iOS Succinctly

by Ryan Hodson

Syncfusion | Technology Resource Portal

iOS Succinctly

By
Ryan Hodson

Foreword by Daniel Jebaraj

Important licensing information. Please read.

Edited by

This publication was edited by Daniel Jebaraj, vice president, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Ryan Hodson began learning ActionScript at age 14, which eventually led to a job creating Flash-based data visualizations for the National Center for Supercomputing Applications at the University of Illinois. Since then, he's worked in a diverse collection of programming fields, building everything from websites to e-publishing platforms, touch-screen thermostats, and natural language processing tools. These experiences have led to a love of exploring new software and a proficiency in several languages (HTML/CSS, JavaScript, PHP, MySQL, Python, Java, Objective-C, PDF) and many frameworks (WordPress, Django, CherryPy, and the iOS and OSX SDKs, to name a few).

In 2012, Ryan founded an independent publishing firm called RyPress and published his first book, *Ry's Friendly Guide to Git*. Since then, he has worked as a freelance technical writer for well-known software companies, including Syncfusion and Atlassian. Ryan continues to publish high-quality software tutorials via RyPress.com.

Introduction

Mobile applications are one of the fastest growing segments of the technology industry, and the iPhone and iPad have been at the forefront of the mobile revolution. Developing applications for these platforms opens the door to a vast number of mobile users. Unfortunately, the variety of underlying technologies can be overwhelming for newcomers to iOS, and the 1,500+ official help documents available from the iOS Developer Library don't exactly provide an approachable introduction to the platform. The goal of *iOS Succinctly* is to provide a concise overview of the iOS landscape.

iOS and the iOS SDK

iOS is the operating system behind iPhone and iPad applications. It takes care of the low-level system tasks like managing memory, opening and closing applications, and rendering pixels to the screen. On top of this core operating system rests a collection of **frameworks**, which are C and Objective-C libraries that provide reusable solutions to common programming problems. For example, the UIKit Framework defines classes for buttons, text fields, and several other user interface components. Instead of implementing your own buttons from the ground up, you can leverage the existing UIButton class.

Together, the core operating system and these higher-level frameworks compose the **iOS software development kit** (SDK). The goal of the iOS SDK is to let you focus on developing *what* your application does instead of getting bogged down by *how* it does it. The SDK is divided into layers based on what level of abstraction they provide. These layers, along with some of the popular frameworks they contain, are shown in the following diagram:

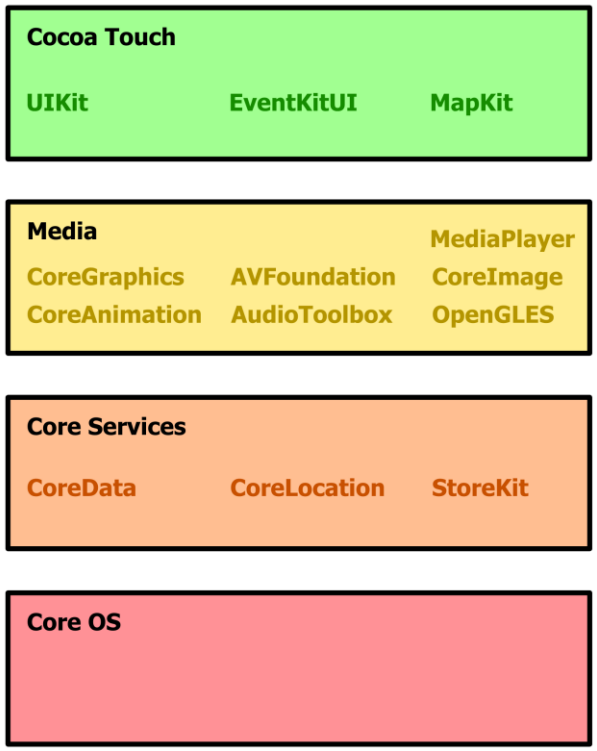

Figure 1: Layers of the iOS SDK frameworks

As a developer, you'll rarely interact directly with the Core OS layer. Most of the time, you'll be working with the frameworks in the Cocoa Touch, Media, or Core Services layers and let them handle the low-level operating system tasks for you.

About *iOS Succinctly*

iOS Succinctly is the second half of a two-part series on iPhone and iPad app development. The first book, *Objective-C Succinctly*, covered the Objective-C language and the core data structures used by virtually all applications. With this in mind, *iOS Succinctly* assumes that you're already comfortable with Objective-C and have at least a basic familiarity with the Xcode integrated development environment (IDE).

This book begins by exploring the basic design patterns behind iOS development. We'll learn how to create a user interface using a very simple, one-scene application. Then, we'll expand this knowledge to a more complicated multi-scene application. By this point, you should have a solid grasp of the iOS workflow. The remaining chapters look at common development tasks like accessing files, localizing assets for different audiences, and playing sounds.

The sample code in this book can be downloaded from https://bitbucket.org/syncfusion/ios-succinctly.

Chapter 1 Hello, iOS!

In this chapter, we'll introduce the three main design patterns underlying all iOS app development: model-view-controller, delegate objects, and target-action. The model-view-controller pattern is used to separate the user interface from its underlying data and logic. The delegate object pattern makes it easy to react to important events by abstracting the handling code into a separate object. Finally, the target-action pattern encapsulates a behavior, which provides a very flexible way to perform actions based on user input.

We'll talk about all of these patterns in more detail while we're building up a simple example application. This will also give us some experience with basic user interface components like buttons, labels, and text fields. By the end of this chapter, you should be able to configure basic layouts and capture user input on your own.

Creating a New Project

First, we need to create a new Xcode project. Open Xcode and navigate to **File** > **New** > **Project**, or press Cmd+Shift+N to open the template selection screen. In this chapter, we'll be creating the simplest possible program: a *Single View Application*. Select the template, and then click **Next**.

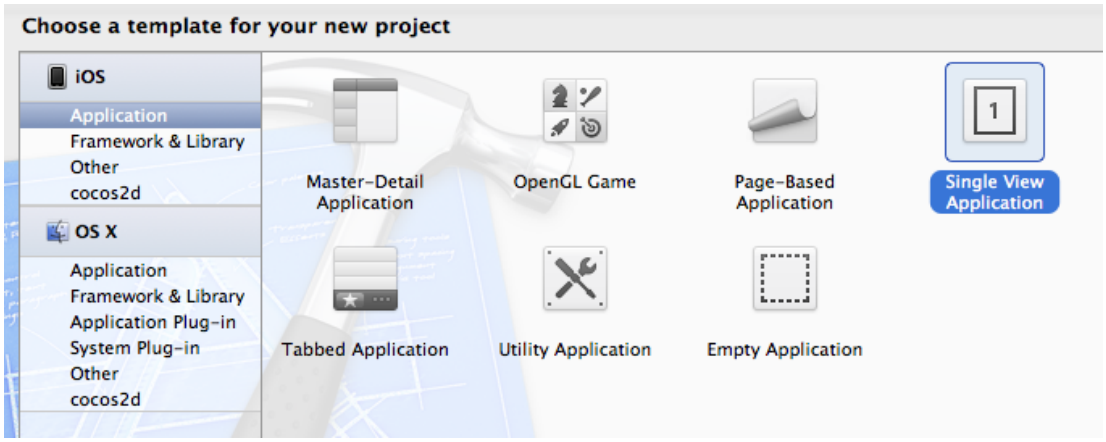

Figure 2: Selecting the Single View Application template

Use *HelloWorld* for the **Product Name**, anything you like for **Organization Name**, and *edu.self* for the **Company Identifier**. Make sure that **Devices** is set to **iPhone** and that the **Use Storyboards** and **Use Automatic Reference Counting** options are selected:

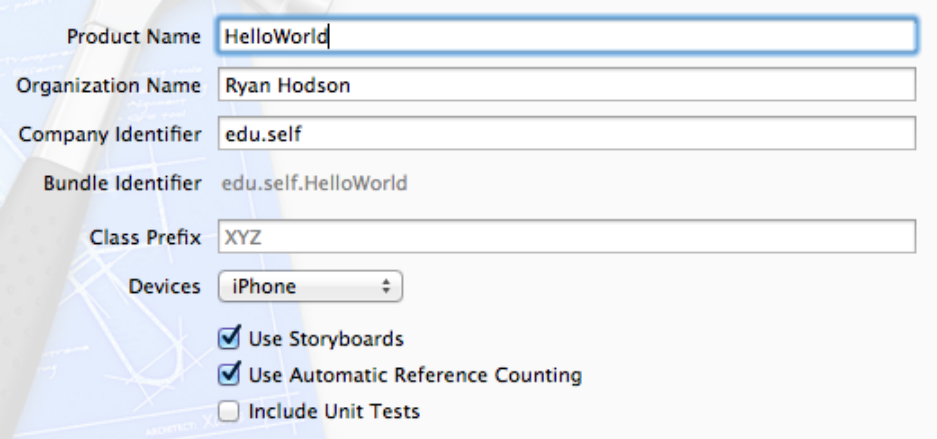

Figure 3: Configuration for our HelloWorld app

Then, choose a location to save the file, and you'll have your very first iOS app to experiment with.

Compiling the App

As with the command-line application from *Objective-C Succinctly*, you can compile the project by clicking the **Run** button in the upper-left corner of Xcode or using the Cmd+R keyboard shortcut. But, unlike *Objective-C Succinctly*, our application is a graphical program that is destined for an iPhone. Instead of simply compiling the code and executing it, Xcode launches it using the **iOS Simulator** application. This allows us to see what our app will look like on the iPhone without having to upload it to an actual device every time we make the slightest change. The template we used is a blank project, so you'll just see a white screen when you run it:

Figure 4: Running the HelloWorld project in the iOS Simulator

While we can't really tell with our current app, the simulator is a pretty detailed replica of the actual iPhone environment. You can click the home button, which will display all the apps that we've launched in the simulator, along with a few built-in ones. As we'll see in a moment, this lets us test the various states of our application.

App Structure Overview

Before we start writing any code, let's take a brief tour of the files provided by the template. This section introduces the most important aspects of our HelloWorld project.

main.m

As with any Objective-C program, an application starts in the **`main()`** function of **main.m**. The **main.m** file for our HelloWorld project can be found in the **Supporting Files** folder in Xcode's **Project Navigator** panel. The default code provided by your template should look like the following:

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

#import "AppDelegate.h"
```

```
int main(int argc, char *argv[]) {
    @autoreleasepool {
        return UIApplicationMain(argc,
                                 argv,
                                 nil,
                                 NSStringFromClass([AppDelegate class]));
    }
}
```

This launches your application by calling the **`UIApplicationMain()`** function, and passing **`[AppDelegate class]`** as the last argument tells the application to transfer control over to our custom **`AppDelegate`** class. We'll discuss this more in the next section.

For most applications, you'll never have to change the default **main.m**—any custom setup can be deferred to the **`AppDelegate`** or **`ViewController`** classes.

AppDelegate.h and AppDelegate.m

The iOS architecture relies heavily on the **delegate design pattern**. This pattern lets an object transfer control over some of its tasks to another object. For example, every iOS application is internally represented as a UIApplication object, but developers rarely create a **`UIApplication`** instance directly. Instead, the **`UIApplicationMain()`** function in **main.m** creates one for you and points it to a delegate object, which then serves as the root of the application. In the case of our HelloWorld project, an instance of the custom **`AppDelegate`** class acts as the delegate object.

This creates a convenient separation of concerns: the **`UIApplication`** object deals with the nitty-gritty details that happen behind the scenes, and it simply informs our custom **`AppDelegate`** class when important things happen. This gives you as a developer the opportunity to react to important events in the app's life cycle without worrying about how those events are detected or processed. The relationship between the built-in **`UIApplication`** instance and our **`AppDelegate`** class can be visualized as follows:

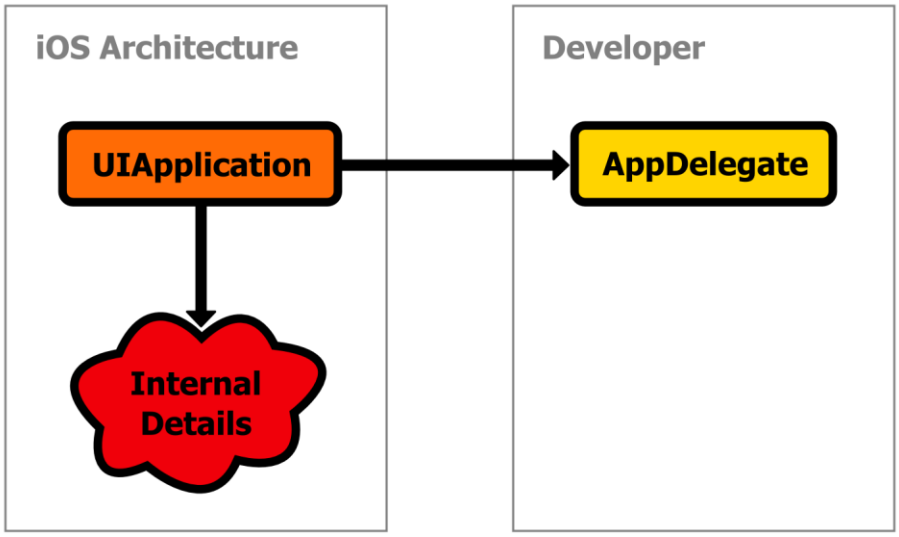

Figure 5: Using AppDelegate as the delegate object for UIApplication

Recall from *Objective-C Succinctly* that a protocol declares an arbitrary group of methods or properties that any class can implement. Since a delegate is designed to take control over an arbitrary set of tasks, this makes protocols the logical choice for representing delegates. The UIApplicationDelegate protocol declares the methods that a delegate for **`UIApplication`** should define, and we can see that our **`AppDelegate`** class adopts it in **AppDelegate.h**:

```
@interface AppDelegate : UIResponder <UIApplicationDelegate>
```

This is what formally turns our **`AppDelegate`** class into the delegate for the main **`UIApplication`** instance. If you open **AppDelegate.m**, you'll also see implementation stubs for the following methods:

```
- (BOOL)application:(UIApplication *)application
      didFinishLaunchingWithOptions:(NSDictionary *)launchOptions;
- (void)applicationWillResignActive:(UIApplication *)application;
- (void)applicationDidEnterBackground:(UIApplication *)application;
- (void)applicationWillEnterForeground:(UIApplication *)application;
- (void)applicationDidBecomeActive:(UIApplication *)application;
- (void)applicationWillTerminate:(UIApplication *)application;
```

These methods are called by **`UIApplication`** when certain events occur internally. For example, the **`application:didFinishLaunchingWithOptions:`** method is called immediately after the application launches. Let's take a look at how this works by adding an **`NSLog()`** call to some of these methods:

```
- (BOOL)application:(UIApplication *)application
        didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    NSLog(@"Application has been launched");
    return YES;
}
- (void)applicationDidEnterBackground:(UIApplication *)application {
    NSLog(@"Entering background");
}
- (void)applicationWillEnterForeground:(UIApplication *)application {
    NSLog(@"Entering foreground");
}
```

Now, when you compile the project and run it in the iOS Simulator, you should see the **`Application has been launched`** message as soon as it opens. You can click the simulator's home button to move the application to the background, and click the application icon on the home screen to move it back to the foreground. Internally, clicking the home button makes the **`UIApplication`** instance call **`applicationDidEnterBackground:`**:

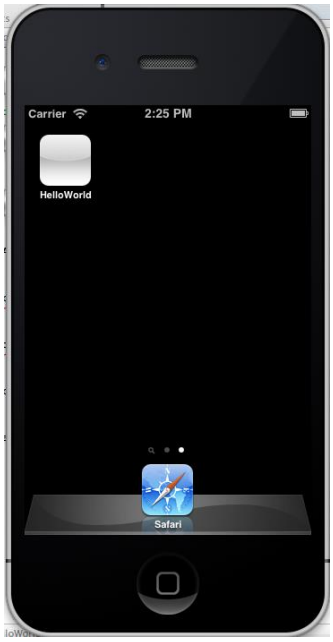

Figure 6: Moving the HelloWorld application to the background

This should display the following messages in Xcode's output panel:

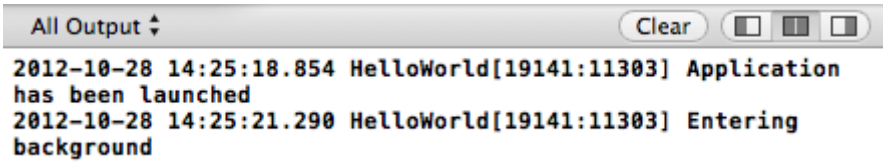

Figure 7: Xcode output after clicking the home button in the iOS Simulator

These **`NSLog()`** messages show us the basic mechanics behind an application delegate, but in the real world, you would write custom setup and cleanup code to these methods. For example, if you were creating a 3-D application with OpenGL, you would need to stop rendering content and free up any associated resources in the **`applicationDidEnterBackground:`** method. This makes sure that your application isn't hogging memory after the user closes it.

To summarize, our **`AppDelegate`** class serves as the practical entry point into our application. Its job is to define what happens when an application opens, closes, or goes into a number of other states. It accomplishes this by acting as a delegate for the **`UIApplication`** instance, which is the internal representation of the entire application.

ViewController.h and ViewController.m

Outside of the application delegate, iOS applications follow a model-view-controller (MVC) design pattern. The model encapsulates the application data, the view is the graphical representation of that data, and the controller manages the model/view components and processes user input.

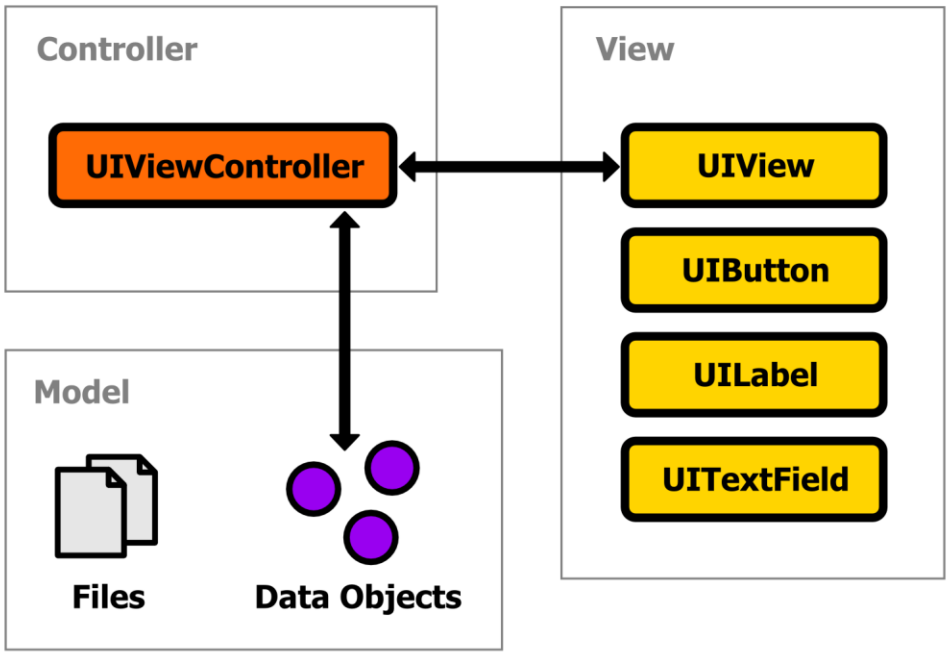

Figure 8: The model-view-controller pattern used by iOS applications

Model data is typically represented as files, objects from the CoreData framework, or custom objects. The application we're building in this chapter doesn't need a dedicated model component; we'll be focusing on the view and controller aspects of the MVC pattern until the next chapter.

View components are represented by the UIView class. Its **`UIButton`**, **`UILabel`**, **`UITextField`** and other subclasses represent specific types of user interface components, and **`UIView`** itself

can act as a generic container for all of these objects. This means that assembling a user interface is really just a matter of configuring **`UIView`** instances. For our example, the **`ViewController`** automatically creates a root **`UIView`** instance, so we don't need to manually instantiate one.

And, as you probably could have guessed, the **`ViewController`** class is the custom controller for our project. Its job is to lay out all of the UI components, handle user input like button clicks, text field input, etc., and update the model data when necessary. You can think of it as a scene manager.

Controllers typically inherit from the UIViewController class, which provide the basic functionality required of any view controller. In our HelloWorld program, the storyboard (discussed in the next section) automatically instantiates the root **`ViewController`** class for us.

While the **`AppDelegate`** is the *programmatic* entry point into the application, our **`ViewController`** is the *graphical* root of the project. The **`viewDidLoad`** method in **ViewController.m** is called after the root **`UIView`** instance is loaded. This is where we can create new user interface components and add them to the scene (we'll do this in a moment).

MainStoryboard.storyboard

The last file we need to take a look at is **MainStoryboard.storyboard**. This is a special type of file that stores the entire flow of your application and lets you edit it visually instead of programmatically. Selecting it in Xcode's **Project Navigator** will open up the **Interface Builder** instead of the normal source code editor, which should look something like this:

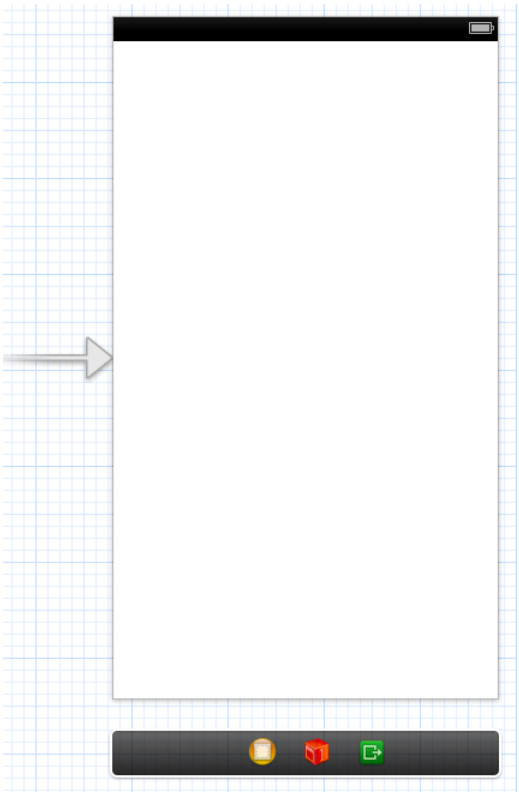

Figure 9: The Interface Builder of our HelloWorld project

The large white area is called a **scene**, and it represents a screen worth of content on the iPhone. This is what you're seeing when you compile and run the empty template, and it's where we can visually create a layout by dragging and dropping user interface components. The arrow pointing into the left of the scene tells us that this is the *root* scene for our app. Underneath it is the **dock**, which contains icons representing relevant classes and other entities. We'll see why this is important once we start making connections between graphical components and our custom classes.

Before we start adding buttons and text fields, let's take a moment to examine the left-most yellow icon in the dock. First, make sure the **Utilities** panel is open by toggling the right-most button in the **View** selection tab:

Figure 10: Displaying the Utilities panel (highlighted in orange)

Then, click the yellow icon in the dock to select it:

Figure 11: Selecting the View Controller icon

This icon represents the controller for the scene. For our project, this is an instance of the custom **`ViewController`** class. We can verify this by selecting the **Identity** inspector in the **Utilities** panel, which will display the class associated with the controller:

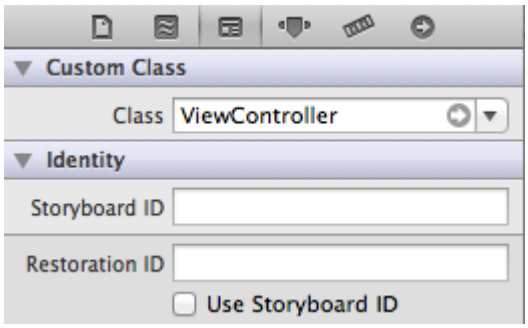

Figure 12: The Identity inspector in the Utilities panel

That **Class** field creates a connection between the storyboard's graphical interface and our source code. This is important to keep in mind when we start accessing user interface components from our classes.

It's also worth taking a look at the **Attributes inspector**, which is the next tab over in the **Utilities** panel:

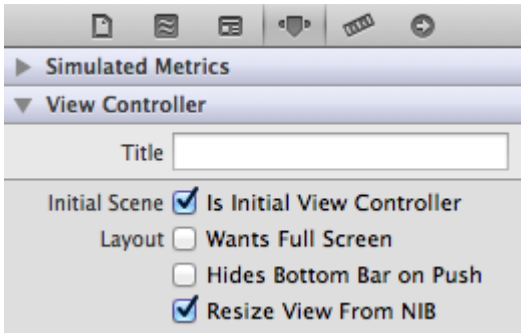

Figure 13: The Attributes inspector for the controller

That **Is Initial View Controller** check box is what makes this the *root* scene. Every app needs to have exactly one root scene, otherwise iOS won't know how to launch your application. If you clear the box, the arrow pointing into the scene will disappear, and you'll get the following message when you try to compile the project:

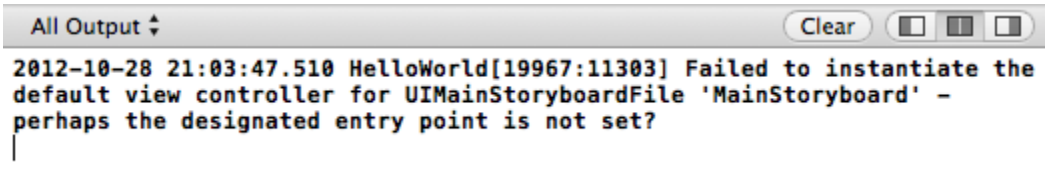

Figure 14: Error message from a missing root scene

Make sure **Is Initial View Controller** is selected before moving on.

Designing a User Interface

There are two ways to design the user interface of your application. You can either programmatically create graphical components by instantiating **`UIView`** and related classes in your source code, or you can visually design layouts by dragging components into the **Interface Builder**. This section takes a brief look at both methods.

Programmatic Layouts

We'll start with the programmatic method, as it shows us what's going on behind the scenes when we construct layouts using the Interface Builder. Remember that one of the main jobs of our **`ViewController`** is to manage UI components, so this is where we should create our layout. In **ViewController.m**, change the **`viewDidLoad`** method to the following:

```
- (void)viewDidLoad {
    [super viewDidLoad];

    UIButton *aButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
    [aButton setTitle:@"Say Hello" forState:UIControlStateNormal];
    aButton.frame = CGRectMake(100.0, 200.0, 120.0, 40.0);
    [[self view] addSubview:aButton];
}
```

First, we create a UIButton object, which is the object-oriented representation of a button. Then, we define its label using the **`setTitle:forState:`** method. The **`UIControlStateNormal`** constant tells the button to use this value for its "up" state. All graphical components use the **`frame`** property to determine their position and location. It accepts a **`CGRect`** struct, which can be created using the **`CGRectMake()`** convenience function. The previous sample tells the button to position itself at (x=100, y=200) and to use a width of 120 pixels and a height of 40 pixels. The most important part is the **`[[self view] addSubview:aButton]`** line. This adds the new **`UIButton`** object to the root **`UIView`** instance (accessed via the **`view`** property of our **`ViewController`**).

After compiling the project, you should see the button in the middle of the iOS Simulator:

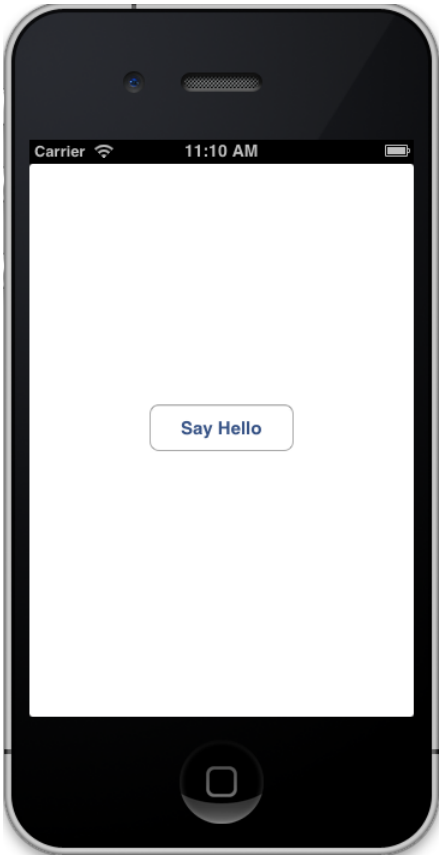

Figure 15: Programmatically creating a UIButton

You can click the button to see the default states, but actually making it *do* anything will take a bit more work. We'll learn how to do this in the Connecting Code with UI Components section.

Remember that **`UIButton`** is but one of many **`UIView`** subclasses that can be added to a scene. Fortunately, all the other user interface components can be managed using the same process: instantiate an object, configure it, and add it with the **`addSubview:`** method of the parent **`UIView`**.

Interface Builder Layouts

Creating components in the **Interface Builder** is a little bit more intuitive than the programmatic method, but it essentially does the same thing behind the scenes. All you need to do is drag a component from the *Object Library* onto a scene. The **Object Library** is located at the bottom of the **Utilities** panel, and it looks something like the following:

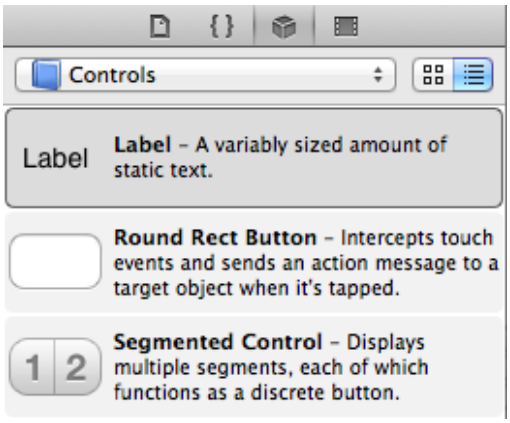

Figure 16: The Object library

In the previous screenshot, we opted to display only the user interface controls by selecting **Controls** from the drop-down menu. These are the basic graphical components for requesting input from the user.

Let's add another **Button**, along with a **Label** and a **Text Field** component by dragging the associated objects from the library onto the large white area representing the root scene. After they are on the stage, you can position them by dragging them around, and you can resize them by clicking the target component then dragging the white squares surrounding it. As you move the components around, you'll notice dotted guidelines popping up to help you align elements and create consistent margins. Try to arrange your layout to look something like Figure 17. The **Say Goodbye** button should be centered on both the x-axis and the y-axis:

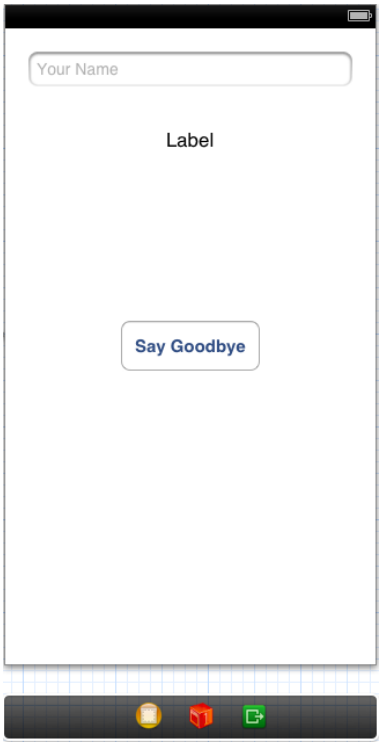

Figure 17: Laying out the Button, Label, and Text Field components

To change the text in the button, simply double-click it and enter the desired title (in this case, *Say Goodbye*). The **Interface Builder** also provides several other tools for editing the appearance and behavior of a component. For instance, you can set the placeholder text of our text field in the **Attribute** panel. Try changing it to **`Your Name`**:

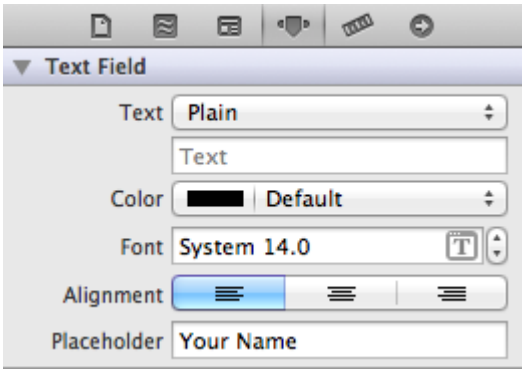

Figure 18: Defining the placeholder text

This will display some instructional text when the field is empty, which is usually a good idea from a user experience standpoint. When you compile your app, it should look something like this (note that we're still using the **Say Hello** button that we added in **`viewDidLoad`**):

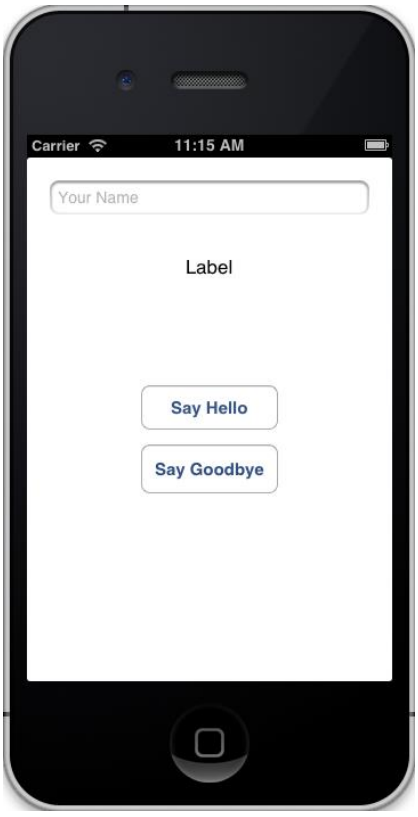

Figure 19: The HelloWorld App

If you click on the text field in the iOS Simulator, it will open the keyboard—just like you would expect from any iOS app. You'll be able to enter text, but you *won't* be able to dismiss the keyboard. We'll fix this issue in the *Delegates* portion of the next section. Until then, don't click the text field while testing the app.

As you might imagine, trying to lay out an interface using both the programmatic method and the Interface Builder can be a little confusing, so it's usually best to stick to one or the other for real-world applications.

These four components are all that we'll need for this project, but notice that we've only learned how to *add* components to a scene—they can't do anything useful yet. The next step is to get these user interface components to communicate with our code.

Connecting Code with UI Components

This section discusses the three most important types of connections between your source code and your user interface components: actions, outlets, and delegates. An **action** is a method that should be called when a particular event happens (e.g., when a user taps a button). An **outlet** connects a source code variable to a graphical component in the Interface Builder. We've already worked with delegates in the **`AppDelegate`** class, but this design pattern is also prevalent in the graphical aspects of iOS. It lets you control the behavior of a component from an arbitrary object (e.g., our custom **`ViewController`**).

Just like adding components to a layout, connecting them to your custom classes can be done either programmatically or through the Interface Builder. We'll introduce both methods in the *Actions* section that follows, but we'll rely on the Interface Builder for outlets and delegates.

Actions

Many interface controls use the target-action design pattern to react to user input. The **target** is the object that knows how to perform the desired action, and the **action** is just a method name. Both the target and the action are stored in the UI component that needs to respond to user input, along with an event that should trigger the action. When the event occurs, the component calls the action method on the specified target.

Programmatic Actions

The UIControl class from which **`UIButton`** inherits defines an **`addTarget:action:forControlEvents:`** method that lets you attach a target-action pair to an event. For example, we can make our *Say Hello* button display a greeting when we tap it by changing the **`viewDidLoad`** method in **ViewController.m** to the following:

```
- (void)viewDidLoad {
    [super viewDidLoad];

    UIButton *aButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
```

```
    [aButton setTitle:@"Say Hello" forState:UIControlStateNormal];
    aButton.frame = CGRectMake(100.0, 200.0, 120.0, 40.0);
    [[self view] addSubview:aButton];
    // Configure an action.
    [aButton addTarget:self
                action:@selector(sayHello:)
      forControlEvents:UIControlEventTouchUpInside];
}
```

This code tells the button to call the **sayHello:** method on **self** when the **UIControlEventTouchUpInside** event occurs. This event is triggered when the user releases a touch on the inside of the button. Other events are defined by the **UIControlEvents** enumeration contained in UIControl.

Of course, for the target-action pair in the previous code sample to work, we need to define the action method. The action should accept a single argument, which represents the user interface component that triggered the event. Add the following method to **ViewController.m**:

```
- (void)sayHello:(id)sender {
    NSLog(@"Hello, World!");
}
```

Now, when you compile the project and click the **Say Hello** button in the iOS Simulator, it should display **Hello, World!** in the Xcode output panel. If you needed to access the **UIButton** that triggered the event, you could do so through the **sender** argument. This could be useful, for example, when you want to disable the button after the user clicks it.

Interface Builder Actions

Configuring actions through the Interface Builder takes a couple more steps, but it's more intuitive when working with components that haven't been created programmatically. In this section, we'll use the Interface Builder to create an action for the **Say Goodbye** button.

Actions need to be publicly declared, so our first task is to add the action method in **ViewController.h**:

```
- (IBAction)sayGoodbye:(id)sender;
```

Notice the **IBAction** return type. Technically, this is just a **typedef** for **void**; however, using it as a return type makes Xcode and the Interface Builder aware of the fact that this is meant to be an *action*—not just an ordinary method. This is reflected by the small circle that appears next to the method in the source code editor:

```
@interface ViewController : UIViewController

- (IBAction)sayGoodbye:(id)sender;

@end
```

Figure 20: Xcode recognizing a method as an action

Next, we need to implement the action method. In **ViewController.m**, add the following method:

```
- (IBAction)sayGoodbye:(id)sender {
    NSLog(@"See you later!");
}
```

Instead of attaching it programmatically with **`addTarget:action:forControlEvents:`**, we'll use the Interface Builder to connect this method to the **Say Goodbye** button. Select the **MainStoryboard.storyboard** file to open the Interface Builder, and select the yellow **View Controller** icon in the dock:

Figure 21: Selecting the View Controller icon

Then, open the **Connections** inspector, which is the right-most tab in the **Utilities** panel:

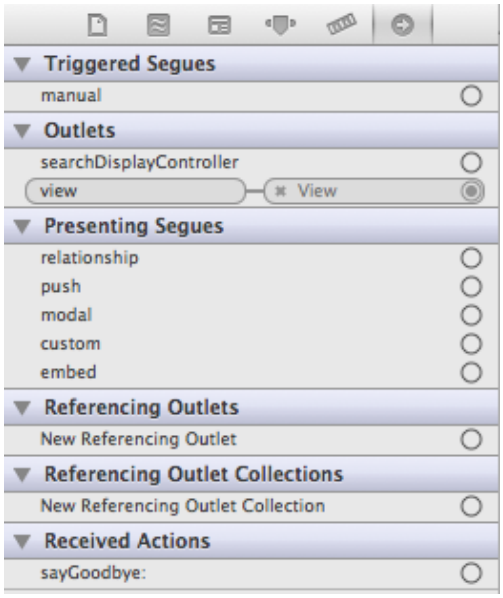

Figure 22: The Connections tab in the Utilities panel

This panel contains all of the relationships available to our **`ViewController`** class. Notice the **`sayGoodbye:`** method listed under **Received Actions**. This is only available because we used the **`IBAction`** return type in the method declaration.

To create a connection between the **`sayGoodbye:`** method and the **Say Goodbye** button, click the circle next to **`sayGoodbye:`** in the **Connections** panel, and then drag it to the button on the scene. You should see a blue line while you drag, as shown in the following figure:

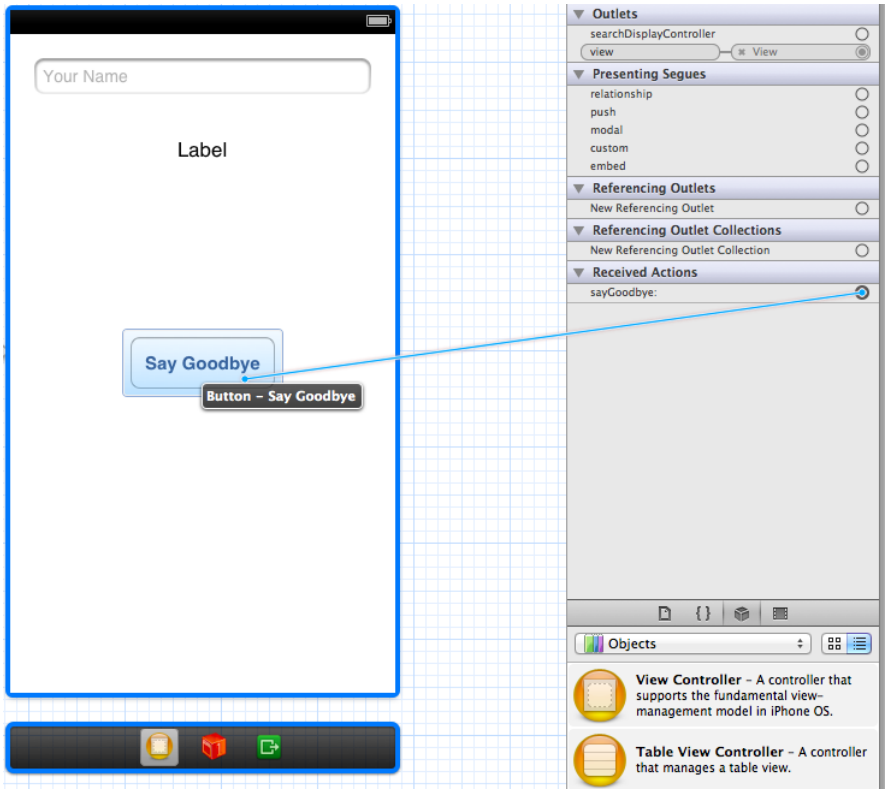

Figure 23: Connecting the sayGoodbye: method to a UIButton in Interface Builder

When you release over the button, a menu will pop up containing all of the available events that can trigger the action. It should look something like the following:

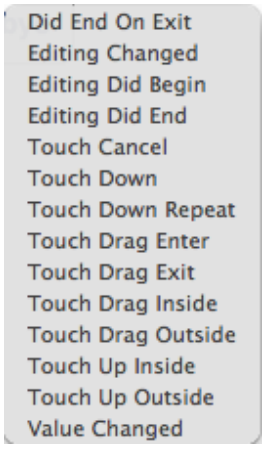

Figure 24: Selecting an event for the action

Select **Touch Up Inside**, which is the equivalent of the **`UIControlEventTouchUpInside`**

enumerator we used in the previous section. This creates a target-action connection between the **`ViewController`** and the **`UIButton`** instance. Essentially, this is the exact same **`addTarget:action:forControlEvents:`** call we used for the **Say Hello** button in the previous section, but we did it entirely through the Interface Builder. You should now be able to compile the project and click the **Say Goodbye** button to display the **`See you later!`** message in the output panel.

Note that instead of being stored as source code, the connection we just created is recorded in the *storyboard*. Again, it can be confusing to maintain actions in both source code and the storyboard, so it's usually best to stick to one method or the other in real-world applications. Typically, if you're creating your layout in the Interface Builder, you'll want to create your connections there as well.

Outlets

An outlet is a simpler type of connection that links a source code variable with a user interface component. This is an important ability, as it lets you access and manipulate properties of the storyboard from custom classes. Outlets always originate from the custom class and are received by a UI component in the Interface Builder. For example, this section creates an outlet from a variable called **`messageLabel`** to a **`UILabel`** component in the storyboard. This can be visualized as follows:

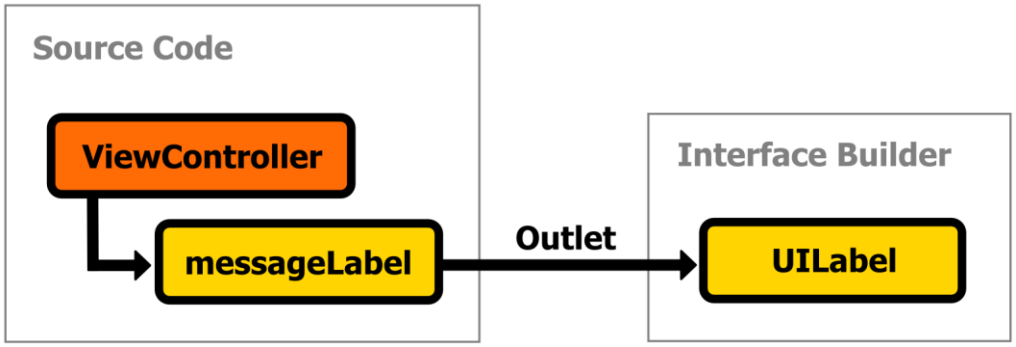

Figure 25: Creating an outlet from the ViewController class to a label component

To create an outlet, we first need to declare the property that will be associated with the UI component. Outlets are typically configured in a controller, so open **ViewController.h** and add the following property:

```
@property (weak, nonatomic) IBOutlet UILabel *messageLabel;
```

This looks like any other property declaration, except for the new **`IBOutlet`** qualifier. Just like **`IBAction`**, this designates the property as an outlet and makes it available through the Interface Builder, but doesn't affect the variable itself. Once we set up the connection, we can use **`messageLabel`** as a direct reference to the **`UILabel`** instance that we added to the storyboard.

But before we do that, we need to synthesize the accessor methods in **ViewController.m**:

```
@synthesize messageLabel = _messageLabel;
```

Back in **`MainStoryboard.storyboard`**, select the yellow **View Controller** icon again and take a look at the **Connections** inspector:

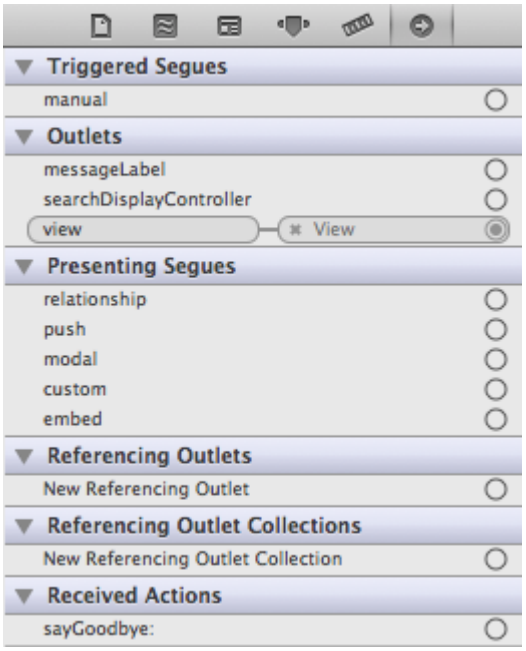

Figure 26: The Connections inspector after adding messageLabel to ViewController

Notice how the **`messageLabel`** property we just created appears in the **Outlets** listing. We can now connect it to a user interface component just like we did with the button action in the previous section. Click and drag from the circle next to **`messageLabel`** to the **`UILabel`** in the scene, like so:

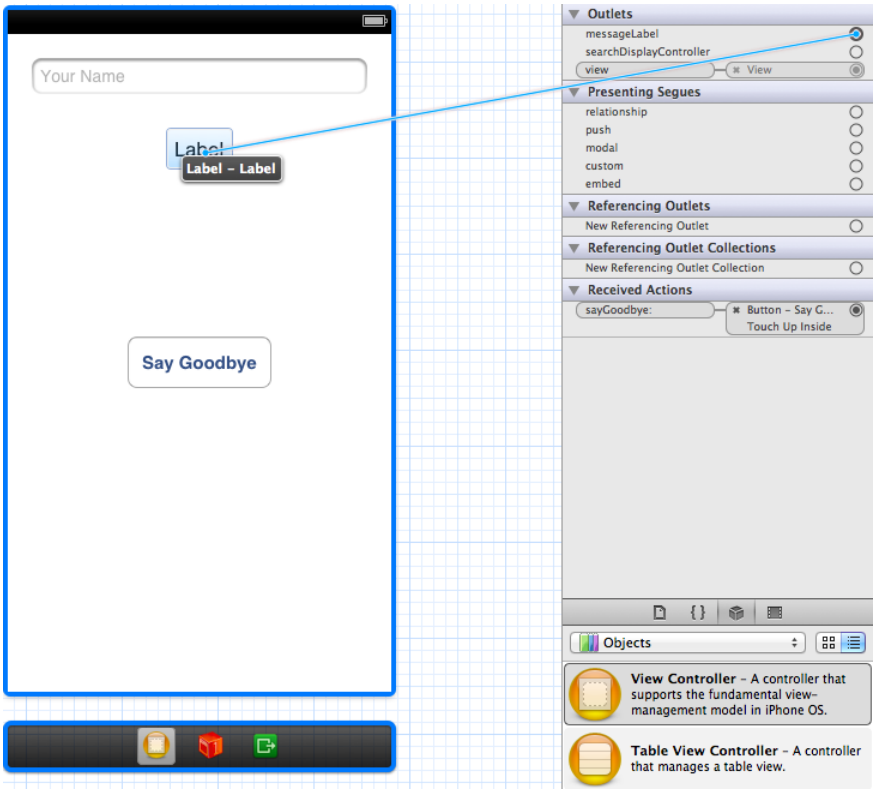

Figure 27: Linking the messageLabel with the UILabel instance

When you release the mouse, the outlet will be created, and you can use **`messageLabel`** to set the properties of the UILabel instance in the Interface Builder. As an example, try changing the text and color of the label in the **`sayHello:`** and **`sayGoodbye:`** methods of **ViewController.m**:

```
- (void)sayHello:(id)sender {
    _messageLabel.text = @"Hello, World!";
    _messageLabel.textColor = [UIColor colorWithRed:0.0
                                              green:0.3
                                               blue:1.0
                                              alpha:1.0];
}

- (IBAction)sayGoodbye:(id)sender {
    _messageLabel.text = @"See you later!";
    _messageLabel.textColor = [UIColor colorWithRed:1.0
                                              green:0.0
                                               blue:0.0
                                              alpha:1.0];
}
```

Now, when you compile the project and click the buttons, they should display different messages in the label. As you can see, outlets are a necessary tool for updating the interface in reaction to user input or changes in the underlying data model.

Before we continue on to delegate connections, we need to set up another outlet for the **`UITextField`** that we added in the Interface Builder. This will be the exact same process as the **`UILabel`**. First, declare the property and synthesize its accessors:

```
// ViewController.h
@property (weak, nonatomic) IBOutlet UITextField *nameField;
// ViewController.m
@synthesize nameField = _nameField;
```

Then, open the Interface Builder, select the yellow **View Controller** icon in the dock, and make sure the **Connections** inspector is visible. To create the connection, click and drag from the circle next to **`nameField`** to the text field component in the scene. In the next section, this outlet will let us access the text entered by the user.

Delegates

The delegate design pattern serves the same purpose for UI components as it does for the **`AppDelegate`** discussed earlier: it allows a component to transfer some of its responsibilities to an arbitrary object. For our current example, we're going to use the **`ViewController`** class as a delegate for the text field we added to the storyboard. As with the **`AppDelegate`**, this allows us to react to important text field events while hiding the complexities of its internal workings.

First, let's turn the **`ViewController`** class into a formal delegate for the text field. Remember that the delegate design pattern is implemented through protocols, so all we have to do is tell **ViewController.h** to adopt the **`UITextFieldDelegate`** protocol, like so:

```
@interface ViewController : UIViewController <UITextFieldDelegate>
```

Next, we need to connect the text field and the **`ViewController`** class using the Interface Builder. This connection flows in the opposite direction as the outlets we created in the previous section, so instead of dragging from the **`ViewController`** to a UI component, we need to drag from the text field to the **`ViewController`**. In the Interface Builder, select the text field component and open the **Connections** panel. You should see a **delegate** field under the **Outlets** section:

Figure 28: The beginning of the Connections panel for the Text Field component

To create the delegate connection, drag from the circle next to **delegate** to the yellow **View Controller** icon in the dock:

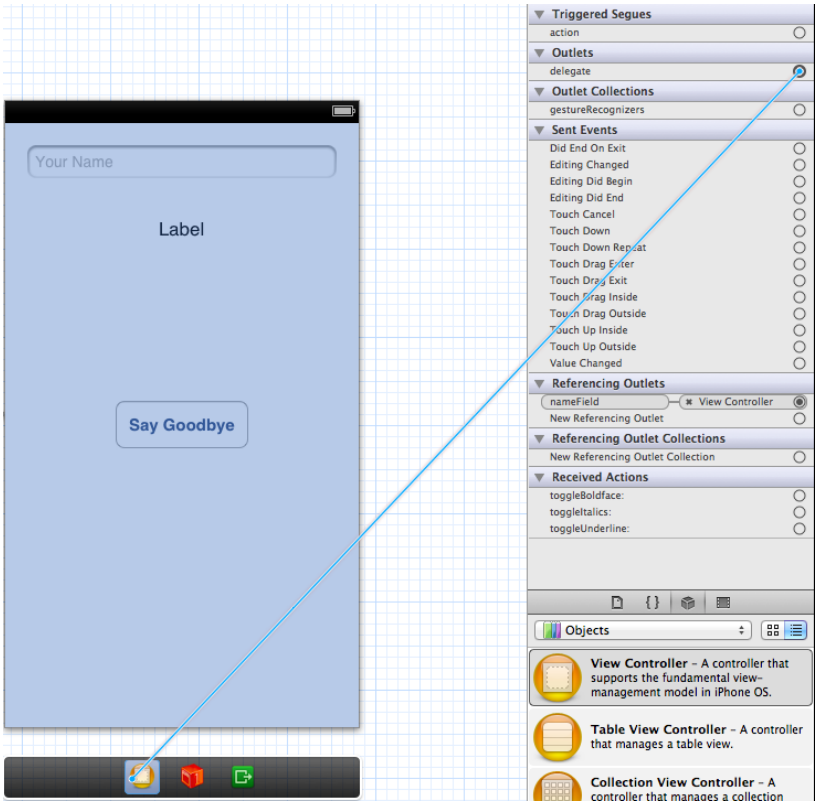

Figure 29: Creating a delegate connection from the Text Field to the View Controller

Now, the **`ViewController`** can control the behavior of the text field by implementing the methods defined in [UITextFieldDelegate]. We're interested in the **`textFieldShouldReturn:`** method, which gets called when the user clicks the **Return** button on the keyboard. In **ViewController.m**, implement the method as follows:

```
- (BOOL)textFieldShouldReturn:(UITextField *)textField {
    self.name = textField.text;
    if (textField == self.nameField) {
        [textField resignFirstResponder];
    }
    return YES;
}
```

This saves the value the user entered (**`textField.text`**) into the **`name`** property after they press the **Return** button. Then, we make the keyboard disappear by removing focus from the text field with the **`resignFirstResponder`** method. The **`textField == self.nameField`** conditional is a best practice to make sure that we're working with the correct component (this isn't actually necessary unless the **`ViewController`** is a delegate for multiple text fields). Note that we still have to declare that **`name`** field in **ViewController.h**:

```
@property (copy, nonatomic) NSString *name;
```

It's always better to isolate the model data in dedicated properties in this fashion rather than rely directly on the values stored in UI components. This makes sure that they will be accessible even if the UI component has been removed or altered in the meantime. Our last step is to use this new **`name`** property to personalize the messages in **`sayHello:`** and **`sayGoodbye:`**. In **ViewController.m**, change these two methods to the following:

```
- (void)sayHello:(id)sender {
    if ([self.name length] == 0) {
        self.name = @"World";
    }
    _messageLabel.text = [NSString stringWithFormat:@"Hello, %@!"
                                                    self.name];
    _messageLabel.textColor = [UIColor colorWithRed:0.0
                                              green:0.3
                                               blue:1.0
                                              alpha:1.0];
}

- (IBAction)sayGoodbye:(id)sender {
    if ([self.name length] == 0) {
        self.name = @"World";
    }
    _messageLabel.text = [NSString stringWithFormat:@"See you later, %@!",
                                                    self.name];
    _messageLabel.textColor = [UIColor colorWithRed:1.0
                                              green:0.0
                                               blue:0.0
                                              alpha:1.0];
}
```

You should now be able to compile the application, edit the text field component, dismiss the keyboard, and see the resulting value when you click the **Say Hello** and **Say Goodbye** buttons.

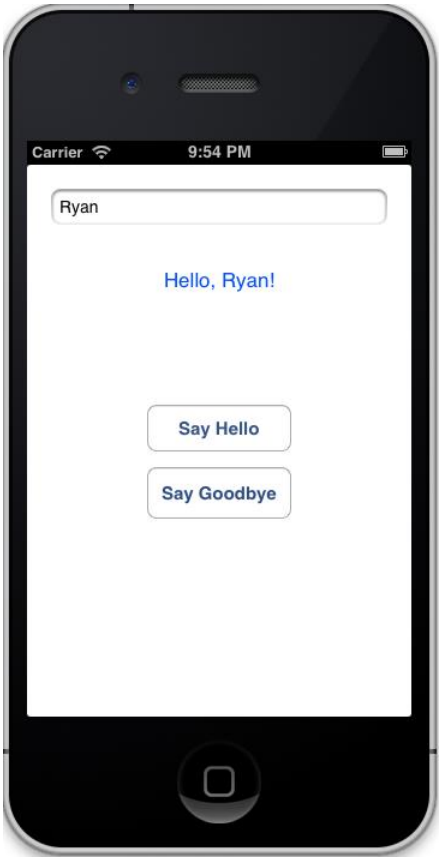

Figure 30: Implementing the Text Field component

Summary

This chapter introduced the fundamentals of iOS development. We learned about the basic file structure of a project: the main file, the application delegate, the custom view controller, and the storyboard. We also designed a layout by programmatically adding components to the stage, as well as by visually editing components in the Interface Builder. And, to enable our code to communicate with buttons, labels, and text fields in the storyboard, we created action, outlet, and delegate connections using the Interface Builder.

That was a lot of work to create such a simple application, but we now have nearly all the skills we need to build real-world applications. Once we understand the basic workflow behind connecting code with user interface elements and capturing user input, all that's left is exploring the capabilities of individual components/frameworks and making them all work together.

The next chapter fleshes out some of the topics we glossed over in the previous example by walking through a more complex application. We'll learn about segues for transitioning between scenes, and we'll also have the opportunity to discuss the model-view-controller pattern in more detail.

Chapter 2 Multi-Scene Applications

The previous chapter introduced the basic workflow of iOS application development, but we worked within the confines of a single-view application. Most real-world applications, however, require multiple scenes to present data hierarchically. While there are many types of organizational patterns for managing multi-scene apps, this chapter looks at one of the most common patterns: the master-detail application.

The minimal master-detail application consists of a “master” scene, which presents a list of data items to choose from, and a “detail” scene, which displays an item’s details when the user selects it from the master scene. Open the Mail app on your iPhone and you’ll find a good example of a master-detail application. The inbox lists your messages, making it the master scene, and when you select one of them, a detail scene is used to display the contents of the message, the sender, any attachments, etc.

For the most part, multi-scene applications use the same workflow discussed in the previous chapter. We’ll still create layouts by adding UI components through the Interface Builder and connect them to our code with actions, outlets, and delegates. However, having multiple scenes means that we’ll have multiple view controllers, so we’ll need to use the new **`UINavigationController`** class to organize their interaction. We’ll also learn how to configure scene transitions using segues.

Creating a Master-Detail Project

The example project for this chapter will be a simple contact list that lets users manage a list of their friends, along with their respective contact information. To create the example project, select **File** > **New** > **Project** and choose the **Master-Detail Application**. This will give us the opportunity to explore new navigation components and organizational structures, as well as how to handle transitions from one scene to another.

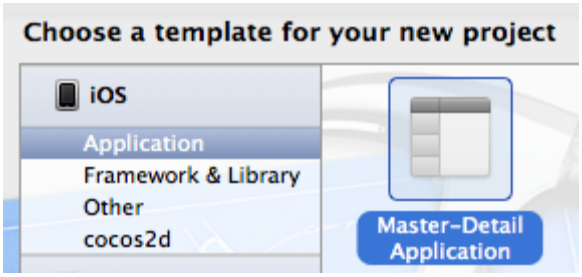

Figure 31: Creating a Master-Detail Application template

Use *FriendList* for the **Product Name** field, whatever you like for the **Organization Name**, and *edu.self* for the **Company Identifier**. Like the previous app, make sure that **iPhone** is the selected **Device** and **Use Storyboards** and **Use Automatic Reference Counting** are selected:

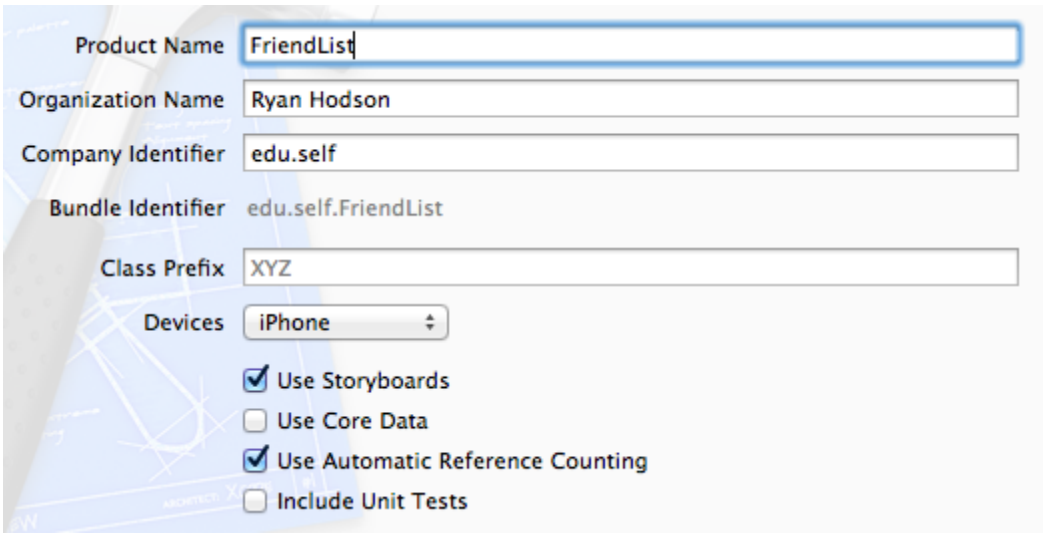

Figure 32: Configuring the project

You can save the project wherever you like.

Template Overview

We'll be building on the template's existing code, so let's take a quick look at the default application. Click the **Run** button in the upper-left corner of Xcode or press Cmd+R to compile the application and launch it in the iOS Simulator. You should see an empty list entitled **Master** with an **Edit** button and an **Add** button (a plus sign) in the navigation bar. Clicking the **Add** button will insert a new item into the list, and selecting that item will transition to the detail scene. Both scenes are shown in the following figure.

Figure 33: The template's default master and detail scenes

The default data items used by the template are dates, but we're going to change the master scene to display a list of names and the detail scene to display their contact information.

We'll be discussing the details behind each source file as we build up the example project, but it will help to have a basic overview of the default classes before we start editing them.

The Application Delegate

As in the previous chapter, the **`AppDelegate`** class lets you react to important events in the application's life cycle. We don't need any custom startup behavior for our friend list application, so we won't be editing this class at all.

The View Controllers

Instead of a single **`ViewController`**, this template has two view controller classes: a **`MasterViewController`** and a **`DetailViewController`**. These manage the master scene and detail scene, and their **`viewDidLoad`** methods serve as the entry point into their respective scenes. The **`MasterViewController`**'s **`viewDidLoad`** method should look like the following:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    self.navigationItem.leftBarButtonItem = self.editButtonItem;

    UIBarButtonItem *addButton = [[UIBarButtonItem alloc]
        initWithBarButtonSystemItem:UIBarButtonSystemItemAdd
                            target:self
                            action:@selector(insertNewObject:)];
    self.navigationItem.rightBarButtonItem = addButton;
}
```

This creates the **Edit** and **Add** buttons that you see at the top of the master scene, and it sets the **`insertNewObject:`** method as the action for the latter. The **`insertNewObject:`** method adds a new **`NSDate`** instance to the private **`_objects`** variable, which is a mutable array containing the master list of data items, and all of the methods after the **`#pragma mark - Table View`** directive control how that list is displayed in the scene. The **`prepareForSegue:sender:`** method is called before transitioning to the detail scene, and it is where the necessary information is transferred from the master scene to the detail scene.

The **`DetailViewController`** class is a little bit simpler. It just declares a **`detailItem`** property to store the selected item and displays it through the **`detailDescriptionLabel`** outlet. We're going to be changing this default implementation to display a person's contact information.

The Storyboard

The storyboard is perhaps the most drastic change from the previous example. If you open **MainStoryboard.storyboard**, you should see the following:

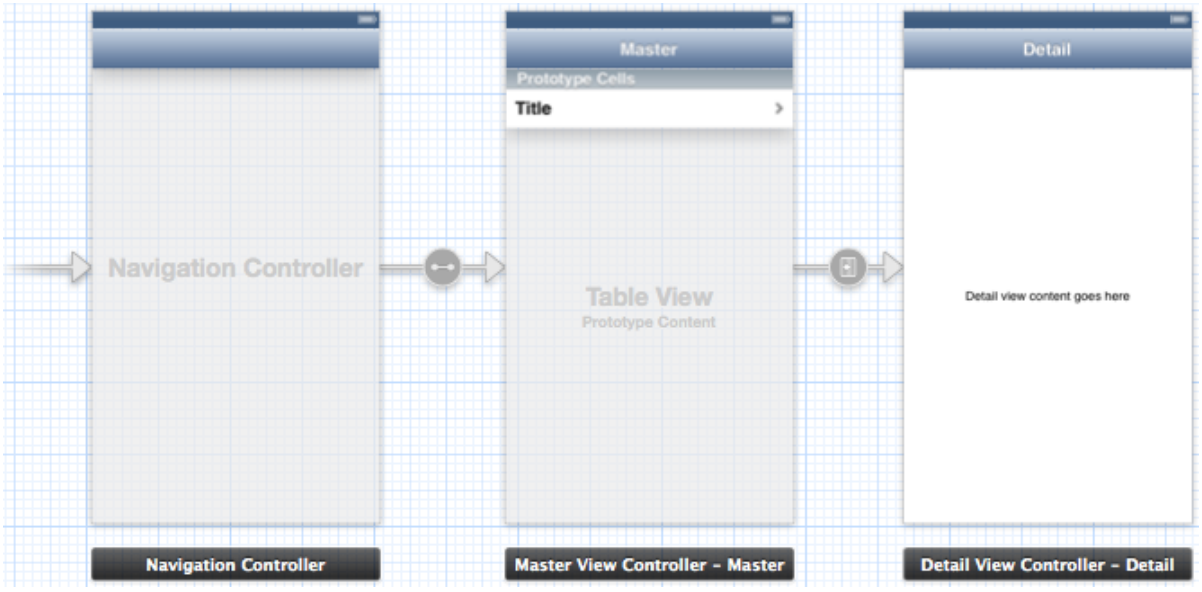

Figure 34: The template's default storyboard

Instead of a single view controller, the Interface Builder now manages *three* controllers. This might seem odd considering our application only has *two* scenes, but both the **`MasterViewController`** and the **`DetailViewController`** are embedded in a **`UINavigationController`** instance. This navigation controller is why we see a navigation bar at the top of the app, and it's what lets us navigate back and forth between the master and detail scenes. We'll talk more about configuring navigation controllers throughout the chapter.

This template should also clarify why the **MainStoryboard.storyboard** file is called a "storyboard"— it visualizes not only the scenes themselves, but the flow between those scenes. As in the previous chapter, the arrow pointing into the navigation controller shows that it is the root controller. But, we also see another arrow from the navigation controller to the **`MasterViewController`** and from the **`MasterViewController`** to the **`DetailViewController`**. These arrows define the relationships and transitions between all of the view controllers in an application.

The Model Data

Unlike the previous chapter, this application will use a dedicated class to represent its model data. We'll use the **`Person`** class to store the contact information of each friend. In Xcode, create a new file, select **Objective-C class**, and enter *Person* for the **Class** field, like so:

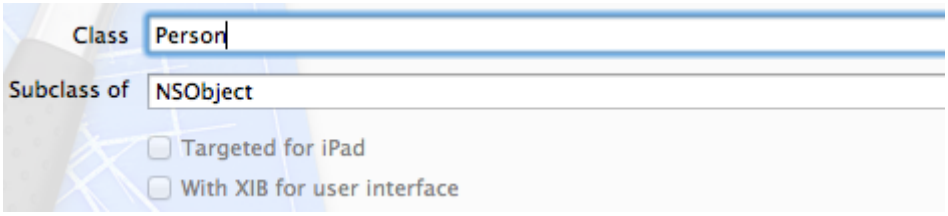

Figure 35: Creating the Person class

Next, we need to declare a few properties to record the name, organization, and phone number of each contact. Open **Person.h**, and change it to the following:

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

@interface Person : NSObject

@property (copy, nonatomic) NSString *firstName;
@property (copy, nonatomic) NSString *lastName;
@property (copy, nonatomic) NSString *organization;
@property (copy, nonatomic) NSString *phoneNumber;


@end
```

Of course, we also need to synthesize these properties in **Person.m**:

```
#import "Person.h"

@implementation Person

@synthesize firstName = _firstName;
@synthesize lastName = _lastName;
@synthesize organization = _organization;
@synthesize phoneNumber = _phoneNumber;

@end
```

That's all we need to represent the data behind our application. Instances of this class will be passed around between the **`MasterViewController`** and **`DetailViewController`** scenes, which will display them using various UI components.

The Master Scene

Next, we'll configure the master scene to display a list of **`Person`** objects. Defining a scene's behavior requires careful interaction between the underlying view controller's source code and the visual representation in the Interface Builder. Before we do any coding, let's take a closer look at the template's master scene in the storyboard.

Relationships

In our storyboard, a **relationship** defines the connection between a navigation controller and another scene. The Interface Builder visualizes the relationship as an arrow from the navigation controller to the other scene with a link icon on top of it. Selecting this icon will highlight the navigation controller, as shown in the following screenshot:

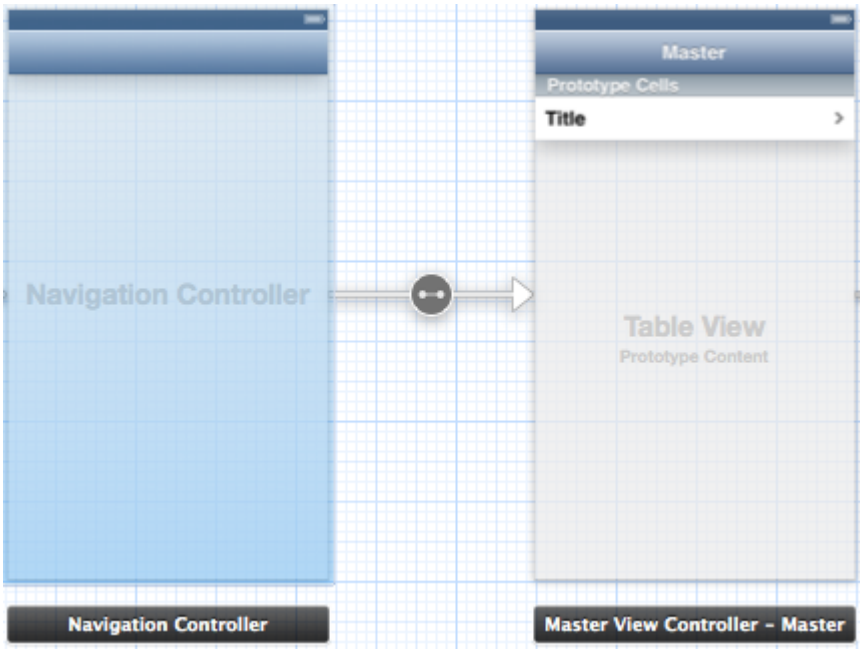

Figure 36: The relationship between the root navigation controller and the master view controller

The template set up this relationship for us, but it's important to be able to configure it on your own. So, go ahead and delete the navigation controller by selecting it and pressing Delete. To re-create the relationship, select the yellow **View Controller** icon in the master view controller, and then navigate to **Editor** in Xcode's menu bar and select **Embed In** > **Navigation Controller**. A new navigation controller should appear, and you should be back to where you started.

It's important to understand that the relationship arrow does *not* signify a transition between the navigation controller and the master controller. Rather, embedding our master scene into a navigation controller in this fashion creates a view controller *hierarchy.* It says that the master scene *belongs* to the navigation controller. This makes it possible to switch between scenes using the navigation controller's built-in transitions and navigation buttons. For example, the **Master** button that appears at the top of the detail scene is automatically added by the navigation controller:

Figure 37: The built-in navigation button provided by the navigation controller

The built-in functionality for switching between scenes makes navigation controllers an easy way to configure the flow of complex applications. The next section discusses how to define transitions between a navigation controller's scenes.

Segues

A **segue** represents a transition from one scene to another. Like relationships, it is visualized as an arrow from the source scene to the destination scene, but it uses a different icon. Notice that

when you click the segue icon, only a single table cell is highlighted. This tells us that the segue is attached to individual table cells instead of the entire master scene.

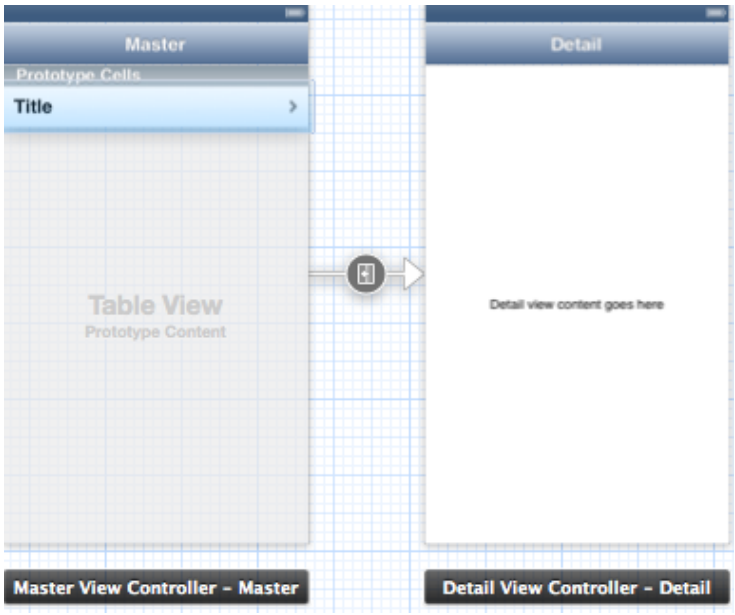

Figure 38: The push segue from the master scene to the detail scene

Again, our template created this segue for us, but it's important to be able to create one from scratch. So, select the segue icon and press Delete to remove it from the storyboard. To re-create it, control-drag from the table cell to the detail scene.

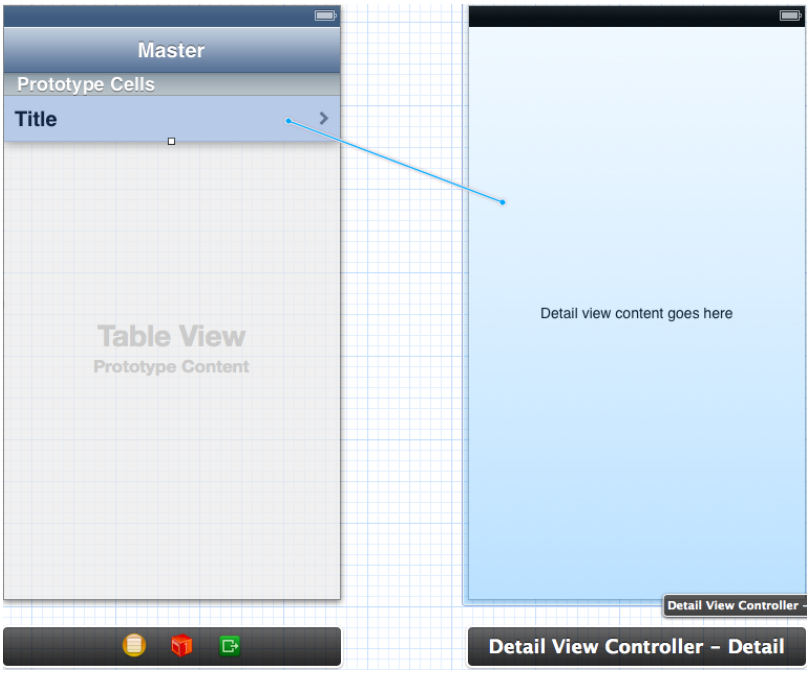

Figure 39: Control-dragging from the master's table cell to the detail scene

This will open a menu prompting you for the **Selection Segue/Accessory Action** type. We want our segue to occur when the user selects the table cell, so choose **push** under the **Selection Segue** group.

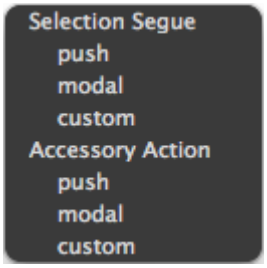

Figure 40: Selecting the type of segue to create

The parent **`UINavigationController`** manages its scenes through a **navigation stack**, and its **`pushViewController:animated:`** and **`popViewControllerAnimated:`** methods let you add or remove view controller instances from the stack. For example, pushing a detail view controller object onto the navigation stack is how you drill down to the detail scene, and clicking the **Master** button in the detail scene's navigation bar pops it from the navigation stack. Selecting **push** from the menu in Figure 40 tells the segue to call the **`pushViewController:animated:`** method to transition from the master scene to the detail scene.

In addition to a type, each segue must also have a unique **identifier** so that it can be accessed from your source code. You can edit a segue's ID by selecting the segue icon and opening the **Attributes inspector** panel. Our segue should have an identifier of **showDetail**, and you should also see the **Push** segue type in the **Style** field:

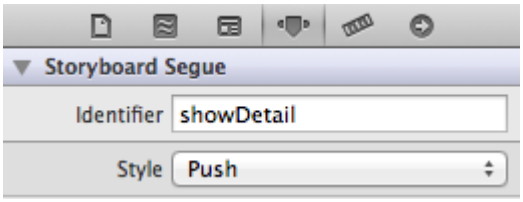

Figure 41: The Attributes inspector for the master-detail segue

The other **Style** option is **Modal**, which presents another scene *on top of* an existing scene, completely independent of the parent navigation controller. You should leave this segue's **Style** as **Push** (we'll create a modal segue toward the end of this chapter).

Tables

One of the main differences between our master scene and the **`ViewController`** from the previous chapter is the fact that it inherits from **`UITableViewController`** instead of **`UIViewController`**. A table view controller manages a **`UITableView`** instance. Table views are composed of a single column of rows, possibly grouped into sections. This makes them well suited to presenting lists of data.

Since table views are graphical containers, it can be hard to select them in the scene editor. The easiest way to select it is from the **document outline** to the left of the scene editor. The document outline is a tree containing all the elements managed by the Interface Builder, and you should find a **Table View** item under the **Master View Controller**, as shown in the following figure:

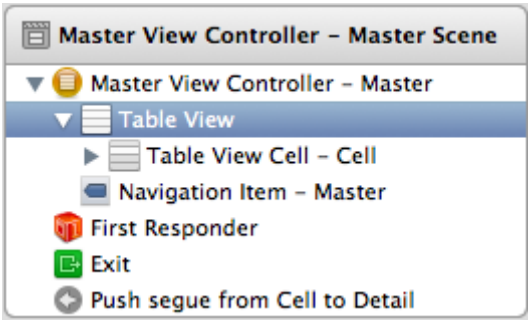

Figure 42: Selecting the UITableView instance from the document outline

When you select the table view, everything under the navigation bar in the master scene should be highlighted in the scene builder. This gives you the chance to edit the table view properties in the **Attributes inspector**. The most important option is the **Content** field, which determines how you will interact with the table from your code:

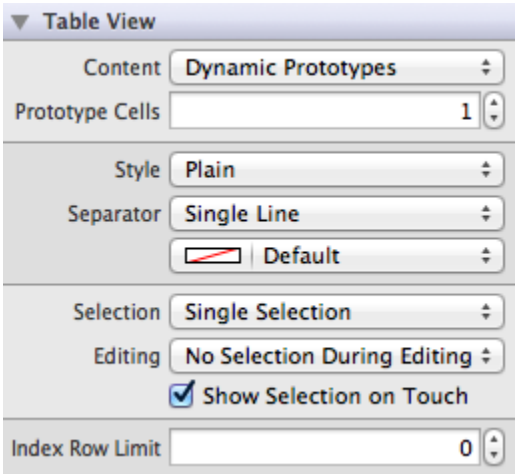

Figure 43: The Attributes inspector for the master scene's table view

If you set the **Content** field to **Dynamic Prototypes**, you can create new cells by duplicating a prototypical cell designed in the Interface Builder. *Static cells*, on the other hand, cannot be duplicated, resulting in a static table. This means that you should use **Dynamic Prototypes** when you want to insert or delete rows on the fly, and use **Static Cells** when your table always shows the same amount of information. Keep the master scene's table dynamic. We'll use a static table for the detail scene.

When you use prototype cells, you need to give each prototype a unique identifier so that it can be accessed from your source code (just like a segue ID). To edit a prototype cell's ID, select the cell in either the scene editor or the interface builder and open the **Attributes inspector**.

The identifier for that particular prototype can be set in the **Identifier** field, as shown in the following figure. Since we're only going to have one prototypical cell in this application, you can leave the default **Cell** value, but for real applications you should give each prototype a descriptive identifier.

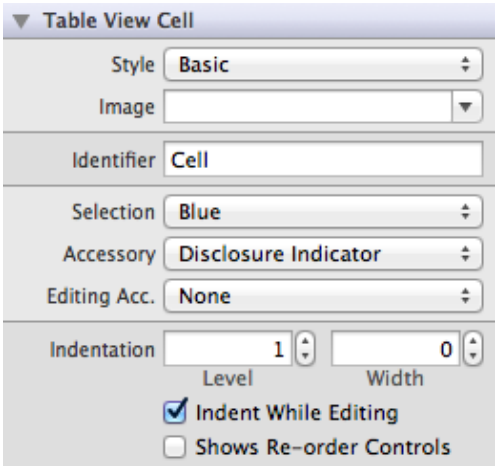

Figure 44: The Attributes inspector for the prototype table cell

It's also worth taking a look at the **Connections inspector** for the **`UITableView`** (not the prototype cell). You should see a **`dataSource`** and a **`delegate`** outlet, and both of them should specify the **`MasterViewController`** class for their destination.

Figure 45: The outlet connections for the master scene's table view

A table view's data source is a special kind of delegate that provides the information for each row in the table. In addition to the raw data, a table view delegate is necessary to define the behavior of the table and the appearance of each row. As with application and text field delegates, these are implemented through protocols called UITableViewDataSource and UITableViewDelegate, respectively.

In this case, the **`MasterViewController`** class acts as both the data source and the delegate, which is why the master-detail template included methods like **`tableView:cellForRowAtIndexPath:`** and **`tableView:canEditRowAtIndexPath:`** in **MasterViewController.m**. In the next section, we'll alter these methods to change the appearance of the friend list.

Coding the Master View Controller

Now that we have a better handle on what's going on in the storyboard, we're ready to start customizing our **`MasterViewController`** class. Right now, the master scene is displaying a list of **`NSDate`** objects, but we want to change those to **`Person`** objects. Of course, this means we'll need access to the **`Person`** class, so import the header in **MasterViewController.m**:

```
#import "Person.h"
```

Remember that the **viewDidLoad:** method tells the master scene's **Add** button to call the **insertNewObject:** method whenever the user taps it. Instead of adding a date object to the **_objects** array, we need **insertNewObject:** to add a **Person** object. Change it to the following:

```
- (void)insertNewObject:(id)sender {
    if (!_objects) {
        _objects = [[NSMutableArray alloc] init];
    }
    Person *friend = [[Person alloc] init];
    friend.firstName = @"<First Name>";
    friend.lastName = @"<Last Name>";
    friend.organization = @"<Organization>";
    friend.phoneNumber = @"<Phone Number>";
    [_objects insertObject:friend atIndex:0];
    NSIndexPath *indexPath = [NSIndexPath indexPathForRow:0 inSection:0];
    [self.tableView insertRowsAtIndexPaths:@[indexPath]
                        withRowAnimation:UITableViewRowAnimationAutomatic];
}
```

This instantiates a new **Person** object and populates it with some dummy values, and then adds it to the front of the **_objects** array with **insertObject:atIndex:**. The **NSIndexPath** instance is a simple data object representing the index of a particular cell, and the **insertRowsAtIndexPaths:withRowAnimation:** adds a new cell at the specified location.

Notice that this last method doesn't actually create the new cell—it just adds an item to the **_objects** array and tells the table that it should have one more row in it. This prompts the table to create a new cell, which is prepared by the **tableView:cellForRowAtIndexPath:** data source delegate method. It should look like the following:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    UITableViewCell *cell = [tableView
                             dequeueReusableCellWithIdentifier:@"Cell"
                             forIndexPath:indexPath];
    Person *friend = _objects[indexPath.row];
    cell.textLabel.text = [NSString stringWithFormat:@"%@ %@",
                           friend.firstName, friend.lastName];
    return cell;
}
```

This method is called every time the table needs to render a given cell, and it should return a **UITableViewCell** object representing the corresponding row. First, we fetch a prototype cell using the identifier defined in the storyboard, and then we use the **NSIndexPath** instance to find

the associated **Person** object. Finally, we display the person's name through the **textLabel** property of the cell.

Now, you should be able to add, view, and delete **Person** objects from the master scene:

Figure 46: Adding a Person object to the master scene

That covers the basic list functionality for the master scene, but we still have one more task before we move on to the detail scene. When a user selects one of the items in the master list, we need to pass that object to the detail scene.

Remember that the **UINavigationController** and the push segue handles the transition for us, but it gives us the opportunity to send data from the source view controller to the destination view controller by calling the **prepareForSegue:sender:** method right before it switches to the detail view. Change **prepareForSegue:sender:** in **MasterViewController.m** to the following (the only real change is to use a **Person** object instead of an **NSDate** instance):

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    if ([[segue identifier] isEqualToString:@"showDetail"]) {
        NSIndexPath *indexPath = [self.tableView indexPathForSelectedRow];
        Person *friend = _objects[indexPath.row];
        [[segue destinationViewController] setDetailItem:friend];
    }
}
```

This method is how we pass data between the master scene and the detail scene. It is called for *every* segue associated with a particular controller, so our first step is to check the segue ID, which was defined in the Interface Builder. Then, we use the **`indexPathForSelectedRow`** method to get the index of the selected row (aren't Objective-C naming conventions great), and we use that index to find the corresponding data item from the **`_objects`** array. Finally, we pass this object off to the detail scene by setting its **`detailItem`** property.

Figure 47: Selecting a Person object from the master scene

Now when you select an item from the master list, you should see a **`Person`** object instead of an **`NSDate`** instance. The default detail scene uses the **`description`** method to convert the object to a string, which is why we see a memory address in Figure 47 instead of any meaningful information. We'll change that in the next section.

To summarize our master scene: we have a relationship connection that embeds it in a **`UINavigationController`** instance, a segue defining the transition to the detail scene, a prototype cell that we use as a template for new table rows, an **Add** button that adds dummy instances to the master list of data items, and a **`prepareForSegue:sender:`** method that passes the selected item off to the detail scene.

The Detail Scene

Next, we need to configure the detail scene to display the selected friend. A single **`Person`** object always has the same amount of information (a name, an organization, and a phone number), so we'll use three static cells to format the output instead of dynamic prototypes. Just

like the master scene, we're going to configure the Interface Builder first, and then code the functionality after we have the UI components laid out.

Switching to a Table View Controller

The master-detail template uses a plain **`ViewController`** for the detail scene, so our first task is to replace it with a **`UITableViewController`**. In the Interface Builder, select the detail scene and press Delete to remove it from the storyboard. Then, drag a **Table View Controller** object from the **Objects Library** onto the scene editor.

Figure 48: The Table View Controller in the Objects Library

The segue was deleted along with the old detail scene, so the new table view isn't a part of the navigation controller hierarchy yet. Re-create the segue by dragging from the master scene's prototype cell to the new detail scene, and then select **push** to create a push segue. After that, be sure to change the ID of the segue back to *showDetail*.

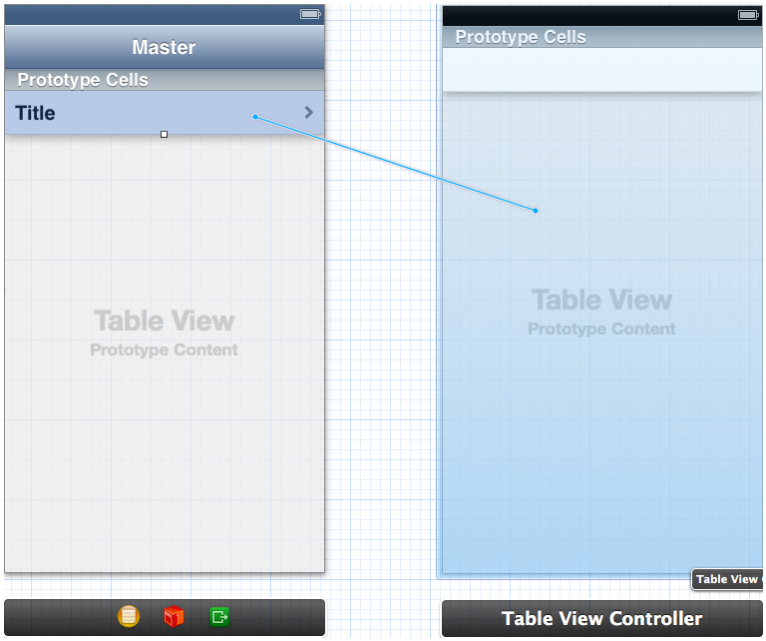

Figure 49: Re-creating the push segue from the master scene to the detail scene

This integrates the **Table View Controller** with the navigation hierarchy, and the Interface Builder reflects this by adding a navigation bar to the top of the detail scene. However, that navigation bar is now blank. Let's fix that by double-clicking in the center of the empty navigation bar and entering *Detail* as the title of the scene, like so:

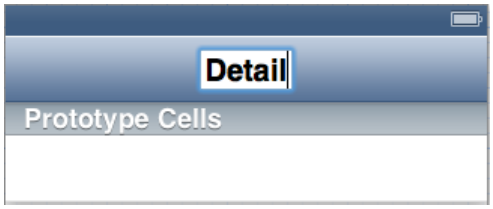

Figure 50: Defining the title of the detail scene

We also need to connect the new scene to our **`DetailViewController`** class. Before changing the class in the interface builder, we need to make **`DetailViewController`** inherit from **`UITableViewController`**. Change the interface declaration in **DetailViewController.h** to the following:

```
@interface DetailViewController : UITableViewController
```

Then, open the storyboard again, select the yellow icon in the **Table View Controller**'s dock, open the **Components inspector**, and change the **Class** to **DetailViewController**.

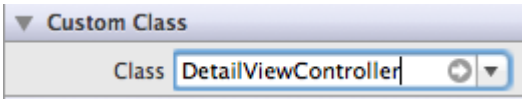

Figure 51: Setting the new Table View Controller's class

Now we're back to where we started, but we have a **`Table View Controller`** instead of a normal **`View Controller`**. Remember that we're going to use a static table to lay out the selected **`Person`** object's information. So, select the detail scene's detail view from the document outline.

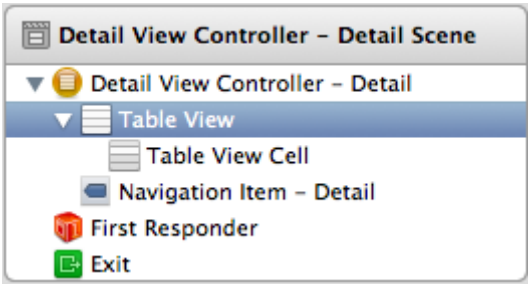

Figure 52: Selecting the detail scene's Table View

Then, change the **Content** field to **Static Cells** in the **Attributes inspector**. You can also change **Separator** to **None** and **Selection** to **No Selection**. This removes the line between the cells and prevents users from selecting them.

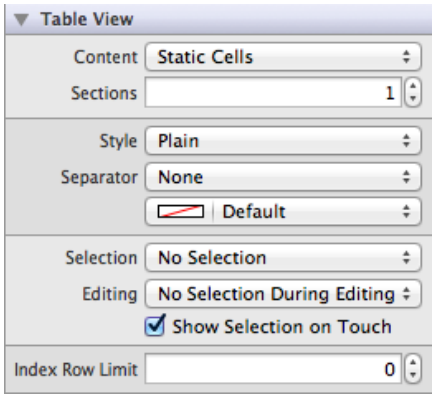

Figure 53: Changing the Table View's content from dynamic prototypes to static cells

You should now see three blank cells in the detail scene. Select all of them by holding Shift and clicking them, and then change their **Style** to **Left Detail** in the **Attributes inspector**. This adds a **Title** and a **Detail** label to each of the cells. Change the title labels to *Name*, *Phone*, and *Organization* so that your detail scene looks like the following:

Figure 54: Configuring the title labels of the static cells

After we add a few properties to the **DetailViewController** class, we'll turn the remaining detail labels into outlets and use them to display the selected **Person**'s information.

Coding the Detail View Controller

That's about all we can do in the Interface Builder for now. Let's add a few properties to **DetailViewController** so that we can access the detail labels that we just added. Change **DetailViewController.h** to the following:

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

@interface DetailViewController : UITableViewController

@property (strong, nonatomic) id detailItem;
@property (weak, nonatomic) IBOutlet UILabel *nameLabel;
@property (weak, nonatomic) IBOutlet UILabel *organizationLabel;
@property (weak, nonatomic) IBOutlet UILabel *phoneNumberLabel;

@end
```

Recall from the previous chapter that the **IBOutlet** modifier is what makes these properties available to the Interface Builder. Next, synthesize these properties in **DetailViewController.m**:

```
#import "DetailViewController.h"
#import "Person.h"

@implementation DetailViewController

@synthesize detailItem = _detailItem;
@synthesize nameLabel = _nameLabel;
@synthesize organizationLabel = _organizationLabel;
@synthesize phoneNumberLabel = _phoneNumberLabel;
```

Then, change the **configureView** method to set the value of the detail labels based on the **Person** object passed in from the master scene:

```
- (void)configureView {
    if (self.detailItem &&
        [self.detailItem isKindOfClass:[Person class]]) {
        NSString *name = [NSString stringWithFormat:@"%@ %@",
                          [self.detailItem firstName],
                          [self.detailItem lastName]];
        self.nameLabel.text = name;
        self.organizationLabel.text = [self.detailItem organization];
        self.phoneNumberLabel.text = [self.detailItem phoneNumber];
    }
```

```
}
```

Also notice that we use the **`isKindOfClass:`** method to ensure that the detail item is in fact a **`Person`** object. This is a best practice step when using dynamically typed variables like **`detailItem`**.

Outlet Connections

Our last step for the detail scene is to connect the **`nameLabel`**, **`organizationLabel`**, and **`phoneNumberLabel`** fields to their corresponding **`UILabel`** components in the storyboard. This can be accomplished by selecting the yellow icon in the detail scene's dock and dragging from the circles in the **Connections inspector** to the label components in the scene editor. Be sure to drag each outlet to the corresponding labels.

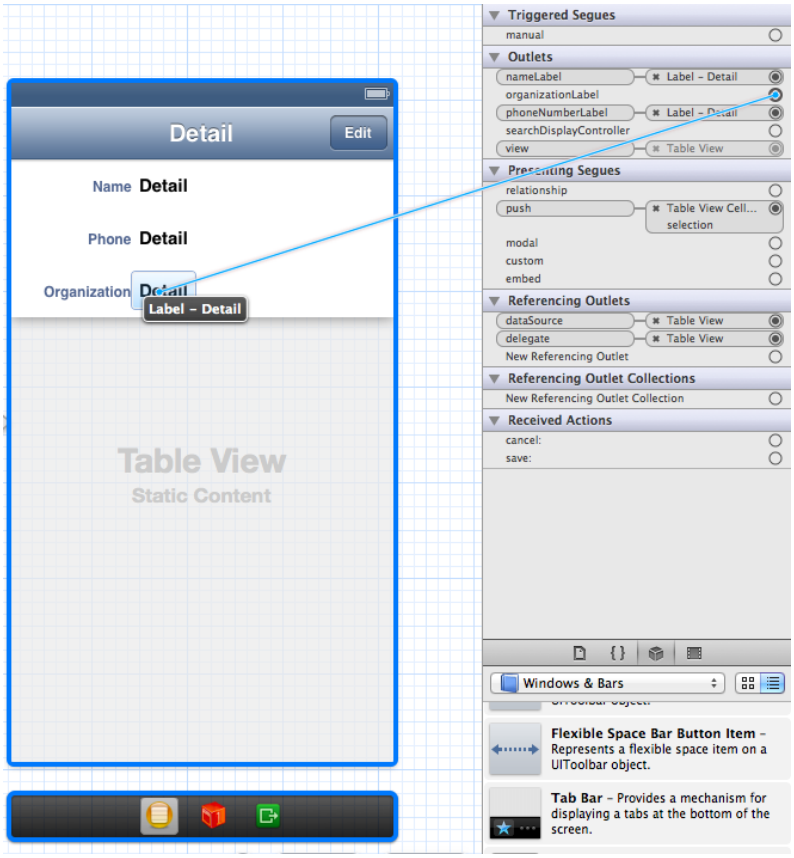

Figure 55: Connecting label components to the Detail View Controller

When you compile the app, you should be able to select items from the master list and view their details in the detail scene. Note that we can only *display* details; we can't edit them yet.

Figure 56: The completed detail scene

To summarize the changes to our detail scene: we replaced the default controller with a **`Table View Controller`** component, changed **`DetailViewController`** to inherit from **`UITableViewController`**, re-created the segue from the master scene to the detail scene, and declared several properties that served as outlets from the **`DetailViewController`** to **`UILabel`** instances. The goal of all of this was to display the properties of the **`Person`** instance that was selected in the master scene.

The Edit View Controller

Our final job for this chapter will be to add another scene that lets us edit the selected item. Instead of a push segue, we're going to implement this new scene using a **modal segue**. A modal segue presents the destination scene "on top of" the existing scene, much like a pop-up window in a desktop computer. This does *not* affect the navigation hierarchy, so instead of a parent **`UINavigationController`** taking responsibility for navigating between the scenes, the modally-presented scene dismisses itself when necessary.

For our example, we'll add a modal segue between our existing detail scene and a new edit scene, then we'll use an **unwind segue** to get back to the original scene. This gives us a new tool for controlling the flow of our application, and it presents the opportunity to get a little bit more comfortable with navigation bars, too.

Creating the Edit Scene

Before we can create a modal segue, we need an edit scene to work with. This scene will work almost exactly like the detail scene, except it will have **`UITextField`** components instead of **`UILabel`**s so that the user can edit each property. First, create a new class called **`EditViewController`** and use **`UITableViewController`** for the superclass:

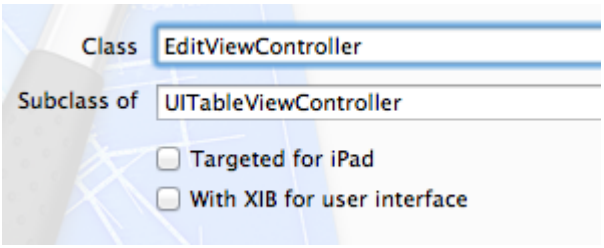

Figure 57: Creating the class for the Edit View Controller

Next, open the Interface Builder and drag another *Table View Controller* from the *Object Library* into the scene editor. Position it above the detail scene, like so:

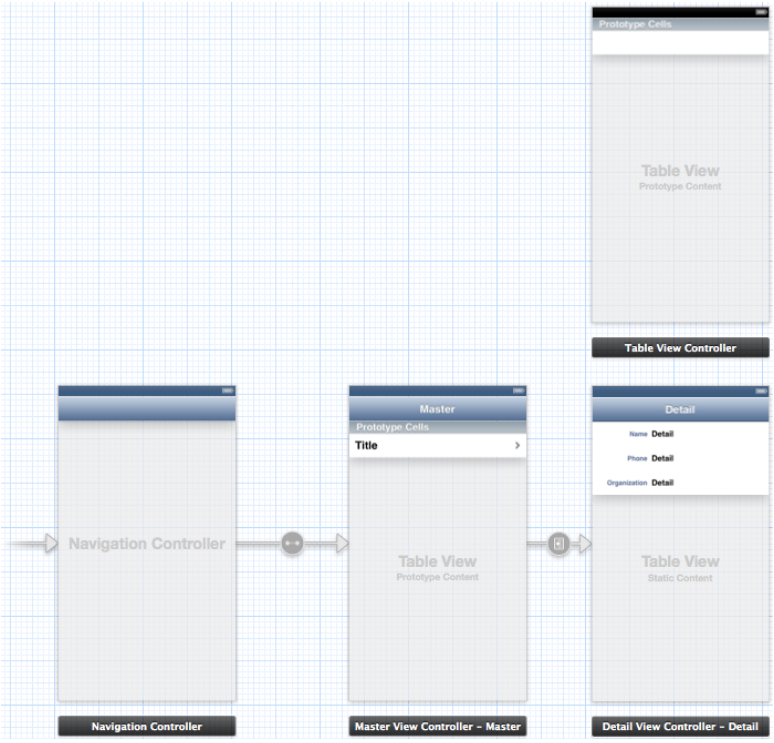

Figure 58: Adding a Table View Controller to the storyboard

This new controller needs to be connected to the **`EditViewController`** class that we just created, so select it in the Interface Editor, open the **Identity inspector** and change the **Class** field to **EditViewController**.

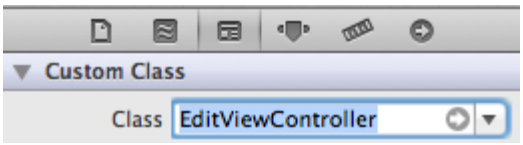

Figure 59: Defining the class of the new table view controller

Navigating to the Edit Scene

Our edit scene will use a navigation bar to present Cancel and Save buttons. We could embed it in the root `UINavigationController`, but remember that we want to present it modally—not by pushing it onto the existing view controller stack. To give it its *own* navigation bar, all we need to do is embed it in its own navigation controller. Select the **Edit View Controller** in the Interface Builder and select **Editor** > **Embed In** > **Navigation Controller** from the Xcode menu.

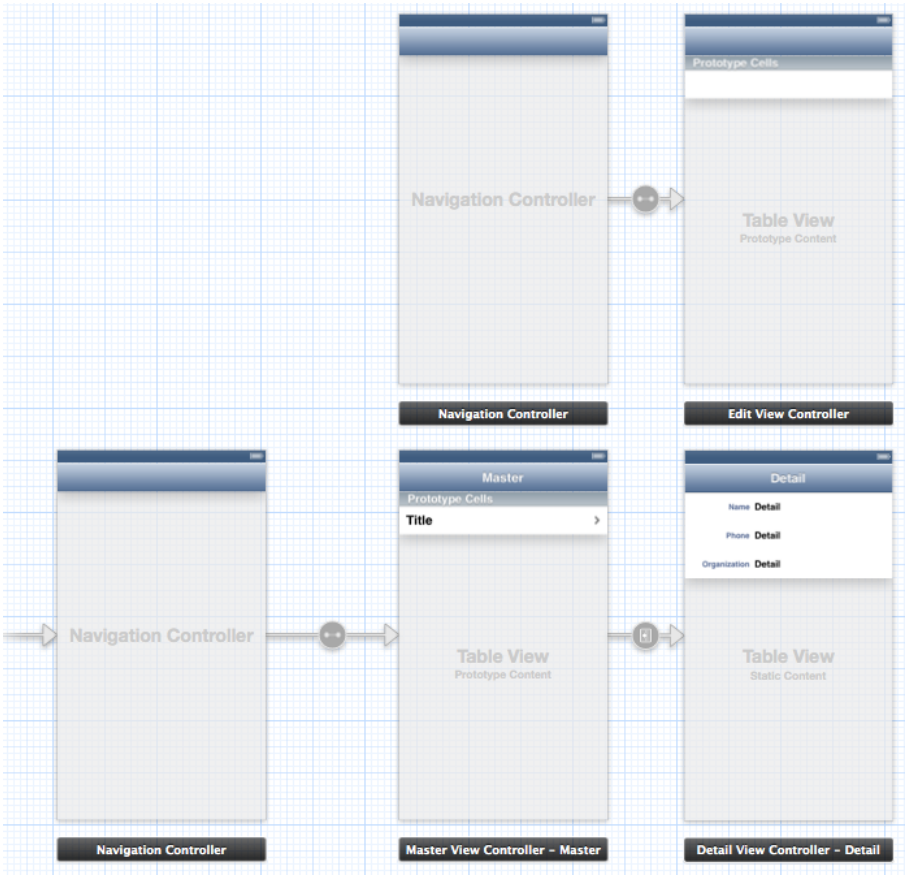

Figure 60: Embedding the edit scene in a new navigation controller

Whereas push segues let the containing navigation controller add navigation buttons for you, we need to add our own buttons for the modal segue. The UIKit Framework uses a special category of controls for use in navigation bars. We're looking for a **bar button item**, which you can find in the **Windows & Bars** section of the **Object Library**.

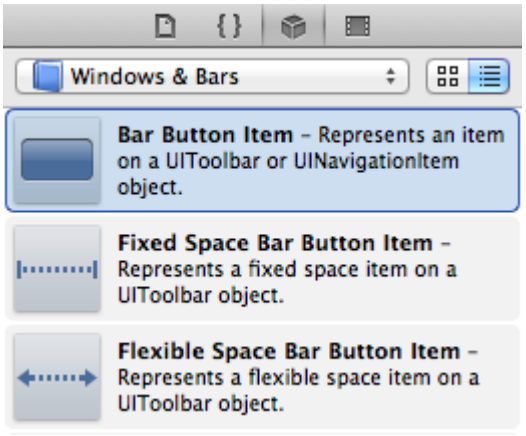

Figure 61: The Bar Button Item in the Object Library

Drag a **Bar Button Item** from the **Object Library** onto the right side of the detail scene's navigation bar. It should snap into place and have a default value of *Item*, as shown in the following screenshot:

Figure 62: Adding an edit button to the detail scene's navigation bar

This button will launch the edit scene, so we should probably change the text to **Edit**. You could do this by manually changing the text in the scene editor, but the preferred way is to select one of the predefined button types from the **Attributes inspector**. Select the **Bar Button Item** and change its **Identifier** field from **Custom** to **Edit**.

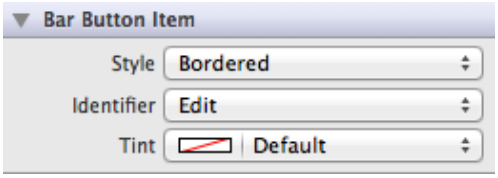

Figure 63: Changing the bar button to an edit button

These predefined types let you access the default system icons which provide a consistent user experience across applications. This isn't a huge deal for the **Add**, **Edit**, **Done**, and other text-based buttons, but can make quite a difference for the iconic types like **Compose**:

Figure 64: The Compose bar button item type

Next, we need to make our new edit button transition to the edit scene. This uses the same process as the push segue from the master table cell to the detail scene. Control-drag from the edit button to the *new* navigation controller, and select **Modal** for the **Action Segue**. You should see a new segue connection with a modal icon on it:

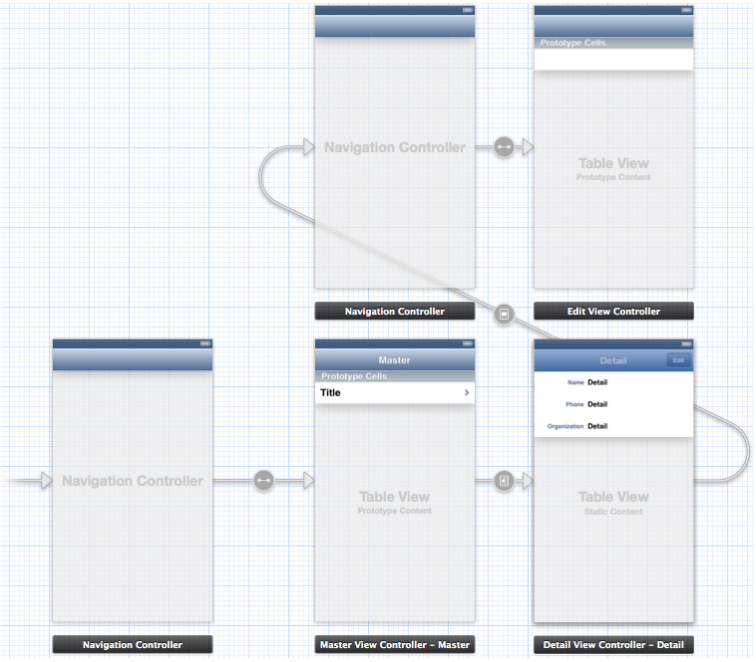

Figure 65: Creating the modal segue

As with all segues, our new modal segue needs a unique identifier. Select the modal segue's icon and enter *editDetail* in the **Identifier** field of the **Attributes inspector**.

You should now be able to compile the project (with a few warnings) and launch an empty edit scene by tapping the **Edit** button in the detail scene. Our next task will be to add some UI components to the edit scene, along with a Cancel and Save button.

Designing the Edit Scene

Next, we're going to design the edit scene. It will look a lot like the detail scene, except it will have text fields instead of labels. Our first task is to add a title to the navigation bar. Double-click the center of the edit scene's navigation bar and type *Edit*. The scene should look like the following afterwards:

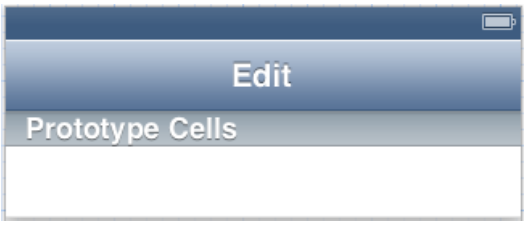

Figure 66: Adding a title to the edit scene

Next, we need to change the **Table View** from a dynamic table to a static one. Select the edit scene's **Table View** object from the **Document Outline**, as shown in the following figure:

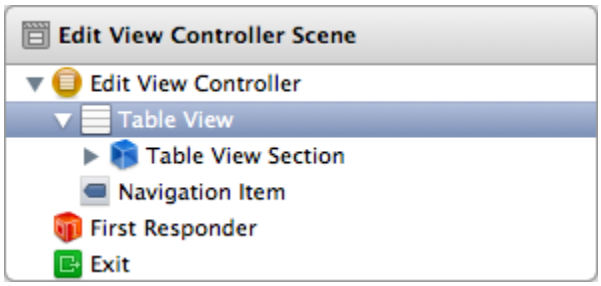

Figure 67: Selecting the Table View object

Then, change the **Content** field of the **Attributes inspector** to **Static Cells**. Delete all but one of the static cells that appear in the scene editor. It's also a good idea to change the **Selection** field to **No Selection** since we're only using the table for layout purposes.

Now, we can't use any of the default **Style** values for the cells since none of them use text fields. Instead, we'll create the cell from scratch. First, drag a **Label** and a **Text Field** object onto the remaining cell and use the guidelines to make sure they are centered vertically. You should also resize both the label and the text field so that they look something like the following:

Figure 68: Adding a label and text field to the edit scene

For the detail scene, we specified **Left Detail** for the cell **Style**. This automatically defined the style, font, and alignment of the components, but since we're creating a custom cell, we need to do this ourselves. All of these settings can be defined in the **Attributes inspector** for the **`UILabel`** and **`UITextField`** objects. For the label, change the text to *First Name*, and then set the color to the same as the title labels in the detail scene. One way to do this is to open the **Colors** panel for the edit scene's label, selecting the magnifying glass (which really acts more like a dropper), and selecting the color from the detail scene's title label. The selected color should be the one in the following figure:

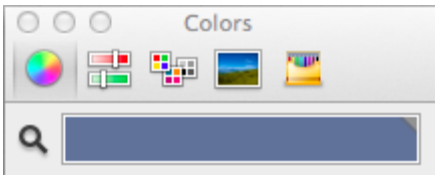

Figure 69: The "dropper" tool in the Colors panel

Finally, change the font to **System Bold** with a size of **12** and change the alignment to **Right**. The final settings are shown in the following screenshot:

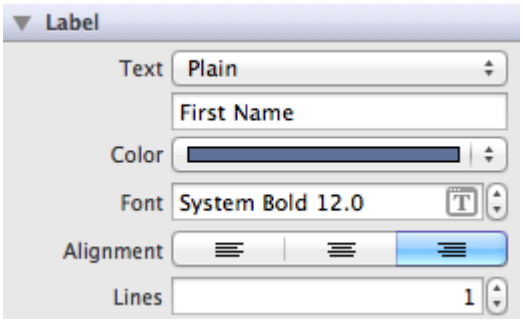

Figure 70: Final attributes for the label

All you need to do for the text field is change the **Capitalization** to **Words**. To create the cells for the other fields, copy and paste the existing cell three times, and change their labels to *Last Name*, *Phone*, and *Organization*. This will give you the following table:

Figure 71: The edit scene table cells with appropriate labels

You should also change the **Keyboard** field for the **Phone** text field to **Number Pad** to display a number pad instead of a QWERTY keyboard. That covers the edit scene's table, but if you try to compile the project right now, you'll notice that all of these cells disappear. This is because the **EditViewController.m** provided by the class template defines several data source methods that treat the table as a prototype cell. We'll delete these in the next section.

But before we do that, let's add two buttons to the navigation bar so that users can choose whether they want to cancel or save their edits. Drag two bar button items from the **Object Library** onto either side of the navigation bar. Change the left button's **Identifier** field to *Cancel* and the right one to *Save.*

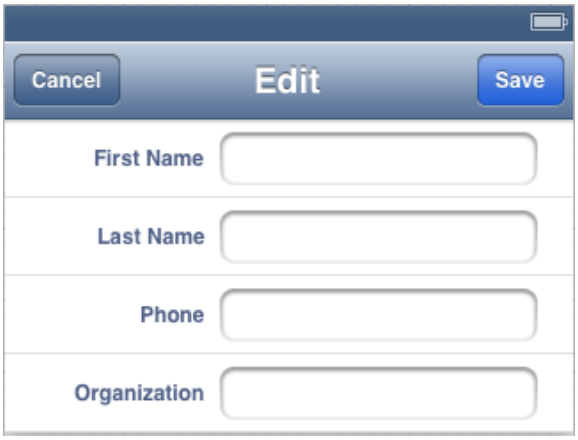

Figure 72: The completed layout for the edit scene

Notice how the **Save** button is bright blue as per the iOS UX conventions. Again, these default **Identifiers** help ensure a consistent user interface across applications.

Coding the Edit View Controller

In this section, we'll code the functionality behind the UI components we just added to the storyboard. The two main tasks are to prepare outlets for the text fields so that we can access them from the **`EditViewController`** class, and implement a text field delegate so users can dismiss the text field. This should all be a review from the previous chapter. First, let's add a few properties to the header file:

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

@interface EditViewController : UITableViewController

@property (strong, nonatomic) id detailItem;

@property (weak, nonatomic) IBOutlet UITextField *firstNameField;
@property (weak, nonatomic) IBOutlet UITextField *lastNameField;
@property (weak, nonatomic) IBOutlet UITextField *phoneNumberField;
@property (weak, nonatomic) IBOutlet UITextField *organizationField;

@end
```

The implementation looks a lot like **DetailViewController.m**. All it does is make sure that the text fields are updated when the **`detailItem`** property is changed:

```
// EditViewController.m
#import "EditViewController.h"
#import "Person.h"
```

```
@implementation EditViewController

@synthesize detailItem = _detailItem;
@synthesize firstNameField = _firstNameField;
@synthesize lastNameField = _lastNameField;
@synthesize phoneNumberField = _phoneNumberField;
@synthesize organizationField = _organizationField;

- (void)setDetailItem:(id)detailItem {
    if (_detailItem != detailItem) {
        _detailItem = detailItem;
        [self configureView];
    }
}

- (void)configureView {
    if (self.detailItem && [self.detailItem isKindOfClass:[Person class]]) {
        self.firstNameField.text = [self.detailItem firstName];
        self.lastNameField.text = [self.detailItem lastName];
        self.phoneNumberField.text = [self.detailItem phoneNumber];
        self.organizationField.text = [self.detailItem organization];
    }
}

- (void)viewDidLoad {
    [super viewDidLoad];
    [self configureView];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}

@end
```

Next, we need to prepare the text field delegate. In **EditViewController.h**, tell the class to adopt the **`UITextFieldDelegate`** protocol with the following line:

```
@interface EditViewController : UITableViewController <UITextFieldDelegate>
```

As in the previous chapter, we can dismiss the keyboard by implementing the **`textFieldShouldReturn:`** method. Add the following to **EditViewController.m**:

```
- (BOOL)textFieldShouldReturn:(UITextField *)textField {
    if ((textField == self.firstNameField) ||
        (textField == self.lastNameField) ||
        (textField == self.phoneNumberField) ||
        (textField == self.organizationField)) {
        [textField resignFirstResponder];
```

```
    }
    return YES;
}
```

Recall that the **`prepareForSegue:sender:`** method is called on the source scene right before iOS switches to the destination scene. Just as we did in the master scene, we'll use this to send the selected item to the edit scene. In **DetailViewController.m**, add the following method:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    if ([[segue identifier] isEqualToString:@"editDetail"]) {
        NSArray *navigationControllers = [[segue destinationViewController]
viewControllers];
        EditViewController *editViewController = [navigationControllers
objectAtIndex:0];
        [editViewController setDetailItem:self.detailItem];
    }
}
```

Remember that the edit scene is embedded in a navigation controller, so the modal segue points to the *navigation controller*, not the edit scene itself. This intervening navigation controller adds an extra step that we didn't need to worry about in the master scene's **`prepareForSegue:sender:`** method. To get the edit scene, we need to query the navigation controller's **`viewControllers`** property, which is an array containing its navigation stack. Since the edit scene is the only child view controller, we can access it via the **`objectAtIndex:0`** call. Once we have the **`EditViewController`** instance, we simply forward the selected item from the detail scene to the edit scene.

Outlet and Delegate Connections

Back in the storyboard, let's connect the outlets and delegates that we just exposed. For the outlets, select the yellow icon in the edit scene's dock, open the **Connections inspector**, and drag from the **firstNameField**, **lastNameField**, **organizationField**, and **phoneNumberField** circles to the corresponding text fields in the scene.

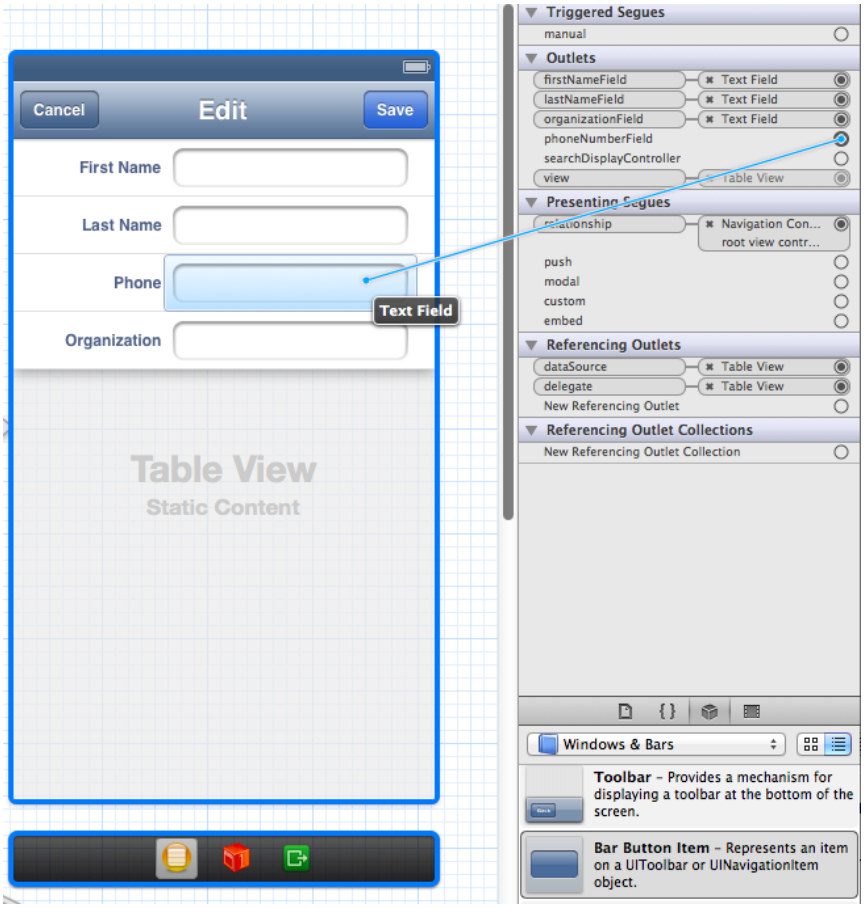

Figure 73: Creating the outlet connections

To set the **`EditViewController`** as the delegate for the text fields, select each text field, open the **Connections inspector**, and drag from the **delegate** circle to the yellow icon in the dock, as shown in the following screenshot. Do this for each text field.

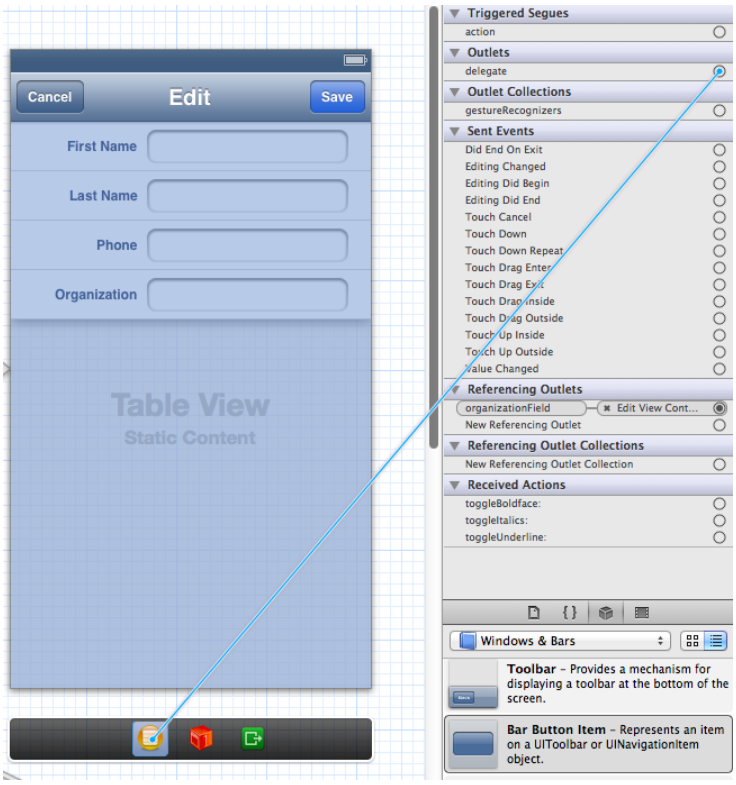

Figure 74: Creating the delegate connections

When you compile the project, you should be able to launch the edit scene and see the text fields populated with the selected **`Person`** object's properties. Hopefully by now, you're relatively comfortable making these kinds of outlet and delegate connections on your own.

You can edit the values, but since we haven't implemented the **Cancel** or **Save** buttons yet, you won't be able to alter the underlying **`Person`** object or even navigate away from the edit scene.

Unwind Segues

Remember that **Master** button that automatically appeared in the detail scene's navigation bar? The navigation controller for the master/detail scenes set up this "back" button for us, but since we're using a modal segue, we need to manually dismiss the modally presented edit scene. We'll use what's called an **unwind segue** to return to the detail scene.

The main difference between an unwind segue and other segues is that the former uses an *existing* scene as the destination, whereas modal and push segues create a *new* instance of their destination scene. This is important to keep in mind if you're doing a lot of transitioning back and forth.

The process of unwinding a scene is also a little bit different than initiating a push segue. It uses the target-action design pattern, which we discussed in the previous chapter. In addition to calling the **`prepareForSegue:sender:`** method on the source scene, an unwind segue calls an arbitrary method on the *destination* scene (**`DetailViewController`**). Let's go ahead and declare a cancel and a save action in **DetailViewController.h**:

```
- (IBAction)save:(UIStoryboardSegue *)sender;
- (IBAction)cancel:(UIStoryboardSegue *)sender;
```

In a moment, we're going to attach these methods to the **Cancel** and **Save** buttons in the edit scene. But first, we need to implement them. Add the following methods to **DetailViewController.m**:

```
- (IBAction)save:(UIStoryboardSegue *)segue {
    if ([[segue identifier] isEqualToString:@"saveInput"]) {
        EditViewController *editController = [segue sourceViewController];
        [self.detailItem setFirstName:editController.firstNameField.text];
        [self.detailItem setLastName:editController.lastNameField.text];
        [self.detailItem setPhoneNumber:editController.phoneNumberField.text];
        [self.detailItem setOrganization:editController.organizationField.text];
        [self configureView];
    }
}

- (IBAction)cancel:(UIStoryboardSegue *)segue {
    if ([[segue identifier] isEqualToString:@"cancelInput"]) {
        // Custom cancel handling can go here.
    }
}
```

These are pretty straightforward. The **`save:`** method updates the **`detailItem`**'s properties based on the text field values from the edit scene, and then updates its labels by calling **`configureView`**. The **`cancel:`** method simply ignores anything that happened in the edit scene.

Now, we can create an unwind segue to dismiss the edit scene and call the appropriate method. Configuring unwind segues is similar to creating push segues: you control-drag from the UI component that initiates the segue to the green **Exit** icon in the dock. This icon is dedicated solely to creating unwind segues.

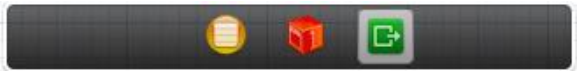

Figure 75: The exit icon in the dock (far right)

So, control-drag from the **Save** button in the edit scene to the **Exit** icon in the dock, as shown in the following figure:

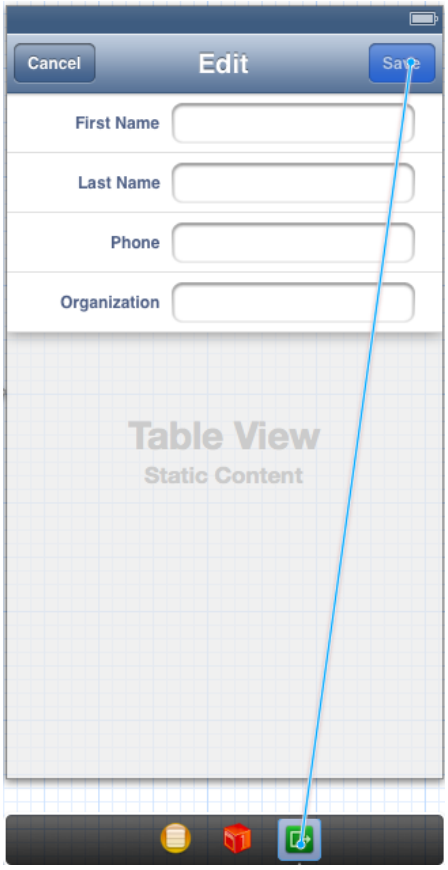

Figure 76: Creating the unwind segue for the Save button

A menu will pop up asking you to associate an action with the segue:

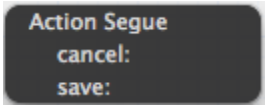

Figure 77: Selecting the action for the unwind segue

Of course, you'll want to choose **`save:`**. That's all you need to do to create the unwind segue. After repeating the process for the **Cancel** button, you should see both unwind segues in the document outline:

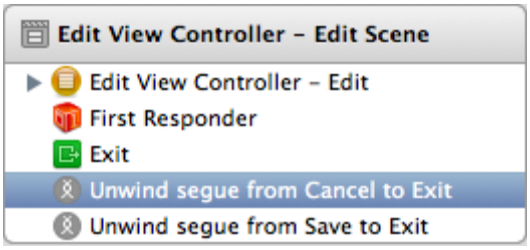

Figure 78: The unwind segues in the document outline

Unlike push and modal segues, unwind segues have no visual representation in the interface builder, so the document outline is the only way you can select them. Our last step will be to add unique identifiers to both of these segues via the **Attributes inspector**. Use **`cancelInput`** for the **Cancel** button and **`saveInput`** for the **Save** button (note that these are the identifiers we checked against in the **`cancel:`** and **`save:`** methods, respectively). Again, since our example app is so simple, adding segue identifiers is more of a best practice step than a necessity.

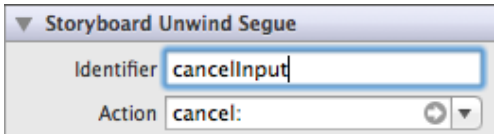

Figure 79: Defining the unwind segue identifiers

You can think of an unwind segue as a combination of a transition and a button. The segue takes care of dismissing the scene (i.e. transitioning to the parent scene), but since it's initiated by a button press, you can also attach a method to it using the target-action pattern.

Our edit scene is now complete, and you should be able to compile the project, enter values into the edit scene's text fields, and choose to cancel or save your changes. Since the **`save:`** method calls **`configureView`** after saving the new values, the detail scene will update to reflect the edits. However, we never told the master scene to update itself, so your changes will *not* be reflected in the master list.

Updating the Master List

The final task for this chapter is to update the master scene's table to reflect any changes in the underlying data. There are a number of ways to do this, but the easiest (though not necessarily the most efficient) is to reload the table each time the master scene is displayed.

`UIViewController` defines a method called **`viewWillAppear:`** and calls it right before the associated scene is displayed. This is different than **`viewDidLoad:`**, which gets the *first* time the view is displayed. Since the parent navigation controller displays the *same* instance of the master scene each time the user navigates to it, we need to use the **`viewWillAppear:`** method instead of **`viewDidAppear:`**. In **MasterViewController.m**, add the following method:

```
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    UITableView *view = (UITableView *)self.view;
    [view reloadData];
}
```

First, we pass the method along to **`super`**, and then we fetch the controller's root **`UIView`** instance through the **`view`** property. We can assume this is a **`UITableView`** because **`MasterViewController`** inherits from **`UITableViewController`**, but we still need to cast it to prevent the compiler from complaining. The **`UITableView`**'s **`reloadData`** method regenerates the table cells based on the underlying data set (the **`_objects`** array), and the master list should now reflect any changes you saved from the edit scene.

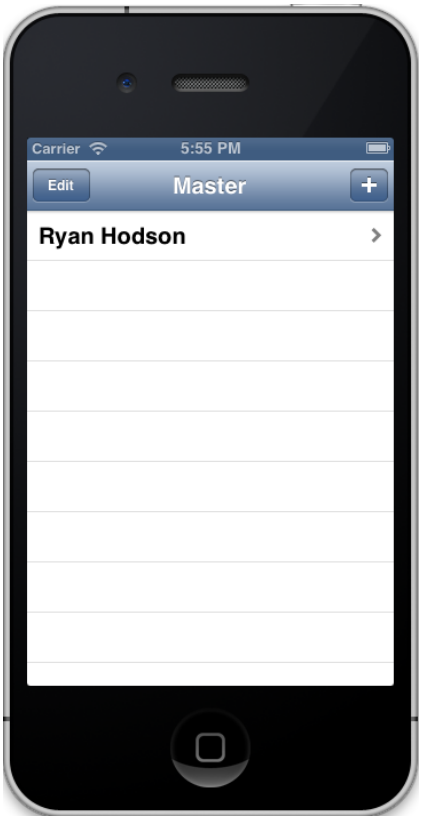

Figure 80: The completed master scene

Summary

In this chapter, we learned how to manage multiple scenes within a single app. We experimented with **`UITableViewControllers`**, **`UINavigationControllers`**, and all sorts of segues. One of the most important concepts to take away from this chapter is how we transferred data between each scene: via the **`prepareForSegue:sender:`** method and the **`save:`** and **`cancel:`** methods for the unwind segue. Most applications are really just user-friendly editors for complex data structures, so understanding how that data is passed around goes a long way toward efficiently organizing iOS projects.

The previous two chapters covered everything you need to know to create simple user interfaces. For the rest of this book, we'll explore other iOS frameworks for accessing media assets, localizing resources, and playing UI sound effects.

Chapter 3 Asset Management

Now that we have a basic understanding of iOS scene management, the next big topic to tackle is how to manage the multimedia assets in an application. iOS apps store their assets using the same hierarchical file system as any other modern operating system. Text, image, audio, and video files are organized into folders and accessed using familiar file paths like **Documents/SomePicture.png**.

In this chapter, we'll learn about the standard file structure for an app; how to add resources to a project; and how to locate, load, and save files. We'll also talk about the required assets for all iOS apps.

Throughout the chapter, we'll talk about files and folders, but keep in mind that the file system should be entirely hidden from iOS users. Instead of showing users the files and folders behind an application, iOS encourages developers to present the file system as user-oriented documents. For example, in a sketching app, drawings should be listed with semantic display names and organized into sketchbooks or a similar abstract organizational structure. You should never show the user file paths like **sketchbook-1/your-drawing.svg**.

Conceptual Overview

The Application Sandbox

The iOS file system was designed with security in mind. Instead of allowing an app to access a device's entire file system, iOS gives each application its own separate file system (a sandbox). This means your application doesn't have access to files generated by other apps. When you need to access information that is not owned by your app (e.g., the user's contact list), you request it from a mediator (e.g., the Address Book Framework) instead of accessing the files directly.

A sandbox is like a mini file system dedicated solely to your app's operation. All apps use a canonical file structure consisting of four top-level directories, each of which store a specific type of file:

- **AppName.app**—The application bundle, which contains your app's executable and all of its required media assets. You can read from this folder, but you should never write to it. The next section discusses bundles in more detail.
- **Documents/**—A folder for user-generated content and other critical data files that cannot be re-created by your app. The contents of this directory are available through iCloud.
- **Library/**—A folder for application files that are not used by the user, but still need to persist between launches.

- **tmp/**—A folder for temporary files used while your application is running. Files in this folder do not necessarily persist between application launches. iOS will automatically delete temporary files when necessary while your application isn't running, but you should manually delete temporary files as soon as you're done with them as a best practice.

When the user installs an app, a new sandbox containing all of these folders is created. After that, your application can dynamically create arbitrary subdirectories in any of these top-level folders. There are also a few pre-defined subdirectories, as described in the following list:

- **Library/Application Support/**—A folder for support files that can be re-created if necessary. This includes downloaded and generated content. You should use the **`com.apple.MobileBackup`** extended attribute to prevent this folder from being backed up.
- **Library/Cache/**—A folder for cache files. These files can be deleted without notice, so your app should be able to re-create them gracefully. This folder is also an appropriate place to store downloaded content.

It's important to put files in the appropriate folder to make sure they are backed up properly without consuming an unnecessary amount of space on the user's device. iTunes automatically backs up files in the **Documents/** and **Library/** folders (with the exception of **Library/Cache/**). Neither the application bundle nor the **tmp/** folder should ever need to be backed up.

Bundles

An iOS application isn't just an executable—it also contains media, data files, and possibly localized text for different regions. To simplify deployment, Xcode wraps the executable and all of its required files into a special kind of folder called an **application bundle**. Despite being a folder, an application bundle uses the **.app** extension. You can think of an application bundle as a ZIP file that runs an app when you open it.

Since your application bundle contains all of your media assets, you'll need to interact with it while your program is running. The **`NSBundle`** class makes it easy to search your application bundle for specific files, which can then be loaded by other classes. For example, you can locate a particular image file using **`NSBundle`**, and then add it to a view using the **`UIImage`** class. We'll do this in another section, "The Application Bundle."

Creating the Example Application

This chapter uses a simple application to explore some of the fundamental methods for accessing files in iOS. First, open Xcode, create a new project, and select **Single View Application** for the template.

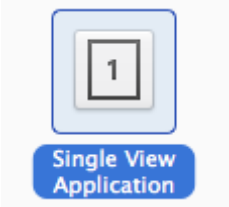

Figure 81: Creating a new Single View Application

Use *AssetManagement* for the **Product Name**, *edu.self* for the **Company Identifier**, and make sure **Use Storyboards** and **Use Automatic Reference Counting** are selected.

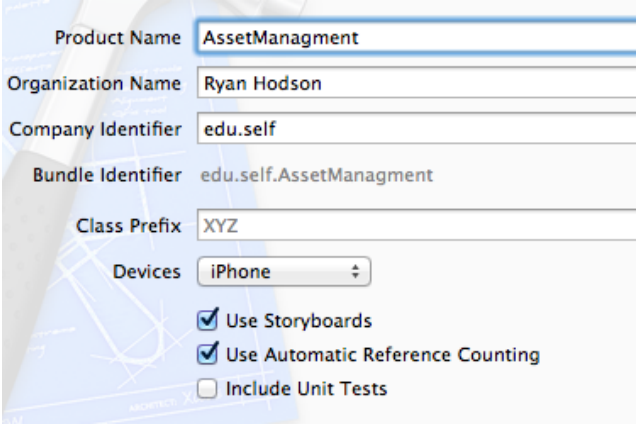

Figure 82: Configuring the new project

You can save the project wherever you like.

The File System

Before we go into an app's multimedia assets, we're going to look at the basic tools for accessing the file system. The upcoming sections discuss how to generate file paths, create plain text files, and load them back into the application.

Locating Standard Directories

One of the most common file system tasks is generating the path to a particular resource. But, before you can gain access to any given file, you need to find the path to the application sandbox or one of the top-level folders discussed previously. The easiest way to do this is via the global **`NSHomeDirectory()`** function, which returns the absolute path to the application sandbox. For example, try changing **ViewController.m**'s **`viewDidLoad`** method to the following:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    NSString *sandboxPath = NSHomeDirectory();
```

```
    NSLog(@"The app sandbox resides at: %@", sandboxPath);
}
```

When the app loads in the iOS Simulator, you should see something like the following in Xcode's output panel.

```
/Users/ryan/Library/Application Support/iPhone Simulator/6.0/Applications/9E38D1C4-8B11-4599-88BE-CD9E36C21A41
```

This path represents the root of your application. If you navigate to this directory in a terminal, you'll find the following four directories:

```
AssetManagement.app
Documents/
Library/
tmp/
```

Not surprisingly, this is the canonical file structure discussed previously. Of course, **`NSHomeDirectory()`** will return a different path when your application is running on an iOS device. The idea behind using **`NSHomeDirectory()`** instead of manually generating the path to your application is to make sure you always have the correct root path, regardless of where your application resides.

The related **`NSTemporaryDirectory()`** function returns the path to the **tmp/** directory. For the other standard application folders, you'll need to use the **`NSFileManager`** class.

```
- (void)viewDidLoad {
    [super viewDidLoad];
    NSFileManager *sharedFM = [NSFileManager defaultManager];
    NSArray *paths = [sharedFM URLsForDirectory:NSLibraryDirectory
                                      inDomains:NSUserDomainMask];
    if ([paths count] > 0) {
        NSLog(@"The Library subfolder: %@", paths[0]);
    }
}
```

As you can see, **`NSFileManager`** is implemented as a singleton, and the shared instance should be accessed via the **`defaultManager`** class method. The **`NSSearchPathDirectory`** enum defines several constants that represent the standard locations used by both OS X and iOS applications. Some of these locations (e.g., **`NSDesktopDirectory`**) are not applicable in iOS apps, however, the **`URLsForDirectory:inDomains:`** method will still return the appropriate subfolder in the application sandbox. The constants for the directories we've discussed are listed as follows.

```
NSDocumentDirectory               /*  Documents/                    */
NSLibraryDirectory                /*  Library/                      */
NSCachesDirectory                 /*  Library/Caches                */
NSApplicationSupportDirectory     /*  Library/Application Support/  */
```

The **URLsForDirectory:inDomains:** method returns an **NSArray** containing NSURL objects, which is an object-oriented representation of a file path.

Generating File Paths

Once you have the location of one of the standard directories, you can manually assemble the path to a specific file using the **NSURL** instance methods (note that **NSString** also provides related utilities, but **NSURL** is the preferred way to represent file paths).

For example, the **URLByAppendingPathComponent:** method provides a straightforward way to generate the path to a specific file. The following snippet creates a path to a file called **someData.txt** in the application's **Library/** directory:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    NSFileManager *sharedFM = [NSFileManager defaultManager];
    NSArray *paths = [sharedFM URLsForDirectory:NSLibraryDirectory
                                      inDomains:NSUserDomainMask];
    if ([paths count] > 0) {
        NSURL *libraryPath = paths[0];
        NSURL *appDataPath = [libraryPath
                              URLByAppendingPathComponent:@"someData.txt"];
        NSLog(@"%@", appDataPath);
    }
}
```

A few of the other useful **NSURL** instance methods are described in the following list. Together, these provide the basic functionality for navigating a file hierarchy and manually determining file names and types.

- **URLByDeletingLastPathComponent**—Returns a new **NSURL** representing the parent folder of the receiving path.
- **lastPathComponent**—Returns the final component in the path as a string. This could be either a folder name or a file name, depending on the path.
- **pathExtension**—Returns the file extension of the path as a string. If the path doesn't contain a period, it returns an empty string, otherwise it returns the last group of characters that follow a period.
- **pathComponents**—Decomposes the path into its component parts and returns them as an **NSArray**.

Saving and Loading Files

It's important to understand that **NSURL** only describes the *location* of a resource—it does not represent the actual file or folder itself. To get at the file data, you need some way to interpret it. The NSData class provides a low-level API for reading in raw bytes, but most of the time you'll want to use a higher-level interface for interpreting the contents of a file.

The iOS frameworks include many classes for saving and loading different types of files. For example, NSString can read and write text files, UIImage can display images inside of a view, and AVAudioPlayer can play music loaded from a file. We'll look at **UIImage** once we get to the application bundle, but for now, let's stick with basic text file manipulation.

To save a file with **NSString**, use the **writeToURL:automatically:encoding:error:** method. The first argument is an **NSURL** representing the file path, the second determines whether or not to save it to an auxiliary file first, the third is one of the constants defined by the **NSStringEncoding** enum, and the **error** argument is a reference to an **NSError** instance that will record error details should the method fail. The following snippet demonstrates **writeToURL:automatically:encoding:error:** by creating a plain text file called **someData.txt** in the **Library/** folder.

```
- (void)viewDidLoad {
    [super viewDidLoad];
    NSFileManager *sharedFM = [NSFileManager defaultManager];
    NSArray *paths = [sharedFM URLsForDirectory:NSLibraryDirectory
                                      inDomains:NSUserDomainMask];
    if ([paths count] > 0) {
        NSURL *libraryPath = paths[0];
        NSURL *appDataPath = [libraryPath
                              URLByAppendingPathComponent:@"someData.txt"];

        NSString *someAppData = @"Hello, World! This is a file I created
dynamically";
        NSError *error = nil;
        BOOL success = [someAppData writeToURL:appDataPath
                                    atomically:YES
                                      encoding:NSUnicodeStringEncoding
                                         error:&error];
        if (success) {
            NSLog(@"Wrote some data to %@", appDataPath);
        } else {
            NSLog(@"Could not write data to file. Error: %@", error);
        }
    }
}
```

When saving or loading text files, it's imperative to specify the file encoding, otherwise your text data could be unexpectedly altered when writing or reading a file. A few of the common encoding constants are included here:

- **NSASCIIStringEncoding**–7-bit ASCII encoding with 8-bit chars (ASCII values 0-127).

- **NSISOLatin1StringEncoding**–8-bit ISO Latin 1 encoding.
- **NSUnicodeStringEncoding**–Unicode encoding.

When in doubt, you'll probably want to use **NSUnicodeStringEncoding** to make sure multi-byte characters are interpreted properly.

To load a text file, you can use the related **stringWithContentsOfURL:encoding:error:**. This works much the same as **writeToURL:automatically:encoding:error:**, but it's implemented as a class method instead of an instance method. The following example loads the text file created by the previous snippet back into the app.

```
- (void)viewDidLoad {
    [super viewDidLoad];
    NSFileManager *sharedFM = [NSFileManager defaultManager];
    NSArray *paths = [sharedFM URLsForDirectory:NSLibraryDirectory
                                      inDomains:NSUserDomainMask];
    if ([paths count] > 0) {
        NSURL *libraryPath = paths[0];
        NSURL *appDataPath = [libraryPath
                              URLByAppendingPathComponent:@"someData.txt"];

        NSError *error = nil;
        NSString *loadedText = [NSString
                                stringWithContentsOfURL:appDataPath
                                encoding:NSUnicodeStringEncoding
                                error:&error];
        if (loadedText != nil) {
            NSLog(@"Successfully loaded text: %@", loadedText);
        } else {
            NSLog(@"Could not load data from file. Error: %@", error);
        }
    }
}
```

In the real world, you'll probably save and load data that was dynamically generated instead of hardcoded as a literal string. For instance, you might store template preferences or user information that needs to persist between application launches in a text file. It's entirely possible to manually load and interpret this data from plain text, but keep in mind that there are several built-in tools for working and storing structured data or even whole Objective-C objects. For example, **NSDictionary** defines a method called **dictionaryWithContentsOfURL:** that loads an XML file containing key-value pairs.

Manipulating Directories

The **NSString** methods described previously combine the creation of a file and the writing of content into a single step, but to create directories, we need to return to the **NSFileManager** class. It defines several methods that let you alter the contents of a directory.

Creating Directories

The **`createDirectoryAtURL:withIntermediateDirectories:attributes:error:`** instance method creates a new directory at the specified path. The second argument is a Boolean value that determines whether or not intermediate directories should be created automatically, **`attributes`** lets you define file attributes for the new directory, and the final argument is a reference to an **`NSError`** instance that will contain the error details should the method fail.

For instance, if your application uses custom templates downloaded from a server, you might save them in **Library/Templates/**. To create this folder, you could use something like the following:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    NSFileManager *sharedFM = [NSFileManager defaultManager];
    NSArray *paths = [sharedFM URLsForDirectory:NSLibraryDirectory
                                      inDomains:NSUserDomainMask];
    if ([paths count] > 0) {
        NSURL *libraryPath = paths[0];
        NSURL *templatesPath = [libraryPath
                                URLByAppendingPathComponent:@"Templates"];

        NSError *error = nil;
        BOOL success = [sharedFM createDirectoryAtURL:templatesPath
                           withIntermediateDirectories:YES
                                            attributes:nil
                                                 error:&error];
        if (success) {
            NSLog(@"Successfully created a directory at %@",
                  templatesPath);
        } else {
            NSLog(@"Could not create the directory. Error: %@", error);
        }
    }
}
```

Leaving the **`attributes`** argument as **`nil`** tells the method to use the default group, owner, and permissions for the current process.

Moving Files/Directories

The **`NSFileManager`** class can also be used to move or rename files and folders via its **`moveItemAtURL:toURL:error:`** instance method. It works as follows:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    NSFileManager *sharedFM = [NSFileManager defaultManager];
    NSArray *paths = [sharedFM URLsForDirectory:NSLibraryDirectory
```

```
                                        inDomains:NSUserDomainMask];
    if ([paths count] > 0) {
        NSURL *libraryPath = paths[0];
        NSURL *sourcePath = [libraryPath
                             URLByAppendingPathComponent:@"someData.txt"];
        NSURL *destinationPath = [libraryPath
                          URLByAppendingPathComponent:@"someOtherData.txt"];

        NSError *error = nil;
        BOOL success = [sharedFM moveItemAtURL:sourcePath
                                         toURL:destinationPath
                                         error:&error];
        if (success) {
            NSLog(@"Successfully moved %@ to %@",
                  sourcePath,
                  destinationPath);
        } else {
            NSLog(@"Could not move the file. Error: %@", error);
        }
    }
}
```

This renames the **someData.txt** file we created earlier to **someOtherData.txt**. If you need to copy files, you can use **`copyItemAtURL:toURL:error:`**, which works the same way, but leaves the source file untouched.

Removing Files/Directories

Finally, **`NSFileManager`**'s **`removeItemAtURL:error:`** method lets you delete files or folders. Simply pass the **`NSURL`** instance containing the path you want to remove, as follows:

```
[sharedFM removeItemAtURL:targetURL error:&error];
```

If the **`targetURL`** is a directory, its contents will be removed recursively.

Listing the Contents of a Directory

It's also possible to list the files and subdirectories in a folder using **`NSFileManager`**'s **`enumeratorAtPath:`** method. This returns an NSDirectoryEnumerator object that you can use to iterate through each file or folder. For example, you can list the contents of the top-level **Documents/** directory as follows:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    NSFileManager *sharedFM = [NSFileManager defaultManager];
    NSArray *paths = [sharedFM URLsForDirectory:NSDocumentDirectory
                                        inDomains:NSUserDomainMask];
```

```
    if ([paths count] > 0) {
        NSString *documentsPath = [paths[0] path];
        NSLog(@"%@", documentsPath);
        NSDirectoryEnumerator *enumerator = [sharedFM
enumeratorAtPath:documentsPath];
        id object;
        while(object = [enumerator nextObject]) {
            NSLog(@"%@", object);
        }
    }
}
```

Note that this enumerator will iterate through *all* subdirectories. You can alter this behavior by calling the **`skipDescendents`** method on the **`NSDirectoryEnumerator`** instance, or using the more sophisticated **`enumeratorAtURL:includingPropertiesForKeys:options:errorHandler:`** method of **`NSFileManager`**.

The Application Bundle

The file access methods discussed previously are typically only used to interact with files that are dynamically created at run time. The standard **Library/**, **Documents/**, and **tmp/** folders are local to the device on which the application is installed, and they are initially empty. For assets that are an essential part of the application itself (as opposed to being *created by* the application), we need another tool—an application bundle.

The application bundle represents your entire project, which is composed of the iOS executable, along with all of the images, sounds, videos, and configuration files required by that executable. The application bundle is what actually installs when a user downloads your app, so—unlike the contents of the sandbox directories—you can assume that all of the supporting resources will be present regardless of the device on which your app resides.

Conceptually, a bundle is just a bunch of files, but interacting with it is a little bit different than using the file system methods from the previous section. Instead of manually generating file paths and saving or loading data with methods like **`writeToURL:automatically:encoding:error:`**, you use the **`NSBundle`** class as a high-level interface to your application's media assets. This delegates some nitty-gritty details behind media assets to the underlying system.

In addition to custom media assets, the application bundle also contains several required resources, like the app icon that appears on the user's home screen and important configuration files. We'll talk about these in the "Required Resources" section.

Adding Assets to the Bundle

Remember that assets in the application bundle are static, so they will always be included at compile-time. To add a media asset to your project, simply drag the file(s) from the **Finder** into

the **Project Navigator** panel in Xcode. We're going to add a file called **syncfusion-logo.jpg**, which you can find in the resource package for this book, but you can use any image you like. In the next section, we'll learn how to access the bundle to display this image in the view.

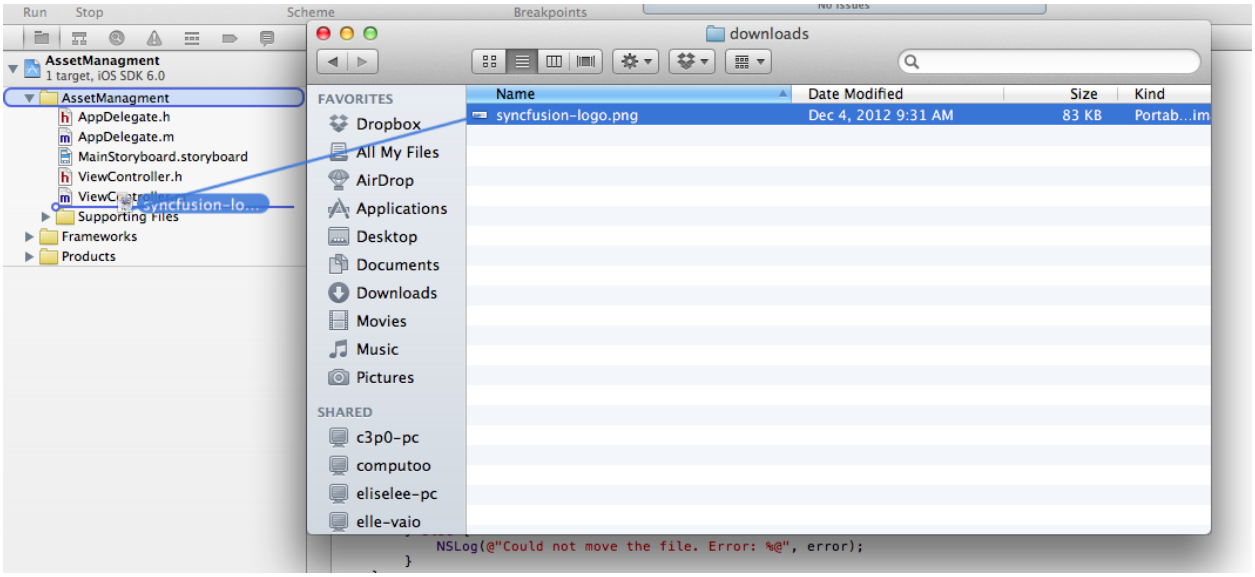

Figure 83: Adding an image to the application bundle

After releasing the mouse, Xcode will present you with a dialog asking for configuration options. The **Copy items into destination group's folder** check box should be selected. This tells Xcode to copy the assets into the project folder, which is typically desirable behavior. The **Create groups for any added folders** option uses the Xcode grouping mechanism. **Add to targets** is the most important configuration option. It defines which build targets the asset will be a compiled with. If **AssetManagement** wasn't selected, it would be like we never added it to the project.

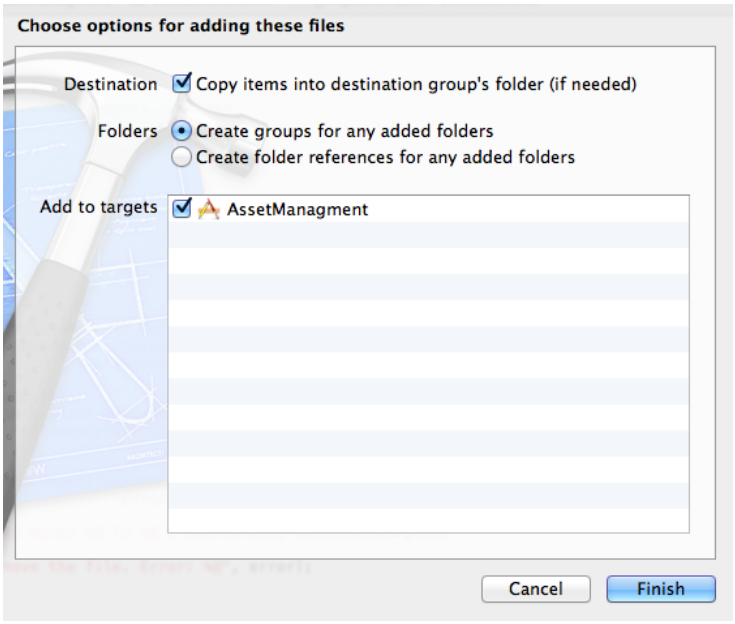

Figure 84: Selecting configuration options for the new media assets

After clicking **Finish**, you should see your file in the Xcode Project Navigator:

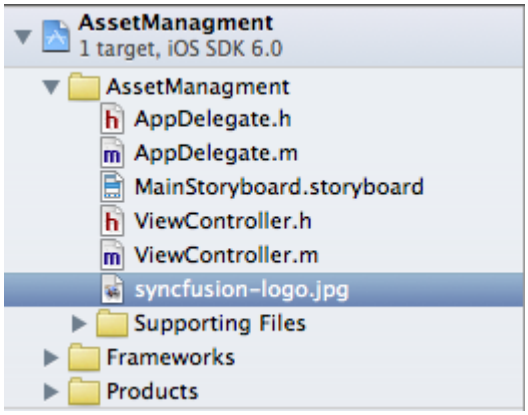

Figure 85: The media asset in the Project Navigator

Now that you have a custom resource in your application bundle, you can access it via **`NSBundle`**.

Accessing Bundled Resources

Instead of manually creating file paths with **`NSURL`**, you should always use the NSBundle class as the programmatic interface to your application bundle. It provides optimized search functionality and built-in internationalization capabilities, which we'll talk about in the next chapter.

The **`mainBundle`** class method returns the **`NSBundle`** instance that represents your application bundle. Once you have that, you can locate resources using the **`pathForResource:ofType:`**

instance method. iOS uses special file naming conventions that make it possible for **NSBundle** to return different files depending on how the resource is going to be used. Separating the file name from the extension allows **pathForResource:ofType:** to figure out which file to use automatically, and it allows **NSBundle** to automatically select localized files.

For example, the following code locates a JPEG called **syncfusion-logo** in the application bundle:

```
- (void)viewDidLoad {
    [super viewDidLoad];

    // Find the resource.
    NSString *imagePath = [[NSBundle mainBundle]
                           pathForResource:@"syncfusion-logo"
                           ofType:@"jpg"];
    NSLog(@"Image path: %@", imagePath);

    // Load the resource.
    UIImage *imageData = [[UIImage alloc]
                          initWithContentsOfFile:imagePath];
    if (imageData != nil) {
        NSLog(@"Image size: %.0fx%.0f",
              imageData.size.width, imageData.size.height);
    } else {
        NSLog(@"Could not load the file");
    }
}
```

Like text files, locating an image and loading it are separate actions. The first **NSLog()** call should display something like **/path/to/sandbox/AssetManagement.app/your-image.jpg** in the Xcode Output Panel.

The **UIImage** class represents the contents of an image file, and it works with virtually any type of image format (JPEG, PNG, GIF, TIFF, BMP, and a few others). Once you've gotten the image location with **NSBundle**, you can pass it to the **initWithContentsOfFile:** method of **UIImage**. If the file loaded successfully, you'll be able to access the image's dimensions through the **size** property, which is a **CGSize** struct containing the **width** and **height** floating-point fields.

While **UIImage** does define a few methods for drawing the associated image to the screen (namely, **drawAtPoint:** and **drawInRect:**), it's often easier to display it using the **UIImageView** class. Since it's a subclass of **UIView**, it can be added to the existing view hierarchy using the **addSubview:** method common to all view instances. **UIImageView** also provides a convenient interface for controlling animation playback. To display **UIImage** in the root view object, change the **viewDidLoad** method of **ViewController.m** to the following:

```
- (void)viewDidLoad {
    [super viewDidLoad];
```

```
    // Find the image.
    NSString *imagePath = [[NSBundle mainBundle]
                           pathForResource:@"syncfusion-logo"
                           ofType:@"jpg"];

    // Load the image.
    UIImage *imageData = [[UIImage alloc]
                          initWithContentsOfFile:imagePath];
    if (imageData != nil) {
        // Display the image.
        UIImageView *imageView = [[UIImageView alloc]
                                  initWithImage:imageData];
        [[self view] addSubview:imageView];
    } else {
        NSLog(@"Could not load the file");
    }
}
```

When you compile the project, you should see your image in the top-left corner of the screen. By default, the image will not be scaled. So, if your image is bigger than the screen, it will be cropped accordingly:

Figure 86: A cropped UIImageView object

To change how your **UIImageView** scales its content, you should use the **contentMode** property of **UIView**. It takes a value of type **UIViewContentMode**, which is an enum defining the following behaviors:

- **UIViewContentModeScaleToFill**—Scale to fill the view's dimensions, changing the image's aspect ratio if necessary.
- **UIViewContentModeScaleAspectFit**—Scale to fit into the view's dimensions, maintaining the image's aspect ratio.
- **UIViewContentModeScaleAspectFill**—Scale to fill the view's dimensions, maintaining the image's aspect ratio. This may cause part of the image to be clipped.
- **UIViewContentModeCenter**—Use the original image's size, but center it horizontally and vertically.
- **UIViewContentModeTop**—Use the original image's size, but center it horizontally and align it to the top of the view.
- **UIViewContentModeBottom**—Use the original image's size, but center it horizontally and align it to the bottom of the view.
- **UIViewContentModeLeft**—Use the original image's size, but center it vertically and align it to the left of the view.
- **UIViewContentModeRight**—Use the original image's size, but center it vertically and align it to the right of the view.
- **UIViewContentModeTopLeft**—Use the original image's size, but align it to the top-left corner of the view.
- **UIViewContentModeTopRight**—Use the original image's size, but align it to the top-right corner of the view.
- **UIViewContentModeBottomLeft**—Use the original image's size, but align it to the bottom-left corner of the view.
- **UIViewContentModeBottomRight**—Use the original image's size, but align it to the bottom-right corner of the view.

For example, if you want your image to shrink to fit into the width of the screen while maintaining its aspect ratio, you would use **UIViewContentModeScaleAspectFit** for the **contentMode**, and then change the width of the image view's dimensions to match the width of the screen (available via the **[UIScreen mainScreen]** object). Add the following to the **viewDidLoad** method from the previous example after the **[[self view] addSubview:imageView];** line:

```
CGRect screenBounds = [[UIScreen mainScreen] bounds];
imageView.contentMode = UIViewContentModeScaleAspectFit;
CGRect frame = imageView.frame;
frame.size.width = screenBounds.size.width;
imageView.frame = frame;
```

The **frame** property of all **UIView** instances defines the position and the visible area of the view (i.e. its dimensions.). After changing the width of the **frame**, the **UIViewContentModeScaleAspectFit** behavior automatically calculated the height of the frame, resulting in the following image:

Figure 87: Shrinking an image to fit the screen width

While this section extracted an *image* from the bundle, remember that it's just as easy to access other types of media. In the previous section, we saw how text files can be loaded with **NSString**; this works the same way with bundles. A video works in a similar fashion to an image in that iOS provides a dedicated class (MPMoviePlayerController) for incorporating it into an existing view hierarchy. Audio files are slightly different, since playback is not necessarily linked to a dedicated view. We'll discuss the audio capabilities of iOS later in this book. For now, let's go back to the application bundle.

Required Resources

In addition to any custom media assets your app might need, there are also three files that are *required* to be in your application bundle. These are described as follows:

- **Information property list**—The configuration file defining critical options like required device capabilities, supported orientations, etc.
- **App icon**—The icon that appears on the user's home screen. This is what the user taps to launch your application.
- **Launch image**—After the user launches your application, this is the image that briefly appears while it's loading.

Information Property List

A property list (also known as a “plist”) is a convenient format for storing structured data. It makes it easy to save arrays, dictionaries, dates, strings, and numbers into a persistent file and load it back into an application at run time. If you’ve ever worked with JSON data, it’s the same idea.

The **information property list** is a special kind of property list stored in a file called **`Info.plist`** in your application bundle. You can think of it as a file-based **`NSDictionary`** whose key-value pairs define the configuration options for your app. The **`Info.plist`** for our example app is generated from a file called **`AssetManagement-Info.plist`**, and you can edit it directly in Xcode by selecting it from the **Supporting Files** folder in the Project Navigator. When you open it, you should see something like the following:

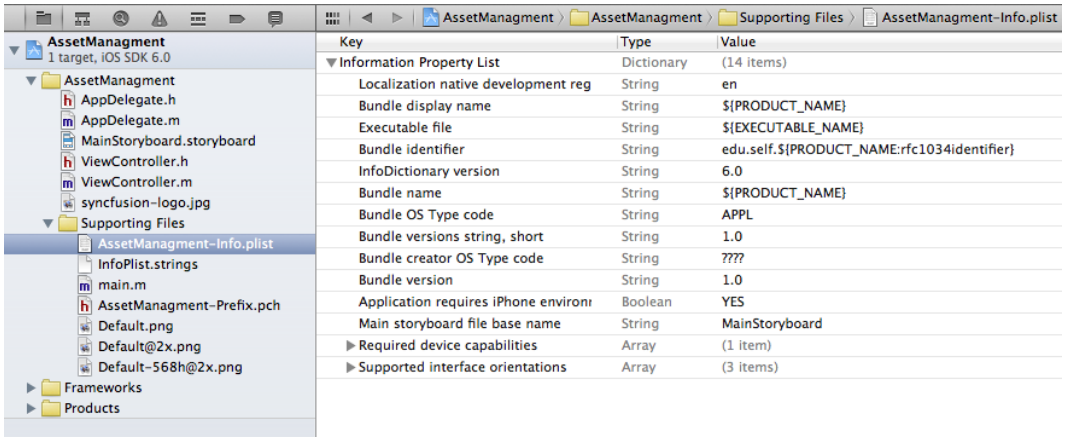

Figure 88: Opening the Info.plist file in Xcode

These are all of the configuration options provided by the template. The left-most column contains the option name, and the right-most one contains its value. Note that values can be Boolean, numbers, strings, arrays, or even entire dictionaries.

By default, keys are shown with human-readable titles, but it helps—especially when learning iOS for the first time—to see the raw keys that are referenced by the official documentation. To display the raw keys, Ctrl+click anywhere in the **`Info.plist`** editor and select **Show Raw Keys/Values**. These are the strings to use when you want to access configuration options programmatically (as discussed in the next section).

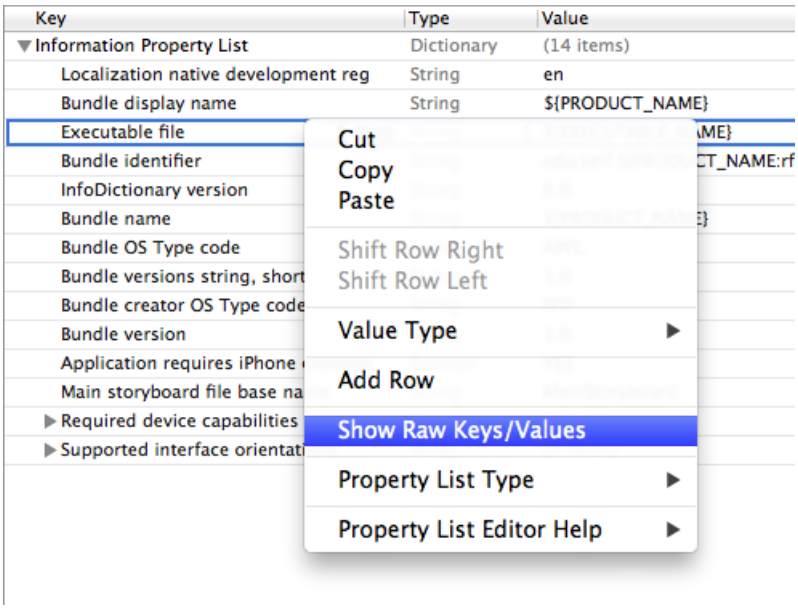

Figure 89: Displaying raw keys

The left-most column should now show keys like **`CFBundleDevelopmentRegion`**, **`CFBundleDisplayName`**, etc. The following keys are required by all **`Info.plist`** files:

- **`UIRequiredDeviceCapabilities`**—An array containing the device requirements for your app. This is one way that Apple determines which users can view your application in the App Store. Possible values are listed in the *UIRequiredDeviceCapabilities* section of the iOS Keys Reference.
- **`UISupportedInterfaceOrientations`**—An array defining the orientations your app supports. Acceptable values are:
  - **`UIInterfaceOrientationPortrait`**
  - **`UIInterfaceOrientationLandscapeLeft`**
  - **`UIInterfaceOrientationLandscapeRight`**
  - **`UIInterfaceOrientationPortraitUpsideDown`**
- **`CFBundleIconFile`**—An array containing the file names of all your app icons. We'll talk more about app icons in a moment.

The template provides defaults for the device requirements and the supported orientations, as shown in Figure 90.

▼UIRequiredDeviceCapabilities	Array	(1 item)
Item 0	String	armv7
▼UISupportedInterfaceOrientations	Array	(3 items)
Item 0	String	UIInterfaceOrientationPortrait
Item 1	String	UIInterfaceOrientationLandscapeLeft
Item 2	String	UIInterfaceOrientationLandscapeRight

Figure 90: Default values for device requirements and supported orientations

Accessing Configuration Options

Most of the configuration options defined in **`Info.plist`** are used internally by iOS, but you may occasionally need to access them manually. You can get an **`NSDictionary`** representation of **`Info.plist`** through the **`infoDictionary`** method of **`NSBundle`**. For example, if you wanted to perform a custom launch behavior based on a key in **`Info.plist`**, you could do a quick check in the **`application:didFinishLaunchingWithOptions:`** method of **AppDelegate.m**, like so:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    NSDictionary* infoDict = [[NSBundle mainBundle] infoDictionary];
    NSArray *supportedOrientations = [infoDict
objectForKey:@"UISupportedInterfaceOrientations"];
    if ([supportedOrientations
containsObject:@"UIInterfaceOrientationPortraitUpsideDown"]) {
        NSLog(@"Do something special to enable an upside-down app");
    } else {
        NSLog(@"Can assume the app won't display upside-down");
    }
    return YES;
}
```

App Icon(s)

Your app bundle must include at least one icon to display on the home screen, but it's also possible to specify multiple icons for different situations. For example, you might want to use a different design for the smaller icon that appears in search results, or use a high-resolution image for devices with Retina displays. Xcode takes care of all of this for you.

The **`CFBundleIconFiles`** key in **`Info.plist`** should be an array containing the file names of all your app icons. With the exception of the App Store icon, you can use any name you like for the files. Note that you do *not* need to specify the intended usage of each file—iOS will select the appropriate icon automatically based on the dimensions.

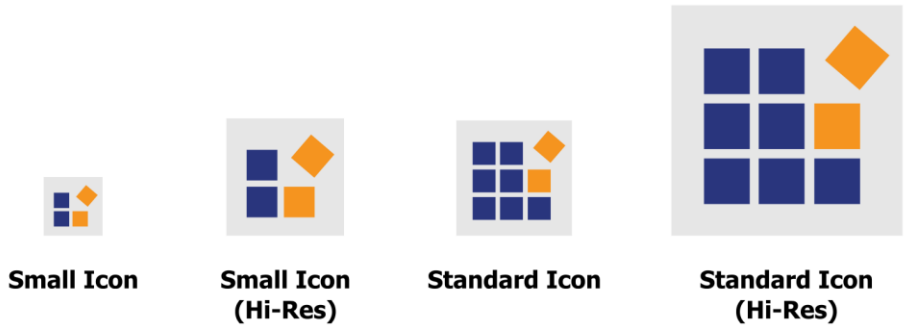

Figure 91: Custom iPhone app icons with high-resolution versions for Retina displays

If your app supports devices with Retina displays, you should also include a high-resolution version of each icon and give it the same file name with **`@2x`** appended to it. For example, if your main icon file was called **`app-icon.png`**, the Retina display version should be called **`app-icon@2x.png`**.

The following table lists the standard iOS app icons, but be sure to visit the iOS Human Interface Guidelines for a more detailed discussion. This document also provides extensive guidelines for creating icon graphics. All icons should use the PNG format.

Icon Type	Platform	Required	Standard Size	Retina Size	Description
App Store Icon	iPhone and iPad	Yes	512 × 512	1024 × 1024	The image presented to customers in iTunes. This file *must* be called **`iTunesArtwork`** or **`iTunesArtwork@2x`** (with no extension).
Main Icon	iPhone	Yes	57 × 57	114 × 114	The icon that appears on the iPhone home screen.
Main Icon	iPad	Yes	72 × 72	144 × 144	The icon that appears on the iPad home screen.
Small Icon	iPhone	No	29 × 29	58 × 58	The icon displayed next to search results and in the Settings app for iPhone.
Small Icon	iPad	No	50 × 50	100 × 100	The icon displayed next to search results and in the Settings app for iPad.

Next, you're going to add an icon to the example application. In the resource package for this book, you'll find four sample icons called **`app-icon.png`**, **`app-icon@2x.png`**, **`app-icon-small.png`**, and **`app-icon-small@2x.png`**. Drag all of these into the Project Navigator to add them to your application bundle, and then open **`AssetManagement-Info.plist`** and make sure you're looking at raw keys or values. Add a new row and type *CFBundleIconFiles* for the key (not to be confused with **`CFBundleIcons`** or **`CFBundleIconFile`**). This will automatically add an empty item to the array, which you can view by clicking the triangle next to the **`CFBundleIconFiles`** item. Add all four of the icon files to the array, so it looks like the following (the order doesn't matter):

CFBundleIconFiles	Array	(4 items)
Item 0	String	app-icon.png
Item 1	String	app-icon@2x.png
Item 2	String	app-icon-small.png
Item 3	String	app-icon-small@2x.png

Figure 92: Adding icon files to Info.plist

Now, when you run your project, you should see a custom app icon in the home screen. You may need to restart the iOS Simulator for this to work.

Figure 93: The custom app icon in the home screen

Try dragging the home screen to the right a few times until you reach the search screen. If you start typing “assetmanagement,” you should see the small version of the icon in the search results, as shown in Figure 94:

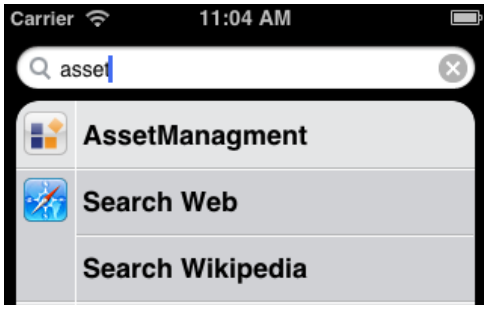

Figure 94: The small app icon used in search results

And that's all there is to customizing the icons for your iPhone application.

Launch Image(s)

The final required media asset for any iOS application is a launch image. A launch image is displayed immediately when the user opens your application. The idea is to give the user the impression that your app launched immediately, even though it may take a few seconds to load. Apple discourages developers from using launch images as an about page or a splash screen. Instead, it should be a skeleton of your app's initial screen.

For example, consider the master-detail application we built in the previous chapter. The ideal launch image would simply be an empty master list:

Figure 95: Appropriate launch image for a master-detail application

As you can see, a launch image is essentially a screenshot of your app's initial screen, minus any dynamic data. This avoids any abrupt changes in the UI. Again, the idea is to downplay the application launch by making the transition as seamless as possible from app selection, to launch image, to the initial screen. The iOS Simulator has a convenient screen capture tool for creating launch images. Once your application is done, run it in the simulator and navigate to **Edit** > **Copy Screen**.

Like app icons, it's possible to have multiple launch images depending on the device or screen resolution. Launch images use the same **@2x** affix for high-resolution images, but it's also possible to target specific devices by appending a **usage modifier** immediately after the base name. For example, iPhone 5 has different screen dimensions than previous generations, and thus requires its own launch image. To tell iOS to use a particular file for iPhone 5 devices, you would append **-568h** to its base name, giving you something like **launch-image-568h@2x.png** (note that iPhone 5 has a Retina display, so the associated launch image will always have **@2x** in its filename).

The following table lists the dimension requirements for iOS launch images.

Platform	Standard Size	Retina Size
iPhone (up to 4th generation)	320 × 480	640 × 960
iPhone (5th generation)	640 × 1136	640 × 1136
iPad	768 × 1004	1536 × 2008

Once you have your launch image, adding it to your application is very similar to configuring app icons. First, add the files to the top level of your application bundle, and then add the base name to the **UILaunchImageFile** key of your **Info.plist**. For example, if your launch images were called **launch-image.png**, **launch-image@2x.png**, and **launch-image-568h@2x.png**, you would use **launch-image** as the value for **UILaunchImageFile**.

If you don't specify a **UILaunchImageFile**, iOS will use the files **Default.png**, **Default@2x.png**, and **Default-568h@2x.png**. These default launch images are provided by the Xcode templates.

Summary

This chapter covered many of the built-in asset management tools in iOS. We talked about an app's sandbox file system and how to read and write dynamic files from various predefined directories. We also looked at the application bundle, which contains all of the resources that will be distributed with the app. In addition to any custom media assets, the application bundle must include an information property list, app icons, and launch images.

Bundles are a convenient way to distribute an application, but they also provide built-in internationalization capabilities. In the next chapter, we'll learn how to show different files to different users based on their language settings. And, thanks to **NSBundle**, this will require very little additional code.

Chapter 4 Localization

So far, all of our example projects have assumed that our apps were destined for English speakers, but many applications can benefit from being available to non-English-speaking audiences. The App Store takes care of presenting our app to the right audience, but it's our job as developers to configure it in such a way that the appropriate resources are displayed to users from different regions. This process is called **localization**.

Fortunately, iOS makes it surprisingly easy to localize resources using bundles. The **`NSBundle`** class automatically selects the appropriate asset by taking into account the user's preferred language. For example, if you've provided different versions of the same image for English speakers versus Spanish speakers, the **`pathForResource:ofType:`** method discussed in the previous chapter returns different file paths depending on the user's settings. This is one of the primary reasons you shouldn't directly access bundle resources using hardcoded paths.

The three aspects of an app that typically need to be localized are images, audio, or videos containing a specific language, hardcoded strings, and storyboards. In this chapter, we'll take a brief look at localizing media resources and hardcoded strings using **`NSBundle`**'s built-in internationalization capabilities. Storyboard files can be localized using the same process.

Creating the Example Application

The example for this chapter is a simple application that displays different images or strings based on the user's preferred language. Create a new Single View Application and call it "Internationalization." As always, **Use Storyboards**, and **Use Automatic Reference Counting** should be selected.

Enabling Localization

The first step to making an application multilingual is to add the supported languages to the project. In the project navigator, select the project icon:

Figure 96: Selecting the project in the Project Navigator

Then, select the **Internationalization** project in the left column (not to be confused with the Internationalization target). Make sure the **Info** tab is selected; you should see the following window:

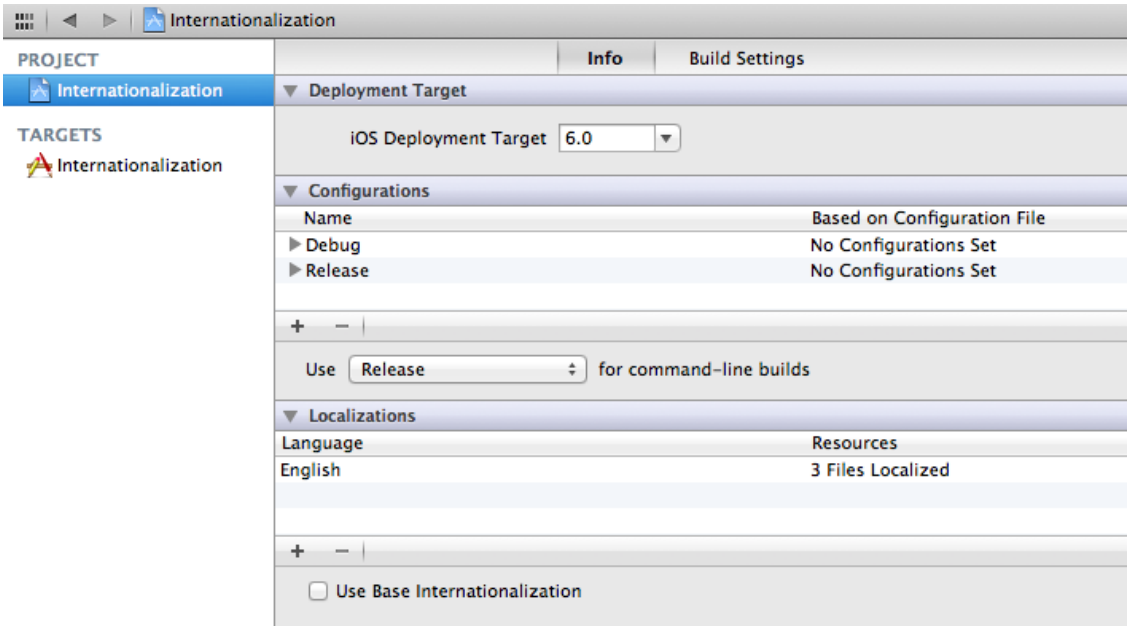

Figure 97: The Project Info window

To add support for another language, select the plus sign under the **Localizations** section. You can pick any language you like, but this book will use Spanish. Selecting a language will open a dialog asking which files should be localized. Clear the selection of **MainStoryboard.storyboard**, but leave **InfoPlist.strings** selected.

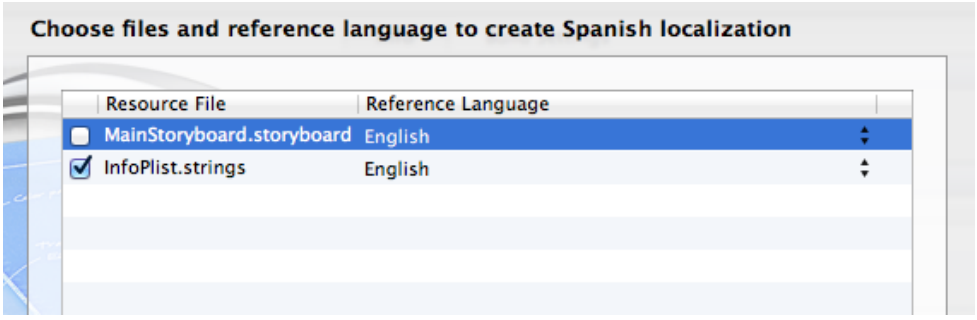

Figure 98: Adding a Spanish localization

It's now possible to add a Spanish version of each resource to the application bundle.

Localizing Images

Next, we'll look at localizing media assets. In the resource package for this book, you'll find a file called **`syncfusion-icon-en.png`**. Add this file to the application bundle by dragging it to the Project Navigator and rename it as **`syncfusion-icon.png`**. Then, display it in the view by changing the **`viewDidLoad`** method in **ViewController.m** to the following:

```
- (void)viewDidLoad {
    [super viewDidLoad];

    // Find the image.
    NSString *imagePath = [[NSBundle mainBundle]
                           pathForResource:@"syncfusion-icon"
                           ofType:@"png"];
    NSLog(@"%@", imagePath);

    // Load the image.
    UIImage *imageData = [[UIImage alloc]
                          initWithContentsOfFile:imagePath];
    if (imageData != nil) {
        // Display the image.
        UIImageView *imageView = [[UIImageView alloc]
                                  initWithImage:imageData];
        CGRect screenBounds = [[UIScreen mainScreen] bounds];
        imageView.contentMode = UIViewContentModeCenter;
        CGRect frame = imageView.frame;
        frame.size.width = screenBounds.size.width;
        frame.size.height = screenBounds.size.height;
        imageView.frame = frame;
        [[self view] addSubview:imageView];
    } else {
        NSLog(@"Could not load the file");
    }
}
```

When you compile the project, you should see a small icon displayed in the middle of the screen:

Figure 99: Programmatically adding an image to the view

You should also see the path **`Internationalization.app/syncfusion-icon.png`** in the Output Panel. Nothing new here, just an image at the top level of the application bundle—but, this is going to change once we localize the image file.

To do this, select the image in the Project Navigator, open the **Utilities** panel, and click **Make localized** under the **Localization** section.

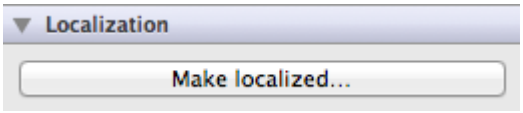

Figure 100: Creating a localized file

The next dialog prompts you to choose a language. Select **English** and click **Localize**.

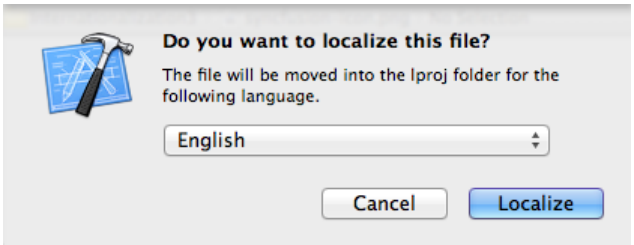

Figure 101: Configuring the localization

This tells iOS that this version of **syncfusion-icon.png** is for English speakers. We'll add a Spanish version in a moment, but first let's look at what's happening behind the scenes. To see your localizations in action, you'll have to reset the iOS Simulator and do a clean build. To reset the simulator, navigate to **iOS Simulator** > **Reset Content and Settings** in the menu bar and select **Reset** in the resulting dialog.

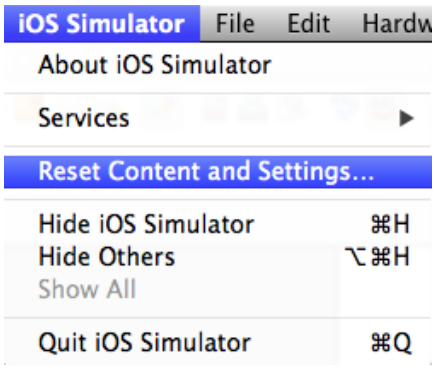

Figure 102: Resetting the iOS Simulator

Quit the simulator and go back to the Internationalization project in Xcode. To do a clean build, navigate to **Product** > **Clean** in the menu bar and compile the project again as you normally would. You should see a different file path in the Output Panel:

```
Internationalization.app/en.lproj/syncfusion-icon.png.
```

The new **`en.lproj/`** subdirectory is the internal way of organizing language-specific files in iOS. All the resources localized in English will appear in this subdirectory, and all of the Spanish versions will appear in the **es.lproj/** subdirectory. But again, we don't actually have to know where the file resides; **`NSBundle`**'s **`pathForResource:ofType:`** method figures it out automatically.

So, our English version of the image is set up. Next, we need to configure the Spanish version. Select the English version of the file in the Project Navigator, and select the check box next to **Spanish** in the **Localization** section of the **Utilities** panel.

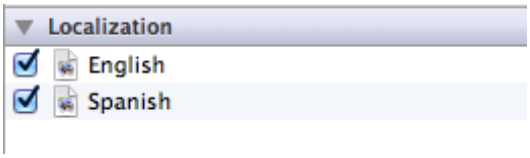

Figure 103: Adding a Spanish version of the file

This copies the existing English-language version of **syncfusion-icon.png** into the **es.lproj/** subdirectory. Back in the Project Navigator, you should be able to see this by expanding the **syncfusion-icon.png** file.

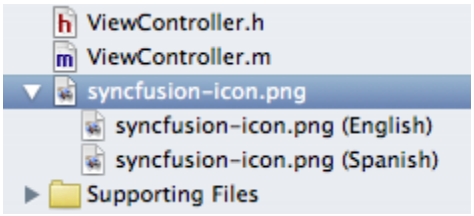

Figure 104: Both versions of the image file in the Project Navigator

Of course, we need to replace the Spanish version with a completely different file. The easiest way to do this is by selecting the **syncfusion-icon.png** (Spanish) file and clicking the arrow icon next to the **Full Path** string in the **Utilities** panel:

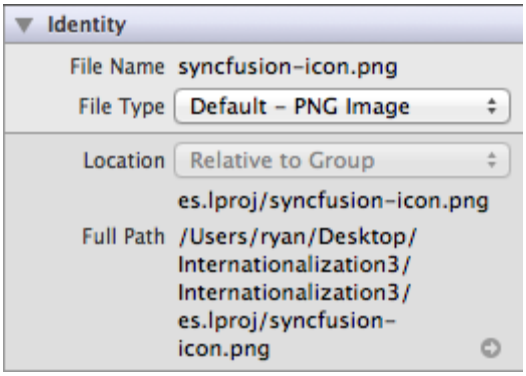

Figure 105: The Utilities Panel for the Spanish image file

This displays the contents of the **es.lproj/** folder in the Finder, which gives us the opportunity to replace the file manually. Delete the existing **syncfusion-icon.png** file and copy the **syncfusion-icon-es.png** file from the resource package into **es.lproj/**. Make sure to rename it as **syncfusion-icon.png**. It's important for localized versions of the same file to have identical file names so **`NSBundle`** can find them. After replacing the file, you should see different images when you select the two localizations in Xcode.

That should be it for localizing our image file. To test it out, you can change the device language the same way you would change it in a real device—through the Settings app. Click the device's home button in the simulator, click and drag the screen to the right, and launch the *Settings* application. Under **General** > **International** > **Language**, you can select the device language.

Figure 106: Changing the device language in iOS Simulator

Choose **Español**, and re-open your application. You should see the Spanish version of **syncfusion-icon.png** (you might need to close the simulator and compile the program again). Also note that the file path output by **`NSLog()`** now reads: **`Internationalization.app/es.lproj/syncfusion-icon.png`**.

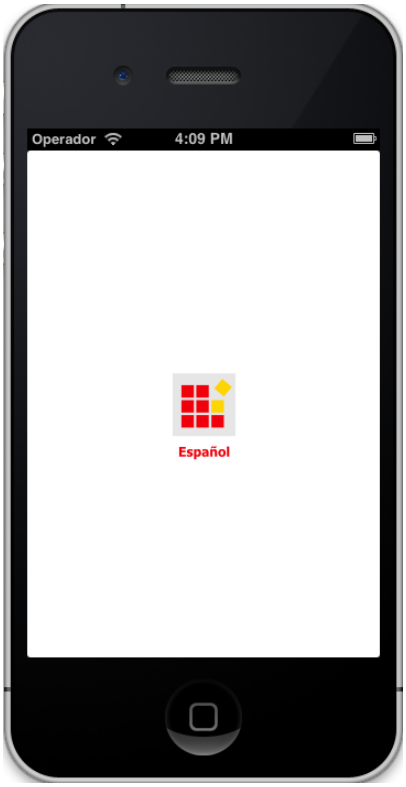

Figure 107: Displaying the localized version of the image file

As you can see, it's incredibly easy to localize files using **`NSBundle`**'s built-in functionality. The idea is to use **`NSBundle`** as an abstraction between your application code and the assets that they rely on. This isolates the localization process from the development process, making it very easy to outsource translations.

Localizing video and audio files uses the exact same process just discussed. However, preparing text for an international audience requires a little bit more work.

Localizing Text

When you're dealing with a multilingual app, hardcoded strings must be abstracted into a bundle asset so that **`NSBundle`** can load the correct language at run time. iOS uses what's called a **strings file** to store translations of all the string literals in your application. After creating this strings file, you can localize it using the same method discussed in the previous section.

Let's change our **`viewDidLoad`** method to display a button and output a message when the user taps it:

```
- (void)viewDidLoad {
    [super viewDidLoad];

    UIButton *aButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
```

```
    [aButton setTitle:@"Say Hello" forState:UIControlStateNormal];
    aButton.frame = CGRectMake(100.0, 200.0, 120.0, 40.0);
    [[self view] addSubview:aButton];

    [aButton addTarget:self
                action:@selector(sayHello:)
      forControlEvents:UIControlEventTouchUpInside];
}

- (void)sayHello:(id)sender {
    NSLog(@"Hello, World!");
}
```

These methods have two string literals that we'll have to move into a strings file: **`@"Say Hello"`** and **`@"Hello, World!"`**.

To create the strings file, create a new file and choose **Resource** > **Strings File**. Use **`Localizable.strings`** for the file name, which is the default strings file that iOS looks for.

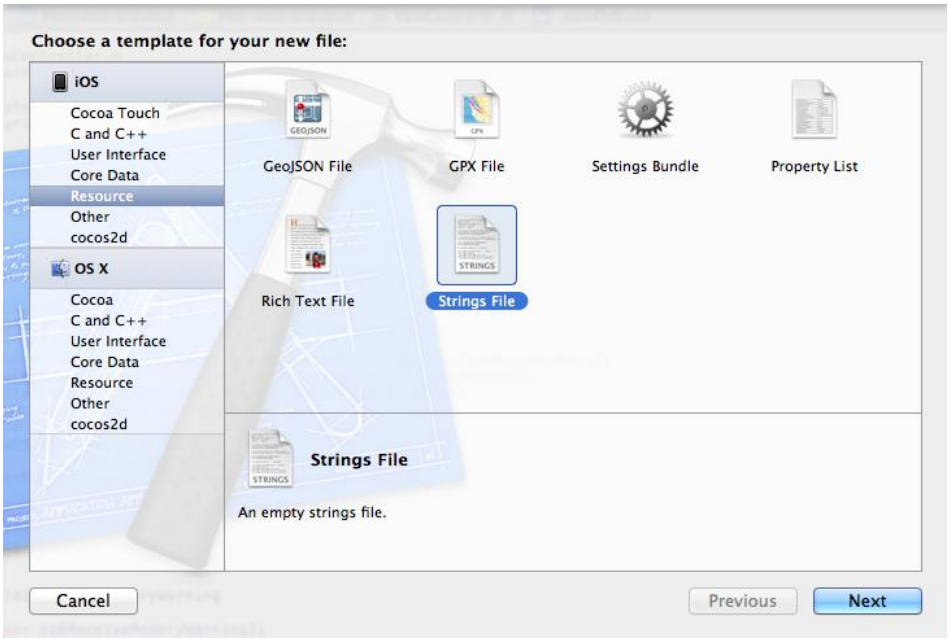

Figure 108: Creating a strings file

The content of the strings file is a simple list of key or value pairs, formatted as follows:

```
"Button Title" = "Say Hello";
"Greeting" = "Hello, World!";
```

The left side is the key that you'll use to reference the translated string in your application code. The keys are arbitrary strings, but developers typically use either a semantic name describing

how the string will be used, or the target phrase in their native language. In our strings file, we opted for the former. The values for each key follow an equal sign. Be sure to include a semicolon at the end of each line or terrible things will happen when you try to run your application.

As with media assets, you can access the contents of **Localizable.strings** via **`NSBundle`**. The **`localizedStringForKey:value:table:`** method returns the value of a key from a particular strings file. The **`value`** argument lets you specify a default return value if the key isn't found, and the **`table`** argument determines which strings file to use. When you specify **`nil`** for **`table`**, the default **Localizable.strings** file is used.

Since accessing translated strings is such a common task, the Foundation Framework also provides a convenient **`NSLocalizedString()`** macro that you can use as a simple shortcut for **`localizedStringForKey:value:table:`**. It passes an empty string for the **`value`** argument and **`nil`** for the **`table`** argument. For most applications, **`NSLocalizedString()`** is all you really need to access localized text.

So, let's change our button's title configuration to use **`NSLocalizedString()`**:

```
[aButton setTitle:NSLocalizedString(@"Button Title", nil)
         forState:UIControlStateNormal];
```

If you compile the project, the button should still read, "Say Hello"—but now it's loaded from **Localizable.strings**. Let's do the same for the **`sayHello`** method:

```
- (void)sayHello:(id)sender {
    NSLog(@"%@", NSLocalizedString(@"Greeting", nil));
}
```

Now that our strings are dynamically loaded instead of being hardcoded, it's trivial to localize them. We'll use the exact same process as with images. In the Project Navigator, select the **`Localizable.strings`** file, then click **Make localized** in the **Utilities** panel. Select **English** in the resulting dialog box to use this version of the file for English-speaking users.

To add a Spanish version, select **`Localizable.strings`** again and select the check box next to **Spanish** in the **Localizations** section.

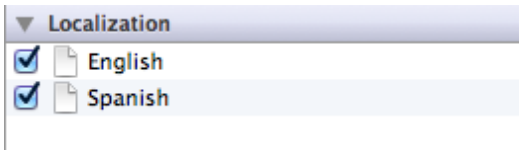

Figure 109: Adding a Spanish version of Localizable.strings

Just like **syncfusion-icon.png**, you should be able to expand the **Localizable.strings** file in the Project Navigator.

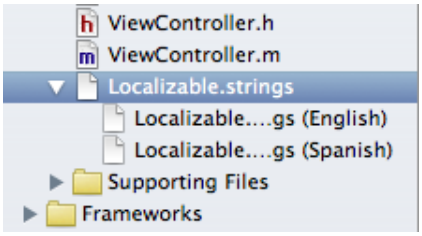

Figure 110: Expanding the strings file to view its localizations

Finally, add some translations to the Spanish version of the file:

```
"Button Title" = "Dice Hola";
"Greeting" = "Hola, Mundo!";
```

You can test it the same way we tested images. Navigate to **Reset Content and Settings** in the simulator, close the simulator, and do a clean build from Xcode. After changing the language to **Español**, your button should read **Dice Hola** instead of **Say Hello**, and clicking it should output "Hola, Mundo!"

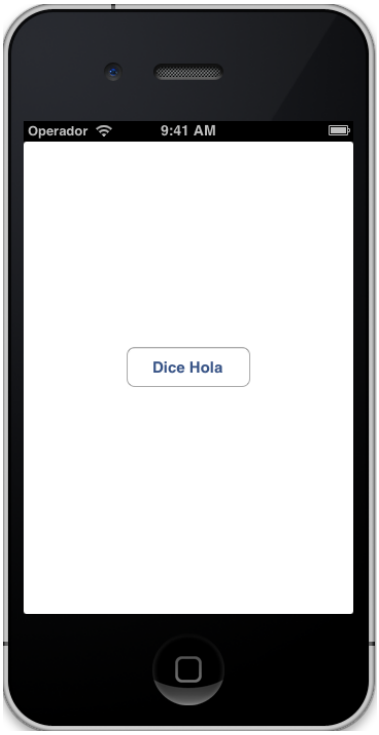

Figure 111: Changing the device language to Spanish

That's all there is to localizing strings in an iOS application. Again, having all your translated text

in a single file—entirely abstracted from your application code—makes it easy to outsource your localization efforts. This is a very good thing, as most developers don't fluently speak all of the languages that they would like to translate their app into.

Localizing Info.plist

There is one important detail that hasn't been addressed yet—localizing the app name. If you take a look at the home screen in the iOS Simulator, you'll notice that the title under your app icon hasn't been translated to Spanish. If you've already gone through the trouble of localizing the string *inside* your app, you might as well take the time to translate a little bit of metadata too.

An app's display name is defined in the **`Info.plist`** under the **`CFBundleDisplayName`** key. Instead of forcing you to translate values in the main **`Internationalization-Info.plist`** file, iOS gives you a dedicated strings file for overwriting certain configuration options with localized values. In the **Supporting Files** group of the Project Navigator, open the **InfoPlist.strings** file. This is just like the Localizable.strings file we created in the previous section, except it should only provide values for **`Info.plist`** keys. Add the following to your **InfoPlist.strings** file:

```
"CFBundleDisplayName" = "Hola, Mundo!";
```

Now, if you reset the simulator and do a clean build, you should see a Spanish title under your application icon.

Figure 112: Localizing the app display name

Summary

In this chapter, we learned how to localize media assets, text, and metadata using **`NSBundle`**. By abstracting the resources that need to be localized into isolated files and referencing them indirectly via methods like **`pathForResource:ofType:`**, it's possible to translate your application into another language without touching a single line of application code. This is a very powerful feature of iOS, especially considering the international prevalence of iPhone and iPad devices.

The final chapter of *iOS Succinctly* takes a brief look at the built-in audio support for iOS applications. As we touched on in previous chapters, audio files use the same bundle structure as images and strings files. However, instead of focusing on how to access those resources, we'll discuss the higher-level tools for controlling audio playback.

Chapter 5 Audio

iOS provides several options for working with audio. In this chapter, we'll introduce two frameworks for dealing with different types of sounds. The Audio Toolbox Framework includes a C library for playing simple system sounds, and the AVFoundation Framework provides an object-oriented interface for intermediate-level playback and recording.

For both frameworks, interacting with audio files is conceptually the same as interacting with images. You still use **`NSBundle`** to locate the file on disk, and then load its contents with another class dedicated to interpreting the different audio formats.

The system sound API is defined in the Audio Toolbox Framework. This entire framework is written in C instead of Objective-C, so we'll be working with the C interface to an iOS application for the first half of this chapter. This changes how we'll be interacting with core iOS objects, but don't let that scare you—we're still dealing with the same objects and concepts as we have been throughout this entire book. For example, instead of using the **`mainBundle`** method of **`NSBundle`**, we're going to use a C function called **`CFBundleGetMainBundle()`** to access the application bundle.

Creating the Example Application

As with the previous chapter, all we're going to need is a simple Single View Application. Create a new project and call it *Audio*. As usual, use *edu.self* for the **Company Identifier**, *iPhone* for **Devices**, and make sure both the **Use Storyboards** and **Use Automatic Reference Counting** check boxes are selected.

Unlike the previous chapter, we need to access two audio frameworks, which aren't included in the template. To add a new framework to the project, click the project icon in the Project Navigator and select the **Audio** target. In the **Summary** tab, scroll down to the **Linked Frameworks and Libraries** section.

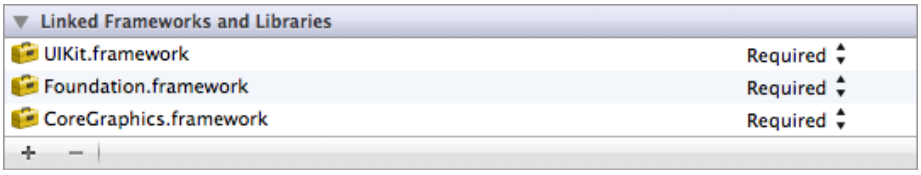

Figure 113: The frameworks currently included in our project

These are the code libraries you have access to from the application code. You need to add the Audio Toolbox Framework in order to work with system sounds, so click the plus sign in the bottom-left corner. The resulting dialog contains a list of all of the available frameworks and libraries. Start typing "audiotoolbox" in the search bar to find our desired framework.

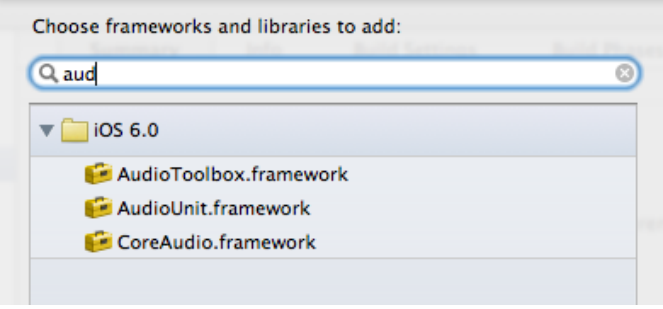

Figure 114: Searching for the Audio Toolbox Framework

Click **AudioToolbox.framework** and then click **Add** to include it in the project. We'll also need access to the AVFoundation Framework for the **`AVAudioPlayer`** portion of this chapter, so repeat the process for this library, too. You should see both **AVFoundation.framework** and **AudioToolbox.framework** in the **Summary** tab before moving on.

Figure 115: All of the required libraries for our example project

We're now ready to start working with system sounds and **`AVAudioPlayer`**.

System Sounds

System sounds are designed to be used for simple things like alerts and UI sound effects (e.g., button clicks). They are very easy to use, but the trade-off is that they do not provide many options for controlling playback. The major limitations of the system sound API are:

- Sounds cannot be longer than 30 seconds.
- You cannot control the volume of system sounds (they use the current device volume).
- Only one sound can play at a time.
- Sounds cannot be looped or played after a certain delay.
- They only support .caf, .aif, and .wav files.
- It's not possible to configure how sounds deal with interruptions (e.g., and incoming call).

If you find these are too restricting, you should use the **`AVAudioPlayer`** discussed later in this chapter. Again, system sounds are meant for specific use cases. If your application is using sounds for more than UI feedback and alerts, you should probably use **`AVAudioPlayer`**.

Accessing the Sound File

The resource package for this book contains a short sound file called **blip.wav**. Add this to the application bundle by dragging it into the Project Navigator. Remember that system sounds use a C interface, so we'll be accessing this file using a slightly different method than previous chapters. In **ViewController.m**, change **`viewDidLoad`** to the following:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    CFURLRef blipURL = CFBundleCopyResourceURL(CFBundleGetMainBundle(),
                                               CFSTR ("blip"),
                                               CFSTR ("wav"),
                                               NULL);
    NSLog(@"%@", blipURL);
}
```

`CFBundleGetMainBundle()` is essentially the same as **`[[NSBundle] mainBundle]`**, except it returns the CoreFoundation object that represents the application bundle instead of the Foundation Framework's version. The CoreFoundation Framework is a lower-level alternative to the Foundation Framework. We need to use CoreFoundation objects here because that's what the Audio Toolbox accepts as arguments.

The **`CFBundleCopyResourceURL()`** is the CoreFoundation version of **`NSBundle`**'s **`pathForResource:ofType:`** method. It simply returns the path to the requested resource, and you should see a familiar file path in the **`NSLog()`** output.

Once you have a path to the sound file, you need to create a sound object using the **`AudioServicesCreateSystemSoundID()`** function. Add the following to **`viewDidLoad`**:

```
AudioServicesCreateSystemSoundID(blipURL, &_blipOne);
```

This function reads in the content of the sound file, turns it into a sound object that the system sound API knows how to play, and returns an ID you can use to reference it later. **`AudioServicesCreateSystemSoundID()`** is defined by the Audio Toolbox Framework, so you need to import it into **ViewController.m** before you can use it. Add the following to the top of the file:

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

You're storing the sound object's ID in a private instance variable called **`_blip`**, so add that to the implementation declaration, too:

```
@implementation ViewController {
    SystemSoundID _blip;
}
```

Playing the Sounds

Next, programmatically add a button to the scene; that button will play the sound when clicked.

```
UIButton *aButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
    [aButton setTitle:@"Blip"
             forState:UIControlStateNormal];
    [aButton addTarget:self
                action:@selector(playBlip:)
      forControlEvents:UIControlEventTouchUpInside];
aButton.frame = CGRectMake(100.0, 200.0, 120.0, 40.0);
[[self view] addSubview:aButton];
```

For the **`playBlip:`** action, we'll call the **`AudioServicesPlaySystemSound()`** function to play the sound. It accepts a single argument, which should be the **`SystemSoundID`** that refers to the desired sound object (e.g., **`_blip`**).

```
- (void)playBlip:(id)sender {
    AudioServicesPlaySystemSound(_blip);
}
```

When you compile the project, you should be able to click the button and hear a short sound (make sure your speakers are not muted).

Figure 116: A blipping button

If you click the button while the sound is still playing, you'll notice that iOS cancels the current sound and begins a new one. Again, system sounds are intended for UI sound effects and other straightforward applications. To play multiple sounds simultaneously, you'll need to upgrade to the **`AVAudioPlayer`** discussed in the following section.

`AudioServicesCreateSystemSoundID()` and **`AudioServicesPlaySystemSound()`** are pretty much all there is to system sounds. You may also find the **`AudioServicesPlayAlertSound()`** function useful. It works exactly like **`AudioServicesPlaySystemSound()`**, but it makes the phone vibrate if the user has it enabled.

AVAudioPlayer

The AVFoundation Framework provides a higher-level interface for managing audio playback. For games, custom music players, and other apps that require sophisticated audio controls, this is the framework you should use. Some of its capabilities are:

- There is no limitation on sound duration.
- You can play multiple sounds at the same time.

- Volume can be controlled for individual sounds.
- It's possible to loop a sound or perform other actions when it finishes playing.
- You can jump to arbitrary points in the sound file.
- It's easy to configure behavior for handling interruptions through a delegate object.

AVAudioPlayer is the main class you'll use from the AVFoundation Framework. It provides an object-oriented way to set the volume, play the sound, stop the sound, specify what part of the sound to play, etc. You can think of it as the audio equivalent of **UIImage/UIImageView**.

Supported file formats for **AVAudioPlayer** include .m4a, .aac, .wav, .mp3, .aif, .pcm, .caf, and a few others. See Apple's Multimedia Programming Guide for a detailed discussion of audio file formats.

Accessing the Song

AVAudioPlayer is designed to be used with longer sound files, like songs, so we've distributed a public domain Billie Holiday tune with this book. Drag the **good-morning-heartache.mp3** file from the resource package into the Project Navigator to add it to the bundle.

Since **AVFoundation** is an Objective-C library, we can go back to using **NSBundle** to access media assets. Go ahead and remove everything in **viewDidLoad**, and replace it with the following:

```
NSURL *soundFileURL = [[NSBundle mainBundle]
                       URLForResource:@"good-morning-heartache"
                        withExtension:@"mp3"];
    _player = [[AVAudioPlayer alloc] initWithContentsOfURL:soundFileURL
                                                     error:nil];
```

The **URLForResource:withExtension:** method is the **NSURL** equivalent to **pathForResource:ofType:**. This was a better option for finding the audio file since you needed an **NSURL** object to initialize the **AVAudioPlayer**. The **initWithContentsOfURL:error:** method loads the content of the sound file into the **AVAudioPlayer** instance, much like **NSString**'s **stringWithContentsOfURL:encoding:error:** method.

AVAudioPlayer resides in the **AVFoundation** library, so be sure to add the following import statement to **ViewController.h** (we'll need to access it from the header file later on).

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

In addition, you'll need a few private instance variables. Change the implementation declaration to the following:

```
@implementation ViewController {
    AVAudioPlayer *_player;
```

```
    UIButton *_playButton;
    UIButton *_stopButton;
}
```

Playing the Song

Next, add a **Play** button to the scene:

```
_playButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
[_playButton setTitle:@"Play"
             forState:UIControlStateNormal];
[_playButton addTarget:self
                action:@selector(playOrPause:)
      forControlEvents:UIControlEventTouchUpInside];

_playButton.frame = CGRectMake(100.0, 170.0, 120.0, 40.0);
[[self view] addSubview:_playButton];
```

The **`playOrPause:`** action should be implemented like this:

```
- (void)playOrPause:(id)sender {
    if (_player.playing) {
        [_player pause];
        [_playButton setTitle:@"Play" forState:UIControlStateNormal];
    } else {
        [_player play];
        [_playButton setTitle:@"Pause" forState:UIControlStateNormal];
    }
}
```

This checks to see if the song is already playing via **`AVAudioPlayer`**'s **`playing`** property. If it is, it pauses the song using the **`pause`** method and changes the button's title accordingly. If it's not, it starts the song by calling the **`play`** method. You should now be able to compile the app and play or pause a song by tapping the button.

Let's take this one step further by creating a **Stop** button. Add the following to the **`viewDidLoad`** method:

```
_stopButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
[_stopButton setTitle:@"Stop"
             forState:UIControlStateNormal];
[_stopButton addTarget:self
                action:@selector(stop:)
      forControlEvents:UIControlEventTouchUpInside];
```

```
_stopButton.frame = CGRectMake(100.0, 230.0, 120.0, 40.0);
[[self view] addSubview:_stopButton];
```

The corresponding action calls the **AVAudioPlayer**'s **stop** method. Note that this method does *not* change current playback position—it differs from the **pause** method only in that it undoes the preloading performed by the **play** method. To jump back to the beginning of the song like you would expect from a typical **Stop** button, you need to manually set the **currentTime** property of **AVAudioPlayer**, like so:

```
- (void)stop:(id)sender {
    [_player stop];
    _player.currentTime = 0;
    [_playButton setTitle:@"Play" forState:UIControlStateNormal];
}
```

AVAudioPlayer Delegates

Another advantage of using **AVAudioPlayer** over the system sounds API is that it allows you to handle interruptions using the familiar delegate pattern. The **AVAudioPlayerDelegate** protocol defines a number of methods that let you know when certain events occur outside of your application.

When an interruption (e.g., an incoming phone call) begins, the **audioPlayerBeginInterruption:** method is called on the delegate object. The sound will pause automatically, but this gives you a chance to write any other custom handling code you may need. Likewise, the **audioPlayerEndInterruption:withOptions:** method is called when the interruption is over and you can use the audio player again. However, the system does not automatically resume playback—you need to manually call the **play** method if you want this to happen.

In addition to interruption handling, the delegate object also lets you know when the sound has finished playing. By defining an **audioPlayerDidFinishPlaying:successfully:** method on the delegate object, you can do custom cleanup work. For example, you can use this to reset the play button and jump back to the beginning of the song, like so:

```
- (void)audioPlayerDidFinishPlaying:(AVAudioPlayer *)player successfully:(BOOL)flag {
    [self stop:nil];
    NSLog(@"Song finished playing!");
}
```

For this code to work, you need to make the view controller a formal delegate for the **AVAudioPlayer**. In **ViewController.h**, change the interface declaration to the following:

```
@interface ViewController : UIViewController <AVAudioPlayerDelegate>
```

Then, the view controller needs to assign itself as the delegate object. In the **`viewDidLoad`** method of **ViewController.m**, add the following line:

```
_player.delegate = self;
```

Now, if you let the song play through to the end, the **Pause** button will automatically turn back into a **Play** button, and clicking it will start the song from the beginning.

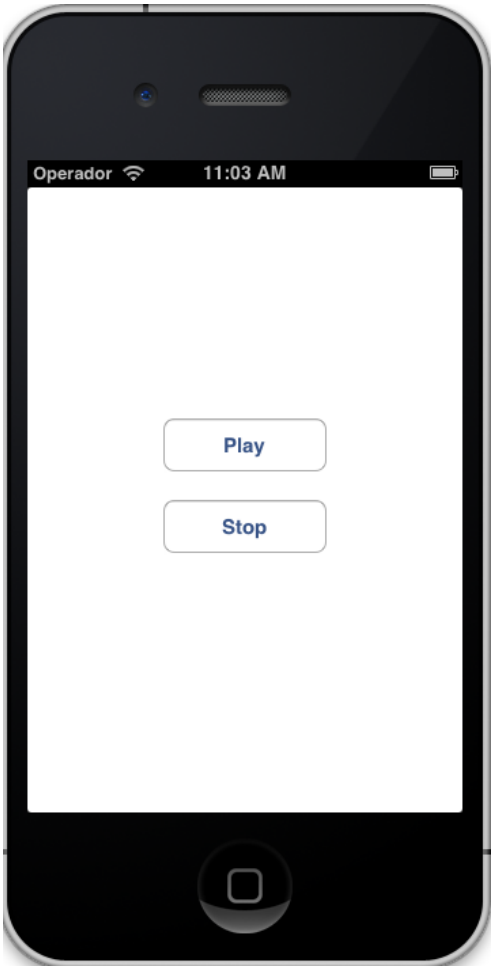

Figure 117: Our simple music player

Summary

In this chapter, we discussed two ways to play audio from an iOS device: system sounds and AVAudioPlayer. The former should be used only for short, simple sounds that don't need much custom configuration, and the latter gives you more control over the audio playback.

For the average iOS app, the AVFoundation Framework provides a good balance between a user-friendly API and fine-grained control over your sounds. But, keep in mind that iOS also provides more advanced, low-level audio capabilities like Audio Queue Services, as well as a higher-level API for interacting with the user's existing iTunes library via the Media Player Framework. For specialized applications, you can even drop down into OpenAL, which is a cross-platform library for simulating a 3-D audio environment.

A lot of work has gone into making iOS a multimedia-friendly platform, so there's no shortage of options for managing images, sounds, and videos from within your application.

Conclusion

This book covered the basics of iOS app development. We started by building a simple user interface, which introduced us to the fundamental design patterns of iOS: model-view-controller, delegate objects, and target-action. Then we dove into multi-scene applications and learned how iOS lets scenes communicate with each other and automatically handles transitions from one scene to another. After that, we discussed the iOS file system, application bundles, and required resources like app icons, launch images, and the information property list. Bundles also happened to be the method for localizing applications to multiple regions in iOS, so we were able to serve translated images and text effortlessly to different language speakers. Finally, we took a brief look at the audio capabilities by creating a simple UI sound effect and music player.

While there are still dozens of frameworks left for you to explore on your own, I hope you feel comfortable navigating an Xcode project. Along with some hands-on experience, what you should take away from this book is a high-level understanding of the iOS architecture—how interfaces are created and laid out using UIKit and storyboards, and what classes and libraries to use for managing media assets. Armed with this knowledge, you should be more than ready to venture into the real world of iOS app development.