC is completely agnostic of the data persistence and access of the 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 mentioned above, 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 or back end. In our To-do application, it's conceivable that we might have a set of screens that give instructions on the use of the application. Those screens will still be created by using views and controllers, but they would not have any real Model data or back end.

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 by 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 then returning the appropriate view. It also listens to requests from the view (say, for example, when a button is clicked), performs the appropriate processing and view modification, and then re-displays the view.

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

Furthermore, the controller is responsible for creating or retrieving the Model from whichever backing store exists in the application, and then populating the view with this 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 might 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

In iOS, the application layer that is responsible for the UI is known as *CocoaTouch*. As part of CocoaTouch, the views and controllers that are available for building applications are known as *UIKit*. UIKit contains all the native controls that iOS users have come to expect in iOS applications.

In UIKit, the `UIViewController` class represents controllers and the `UIView` class represents views.

When creating iOS applications, the topmost (first) controller added to the window is called the root controller. When using storyboards, as in the PhoneWord application, the window is created automatically and its root controller is set within the storyboard.

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 in several of the 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. The actual title displayed in the title bar comes from the `Title` property of whichever `UIViewController` instance is currently at the top of the `UINavigationController's` controller stack.

It's important to note that there are other ways to handle navigation in your applications that can be used instead of, or in conjunction with, a navigation controller. 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 the most commonly used way to provide multi-screen functionality in an application, and it's often used with other navigation controls, such as the Tab Bar and the Split View controller.

Storyboards

Storyboards are a new way to build application user interfaces in Xcode. Previously, developers created XIB files for each view controller, and programmed the navigation between each view manually. A storyboard uses a design surface that offers WYSIWYG editing of the application user interface. More specifically, a storyboard offers developers a visual way to interact with view controllers.

An application can have its entire UI defined in a single storyboard, or the UI can be broken down into multiple storyboards. Storyboards are compiled into individual views to avoid any delays that might be caused by loading large scenes.

We used a storyboard to create our single screen version of PhoneWord in Chapter One, but we didn't delve much into working with storyboards at that point, since their main use is to provide navigation between multiple screens.

In the next section, we'll give step-by-step instructions on how to use a storyboard with Xamarin.iOS to create a simple screen navigation flow.

Creating our Application

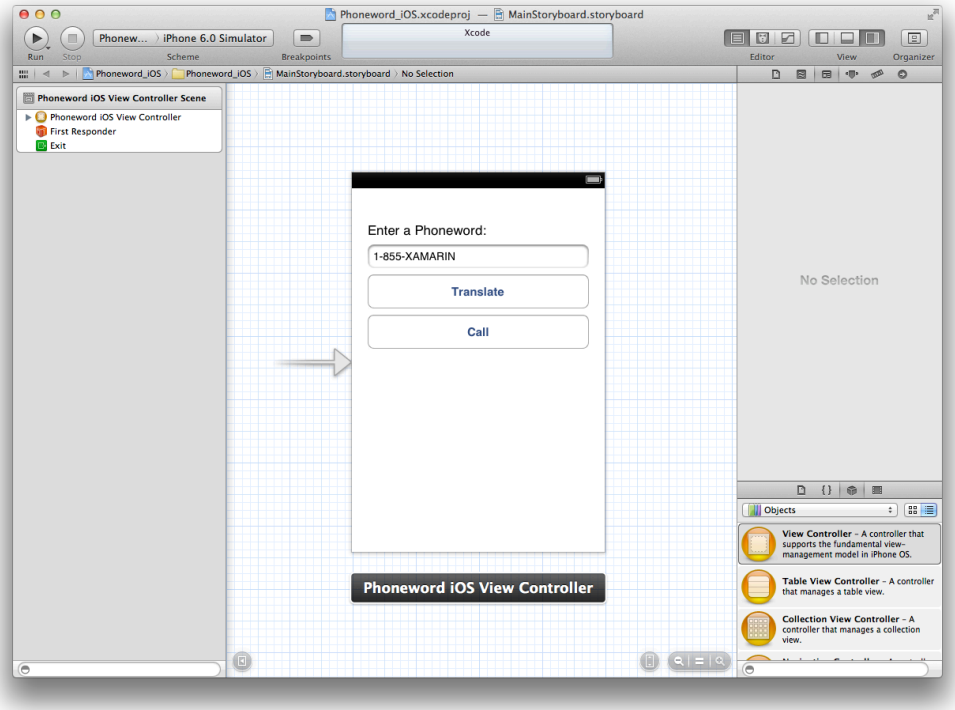
Now that we have an understanding of the MVC pattern and the navigation controller, and how they're used in iOS programming, let's extend the PhoneWord application to include multiple screens.

We'll begin by using the PhoneWord example we created in Chapter One.

Adding a Navigation Controller

Let's change the PhoneWord application to use a `UINavigationController`.

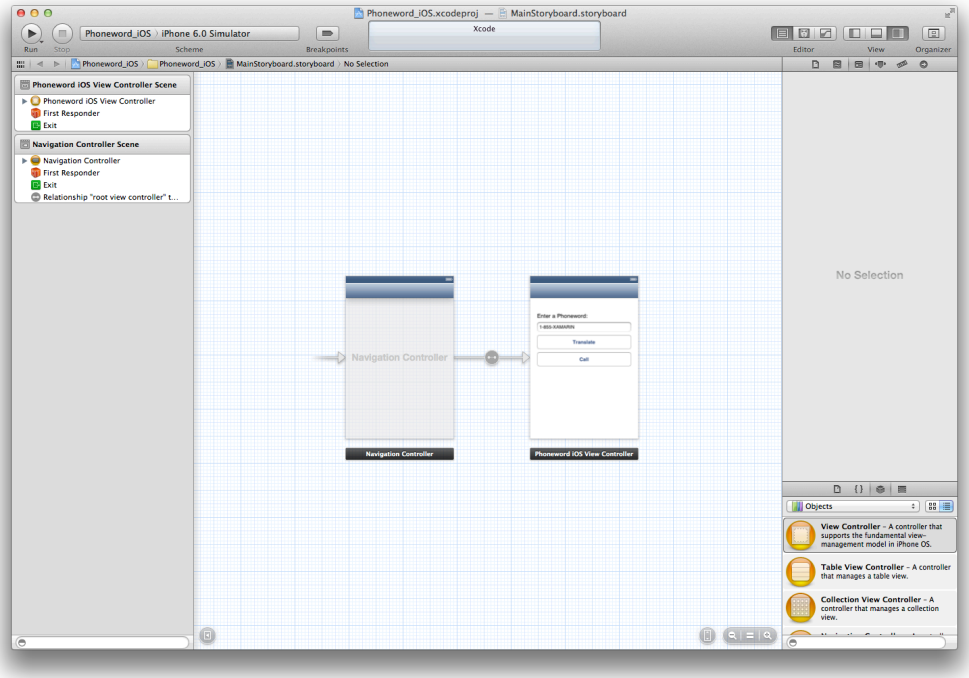
Open the PhoneWord project in Xamarin Studio and double-click the **MainStoryboard.storyboard** file to load the storyboard in Xcode:



Adding A Navigation Controller

The first thing we want to do is create a navigation controller and set our main screen (`Phoneword_iOSViewController`) as its root view controller.

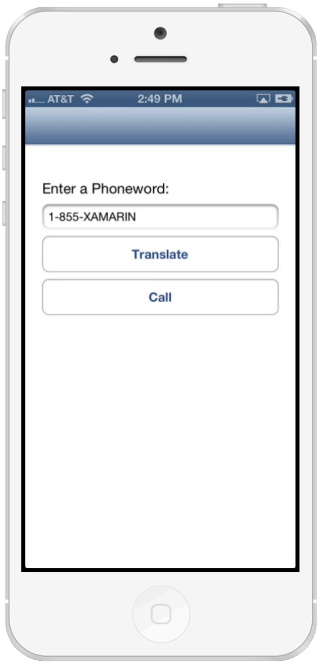
Interface Builder makes this incredibly easy. All we have to do is select the view controller in IB, and then in the **Editor Menu** choose **Embed In > Navigation Controller**. IB will automatically add the navigation controller to the storyboard and set our screen as its root view controller:



Note the arrow on the left of our navigation controller, as well as the arrow connecting it to the screen.

Both of these arrows represent *Segues*, which we'll examine more in a moment. The arrow on the left is known as a *Sourceless* Segue, and the arrow connecting the two items with the line and two dots in the middle of it denotes a root view controller relationship. A Sourceless Segue is the default entry in a storyboard; it's the first controller that will run in our application.

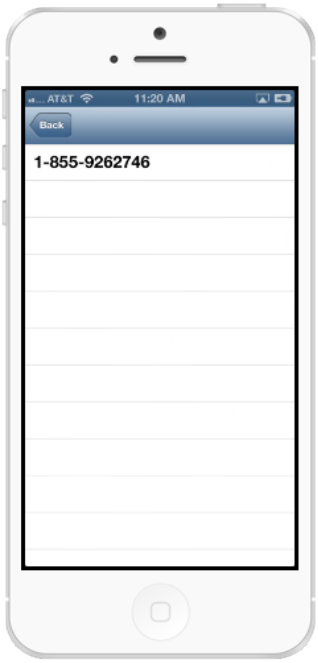
If we run the application now, we see that the initial screen contains a navigation controller with the PhoneWord controller as its root:



Now that we have a navigation controller wired up, we can use it to load a second controller. First, we'll need to create the second controller.

Adding the Call History

For the second screen in this application, we'll display the call history in a table as shown below:

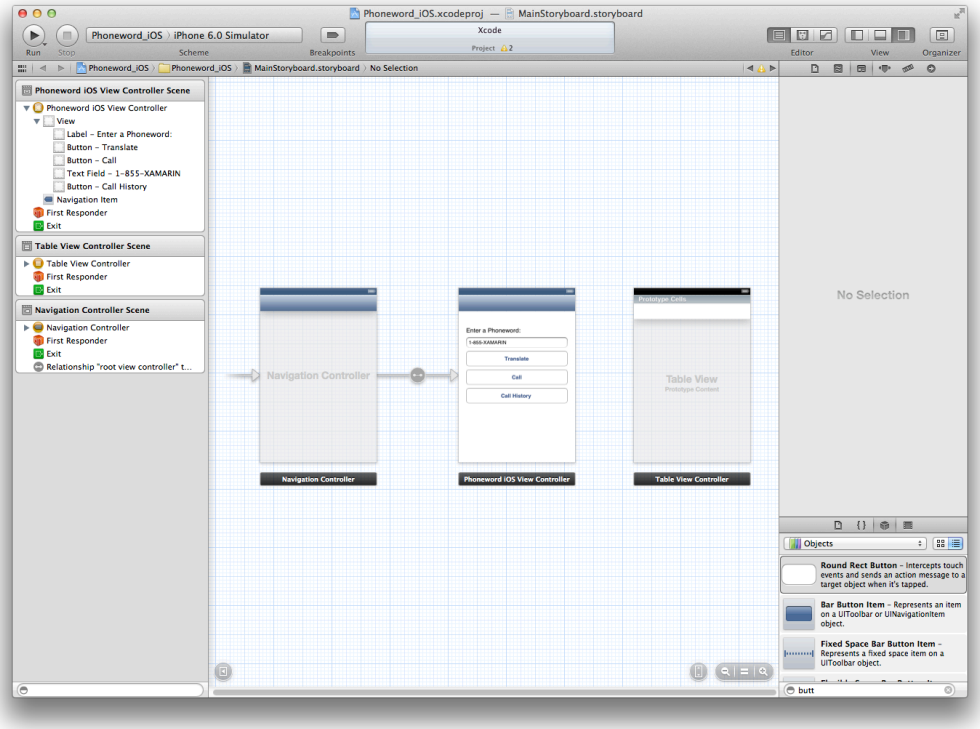


Adding the Table Controller

First, let's create the table in the storyboard.

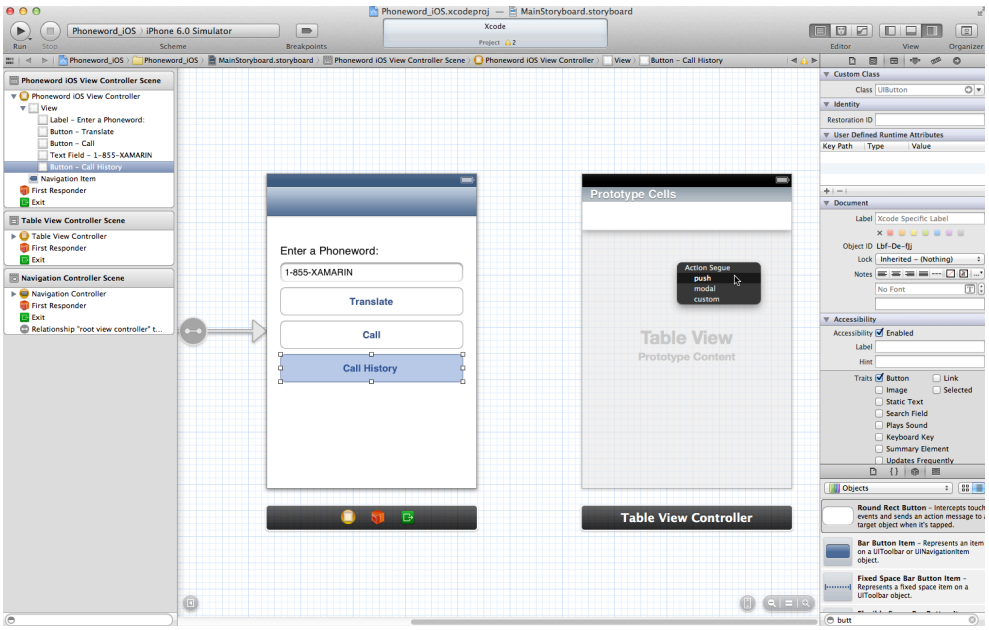
Drag a **Table View Controller** from the **Object Library** onto the designer. We're going to navigate to the call history when the user taps a button on the first screen, so add a button to the PhoneWord controller's view as well.

Double-click the button to change its title to **Call History**. The following screenshot shows the storyboard with button and table controller added:

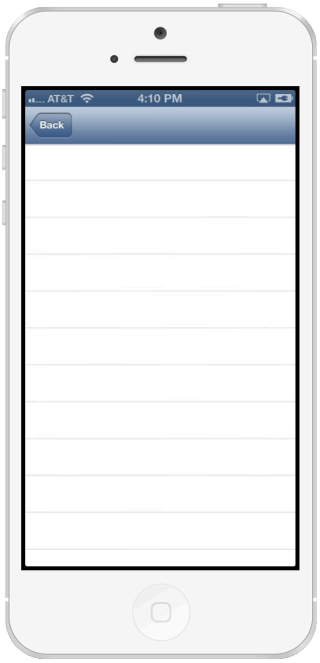


Adding a Push Segue

To make the table display when the user taps **Call History**, we need to create a *Push Segue* to the table controller. Control Drag from the button to the table controller, and then select **push** under the **Action Segue** popup menu, as shown below:



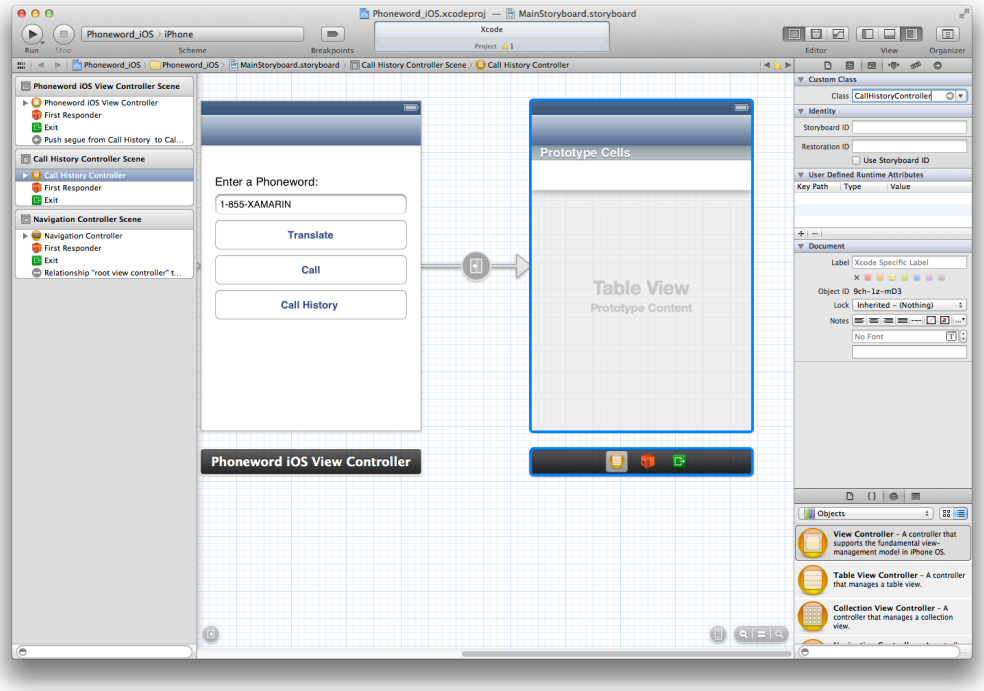
Now, when the user taps **Call History**, the table controller's view is displayed, showing an empty table view:



Implementing the Table Controller

To implement the table controller, we need to have a backing class for the table controller in the storyboard. Xamarin Studio will generate a backing class we can use when we change the name of the class in Xcode.

With the Table Controller selected, open the **Identity Inspector** and change the **Class** name to `CallHistoryController`, as shown below:



Press the RETURN key on the keyboard and switch back to Xamarin Studio. In the **Solution Pad**, notice that a `CallHistoryController.cs` file has been generated, which contains the following `UITableViewController` subclass:

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

namespace Phoneword_iOS
{
        public partial class CallHistoryController : UITableViewController
        {
                public CallHistoryController (IntPtr handle) :
                        base (handle)
                {
                }
        }
}
```

We are now ready to add our implementation to this class to display the call history. The code to do this involves populating the table's data source from a backing data store, which, for simplicity, is an in-memory `List<String>` in this case. We'll discuss tables in depth later in the course.

Add the following code to the `CallHistoryController` class:

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

namespace Phoneword_iOS
```

```
{
        public partial class CallHistoryController : UITableViewController
        {
                public List<String> PhoneNumbers { get; set; }

                static NSString callHistoryCellId =
                        new NSString ("CallHistoryCell");

                public CallHistoryController (IntPtr handle) :
                        base (handle)
                {
                        TableView.RegisterClassForCellReuse (
                                typeof(UITableViewCell), callHistoryCellId);
                        TableView.Source = new CallHistoryDataSource (this);
                        PhoneNumbers = new List<string>();
                }

                class CallHistoryDataSource : UITableViewSource
                {
                        CallHistoryController controller;
                        CallAlertDelegate alertDelegate;

                        public CallHistoryDataSource (
                                CallHistoryController controller)
                        {
                                this.controller = controller;
                                alertDelegate = new CallAlertDelegate ();
                        }

                        public override int RowsInSection (UITableView
                                tableView, int section)
                        {
                                return controller.PhoneNumbers.Count;
                        }

                        public override UITableViewCell GetCell (UITableView
                                tableView, NSIndexPath indexPath)
                        {
                                var cell = tableView.DequeueReusableCell (
                                        CallHistoryController
                                        .callHistoryCellId);

                                int row = indexPath.Row;

                                cell.TextLabel.Text =
                                        controller.PhoneNumbers [row];

                                return cell;
                        }

                        public override void RowSelected (UITableView
                                tableView, NSIndexPath indexPath)
                        {
```

```
                                tableView.DeselectRow (indexPath, false);

                                alertDelegate.PhoneNumber = controller
                                        .PhoneNumbers [indexPath.Row];

                                var alert = new UIAlertView ("Dial Number"
                                        , "Would you like to call this
                                                number?"
                                        , alertDelegate
                                        , "No"
                                        , "Yes");
                                alert.Show ();
                        }

                        class CallAlertDelegate : UIAlertViewDelegate
                        {
                                public string PhoneNumber { get; set; }

                                public override void Clicked (UIAlertView
                                        alertview, int buttonIndex)
                                {
                                        if (buttonIndex == 1) {
                                                PhoneUtility.Dial (
                                                        PhoneNumber);
                                        }
                                }
                        }
                }
        }
}
```

This code makes use of a small `PhoneUtility` class that contains a function that wraps dialing a number. This allows us to avoid redundancy since the application invokes the dialer in a couple of places. The code is included below:

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

namespace Phoneword_iOS
{
        public static class PhoneUtility
        {
                public static void Dial(string phoneNumber)
                {
                        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();
                        }
                }
        }
}
```

Navigating to the Call History

Finally, we need to add code to the `Phoneword_iOSViewController` class to add a number to the history when **dial** is pressed.

As with the `CallHistoryController`, we simply use a `List<String>` for the call history, which we populate when **dial** is pressed, just before launching the dialer, as shown below:

```
string translatedNumber = "";

public List<String> PhoneNumbers { get; set; }

public Phoneword_iOSViewController (IntPtr handle) : base (handle)
{
        PhoneNumbers = new List<String> ();
}

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

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

                // *** SHARED CODE ***
                translatedNumber = Core.PhonewordTranslator.ToNumber(
                        PhoneNumberText.Text);

                if (translatedNumber == "") {
                        CallButton.SetTitle ("Call ",
                                UIControlState.Normal);
                        CallButton.Enabled = false;
                } else {
                        CallButton.SetTitle ("Call " + translatedNumber,
                                UIControlState.Normal);
                        CallButton.Enabled = true;
                }
        };

        CallButton.TouchUpInside += (object sender, EventArgs e) => {

                PhoneNumbers.Add (translatedNumber);
                PhoneUtility.Dial (translatedNumber);
        };
}
```

For the call history to be accessible by the `CallHistoryController`, we can set the `PhoneNumbers` property of the `CallHistoryController` instance before we

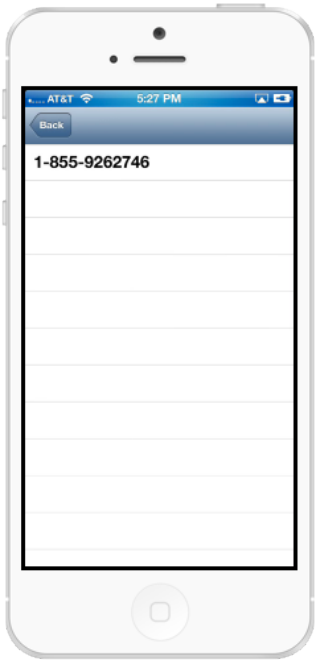
navigate to it. When using storyboards, the `PrepareForSegue` method is called just before transitioning to the destination controller, which is the `CallHistoryController` instance in this case. From this method, we can set the `PhoneNumbers` property, as shown below:

```
public override void PrepareForSegue (UIStoryboardSegue segue, NSObject
sender)
{
        base.PrepareForSegue (segue, sender);

        var callHistoryContoller = segue.DestinationViewController
                as CallHistoryController;

        if (callHistoryContoller != null) {
                callHistoryContoller.PhoneNumbers = PhoneNumbers;
        }
}
```

If we run the application now, we see that navigating to the call history after dialing shows the list of numbers that were dialed:



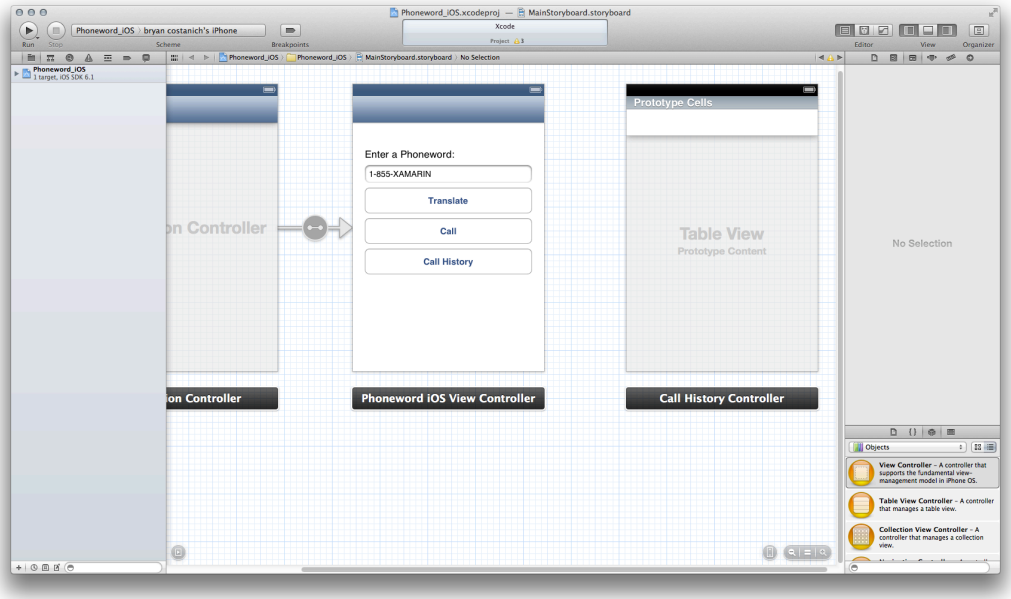
Showing Screens Programmatically with PushViewController

Segues are a great way to create transitions between view controllers in a storyboard, but can only be used for known fixed paths. For instance, imagine that we are creating an application that launches different screens from the same button, depending on the current state of an application. In this case, we need to load the view controller programmatically.

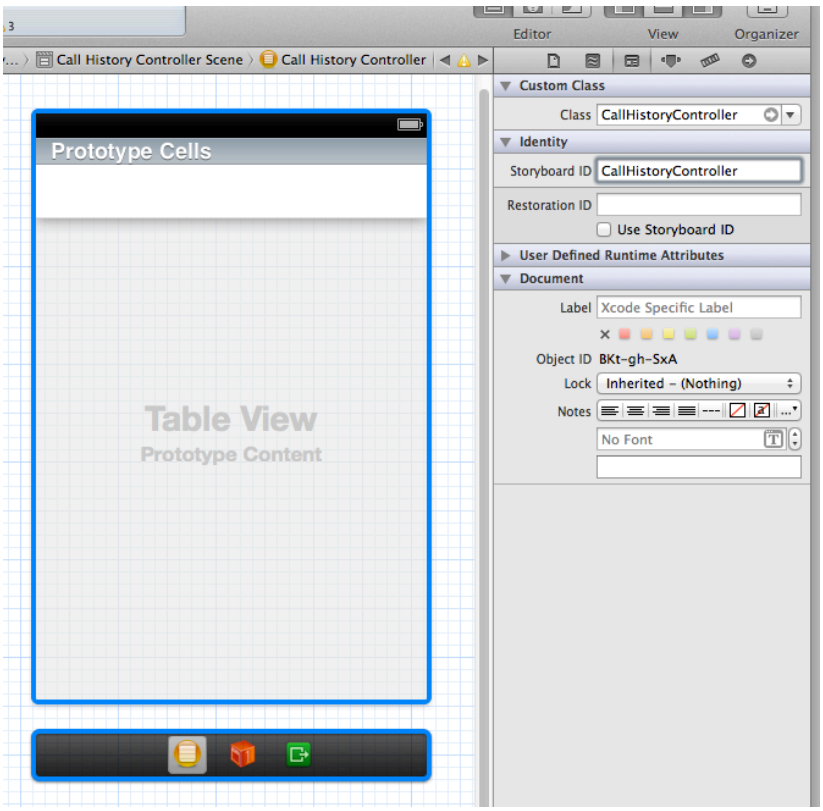
The navigation controller exposes a method called *PushViewController* that allows us to do just that. To use it, we simply instantiate the view controller that we want to push onto the navigation controller, and then call it.

Let's modify our PhoneWord application to launch our Call Log programmatically, rather than with a Segue:

1. First, open the Storyboard in Xcode and delete the Push Segue by selecting it and hitting **⌘+Backspace** to delete it. Our storyboard should now look like this:



2. Next, create an Outlet for the **Call History** button called `CallHistoryButton`.
3. Now, we need to add a *Storyboard Identifier* to our Call History view controller so that we can load it programmatically from the storyboard. To do this, select the **Call History Controller** in Xcode, then select the **Identity Inspector** tab, and in the Identity section, set the **Storyboard ID** to be `CallHistoryController`:



4. Save your work, and then open the Phoneword_iOSViewController.cs file and delete the `PrepareForSegue` method, as we'll no longer need it.

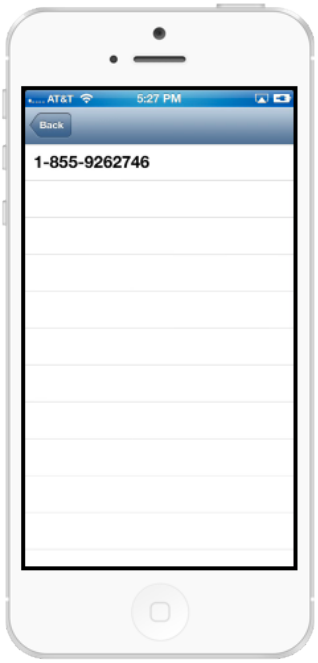
5. Finally, wire up the `CallHistoryButton.TouchUpInside` event as follows:

```
CallHistoryButton.TouchUpInside += (object sender, EventArgs e) => {
      CallHistoryController callHistory =
      this.Storyboard.InstantiateViewController
      ("CallHistoryController") as CallHistoryController;
      this.NavigationController.PushViewController (callHistory,
      true);
};
```

We do two things in this event. First, we create a new instance of our Call History controller via the current storyboard's `InstantiateViewController` method (passing the Storyboard ID we defined in step 3), and then we call `PushViewController` on the current controller's `NavigationController` object.

All view controllers have a `NavigationController` property, and as long as the view controller has been added to a navigation controller, then the property won't be `null`. `PushViewController` takes two parameters: first, the controller to push, and second, a Boolean value that specifies whether or not the controller should animate onto screen. Usually we want to pass `true`, unless we're pushing the root controller onto it, in which case we would just want it to be there.

If we run the application now and click **Call History**, we should see the Call History screen animate on, just as before:



Summary

In this chapter, 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 also presented step-by-step instructions that showed how to use storyboards to build an application. We discussed creating new view controllers and showed how to link them together with Segues.

In the next chapter, we're going to switch to the iPad and create our first Xamarin.iOS iPad application, and then learn how to make universal applications that target both the iPhone and iPad from a single executable.