

Learn to build powerful iPhone and iPad apps
using Xcode 6 and Swift

Includes
Xcode 6



Beginning Xcode

SWIFT EDITION

Matthew Knott

Apress®

www.allitebooks.com

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.



Apress®

Contents at a Glance

About the Author	xv
About the Technical Reviewer	xvii
Acknowledgments	xix
Introduction	xxi
■ Part 1: Getting Acquainted	1
■ Chapter 1: Welcome to Xcode.....	3
■ Chapter 2: Diving Right In	17
■ Chapter 3: Project Templates and Getting Around.....	43
■ Chapter 4: Building Interfaces.....	75
■ Chapter 5: Getting Help and Code Completion	119
■ Chapter 6: Constraints.....	143
■ Part 2: Diving Deeper.....	179
■ Chapter 7: Storyboards.....	181
■ Chapter 8: Table and Collection Views	225
■ Chapter 9: Frameworks, Libraries, and Targets.....	287

■ Chapter 10: Advanced Editing	315
■ Chapter 11: Debugging and Analysis	343
■ Part 3: Final Preparations and Releasing	377
■ Chapter 12: Version Control with Git	379
■ Chapter 13: Localization.....	413
■ Chapter 14: Devices and the Organizer	455
■ Chapter 15: Building, Sharing, and Distributing Applications	483
Index.....	525

Introduction

Welcome to *Beginning Xcode*, the book that aims to give you all the knowledge to start writing applications using what is probably the most powerful integrated development environment (IDE) ever. And that IDE is free.

As with many Apple products, Xcode has simplicity and ease of use in abundance, but don't be fooled: the shiny exterior masks a workhorse of a tool, incredibly powerful and with an extensive set of integrated tools for every eventuality. Xcode is the development environment that all other IDEs have a poster of pinned to their bedroom walls.

Xcode hasn't always been this shining Rock God of awesomeness; it used to be a sorry band of ragtag applications. When I first picked up Xcode 3 in 2007, I remember going through a multitude of different applications to perform varying tasks, such as the very basic Interface Builder, and finding out how to adapt my knowledge of C into Objective-C. Back then, what I really wanted was something that showed me how to get the most out of Xcode and could give me the understanding I needed to get going with the hundreds of app ideas I had in my head.

Fast-forward seven years, and both Xcode and I have come a long way. I feel as if I've gone from being a kid, bumping my leaky paddleboat aimlessly around a lake, to being a handsome sea captain at the prow of my vessel, gazing forth as I slice through choppy waves with grace and ease. Well, aside from the handsome part, the analogy is a good one. Xcode 6 is more complete and powerful than ever before: it's an integrated product that puts in your hands the same power used by the developers at Apple who write the apps found in iOS and Mac OS X.

I've always had a mixed experience with development books and have often been sick of building the same old apps again and again. In this book I've aimed to take you through almost every facet of Xcode, helping you to understand the capabilities of each of the key areas as you build a number of cool and exciting projects along the way and come to grips with the new Swift programming language. By the end of the book, you should be ready to turn the ideas in your head into reality—and I can't wait to see what that looks like.

Part

1

Getting Acquainted

Welcome to Xcode

Apple provides Xcode to developers to help them create applications for Macs, iPhones, and iPads (OS X and iOS). Xcode was used to create many of your favorite iOS and OS X applications. Arguably, without such a powerful, refined, integrated development environment (IDE), the thriving ecosystem that is the App Store would not exist as it does today.

What Is Xcode?

All developers, regardless of the platform for which they're developing, use an array of tools to take an application from an idea to something that is readily available to millions of people. Apple is no exception, and it provides a cultured, powerful, and polished set of development tools. These tools are brought together within one application called Xcode. Xcode provides everything you need to create, test, deploy, and distribute applications for iOS and OS X. With the release of iOS 8 and Xcode 6, Apple has radically overhauled its toolset and created powerful new technologies that aid developers in making the process of creating an application fun and rewarding; in addition to that, and for the first time since Xcode was released, it now supports a brand-new proprietary programming language called Swift.

The purpose of this book is to guide you in becoming familiar with Xcode 6, in the hope that you'll become more confident and embrace it to create amazing, innovative new applications for iOS and OS X. As in many other technical books, as you progress through each chapter, you build on your knowledge and systematically create a number of iOS applications.

Although Xcode was created primarily for developers working on iOS and OS X applications, it's also great if you work with other languages such as C, Java, and C++, among others. Xcode has a long, interesting history of releases, some having a very good reception and some less so. First released in 2003, Xcode has had six major releases and seen a couple of major interface overhauls. After more than 10 years of active development, it's safe to say that Xcode is incredibly powerful and a leading professional set of development tools. What's more, Xcode is available to developers at absolutely no cost; all you need is an iTunes account, and you're good to go.

What Is Swift?

Before WWDC 2014, Apple's World Wide Developer conference, if you created a project in Xcode, the code that was added behind the scenes was written in a programming language called Objective-C. At WWDC14, Apple unveiled a brand-new and highly streamlined programming language called Swift. This new language greatly reduced the amount of code needed to achieve equivalent outcomes in Objective-C. Swift is far less rigid than Objective-C and is very similar in structure to Python, a move that will no doubt encourage more developers into what is already one of the most accessible development ecosystems in existence today.

To make Swift even more appealing, Apple also introduced a new piece of software called Swift Playground with Xcode 6. Playground gives you the ability to experiment with snippets of code to see the result without putting them into your application, meaning you have a code sandbox in which to try things such as loops or regular expressions and instantly see the result.

The purpose of this book is to teach you how to create apps using Swift in Xcode 6—it isn't a definitive Swift language guide. Apple, however, has released a free book that *is* a definitive guide to the Swift programming language and has made it available via iBooks. Search iBooks for Swift programming, or visit <https://itunes.apple.com/us/book/swift-programming-language/id881256329?mt=11>.

Why Choose Xcode?

If you have experience developing for other platforms, then you probably want to know what makes Xcode so great. The main thing is that there's simply no other IDE like it. It's unique in the sense that Apple has created it to be simple, yet at the same time it masks a powerful interior. You have the ability to work with a range of technologies, and you also have a phenomenal developer toolkit at your disposal. Xcode contains everything you could need: an intuitive code editor, advanced debugging, seamless interface-editing features, and the benefit of being constantly updated and maintained by Apple.

In addition, using Xcode is arguably the only practical way to develop applications for iOS and OS X that can truly be called native. Xcode is what Apple itself uses to produce its own innovative software, which is used by millions of people.

Aside from Xcode, it's hard to find a commendable alternative if you'd like to develop native iOS or OS X applications. Of course, there are third-party services and tools; but when using these you may often find yourself battling inconsistencies and a lack of compatibility rather than focusing on what's really important: creating great apps (and enjoying doing so). The purpose of Xcode isn't to simply be an IDE: it also helps and guides you on your quest to create something that has the potential to reach a staggeringly large audience. For that reason, Xcode is a fantastic choice.

Prior Assumptions

Before you dive in and start reading this book, it's assumed that you have at least some familiarity with developing for Cocoa Touch and are familiar with the concepts of object-oriented programming. This book is geared toward those developing for iOS; however, it's possible to get a lot out of this book if you're developing OS X applications, because many of the principles presented can be applied to either platform.

It's assumed that you are using a Mac and are preferably running the latest version of OS X. Unlike the Objective-C based equivalent of this title, it's absolutely necessary that you run the latest version of Xcode. This book is written specifically for Xcode 6 and Swift, and the technology doesn't exist in previous versions of the IDE. There is a common misconception that you need the greatest and latest "souped-up" Mac, but many previous-generation iMacs, MacBooks, Mac Minis and Mac Pros will work just fine.

It's also assumed that you know how to operate your Mac and how to use OS X. For example, you need to know how to use the Finder, save files, and so forth—all the basics. Finally, a couple of the chapters present scenarios in which an active Internet connection is required, and some features of Xcode perform better when you're connected. Additionally, some later chapters require a physical device and a paid developer account to complete, but for the most part you can use an iOS simulator to run your apps.

It's also worth mentioning that the purpose of this book is not to teach you how to create applications for iOS or teach you how to program in Swift or Objective-C; the purpose of this book is to get you up and running with Xcode so you can apply your current knowledge of Swift and OS X/iOS development and use the latest version of Xcode to its full potential to enable you to work more productively and create fantastic applications.

A final note, this book was written at a time of transition, when OS X 10.10 Yosemite was not released. All screenshots are taken using OS X 10.9. Despite being the same version of Xcode, there are differences in icons between the versions, and when this happens I call it out.

What's Covered in This Book

Part 1: Getting Acquainted

- *Chapter 1:* This chapter starts you on your journey into the world of Xcode and explains how to get Xcode onto your machine and prepare it for first use. You are shown how to sign up as an Apple developer, and you get a look at the wealth of resources provided by Apple to iOS and OS X developers.
- *Chapter 2:* Here, you start a project and get the ball rolling in terms of becoming familiar with Xcode. You learn the basics of how to create projects and build applications, along with how to get around in Xcode.
- *Chapter 3:* Next, the focus shifts to how to choose from Xcode's different project templates. You also get a guided tour around Xcode's interface along with an introduction to many of the menus, inspectors, and panels you should use to work efficiently.
- *Chapter 4:* This chapter focuses solely on how to design your interfaces using Xcode's built-in interface editor, Interface Builder. It gives you an in-depth look at the libraries and inspectors available.

- *Chapter 5:* Next, you're shown how to access the invaluable help resources that are built right in to Xcode and also how to make the most of its intelligent code-completion feature.
- *Chapter 6:* Building on Chapter 4, you see the Auto Layout system and learn how it works with constraints and the new size classes introduced in Xcode 6 to create a single layout for any device.

Part 2: Diving Deeper

- *Chapter 7:* This chapter shows you how to use a key feature for rapid development in Xcode: Storyboards. You see how it can add a certain degree of logic to how you display and push views in your application.
- *Chapter 8:* This chapter explains how Xcode makes it easy to populate and create table and collection views, with the addition of how to customize their appearance and functionality.
- *Chapter 9:* Here you learn how to add features to your application by adding frameworks and libraries. You also learn how to create a different version of your application in the same project with targets.
- *Chapter 10:* This chapter shows you how to add your own personal touches to Xcode in terms of editing code. In particular, the code editor is the focus of this chapter, and you see how to work more productively and how to customize its appearance and behavior to suit your tastes and requirements.
- *Chapter 11:* This chapter presents the idea of making your application run more efficiently and faster. This is done by looking at the range of different tools and methods included in Xcode: for example, using breakpoints to step through your code systematically. You also learn about the Swift Playground for prototyping and testing your code.

Part 3: Final Preparations and Releasing

- *Chapter 12:* Here you learn how you can protect your code and work effectively as a team by using Git, Xcode's integrated version control software.
- *Chapter 13:* This chapter examines the idea of localization and how to use Xcode to accurately support multiple languages in your app.
- *Chapter 14:* This chapter looks at the Organizer, what it's for, how to navigate around in it, and how to keep your developer assets in good standing order.
- *Chapter 15:* To conclude, you make final touches to your application, build it for release, and then share it either as an IPA file or via the App Store using either Application Loader or the Organizer.

Getting and Installing Xcode

Before you can download Xcode, there are a couple of things you need to do. You need an iTunes account (or an Apple ID) that allows you to download content from the Mac App Store; then you're good to go. If you don't have an Apple ID, you can sign up for one at no cost at <http://appleid.apple.com>. This book is written for Xcode 6, and to run it you also need a Mac that's running the latest version of OS X or at least OS X 10.9.3.

Once you're equipped with an Apple ID and a Mac running OS X 10.9.3+, you can begin downloading Xcode. As with many other Mac apps, you simply download it from the Mac App Store at no additional cost. Open the App Store on your Mac, select Categories from the top bar of the window, and then click the Developer Tools category. Usually, you can find Xcode right away either at the top of the window or in the sidebar on the right displaying the top free apps. Alternatively, you can use the Search bar at top right and enter "xcode". Xcode's icon is a hammer over an "A" blueprint, as shown in Figure 1-1.



Figure 1-1. Xcode in the Mac App Store

Note If you don't have access to the latest version of OS X or are running an older version that isn't supported, you can download previous versions of Xcode from the iOS Dev Center, but for this you need to have a registered Apple developer account. This is explained later. However, this book covers the latest version of Xcode (which is 6.0 at the time of writing).

Select the icon, and you're taken to Xcode's App Store page. Here you can view all the features of Xcode along with the latest additions to the current version of Xcode (at the time of writing, this is 6.0) and also preview some screenshots of Xcode. To download Xcode, click the gray Free button and enter your Apple ID e-mail address and password, and your download will commence. Xcode is about 2.4 GB, so you can go and make some coffee while you wait for the download to finish, as shown in Figure 1-2.



Figure 1-2. Xcode in the Mac App Store, ready to be downloaded

With Xcode downloaded, open it from your Applications folder. You're prompted to install some additional packages: click Install, and enter your user password. This installation should take a matter of seconds, as shown in Figure 1-3.

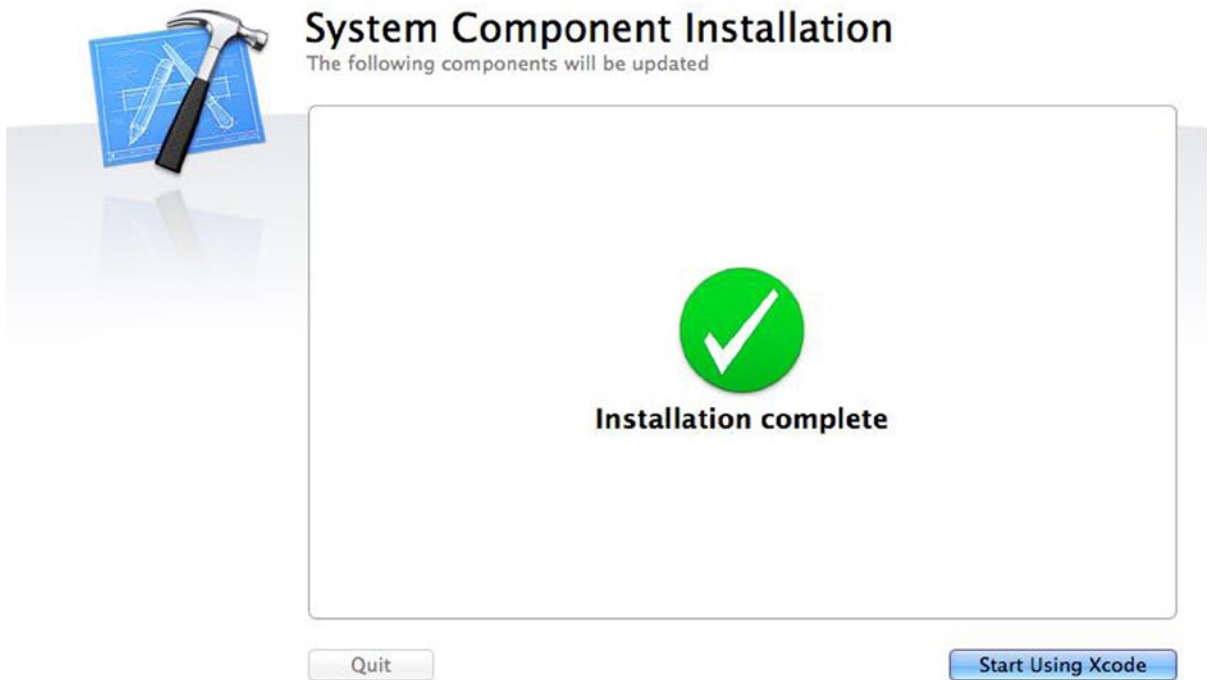


Figure 1-3. Installing additional tools required by Xcode

Firing Up Xcode

Once you've successfully downloaded and installed the additional components, you can begin using Xcode. In Chapter 2, you create your first project and become familiar with the basic areas of Xcode; but for now, just make sure everything is in good order so you don't encounter any problems later.

When you first launch Xcode, you're presented with a Welcome splash screen. From here you can create a new project, connect to an external repository, open documentation, visit Apple's developer portal, and also browse a list of your recent projects. For some, this screen causes irritation—you can prevent it from appearing each time you open Xcode by simply checking or unchecking the Show This Window When Xcode Launches box, as shown in Figure 1-4.



Figure 1-4. Xcode's Welcome window, which is displayed optionally each time you open Xcode

To create a new project, you can click the Create A New Xcode Project button on the Welcome screen or navigate to File ► New ► Project, where you're presented with a range of templates provided by default by Xcode.

If you have gotten to this point, it's safe to assume that you've successfully installed Xcode and that you're ready to start creating projects. However, let's save this for a deeper explanation in Chapter 2 and for now look at the variety of resources provided to developers by Apple.

Apple's Resources for Developers

At this point, you have Xcode downloaded to your machine, and you've fired it up to make sure it runs. If there's one thing that makes Apple stand out from its competitors, it's the wealth of knowledge, resources, and tools that are made just for developers. There are thousands of documents, thousands of samples to download, and dozens upon dozens of videos you can watch. Currently you have Xcode installed, but that alone isn't going to make you a great developer of iOS and OS X applications. You also need to use the vast library provided by Apple. To gain access to Apple's resources, I urge you to sign up as a registered Apple developer. To do this, all you need is an Apple ID; you can create a new one or use the same ID you use to download content from iTunes or the App Store.

First, head over to <http://developer.apple.com>. This is the central web site for Apple developers. On the home page of the site, click iOS Dev Center. The iOS Dev Center is the central location for all the resources provided to those who create iPhone, iPad, and iPod Touch applications, as shown in Figure 1-5.

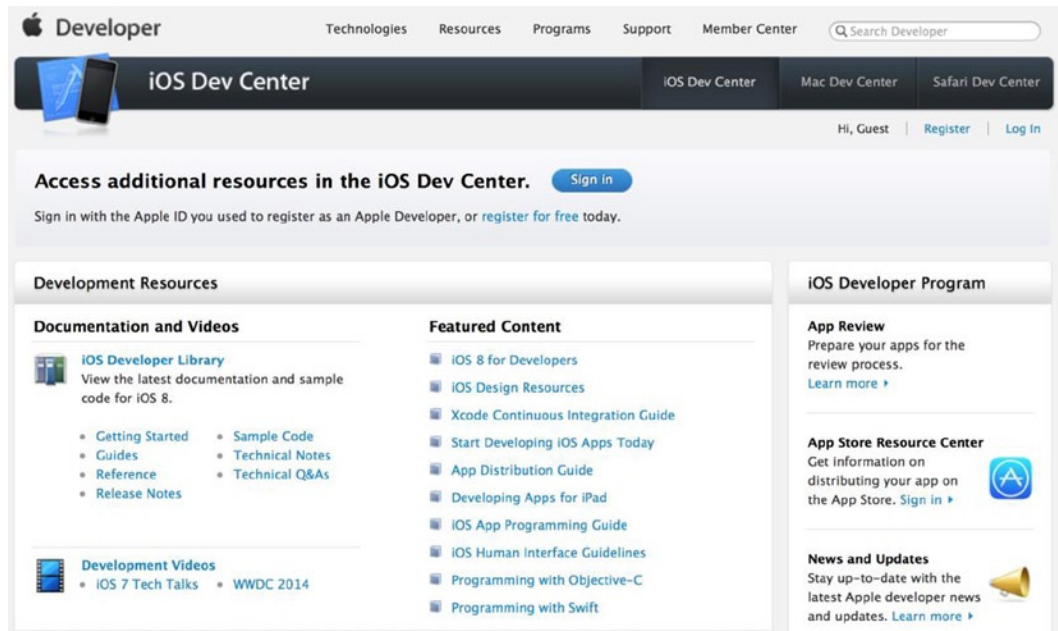


Figure 1-5. The iOS Dev Center—home of Apple resources for iOS developers

You aren't required to sign up in order to gain access to many of the resources, including the Developer Library, an overwhelming wealth of example source code, release notes, and many more things. You can happily browse through the iOS Dev Center right now.

However, there are great advantages to signing up as a registered Apple developer, and it will become essential at some point if you're planning to distribute applications via the App Store. Therefore, it's a good idea to sign up right from the start. To begin the process of signing up, click the Register For Free text just below the Sign In button; alternatively, you can visit <http://developer.apple.com/programs/register/>. In order to sign up, you need a valid Apple ID; if you don't have one or would like to dedicate an Apple ID to your developer account, create a new one (don't worry, none of your purchases or downloaded content from the App Store or iTunes Store will be affected if you use your current one).

Once you're happy with your Apple ID, go to <http://developer.apple.com/programs/register/> and sign up for an account. In order to complete the process of signing up, you need to create a personal and professional profile; you can change these at any time if you need to.

You're required to complete your professional profile by telling Apple any previous platforms you've developed for along with your primary markets and experience with Apple's platforms. This information is used by Apple to get an idea of the spectrum of people who are signing up as developers. Again, once you have completed this, click the Next button. Also, it's important to note that what you select when updating your professional profile doesn't bind you to anything, and that you're able to develop and release applications to any of the App Store's markets. Furthermore, you can, if needed, make any amendments to your professional profile (and personal profile, for that matter) after you've signed up, as shown in Figure 1-6.

Apple Developer Registration

Apple ID Personal Profile Professional Profile Legal Agreement Email verification

Complete your professional profile

(All form fields are required)

Which Apple platforms do you develop with? Select all that apply.

iOS

Mac OS X

Safari

What is your primary market?

Business Medical Reference

Education Music Social Networking

Entertainment Navigation Sports

Finance News Travel

Games Photography Utilities

Health & Fitness Productivity Weather

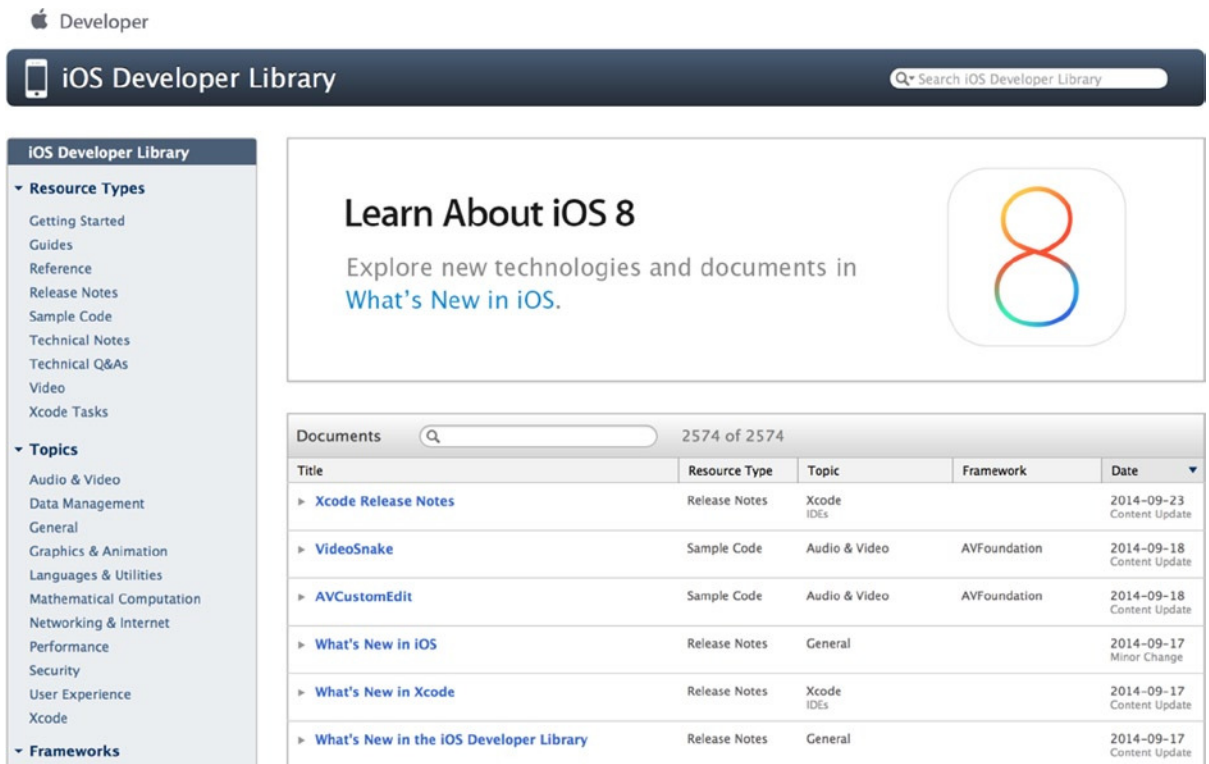
Check this box if you are currently enrolled in a college or university.

Figure 1-6. Completing your developer professional profile

Finally, you reach the tiresome agreement that comes with many of Apple's products; read it, click to agree, and then continue with the process. To finish, all you need to do is verify that the e-mail address supplied is valid; you do this by opening the e-mail sent to you by Apple and entering the verification code contained within.

The Dev Center

As mentioned previously, Apple really does like to take care of its developers. As a developer, your first port of call is the Developer Library, because it houses most of the resources provided by Apple. If you select the iOS Developer Library link under Documentation And Videos, you're taken to an invaluable section of Apple's developer web site. The Developer Library is a simple and straightforward site: simply use the links on the left to navigate around and to filter the results. You can either search for specific keywords or sort the results using one of the column titles, as shown in Figure 1-7.



Apple Developer

iOS Developer Library

Search iOS Developer Library

iOS Developer Library

Resource Types

- Getting Started
- Guides
- Reference
- Release Notes
- Sample Code
- Technical Notes
- Technical Q&As
- Video
- Xcode Tasks

Topics

- Audio & Video
- Data Management
- General
- Graphics & Animation
- Languages & Utilities
- Mathematical Computation
- Networking & Internet
- Performance
- Security
- User Experience
- Xcode

Frameworks

Learn About iOS 8

Explore new technologies and documents in [What's New in iOS](#).

Documents 2574 of 2574

Title	Resource Type	Topic	Framework	Date
▶ Xcode Release Notes	Release Notes	Xcode IDEs		2014-09-23 Content Update
▶ VideoSnake	Sample Code	Audio & Video	AVFoundation	2014-09-18 Content Update
▶ AVCustomEdit	Sample Code	Audio & Video	AVFoundation	2014-09-18 Content Update
▶ What's New in iOS	Release Notes	General		2014-09-17 Minor Change
▶ What's New in Xcode	Release Notes	Xcode IDEs		2014-09-17 Content Update
▶ What's New in the iOS Developer Library	Release Notes	General		2014-09-17 Content Update

Figure 1-7. The iOS Developer Library

In addition to the iOS Developer Library, you also have access to an array of getting-started videos that explain core Objective-C and Cocoa Touch concepts. You're also given access to a direct link to the latest version of Xcode on the Mac App Store and the ability to download previous versions of Xcode if you're not running the latest version of OS X or would like to target older versions of iOS.

Your Developer Account

Currently, your level of membership is that of a free account, meaning you have access to a staggeringly vast amount of resources but not to all the resources you need if you're planning to release applications to the App Store. Although this isn't necessary at this point, it's a good idea to sign up as a paid developer, because doing so gives you access to the Apple developer forums, prerelease versions of iOS before they're available to the public, prerelease versions of Xcode, the ability to test your applications on your iOS devices, and, of course, the ability to submit applications to the iOS App Store. The cost of signing up at the time of the writing of this book is \$99 per year, and it's required for some of the concepts presented toward the end of the book.

As mentioned previously, it isn't necessary to sign up this instant, but it's recommended that you do so at some point. To sign up for a paid account, visit <https://developer.apple.com/programs/ios/> and click the Enroll Now button. You're then guided through the process of signing up; it's straightforward if you follow the steps onscreen, as shown in Figure 1-8.

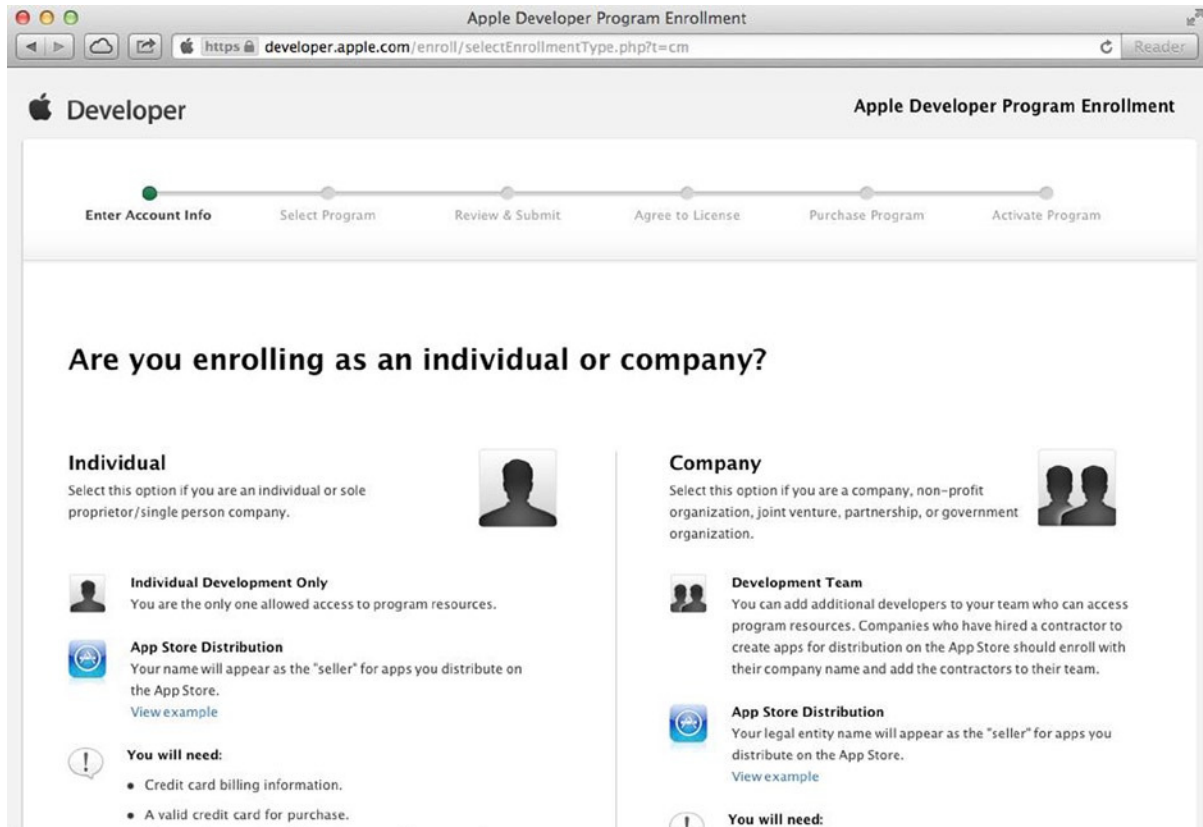


Figure 1-8. Choosing between an individual or a company developer account

It's useful to note that when you're prompted to choose between an individual or company account, if you're planning to operate under a name other than your own, you have to register as an official company (this is verified by Apple) and then acquire what's called a Data Universal Numbering System (DUNS) number that uniquely identifies your company; this takes around 7 days to process, so plan ahead. If selling applications under your own name suffices, then go for the simpler option of signing up as an individual; both accounts are essentially equal in terms of the resources you're able to access. This choice mainly determines the name with which you operate under on the App Store.

Don't worry if you're not ready to do this right now—it's covered in detail later in the book when it becomes essential if you're planning to release applications on the App Store (free or paid) or want to test your apps on an iOS device. Chapter 14 looks at using Provisioning Profiles and deployment onto actual iOS devices as opposed to the virtual iOS Simulator, so you then need access to a paid developer account.

Source Code

I strongly recommend that one of the first things you do is to go to the Apress web site for this book and download the entire source code. Either search for the book at www.apress.com or go directly to www.apress.com/9781430250043. When you get to the page for this book, scroll down until you see the section of the web site with four tabs, the third of which is Source Code/Downloads, as shown in Figure 1-9.

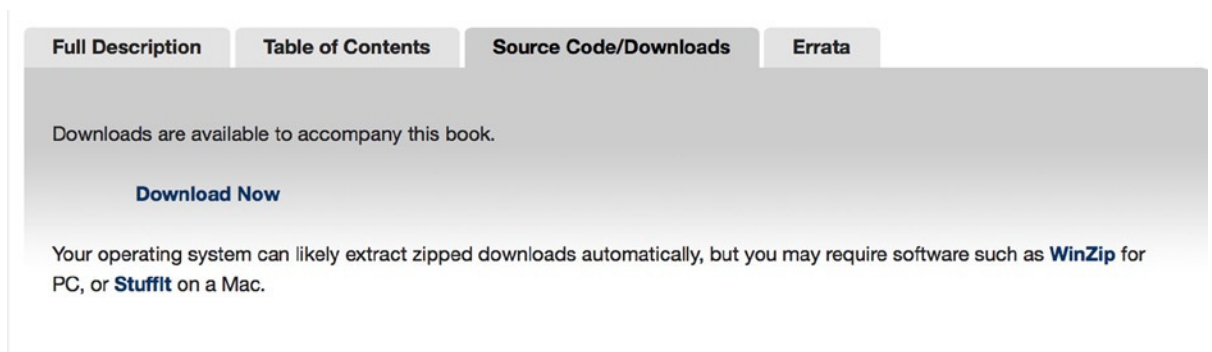


Figure 1-9. The source code download page for this book

Additional Resources

In addition to Apple's own resources, an extensive amount of third-party resources are available if you ever have a burning question or get stuck somewhere:

Forums: Forums are a great way to ask questions, learn from other people's questions, and help other people. In particular, Stack Overflow (<http://stackoverflow.com/>) has been invaluable to the entire developer spectrum for years and has a vibrant, active collection of iOS developers. There are also the Apple developer forums, which are available to those with a paid Apple developer account.

Mailing lists: There's a handy Xcode users mailing list that I'd recommend you subscribe to and periodically check. Many other developers, including myself, participate in answering questions relating to Xcode. You can subscribe at <https://lists.apple.com/mailman/listinfo/xcode-users>.

Xcode Overview: Apple provides a handy user's guide that's always being updated to accompany the latest release of Xcode, so it's a good idea to refer to it when there's a new update or if you'd like to follow up on something. It's available at https://developer.apple.com/library/ios/documentation/ToolsLanguages/Conceptual/Xcode_Overview. Similarly, it's also handy to glance over the latest release notes when Xcode is updated. These are available at <https://developer.apple.com/library/ios/releasenotes/DeveloperTools/RN-Xcode>.

Search engines: It's easy to underestimate the power of a simple Google search (and it's apparent many people on online forums don't have access to them). It can save you a lot of time, because someone, somewhere, at some point has undoubtedly had the same question you do—all you need to do is find where they asked it!

Videos: If you type "Xcode" into iTunes U search, you'll find a couple of good university courses that focus not only on Xcode but also on iOS development in general. Similarly, type "Xcode" into YouTube search, and you'll be amazed at what you can learn from the short screencasts that have been uploaded.

Contact me: I am happy to field questions via email at matthewknott@me.com or via my blog at www.mattnott.com.

Summary

In this chapter, you have:

- Successfully downloaded and installed Xcode
- Had a look around the iOS Dev Center and also looked at the resources provided by Apple to aid developers
- Signed up and registered as an Apple developer and become aware of the option of signing up for a paid developer account

Chapter 2 explains how to create your first project and helps you become more familiar with Xcode's interface and basic concepts.

Diving Right In

In Chapter 1, you downloaded Xcode, made sure it was correctly configured, signed up for a developer account, and explored the wealth of resources provided by Apple to help you get started with not only Xcode but also some of its fantastic new technologies. This chapter explains how to create a working application using Xcode's visual interface building tool (aptly named Interface Builder) and its built-in code editor and then run the app on your machine.

As mentioned, as you progress through this book, the ultimate goal is not only to get a grip on the latest and greatest version of Xcode but also, by the end of the book, to have walked you through building a series of varied applications that give you many of the essential skills needed to go out and start writing your own applications. The application you build in this chapter familiarizes yourself with Xcode as a development environment before you start looking at sharing data between pieces of your application in Chapter 3. For now, you develop a very simple application that has a custom background color and a label, and you programmatically update the text in the label.

Be forewarned that in this chapter, a lot of the concepts are new and therefore require more explanation to do them justice. As a result, on several occasions you're told that later chapters revisit many of the concepts presented. This is because the main goal of this chapter isn't to turn you into an Xcode pro, but rather to get you started and give you the confidence to believe that Xcode isn't as overwhelming as it may first appear. In Figure 2-1, you get a glimpse of the example application; although simple, it will make you at least a little familiar with the workings of Xcode and help you to understand that Xcode can help you produce a working application in next to no time.

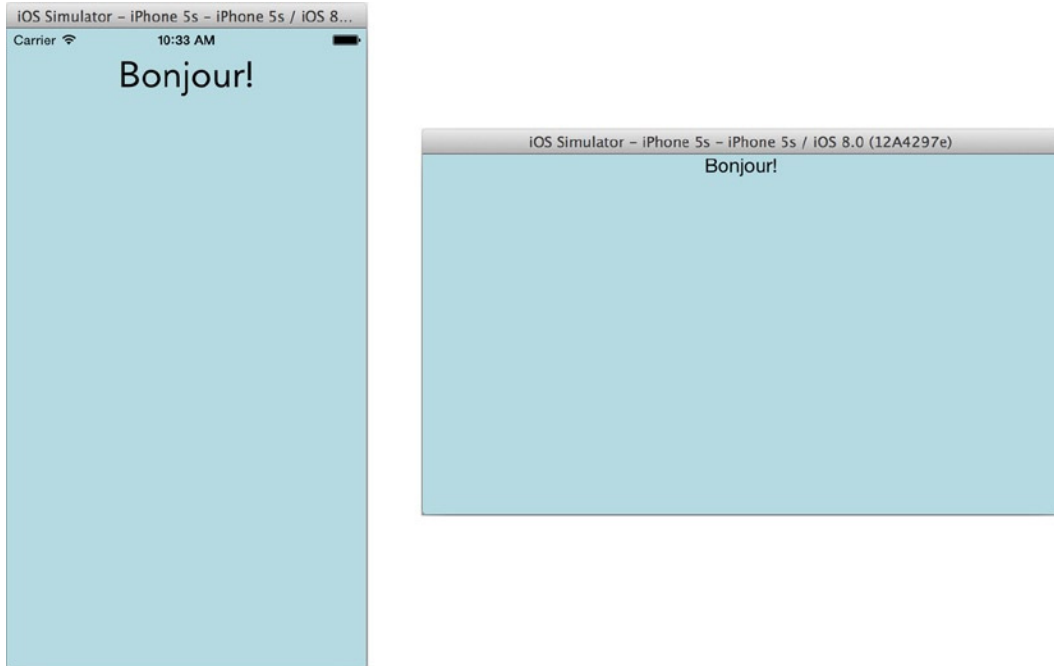


Figure 2-1. The application you create in this chapter

Creating Your First Xcode Project

First you need to bring this project into existence. To do this, start by running the Xcode application, and then click Create A New Xcode Project on the Welcome Screen or choose File ► New ► Project (⌘+Shift+N). You're presented with a new window asking what kind of project you'd like to create. Apple provides, by default, a variety of different project templates for both OS X and iOS, each of which is useful for getting started on different types of projects; Chapter 3 covers each of them in more detail. Continue as follows:

1. Because you're creating a basic one-view application, it seems appropriate to choose Single View Application, which can be found in the Application category under iOS on the left side of the dialog.
2. Once you've selected the Single View Application project template, click the Next arrow in the bottom-right corner. Figure 2-2 shows the template screen.

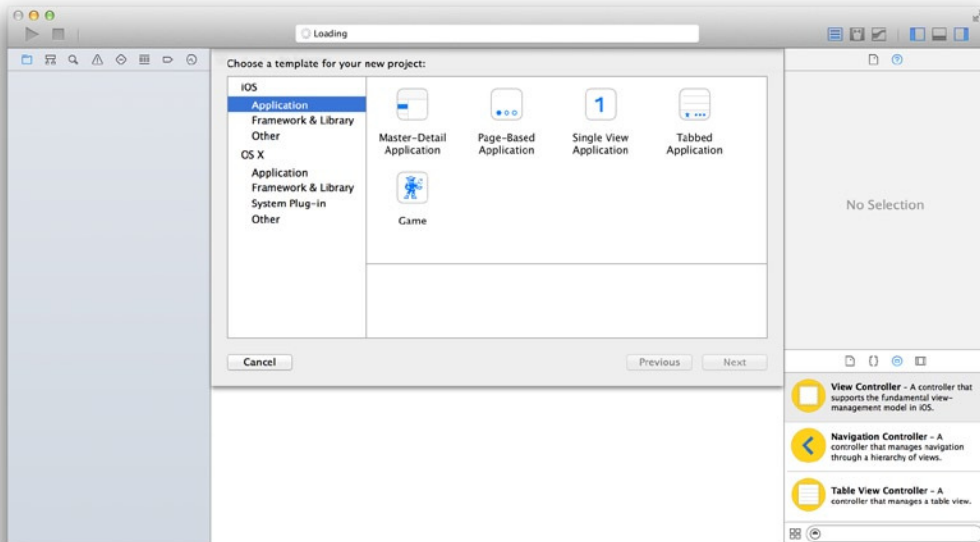


Figure 2-2. The variety of different templates provided by Apple to help you get started with creating your app quickly

You need to specify a couple of things before you can actually get started. Once you select your project template, a screen identical to that in Figure 2-3 is displayed. Following is a brief overview of each of the values required to proceed—bear in mind that you learn more about the significance of some of the values you enter here as the book progresses:

- **Product Name:** What you would like to call your application. For example, if you wanted to create an application called Chocolate Recipes, you'd specify the Product Name to be something along the line of Chocolate Recipes. Although not required, it's generally good practice to omit any spaces and instead capitalize each new word. The Product Name can be amended during the development of your application, so you're not obliged to stick with what you specify; but regardless, the Product Name is a rather important detail that you're required to specify at this stage.
- **Organization Name:** Required whether you're working independently or you're part of a software development company. For now, your own name is adequate. If you're looking to submit an application to the App Store, it's in your best interest to specify the correct name; although not required, it's recommended. When you create a new file, your Organization Name automatically appears along with copyright details at the top; that's something to bear in mind if you plan to work on a team or hand your project off to someone else.
- **Organization Identifier:** Only really required if you're planning to distribute your application in some capacity. For example, to distribute an app via the App Store, you're required to specify an App ID along with a Bundle Identifier, which is created by Xcode depending on what you input as your Organization Identifier. The company identifier is written in the style of reverse domain name notation; my web site, for example, is mattknott.com, so my Organization Identifier is `com.mattknott`.

- *Bundle Identifier*: By default, a combination of the Organization Identifier and the Product Name, to avoid confusion (I won't focus on this too much right now). You can't edit this.
- *Language*: Swift or Objective-C. For the first time, in Xcode 6, you get to choose between two possible languages. This book covers the Swift programming language, so please ensure that you select Swift for each example project.
- *Devices*: The device you'd like your application to run on. This is possibly the most straightforward part of getting up and running with your project. You have three choices: iPhone, iPad, and Universal. The iPhone and iPad choices are self-explanatory. A Universal application is one that is compatible with both the iPhone (and iPod Touch) and iPad. Your selection here isn't final, but it's good to make the right choice.
- *Use Core Data*: Core Data is a large framework designed by Apple to simplify and unify the methods for storing data in iOS. For example, if you wanted to create a database for storing relational information in your application, you might want to set up an SQLite database. Core Data does all this for you and gives you a simple interface to set up the tables, fields, and relationships.

Choose options for your new project:

Product Name: HelloWorld

Organization Name: Matthew Knott

Organization Identifier: com.mattknott

Bundle Identifier: com.mattknott.HelloWorld

Language: Swift

Devices: iPhone

Use Core Data

Cancel Previous Next

Figure 2-3. Specifying the project's details

Note If you've been using an older version of Xcode, you may have an annoying feeling that a field is missing. In version 6 of Xcode, Apple decided to remove the ability to specify a Class Prefix value. The Class Prefix was added to the start of every new class you created to help differentiate your classes from other class files that might be imported into the project.

3. Now that you vaguely know what these values are for and what they correspond to, you're probably wondering what you should input to create this project. As shown in Figure 2-3, type in **HelloWorld** as the Product Name; input your own first and last name as your Organization Name; use **com.LASTNAME** as your Organization Identifier (obviously change **LASTNAME** to your actual last name), set the Language value to Swift if it isn't already, specify iPhone as the Device, and, finally, ensure that Use Core Data isn't selected.
4. Once you've made sure all your values are correct, click Next. You're required to save your project to disk.
5. When prompted to, use the familiar OS X dialog to find a location. Make sure the box next to Source Control For This Project is unchecked, and then click Create.

Note Git is a popular system used for version control and source-code management. You can integrate a local Git repository with a web site such as GitHub or Bitbucket if you want to back up or share your code online. If none of these things are familiar to you, Chapter 12 explains.

So, you've given Xcode all the relevant details and specified what kind of project you're looking to create. As a result, Xcode conveniently creates a basic, functioning application for you to use as a starting point. The code that Xcode creates for you is just enough to get the application to run; it's a working, if slightly pointless, app that you can run right now if you like.

Choose Product ► Run (⌘+R), and you'll find the application builds successfully and the iOS Simulator pops up with the app running, as shown in Figure 2-4. It's nothing spectacular, nor will it reach the top 25 in the App Store anytime soon, but it's a functioning application created by Xcode with very little input from you. Return to Xcode, and click the Stop button in the top-left corner or choose Product ► Stop (⌘+.).



Figure 2-4. The initial application created by Xcode

Tip With the high resolution of modern iOS devices, unless you have a top-of-the-range Mac, the simulator may be too large for your computer screen. If this is the case, then with the simulator selected, choose **Window** ► **Scale** ► **50%** or use the key shortcut **⌘+3**. The **Scale** menu also gives you the option to go to **100%** or **75%** scale using **⌘+1** or **⌘+2**, respectively.

The Project

In order to make the app a little more interesting than a simple white screen, you need to open some files that Xcode created. As with previous versions of Xcode, the way in which it organizes your project's file is somewhat strange. Upon returning to Xcode, if you look to the left of the interface, you should see what appears to be an arrangement of folders and files. These are the files that make up your project (see Figure 2-5). This part of Xcode is called the Project Navigator. If you're unable to find it, choose View ► Navigators ► Show Project Navigator (⌘+1). It's important to note that when you create a folder in the Project Navigator, it doesn't correspond to the structure in which the files are saved in the Finder. The folders and organization of the Project Navigator are purely to help you locate files in Xcode. If you create a folder in the Navigator, the same folder isn't present in your project when you browse in Finder. I revisit this when you add a file to your project later in this chapter.

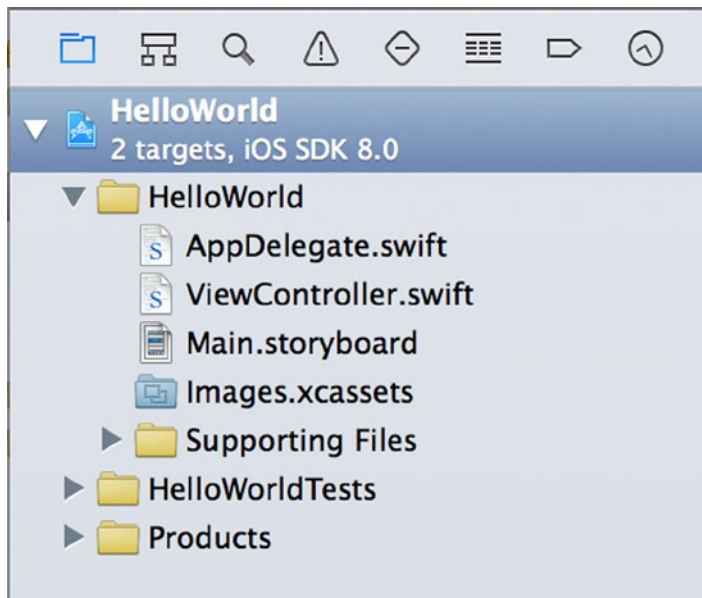


Figure 2-5. The Project Navigator

If you've used Xcode with Objective-C in the past, then you've almost certainly noted that when using the Swift language, there are far fewer files in the project. The reason is that in Objective-C, each class file is created with a header (.h) file and an implementation (.m) file. Swift combines all class information into a single file.

With that in mind, select `Main.storyboard` from the Project Navigator. Xcode opens its built-in graphical user interface (GUI) design tool, usually referred to as Interface Builder. Xcode 4.0 represented a major overhaul of Apple's developer tools: Interface Builder, which was previously a separate application, was conveniently integrated into Xcode, making it easy to switch between the built-in code editor and interface design tool in a single application, as shown in Figure 2-6. One warning is worth mentioning: the more you become familiar with Xcode, the more you may wish for a larger screen!

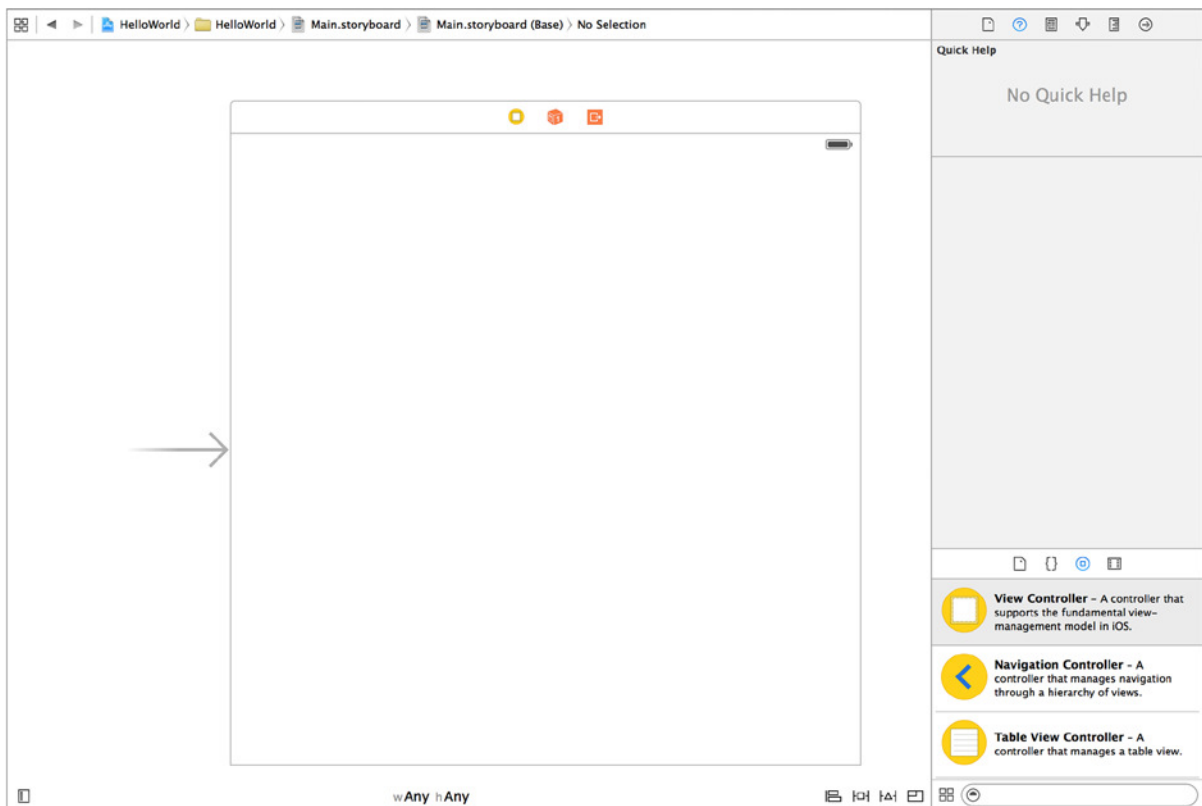


Figure 2-6. Xcode's built-in graphical interface designer

Designing the Interface

As noted previously, the app’s interface is neither exceptional nor revolutionary—in this chapter you create something simple and straightforward. However, the basics presented here are echoed later in the book in much more detail, and they’re the building blocks behind almost any app you could want to build. To begin with, let’s look at the Attributes Inspector. You can find this by selecting the fourth tab in the sidebar on the right side of Xcode’s interface; alternatively, you can choose View ► Utilities ► Show Attributes Inspector (⌘+⇧+4). The Attributes Inspector plays an important role when it comes to layout and fine-tuning interface elements. Now follow these steps:

1. To change the background color of the application, first make sure the view is selected by clicking the white area with an arrow pointing to it in Figure 2-6.
2. Under the View heading of the Attributes Inspector, select the color-picking option for the Background attribute. Then use OS X’s default color picker to choose a background color, as shown in Figure 2-7. In this example, I’ve used the RGB sliders and chosen a background of Red: 181, Green: 218, and Blue: 225, but you’re free to choose whichever colors you wish.

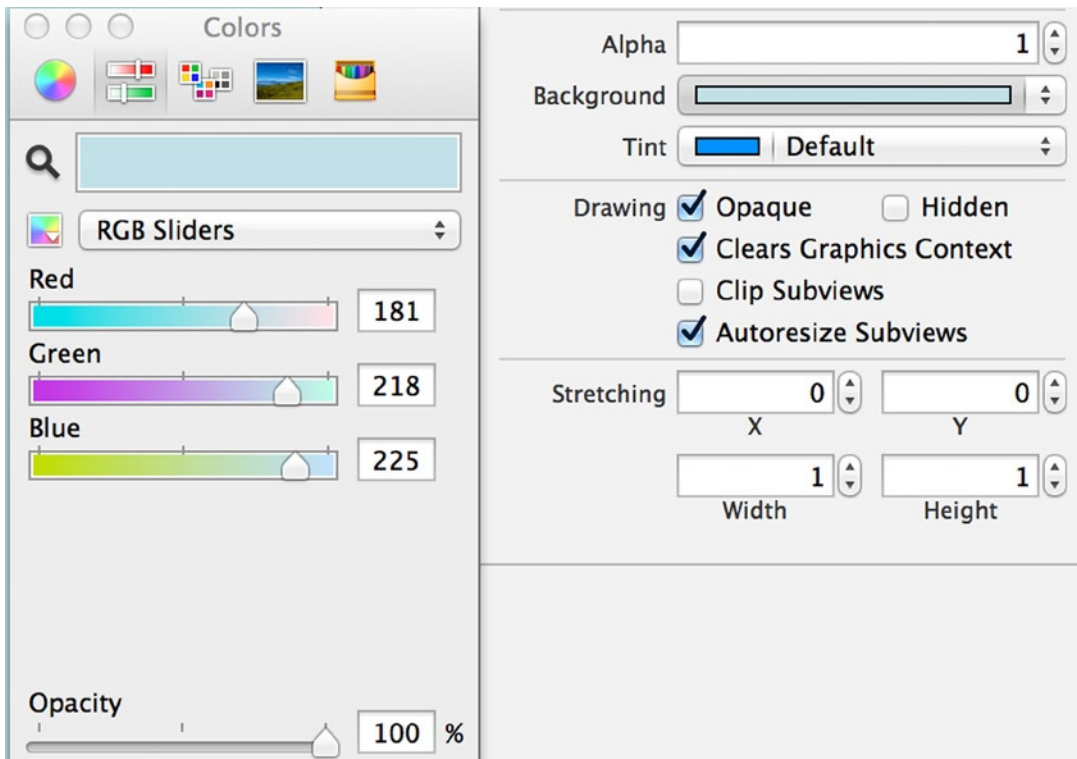


Figure 2-7. Changing the view’s background using the Attributes Inspector

3. Add a label to your view. To do this, open the Object Library (shown in Figure 2-8), and drag a label object to your view. Generally, the Object Library is right below the Attributes Inspector and accessible by selecting the third tab; you can also access it via View ► Utilities ► Show Object Library (^+⌘+⌘+3).

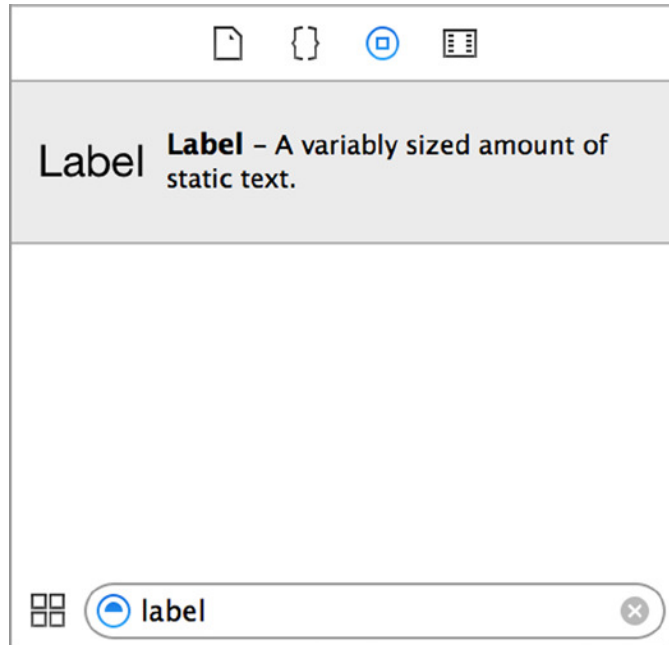


Figure 2-8. The Object Library, filtered for “label”

4. With the library open, use the small search bar to search for “label”.
5. Once you’ve found the label object, drag it to your interface at the top of the view, as shown in Figure 2-9. As you position the label near the top, the Guides shown in Figure 2-9 appear, and the label snaps into place.

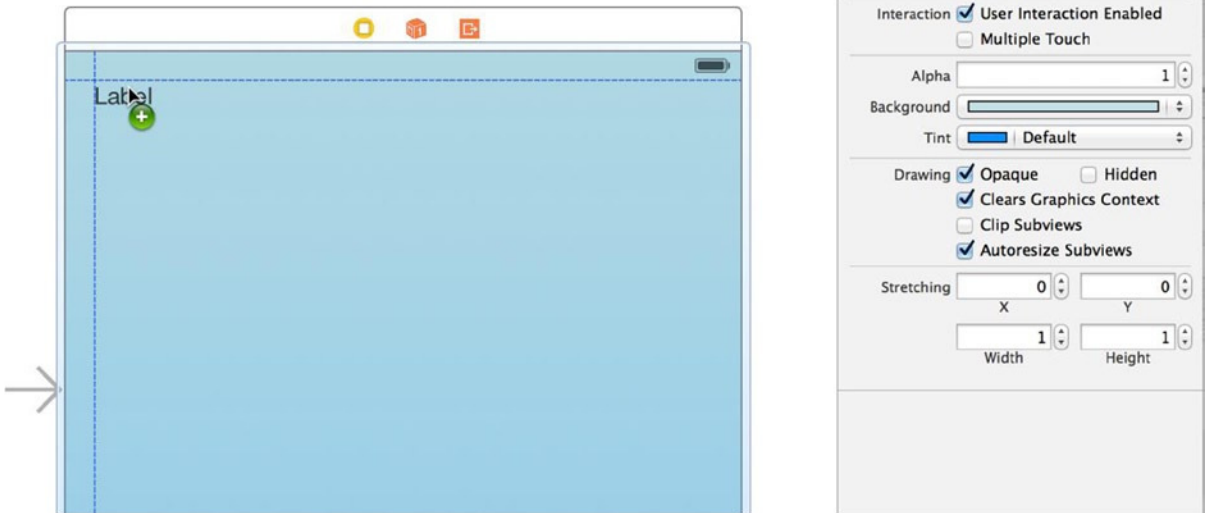


Figure 2-9. Dragging the label onto the view

Note Regardless of whether you've used Xcode before, you may wonder why you're only seeing what appears to be the top portion of the device's screen. This is because of a new Auto Layout feature that Apple introduced with Xcode 6 and iOS 8, called Size Classes. I explain in greater detail later in the book.

6. Select your new label, and use the handles to extend its size so it fills the width of the View. Then double its height to accommodate a larger font size.
7. Set the Alignment attribute in the Attributes Inspector to Center.
8. Click the T symbol in the Font attribute to alter the font. Set Font to Custom, Family to Avenir, and Size to 32, as shown in Figure 2-10.

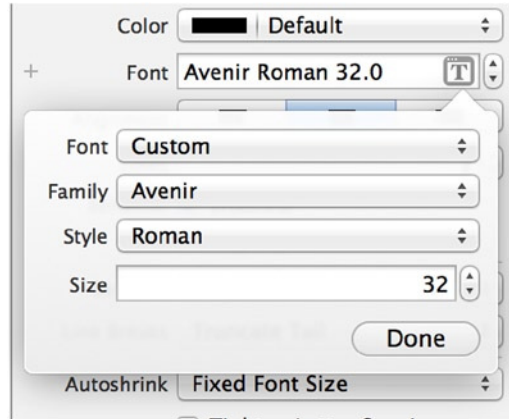


Figure 2-10. The Attributes Inspector's Font property

9. Because you have Size Classes enabled for this view, it's essential to set a couple of parameters called *constraints* that tell iOS how to position the label in the view. I cover this in detail later in the book, but for now, locate the pin icon in the bottom-right corner of the design area and click it. A popover appears.
10. At the top of the pin popover, you see a square with an I bar on each side followed by a numeric value. Click the top, left, and right I bars to highlight them red, as shown in Figure 2-11. Then click the Add 3 Constraints button.

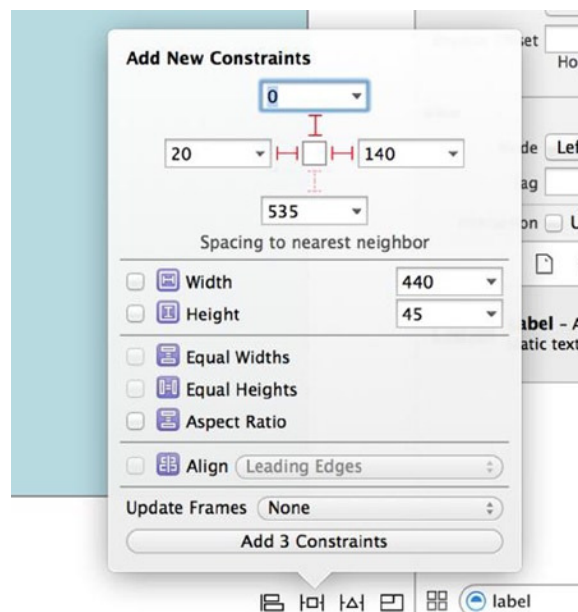


Figure 2-11. Setting some constraints for the label

At this stage, although you *can* use the Attributes Inspector to specify the text to be displayed in the label, it's important to realize that Xcode isn't just about creating graphical interfaces. It also houses a very powerful code editor. So, as you progress through this chapter, you update the contents of your label programmatically as opposed to graphically.

Making Connections

Before you leave Interface Builder and move on to focus on Xcode's code editor, let's look at a powerful feature that allows you to use both simultaneously. Open the Assistant Editor by selecting the shirt-and-bowtie icon in the top-right corner of Xcode, as shown in Figure 2-12, or by selecting View ► Assistant Editor ► Show Assistant Editor (⌘+⇧+Return).

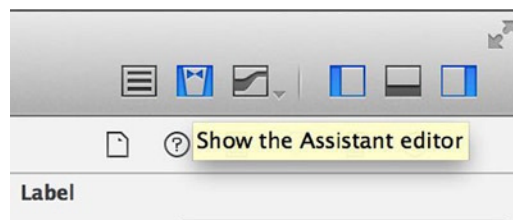


Figure 2-12. The button to select the Assistant Editor looks like a shirt with a bowtie

Opening the Assistant Editor splits your screen, with Interface Builder occupying the left frame and the code editor occupying the right (unless you've customized this appearance, as I show you later in the book). Before you continue, you need to make sure Xcode has opened the correct file. You should be looking at a file called `ViewController.swift`: you can verify this by looking at the jump bar just above the code, as shown in Figure 2-13. Continue as follows:

1. With both Interface Builder and the code displayed using the Assistant Editor, click the label you added to your view in Interface Builder to highlight it.



Figure 2-13. The jump bar in the Assistant Editor shows which file is open

Now you're going to create a variable called an outlet (IBOutlet) to make the label accessible through your code. In older versions of Xcode, the process of creating an outlet and then wiring it into Interface Builder was quite long-winded, but Apple has simplified this greatly over the past few versions of Xcode by allowing you to drag connections directly from Interface Builder into the code.

2. Holding down the Control key, click the label and drag a connection to the `ViewController.swift` file. Position the cursor in the class scope, just below the line `class ViewController: UIViewController {`, as shown in Figure 2-14.

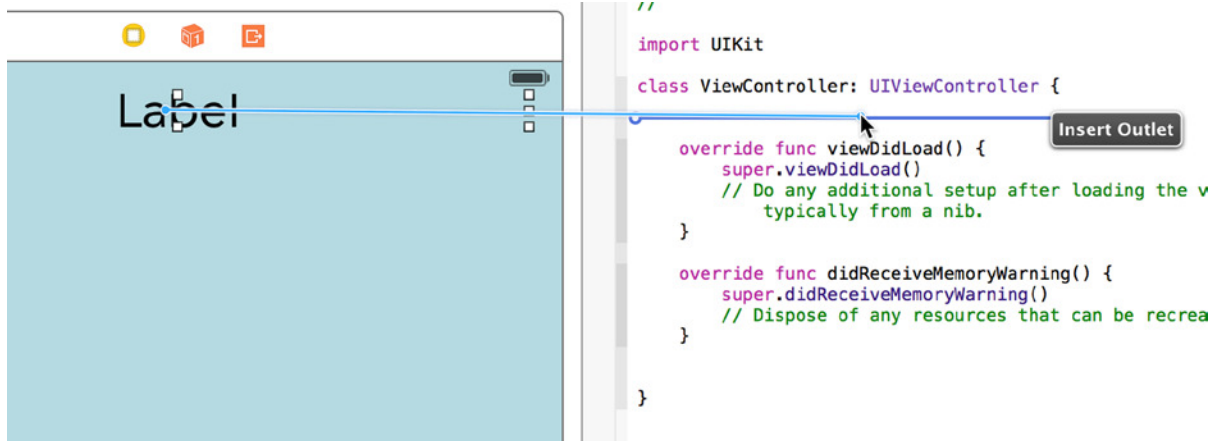


Figure 2-14. Creating an outlet using the Assistant Editor

When you release the mouse button, a Connection dialog appears, asking for a number of values (see Figure 2-15). The key option you need to be aware of here is the Name text field. If the object you're connecting to code can be tapped or trigger an event, you can choose one of two options for your connection: Outlet or Action. But in this instance Xcode intelligently knows that this label isn't interactive and therefore restricts your choices.

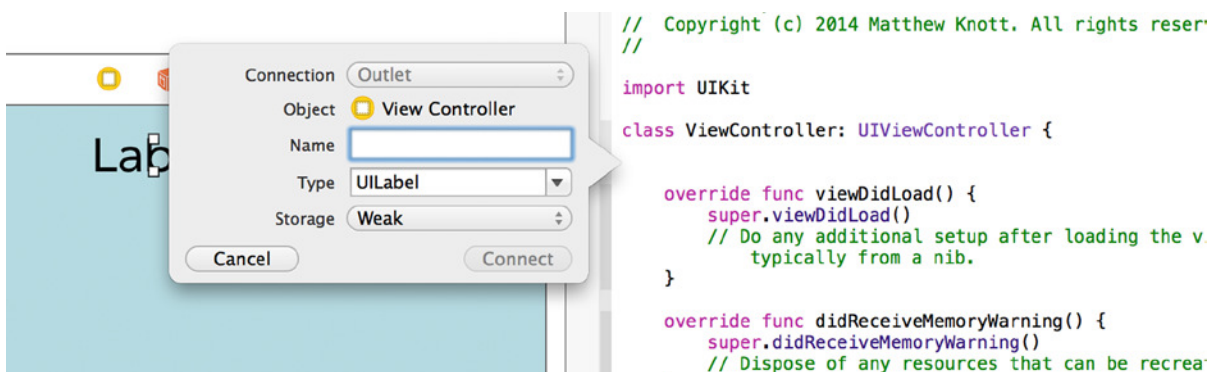


Figure 2-15. Creating an outlet for your label

3. The Name text field value determines how you refer to your label in code. For now, type in `lblOutput` and click Connect.

If everything's gone according to plan, the first few lines of code should look like this:

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var lblOutput: UILabel!
```

4. You're finished with the Assistant Editor for now, so switch back to the Standard Editor by selecting the icon from the toolbar with three lines in a box, to the left of the Assistant Editor icon (see Figure 2-12).

You've finished with Interface Builder for this project. Now you need to write some code to manipulate your label. Go to the Project Navigator and select `ViewController.swift`. The View Controller's class file opens in the code editor, as shown in Figure 2-16. This chapter touches on many areas and concepts I explain throughout the book; but at this point, you're going to start using Xcode's powerful code editor and see some of the intuitive features that make Xcode one of the best IDEs ever.

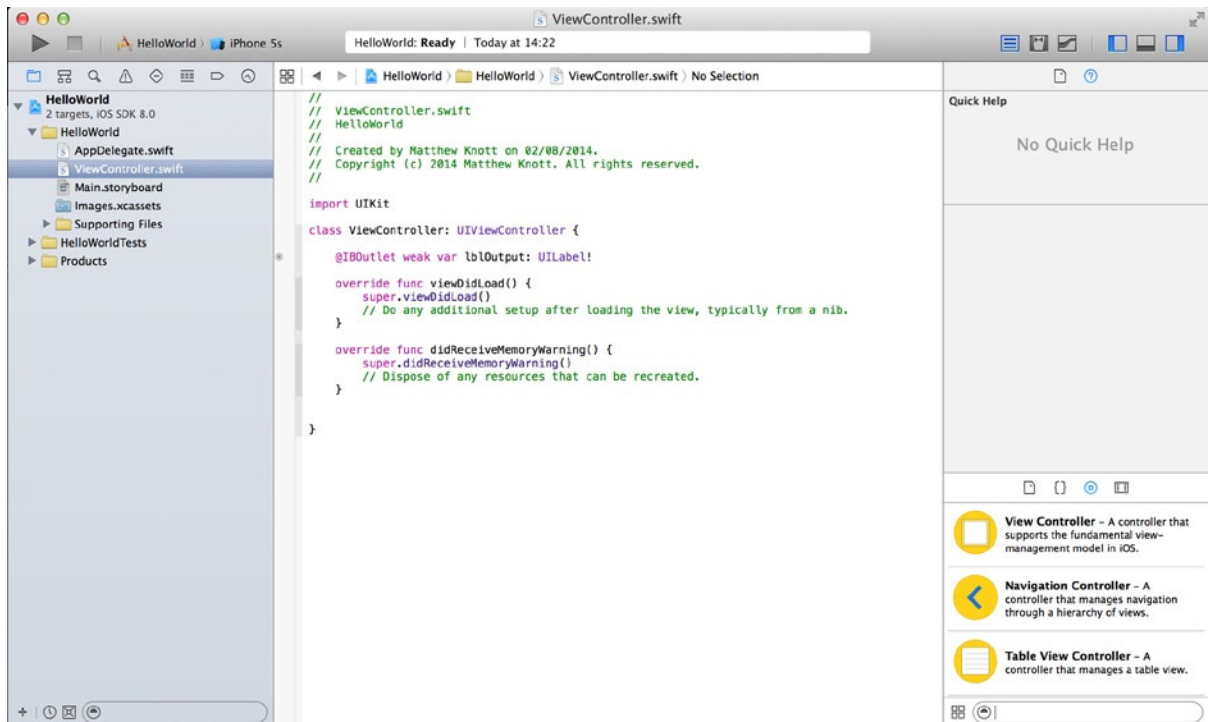


Figure 2-16. Xcode's built-in code editor with the `ViewController.swift` file open

With the class file open, notice that it has a number of lines of code by default. This boilerplate code gives the application a starting point you can build on.

In the code of the class file, look for a line that begins with `override func viewDidLoad()`: this is the start of the `viewDidLoad` function. To complete the very simple code for this application, you need to tell the View Controller that when the view loads, it should set the label's text to "Bonjour!". Add the highlighted code to the `viewDidLoad` function, as shown:

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Do any additional setup after loading the view, typically from a nib.

    lblOutput.text = "Bonjour!"
}
```

Here you can see Xcode's powerful code-completion feature in action; it can assist you in writing code much more efficiently. As you type `lbl`, a pop-up appears that says `UILabel! lblOutput`; when this is highlighted, press the Tab or Return key to complete the word `lblOutput`. Next type `.te`. Again Xcode's code completion snaps into action and shows a number of options, but the first one in the list is the one you want: `String! text`. With that item selected, press Return and continue typing the code. You can easily see from this example how Xcode's code completion helps you become a really efficient programmer, as well as how it helps cut down on errors.

To recap, what you've done here is to declare a variable that is linked to the `UILabel` you added to the view in Interface Builder using the Assistant Editor. You then added a single line of code to the `viewDidLoad` function to set the text of the label programmatically—well done!

Running and Testing

It's hard to stress enough how important it is to test your application thoroughly before even thinking about submitting it to the App Store. There are many reasons for this. First, the App Store review process is very thorough—if your app isn't up to par, Apple isn't afraid to let you know in the form of a rejection. So testing means you reduce your chances of being rejected by Apple. When you submit your app, if you're rejected, you have to make the amendments and then resubmit your application, all of which is time consuming—time that could otherwise have been used to sell your app. Second, when someone downloads your application, they're parting with their money and expect a certain standard. When they purchase and download an app, it's disappointing to find that it's slow and hard to use. Finally, testing makes you a better developer. Smoothing out the creases in your applications now helps you build good habits, and you carry these on until they become second nature. Testing can save you a lot of time when working on larger, more demanding projects.

Now that your application is ready to be run, the quickest way to check if it will build successfully without crashing is to choose **Product** ► **Build** (⌘+B). If everything's in order, you should see a small dialog stating that the build has completed successfully. It's time to run your application: choose **Product** ► **Run** (⌘+R), and Xcode will build and then run the application using the target specified, which is (by default at this stage) the iOS Simulator.

The iOS Simulator is invaluable when you need to test your application quickly or test a feature that you've recently implemented. However, it's important to note that testing your app using the iOS Simulator isn't the same as testing it on an iOS device—that is, an actual iPhone/iPod Touch

or an iPad. Applications may not perform the same on a device as they do on iOS Simulator, because the simulator doesn't simulate all software and hardware functionality. To change the type of device you'd like your application to be tested on via the iOS Simulator, go back to Xcode and click the Stop button in the top-left corner. With the application no longer running, go back to the iOS Simulator and choose Hardware ► Device, and then select from the list of devices available. Figure 2-17 shows the application running in the iOS Simulator.

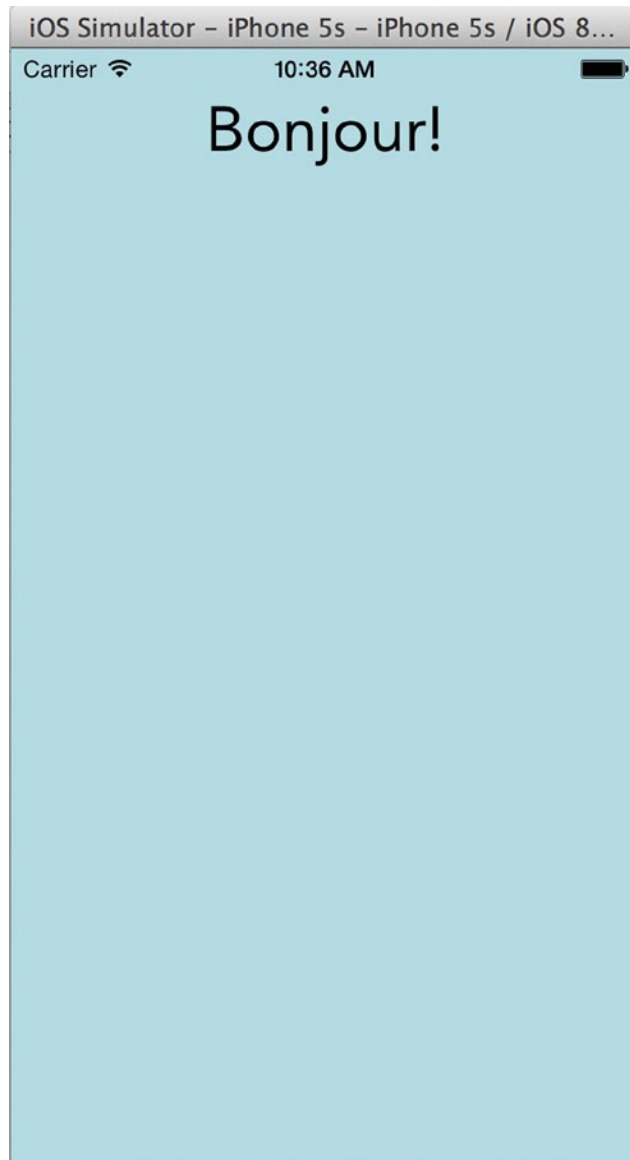


Figure 2-17. The iOS Simulator running the app using the iPhone 5s as the simulated hardware

Additionally, in the iOS Simulator you can change the orientation of the device, the scale at which the device displays, and an array of other options covered later in this book. What's recommended now is that you browse through the menus of the iOS Simulator while your app is running and play around with the options to get a little familiar with the different features.

Adding Files

One final thing worth mentioning at this early stage is how to add your own files to your project. Applications can be made up of literally hundreds upon hundreds of files, ranging from images to sounds. Let's add some images to the example application: let's change the app's icon without writing any code and then add a background image to the main view.

Before you add an icon file, you either need to create one or download the source code for this book from the Apress web site and use the included files. Because this isn't a book on iOS design or even iOS development, I won't digress about how to create perfect iOS app icons. Instead, I'll just state that you need to create a PNG file that, in this instance, I'm calling `icon120.png`, with dimensions 120 px by 120 px. Include whatever you like as the graphic, making sure it conforms to these specifications. I created a file with a basic gradient and a speech bubble saying "Hi!" in the middle. To set the application icon, you work with a feature that Apple introduced in Xcode 5, called Asset Catalogs; these are covered in more detail later in the book, but suffice to say they make the headache of managing retina and standard-resolution images far easier than having a folder with lots of different-sized images. Here are the steps:

1. From the Project Navigator, select `Images.xcassets`.
2. You're presented with two items in the left column of the Asset Catalog. Click `AppIcon`.
3. Bring the Finder window with the icon file in it over the top of Xcode, and then drag the icon file to the box labeled `iPhone App Icon iOS 7 60pt`, as shown in Figure 2-18.

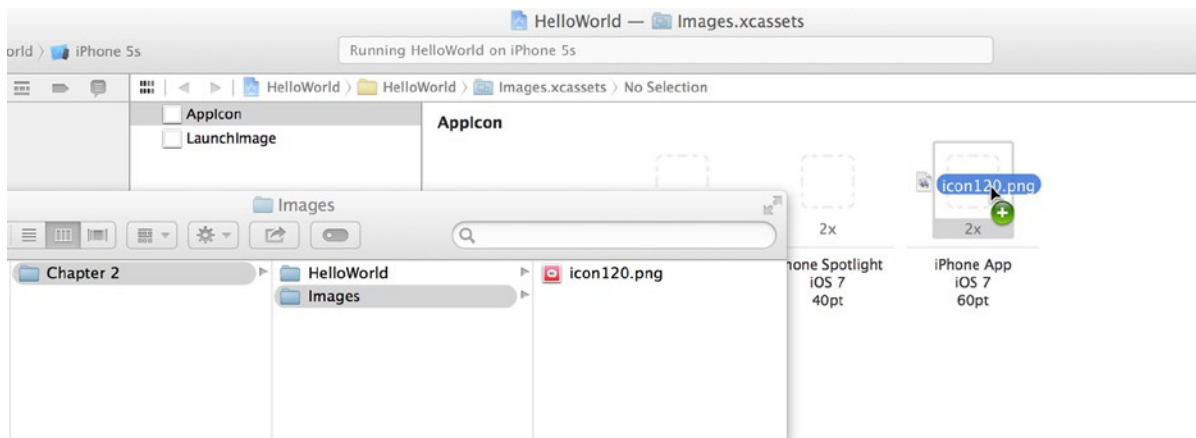


Figure 2-18. The Images Asset Catalog where you set the application icon

Run the application to see the icon in action. Once the app is running, choose Hardware ► Home (⌘+ Shift+H). If you’ve done everything right, you should see something like the image in Figure 2-19.

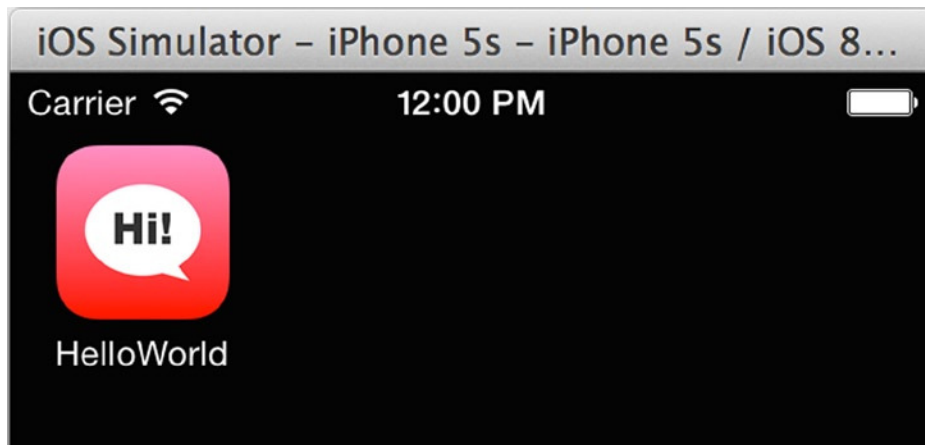


Figure 2-19. The application's new icon in the iOS Simulator

Note As you’ll learn as you progress through the book, Apple requires you to create icons in numerous sizes for the App Store, currently including up to 1,024 pixels square. It’s good to get into the habit early of adding the resolution to the filename to help you keep track of your assets.

With the icon successfully set, let’s look at another, more traditional way of adding files to the project. (You expand your Asset Catalog knowledge later in the book.)

They say there’s more than one way to skin a cat, and the same can be said for accessing Xcode’s Add Files dialog. First, you can choose File ► Add Files to “HelloWorld” (⌘+ ⌘+A), or you can right-click in the Project Navigator area and choose Add Files to “HelloWorld”. But the method I want you to use is to click the plus icon in the bottom-left corner of the Project Navigator and then click Add Files to “HelloWorld”, as shown in Figure 2-20.



Figure 2-20. The Add Files dialog available from the Project Navigator

The Add Files dialog will be instantly familiar to any user of OS X. Now you need to locate an image file you would like to use as the application background. In this example, I have downloaded an image from the fantastic website www.unsplash.com, which has a collection of images licensed under the Creative Commons license (they're public domain). The image is included with the downloadable resources for this chapter.

Once you've located your file, select it, and then make sure Copy Items If Needed is checked, as shown in Figure 2-21. Click Add.

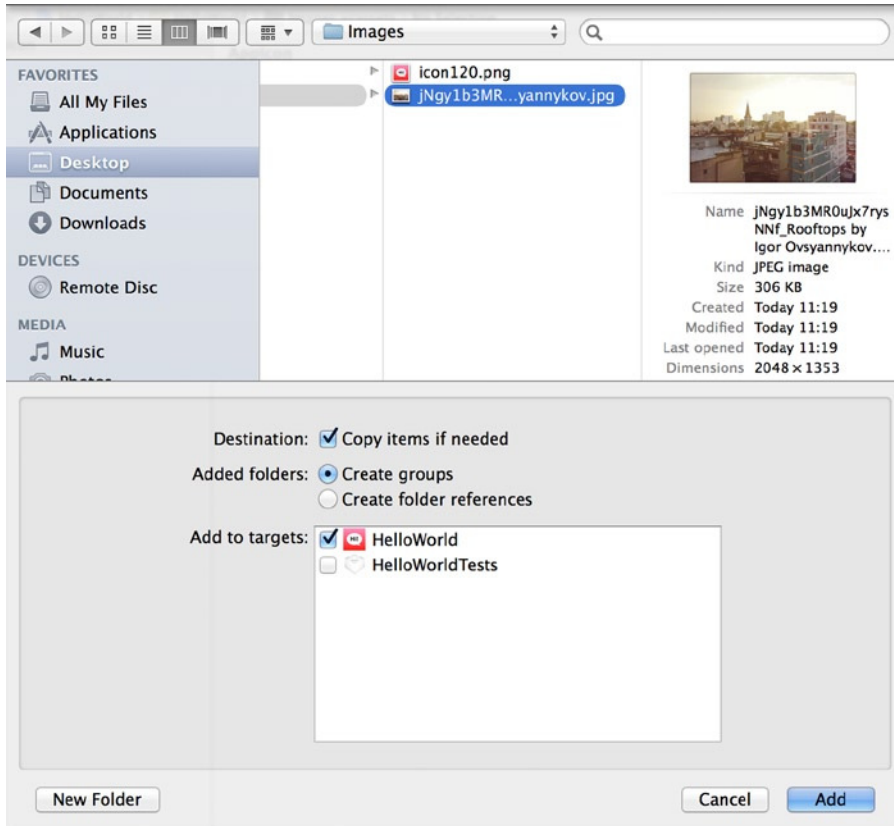


Figure 2-21. The Add Files dialog. Note that Copy Items If Needed is checked

At this point you return to Xcode. Look at the Project Navigator, where you see that your file has been added to the project's file structure. When I asked you to add the file, I also asked you to ensure that Copy Items If Needed was checked. The reason is that if you don't check this option, the file appears in the project structure as it does now, but the file itself isn't copied into the project. Hence, if you were to send the project to someone or to archive it, the image would be omitted.

Organizing Files in Xcode

Before you proceed and make this image appear in your view, let's talk about organizing files. As I mentioned earlier, Xcode gives the illusion of organization: a kind of faux folder structure that in Xcode is called Groups. Take the file you just added, and move it to the Supporting Files group by clicking it and dragging it until Supporting Files is highlighted. Figure 2-22 shows a comparison of the structure of the files in Finder compared to the structure of the files in Xcode.

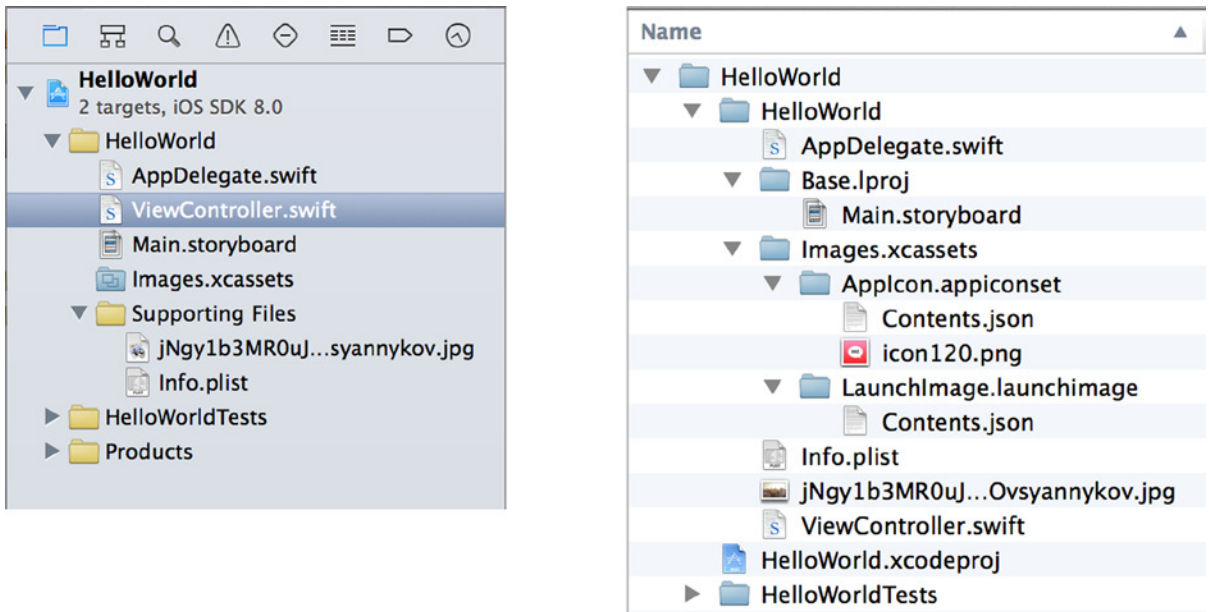


Figure 2-22. The structure of the files in the Project Navigator in Xcode (left) compared to those in Finder (right)

As you can see, there are some similarities, but there are items that are visible in one view and yet hidden in the other. This peek behind the scenes teaches you that what you see in Xcode may or may not physically exist on your file system.

You've done the hard work of adding the file to Xcode. Now let's do the fun part—adding the image to the View Controller using the Storyboard. Start by selecting `Main.storyboard` from the Project Navigator. Now, from the Object Library, select an image view (`UIImageView`) object, and drag it to your View Controller. If you're having trouble finding it in the list, remember that you can filter the list by typing "image" in the search field. Resize the Image View so it fills the entire view. Your screen should look something like Figure 2-23.

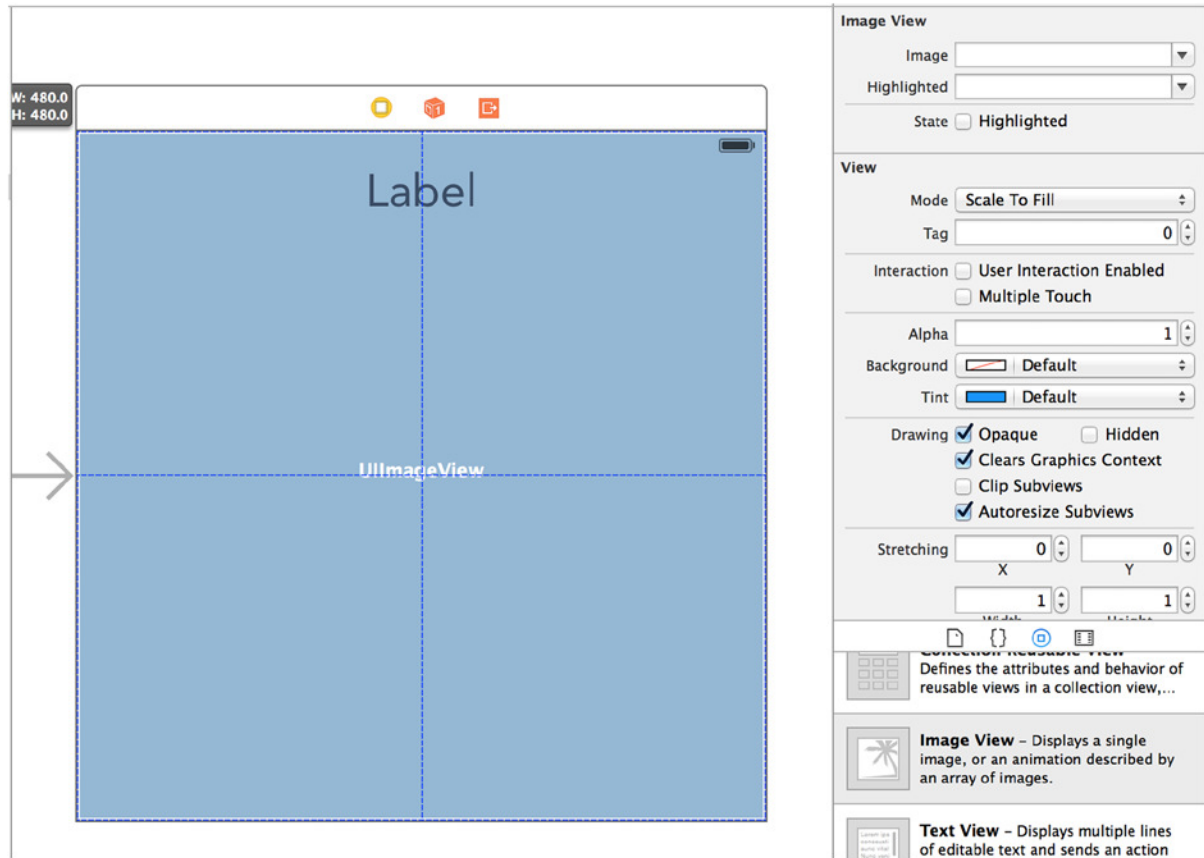


Figure 2-23. Adding the image view and resizing it to fill the view

Tip Remember, if you can't see the Object Library, you can access it via **View** ► **Utilities** ► **Show Object Library** (^+⌘+⇧+3).

With the image view positioned, it's time to specify which image to use:

1. With the image view selected, go to the Attributes Inspector.
2. From the Image drop-down, select the file you added. If you're using the image from the download, then this is the file starting with jNgy.
3. The image fills the image view, but it may have been distorted in doing so. You want the photo to fill the image view but maintain its ratio. To achieve this, click the Mode drop-down list, and change it from the default Scale To Fill option to Aspect Fill. Your image still fills the image view, but the ratio is maintained, preserving the original look of the image.

- You once again need to apply constraints, this time to the image view. Ensuring that the image view is still selected, click the Pin button at the bottom of the design area. This time, all four values should say 0; if they don't, change the values to zero and then click all four I bars as shown in Figure 2-24. Click the Add 4 Constraints button.

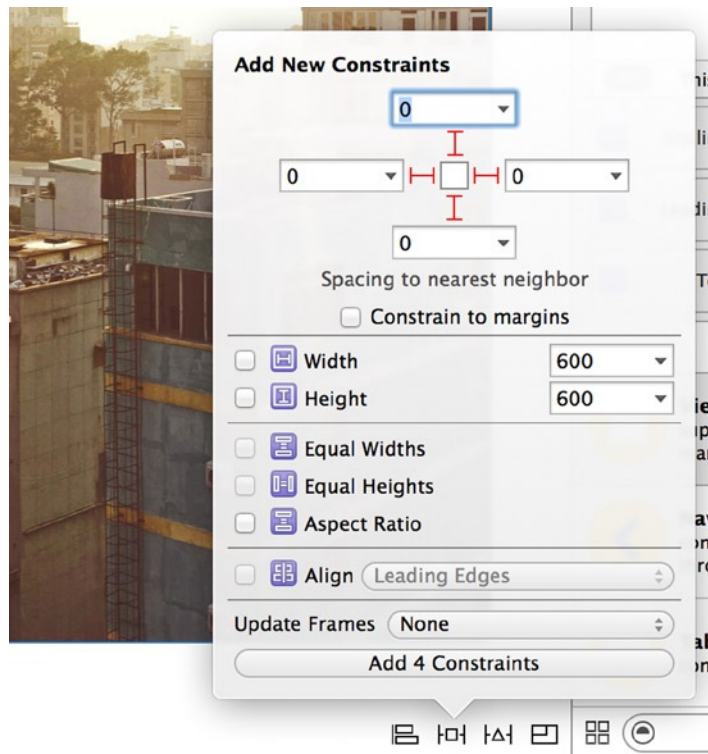


Figure 2-24. Applying constraints to the image view

So you're finished, right? Not quite. With the image view filling all the available space, you can't see the label you added at the start of the project! This is because of the way these two items are ordered, or rather how they're layered: the image view is rendered in a layer above the label, obscuring it. To resolve this, let's look at the Document Outline. If you can't see the Document Outline (the column between the design area and the Project Navigator), click the Show Document Outline button in the bottom-left corner of the Storyboard design area, or choose Editor ► Show Document Outline.

Expand all the items in the Document Outline. Beneath View, you should have your label, followed by the image view. Drag the image view carefully to move it above the label, as shown in Figure 2-25. Because of the hierarchy of the objects in the view, the image view is now rendered beneath the label, although you may need to tweak the color of your label to make sure it's visible against the image background.

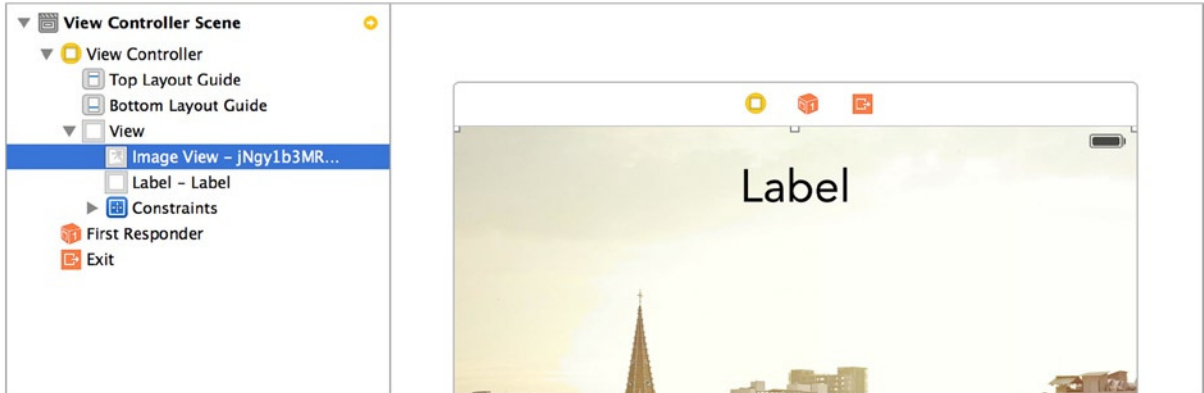


Figure 2-25. The Document Outline after rearranging the order of the elements in the view

You're ready to run your app! Figure 2-26 presents the finished product.



Figure 2-26. The finished app

Summary

You've learned a lot in this chapter. Specifically, you have done the following:

- Set up a new Xcode project and found that Xcode generates a functioning, yet boring, application
- Designed an interface using Xcode's built-in graphical interface builder, and used the Attributes Inspector to change some basic properties of the objects you added to your view
- Used the Assistant Editor to visually create connected outlets quickly and efficiently with drag and drop
- Looked at Xcode's code editor, updated the contents of a UILabel programmatically, and had an introduction to Xcode's code-completion tool
- Built and run your application in iOS Simulator, and looked at some different features of the iOS Simulator
- Compared and contrasted the basics of the structure of your files in the Project Navigator and the structure of your project in the Finder, and added new files to your project, which gave your application an icon without your having to write any code
- Added an image view (or UIImageView, to use its correct but less-friendly name) to your view controller, and set its image in the Attribute Inspector
- Moved objects in the Document Outline hierarchy

Essentially, the main purpose of this chapter wasn't to create a groundbreaking application but rather to give you a degree of comfort when it comes to working with the latest version of Xcode. It's easy to become overwhelmed with the sheer number of menus, tools, dialogs, and inspectors; however, as you've seen, creating an app isn't too daunting when you know where to look and what to press.

Chapter 3 introduces you to the different project templates that come with Xcode. You set up a new project and get a guided tour of the array of panels, windows, and menus that come with Xcode, along with how to quickly access them.

Project Templates and Getting Around

In Chapter 2, you created a very basic application and then tested it on the iOS Simulator. You were also introduced to the basics of the Project Navigator and looked at the Attributes Inspector. In addition, you programmatically updated the contents of a label that had been placed onto the view using Interface Builder and made it show an image file that had been added to the project, and you used an Asset Catalog to set the application's icon. In the first half of this chapter, you take a step back from app creation and look at the array of different tabs, inspectors, panels, buttons, and windows that come with Xcode, along with the different project templates that Xcode provides.

In order to be an accomplished app developer, it's important that you become intimately familiar with the key areas of your IDE; there's a good chance you already knew this, which is why you bought this book. By the end of this chapter, you should be a lot more familiar with many of the different interface elements of Xcode and should be able to quickly access Xcode features. You should also be able to choose a project template without having to worry about whether you've picked the right one, and you'll have some of the key knowledge required to start creating your own applications.

In this chapter, you create a working multiview application; then you learn how to pass information from one view to another and display that information in the ShowMe application. Passing information between views is essential for many applications; in this chapter you simply pass text between views, but in Chapter 8 you discover how to pass a selection from a table view to another view and also how to pass certain objects.

Without further ado, let's get started!

Project Templates

As a developer, you have the somewhat daunting task of making many, many decisions throughout the development of your application. With iOS and OS X apps, arguably the first decision you need to make is which project template to choose in Xcode. At this point you're optimistic, excited about the adventure ahead, and eager to get in there and begin writing your application, but not so fast! Choosing the right project template can have a huge impact on the direction your application takes, and that's why this section goes through each of them and explains the cases in which you should choose a project template provided by Xcode. It's worth mentioning here that, because the main focus of this book is iOS development, I don't go into detail about the OS X templates and instead focus more on templates targeted at iOS application development.

To begin, you need to fire up Xcode if you haven't done so already, and either choose Create A New Xcode Project from the Welcome screen or go to File ► New ► Project (⌘+Shift+N). You're greeted with a screen that presents an array of different project templates to choose from, as illustrated in Figure 3-1.

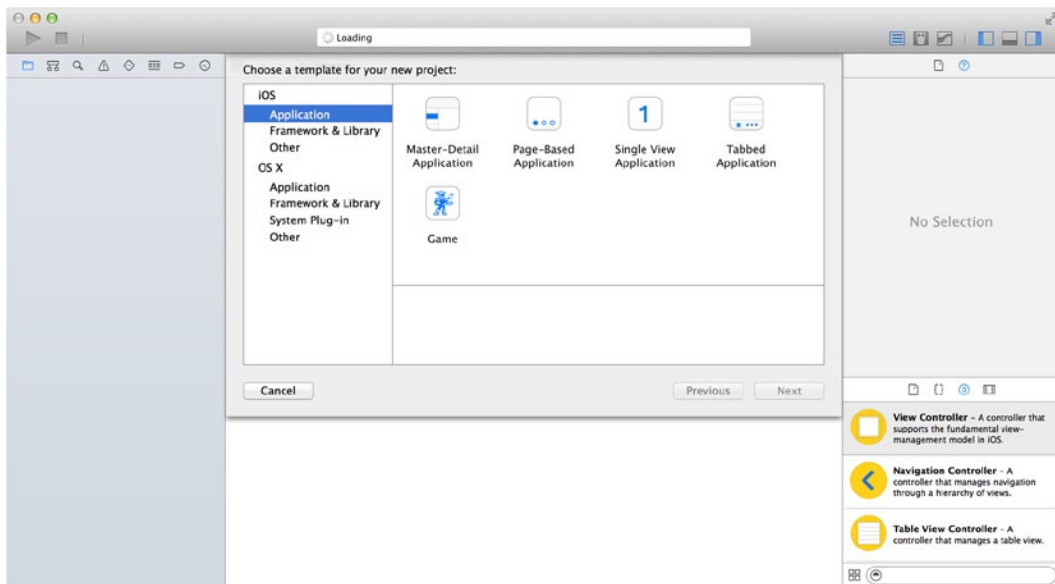


Figure 3-1. Xcode's different project templates

By default, when you first open Xcode and choose a project template, you're given the option to choose only from the ones provided by Apple; however, if you'd like to see what really goes into making an Xcode project and perhaps tinker with one yourself, the default location of Xcode's project template is `/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/Library/Xcode/Templates/Project Templates` (this is, of course, assuming that you have Xcode located in your `/Applications` folder). Here, you can open the different project templates provided by Apple and dissect them. However, I recommend that before you do this, you back up that folder just in case you change something that corrupts the template.

Note Notice that throughout this book you're given the keyboard shortcut equivalent whenever you need to access a menu item, open a window, or show a navigator or inspector. I strongly encourage you to take advantage of keyboard shortcuts, because using them can drastically improve your workflow and allow you to become a more productive developer—or, at the very least, make you appear to know what you're doing. It can also help make tiresome tasks somewhat bearable. What's more, many of the shortcuts that apply to Xcode can be brought over to other applications: for example, to Finder. You can also visit Xcode's preferences (⌘+,) and modify some of the shortcuts if they're not quite to your liking.

Master Detail View

The *Master Detail View* template is a starting point if you're looking to create an application that presents the user with a `UITableView` and then pushes a detail view when the user taps a row. By default, Xcode creates a project that, if targeting an iPhone, has one table view; the user can add rows by tapping the plus button in the top-right corner of the navigation bar. If targeting the iPad, a new row is added to the table view; however, the layouts of both the table and detail view fit much more nicely in the iPad's larger display.

Figure 3-2 shows the default project created when you specify Master Detail View as your project template. It's running universally via the iOS Simulator: iPad on the left, and iPhone on the right.

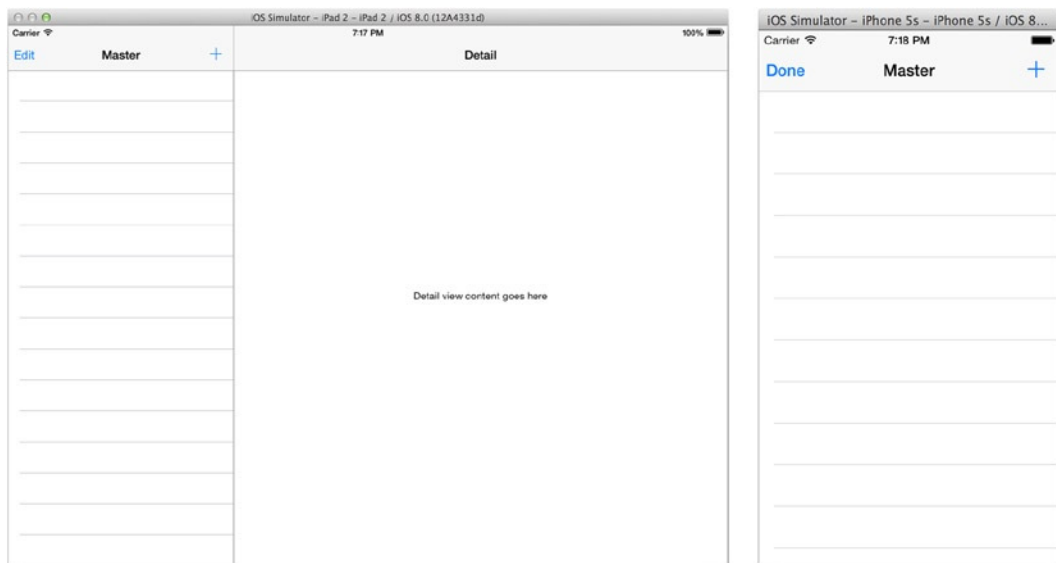


Figure 3-2. Master Detail View template running on both iPad (left) and iPhone (right)

Page-Based Applications

Creating an application using the *Page-Based* template gives users the impression that they're swiping through the pages of a book. With a page-based project, Xcode gives you everything you need to create an application that displays information in a book-like format: that is, it reveals information as the user swipes the screen either left or right. By default, you're provided with an object that adheres to the `UIPageViewControllerDelegate` protocol, which specifies the root view controller and initializes the view by loading `PageViewModelController`.

Figure 3-3 shows the default project created by Xcode when you choose to create a page-based application; on the left you can see it running on the iPad, and on the right it's running on the iPhone. If you swipe or click the left- or right-most side of the screen, the content is pulled over as if you're reading a book.

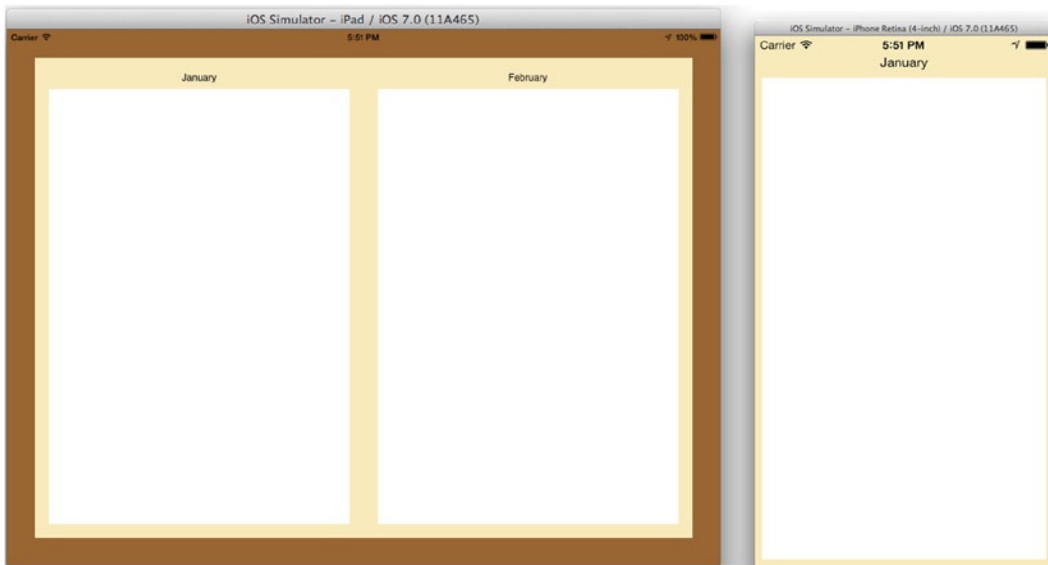


Figure 3-3. Page-Based template running on both iPad (left) and iPhone (right)

Single View Applications

This is perhaps the most organic project template provided by Xcode, and it will inevitably be the starting point for many applications. The *Single View* project template provides you with a single `UIViewController` that's loaded when the application runs. It's like a completely blank canvas in which the application can take any shape you like. This is especially useful if you're creating a custom iOS application, if you aren't sure of the exact approach you're going to take, or if the alternative templates don't seem appropriate for your project.

Figure 3-4 illustrates what you're given by Xcode when you choose this project template. Surprisingly, it's a blank, white view.

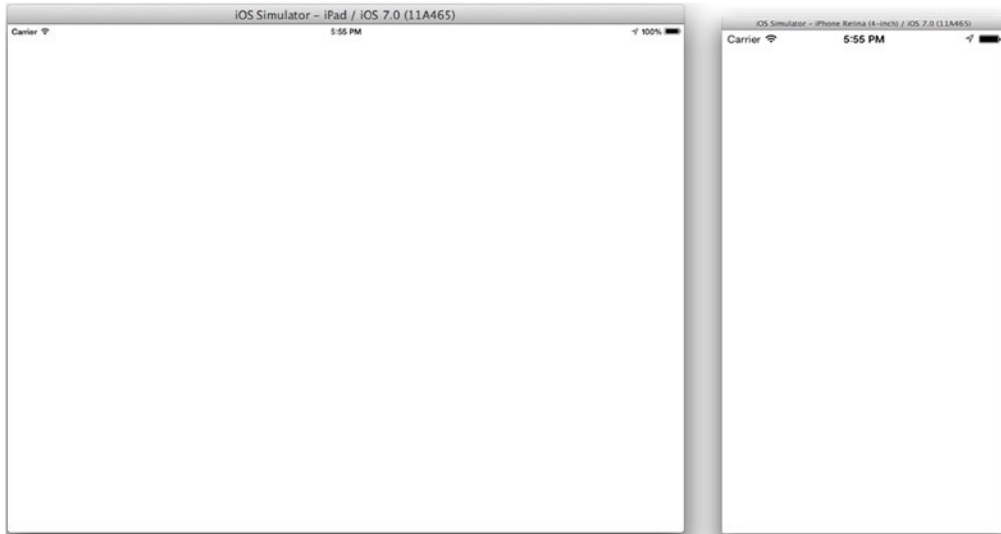


Figure 3-4. Single View template running on both iPad (left) and iPhone (right)

Tabbed Applications

Because many applications use tab bars to display different parts, it's no surprise that Apple has created a project template that allows you to quickly implement `UIViewController`s in a `UITabBarController`. By default you're provided with two view controllers, each of which has its own tabs.

Figure 3-5 shows a tabbed application. As you can see, the application consists of a tab bar with two tabs: the first loads `FirstViewController`, and the second `SecondViewController`.

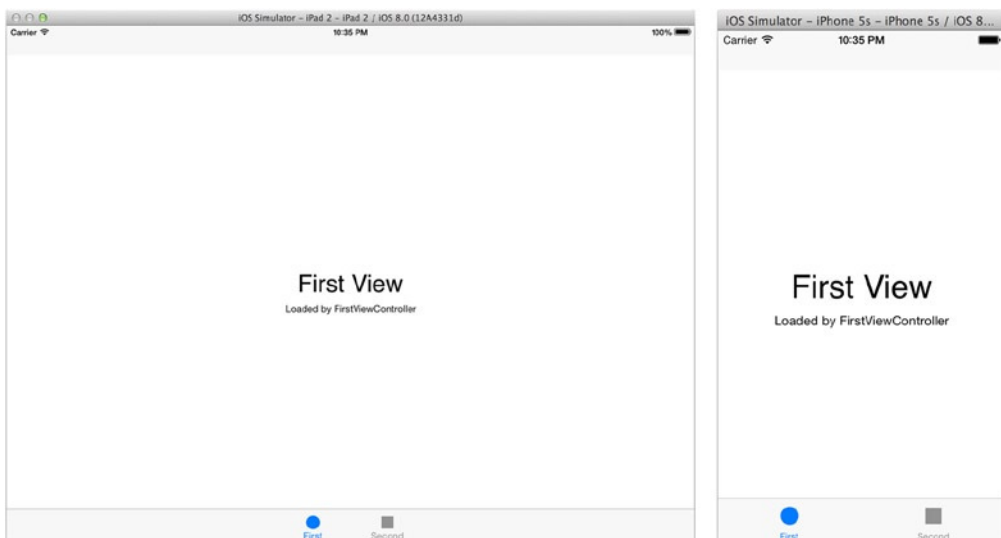


Figure 3-5. Tabbed template running on both iPad (left) and iPhone (right)

Games

In Xcode 6, Apple has consolidated the two separate game templates that existed in Xcode 5 (OpenGL Game and Sprite Kit Game) into a single template. What's more, Apple has also added two new game templates to the mix: the Scene Kit template and Metal, Apple's new high-performance alternative to OpenGL. Although this book isn't intended for game development, a project later in the book gives you a flavor of developing 2D games for iOS. Let's look a little more closely at each of the four variants of the game template.

OpenGL ES

The *OpenGL ES Game* template is an excellent choice if you're planning to create a game using Apple's OpenGL ES and GLKit frameworks. OpenGL is used across multiple platforms including consoles, PCs, and other mobile operating systems like Android; hence it's a good choice for cross-platform game development.

Sprite Kit

The Sprite Kit framework was introduced for the first time in Xcode 5 with iOS 7. Sprite Kit is Apple's answer to third-party game engines like Cocos2D. It may not have the features of Cocos2D, but its simplicity, coupled with its powerful physics and the animation tools it provides to developers, made it one of the hottest new frameworks in Xcode 5. In addition to giving you the tools to create your own version of your favorite 2D physics game, you can also use Sprite Kit to add complex animations to more traditional applications.

Note If you're interested, go to my blog at www.mattnott.com and search for Sprite Kit. I've written a tutorial on adding Sprite Kit particle effects to regular iOS apps.

Scene Kit

Scene Kit is a 3D graphics API that was first introduced with OS X 10.8 and has now made its way into iOS in version 8. Whereas Sprite Kit is a complete game engine, Scene Kit is designed to create and render assets and integrate with other technologies such as Sprite Kit, Core Image, and Core Animation. Scene Kit lets you render and manipulate 3D models in a regular app, such as presenting a strand of DNA that can be rotated, pinched, and pulled around.

Metal

Metal is a huge deal for iOS game developers; there isn't any other way to describe it. In a huge leap forward, Apple has created this new technology as an alternative OpenGL ES in iOS 8. Apple's developers have basically created a bespoke API that can squeeze every ounce of power out of the current A7 chip in the latest iOS hardware. The performance specs being touted by Apple imply that Metal will be able to redraw models on the screen ten times faster than with OpenGL ES. This means better graphics and better performance without the need for a hardware upgrade.

Note In Xcode 6, in addition to consolidating the game templates, Apple also decided to retire two templates that users of older versions of Xcode may be looking for: Empty Application and Utility Application. The animation found in the utility application fell out of fashion with the iOS 8 UI, and the Empty application, as you learn in this chapter, became easily replicable with the Single View Application template.

Template Selection

Now that I've explained each of the five default project templates provided by Apple, let's start looking at the various panels and panes you see in Xcode. To help with this, you're going to build a simple with two views or screens, using an innovative system for constructing interfaces and linking views together. Storyboards make the development process much quicker, more visual, and more accessible. I hope that through this example, even though it isn't complex, you can appreciate the real benefits of storyboards ahead of later chapters where they're used more extensively. For this project, called ShowMe, you use the Single View Application template.

Once you've started the process of creating this app, you need to specify which application template you would like to use—that is, which one will best suit this application. In this instance, and as I just mentioned, select the Single View Application template. Typically, when choosing a template, you go through a thought process like the following to make sure you start with the right template:

- *How users will navigate around your application:* If you're using a good-old UINavigationController as the crux of the application, chances are you need to choose a single-view application and then implement a UINavigationController manually. However, if users will navigate using the UITabBarController, then your best choice, surprisingly, is a tabbed application.
- *How you'd like your screens to be laid out:* Again, if screens will be pushed via a UINavigationController or displayed as a single UIViewController, a single-view application will suffice. However, if you're creating a book or magazine, the Page Based template is your best bet.
- *Whether you're creating a game:* If you're creating a game, Apple provides the Game template, which supports OpenGL ES; Sprite Kit for 2D games; Metal for 3D games; and Scene Kit to render 3D assets. Combined, these give you the tools and features for almost any game project.

As with any other application created using Xcode, you need to start by creating a new project. Let's begin:

1. Create a new Xcode project by going to File ► New ► New Project (⌘+Shift+N) or, alternatively, clicking Create A New Xcode Project from the Welcome screen (⌘+Shift+1).
2. As I've already specified, select Single View Application from the Project Templates dialog, and click Next.

3. You're required to provide Xcode with those all-important little details such as Product Name, Organization Name, and so on. Figure 3-6 illustrates the values to put in (remember to enter your own first and last names in the relevant fields, though!). For Product Name, use ShowMe.

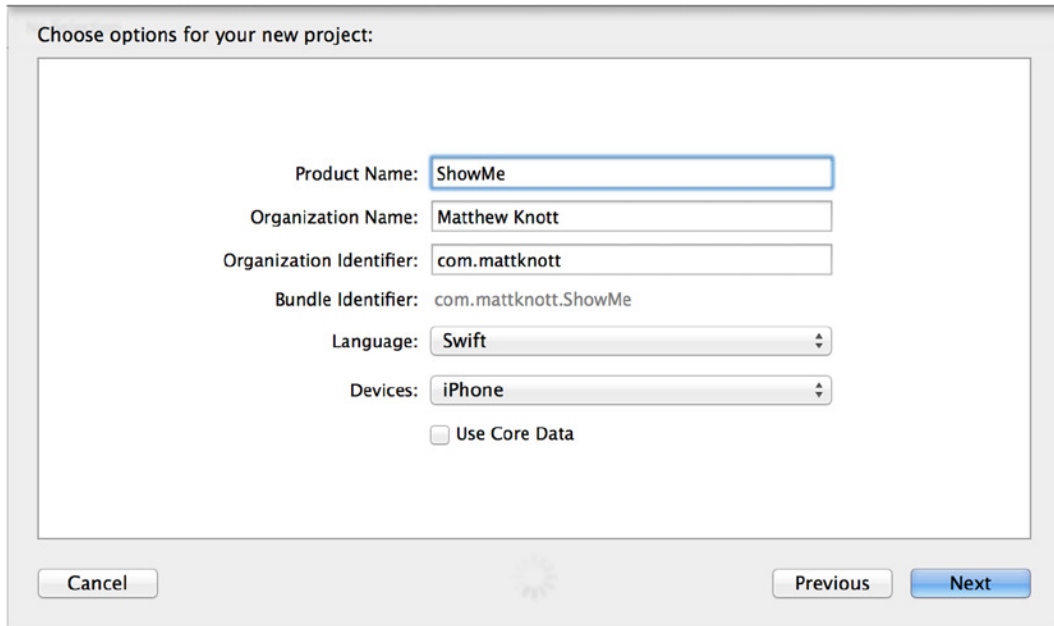


Figure 3-6. Inputting the options to create the application

4. For this project, you can specify your own name (that is, your first and last names) as the Organization Name; for example, in my case this would be Matthew Knott.
5. For Organization Identifier, I used reverse domain notation and entered `com.mattknott`, but you could just as easily use `com.YOURSURNAME`.
6. Ensure that Device is set to iPhone and Language is set to Swift, and that you've unchecked Use Core Data.
7. To finish, click Next. You're prompted to choose a location for your project. Save it somewhere that's easy for you to find, and ensure that Source Control is unchecked. Click Create. Now you're now ready to explore the many different areas of Xcode.

Getting Around

Now that your application is ready and the project is set up, it's be useful to become familiar with the main areas of Xcode's interface: the navigators, toolbar, editor, utilities panel, and debugging area. Essentially, most actions you need to perform are in those main areas of the interface, with the exception of actions contained in the menu bar. This section focuses on each of these areas so that when attention is brought to them later in the book, you know where to look and what purpose they serve. Figure 3-7 shows a breakdown of the main area of Xcode's interface.

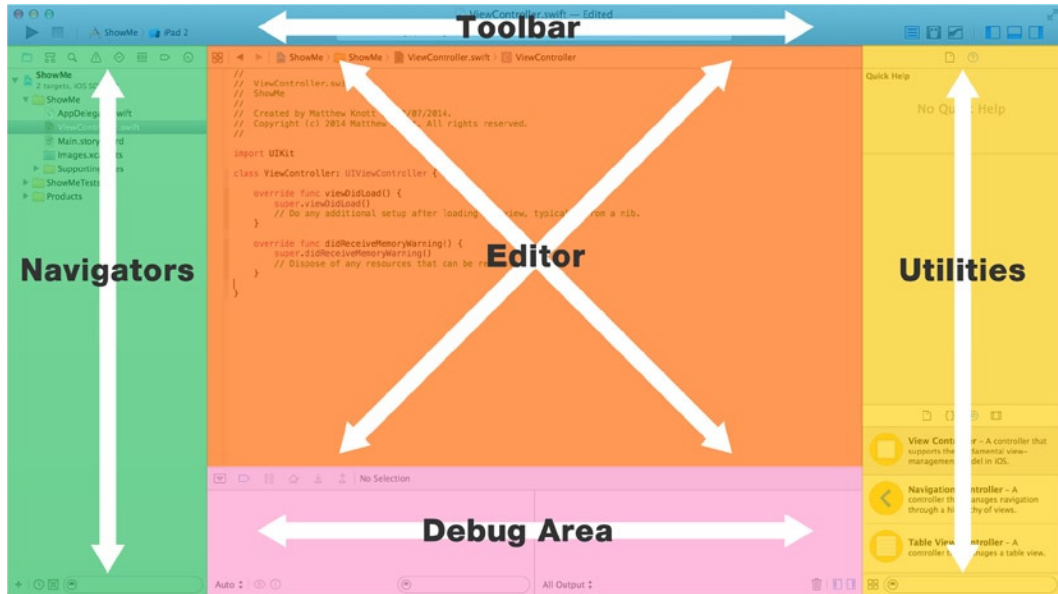


Figure 3-7. The five main parts of Xcode's interface

Navigators

Let's first focus on the far-left side of Xcode: the section that houses the navigators. A *navigator* in Xcode is something that allows you to browse various things, whether files, folders, warnings, build errors, memory leaks, breakpoints, and so on. If you look at the top of the navigator panel, you see that you can toggle the view using eight different tabs, as shown in Figure 3-8:



Figure 3-8. Xcode's different navigators, among which you can toggle

- *Project Navigator*: Perhaps one of the most important features of the enter navigator panel. It allows you to explore the files in your project and also displays what frameworks and interface files your project is made of. As previously mentioned, the file structure of the Project Navigator doesn't correspond to that of the Finder; things like folders are there just to make it easier for you to navigate your project in Xcode.
- *Symbol Navigator*: Where you can browse what Xcode considers a *symbol* in your project. A symbol is generally the name of a class or a function. In the bar at the bottom of the Symbol Navigator, you can filter what's displayed and view Cocoa's built-in symbols or symbols defined in your project. The Symbol Navigator is especially useful when you have dozens of Swift class files and you need to quickly browse them to find a specific class declaration or function.
- *Search Navigator*: A very useful means of searching your project to find a certain bit of code. If you select the small magnifying glass in the search area, you can specify search options such as where Xcode should look, and you can more accurately define what you're looking for. The Search Navigator is very useful when you need to quickly find something and you have hundreds of different files and thousands of lines of code.
- *Issue Navigator*: Alert you to any issues by flagging warnings and errors as you develop your application. The Issue Navigator not only tells you what's wrong but also attempts to accurately pinpoint exactly where the issue lies.
- *Test Navigator*: Where you find your test targets and test classes. From here, you manage all of your tests. They're coded in a way similar to how the Project Navigator and the editor work together: selecting a test opens the relevant code and allows you to write unit tests that ensure individual classes and functions work as they should.
- *Debug Navigator*: Used when your code pauses. By default, it opens if a pause is encountered. It also appears when a breakpoint is reached in your code. Otherwise, the Debug Navigator remains dormant. When in use, it displays call stacks along with the names of nested functions. If you click a function name, you can navigate through it further. In addition, useful CPU and memory monitors display the real-time impact of your code. The Debug Navigator should not be confused with the debug area of Xcode, which I cover later in this chapter.

Note *Breakpoints* essentially tell Xcode when to pause your program. They're especially useful when you're trying to pin down an issue with your code. To add a breakpoint, open the editor and click the line number in the gray area on the left, just between the navigator pane and the editor.

- *Breakpoint Navigator*: The hub in which you manage breakpoints. With a project that has dozens of breakpoints, you'll soon become accustomed to using this tab. In the Breakpoint Navigator, you can also create different types of breakpoints: for example, symbolic breakpoints.
- *Log Navigator*: Like the history option of your Internet browser, except that instead of recording what you open, it records your actions. Specifically, it lists the status of a build (whether it failed, succeeded, or succeeded but has errors). To reveal all the details of something like a build, simply click the log item; Xcode brings up a new dialog in the editor, showing all the necessary details regarding what you clicked.

Now that you know exactly what each of the eight different tabs corresponds to, there must be a quicker way to access them, as opposed to having to click them each time. Well, first and foremost, Xcode has a tendency to spontaneously hide some of its interface elements (usually because you've clicked a button mistakenly). If you ever lose the navigator sidebar, go to View ► Navigators ► Show Project Navigator to bring up the Project Navigator. To quickly switch between the tabs of the navigator, press ⌘+1 (for the Project Navigator) or ⌘+8 (for the Log Navigator). Again, it's handy to use keyboard shortcuts, because they can dramatically increase your productivity—which, to a developer working through the night to complete a project for the morning, is everything. If you're short on space, you can press ⌘+0 to hide the navigator pane.

Toolbar

Moving on from the navigators section, there is the Toolbar. The Toolbar is present throughout many familiar OS X applications (such as Finder), and it houses many useful buttons and displays important information regarding build results. If you've previously used Xcode 4 or an earlier version, you see a number of changes, most notably the size: the Toolbar has been compressed somewhat in Xcode 5. To tackle the Toolbar, let's examine each of the default buttons, starting on the left and moving to the right. Figure 3-9 shows the default layout of the Toolbar.



Figure 3-9. The Xcode Toolbar's default layout

First you see two buttons on the left: Run and Stop. These are rather self-explanatory at this point, but clicking the Run button starts a build of your project and then launches it using whatever target is specified in the active scheme, just to the right. In this instance, the scheme is set to ShowMe and iPhone 5s. Once it's running, you can stop your project by clicking the Stop button. Additionally, if you click and hold your mouse over the Run button, you can choose from Run, Test, Profile, and Analyze. I explain what these do later in this book; briefly, if you select an option from the menu, it takes the place of Run and performs the specified action each time you click it.

Next are the active scheme and device target. This is where you can choose a scheme, which specifies how you'd like to run your project. Select ShowMe, and you're given the option to choose Edit Scheme, New Scheme, or Manage Scheme. A scheme allows you to specify in more detail how you'd like your application to be run or debugged. If you click the iPhone 5s section, a drop-down menu appears in which you can choose from different platforms on which to test your project.

Next is the Activity Viewer, which tells you what is happening when Xcode is performing an action. For example, if you choose to clean your project, the Activity Viewer displays the progress of the clean, similarly to when you're building an application. What's also nice about the Activity Viewer is that, if you're running the latest versions of Xcode and OS X, it displays the last action performed along with when it was performed. Finally, the Activity Viewer displays small icons near the button that let you quickly see the number of issues or errors found in your project.

On the right of the Activity Viewer are three editor buttons that change how the editor in Xcode looks and behaves. You can choose the Standard, Assistant, or Version editor, respectively. Click to open Main.storyboard from the Project Navigator (⌘+1), and then toggle between the three different editors and see what happens. These are covered in the next section of this chapter.

Finally, you have three view buttons. These are very useful when you lose one of the main elements of Xcode's interface. The first button toggles the navigators section, the middle button toggles the debug area, and the third button toggles the utilities section.

As with many other OS X applications, if you right-click a gray area of the Toolbar, you can customize whether you'd like icons and text to appear, just icons, or just text. This isn't particularly groundbreaking, but if you prefer to have only icons, text, or both of them together, feel free to change this option.

It's also worth mentioning that Xcode does support full-screen mode as introduced natively with OS X Lion. This is especially useful when you're working with the Assistant editor, previewing layouts with size classes or storyboards, or designing iPad application interfaces. To toggle full-screen mode, select the small arrows in the top-right corner of Xcode.

Editor

Perhaps the most important part of any integrated development environment is its code editor. Xcode's editor is exceptional in many ways. It has three different view options—Standard, Assistant, and Version—each of which is covered shortly.

If you open AppDelegate.swift, you see that the editor is front and center. Simply click where you'd like to begin coding, and then code away. As you type code, Xcode's code-completion feature appears. To choose an option from the code-completion dialog, use the arrow keys on your keyboard to navigate the suggestions (sometimes there are multiple ways to instantiate a class), and press Return or Tab.

Notice that just above the editor window is a small jump bar. You can use this to open files, see function declarations, and more efficiently navigate through your project and your code. I revisit this in Chapter 10.

Standard Editor

The Standard editor displays a single window and focuses on what has been selected from the Project Navigator on the left (Figure 3-10). This is the preferred way of coding mainly because of its simplicity.

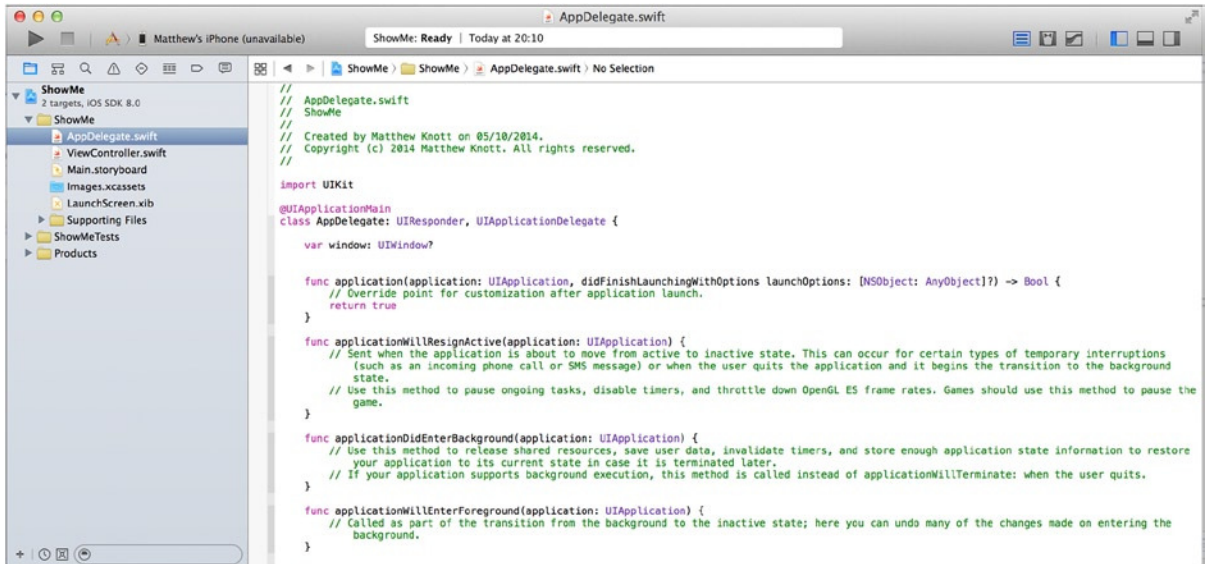


Figure 3-10. The Standard editor

Assistant Editor

The Assistant editor offers a much more interesting approach, and chances are it will make you want to go out and purchase a larger display. The Assistant editor displays separate windows and contains logical contents depending on which file you're working with. For example, in Figure 3-11, I have the Main.storyboard open on the left; as a result, Xcode opens ViewController.swift on the right automatically. This allows you to work simultaneously on both files without having to worry about switching constantly.

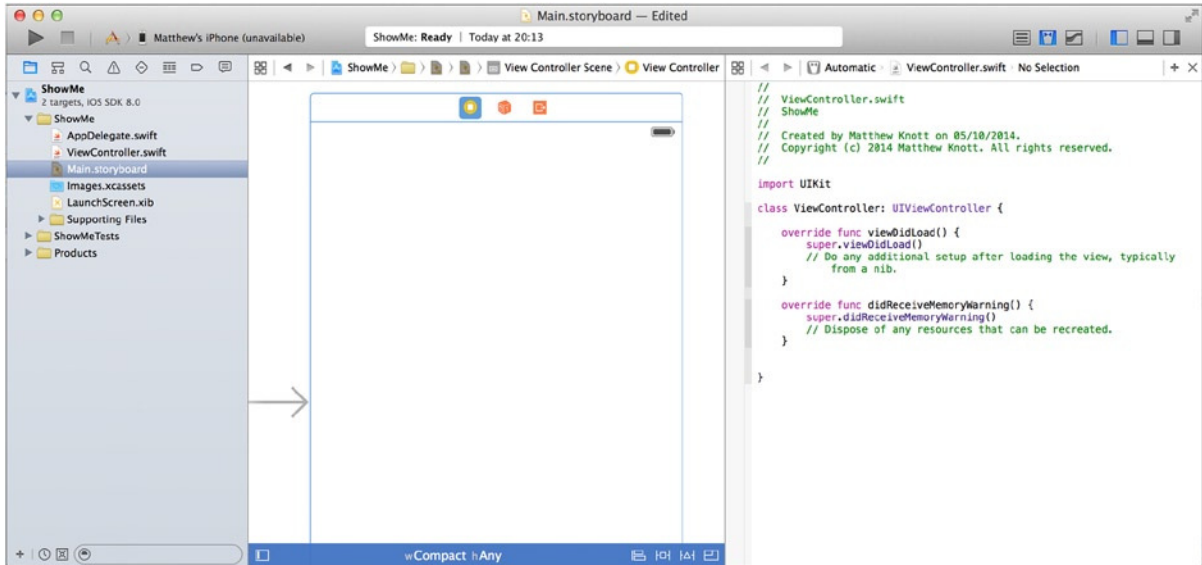


Figure 3-11. *The Assistant editor*

Notice on the far right, along the jump bar, a small button with a plus symbol on it. If you click this, Xcode allows you to have multiple editors open—as many as you and your display can handle. In addition, it's important to understand that you aren't restricted to the automated file: you can choose any file in the project to view. This can be useful when you're referencing keys in a strings file for localization of an app, which is something you do toward the end of the book.

Version Editor

Because you haven't made use of version control yet, the Version editor isn't too significant right now. All you need to know is that, as you can see in Figure 3-12, the most recent version of a file is selected on the left, and Xcode opens another version of that file on the right and lets you track and view changes made to this file.

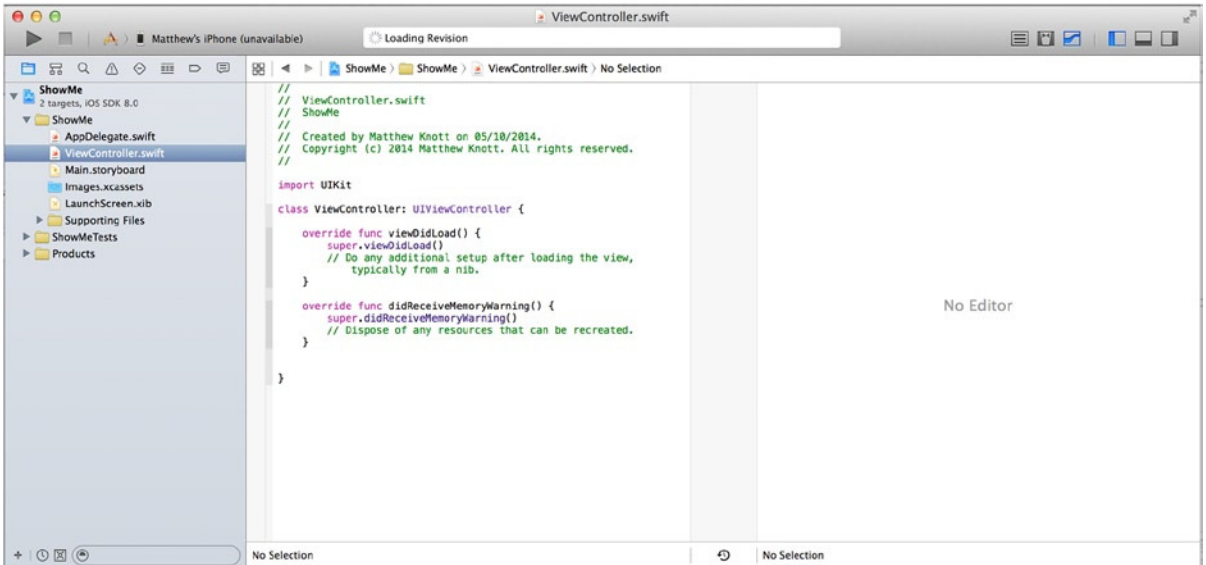


Figure 3-12. The Version editor

Utilities

The utilities area provides essential tools for your project. What's useful about the utilities area is that it varies in terms of what it displays, depending on what you're using. Like the navigator area, the utilities area consists of different tabs along the top, but it also includes tabs toward the middle and bottom.

Let's first focus on the main tabs along the top. Providing you have ViewController.swift open, you see two tabs along the top, as shown in Figure 3-13.



Figure 3-13. The tabs in the utilities area of Xcode with a code file open

First is the File Inspector. It lets you manage attributes of a file: for example, its name, type path, and location in your project. As with many other inspectors in the utilities area, additional options can be changed if you scroll down. The File Inspector is one of two inspectors that are always present in the utilities area, regardless of which file you're working with.

Second is the Quick Help Inspector. Here you can easily access information about a symbol in Xcode. This is especially useful when you'd like to know where something has been declared, how it was declared, and its scope and parameters. This is the other tab that is always present in the utilities area.

The Utility Navigator really comes into its own when you're working with Interface Builder. Interface Builder was introduced in Chapter 2, when you used it in conjunction with the `Main.Storyboard` file. This is the file you start with in this chapter:

1. Open `Main.storyboard` from the Project Navigator. You have only a single view in the storyboard; but this will be a multiview application, so let's add a second view. Just as you did with the label and image views in the previous chapter, view controllers can be dragged in from the Object Library. Drag in a view controller from the Object Library, and position it to the right of the current view controller, as shown in Figure 3-14.



Figure 3-14. Dragging a view controller from the Object Library to the design area

2. Once you've released the view controller, use the small bar at the top of the view controller to maneuver it neatly beside the existing view controller, as shown in Figure 3-15.

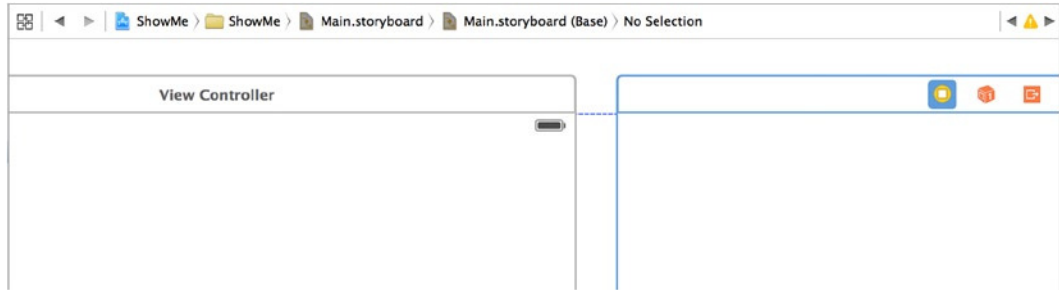


Figure 3-15. Moving the view controller so that it snaps neatly into place beside the existing view controller

3. At this point you have two view controllers on the storyboard. But remember from Chapter 2 that a view controller in a storyboard needs a view controller code file in order to interact with the visual portion. You need to create a new view controller file called `MessageViewController` that subclasses `UIViewController`. I explain subclassing in great detail later in the book; for now create the file by going to `File > New (⌘+N)` and selecting `Cocoa Touch Class`, as shown in Figure 3-16. Click `Next`.

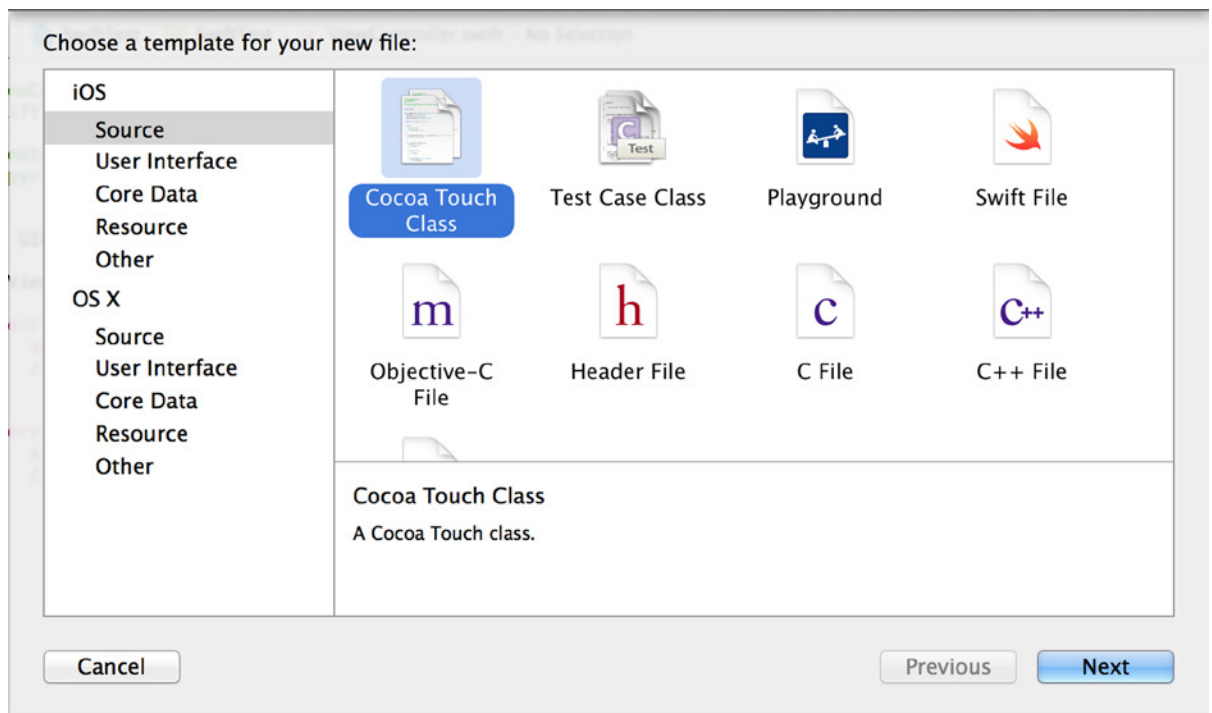


Figure 3-16. The new file template selection

4. On the next screen, you're asked for two values: Class and Subclass Of. For the Subclass Of value, you need to tell Xcode which class your new class is based on. If you were creating a class to hold custom properties, such as a `Car` class or an `Animal` class, you would use a generic `NSObject`; but in this instance you need a view controller, so set the value to `UIViewController`.
5. The Class value is largely up to you; this is the name you use to instantiate this view controller. When naming classes, always try to make the names semantically accurate—that is, they should describe the function of the class. This view controller displays the message it's sent, so I named it `MessageViewController`.
6. Ensure that Also Create XIB File is not checked, that the device shown is iPhone, and, of course, that Language is Swift. Check that your values match Figure 3-17, and click Next.

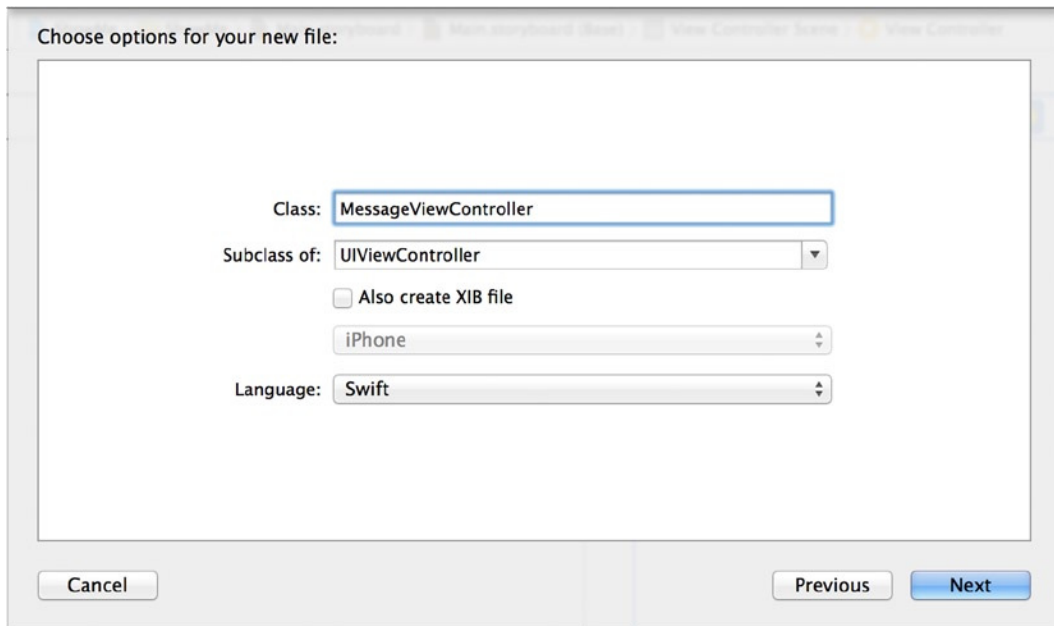


Figure 3-17. Subclassing `UIViewController`

7. On the next screen, you're prompted for a save location for your new file. Stick with the default settings. Ensure that in the Targets box, `ShowMe` is selected; then click Create.

You should now have a new file in your Project Navigator called `MessageViewController.swift`, which means you have all the files needed for the project and you're ready to put together the interface for the first view controller. Reopen `Main.storyboard` from the Project Navigator. Before you add anything to the view, click the left view in the storyboard, and then look back to the utilities area on the right. You should see that the two tabs in Figure 3-13 have become six tabs, as shown in Figure 3-18.



Figure 3-18. The utilities tabs in Interface Builder

I've already covered the first two tabs, so let's look at the remaining four:

- *Identity Inspector*: Here you can change details regarding an object, similar to the File Inspector. You can change and access the class name, accessibility details, runtime attributes, and so on. The Identity Inspector is only active when you have an object selected. With `MainViewController.xib` open, select the view, and the information in the Identity Inspector should become visible. You'll use this tab often when adding views to a storyboard, as I cover later.
- *Attributes Inspector*: A very useful inspector to work with when you're designing interfaces graphically. With the view selected, you can change many properties, such as background color and so forth. Without this tab, you'd have to make changes programmatically, which would be not only time-consuming but also tedious and tiresome. All the different objects you can add through Interface Builder have different properties that can be configured.
- *Size Inspector*: Allows you to specify the positioning of objects that are selected, along with minimum and maximum sizes and so on. This is also one of the places you can view and manage constraints, as covered in detail later in this book.
- *Connections Inspector*: Lets you connect outlets to interface objects as well as make new connections and break existing ones. The Connections Inspector is essentially an overview of which parts of your code the visual elements are connected to. For example, when a label is populated, it shows the name of the outlet you call in code; and in the case of a button, it indicates which action the button triggers when clicked.

Like the Project Navigator, each of these tabs is accessible via `View > Utilities > Show File Inspector`; more important, the tabs can be accessed using similar keyboard shortcuts. To access the File Inspector, simply press `⌘+⌘+1`. For the Attributes Inspector, press `⌘+⌘+4`. If you want to quickly dismiss the utilities area, use the keyboard shortcut `⌘+⌘+0`.

Just below the inspectors are four more tabs: File Templates Library, Code Snippets Library, Object Library, and Media Files, as shown in Figure 3-19:

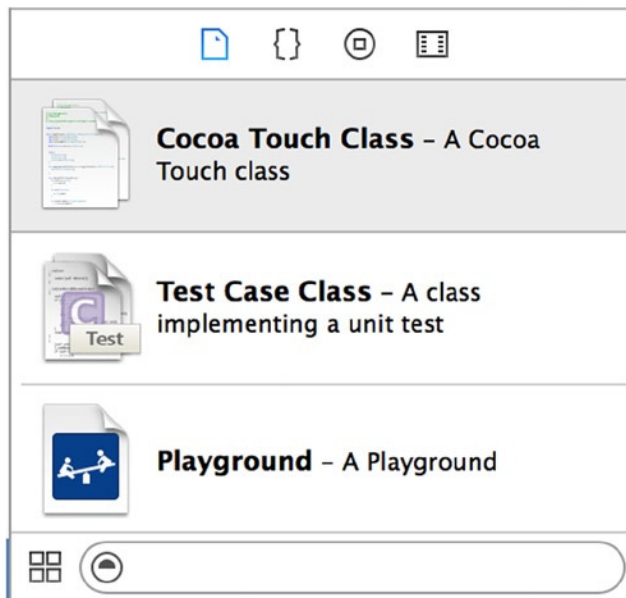


Figure 3-19. The File Template, Code Snippets, and Object libraries, along with the Media Files tab

- The *File Templates Library* contains templates for common classes, heads, protocols, and so forth. To add one to your project, click and drag it to the Project Navigator.
- The *Code Snippet Library* contains short pieces of code that you can use by clicking and dragging them into the editor with a code file open.
- The *Object Library* is where you find the standard Cocoa controls for applications. You make extensive use of this library throughout this book. Simply click and drop an item onto a view with Interface Builder open.
- The *Media Files* tab contains useful graphics, sounds, and icons you can use in your interfaces, again, by dragging and dropping them onto your interface.

These four tabs are accessible via View ► Utilities ► Show File Template Library or with the keyboard shortcut $\text{^+}\text{⌘+1}$. The tabs are always visible, regardless of what file type you're working with; however, they're most useful while working with Interface Builder.

Before adding any objects to your view, you need to add an element called a *navigation controller* to manage the navigation back and forth through the different views. Xcode makes this very easy to do:

1. If you haven't already, select the left view in the storyboard by clicking the large white area in the design area.

2. Go to Editor ► Embed In ► Navigation Controller, as shown in Figure 3-20. You should see a navigation controller appear to the left of the view. Once it's there, you can pretty much ignore it for the rest of the project, because as the focus will be on the two views; but without it, navigating views would be extremely troublesome.

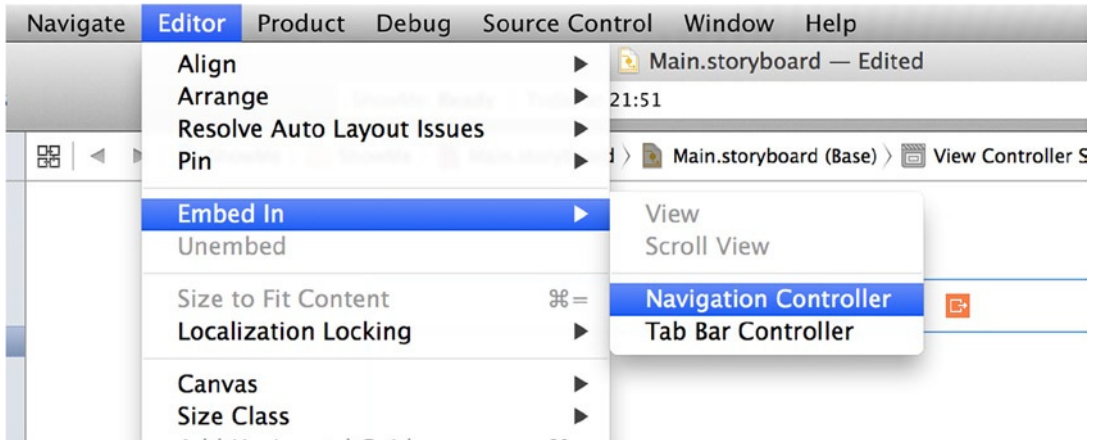


Figure 3-20. Embedding in a navigation controller

3. To make the design process easier, let's change how the view is displayed. At the bottom of the design area are the words *w Any h Any*. Click this location, as shown in Figure 3-21, and move your mouse around until you get the vertical compact area. This make the view the shape it will appear on the simulated iPhone, so more of the view is visible; this only affects Interface Builder. After you click the new area, it says *w Compact h Any*.

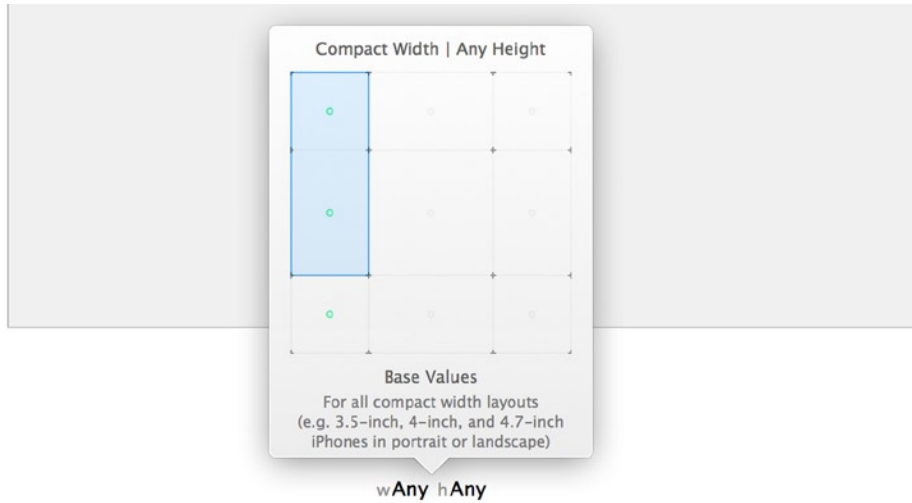


Figure 3-21. Changing the dimensions of the view

4. Position the design area so you can see the first view, which has a line attaching it to the navigation controller.
5. From the Object library, you need to add three items to the view. First, drag a label onto the view, and position it toward the top of the view in the center.
6. Click the label once to select it, and then select the Attributes Inspector. Here you can really appreciate the range of minute customizations available to you. The second property in the Attributes Inspector for this label is a text field that says Label. Change this to say **Text to send**. You can also edit labels by double-clicking them on the view, but this method helps you see some of the minute adjustments you can make in the Attributes Inspector.
7. Click the T in the Font box, and select Headline from the font list, as shown in Figure 3-22.

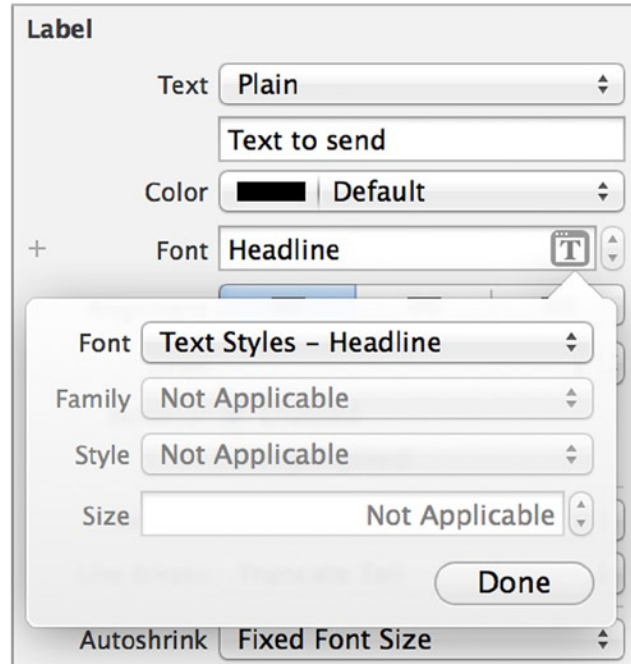


Figure 3-22. Selecting the Headline text style for your label

8. When you changed the text, the label probably truncated most of it. Rearrange the label by dragging one of the small handles in the corners until all the text is visible. You may need to re-center it after this.
9. Now that the label is in place, drag a text field from the Object Library onto the view, and position it below the label. As you move the text field, you should get a feel for the vertical positioning as the object snaps into place below the label. Drag out the sides of the text field until a blue line appears on the side of the view; this indicates the view's margin.
10. Drag a button onto the view. Position it a little below the text field. Then double-click the button and change its text to **Show Me**. That's it! You've built your interface, and you should have something resembling Figure 3-23.

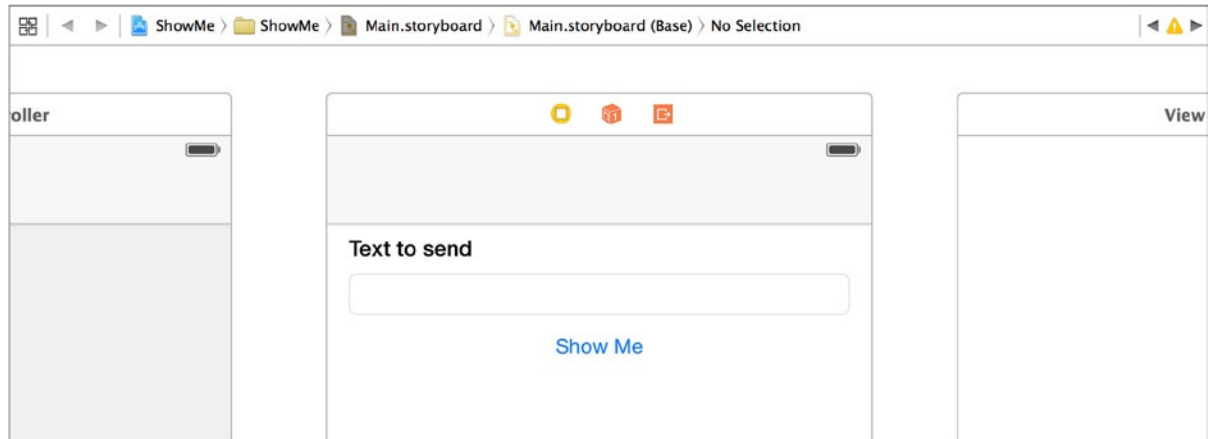


Figure 3-23. The `MainViewController.xib` file with all of its elements

11. You need to add some constraints in order to make sure the elements line up correctly when the application runs. Click a white area of the view, and then click the Resolve Auto Layout Issues button in the bottom-right corner of the design area. Select Add Missing Constraints, and Xcode will do the hard work for you.

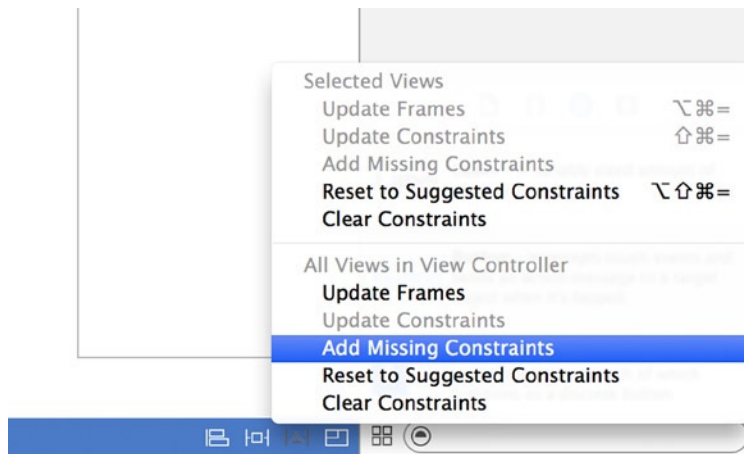


Figure 3-24. Adding missing constraints to the view to get all elements lined up nicely

The last thing you need to do is link the objects from Interface Builder into the view controller code as outlets and actions. These next steps reuse knowledge and skills you picked up in this chapter and Chapter 2. So, as a challenge, I'll present the steps for you without any visual aids. You can see how everything matches up at the end:

1. Open the Assistant Editor. You should see `ViewController.swift` displayed in the code editor portion.
2. Select the text field. Holding down the `Ctrl` key, click and drag a connection to the class file, positioning it just below `class ViewController: UIViewController`. Create an outlet named `textToSendField`.
3. Perform a similar action on the Show Me button. `Ctrl`-drag another connection to the class file, positioning it below the outlet you just created; but this time when you release the mouse button, specify that you're creating an action, not an outlet, and name it `showMe`.

That's it for the first view for now. But before you move on, check that the code in your `ViewController.swift` file is the same as the following:

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var textToSendField: UITextField!

    @IBAction func showMe(sender: AnyObject) {

    }

    override func viewDidLoad() {
        super.viewDidLoad()

        // Do any additional setup after loading the view.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }

}
```

You've covered a lot of ground so far in this chapter, so let's take a look at the application in action. Run the application in the Simulator, as you were shown in Chapter 2; the output should resemble Figure 3-25.

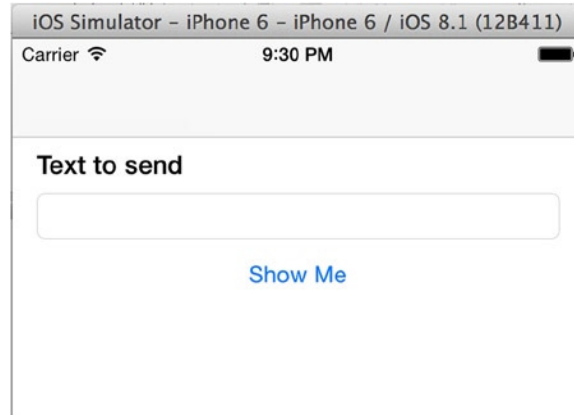


Figure 3-25. The application running in the Simulator

At this point, if your app runs successfully and you're staring at the result of your hard work, give yourself a pat on the back. You've built the first part of the application; now it's time to configure the second view controller and bring the two together using a linkage called a *segue*.

Configuring the Second View Controller

Hopefully you're feeling pleased with what you've done so far. You should be—but the application isn't finished. The idea behind this application is to type some text in the text field and have it display on another view controller when you click the Show Me button. Using storyboards to build an application comes into its own when you're working with multiple views. First you need to create a linkage called a *segue* between the Show Me button and what will be the Message view controller; by creating this linkage, you can move between view controllers without writing a single line of code. Next you need to create the interface and, finally, the underlying code to tie it all together.

Now that you know what you're aiming for, let's get started. Again, let's see how much you remember from creating the previous view controller:

1. Switch back to the Standard Editor by clicking the button to the left of the Assistant Editor on the toolbar.
2. Position the storyboard so that you can see both view controllers.
3. Click the Show Me Button, and then Ctrl-drag a connection from it to the view controller on the right, as shown in Figure 3-26.



Figure 3-26. Making a connection from the Show Me Button to the second view controller

4. When you release the mouse button, a dialog appears with a number of options in it; these are the different types of segues available. Select Show, as shown in Figure 3-27.

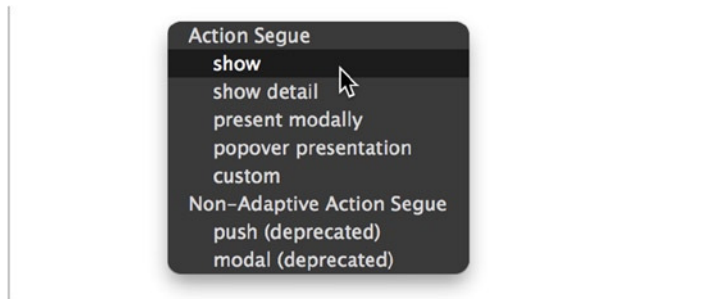


Figure 3-27. Selecting the Show segue type

Note I explain the different types of segues when you take a closer look at storyboards in Chapter 7.

You've created a relationship between the button and the second view controller. To see what this means, run the application again. Note that when you click the button, the second view slides in nicely, and there is a Back button to take you back to the initial view, but it's all a bit simplistic at the moment. Let's finish configuring the second view controller:

5. Stop the Simulator, and then align the storyboard so that you can see the second view controller.
6. Drag two labels onto the view, one below the other, and position them near the top of the view under the navigation bar placeholder. Double-click the first label, and set its text to **You Said....** Then resize the second label so that it's the width of the view.

When you created `MessageViewController.swift`, it was so that you could interact with the visual part of the view controller. So, the next step is to link this view controller to the custom `MessageViewController` class, using the Identity Inspector.

7. Click the bar at the very top of the right view controller, and then open the Identity Inspector, just to the left of the Attribute Inspector. Change Class to `MessageViewController`, as shown in Figure 3-28.

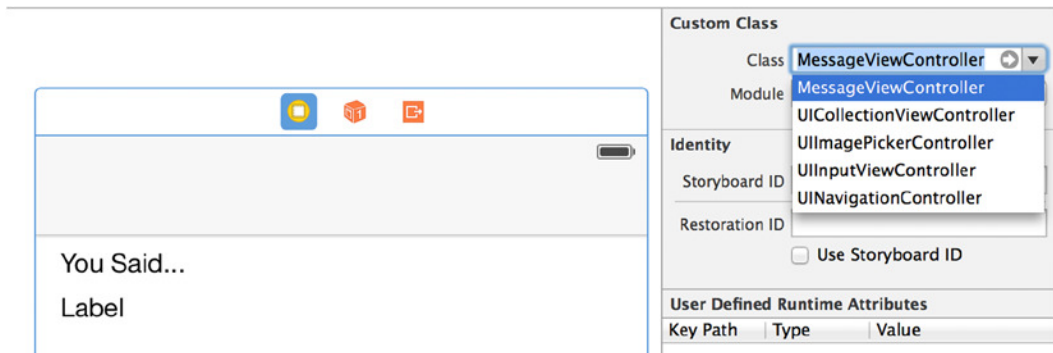


Figure 3-28. Selecting the `MessageViewController` class for the second view controller

8. Now that you've created this relationship, turn on the Assistant Editor and, with `MessageViewController.swift` selected, Ctrl-drag a connection from the bottom label to below the line starting `class MessageViewController` and create an outlet named `messageLabel`.
9. Click the view and then add the constraints required for a flexible layout by clicking the Resolve Auto Layout Issues button and clicking Add Missing Constraints.

Hopefully, your view should now resemble Figure 3-29.

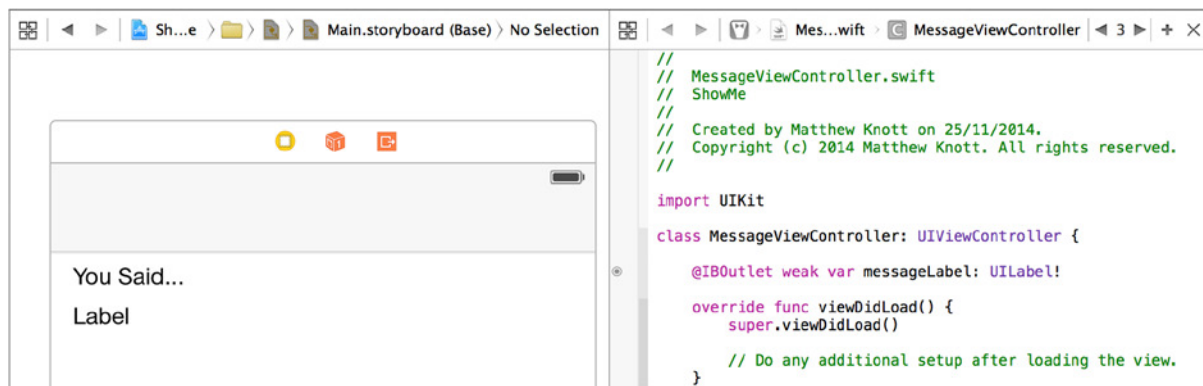


Figure 3-29. The complete, very simple layout for the second view controller, called `MessageViewController`

Before I go any further, let's think about what the objective is. The user should be able to push the Show Me button on the ViewController and have whatever they have written in the text field appear in the MessageViewController. To make this happen, you need to add some code to the MessageViewController class file so it can receive the message from the MainViewController.

Note When a new view is loaded on to the screen, replacing another one in an iOS application, this is referred to as *pushing* a view.

The MainViewController will interface with the MessageViewController using a custom initializer that will accept the text passed from the ViewController.

1. Start by switching back to the Standard editor and opening MessageViewController.swift from the Project Navigator.
2. Create a variable to hold the message data to be displayed. The data coming from the first view controller is text, so create a String variable by adding the following highlighted code just below your outlet:

```
class MessageViewController: UIViewController {
    @IBOutlet weak var messageLabel: UILabel!

    var messageData: String?
```

3. Next, you need to take the supplied text and display it with the Label, you do this by setting the text property of the messageLabel object. Add the highlighted code below to the viewDidLoad method:

```
override func viewDidLoad() {
    super.viewDidLoad()

    // Do any additional setup after loading the view.
    messageLabel.text = messageData
}
```

4. Finally, scroll down to the bottom of this file and locate the code shown below, a function called prepareForSegue. Highlight the function and copy it with Edit ► Copy (⌘+C); you're going to need this in just a minute.

```
// MARK: - Navigation

// In a storyboard-based application, you will often want to do a little preparation before navigation
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    // Get the new view controller using segue.destinationViewController.
    // Pass the selected object to the new view controller.
}
```

Now all that remains is to complete the code to send the data to the `MessageViewController`.

1. Open `ViewController.swift`, from the Project Navigator.
2. Underneath the `didReceiveMemoryWarning` function, paste in the code you copied from the other view controller using Edit ► Paste (⌘+V) so that the bottom of your code file looks like this:

```
override func didReceiveMemoryWarning() {
    super.didReceiveMemoryWarning()
    // Dispose of any resources that can be recreated.
}

// MARK: - Navigation
// In a storyboard-based application, you will often want to do a little preparation before navigation
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    // Get the new view controller using segue.destinationViewController.
    // Pass the selected object to the new view controller.
}
```

3. The `prepareForSegue` method is called just before a Segue linking views is called. Normally, you would name your Segue's and control the actions based on the Segue that has been triggered, but in this instance, you only have one Segue, so when this method is called, you know that you that the destination of the Segue is the `MessageViewController`, so you're going to create an instance of `MessageViewController` based on the Segue's destination controller, and then set the `messageData` String with the text of your text field. Add the following highlighted code into the `prepareForSegue` method, removing the two comments:

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    let messageController = segue.destinationViewController as MessageViewController
    messageController.messageData = textToSendField.text?
}
```

Xcode automatically understands that you have a `MessageViewController` class without having to make any specific references to it in your code file, and when you type `messageController.` on the second line of the method, it knows you have an object called `messageData` ready and waiting for you to pass it the contents of the text field.

That's it, you should now be able to run the application and find that you can click in the text field, type a message, and click Show Me, which takes you to the `MessageViewController` and shows whatever you typed, as shown in Figure 3-30

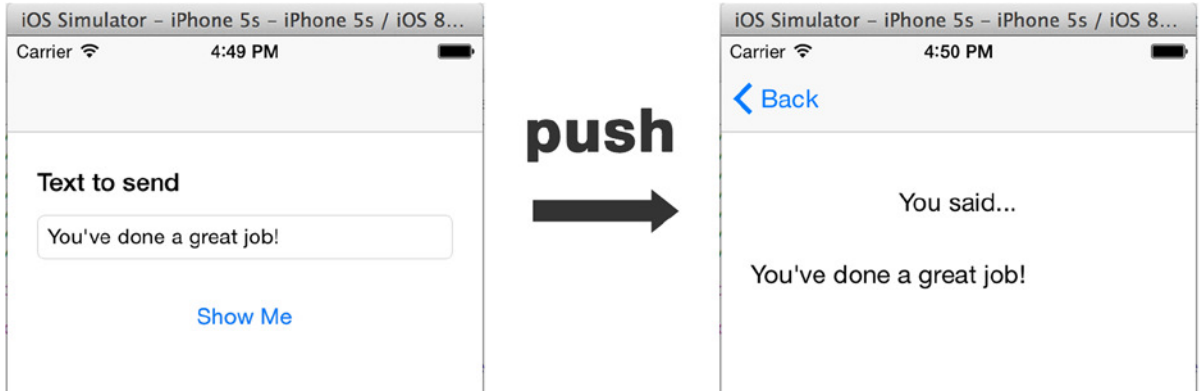


Figure 3-30. The finished application

Debugging Area

The final focus is the debugging area. In order to actually see this in action, you need to add some code to the project that captures the text that was entered when the button is clicked, and you use `NSLog` to add a message to the console. Go to `ViewController.swift` and go to the `showMe` action that you've left empty so far. Add the following highlighted code:

```
@IBAction func showMe(sender: AnyObject) {
    NSLog("User Wrote: %@", textToSendField.text)
}
```

The debug area allows you to pin down any issues with your program. Because the debug area can quickly become very complex and can be used for a variety of different things, I'll only show you the basics for now and revisit it later in the book. Now run your application and try to click the Show Me button without adding any text to the text field. You should see that when you click the button, a message is added to the output console, as shown in Figure 3-31.



Figure 3-31. The result of the `NSLog` method call is displayed in the output console

For now all you need to know is that the debugger included with Xcode is the LLVM-GCC debugger, this means you can debug a variety of code in a variety of languages. This is especially useful as Swift can use frameworks and libraries written in Objective-C. As a nice bonus, you've also added some basic input validation to your application to help prevent the user making mistakes.

Summary

This chapter covered two rather contrasting topics: project templates and the basics of Xcode's interface. The purpose of doing so was, first, to give you the confidence to start an Xcode project and choose correctly a project template suitable for your projects and, second, to drop you in at the deep end to understand how to start with nothing and build a working application, while providing a basic familiarization with the key parts of Xcode's interface.

More specifically, in this chapter, you have:

- Created an application with multiple views
- Passed an object from one view controller to another
- Learned about Segues and the hugely important `prepareForSegue` method
- Looked at each of Xcode's default project templates and when you might use them
- Explored key parts of Xcode's interface

The next chapter will look at Xcode's graphical Interface Builder in greater detail and learn how to use this to build interfaces efficiently.

Building Interfaces

A lot was covered in Chapter 3: you looked at each of Xcode’s default iOS project templates, examined when exactly you should use them, and then took a tour of the main areas of Xcode’s workspace. And if that wasn’t enough, you created an application with multiple views that could share information between the views.

The focus of this chapter is to delve deeper into Xcode’s graphical interface design tool, Interface Builder. Interface Builder has always been a key part of the Xcode set of development tools. However, with the release of Xcode 4, Interface Builder became part of Xcode itself, as opposed to previous versions in which it was a separate application. As already discussed in previous chapters, what makes Interface Builder an attractive addition to Apple’s developer tools is that it removes the need to write code in order to design great interfaces for your applications. It allows you to lay out your views and windows by dragging built-in Cocoa objects from the Object Library and placing them on the screen.

What’s even more useful is that by using the Attributes Inspector, you can make many changes that would otherwise require lines upon lines of code. As a developer, this is good news for two reasons. First, you don’t have to continuously test, build, and run your application in order to see if what you’re designing with code looks good. With Interface Builder, you can see this right away. Second, similar to what’s just been mentioned, you can make changes graphically, which saves a lot of time and effort. All this—plus using Interface Builder makes designing views fun!

This chapter explains how to set up an application using the Tabbed Application template. The great thing about using a Tabbed Application is that each of the tabs can act as an app within an app, each one showing a drastically different set of tools and styles. The two initial tabs you set up will showcase some of the interface elements you haven’t seen yet, as well as use your device’s GPS function. Once you’ve done this, you set up a third tab that will allow you to demonstrate some of the important interface elements that can’t be added using Interface Builder. A goal of this chapter is to show how much you can achieve while using as little code as possible, and it’s important to note how little code this example requires compared to how much you would need to write if Interface Builder were not a part of Xcode. The last thing you look at is how you can alter interface elements with code to achieve results that Interface Builder alone can’t do but that are

important in building beautiful, easy-to-use interfaces. Following is an outline of what each of the tabs will include:

- *Track It*: Here you create a text view that displays detailed telemetry from the GPS receiver, on either a physical device or in the simulator. You also use a switch to turn the GPS on and off.
- *Slide It*: As the name implies, in the second tab you look at how to implement the slider tab, where to build a series of sliders to alter the background color of the entire view, and how to output their values into text fields. You also learn the answer to one of the burning questions all iOS developers ask: “How do I dismiss the keyboard?”
- *Alert*: In the final tab, you see how to use a segmented control to determine what happens when a button is pushed. The choice is between an alert view and an action sheet, two popular elements in many applications.

Getting Ready

Now that you’ve had a sneak peek at the aim of this chapter, hopefully you’re raring to go. Let’s get to it:

1. Open Xcode, and create a new project by clicking Create A New Xcode Project on the Welcome screen or choosing File ► New ► Project (⌘+Shift+N). Select the Tabbed Application template, and click Next.
2. Name your project *Showcase*, and ensure that the device is set to iPhone. Configure the other settings as you did in previous applications. Make sure the key settings match those in Figure 4-1, and click Next.

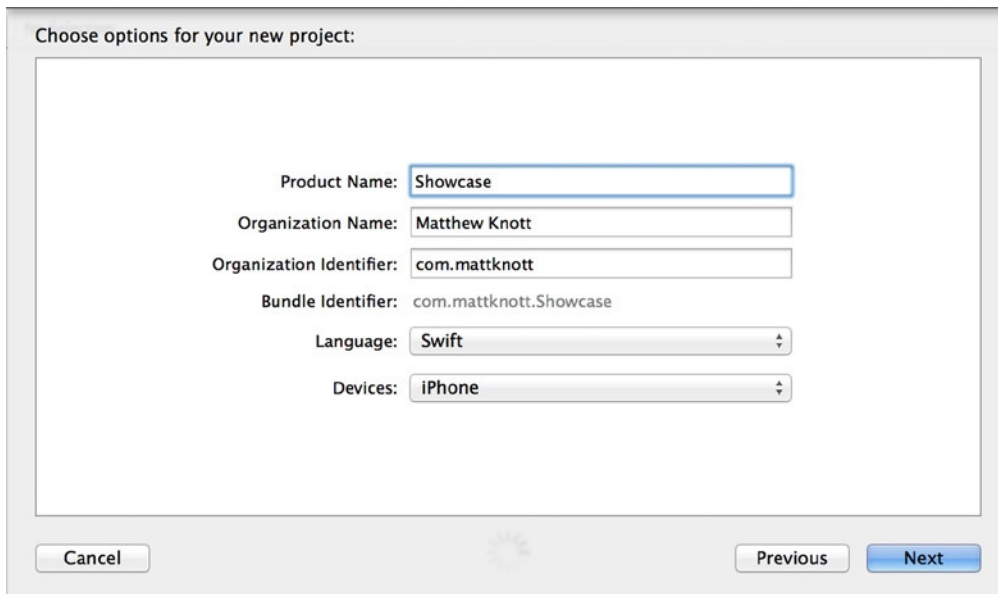


Figure 4-1. Configuring the Showcase application

3. You don't want to create a Git repository, so leave the Source Control option unchecked. Ensure that your project is going to be saved where you want it to be, and click Create.

You've now created the bare bones of your Showcase tabbed application. To see what Apple's template has given you as a starting point, click `Main.Storyboard`; you should see a screen resembling that shown in Figure 4-2.

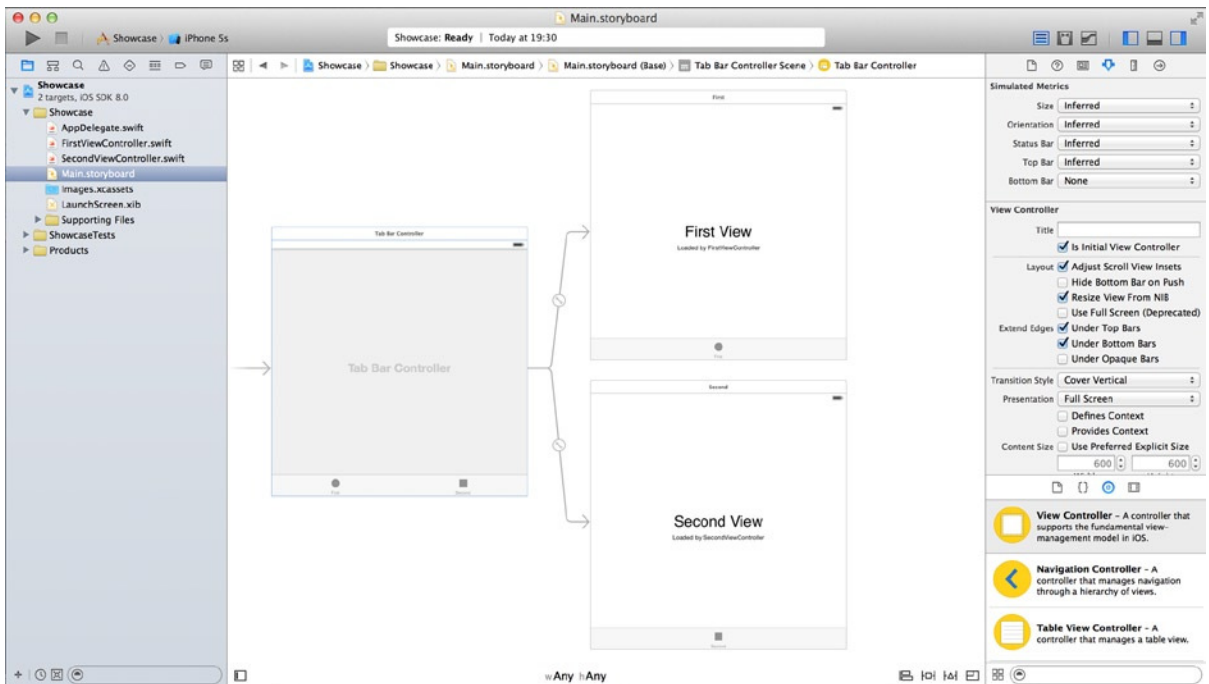


Figure 4-2. The starting point for the Showcase application

By default, the template gives you a `UITabBarController` with two `UIViewController`s attached named `FirstViewController` and `SecondViewController`. Although these names are perfectly good, tab orders can change as a project develops, so it's always better to use names that are semantically accurate. So, before you add a third tab, let's rename the files to something more appropriate.

With the Project Navigator open (+1), highlight `FirstViewController.swift` and press Return on your keyboard. You should now be able to rename the file. Remove the text, and type `TrackViewController.swift` (remember to add the `.swift` extension). Repeat this for `SecondViewController.swift`, but call it `SliderViewController.swift`. Your Project Navigator should closely resemble what you see in Figure 4-3.

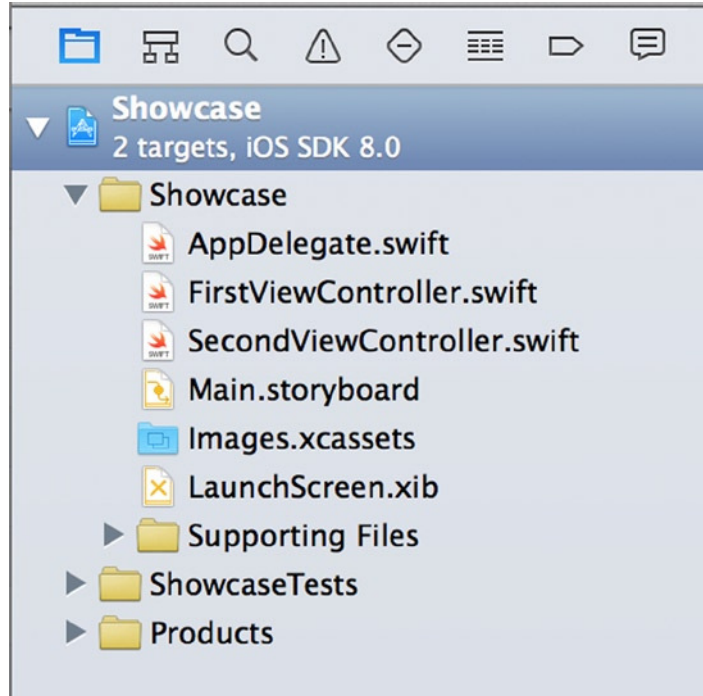


Figure 4-3. The current project's files as shown in the Project Navigator

Next you need to update your code files to use these new file names, and to do this you use the Search Navigator ($\mathbb{H}+3$). You need to set up the Search Navigator to rename every instance of `FirstViewController` to `TrackViewController`. By default, you see `Find > Text > Containing` above the search criteria. Click the word `Find`, and select `Replace > Text > Containing`, as shown in Figure 4-4.

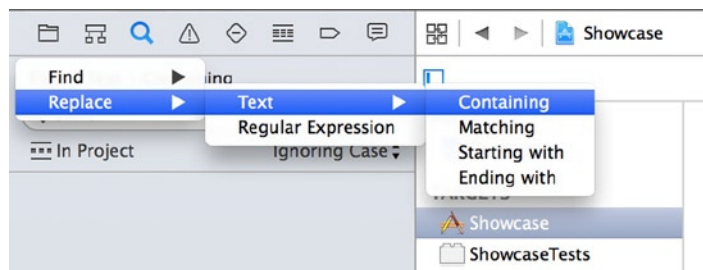


Figure 4-4. Configuring the Search Navigator to perform a find-and-replace task

In the first text field, type `FirstViewController`; and in the second, type `TrackViewController`. At this point you encounter an uncharacteristically poor piece of interface design: you need to press `Return` to perform the search, although Xcode doesn't make this clear. Click `Replace All`, and Xcode will go through all the files listed and replace the word `FirstViewController` with `TrackViewController`. Figure 4-5 illustrates the results of this find-and-replace operation in the Search Navigator.



Figure 4-5. The Search Navigator updating references to old file and class names

Note When performing a batch find-and-replace operation, Xcode may ask if you want to enable automatic snapshots. It's recommended that you enable this, because developers often perform this step in error. Having a snapshot means you can roll back in the event of a catastrophic mistake in your application. I'll cover this in Chapter 14.

Once the find-and-replace operation has completed, repeat the task, but in the first box enter `SecondViewController` as the text you're searching for and in the second type `SliderViewController` as the text you want to replace it with. Press Return, and then click Replace All.

You've now updated all references to your renamed view controllers. Next you create a third view controller called `ActionViewController`:

1. Switch back to the Project Navigator from the Search Navigator so you're ready to start interacting with the project files again.
2. Create a new file ($\mathbb{K}+N$). Select Source from the left sidebar under iOS, and then choose Cocoa Touch Class, as you did in Chapter 3. Click Next.

3. Specify `ActionViewController` as the class name. Type `UIViewController` in the Subclass Of field, and ensure that Also Create XIB File is *not* checked. Click Next. Create this file in the same location where all your other files are stored, and click Create.

Adding Tab Bar Icons to an Asset Catalog

Since they were introduced in Xcode 5, Asset Catalogs have been used to store the icons that appear on tabs in the Tabbed Application template. Although this isn't the way you *have* to store the images, it's certainly best practice, and it makes storing retina and regular versions of the same icon much easier and less cluttered. Chapter 2 briefly explained the Asset Catalogs, but here I go into the topic a little deeper.

For this project, I have created three purpose-built icons that are available in the Chapter 4 source code available from the Apress web site. If you don't want to create your own icons, a fantastic range of free tab-bar icons created by Charlene are available for download at www.iconbeast.com. Once you've downloaded the icons, you're ready to begin working with the Asset Catalog in this project:

1. Head back to Xcode, and select `Images.xcassets`. You should see the three images shown in Figure 4-6.

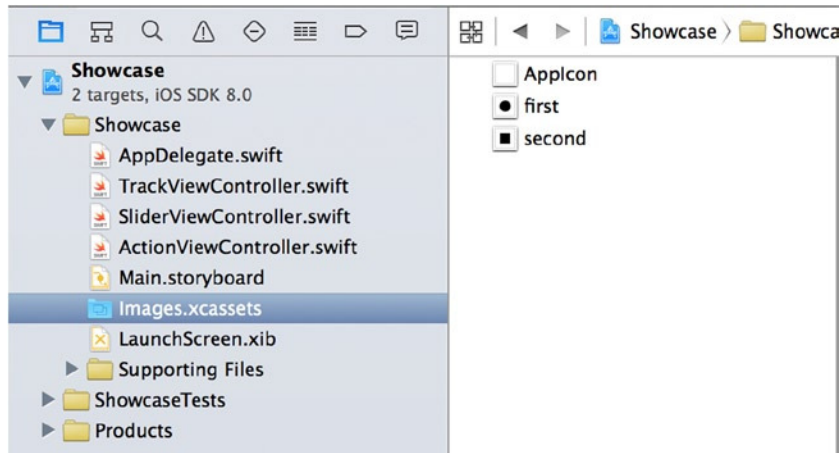


Figure 4-6. The contents of the `Images.xcassets` library

2. Select the image named `first`, and delete it by pressing the Backspace key or right-clicking and selecting Remove Selected Items. Repeat this step for the image named `second`.
3. Click the plus symbol at the bottom of the list of images, and from the menu that appears select New Image Set, as shown in Figure 4-7. This creates a new image set called `image`.

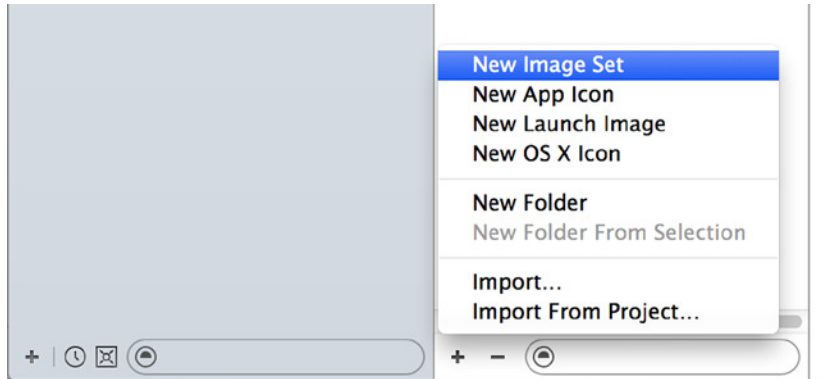


Figure 4-7. Creating a new image set

4. Select the new image set, and press Return so that you can rename the file; rename it *Track*. Repeat step 3 twice to create two more new image sets, and name them *Slider* and *Alert*, respectively. You've now created three image sets, which will contain the tab-bar icons for the three tabs.
5. Open a Finder window, and browse to where you extracted the icons from the book's source code.
6. Select the *Track* image set. You see something resembling the screens in [Figure 4-8](#).

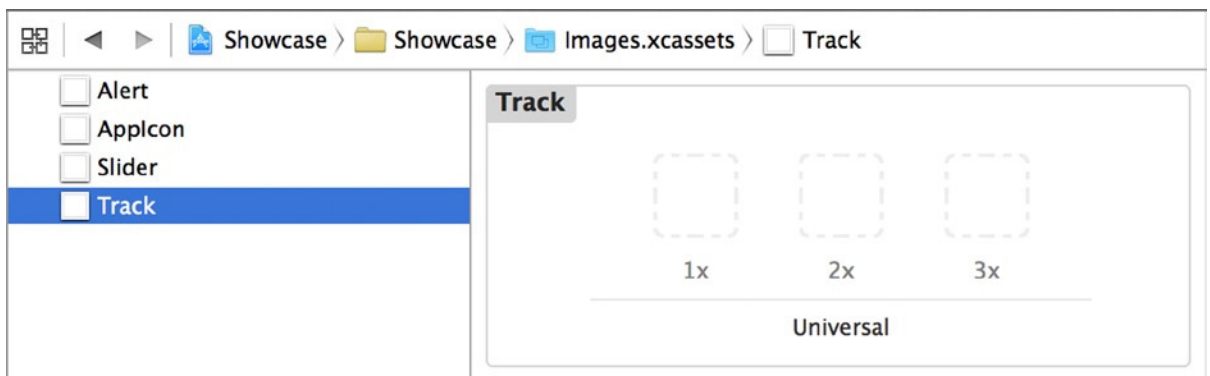


Figure 4-8. The three new image sets

One of the benefits of using Asset Catalogs for storing images is that they make it easy to group images that have different resolutions. In this case, you have a 1x container and a 2x container. Traditionally, in the 1x container you would place the standard-resolution image, and in the 2x container the retina, or higher-resolution image. Unlike with the *AppIcon* image set, there are fewer restrictions on image size here; but as a guideline for tab-bar icons, use 30px × 30px for

standard-resolution icons, 60px × 60px for 2x retina icons, and 90px × 90px for 3x retina icons (used for the new iPhone 6), which is what the icons are currently set to.

7. In the Finder window, locate the icon named `mapicon.png`, and drag it into the 1x container. Then drag the `mapicon@2x.png` file into the 2x container and `mapicon@3x.png` into the 3x container.
8. Repeat step 7 for the two remaining image sets, dragging `slidericon.png`, `slidericon@2x.png`, and `slidericon@3x.png` into the containers for the Slider image set, and `alerticon.png`, `alerticon@2x.png` and `alerticon@3x.png` into the Alert image set. Your Asset Catalog should now resemble that shown in Figure 4-9.

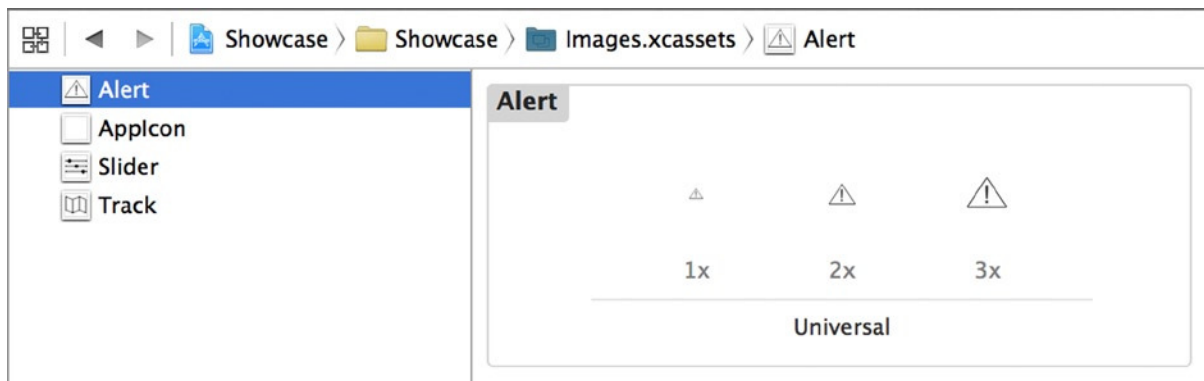


Figure 4-9. The three image sets with the icons in place

A great benefit of using Asset Catalogs for managing images is that you're left with a much neater project in the Project Navigator! Now you're set up and can begin working on your interfaces using Interface Builder, let's start by taking a closer look at the different areas of the Interface Builder.

Before You Start ...

Notice that, unlike in Chapter 3, your project has no `.xib` files; instead, you work with `Main.storyboard`. Storyboards were discussed briefly in Chapter 2, and I explain them in increasing detail as I progress through the book; but let's stop for a minute and take a closer look at what is involved in a storyboard.

Storyboards are a relatively new feature of Xcode. They allow you to logically lay out how views are pushed and managed as a user navigates through your application. They can greatly simplify applications, plus they add a degree of logic to how you develop your projects.

Because you work with them a little in this chapter, it's important to know the basics because, after all, they're part of Interface Builder. Open `Main.storyboard`, and let's take a look at the key controls. Conveniently, all the controls for your storyboard are located at the bottom of the storyboard design area and are separated into three groups, as shown in Figure 4-10.

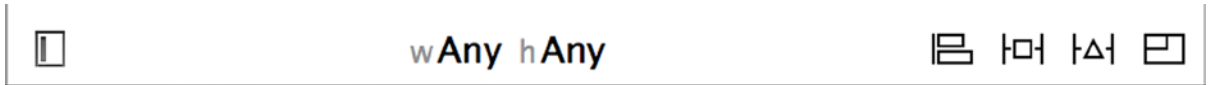


Figure 4-10. The storyboard controls

Let's look at the groups and their icons:

- *Document Outline toggle*: This first button, located by itself in the bottom-left corner of the design area, hides and displays the document outline, which I cover in more detail in Chapter 7 and throughout the book.
- *Form Factor toggle*: Located in the middle of the design area icons, this control allows you to alter how view controllers in the storyboard are displayed. This is incredibly useful if you're designing a single interface for multiple form factors, because you can quickly move between orientations and sizes and have that reflected for the entire storyboard.
- *Constraint controls*: These are described in greater detail in Chapter 6. For now, these four buttons, which are grouped together, let you control the behavior of the elements in your view when faced with differing resolutions or different screen resolutions:
 - *Align*: Allows you to position elements in relation to the view, letting you set a range of alignments including centering and aligning to the top of the view.
 - *Pin*: Fixes an element in place by manually setting its constraints.
 - *Resolve Auto Layout Issues*: One of the most useful buttons in Xcode 6. You can often use the powerful auto-layout functions offered from this menu to do all the hard work for you.
 - *Resizing Behavior*: Controls how views handle resizing. You can choose from one, none, or both of the options available.

If you have used an older version of Xcode, you may have noticed that the zoom controls have been removed in Xcode 6. This is in part because many developers handle zoom by using the pinch and squeeze gestures on a multitouch-enabled device or by double-clicking the whitespace around the views to snap in or out of the storyboard.

This concludes our brief look at the storyboard design area controls. Chapters 6 and 7 examine all of these in intimate detail, but for now let's move on and build the interface.

Building the Interface

Now that you're familiar with the storyboard controls, you can start to get your interface in order. To do that, you first remove the two views that were added by default. With `Main.storyboard` selected, you have two view controllers, as you saw in Figure 4-2. As you did in Chapter 3, you begin by removing the bulk of the Xcode-created content so that you can see for yourself exactly how the different elements are created:

1. Above each of the two view controllers is a bar. Click the bar, and three icons appear. Select the bar, as shown in Figure 4-11, and press the Backspace key to delete it.

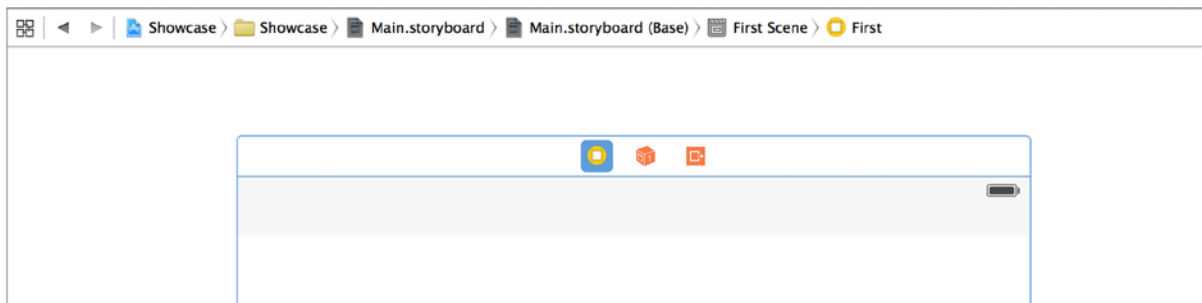


Figure 4-11. The view controller, outlined in blue after selecting the top bar

2. Repeat this step for the other view controller so you're left with only a tab bar controller.
3. To make the views easier to fit on the design area, use the `w Any h Any` control to set a compact appearance for the views: change it so that it reads `w Compact h Any`.
4. All your views are going to be based on standard view controllers, so locate the View Controller object in the Object Library and drag three of them to the design area. Position them as shown in Figure 4-12.

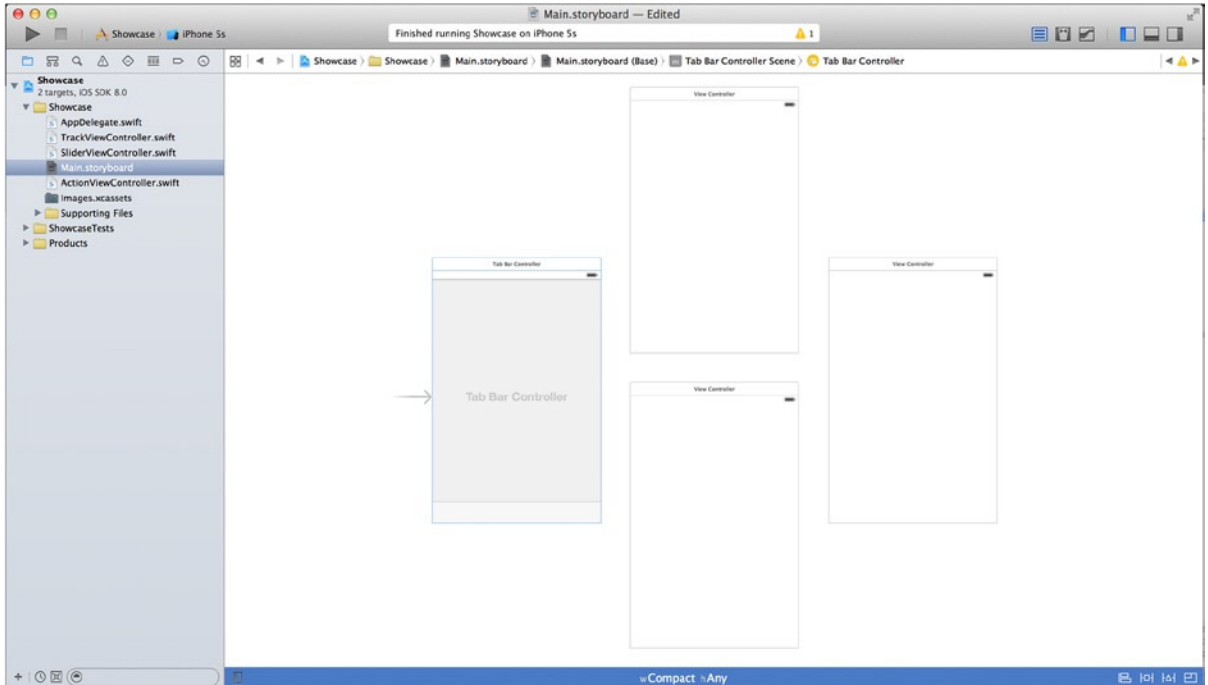


Figure 4-12. *Main.storyboard with the three orphaned view controllers. The Utilities pane can be hidden to give extra room*

Although you've added the three view controllers, they're currently orphans—that is, there is no relationship between the view controllers and the tab bar controller, so you need to create one. The process for creating a relationship between the tab bar controller and the view controllers is similar to how you connected objects to their actions and methods in Chapter 3:

1. Set the zoom level so that you can see all of the view controllers. If you're having trouble, try double-clicking some whitespace in the design area; the view should zoom out to show all three view controllers and the tab bar controller.
2. Select the tab bar controller by clicking it once.
3. Holding the control key (^), click the tab bar controller, and drag a connection to the top view controller, as shown in Figure 4-13.

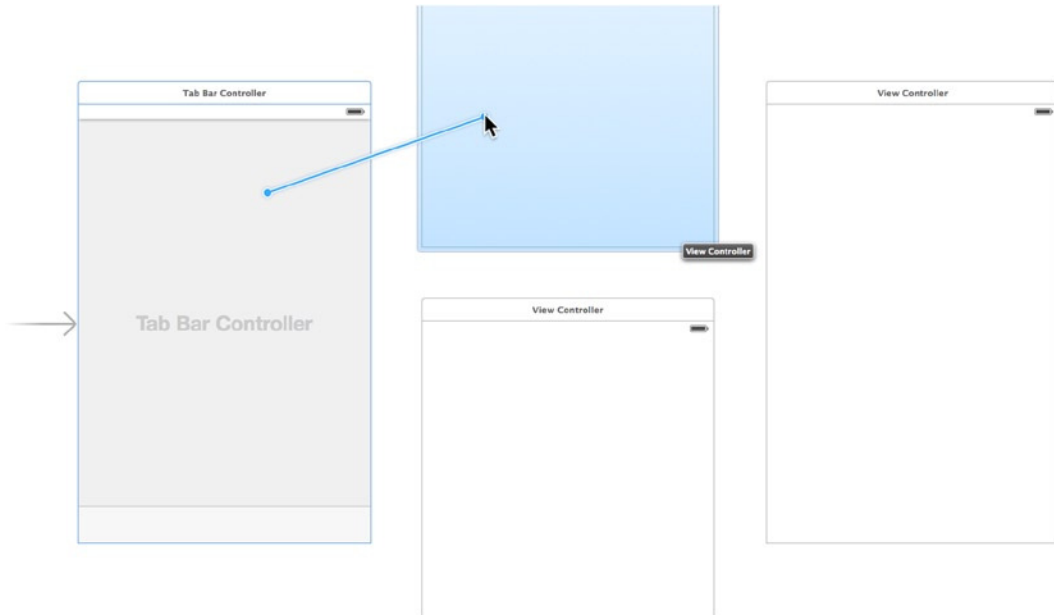


Figure 4-13. Connecting the tab bar controller to the view controller

4. When you release the mouse button, a dialog appears, asking you to choose the segue type. Chapter 7 covers segues; for now, select View Controllers under the Relationship Segue heading, as shown in Figure 4-14.

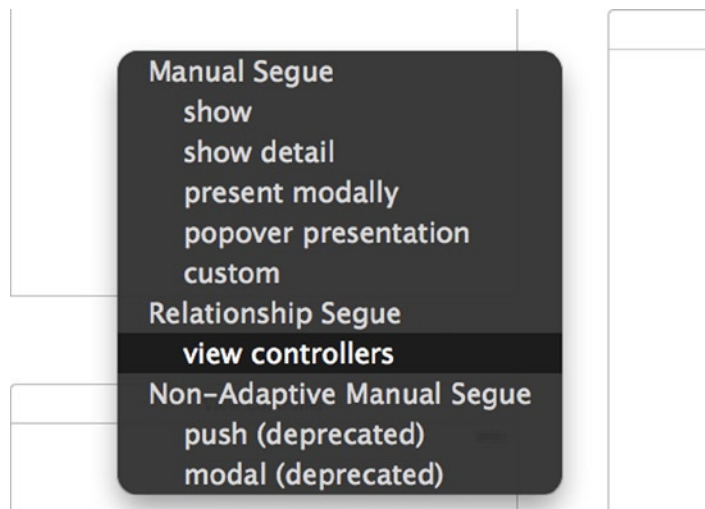


Figure 4-14. The segue selection dialog

Now that you've created a relationship between the tab bar controller and the view controller, notice that a few changes have been made in your design area. The tab bar controller has a tab showing on the tab bar, as does the view controller. Also, a line connects the tab bar controller to the top view controller; this is called a *segue*, and it's a visual representation of the relationship between two elements in a storyboard. Segues can link elements in several different ways, but on this occasion you only choose the View Controllers branch to create a relationship segue.

5. Repeat step 4 for the remaining two view controllers—first the bottom view controller, then the middle one—until you're left with something resembling the screen shown in Figure 4-15.

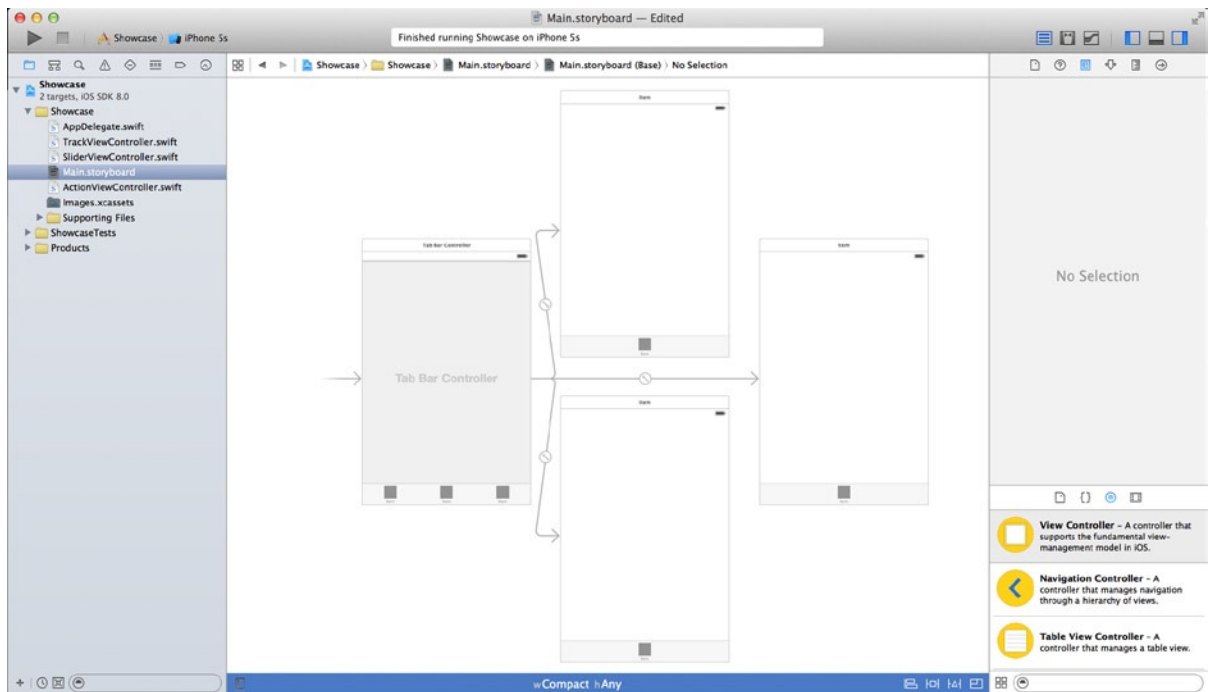


Figure 4-15. The view controllers are all connected to the tab bar controller

Setting the Tab Icons

You're nearly ready to focus on the individual views in your application, but before you do, there are a couple more tasks to complete. You need to implement the icons you added to the image Asset Catalog:

1. Zoom back in to the design area.
2. Location the topmost view controller, and select the square icon above the text Item, as shown in Figure 4-16.



Figure 4-16. *Selecting the tab bar icon in the top view controller*

3. Open the Attributes Inspector ($\text{⌘}+\text{⌘}+4$). Set the Title attribute to Track It and the Image attribute to Track. Be sure not to set the Selected Image attribute—it needs to stay blank.
4. Select the bottom view controller’s tab bar item, and set Title to Slide It and Image to Slider.
5. Select the tab bar item from the middle view controller, and set Title to Action and Image to Alert.

Using the images Asset Catalog and the storyboard, you’ve successfully named your tabs and set their icons. The tabbed application is really starting to take shape. The visual relationship between the tab bar controller and the view controllers is in place, and if you want to, you can build and run the application in the simulator—it will work fine. However, there is one other relationship you’ve yet to establish.

The three views on the design area are currently controlled by the default view controller class. But you want to use the purpose-made view controllers that you created earlier in this chapter. In Chapter 3, you created view controllers with `.xib` files, so this relationship was created for you; but now you have to do this yourself:

1. Select the top bar above the Track It view controller, as in Figure 4-11, and then open the Identity Inspector ($\text{⌘}+\text{⌘}+3$).
2. Click the drop-down list for the Class attribute, and select `TrackViewController`, as shown in Figure 4-17.

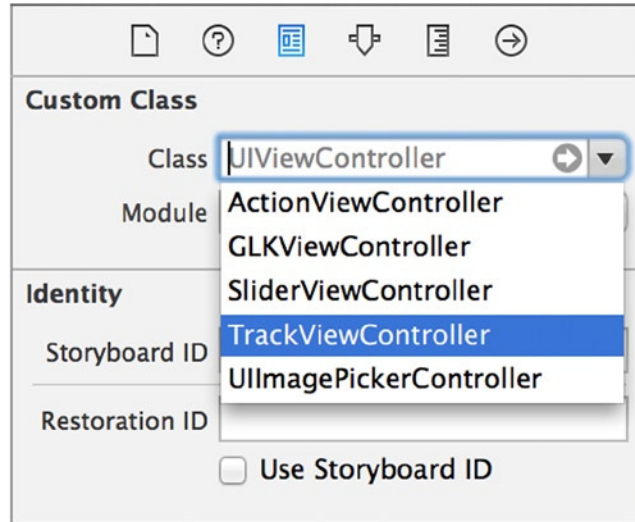


Figure 4-17. Selecting the custom view controller class

3. Select the top bar above the bottom view controller, and this time set the class to `SliderViewController`.
4. Repeat step 3 with the middle view controller, and set its class to `ActionViewController`.
5. Build and run the application using the simulator; you should find that at this stage you have three bland but working tabs.

That's it: the preparation work is complete! So far you've renamed the default view controller classes and created an extra one, created entries in the Asset Catalog and populated it with some icons, and removed the default view controllers from the storyboard and replaced them with three brand-new ones, all before setting the classes, icons, and titles of each one. You're ready to learn more about building great interfaces.

Tracking Location with the Track It Tab

For the first tab, you create a view that allows you to display detailed information about the current location, including speed, course, longitude, latitude, and positional accuracy. To do this, you will use the `CoreLocation` framework.

`CoreLocation` is used in many applications in the App Store, whether in an obvious way such as in a map-based application or in a more subtle way such as providing localized information wherever you go. The skills you learn here will give you a good grounding in applying `CoreLocation` in your own applications.

`CoreLocation` by itself isn't that useful without something to control and display its information. To do this, you add a switch control to turn positional tracking on and off and a text view to display the output information. By the end of this section, you should have created something resembling the screen shown in [Figure 4-18](#).

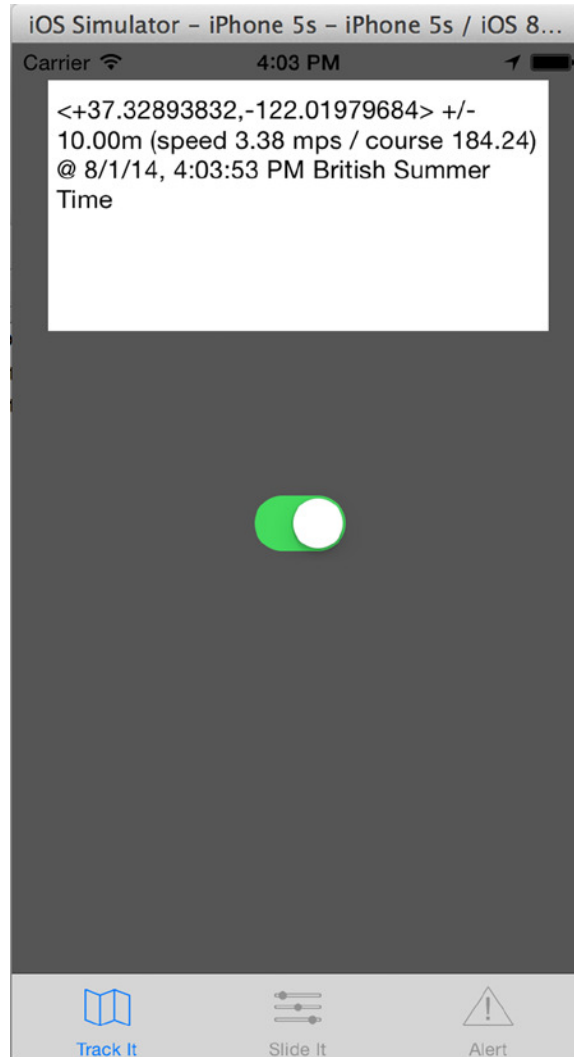


Figure 4-18. The Track It tab in action

Although you're working with a storyboard file and your views are laid out together on the design area, you can still design your views as you did in Chapter 3 with the .xib files. But you have the added bonus of being able to edit all your views in one location, without having to load up different files from the Project Navigator.

UITextView and UISwitches

To build a simple but effective view, let's use two of the most common and useful controls provided by Apple: UITextView and UISwitch. The UISwitch control (or Switch, as it appears in the Object Library) is found throughout the Settings app on your iPhone or iPad. It has on and off states, and you use it to turn tracking on and off again:

1. Search the Object Library for Switch, and drag the object to the view. Position it in the middle, as shown in Figure 4-19. Blue guidelines appear as you approach the middle of the view.

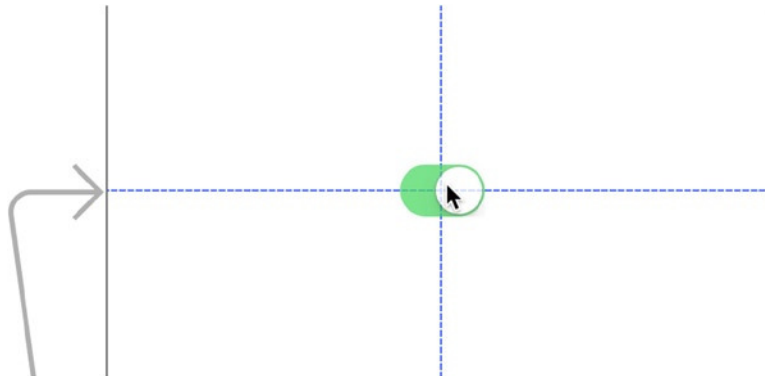


Figure 4-19. Snapping the switch into place using the guidelines

2. With the switch selected, open the Attribute Inspector and change the State attribute to Off.
3. You need to add a UITextView (or Text View, as it appears in the Object Library). A text view can contain a large amount of text; the user can type in it, like a text field, or scroll through it. Search the Object Library for Text View, and drag it so it appears just above the switch (refer back to Figure 4-18 for size and positioning reference).
4. With the text view selected, open the Attributes Inspector and remove the placeholder text from the Text attribute.
5. To change the view's background color, select a patch of whitespace on the view and then, in the Attributes Inspector, click the Background drop-down list and select Dark Gray Color from the list of prespecified colors to the right of the color indicator.
6. As you have in previous chapters, with the view still selected, bring up the Fix Auto Layout Issues menu and click Reset To Suggested Constraints under the All Views In View Controller heading.

You've added the two controls needed for this tab. Next you need to create the actions and outlets in the `TrackViewController` class file that will make them come to life and give them purpose:

1. Switch to the Assistant Editor, and ensure that you have the `TrackViewController.swift` file selected. If you have a different file open, go back to double-check whether you correctly set the view controller's class.
2. Select the text view. As in previous chapters, control-drag a connection from the text view to the class file, just below the line that says `class TrackViewController: UIViewController`.
3. You're creating an outlet named `locationText`. Type this in, and click the Connect button.
4. Repeat step 3 for the switch, this time naming the outlet `toggleSwitch`.
5. Drag another connection from the switch, this time being sure to create an action, and name it `changeToggle`.

The code of your header file should look like this:

```
import UIKit

class TrackViewController: UIViewController {

    @IBOutlet weak var locationText: UITextView!
    @IBOutlet weak var toggleSwitch: UISwitch!
    @IBAction func changeToggle(sender: AnyObject) {
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
}
```

You've created all your outlets and actions. But before you can write any code, you need to add the CoreLocation framework to the project so you can interact with the GPS features of your device.

Tip If you accidentally create an outlet instead of an action, as I often find myself doing, you may have trouble running your application after removing the erroneous line. This is because your control is still looking for that outlet. Select the control, and open the Connections Inspector. You can remove the reference to the defunct outlet there.

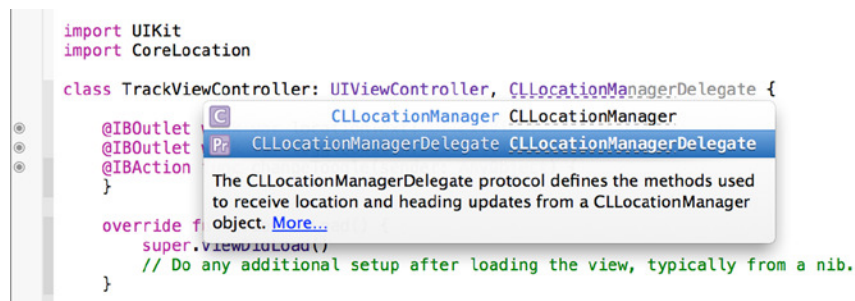
Adding Frameworks to a Project

A *framework* is a collection of classes and functions that provides additional functionality to your project. In iOS, all the GPS and location-based features are accessed through the CoreLocation framework. Some of the most commonly used frameworks are CoreData, MapKit, and CoreImage, among others.

As of iOS 7 and Xcode 5, Apple gave Objective-C developers an alternative to manually adding frameworks, called *modules*. Modules are a very simple concept: instead of going through Xcode to select a framework, physically add it to a project, and reference it in code with a `#import` statement, you simply reference it with the `@import` statement, and Xcode automatically identifies the framework and adds it into your project.

In Swift, Apple has kept this functionality and made it the default approach. Thus you never again need to add a core Apple framework manually. Chapter 9 explains modules and frameworks in more detail. As I've already mentioned, you need to add CoreLocation, and what have previously been a protracted process of locating and importing the framework is now as simple as writing a single line of code:

1. From the Project Navigator, select the `TrackViewController.swift` file, and close the Assistant Editor in favor of the Standard Editor.
2. Drop down a line from `import UIKit`, and type `import CoreLocation`. This single line makes the classes, functions, and protocols of the CoreLocation framework available to your application.
3. You need to specify that the view controller can act as a delegate for the `CLLocationManager` class. This means when the location manager is running, it knows this file contains the functions that handle certain events, such as the position of the device changing. Add `CLLocationManagerDelegate` after `class TrackViewController: UIViewController`. As shown in Figure 4-20, code completion appears, to help you complete the protocol name.



```
import UIKit
import CoreLocation

class TrackViewController: UIViewController, CLLocationManagerDelegate {
    @IBOutlet
    @IBOutlet
    @IBOutlet
    @IBAction
}

override func viewDidLoad() {
    super.viewDidLoad()
    // Do any additional setup after loading the view, typically from a nib.
}
```

The CLLocationManagerDelegate protocol defines the methods used to receive location and heading updates from a CLLocationManager object. [More...](#)

Figure 4-20. Adding the `CLLocationManagerDelegate` protocol to `TrackViewController`

- The first few lines of your `TrackViewController.swift` file should resemble the following code:

```
import UIKit
import CoreLocation

class TrackViewController: UIViewController, CLLocationManagerDelegate {

    @IBOutlet weak var locationText: UITextView!
    @IBOutlet weak var toggleSwitch: UISwitch!
    @IBAction func changeToggle(sender: AnyObject) {
    }
}
```

- You're ready to start setting up your interface in the device's GPS and location tracking technology, which is done via a class called `CLLocationManager`. Before the first `@IBOutlet`, add the following highlighted code to create an instance of the location manager class:

```
var locationManager: CLLocationManager!
@IBOutlet weak var locationText: UITextView!
```

Note You have created the instance of the location manager in what is called the global scope; this means any of your functions in the `TrackViewController` class can see and use the location manager. This is important because the location manager is the primary interface into the location functions, and you want to be efficient and consistent by having only a single instance of the class declared in your application.

Next, you need to implement the action of the toggle switch being turned on or off. The code you add here does the majority of the work in this tab. A little below the line you just added should be the `changeToggle` action. Let's go through the code step by step before you see the final code block:

- You need to determine whether the switch was turned on or off when the action is called. You do this with an `if ... else ...` statement. Add the highlighted code into the action:

```
@IBAction func changeToggle(sender: AnyObject) {
    if toggleSwitch.on {

    }
    else
    {

    }
}
```

The `.on` property of the `UISwitch` class returns either a true or a false value, depending on the switch's position. If true or on, the code in the first set of braces is executed; otherwise, if false or off, the code in the second set of parentheses is executed.

Tip In Swift, parentheses containing the conditions of an `if` statement are optional.

2. All the code you write in this view controller will be completely useless if the device's location services are disabled. To account for this, the next block of code will prevent the switch being turned on if location services are disabled. Add the highlighted code:

```
@IBAction func changeToggle(sender: AnyObject) {
    if toggleSwitch.on {
        if (CLLocationManager.locationServicesEnabled() == false) {
            self.toggleSwitch.on = false
        }
    }
    else
    {
    }
}
```

3. The next step is to check whether the `locationManager` object has been initialized and, if not, to initialize it. There are numerous ways of initializing a `CLLocationManager` object, but in this case you do four things: initialize the object, tell it that this view controller is acting as its delegate, tell it to be accurate within 10 meters, and tell it to update when it moves more than 10 meters from the last recorded position. So, drop down a line after the last statement, and add the following highlighted code:

```
@IBAction func changeToggle(sender: AnyObject) {
    if toggleSwitch.on {
        if (CLLocationManager.locationServicesEnabled() == false) {
            self.toggleSwitch.on = false
        }

        if locationManager == nil {
            locationManager = CLLocationManager()
            locationManager.delegate = self
            locationManager.distanceFilter = 10.0
            locationManager.desiredAccuracy = kCLLocationAccuracyNearestTenMeters
            locationManager.requestWhenInUseAuthorization()
        }
    }
}
```

```

    }
    else
    {

    }
}

```

4. The last thing you need to do in this half of the `if` statement is to tell the `locationManager` object to start updating the location. You do this by calling the `startUpdatingLocation` function. Once activated, it begins tracking your location; then, every time the conditions you initialized it with are met, it fires the delegate function `didUpdateLocations`, which you add later. For now, drop down a line and add the following highlighted code:

```

@IBAction func changeToggle(sender: AnyObject) {
    if(toggleSwitch.on)
    {
        if (CLLocationManager.locationServicesEnabled() == false) {
            self.toggleSwitch.on = false
        }

        if locationManager == nil {
            locationManager = CLLocationManager()
            locationManager.delegate = self
            locationManager.distanceFilter = 10.0
            locationManager.desiredAccuracy = kCLLocationAccuracyNearestTenMeters
            locationManager.requestWhenInUseAuthorization()
        }

        locationManager.startUpdatingLocation()
    }
    else
    {

    }
}

```

5. You need to code the `else` outcome, which is triggered when the switch is set to off. All you want to do in this instance is tell the `locationManager` object to stop tracking by calling the `stopUpdatingLocation` function. Complete the action by adding the highlighted code between the `else` braces:

```

@IBAction func changeToggle(sender: AnyObject) {
    if(toggleSwitch.on)
    {
        if (CLLocationManager.locationServicesEnabled() == false) {
            self.toggleSwitch.on = false
        }
    }
}

```



```

    if locationManager == nil {
        locationManager = CLLocationManager()
        locationManager.delegate = self
        locationManager.distanceFilter = 10.0
        locationManager.desiredAccuracy = kCLLocationAccuracyNearestTenMeters
        locationManager.requestWhenInUseAuthorization()
    }

    locationManager.startUpdatingLocation()
}
else
{
    if locationManager != nil {
        locationManager.stopUpdatingLocation()
    }
}
}

```

Those few lines can be used as boilerplate code any time you want to initialize a `CLLocationManager`. I mentioned the `didUpdateLocations` delegate method that the `locationManager` object looks for every time an update is triggered. It's a very simple implementation that takes the last reported location information and outputs its description value to the text view. To do this, add the following highlighted code after the `didReceiveMemoryWarning` function:

```

override func didReceiveMemoryWarning() {
    super.didReceiveMemoryWarning()
    // Dispose of any resources that can be recreated.
}

func locationManager(manager: CLLocationManager!, didUpdateLocations locations: [AnyObject]!) {
    var location:CLLocation = locations[locations.endIndex-1] as CLLocation
    self.locationText.text = location.description
}

```

You need to implement one final delegate method: the `didFailWithError` function that is called if there is a fault while trying to obtain a location and that writes the error description to the text view. It's not essential for this example, but I'm trying to give you some useful boilerplate code; plus, you should always account for and handle failures such as this. Add the highlighted function below your last delegate function:

```

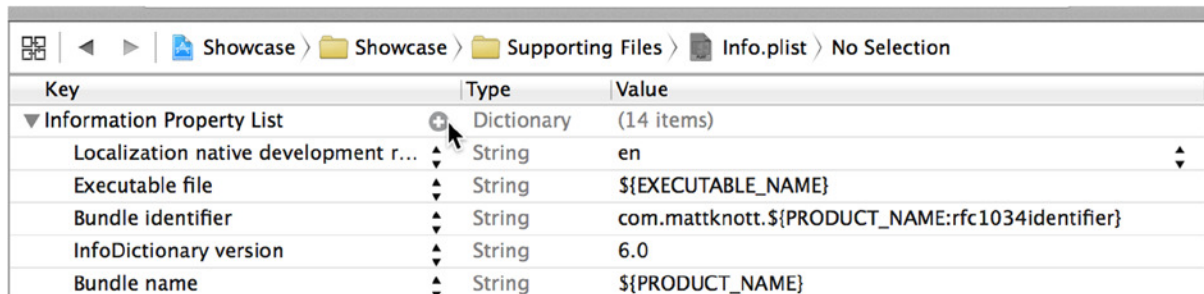
func locationManager(manager: CLLocationManager!, didUpdateLocations locations: [AnyObject]!) {
    var location:CLLocation = locations[locations.endIndex-1] as CLLocation
    self.locationText.text = location.description;
}

func locationManager(manager: CLLocationManager!, didFailWithError error: NSError!) {
    locationText.text = "failed with error \"(error.description) "
}

```

That's it—you've finished the code for the Track It tab. But before you run it, you need to do something new: add several entries to the application's `info.plist` file. For a number of frameworks and classes, Apple likes you to add a privacy declaration that explains to the user what you're doing with their location information. In iOS 8, these are mandatory, and the code won't function without them:

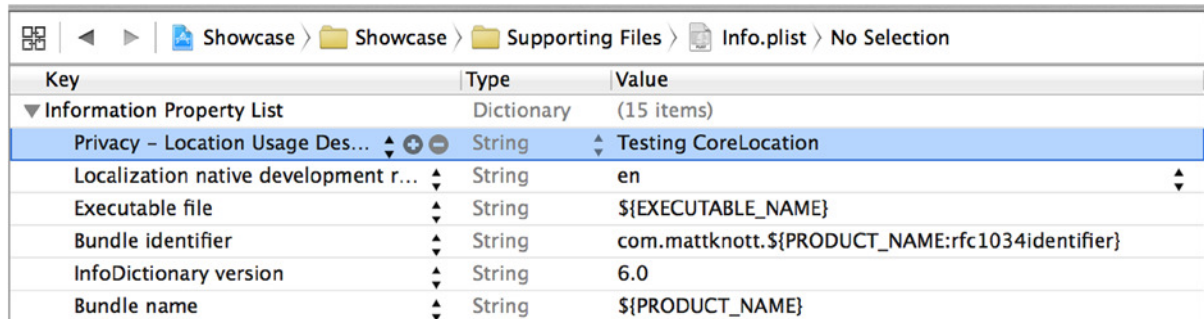
1. In the Project Navigator, expand the Supporting Files group, and select `Info.plist`.
2. Move your mouse cursor over the first line, titled Information Property List. A small plus symbol appears, as shown in Figure 4-21. Click it.



Key	Type	Value
▼ Information Property List	Dictionary	(14 items)
Localization native development r...	String	en
Executable file	String	\${EXECUTABLE_NAME}
Bundle identifier	String	com.mattknott.\${PRODUCT_NAME:rfc1034identifier}
InfoDictionary version	String	6.0
Bundle name	String	\${PRODUCT_NAME}

Figure 4-21. Adding an item to the application's `info.plist` file

3. A new row is inserted. In the list on the left, scroll until you see the item Privacy - Location Usage Description. Select it, double-click the empty Value field, and type **Testing CoreLocation** or whatever message you want to present to the user. Your finished entry should resemble that shown in Figure 4-22.



Key	Type	Value
▼ Information Property List	Dictionary	(15 items)
Privacy - Location Usage Des...	String	Testing CoreLocation
Localization native development r...	String	en
Executable file	String	\${EXECUTABLE_NAME}
Bundle identifier	String	com.mattknott.\${PRODUCT_NAME:rfc1034identifier}
InfoDictionary version	String	6.0
Bundle name	String	\${PRODUCT_NAME}

Figure 4-22. The privacy statement in the `info.plist` file

- Repeat step 2 to create another entry under Information Property List. This time, you need to type the description as well as the value. Type in `CLLocationWhenInUseUsageDescription`; then, under Value, again enter **Testing CoreLocation**.

This may seem like an unnecessary chore, but without it, not only won't your app work, but it will also be rejected by Apple if you submit it to the App Store.

With the privacy message set, the last thing to do is test it in the simulator. When you flip the switch, you should see the privacy message, as shown in Figure 4-23.

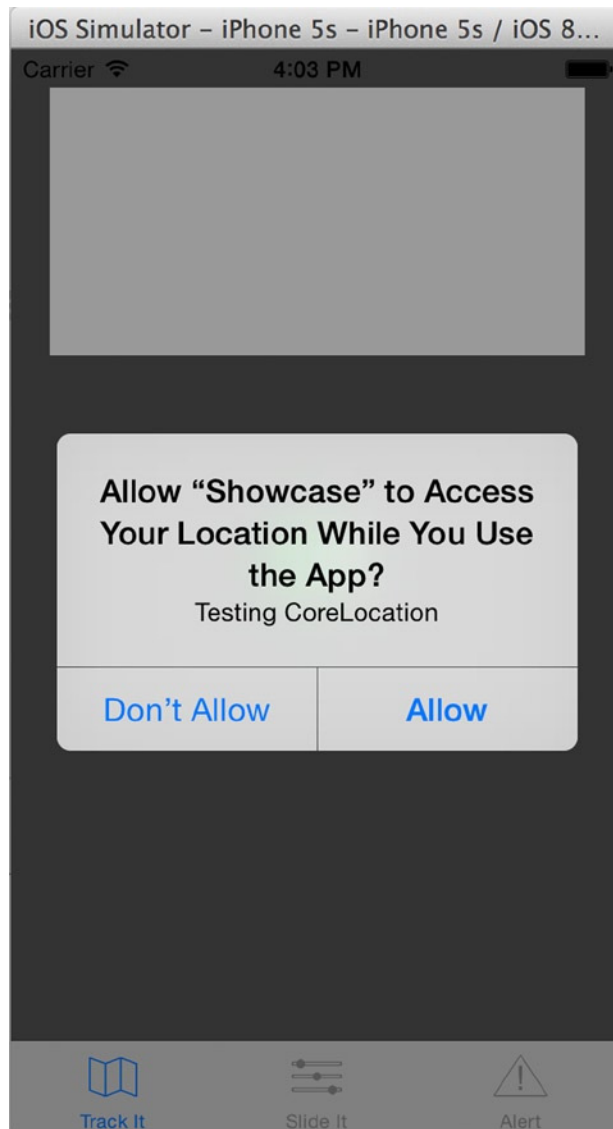


Figure 4-23. The custom privacy message being displayed to the user

Simulating a Location

When you run your application in the simulator and accept the privacy message, you may find that nothing happens. The reason for this is simple: by default, the simulator doesn't have a location, and therefore it's unable to give you any details about a location, let alone update the location as it's moving.

Note If nothing happens, you may not have Location Services enabled. Return to the home screen on your virtual device by going to Hardware ► Home, and then open the Settings app. In Privacy, select Location, and then ensure that Location Services is enabled.

Fortunately, Apple has provided some pretty nifty tools for specifying a location. It can also simulate a drive or bike ride, which is the preset you use in this case. In the simulator menu bar, select Debug ► Location ► City Bike Ride. All of a sudden your text view begins filling as a virtual bike peddles through California, near Apple headquarters in Cupertino. (Chapter 11 explains more about location debugging.)

That does it for this tab! You've created a really neat app that you could deploy to your phone while you take a run to view your location and meters run per second reflected in real time, which is pretty amazing.

Mixing Colors with the Slide It Tab

The second tab uses `UISlider` controls to create a RGB (red green blue) color mixer that alters the background color in real time and outputs the values to a series of text fields. This is another tab with real-world, practical applications. RGB is a color system that defines colors by assigning three values between 0 and 255 to each primary color. Any web developer, graphic designer, or even iOS app programmer will at some point need a tool that gives them the RGB value for a certain color. With this tab, you can play around with different combinations before implementing the one you like.

The interface for this tab is by far the most complex of the three, so let's begin. You create one block of elements for the red color and then repeat the steps two times for the green and blue colors:

1. Add a Label object from the Object Library to the Slide It view. Position it near the top of the view, and then double-click it and change the text to say Red, as shown in Figure 4-24.

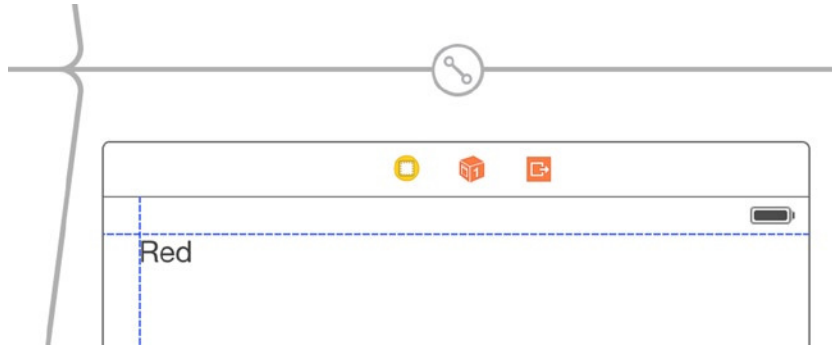


Figure 4-24. The color label in position

2. Search for Slider in the Object Library, and drag it onto the view. Position it below the label, and resize it so it fills about two-thirds of the view's width, as shown in Figure 4-25.

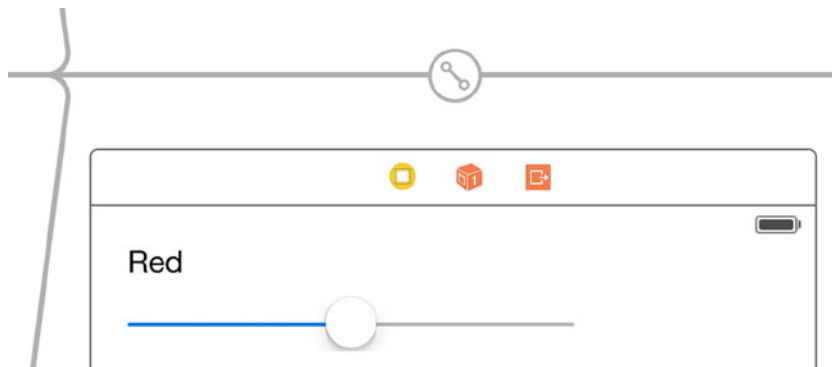


Figure 4-25. The slider added to the view and made wider

3. You want to add a text field to display the RGB value. Drag in a text field from the Object Library, and position it to the right of the slider, as shown in Figure 4-26.

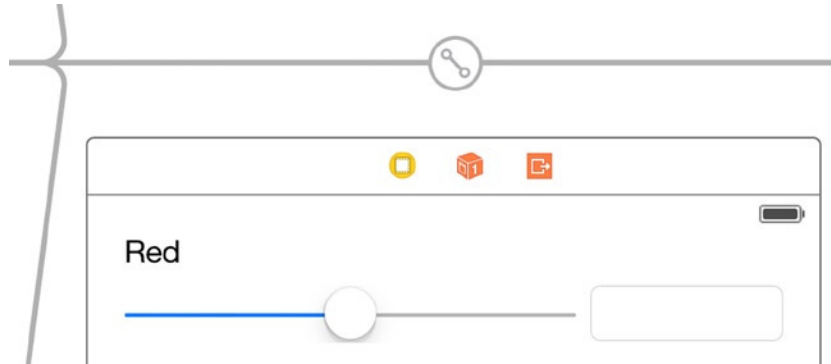


Figure 4-26. The text field added to the view and positioned to the right of the slider

- Repeat steps 1 through 3, positioning each group of elements one under the other until your view resembles that shown in Figure 4-27.

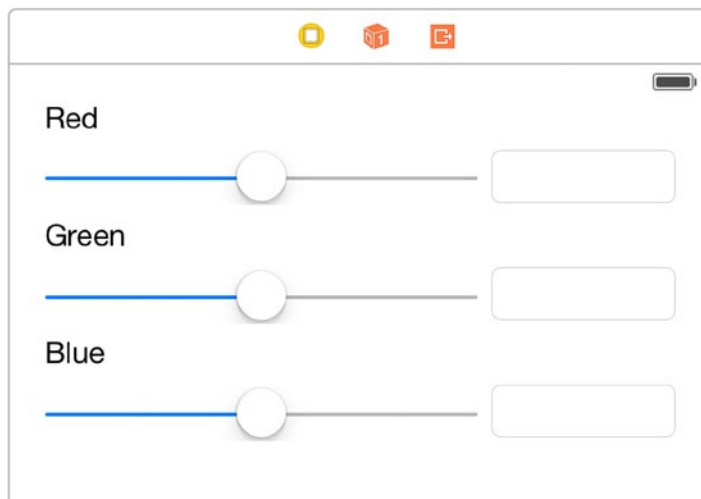


Figure 4-27. The finished interface

- To ensure that all the elements line up when you run the application, take a moment to add the constraints for the view. Select a white area of the view, click the Fix Auto Layout Issues button, and click Add Missing Constraints.
- Select the red slider, and examine its values in the Attributes Inspector. Its value range is set with 0 as a minimum value and 1 as a maximum. Your instinct might be to set the maximum to 255, the upper value of a color in the RGB format; but the technique used here expects a value between 0 and 1, so this fits your needs perfectly. You do, however, want to change the starting point for the slider to be the maximum value, so change the Current value to 1 from 0.5. Repeat this for the green and blue sliders.

This completes the interface, leaving you ready to create your outlets and actions before moving on to the code, which is very simple for this tab. As you did in the previous tab, open the Assistant Editor and ensure that it shows the `SliderViewController.swift` file:

1. Create an outlet for each of the `UISlider` controls, naming them `redSlider`, `greenSlider`, and `blueSlider`, respectively.
2. Create outlets for each of the `UITextField`s, naming them `redValue`, `greenValue`, and `blueValue`, respectively.
3. Create actions for all the `UISlider` controls, naming them `changeRed`, `changeGreen`, and `changeBlue`, respectively.
4. For reasons that I go into shortly, make your view controller a text view delegate by adding, `UITextFieldDelegate` after `class SliderViewController: UIViewController`.

Before you move on, check that the start of your code looks like this:

```
import UIKit

class SliderViewController: UIViewController, UITextFieldDelegate {

    @IBOutlet weak var redSlider: UISlider!
    @IBOutlet weak var greenSlider: UISlider!
    @IBOutlet weak var blueSlider: UISlider!
    @IBOutlet weak var redValue: UITextField!
    @IBOutlet weak var greenValue: UITextField!
    @IBOutlet weak var blueValue: UITextField!
    @IBAction func changeRed(sender: AnyObject) {
    }
    @IBAction func changeGreen(sender: AnyObject) {
    }
    @IBAction func changeBlue(sender: AnyObject) {
    }
}
```

That's it for Interface Builder for this tab. This has been one of the most complex interfaces you've encountered so far. Switch back to the Standard Editor, and open `SliderViewController.swift` from the Project Navigator:

1. As with the previous tab, you need to store the value specified by the sliders by declaring and initializing some global variables that are of `CGFloat` type. Add the following code after the line `class SliderViewController: UIViewController, UITextFieldDelegate`:

```
var redColor:CGFloat = 1.0
var greenColor:CGFloat = 1.0
var blueColor:CGFloat = 1.0
```

2. Navigate to the `viewDidLoad` function. Under the line `super.viewDidLoad()`, you need to set the `delegate` property of the text fields in order to use the `UITextViewDelegate` protocol. Add these lines:

```
self.redValue.delegate = self
self.greenValue.delegate = self
self.blueValue.delegate = self
```

3. You're going to call a function that you haven't written yet, so don't panic when Xcode doesn't help you through code completion and then adds a red exclamation mark next to this line. You call the function by adding `updateColor()` to the `viewDidLoad` function after the last code you wrote. Your completed `viewDidLoad` function should now look like this:

```
override func viewDidLoad() {
    super.viewDidLoad()

    redValue.delegate = self
    greenValue.delegate = self
    blueValue.delegate = self

    updateColor()
}
```

4. You need to write the `updateColor` function, which basically takes the red, green, and blue values and uses them to set the view's background color. Under the `viewDidLoad` function, add the following code:

```
func updateColor() {
    self.view.backgroundColor =
    UIColor(red: redColor, green: greenColor, blue: blueColor, alpha: 1.0)
}
```

In this code, you create a `UIColor` object from the red, green, and blue values. The `alpha` property controls the opacity of the background, with 1.0 being totally opaque and 0.0 being transparent.

Now you need to add the code to the three actions that are linked with their corresponding sliders: `changeRed`, `changeGreen`, and `changeBlue`. All of these actions use practically the same code—only the variable and outlet names change, depending on the color. Let's set the `changeRed` code step by step, after which you complete the remaining two methods yourself:

1. You need to get the value from this color's slider and assign it to the `redColor` float. In the action for the red slider, write `redColor = CGFloat(redSlider.value)`.
2. You want to update the text field with the correct RGB value. To do that, you need to convert the value from between 0.0 and 1.0 to between 0 and 255; so, you multiply the value of `redColor` by 255. Finally, you ensure that there are no decimal places by using the `String(format: function` and the `%.0f` placeholder, which in plain English means “put the float value

here but limit it to 0 decimal places.” The number before `f` controls the number of decimal places shown in the string. Also, in order to make the format function recognize the float, you need to convert it from a `CGFloat` to a `Float`. The code to achieve this is `redValue.text = String(format: "%.0f", Float(redColor*255.0))`.

3. A change has been made, so you need to call the `updateColor` function to make sure the change is reflected in the color set in the view’s background. The code for this is the same as in the `viewDidLoad` function, so type `updateColor()`.

The code for the finished action should look like this:

```
@IBAction func changeRed(sender: AnyObject) {
    redColor = CGFloat(redSlider.value)
    redValue.text = String(format: "%.0f", Float(redColor*255.0))
    updateColor()
}
```

Your challenge is to implement the remaining two actions by yourself. When you’re done, check that your code matches mine:

```
@IBAction func changeGreen(sender: AnyObject) {
    greenColor = CGFloat(greenSlider.value)
    greenValue.text = String(format: "%.0f", Float(greenColor*255.0))
    updateColor()
}
@IBAction func changeBlue(sender: AnyObject) {
    blueColor = CGFloat(blueSlider.value)
    blueValue.text = String(format: "%.0f", Float(blueColor*255.0))
    updateColor()
}
```

You need to write one final function to complete this tab: the `textFieldShouldReturn` method, which the text fields will look for now that they know this view controller is acting as a delegate for those text fields.

The UITextViewDelegate Implementation

Text fields are probably the most common control in an iOS app—they’re everywhere. You tap inside them, the keyboard slides in, and you add your text. It’s probably second nature to you that tapping the Return key dismisses the keyboard. Hold that thought; go ahead and run your application, and select the Slide It tab.

Play around with the sliders, and see how the background color changes as you modify the values. You’ve created something that can be usefully applied in the real world, which, as I mentioned previously, is done by giving the RGB values so they can be selected. Let’s test this. Tap in one of the text fields: as expected, the keyboard slides in. Great—now try to go to the Track It tab. Hmm, not so great: the keyboard is blocking the path, so you’re effectively stuck and must quit and relaunch the app to have any hope of accessing the other tabs.

You want to make it so that when you press Return, the keyboard dismisses itself. This is why you made your view controller take on the `UITextViewDelegate` role. By doing this, when you press Return, the text field tries to call the `textFieldShouldReturn` function; but because you haven't added this function yet, it doesn't do anything. Add the following code beneath your `viewDidLoad` function:

```
func textFieldShouldReturn(textField: UITextField!) -> Bool {
    textField.resignFirstResponder()
    return true
}
```

When you tap the text field, the text field assumes responsibility for everything that happens thereafter—in other words, it becomes the *first responder*. When this function is called, you're telling it to give up this status with the `resignFirstResponder` function before returning a Boolean value, which in this case can be either true or false (the result is the same). Rerun your application: you should find that you can dismiss the keyboard with the Return key and that you have a fully functional color slider view, as shown in Figure 4-28.

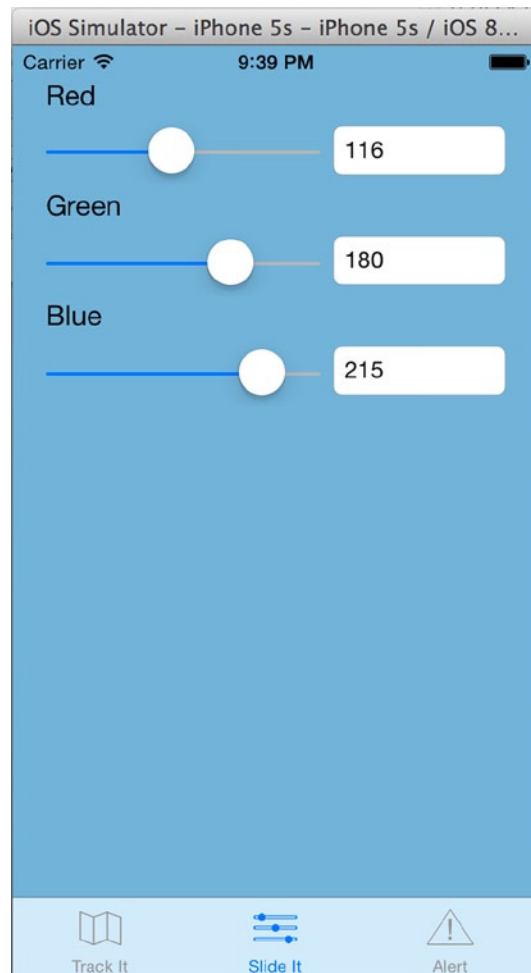


Figure 4-28. The finished Slide It tab, complete with dismissible keyboard

Note If you're running this in the simulator, as of Xcode 6 and iOS 8, the keyboard doesn't automatically show—the simulator assumes you want to use your physical keyboard. But on a device, you experience the problem of not being able to dismiss the keyboard, which is why you must always test on a physical device. To summon the keyboard in the simulator, go to Hardware ► Keyboard ► Toggle Software Keyboard (⌘+K).

Adding “Off the Menu” Controls

You've created two hugely different but incredibly cool tab views so far. For the third tab, you look at another common control you add through Interface Builder: the segmented control. In addition, you also look at two important controls that you can't add through Interface Builder: the alert view and the action sheet.

Alert Views and Action Sheets with UIAlertController

Before you begin building your interface, let's clarify what I mean when I talk about alert views and action sheets. Figure 4-29 shows how both are used in the iCloud settings area of the Settings application in iOS 8. You already encountered an alert view, when the Track It tab asked for access to your location.

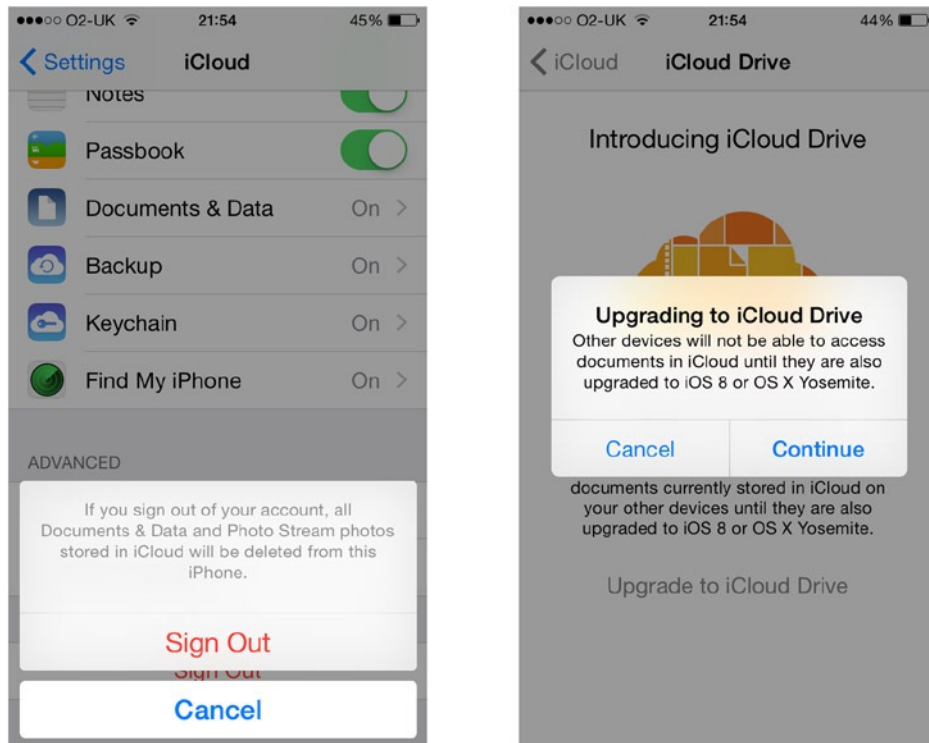


Figure 4-29. An example of an action sheet (left) and an alert view (right)

Action sheets, as their name implies, can be used to present the user with several options for a specific action. For example, when you tap the flag icon while looking at an e-mail, you're asked whether you want to Flag, Mark as Read / Unread, or Move To Junk. If you give the user the option to add account details to your application, you might use an action sheet to ask whether the user wants to add an account for your site or a third-party account, such as an OpenID account.

Alert views are coded in a way very similar to action sheets. Alert views are used to draw the user's attention to an event, such as a timer ending, or to confirm whether the user wants to activate a feature or delete some data. You'll use them often, and the good news is that they're easy to set up and use.

In iOS 8, Apple introduced a new class called `UIAlertController`, which combines the older `UIAlertView` and `UIActionSheet` classes into a single class. This is a fairly sensible move on Apple's part; the legacy classes were extremely similar.

Building the Action Tab Interface

Now that you have a clearer understanding of what alert views and action sheets do, you're ready to build the third and final tab: the Action tab. Just the middle view controller needs to be built. Adjust your storyboard so it's visible in the design area:

1. Search for Segmented Control in the Object Library, as shown in Figure 4-30. Drag it onto the view and position it in the center, at the top of the view.

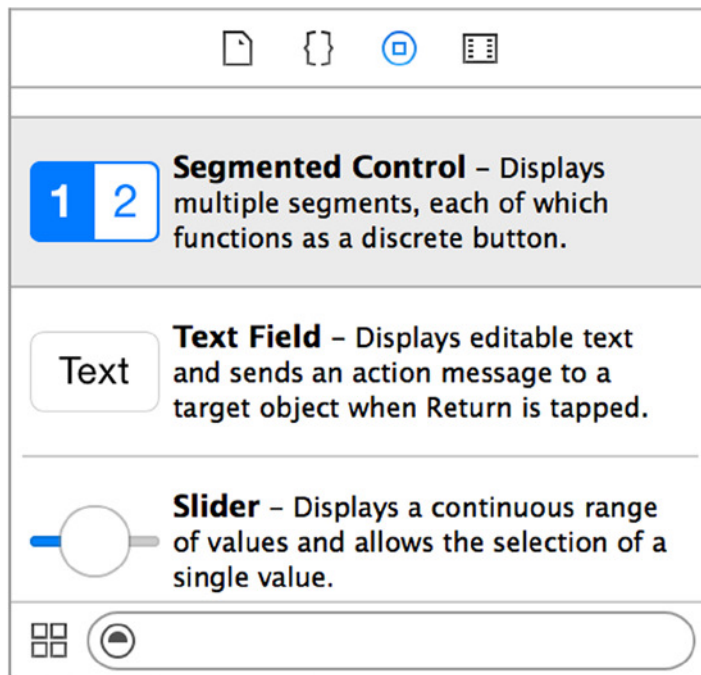


Figure 4-30. Searching for Segmented Control in the Object Library

2. You need to change the values of the segments. To do this, select the segmented control you added to the view, and open the Attributes Inspector. Change the segment Title attribute from First to Alert.
3. Changing the second segment's title isn't as obvious. Looking at the Attributes Inspector for the segmented control, notice the drop-down list above the Title attribute that you just changed. Click it, and select Segment 1- Second.
4. You can now change the Title attribute of the second segment from Second to Action Sheet, as shown in Figure 4-31.



Figure 4-31. Changing the second segment's Title attribute

5. Using the square handles on either side of the segmented control, resize it so that you can see all of the text in the second segment. Reposition it so it's centered again.
6. You want to add a button control to the view to trigger whichever option is selected. Search for Button in the Object Library, and drag it on to the view, positioning it dead center as you did the switch in the first tab.
7. Using the Attributes Inspector, change the button's Title attribute from Button to Show Me. Again, you need to reposition it to be dead center after changing the text. Your view should resemble that shown in Figure 4-32.

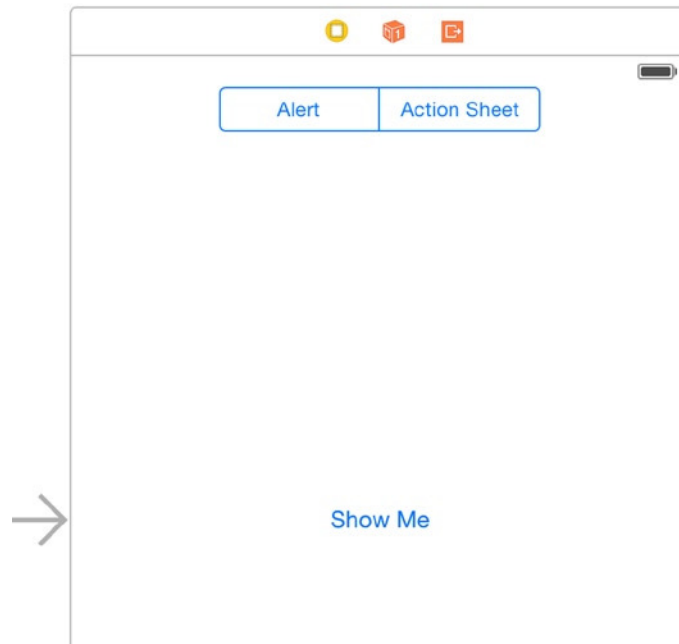


Figure 4-32. The completed interface for the Action tab

8. Click the view, go to Fix Auto Layout Issues, and click Add Missing Constraints.
9. You're ready to create the outlets and actions. As usual, switch to the Assistant Editor, and ensure that you have `ActionViewController.swift` selected. Control-drag a connection from the segmented control into the header, and create an outlet named `actionControl`, as shown in Figure 4-33.

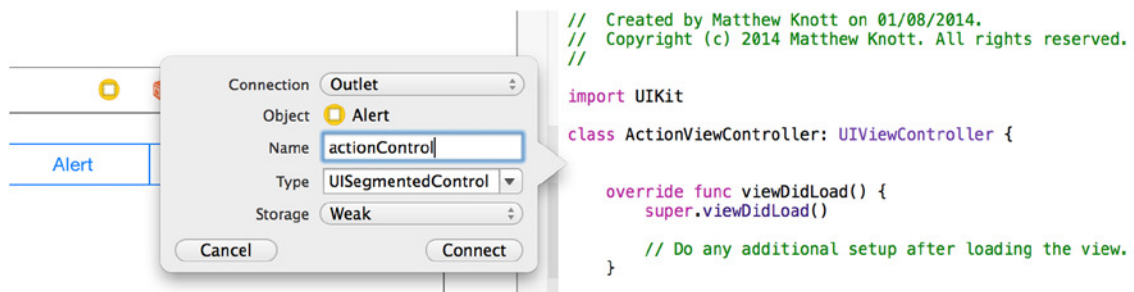


Figure 4-33. Creating the `actionControl` outlet

10. Create an outlet for the button called `showmeButton`, and then create an action for it named `performAction`.

The first lines of your `ActionViewController.swift` file should now resemble the following code:

```
import UIKit

class ActionViewController: UIViewController {

    @IBOutlet weak var actionControl: UISegmentedControl!
    @IBOutlet weak var showmeButton: UIButton!
    @IBAction func performAction(sender: AnyObject) {
    }

    override func viewDidLoad() {
        super.viewDidLoad()

        // Do any additional setup after loading the view.
    }
}
```

It's important to quickly check the official documentation when experimenting with different controls and frameworks, because quite often you need to specify that your view controller is acting as a delegate for the classes you're adding. This can be the case with action sheets and alert views if you want to take advantage of any of their delegate methods for handling user responses. Missing a delegate reference can lead to your application failing or your code not being called in some situations. Because you won't be using the delegate methods in this example, there is no need to add them.

You're now ready to begin coding the action in this class file. Switch back to the Standard Editor, and open `ActionViewController.swift` from the Project Navigator. All you need to look at in the file is the stub for the `performAction` action:

1. Scroll down until you find the `performAction` action. Inside its braces, you'll type an `if ... else ...` statement to see which segment is currently selected and determine the appropriate action to perform. You do this by checking the `UISegmentedControl`'s `selectedSegmentIndex` property. The segments are held in an array, and the index is an incremental number assigned to each entry. The index starts at 0, so if the selected index is 0, that means Alert is selected; if it's 1, that means Action Sheet is selected. Type the highlighted code into the action:

```
@IBAction func performAction(sender: AnyObject) {
    if actionControl.selectedSegmentIndex == 0 {
    }
    else
    {
    }
}
```

2. You need to initialize and show the alert view. The new `UIAlertController` takes far more code to initialize than its predecessor, but it's far more flexible. Type the highlighted code; once the action sheet code is written, you can see the completed result:

```
@IBAction func performAction(sender: AnyObject) {
    if actionControl.selectedSegmentIndex == 0 {
        var controller : UIAlertController = UIAlertController(title: "This is an alert",
            message: "You've created an alert view",
            preferredStyle: UIAlertControllerStyle.Alert);

        var okAction : UIAlertAction = UIAlertAction(title: "Okay",
            style: UIAlertActionStyle.Default,
            handler: {
                (alert: UIAlertAction!) in controller.dismissViewControllerAnimated(true,
                    completion: nil)
            })

        controller.addAction(okAction);

        self.presentViewController(controller, animated: true, completion: nil)
    }
    else
    {
    }
}
```

3. To code the else eventuality, type this very similar highlighted code inside the second set of parentheses:

```
@IBAction func performAction(sender: AnyObject) {
    if actionControl.selectedSegmentIndex == 0 {
        var controller : UIAlertController = UIAlertController(title: "This is an alert",
            message: "You've created an alert view",
            preferredStyle: UIAlertControllerStyle.Alert);

        var okAction : UIAlertAction = UIAlertAction(title: "Okay",
            style: UIAlertActionStyle.Default,
            handler: {
                (alert: UIAlertAction!) in controller.dismissViewControllerAnimated(true,
                    completion: nil)
            })

        controller.addAction(okAction);

        self.presentViewController(controller, animated: true, completion: nil)
    }
    else
    {
        var controller : UIAlertController = UIAlertController(title: "This is an
        action sheet",
```



```
        message: "You've created an action sheet",
        preferredStyle: UIAlertControllerStyle.ActionSheet);

    var okAction : UIAlertAction = UIAlertAction(title: "Okay",
        style: UIAlertActionStyle.Default,
        handler: {
            (alert: UIAlertAction!) in controller.dismissViewControllerAnimated(true,
                completion: nil)
        })

    controller.addAction(okAction);

    self.presentViewController(controller, animated: true, completion: nil) }
}
```

You've just written code that performs four distinct tasks. First, you define a `UIAlertController` called `controller`. Next, you define a `UIAlertAction` that adds a button to either the alert view or the action sheet to dismiss the controller. Third, you add the action to the controller, associating the two. Finally, you tell the main view to present the `UIAlertController`. The only difference between these two pieces of code is that `UIAlertControllerStyle` is `Alert` for an alert view and `ActionSheet` for an action sheet.

That's it—you've configured your view controller to show either an alert view or an action sheet depending on the selected index of a segmented control. Because the focus of this book is Xcode, not iOS app development, I'm only scratching the surface of what you can do with these two controls, but they're extremely easy to build on and are a key addition to any developer's bag of tricks.

You've coded the third and final tab, so go ahead and run the application in the simulator. It should produce results similar to those shown in Figure 4-34. Look at all the great things you've been able to achieve in this chapter, with a relatively small amount of effort and code! Hopefully your confidence with the Xcode IDE, iOS, and the Swift language is beginning to build.

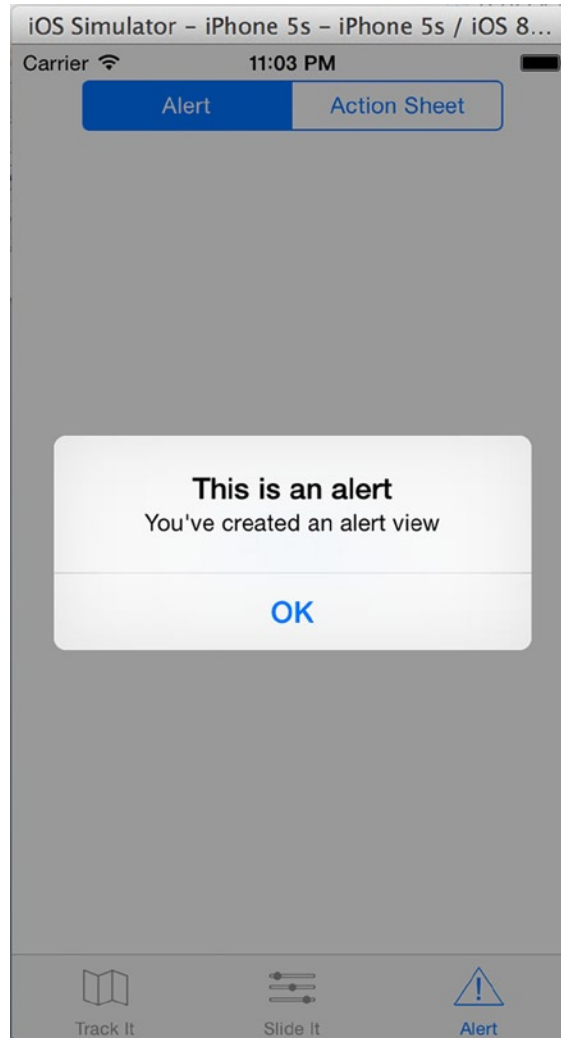


Figure 4-34. The alert view shown in the Alert tab

Changing the Interface with Code

In this chapter, you've taken a good look at how you can adjust the interface elements' attributes using the Attribute Inspector. But just as you can't use Interface Builder to add action sheets and alert views, there are some visual effects you can only achieve through code. You've already done a lot of hard work in this chapter, and you won't learn any more about Xcode here. So, look at this section as totally optional. However, you'll probably want to use the skills you can develop here to build your own applications for iOS devices, in which case these examples will prove invaluable.

Buttons and iOS 8

With iOS 7, Apple introduced the most radical change in design since the launch of the first iPhone: moving away from *skeuomorphism* to a *flat* design style. The decision was controversial when announced, but many are now warming to the change and have adapted their applications to fit with the new styles.

One area that changed that many want to alter in their applications is the standard button. Figure 4-35 shows the three buttons from the Contact screen in iOS 6 and iOS 7. In iOS 6, buttons *looked* like buttons, whereas in iOS 7 and iOS 8, they're shown in the same style as hyperlinks on a web page.

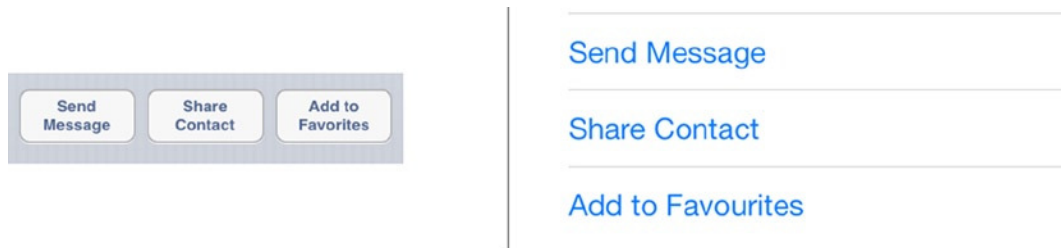


Figure 4-35. The three buttons from the Contact app's detail view for iOS 6 (left) and updated for iOS 7 (right)

Although you can change the background color of the button, you can't add rounded corners in Interface Builder, so you need to delve into code to make these alterations:

1. From the Project Navigator, open `ActionViewController.swift` and scroll down to the `viewDidLoad` function. Drop down a line after `super.viewDidLoad()`, and you're ready to add some custom code.
2. You're going to change the background color to a dark blue. You do this similarly to how you changed the background color in the Slide It tab, by creating a color using RGB values. But this time, you need to convert real RGB values, which range from 0 to 255, to fit in with what the method expects, which is a value between 0.0 and 1.0. To do this, you divide the value by 255.0. Add this line of code:

```
showmeButton.backgroundColor = UIColor(red: 9/255.0, green: 95/255.0, blue: 134/255.0, alpha: 1.0)
```

3. The button will be hard to read with blue text on a blue background, so the next task is to change the text color to white. You could do this in Interface Builder, but then you wouldn't be able to read the button's text when looking at the storyboard. Type the following code on the next line:

```
showmeButton.setTitleColor(UIColor.whiteColor(), forState: UIControlState.Normal)
```

- You can easily apply a curved corner to the button by specifying a float value greater than 0.0 to the button's `cornerRadius` property. You do this using the following code:

```
showmeButton.layer.cornerRadius = 4.0
```

Your `viewDidLoad` function should now look like this:

```
override func viewDidLoad() {
    super.viewDidLoad()
    showmeButton.backgroundColor =
        UIColor(red: 9/255.0, green: 95/255.0, blue: 134/255.0, alpha: 1.0)
    showmeButton.setTitleColor(UIColor.whiteColor(), forState: UIControlState.Normal)
    showmeButton.layer.cornerRadius = 4.0
}
```

Run the application in the simulator: you see the difference immediately in your button on the Action tab. The problem is that the button isn't set at a suitable size to make the most of your effects. Open `Main.Storyboard` from the Project Navigator, make the button on the Action view much bigger, and then reposition it to center it. Now run the application again: your button should look great and resemble that shown in Figure 4-36.

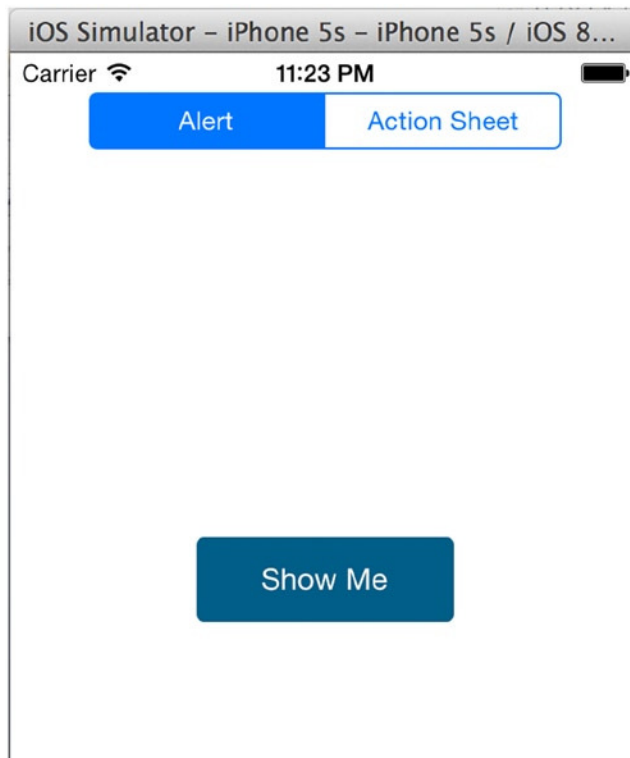


Figure 4-36. The customized button

That's it for this chapter. As a final challenge, try to apply curved corners to the text view in the Track It tab using the code you used to curve the button. If you get stuck, the answer can be found at the end of the summary for this chapter.

Summary

This has been a long chapter, but you've made it through and should be really proud of what you've achieved. The objective for the chapter was to learn more about creating interfaces in Xcode, and you did that with a mix of Interface Builder and writing custom code. The application you created was called Showcase, mostly because it gives you a cool app that you can load on your phone so you can show your friends and colleagues the kind of amazing things you're now able to develop!

Specifically, in this chapter, you did the following:

- Created an application from the Tabbed Application template
- Renamed the default view controllers, and created your own from scratch
- Removed the default views from the storyboard, and created three of your own
- Tied your new view controllers to their respective classes
- Created image sets in the images Asset Catalog, and populated them
- Linked views to a tab bar controller in a storyboard
- Learned about frameworks, and accessed the device's GPS function
- Learned about UITextView, UISegmentedControl, UISwitch, and UISlider controls
- Programmatically created a UIAlertView and a UIActionSheet
- Learned how to modify the visual appearance of controls using code

When you go through that list, you can see how many new skills you've learned in this chapter. Before moving on, though, I promised the solution to rounding the corners of the text view in the Track It tab. If you did it right, you should have added the following line to the viewDidLoad function in TrackViewController.swift:

```
locationText.layer.cornerRadius = 5.0
```

Very well done if you got that right.

Now on to Chapter 5, where you begin to look at the help provided by Apple through Xcode, along with how Xcode's intelligent code-completion feature makes coding much quicker and more efficient.

Getting Help and Code Completion

In Chapter 4, you accomplished quite a lot. You should be starting to feel more confident with the tools and features available in Xcode, and hopefully you're seeing how it can help you build your own applications.

This chapter focuses on the wealth of help that Xcode offers while you create the next big OS X and iOS apps. You see how Xcode makes writing code quick and easy with its intelligent code-completion feature. Looking at code completion will also help you grasp the basics of working with Xcode's code editor. The main focus of Chapter 4 was Interface Builder with a dash of storyboarding. You went from having a default tab bar application to an application that had three very different tabs. This chapter explains how to create the project shown in Figure 5-1, which demonstrates how to interact with some of the built-in applications: Mail, Messaging, and Safari.

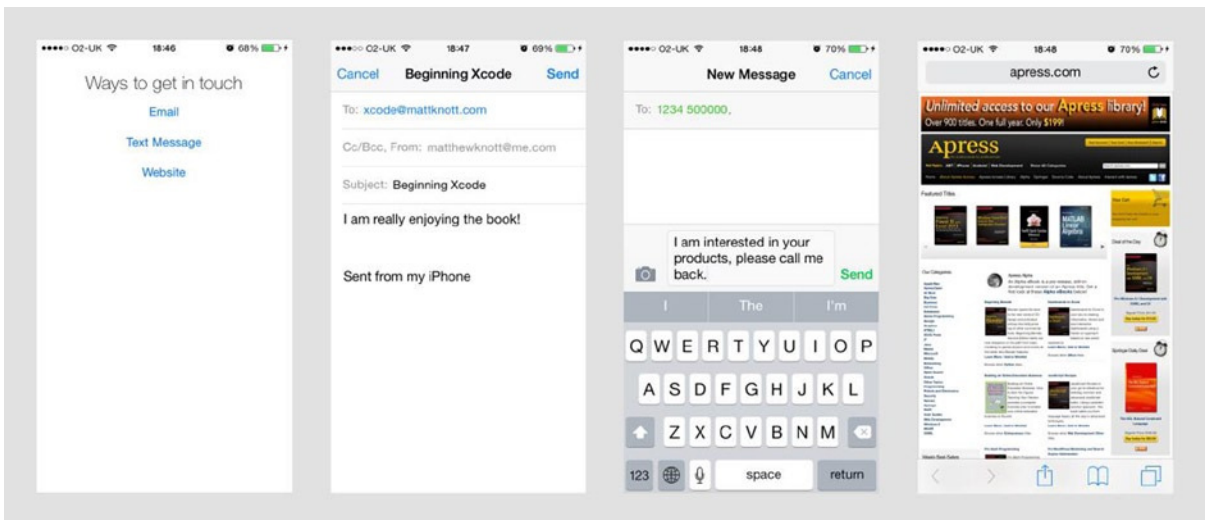


Figure 5-1. The InTouch application

Getting Help

Xcode provides help in a variety of different ways, not only on how to use Xcode but also on how to program using Objective-C and Apple’s frameworks. This section explains how Xcode helps you find a solution when you’re stuck, whether you’re using the code editor, the property list editor, or Interface Builder, or you’ve just encountered a problem while coding.

Creating the Project

Okay, you know what the aim of the chapter is, let’s start building the project.

1. Open Xcode, and create a new project by clicking Create A New Xcode Project on the Welcome screen or going to File ► New ► Project (⌘+Shift+N). Select the Single View Application template, and click Next.
2. Name your project InTouch, and ensure that Language is set to Swift and the Devices option is set to iPhone. Configure the other settings as you did in previous applications. Make sure the key settings match those shown in Figure 5-2, and click Next.

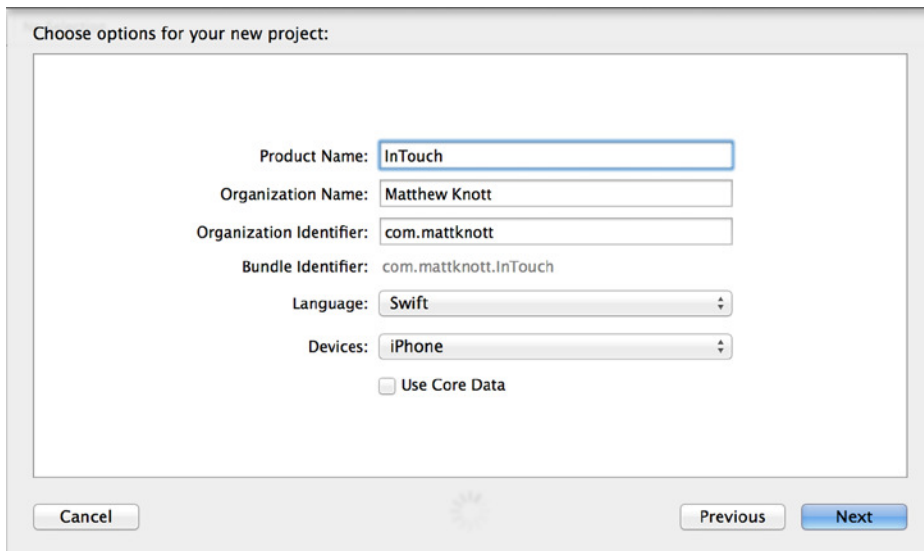


Figure 5-2. The initial settings for the InTouch application

3. You don’t want to create a Git repository, so leave Source Control unchecked. Make sure your project will be saved where you want it to be, and click Create.

That’s the foundation of the project setup. You’ll be amazed how much you can achieve beyond this with very little code and effort. Before you proceed, it’s important to understand that you don’t have to go to your favorite search engine if you’re stuck. Xcode has one of the best support systems of any IDE out there, if not *the* best.

Downloading Additional Documentation

Large parts of this chapter are dependent on you having the relevant documentation installed on your computer. Xcode often does this by default, but it's well worth checking that you have everything you need installed; otherwise you miss out on some excellent application programming interface (API) and system documentation. To check the state of your documentation, start by selecting Xcode ► Preferences from the menu bar (⌘+,). Click the Downloads tab, and you're presented with two lists, as shown in Figure 5-3. You can download legacy simulators and additional documentation sets. It's optional for this chapter, but if you want to bolster your documentation, you should download Xcode 6 library and iOS 8 library. Click the down-pointing arrow next to the file size, and the documentation begins downloading.

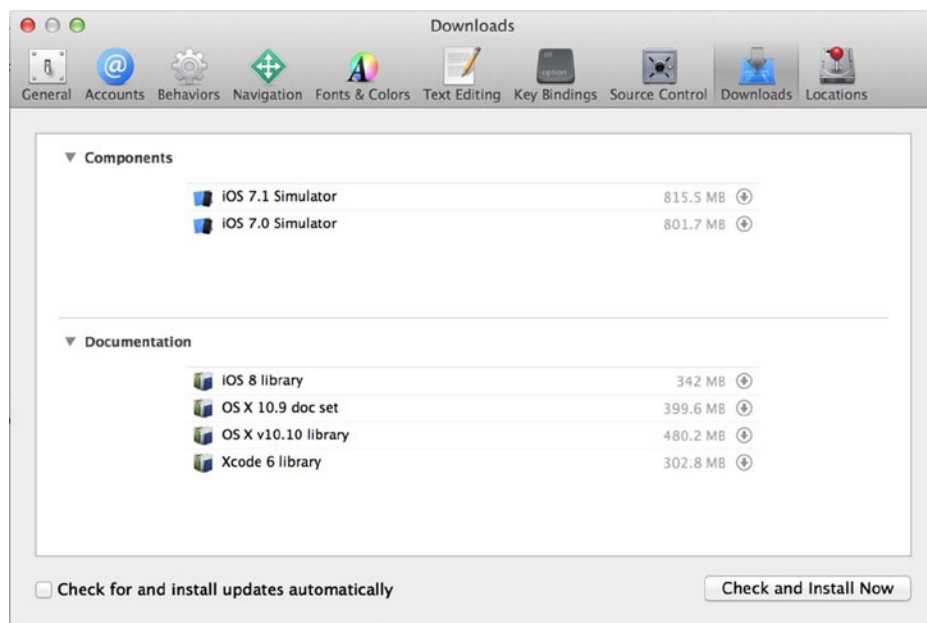


Figure 5-3. The Xcode Downloads tab in the Preferences window

Quick Help

To begin, let's focus on Quick Help. Quick Help provides a concise definition of symbols, interface objects, and build settings. The beauty of using Quick Help is that it resides in the Utilities area of Xcode and doesn't take away your focus when you're working on a project. To access the Quick Help Inspector, go to View ► Utilities ► Show Quick Help Inspector (⌘+⌘+2). To see Quick Help in action, open AppDelegate.swift and highlight UIResponder. Quick Help instantly updates to give you useful information. Figure 5-4 shows Quick Help in action.

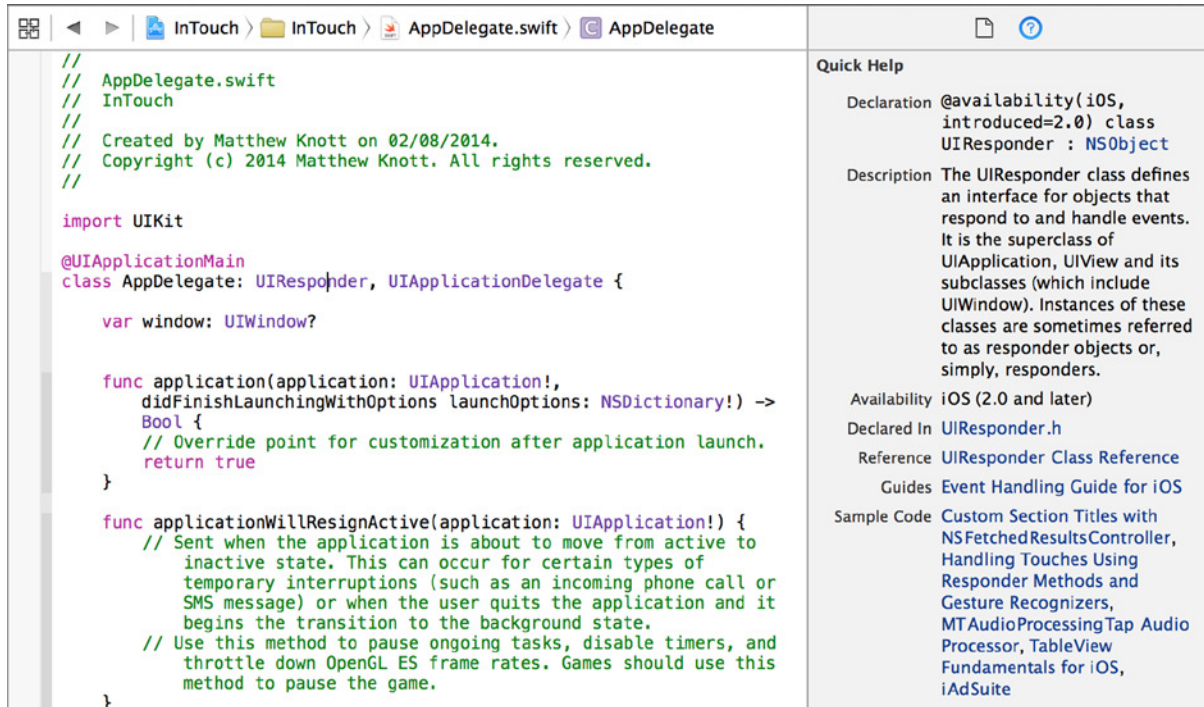


Figure 5-4. Quick Help showing information about the UIResponder class

You can see from Figure 5-4 that a range of information appears—exactly what is displayed varies with what you select. However, following are the main entries of a Quick Help entity:

- **Declaration:** The declaration gives you an overview of the class's definition, including its base class and any adopted protocols, to help you understand the class's capabilities.
- **Description:** This is rather self-explanatory, but the main point is that the description of a symbol, an object, or a setting covers how it should be used and also gives an overview of event handling.
- **Availability:** The availability states the minimum requirements in terms of the version of iOS the user can be running in order for the object or symbol to function. Many classes have been available since the release of the iPhone software development kit (SDK); however, it's a good idea to keep an eye on newer technologies to ensure that you don't run into compatibility issues.
- **Declared In:** Here you're given the name of the header file in which the object or symbol is defined. When you click it, you can view the header's source code in the code editor.
- **Reference:** Each of the main classes has a class reference that fully explores and explains the class and its protocols and functions.

- **Guides:** If you installed the iOS 8 Library documentation, guides are available for certain classes that give you straightforward help for implementing and using the class, as well as how it can work with other classes.
- **Sample Code:** Apple has made dozens and dozens of functioning applications available to demonstrate different classes and their implementation. These sample code applications can be very simple or full-blown games. Again, the availability of sample code in Quick Help depends on you've downloaded the iOS 8 Library.

If you come across a symbol, an object, or a setting that doesn't have a Quick Help entry, you can search Xcode's documentation for whatever you've selected. For instance, in `AppDelegate.swift`, if you highlight `func`, you see that there isn't a Quick Help entry. With `@func` still highlighted, click Search Documentation, as shown in Figure 5-5, and the Documentation Viewer window appears.

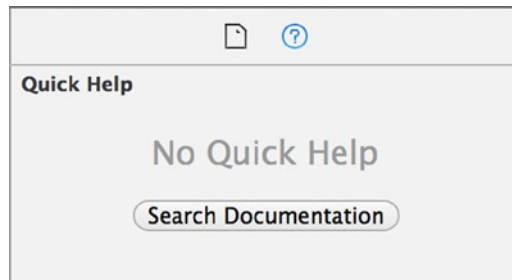


Figure 5-5. Quick Help's Search Documentation button, which you can click when no entry exists for the highlighted entity

Another way to access Quick Help is to press the Option key (`⌘`) and select a class in the code editor. Figure 5-6 illustrates the dialog displayed when the `UIWindow` class is selected in `AppDelegate.swift`. This is only accessible from the code editor, but it's very useful if you need to quickly look up a class definition. You're provided with a description of the class, the version of iOS in which it was introduced, where it's declared, and a link to additional documentation (any text that's blue is a link that opens an external file, either source code in the code editor or a class reference in the Documentation Viewer).

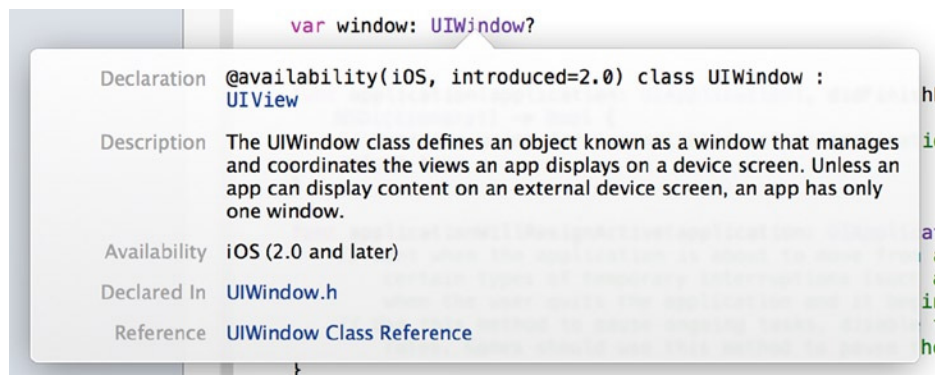


Figure 5-6. A more compact version of Quick Help, displaying information regarding the `UIWindow` class

Documentation Viewer

Before I focus on the Documentation Viewer in detail, I want to talk about how this facet of Xcode has changed from previous versions. Documentation Viewer was introduced in Xcode 5, having previously been integrated into the Organizer tool, which is examined later in the book in Chapter 14.

As you become more familiar with the workings of Xcode, you'll learn to depend on the Documentation Viewer for its quick access to documentation and SDK references, which, given the literally thousands of APIs, are remarkably detailed. Figure 5-7 shows the Documentation Viewer. To open it, go to Help ► Documentation And API Reference ($\text{⌘}+\text{⌘}+0$).

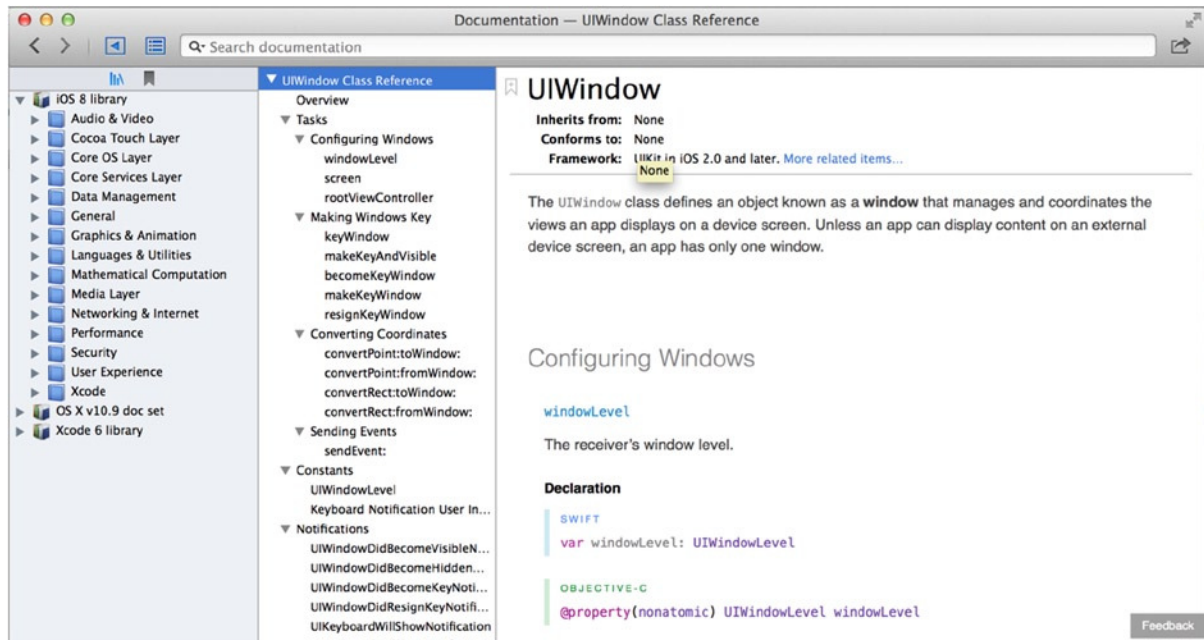


Figure 5-7. The Documentation Viewer

The Documentation Viewer allows you to search and browse a variety of different documents and resources, including these:

- Class, framework, protocol, and object references
- Technical guides
- Getting started documents
- Technical Q&A
- Change logs and revision histories
- Technical videos
- Sample code

Notice that in addition to the document being viewed, the Documentation Viewer has three key areas: the Toolbar, the Navigator, and the Table of Contents; I cover all three of these in detail in the following sections.

Note What's really useful about the documentation sets in Xcode is the fact that they're completely accessible offline. This means if you're working where there isn't an Internet connection, you still have access to the wealth of documentation and references.

Toolbar

The Toolbar contains all of Documentation Viewer's navigation and sharing features. They're as follows, moving left to right across the bar:

- *Backward/forward navigation*: The Documentation Viewer works very much like a web browser in that the content is HTML based. You can bookmark pages of interest; you can navigate backward and forward through the history of your research. This can be invaluable as you dip in and out of API references, trying to find how to correctly instantiate a new class or get a better understanding of the class properties.
- *Sidebar controls*: Immediately after the backward and forward navigation arrows are two buttons. The first controls the visibility of the Navigator, and the second controls the visibility of the Table of Contents.
- *Search*: The sheer amount of documentation provided by Apple can overwhelm even the most seasoned iOS developers, and it would be absurd for Apple to assume developers can find the correct documentation for the nitty-gritty details of underlying Swift and Objective-C technologies, classes, and functions. That's why the Documentation Organizer has a very useful search functionality that allows you to search for a particular term.

To demonstrate, search for a broad term: for example, *gesture*. This can relate to a variety of different things, but luckily Xcode helps you find just what you're looking for. Figure 5-8 illustrates the results that are displayed when searching for *gesture*.

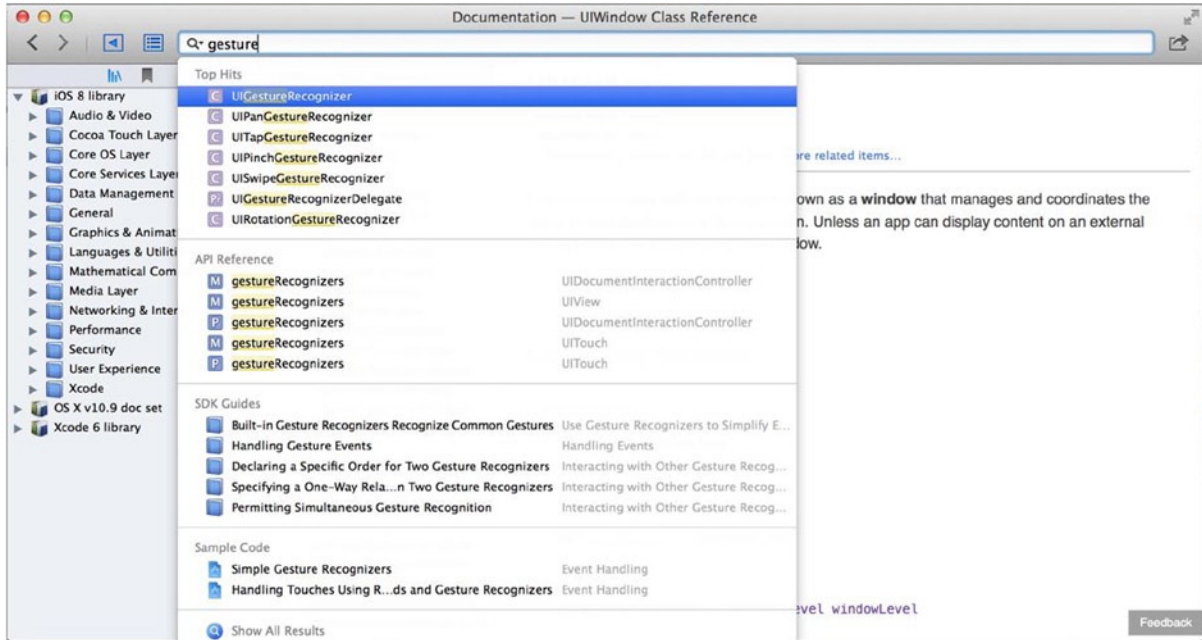


Figure 5-8. The top results displayed when searching for gesture

Notice that as you type, only a few results are displayed; click Show All Results to see the full set of results spread across a number of sections. In this case, API Reference, SDK Guides, Tool Guides, and Sample Code are shown.

- **Share:** A familiar icon to most people, and almost identical in function to its equivalent in Safari, the Share button presents a list of ways to share or export the current document: Open In Safari, Add Bookmark, Email Link, Message, and, if available, the option to open a PDF copy of the document. Because documentation in the Documentation Viewer is often spread over numerous pages, it can be difficult to search in the scope of the entire document. This is where opening the PDF copy can be useful, and most classes and APIs support this. Another new feature is the ability to open sample code in the Swift Playground, which you see in detail in Chapter 11.

The Navigator Sidebar

The Navigator provides you with two methods of accessing help and documentation. First, you can browse the entire library of documentation installed on the machine; and second, you can place bookmarks in various pieces of documentation. Access to each feature is controlled by two icons at the top of the Navigator, as shown in Figure 5-9.

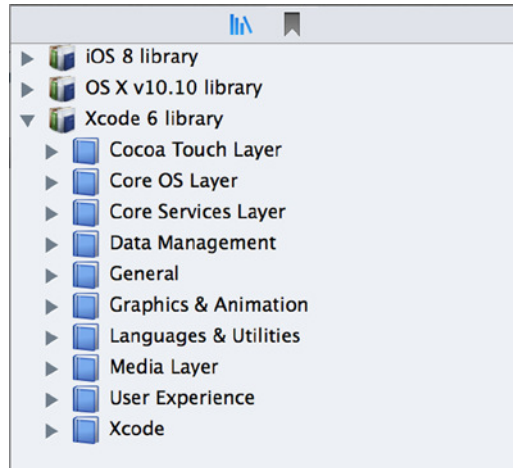


Figure 5-9. *The Navigator gives you access to the document library and your bookmarks*

The ability to browse the documentation lets you start with a topic area, such as file management, and then expand that topic, viewing all sample code, guides, and class references. For many developers, this is the best way to use the system documentation to approach a problem or area of development, because all the relevant resources are available when you reach the topic area; you don't have to start with a specific class or framework.

The ability to bookmark articles, guides, references, and sample code is another heavily used feature of the Documentation Viewer. This makes it easy to refer to a piece of documentation at a later date. It's inevitable that at some point you're going to stumble on something that may not be useful right away but that you may want to refer to later—so a bookmark is just what you need!

Adding a Bookmark

To bookmark a piece of documentation, Figure 5-10 shows that you have three choices. You've already seen the Add Bookmark option from the Share icon (1) on the Toolbar. Alternatively, you can simply click the bookmark icon found throughout the documentation (2), or you can right-click the documentation while you're viewing it and select Add Bookmark (3).

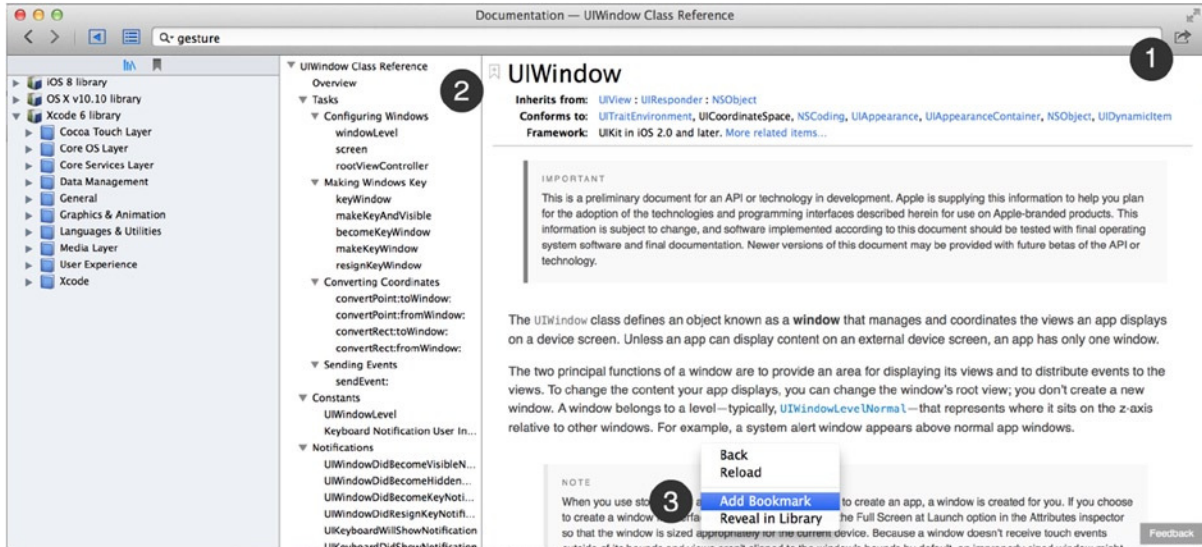


Figure 5-10. Three ways to bookmark documentation while viewing it

Viewing Your Bookmarks

Once you've bookmarked documentation, when you want to access it again, ensure that the Navigator is active using the relevant Toolbar icon and then select the bookmarks icon. You see a list of everything you've bookmarked. Simply click the documentation item in the Navigator, and it opens. To delete a bookmark, select the item from the sidebar and then press Backspace, or right-click the item and select Delete.

Unfortunately, your bookmarks aren't synced anywhere, so you have to be on your device to access them. It's sometimes useful to use Apple's online documentation library if you want to access your bookmarks on other devices, such as your iPhone or iPad.

The Table of Contents Sidebar

The Table of Contents sits to the left of the document you're viewing. As you would expect, it provides a hierarchical overview, allowing you to quickly navigate a large document and see any associated source code or example projects (see Figure 5-11).

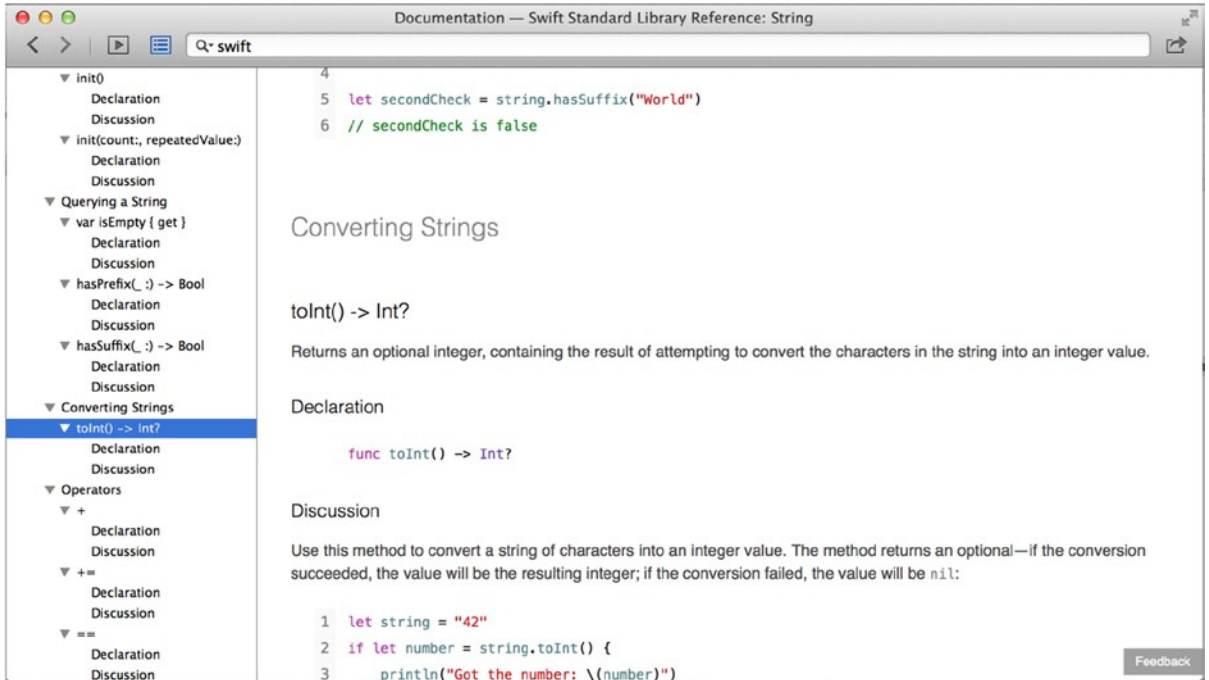


Figure 5-11. The Table of Contents lets you navigate large documents quickly

Quickly Accessing Documentation

Xcode makes it easy to access help while you're working on your project without having to open the Documentation Viewer each time you want to look something up. Simply right-click, and, depending on what part of Xcode you're working with, a help menu appears in which you can access popular documentation guides that are displayed as submenus. You can access these menus from the Source Editor (Figure 5-12), Interface Builder (Figure 5-13), and the Project Navigator (Figure 5-14), along with a variety of others. When you choose an item from the help submenu, the Documentation Viewer is displayed and shows the relevant guide or reference. This is useful when you would like to look something up quickly or want to know how to perform a task in the part of Xcode you're using.

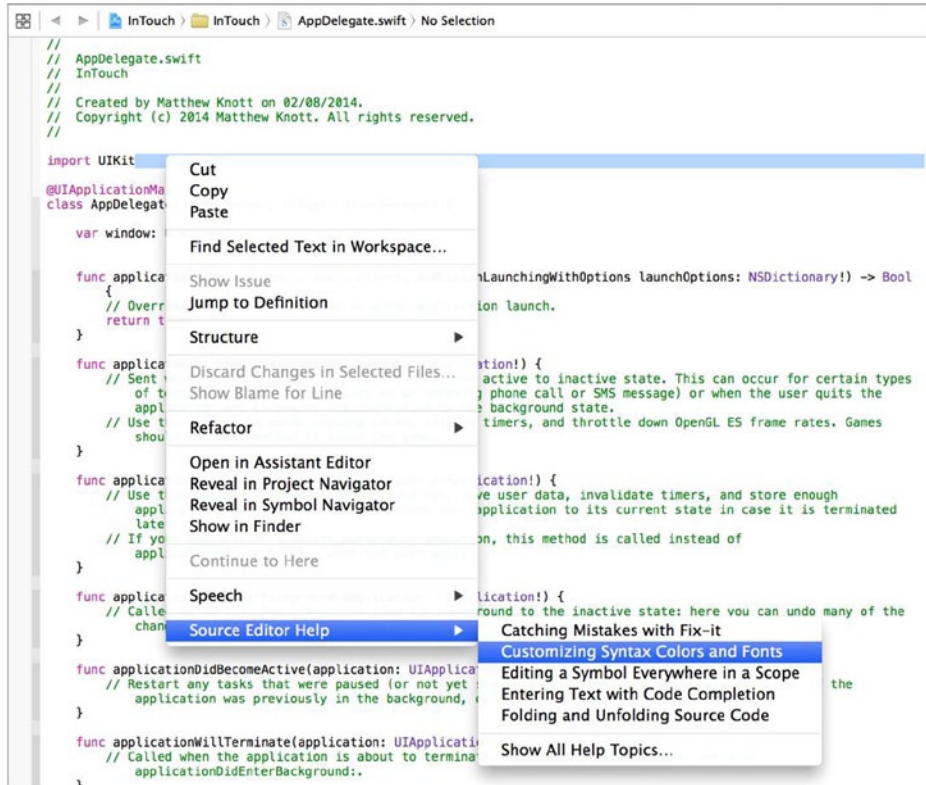


Figure 5-12. The Source Editor help menu

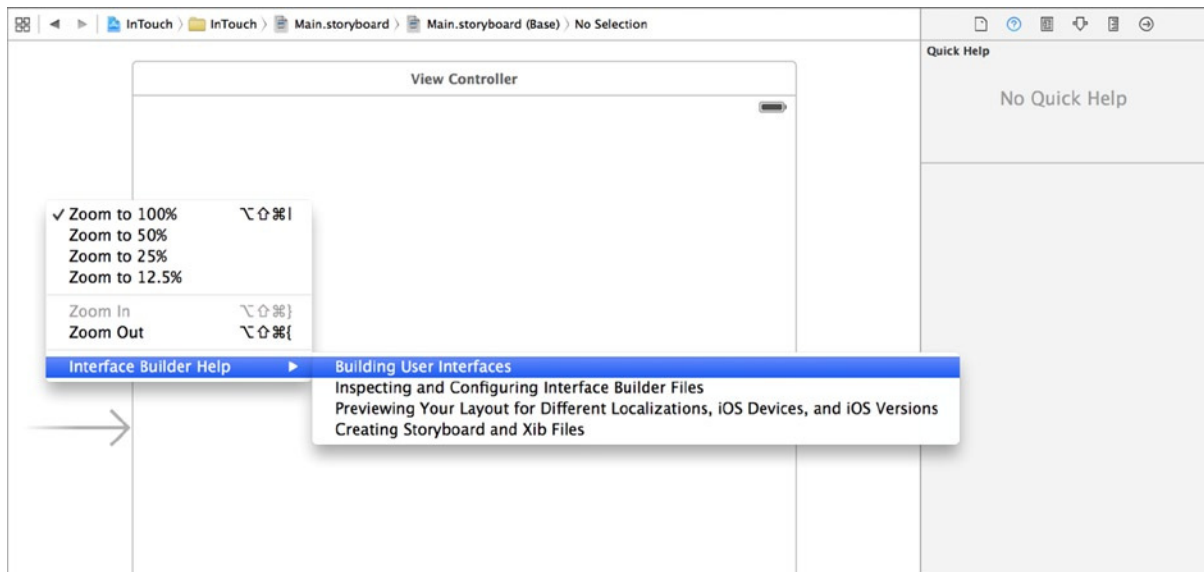


Figure 5-13. The Interface Builder help menu

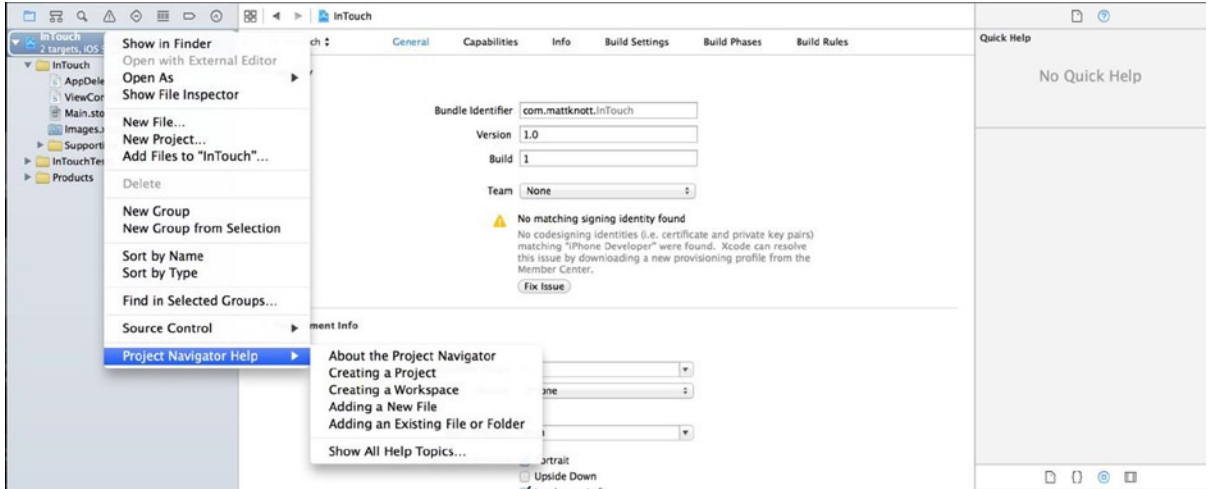


Figure 5-14. The Project Navigator help menu

Apple's Web Site

As mentioned previously in this chapter and also at the beginning of this book, Apple's developer web site also provides an excellent library of information. In fact, when you download a documentation set, you're in fact downloading an offline version of Apple's reference library—the Documentation Viewer is essentially a glorified web browser.

Here is a list of useful online resources that are provided by Apple, aimed mainly at iOS developers:

- <http://developer.apple.com>: The main home of Apple's developer web site. Here you can access the three developer centers available; see the latest and greatest news in the world of Apple, specifically for developers; and access many other parts of Apple's developer world, such as resource centers, information about copyright, and much more. I discuss these later when it's time to build and share your application.
- <https://developer.apple.com/devcenter/ios/index.action>: The iOS Dev Center is the main hub of all Apple resources provided to iOS developers and is immensely useful when you're developing applications. You can access the iOS Provisioning Portal, Member Center, iOS Reference Library, and also available downloads.
- <https://developer.apple.com/library/ios/navigation>: The iOS Developer Library is what's used to populate the Documentation Organizer. The library includes technical guides, a wealth of references, sample code, and documentation. Apple really sets itself apart with the overwhelming amount of support provided to developers. If you're working with a certain technology in iOS or OS X, chances are there's a detailed guide for it. You can search the library using the search bar to the left. It's easy to become overwhelmed, but once you get familiar with its layout, finding what you need will become easy.

- <https://developer.apple.com/library/ios/navigation/#section=Resource%20Types&topic=Sample%20Code>: It's always useful to see something up and running, and seeing a working example can also save you a lot of time. The sample code provided by Apple allows you to test a particular technology yourself, dissect the code, and even use the code in your own applications.
- <https://devforums.apple.com>: Something that I haven't really mentioned is Apple's Developer Forums. The Developer Forums aren't as active as other forums available online, but the users are much more helpful and willing to offer advice and solutions.
- <https://developer.apple.com/videos/ios>: Apple also provides a host of useful videos. Topics range from low-level technologies to high-level technologies and frameworks provided by Apple. Because many developers don't have the chance to attend WWDC, Apple's annual World Wide Developers Conference, the video coverage of the conference can also be useful. This too can be found in the videos section of the developer web site.

Code Completion

Code completion can greatly increase any developer's productivity and can also save you a lot of time—that is, if you know how to use it correctly. Using code completion in Xcode can take some getting used to, depending on your prior experience; however, Xcode is much more intelligent than other IDEs.

To get a taste of code completion and also using the Source Editor as a whole, let's do things a bit differently this time. Specifically, you'll code your actions manually and then wire them up using Interface Builder, instead of using Interface Builder to create the action stubs and linkages as you did in previous chapters:

1. Open `ViewController.swift` and, under the line `import UIKit`, begin typing the following (remembering that it's case sensitive):

```
import MessageUI
```

Xcode's code completion may appear, but in this case it makes no suggestions; this isn't particularly helpful, but it's one of the downsides of the modules approach. Once you've typed the code, Xcode happily accepts it. Fortunately, importing frameworks is a minor part of your application, and the code completion is excellent for everything else.

You've imported the `MessageUI` framework because it gives you access to `MFMailComposeViewController`, among other classes, so try out some of the skills you learned earlier in this chapter. Go ahead and search for it in Documentation Viewer; you'll find a wealth of information, including confirmation of its parent framework.

- You need to tell the view controller to act as a delegate for `MFMMessageComposeViewControllerDelegate` and `MFMailComposeViewControllerDelegate`. To do this, immediately next to class `ViewController: UIViewController`, type the following, using the code-completion dialog to insert the correct code as shown in Figure 5-15:

```
, MFMMessageComposeViewControllerDelegate, MFMailComposeViewControllerDelegate
```

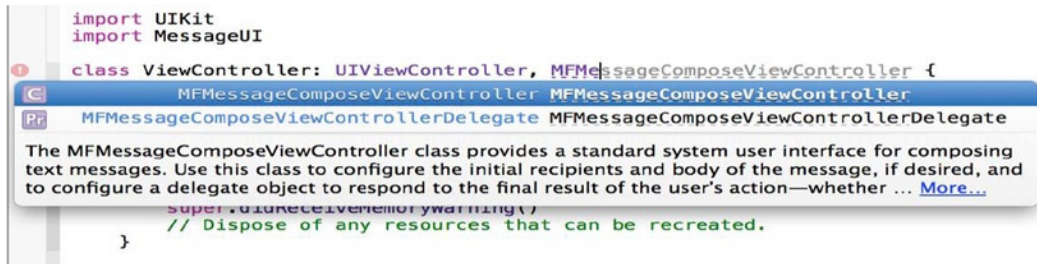


Figure 5-15. The code-completion dialog appears as you add the delegate protocols

Note You can use the up and down arrows to change the selection in the code-completion dialog. Then, with the correct item highlighted, press Enter: your cursor focuses on the end of the line, and the code is entered.

When you add the `MFMMessageComposeViewControllerDelegate` protocol, you receive an immediate error; it's important to note that you haven't done anything wrong. The issue is that this protocol has a single delegate function that must be implemented in the class adopting the protocol. This means if you want to add that protocol onto this view controller, the next thing to do is to add the delegate function. Before you do that, ensure that the start of your view controller looks like this:

```
import UIKit
import MessageUI

class ViewController: UIViewController, MFMMessageComposeViewControllerDelegate,
MFMailComposeViewControllerDelegate {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
    }
}
```

- To remove the error, you can use Xcode's powerful code completion to quickly add the missing delegate method. After the `viewDidLoad` function, drop down a couple of lines and start typing `messageComposeViewController`. As you do, code completion kicks in and presents you with the delegate function, as shown in Figure 5-16. Press the Tab key to create the method.



Figure 5-16. Using code completion to create the missing delegate function

- The method was generated with a code placeholder. Remove this with the Backspace key so you're left with an empty function.
- You need to manually write three actions that the buttons in your app's interface will use to open the web site or begin composing an e-mail or text message. Begin typing the following highlighted code just before the `override func viewDidLoad()` line:

```

import UIKit
import MessageUI

class ViewController: UIViewController, MFMessageComposeViewControllerDelegate,
MFMailComposeViewControllerDelegate {

    @IBAction func sendEmail(sender: AnyObject) {

    }
    @IBAction func sendText(sender: AnyObject) {

    }
    @IBAction func openWebsite(sender: AnyObject) {

    }
}

```

Now that you've created the stubs for each action, let's go through the actions and focus on what the code will do before writing it.

Opening Web Sites in Safari

Many applications that you download from the App Store use web views in the native application to load visual information from the Web or from locally stored assets. This is generally frowned on by Apple, which prefers you to write everything natively. In the InTouch app, your goal is to direct users to your company web site, and you do this by forcing Safari to open and display a specified web address. There are some good reasons for using Safari in this instance: first, it's overkill to implement a web view for something that will require a lot of work to create a completely functional implementation with back and forward controls; and, second, if the user opens the home page in Safari, they will be able to bookmark it, sync the tab with other iOS devices, and share it on social media.

Locate the `openWebsite` action stub you just created. Between the braces, begin to type the highlighted code, but feel free to replace `http://apress.com` with your own URL:

```
@IBAction func openWebsite(sender: AnyObject) {  
    UIApplication.sharedApplication().openURL(NSURL(string: "http://apress.com"))  
}
```

In this code, notice that you aren't creating any variables: you access the `UIApplication` class and use what are called *type methods* in Swift. Although this book isn't a guide to the Swift language as such, it's important to understand some of the basic concepts of the language. If you have experience with other C-based languages, you may be familiar with static methods; a type method is the same thing. In essence, a type method is a function that you can access without instantiating the parent class—that is, without assigning it to a variable. Type methods are great when you want to quickly access a function without setting a bunch of parameters.

Note Each time you add a bracket, notice that for a brief moment a little yellow box appears around its counterpart (that is, the one you're closing). This is to make sure you don't add too many or too few brackets; this also applies to braces.

Sending an E-mail with `MFMailComposeViewController`

Next, let's write the code that will allow the user to send an e-mail from your application. What's significant is that you don't need to create an interface for this; you simply use `MFMailComposeViewController` and preset the values. This is a great approach because unless Apple changes the class, your application will always use the iOS Mail application's compose view, instantly making it familiar to users, more future-proof, and less work than writing your own view.

To implement the view controller, you also have to write another delegate function to handle what happens after the e-mail has been sent. First write the action by adding the following highlighted code:

```
@IBAction func sendEmail(sender: AnyObject) {
    if MFMailComposeViewController.canSendMail()
    {
        var mailVC = MFMailComposeViewController()

        mailVC.setSubject("Beginning Xcode")
        mailVC.setToRecipients(["xcode@mattnkott.com"])
        mailVC.setMessageBody("<p>I am really enjoying the book!</p>", isHTML: false)
        mailVC.mailComposeDelegate = self;

        self.presentViewController(mailVC, animated: true, completion: nil)
    }
    else
    {
        println("This device is currently unable to send email")
    }
}
```

Feel free to change `xcode@mattnkott.com` to your own e-mail address, and also feel free to change the subject and presumptuous contents of the e-mail message to whatever you'd like the user to see before they begin to compose their e-mail message to you.

Next you need to create a new function `mailComposeController: didFinishWithResult`. This is a delegate function that is called when the user wants to dismiss the mail-compose view controller. In this instance, you account for each of the possible outcomes of trying to send an e-mail before you let the user dismiss the compose view, which they can do after they've sent their message or if they decide to cancel it. Add the following function before the other delegate function, `messageComposeViewController: didFinishWithResult`:

```
func mailComposeController(controller: MFMailComposeViewController!,
    didFinishWithResult result: MFMailComposeResult, error: NSError!) {

    switch result.value {

        case MFMailComposeResultSent.value:
            println("Result: Email Sent!")

        case MFMailComposeResultCancelled.value:
            println("Result: Email Cancelled.")

        case MFMailComposeResultFailed.value:
            println("Result: Error, Unable to Send Email.")

        case MFMailComposeResultSaved.value:
            println("Result: Mail Saved as Draft.")
```

```

        default:
            println("unknown");
    }

    self.dismissViewControllerAnimated(true, completion: nil)
}

```

You've written all the code needed to send an e-mail. Next you look at text messaging and how the process is similar to sending an e-mail.

Sending a Text Message

Short Message Service (SMS) messaging is still one of the most popular forms of communication in the world today, and just like e-mail, Apple makes it easy to send a text message from your application. The code is very similar to the previous two methods, but there is one big distinction: you have to test this on a physical device, because the simulator can't simulate SMS.

With that in mind, in the `sendText` action, type the following highlighted code:

```

@IBAction func sendText(sender: AnyObject) {
    if MFMessageComposeViewController.canSendText()
    {
        var smsVC : MFMessageComposeViewController = MFMessageComposeViewController()
        smsVC.messageComposeDelegate = self
        smsVC.recipients = ["1234500000"]
        smsVC.body = "I am interested in your products, please call me back."
        self.presentViewController(smsVC, animated: true, completion: nil)
    }
    else
    {
        println("This device is currently unable to send text messages")
    }
}

```

Just as with the e-mail implementation, you now need to complete the code for the delegate function that is called when the process of sending the text message is completed. Once again you compare the result of the attempt to send a text message against several possible results and print text to the console based on the outcome. Add the highlighted code to the `messageComposeViewController: didFinishWithResult:` function:

```

func messageComposeViewController(controller: MFMessageComposeViewController!,
    didFinishWithResult result: MessageComposeResult) {

    switch result.value {

        case MessageComposeResultSent.value:
            println("Result: Text Message Sent!")

        case MessageComposeResultCancelled.value:
            println("Result: Text Message Cancelled.")
    }
}

```



```

case MessageComposeResultFailed.value:
    println("Result: Error, Unable to Send Text Message.")
default:
    println("unknown");
}

self.dismissViewControllerAnimated(true, completion: nil)
}

```

Building the Interface

You've written all the code your application needs to perform three essential communication tasks. Now you need to build and connect your interface to harness the code you've just written:

1. Open `Main.storyboard` from the Project Navigator.
2. Drag a label and three buttons onto the view. Position the label at the top of the view and the three buttons beneath it, one on top of the other.
3. Resize the label so it fills the full width of the view, and then open the Attributes Inspector (`⌘+⌘+4`). Set the Text attribute to say "Ways to get in touch". In the Font attribute, click the T to customize the font. Set Font to Custom, Family to Helvetica Neue, Style to Thin, and Size to 23, as shown in Figure 5-17. You may need to increase the height of the label.

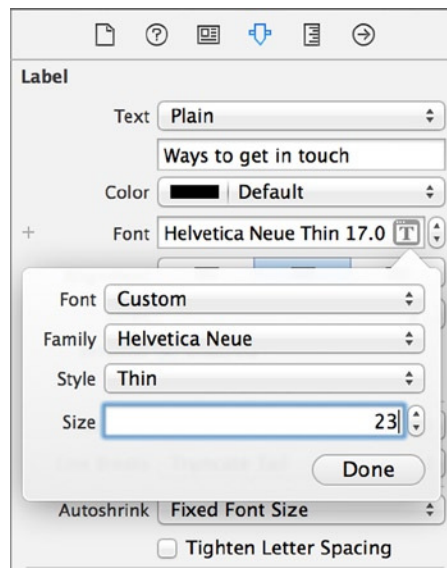


Figure 5-17. Setting the custom font properties

4. In order, double-click each of the buttons and name them Email, Text Message, and Website, respectively, before centering them.
5. Use the Resolve Auto Layout Issues button, and select Reset To Suggested Constraints under the All Views In View Controller heading to pin the elements in place.

Your finished interface should look something like that shown in Figure 5-18.

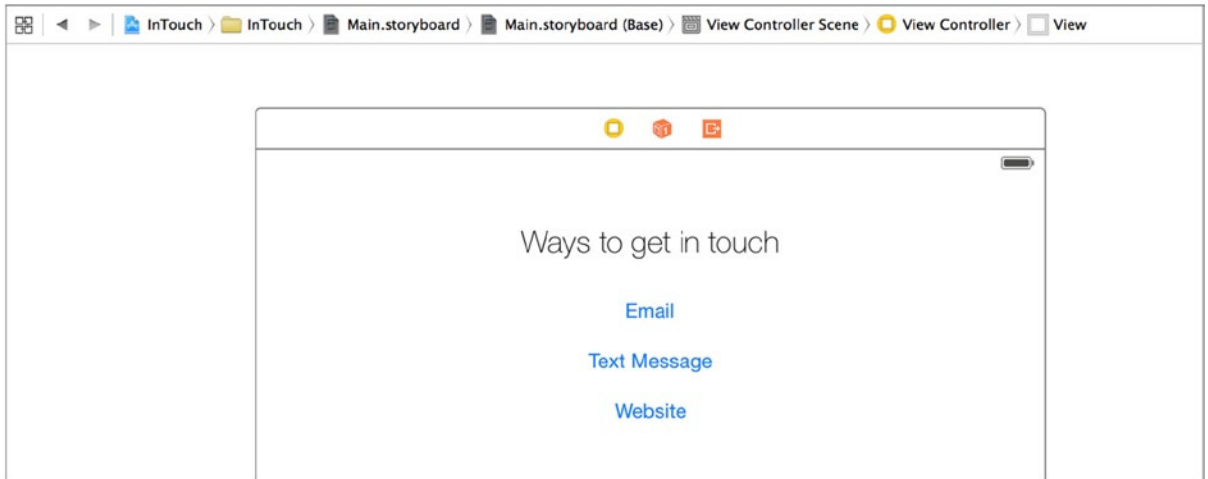


Figure 5-18. The finished interface

Making Connections

The code is written and the interface is assembled, but there is no linkage between the two. To address this, you need to connect the actions you've created to the buttons using the Connections Inspector (the sixth and final inspector):

1. Be sure you still have `MainStoryboard.storyboard` open. If it isn't, open it from the Project Navigator.
2. Open the Connections Inspector ($\text{⌘}+\text{⌘}+6$) with the view controller selected from the document outline, as shown in Figure 5-19. Note that you can select the view controller by clicking the bar at the top in the design area.

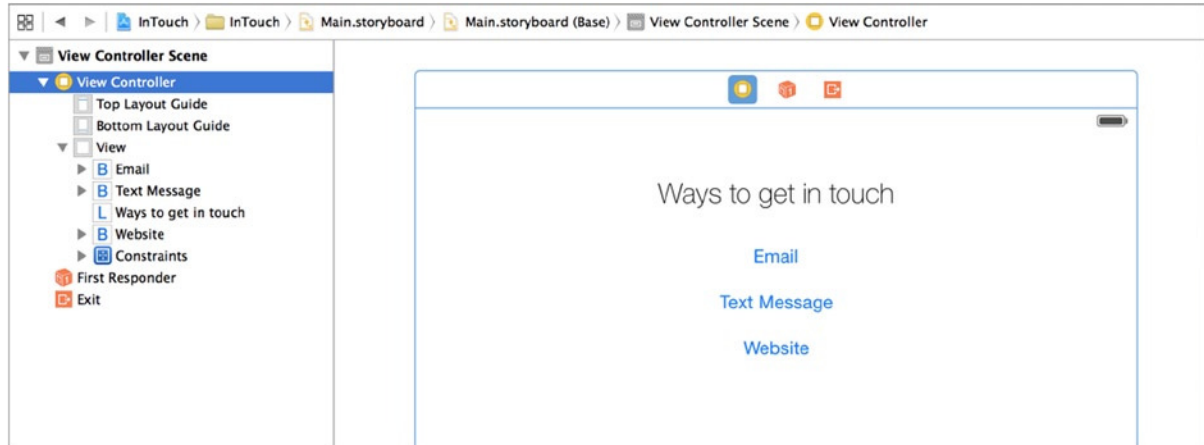


Figure 5-19. *Main.storyboard* with the view controller selected and the Connections Inspector open

- Under the Received Actions heading in the Connections Inspector, you see the three actions with a hollow circle next to each one. From the `sendEmail` method's circle, click and drag a connection to the button, as shown in Figure 5-20.

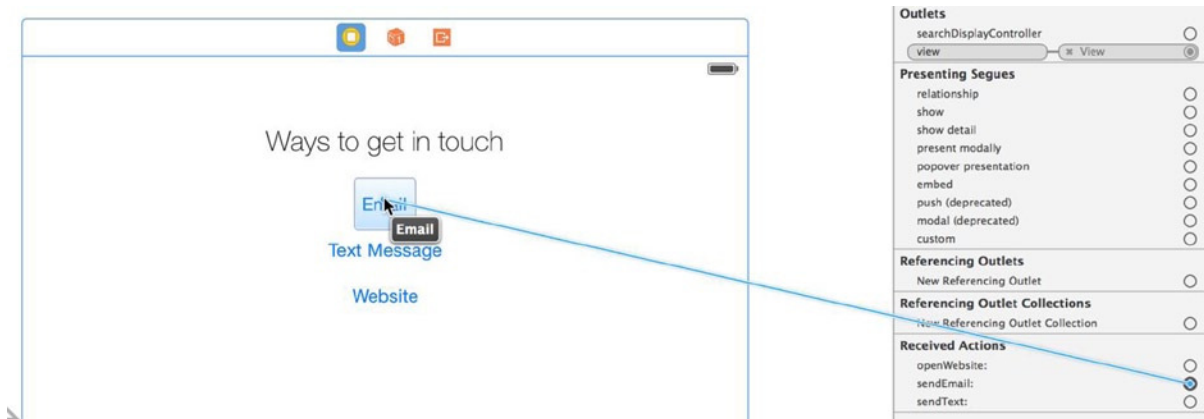


Figure 5-20. Connecting an action to a button from the Connections Inspector

- A menu appears when you release the connection, presenting you with a list of trigger events. The action is called when the correct type of event occurs. Select `Touch Up Inside` from toward the bottom of the list.

Note When a button in an iOS application is tapped, the Touch Up Inside event is triggered. By linking the action to this event, you can be sure the code will be executed when the user taps your button.

- Repeat these steps, linking the two remaining actions to their respective buttons, and making sure to select the Touch Up Inside event from the list.

You've now learned one of several ways to link buttons to preexisting actions! Well done—you're well on your way to becoming an Xcode master.

Running the Application

Run the application either on your device or on the iOS simulator. When you tap the Website button, InTouch is placed in the background and Safari opens. Similarly, if you click the Email or Text Message button, a view is pushed in which the user can send an e-mail or SMS. Figure 5-21 illustrates, in order on the main view, the Email compose screen, the Text Message compose screen, and Safari with the web site.

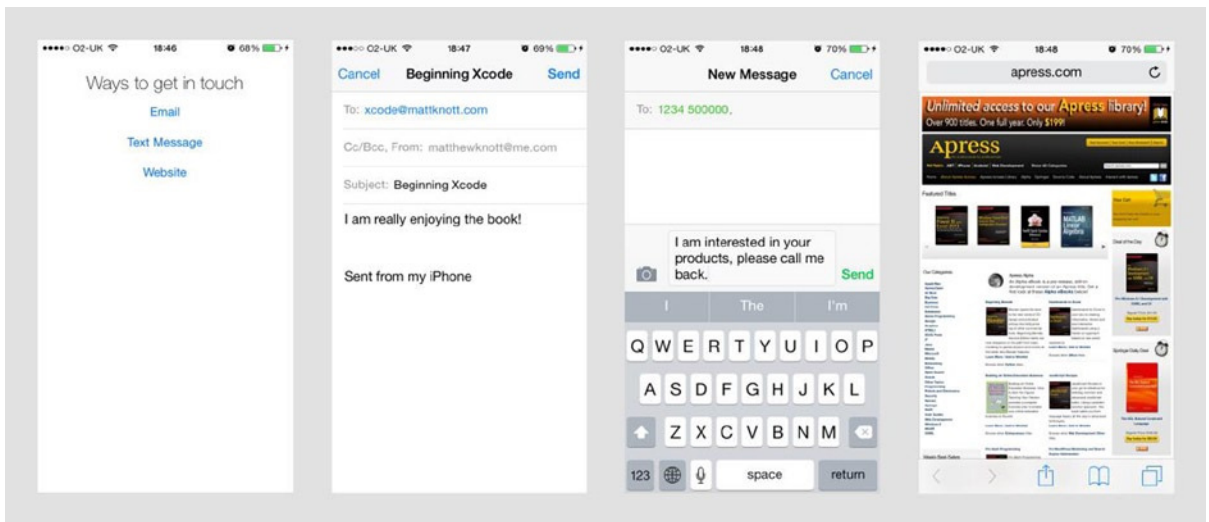


Figure 5-21. All the different views of the application

Note You can't send an e-mail from the iOS simulator, and you can't even see the SMS dialog in it; so in order to fully test this feature, you need to run InTouch on an actual iOS device that has an e-mail account configured. You commence testing on a physical device in Chapter 14; if you haven't skipped ahead to find out how to test on a physical device, this will be a great project to come back to. For now, though, take my word for it and assume that it works.

Summary

This chapter explored quite a few different topics, and you added some interesting communication features to an application. Here is what you achieved:

- Looked at the Documentation Viewer
- Saw how Xcode makes it easy to access help from wherever you're working in Xcode
- Learned about Quick Help
- Explored Apple's online documentation
- Became more familiar with the Source Editor
- Used code completion to speed up how you code
- Connected actions using the Connections Inspector
- Added a framework to your project

The next chapter looks at constraints. If you've been testing on a physical device or have tried rotating the simulator, you've probably noticed that things can get a little messed up. I explain in detail how to quickly fix this in Interface Builder, and I also provide your first look at programmatically adding objects to the view controller while using programmatically generated constraints to keep them aligned.

Constraints

Chapter 5 detailed the many ways Xcode gives you access to help, documentation, and guidance. You created a handy communications application along the way that could compose a text message or an e-mail, or even open a web site in another browser. You also looked at an alternative to the Assistant Editor for making connections between Interface Builder controls and your outlets and actions, and I talked briefly about how the Organizer had changed since Xcode 5.

This chapter introduces you to an area of Xcode that changed significantly between versions 4 and 5 and has undergone another significant revision in Xcode 6: the Auto Layout system. Auto Layout (or Auto layout, as Apple sometimes refers to it) has been updated in Xcode 6 with the addition of size classes to allow for more adaptable storyboards, the subject of the next chapter. In this chapter, you use Auto Layout to build an example application that adapts to changing resolutions and screen orientations the way you want it to. What's great is that the techniques you learn here are largely applicable to both iOS and OS X development.

Understanding Auto Layout

In Xcode 5, Auto Layout provides you with a comprehensive set of tools to automatically lay out your controls in a view and constrain how different controls react to each other when the resolution changes or when the iOS device is rotated. In the past there was a lot of stigma around Auto Layout because of its shortcomings: it was inaccurate and offered poor flexibility. In Xcode 5, Apple completely overhauled Auto Layout, creating something totally new that gave developers very fine control over the behavior of the elements in a view.

Although many of the tools and principals remain, Apple has added an extra layer of configuration with the introduction of *size classes*. This mechanism means a single storyboard can work on both iPhone and iPad, which in the past were separate storyboards. The catalysts for this were the iPhone 6 and, even more, the iPhone 6 Plus, which bridge the gap between phone and tablet. The so-called *phablet* blurred the lines, and Xcode 6 changed to embrace this, making life easier for developers by allowing them to use a single storyboard if they want to.

This chapter takes you through the principles of Auto Layout before focusing specifically on how size classes affects these principles. I'll present four ways to add constraints to your controls:

- *Manually*, using the Control + click-and-drag method you're familiar with for creating connections
- Using the *Add Missing Constraints* function to automatically add constraints
- Using *Reset To Suggested Constraints* to update constraints when you move constrained controls
- Using the *Pin* menu to set constraints with numeric precision

As a context for demonstrating the power of Auto Layout, you create a login dialog similar to those in many password-protected services. Let's begin!

Building an Authentication View

The authentication view you create in this chapter will be a familiar sight to users of Twitter, Facebook, or any of the countless other web service–based apps in the App Store. You build the project in this chapter in a way that teaches you how to lay out the elements of a view with Auto Layout and constraints; at the same time, you learn some of the finer points of configuring text fields that will be crucial when you develop your own applications.

Figure 6-1 shows LoginApp, the project you create in this chapter. Here you can see constraints in action. When the device rotates, the text field resizes and adapts to the new orientation. I also explain in depth the text field's attributes; you can see in the finished application that you set placeholder text on the e-mail address, but a number of hidden refinements contribute to a rich user experience.

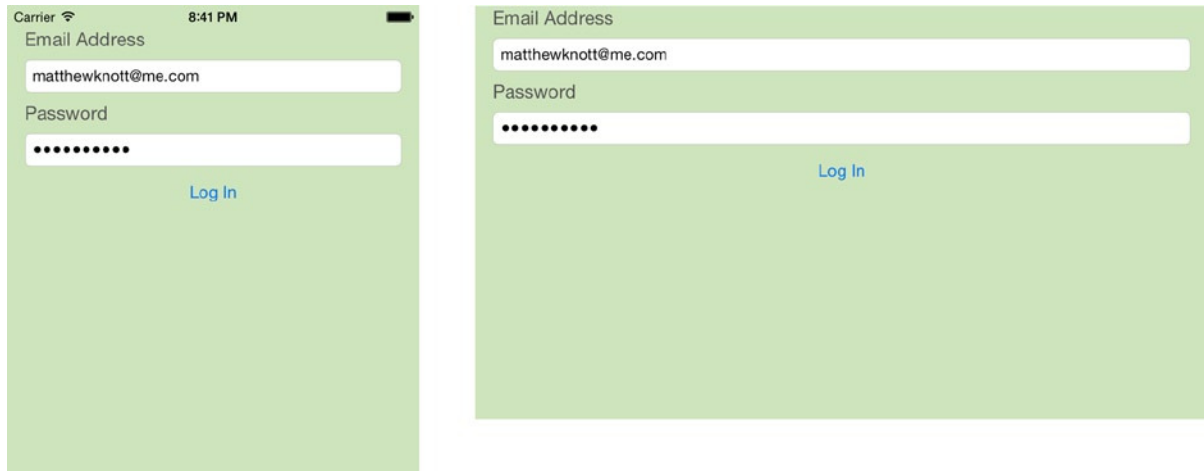


Figure 6-1. The login page for the app resizes automatically when rotated

You've done this several times already, so you should be pretty familiar with creating new projects by now. But to give you a heads-up, it's always important to read the setup steps, because in the next chapter you try something new:

1. Open Xcode, and create a new project by going to File ► New ► New Project (⌘+Shift+N) or, alternatively, choosing Create A New Xcode Project from the Welcome screen (⌘+Shift+1).
2. Select Single View Application, and click Next.
3. Name the project LoginApp, ensure that Devices is set to Universal, and leave the other options at their defaults, as shown in Figure 6-2. Click Next.

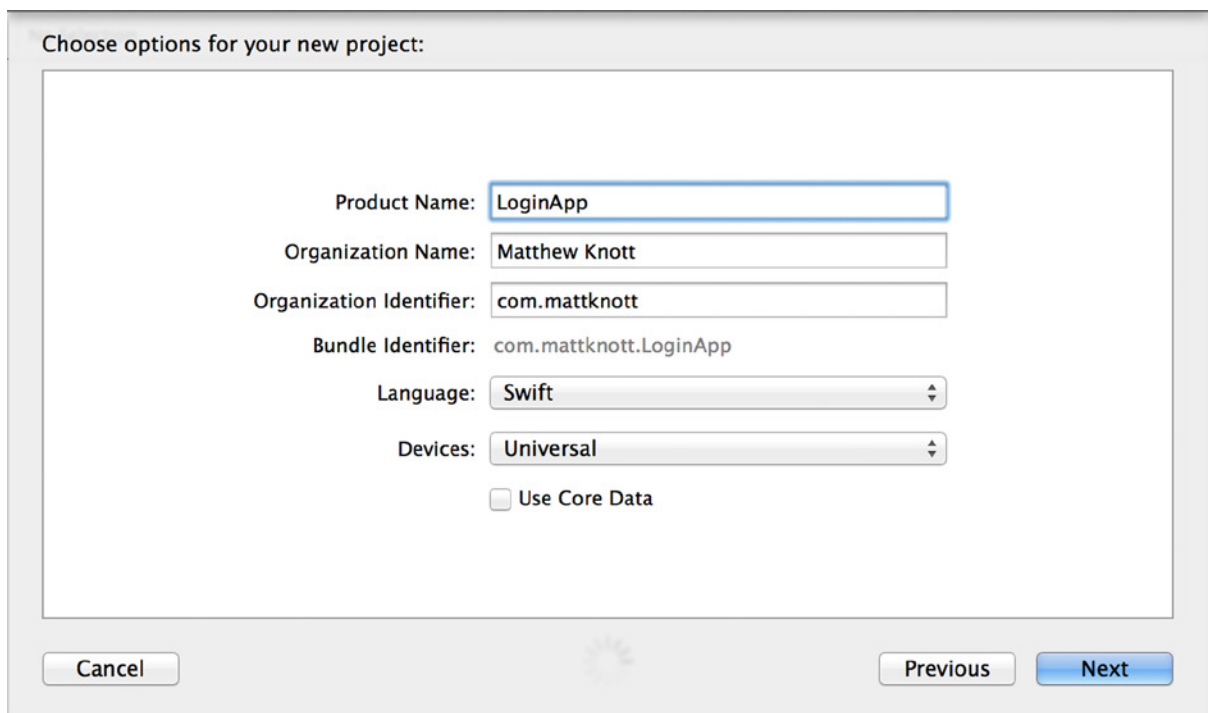


Figure 6-2. Setting up the project

4. Select a location to save the project, and click Create.

That's it! You're ready to start building your application. But before you begin building up your view, note that everything you do in this chapter relating to Auto Layout and setting constraints is done purely from Interface Builder—you don't write a single line of code. However, at the end of the chapter you look at using a little code just to add the finishing touches to your form. With that in mind, let's open `Main.storyboard` and get to work on the interface.

Design Considerations

If you haven't already, I hope that after reading this book you start writing your own applications using Xcode, whether for fun, to solve a problem you encounter, or maybe because of a gap you've spotted in the market. When you're a beginner, you'll make design decisions that, when you run the application, make you realize you've made a terrible error in judgment about how you've arranged the layout.

The good thing is that most of the time there's a simple solution, and the whole thing becomes a valuable learning experience. Login dialogs are a potential banana skin when you're starting out: when you design a beautiful layout on a static view, it's easy to forget about the keyboard that in many cases slides up and covers the fields, making them inaccessible. To address this problem, make sure you position the text fields and labels in a way that ensures that the keyboard will not obscure them in any of the iOS screen formats. Follow these steps:

1. After you open `Main.storyboard`, open the Document Outline, if it isn't already visible, by clicking the button in the bottom-left corner of the design area that has a box with a solid line down the left inside it, as shown in Figure 6-3.

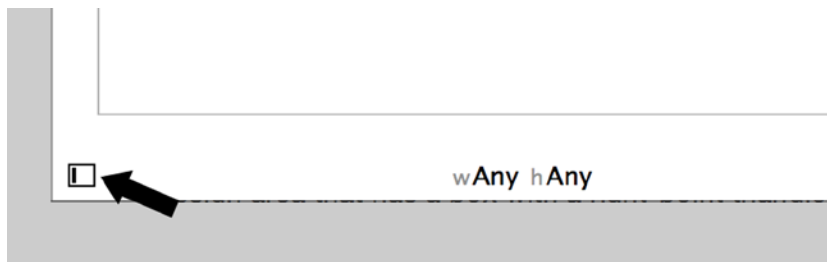


Figure 6-3. The Show Document Outline button in the bottom-left corner of the design area (indicated by the arrow)

2. Click the disclosure triangle to the left of View Controller Scene, and then click the disclosure triangle to the left of View Controller. Click the View item, as shown in Figure 6-4, and then open the Attributes Inspector ($\text{⌘}+\text{⌘}+4$).

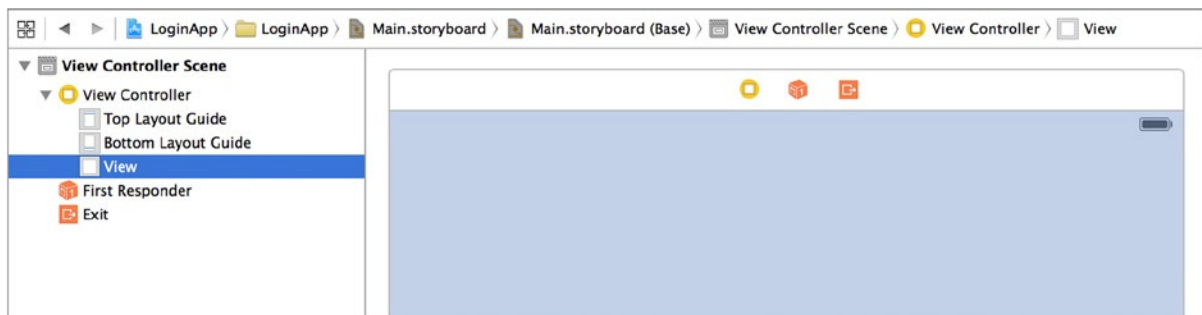


Figure 6-4. Selecting the view from the Document Outline

Note The Document Outline bar can be extremely useful when you're working with a large number of controls in a view. You can alter the hierarchy of the elements to make one appear above or below another, or you can add controls to a scroll view instead of the main view. It's also useful for creating connections between view controllers that are physically far apart in the design area, to save zooming out many times.

3. To make this view more appealing, give it a background color. I selected a pale green color for my view, but you can select whichever color you like. Click the Background drop-down list, and either select a preset color or click Other if you want to choose from the pallet or specify an RGB color. I used Red 206, Green 228, and Blue 188: you can use the same colors by using the color sliders from the color picker and setting RGB sliders from the drop-down menu, as shown in Figure 6-5.

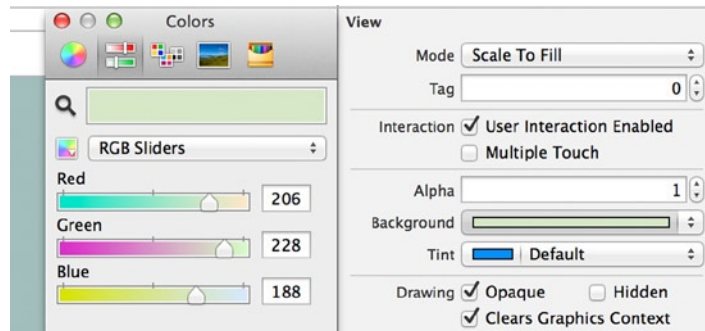


Figure 6-5. Setting an RGB color

4. You're ready to add controls to the view. Drag a label and a text field from the Object Library onto the view: put the label in the top-left corner, where it snaps to the blue guidelines, and snap the text field into place directly beneath it, as shown in Figure 6-6.

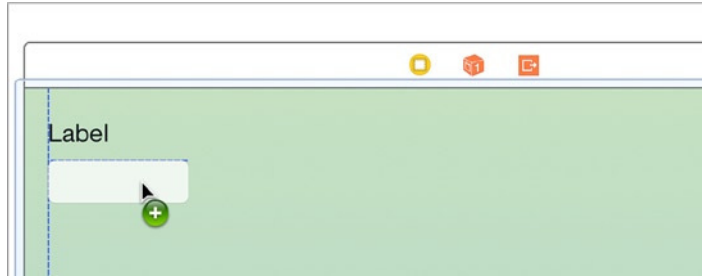


Figure 6-6. The text field snaps into place below the label

5. Use the handle on the right of the text-field box to resize it. Drag it to the right until, again, the blue guidelines appear.
6. Because the second row is a copy of these elements, you can duplicate them. Holding down the Command (⌘) key, in the Document Outline, click to highlight both Label and Round Style Text Field. You should see handles appear on both items in the view, as shown in Figure 6-7.

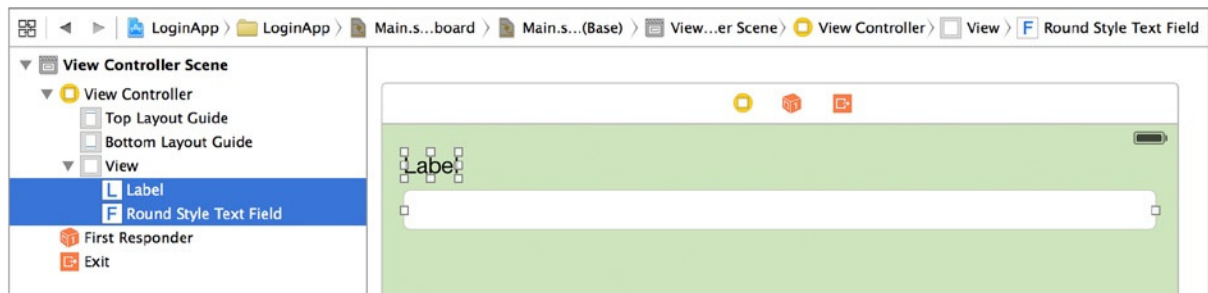


Figure 6-7. Both elements of the view selected

7. To duplicate the items, copy them with ⌘+C and then click a blank area of the view or select View from the Document Outline. Click the light green view, and paste the elements back into the view with ⌘+V. When the two items appear on the view, they're grouped together: move them as one, and snap them into place below the first text field.
8. Drag in a button from the Object Library and position it centrally in the view, a little below the last text field. If everything has gone to plan, your view should resemble that in Figure 6-8.

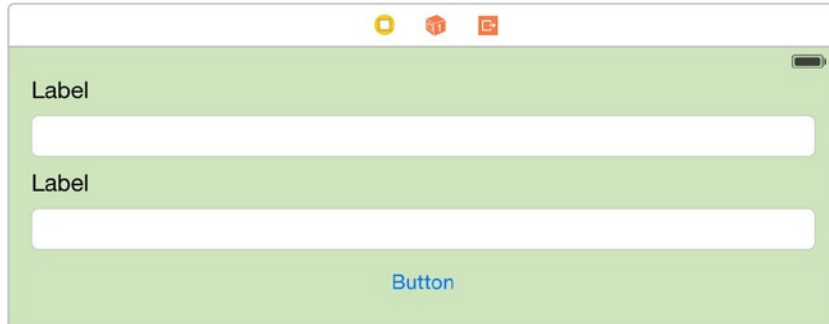


Figure 6-8. The skeleton of the LoginApp

9. With the elements all in place, you need to set the titles and text colors of the labels and the button. Select the first label, and from the Attributes Inspector, set Color to Dark Gray Color from the list of presets. Change Text from Label to Email Address. Your attributes for the first label should now resemble those shown in Figure 6-9.

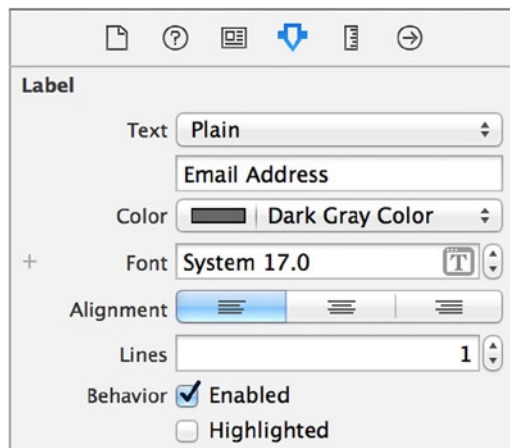


Figure 6-9. The attributes of the first label

10. Resize the label to the right just enough to display the full text.
11. Repeat the previous steps on the second label, but set its Title to Password. Then set the button's Title to Log In. When you resize the Password label, drag it to the same width as Email Address: when they're the same size, a blue guide line appears.

Before you go any further, let's run the application as it stands using the iPhone 6 Simulator. The Simulator opens, and the text fields appear off the view. Rotate the Simulator by selecting Hardware ► Rotate Left from the menu bar or pressing ⌘+left arrow. Now, the problem is that the fields are too short and aligned on the left rather than spanning the entire view; as you can see in Figure 6-10, the elements stay the same size but also stay in the same position.



Figure 6-10. The misaligned view, badly needing some constraints

The elements don't move because they have no behaviors applied to them, telling them what to do when the screen rotates or the view is bigger or smaller than the storyboard. In Xcode, these behaviors are called *constraints*.

Debugging Views in Xcode

Before you apply constraints to the view to snap everything into its proper place, let's take a moment to look at Xcode's incredibly useful view-debugging tool. This tool allows you to pause the execution of the application, analyze each control in the view, and see views that may have rendered offscreen—something the Simulator can't help you with. It's important to be aware of this facility as you begin to implement constraints, because in addition to giving you a flexible layout that adjusts intelligently to varying form factors and orientations, they can also have unforeseen and confusing effects, such as moving elements way off screen when rotated.

Follow these steps:

1. In the Simulator, rotate the screen back to portrait mode by going to Hardware ► Rotate Right.
2. Leave the app running, and switch back to Xcode.
3. At the bottom of the screen is the icon for the View Debugger, as shown in Figure 6-11. Alternatively, go to Debug ► View Debugging ► Capture View Hierarchy.

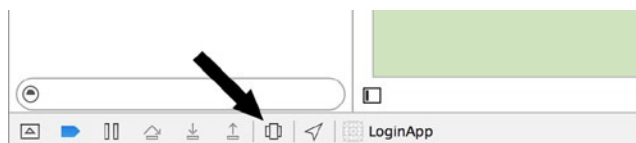


Figure 6-11. Accessing the View Debugger

When the View Debugger runs, the first thing you see are the two fields extending outside the bounds of the view, as shown in Figure 6-12.

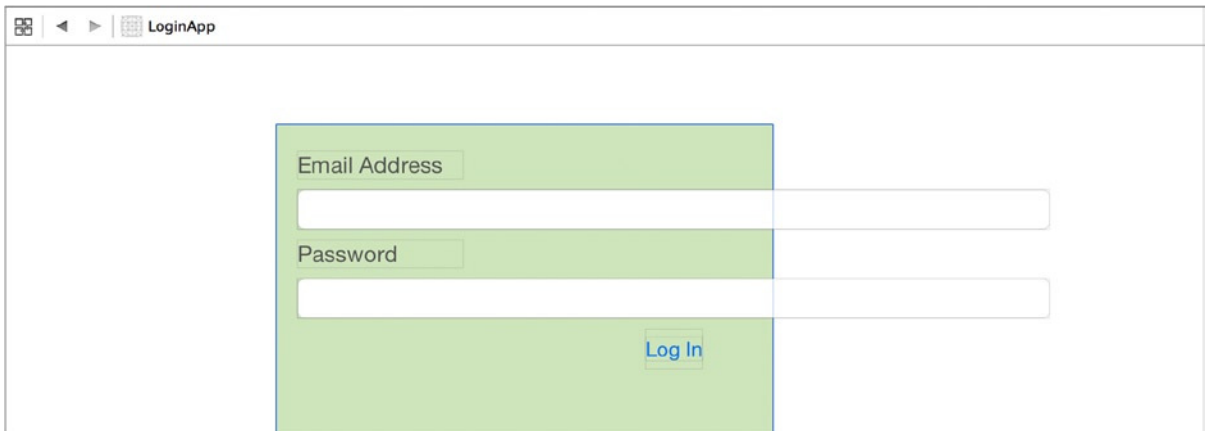


Figure 6-12. Using the View Debugger in Xcode to view elements outside the bounds of the view

The View Debugger gives you a huge array of tools to help track down stray elements, but for the most part, you can get everything you need from the basic view hierarchy. Stop the application in the Simulator, and you're returned to the storyboard so that you can begin adding constraints to the view.

Manually Adding Constraints

The first method I'll explain for adding constraints is the manual method, in which you use the familiar Control + click-and-drag technique to specify a constraining relationship between multiple elements. You should be used to holding the Control key while clicking and dragging—you've done it a number of times in previous chapters to create connections to actions and outlets.

Note Even though the constraints are being added manually, they're still part of Auto Layout, which is a bit of a contradiction and can be confusing.

Here are the steps:

1. Let's add a number of manual constraints to the layout. Select the first text field, and then, while holding down the Control key, click the text field and drag a line to the left side of the view. When you release the mouse button, a contextual dialog appears, as shown in Figure 6-13.



Figure 6-13. Dragging a line from the text field to the side of the view (left) presents a context menu (right)

2. You want to tell Interface Builder to constrain the position of the text field to the sides of the view so that when the view rotates, the text field resizes. Select **Leading Space To Container Margin** from the menu, and an orange guideline appears to the left of the text field.
3. Control-drag a connection from the first text field to the right side of the view. When you release the mouse button, select **Trailing Space To Container Margin**. That's all you need to do to constrain the Email Address text field to the sides of the view.
4. Repeat the process by selecting the Password text field, Control-dragging a connection to the left side of the view, and selecting **Leading Space To Container Margin**. Next, Control-drag a connection from the text field to the right side of the text field, and choose **Trailing Space To Container Margin**.
5. The constraint for the Log In button is slightly different. At this point you're probably happy with the vertical position and size of the button, but to keep it that way, you need to add some constraints. First, you want to constrain the button so that it stays centered horizontally. To do this, select the button, and Control-drag a line directly beneath the button, as shown in Figure 6-14.

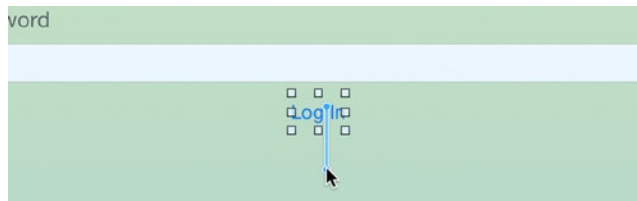


Figure 6-14. Dragging a connection below the Log In button so that you can constrain it horizontally

6. When you release the mouse button, the context menu opens again: this time select Center Horizontally In Container. A guideline appears from the top of the view to the bottom.
7. One final constraint: whatever happens, you want the Log In button to remain the same distance beneath the Password text field as it is now. Control-drag a connection from the button to the Password text field, and select Vertical Spacing.

That's it! With a few clicks, you've done enough to make your layout respond to changes in orientation and form factor. All that's left to do is to test it in the Simulator. click the Run button (⌘+R). When the Simulator launches, the elements are aligned nicely in the narrow screen size. Rotate the interface by selecting Hardware ► Rotate Left from the menu bar or pressing ⌘+left arrow, as you did earlier. The elements should resize just as you want and exactly as previewed in Figure 6-1, with the button staying in the middle of the view and the text fields resizing because you've constrained them to be a fixed distance from the side of the view.

Even though you've added six constraints to this relatively sparse view and the layout is completely satisfactory, Xcode is less than happy. At the top of Xcode, in the Activity Viewer, is a yellow warning triangle; and in the Document Outline, next to the View Controller Scene node, is a red arrow, as shown in Figure 6-15. Both of these warnings stem from a lack of constraints applied to the elements in the view.

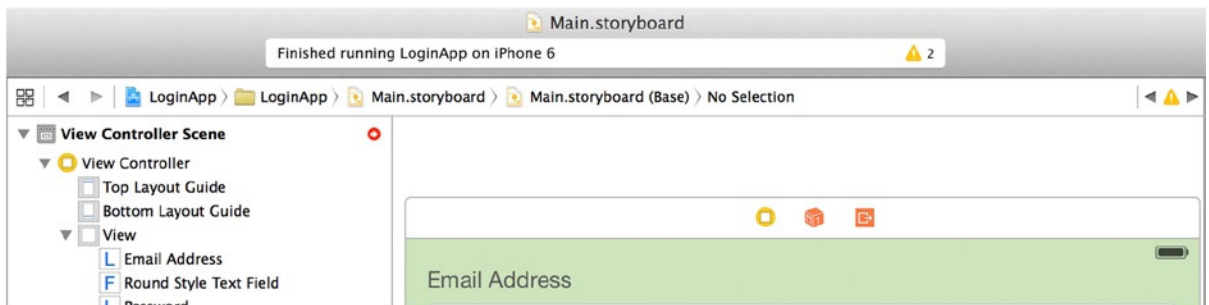


Figure 6-15. Xcode displaying two separate warnings because of a lack of constraints

Xcode wants you to specify constraints for every object in the view, including vertical and horizontal positioning, whether in relation to the view itself or to other elements in the view. When you move on to the next segment, you'll find out what constraints you need to manually add to satisfy Xcode's exacting standards.

Before moving on, let's quickly look at the attributes of a constraint. Back in the Interface Builder, select the Log In button. Now, select one of the constraint guidelines by single-clicking it; it becomes highlighted, as shown in Figure 6-16. Alternatively, with the View item inspected in the Document Outline, you can expand Constraints and select the last constraint in the list, also shown in Figure 6-16.

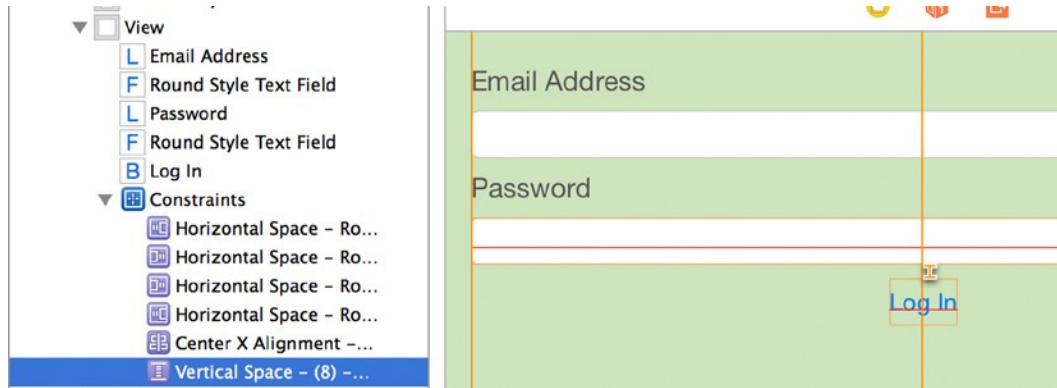


Figure 6-16. *Selecting the constraint for the button's vertical spacing*

If you look at the Attributes Inspector for a moment, as shown in Figure 6-17, you see a variety of ways to fine-tune how a constraint works. First, look at the Relation attribute, which should currently be set to Equal. You can change it to Less Than Or Equal To or More Than Or Equal To. These two options allow the constraint to be flexible, whereas Equal is a fixed value that can't be deviated from.

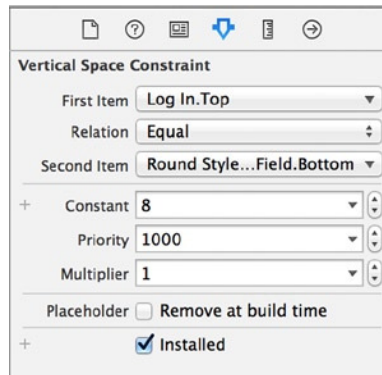


Figure 6-17. *The attributes of the constraint*

The Constant attribute contains the numerical value assigned to the constraint. Currently, if you positioned your button in the same place as mine, the value is 8. Putting these attributes into plain English, this means the text field will stay a distance *equal to* 8 points from the Password text field; increasing or decreasing this value has an impact on the spacing when the application runs. You learn more about customizing these values as the chapter and the book progress. For now; let's move on to look at some other aspects of Auto Layout and constraints.

Specifying Constraints with the Align Menu

The Control-drag method of specifying constraints isn't everyone's preferred way of working, so let's look at how to apply the same constraints using the Align menu and then the Pin menu. The Align menu is used to specify how controls align to each other and the wider view. Because the constraint

you applied to the button is an alignment constraint, you can use the Align menu to constrain it the same way you did earlier with the Control-drag method. The Align button is one of several buttons available at the bottom of Interface Builder: it's the first button in the cluster of buttons at lower right, and it resembles a small box on top of a larger one. Refer back to Chapter 4 if you need to reacquaint yourself with these buttons.

Before you use either the Align menu or the Pin menu, you need to remove the constraints you've applied to your controls. Select the view controller, either from the Document Outline or by clicking a green portion of the view, and then click the Resolve Auto Layout Issues button. Choose Clear Constraints under the All Views In View Controller heading, as shown in Figure 6-18, to remove all the constraints set in this view controller.

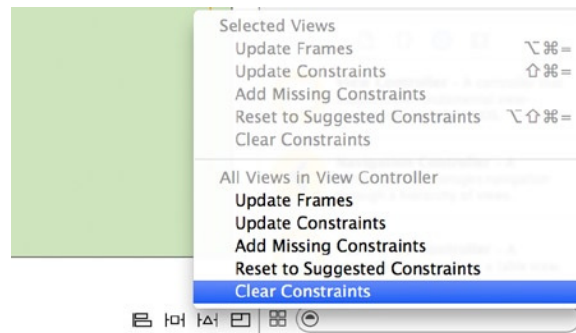


Figure 6-18. Selecting Clear Constraints from the Resolve Auto Layout Issues menu

The constraints you applied earlier are removed, and the view is ready for you to reapply them using the Align and Pin menus. First, select just the Log In button, and then click the Align button. In the menu that appears, select the check box next to Horizontal Center In Container, as shown in Figure 6-19.

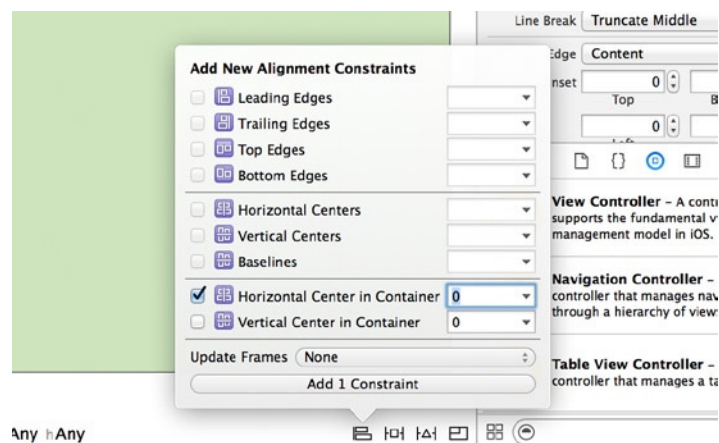


Figure 6-19. Adding a horizontal alignment constraint using the Align menu

Clicking the button currently showing Add 1 Constraint applies the constraint to the button, exactly the same as when you used the Control-drag method, but without the need to be precise with your mouse movements.

Specifying Constraints with the Pin Menu

So far, I've shown you how to center the Log In button using the Align menu. Now it's time to constrain the text fields so that their leading and trailing edges stay fixed to the side of the view's margin at all times. This technique is known as *pinning*, because you're fixing a positional attribute of the control. Therefore, it makes sense to reapply the constraints to the text fields by using the Pin menu. The Pin menu is the second button in the cluster of buttons at lower right in the Interface Builder design area; it sits next to the Align button.

What's great about the Pin menu is that you can apply constraints to both text fields in a single action. Click the Email Address text field to select it, and then hold the ⌘ key and click the Password text field to select it too. Click the Pin button, and a menu appears, as shown in Figure 6-20.

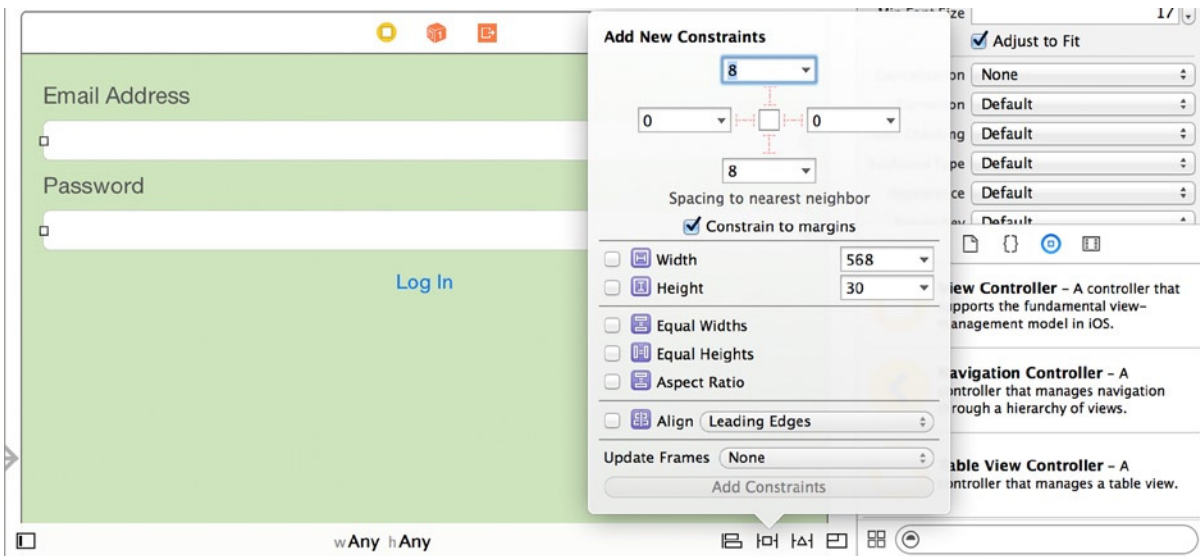


Figure 6-20. The Pin menu

Let's focus on the top area of the Pin menu, which contains four text fields with a value of either 8 or 0 in Figure 6-20. This area is used to set the spacing to the nearest neighbor. Just below these boxes, note that there is a Constrain To Margins check box that is currently checked; this is perfect because it's important to respect the margin when designing your interface. If you added an element that you wanted to be fixed to the edge of the screen, you would uncheck this box to pin the element to the side of the view.

Thinking about this specific situation, the nearest neighbor that you want to fix to is the view itself; you want to fix the leading and trailing edges of the text fields to that neighbor. Earlier in this chapter, when I discussed the attributes of one of the constraints, it had a value of 8 points. Conveniently, Xcode has anticipated that you might want to fix the leading and trailing edges; but the value in the left and right boxes that control the leading- and trailing-edge constraints is 0. Can that be right? Absolutely: it's 0 because the elements are currently positioned against the left and right margin. The Constrain To Margins check box is selected, so all you need to do here is tell Xcode that you want to apply the constraints.

At this point, it's easy to become confused. The Add Constraints button is grayed out, so how do you set the constraints? Luckily, the answer is easy. At the center of the four text fields in the Pin menu is a square shape with red I-bars going to each of the text fields; click the left and right I-bars, and they become bright red. Also notice that the button at the bottom of the Pin menu now says Add 4 Constraints, as shown in Figure 6-21.

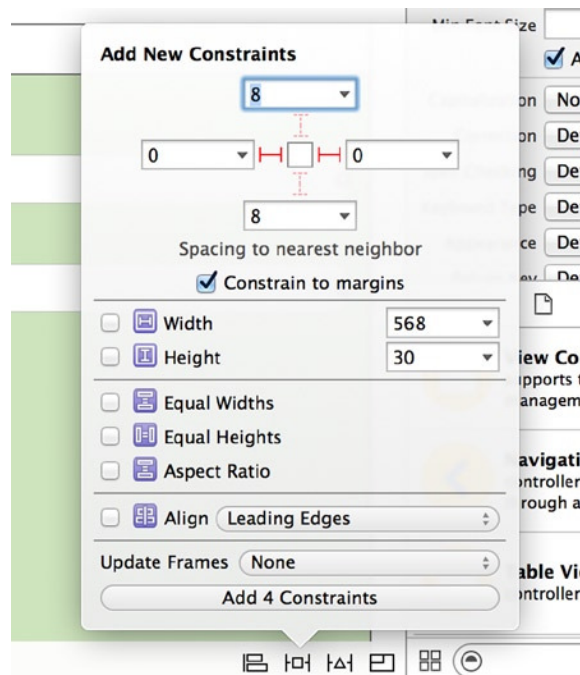


Figure 6-21. Specifying the Pin constraints for the text fields. Note the emphasized left and right I-bars

Click the Add 4 Constraints button to apply the constraints to your text fields. This is the quickest and most efficient way to add the same set of constraints to a number of controls—and sometimes, being a great developer is as much about knowing the shortcuts in your development environment as it is about knowing the code.

If you run the app now, it reacts exactly the same way it did when you used the Control-drag method. But Xcode is still not happy; to fill in the blanks and satisfy Xcode's exacting standards you will learn how to let Xcode determine the constraints for the layout.

You now know two different ways to use Xcode to manage your layout with constraints. Although next I explain how to use automatically set constraints, there will always be a need to override or add additional constraints manually; the skills you've learned here will be extremely useful when building your own applications.

Automatically Adding Constraints

So far in this chapter, the techniques you've used can definitely be classified as manual, although the process certainly hasn't been complex. Apple has gone to a lot of effort to make managing layouts even easier by providing two great methods for automatically setting constraints. It's not perfect, but it's the quickest way to put the bulk of your constraints in place with the click of a button.

In the previous section, you may have noticed that Xcode isn't happy with the constraints you applied manually in this application, even though the application functions exactly as required. This is because Xcode feels it doesn't have all the information it requires to position the design elements and is making some decisions itself. It would be far happier if you were making all the decisions.

At first, the logic of the situation can be hard to comprehend. Before you added the constraints, there were no warnings, but now that there are five *working* constraints in place, Xcode isn't happy. Fortunately, there are two ways you can find out more about why Xcode is upset, as I indicated back in Figure 6-15 the warning triangle in the Activity Viewer and the red arrow in the Document Outline. Click both now. The Issues Navigator appears in place of the Project Navigator, and a list of all the Auto Layout issues replaces the Document Outline, as shown in Figure 6-22.

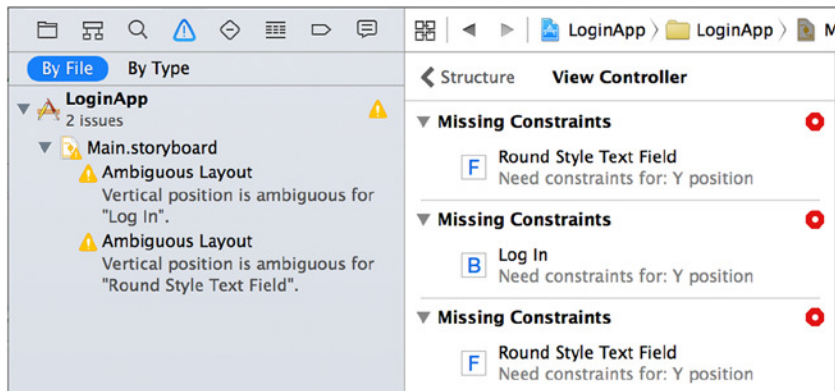


Figure 6-22. The Issues Navigator and the Document Outline showing that there are issues with the layout

Troubleshooting from the Document Outline gives you the most detail about the issues Xcode has flagged. You can see that the first three warnings, which are more serious, were triggered because you haven't yet provided all the constraints that Xcode expects for each control. All the constraints have focused on the x (horizontal) axis and ignored the y (vertical) axis. Let's address these missing constraints to satisfy Xcode. Luckily, Apple has made this incredibly straightforward in Xcode 6.

Adding Missing Constraints

The Constraint Warning Details view accessed from the Document Outline allows you to automatically resolve layout issues one by one. But in many cases, you just want to let Xcode do the fixing for you. Let's try both methods.

In the Constraint Warning Details view, click the red dot next to the first constraint warning. As shown in Figure 6-23, you can easily fix the constraint issue with a single click of the Add Missing Constraints button.

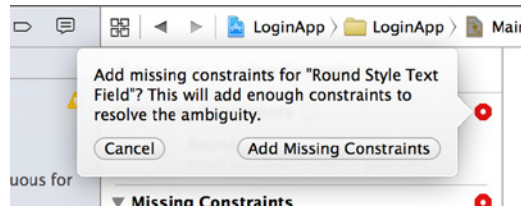


Figure 6-23. Fixing individual issues with the Constraint Warning Details view

Ironically, sometimes when you fix this issue this way, the number of issues can go up before it goes down. Keep repeating this step until all the issues are gone. That wasn't too hard, right? In the Document Outline, click the Structure button in the top-left corner to return to the standard Document Outline view. Then expand the Constraints item: Xcode has created many more than five constraints, as shown in Figure 6-24, where I have highlighted each constraint so that its position on the view is shown. Finally, run the application just to make sure everything is in place.

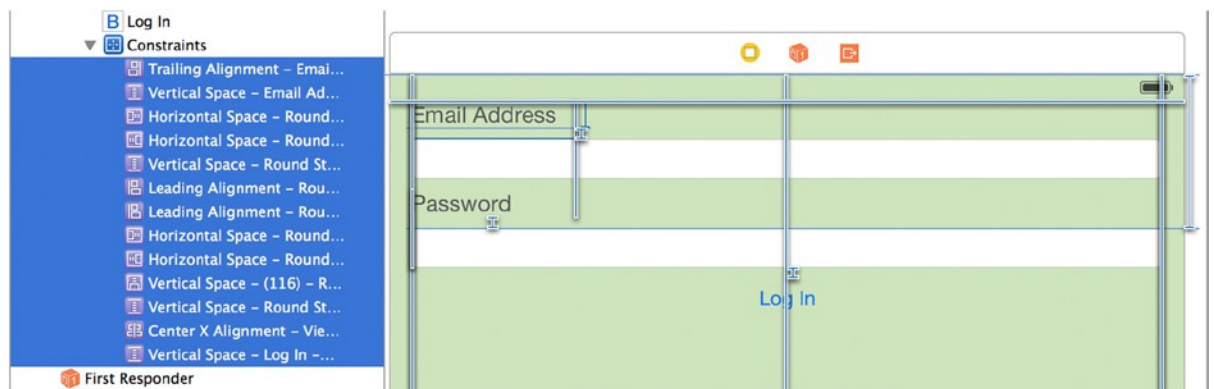


Figure 6-24. Examining the constraints that Xcode has created

Back in Interface Builder, note that the previously orange guidelines have become blue! The orange guidelines are Xcode's way of telling you that there is more to do.

Just as you've achieved a perfect set of constraints, let's wipe the slate clean and reset all of them so you can dip a toe into fully automated constraint-setting. To do this, select View Controller from the Document Outline; then click the Resolve Auto Layout Issues button, and choose Clear Constraints under the All Views In View Controller heading to again remove all the constraints that were set in this view controller.

You're back to square one as far as constraints are concerned. Now, reselect the Resolve Auto Layout Issues button, and click Add Missing Constraints under the All Views In View Controller heading, as shown in Figure 6-25. This makes Xcode look at every element in the view controller and add the constraints it feels are needed to make the layout adjust to a change in the shape of the view, such as rotating the device or using a different form factor. Because this is a universal app with size classes enabled, the constraints need to be thoroughly tested and customized so your app can be deployed to any iOS device running iOS 8.

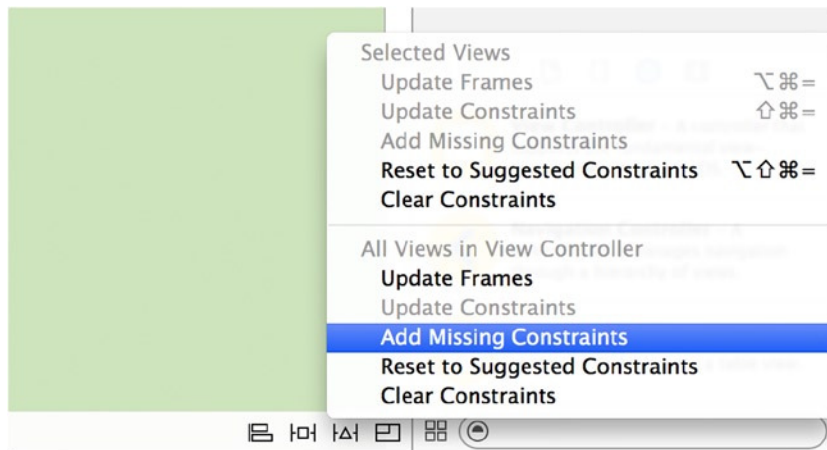


Figure 6-25. Using Add Missing Constraints to automatically set the constraints for the view

While I'm on the topic of the Resolve Auto Layout Issues menu, I want to quickly draw your attention to the Reset To Suggested Constraints option. It can be used just like Add Missing Constraints, in that even if you have no constraints, you can select this option to automatically generate all the constraints for your view. Where Reset To Suggested Constraints comes into its own, however, is when you've heavily modified your constraints and gotten into a mess: you can use Reset To Suggested Constraints to return to firm footing and restart the modification process, being more careful to test as you go and ensuring that you use the Preview facility (which I cover shortly).

Updating Constraints

In a short space of time, you've increased your knowledge and now know a number of ways to add constraints to a layout. Adding constraints is well and good, but layouts change, and constraints need updating. There are two ways to do this, depending on the severity of your changes.

In Interface Builder, move the Log In button further from the Password text field; this causes the blue guideline to turn orange, as shown in Figure 6-26. The constraint is orange because in this example, it was originally set to pin the button 8 points below the Password text field: you moved the button further from the text field, so the constraint is no longer correct. Interface Builder shows that in this instance, it has moved 12 points further from the text field, as indicated by the +12 value attached to the now-invalid constraint.

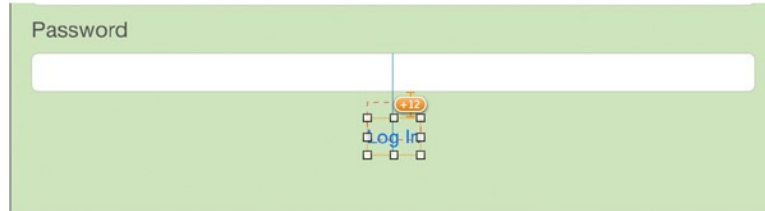


Figure 6-26. An invalid constraint highlighted in Interface Builder, showing that the constraint is off by 12 points

These types of minor interface changes happen all the time when you're tweaking your layout to be pixel perfect, and fortunately they're very easy to fix. With the Log In button selected, go back to the Resolve Auto Layout Issues button. This time, select Update Constraints (Shift+⌘+=): the constraint guideline turns back to blue, happy that it is now satisfying the terms of the constraint.

If you're in a situation where you've fine-tuned a number of elements in your layout and want to keep the constraints but with the updated values, go back to your best friend when using Auto Layout: the Resolve Auto Layout Issues button. Select Update Constants under the All Views In view controller heading, which, as it implies, updates all the altered constraints in your view controller to their correct values.

That's it for the principles of Auto Layout in this chapter. You've learned a lot about how to apply constraints, how to fix issues with your layout, and a lot more. Now let's change pace a little and examine how to preview a layout in Xcode, using size classes to change constraints depending on the form factor, and customizing text fields to create a great user experience.

Previewing Your Layout

As you've gone through this book, you've created a number of small projects that show off particular features of Xcode or iOS. But in the real world, you're potentially creating massive applications, and you may need to go through complex processes to produce a specific layout that you want to test for potential issues in real time without having to rely on the Simulator and repetitively go through the application to get to the view you're working on each time. And when you're facing a tight deadline, you need to get things done as quickly as possible. This is where previewing can come in handy.

The Preview tool is a fantastic addition that Apple built into Xcode 5—but it was not easy to find or use. Fortunately that has changed in Xcode 6. Enable the Assistant Editor in Xcode, click Automatic on the jump bar, and then mouse over Preview to expose Main.storyboard (Preview), as shown in Figure 6-27.

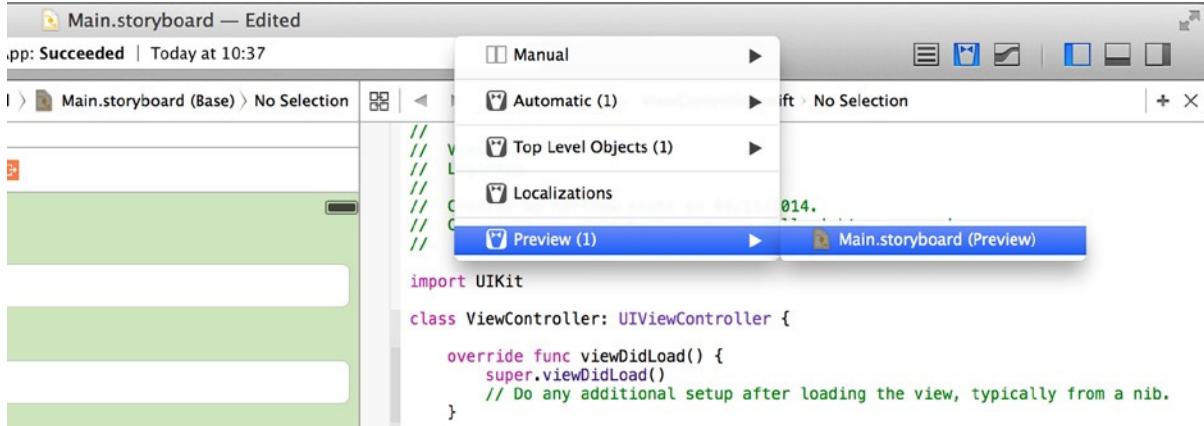


Figure 6-27. Accessing the Preview menu

You may have to resize your windows slightly to accommodate both panes. The result is that you have a preview of your layout on a specific device: in this case, a 4-inch iPhone as shown in Figure 6-28.

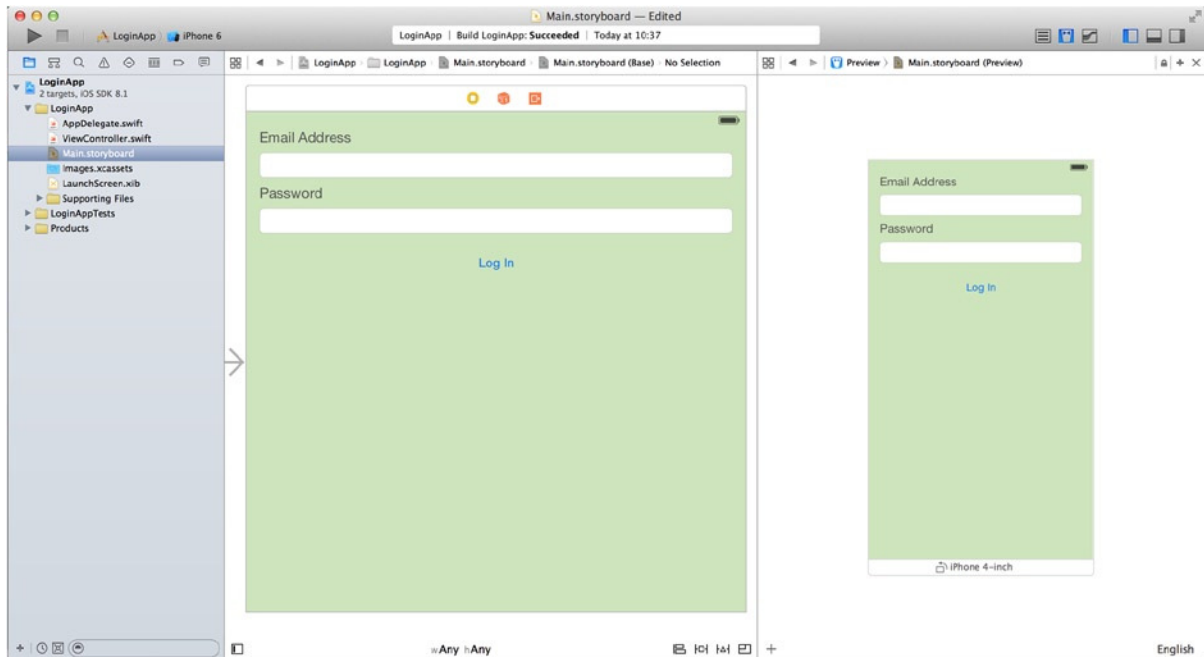


Figure 6-28. Xcode with the Preview tool enabled

You can zoom out the preview by using the standard pinch gesture on a multitouch device or by double-clicking a white area of the Preview background to toggle zoom levels. You can't make any changes to the application in the Preview tool, but anything you change in the Interface Builder is immediately reflected here.

In the bottom-right corner are two buttons. On the left is a plus symbol: clicking it allows you to add another form factor to the preview pane, meaning you can preview different devices simultaneously, greatly simplifying the process of configuring your views. On the right is a language (quite possibly English, as in my case). If you're working on a localized application, you can use this to switch between languages; or you can use a double-length pseudo-language to test how your view reacts even when you don't have any other languages set up, meaning you can get a solid interface configured from the very start.

All of these features clearly make the Preview tool a necessity for all Xcode developers. Let's make some changes to the way the application looks when it runs on the iPad without changing the iPhone version, all in a single storyboard, using size classes.

Size Classes

One of the most significant changes in application development that Apple introduced in Xcode 6, aside from the Swift programming language, has to be size classes. Through this new mechanism, there is no longer a need for separate storyboards for iPhone and iPad. All layout is done by default in a much squarer view, rather than the typically rectangular layouts provided in previous version of Xcode.

Although using size classes isn't mandatory at this point, they're certainly preferable to maintaining two storyboards, although there are instances when you would want to do this. One of the first apps I wrote was universal, but the iPhone portion was based around a tab bar controller, whereas the iPad portion adopted a very different dashboard approach. This significant difference in styles couldn't have been achieved with size classes, so it's important to remember that although they're the standard for new iOS applications in Xcode 6, they aren't your only option.

At the bottom of the design area, notice that it says `wAny hAny`, as shown in Figure 6-29. This means the interface you design here and the constraints you apply will be used regardless of the width or height of the screen.



Figure 6-29. Xcode indicates that the interface being built is for any width and any height

To understand a bit better what this means, let's use the Preview tool you just discovered to show what the interface will look like on an iPad as well as an iPhone. If you've closed the preview, reopen it as instructed in the previous section. Then click the + symbol in the bottom-left corner, and select iPad. An iPad appears alongside the iPhone. Remember to zoom out to see both devices, as shown in Figure 6-30.

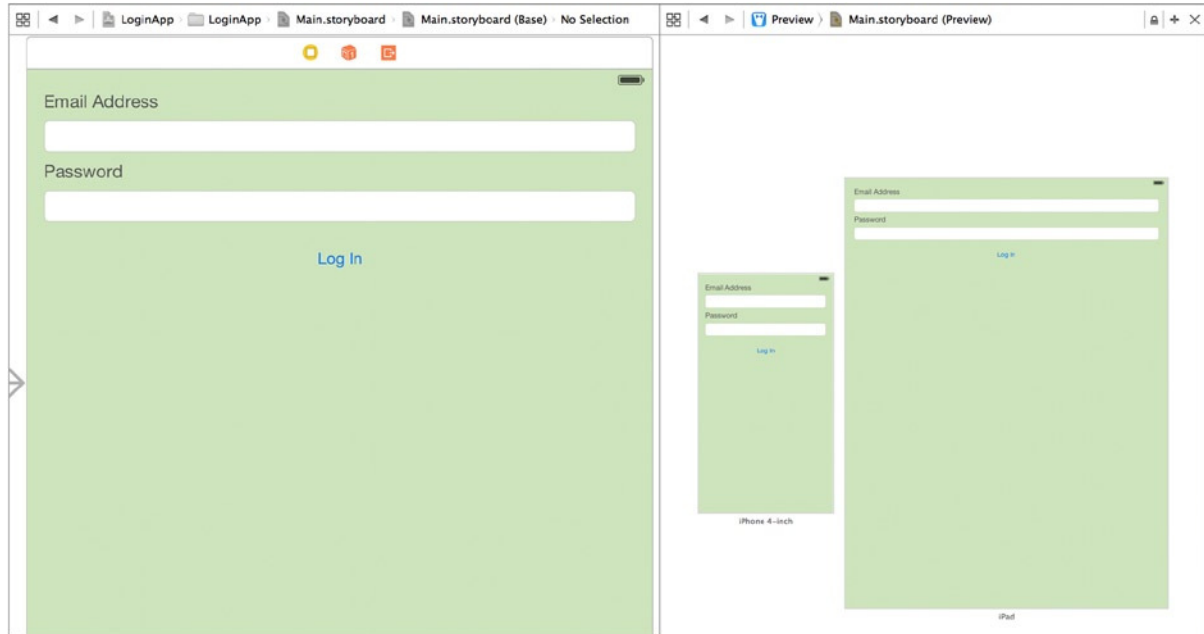


Figure 6-30. The Preview tool showing an iPhone and an iPad side by side

Although there is nothing technically wrong with the way the text fields span the view, they're realistically far too big and will add to the impression that the user is running a scaled-up iPhone app, which isn't desirable. To demonstrate the power of size classes, let's alter the layout for iPad by centering the Email Address and Password fields and making them a fixed width.

Click `wAny hAny`, and move the mouse over the grid until you get to `Regular Width | Any Height`, as shown in Figure 6-31. Note that the description below Base Values indicates that this layout is for iPads in portrait or landscape orientation.

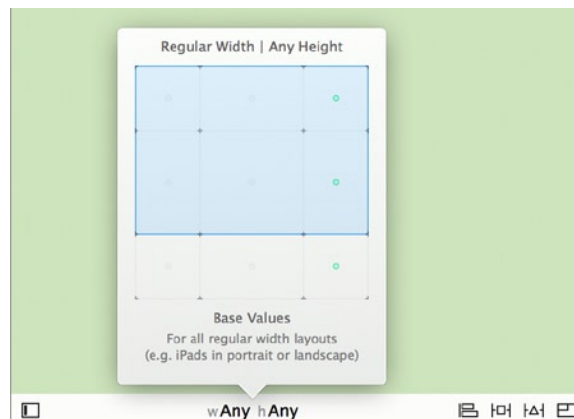


Figure 6-31. Selecting `Regular Width | Any Height` as a size class to develop the iPad layout

This is where things can get fiddly because of the width of the view and the number of panes. For now, let's experiment with moving the Assistant Editor's location. Go to **View** ► **Assistant Editor** ► **Assistant Editors On Bottom**. Immediately the preview shifts below the Interface Builder: it's obvious that this is the best way to work on an iPad layout, as you can see in Figure 6-32.

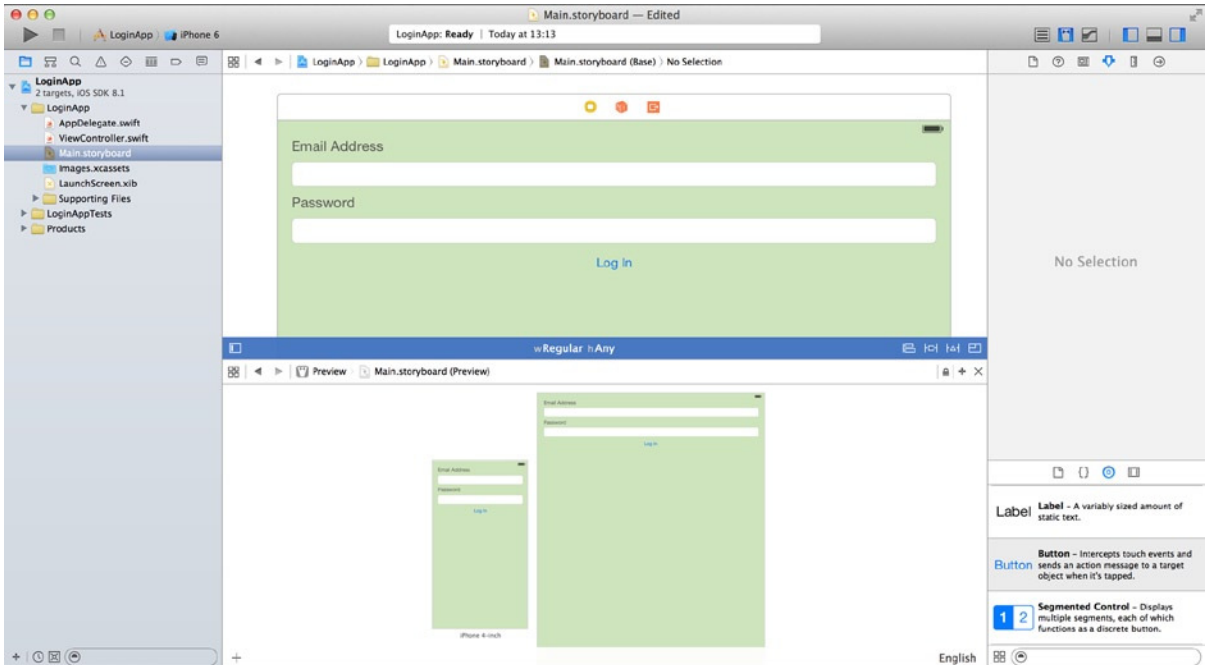


Figure 6-32. Moving the Assistant Editor below the Interface Builder

This may be a slightly daunting task, and the layout will appear to fall apart before it's finally drawn together, so stay with me while you make some significant changes:

1. Open the Document Outline, and expand the Constraints portion for the view.
2. Click the second constraint, which should be Trailing Alignment. Holding down Shift, click the bottommost Leading Alignment constraint, as shown in Figure 6-33. You've selected all the constraints that position the elements in the view that you're going to resize and reposition. You've left out the first constraint, which controls the vertical spacing from the top of the view, because that is consistent across both layouts. You've also left out the Log In button, which aligns itself to the center of the view and is a fixed distance below the Password text field; again, there is no need to change this.

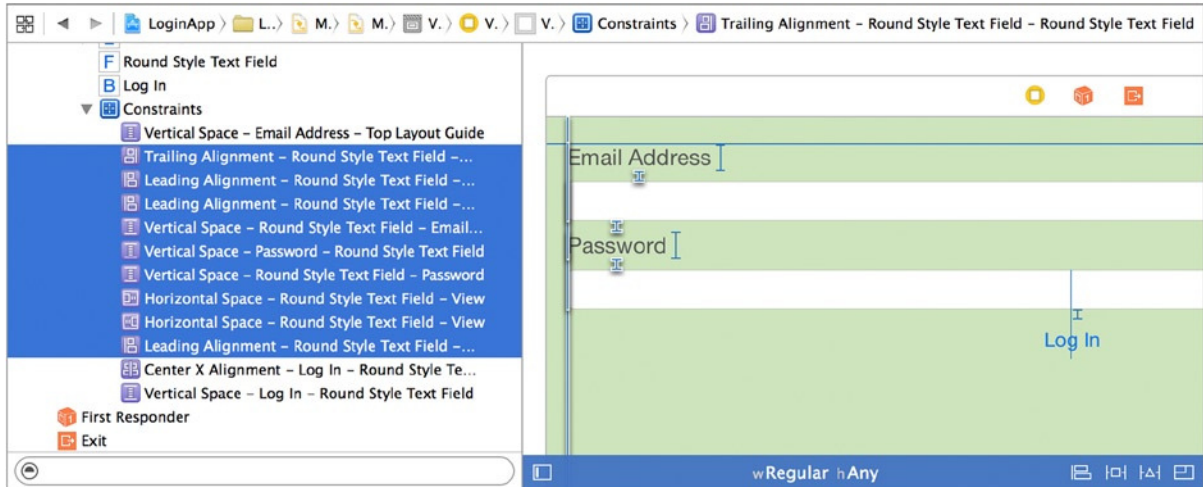


Figure 6-33. Selecting the iPhone-specific constraints

- Open the Attributes Inspector. Notice that the last item in the Attributes Inspector for a constraint is the Installed attribute. Importantly, to the left of this attribute is a plus symbol. This tiny + symbol appears next to a number of attributes and settings in the different inspectors; it's used when you want to add an exception or customization for a specific size class. Click the + symbol next to the Installed attribute, and choose Regular Width | Any Height (Current), as shown in Figure 6-34.

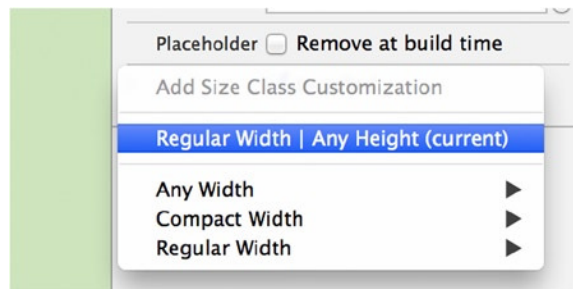


Figure 6-34. Selecting a size class for which to customize the Installed attribute

- When you've made your selection, you see a second Installed attribute: to the left, it says wR hAny. Uncheck this new Installed attribute so that it resembles Figure 6-35.

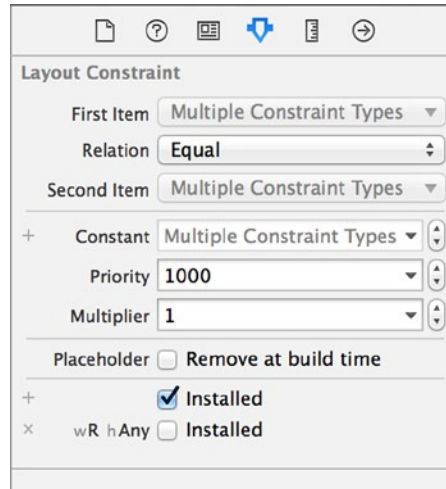


Figure 6-35. Disabling the selected constraints for the iPad's size class

At this point, the preview layout on the iPad may collapse and all the components scrunch up into the top-left corner; this is because when you uncheck *Installed* for this size class, Xcode has no idea what size the elements should be or where to put them. Although this may seem pretty catastrophic, it will be sorted out quickly.

5. Back in the Document Outline or on the view itself, hold down the \mathbb{A} key and click the two labels and the two text fields so they're highlighted as shown in Figure 6-36.

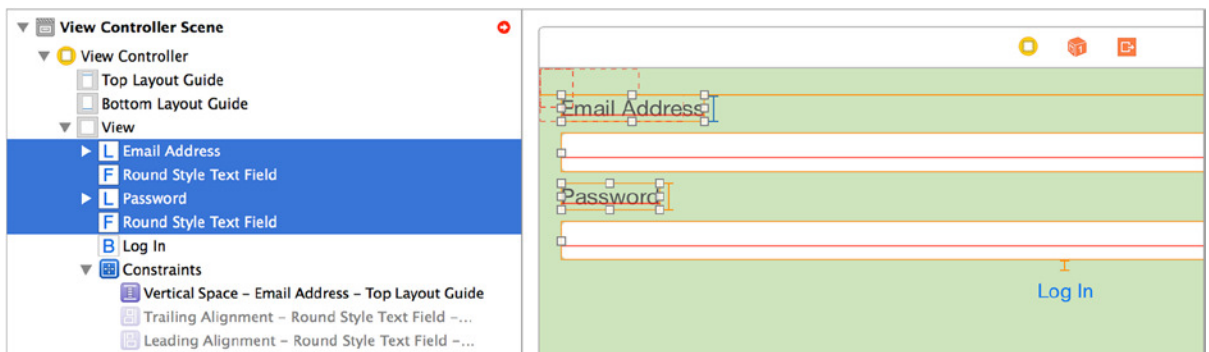


Figure 6-36. Selecting the labels and text fields in the view

- Click the Pin button. Check the Width box, and set the value to 400, as shown in Figure 6-37. This makes life easier by locking all four elements to the same width.

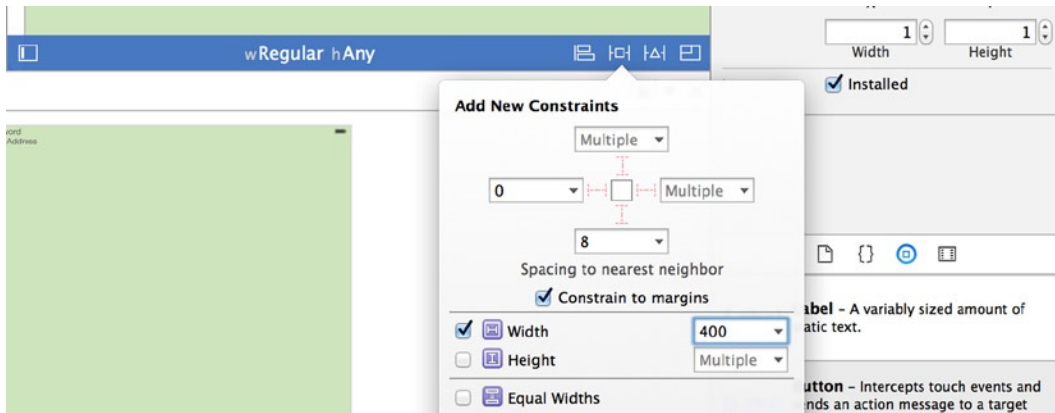


Figure 6-37. Fixing the width of the labels and text fields to 400 points

- Click the Add 4 Constraints button. Xcode continues to display a number of warnings, but these should diminish as you add more constraints.
- Let's sort out the positioning. With the four elements still selected, click the Align button and then check Horizontal Center In Container, as shown in Figure 6-38. Click Add 4 Constraints. Your preview still looks horrendous, but you're a couple of clicks away from finishing.

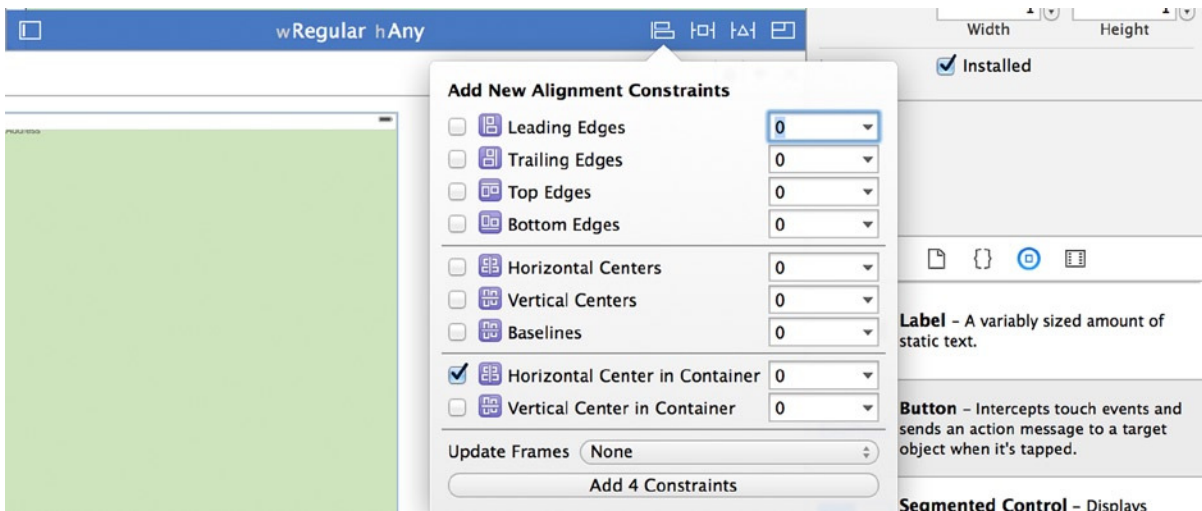


Figure 6-38. Centering the four elements in the view with constraints

9. Tap a green area of the view in the Interface Builder to deselect those four elements. The only element that is missing in terms of constraints is the vertical spacing between labels and text fields; fix this by going to Resolve Auto Layout Issues and choosing Add Missing Constraints under the All Views In View Controller, just as you did back in Figure 6-25.
10. If everything has worked as planned, your layout for the iPad should snap into place as shown in Figure 6-39. There are still Xcode warnings, because the elements being displayed in the Interface Builder are in a different position due to the constraints you've added. To resolve this, again select Resolve Auto Layout Issues and choose Update Frames under the All Views In View Controller heading. The layout changes to match the one shown in the preview.

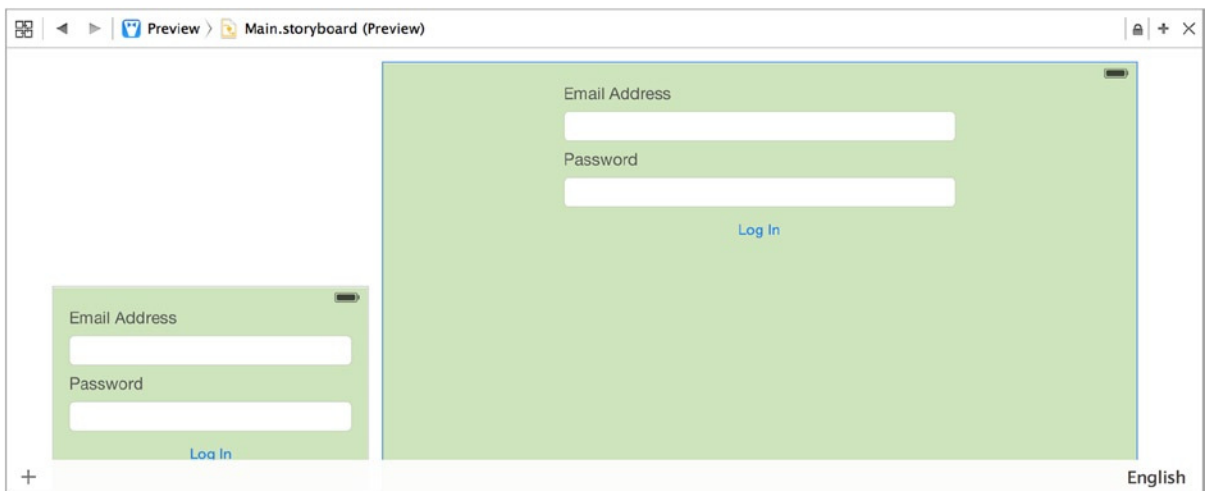


Figure 6-39. The Preview tool showing two different layouts created by using size classes

11. Change to the iPad Air Simulator, and run the application. Feel free to rotate the virtual device and see how the layout stays true to your constraints, regardless of the orientation. What's more, if you change the size class back to wAny hAny, the old layout and constraints are exactly where you left them.
12. You need to know how to disable size classes. In the storyboard, open the File Inspector, and scroll down until you see the Use Size Classes check box. Don't uncheck it on this occasion, but if you need to in the future, you know where it is.

Finishing Touches

You're getting close to the end of this chapter. With the text fields you've added to your view, you have an opportunity to look at how you can use the Interface Builder to create a tailored experience that makes it easy for users to fill out this form.

Customizing Text Fields

Even though you only have a couple of text fields in the view, you can apply a wealth of customizations to make the form fit the purpose. You can also add some neat features so that your users can fly through it in an intuitive manner.

Hiding Passwords

How seriously you take security in your application can make or break it on the App Store, so you need to make sure the basic features a user expects to see in a password-protected app are in place. Therefore, the first customization concerns how to set your Password text field to behave like a typical password field by obscuring the user's password as they type it, which requires absolutely no code at all:

1. Close the Assistant Editor by clicking Standard Editor.
2. Select the Password text field in the Interface Builder, and then open the Attributes Inspector.
3. Scroll down the list of attributes until you see the Secure Text Entry check box; select it, as shown in Figure 6-40.

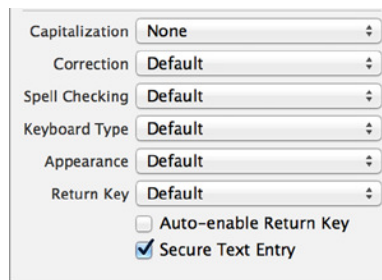


Figure 6-40. Making the Password text field secure

4. Run the application in the Simulator, and try typing in the Password field. You see that as in other applications, the password is obfuscated as you type it.
5. Another behavior you may want to add to the Password field is automatic clearing of the contents when you tap in the field. This is useful when you can't see what you've typed. Go back to the Interface Builder, and back to the Password field's Attributes Inspector. The Clear Button attribute section has a Clears When Editing Begins check box; select it, and change the drop-down option above it from Never Appears to Appears While Editing.
6. Rerun your application in the Simulator. Type in a password, click the Email Address text field, and then click back into the Password field. The contents should be cleared, ready for you to have another go at remembering the password. You can also clear the field while editing.

Configuring a Text Field for E-mail Addresses

You've configured the Password field to fit with your users' expectations, but what can you do to make the Email Address field easier to use? Quite a bit, as it happens. Getting the user interface right can go a long way toward making your application a hit on the App Store. These may seem like small changes, but in an oversaturated market, having an immaculate, intuitive interface can make a big difference:

1. Select the Email Address text field, and open the Attributes Inspector.
2. The first thing you want to do is emphasize that users should type an e-mail address. You do this by adding placeholder text to guide them. In the Placeholder text field, I typed **E.g. matthewknott@me.com**, but you can type whatever you want.
3. The placeholder text you type is immediately reflected in the text field, as shown in Figure 6-41.

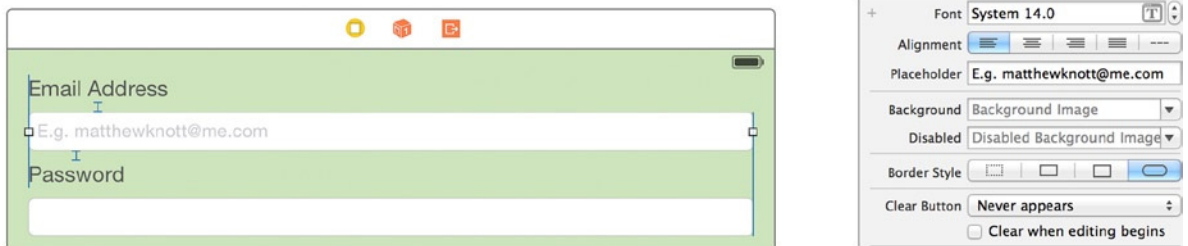


Figure 6-41. Setting placeholder text in the Attributes Inspector

4. Think back on how you cleared the Password field when editing. You should make it easier for users to clear the Email Address field, but it shouldn't be the default action. Go to the Clear Button section in Attributes Inspector, and change the drop-down option from Never Appears to Appears While Editing. With this option set, the user is be given the opportunity to clear the field whenever they're focused on it.
5. Run your application in the Simulator, and type something in the Email Address field. As you can see in Figure 6-42, the Clear button (an X inside a circle) appears nicely, which is great. What's not so great is that it's suggesting I've misspelled my e-mail address, which is annoying.

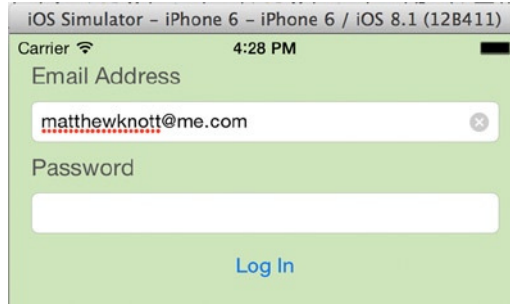


Figure 6-42. The Clear button appears when the user edits the e-mail address. But, annoyingly, it's reporting spelling errors

6. Back in the Attributes Inspector, find the Correction attribute, and select No from the list of options. Below that, change Spell Checking to No as well. Now iOS will ignore the spellings of e-mail addresses.
7. Another common feature that users expect and value when entering an e-mail address is having the keyboard presented in a way that gives priority to common keys such as @. iOS has a number of options for configuring the keyboard, and all are available from the Keyboard attribute, found directly below the Spell Checking attribute. Figure 6-43 shows the great variety of context-specific keyboard options Apple provides by default, which in turn let you, the developer, make life that much easier for your users. In this case, choose E-Mail Address from the list.

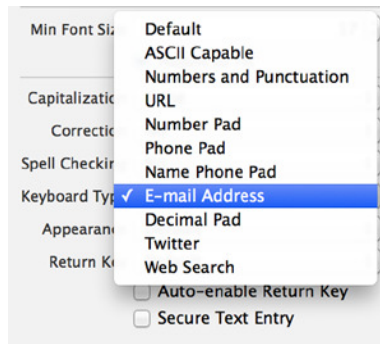


Figure 6-43. Choosing the E-mail Address keyboard options from a very rich list of context-specific keyboard options

8. Rerun the application in the Simulator, and tap into the Email Address field. The keyboard shown to the user makes it easier to quickly type an e-mail address without having to go hunting for the @ symbol.

You're well on your way to having happy users. There is just one more tweak I want to add to make this application as easy as possible to navigate.

Navigating Forms

A lot of what you've looked at regarding customizing text fields has focused on simplifying the experience for the user. This final example is no exception, but it requires you to write some code, even though I said this was a code-free chapter. I lied (sorry!).

The two text fields don't present much of a burden to the user in terms of navigating, but with a couple of lines of code, you can add the icing on the cake by allowing the user to navigate through the text fields using the keyboard. This makes it easier for users to complete the fields in double time. The technique learned here can be scaled up to larger forms, where your users will really appreciate it:

1. Open the Assistant Editor, and ensure that the `ViewController.swift` file is loaded. If you want the Assistant Editor located back on the right, go to **View > Assistant Editor > Assistant Editors On Right**.
2. Control-drag a connection from the Email Address text field to just below the class declaration, and create an outlet named `usernameField`.
3. Do the same for the Password text field, and name this outlet `passwordField`.
4. Switch back to the Standard Editor. You need to make some changes to the interface before moving on to the implementation file.
5. When the user selects the Email Address field, you want them to see a Next button instead of the Return button so that they can tap it to move to the Password field. To set this, select the Email Address text field, and, in the Attributes Inspector, look for the Return Key attribute. Select Next from its list of options.
6. Scroll down the list of attributes until you find the View section and, specifically, the Tag attribute. Enter **1** as the attribute value. Tags are integer values that are used to identify different elements in a layout when you look at them in code. If you have 30 text fields in your view, the only way in code to differentiate one `UITextField` from another is to examine its tag.
7. Select the Password field, and change its Return Key attribute to Done and its Tag attribute to 2. Run the application in the Simulator to see these buttons in action, as shown in Figure 6-44.

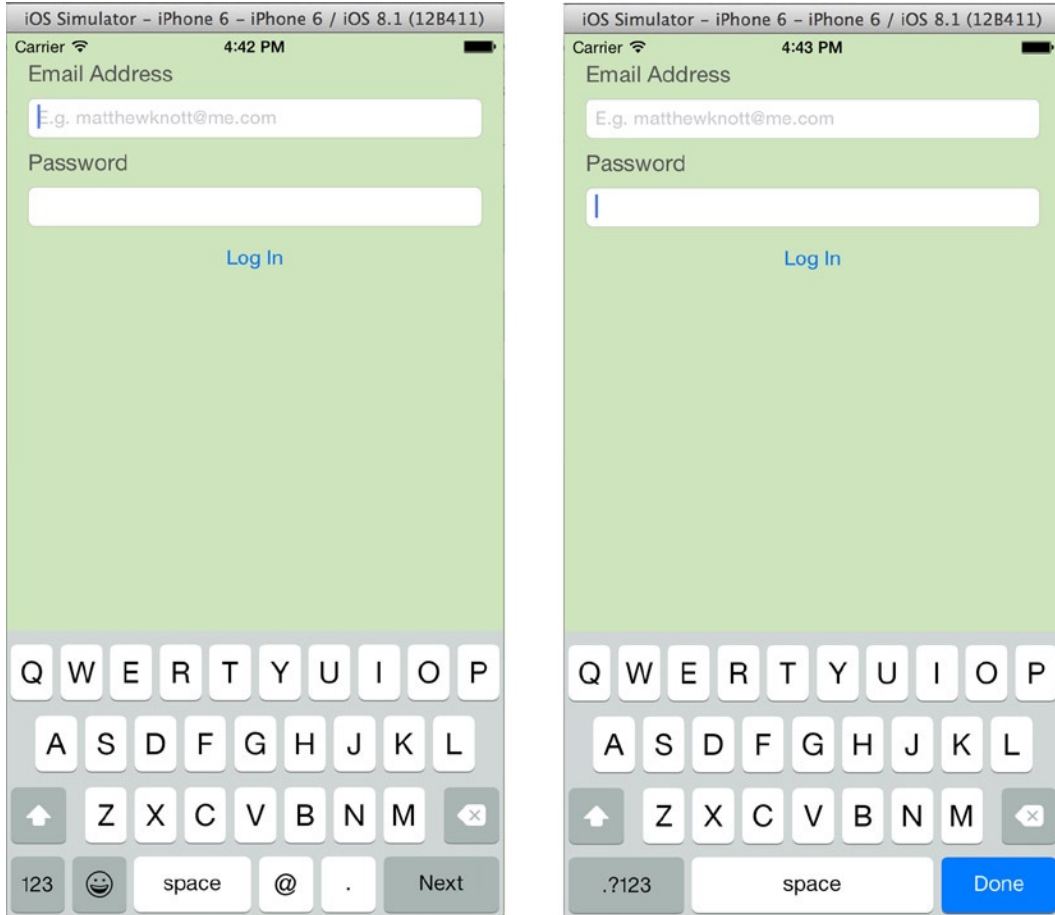


Figure 6-44. The keyboard changes depending on which field is selected

8. The buttons look good but don't currently do anything. Open `ViewController.swift` from the Project Navigator, where you add the functionality needed to finish this application.
9. Add the `UITextFieldDelegate` protocol to the class by adding the highlighted code to the class declaration:

```
import UIKit

class ViewController: UIViewController, UITextFieldDelegate {

    @IBOutlet weak var usernameField: UITextField!
    @IBOutlet weak var passwordField: UITextField!
```

10. Go to the `viewDidLoad` method, and add the following highlighted code to specify that the view controller is the delegate for the text fields:

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Do any additional setup after loading the view, typically from a nib.
    usernameField.delegate = self
    passwordField.delegate = self
}
```

11. You need to add the `textFieldShouldReturn` method, as you did in Chapter 4, to understand what happens when the user presses the Return key, whether you've configured it to say Next or Done. Drop down a few lines after the `viewDidLoad` method, and type the stub for the method as follows:

```
func textFieldShouldReturn(textField: UITextField) -> Bool {
}
```

This method returns a Boolean value and shows an error until that is returned; for now, ignore it.

12. You want the method to work out the tag for the next field, assuming it's one more than the current `textField.tag` property, and assign it to a variable called `nextTag` for analysis. Add this highlighted code:

```
func textFieldShouldReturn(textField: UITextField) -> Bool {
    let nextTag = textField.tag + 1 as Int
}
```

13. You need to see if there is an element in the view with a tag that matches the `nextTag` integer value so that you can either move to the Password field or dismiss the keyboard. You do this by creating an instance of the `UIResponder` class and setting it by going to the current text field's parent view and searching based on the tag. Do that by adding this line of code to your method:

```
func textFieldShouldReturn(textField: UITextField) -> Bool {
    let nextTag = textField.tag + 1 as Int
    var nextField : UIResponder? = textField.superview?.viewWithTag(nextTag)
}
```

14. The following `if else` statement determines the course of action based on whether the `nextField` object was found:

```
func textFieldShouldReturn(textField: UITextField) -> Bool {
    let nextTag = textField.tag + 1 as Int
    var nextField : UIResponder? = textField.superview?.viewWithTag(nextTag)
```

```

    if let field : UIResponder = nextField {
    }
    else
    {
    }
}

```

15. If the current field is the Email Address field, the code in the first set of braces is executed. In this case, you want to focus the cursor in the password field so that the user can type their password. To do this, make that field the first responder by adding the following highlighted code:

```

func textFieldShouldReturn(textField: UITextField) -> Bool {
    let nextTag = textField.tag + 1 as Int
    var nextField : UIResponder? = textField.superview?.viewWithTag(nextTag)

    if let field : UIResponder = nextField {
        field.becomeFirstResponder()
    }
    else
    {
    }
}

```

16. The else code executes if the current field is the Password field. In many cases you would execute your login routine at this stage, but because you don't have one, let's just dismiss the keyboard by resigning the first-responder state. After the if else statement, you need to return a Boolean value indicating whether the use of the Return key in iOS should insert a line break, which it should not. Type the following highlighted code to complete the method:

```

func textFieldShouldReturn(textField: UITextField) -> Bool {
    let nextTag = textField.tag + 1 as Int
    var nextField : UIResponder? = textField.superview?.viewWithTag(nextTag)

    if let field : UIResponder = nextField {
        field.becomeFirstResponder()
    }
    else
    {
        textField.resignFirstResponder()
    }

    return false
}

```

Run your application in the Simulator. You should be able to tap into the Email Address field and navigate to the Password field using the Next button; then use the Done button to dismiss the keyboard. The great thing is that you can use the same `textFieldShouldReturn` method whether you have 2 text fields or 30, as long as you make sure each field has a unique tag value that is one more than its predecessor.

Summary

Whatever your feelings were about constraints, Auto Layout and size classes before this chapter, I hope that now you feel at least a bit more confident about how to manipulate them with Xcode. Specifically in this chapter, you have done the following:

- Learned about how Auto Layout has changed in Xcode 6
- Manually added constraints with the Control-drag method as well as with the Align and Pin menus
- Learned how to add missing constraints and much more with the Resolve Auto Layout Issues menu
- Added a layer of polish to your text fields with the Secure and Placeholder attributes
- Used a small piece of code to take control of how the Return key works

You've covered a lot this chapter, but hopefully it's all contributing to an application you're working on or giving you the confidence to begin writing that application you've been thinking about for months.

The next chapter focuses on storyboards: the visual approach to building applications through Xcode that lets you create large portions of your application without the need to write a single line of code.

Part **2**

Diving Deeper

Storyboards

Chapter 6 for the most part took a break from writing code to look in detail at Auto Layout, Xcode’s system for arranging layouts and specifying how they react to changes in form factor or orientation. It also explained how to craft a tailor-made user experience by customizing keyboards and text fields.

You’re now in the second part of this book, “Diving Deeper,” and you’ll see that demonstrated from the outset as you get into the nitty-gritty of building a complex multiview application using storyboards. First I present the background of storyboards and the concepts behind them, and then you see the key feature of Xcode storyboards—the segue—and how to make the most of segues when rapidly creating applications.

So far in this book, in every chapter but Chapter 1 you’ve created an application as a context for your journey through Xcode. The only difference with this chapter’s project is that you’re building it over this *and* the next chapter. The reason is that you’re creating a functional Twitter client for iPad. You begin by laying out and connecting the views using storyboard techniques, but the client is built around customized table views (the subject of Chapter 8). There’s a lot to cover, and I rely heavily on two Apple-provided frameworks that take a lot of the pain out of communicating and authenticating with Twitter: the Accounts and Social frameworks.

Although this project won’t have all the bells and whistles you might expect from a full Twitter client, you can still choose from multiple accounts, see a full Twitter feed, and compose and post tweets. To get a flavor of what this application will look like, see [Figure 7-1](#).

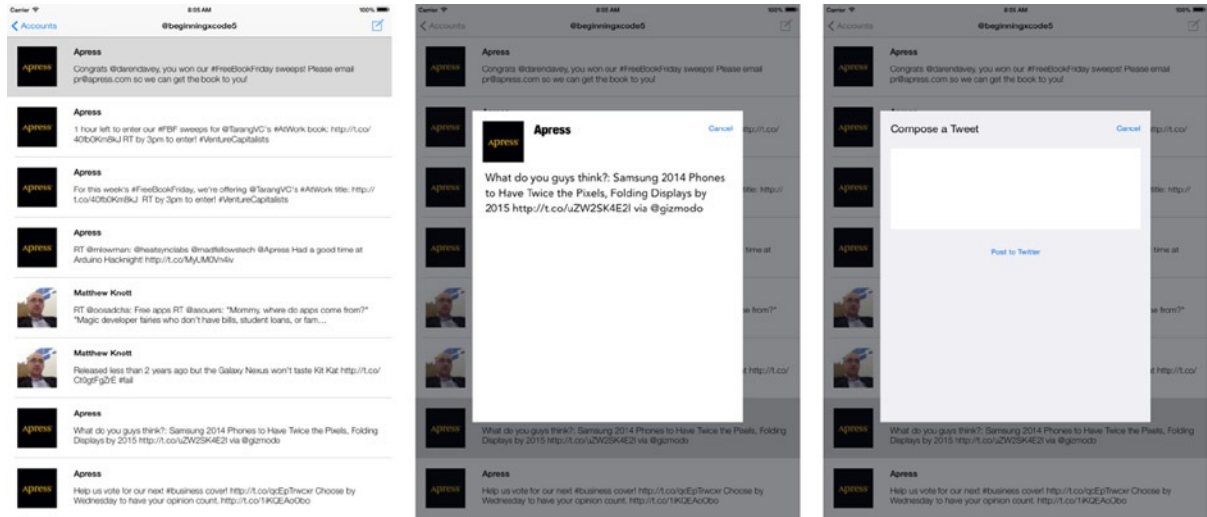


Figure 7-1. Some of the key screens in SocialApp, your functional Twitter client

A Brief History of Storyboards

Apple introduced the storyboard approach to building iOS apps with iOS 5. Although it was initially seen as a novelty, it has grown to where it is now: the preferred system for application development used with the majority of Xcode's iOS templates.

Storyboards aren't a new concept; they have been used by developers for decades. They're used today as part of the software planning and prototyping process. When my team and I are brainstorming for a particular solution, we storyboard using whiteboards or flip-chart pages because it's a way to rapidly express relationships between web pages or views in a mobile application. It's this system for rapid prototyping that Apple has successfully captured in Xcode 6—but Apple has taken it to another level allowing for agile application development.

Outside of development, storyboards originally came from the world of cinema. They were developed by Walt Disney Studios in the 1930s to plan out animations scene by scene, which is a process that is still used today even in major motion pictures. Although there are parallels in terminology, the major difference between storyboards in animation and storyboards for software development is that in animation, the progress of the story is linear: scene B always follows scene A. But in software development, this is rarely the case: perhaps scene A links to scenes B and C, with scene B linking to scene D and scene C linking to scene E, which links back to scene A. It's because of the complexity of designing a multipage application's user experience that storyboards are so valuable. The thing that movie storyboards and application storyboards share is their ability to show us the bigger picture without having all the footage—or, in our case, code.

For the SocialApp Twitter application developed in this chapter and the next, I've made a basic storyboard using a graphics package, as shown in Figure 7-2.

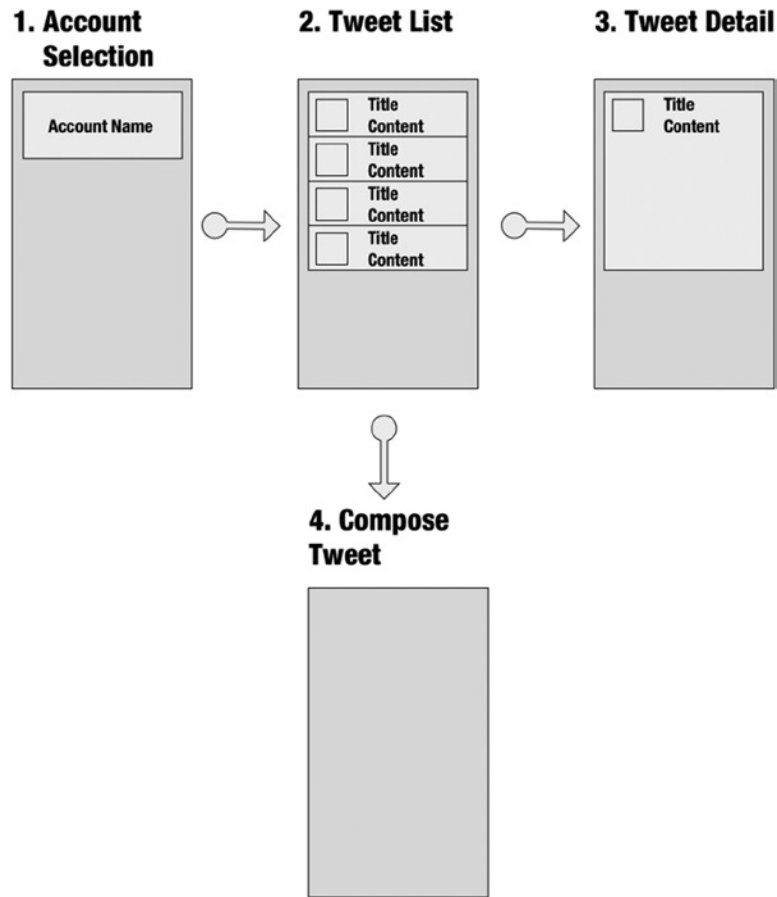


Figure 7-2. The basic composition of *SocialApp*, the example Twitter client

This is the process that my team of developers and I go through when we start thinking about the composition of an application and the functionality we want to add. We use this storyboarding process to explore ways of assembling those functions in a way that results in an easy-to-use application. When dealing with storyboards in Xcode, Apple refers to the views as *scenes*. As you can see in Figure 7-2, this app has four scenes, and one scene leads to another. Before you begin developing this app, let's look at each scene in more detail:

1. *Account Selection:* Today people often manage several Twitter accounts, so the first scene is a *grouped style* table view controller that lists each account available on the device.
2. *Tweet List:* Once the user has selected their preferred account, you want to show the 20 most recent tweets on that user's timeline. These are displayed in a *plain style* table view with a custom table cell.

3. *Tweet Detail*: The user can see more details about the tweet and its author in the Tweet Detail scene. This is based on a *standard* view controller and lists the user's name, their avatar, and the full tweet content in a text view.
4. *Compose Tweet*: Accessed from the compose icon in the Tweet List, this *standard* view controller uses a text view to compose a tweet and then posts it to Twitter on behalf of the user.

If you want to reference the scenes in this storyboard to the actual application screenshots shown in Figure 7-1, the first screenshot is scene 2, Tweet List; the second is 3, Tweet Detail; and the third is 4, Compose Tweet.

Now that you know a little more about storyboards, their origin, and how they're used by developers every day, it's time to begin putting this application together. This chapter focuses on laying out the scenes in the storyboard and putting the connecting segues in place, as well as embedding navigation controllers and creating the custom classes behind the view controllers, so let's get started.

Creating a New Project Called SocialApp

Before I get into adding in the finer details of the interface, you need to create the project and then lay out the views for this application:

1. Open Xcode, and create a new project by clicking Create a New Xcode Project from the Welcome screen or going to File ► New ► Project (⌘+Shift+N). Select the Single View Application template, and click Next.
2. Name your project SocialApp, and ensure that the targeted device is set to iPad, not iPhone or Universal. Configure the other settings as you've done in previous applications so they match Figure 7-3 (again substituting your name for mine), and click Next.

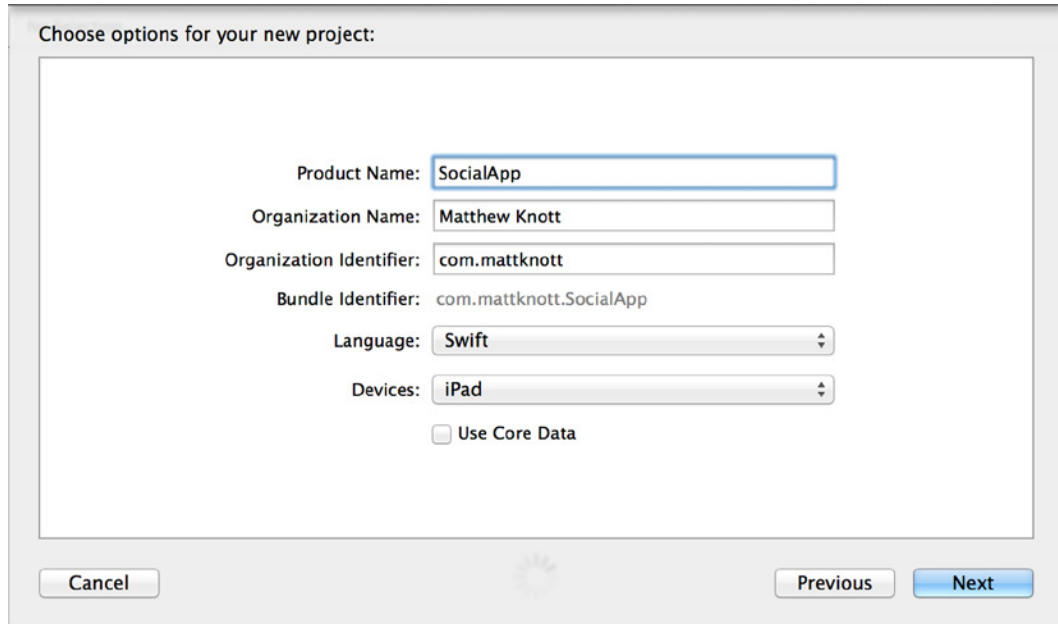


Figure 7-3. Setting the project options

3. You don't need to create a Git repository this time, so leave that option unchecked and make sure your project will be saved in the right place. Then click Create.
4. As should be familiar by now, you're ready to begin your application in earnest. The focus of this chapter is storyboards, so it makes sense to open `Main.Storyboard`. Select it from the Project Navigator.

This is the first time you've created an iPad app, but due to Size Classes (introduced with iOS 8 and Xcode 6) the view hasn't changed from when you were working on iPhone apps. Although this isn't why Apple introduced Size Classes, it means iPad apps are much easier to develop because the views occupy less screen real estate.

Tip If at any time you want to get a bird's-eye view of the burgeoning storyboard, zoom out by using pinch and zoom on a trackpad, by double-clicking an empty part of the design area, or by pressing `⌘+⌘+Shift+[`.

Note When your zoom level is less than 100%, you can still create segues between scenes and reposition them, but you can't add controls to your views. For that, you need to be zoomed in to 100% or greater.

At this point in the process, you may need to refer back to the initial layout storyboard created for Figure 7-2. As I explained at that time, the first scene in the application is a table view controller that lists the available accounts. You could add a table view to the default view controller that was added to the storyboard, but it's easier to add a completely separate table view controller:

1. The table view controller is the third item in the Object Library. Drag one onto the design area and drop it next to the existing view controller, as shown in Figure 7-4. You may need to move it around a little to get a tidy design area.

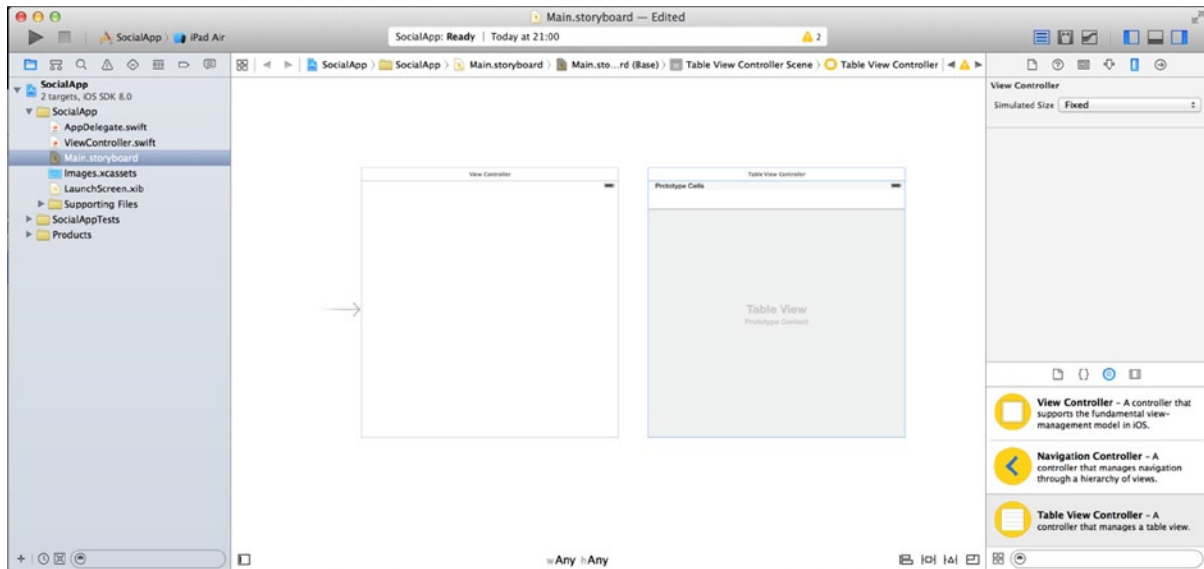


Figure 7-4. The table view controller scene, next to the initial scene the project was created with

2. Before you add any more scenes, let's run the application in the simulator. Click the Run button on the Toolbar, or press $\text{⌘}+\text{R}$. Notice that the default view controller is loaded instead of the table view controller, and there is no obvious way of accessing the Table View Controller.
3. Quit the simulator, and return to Xcode.

The default view controller is the starting point because there is an arrow pointing to the left side of it, known as the *starting arrow*. As you might expect, you can drag and drop the starting arrow onto the table view controller. When the arrow is over the table view controller, the scene becomes highlighted in blue, as shown in Figure 7-5. The starting arrow now points to the table view controller, just as it once pointed to the default view controller.

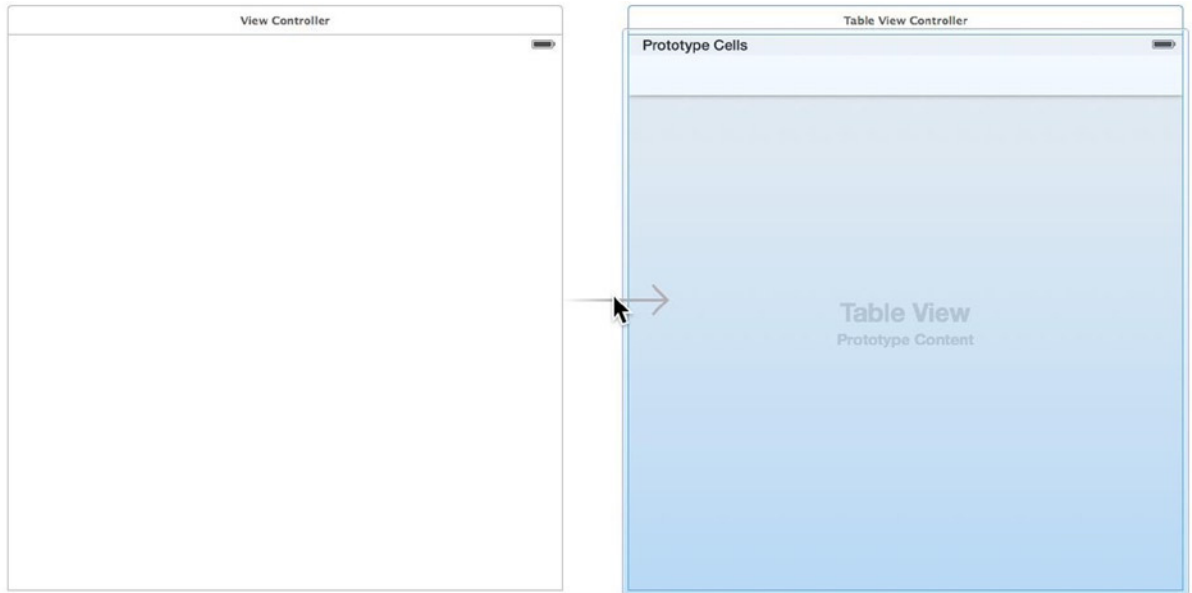


Figure 7-5. The starting arrow being hovered over the table view controller

If you run the application again, you should be greeted with an empty table view. It was that easy to change the starting point for this application! Before storyboards, you would have had to modify the application delegate to tell it which view controller to start with; now you can just drag and drop a visual aid.

If you want to be a bit more precise when setting the initial scene, and you don't want to drag and drop the starting arrow, there is another way to do it:

1. Select the table view controller in the design area, either by clicking the scene while zoomed out or by selecting Table View Controller from the Document Outline.
2. Open the Attributes Inspector, and look down to the View Controller section. Notice that **Is Initial View Controller** is select. Unselect it, and the starting arrow disappears! You'd better bring it back; otherwise the application will run with a black screen.
3. It would be a good idea to set a title while you're here. To do so, click in the Title box and set the title to **Accounts**.

There is a final way to make the table view controller the initial view controller: by deleting the default view controller. Let's do that now, as well as delete its code files:

1. Select the blank view by clicking its scene while zoomed out or by selecting View Controller from the Document Outline, as shown in Figure 7-6.

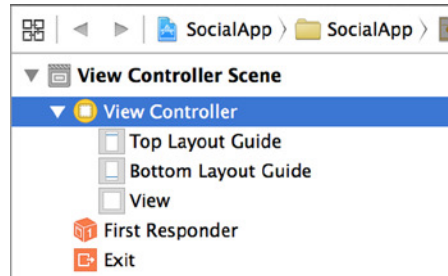


Figure 7-6. Selecting the initial view controller from the Document Outline

2. Delete the view controller by pressing the Backspace key or by selecting Edit ► Delete.
3. You need to remove the file that Xcode added for this view controller. Using the Project Navigator, select `ViewController.swift` and, again, press the Backspace key or select Edit ► Delete.
4. You're presented with the dialog shown in Figure 7-7, giving you options for file deletion. The Remove References button removes files from the project but leaves them in place in the project folder on your Mac. In this case you want to delete the file altogether, so select the Move to Trash option.

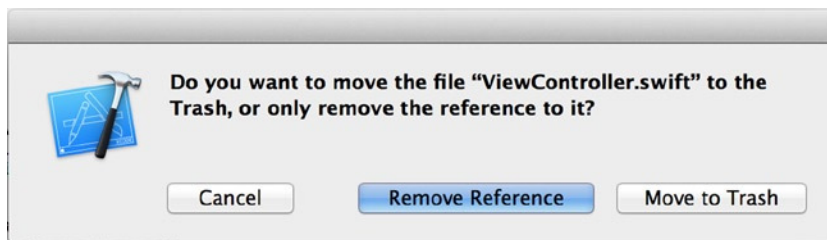


Figure 7-7. The dialog presented by Xcode when removing files via the Project Navigator

Congratulations: you've removed all the unnecessary files and views from the project! You're now going to step away from the storyboard for a moment to create four custom view controllers for the views by subclassing either `UITableViewController` or `UIViewController`.

Creating View Controllers

You can add as many view controllers to the storyboard as you like. But if you don't tie them to a view controller class file, the application will be extremely limited, because you'll have no way of interacting with the view controllers or controlling them using code. Therefore, let's break away from the application and focus for a moment on the design pattern you're using. It's called model-view-controller, more commonly shortened to MVC.

The Model-View-Controller Design Pattern

Using Xcode with the OS X and iOS SDKs is one of the most natural environments for developing using the MVC principle. First, you don't have to configure Xcode for MVC. Xcode was built from the ground up for MVC, and all the application templates except the Empty Application template are set up using the MVC principle. Second, the semantics of the terminology are completely logical:

Model: An object that stores data in a structured way. Core Data lets you create data models to interface with stored data. You can also create custom classes to represent objects, such as a vehicle class, which might have a type property, a wheels property, a make property, and many more.

View: Unsurprisingly, consists of your views, as laid out in your storyboard. The view should be all the visual elements of an application, held in isolation from any code.

Controller: The part that manages the views and the models. It acts as an intermediary between the two, taking information from the model and using it to coordinate changes in the view.

SocialApp currently has a view, and you know you're going to add several more. Before you do, let's create all the view controllers so that when you add the views, you can tie them directly to a controller. All your view controllers subclass either UITableViewController or UIViewController to create an individual class file for each view. You've done this in previous chapters, but this time you need to create four view controllers of different types.

Subclassing UIViewController

UIViewController is the class name given to the standard view controller that has been used so far in all the applications in this book. When you declare a class that subclasses UIViewController, you type, for example,

```
class MyCustomViewController: UIViewController
```

This says that the view controller called `MyCustomViewController` *subclass*s `UIViewController`. *Subclassing* means taking on all the attributes of another class, but with the ability to add your own methods and properties and override others. Two of your views subclass `UIViewController`. The two view controllers will be called `TweetViewController` and `ComposeViewController`. First, let's create `TweetViewController`, and then see if you can repeat the process for `ComposeViewController`:

1. Right-click the `SocialApp` group in the Project Navigator and select `New File`, as shown in Figure 7-8; or select the `SocialApp` group and press `⌘+N`.

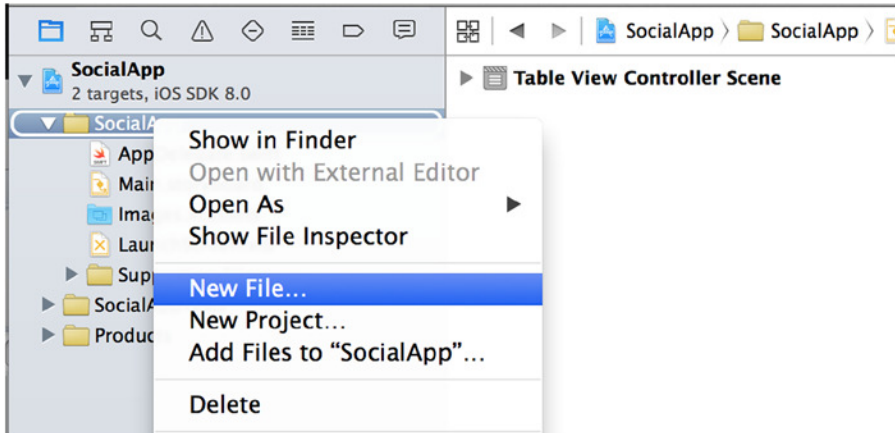


Figure 7-8. Adding a file to the `SocialApp` group in the Project Navigator

2. You're presented with the file template selection screen. Ensure that `Source` is selected from the list under the `iOS` heading, select `Cocoa Touch Class`, as shown in Figure 7-9, and click `Next`.

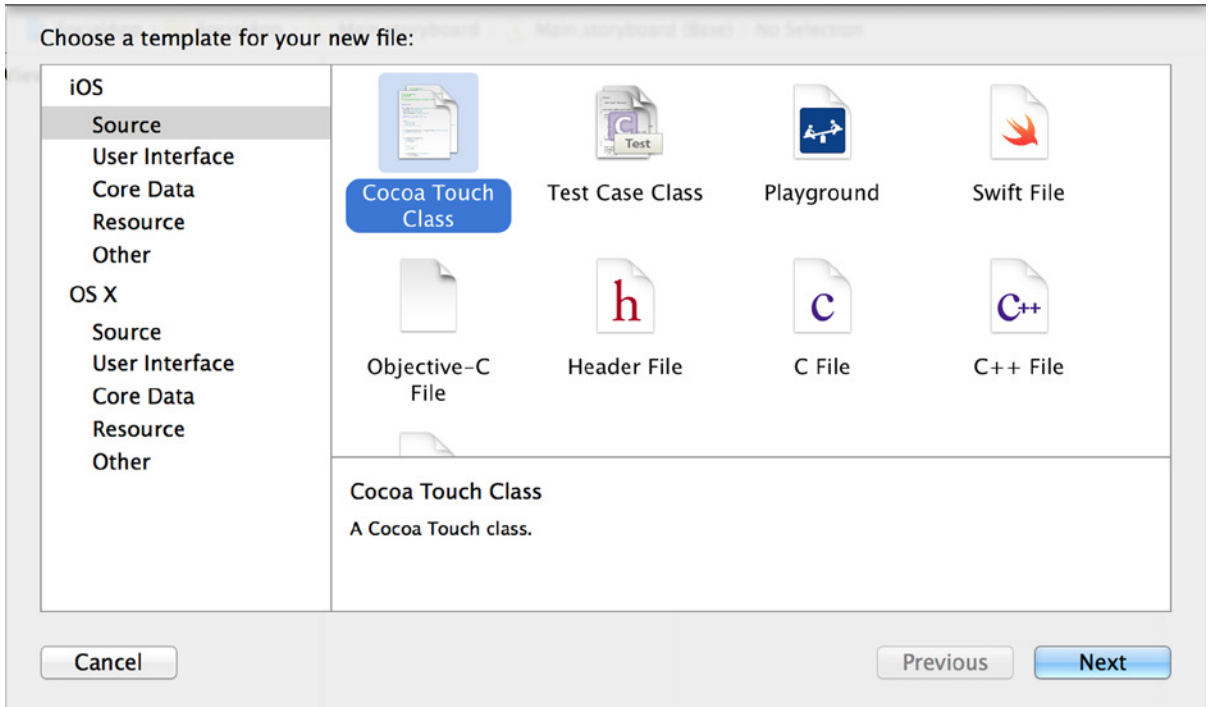


Figure 7-9. Selecting the Cocoa Touch Class file template

3. Set the Subclass Of field to UIViewController. This defaults the Class field to ViewController, making it easy for you to change it to TweetViewController. Be sure the Also Create XIB File check box is unchecked, as shown in Figure 7-10, and click Next.

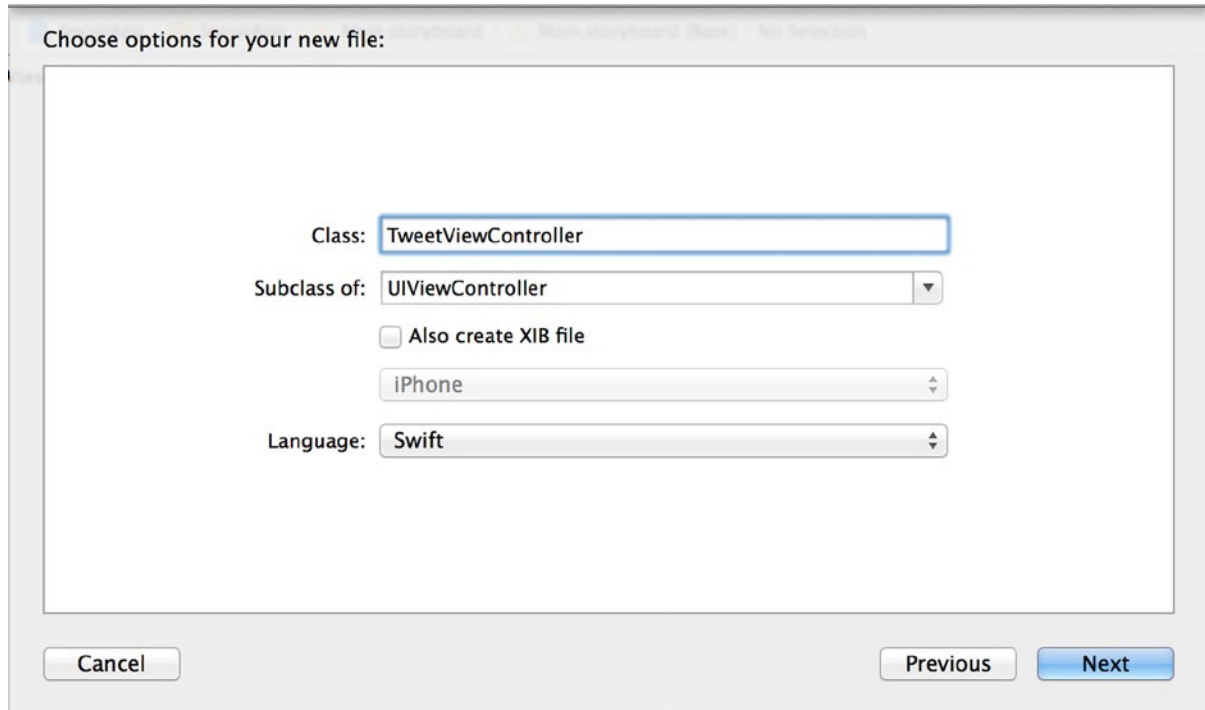


Figure 7-10. Setting the options for the *TweetViewController* file

4. You need to choose a location to save the file. Xcode automatically suggests the project folder, which is what you want. Be sure Group is set to SocialApp and that the SocialApp target is selected, as shown in Figure 7-11, and click Create.

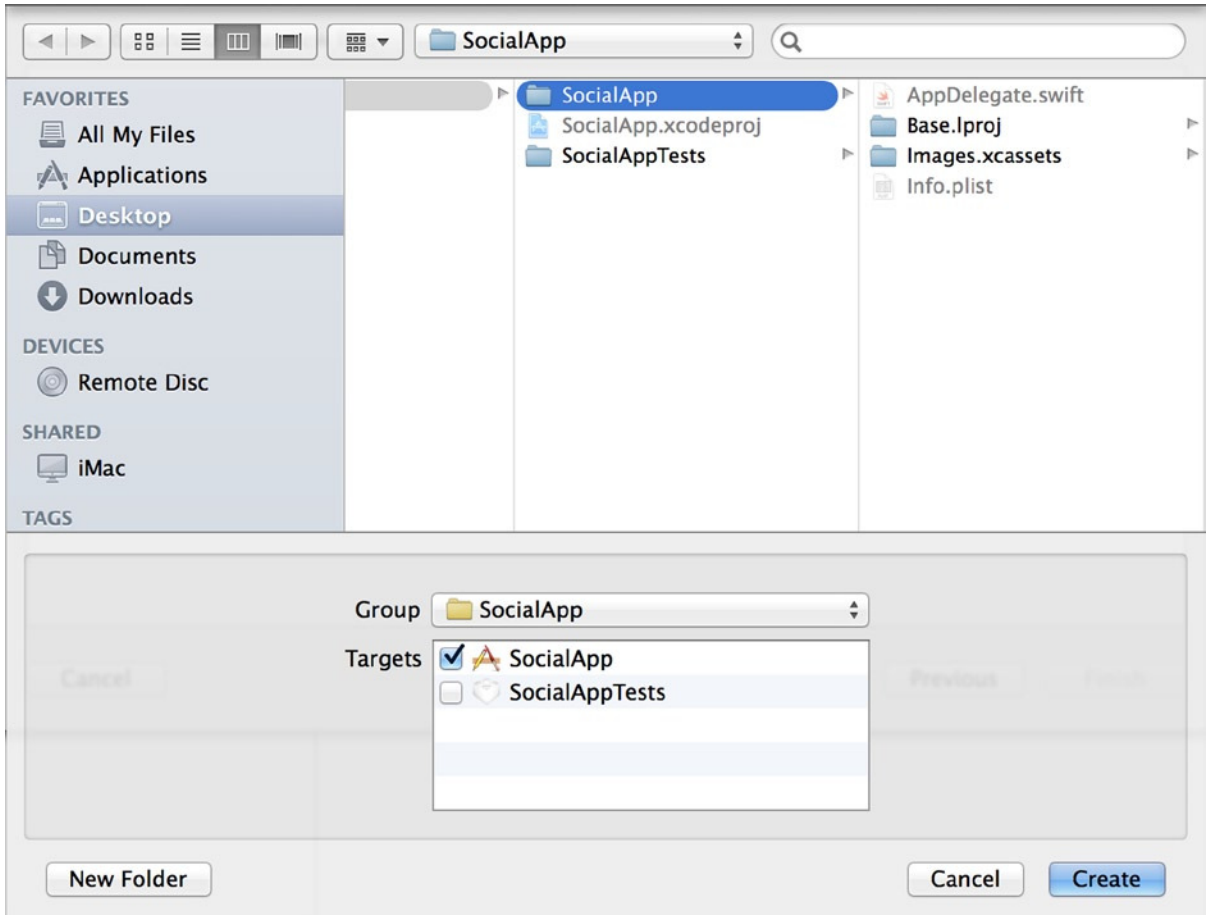


Figure 7-11. Choosing a save location and specifying the group and target

You're returned to Xcode, where you can see that `TweetViewController.swift` has been added to the project. Great! That's one view controller; now repeat this process for `ComposeViewController`. When you're done, your Project Navigator should resemble that shown in [Figure 7-12](#).

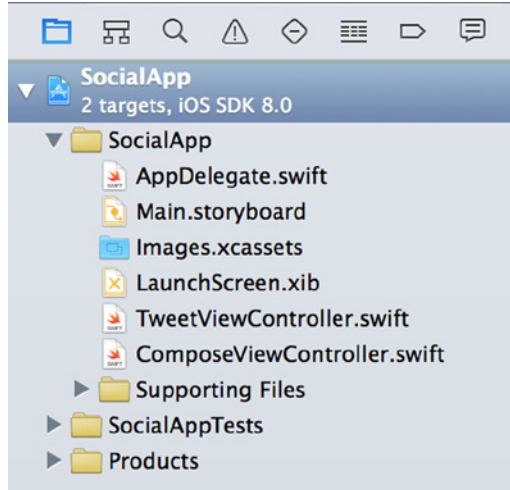


Figure 7-12. The growing project in the Project Navigator

Well done! Creating view controllers and subclassing other classes and objects is a core task for developing applications, so you're perfecting a valuable skill.

One thing you can foresee, looking at Figure 7-12, is that the file list is growing, and you still have three more view controllers to add. You need to tidy up the structure by grouping the view controllers together:

1. At the bottom of the Project Navigator, type **View** in the Show Files With Matching Name filter to make sure you only see your view controllers.
2. Click the first view controller file (in my case, it's `TweetViewController.swift`). Holding the Shift key, select the last file (in my case, `ComposeViewController.swift`). Both view controllers should be selected, as shown on the left in Figure 7-13.

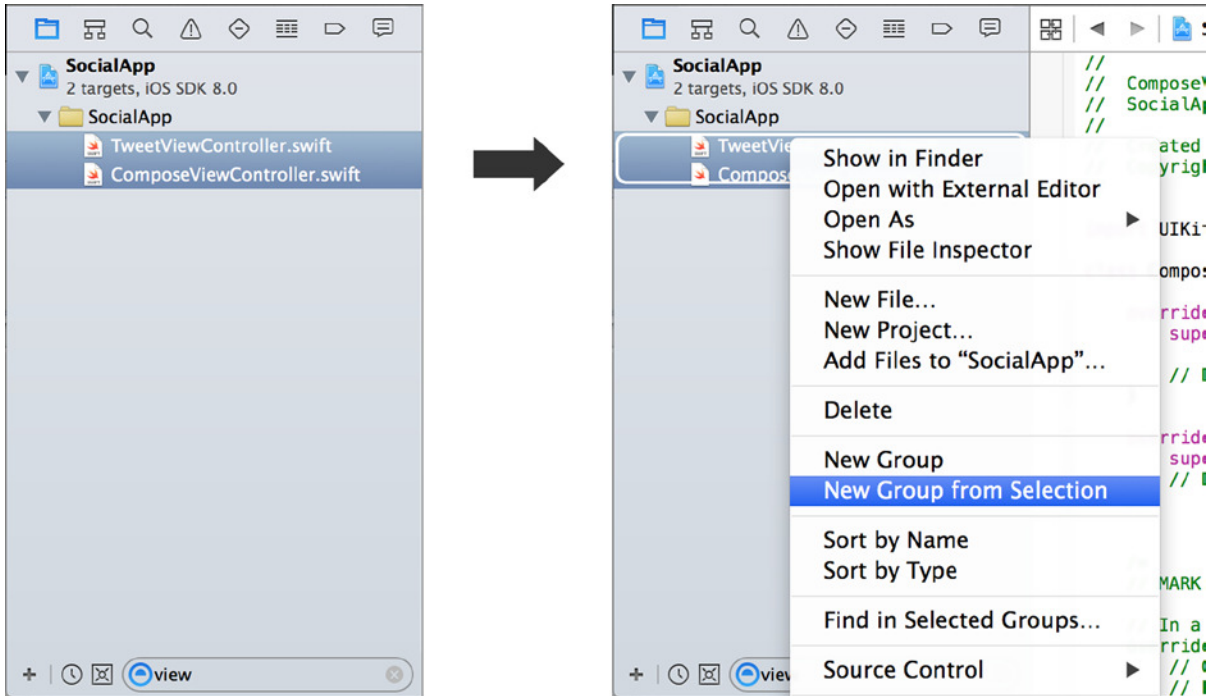


Figure 7-13. Selecting the view controllers, and creating a new group to contain them

3. Right-click the selected files, and choose New Group from Selection, as shown on the right in Figure 7-13.
4. When prompted, name your new group View Controllers. Clear the filter by clicking the X at the end of it. You should be left with a neat project, as shown in Figure 7-14.

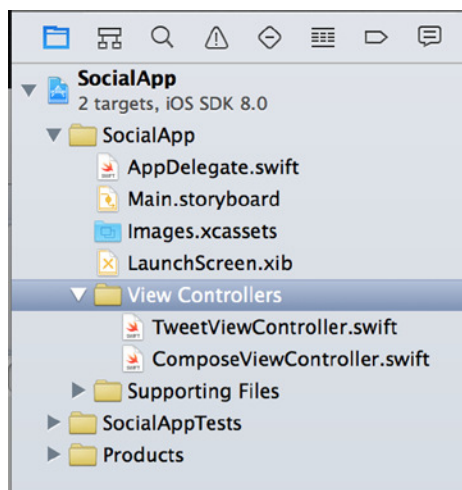


Figure 7-14. The view controllers grouped neatly together

Organizing your project neatly and logically ensures the admiration of your development team colleagues. They *will* thank you for it. Organizing files in big projects means if someone else has to work on your project, they don't have to hunt for the custom classes or the view controllers, because you've applied logic and good housekeeping to your project structure.

Subclassing UITableViewController

You've created two of the four view controllers, and it's time to create the remainder. The process is more or less the same, but there are some subtle changes. UIViewControllers are fairly straightforward: the views themselves are blank canvases, ready for you to add controls; and their methods are also very minimal, giving you just a `viewDidLoad` function and a handler for low memory. UITableViewController, however, is a more complex system, designed for displaying large amounts of data through a structured interface. It has a number of intrinsic attributes that result in the code files containing methods that are used to control the number of rows, sections, and more. I explain these in detail in Chapter 8; for now, let's just create them so you can get back to the storyboard.

The process is largely the same as before, but this time you start by selecting the View Controllers group instead of SocialApp in the Project Navigator. You need to create the two instances of UITableViewController that this application uses, called AccountsViewController and FeedViewController:

1. Right-click the View Controllers group and select New File, as shown in Figure 7-15, or press ⌘+N. You're presented with the file template selection screen.

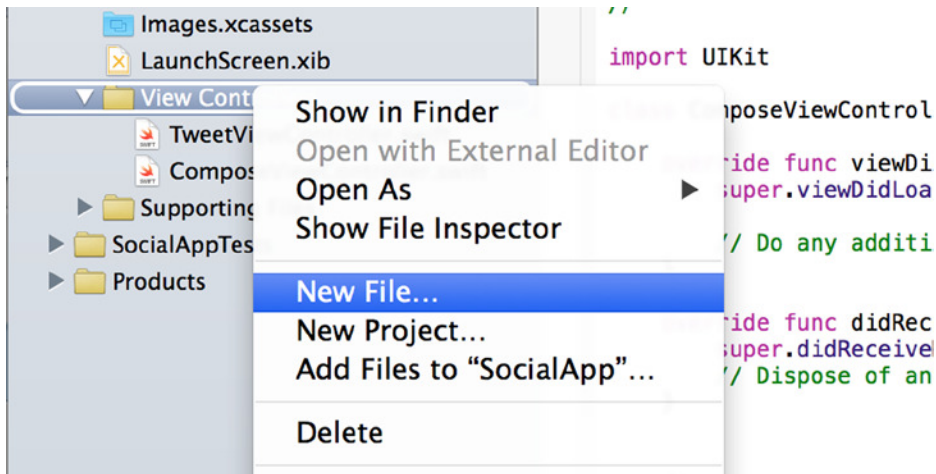


Figure 7-15. Adding a new file to the View Controllers group

2. As with the UIViewControllers, select the Cocoa Touch Class file template, and click Next. Set the Class field to AccountsViewController and the Subclass Of field to UITableViewController, as shown in Figure 7-16, and click Next.

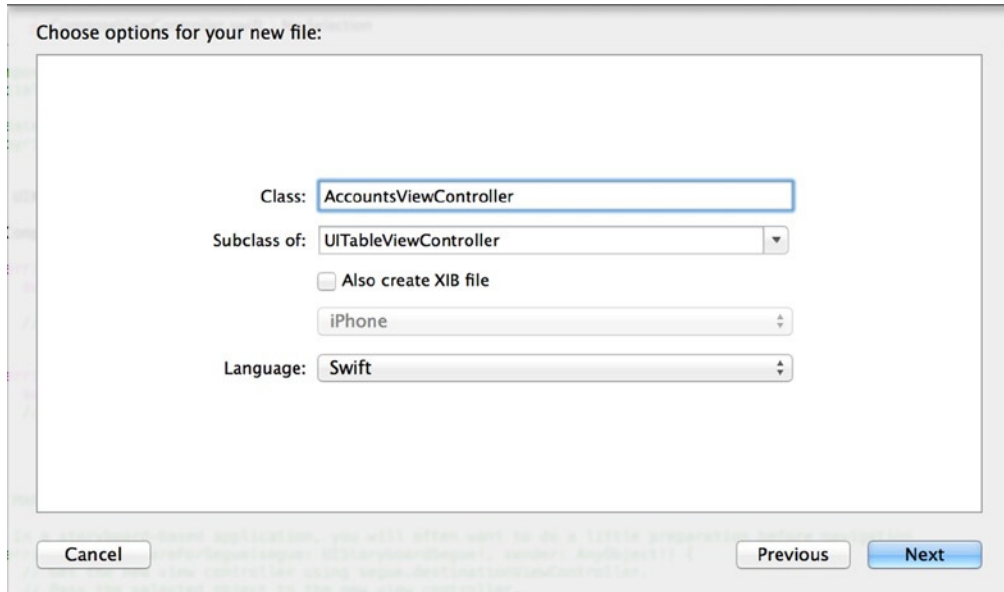


Figure 7-16. Specifying the options for the `UITableViewController`

3. Accept the suggested file location, and click Create.
4. You should be a master at creating new view controllers, so repeat these steps and create another `UITableViewController` called `FeedViewController`.

You've now created four view controllers. Let's finish adding the views to the storyboard and tie them to their respective controllers. Your Project Navigator should look something like the screen shown in Figure 7-17.

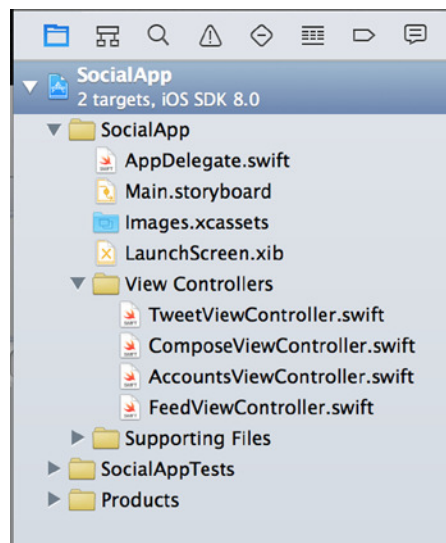


Figure 7-17. The Project Navigator with all the view controllers nicely organized

Pairing the View to the Controller

I've explained the fundamentals of the MVC design pattern on which these applications are based. You've created the controllers, and next you need to add the views to the storyboard and tie them to their specific view controller by using the Identity Inspector from the Utilities bar:

1. Open `Main.Storyboard`, and select `Accounts` from below `Accounts Scene` in the Document Outline. When setting the class of a view, you need to select its view controller before you apply the class; otherwise things get messy. That's why you use the Document Outline to make sure of your selection.
2. Open the Identity Inspector from the Utilities bar, or press `⌘+⌘+1`. Xcode should resemble the screen shown in Figure 7-18.

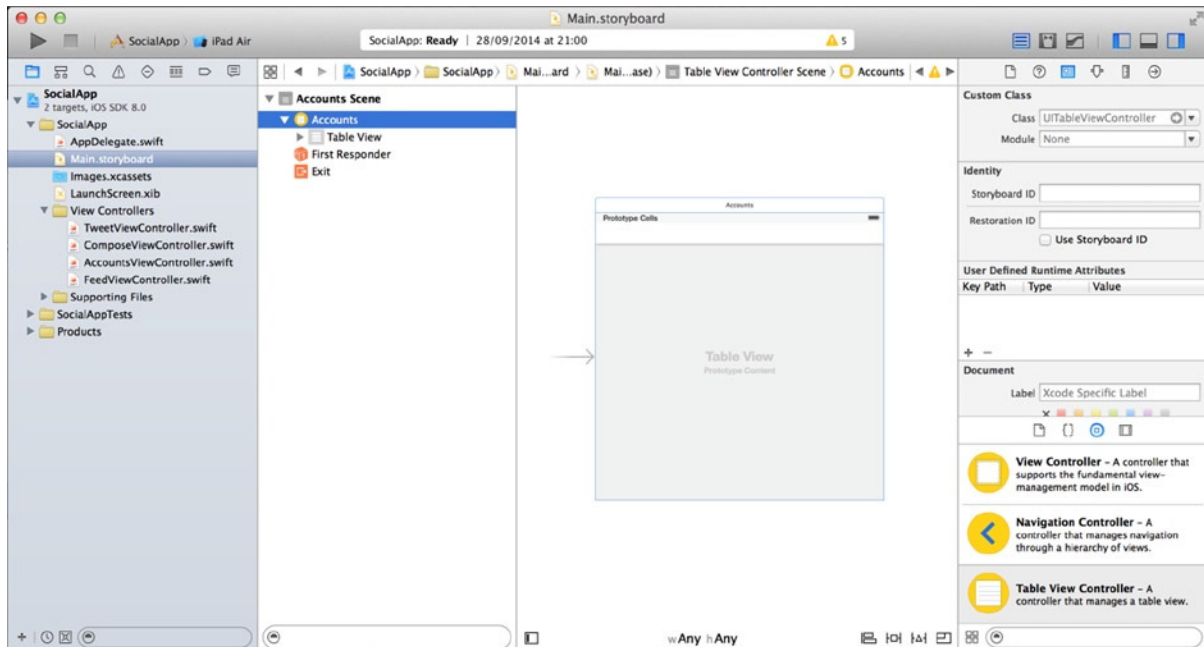


Figure 7-18. Xcode with the view controller selected and the Identity Inspector open

3. In the Identity Inspector, look for the Custom Class section. This is where you bind your view controller's visual element to the actual view controller. In the Class field, it currently says that this view controller's class is `UITableViewController`. It's grayed out because although it knows what its base type is, you haven't yet tied it to a custom view controller. Click the down arrow at the end of the field, and you should see three selections: the base class and the two custom view controllers, as shown in Figure 7-19. Select `AccountsViewController`.

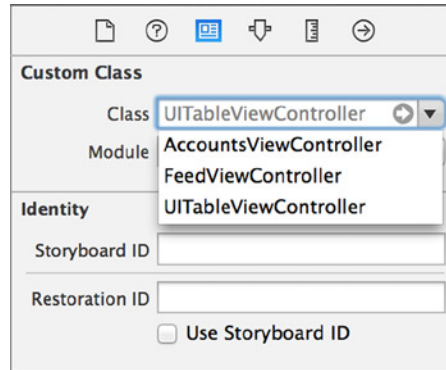


Figure 7-19. The list of available view controllers

Note If at any time you don't see one of your view controllers listed as an option in the Custom Class list and you have the view controller selected in Interface Builder, either quit Xcode and relaunch it or write the class name in yourself—but remember, it's case sensitive. Sometimes Xcode loses track of which classes have been created.

You've told the view controller on your storyboard that it no longer complies with its base class `UITableViewController`; now it's controlled by the `AccountsViewController` class. Xcode immediately reacts to the change in custom class. Try opening the Assistant Editor: it displays the implementation file for the `Accounts` view controller, confirming that the class is valid and the link to the storyboard is working.

Understanding Inheritance

Subclassing and *base classes* are two terms that feature heavily when describing the concept of inheritance. *Inheritance* is one of the key principles of object-oriented programming. It describes a link between two classes: the *base* or *superclass*, and the *custom class*, which can be referred to as a *subclass* or *derived class*.

The best way to think about a base class and subclass is as a parent and child; the child descends from the parent and shares the same genes, but the child is not identical to the parent. And although the parent has its own attributes, the child has the ability to gain its own new attributes. In Wales, people are often named along the lines of David ap Gwillim, meaning David son of Gwillim. That is exactly what `class AccountsViewController: UITableViewController` is saying in the `AccountsViewController.swift` file: that it descends directly from `UITableViewController`.

Now that you have a better understanding of inheritance and why you create custom classes, you can finish building the storyboard.

Building Up the Storyboard

With `Main.Storyboard` open, let's get back to the focus of this chapter: storyboards. You've created the first scene for account selection; it's time to create the second scene, which is the list of tweets, more commonly known as the Twitter feed, which is controlled by `FeedViewController`:

1. This is another `UITableViewController` class, so you need to drag a table view controller from the Object Library and drop it next to the first scene, as shown in Figure 7-20.

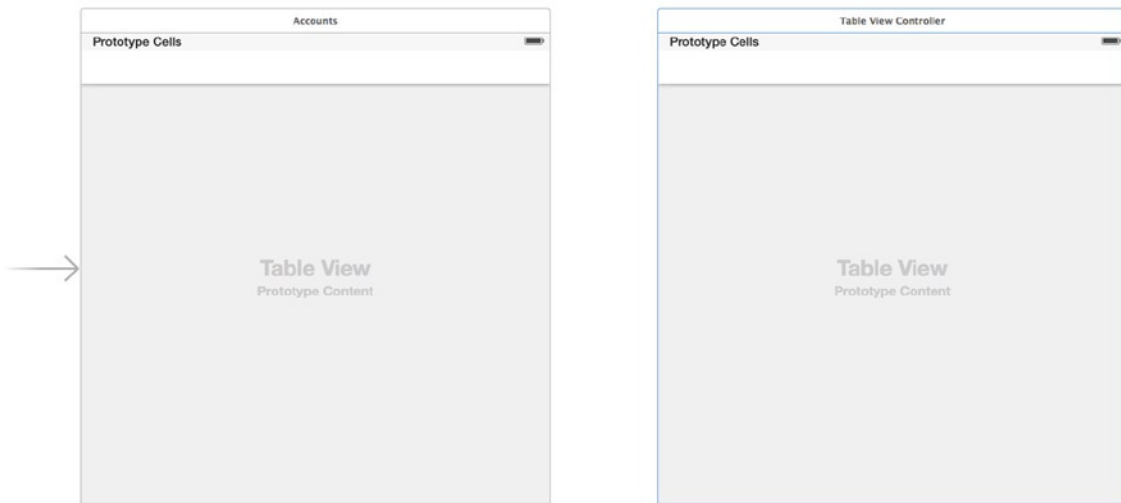


Figure 7-20. The storyboard with two table view controllers side by side

2. Select the new table view controller, and open the Attributes Inspector. Set the Title attribute to Feed; this will help you keep track of your scenes in what will be a full storyboard.
3. Select the Identity Inspector, click the Class drop-down list, and choose `FeedViewController`.
4. You have two scenes on your storyboard. Let's add the last two so you can move on to a careful examination of one of the key features of any storyboard: segues. The third scene is a regular view controller that is used to show the details of the tweet. Drag a view controller from the Object Library, and drop it above and to the right of the Feed view controller.
5. Select the new view controller, and open the Attributes Inspector. Set the Title attribute to Tweet. To set the class, open the Identity Inspector and set the Class value to `TweetViewController`. Your growing storyboard should resemble that shown in Figure 7-21.

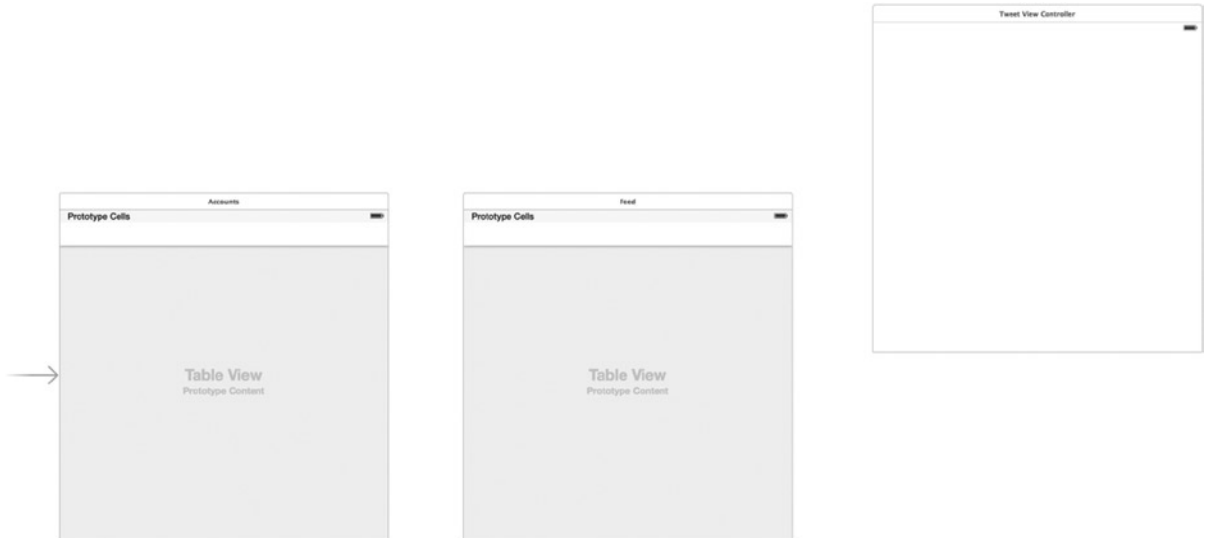


Figure 7-21. The three scenes, positioned nicely and ready for the fourth

6. By now you can start to appreciate how it can sometimes be challenging to work on large, multiview applications using storyboards. This is why it's best to meticulously name every scene as it's created, so you can quickly identify a scene via the Document Outline. The next scene is `ComposeViewController`, which you use to write tweets and post them to Twitter. Drag in another view controller, and position it directly below `TweetViewController`.
7. Click the new view controller, and open the Attributes Inspector. Set the Title attribute to `Compose`. As you've done previously, open the Identity Inspector, and set the Class value to `ComposeViewController`.

All the scenes of the storyboard are laid out neatly. Each view controller on the storyboard is tied to its respective view controller class. Although you have the basic structure in terms of the scenes, the scenes themselves are largely empty. Let's focus on the individual scenes and begin adding the elements that will make up the interface. Before you move on, check that your interface resembles that shown in Figure 7-22.

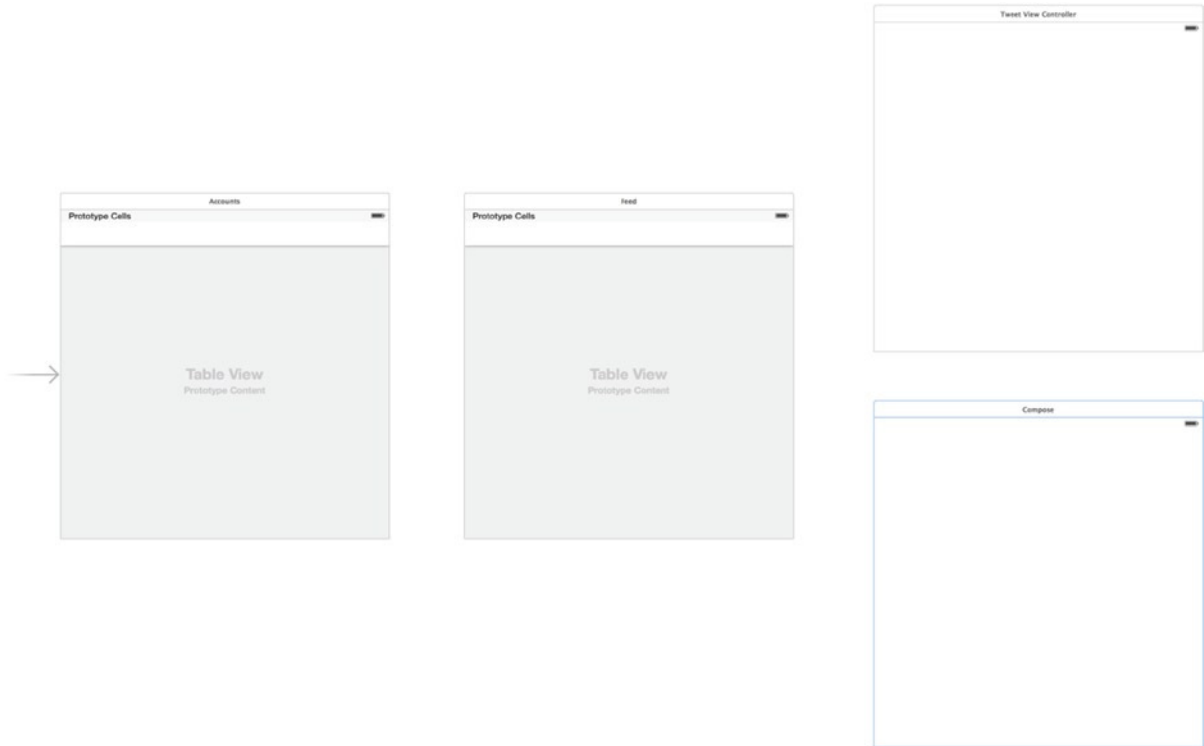


Figure 7-22. The storyboard as it stands, with all the scenes in place, but without their interface elements

Linking Scenes and Building Interfaces

One message that should be coming across in this chapter is that building an application with storyboards is part of a structured process. First you plan your application and its scenes, and then you create the view controllers before tying them into their respective visual counterparts on the storyboard, giving you separate view controllers that have their classes set but are ultimately strangers to one another. To address this issue, you need to progress through each scene from 1 through 4, building and connecting the interface with segues.

What Are Segues?

Just as the concept of storyboards is rooted in the movie industry, so is the term *segue* (pronounced “seg-way”). In a film, a segue is a transition between scenes, so you can immediately see how it’s appropriate as a term that describes the mechanism used by Xcode to transition between storyboard scenes.

In Xcode, segues need a start point and an end point. Typically, the start point is a button or a table cell. The end point is almost always another view controller. Think back to Chapter 3 for a moment; there you wrote several lines of code to push the second view controller onto the screen. With storyboards, a segue allows you to do this with a couple of clicks.

To demonstrate how to create a segue, let's start by linking the Accounts view controller to the Feed view controller. It's important to note that the Accounts view controller doesn't need any specific interface work at this point; I explain that in more detail in Chapter 8. Here are the steps:

1. Double-click the design area to zoom back to 100%. Move the storyboard around so you can get as much of the Accounts view controller and the Feed view controller in view as possible, as shown in Figure 7-23.

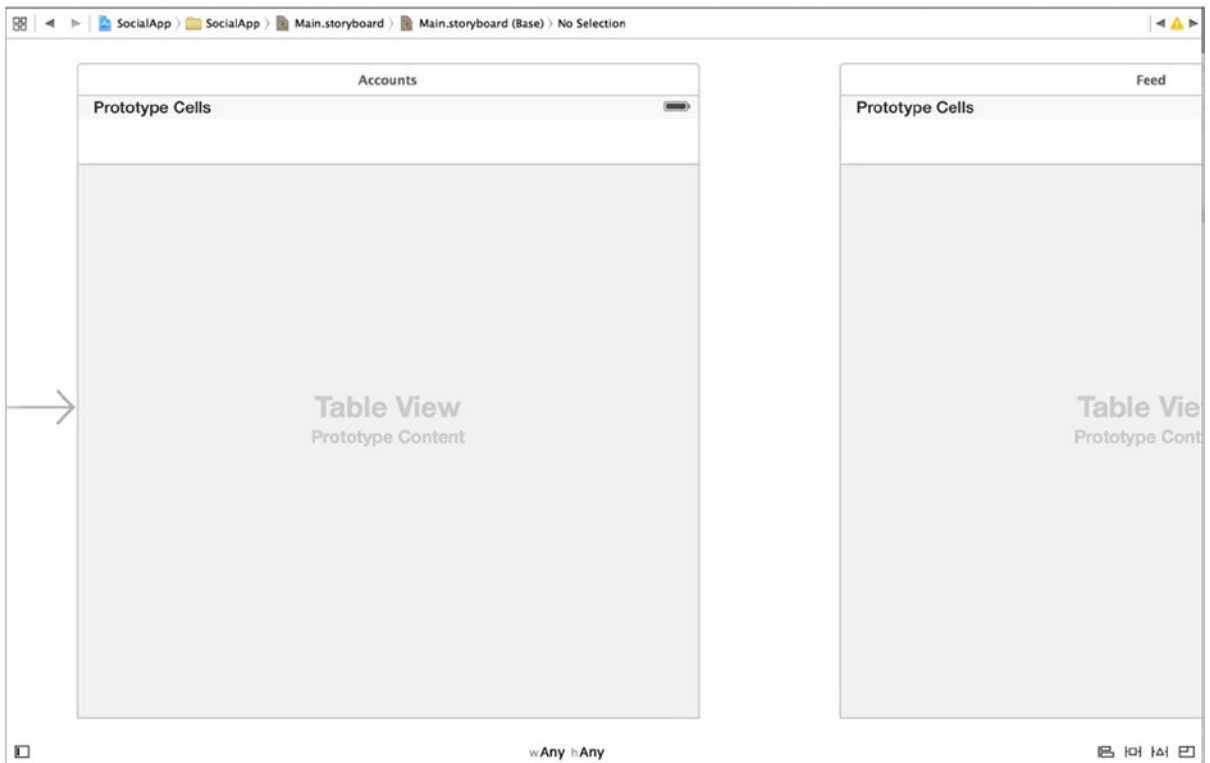


Figure 7-23. Scenes 1 and 2 side by side: the Accounts view controller and the Feed view controller

2. The Accounts view controller lists all the Twitter accounts that are set up on the device in a table view. Selecting one of the rows takes you to the Feed view controller. To make this happen, you need to create a Show segue from the table cell to the Feed view controller. Highlight the table cell in the Accounts view controller by clicking it, as shown in Figure 7-24.

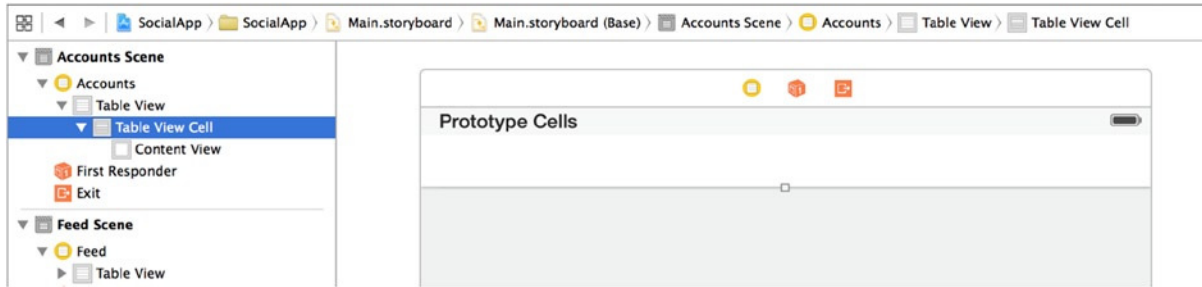


Figure 7-24. The Accounts view controller table cell highlighted

Note It's not immediately obvious that you've selected the table cell, but if you look at the jump bar above the design area, or in the Document Outline shown in Figure 7-24, you see that it is indeed selected.

3. Hold down the Control key, click the table cell, and drag a connecting line from the cell to the Feed view controller, as shown in Figure 7-25.

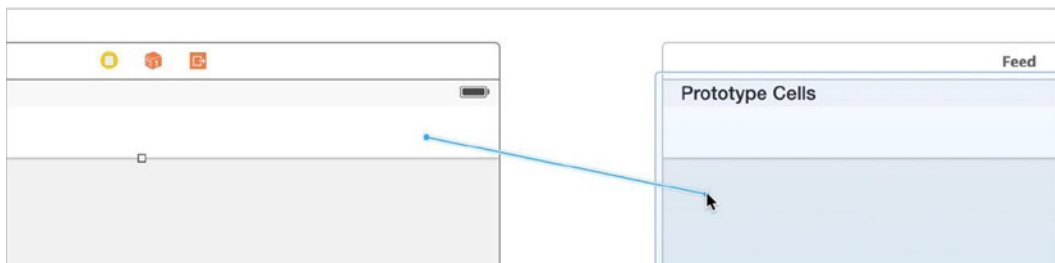


Figure 7-25. Dragging a connecting line between the table cell and the Feed view controller

4. When you release the mouse button, you're presented with a contextual dialog asking about the type of segue you want to create, as shown in Figure 7-26. In this instance, you want to create a Show segue from the Selection Segue list of types.

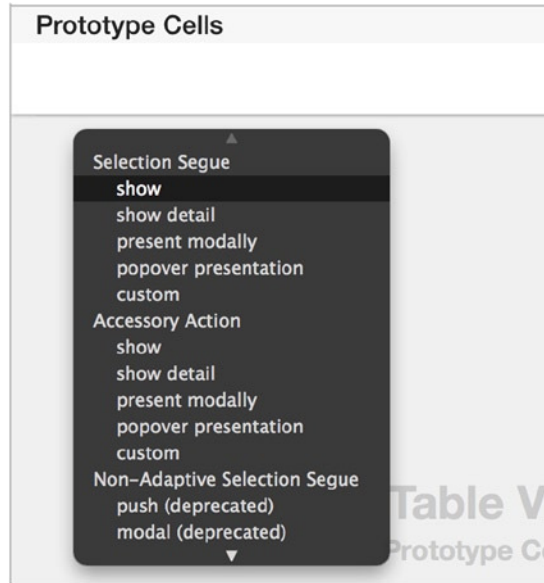


Figure 7-26. The contextual menu presented when you create a segue from a table cell

Note The dialog that appears when you create a segue is contextual because depending on your start point, the options for creating a segue are different. The start point for this segue is a table cell, so the options you're presented with are specific to this scenario. As you can see, there are three headings: Selection Segue, Accessory Action, and Non-Adaptive Selection Segue. This is because a table cell has two elements that support user interaction: the cell itself triggers the Selection segue, whereas the cell accessory triggers the Accessory Action segue, the Non-Adaptive heading encapsulates the segues that were deprecated with Xcode 6 and iOS 8.

5. A segue is created between your two view controllers. Select it, and then open the Identity Inspector, as shown in Figure 7-27. In the Identifier field, type **ShowTweets**.

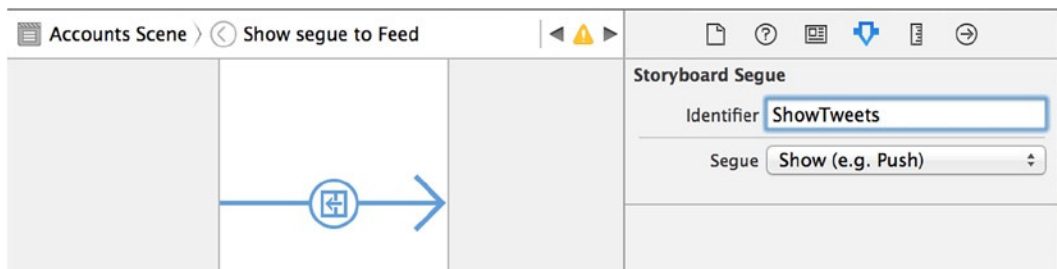


Figure 7-27. Inspecting the segue

This segue is triggered when the user taps the table cell for the Twitter account they want to use. When this happens, you need to pass the selected account details to the Feed view controller. In Chapter 3, you did this by writing code that passed an object to the next view controller; but when working with segues, you use the `prepareForSegue` method, which is called every time a segue on the view controller is triggered. I explain this method in more detail later, when you write the code that performs different actions depending on the segue that is being triggered. But to identify which segue is being triggered, you must give each an identifier.

Tip Even when you have only one segue coming from a view controller, as in this case, it's still a good practice to give it an identifier so that if you add more segues to the view controller in the future, you won't have a situation where you're passing information to the wrong target view controller.

Before you go any further, let's take a minute to look at the different segue styles available and in what situations you might use each one:

Show: Prior to Xcode 6 and iOS 8, this was referred to as a Push segue. This segue dismisses the current view and pushes the target of the segue onto the screen. Behind the scenes, the Show segue adds the target view controller onto the navigation stack, which is why you always need a navigation controller to be present when using a Show segue. Xcode does all the work of managing the navigation stack for you: when your Show segue is triggered, the view controller that is presented automatically has a button to go back to the previous view controller.

Present Modally: Modal segues are definitely the most interesting and varied type you can use, especially when working with iPad applications. A modal segue presents another view controller without the need of a navigation controller. You can slide these over the top of the view that calls the segue. A number of transition animations and presentation styles are available, some of which I explain later in this chapter. Figure 7-28 shows a modal segue in action, presenting the tweet view controller from SocialApp modally with the Form Sheet presentation style.

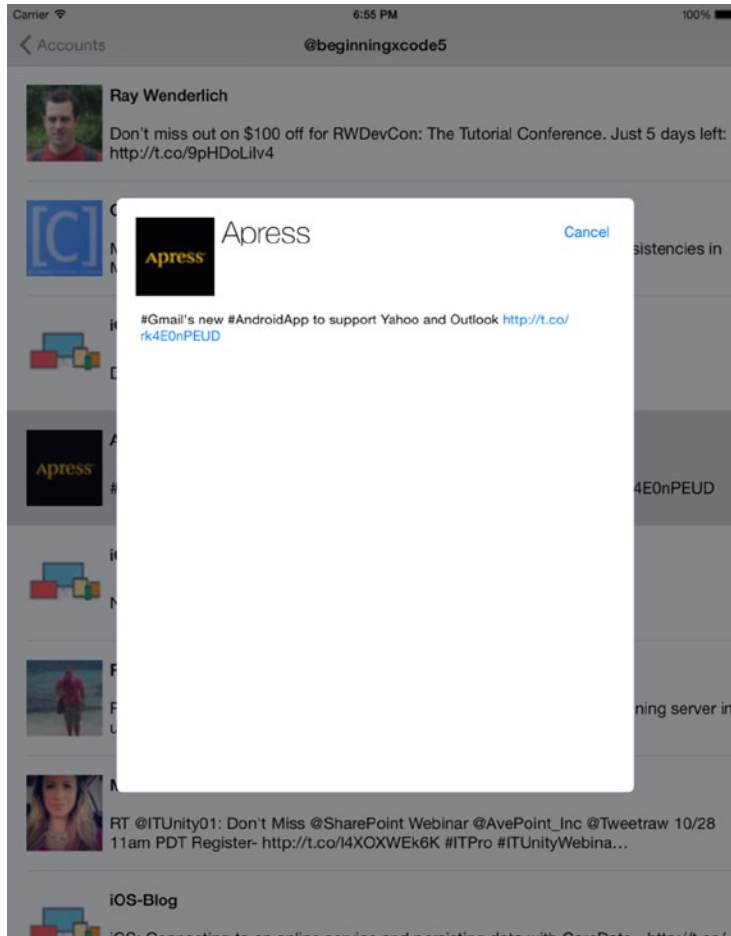


Figure 7-28. A modal segue in action

Popover Presentation: Popovers are visually similar to some modal segues in that they cause a view controller to appear above the view controller that originated the segue. These are useful for displaying contextual information; for example, if you created an application for an online store, you could use popover segues to provide a quick view of your products. You can see an example of a popover in Figure 7-29; note that the arrow at the top of the view controller is generated by Xcode and can be configured.

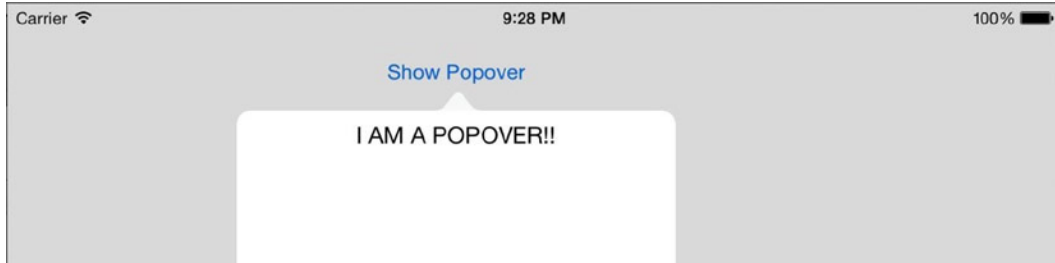


Figure 7-29. An example of a popover segue presenting another view controller on top of the view

Show Detail: Known as a Replace segue prior to Xcode 6, this style was previously available only when working with an iPad storyboard. It's mainly used with master-detail applications. (Refer to Chapter 3 if you're unfamiliar with the Master-Detail Application template.) A Show Detail segue replaces the originating view controller with the target one in the navigation stack.

Custom: As you might expect, a custom segue can be anything you tell it to be. With a custom segue, you specify a custom `UIStoryboardSegue` class in a way similar to how you would set custom classes for the view controllers.

Now that you understand the different segue styles, perhaps you've noticed from the description of the Show segue that your application is missing something essential: a navigation controller.

Adding a Navigation Controller

A navigation controller, or `UINavigationController`, is used to manage navigation through the various view controllers in your application. It keeps track of where the user has been, adding each successive view controller to the navigation stack, and provides a mechanism for the user to navigate backward through the navigation stack, all without you having to write any code. You first encountered navigation controllers in Chapter 3, where you added one programmatically to your application; however, this time I explain how Xcode makes this possible in just a couple of clicks.

Because the Accounts view controller is the originator of the Push segue, and because it's the initial scene on the storyboard, the navigation controller needs to be applied here:

1. Select Accounts from the Document Outline, as shown in Figure 7-30, because you're applying the navigation controller explicitly to the view controller.

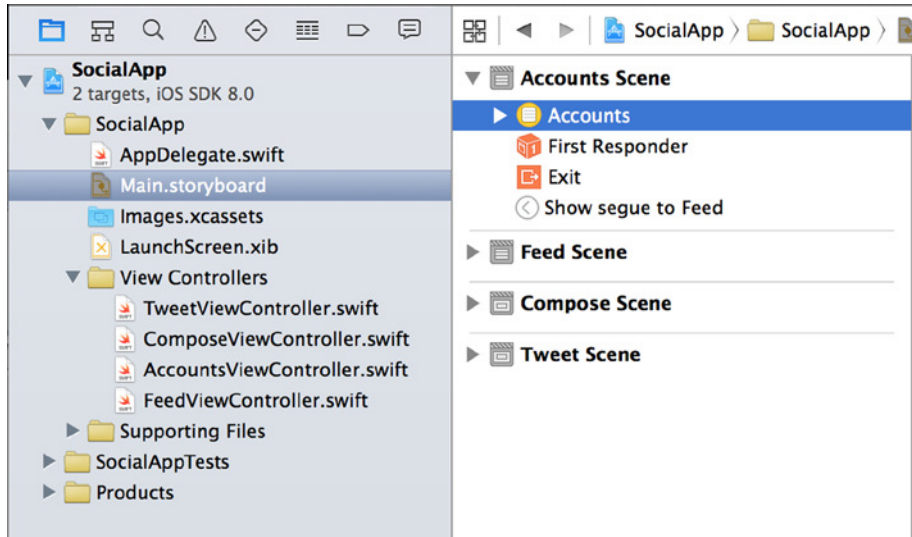


Figure 7-30. Selecting the Accounts view controller from the Document Outline

2. To add a navigation controller to this view controller, from the menu bar select Editor ► Embed In ► Navigation Controller. Xcode adds a navigation controller to the storyboard and attaches it to the Accounts view controller for you, as shown in Figure 7-31.

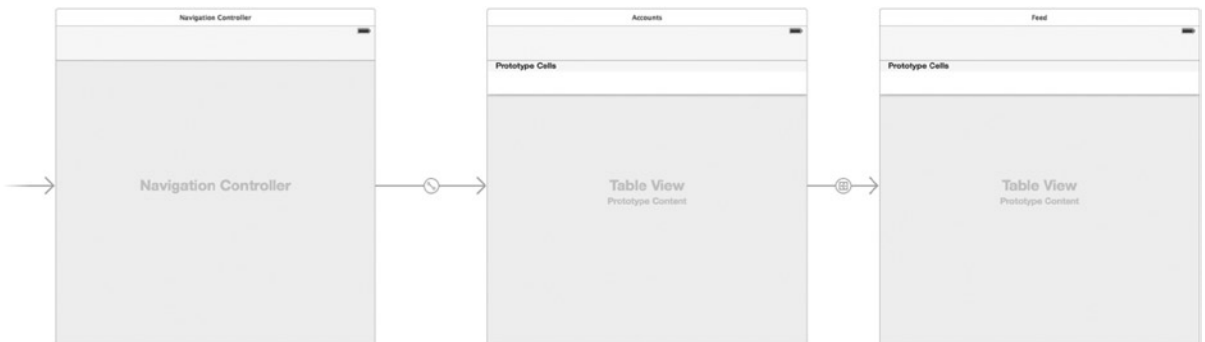


Figure 7-31. The navigation controller added to the storyboard and linked to the Accounts view controller

3. You can set a title for the view that is visible to the user and provides meaningful text for the Back button. Zoom back in to the storyboard. In Accounts, highlight the navigation bar at the top of the view or select Navigation Item from the Document Outline, as shown in Figure 7-32.

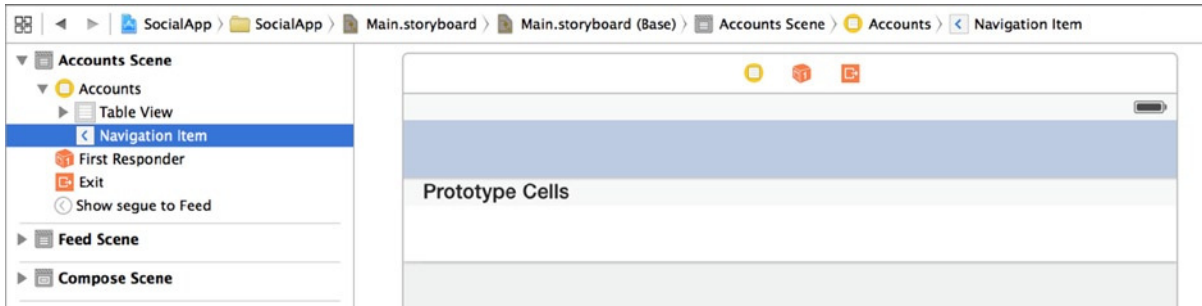


Figure 7-32. Selecting the Navigation Item from the Document Outline for the Accounts view controller

4. Open the Attributes Inspector, and set the Title attribute to Accounts. Notice how the title appears at the top of the view controller, as shown in Figure 7-33. What's also neat is that when you select an account and segue to the Feed view controller, the Back button is automatically labeled Accounts.

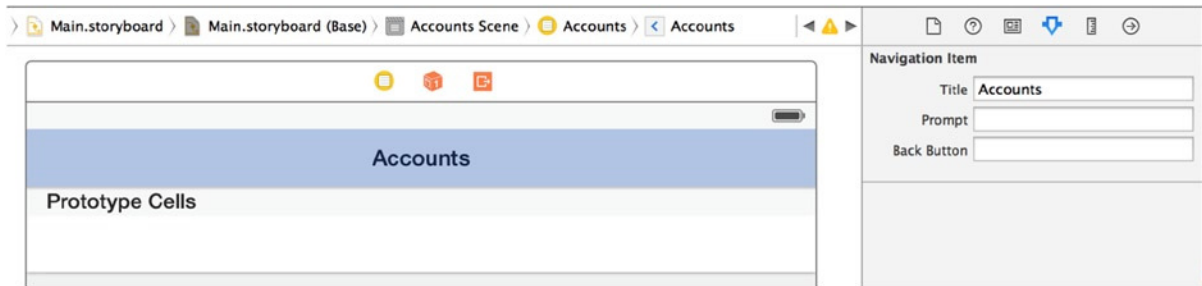


Figure 7-33. The Title attribute for the navigation bar in place

With the navigation controller in place and the Accounts view controller complete for now, you're ready to start building the interface on the remaining three scenes. Next up: the Feed view controller.

Creating an Interface for the Feed View Controller

The Feed view controller is responsible for showing all the tweets in the user's Twitter timeline. In Chapter 8, you create a custom table cell to display the actual tweet. But for now you need to add a button to the navigation bar so you can compose new tweets, and then create modal segues to both the Compose scene and the Tweet scene. Zoom to 100%, and ensure that the Feed view controller is front and center:

1. Position the Feed view controller in the design area.
2. Locate Navigation Item in the Object Library, as shown in Figure 7-34, and drag it onto the Feed view controller. Remember to use the filter and search for *navigation* to make your search easier.

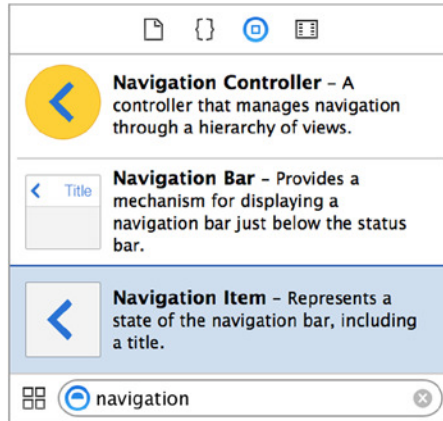


Figure 7-34. Searching for Navigation Item in the Object Library

3. You need to add a button to the navigation bar so that you can create a modal segue to the Compose view controller. To do this, search for *button* in the Object Library. You're looking specifically for Bar Button Item, which should be the second item in the Object Library. Drag it onto the right side of the navigation bar, positioning it as shown in Figure 7-35.

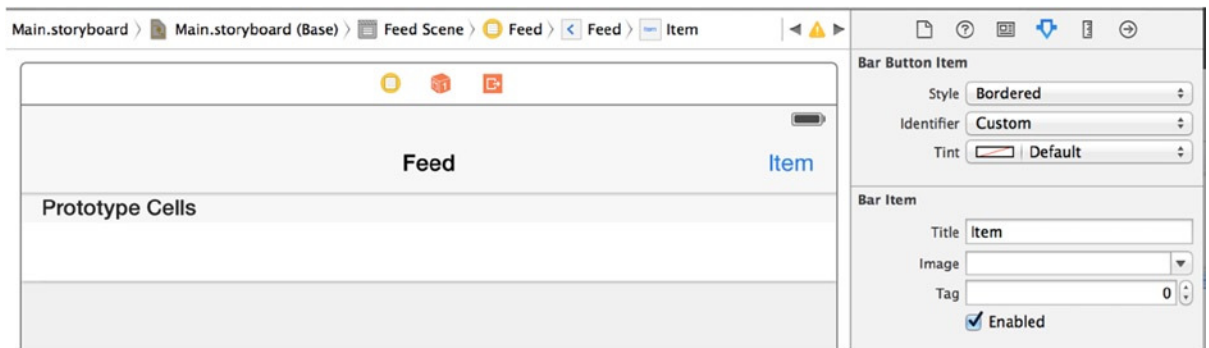


Figure 7-35. Adding a bar button item to the navigation bar

4. If you still have the Attributes Inspector open, when you drop the bar button item on the navigation bar, its attributes are displayed. If not, open the Attributes Inspector and select the new button.
5. You could change the Title attribute of this button to read Compose, because that is the scene it will link to. However, Apple provides a standard set of icons you can use for this button by choosing one of the predefined Identifiers. Click the Identifier attribute list and select Compose, as shown in Figure 7-36. Your button changes from text to an icon with a pencil in a box.

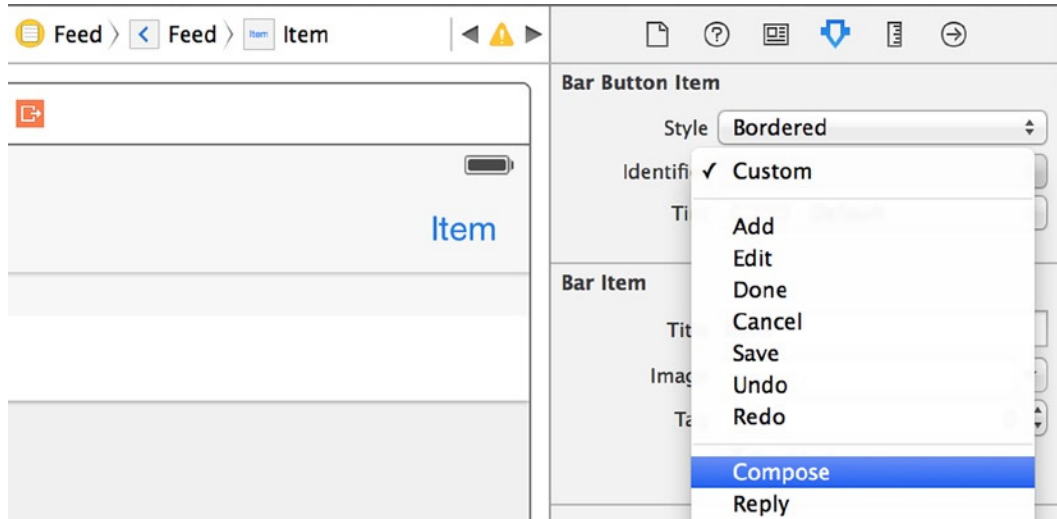


Figure 7-36. Selecting the Compose identifier for the bar button item

Note Apple provides a number of identifiers that can be used for many common tasks. You should use these whenever possible, to provide users with icons they're familiar with and also to future-proof your application. If Apple updates that identifier in the future, your applications will be easy to update to the new design.

6. While you're working with the navigation bar and the Attributes Inspector, select it as you did for the previous scene and change the Title attribute from Title to Feed.
7. You've created the interface for your second scene. Now you need to create modal segues to the Compose view controller and the Tweet view controller. This time, because the scenes are spread out in the storyboard, you can use the Document Outline to add an element of simple precision to the task of creating segues. Compress all the scenes except the Feed, Compose, and Tweet view controllers by clicking the triangles to the left of each scene's title. Then, expand Feed, and beneath that expand Table View, so your Document Outline resembles that shown in Figure 7-37.

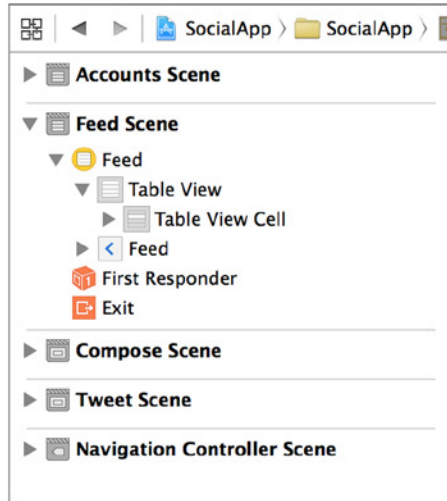


Figure 7-37. Preparing to create segues using the Document Outline

- The first segue you create is to the Tweet view controller. You get to this scene by selecting one of the table cells, so highlight Table View Cell and Control-drag a connection from there to the Tweet view controller, as shown in Figure 7-38.

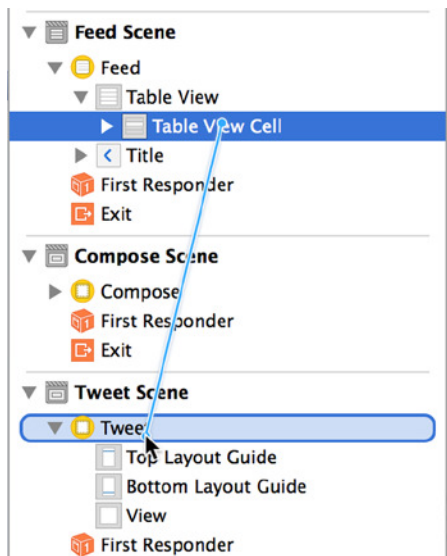


Figure 7-38. Creating a segue using the Document Outline

9. When you release the mouse button, the contextual dialog appears, just as it did in Figure 7-26. This time select Present Modally under the Selection Segue heading. You now have a modal segue connecting the table cell in your scene to the Tweet view controller, but you need to customize it slightly.
10. In the Feed Scene section of the Document Outline, notice the item Present Modally Segue to Tweet. This is the segue: select it to highlight it, and then open the Attributes Inspector.
11. Set the Identifier attribute to ShowTweet, the Presentation attribute to Form Sheet, and the Transition attribute to Cover Vertical.
12. You need to create a modal segue from the Compose bar button item to the Compose view controller. Expand Feed in the Document Outline to reveal Compose, as shown in Figure 7-39.

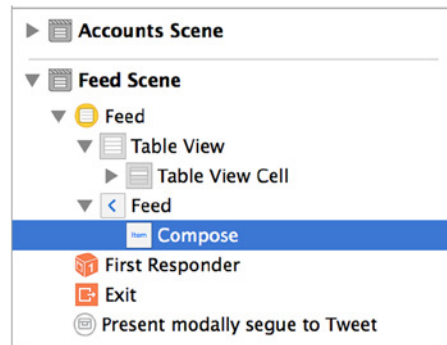


Figure 7-39. Exposing the bar button item in the Document Outline

13. In the Document Outline, Control-drag a connection from the Compose button to the Compose view controller, as shown in Figure 7-40. When you release the button, select Present Modally. Notice this time that the menu is different because you're dragging from a button, not a table cell.

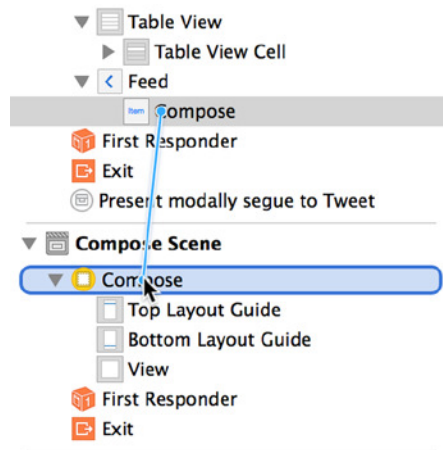


Figure 7-40. Making a connection between the button and the Compose view controller

- Two segues are listed in the Document Outline for your scene. Select the one named Present Modally Segue to Compose. Open the Attributes Inspector, and set the Identifier attribute to ComposeTweet, the Presentation attribute to Form Sheet, and Transition to Cover Vertical.

Let's review where you are with the project. Zoom out so you can see more of the storyboard, as shown in Figure 7-41. Notice how much smaller the Compose and Tweet view controllers are because you created segues for them. The reason is that they're adjusting to the presentation style you specified on each segue: Form Sheet. If you refer back to Figure 7-28, you see the finished Tweet view controller in action, displayed modally over the Feed view controller.



Figure 7-41. The storyboard changes in appearance after you set up two scenes

The way the view controller reacts dynamically to the segue means you can be fairly confident that the interface you build in Interface Builder will be what you get on a physical device or in the simulator. You're ready to move on to the third scene: the Tweet view controller.

Creating an Interface for Tweet View Controller

The purpose of the third scene is to display details about the tweet the user selects from the Twitter feed. You could display all types of information, but for this application you simply display the tweet author's name, the tweet content, and the tweet author's avatar image. Because you've done these steps a few times already in this chapter, I use screenshots at key points in the process so you can verify that you haven't missed anything:

1. Because this is a modal scene, it has no Back button. You need to be able to dismiss the view controller when the user wants to return to the Twitter feed. Add a button at upper right in the view by dragging one from the Object Library. In the Attributes Inspector, set the Title attribute to Cancel.
2. You may need to resize the button slightly to make the text visible. When you're happy with the button's appearance, position it as shown in Figure 7-42. You'll set the auto layout constraints once the view is built.

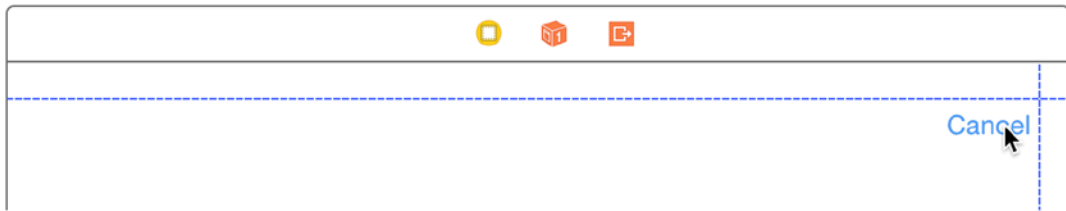


Figure 7-42. *Placing the button on the Tweet view controller*

3. You need to add an image view from the Object Library to the view controller. Position it somewhere in the view, and then open the Size Inspector. This is a handy tool for gaining pinpoint precision over an object's size and position. Set the X axis value to 16 and the Y axis value to 20; these control where the top-left corner of the image view is positioned relative to the parent view. Set the Width and Height values to 82, as shown in Figure 7-43, to create a square image view.

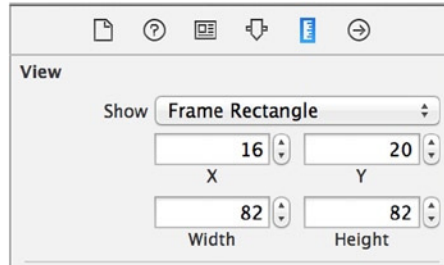


Figure 7-43. Setting the width, height, and x and y axis positions of the image view

4. From the Object Library, drag in a label to contain the tweet author's name. Position it loosely between the image view and the Cancel button. You'll adjust the size and the font, and it makes sense not to finalize the position until after you've done that.
5. Open the Attributes Inspector with the label selected. Select T by the Font attribute. Set Font to Custom, Family to Helvetica Neue, Style to Thin, and Size to 32, as shown in Figure 7-44.

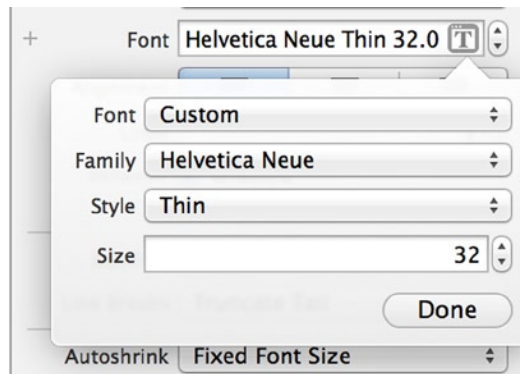


Figure 7-44. Adjusting the Font attribute for the label that shows the tweet author's name

6. Notice that you can't see any text in the label, due to its size and shape. Position the left side and top of the label so it snaps into alignment with the top of the image view. Then resize it so that it uses the rest of the available width in the view controller and the height is sufficient to show the text clearly, as shown in Figure 7-45.

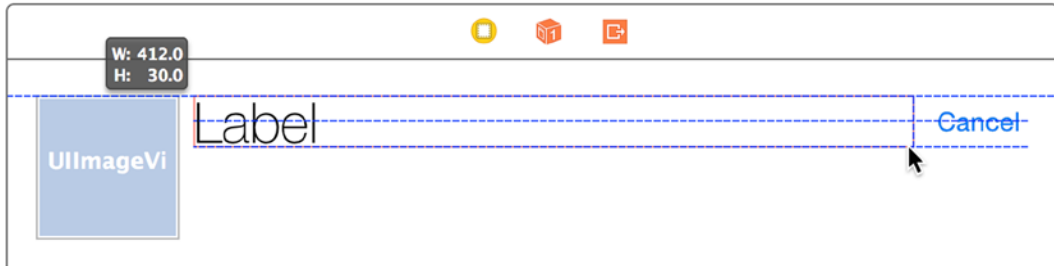


Figure 7-45. Setting the label to fill the remaining width

7. You need something to show the textual content of the selected tweet. Tweets vary in size, so the best choice is a text view. Drag one from the Object Library, snapping it into place below the image view. Resize it to give it a generous size, as shown in Figure 7-46.



Figure 7-46. The text view added to the view controller

8. You don't want this text view to behave like a regular text view, because it shouldn't be editable. Open the Attributes Editor, and uncheck the Editable attribute. Leave Selectable checked so the user can copy the tweet text.
9. While the attributes of the text view are open, look at the Detection section. Twitter posts often contain links, so check the Links attribute, which tells Xcode to detect links and provides a code-free way of opening the link in Safari.
10. Click the Resolve Auto Layout Issues button. Under the heading All Views in Tweet View Controller, click Add Missing Constraints. This ensures that your layout adapts correctly.

11. Even though you're using storyboards, you still need to create actions and outlets for the interface objects. Open the Assistant Editor, and be sure it displays `TweetViewController.swift` in the right pane. If it doesn't, you need to go back and set the class for this view controller to `TweetViewController`.
12. As you've done numerous times in this book, Control-drag the following outlets: `tweetAuthorAvatar` for the image view, `tweetAuthorName` for the author's name label, and `tweetText` for the text view. Create an action for the Cancel button called `dismissView`. Your header should contain the following highlighted code:

```
import UIKit

class TweetViewController: UIViewController {

    @IBOutlet weak var tweetAuthorAvatar: UIImageView!
    @IBOutlet weak var tweetAuthorName: UILabel!
    @IBOutlet weak var tweetText: UITextView!
    @IBAction func dismissView(sender: AnyObject) {

    }
}
```

You're finished setting up the third scene for viewing tweets in detail. Next you need to set up the fourth and final scene: the Compose scene.

Creating an Interface for the Compose View Controller

The Compose view controller provides an interface for the user to create a tweet and post it to Twitter. You use a text view for the composition of the tweet, and you give the user two buttons—one to dismiss the view and one to post the tweet—and an activity indicator that triggers when posting the tweet. Make sense? Great! Let's get started:

1. To make the text view stand out a bit, let's change the view's background color. Click a blank area of the view, and open the Attributes Inspector. Choose a light gray background color; I changed the Background attribute to one of the predefined colors (Group Table View Background Color).
2. Before you add the interface objects just mentioned, let's add a label to act as a title for the view. Drag in a label from the Object Library, and position it in the upper-left corner of the view.
3. In the Attributes Inspector, change the label's Text attribute to **Compose a Tweet**. You also want to make it stand out, so click the T in the Font attribute and set Font to Custom, Family to Helvetica Neue, Style to Thin, and Size to 32. Resize the Label so that all the text is visible, but don't resize it to fill the width of the view. Reposition the label in the upper-left corner so the guidelines for the margin appear.

4. Drag a button to the upper-right corner of the view; this button will dismiss the view controller. Open the Attributes Inspector, and change the button's Title attribute to Cancel. Before you move on, the top of your scene should resemble that shown in Figure 7-47.

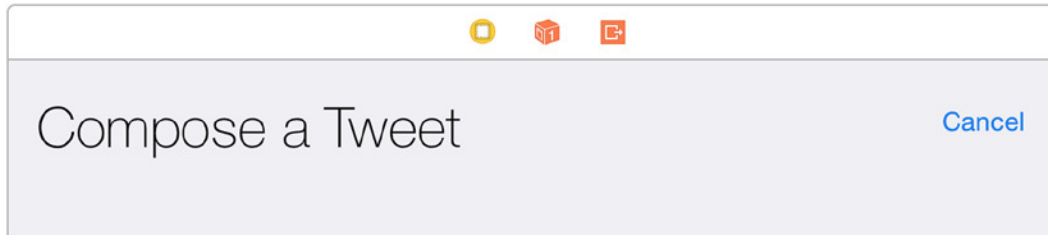


Figure 7-47. The label and button in position on the Compose scene

5. Drag in a text view, and position it below the label and button. This is where the user composes posts to Twitter, so make it a decent size. Be sure to expand the text view left and right until the margin appears.
6. One downside is that by default, the text view is populated with Lorem Ipsum placeholder text. Open the Attributes Inspector, and remove all the default text from the Text attribute. Your scene should be progressing nicely and resemble that shown in Figure 7-48.

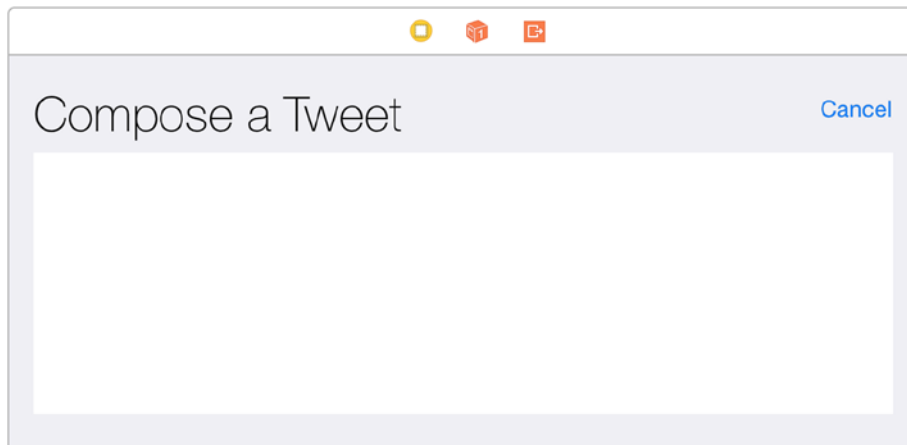


Figure 7-48. The text view in position below the button and label

7. You need to add a second button to allow the user to post the content to Twitter. Drag in a button, and position it below to the text view. In the Attributes Inspector, change the Title attribute to **Post**. Set the Background attribute to White Color. Then make the button a little larger and move it toward the center of the view until the blue guideline appears and it snaps into place, centered horizontally and positioned just below the text view.

8. This is the first time you've dealt with the next object: an activity indicator. Activity indicators are very common in applications that rely on data from the Internet. The control produces the familiar spinning wheel that has become synonymous with data transfer over the past decade; it has long been used with AJAX-based applications on the Web and in iOS applications since iOS 2.0. Drag one in from the Object Library, and position it to the left of the Post button.
9. By default, the activity indicator is visible and static; you want it to be hidden until it's told to start animating. Xcode provides a simple attribute to achieve this: open the Attributes Inspector with the activity indicator selected, and check the Hides When Stopped attribute. Figure 7-49 shows the finished scene.

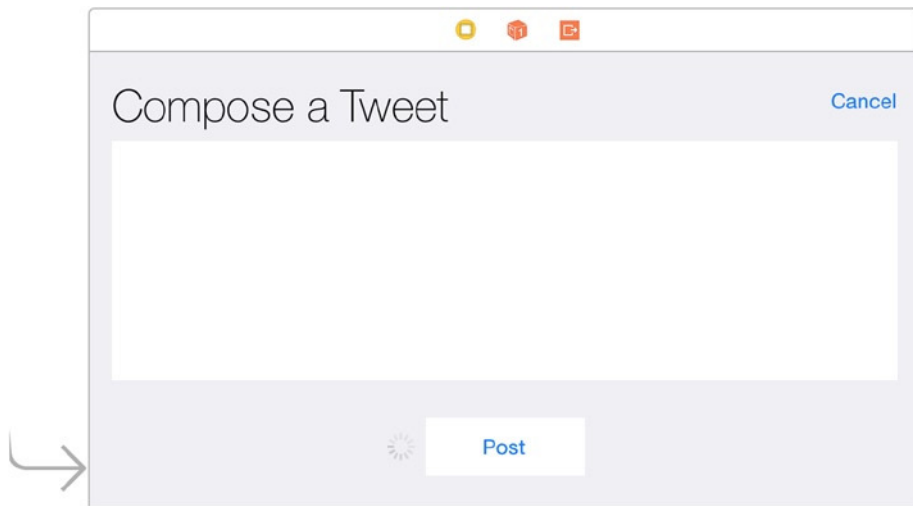


Figure 7-49. The finished Compose scene

10. With the interface elements laid out, let's fix them in place. Click Resolve Auto Layout Issues. Then, under the All Views in Compose View Controller heading, click Add Missing Constraints.
11. The text view needs to remain at a constant height, so click the text view and then click Pin. Click the Height check box, and be sure the value is around the 170 mark.
12. If you click the Post button, you see a constraint travelling from it to the bottom of the view. Click the constraint, and then, in the Attributes Editor, change Priority to 250, as shown in Figure 7-50. This change means iOS won't try to stretch your button down to the bottom of the view.

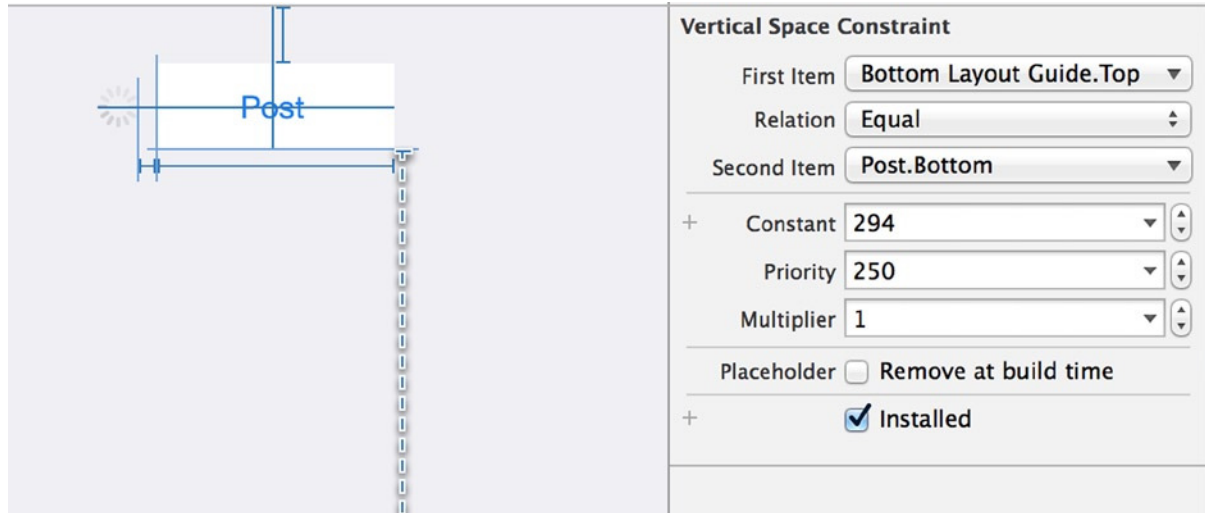


Figure 7-50. Adjusting the vertical space constraint beneath the Post button

13. There are no segues from this scene, so all that remains is to create the outlets and actions. Open the Assistant Editor, and ensure that `ComposeViewController.swift` is showing on the right.
14. Create an outlet for the text view called `tweetContent`, one for the Post button called `postButton`, and one for the activity indicator called `postActivity`. Create an action for the Cancel button called `dismissView` and one for the Post button called `postToTwitter`.

You should have the following highlighted outlets and actions in your header file:

```
class ComposeViewController: UIViewController {

    @IBOutlet weak var tweetContent: UITextView!
    @IBOutlet weak var postButton: UIButton!
    @IBOutlet weak var postActivity: UIActivityIndicatorView!

    @IBAction func dismissView(sender: AnyObject) {
    }

    @IBAction func postToTwitter(sender: AnyObject) {
    }
}
```

That's it for the fourth and final scene—you have all the elements in place for your users to compose messages and post them to Twitter. You've worked really hard getting to this stage in the project, and you've earned a rest before you learn about table and collection views in the next chapter.

Summary

It can be hard going through an entire chapter without having a completed application to show for it at the end; but in any project, you have to do preparation work, which is what you've done here. In the next chapter, you see the application come to life in the first few pages as you configure table views and take the first steps in the use of the Social and Accounts frameworks.

In this chapter, you've been shown all the skills essential to using Xcode to build an application structure with storyboards. At its core, such an app consists of scenes and segues. But specifically you've learned about the following:

- Organizing files in the Project Navigator using groups
- The model-view-controller design pattern and how Xcode is built around it
- Applying custom view controller classes to view controllers in the storyboard
- The inheritance principle of object-oriented programming
- Different ways of creating segues
- Types of segues
- Specifying identifiers for segues
- Embedding navigation controllers
- Using the Size Inspector to precisely position elements in the view

When you've had a well-deserved rest, move on to the next chapter to finish the SocialApp.

Table and Collection Views

In Chapter 7, you started work on SocialApp, a Twitter client; I presented an in-depth look at building an application structure with storyboards, explaining how to tie scenes together with segues. You also learned about the principles of the model-view-controller design pattern, combining the View element with the Controller element. In addition, you learned about subclassing and how inheritance is an important concept of object-oriented programming. Most important, you worked hard preparing the scenes for SocialApp and tying them all together with segues.

In this age of big data, it seems in every facet of our lives we're being bombarded by more and more data, and as developers, we often find ourselves needing a way to display large amounts of data to users in a concise and structured manner. In iOS, Apple has provided the table view and collection view for this purpose.

This chapter focuses on the table view and collection view. You explore how each view is structured and how you can use Xcode to alter their appearance. Additionally, you learn about creating custom cells, where you subclass `UITableViewCell` to customize the elements in your cells. As in the other chapters in this book, many additional lessons are learned along the way; you see how the segue identifiers specified in Chapter 7 allow you to share data between view controllers, and you learn about a variety of ways to obtain data from the Internet and display that in an application.

Because this chapter is reliant on your having access to at least one Twitter account, if you don't have one, it would be a good idea to register at www.twitter.com. Ensure that you've created the account and that you've "followed" some other Twitter users; whatever your personal feelings are about Twitter, remember that you don't have to use it beyond this chapter, and you can delete your account when you're ready.

After you finish SocialApp, I explain collection views and how they differ from table views. They're very close cousins—collection views have methods and properties similar to those of table views—so much of what you learn about table views can be directly applied to collection views.

What Is a Table View?

A table view represents an instance of the `UITableView` class; it presents the user with a single column of cells listed vertically. Table views provide developers with a great way of displaying a large number of options or data to the user, such as in a Twitter application where you can scroll through potentially hundreds or even thousands of tweets. They can also be used to neatly lay out application settings, exploiting the table view's hierarchical structure to take the user from high-level categories right down to granular details and micro settings. The flexibility of table views means you can find one in nearly every application, but you may not recognize them right away. Figure 8-1 shows some of the table views in use through iOS and the default applications.

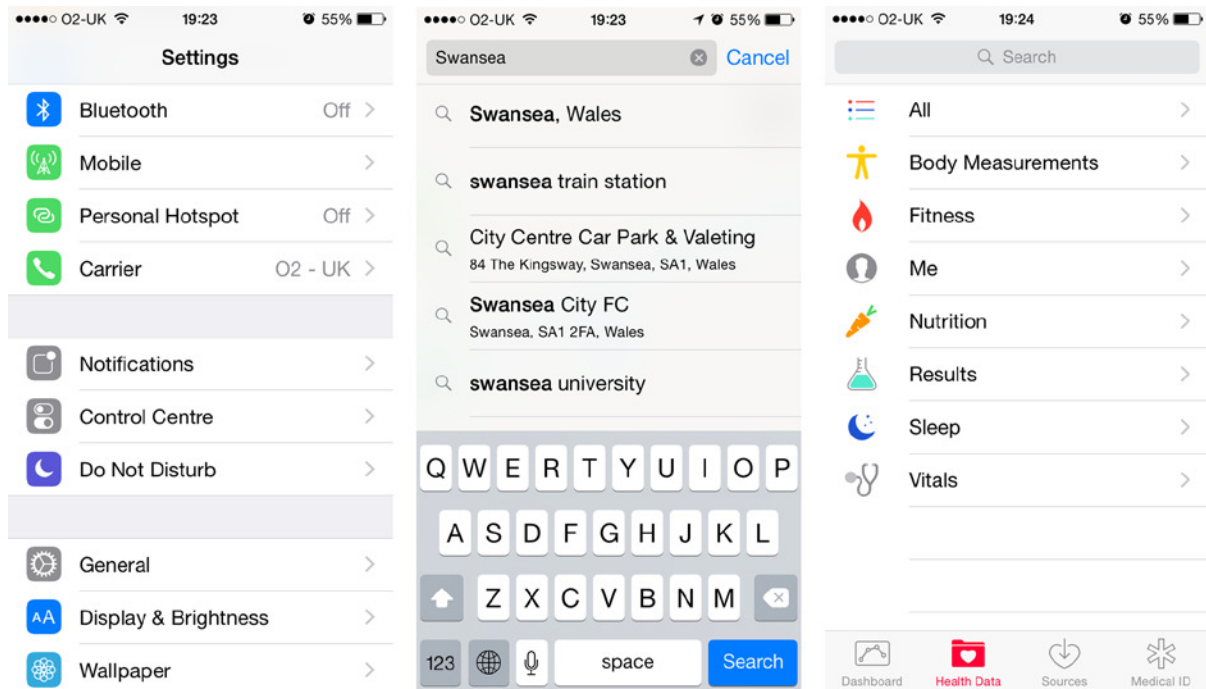


Figure 8-1. Table views in various iOS apps

Because of their popularity, Apple spends a lot of time improving the flexibility and feature set of table views with each successive release of iOS. This makes it easy to add features such as Pull To Refresh.

Table View Composition

Before you get into the configuration of table views, it's good to have a basic idea of their composition and key components, which are layered on top of each other. Figure 8-2 shows a visual breakdown of the different elements in a table view. Let's examine these components in more detail:

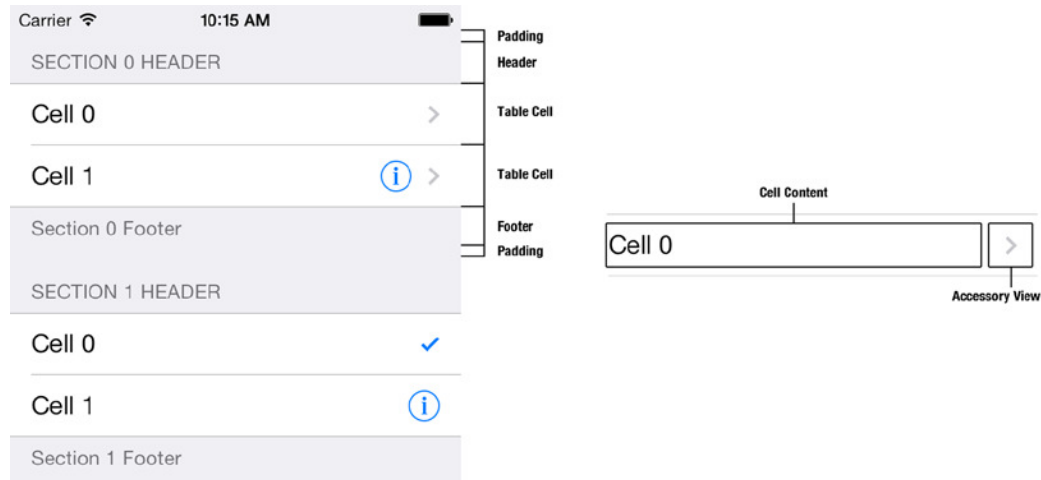


Figure 8-2. A breakdown of a table view's anatomy

- **View:** The foundation in the hierarchy of element. It sits at the bottom of the stack, coordinating all the child components. The View element controls the overall look and feel of the table and anchors all the delegate methods together.
- **Section:** The next item in the table-view stack. Sections are useful for breaking up tables, grouping cells together, and providing a header and footer for the group.
- **Cell:** Represents a row in the table view and can have varying states, such as when it's being edited, that affects the number of key areas in a cell. The two default areas of a table cell are the cell content and the accessory view. You may think the cell content area is self-explanatory, but it's actually a varied element and, depending on the style of cell, can contain an image, a title, and a subtitle. The accessory view can contain a disclosure indicator, such as an arrow, a detail accessory for providing more information about the row, or both.

Table View Styles

Table views in iOS 8 look exactly as they did in iOS 7. It's worth noting that when Apple released iOS 7, most areas received a visual makeover, but the greatest change was seen in table views. There are two separate styles for table views, but with iOS 7, they're no longer all that visually distinct. Figure 8-3 shows a plain style table view on the left and a grouped style table view on the right.

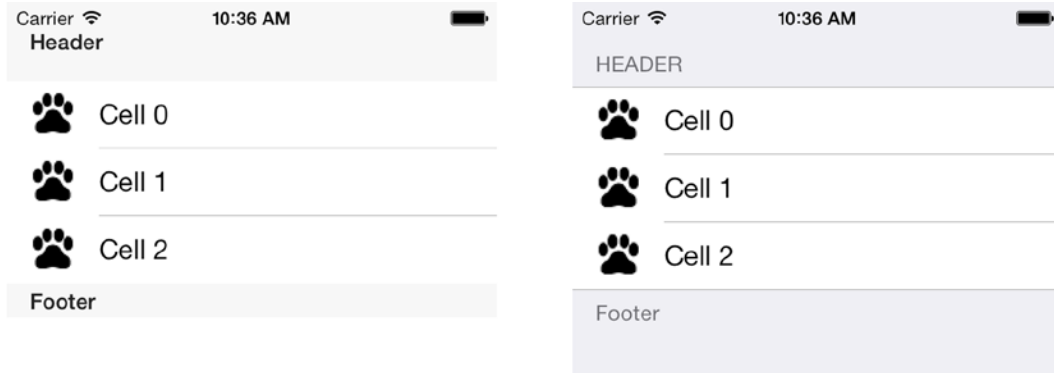


Figure 8-3. A plain table view (left) and a grouped table view (right)

In practical terms, you would typically use the plain style to list large amounts of data, usually in a single section with no header or footer. The grouped style is ideal for situations where you have a static list of data and you want to group similar items together, such as the different configuration options in the Settings application in iOS (first image in Figure 8-1).

Configuring the Accounts View

In the Twitter client SocialApp, the Accounts view is one of the simplest to set up, because it uses a plain table view with very little customization. However, you also get your first introduction to the Social and Accounts frameworks, which allow you to access details about accounts set up on the device and then use the account to authorize requests to social media sites such as Twitter or Facebook.

This is the first time you've had to go back to work on an existing project. If you haven't already opened the SocialApp project that you started in Chapter 7, start by opening Xcode, and then use File ► Open (⌘+O) and locate SocialApp.xcodeproj in the folder where you created it. Click Open, as shown in Figure 8-4.

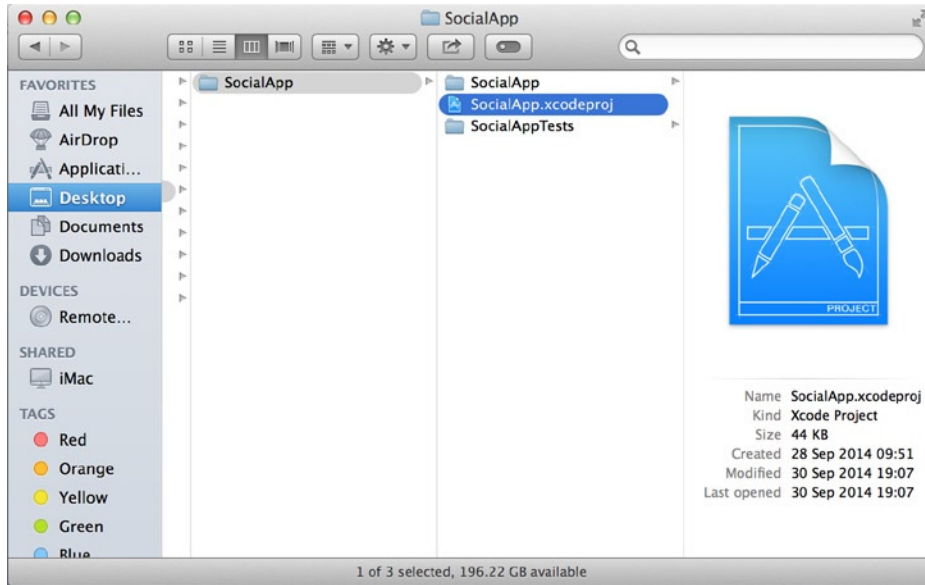


Figure 8-4. Opening the *SocialApp* project that you started in Chapter 7

Note If you've downloaded the book's source code, then you can open the project from the Chapter 7 folder. Don't open the Chapter 8 *SocialApp* project, because it's complete.

Before you configure the table view for the Accounts scene, let's explore the options available in Xcode for altering the table view itself; later in this chapter, you customize the cells. With the project open, select `Main.Storyboard` from the Project Navigator, and position the storyboard so that you're looking at the Accounts scene, as shown in Figure 8-5.

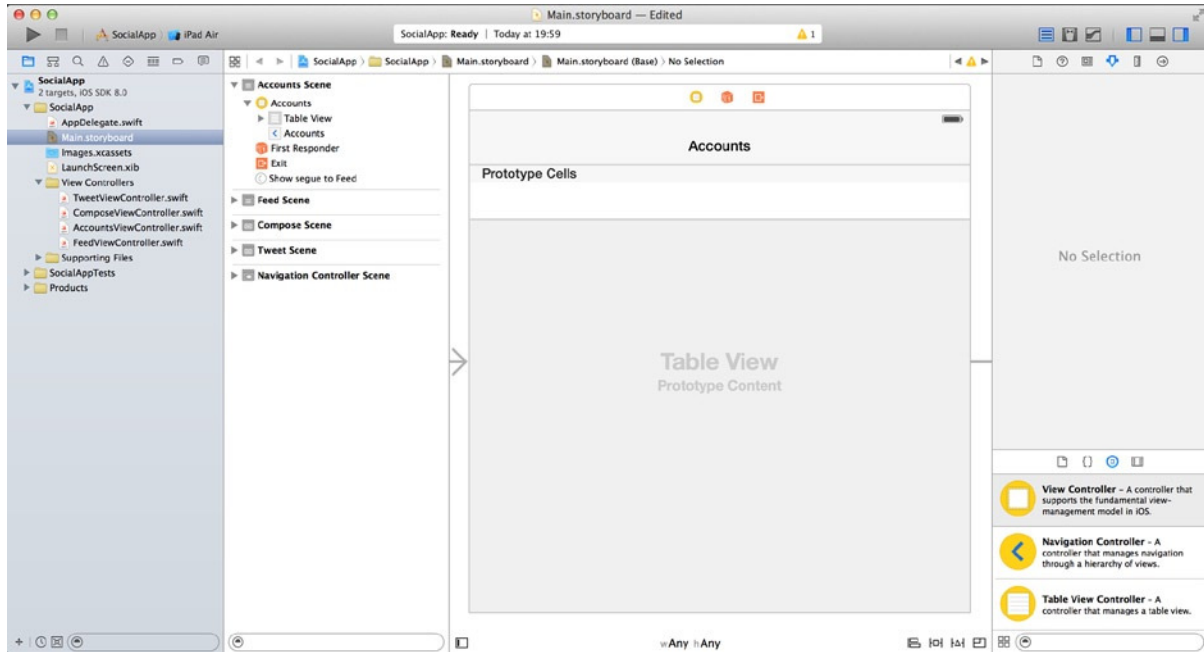


Figure 8-5. The Accounts scene positioned and ready to go

The Key Attributes of Table Views

Select the table view in the Accounts scene; you can do this by either clicking the words Table View Prototype Content in the middle of the view or selecting Table View from the Accounts scene in the Document Outline. Now open the Attributes Inspector. If you're looking at the right object, the first segment in the Attributes Inspector is Table View. Figure 8-6 shows the default attributes of a table view in Xcode 6.

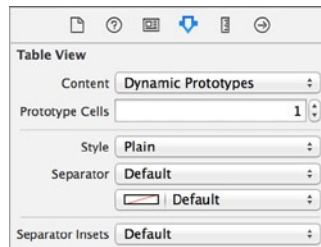


Figure 8-6. The default attributes of a table view in Xcode 6

When configuring a table view, most of the time you're after one of these first five attributes, because they have the largest influence on the table view's structure and style. Let's look at these options in more detail.

- *Content (Dynamic Prototypes)*: Unlike many of the attributes you can configure in Xcode, the content type can't be set programmatically. It's purely an Xcode thing. Selecting Dynamic Prototypes as the content type gives you a single table cell by default. The idea is that often, you have only one cell style in a table, so you configure the one cell, and in code you reuse it for each row. Using the Prototype Cells attribute that is available for this content type, you can increase the number of prototypes and have one for each distinct cell type. If you're using dynamic prototypes, you must customize the code to be able to view any cells in your view. This is the content type you'll use for all the table views in this application.
- *Content (Static Cells)*: When using static cells, you use Xcode to specify how many sections exist in your table view and how many cells appear in each section. You can then create segues from these individual cells to other scenes in your storyboard. One of the biggest differences between static cells and dynamic prototypes is that although the delegate methods that table views use to specify the number of sections and cells *can* be used to influence the table view, they can also be completely removed, leaving the attributes specified in Xcode as the controlling factor.
- *Style*: This is where you choose between plain or grouped styles. As you can see in Figure 8-3, there is very little difference between the two in iOS 8. Both styles can be extensively customized using code to change sizes and colors.
- *Separator*: The separator in a table view is the line that appears between cells. This attribute gives you four options to choose from: Default, Single Line, Single Line Etched, and None. In reality, these four options are only two. The Default style is the same as the Single Line style, and Single Line Etched is the same as None, because as it was deemed not compatible with the flat design approach Apple took with iOS 7; effectively, this style and its code equivalent `UITableViewCellSeparatorStyleSingleLineEtched` have been deprecated.
- *Separator Insets*: This was introduced in iOS 7 and Xcode 5. In earlier versions of iOS, the separator spanned the full width of the cell; however, in iOS 8, there is a small indent on the left side of the cell by default. By setting the Separator Insets attribute to Custom, you can specify a custom left and right indent, depending on the style you want to achieve.

Note In programming, when a class is *deprecated*, it has been decommissioned, isn't supported, and is no longer considered acceptable for use. Often when a class is deprecated, it remains available for use; however, because it's unsupported, it may have consequences with other areas of your application and may cause unexpected issues with other classes.

Manipulating Static Table Views

Let's take a look at how Xcode allows you to manipulate a static layout, before going back and implementing the dynamic prototype system you're using for this scene:

1. Select the Accounts table view, and then open the Attributes Inspector. Set the Content attribute to Static Cells; you should notice that the single cell you had becomes three cells, and the second attribute becomes Sections.
2. Let's increase the number of sections so that there are two groups of cells to work with: change the Sections attribute to 2. Your table should now resemble the one shown in Figure 8-7.

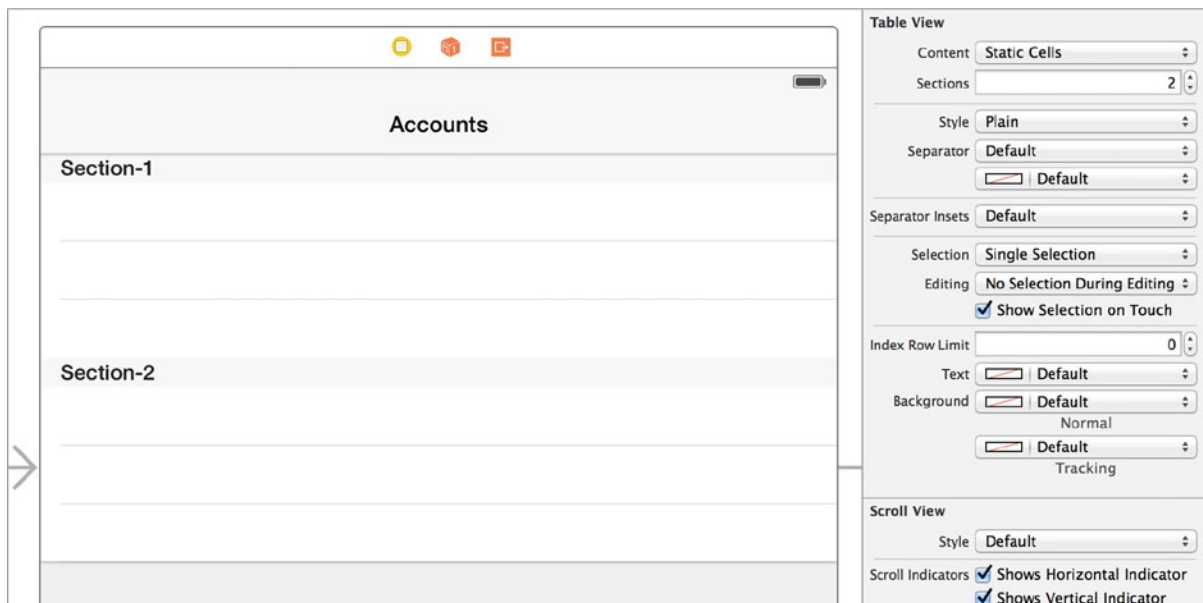


Figure 8-7. The static layout with two sections

3. The number of cells in a section is controlled directly by the section attributes. Select the first section by clicking Section-1 in the view or by expanding Table View in the Document Outline and selecting Section-1, as shown in Figure 8-8.

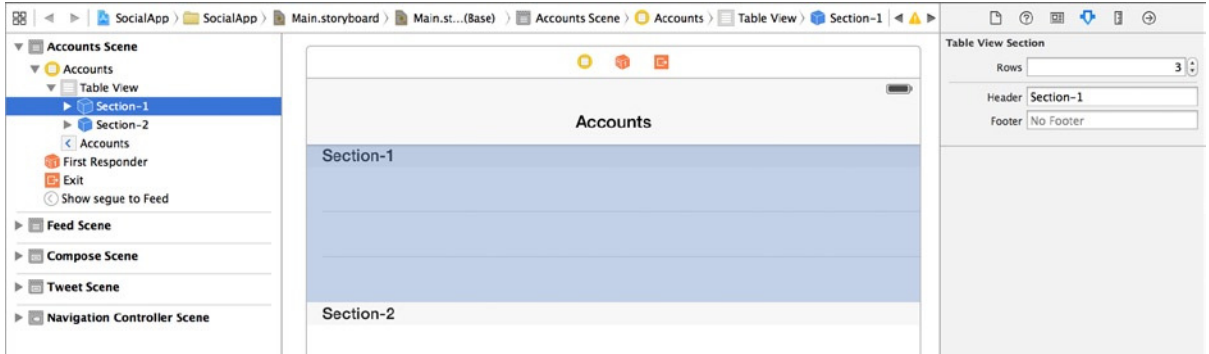


Figure 8-8. Selecting *Section-1* and viewing the section attributes

4. In the Attributes Inspector, the terminology changes slightly from cells to rows. Looking at the attributes, you can see that here you specify the header and footer value of your section and also the number of rows. Feel free to try setting your own values for these attributes and see how the table changes.
5. You can also delete individual cells and move them around. In *Section-2*, delete two cells; click the cells individually, and then press the Backspace key to remove them. Select the single remaining cell.
6. In the Attributes Inspector, change *Style* to *Basic*; doing so adds the word *Title* to the cell.
7. Double-click the word *Title* to edit it, as shown in [Figure 8-9](#), and change it to read *Cell 1*.



Figure 8-9. Changing the title of the basic cell

8. Press the Return key to commit the change, and then reselect *Section-2*. Now increase the number of rows from one to three by changing the *Rows* attribute to 3. This allows you to clone your row three times. This is a really handy way of duplicating a custom cell when using static cells instead of dynamic prototypes.
9. Rename each of your new cells by double-clicking *Cell 1* and changing them to 2 and 3, respectively, so that *Section-2* resembles [Figure 8-10](#).

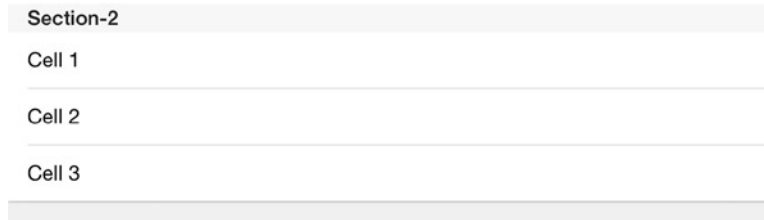


Figure 8-10. Section-2 with three basic style cells

- To demonstrate how easy it is to reorder a static table view, highlight Cell 3 with a single click and then drag it to the top of Section-2. A solid blue line appears, as shown in Figure 8-11. You can also move cells between sections, meaning that changing your layout needn't be a chore.

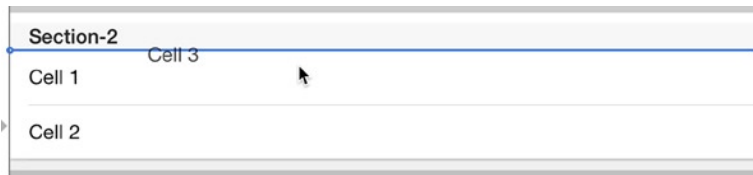


Figure 8-11. Selecting the third cell and dragging it to the first position

- Now that you've seen the various ways Xcode lets you manipulate static layouts, you're ready to get the Accounts scene built up and working. Reselect the table view, change the Content attribute to Dynamic Prototypes, and ensure that Style is set to Grouped. This will leave you with three cells in the single section.
- You only want one prototype cell, so delete any excess cells by manually highlighting them and pressing the Backspace key until you're left with a single cell, ready for customizing.
- Highlight the one remaining cell, and go to the Attributes Inspector. The style should currently be set to Custom, which is fine because you'll be setting the content programmatically.
- Set the Accessory attribute to Disclosure Indicator, which adds the indicator arrow on the right side of the cell.
- Give your cell an identifier so that you can refer to it in code and reuse it efficiently. Set the Identifier attribute to `AccountCell`. The cell's attributes should resemble those shown in Figure 8-12.

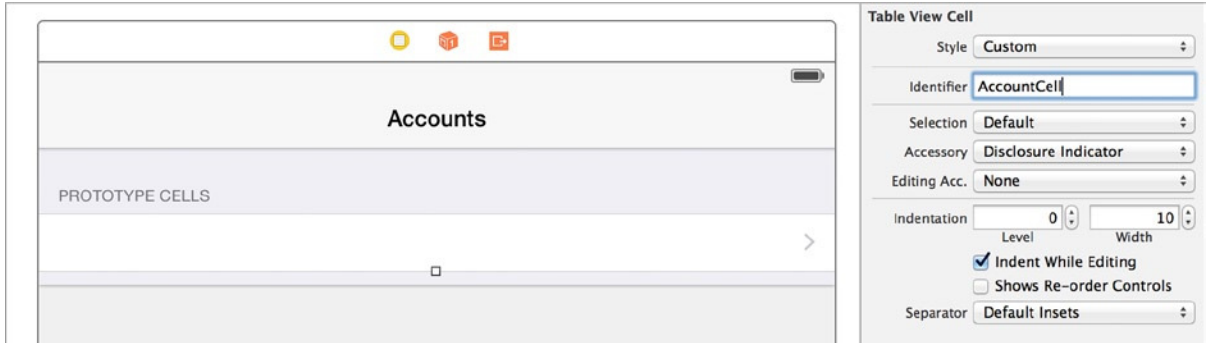


Figure 8-12. The table cell configured, ready to have its content set

You're finished with the layout and design of your table view, and it's time to add the code to access the Twitter accounts on the device and display them in the table. To do this, you need to add the Accounts and Social frameworks to the project.

The Accounts and Social Framework

In previous chapters, you've taken advantage of some of the Apple-provided frameworks, and in this chapter you use two: `Accounts.framework` and `Social.framework`. Like the other frameworks you've used, they make potentially complex and intensive tasks much simpler. Based on the fact that this is the first time you've worked with these two frameworks, it's also worth noting that they work really well together, and you sometimes need to use both in order to make the most of their functionality. Accounts framework classes are prefixed with `AC`; in this project, you create an instance of the `ACAccount` class to hold the details of a selected Twitter account. You hand that `ACAccount` object to other view controllers as you navigate through the project and then combine it with one of the Social framework's classes, `SLRequest`, which uses the `ACAccount` object to authenticate requests with Twitter.

If you've worked with Twitter's APIs in the past, you know that authentication involves a process called *three-legged OAuth* that sends a number of requests back and forth with Twitter. The combination of the `ACAccount` object with the `SLRequests` means you don't have to do any of that. iOS does all the running around so that you're free to focus on functionality and how you handle the requests to Twitter.

Now that you understand the roles of the frameworks you need to add to the project, you're ready to begin writing the code that will display Twitter accounts in the table view.

Retrieving and Displaying Twitter Accounts

To finish this scene, you need to achieve two objectives: retrieve an array of Twitter accounts registered on the device or in the Simulator, and display them for users to choose from. Let's start by setting up the view controller's header, importing the frameworks, and creating the properties that are needed for this scene. As in previous chapters, I explain what needs to be done bit by bit and then review the code at the end of the process:

1. Open `AccountsViewController.swift`. You first need to import the Accounts framework that is used for this scene. After the line `import UIKit`, add the following highlighted line:

```
import UIKit
import Accounts
```

Note API stands for *application programming interface*. An API is a mechanism that specifies how different pieces of computer software interact with one another. In iOS, Apple uses the term *API* to describe new classes in frameworks. When I talk about the Twitter API, I don't mean classes in the Social framework, but rather the Twitter REST API, which you can learn more about at <http://dev.twitter.com>.

2. You need to create an array to store the retrieved accounts and make them available to all functions in this class. Add the highlighted code:

```
import UIKit
import Accounts

class AccountsViewController: UITableViewController {

    var twitterAccounts : NSArray?
```

3. You need to declare an instance of `ACAccountStore`. The `ACAccountStore` class is the gateway to the list of Twitter and other social media accounts stored in iOS; you declare it and then try to get permission from the user to access the Twitter account. If the user grants permission, the object becomes initialized. Add the highlighted code after the `NSArray`:

```
var twitterAccounts : NSArray?
var accountStore : ACAccountStore?
```


So far, you've added a reference to the Accounts framework, created an array called `twitterAccounts`, and created an `ACAccountStore` object to manage the retrieval of Twitter accounts. Your initial code should look like this:

```
import UIKit
import Accounts

class AccountsViewController: UITableViewController {

    var twitterAccounts : NSArray?
    var accountStore : ACAccountStore?
```

4. To get into the nitty-gritty, scroll down until you see the `viewDidLoad` function. The first thing you want to do when the view loads is initialize the `ACAccountStore` instance. That doesn't mean you're accessing the accounts; you're just initializing the object so that it can be interacted with. Drop down a line after `[super viewDidLoad];` and type the following highlighted line:

```
super.viewDidLoad()
accountStore = ACAccountStore()
```

5. You'll use the `requestAccessToAccountsWithType` method of the `accountStore` object. This method needs to be told the type of account to which you're requesting access. You do this by creating an `ACAccountType` object and then using another `accountStore` method: `accountTypeWithAccountTypeIdentifier`. Drop down a line, and add this highlighted code (as a single line):

```
super.viewDidLoad()
accountStore = ACAccountStore()
var accountType : ACAccountType = accountStore!.  
accountTypeWithAccountTypeIdentifier(ACAccountTypeIdentifierTwitter)
```

6. You're at the stage where you want to ask the user for permission to use their Twitter accounts in the application using the `requestAccessToAccountsWithType` method. When this method is accessed, it creates an alert for the user to either grant or deny the request to access their Twitter account. Add the following highlighted code:

```
var accountType : ACAccountType = accountStore!.  
accountTypeWithAccountTypeIdentifier(ACAccountTypeIdentifierTwitter)

accountStore?.requestAccessToAccountsWithType(accountType, options: nil,  
completion: { granted, error in

}}
```

7. Notice that you pass the `accountType` object into the method to specify that it wants Twitter account access, and you handle completion using a code block into which you add the logic as to whether granted returned yes or no. Because you only want to look at the accounts available *if* access was granted, that should be the next thing you check. To do so, add the highlighted `if` statement in the code block as shown next:

```
var accountType : ACAccountType =
    accountStore!.accountTypeWithIdentifier(ACAccountTypeIdentifierTwitter)

accountStore?.requestAccessToAccountsWithType(accountType, options: nil,
    completion: { granted, error in
        if(granted)
        {

        }
    })
```

8. With access granted, you need to populate the `twitterAccounts` object with all the available Twitter accounts on the device. Here you use the `accountsWithType` method of the `accountStore` and reuse the `accountType` object to restrict the request to Twitter accounts. In the `if` statement, add the following highlighted code:

```
if(granted)
{
    self.twitterAccounts = self.accountStore!.accountsWithType(accountType)
}
```

9. Although the user has granted access, you need to check whether user has added any Twitter accounts to the device. This is done by checking the `twitterAccounts` `count` property in an `if else` statement. After the previous line, drop down and add this code:

```
if(granted)
{
    self.twitterAccounts = self.accountStore!.accountsWithType(accountType)

    if (self.twitterAccounts?.count == 0)
    {

    }
    else
    {

    }
}
```

10. If there are no Twitter accounts stored in iOS, you want to summon an alert view and tell the user that no accounts were found. Because you're running on an arbitrary thread and all interface changes need to be executed on the main thread, you add a Grand Central Dispatch call to execute the display of the alert view. Add the following highlighted code in the first set of braces:

```

if (self.twitterAccounts?.count == 0)
{
    var noAccountsAlert : UIAlertController = UIAlertController(title: "No Accounts Found",
        message: "No Twitter accounts were found.",
        preferredStyle: UIAlertControllerStyle.Alert)

    var dismissButton : UIAlertAction = UIAlertAction(title: "Okay",
        style: UIAlertActionStyle.Cancel) {
        alert in
        noAccountsAlert.dismissViewControllerAnimated(true, completion: nil)
    }

    noAccountsAlert.addAction(dismissButton)

    dispatch_async(dispatch_get_main_queue()) {
        self.presentViewController(noAccountsAlert, animated: true, completion: nil)
    }
}
else
{
}

```

11. If there are Twitter accounts, they're added to the `twitterAccounts` object, and you just need to tell the table view to reload the data shown in the table. This is done by accessing the `reloadData` method. In the second set of braces for the `else` statement, add this highlighted lines of code:

```

else
{
    dispatch_async(dispatch_get_main_queue()) {
        self.tableView.reloadData()
    }
}

```

You've added the code that pulls together a dataset for the table view in the shape of the `twitterAccounts` array. Now you need to get that data into the cells by using two of the `UITableView` class's delegate methods: `numberOfSectionsInTableView` and `numberOfRowsInSection`. These two methods control the number of sections displayed in the table and also the number of table cells per section by returning an `NSInteger`, a numeric value that the table view interprets. Setting up these two methods is pretty simple in this application: you only ever have one section, and the number of cells is the count property from the `twitterAccounts` object. Furthermore, Apple has already added stubs for these two methods.

12. Scroll down until you find the `numberOfSectionsInTableView` method; or click `AccountsViewController` in the jump bar and select `numberOfSectionsInTableView`, as shown in Figure 8-13.

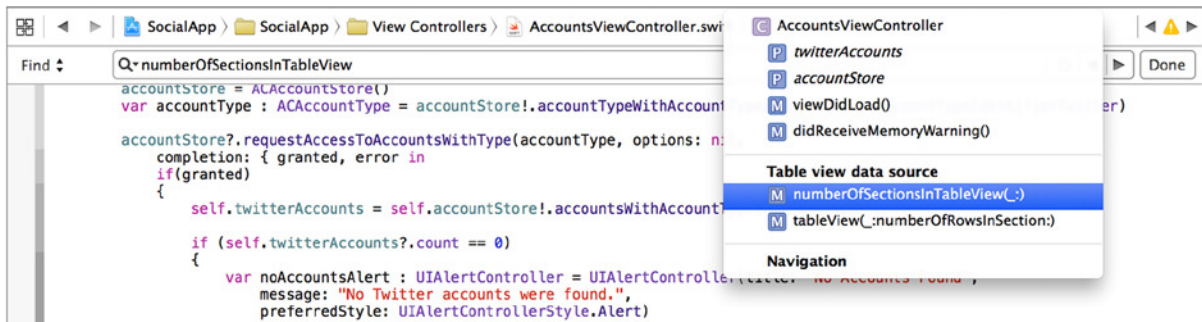


Figure 8-13. Navigating code using the jump bar

13. You may notice that, confusingly, a warning appears beside the method declaration and also the one below it. This is a casualty of Apple changing the way Swift is handled by Xcode; it's not something you've done. No doubt Apple will resolve this in the future. If you don't see the warning, ignore this step; but if you do, click the warning triangle. A pop-over appears above the method, as shown in Figure 8-14; click the first Fix-it option to remove the `!` from the parameter. Repeat this for the warning against the other method as well.

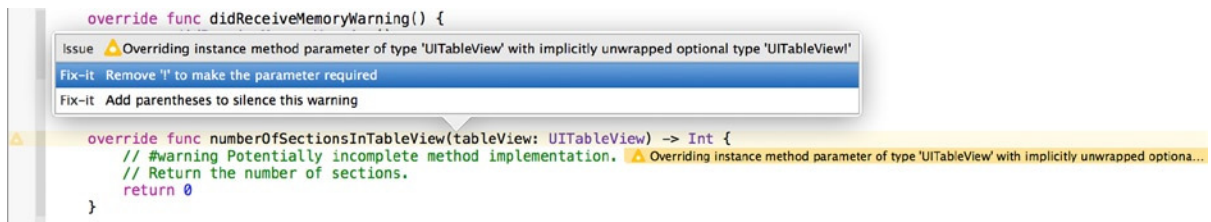


Figure 8-14. Fixing a warning in Xcode

14. As previously mentioned, the `numberOfSectionsInTableView` method returns a value that determines how many sections appear in the table view. In this instance, you only want a single section, so change the return value to 1 as shown next:

```
override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
    // Return the number of sections.
    return 1
}
```

15. You need to set the `numberOfRowsInSection` method to return the number of items held in the array so that the application knows how many cells to create in the table view. Because there may be no items in the array, and you don't want an exception, add the highlighted `if` statement as shown next:

```
override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    // Return the number of rows in the section.

    if let cellCount = self.twitterAccounts?.count {
        return cellCount
    }
    else
    {
        return 0
    }
}
```

16. The application will try to create an instance of your prototype cell if there are any Twitter accounts set up in the Simulator, so it's important to set the correct reference name for the cell before trying to run the application. I explain this in more detail once you've run the application. Look for the `cellForRowAtIndexPath` method: it should be just below the last method you changed. It's commented out by default, so remove the `/*` from the start and the `*/` from the end of the method.
17. At this point, you may get a red error indicator next to the overridden function. To resolve this, remove all `!` from the function definition so that it matches the following line:

```
override func tableView(tableView: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
```

18. Focusing on the contents of the function, you can see that an object called `cell` is initialized and then the last line returns the `cell` object. This object is a `UITableViewCell`. When it's initialized, it's passed a value that currently says "reuseIdentifier" but that needs to say "AccountCell", which is the name you gave the table cell when you configured the table view in the storyboard. Go ahead and change the highlighted value as shown next:

```
let cell = tableView.dequeueReusableCellWithIdentifier("AccountCell",
    forIndexPath: indexPath) as UITableViewCell
```

19. It's been a very long time coming, but you're at a point where you can run the application in the Simulator. Go to Product ► Run (⌘+R) to launch the application.

The first thing you should see is a prompt for access to the device's Twitter accounts, as shown in Figure 8-15. It's important to click OK at this point to grant access; you're then presented with one of the two outcomes, depending on the number of Twitter accounts you have. If there are no Twitter accounts installed, you get the alert view warning you that no accounts were found; otherwise you see a single row with an arrow in your table view.

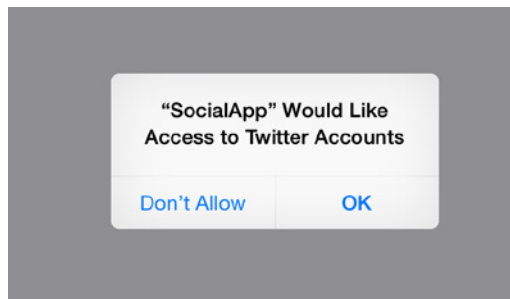


Figure 8-15. The security prompt asking for access to the Twitter accounts for the application

20. If you don't have any Twitter accounts added in iOS and you saw the alert, adding your Twitter account is very easy. In the Simulator, return to the home screen by going to Hardware ► Home (⌘+Shift+H). Navigate to the first page of icons, click the Settings icon, scroll down, and select the Twitter option from the left column, as shown in Figure 8-16.

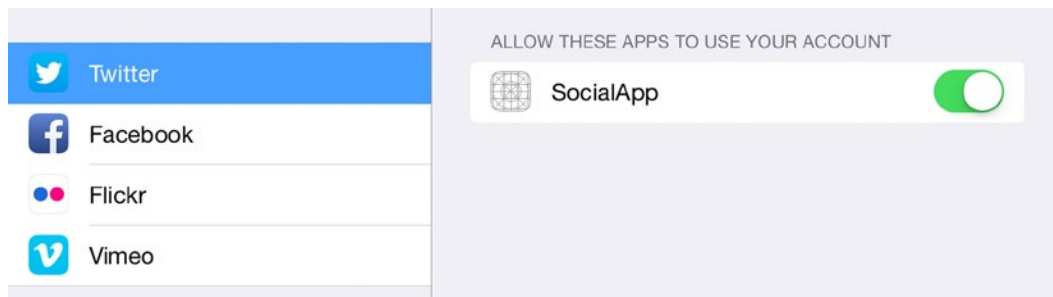


Figure 8-16. Selecting the Twitter settings

Note If you accidentally refuse permission to the application, you can grant permission from the Twitter settings. SocialApp is listed at the bottom of the Twitter settings with a switch beside it, as shown in Figure 8-16. Turn this to the on position, and permission will be granted to access the accounts.

21. Type in your Twitter account name and password, and click Sign In. You can repeat this step to add more Twitter accounts if you wish. When you're done, go back to Xcode and rerun your application; this time your view should resemble Figure 8-17.



Figure 8-17. The Accounts view with a single row showing

Although the application runs, you still have to make the table view display the account name in the cell (currently it's empty); and to do that, you need to add some code to the delegate method `cellForRowAtIndexPath`. This is the method you quickly altered before running the application. All delegate methods are called as the result of an event occurring. In this case, it's the table view responding to the number of rows it has been told to display and then calling the `cellForRowAtIndexPath` method to allow it to populate the specific cell's contents. To display the correct account information in the cell, the application needs to know which row it's on so that the corresponding entry can be fetched from the array. You can establish this by looking at the method variable `indexPath`. This object has properties indicating the cell's section and row numbers: both follow the array format for positioning, starting at 0 and incrementing by 1. Because there is only one section, the row value corresponds to the position of elements in the `twitterAccounts` array. For example, if the `indexPath` row property is 0, the application will fetch the account at position 0 in the array.

22. Remove the comment `// Configure the cell...` and in its place create an `ACAccount` object based on the account stored at the supplied position in the array:

```
let account = self.twitterAccounts!.objectAtIndex(indexPath.row) as ACAccount
```

23. Put a value in the table view cell that shows the name of the account in the array. The table cell is currently the default `UITableViewCell`, which has three controls that can be manipulated: a label called `textLabel`, a subtitle label called `detailTextLabel`, and an image view called `imageView`. Let's take the `accountDescription` property of the account object and use it to set the `textLabel`'s `Text` property, as shown in the highlighted code:

```
let account = self.twitterAccounts!.objectAtIndex(indexPath.row) as ACAccount
```

```
cell.textLabel.text = account.accountDescription
```

24. The complete method should look like this:

```
override func tableView(tableView: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {

    let cell = tableView.dequeueReusableCellWithIdentifier("AccountCell",
        forIndexPath: indexPath) as UITableViewCell

    let account = self.twitterAccounts!.objectAtIndex(indexPath.row) as ACAccount

    cell.textLabel.text = account.accountDescription

    return cell
}
```

25. Run your application again; this time the table view should be populated as shown in Figure 8-18. What's more, if you select an account, you're taken to the `FeedViewController` scene. But beware: if you click the `Compose` button, the view appears and looks great, but you haven't yet written the code to dismiss the view controller and thus are stuck unless you rerun the application from Xcode.



Figure 8-18. The table view showing a list of Twitter accounts

One great feature to take note of here is that as you show the Feed view controller, you're automatically given a button that takes you back to the Accounts view controller without even having to write any code.

Before you focus on the next scene—the Feed view controller—you need to think ahead slightly. When an account is selected, that selection is currently retained in the Accounts view controller; you need to pass it on, a bit like a baton in a relay, to the Feed view controller when the segue is triggered. To do this, you need to create a custom initializer in the Feed view controller to receive the selected `ACAccount` object to be passed across.

26. Open `FeedViewController.swift` from the Project Navigator. Right after the class definition, type the following highlighted line:

```
class FeedViewController: UITableViewController {

    var selectedAccount : ACAccount!
```

27. Because you need to refer to both the Accounts and the Social framework, add the following two highlighted import statements after `import UIKit`:

```
import UIKit
import Accounts
import Social
```

You've now created a property in your `FeedViewController` class that can receive the baton, or in this case the selected `ACAccount`, from the Accounts view controller. All that remains is to pass the object across when the segue is called.

28. Open `AccountsViewController.swift` once more from the Project Navigator.
29. If you weren't using segues to navigate between view controllers, you would use the `didSelectRowAtIndexPath` method to determine what to do next; but because you're using a segue, you use the `prepareForSegue` method. Yet again, the good folks at Apple have already written a basic implementation of the method for you, but it's currently commented out. At the bottom of the `AccountsViewController.swift` file, you should see the method you need, commented out in green (assuming you haven't deviated from the color scheme). Before the line `// MARK: - Navigation` are the characters `/*` that index the start of a commented block of code; remove them. Next, look for `*/` just before the last `};`; this signifies the end of the comment block. Remove it as well. The method is now uncommented and ready for use.
30. Remove the erroneous `!` exclamation mark after the `UIStoryboardSegue` parameter from the `prepareForSegue` method.

31. The `prepareForSegue` method is called when a segue is about to be triggered. It gives you a chance to perform any actions that need to be processed before the view changes. One of the parameters passed to this method is a `UIStoryboardSegue` object called `segue`; you can use this to check the segue identifier and then take appropriate action. The identifier for this segue to the Feed view controller scene is `ShowTweets`. To check for this in an `if` statement, add the highlighted code to the method:

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject!) {  
    if(segue.identifier == "ShowTweets")  
    {  
    }  
}
```

32. You need to find out which account was selected. Create an `ACAccount` object from the selection, and pass that to the Feed view controller's `selectedAccount` property. The first task is to determine which row was selected. You do this by creating an `NSIndexPath` object called `path` based on the result of the `indexPathForSelectedRow` method. Add the following highlighted line to your `if` statement:

```
if(segue.identifier == "ShowTweets")  
{  
    var path : NSIndexPath = self.tableView.indexPathForSelectedRow()!  
}
```

33. Create the `ACAccount` object, called `account`. This is almost an exact duplicate of when you instantiated an `ACAccount` object in the `cellForRowAtIndexPath` method, as the highlighted code shows:

```
if(segue.identifier == "ShowTweets")  
{  
    var path : NSIndexPath = self.tableView.indexPathForSelectedRow()!  
  
    let account = self.twitterAccounts!.objectAtIndex(path.row) as ACAccount  
}
```

34. Pass the account object to the Feed view controller. When a segue is triggered and this method is called, the destination view controller is stored in a property of the segue object called `destinationViewController`. Because you know that the destination view controller is a Feed view controller, you cast `destinationViewController` from a generic `AnyObject` type to be a `FeedViewController` type. Once you've created a `FeedViewController` object, you simply take the account object and give it to `selectedAccount`. All of these actions are done in the following highlighted code:

```
if(segue.identifier == "ShowTweets")
{
    var path : NSIndexPath = self.tableView.indexPathForSelectedRow()!

    let account = self.twitterAccounts!.objectAtIndex(path.row) as ACAccount

    let targetController = segue.destinationViewController as FeedViewController
    targetController.selectedAccount = account
}
```

That's it! You've finished writing the code for the Accounts view controller and even added a bit to the Feed view controller. That's one down and five to go. Let's move on to the Feed view controller, where you build on your table-view skills and learn about creating custom cells and subclassing `UITableViewCell` to take customizations to another level.

Configuring the Feed View

The Feed view is the center point of `SocialApp`; it lists the 20 latest tweets using some of the methods and classes used in the previous Accounts view, along with many that haven't been encountered in the app so far. In the sections that follow, you learn how to

- Use the `SLRequest` class to fetch the JSON-formatted data from the Internet
- Use an `NSCache` object to handle some basic caching
- Use an `NSOperationQueue` to streamline the retrieval of Twitter avatar images
- Subclass `UITableViewCell` to create a custom class for the cells in the table view

Because I really want to focus on the table view and how to populate it, I won't go into much detail about the code used for retrieving the data from the Twitter API or processing it. You've already learned how to access the documentation for different classes, so if you want to know more, I encourage you to look at the documentation and then branch out onto the Internet if you want to know more.

Figure 8-19 shows what the rows in the finished table view will look like. Before I get into the code, you need to build the cell's interface and link it to a custom `UITableViewCell` class. Here are the steps:

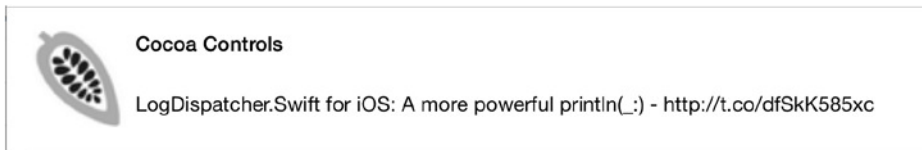


Figure 8-19. A look ahead at a row from the finished Feed view

1. You subclass `UITableViewCell` the same way you did `UITableViewController`. Right-click the `SocialApp` group in the Project Navigator, and select New File ($\mathbb{C}+N$), as shown in Figure 8-20.

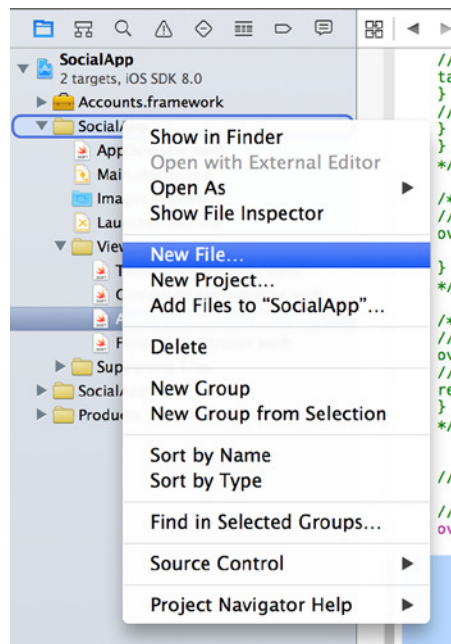


Figure 8-20. Adding a new file to the `SocialApp` group in the Project Navigator

2. Select the Cocoa Touch Class option, which should be selected by default, and click Next. Set the Subclass Of value to `UITableViewCell` and the Class value to `TweetCell`. There is no need to create an XIB file, so leave that unchecked and click Next. As always, you want to save the file in the project folder. Click Create to create the file and add it to the project. You're now ready to set up the visual elements of the table view.

3. Open Main.storyboard from the Project Navigator, and arrange the view so you can see the Feed scene and are at 100% zoom, as shown in Figure 8-21.

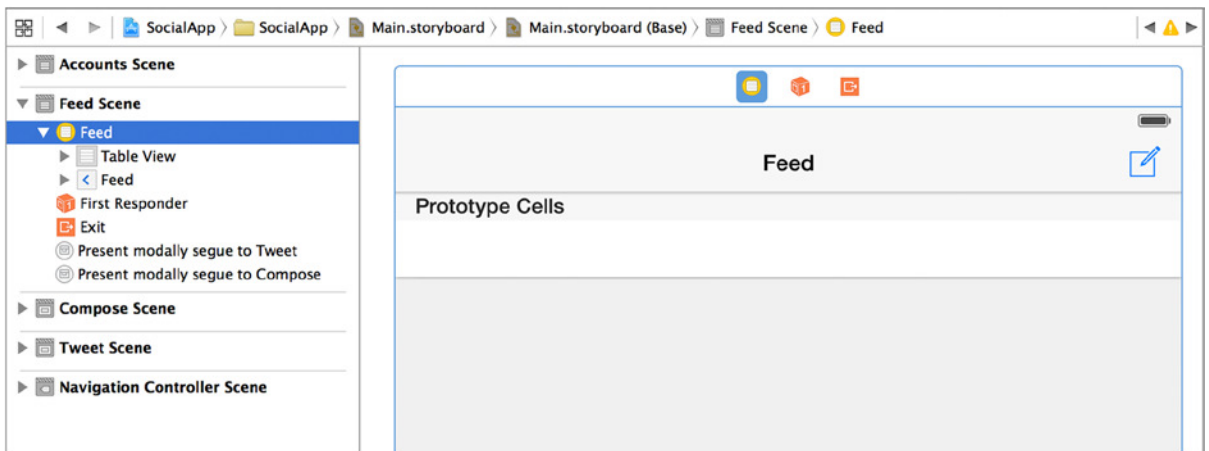


Figure 8-21. The storyboard file, open and ready for you to build the interface

4. The default row height of the cell is far too small to display everything nicely. To resize the cell, select the table view as you did in the previous scene by clicking the view where it says Table View Prototype Content or by selecting Table View from the Feed view controller scene in the Document Outline. Next, open the Size Inspector and set the Row Height value to 120, as shown in Figure 8-22.

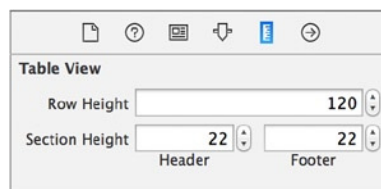


Figure 8-22. Adjusting the height of the row in the Size Inspector

5. Before you start creating the interface of the cell, you need to specify that the cell is controlled by the new TweetCell class. Select the cell, open the Identity Inspector, and, from the Class drop-down list, select TweetCell.
6. Open the Attributes Inspector for the cell. You need to specify a reuse identifier here, so in the Identifier attribute, type **TweetCell**.

- Now that there is plenty of room to work, you can add the controls: an image view and two labels. Start by dragging in an image view from the Object Library onto the cell. It tries to fill the view, but don't worry; put it anywhere, and go back the Size Inspector. Set the X and Y values to 20 and the Height and Width values to 79. The view should resemble that shown in Figure 8-23.

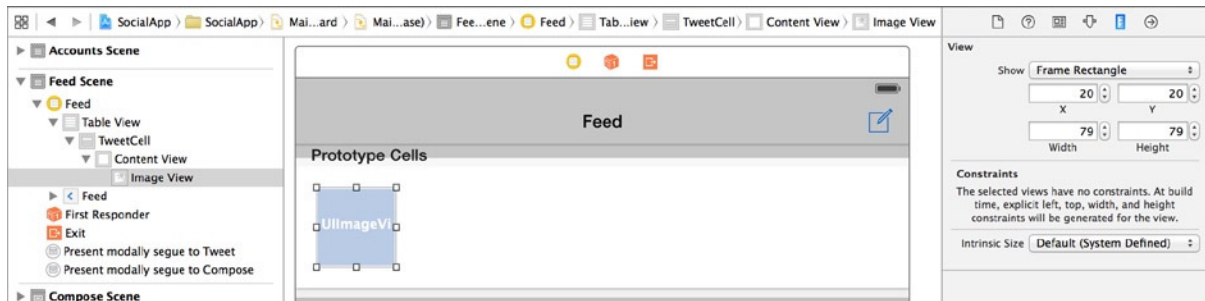


Figure 8-23. Manually sizing and positioning the image view

- This image view will display the Twitter photo of the tweet's author. Before you leave the image view, you need to set a default image to act as a placeholder while the image is downloading from the Internet. Open `Images.xcassets` from the Project Navigator: in the project files for this chapter is a folder called `images` that contains a file called `camera.png`. Drag that file from Finder to the Asset Catalog sidebar, as shown in Figure 8-24.

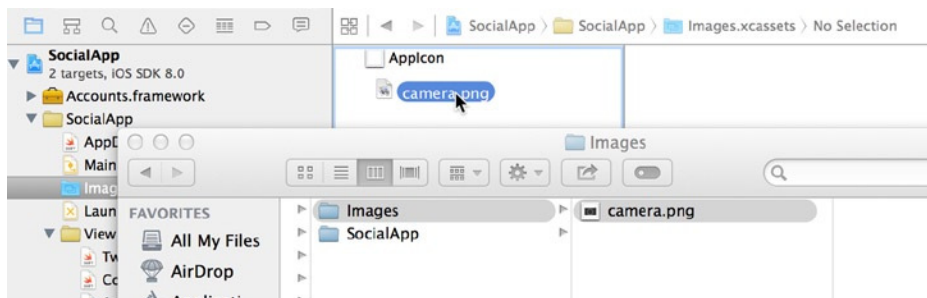


Figure 8-24. Dragging in `camera.png`

- When you release the file, it automatically creates an image set named `camera`. Switch back to the storyboard, select the image view if it isn't already selected, and open the Attributes Inspector.
- Set the Image attribute to `camera`, which should be shown in the list of available images. Your image view should now resemble Figure 8-25.

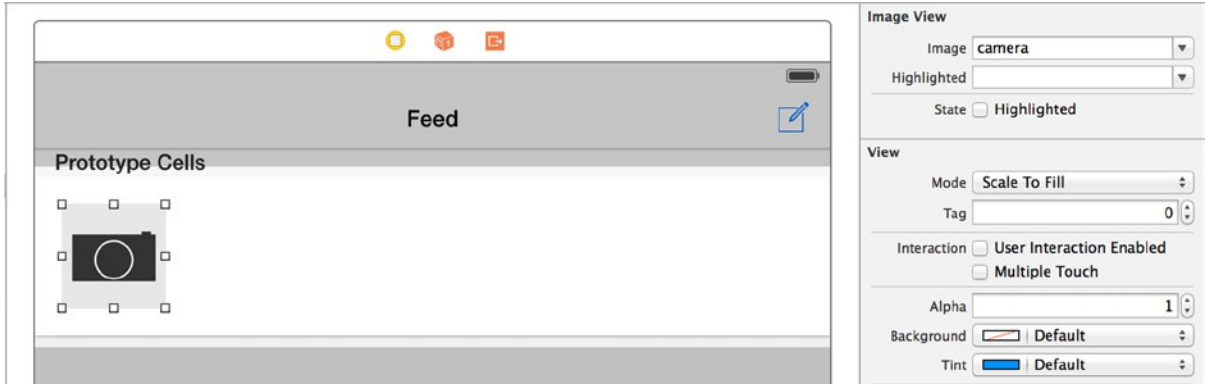


Figure 8-25. Setting the attributes of the image view so that it has a placeholder image

11. Add a label for the user name of the author of the tweet. To do so, drag in a label from the Object Library, and align it with the top of the image view. In the Attributes Inspector, change the Text attribute to User Name. Using the T icon, change the Font attribute to System Bold and Size to 17.
12. With the font and the placeholder text set correctly, size the label appropriately. Keep it at a single-line height, but increase the width to the right until the blue margin appears. It should resemble Figure 8-26.

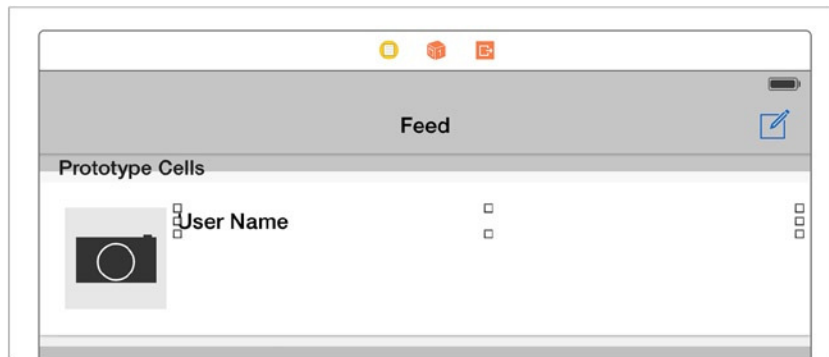


Figure 8-26. Aligning the User Name label

13. The final object that needs to be added to the cell is a label for the tweet content. Drag in a label, and position it just below the User Name label. Make it the same width as the User Name label, and then increase the height until the margin guidelines at the bottom of the cell appear. It should resemble Figure 8-27.

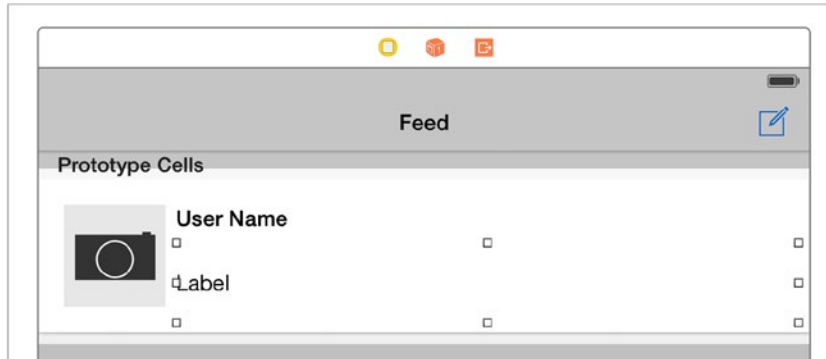


Figure 8-27. *Aligning the tweet Content label*

14. In the Attributes Inspector, set the Text attribute to Content and the Lines attribute to 2. Xcode will wrap the text onto a second line if needed. If the text length exceeds what will fit on two lines, the Line Breaks attribute will determine what will happen. The default option is Truncate Tail, which cuts the text short and appends an ellipsis or ... to the end of the text.
15. To align the interface elements, click the Resolve Auto Layout Issues button. Under All Views in TweetCell, choose Add Missing Constraints.
16. You've now built the interface for the custom cell. Next you need to create the outlets for the three objects. Open the Assistant Editor with the cell selected, and this time ensure that the code file that is loaded is TweetCell.swift; it's likely that the file loaded is actually FeedViewController.swift. Click the file name on the jump bar, as shown in Figure 8-28, and then choose TweetCell.swift.

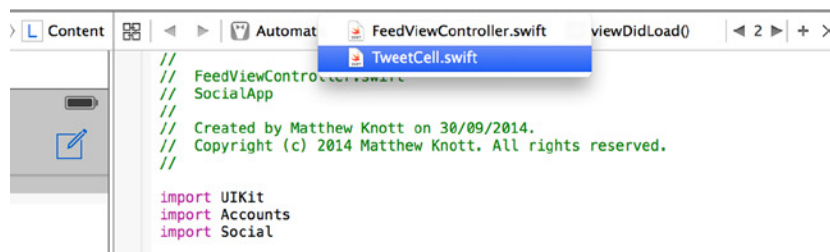


Figure 8-28. *Ensuring that TweetCell.swift is loaded before creating the outlet*

17. Control-drag a connection from the image view to just below class TweetCell, and release the mouse. Name this outlet tweetUserAvatar, as shown in Figure 8-29.

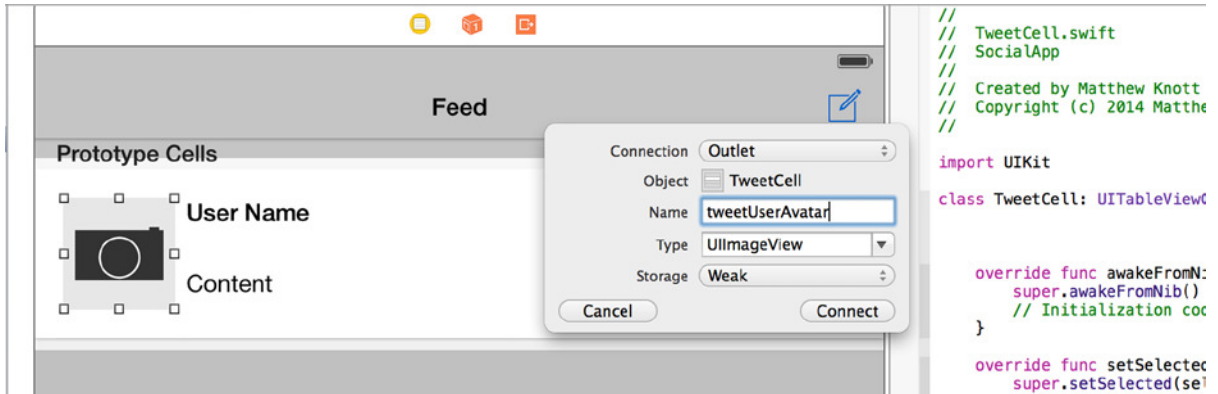


Figure 8-29. Creating an outlet for the image view called *tweetUserAvatar*

18. Create an outlet in the same way for the User Name label, naming it `tweetUserName`. Finally, create an outlet for the Content label named `tweetContent`. The start of your custom `UITableViewCell` class should resemble the following code:

```
import UIKit

class TweetCell: UITableViewCell {
    @IBOutlet weak var tweetUserAvatar: UIImageView!
    @IBOutlet weak var tweetUserName: UILabel!
    @IBOutlet weak var tweetContent: UILabel!
}
```

Now that you've created the interface and the outlets for the objects in the cell's view, you can begin to bring all the different elements and classes together in the Feed view controller. Here you write the code that fetches the Twitter feed and then parses the returned data before displaying it in the custom table cell:

1. To get started, prepare the header file. Switch back to the Standard Editor, and then open `FeedViewController.swift` from the Project Navigator. You need to create some instance variables, just as you did for `AccountsViewController`, but this time you're creating three. After `var selectedAccount : AAccount!`, add the following highlighted code:

```
class FeedViewController: UITableViewController {

    var selectedAccount : AAccount!
    var tweets : NSMutableArray?
    var imageCache : NSCache?
    var queue : NSOperationQueue?
```

Note You've got a lot of code to get through for this view controller. As I already mentioned, I won't be going into a huge amount of detail, but be assured that much of the code you're writing in this chapter is interchangeable with any kind of application that fetches data from the Internet.

2. Scroll down to the `viewDidLoad` method. Because this method is called when the view loads, you perform a few key tasks here. First, clear out all the green commented lines of code so you're left with just `super.viewDidLoad()`.
3. You need to set the title of the view to the Twitter account name that was passed to the view from the previous Accounts view controller, initialize `NSOperationQueue` and configure its basic settings, and finally add a call to a function that hasn't been written yet called `retrieveTweets`. Add the highlighted code to your `viewDidLoad` method:

```
override func viewDidLoad() {
    super.viewDidLoad()

    self.navigationItem.title = selectedAccount.accountDescription

    queue = NSOperationQueue()
    queue!.maxConcurrentOperationCount = 4

    retrieveTweets()
}
```

4. Xcode correctly flags the last line of the `viewDidLoad` method as being in error. This is because you haven't written that function yet. Drop down a few lines after the `viewDidLoad` method, and declare the function stub as shown next:

```
func retrieveTweets() {
}

```

5. Xcode is happy that everything is back in order. You still need to write this method's substantial code. I'll take you through each major block of code and explain the function as I go, as opposed to going line by line as I have in the past. First you need to clear the `tweets` array to remove any previously stored tweets. Add the highlighted code to start the function:

```
func retrieveTweets() {
    tweets?.removeAllObjects()
}
```

6. You need to check that you do indeed have a valid `ACAccount` object. If so, you declare and initialize an `SLRequest` object with the URL to the Twitter API that provides the home timeline data you want to display in the table view. You then authenticate the request using the `selectedAccount` `ACAccount` object. Drop down a line and add this highlighted code:

```
func retrieveTweets() {
    tweets?.removeAllObjects()

    if let account = selectedAccount {
        let requestURL =
            NSURL(string: "https://api.twitter.com/1.1/statuses/home_timeline.json")

        let request = SLRequest(forServiceType: SLServiceTypeTwitter,
                                requestMethod: SLRequestMethod.GET,
                                URL: requestURL,
                                parameters: nil)

        request.account = account
    }
}
```

7. For the final block of this method, you've given the request object all the parameters it needs, and now you execute the `performRequestWithHandler` method. This method accesses the supplied URL and returns the response from the request to a code block. If the request is successful, it returns a status code of 200. When this happens, you parse the JSON code into an array and use that as the contents of the tweets array. Finally, you call the `reloadData` method of the table view to update the information shown on the screen. Add the highlighted code after the last line you wrote:

```
request.account = account
request.performRequestWithHandler()
{
    responseData, urlResponse, error in

    if(urlResponse.statusCode == 200)
    {
        var jsonParseError : NSError?
        self.tweets = NSJSONSerialization.JSONObjectWithData(responseData,
                                                                options: NSJSONReadingOptions.MutableContainers,
                                                                error: &jsonParseError) as? NSMutableArray
    }

    dispatch_async(dispatch_get_main_queue()) {
        self.tableView.reloadData()
    }
}
```

Note If you aren't familiar with http response codes, it may be worth looking up the possible codes online. Even if you've never heard the term before, you've almost certainly come across them while browsing the Internet. Errors 404 and 500 are two of the more visible error codes that you may have seen on a web site in the past, but there are many others, and it's worth doing some research on them if you intend to use web APIs to get data into your application.

8. The completed code for the `retrieveTweets` method should look like this:

```
func retrieveTweets() {
    tweets?.removeAllObjects()

    if let account = selectedAccount {
        let requestURL =
            NSURL(string: "https://api.twitter.com/1.1/statuses/home_timeline.json")
        let request = SLRequest(forServiceType: SLServiceTypeTwitter,
            requestMethod: SLRequestMethod.GET,
            URL: requestURL,
            parameters: nil)

        request.account = account
        request.performRequestWithHandler()
        {
            responseData, urlResponse, error in

            if(urlResponse.statusCode == 200)
            {
                var jsonParseError : NSError?
                self.tweets = NSJSONSerialization.JSONObjectWithData(responseData,
                    options: NSJSONReadingOptions.MutableContainers,
                    error: &jsonParseError) as? NSMutableArray
            }

            dispatch_async(dispatch_get_main_queue()) {
                self.tableView.reloadData()
            }
        }
    }
}
```

9. This is a good point at which to run your application to check for errors. The application should compile and allow you to select a Twitter account. On the feed screen, you don't see anything yet; but more important, you shouldn't see any errors. If you do, check things such as the correctness of the name of the segue in the storyboard and whether you typed the URL correctly.

10. Back in `FeedViewController.swift`, it's time to look at the table-view delegate methods. Starting with `numberOfSectionsInTableView` and `numberOfRowsInSection`, you need to return 1 for the single section you want to have and the number of tweets in the array to set the number of rows in the table. Again, Xcode may show warnings against these two methods; as before, accept the first suggestion from the Fix-it dialog. The completed methods should look like this:

```
override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
    // Return the number of sections.
    return 1
}

override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    // Return the number of rows in the section.

    if let tweetCount = self.tweets?.count {
        return tweetCount
    }
    else
    {
        return 0
    }
}
```

11. It's time to pair the data you've received and stored in the array with the custom `TweetCell` table cell with the `cellForRowAtIndexPath` method. Delete the comments surrounding the method so you're left with just the stub (and an Xcode warning).
12. Remove the exclamation marks from the method declaration, and change the highlighted values shown next:

```
override func tableView(tableView: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {

    let cell = tableView.dequeueReusableCellWithIdentifier("TweetCell",
        forIndexPath: indexPath) as TweetCell

    // Configure the cell...

    return cell
}
```

13. You need to create two `NSDictionary` objects to store different parts of the Twitter feed data. One stores the main message data, the other stores the portion that relates directly to the user who created the tweet. An `NSDictionary` is a type of array that uses a key-value pairing system to store and access data. This means instead of asking for the value at position 0, as you would with an array, you ask for the value that corresponds to “name.” Add the following highlighted code into your method:

```
let cell = tableView.dequeueReusableCellWithIdentifier("TweetCell",
    forIndexPath: indexPath) as TweetCell
```

```
let tweetData = tweets?.objectAtIndex(indexPath.row) as NSDictionary
let userData = tweetData.objectForKey("user") as NSDictionary
```

```
return cell
```

14. Let’s take data from those `NSDictionary` objects and populate the interface. Add the following highlighted code to set the values of the two labels and then return the cell object to stop the error from being reported in Xcode:

```
let tweetData = tweets?.objectAtIndex(indexPath.row) as NSDictionary
let userData = tweetData.objectForKey("user") as NSDictionary
```

```
cell(tweetContent).text? = tweetData.objectForKey("text") as String
cell(tweetUserName).text? = userData.objectForKey("name") as String
```

```
return cell
```

15. Because you returned the cell object, you’re now error free and can run the application. You haven’t set the image yet, but you should be able to select your Twitter account and see the user and content values in each cell, as shown in Figure 8-30. If you get an exception when you go to the Feed view controller, check that you specified the correct identifier on the cell in the storyboard as well as in the code.



Figure 8-30. The avatar-less Twitter feed being displayed

16. I hope you have a huge sense of satisfaction at seeing your application finally come to life as it reads live data from the Internet. There is one final block of code for this method, which focuses on retrieving, caching, and displaying Twitter users' avatars. First you try to retrieve the image from the cache; if that fails, you create an operation for the `NSOperationQueue` queue object to fetch the image's data and create an image from it before displaying it and then caching it for future use. Add the following highlighted code:

```

cell.tweetUserName.text? = userData.objectForKey("name") as String

let imageURLString = userData.objectForKey("profile_image_url") as String
let image = imageCache?.objectForKey(imageURLString) as UIImage?

if let cachedImage = image {
    cell.tweetUserAvatar.image = cachedImage
}
else
{
    cell.tweetUserAvatar.image = UIImage(named: "camera.png")

    queue?.addOperationWithBlock() {
        let imageURL = NSURL(string: imageURLString) as NSURL!
        let imageData = NSData(contentsOfURL: imageURL) as NSData?
        let image = UIImage(data: imageData!) as UIImage?

        if let downloadedImage = image {
            NSOperationQueue.mainQueue().addOperationWithBlock(){
                let cell = tableView.cellForRowAtIndexPath(indexPath) as TweetCell

                cell.tweetUserAvatar.image = downloadedImage
            }

            self.imageCache?.setObject(downloadedImage, forKey: imageURLString)
        }
    }
}

return cell

```

17. Take this opportunity to rerun the application. This time, after a brief delay, images should appear instead of the placeholder `camera.png` image, as shown in Figure 8-31.

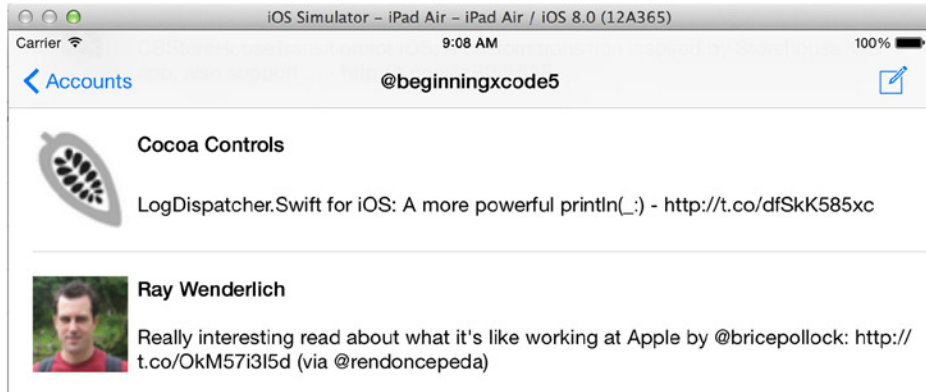


Figure 8-31. The Twitter feed, with avatar images included this time

Note There is always a delay when fetching data from the Internet. But because you're using the `NSOperationQueue` object and efficiently switched between the main and arbitrary threads, there is no slowdown in the application, which would have guaranteed you negative app store reviews. Notice how quickly you can scroll up and down the list of tweets, all because the `NSCache` stores them for later use.

Before moving on, here is the full code for the `cellForRowAtIndexPath` method:

```
override func tableView(tableView: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {

    let cell = tableView.dequeueReusableCellWithIdentifier("TweetCell",
        forIndexPath: indexPath) as TweetCell

    let tweetData = tweets?.objectAtIndex(indexPath.row) as NSDictionary
    let userData = tweetData.objectForKey("user") as NSDictionary

    cell.tweetContent.text? = tweetData.objectForKey("text") as String
    cell.tweetUserName.text? = userData.objectForKey("name") as String

    let urlString = userData.objectForKey("profile_image_url") as String
    let image = imageCache?.objectForKey(urlString) as UIImage?

    if let cachedImage = image {
        cell.tweetUserAvatar.image = cachedImage
    }
}
```



```

else
{
    cell.tweetUserAvatar.image = UIImage(named: "camera.png")

    queue?.addOperationWithBlock() {
        let imageURL = NSURL(string: imageURLString) as NSURL!
        let imageData = NSData(contentsOfURL: imageURL) as NSData?
        let image = UIImage(data: imageData!) as UIImage?

        if let downloadedImage = image {
            NSOperationQueue.mainQueue().addOperationWithBlock(){
                let cell = tableView.cellForRowAtIndexPath(indexPath) as TweetCell

                cell.tweetUserAvatar.image = downloadedImage
            }

            self.imageCache?.setObject(downloadedImage, forKey: imageURLString)
        }
    }
}

return cell
}

```

18. In this file, let's create the stubs for the two segues away from this view controller: `ComposeTweet` and `ShowTweet`. Scroll down to the bottom of the file, uncomment the `prepareForSegue` method, and remove the `!` from the method declaration if Xcode gives you a warning.
19. Handle the two possible segues by adding the following highlighted code:

```

override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject!) {
    if(segue.identifier == "ComposeTweet")
    {
        }
    else if(segue.identifier == "ShowTweet")
    {
        }
    }
}

```

That's it for the Feed view controller for now. Once the other views have been configured, you'll come back to this file and add the code to pass the baton when the segue is triggered. For now, let's move on to the third view controller: Tweet view controller.

Because the last two views in the application subclass the `UIViewController`, in the following section I only focus on adding the code. This is mainly so that you can see how the application builds up in stages and that it's rarely spit into neat chunks. But this will also give you the satisfaction of building your own working Twitter client.

Configuring the Tweet View

The Tweet view controller allows the user to see the full text of the tweet they selected from the Twitter feed, as well as any associated metadata. This is useful if the text in the table-view cell has been truncated. If you were making a complete Twitter client, you would have to add numerous bits of additional information, such as how many times the tweet has become a favorite or been retweeted. For SocialApp, however, you configure the Feed view controller to pass across an `NSDictionary` object containing the data of the selected tweet; you then pick relevant information from that object to be displayed in the view.

First, as you did with the Feed view controller, let's configure the Tweet view controller to receive the `NSDictionary`. Because you're just pulling information from an `NSDictionary` object and not interacting with Twitter or the Internet, you don't need the Accounts or Social framework in this view controller. Follow these steps:

1. Open `TweetViewController.swift` from the Project Navigator. After the `@IBOutlet`s and `@IBActions`, create an `NSDictionary` property called `selectedTweet` with this line of code:

```
var selectedTweet : NSDictionary?
```

This creates the property—or, to think of it another way, it's a runner on the track waiting to receive the `NSDictionary` baton from the runner or view controller before it. Now you need to go back to Feed view controller and pass across an `NSDictionary` of tweet data.

2. Back in `FeedViewController.swift`, scroll down to the `prepareForSegue` method. You've already created the `if` statement that checks for the `ShowTweet` segue, but in that `if` statement you need to determine the selected row's index, retrieve the relevant entry from the `tweets` array, and pass it to the Tweet view controller by setting its `selectedTweet` property. Following is the `if` statement and all the required highlighted code:

```
else if(segue.identifier == "ShowTweet")
{
    var path : NSIndexPath = self.tableView.indexPathForSelectedRow()!

    let tweetData = self.tweets!.objectAtIndex(path.row) as NSDictionary

    let targetController = segue.destinationViewController as TweetViewController
    targetController.selectedTweet = tweetData
}
```

3. Now that the information is being passed across, it's easy to access the information you want to display. Open `TweetViewController.swift` again.

4. All the processing of data is done in the `viewDidLoad` method. You fetch the user's avatar directly from the Internet, rather than from the cache. You should be familiar with the rest of the code from the previous view controller. Add the highlighted code to the `viewDidLoad` method:

```

override func viewDidLoad() {
    super.viewDidLoad()

    let userData = selectedTweet?.objectForKey("user") as NSDictionary

    tweetText.text? = selectedTweet?.objectForKey("text") as NSString
    tweetAuthorName.text? = userData.objectForKey("name") as String

    let imageURLString = userData.objectForKey("profile_image_url") as String
    let imageURL = NSURL(string: imageURLString) as NSURL!
    let imageData = NSData(contentsOfURL: imageURL!) as NSData!

    dispatch_async(dispatch_get_main_queue()) {
        self.tweetAuthorAvatar.image = UIImage(data: imageData!)
    }
}

```

5. To make the tweet close with the Cancel button, you implement the `dismissView` method, which uses the `UIViewController` method of `dismissViewControllerAnimated`. Because it's a `UIViewController` method and this is the implementation file for a class that subclasses `UIViewController`, you access your base class methods by using `self`, although it isn't always necessary with Swift. Add the highlighted line of code to your action:

```

@IBAction func dismissView(sender: AnyObject) {
    self.dismissViewControllerAnimated(true, completion: nil)
}

```

That's it! Go ahead and run your application. Select a tweet from the feed, and the tweet should be expanded in the modal dialog. For the first time, you can see the form sheet presentation style in effect, as shown in Figure 8-32. Now let's move on to the final view controller in this application: the Compose view controller.

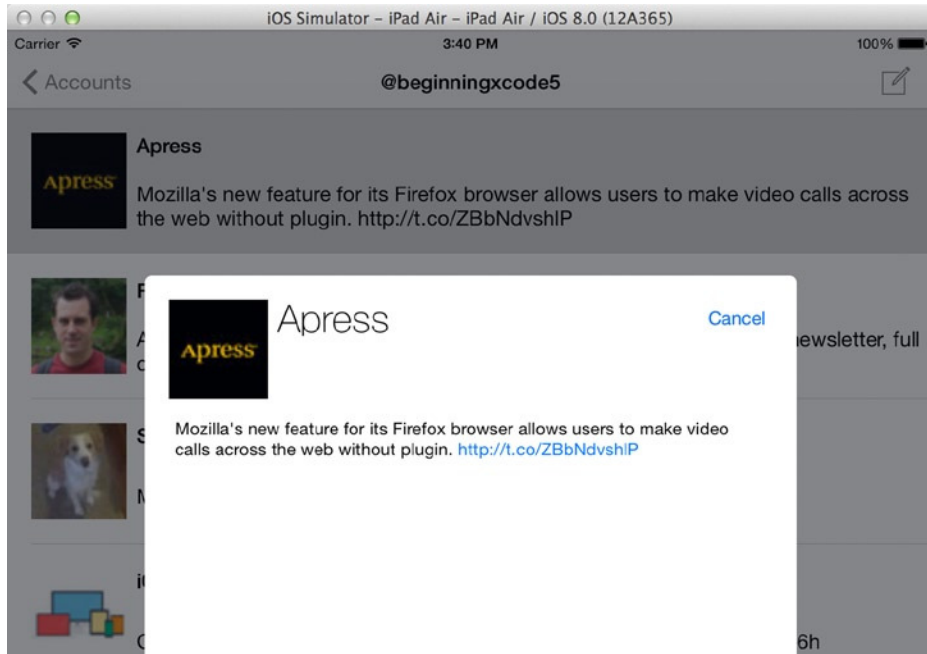


Figure 8-32. The Tweet view controller showing the data that was passed to it along with an image pulled directly from the Internet. Notice the link detection in effect

Configuring the Compose View

The final view for this application is the Compose view controller. This is where the user can compose a message and post it to Twitter. Let's enforce the 140-character limit for tweets by using a UITextView delegate method and then animate the activity indicator when it's sending the tweet data to Twitter. Here are the steps:

1. You've created the visual element and set up the outlets, so open `ComposeViewController.swift`. As previously mentioned, you're using a UITextView delegate method, so the first thing you need to do is implement the UITextViewDelegate protocol. Add the highlighted code to the class line so that it looks like this:

```
class ComposeViewController: UIViewController, UITextViewDelegate {
```

2. You need the Social and Accounts frameworks for this view controller, so add their import statements beneath the line that says `import UIKit`:

```
import UIKit
import Accounts
import Social
```

- To create a property to receive the `ACAccount` object for the selected account from the Feed view controller, after the class line, add the following highlighted code:

```
class ComposeViewController: UIViewController, UITextViewDelegate {
    var selectedAccount : ACAccount!
}
```

- Go back to `FeedViewController.swift` to pass the selected account details over to your newly created property.
- Scroll down until you see the `prepareForSegue` method. You have an empty `if` statement set up for the `ComposeTweet` segue; modify it so that it passes the `selectedAccount` object to the Compose view controller, as shown next:

```
if(segue.identifier == "ComposeTweet")
{
    let targetController = segue.destinationViewController as ComposeViewController
    targetController.selectedAccount = selectedAccount
}
```

- Switch back to `ComposeViewController.swift`.
- You need to create a custom function and a delegate method as well as two actions in this file. The good news is that none of them require a great deal of code. Scroll down until you see the `viewDidLoad` method. All you need to do here is specify the delegate property of the text view, which as I've mentioned previously is this view controller, so it's set to `self`. Although you can add this in the storyboard, let's do it here for the sake of variety. After the line `super.viewDidLoad()`, add the following code:

```
override func viewDidLoad() {
    super.viewDidLoad()
    self.tweetContent.delegate = self
}
```

- You need to create a function called `postContent` to handle the transmission of the text that is sent to the Twitter account passed to the view controller. Create the function stub just below the `viewDidLoad` method:

```
func postContent(post : String) {
}
}
```

9. As you can see, the function takes one parameter: a `String` object called `post`. The first thing you want to do when this function is called is to start animating the `postActivity` activity indicator. You do this by sending the `startAnimating` message. Due to the way the activity indicator was configured in Chapter 7, it's at this point that it appears and begins its animation:

```
func postContent(post : String){
    postActivity.startAnimating()
}
```

10. You're ready to prepare the Twitter request. Just as you did in the Feed view controller, let's create an `NSURL` object with the appropriate API URL for the type of request you want to make, which is used when instantiating the `SLRequest` object. A key difference this time is that you use a different HTTP method. You no longer issue a get request but instead issue a post request. When a post request is made with the `SLRequest` object, you supply the required parameters in an `NSDictionary` object. If you refer to the Twitter documentation for the status update API at <https://dev.twitter.com/docs/api/1.1/post/statuses/update>, you see that the only required parameter is called `status`. The `status` parameter should be the textual content of the status update, which is the contents of the `post` parameter the function is supplied with. Drop down a line in the function, and add the following code:

```
func postContent(post : String){
    postActivity.startAnimating()

    if let account = selectedAccount {
        let requestURL = NSURL(string: "https://api.twitter.com/1.1/statuses/update.
json")
        let request = SLRequest(forServiceType: SLServiceTypeTwitter,
requestMethod: SLRequestMethod.POST,
URL: requestURL,
parameters: NSDictionary(object: post, forKey: "status"))

        request.account = account
    }
}
```

11. Access the `performRequestWithHandler` method of the `SLRequest` object just as you did in the Feed view controller. This time, however, when you receive a successful response code, you want to stop animating the activity indicator

and dismiss the view controller. Drop down a line, and add the following highlighted code to complete the method:

```
request.account = account

request.performRequestWithHandler()
{
    responseData, urlResponse, error in

    if(urlResponse.statusCode == 200)
    {
        println("Status Posted")

        dispatch_async(dispatch_get_main_queue())
        {
            self.postActivity.stopAnimating()
            self.dismissViewControllerAnimated(true, completion: nil)
        }
    }
}
```

Your finished method code should look like this:

```
func postContent(post : String){
    postActivity.startAnimating()

    if let account = selectedAccount {
        let requestURL = NSURL(string: "https://api.twitter.com/1.1/statuses/update.json")
        let request = SLRequest(forServiceType: SLServiceTypeTwitter,
            requestMethod: SLRequestMethod.POST,
            URL: requestURL,
            parameters: NSDictionary(object: post, forKey: "status"))

        request.account = account

        request.performRequestWithHandler()
        {
            responseData, urlResponse, error in

            if(urlResponse.statusCode == 200)
            {
                println("Status Posted")

                dispatch_async(dispatch_get_main_queue())
                {
                    self.postActivity.stopAnimating()
                    self.dismissViewControllerAnimated(true, completion: nil)
                }
            }
        }
    }
}
```

12. It's time to address the two action methods: `dismissView` and `postToTwitter`. These are both one-liners; `dismissView` is a duplicate of the method used in the `Tweet` view controller, and `postToTwitter` simply calls the `postContent` method you just finished writing. Implement them both as follows:

```
@IBAction func dismissView(sender: AnyObject) {
    dismissViewControllerAnimated(true, completion: nil)
}
```

```
@IBAction func postToTwitter(sender: AnyObject) {
    postContent(self.tweetContent.text)
}
```

13. You need to implement a `UITextView` delegate method that restricts the text view's content to 140 characters. This is done by using the `shouldChangeTextInRange` method, which is called every time a character is typed. The method checks that the text view's content isn't greater than 140 characters and that it won't exceed 140 characters if someone pastes in some text. If the content is too large, the method returns `false`, and no more text can be typed. Add the following method just below `viewDidLoad`:

```
func textView(textView: UITextView,
    shouldChangeTextInRange range: NSRange,
    replacementText text: String) -> Bool {
    let targetlength : Int = 140
    return countElements(textView.text) <= targetlength
}
```

That was the last line of code for this application! Go ahead and run it and see how all the hard work you've put in over these two chapters has finally paid off. You should be able to successfully access your Twitter accounts, view the Twitter feed, see a tweet in detail, and even post your own.

Importantly, in this chapter, you've learned all about configuring table views and the different methods and properties of the `UITableView` class, which you no doubt will use heavily in your own applications.

Discovering the Collection View

A collection view is a fantastic class that Apple introduced fairly recently in iOS 6 (compared to most other objects, which have existed since the first version). Collection views offer developers a flexible way to display large amounts of data just like their cousin the table view, with the difference being that you can display data in columns as well as rows. Another neat feature is that collection views can scroll either vertically or horizontally, giving you that extra dimension as a developer.

Although structurally they're quite similar, one of the largest differences between the collection view and the table view is that the collection view's layout is completely separate from the view. It can be set to either a default or a custom `UICollectionViewLayout`, giving you a massive amount of flexibility over your design.

To demonstrate the implementation and configuration of a collection view, let's make some pretty drastic changes to SocialApp. First, let's turn it into a tabbed application, and then look at storing user preferences to automate account selection.

Embedding a Tab Bar Controller

The change I'm aiming for here is to have a Feed tab and a Following tab in the application, with the Feed tab obviously being the Feed view controller. Let's create the Following view controller, which is a collection view controller that shows the avatars of all the users that the selected Twitter account follows. To turn SocialApp into a tabbed application, you need to add a tab bar controller between the Accounts view and the Feed view:

1. Open Main.storyboard from the Project Navigator. Navigate around the storyboard until you're able to see the segue connection from the Accounts scene to the Feed scene. Highlight the segue and delete it, as shown in Figure 8-33.



Figure 8-33. The SocialApp storyboard with the ShowTweets segue removed

2. Add a tab bar controller, with the Feed scene as one of the tabs. You could drag in a tab bar controller and manually link it up, but instead let's allow Xcode do the hard work for you. Select the Feed view controller either by clicking it directly in the storyboard or by selecting Feed from beneath Feed Scene in the Document Outline.

- From the menu bar, select Editor ► Embed In ► Tab Bar Controller, as shown in Figure 8-34. This adds a tab bar controller to your storyboard, sets the Feed scene as the first tab, and arranges the views to suit your needs.

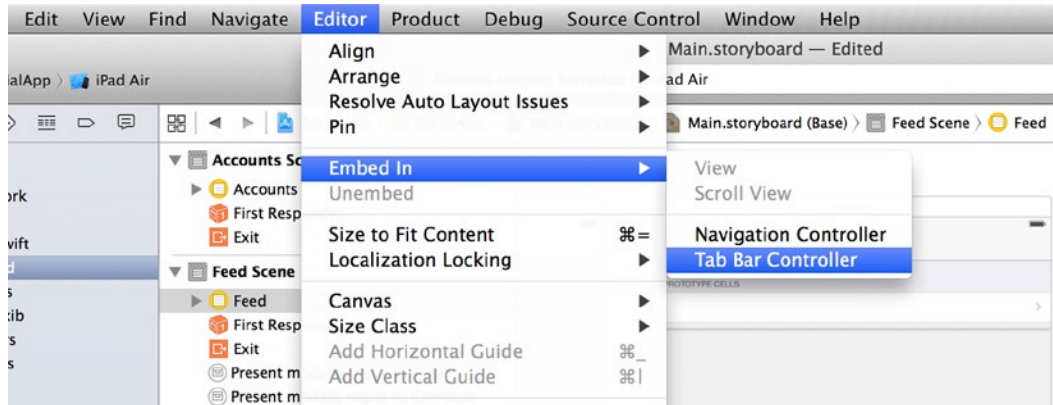


Figure 8-34. Embedding a tab bar controller into SocialApp

- You need to re-create the ShowTweets segue, but this time you're doing things differently by linking from the Accounts view controller to the tab bar controller. This is called a *manual* segue because it isn't tied to a button or table cell that can be triggered by the user. Instead, the segue is triggered programmatically, because it's not possible to segue from a table cell to a tab bar controller and then on to the view controller displayed by the controller. Zoom out, and control-drag a connection from the yellow Accounts icon to the tab bar controller, as shown in Figure 8-35.

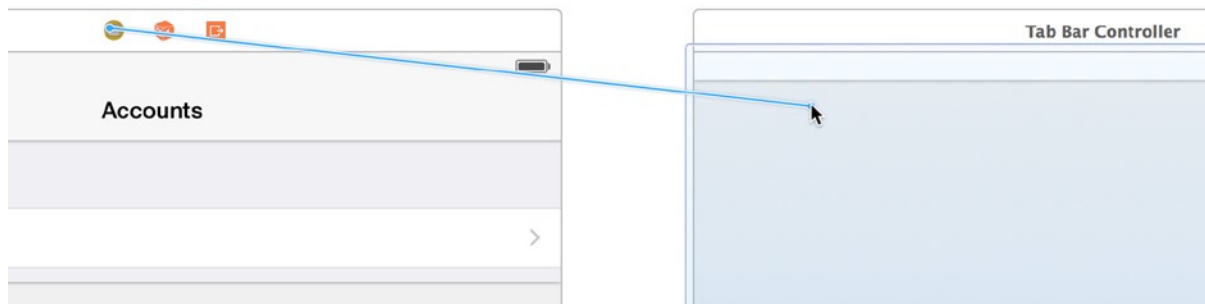


Figure 8-35. Connecting the Accounts view controller to the tab bar controller with a manual segue

- When you release the mouse button, select Show as the segue type. Select the newly created segue, and open the Attributes Inspector. Set Identifier back to ShowTweets.

You're finished with the storyboard for now, but before you add the Collection view controller, let's get the application back to a working condition. This involves executing a manual segue and storing the user's account selection so that whenever the application runs in the future, it will default to the account that the user selected and go straight to the feed.

Persisting User Preferences with NSUserDefaults

In a real-world scenario, a user picking up SocialApp would find it slightly annoying to have to select their account every time the application runs. Fortunately, iOS gives you a number of ways to persist user preferences, including iCloud and Core Data. In this instance, you're using the incredibly handy `NSUserDefaults` class.

`NSUserDefaults` allows the app to store values or certain types of objects against a textual key and can't be accessed from other applications. Whenever the application is closed and rerun, the preferences stored in `NSUserDefaults` are preserved, but the user can access and change the saved preferences to their heart's content. The `NSUserDefaults` class has methods that make it easy to both store and access a range of common types such as Booleans, floats, integers, doubles, and URLs, and it also supports the storage of the following objects:

- `NSData`
- `NSString`
- `NSNumber`
- `NSDate`
- `NSArray`
- `NSDictionary`

The object you want to store here is an `ACAccount`, so you have to convert it to an `NSData` object, but I'll get to that in a moment. Follow these steps:

1. Open `AccountsViewController.swift` from the Project Navigator. You need to create an `NSUserDefaults` instance variable to allow you to access the preferences from different methods without having to instantiate the method each time. After the line `var accountStore : ACAccountStore?`, drop down a line and add the highlighted code:

```
import UIKit
import Accounts

class AccountsViewController: UITableViewController {

    var twitterAccounts : NSArray?
    var accountStore : ACAccountStore?
    var userDefaults : NSUserDefaults?
```

2. Scroll down to the `viewDidLoad` method. The first thing you need to do when the view loads is to initialize the `userDefaults` object and then determine whether a preference called `selectedAccount` has already been saved; if so,

you execute the manual segue with the `performSegueWithIdentifier` method and go straight to the Feed view controller. After the line `accountStore = ACAccountStore()`, add the highlighted code:

```
override func viewDidLoad() {
    super.viewDidLoad()
    accountStore = ACAccountStore()

    userDefaults = UserDefaults.standardUserDefaults()

    if (userDefaults?.objectForKey("selectedAccount") != nil) {
        performSegueWithIdentifier("ShowTweets", sender: self)
    }

    var accountType : ACAccountType =
        accountStore!.accountTypeWithIdentifier(ACAccountTypeIdentifierTwitter)
```

- That's it for the `viewDidLoad` method. It's time to address what happens when you tap on a cell. In the past, the segue from the cell was triggered, and the `prepareForSegue` method then passed the selected account across to the feed. This time, however, the application is going to save the selection before moving away. To do this, you use another key `UITableView` method called `didSelectRowAtIndexPath`, which is triggered every time a table cell is selected. Before you implement this method, *delete the entire* `prepareForSegue` *method*; there is no longer a segue associated with the cell, and the method won't be needed from here on out.
- To create the `didSelectRowAtIndexPath` method stub, after the `cellForRowAtIndexPath` method, type the following highlighted code:

```
let account = self.twitterAccounts!.objectAtIndex(indexPath.row) as ACAccount
cell.textLabel?.text = account.accountDescription

return cell
}
```

```
override func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath:
NSIndexPath) {
```

```
}
```

- In this method, you use the `indexPath` object to allocate an `ACAccount` object from `twitterAccounts` array based on the selected cell's index. Add the following highlighted code to the `didSelectRowAtIndexPath` method:

```
override func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath:
NSIndexPath) {
    let account = self.twitterAccounts!.objectAtIndex(indexPath.row) as ACAccount
```

- As mentioned previously, you need to convert the account object into something that can be stored in the `NSUserDefaults`; in this case, it will be converted to an `NSData` object using the `NSKeyedArchiver` class. After the previous line, add the following highlighted code:

```
override func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath:
NSIndexPath) {
    let account = self.twitterAccounts!.objectAtIndex(indexPath.row) as Account
    let accountData = NSKeyedArchiver.archivedDataWithRootObject(account) as NSData
```

- The selected account is in a compatible format, so it can be saved to the `NSUserDefaults` instance, `userDefaults`. The process for saving a preference comes in two parts: first, use the `setObject: forKey:` method, which associates the `accountData` object with a key; then, call the `synchronize` method, which saves the preference to the system. Add the following code to the method:

```
let accountData = NSKeyedArchiver.archivedDataWithRootObject(account) as NSData
userDefaults?.setObject(accountData, forKey: "selectedAccount")
userDefaults?.synchronize()
```

- Now that you've saved the user's selection, you can manually trigger the segue. Just as you did in the `viewDidLoad` method, you need to call the `performSegueWithIdentifier` method. Add this code to complete the method:

```
override func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath:
NSIndexPath) {
    let account = self.twitterAccounts!.objectAtIndex(indexPath.row) as Account
    let accountData = NSKeyedArchiver.archivedDataWithRootObject(account) as NSData
    userDefaults?.setObject(accountData, forKey: "selectedAccount")
    userDefaults?.synchronize()

    performSegueWithIdentifier("ShowTweets", sender: self)
}
```

That's it for the Accounts view controller. You've removed the previous mechanisms for selecting an account, considered the overall user experience, and replaced those mechanisms with something that will be much more user friendly.

Now that you've saved the user's selection, you need to implement the retrieval of that selection when the application moves to the Feed view controller:

- Open `FeedViewController.swift`, and scroll down to the `viewDidLoad` method.

- When the view loads, you create an instance of `NSUserDefaults`; there is no point in creating an instance variable, because this is the only time you need to access it in this file. You then retrieve the selected account from the preferences using the `objectForKey` method, which retrieves the object associated with the key that is supplied: in this case, `selectedAccount`. Finally, you reverse the conversion process with the `NSKeyedUnarchiver` class, which allows the conversion of the `NSData` object back into an `ACAccount` object. After the line `super.viewDidLoad()`, add the following highlighted code:

```
override func viewDidLoad() {
    super.viewDidLoad()

    let userDefaults = NSUserDefaults.standardUserDefaults()
    let accountData = userDefaults.objectForKey("selectedAccount") as NSData
    selectedAccount = NSKeyedUnarchiver.unarchiveObjectWithData(accountData) as ACAccount

    self.navigationItem.title = selectedAccount.accountDescription

    queue = NSOperationQueue()
    queue!.maxConcurrentOperationCount = 4

    retrieveTweets()
}
```

- Run the application in the Simulator. Select an account, and you segue across to the newly tabbed Feed view. Quit the Simulator, and then rerun the application. If everything has been done correctly, you should start facing the Feed view instead of the Accounts view! As a user, this is a much more favorable situation to be in. There is, however, one small issue you need to address: after embedding a tab bar controller, the table-view positioning changed, and now the first row renders underneath the navigation bar and the title has vanished, as shown in Figure 8-36. This is far from ideal, and unfortunately Xcode doesn't give you an easy way to fix this; it has to be done in code.

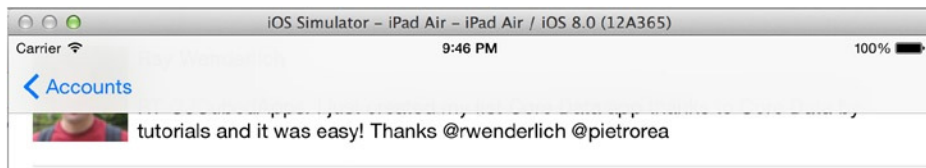


Figure 8-36. The first table row obscured by the navigation bar

4. The title no longer applies itself because when you embedded the tab bar controller, you effectively inserted another level between the view and the navigation bar controller. Ensure that you still have `FeedViewController.swift` open. Then, in the `viewDidLoad` method, change the line `self.navigationItem.title = selectedAccount.accountDescription` to the following:

```
self.tabBarController?.navigationItem.title = selectedAccount.accountDescription
```

5. Drop down another line. The issue of the table row being obscured by the navigation bar is an iOS 7 & 8–specific issue with an iOS 7 & 8–specific fix, but this doesn’t mean your application can’t be backward compatible. Let’s use a handy method called `respondsToSelector` to evaluate whether the method you need to call exists before calling it. Add the following highlighted code:

```
self.tabBarController?.navigationItem.title = selectedAccount.accountDescription  
self.tabBarController?.edgesForExtendedLayout = UIRectEdge.None
```

6. Rerun the application, and it should function perfectly! You’ve successfully implemented a user preferences system that will make life much easier for your users. You’re now ready to start creating the Collection view controller.

Adding a Collection View Controller

You’ve successfully pulled apart and reassembled your application. It’s time to turn your attention back to the storyboard and, in particular, adding a Collection view controller into the application:

1. Open `Main.storyboard` from the Project Navigator, and position the scenes as shown in Figure 8-37.

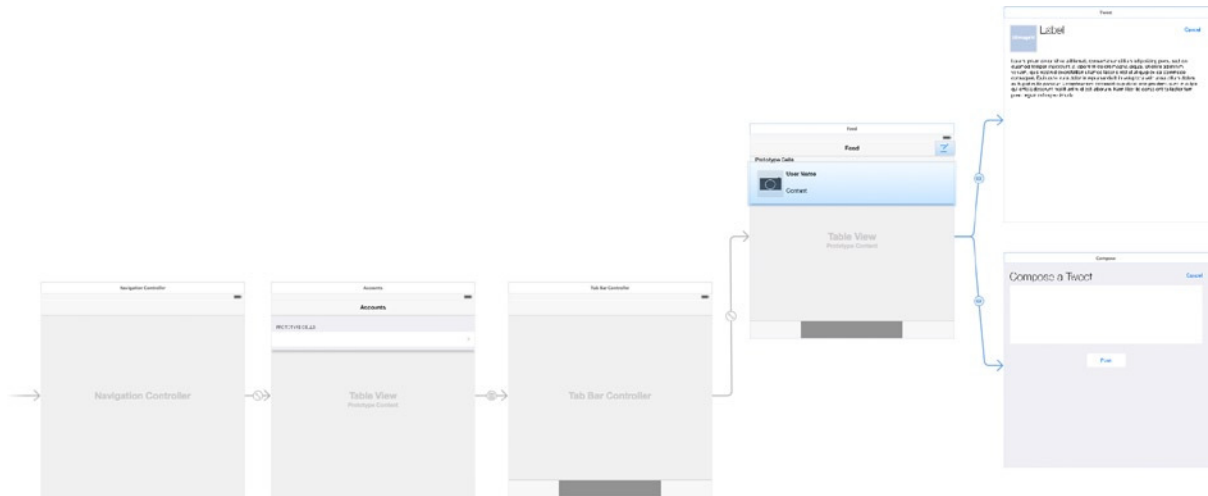


Figure 8-37. Arranging the storyboard in anticipation of the Collection view controller

2. Drag in a Collection view controller from the Object Library, and position it below the Feed scene, as shown in Figure 8-38.

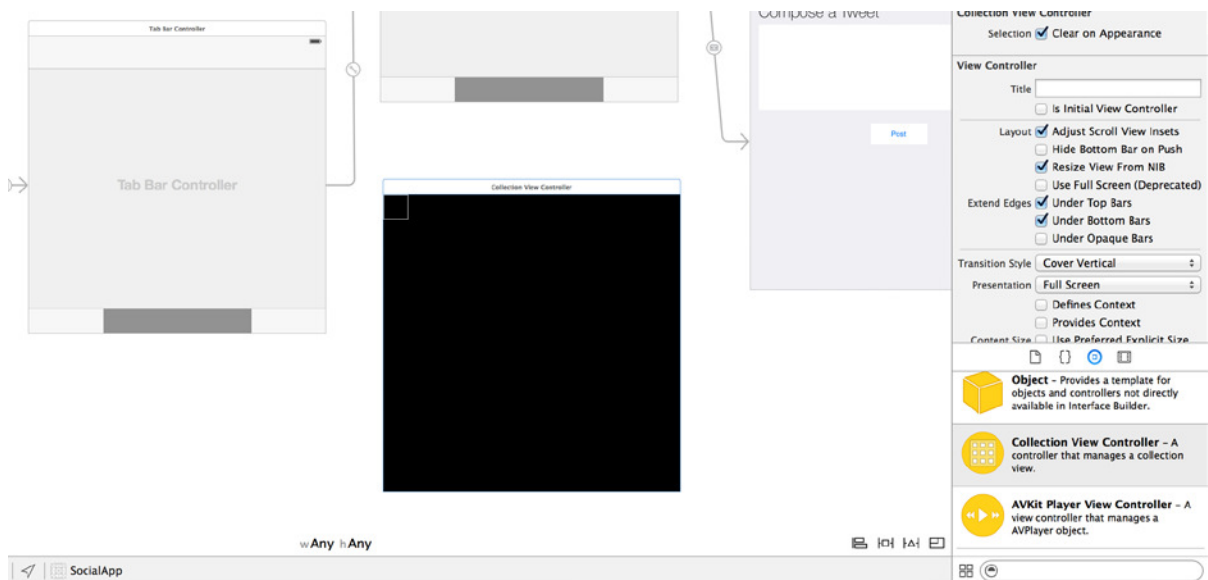


Figure 8-38. Dragging in a Collection view controller from the Object Library

3. Create a relationship between the tab bar controller and the Collection view controller. To do this, select the tab bar controller and then control-drag a connection to the Collection view controller, as shown in Figure 8-39.

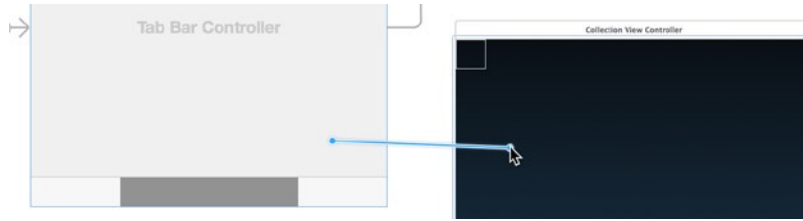


Figure 8-39. Control-dragging a connection from the tab bar controller to the Collection view controller

4. When you release the mouse, select View Controllers under Relationship Segues. You now have two tabs in the application. The tab bar controller may look like a solid grey line, but if you were to run the app, you would have two tabs called Item—which isn't great.
5. To set the Feed tab bar button title, position your storyboard so that you can select the tab bar button on the Feed scene. Open the Attributes Inspector, as shown in Figure 8-40.

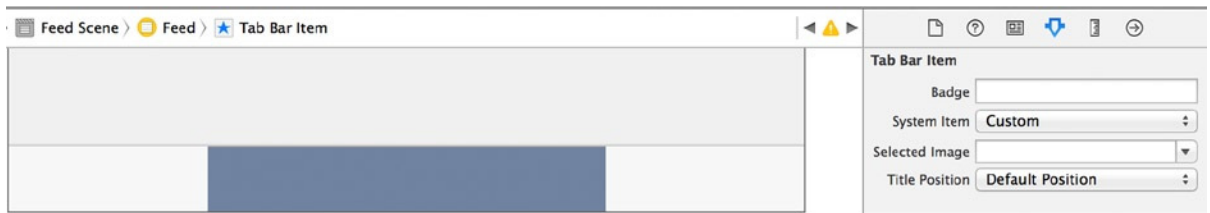


Figure 8-40. Selecting the Feed tab bar button

6. Change the System Item drop-down list from Custom to Most Recent. A neat clock icon appears. Repeat this process for the Collection view controller you added to the storyboard, changing System Item from Custom to Contacts. Your tab bar controller should now look like the one shown in Figure 8-41.

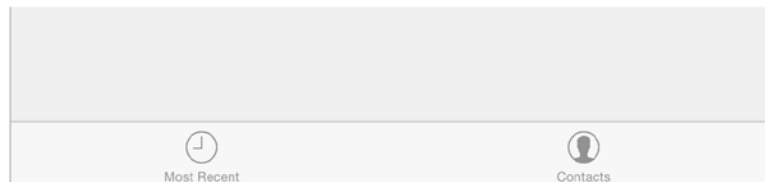


Figure 8-41. The tab bar controller featuring two correctly named tab bar items

7. As you did with the other view controllers, you need to create a customer view controller class file. In this instance, you're subclassing `UICollectionViewController`. Right-click the View Controllers group in the Project Navigator, and select New File.

8. When prompted, select the Cocoa Touch Class option, and click Next. Set the Subclass Of value to `UICollectionViewController` and the Class value to `FollowingViewController`, and then click Next. Accept the default folder Xcode suggests to save the file, and click Create.

Now that you've successfully created the last custom view controller for this application, you're ready to configure the visual aspects of the collection view before fetching the user details followed by the selected account from Twitter.

Note When adding the new view controller, a few errors may appear in Xcode. Don't panic: you'll fix those very shortly. They're not your fault.

Configuring a Collection View

I've already mentioned that the `UICollectionView` class is very similar to the `UITableView` class in terms of methods and the fact that they both use cells to present large amounts of data to the user. They also have sections with independent headers and footers. Yet despite these similarities, the collection-view configuration in Xcode is drastically different from that of the table view.

To begin, open `Main.storyboard` from the Project Navigator, and move the storyboard until you can see the collection view, as shown in Figure 8-42. Structurally, what you're looking at isn't really any different from what you started with in the table view, as shown in Figure 8-21. The white-bordered box in the top-left corner of the view is the prototype cell.

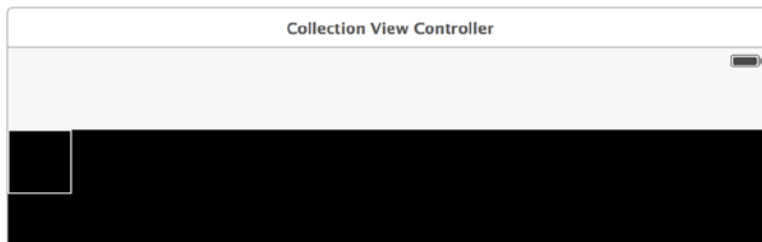


Figure 8-42. The top of the Collection view controller

Select the collection view so that you can look at the key attributes in the Attributes Inspector. To do this, click the main area of the collection view or select Collection View from the Document Outline, as shown in Figure 8-43.

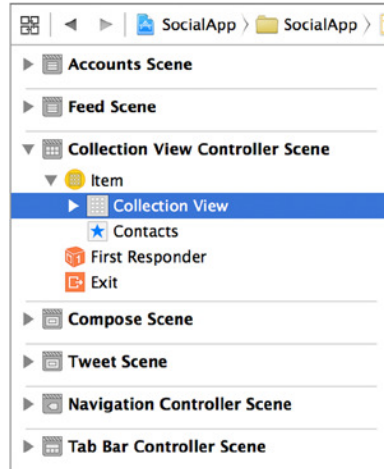


Figure 8-43. Selecting the collection view from the Document Outline

Open the Attributes Inspector. Let's take a closer look at the key options available to you:

- *Items:* Unlike the table view, collection views don't have a static mode. The Items attribute increases and decreases the number of prototype cells. No matter how many items you have to display, if they have a single type of appearance, then you only need one cell, which you reuse.
- *Layout:* In a collection view, the layout is a separate entity from the view. The default layout is Flow, which provides a grid of items continuing uninterrupted in a fixed direction. Changing the attribute to Custom exposes a class selector whereby you can specify a custom UICollectionViewLayout.
- *Scroll Direction:* As you might expect, this attribute controls the direction in which the cells are positioned for scrolling.
- *Accessories:* The Section Header and Section Footer options allow you to add a prototype header and footer to the section. Unlike with table views, you can't manually specify any text in either container; it must be set programmatically.

Unlike in other views, much in collection views depends on the settings of the Size Inspector. When you open the Size Inspector, you see many configurable values; Figure 8-44 shows the different sizes and where they take effect. In this example, the number of items is set to 8 to help you visualize how the cells react to one another.

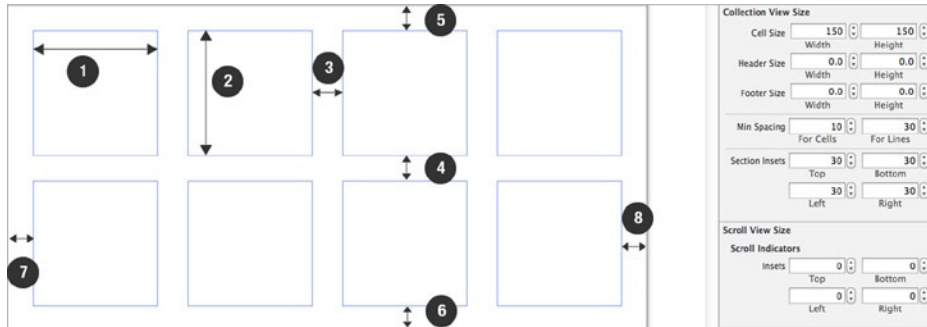


Figure 8-44. The different sizes you can alter in a collection view

- **Cell Width and Height:** Points 1 and 2 represent the width and height of the cell, respectively. The cell doesn't have to be square; the two values can change independently of each other.
- **Minimum Spacing:** The first value (highlighted by point 3), For Cells, sets a minimum value for the horizontal spacing between cells. This is useful because by default, the cells are spaced nicely, and the horizontal gap is far greater than 10 points. However, you know it won't slip below 10 points if the size of the view changes. The For Lines value shown by point 4 sets a minimum width between the rows of cells.
- **Section Insets:** These four values control the spacing around the outside of the section of cells, so the cells function as a collective entity. When you increase the value of any of the sizes illustrated by points 5–8, you move the cells further from that side of the view. By default, the Section Insets values are set to 0, which can leave content feeling squashed; set a nice inset value to bring the cells in from the edge, which is more visually appealing.

Follow these steps:

1. For the Followers collection view, set the Cell Size Width and Height values to 75, set Minimum Spacing to 10 For Cells and 30 For Lines, and set all Section Insets to 30.
2. Before you start adding code to the Followers view controller, you need to specify the class the view controller uses. Select the Collection view controller by either clicking the top bar of the view controller or selecting Item under Collection View Controller Scene in the Document Outline.
3. Open the Identity Inspector, and set the class to FollowingViewController, as shown in Figure 8-45.

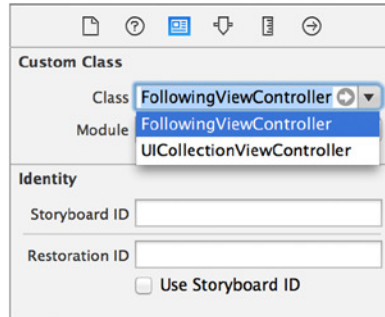


Figure 8-45. Setting Class to *FollowingViewController*

4. Select the single cell in the *Following* view controller. Open the Attributes Inspector, and set the Identifier value to *Cell*.

Displaying Items in a Collection View

You've configured the collection view in Xcode, but to finish the view you need to write the code that retrieves the list of users the app follows. I won't focus too much on the code for retrieving the list of followed users, but rather on the key methods of the *UICollectionViewController* class:

1. Open *FollowingViewController.swift* from the Project Navigator. The first thing you need to do is import the *Social* and *Accounts* frameworks. After the line `import UIKit`, add the following highlighted import statements:

```
import UIKit
import Accounts
import Social
```

2. Unless Apple has fixed this in Xcode 6.1, you may see a number of errors in the code; let's fix those now. Almost all the errors can be resolved by removing the `!` suffix after the class names in method declarations in the file. Additionally, you may need to add a `?` to an object, as highlighted here in the `viewDidLoad` method:

```
self.collectionView?.registerClass(UICollectionViewCell.self,
    forCellWithReuseIdentifier: reuseIdentifier)
```

3. Declare a number of instance variables, just as you did with the Feed view controller. You need an NSMutableArray instance called following to store the details of each user the selected account follows, an NSCache object called imageCache, and an NSOperationQueue object imaginatively called queue. The start of your file with these items added should look like this:

```
import UIKit
import Accounts
import Social

let reuseIdentifier = "Cell"

class FollowingViewController: UICollectionViewController {

    var following : NSMutableArray?
    var imageCache : NSCache?
    var queue : NSOperationQueue?
```

4. Move down to the viewDidLoad method; in this method, just as you did with the Feed view controller, you want to initialize the queue object, set the navigation bar title to Following, and then call the retrieveUsers function, which you'll add shortly. Add the highlighted code to your viewDidLoad method:

```
override func viewDidLoad() {
    super.viewDidLoad()

    // Register cell classes
    self.collectionView?.registerClass(UICollectionViewCell.self,
        forCellWithReuseIdentifier: reuseIdentifier)

    queue = NSOperationQueue()
    queue?.maxConcurrentOperationCount = 4

    self.tabBarController?.navigationItem.title = "Following"

    retrieveUsers()
}
```

5. Look at the retrieveUsers function. Create a stub for the function below the viewDidLoad method, as shown next:

```
    retrieveUsers()
}

func retrieveUsers() {

}
```

6. Much of this code is the same as that used in the Feed view controller, so I don't present the code in any detail. Clear the following array, and then retrieve `selectAccount` from the stored user preferences, as shown next:

```
func retrieveUsers() {
    following?.removeAllObjects()

    let userDefaults = NSUserDefaults.standardUserDefaults()
    let accountData = userDefaults.objectForKey("selectedAccount") as NSData
    let selectedAccount = NSKeyedUnarchiver.unarchiveObjectWithData(accountData) as
    ACAccount
}
```

7. Declare an `SLRequest` object, and instantiate it using the URL specified by the Twitter API for retrieving a list of "friends," as Twitter refers to the API that returns "up to 200 users," followed by the supplied account:

```
let accountData = userDefaults.objectForKey("selectedAccount") as NSData
let selectedAccount = NSKeyedUnarchiver.unarchiveObjectWithData(accountData) as ACAccount

let requestURL = NSURL(string: "https://api.twitter.com/1.1/friends/list.json?count=200")

let request = SLRequest(forServiceType: SLServiceTypeTwitter,
    requestMethod: SLRequestMethod.GET,
    URL: requestURL,
    parameters: nil)

request.account = selectedAccount
```

Note For more information on configuring the Twitter Friends/List API, visit <https://dev.twitter.com/docs/api/1.1/get/friends/list>.

8. You need to call the `performRequestWithHandler` method of the `SLRequest`. Just as before, you check for a valid status code and parse the JSON response before picking the "users" array from the parsed code and assigning it to the `following` array. Calling the `UICollectionView` method `reloadData` causes three methods to be called. Add the following code to complete this method:

```
request.account = selectedAccount

request.performRequestWithHandler()
{
    responseData, urlResponse, error in
```

```

if(urlResponse.statusCode == 200)
{
    var jsonParseError : NSError?
        let followingData = NSJSONSerialization.JSONObjectWithData(responseData,
            options: NSJSONReadingOptions.MutableContainers,
            error: &jsonParseError) as NSDictionary

        self.following = followingData.objectForKey("users") as? NSMutableArray
    }

    dispatch_async(dispatch_get_main_queue()) {
        self.collectionView.reloadData()
    }
}

```

9. On to the delegate methods. Just as with the table view, you have to specify the number of sections via the `numberOfSectionsInCollectionView` method. Set it to return 1 as shown next:

```

override func numberOfSectionsInCollectionView(collectionView: UICollectionView) -> Int {
    return 1
}

```

10. You need to specify how many of the potential 200 cells to render should appear via the `numberOfItemsInSection` method, which returns the number of rows in the following array:

```

override func collectionView(collectionView: UICollectionView,
    numberOfItemsInSection section: Int) -> Int {
    if let followCount = following?.count {
        return followCount
    }
    else
    {
        return 0
    }
}

```

11. Scroll down, and locate the `cellForItemAtIndexPath` method. It's used to initialize the cell and set its content, just as its `UITableView` equivalent does.
12. Using the highlighted code to pull the relevant data for the current index from the following array and store it in an `NSDictionary` object before extracting the URL for the user's profile image, as you did in the Feed view controller:

```

override func collectionView(collectionView: UICollectionView,
    cellForItemAtIndexPath indexPath: NSIndexPath) -> UICollectionViewCell {
    let cell = collectionView.dequeueReusableCellWithReuseIdentifier(reuseIdentifier,
        forIndexPath: indexPath) as UICollectionViewCell

```



```

    let userData = following?.objectAtIndex(indexPath.row) as NSDictionary
    let imageURLString = userData objectForKey("profile_image_url") as String

    return cell
}

```

13. You need to set up the image that is programmatically added to the cell using the `addSubview` method because no `UIImageView` exists in the cell, and then return the cell object. Complete the method with this highlighted code:

```

override func collectionView(collectionView: UICollectionView,
    cellForItemAtIndexPath indexPath: NSIndexPath) -> UICollectionViewCell {
    let cell = collectionView.dequeueReusableCellWithReuseIdentifier(reuseIdentifier,
        forIndexPath: indexPath) as UICollectionViewCell

    let userData = following?.objectAtIndex(indexPath.row) as NSDictionary
    let imageURLString = userData objectForKey("profile_image_url") as String

    if let image = imageCache?.objectForKey(imageURLString) as? UIImage {
        let imageView = UIImageView(image: image) as UIImageView
        imageView.bounds = cell.frame
        cell.addSubview(imageView)
    }
    else
    {
        queue?.addOperationWithBlock() {
            let imageURL = NSURL(string: imageURLString) as NSURL?
            let imageData = NSData(contentsOfURL: imageURL!) as NSData?
            let image = UIImage(data: imageData!) as UIImage?

            if let downloadedImage = image {
                NSOperationQueue.mainQueue().addOperationWithBlock(){
                    let imageView = UIImageView(image: image)
                    imageView.bounds = cell.frame

                    if let cell = self.collectionView.cellForItemAtIndexPath(indexPath)
                    as UICollectionViewCell! {
                        cell.addSubview(imageView)
                    }
                }
            }

            self.imageCache?.setObject(image!, forKey: imageURLString)
        }
    }
    return cell
}

```

That completes the collection view and the chapter! Go ahead and run your application. As long as you're following some other Twitter users, your collection view should populate with user avatars as shown in Figure 8-46. Note that I've used the Xcode logo instead of some of the faces, for privacy reasons.

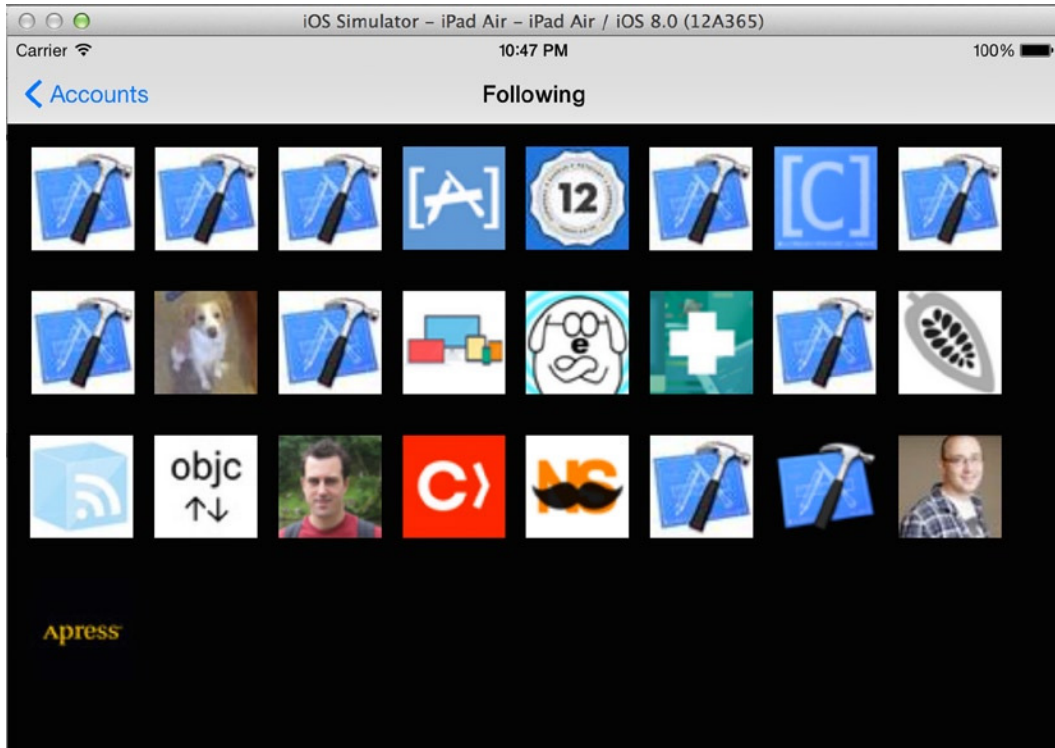


Figure 8-46. The finished collection view showing the avatars of users you follow

Summary

There's no doubt that you covered a lot in this chapter. Most important, you've taken SocialApp from the shell it was at the end of Chapter 7 and created a working Twitter client. This was no mean feat, and hopefully you're feeling really pleased with yourself. Take a break and reflect on all the things you've learned in this chapter:

- The difference between static and prototype table cells
- When to use a grouped or a plain style table view
- How to create a custom table cell
- How to fetch data from the Internet
- How to parse JSON data
- How to embed a tab bar controller into your application
- How to persist user preferences even when the application has closed

The next chapter looks at other ways you can use frameworks in Xcode, as well as libraries and how Xcode lets you create different applications with the same code using targets.

Frameworks, Libraries, and Targets

In Chapter 8, you learned about how to configure and implement table and collection views as you completed a Twitter client, `SocialApp`. You also used two frameworks, `Social` and `Accounts`, to access the Twitter API; frameworks are a topic I explain in more detail in this chapter. You also parsed the JSON-formatted data and looked at `NSDictionaries` in action. The chapter covered many important aspects of Xcode application development, and I hope you found the project useful and enjoyable—maybe even a bit exciting!

Chapter 8 had a lot of code to go through as you subclassed cells and view controllers, but in this chapter you see how a little code can go a long way. This chapter explains how Xcode uses frameworks to add extra functionality; libraries to encapsulate lots of classes, methods, and resources neatly; and targets to create different versions of your application in a single project.

The project for this chapter is an application based on the Map Kit framework that displays pushpins on a map. You create two versions of the application by using targets: both use the same base code, but the output changes based on the version.

Map Kit is a framework provided by Apple that renders an interactive map. It has numerous classes for modifying and complementing the map. Whether you're adding pushpins or custom markers, plotting routes, or outlining areas, the Map Kit framework has everything you need to create a rich, map-based application.

Understanding Frameworks

Chapter 7 introduced inheritance as being one of the core principles of object-oriented programming. Frameworks embody another of these core principles: encapsulation. Encapsulation is usually defined as one of two things:

- A mechanism by which information (code) is hidden
- A construct for bundling data together

A framework groups classes, resources, interface files, and more together in a hierarchical package. Although you can view some of the resources in the framework, the implementation of the classes—the key code—is hidden. Figure 9-1 shows the Map Kit framework you’re working with in this chapter.

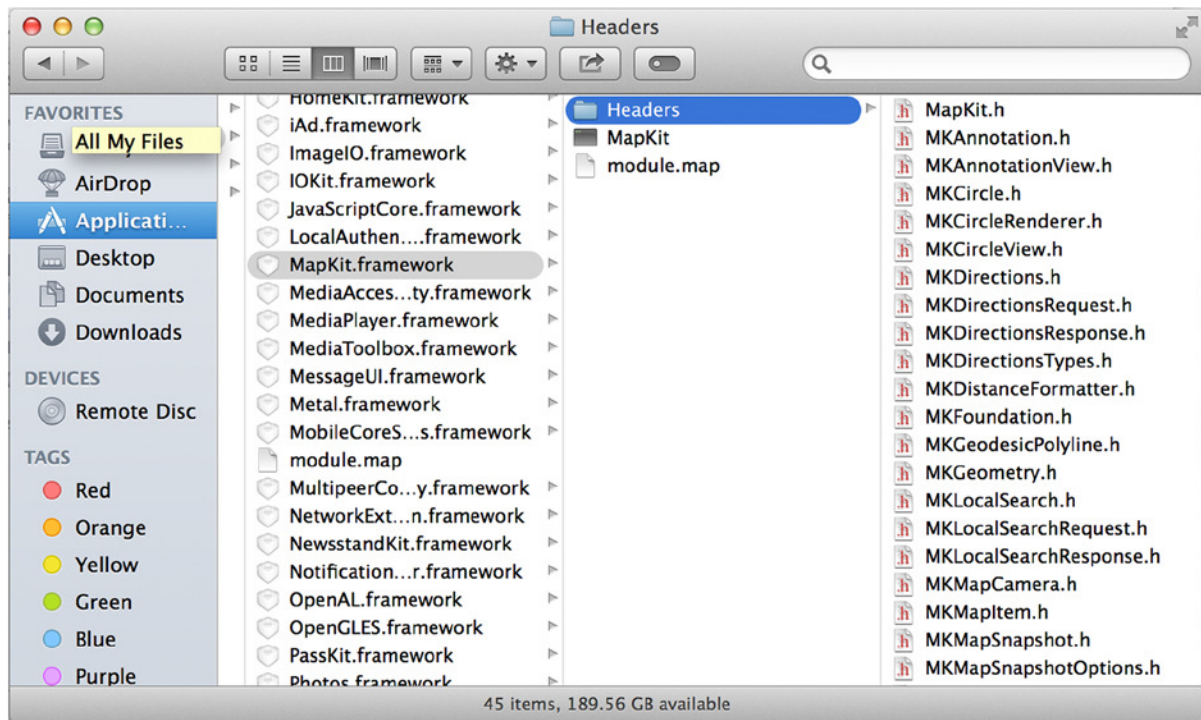


Figure 9-1. A look at the large number of class headers held in MapKit.framework

As you learn later in this chapter, frameworks don’t differ greatly from libraries in terms of definition: the key difference is in how you define their purpose. Frameworks encapsulate a wide range of functions; Map Kit has a mass of classes and different resources. Libraries, on the other hand, are intended to fulfill smaller, more specific tasks, such as caching images or grouping code you may use often for a specific type of project, such as with Map Kit applications. Frameworks provide a way to unlock the features and functions of the operating system and the hardware.

Creating the Project

The project for this chapter is called *MapPin*; it shows a number of pushpins on a map with a textual annotation. Many applications use the Apple-provided Map Kit to display information in really interesting ways, and if this is something you want to add to your own applications, hopefully you'll be encouraged by how easy it is to add a map view to your application and to make it display information with just a few lines of code:

1. Open Xcode, and create a new project by clicking Create A New Xcode Project from the Welcome screen or going to File ► New ► Project (⌘+Shift+N). Select the Single View Application template, and click Next.
2. Name your project MapPin, and ensure that Language is set to Swift and Devices is set to iPhone, not iPad or Universal. Configure the other values to your own preferences; mine are shown in Figure 9-2. Click Next.

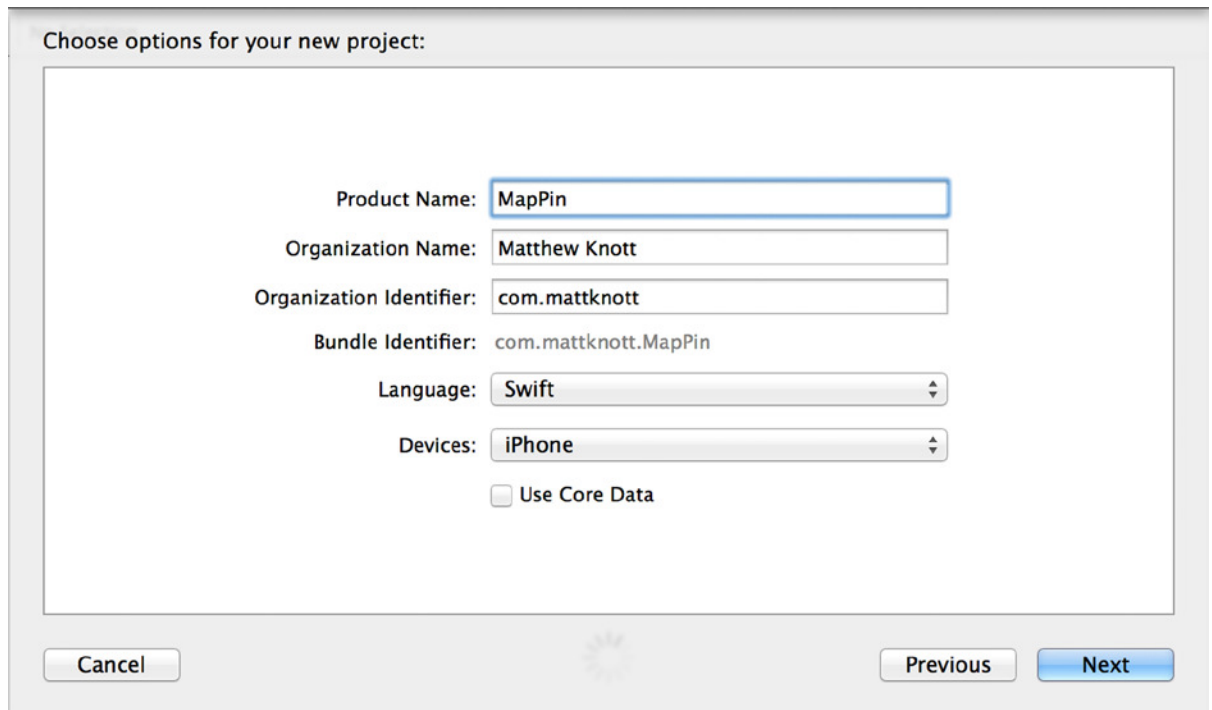


Figure 9-2. The initial settings for the MapPin project

3. You don't want to create a Git repository, so leave that option unchecked. You also don't want to add this to another project. With those options set, click Create.
4. You now have a blank project—a fresh canvas to which you need to add a map view. Open `Main.storyboard`, and locate a map view object in the Object Library. Drag it onto the view controller in the design area, as shown in Figure 9-3.

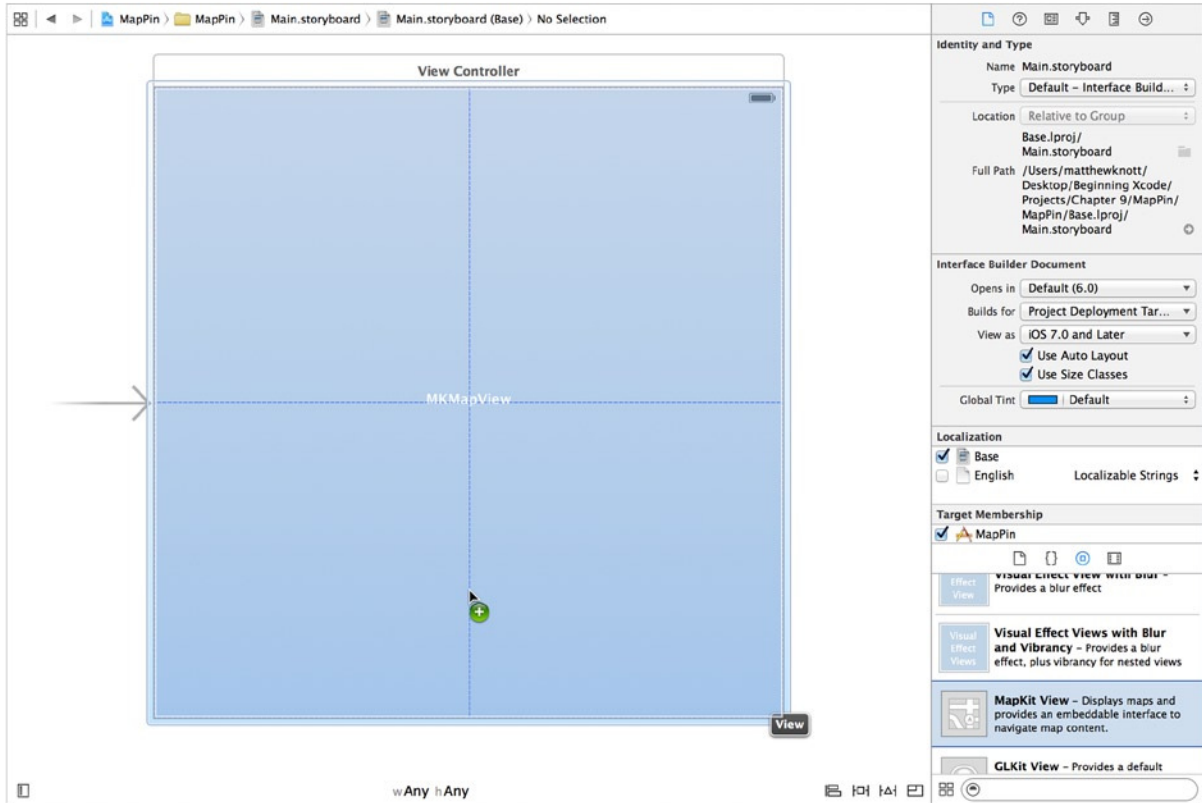


Figure 9-3. Dragging a map view object onto the view controller

The map view represents an `MKMapView` object and is the only object needed to display maps in your application. Go ahead and run the application to see what happens after you add the object to the view controller. The application builds successfully and launches in the simulator; then it drops back to Xcode with an exception, as shown in Figure 9-4, because the application has no knowledge of the `MKMapView` class. The prerequisite for adding a map view to your applications is that you also need to add the Map Kit framework.

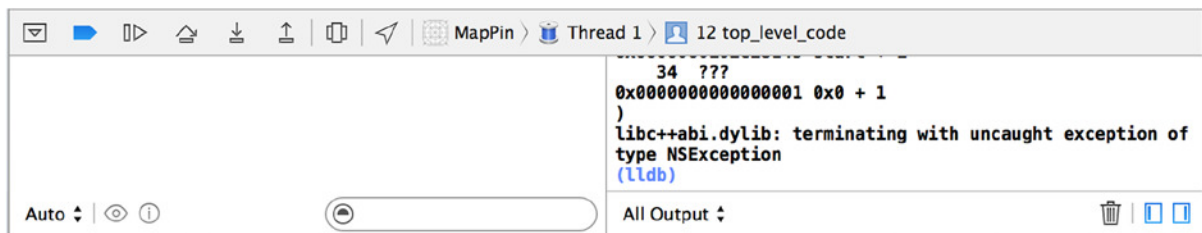


Figure 9-4. The exception shown in the debug area when you add a map view without the Map Kit framework

Adding a Framework

Back as early as Chapter 4, in the Showcase application, you first encountered frameworks. At that point I gave a very brief explanation of how to import one and how the modules concept works. Let's recap.

With iOS 7 and Xcode 5, before Swift came along, Apple gave developers a new alternative called *modules* for manually adding frameworks to a project. The concept behind modules is that instead of going through Xcode to find and add a selected framework to physically integrate into a project and *then* go on to reference it in code with an `#import` statement, you can simply reference it with a single line of code using the `@import` statement. Xcode automatically identifies the framework and links the frameworks headers at build time behind the scenes.

As you can imagine, not having to go through the time-consuming process of locating and adding frameworks was a big hit with developers, and with Swift, Apple has kept this functionality and made it the default approach. You rarely need to manually import a framework in a Swift application.

The only downside is that before you add a framework, you need to know its name. To help you, Table 9-1 lists some of the more important iOS 8 frameworks that you haven't come across yet and explains some of the functionality they unlock.

Table 9-1. Key Frameworks for iOS 8

Framework	Purpose
Core Data	Interface to interact with efficient relational databases. Great for managing large amounts of data.
Local Authentication	New for iOS 8, provides access to the Touch ID API for devices with a fingerprint reader.
HealthKit	Also new in iOS 8, provides numerous APIs to access the M7 motion chip's functions, such as the pedometer.
Web Kit	Allows you to use much more Safari-level functionality for your web view and is highly customizable.
Notification Center	Lets you add your own custom widgets to the notification center in iOS 8. This is a great way to extend your app's functionality.
Photos and Photos UI	Two new frameworks that allow you to manipulate photos and add custom functionality to the iOS 8 Photos app.
Message UI	Allows you to create emails and text messages programmatically.

These are just a few examples of the dozens and dozens of frameworks available to you with iOS 8. I mentioned that you rarely have to add a framework to a project. However, this is one of those occasions when you do have to perform this task, because you're using the Map Kit framework:

1. Adding a framework to the project is a breeze. Start by selecting the MapPin project in the Project Navigator, as shown in Figure 9-5.

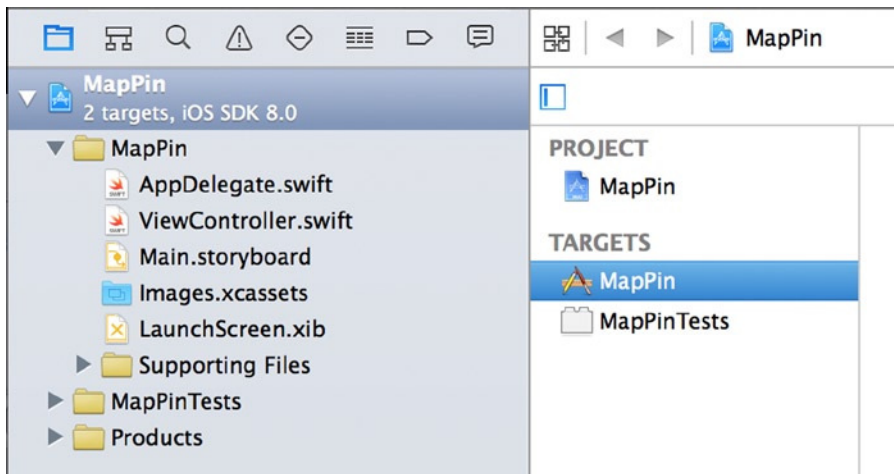


Figure 9-5. Selecting the MapPin project from the Project Navigator

2. On the screen that appears, select MapPin from the list of targets, and then choose the General tab. Scroll down to the Linked Frameworks And Libraries section, as shown in Figure 9-6.

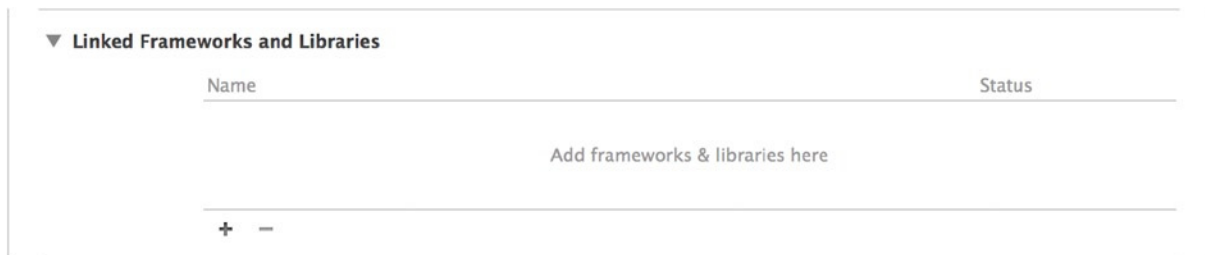


Figure 9-6. The Linked Frameworks And Libraries area of your project's settings

3. You can display a list of available frameworks, as shown in Figure 9-7, by clicking the + symbol at the bottom of the section, below Add Frameworks & Libraries Here.

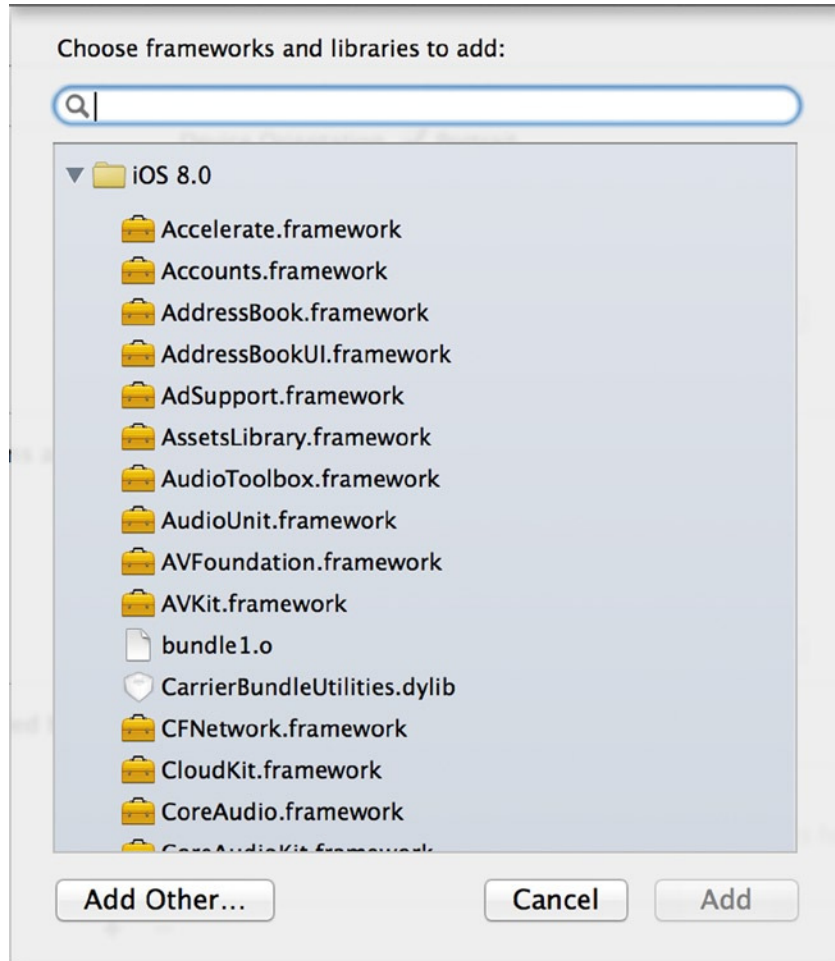


Figure 9-7. The list of frameworks and libraries available with the SDK in Xcode 6

4. Scroll through the list until you see `MapKit.framework`, or use the filter bar and type “map” to narrow the list substantially.
5. Select the Map Kit framework, and click the Add button. You return to Xcode, and the Map Kit framework is added to the project.

It’s important to be aware of the APIs available for your development platform, so you can create the best, most functional and integrated application. Also, frameworks become deprecated between releases of iOS and are replaced with new classes and methods. Fortunately, the documentation in Xcode lists all the frameworks and describes their purpose.

To see this, open the Documentation Viewer by going to Help ► Documentation and API Reference ($\text{⌘}+\text{⌘}+0$). Search for “Device Frameworks”, and open the document. As you can see in Figure 9-8, the document contains a table listing all the available frameworks and the version of iOS in which they were introduced. This page is updated with each iOS release, so it makes sense to bookmark it.

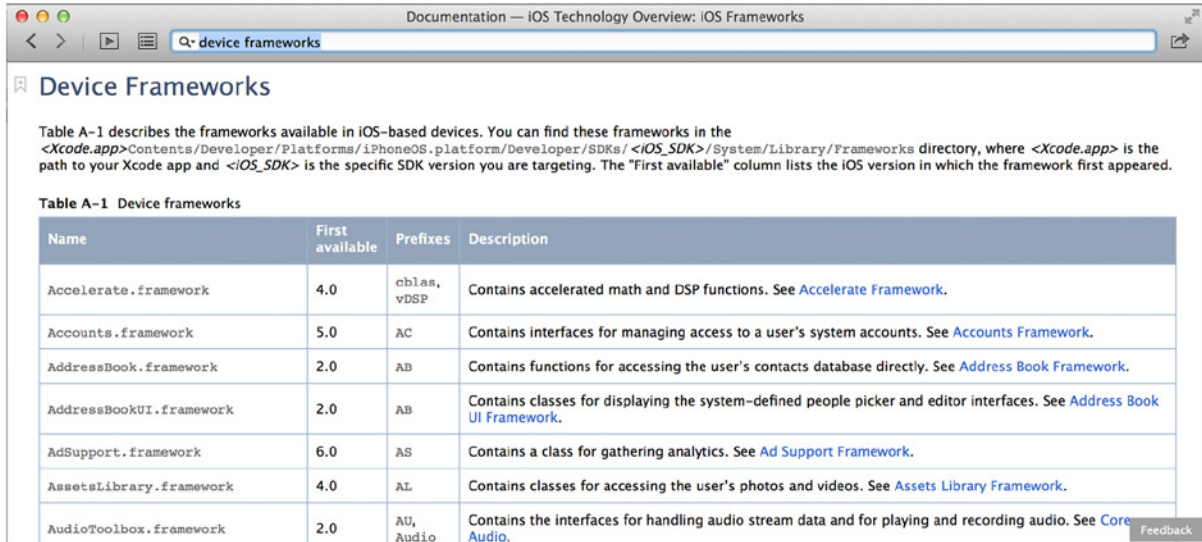


Figure 9-8. Listing the frameworks available, in the Documentation Viewer

Bookmarking was covered in Chapter 5, but to remind you, simply click the bookmark symbol as shown in Figure 9-9. This section will now be quickly accessible in future updates.



Figure 9-9. Bookmarking the Device Frameworks document

Now that you've added the prerequisite framework for using a map view, you're in a position to run the application again. Run it, you should be greeted by a map showing an area of the world, as shown in Figure 9-10.



Figure 9-10. The MapPin application displays an interactive map without you even writing a line of code

Manipulating a Map View

You haven't written a single line of code, but already what the application does is fairly impressive; you can pan and zoom the map. Although this is initially fun, it's not very useful. If you're writing a Map Kit–based application, you'll instinctively want to add your own touches, setting the initial position of the map and adding some landmarks, as I now explain.

In this application, you set the region property of the map view to show a map of Wales in the UK, and then you add points of interest that display as pushpins. But the first thing to do is to align the map correctly using constraints and then create an outlet for the map view:

1. Open `Main.storyboard` from the Project Navigator.
2. Select the map view, and click the Pin icon.
3. Uncheck `Prefer Margin Relative`, and then click the four I bars at value 0 to lock the map view to the screen edges, as shown in Figure 9-11. Click `Add 4 Constraints`.

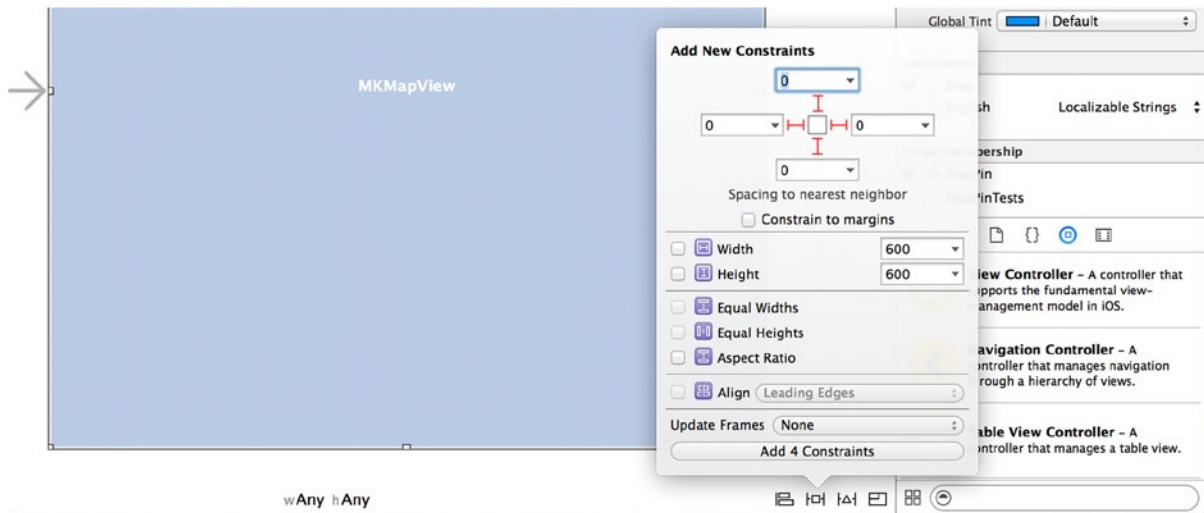


Figure 9-11. *Pinning the map view in place*

4. Now that the view is pinned in place, enable the Assistant Editor so you can create an outlet. Make sure the file displayed in the code editor is `ViewController.swift`.
5. Control-drag a connection from the map view to the header just above the `@end` line, as shown in Figure 9-12.

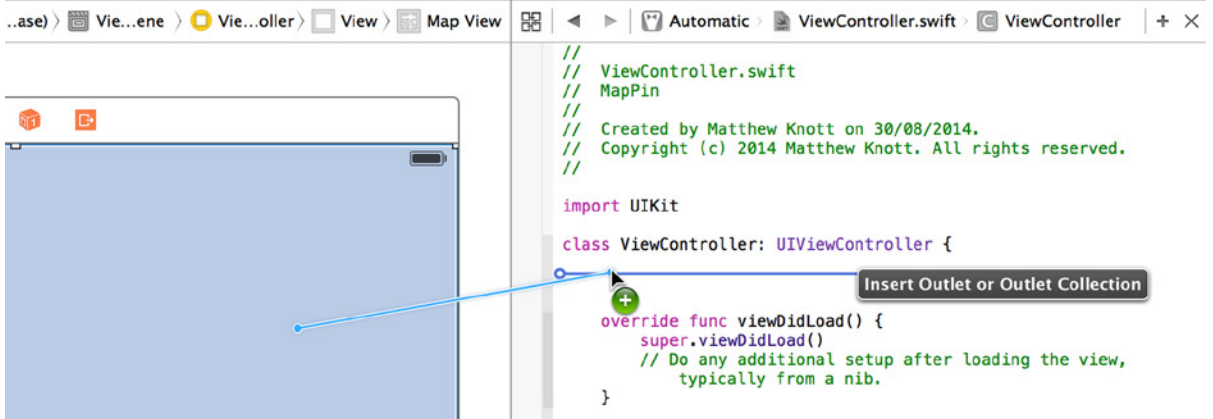


Figure 9-12. Dragging a connection from the map view for an outlet

6. Name the outlet `mapView`, and then click Connect. You see an error and a warning for the outlet because the import reference for the Map Kit framework hasn't been added to the view controller's header file.
7. From the Project Navigator, open `ViewController.swift`. Switch back to the Standard Editor. Import the Map Kit framework so you have access to the map view class and its associated classes. To do this, add the following code after the `import UIKit` line (remember, it's case sensitive):

```
import MapKit
```

8. In order to manipulate the map view from the view controller, you need to set it up as the delegate. Therefore, the next thing to do is add the `MKMapViewDelegate` protocol to the view controller. Amend the `class ViewController: UIViewController` line with the highlighted code as follows:

```
class ViewController: UIViewController, MKMapViewDelegate {
```

9. Before moving on, check that the beginning of your view controller matches that shown here:

```
import UIKit
import MapKit
```

```
class ViewController: UIViewController, MKMapViewDelegate {

    @IBOutlet weak var mapView: MKMapView!
```

10. You're finished with the header and are ready to move on to the implementation file, where you can start manipulating the map. Open `ViewController.m` from the Project Navigator. Scroll to the `viewDidLoad` method.
11. You need to tell the map view that the view controller will be its delegate, so it obeys the instructions you send it. To do this, you use the `delegate` property of the `mapView` object and set it to `self`. In the `viewDidLoad` function, after the `super.viewDidLoad()` line, add the following statement:

```
mapView.delegate = self
```

12. To position the map view in a specific position, you need to create a region, represented by `MKCoordinateRegion`. Think of a region as an invisible window that can be set at a specific location and that shows a specific amount of the map. To create your region, you need two sets of values: the latitude and longitude for the center point of the region, and the north-to-south and east-to-west span values. Drop down a line, and add the following three lines of highlighted code:

```
override func viewDidLoad() {
    super.viewDidLoad()
    mapView.delegate = self
    let centerPoint = CLLocationCoordinate2D(latitude: 52.011937, longitude: -3.713379)
    let coordinateSpan = MKCoordinateSpanMake(3.5, 3.5)
    let coordinateRegion = MKCoordinateRegionMake(centerPoint, coordinateSpan)
}
```

Note You're probably familiar with longitude and latitude, but the *span* is a concept that is unique to Apple Maps. It consists of a *latitude delta*, which is a measurement in degrees north to south that equates to roughly 111 kilometers; and a *longitude delta*, which is also measured in degrees but, unlike the latitude, equates to a distance that varies from 111 kilometers at the equator to 0 at either pole. If you want to show a fixed zoom level where the location can't be guaranteed, you'll get more consistent results with the `MKCoordinateRegionMakeWithDistance` method.

13. You need to apply this region to the `mapView` object. Do this by calling two methods that apply the region in a way that ensures the map displays properly. Drop down a line, and add the following code:

```
let centerPoint = CLLocationCoordinate2D(latitude: 52.011937, longitude: -3.713379)
let coordinateSpan = MKCoordinateSpanMake(3.5, 3.5)
let coordinateRegion = MKCoordinateRegionMake(centerPoint, coordinateSpan)

mapView.setRegion(coordinateRegion, animated: false)
mapView.regionThatFits(coordinateRegion)
```


- Run the application. When the map loads, it should be focused over Wales, as shown in Figure 9-13. If your view doesn't match the figure, check that you set the delegate correctly and that your latitude and longitude values are spot on.

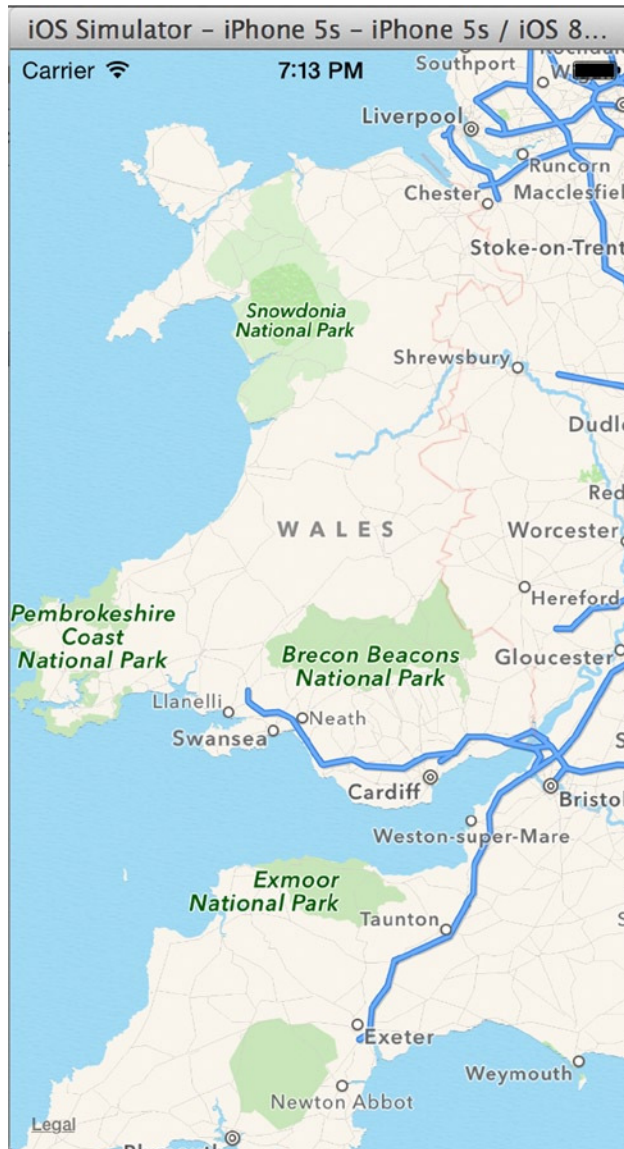


Figure 9-13. The map view now that you've specified a region to display

Knowing how to move the map where you want it to be and setting the correct region to view are basic but essential skills when working with a map view. Another common requirement in a Map Kit application is marking locations on the map using pushpins.

You do this by creating instances of the `MKPointAnnotation` class and adding them to the map individually using the `addAnnotation` method. To instantiate an `MKPointAnnotation`, you set three attributes—`title`, `subtitle`, and `coordinates`—as follows:

15. Drop down a line in the `viewDidLoad` method, and add the following highlighted code:

```
mapView.setRegion(coordinateRegion, animated: false)
mapView.regionThatFits(coordinateRegion)

var annotation1 = MKPointAnnotation()
annotation1.title = "Swansea Bay"
annotation1.subtitle = "Beautiful Beaches"
annotation1.coordinate = CLLocationCoordinate2DMake(51.587736, -3.90152)

var annotation2 = MKPointAnnotation()
annotation2.title = "Menai Bridge"
annotation2.subtitle = "Fantastic Engineering"
annotation2.coordinate = CLLocationCoordinate2DMake(53.220527, -4.163561)

var annotation3 = MKPointAnnotation()
annotation3.title = "Parc Y Scarlets"
annotation3.subtitle = "Oh Dear"
annotation3.coordinate = CLLocationCoordinate2DMake(51.678809, -4.127469)

var annotation4 = MKPointAnnotation()
annotation4.title = "Castell Coch"
annotation4.subtitle = "A Fairytale Castle"
annotation4.coordinate = CLLocationCoordinate2DMake(51.535819, -3.2547)

var annotation5 = MKPointAnnotation()
annotation5.title = "Arthur's Stone"
annotation5.subtitle = "Rock Of Legend"
annotation5.coordinate = CLLocationCoordinate2DMake(51.593735, -4.179525)

mapView.addAnnotation(annotation1)
mapView.addAnnotation(annotation2)
mapView.addAnnotation(annotation3)
mapView.addAnnotation(annotation4)
mapView.addAnnotation(annotation5)
```

16. Rerun the application. You should see five annotations. Tap an annotation, as shown in Figure 9-14, to display the text associated with that pushpin.



Figure 9-14. The map view, now with five annotations showing various attractions in Wales

With very little code, you’ve made a really useful and interactive application! You could make endless customizations, such as replacing the pushpins with an image or adding controls to the callout that display the annotation text. I won’t explain how to customize the pushpins any further, because the focus of this book is Xcode. However, I’ll show you one of the key steps you typically perform before any customization of a class’s behavior: subclassing the `MKPointAnnotation` class and then creating a custom initializer to simplify creation of the annotations.

Subclassing `MKPointAnnotation`

Because this code is reusable in different projects, you create it in a separate class file. You’re working with annotations, so it probably comes as no surprise that you subclass the `MKPointAnnotation` class and replace the pushpin objects you just created with this new class, `MyPin`:

1. To create the new class, select `File` ► `New` ► `File (⌘+N)`. Choose `Cocoa Touch` from the left menu, and then select the `Cocoa Touch Class` template, as shown in Figure 9-15. Click `Next`.

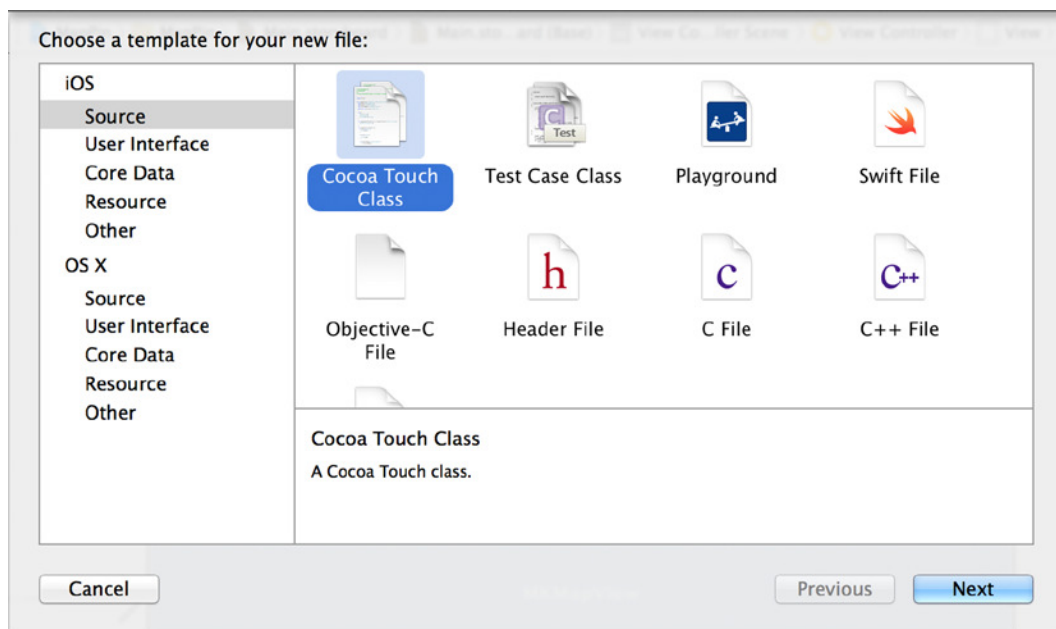


Figure 9-15. Creating a new Objective-C class

2. Name the class `MyPin`, and set `Subclass Of` to `MKPointAnnotation`. Click `Next`. Then accept the default save location, and click `Create`.
3. You have successfully created your custom class, but to make it work you need to make some modifications. Open `MyPin.swift` in the Project Navigator.

4. You can add a custom initializer to simplify the process of creating the annotation. The initializer takes the three parameters that are required: title, subtitle, and coordinate. Add the following highlighted code:

```
import UIKit
import MapKit

class MyPin: MKPointAnnotation {

    init(title : String, subtitle : String, coordinate : CLLocationCoordinate2D) {
        super.init()
        self.title = title
        self.subtitle = subtitle
        self.coordinate = coordinate
    }
}
```

As you can see, the initializer is straightforward. The `MKPointAnnotation` class already has `title`, `subtitle`, and `coordinate` properties; you've written an initializer that lets you create the object with all the parameters you need for the application in one go. If there were others you wanted to set when you initialize the object, you could add them to the initializer and eliminate the need to set properties after initialization.

5. With the custom initialization function written, you can modify the annotations declared in the `viewDidLoad` function to use the new class. Open `ViewController.swift` in the Project Navigator.
6. Scroll down to the `viewDidLoad` method where you created the five instances of `MKPointAnnotation`, and replace them with the highlighted code:

```
override func viewDidLoad() {
    super.viewDidLoad()

    mapView.delegate = self
    let centerPoint = CLLocationCoordinate2D(latitude: 52.011937, longitude: -3.713379)
    let coordinateSpan = MKCoordinateSpanMake(3.5, 3.5)
    let coordinateRegion = MKCoordinateRegionMake(centerPoint, coordinateSpan)

    mapView.setRegion(coordinateRegion, animated: false)
    mapView.regionThatFits(coordinateRegion)

    var annotation1 = MyPin(title: "Swansea Bay",
        subtitle: "Beautiful Beaches",
        coordinate: CLLocationCoordinate2DMake(51.587736, -3.90152))

    var annotation2 = MyPin(title: "Menai Bridge",
        subtitle: "Fantastic Engineering",
        coordinate: CLLocationCoordinate2DMake(53.220527, -4.163561))
}
```

```

var annotation3 = MyPin(title: "Parc Y Scarlets",
    subtitle: "Oh Dear",
    coordinate: CLLocationCoordinate2DMake(51.678809, -4.127469))

var annotation4 = MyPin(title: "Castell Coch",
    subtitle: "A Fairytale Castle",
    coordinate: CLLocationCoordinate2DMake(51.535819, -3.2547))

var annotation5 = MyPin(title: "Arthur's Stone",
    subtitle: "Rock Of Legend",
    coordinate: CLLocationCoordinate2DMake(51.593735, -4.179525))

mapView.addAnnotation(annotation1)
mapView.addAnnotation(annotation2)
mapView.addAnnotation(annotation3)
mapView.addAnnotation(annotation4)
mapView.addAnnotation(annotation5)
}

```

Note In Objective-C, you would have had to declare the initializer in the class header, implement the initializer in the implementation file, and then import the header into the view controller in order to use the class. Swift has changed all that and created something far more straightforward.

Go ahead and rerun your application. The pushpins are still in place, but you did this with a fraction of the code.

Static Libraries, Frameworks, and Swift

Nothing exposes Swift's infancy more than its lack of support for native frameworks and static libraries. Apple doesn't support the creation of compiled frameworks or static libraries for distribution.

In the Objective-C version of this book, I would take you through creating a static library that could reuse with multiple projects—but, alas, this is isn't possible with Swift. To explain a little better, here is a quote from the official Swift blog that explains the reasoning behind this decision:

While your app's runtime compatibility is ensured, the Swift language itself will continue to evolve, and the binary interface will also change. To be safe, all components of your app should be built with the same version of Xcode and the Swift compiler to ensure that they work together.

This means that frameworks need to be managed carefully. For instance, if your project uses frameworks to share code with an embedded extension, you will want to build the frameworks, app, and extensions together. It would be dangerous to rely upon binary frameworks that use Swift — especially from third parties. As Swift changes, those frameworks will be incompatible with the rest of your app. When the binary interface stabilizes in a year or two, the Swift runtime will become part of the host OS and this limitation will no longer exist.

Apple Swift blog, July 11, 2014, <https://developer.apple.com/swift/blog/?id=2>

What can be confusing is that Xcode contains the templates for frameworks and static libraries and even offers Swift as a language. But when you create a project, it's created in Objective-C. Because the language focus of this book is Swift, it doesn't make sense to mix and match different languages at this stage.

On a more positive note, when the language matures, Apple will almost certainly add support for native Swift frameworks and libraries. Let's move on to creating different versions of the application in the same project by using targets.

Working with Multiple Targets

I hope that when you finish this book, you're ready to start writing your own applications for the App Store. When you create an application that you want to charge for, it's likely that you'll also want to create a free version with fewer features to tempt users into upgrading to the full version. You can create a new project and copy all your code over to it, but then you've fallen into the snare of having to maintain two versions of the same code.

By using different targets, Xcode allows you to maintain multiple versions of the same application in the same project and then, in the code, identify which version of the application is running and adjust the functionality to suit. For this example, let's create another target called MapPinSatellite that displays the map in satellite mode instead of the default standard mode.

Rather than create a new target and apply a lot of settings, you can duplicate the existing MapPin target:

1. Select the MapPin project from the Project Navigator. When the project settings load, select the MapPin target and press ⌘+D, or right-click the MapPin target and click Duplicate, as shown in Figure 9-16.

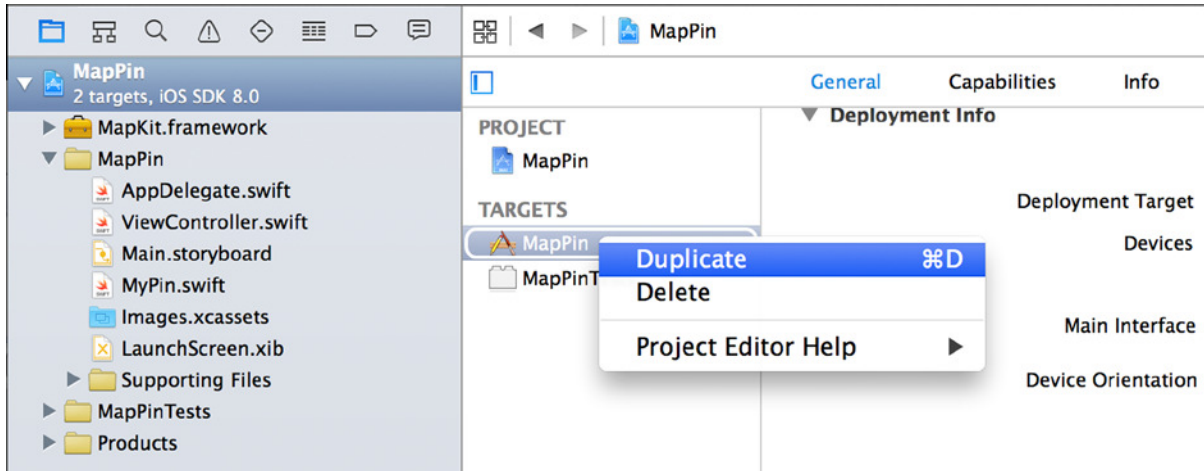


Figure 9-16. Duplicating the MapPin target

2. Xcode detects that you're duplicating an iPhone-specific target and asks if you want to convert it for use with an iPad, as shown in Figure 9-17. In this instance, you just want to duplicate the target, so select Duplicate Only.

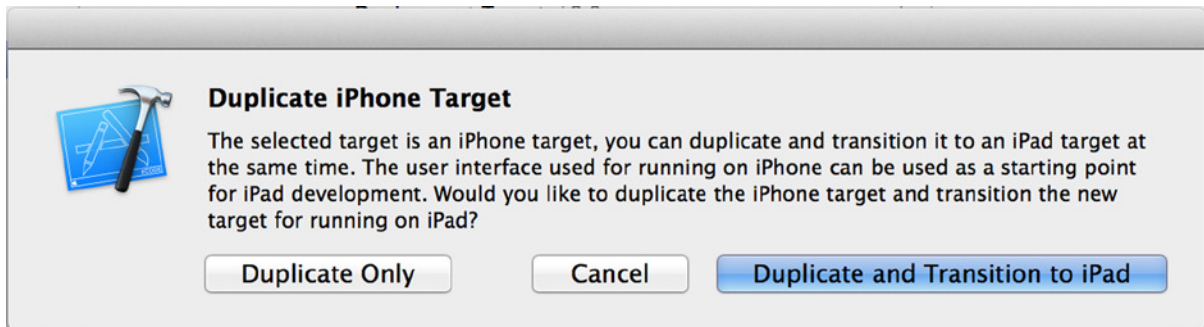


Figure 9-17. Xcode prompts you if you try to duplicate an iPhone-specific target

3. Xcode duplicates the target and names it MapPin Copy. This is great but not really what you want your target to be named. Click the MapPin Copy target, and then click it again so you can edit its name. Change the name to MapPinSatellite, as shown in Figure 9-18.

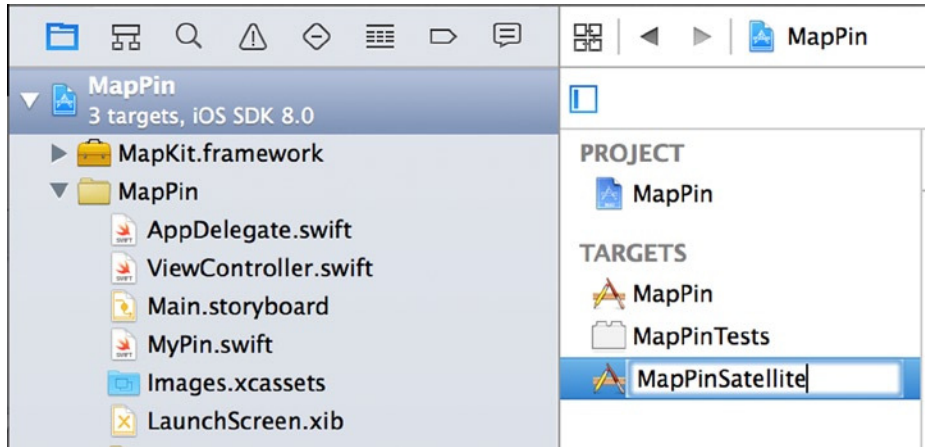


Figure 9-18. Renaming the new target

4. You need to change a couple of the target's settings to reflect its new name. With the `MapPinSatellite` target selected, open the Build Settings tab. There are dozens of settings in this list! To make things easier, use the search filter at the top of the page. First, change the Product Name property to `MapPinSatellite`; to do this, search for *product name*, double-click the words *MapPin copy*, and change them to *MapPinSatellite*, as shown in Figure 9-19.

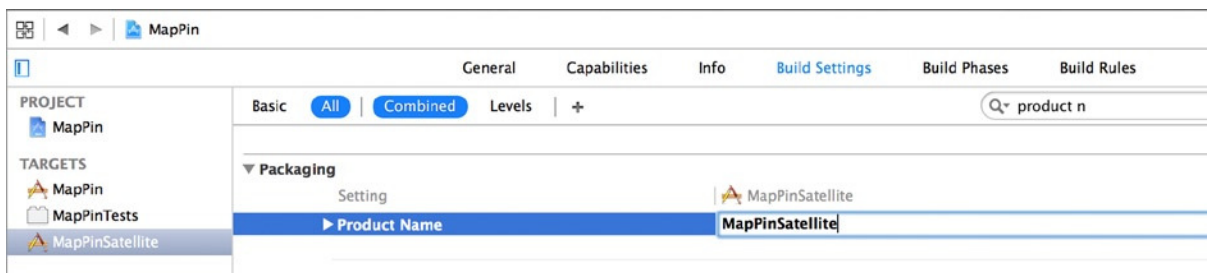


Figure 9-19. Changing the product name setting to `MapPinSatellite`

5. The next setting to change is the name of the `info.plist` file. Change the filter, and then rename the value from `MapPin copy-Info.plist` to `MapPinSatellite-Info.plist`, as shown in Figure 9-20.

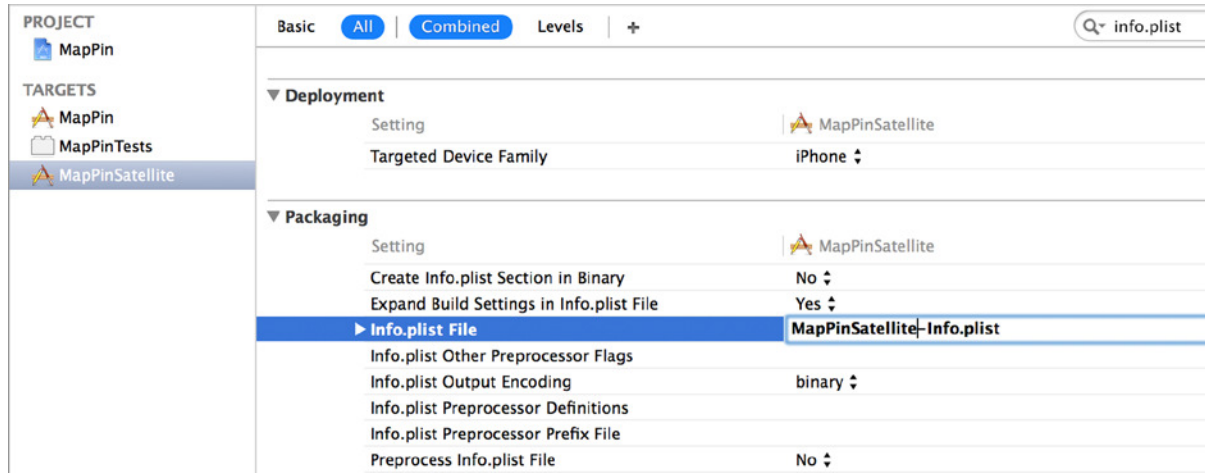


Figure 9-20. Renaming the `info.plist` setting

- When you duplicated the target, Xcode actually duplicated three things: the target, the `info.plist` file, and the targets scheme. You've just named the `info.plist` file in the settings for your target, so you should change the `info.plist` file name next. In the Project Navigator, notice that at the bottom of the project there is now a `plist` file named `MapPin copy-Info.plist`. Highlight the file, press `Return` to begin editing, and change the name to `MapPinSatellite-Info.plist`. Your Product Navigator should resemble Figure 9-21.

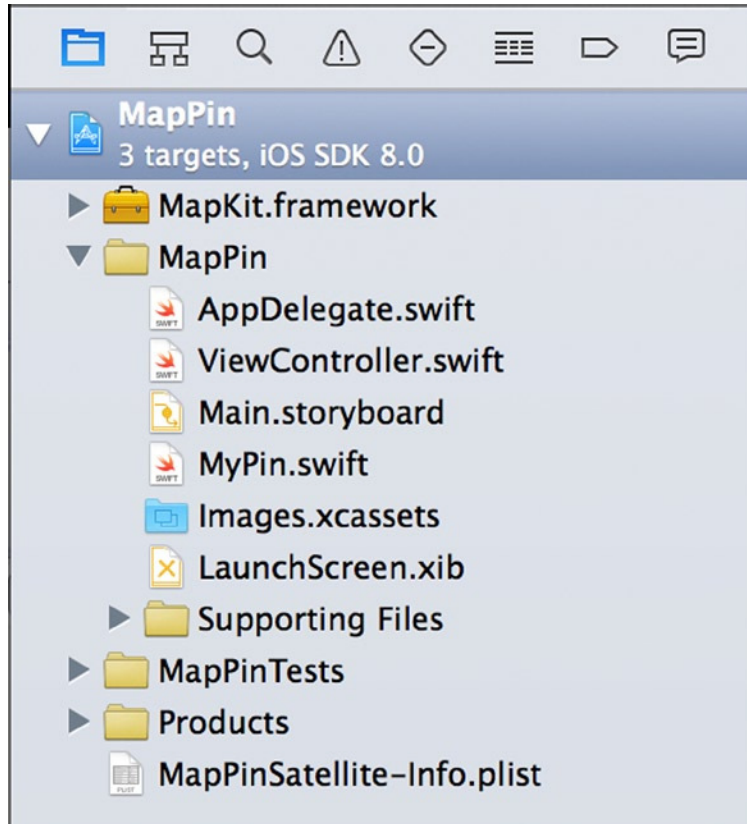


Figure 9-21. The Project Navigator after renaming the file

7. There is one final item to change: the scheme. Go to Product ► Scheme ► Manage Schemes, and you're presented with a list of available schemes, as shown in Figure 9-22.

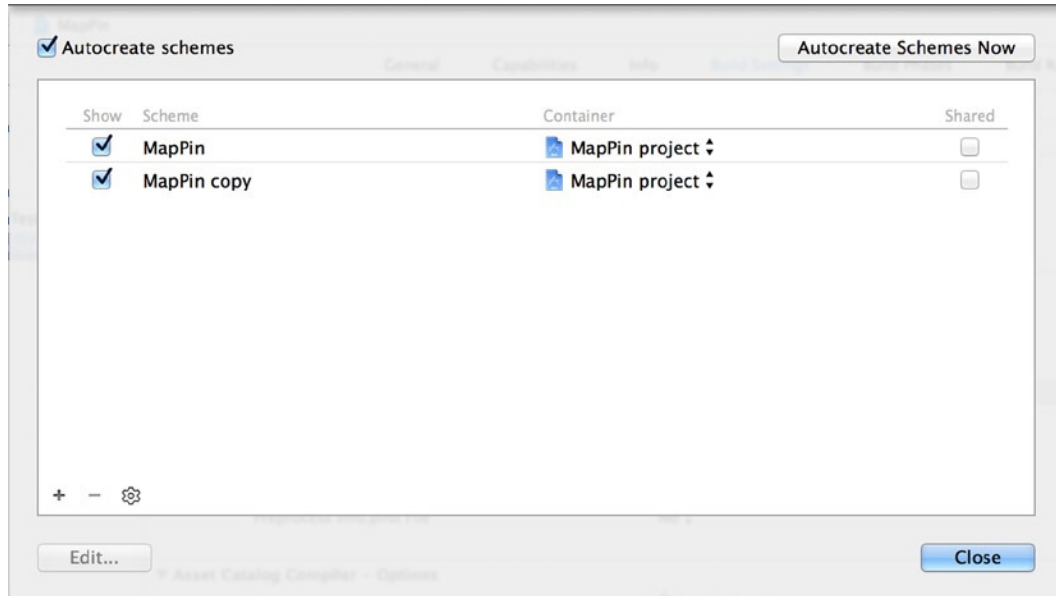


Figure 9-22. Xcode's Manage Schemes view, listing the available schemes

8. Highlight the bottom scheme named MapPin Copy, and press Return. Change the scheme name to MapPinSatellite. You've now updated everything required to start taking advantage of your new target. Click OK to close the window.
9. Open `ViewController.swift` from the Project Navigator, and scroll down to the `viewDidLoad` function. To separate functionality based on the active target, you first need to identify which version of the application is being run. You find this by examining the application's bundle identifier, which changes depending on which target scheme is being run. The bundle identifier is a combination of the Company Identifier specified in Figure 9-2 and the Product Name.
10. You need to retrieve the bundle identifier, assign it to a `String` object, and then use `println` to output the identifier to the console. After the `super.viewDidLoad()`, drop down a line and add the following highlighted code:

```
override func viewDidLoad() {
    super.viewDidLoad()

    var currentBundle : String =
    NSBundle mainBundle().objectForInfoDictionaryKey("CFBundleIdentifier") as String
    println(currentBundle)
}
```

- Run the application, and you see the bundle identifier in the debug console, as shown in Figure 9-23.



Figure 9-23. The bundle identifier shown in the debug console

- To change the scheme for this project from MapPin to MapPinSatellite, select the MapPin scheme next to the Run and Stop buttons in the Toolbar area, and select MapPinSatellite, as shown in Figure 9-24.

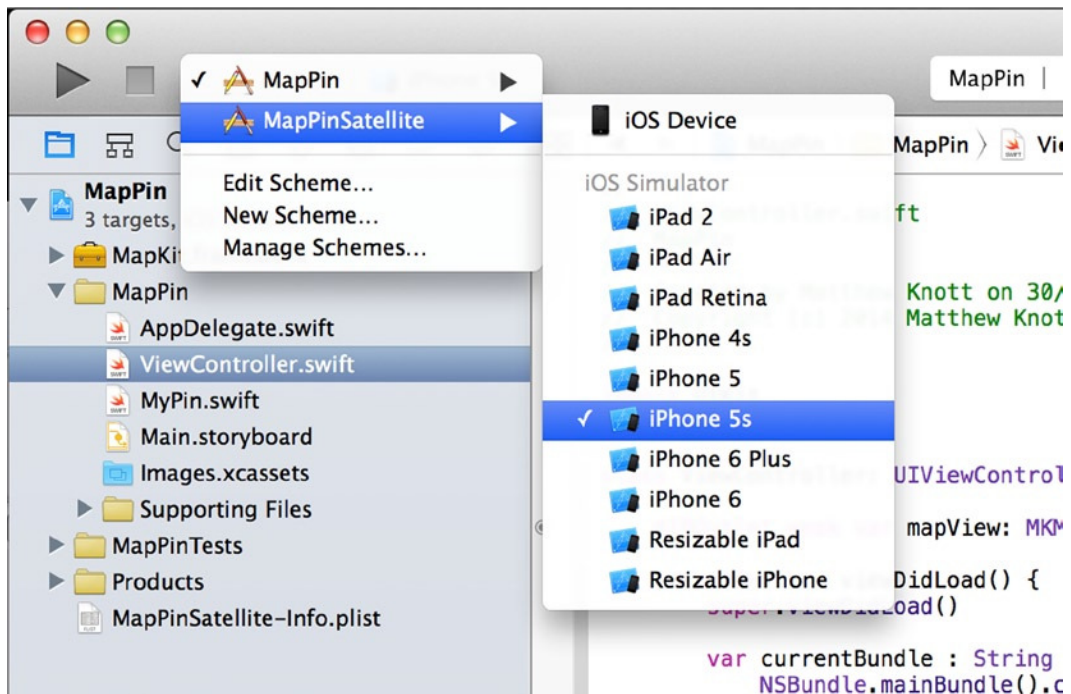


Figure 9-24. Changing to the MapPinSatellite scheme

13. If the Stop button is active, click it to terminate the application running under the other scheme; otherwise Xcode will throw an error about the simulator being in use. Run the application again with the new scheme: the bundle identifier in the debug area changes to match the product name specified for this scheme, which ends with MapPinSatellite. You now have two distinct values that let you implement different functionality.
14. The different functionality in this project is that the original target runs using the standard map type, but the satellite version runs with the satellite map type. In the `ViewController.swift` file, after the line `mapView.delegate = self` line, add the following highlighted code, remembering that MapPinSatellite is case sensitive and must match what you've written as a bundle identifier:

```
if currentBundle.hasSuffix("MapPinSatellite")
{
    mapView.mapType = MKMapType.Satellite
}
```

Note `hasSuffix` is a helper function that examines the end of the string to see if it matches the supplied string. This is quicker than typing the full string for comparison and reduces the chance of a typo.

15. Run your application once with the MapPinSatellite scheme and again with the MapPin scheme to appreciate the difference, as shown in Figure 9-25. Remember to stop the application when switching schemes.

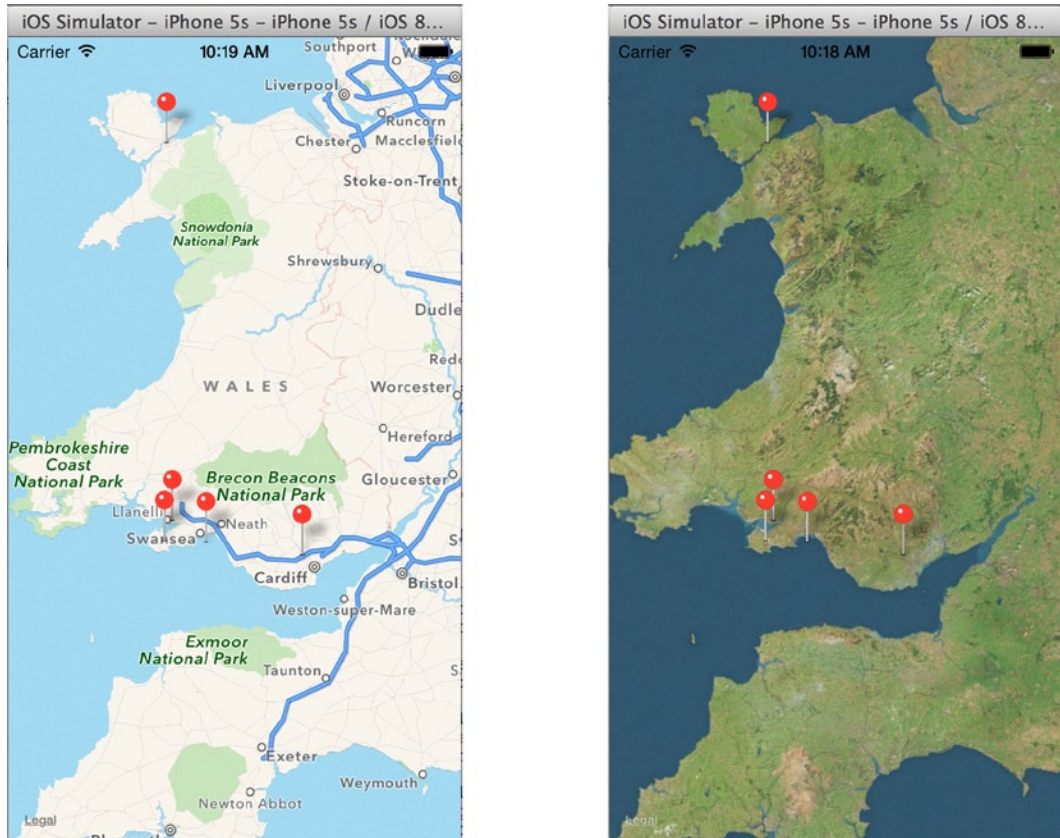


Figure 9-25. The application running under both schemes

Summary

As you get further into this book, the topics become more advanced. If you're serious about becoming an iOS or Mac OS developer, then the skills you've learned in this chapter will help you do super-efficient work, using frameworks and custom initializers to good effect.

This chapter combined your knowledge of frameworks and targets to create a Map Kit-based application that shows just a few of the many hundreds of points of interest in Wales. You learned how to add annotations to the map view and then subclassed `MKPointAnnotation` to create a one-hit initializer for the annotation object.

Specifically, in this chapter you have done the following:

- Learned more about modules, which are an efficient way to expose new APIs in your project
- Learned about manually adding frameworks to a project
- Looked at the Device Frameworks section in the help documentation and bookmarked it for future reference

- Discovered how to use map views and manipulate them in code
- Duplicated a target to create different sets of features from the same code
- Learned about Swift and why native libraries and frameworks aren't supported

In the next chapter, you learn how to mold Xcode into a more personalized development environment and how to get more out of Xcode as you explore new customizations that will make you a better and more efficient developer.

Advanced Editing

Chapter 9 looked at how to add frameworks to a project and efficient ways of managing code through the use of libraries, as well as using multiple targets and aggregate targets. These skills were combined and developed as you created a basic Map Kit application that showed a selection of interesting points on a map of Wales, the country I live in. If you want to become an accomplished iOS or Xcode developer, either on your own or in a team of developers, then learning how to efficiently reuse code is essential.

This chapter maintains the efficiency theme and focuses on the code editor while you create a fun Sprite Kit–based application called AlienDev that shows our hero, the alien dev (Alien Cyborg Dev, to give him his full name) surrounded by an increasing numbers of bugs (evil ones). I explain how small bits of reusable code can be saved as snippets, readily available to be dropped into your application, as well as look at the many ways you can customize Xcode to work for you. It’s a cliché, but everyone is different, and developers are no exception. Xcode is hugely customizable, from the font and colors used in the code editor to the way Xcode reacts to different events. There are many ways you can tweak the IDE to be a better environment for you to code in. I also discuss how to work with large implementation files by using code folding and some of the subtle ways you can efficiently navigate your code using the jump bar and its pragma marks to bring order to your jump-bar hierarchy.

Without further ado, let’s get started on the project—let the bug wars commence!

Getting Started

The project for this chapter is created using the Sprite Kit application template. As mentioned in Chapter 3, Sprite Kit is an exciting new framework that Apple introduced with Xcode 5 and iOS 7 to let developers easily create 2D animations and even games without having to use a third-party system such as Cocos2D or Unity.

Note I recommend that you download the resources for this project and all the others in the book from the Apress web site at www.apress.com. Alternatively, if you're feeling adventurous, you can create your own characters.

Follow these steps:

1. Open Xcode, and create a new project by clicking Create A New Xcode Project on the Welcome screen or going to File ► New ► Project (⌘+Shift+N). Select the Game template, and click Next.
2. Name your project AlienDev. Be sure Game Technology is set to SpriteKit and Devices is set to iPhone, not iPad or Universal. Configure the other values to your own preference; mine are shown in Figure 10-1. Click Next.

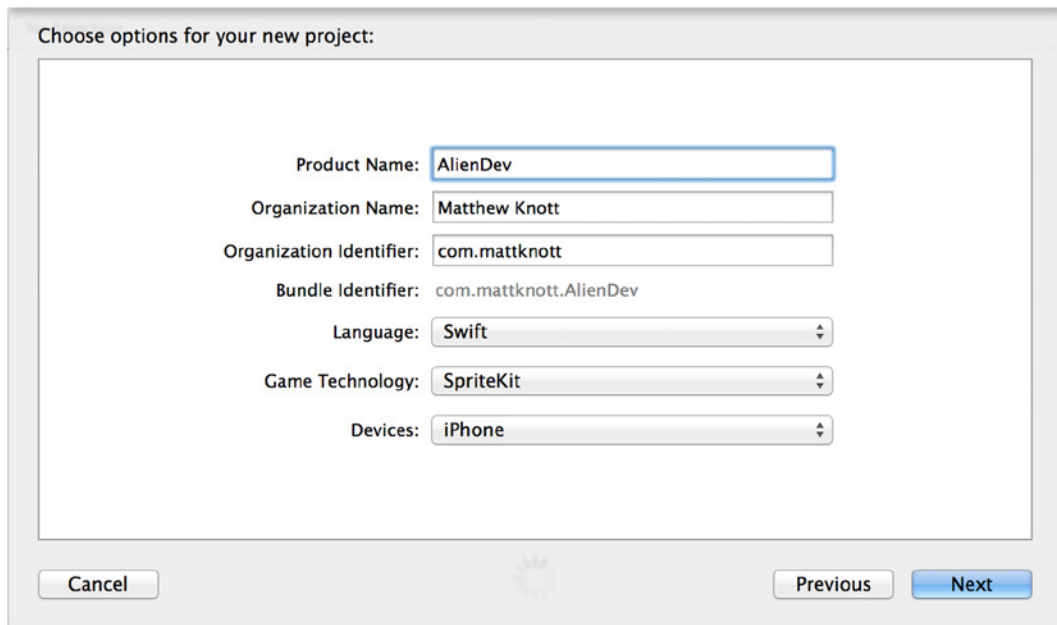


Figure 10-1. Setting up the initial project

3. By now you should be familiar with the process: you want to save in the default location, and you don't want to create a Git repository, so go ahead and click Create.

4. Xcode takes you straight into the project settings. You want the application to run in landscape mode only, so scroll down to the Deployment Info section. The default-supported orientations for an iPhone application are Portrait, Landscape Left, and Landscape Right; you need to uncheck Portrait so that only the landscape options remain, as shown in Figure 10-2.

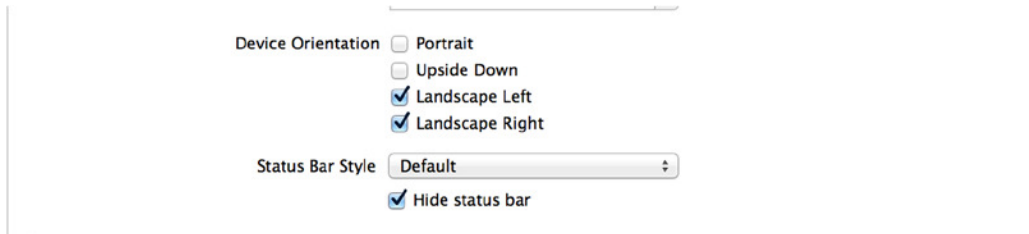


Figure 10-2. Restricting Device Orientation to landscape

5. Before you go any further, let's stop to take a look at the assets for the application. Figure 10-3 shows the two characters you use: the hero—the alien dev—on the left and the villainous bug on the right. One thing you should be able to gather is that I am *not* a graphic artist. Each character is represented by a single high-resolution image: `Dev.png` and `Bug.png`. Both are transparent png images, which means the background appears in the white space behind them. Downloaded these images, or create your own with the same names.

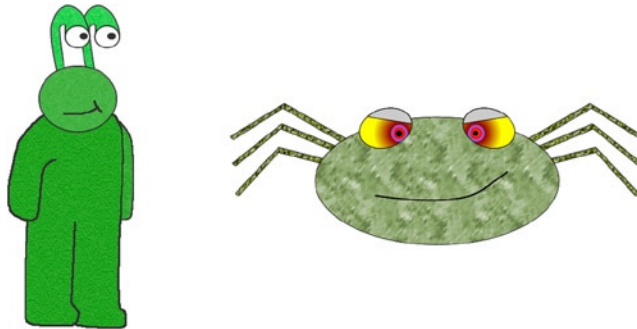


Figure 10-3. The hero and villain for this application

6. Open `Images.xcassets` from the Project Navigator, and drag the files from the Finder into the sidebar of the asset catalog in Xcode, as shown in Figure 10-4.

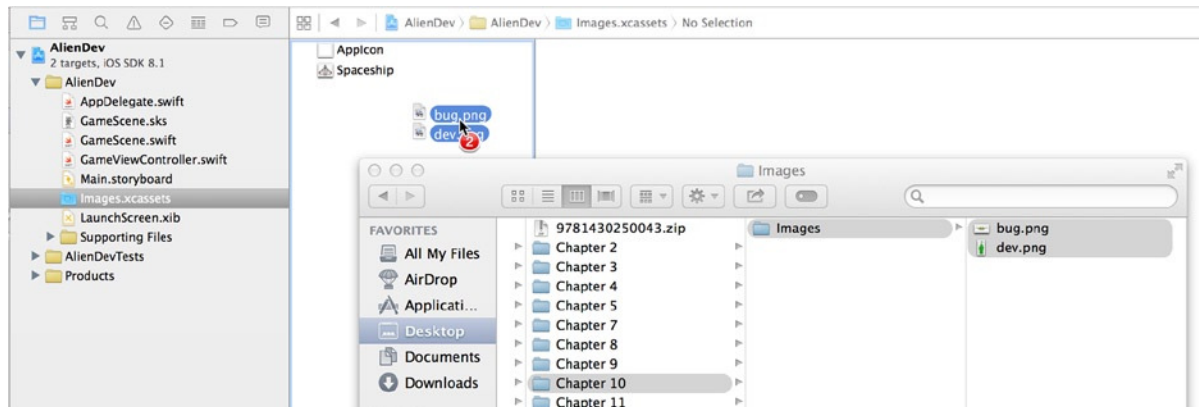


Figure 10-4. Dragging the image files into the Supporting Files group

- When you release the files, image sets are created automatically. Highlight the spaceship image set, and remove it with the Backspace key so that your asset catalog resembles Figure 10-5.



Figure 10-5. The images asset catalog with the new image sets

With the assets in place, let's start to look at the code for the application. The rest of this chapter is dedicated to code: how to look at it, how to manage it efficiently, and how to quickly navigate it.

Efficient Editing

A good developer can be compared to a master craftsman, except that instead of a hammer and chisel, developers have IDEs and compilers. Like a master craftsman, good developers take pride in their work, taking time to achieve perfection and adding the painstakingly small touches that set their product apart from the rest. Unlike a master craftsman, though, when the tool isn't right, you don't have to pick up a different one, because developer tools aren't static; Xcode's interface is an organic entity that can be tailored to your needs and made to work the way you want it to.

Changing Color Schemes

One of the things you find when working in a team of developers is that everyone has a different way of working, whether it's how they write their comments or how they indent their code. But when you're working with a flexible IDE like Xcode, most developers tailor the color scheme to meet their needs. You may not have realized it, but changing the color scheme can have a drastic effect on productivity; sometimes I find the lightness of the standard color scheme can cause eye strain or even migraines, so I switch to a dark background scheme with high contrast to make the code easier on the eyes.

To see what Xcode lets you change, open the Xcode preferences by selecting Xcode ► Preferences (⌘+); when the preferences open, select the Fonts & Colors tab, as shown in Figure 10-6.

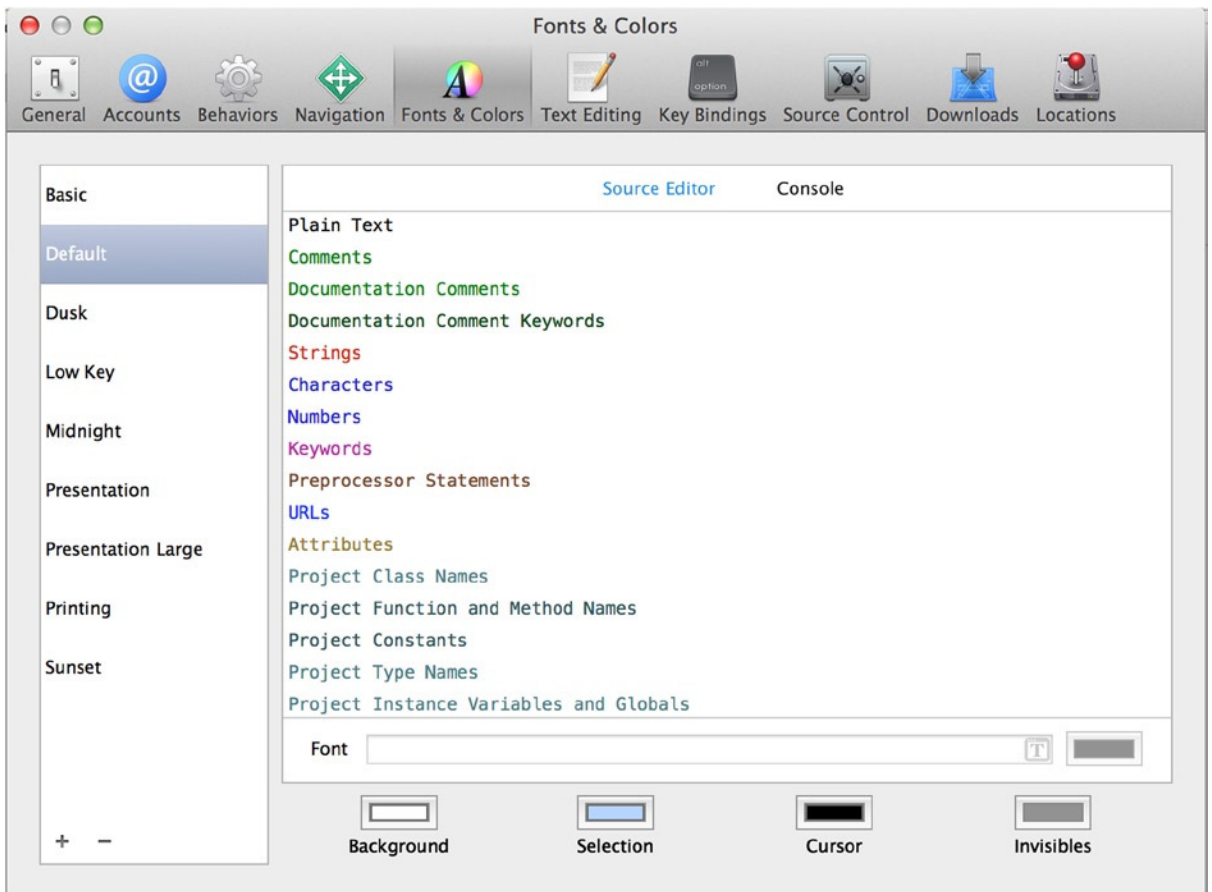


Figure 10-6. The Fonts & Colors tab in Xcode's preferences

The Fonts & Colors tab is divided into four key areas, as highlighted in Figure 10-7.

- *Theme list:* This area lists the preset themes Xcode makes available. A *theme* is a predetermined combination of fonts and colors.
- *Detail view:* The items listed in the detail view are known as *syntax categories*. These represent all the conceivably customizable elements of code in either the source editor or console, depending on which tab is selected at the top of the detail view.
- *Font configuration:* This area allows you to customize the font and color for the selected syntax category.
- *General colors:* The general colors control the background color; the selection color, which is used when highlighting text; the cursor color; and the “invisibles” color or the instruction pointer if you’re looking at the Console tab.



Figure 10-7. The key areas of the Fonts & Colors tab

Try changing the different themes: Dusk and Midnight are good if you want a high-contrast theme, whereas Low Key and Sunset are great if you want something that’s a little washed out. There are specialist themes too, with Printing giving a monochrome look and Presentation using an enlarged font size for when you’re hooked up to a projector to demonstrate your code.

Select the Default theme for now, and then choose the Comments syntax category. The font configuration area displays the key details about the selected syntax category, as shown in Figure 10-8. You can see that the font used for comments is Menlo Regular, size 11, and green. To change the font details, click the T icon as you would when setting font information for labels and text fields in Interface Builder. Unlike in Interface Builder, however, you’re presented with the standard font-selection dialog that will be familiar to anyone who has used Mac OS X for a while; this is shown in Figure 10-9.

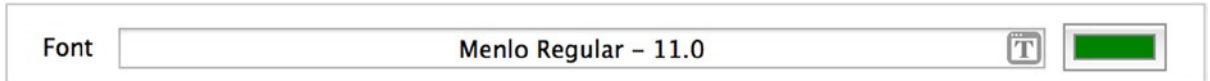


Figure 10-8. The font configuration details

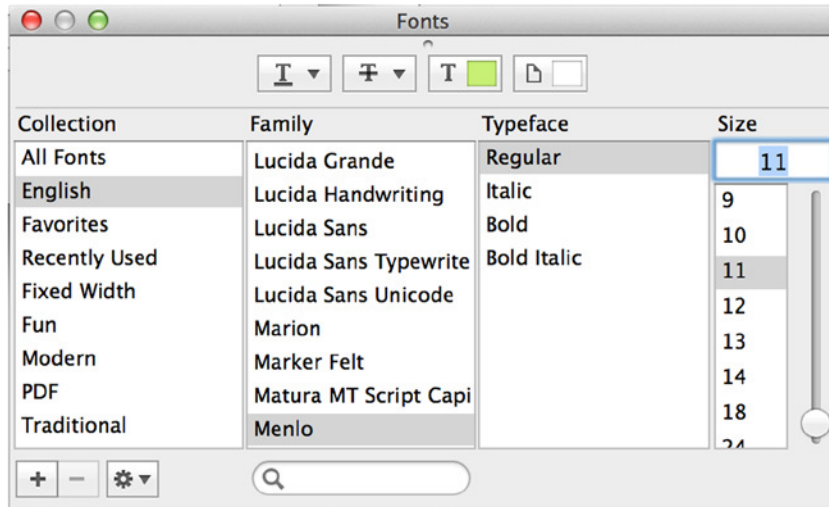


Figure 10-9. The font-selection dialog

Creating a New Theme

Although it's easy to modify a theme, there is no Reset button to revert your changes to the theme's original settings. If you modify a theme and want to get it back to its original state, the best way is to start over and create a new theme based on one of the preset themes. This may sound drastic, but it's perfectly all right; all the themes that come with Xcode by default are stored as templates in Xcode that can be re-created in a couple of clicks.

First, let's examine the Add button at the bottom of the theme list. Click the + symbol, as shown in Figure 10-10. You're presented with a pop-up menu that effectively gives you two choices: you can either duplicate your current theme or create a new theme from one of the default templates.

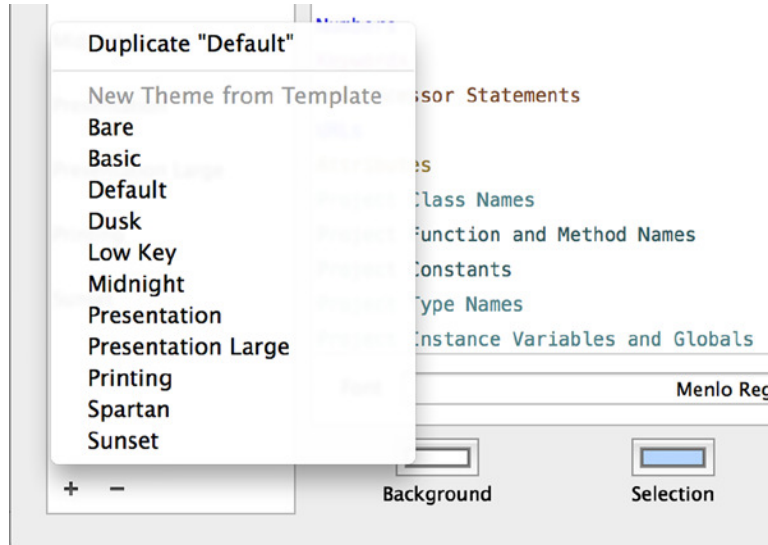


Figure 10-10. The Add menu in the themes list

Select Midnight, and a new version of the Midnight theme is added to your themes list, as shown in Figure 10-11. At this point you can change the name of the theme to anything you want. I'll name my theme Matt's Midnight; call your theme whatever suits your fancy.

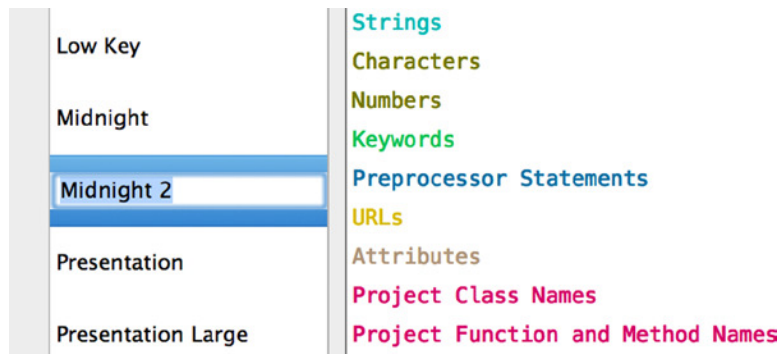


Figure 10-11. The new theme, ready to be customized

There may be a time where you get bored with your theme and want to remove it; this is done by clicking the minus symbol next to the + symbol in the themes list.

Sharing or Importing a Theme

You've spent hours customizing your theme to suit your preferences, fine-tuning every syntax category until the color and tone are perfect, and now you want to share it with the world. The good news is that this is really easy to do!

All the themes you create are stored on your computer in a `dvtcolortheme` file format. To locate these files, open the Finder, and then select Go from the menu bar. The Library option is hidden by default, but if you press the Option (`⌘`) key, Library appears; choose it, and the user library appears, as shown in Figure 10-12.



Figure 10-12. Viewing the library for my user account

The path to the themes is quite deep in the library: select the Developer folder and then Xcode **➤** UserData **➤** FontAndColorThemes. You can see the custom theme I created called Matt's Midnight. `dvtcolortheme`, as shown in Figure 10-13. From here, you can add themes you've downloaded from the Internet or copy them to share online.

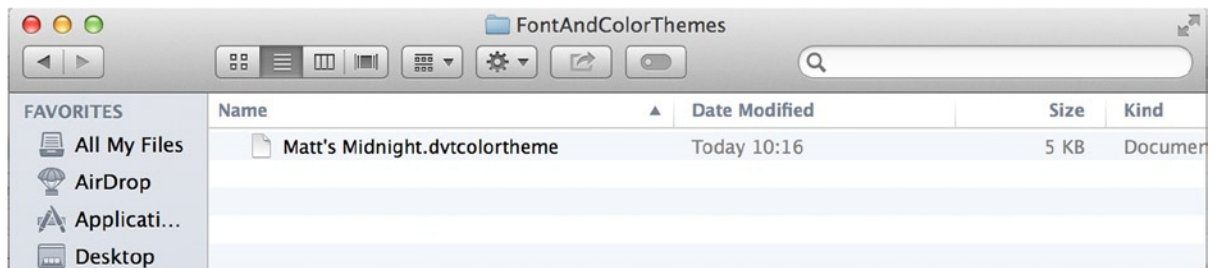


Figure 10-13. The contents of my FontsAndCoLoRThemes folder

Note The `FontAndCoLoRThemes` folder is only visible if you've duplicated a theme, so if you skipped the previous section, you may not see this folder.

Note Whenever you make changes in this folder, you must restart Xcode for those changes to take effect.

Back in Xcode, choose the theme you're happiest with, and close the preferences by clicking the red ball in the top-left corner. You now know everything there is to know about themes in Xcode! Next, I delve into the code for AlienDev and show you how to make dealing with large amounts of code less of a chore.

Organizing and Navigating Code

You've learned how to alter the visual appearance of code, so it's time to go a bit deeper and look at the fantastic shortcuts Xcode provides to help you be super-efficient in how you code. In order to see some of the finer points of organizing and navigating through code, you first need to add that code. At this point, you've added the assets to display in the Sprite Kit scene, but if you were to run the application, it would still have all the behaviors of the default Sprite Kit template. The first thing you need to do is make some modifications to `GameViewController.swift` so that the stage is set for adding the hero and the villainous bugs.

Start by opening `GameViewController.swift` from the Project Navigator. Because you're using a landscape-only orientation, and because of the point at which `viewDidLoad` is called, you need to create a method that performs the initialization once the view has been added to the stack. This method is called `viewWillLayoutSubviews`, and you need to add it just after `viewDidLoad`, as shown highlighted here:

```
override func viewDidLoad() {
    super.viewDidLoad()

    if let scene = GameScene.unarchiveFromFile("GameScene") as? GameScene {
        // Configure the view.
        let skView = self.view as SKView
        skView.showsFPS = true
        skView.showsNodeCount = true

        /* Sprite Kit applies additional optimizations to improve rendering performance */
        skView.ignoresSiblingOrder = true

        /* Set the scale mode to scale to fit the window */
        scene.scaleMode = .AspectFill

        skView.presentScene(scene)
    }
}

override func viewWillLayoutSubviews() {
    super.viewWillLayoutSubviews()
}
```


After you've created the new method with the single line of code, you need to move the bulk of the code from `viewDidLoad` into `viewWillLayoutSubviews`. Do this by highlighting all the code after `super.viewDidLoad()`, as shown in Figure 10-14, and using `Edit > Cut (⌘+X)` to cut the code and `Edit > Paste (⌘+V)` to paste the code into the method after the line `super.viewWillLayoutSubviews()`.

```
class GameViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

        if let scene = GameScene.unarchiveFromFile("GameScene") as? GameScene {
            // Configure the view.
            let skView = self.view as SKView
            skView.showsFPS = true
            skView.showsNodeCount = true

            /* Sprite Kit applies additional optimizations to improve rendering performance */
            skView.ignoresSiblingOrder = true

            /* Set the scale mode to scale to fit the window */
            scene.scaleMode = .AspectFill

            skView.presentScene(scene)
        }
    }
}
```

Figure 10-14. Highlighting the code in the `viewDidLoad` method that needs to be moved

After you move the code, the finished structure of the two methods should resemble the code shown next:

```
override func viewDidLoad() {
    super.viewDidLoad()
}

override func viewWillLayoutSubviews() {
    super.viewWillLayoutSubviews()

    if let scene = GameScene.unarchiveFromFile("GameScene") as? GameScene {
        // Configure the view.
        let skView = self.view as SKView
        skView.showsFPS = true
        skView.showsNodeCount = true

        /* Sprite Kit applies additional optimizations to improve rendering performance */
        skView.ignoresSiblingOrder = true

        /* Set the scale mode to scale to fit the window */
        scene.scaleMode = .AspectFill

        skView.presentScene(scene)
    }
}
```

The outcome of any code tutorial is usually a known entity, but in this case let's pretend you don't know what's going to happen as you develop this application. You've emptied all the bespoke code from the override of the `viewDidLoad` method, because it was effectively just taking up space at this point, but you *might* need it in the future. So let's set a reminder to clean up the method if it doesn't get used.

Creating Code Reminders

Xcode provides several handy tags that can be used in code comments to help you remember to deal with different tasks such as adding code to a method, fixing something that isn't quite right but doesn't break the compiler, and or adding a general reminder to either research something further or double-check whether you've added all the required elements to a view.

In Swift, you comment a single line of code by prefixing it with two forward slashes (`//`). If you then start your comment in one of the following two ways, Xcode detects it and displays it in the jump bar, which I'll explain shortly:

```
// TODO: TODO reminders should be used when you want to create a quick reminder about a piece of work you haven't done. This can be great when you're writing a large method and you want to focus on the key functionality, but you know you need to come back later and write the error checking.
```

```
// FIXME: I use FIXME mainly when transitioning code between two versions of iOS. When iOS 8 came out, some of my iOS 7 applications developed small code glitches. I pinpointed these glitches in one go, adding FIXME comments to the errors of concern so that I could work through them one by one, checking them off.
```

Because this is something you want to look at later on, let's use the `TODO` mark to set a reminder. After the line `super.viewDidLoad()`; add your `TODO` comment so the method looks like this:

```
override func viewDidLoad() {
    super.viewDidLoad()

    //TODO: Remove if not used
}
```

Note These code words are case sensitive. If you use any lowercase characters or fail to use the colon correctly, they won't display in the jump bar.

That's it: you've added a handy reference that will stand out in the jump bar like a sore thumb and remind you to tidy your code when you finish the application.

Using the Jump Bar

I've mentioned the jump bar several times, so let's take a closer look at it. The *jump bar* is the series of items at the top of the code view, as shown in Figure 10-15. It's called the jump bar because, depending on which part you choose, it allows you to jump quickly between files, folders, and different areas of a code file.



Figure 10-15. The jump bar is located at the top of the code window

The very last block of the jump bar, which in Figure 10-15 says `viewDidLoad()`, is by far the most commonly used part. For many developers, this is the jump bar, and it's the only part of it they ever use. Select this block of the jump bar, and you should see the effect of the added TODO comment, as shown in Figure 10-16. Although there are many comments in the code, the only one that appears in the jump view among the methods is the cleanup message.

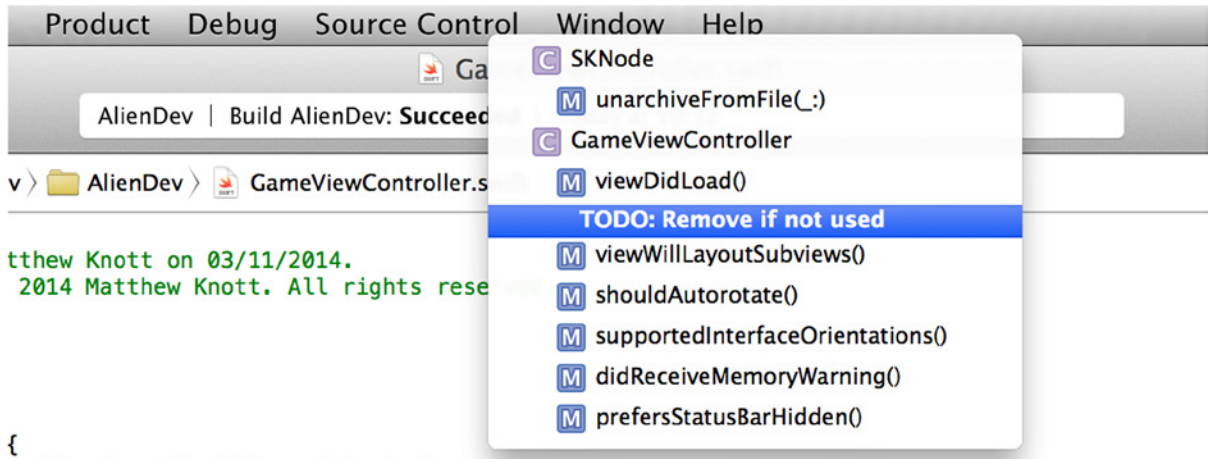


Figure 10-16. The structure of the code outlined in the jump view

Notice in the jump bar that each of the methods is listed, as well as the class implementation. Selecting any of these takes you directly to the relevant portion of code.

Organizing Code with Pragma Marks

If you have any experience programming with a C-based language, you're probably familiar with the `#pragma` directive. Traditionally it's used to provide additional information to the compiler in addition to what the language itself can express. You can also provide additional information to the IDE by using the `#pragma mark` directive to create sections for the methods in the jump bar. In Swift, the pragma mark has been streamlined to fit the same format as the other bookmarking tags and is now written `// MARK:`, as you may have noticed in previous chapters.

Let's say you want to isolate the bottom three methods in the `GameViewController.swift` file. You can add a directive to indicate that these methods are not to be touched. Before the override `func shouldAutorotate()` method, type the following code:

```
//MARK: Standard Methods : Ignore
```

Now go back and look at what the jump view shows for the code file. As shown in Figure 10-17, the mark has been added to the hierarchy and now provides a heading for the bottom three methods.

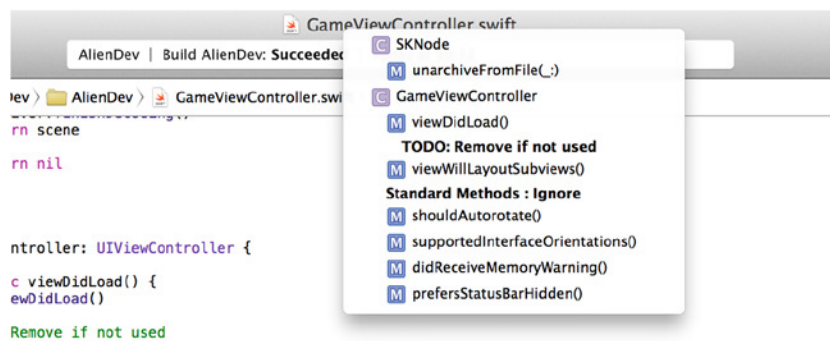


Figure 10-17. The mark appearing in the jump bar to isolate the standard methods

These marks are incredibly useful, especially when you're working with dozens of methods. Grouping your methods together by area of function makes your life easier and also that of anyone else who needs to look at your code.

Building the Scene

Unlike other applications you've built so far in this book, a Sprite Kit application doesn't need much work in the view controller, which initializes the environment. All the logic that controls what you see on the screen comes from the scene. By default, the Sprite Kit application comes with two custom classes: `GameViewController` and `GameScene`. Open `GameScene.swift`, and you see that it subclasses `SKScene`.

In a Sprite Kit application, the scene is responsible for calculating what is shown onscreen in each frame. Let's modify `GameScene` to add the alien dev hero to the screen, before swamping him with bugs:

1. If you haven't already, open `GameScene.swift`. Create an `SKSpriteNode` variable called `alienDev` to hold all the information for the alien dev's character in the scene. Add the following highlighted code:

```
import SpriteKit

class GameScene: SKScene {

    var alienDev : SKSpriteNode?
```

2. Let's clean up this file a little. Scroll down, and remove the `touchesBegan` method.
3. Remove all the code in the `didMoveToView` method. The entire file should now look like this:

```
import SpriteKit

class GameScene: SKScene {

    var alienDev : SKSpriteNode?

    override func didMoveToView(view: SKView) {
        /* Setup your scene here */
    }

    override func update(currentTime: CTimeInterval) {
        /* Called before each frame is rendered */
    }
}
```

4. Let's set the scene for the application by choosing the background color and initializing and adding the `alienDev` object. The `SKScene` object is always the root node in the hierarchy of Sprite Kit nodes; so you need to add the sprites to the scene to begin creating the hierarchy. In the `didMoveToView` method, add the following highlighted code:

```
override func didMoveToView(view: SKView) {
    /* Setup your scene here */
    self.backgroundColor = SKColor.whiteColor()
    alienDev = SKSpriteNode(imageNamed: "dev")
    alienDev!.position = CGPointMake(CGRectGetMidX(self.frame), CGRectGetMidY(self.frame))
    alienDev?.size = CGSizeMake(120, 220)

    self.addChild(alienDev!)
}
```

- Run the application. If everything works as it should, the hero stands alone in the middle of the screen, as shown in Figure 10-18, along with the node and frames per second (fps) counts. These two values are great for debugging poor performance in your animation and can help to identify choke points where you may have too many sprites (nodes) on the screen at any one time. They can also help you scale your application for different devices; more capable hardware can handle more sprites without a drop in performance, whereas an older device may need to have lower-quality images and fewer sprites. When you're finished with them, you can disable them by changing the `showsFPS` and `showsNodeCount` properties of `SKView` to `NO` in `GameViewController.swift`.



Figure 10-18. The *AlienDev* app in action, with the single, static sprite

- The hero is a bit lonely on the screen, which looks quite sparse. Before you add his nemesis, the bug, let's add a title to the scene as another node. It's an instance of the `SKLabelNode` class. Although this will be the only text you add, adding text nodes to the scene is a common requirement, so let's be efficient and create a function that adds to the scene whatever text you send it. Drop down a few lines after the `didMoveToView` method, and add the following function stub:

```
func createTextNode(text: String, nodeName: String, position: CGPoint) -> SKLabelNode {
}
```

- The function returns an `SKLabelNode` object, so you need to declare and initialize an object of this class and return it. In the function, add the following lines of highlighted code:

```
func createTextNode(text: String, nodeName: String, position: CGPoint) -> SKLabelNode {
    let labelNode = SKLabelNode(fontNamed: "Futura")

    return labelNode
}
```

- Set the attributes of the `labelNode` object to specify the text, size, color, and position of the label, as well as give it a name as an identifier. Once the attributes are set, return the label. To do so, add the following highlighted code:

```
func createTextNode(text: String, nodeName: String, position: CGPoint) -> SKLabelNode {
    let labelNode = SKLabelNode(fontNamed: "Futura")
    labelNode.name = nodeName
    labelNode.text = text
    labelNode.fontSize = 30
    labelNode.fontColor = SKColor.blackColor()
    labelNode.position = position
    return labelNode
}
```

Note Sprite Kit applications don't use `UIKit` like the other solutions have, so in this instance you use `SKColor` to set the color rather than `UIColor`. They're separate classes in separate frameworks but perform largely the same function with almost identical syntax.

- Now that you have a function for generating labels, let's give the application a title. Go back to the `didMoveToView` method and, after the line `self.addChild(alienDev!)` add the following code:

```
self.addChild(alienDev!)

let title = createTextNode("Welcome to Alien Dev",
    nodeName: "titleNode",
    position: CGPointMake(CGRectGetMidX(self.frame), CGRectGetMaxY(self.frame)-150))
self.addChild(title)
```

- Run the application again. This time it looks a little less sparse, as you can see in Figure 10-19.

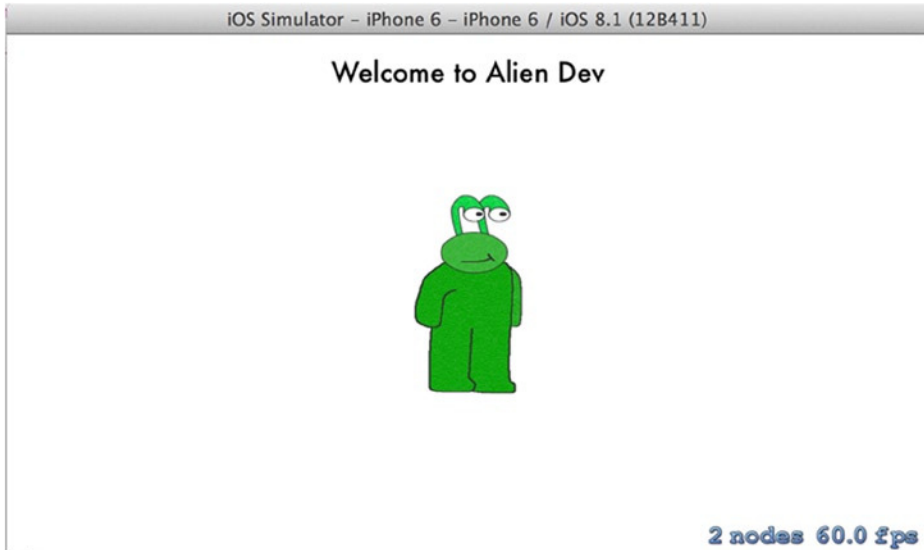


Figure 10-19. The application, now with two nodes: the dev and the title

11. You've got everything in this scene except the bugs that plague the hero. In this application, the bugs appear from the top of the screen and slide down toward the hero—the alien dev. To achieve this, let's create a function that adds a bug at a random position offscreen. This is called by the SKScene update method at regular intervals. Add the stub for this function beneath the createTextNode function:

```
func createBug() {
}
```

12. This will be a particularly large function. As in previous chapters, I won't dwell too much on the actual code—just how Xcode ultimately makes it easy to manage and interact with this code. In pseudo code, let's create an SKSpriteNode as you did when you initialized the dev character. You set the bug to appear just offscreen above the top of the screen, but then randomly determine where it "spawns" on the x (horizontal) axis. After adding the bug to the scene, you indicate how long it should appear and then send it down the screen before it disappears and is removed from the scene. Add the following highlighted code in the createBug function:

```
func createBug() {
    let evilBug = SKSpriteNode(imageNamed: "bug")
    evilBug.size = CGSizeMake(220, 120)
}
```



```

let minX = (evilBug.size.width / 2)
let maxX = (self.frame.size.width - evilBug.size.width)
let rangeX : UInt32 = UInt32(maxX - minX)

let finalX = Int(arc4random() % rangeX) + Int(minX)

evilBug.position = CGPointMake(CGFloat(finalX),
    self.frame.size.height + evilBug.size.height/2)
self.addChild(evilBug)

let minDuration : Int = 3
let maxDuration : Int = 8
let rangeDuration : UInt32 = UInt32(maxDuration - minDuration)

let finalDuration = Int(arc4random() % rangeDuration) + minDuration

let actionMove = SKAction.moveTo(CGPointMake(CGFloat(finalX),
    -evilBug.size.height/2), duration:NSTimeInterval(finalDuration))
let actionMoveDone = SKAction.removeFromParent()

evilBug.runAction(SKAction.sequence([actionMove, actionMoveDone]))
}

```

Folding Code

The `GameScene.swift` file is really starting to fill out now, but there are still two methods and some instance variables to add so that you can see the application in all its glory. Xcode kindly provides a way to make this file easier to navigate and modify, in the shape of *code folding*.

Code folding is the concept of compressing code that is encapsulated by brackets—such as in methods, `if` statements, and other logical structures—into a single line, thus hiding the code from view and allowing you to focus on a specific segment of code. You can fold code in the editor as well as through menus and key combinations. Code folding isn't unique to Xcode—the concept exists in many popular IDEs and code editors—but it's not as obvious how to access it in Xcode as it is in those other systems.

First, let's look at code folding and how it's done in the editor. Scroll to the top of the `GameScene.swift` file, to the start of the `didMoveToView` method. Move your mouse cursor to the gutter next to the code just adjacent to the start of the method, as shown in Figure 10-20.

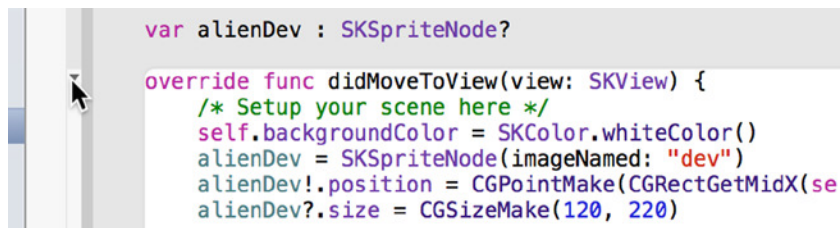


Figure 10-20. Exposing the fold toggle in the code editor

Notice that a downward-pointing arrow appears: Xcode highlights all the code that will be hidden if you click the mouse on that area of the gutter. Click the arrow, and the entire method is compressed, as shown in Figure 10-21. As you can see, the code shrinks to a single line, greatly reducing the visual clutter. There is now a right-pointing arrow in the gutter: click it, and your method is restored.



```
class GameScene: SKScene {
    var alienDev : SKSpriteNode?
    override func didMoveToView(view: SKView) {...}
    func createTextNode(text: String, nodeName: String, position: CGPoint) -> SKLabelNode {
        let labelNode = SKLabelNode(fontNamed: "Futura")
        labelNode.name = nodeName
    }
}
```

Figure 10-21. The “folded” `didMoveToView` method

Folding one method at a time can be time consuming, which is where the menu options come into play. From the menu bar, select Editor ► Code Folding. As you can see in Figure 10-22, Xcode gives you a number of options for folding the code in this file: Fold, Unfold, and Unfold All. In addition, Xcode separates itself from other, lesser IDEs by giving you the option to specifically Fold Methods & Functions or Fold Comment Blocks. This fine level of control is really satisfying for developers who like to fold methods but not comments, rather than folding everything that could possibly be folded.

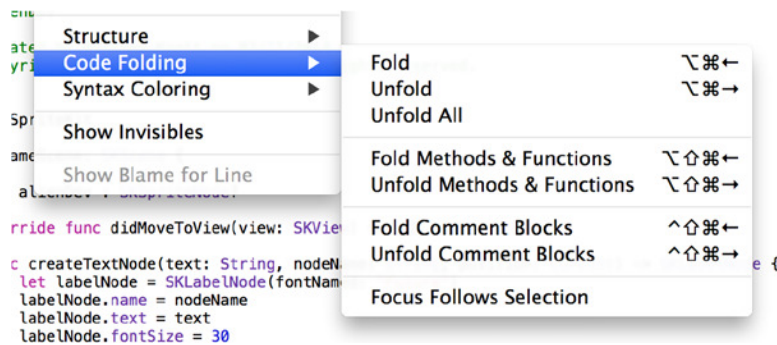


Figure 10-22. Options offered in Xcode’s Editor tab

Select the Fold Methods & Functions option. Instantly, the implementation is compressed from 60 lines to just 14, as shown in Figure 10-23!

```

import SpriteKit

class GameScene: SKScene {
    var alienDev : SKSpriteNode?

    override func didMoveToView(view: SKView) { ... }

    func createTextNode(text: String, nodeName: String, position: CGPoint) -> SKLabelNode { ... }

    func createBug() { ... }

    override func update(currentTime: CTimeInterval) { ... }
}

```

Figure 10-23. The neatly folded implementation file

Now that there is some room to breathe in the file, you can add the remaining two methods and the instance variables. This code is provided by Apple as part of its Sprite Kit adventure template, and it allows you to call the `createBug` method at regular intervals. At the top of `GameScene.swift`, add the two instance variables as follows:

```

class GameScene: SKScene {

    var alienDev : SKSpriteNode?
    var lastSpawnTimeInterval : CTimeInterval?
    var lastUpdateTimeInterval : CTimeInterval?

```

These variables are used to calculate the time elapsed between frames. Next, you need to create one function stub and use an existing one to handle the updates: `updateWithTimeSinceLastUpdate` and `update`. The `update` method is a class method that is called each frame, and `updateWithTimeSinceLastUpdate` is a custom function that ensures that bugs are added at a constant rate. Add the `updateWithTimeSinceLastUpdate` function stub before the `update` method in the implementation file, as shown in the following highlighted code:

```

func updateWithTimeSinceLastUpdate(timeSinceLast : CTimeInterval) {
}

```

```

override func update(currentTime: CTimeInterval) {

```

In the `update` method, add this highlighted code, which accurately calculates the elapsed time and calls the custom method:

```

override func update(currentTime: CTimeInterval) {

    if let lastUpdate = lastUpdateTimeInterval {

        var timeSinceLast = currentTime - lastUpdate as CTimeInterval

```

```

    lastUpdateTimeInterval = currentTime
    if (timeSinceLast > 1) {
        timeSinceLast = 1.0 / 60.0
        lastUpdateTimeInterval = currentTime
    }

    updateTimeSinceLastUpdate(timeSinceLast)
}
else
{
    lastUpdateTimeInterval = currentTime
}
}

```

For the last method in this implementation file, let's look at another weapon in the efficient developer's arsenal: code snippets.

The Code Snippet Library

The Code Snippet library is a collection of small pieces of code, like microtemplates, that allow you to quickly create commonly written blocks of code by simply dragging and dropping the code into the code editor. The concept of code snippets, like code folding, is not unique to Xcode; but like code folding, it's implemented in a clear and intuitive manner. The Code Snippet library, shown in Figure 10-24, is located in the utilities bar and is accessed by clicking the { } icon (Control+⌘+⌘+2).

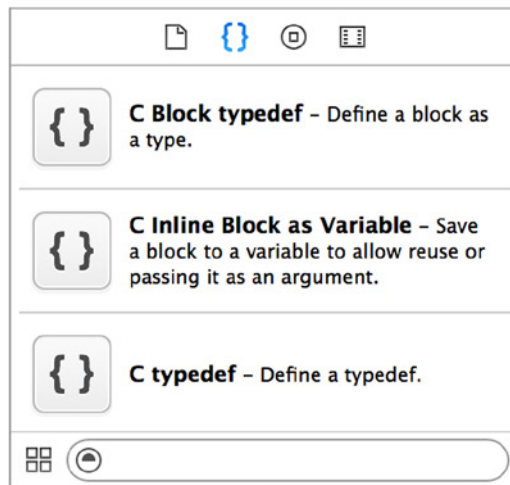


Figure 10-24. The Code Snippet library

Using Code Snippets

I'll explain how to create an `if` statement using a code snippet, but before you can do that, you need to add a line of code to the `updateWithTimeSinceLastUpdate` function. The following highlighted line updates the `lastSpawnTimeInterval` object:

```
func updateWithTimeSinceLastUpdate(timeSinceLast : CTimeInterval) {
    if let lastSpawn = lastSpawnTimeInterval {
        lastSpawnTimeInterval! += timeSinceLast
    }
    else
    {
        lastSpawnTimeInterval = 0
    }
}
```

You want to spawn a bug every 1 second, so you need an `if` statement to check that the `lastSpawnTimeInterval` object, which counts milliseconds, is greater than 1. Look in the Code Snippet library for `If Statement`, or type `if state` in the filter box, as shown in Figure 10-25.

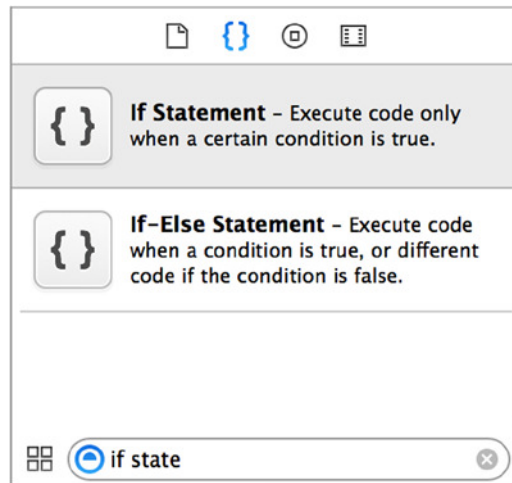


Figure 10-25. Filtering the Code Snippet library for an `if` statement

Create an empty line under the last line of code you added, and then drag the `if` statement from the Code Snippet library. Position it just below that last line of code, as shown in Figure 10-26.

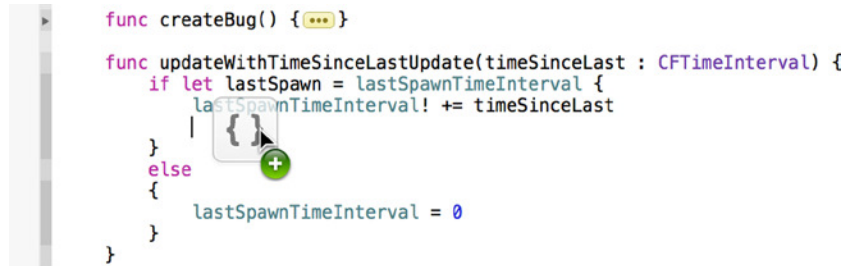


Figure 10-26. Dragging an `if` statement from the Code Snippet library

When you release the snippet, it creates the outline of the `if` statement exactly as it does when you use the code-completion method for creating `if` statements. Change the conditions placeholder to say `lastSpawnTimeInterval! > 1` and the statements to `lastSpawnTimeInterval! = 0` and, below that, `createBug()`.

The finished method code looks like this:

```

func updateWithTimeSinceLastUpdate(timeSinceLast : CTimeInterval) {
    if let lastSpawn = lastSpawnTimeInterval {
        lastSpawnTimeInterval! += timeSinceLast
        if (lastSpawnTimeInterval > 1 ) {
            lastSpawnTimeInterval = 0
            createBug()
        }
    }
    else
    {
        lastSpawnTimeInterval = 0
    }
}

```

Tip If your code is a bit messy after dragging in the code snippet, highlight the whole function and go to Editor ► Structure ► Re-Indent (Ctrl+I) to fix the indenting.


Using code snippets to create an `if` statement isn't the most efficient way to use them, but look through the list of snippets—there are dozens of premade snippets for a wide range of scenarios.

Creating Code Snippets

Where code snippets come into their own is when you create them from your own code. You know better than anyone else the code you type time and again—whether it’s a template for a web request or a pattern you use for error handling—so creating code snippets is a great way to simplify code reuse.

Now for the paradox of creating code snippets—it’s as easy as drag and drop, but drag and drop isn’t necessarily easy. To explain, you create a code snippet by highlighting the code you wish to save and dragging it into the Code Snippet library. Unfortunately, if you try this, you probably find you just end up selecting different code when you try to drag; this is where the art of dragging code in Xcode comes into play.

Let’s say you want to save the last two methods you created as they’re the best way to perform a Sprite Kit action at a regular interval. Start by highlighting the code, as shown in Figure 10-27.



```
func updateWithTimeSinceLastUpdate(timeSinceLast : CTimeInterval) {
    if let lastSpawn = lastSpawnTimeInterval {
        lastSpawnTimeInterval! += timeSinceLast
        if (lastSpawnTimeInterval > 1) {
            lastSpawnTimeInterval = 0
            createBug()
        }
    }
    else
    {
        lastSpawnTimeInterval = 0
    }
}

override func update(currentTime: CTimeInterval) {
    if let lastUpdate = lastUpdateTimeInterval {
        var timeSinceLast = currentTime - lastUpdate as CTimeInterval

        lastUpdateTimeInterval = currentTime
        if (timeSinceLast > 1) {
            timeSinceLast = 1.0 / 60.0
            lastUpdateTimeInterval = currentTime
        }

        updateWithTimeSinceLastUpdate(timeSinceLast)
    }
    else
    {
        lastUpdateTimeInterval = currentTime
    }
}
```

Figure 10-27. The highlighted code that you want to create a snippet from

Now, the technique—click and *hold* the mouse pointer in place until it changes from an I bar to a normal mouse cursor and drag the code to the Code Snippet library, as shown in Figure 10-28.

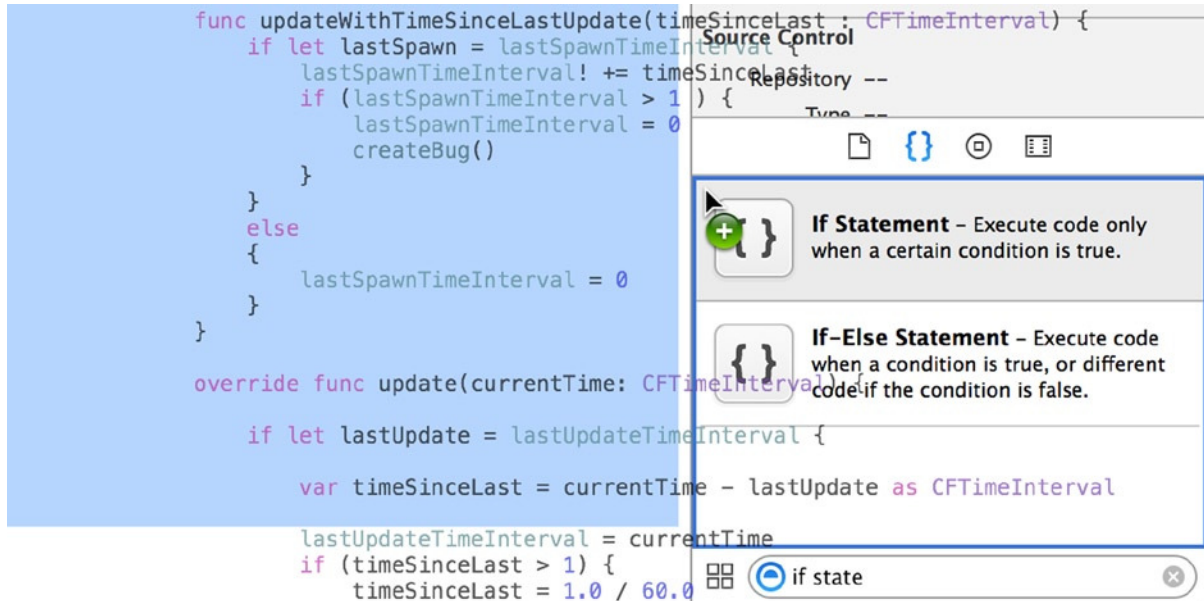


Figure 10-28. Dragging the code to the Code Snippet library

When you release the mouse, it will create an entry in the library called My Code Snippet. To save it from getting lost, it's important to name the snippet and set its attributes before you forget why you created it. Double-click the snippet to see a preview of the code, as shown in Figure 10-29; click the Edit button.

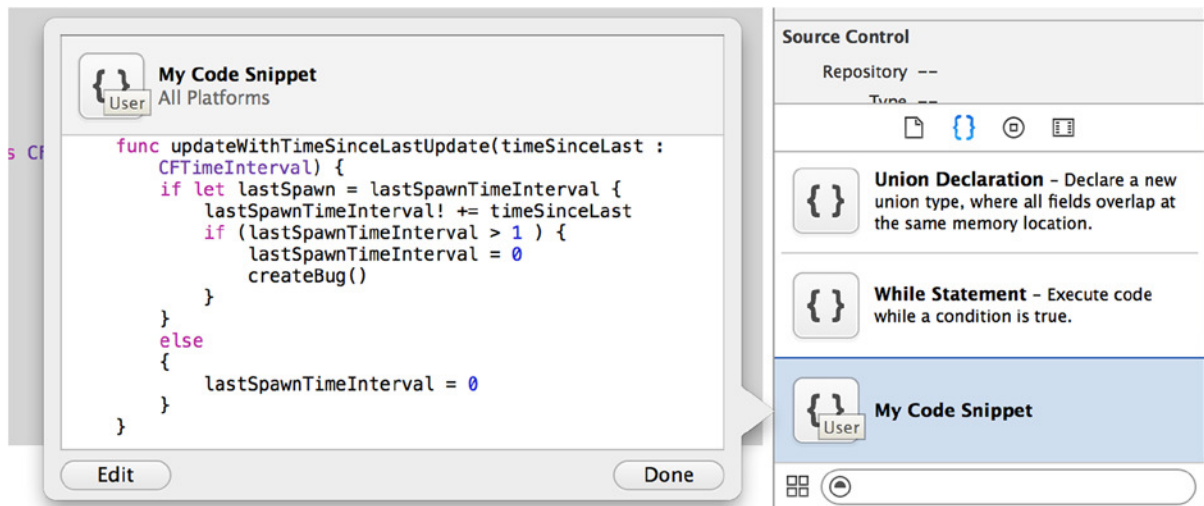


Figure 10-29. Previewing the code snippet you created

You can see that you can now edit the entire code snippet here, but what you want to do is set the Title to SKScene Update Functions, the Platform to iOS, and the Completion Shortcut to SKU, as shown in Figure 10-30, then click Done.

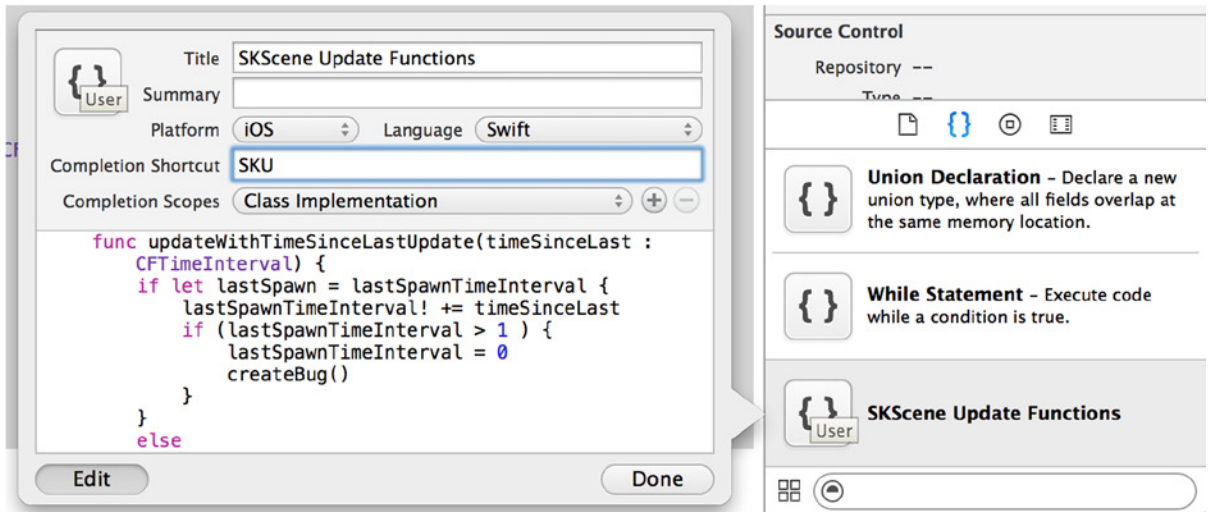


Figure 10-30. Setting the properties of the snippet

You can see as soon as the Title property is set, it's updated in the library. One important value you set is the Completion Shortcut. I mentioned earlier that an if statement isn't the most efficient use of a code snippet, but in fact the drag and drop of the code snippet wasn't efficient because you've learned that when you type if in the code editor, you can press the Tab key to complete the entire statement, which is faster than dragging the snippet. All that actually happens is when you type if, it sees that you have a snippet in your library with a completion shortcut of if.

Go to your code editor and after the last method, type SKU. You see that you can now quickly create the two update methods in your snippet simply by typing that completion shortcut, as shown in Figure 10-31.

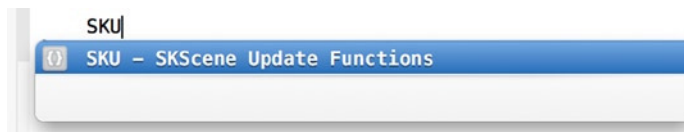


Figure 10-31. Quickly adding the snippet to the implementation file

It's this feature that exemplifies how Xcode has many of the same features you find in other IDEs, but Apple has taken the concept and refined it to make the developer's life so much easier. All that remains now is to run the application and watch the alien dev being bombarded by bugs, as shown in Figure 10-32.

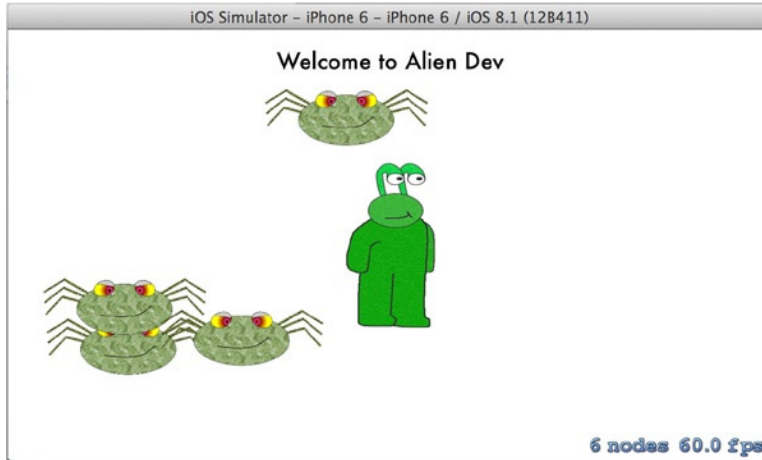


Figure 10-32. The completed AlienDev application

Summary

This chapter used a Sprite Kit application as a backdrop to show the numerous ways Xcode can help reduce the overhead of writing code by streamlining how you code and by fine-tuning the development environment to suit your personal preferences.

Specifically in this chapter, you've learned:

- How to select a theme for the code editor
- How to customize a theme, share it, and import third-party themes
- Which keywords to use in code comments and when to use them
- How to use the jump bar to quickly navigate to a block of code or a comment
- How to fold blocks of code to allow you to focus on specific areas of the code
- How to create and use code snippets to allow you to reuse code

The next chapter will take a detailed look at how to debug issues in applications and how Xcode can give insight into what's going on behind the scenes.

Debugging and Analysis

Chapter 10 looked at some of the ways Xcode can empower you to be a more effective and efficient developer, from tweaking the theme used to display code in the code editor, to using the jump bar and code snippets to speed up development. You learned all that by creating a Sprite Kit–based animation application where the hero, the alien dev, had bugs raining down on him from above.

This leads nicely into this chapter, where you learn about debugging and analysis and, hopefully, answer the question, “How can Xcode help when the bugs start raining down?” Xcode has a whole suite of tools dedicated to making your life easier when it comes to determining why your code throws an exception or why in some cases nothing happens at all. This chapter takes a detailed look at all of these reasons. Additionally, I discuss some of the lesser-known debugging tools that aren’t integrated into Xcode but are essential as you explore the breadth of the features of iOS application development with Xcode.

At the end of the chapter, I introduce you to one of the major new features of Xcode 6 for Swift programmers: the Swift playground. This a code sandbox lets you focus on trying different pieces of logic and sampling the outcome.

This chapter initially focuses on three common debugging scenarios and how Xcode can be used to address them:

- *Logic errors*: Sometimes the hardest to debug, logic errors occur when your application doesn’t do what you expect it to do, but they don’t cause your application to trigger an exception or warning at either compile time or runtime. An example would be a button that doesn’t do anything or a map view not displaying the specified area.
- *Runtime errors*: A runtime error is one that is detected after the application has compiled and it’s either launching or running. Unhandled runtime errors are usually fatal to the application and cause it to crash.
- *Compile-time errors*: When you tell Xcode to run or build your application, it uses the compiler to take all of your code, linked files, and libraries and compile them into a binary. A compile-time error stops your application from compiling into a binary, so it must be resolved before you can run the application.

This chapter explains how to create an application that lists some of the European Union (abbreviated to EU throughout this chapter) member states in a table view calls EUStates. The table view uses an array as its data source, which is traditionally a great way of demonstrating runtime and logic errors because of their precise nature (they have a set number of items known as the *bounds* of the array, and going outside of those bounds can trigger a runtime or logic error).

Building the Application

EUStates is a very simple application to build. The focus here is on how to use Xcode to debug an application, so the code is minimal. To create the EUStates application, let's start with a Single View Application template and add a Table view controller. Many of these steps will be familiar to you from when you created the Twitter client in Chapters 7 and 8:

1. Open Xcode and create a new project by going to File ► New ► New Project (⌘+Shift+N) or, alternatively, choosing Create A New Xcode Project on the Welcome screen (⌘+Shift+1).
2. Select the Single View Application Template, and click Next.
3. Name the product EUStates, substitute your personal information for mine, ensure that Device is set to iPhone, and leave the other options set to their defaults, as shown in Figure 11-1. Click Next.

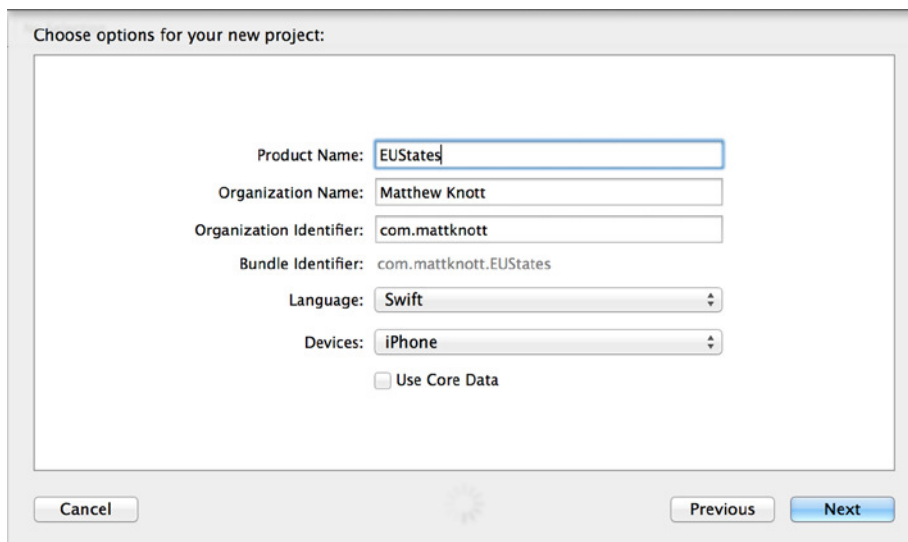


Figure 11-1. Setting up the application

4. The default save location is okay, so create the application by clicking Create.
5. Open `Main.storyboard` from the Project Navigator. It resents a single view on the storyboard; select the view and remove it so that you're left with an empty storyboard, as shown in Figure 11-2.

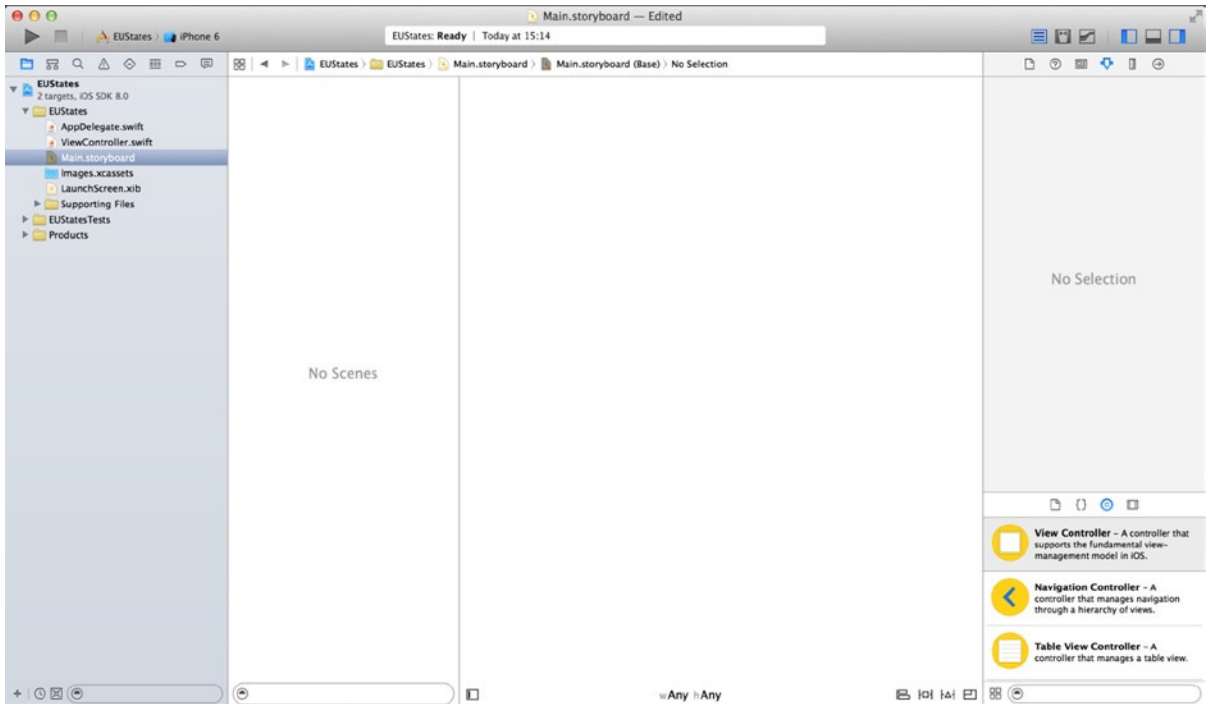


Figure 11-2. Starting with a blank storyboard

6. To complete the initial set up of the application, select `ViewController.swift` in the Project Navigator. Remove it either by pressing the Backspace key or by right-clicking the file and selecting Delete and then Move to Trash when prompted.
7. The application you're creating is based around a single Table view controller. That means you need to subclass `UITableViewController` to create a custom view controller and then tie that into a Table view controller on the storyboard. Select `File > New > File (⌘+N)`, choose Cocoa Touch Class from the list of templates, and click Next.
8. Set the Subclass Of value to `UITableViewController` and the Class value to `StatesViewController`. Leave Targeted For iPad and With XIB For User Interface unselected, and click Next. Accept the default location to save the new class files, and click Create.

- Let's add the Table view controller to the storyboard. Open `Main.storyboard` from the Project Navigator. Open the Object Library, and locate the Table View Controller object near the top of the list.
- Drag a Table view controller onto the storyboard's design area, as shown in Figure 11-3.

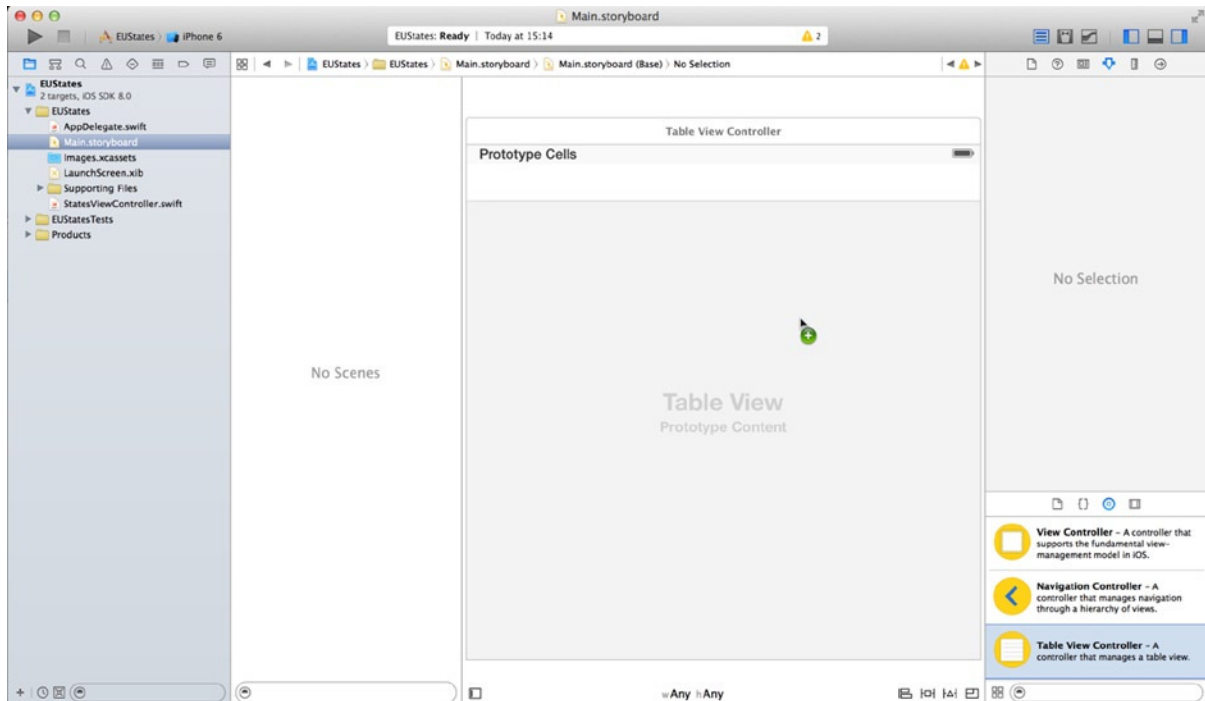


Figure 11-3. Dragging a Table view controller onto the storyboard

- You need to specify the class for the new Table view controller. If the Table view controller isn't select it, do so, and then open the Identity Inspector. Change the Class value to `StatesViewController`, as shown in Figure 11-4.

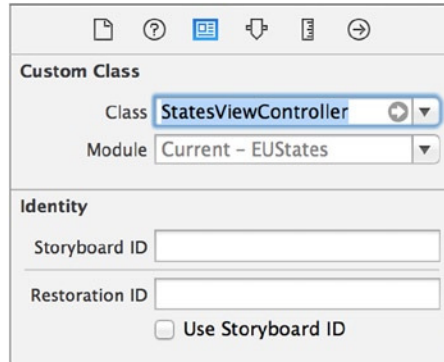


Figure 11-4. Setting the Class value for the Table view controller to `StatesViewController`

- Click the Attributes Inspector. If it isn't selected, check Is Initial View Controller as shown in Figure 11-5.

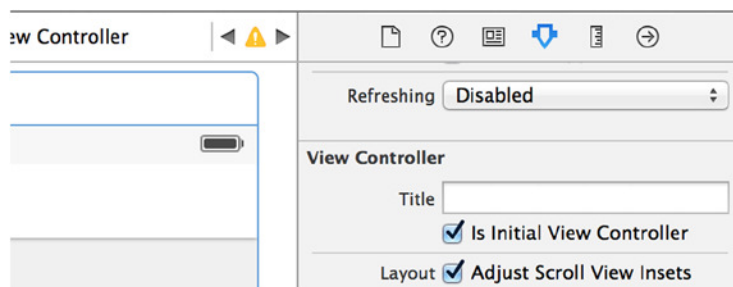


Figure 11-5. Specifying the initial view for the storyboard

- With the structure of the application complete, it's time to create the array to hold 15 EU member states. This is done by creating an `NSMutableArray` as an instance variable and then populating it in a custom method. Open `StatesViewController.swift` from the Project Navigator. To declare the instance variable, add the highlighted code to the file as shown next:

```
class StatesViewController: UITableViewController {
    let states = NSMutableArray(capacity: 15)
}
```

- You need to create a method to initialize the array and populate it with 15 EU member states. After the `viewDidLoad` method, to create a new function called `initStates`, drop down a few lines and add the following stub:

```
func initStates() {
}
}
```

15. To populate `NSMutableArray`, you add string values for the first 15 member states of the EU to the array. Add the highlighted code to the `initStates` function:

```
func initStates() {
    states[0] = "Austria"
    states[1] = "Belgium"
    states[2] = "Bulgaria"
    states[3] = "Croatia"
    states[4] = "Cyprus"
    states[5] = "Czech Republic"
    states[6] = "Denmark"
    states[7] = "Estonia"
    states[8] = "Finland"
    states[9] = "France"
    states[10] = "Germany"
    states[11] = "Greece"
    states[12] = "Hungary"
    states[13] = "Ireland"
    states[14] = "Italy"
}
```

16. Now that you have a data source, all that remains is to use the `states` array to populate the table view. You may recall from Chapter 8 that three methods need to be altered: `numberOfSectionsInTableView`, `numberOfRowsInSection`, and `cellForRowAtIndexPath`. All of these methods sit beneath `// MARK: - Table view data source`. Click the jump bar, and you see that this mark is used to neatly separate the table view methods from the rest of the view controller. Select `numberOfSectionsInTableView(_:)`, as shown in Figure 11-6.

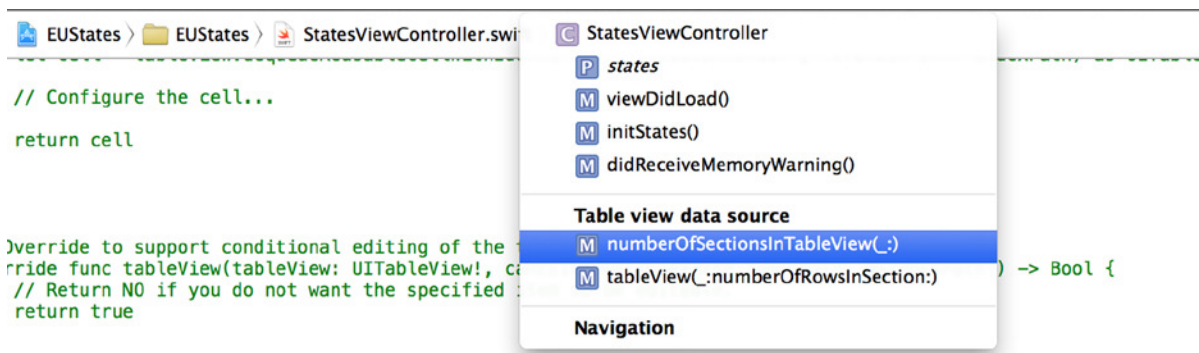


Figure 11-6. Selecting the `numberOfSectionsInTableView` method from the jump bar

Note If you aren't running the latest version of Xcode, you may see some warnings. If so, click the warning: Xcode automatically fixes the issue by removing erroneous exclamation marks from the method declaration.

17. Change the return value from 0 to 1, as shown next, and feel free to remove the comments:

```
override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
    return 1
}
```

18. Move down to the `numberOfRowsInSection` method, which dictates how many rows to render in the table view's section. Because you want this to be the number of elements in your array, you simply return `states.count`. The finished method should now resemble the following code:

```
override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    return states.count
}
```

19. The final part of this implementation is to set the text of the table cell to the value of the corresponding index in the array. You use a string formatted to append the index number onto the table cell so that each cell appears like this: *2: Bulgaria*. Go to the `cellForRowAtIndexPath` method, which is currently commented out, and remove the `/*` from the start and `*/` from the end of the method.
20. Remove any exclamation marks from the method declaration that may cause an error, and also remove any comments from the method. Then add the highlighted code:

```
override func tableView(tableView: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCellWithIdentifier("StateCell",
        forIndexPath: indexPath) as UITableViewCell

    let cellText = "\(indexPath.row): \({states[indexPath.row]})"
    cell.textLabel.text = cellText

    return cell
}
```

That's the last of the code you need to write for this application. Now you can run the application in the Simulator. Unfortunately, when you do, not a lot happens, as shown in Figure 11-7. All the methods are set correctly, so where's the problem? To get to the bottom of this conundrum, you need to see what's happening behind the scenes; and to do that, you need to use breakpoints.

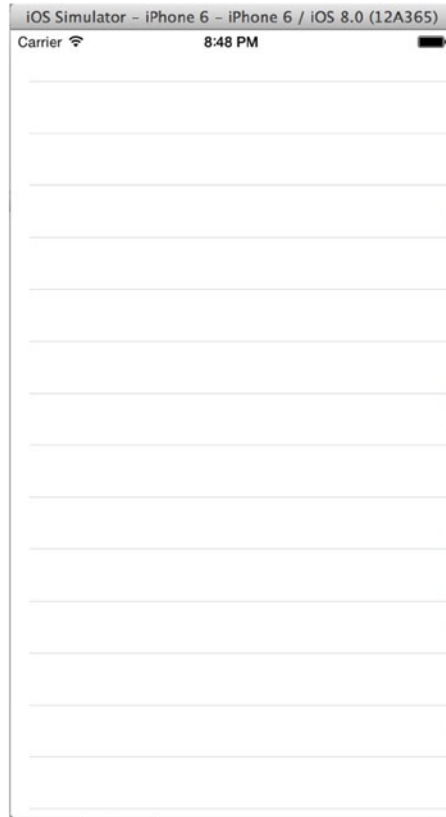


Figure 11-7. The application running in the Simulator, but not doing very much

Using Breakpoints to Resolve Logical Errors

You've come across the first issue with the application. You need to use Xcode's debugging tools to analyze the problem and understand what has gone wrong and what you should do to fix it. The first tool in a developer's arsenal is the humble breakpoint.

Breakpoints allow you to specify a point in the application at which to pause the execution of the code and see what's happening behind the scenes. This way, you can associate a specific event with the line of code at fault. When a breakpoint is reached and the application pauses, you get to see the state of all of your objects and what values they contain. The application checks the `numberOfSectionsInTableView` and `numberOfRowsInSection` methods to set the parameters for the table view, so these are good places to start the analysis. Because the `numberOfSectionsInTableView` method returns a fixed value, let's concentrate on the `numberOfRowsInSection` method first.

Setting a Breakpoint

If you've used other development environments in the past, you may find that adding a breakpoint in Xcode is very familiar; even if you haven't, you'll find that the process is intuitive. Scroll down to the `numberOfRowsInSection` method, and click in the grey bar to the left of the method declaration to place a breakpoint, as shown in Figure 11-8; if you have line numbers turned on, click the line number. When you add a breakpoint, a dark blue arrow appears on top of the line number.

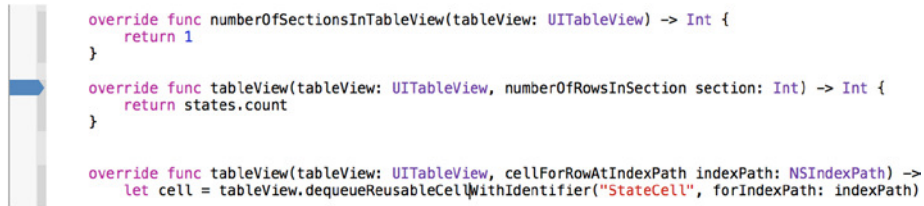


Figure 11-8. The breakpoint added for a line of code

During runtime, when this line of code is reached, the application will pause and, hopefully, provide insight into why there isn't any information in the table view. Try running the application now. It launches in the Simulator, and then you're dropped back to Xcode with the Debug Navigator (Figure 11-9) showing the state of the application and the debug area (Figure 11-10) showing a selection of objects that are relevant to where the application has paused.

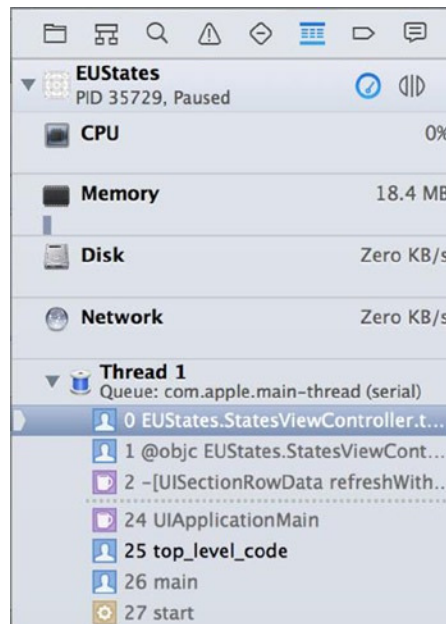


Figure 11-9. The Debug Navigator, showing key information about the application at this point in time

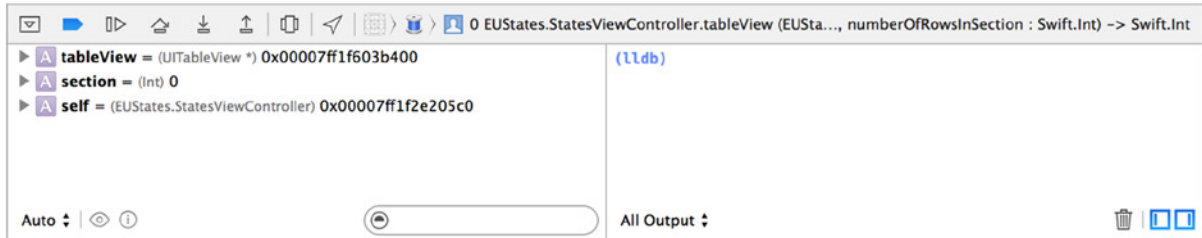


Figure 11-10. The debug area and, on the left, a selection of objects relevant in the context of the breakpoint

Before you look at why the application isn't working, let's examine both of these key areas in some detail to see what information they provide.

The Debug Navigator

While the application is paused, the Debug Navigator, as shown in Figure 11-11, can be used to provide a live snapshot of the activities taking place in the application. It displays the application's performance in terms of resource usage and the call stack of each thread in the thread list.

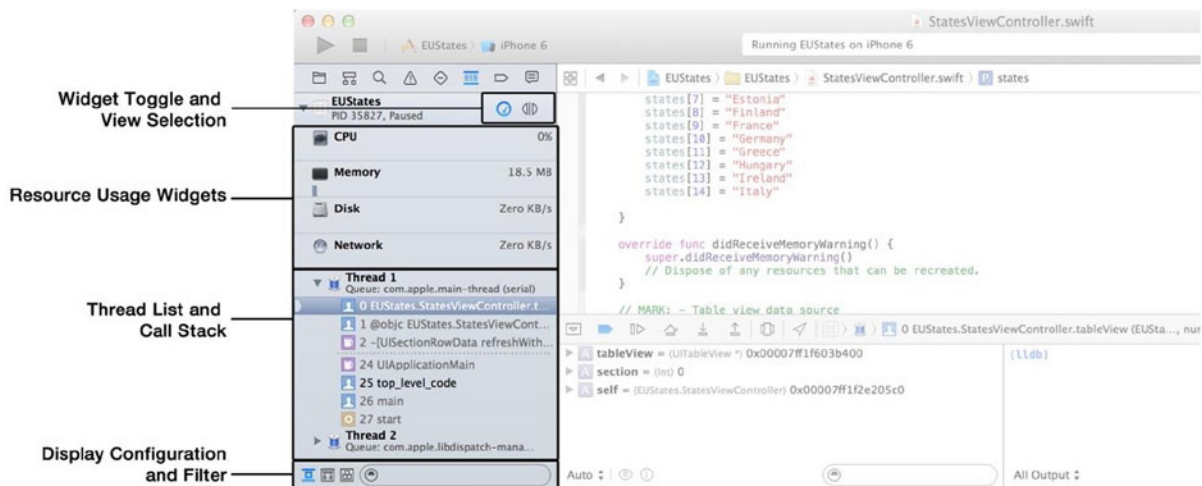


Figure 11-11. The three key areas of the Debug Navigator, shown while debugging an application

The resource usage is the only portion of the Debug Navigator that appears while the application is running as well as when it's paused. When you're developing for iOS, memory management is essential, but the task of ensuring that objects are correctly disposed of has been automated to a large extent with the introduction of automatic reference counting (ARC) in iOS 5. Before Xcode 5, ARC could be turned on or off depending on your preference, but with Xcode 5 it ceased to be presented as an option and is enabled by default for all iOS application templates. ARC basically automates the disposal of objects when they're no longer needed, handling almost all memory management and therefore giving you a lot of insurance against memory leaks.

The thread list and the call stack in each thread show which methods are currently being executed by each thread. You can see from the expanded call stack for the main thread (thread 1) in Figure 11-12 that the currently executed method is `tableView:numberOfRowsInSection:`, which was called by `[UISectionRowData refreshWithSection:tableView:tableViewRowData:]`. This can make understanding the flow of your application much easier.

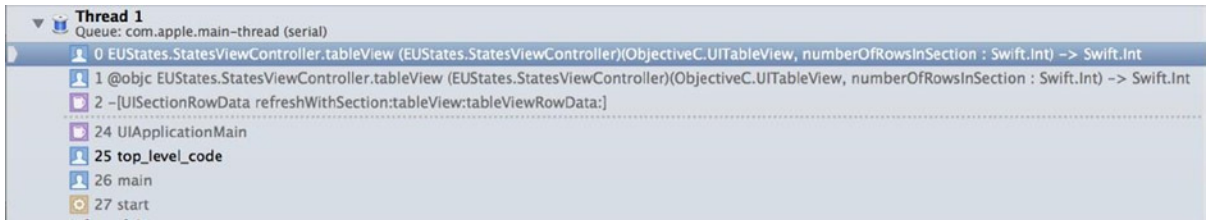


Figure 11-12. A detailed look at the call stack of the main thread

An icon adjacent to each thread reflects the thread's status:

- *No icon / blue icon:* The thread is running normally.
- *Yellow icon:* The thread is being blocked and is waiting for another thread to be unlocked or a certain condition to be met.
- *Red icon:* The thread has been suspended. While suspended, the thread doesn't execute any code when you proceed from the breakpoint.

The Debug Area

The debug area was covered in some detail in Chapter 3, so I won't go over old ground. But unlike in Chapter 3, the debug area has now come to life so you can interact with some of the controls and features you learned about earlier in the book.

Referring back to Figure 11-10 and your own instances of Xcode use for a moment, notice that the main output of the debug area at this time is the list of four variables that are relevant in the context of the method you're paused on. Figure 11-13 shows the control bar for the debug area. I explained what each icon means in Chapter 3, but at that time you didn't necessarily have the ability to see what they do. As you go through the rest of this chapter, feel free to click the buttons to see their effect and then stop and rerun the application to get back to the breakpoint.



Figure 11-13. The control bar for the debug area

The buttons that appear in the debug area, listed left to right, are as follows:

- *Hide*: Show or hide the debug area.
- *Breakpoint toggle*: Currently dark blue; toggles all breakpoints between enabled and disabled.
- *Continue*: Resumes execution of the application and then becomes a pause button.
- *Step Over*: Allows you to move out of the function being executed onto the next instruction.
- *Step Into*: Allows you to see in intricate detail each step of the application's execution. Think of the application as a film that's been paused: each time you click Step Into, you move a single frame ahead.
- *Step Out*: Shows the next piece of code to be executed. Use this if you've stepped too deep into your application's inner workings.
- *Debug View Hierarchy*: A new feature in Xcode 6 that helps you understand where elements of your view may be sitting, either off-screen or on. Sometimes a constraint may seem sensible enough when you add it but can have unexpected consequences when your application runs; this tool is invaluable in understanding what has happened.
- *Simulate Location*: Explained later in the chapter.

In addition to these controls, the debug area has its own jump bar. This jump bar shows the threads in the application and, in turn, each thread's call stack, as shown in Figure 11-14.



Figure 11-14. The debug area's jump bar

While trying to solve the current predicament of a table view with no content, you essentially have two initial lines of investigation: the data source (is there anything for the table to display?) and the application logic (did you correctly pass the information to the table view?). Looking at the objects in the debug area as shown back in Figure 11-10, you have an item called `self`. Click the disclosure indicator triangle next to it, and beneath `self` you see an array called `states`. Looking at the information about `states`, it should be obvious where the fault lies: the `states` array, instead of saying “15 values,” says “0 values,” meaning it hasn't been populated. Before you investigate why the array hasn't been populated, there are a few more things to cover in relation to breakpoints: most important, the Breakpoint Navigator.

The Breakpoint Navigator

At this point you've only created a single breakpoint, which is fine for working through the current problem, but in day-to-day development you'll come across various issues that call for a range of different breakpoints. You'll often set breakpoints on key pieces of logic and turn them on or off as required when flowing through an application. To manage these numerous breakpoints, you need to take advantage of the Breakpoint Navigator (⌘+7), which is the seventh icon in the list of navigators and resembles the shape of a breakpoint indicator.

Figure 11-15 shows the Breakpoint Navigator as it stands for this application. It gives you access to all the breakpoints in the project from a single location, and you can also use it to create an array of special breakpoints from the + symbol at the bottom of the navigator.

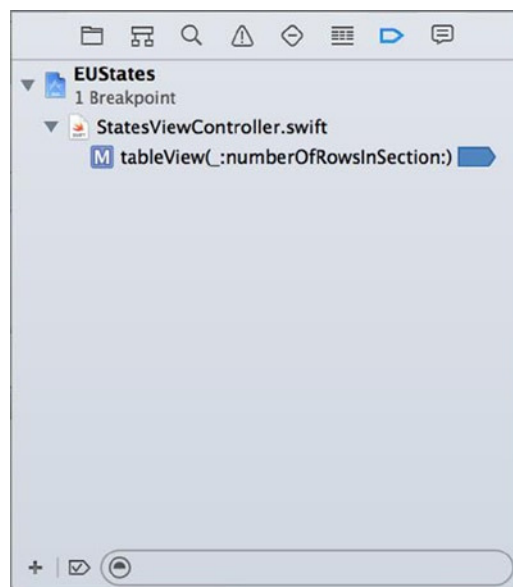


Figure 11-15. The Breakpoint Navigator

In addition to a standard breakpoint, Xcode lets you add the following breakpoints, which are used in specific scenarios:

- *Exception breakpoint:* Triggers when any Objective-C or C++ exception is thrown or when a specific C++ exception is thrown.
- *Symbolic breakpoint:* Triggers when a specific method is triggered, which can be refined to a specific method in a specific class or even a specific function.
- *OpenGL ES error breakpoint:* Used when creating OpenGL ES-based applications (mainly games). As with standard breakpoints, these can be configured to be conditional, as I explain shortly.
- *Test failure breakpoint:* Triggers when a unit-test assertion fails, giving you an even greater level of granular analysis during testing.

Xcode also gives you a large degree of control over the behavior of your breakpoints. Right-click the breakpoint, as shown in Figure 11-16.

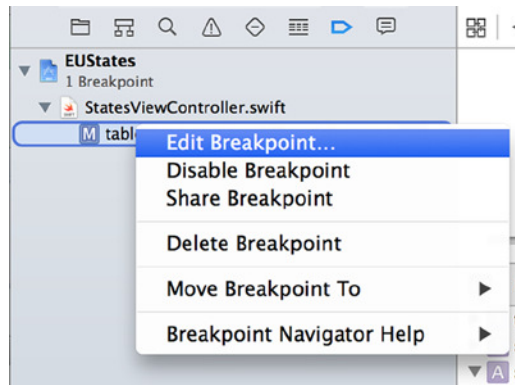


Figure 11-16. The contextual menu displayed when you right-click a breakpoint in the Breakpoint Navigator

You can use this menu to *edit* the attributes of the breakpoint; you can also *disable* or *delete* the breakpoint. Sharing a breakpoint makes it available to other users of the same project. This menu also continues Xcode's great support for contextual help, providing numerous useful and relevant pages on the Breakpoint Navigator and its features. Here, you focus on the edit ability.

Click Edit Breakpoint, and you're presented with a pop-over dialog, as shown in Figure 11-17. This dialog first shows the file and line number where the breakpoint has been added, and then provides several properties that can be set for the breakpoint:

- **Condition:** A specific programming condition that must exist for the breakpoint to trigger, such as the incremental number in a for loop being equal to 10.
- **Ignore:** The number of times the condition needs to be met before it triggers a pause.
- **Action:** The action menu exemplifies Xcode's flexibility as an IDE. When the condition of the breakpoint is met, Xcode can perform any number of combinations of the following actions:
 - Execute a piece of AppleScript
 - Capture an OpenGL frame
 - Issue a debugger command
 - Log a message to the console (or have it spoken to you!)
 - Run a shell command
 - Play a sound
- **Options:** Although plural, offers only one option, which is to continue after the breakpoint has triggered. This may seem counterintuitive, but if you just want the actions you add to be executed and the program to continue, this saves you from having to manually resume the application.

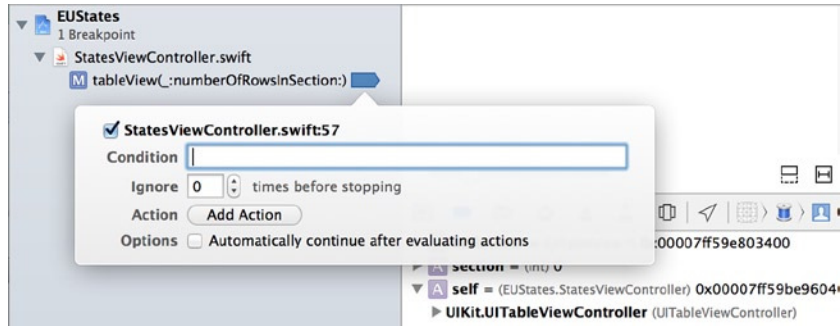


Figure 11-17. Editing a breakpoint

Conditional Breakpoints

Conditional breakpoints only trigger when certain conditions are met. In this case, it would be great if even after you fix the current issue, the debugger would notify you any time you get to a breakpoint and the states array is in a null state. Let's make the breakpoint conditional and add some humor to it at the same time (yes, breakpoints can be fun):

1. Right-click the breakpoint in the Breakpoint Navigator, and choose Edit Breakpoint.
2. In the Condition box, type **states.count == 0**.
3. Click Add Action, and choose Log Message from the list.
4. Set the Message value to **I've fallen and I can't get up at %B**.
5. Change the radio button selection from Log Message To Console to Speak Message. Your finished breakpoint should resemble that shown in Figure 11-18.

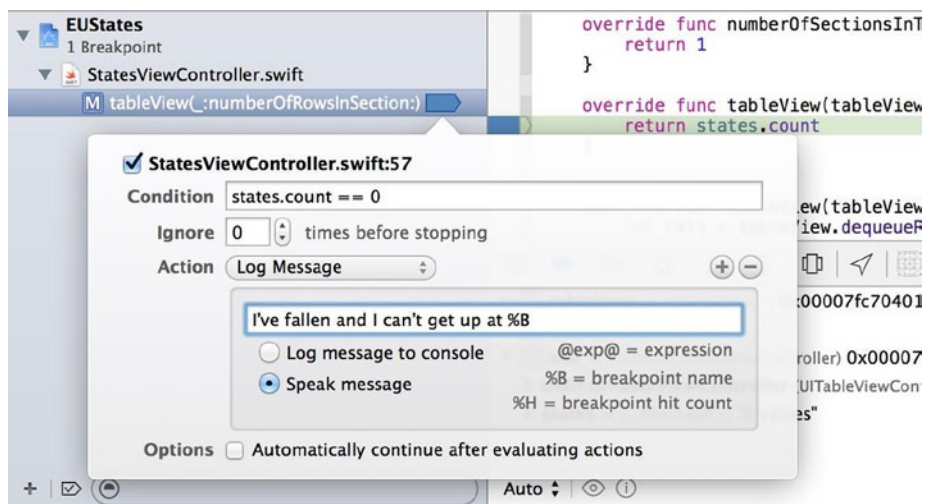


Figure 11-18. The customized breakpoint with a little added humor

Rerun the application, and you're greeted by a synthesized voice saying, "I've fallen and I can't get up at table view `_:numberOfRowsInSection`." Although this is amusing, the issue with the application—that the array is empty—still has not been resolved. The fact that the array is empty means that the `initStates` method isn't being called before it tries to build the table. The real issue is that the method isn't called *at all*.

Switch back to the Project Navigator, and open `StatesViewController.swift`. Go to the `viewDidLoad` method, and add the highlighted code to the method:

```
override func viewDidLoad() {
    super.viewDidLoad()

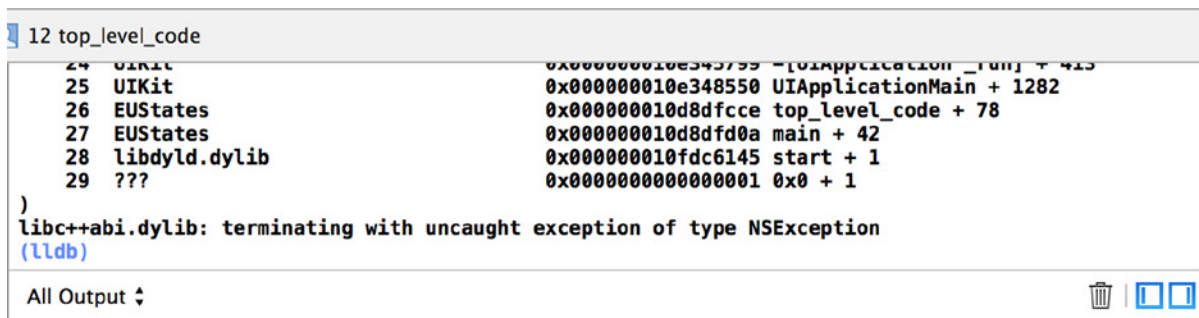
    initStates()
}
```

Now rerun the application. It runs, and you're dropped back to Xcode. This isn't because of the breakpoint—as you should have noticed, there was no voice expressing its need for assistance—but rather because of a runtime error.

Runtime Errors

Although logic errors are frustrating, runtime errors are far more destructive to your application and your reputation. Outside of your IDE, a runtime error will cause your application to crash, which will annoy your users no end and harm your reputation as a developer of bulletproof software. Because of the risk associated with software glitches, it's essential to thoroughly test your software for robustness.

The question here is, how do you address a runtime error? It's often one of the hardest errors to debug, but there is a wealth of information to help you get to the bottom of the issue. When the application crashed, it dumped the exception details and the call stack into the console in the debug area, as shown in Figure 11-19.



```
12 top_level_code
24 UIKit                                0x000000010e343799 -[UIApplication _run] + 415
25 UIKit                                0x000000010e348550 UIApplicationMain + 1282
26 EUStates                             0x000000010d8dfcce top_level_code + 78
27 EUStates                             0x000000010d8dfd0a main + 42
28 libdyld.dylib                        0x000000010fdc6145 start + 1
29 ???                                  0x0000000000000001 0x0 + 1
)
libc++abi.dylib: terminating with uncaught exception of type NSRangeException
(lldb)
```

All Output ↓

Figure 11-19. The console in the debug area after a runtime error has occurred

When you first look at this mass of detail, it can be daunting, but it's actually incredibly useful. Starting at the end of the message, you can see that the application threw a standard `NSException`. This isn't particularly helpful at this point. Then you have the call stack in reverse order, meaning you need to scroll up through the console to get to the point of failure. As you scroll up, look at item number 4, as shown next:

```
4 UIKit 0x000000010e432084 -[UITableView dequeueReusableCellWithIdentifier:forIndexPath:] + 153
```

The call stack is like the black-box flight recorder on an aircraft: just like a real black box, it gives a detailed log of what happened leading up to a crash. Item 4 is significant because this is the last event that happened before Xcode started reporting the calls to the exception handlers in items 3 to 0, which can largely be discarded.

Item 4 is significant because it shows a call to the `UITableView` class's `dequeueReusableCellWithIdentifier:forIndexPath:` method. If you open `StatesViewController.swift` from the Project Navigator and scroll down to the `cellForRowAtIndexPath` method, you see that this is called in the first line of the method.

Using Exception Breakpoints

To confirm your suspicions about this line, you can create an exception breakpoint to confirm the true source of the exception. Open the Breakpoint Navigator, click the + symbol at the bottom of the navigator, and then select `Add Exception Breakpoint`, as shown in Figure 11-20.

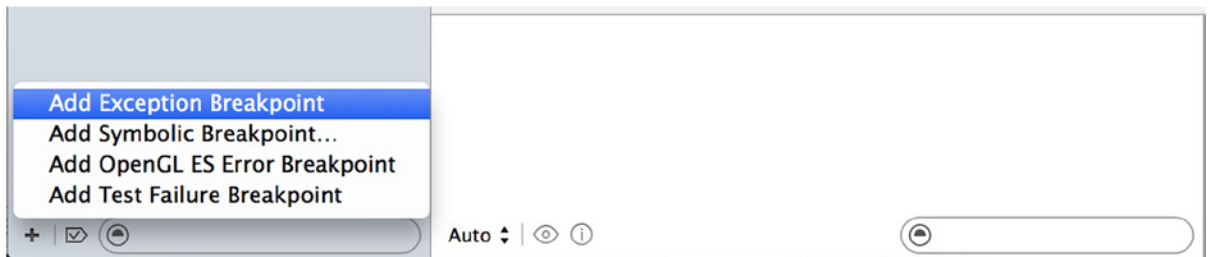


Figure 11-20. Adding an exception breakpoint

A new breakpoint to capture all exceptions is added to the list of breakpoints, as shown in Figure 11-21. All that remains is to run the application and see what Xcode can tell you about this particular issue.

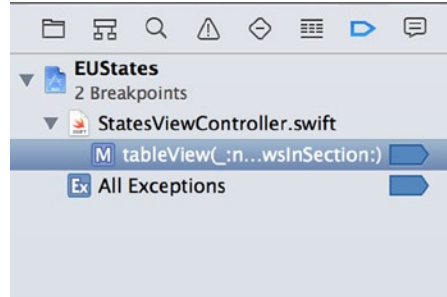


Figure 11-21. A breakpoint to capture all exceptions has been added to the Breakpoint Navigator

Run the application again. Immediately the new breakpoint intercepts the exception and confirms your suspicions about the exception's source, which is indeed the call to the `dequeueReusableCellWithIdentifier:forIndexPath:` method, as shown in Figure 11-22.

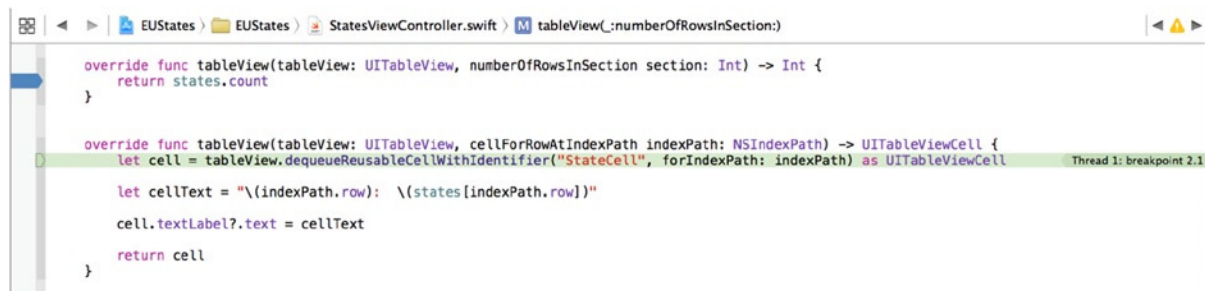


Figure 11-22. The exception breakpoint pinpointing the source of the exception

So, now that you know the source of the exception, how do you resolve it? It isn't always obvious, but in this case, Xcode gives you all the answers needed to get the application up, running, and error free. What's more, the solution has been there since the initial exception.

Click the Breakpoints toggle button in the debug area to disable all the breakpoints, and then rerun the application. It crashes back to Xcode and again provides a mass of detail about the crash. This time, scroll up through the details in the console until you reach the top and see the following line:

```
2014-10-28 21:12:00.089 EUStates[38309:2650763] *** Terminating app due to uncaught exception
'NSInternalInconsistencyException', reason: 'unable to dequeue a cell with identifier StateCell -
must register a nib or a class for the identifier or connect a prototype cell in a storyboard'
```

Roughly translated, this line says that you told the `dequeueReusableCellWithIdentifier` method to dequeue a cell called `StateCell`, which indeed you did in the first line of the `cellForRowAtIndexPath` method:

```
let cell = tableView.dequeueReusableCellWithIdentifier("StateCell",
    forIndexPath: indexPath) as UITableViewCell
```

It's saying that you haven't set up a prototype cell in the storyboard with the identifier of `StateCell`, so it doesn't know what you're asking it to do. As you may recall from looking at table views in Chapter 8, you need to specify a reuse identifier for each cell you want to reference, so now you can see what happens if you don't.

To resolve this exception and move one step closer toward a working application, open `main.storyboard` from the Project Navigator. Select the prototype cell from the table view, and then open the Attributes Inspector as shown in Figure 11-23.

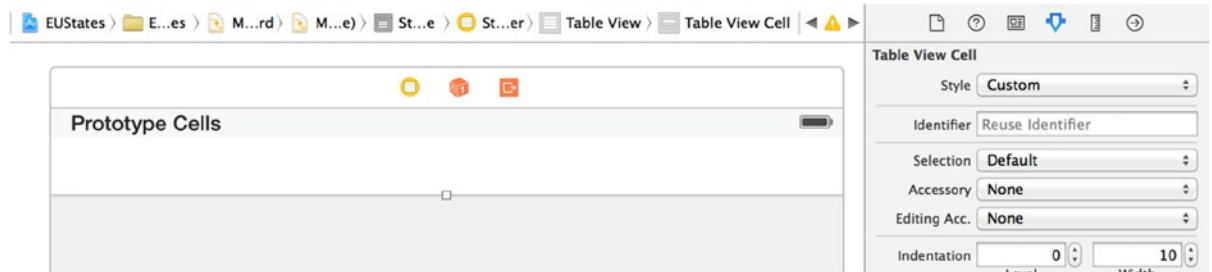


Figure 11-23. Selecting the prototype cell from the table view in the storyboard

In the Identifier attribute, match the cell identifier to the code to set the value as `StateCell`, as shown in Figure 11-24.

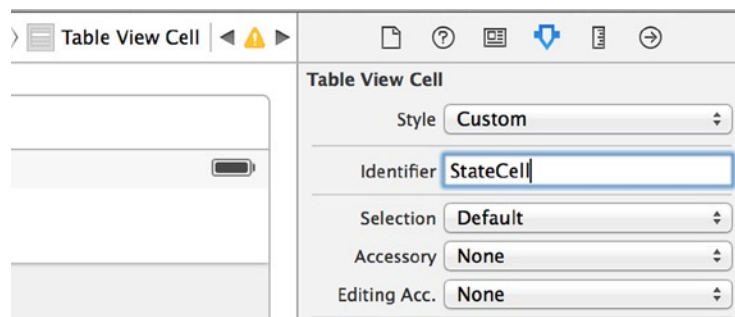


Figure 11-24. Setting the cell Identifier attribute

It's taken a while to reach this point, but it's finally time to rerun the application and see it in action. Click the Run button, and your application should run successfully, as shown in Figure 11-25.

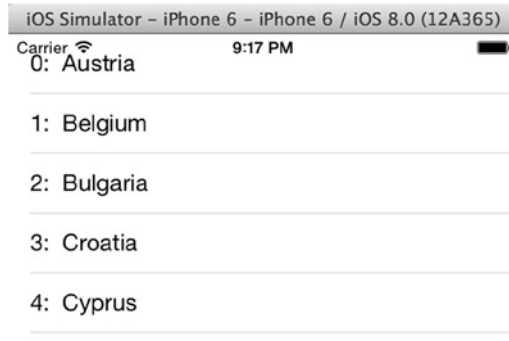


Figure 11-25. The application, finally running!

Compile-Time Errors

So far, this chapter has discussed logic errors, where the application runs but doesn't work, and runtime errors, where the application runs but then crashes. In both situations, when Xcode is initiated to run the application, the compiler is happy that all the code's syntax is correct, and it compiles the application.

A compile-time error occurs when the compiler is processing the code and encounters an issue with either the syntax of the code or one of the linked files. The good news is that often, compile-time errors are easily overcome, and many times Xcode even helps you overcome them by suggesting solutions.

Let's start to introduce some syntax errors into the application so you can see this in action. Open `StatesViewController.swift` from the project, and scroll down to the `viewDidLoad` method. After the `super.viewDidLoad()` line of code, let's specify a background color for the table. Add the highlighted code exactly as shown—the casing of words is extremely important:

```
override func viewDidLoad() {
    super.viewDidLoad()

    self.tableView.backgroundColor = UIColor.redColor()?

    initState()
}
```

Instantly, Xcode shows you that you have made a mistake by indicating the error with a red circle next to the line, as shown in Figure 11-26.

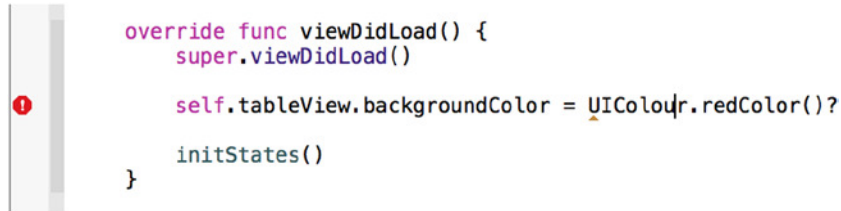


Figure 11-26. Xcode indicating an error with the syntax

Just to prove that you’ve created a compile-time error, try to run the application; it will fail and report the failure to build in the Activity Bar, as shown in Figure 11-27. Notice that the number of compile-time errors is reflected in the Activity Bar as a red circle with a white exclamation mark in it, next to the number of issues. To get an overview of the issues, click the red circle, which takes you to the Issue Navigator.



Figure 11-27. The Activity Bar reflecting the compile-time error

The Issue Navigator

This small issue serves two purposes: it shows you how Xcode spots syntax issues, and it serves as a great example that although Swift is a very comprehensive and advanced language, it has plenty of room for growth. You’ve actually created two errors in this single line of code; in Swift, Xcode identifies that there is a syntax error and shows you the line the error occurs on. If you were deliberately creating this error in Objective-C, Xcode would spot that there were two errors and offer to fix both for you. No doubt Apple will continue to develop the language throughout version 6 of Xcode and through each successive release, but for now, you have to work things out for yourself.

To get an overview of any warnings or errors in the project, use the Issue Navigator (+), as shown in Figure 11-28.

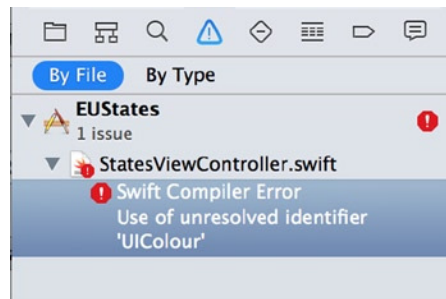


Figure 11-28. The issue listed in the Issue Navigator

As you can see, Xcode has seen the misspelled class name and is effectively saying it doesn't know what you're referring to. Go into the code editor and correct this by removing the *u* from the class name so that it says `UIColor`. Although the class name is fixed, Xcode now picks up on the other issue.

Issues that are highlighted with a white dot in the red circle in the Issue Navigator can be fixed automatically by Xcode. Select the Swift Compiler Error from the list, and you're directed to the specific point in the code that Xcode feels is in error, as shown in Figure 11-29.



Figure 11-29. The issue detail pop-over, explaining the issue and offering to fix it

As you can see, Xcode adds a pop-over explaining what the issue is, and it also gives a suggested fix. Click the Fix-it option and Xcode corrects the syntax issue. Immediately the Activity Bar loses the error indicator and the Issues Navigator says No Issues.

And that is how easy Xcode makes it to manage compile-time errors! Sometimes you will find it slightly more challenging than this, but powerful code completion coupled with a highly responsive editor and debug system largely prevent errors or allow you to resolve them the second they crop up.

Tools to Help with Debugging

So far, this chapter has looked at the debug tools that are built into Xcode, but you can also access a couple of extremely useful tools through the Simulator to help you debug your applications. Although these tools don't specifically debug anything, they allow you to test certain functions of your application in a way that can trigger an exception and therefore let you debug the error before you release your product.

In the next section, I explain the tools you can use to help debug Map Kit–based applications and applications that have a print function. Rather than write another application to demonstrate these features, let's use the default iOS Maps application.

To access Maps, first make sure you stop EUSates, which returns the Simulator to the home screen—but not the first page. To get to the first page, open the Simulator and then use the mouse to slide the pages across until you reach the page with the Maps icon on it, as shown in Figure 11-30.

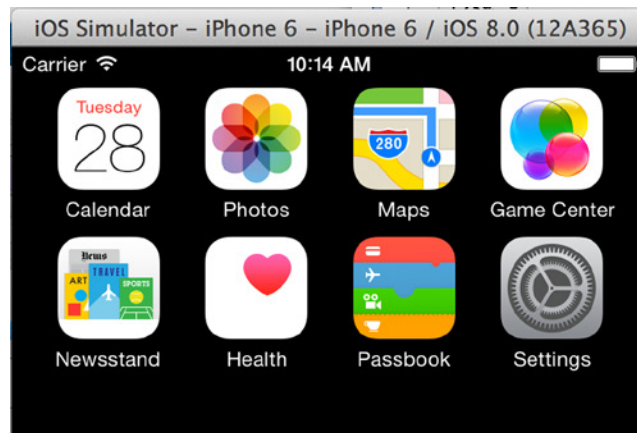


Figure 11-30. The main icons all appear on the first page in the Simulator

Run the Maps application from the Simulator. Depending on previous usage of both the Simulator and Xcode, you should see a view resembling that in Figure 11-31 a map with no user location and no movement.



Figure 11-31. The default starting point in the Maps application

Debugging a Location

The challenge when developing a Map Kit–based application is your location. Unless you’re commuting, using a physical device won’t really help you develop your application because you want to try the application in different locations—perhaps even at different speeds, such as when driving or cycling—and using a MacBook while cycling isn’t advisable!

Because of this conundrum, Apple introduced location simulation in Xcode and the Simulator, which lets you use preset locations and scenarios or create your own. As shown in Figure 11-32, you can debug the location by selecting **Debug** ► **Location** and then selecting **Apple**. Almost immediately, a blue circle appears over California. Clicking the arrow in the bottom-left corner of the Maps application zooms you to Apple headquarters in Cupertino.

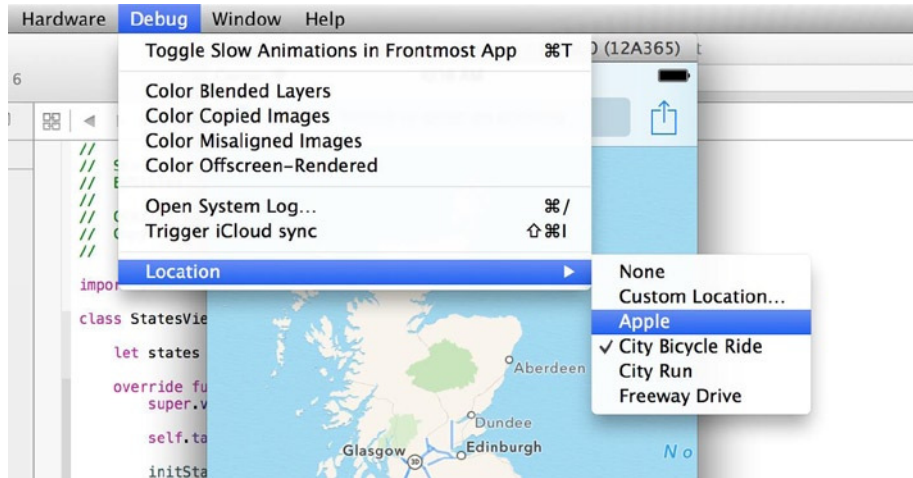


Figure 11-32. Accessing the options for debugging location from the Simulator

Next, return to the Debug ► Location menu and select Freeway Drive. The simulated position starts speeding around the Junipero Serra freeway, which is a great way to test an application that measures speed or distance.

Note If nothing happens, click the location arrow in the bottom-left corner of the Maps application. It will request access to your location and then begin tracking.

To simulate a specific location, return to the menu and choose Custom Location. From here you can specify a longitude and latitude as the user's location. Specify a longitude of 51.62228 and a latitude of -3.943491 to put the user in the middle of the city of Swansea. Simulating location is great for applications that use routing or that perhaps make recommendations based on a location.

Print Debugging with the Printer Simulator

Adding print functionality to your application is a great way to enrich its capabilities. Although the ability to print from in an application isn't overly complicated, you need to be able to test the actual print functionality. If you don't have a printer that supports Air Print, the technology for printing from an iOS device, fear not, because the Printer Simulator can solve all your worries.

Unfortunately, the Printer Simulator is no longer bundled with Xcode and must be downloaded from the Apple web site. In Xcode, go to Xcode ► Open Developer Tool ► More Developer Tools, which prompts you to sign in with your Apple developer credentials.

Once you have done this, look for the newest release of Hardware IO Tools for Xcode; the releases are in reverse chronological order, so the newest version is near the top. Click the item, and, on the right, you'll see a .dmg file for download, as shown in Figure 11-33. Note that I am using Xcode 6.0.1 final release, not the 6.1 beta.

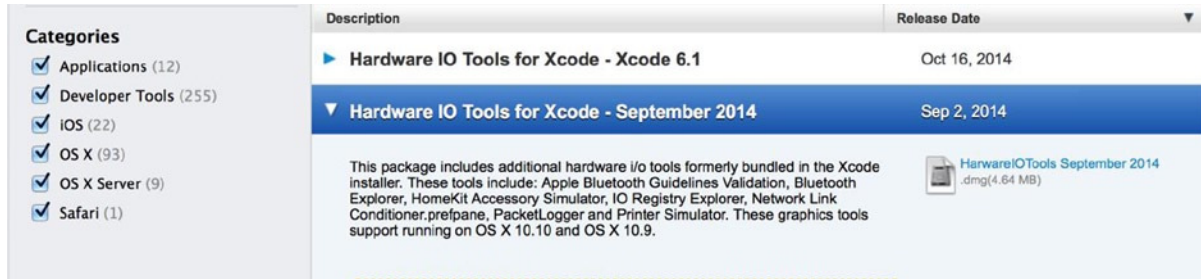


Figure 11-33. Downloading the Hardware IO Tools for Xcode

Download and then open the .dmg file. Open it, and then open the Printer Simulator. It launches as a simple console and reports that it has set up several types of printers, as shown in Figure 11-34.

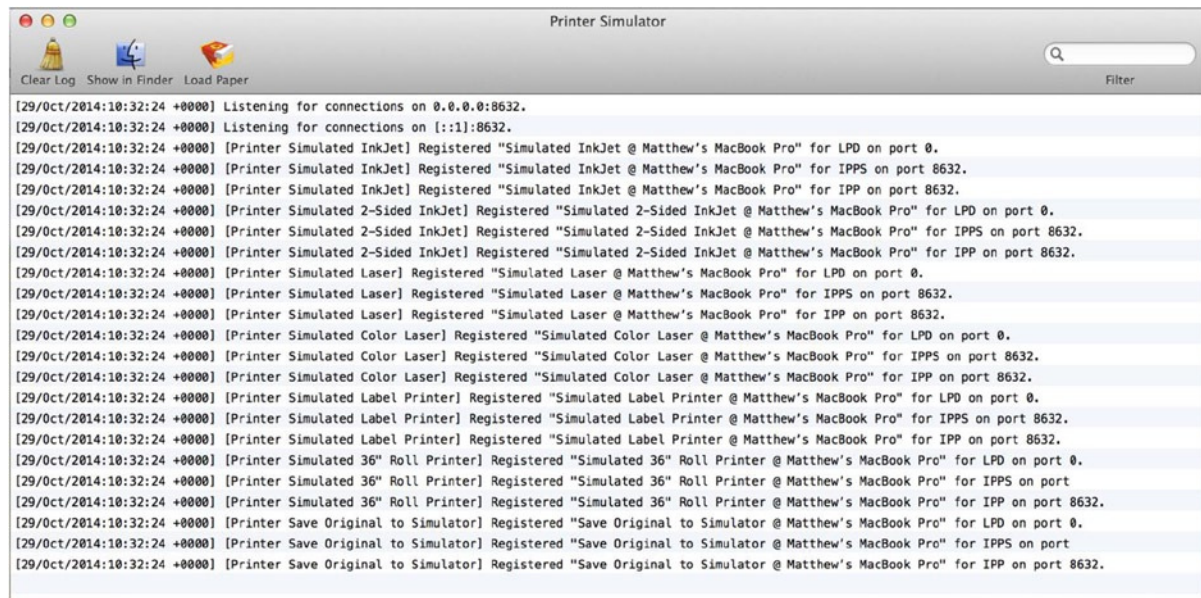


Figure 11-34. The Printer Simulator running and providing a selection of printers

The Simulator may not be what you expected, but it's extremely powerful and can simulate an incredible number of scenarios. To see an example of its power, click the Load Paper icon from the toolbar. A dialog slides down, showing all six of the simulated printers, with the options of customizing their functions and paper sizes. Dismiss the dialog by clicking OK.

To use one of these virtual printers, return to the home screen in the Simulator and choose the Photos application. It should load with four sample images in the library, as shown in Figure 11-35.

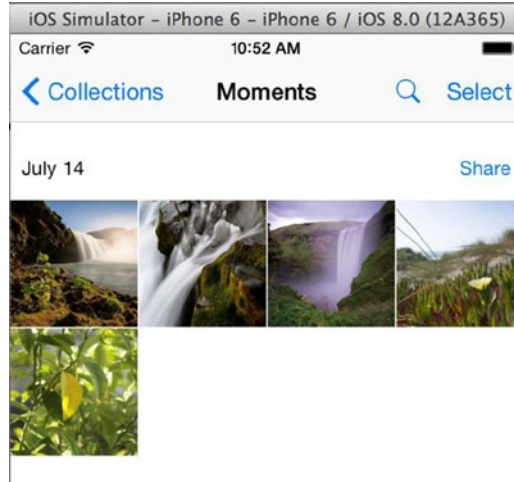


Figure 11-35. The Photos app with default photos

Open a photo, and click the Share icon in the bottom-left corner. The sharing action sheet appears, as shown in Figure 11-36.

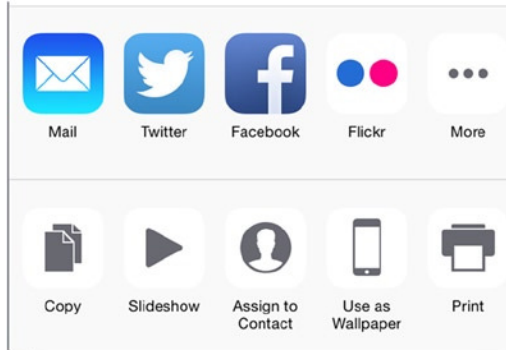


Figure 11-36. The iOS sharing menu

From this action sheet, select the Print option. You're presented with the standard iOS print dialog, as shown in Figure 11-37.



Figure 11-37. The iOS printer options

Choose the first option to select a printer. You're then presented with a list of simulated printers; choose Simulated Inkjet, and click the Print button. After a brief pause, the Printer Simulator springs to life. Details about the print job appear in the console; then the Preview application appears, displaying the results of your print job, as shown in Figure 11-38.

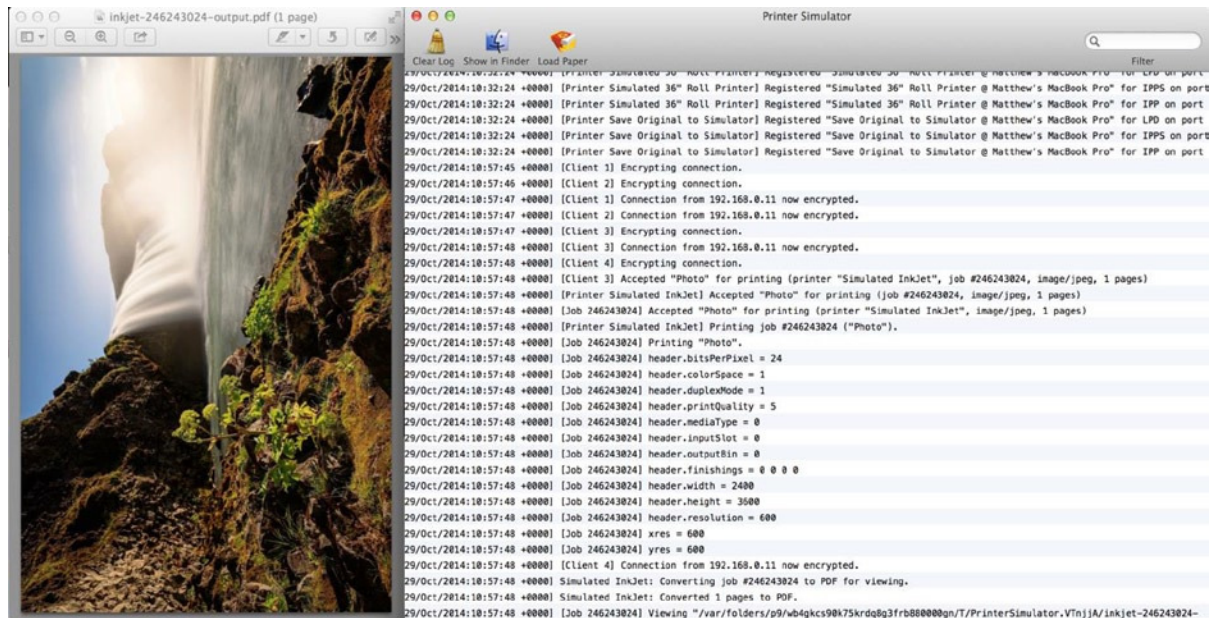


Figure 11-38. The output of the simulated print job

Setting up printing for your application is often done blind, with no visual indication of the output until you print, so being able to quickly test this functionality and debug any issues (such as missing pages) is invaluable. The depth offered by the Printer Simulator means that you can test print functionality on a range of device types at different resolutions and have a high degree of confidence in your finished product—and you also aren't contributing to the erosion of the rainforests with reams of test prints.

Playground

Accompanying the announcement of the Swift programming language was the Playground feature of Xcode 6. A *playground* is effectively a code sandbox or scratchpad where you can drop in, write some Swift code, and experiment with it outside the confines of your application.

This can be great for learning the language, or even for more seasoned developers who want to perfect a regular expression, for example, or see a visual representation of the curve they're calculating. Playgrounds can be saved and shared among friends and colleagues, meaning you no longer have to rely on code snippets or large projects as ways of sharing code—you can also share playgrounds. And because Xcode is free, playgrounds surely lend themselves to computer science in schools as a great way of writing a piece of logic without having to build an interface or make full use of an IDE.

There are literally hundreds of uses for playgrounds. Although they don't strictly have to be used for debugging purposes, they can certainly help when you hit a sticky patch in your application's logic.

Let's take a moment to demonstrate the power of playgrounds. Start a new playground in Xcode by going to File ► New ► Playground. Then enter a name for your playground: I chose BeginningXcode, as shown in Figure 11-39. Click Next, and specify a location to save the playground.

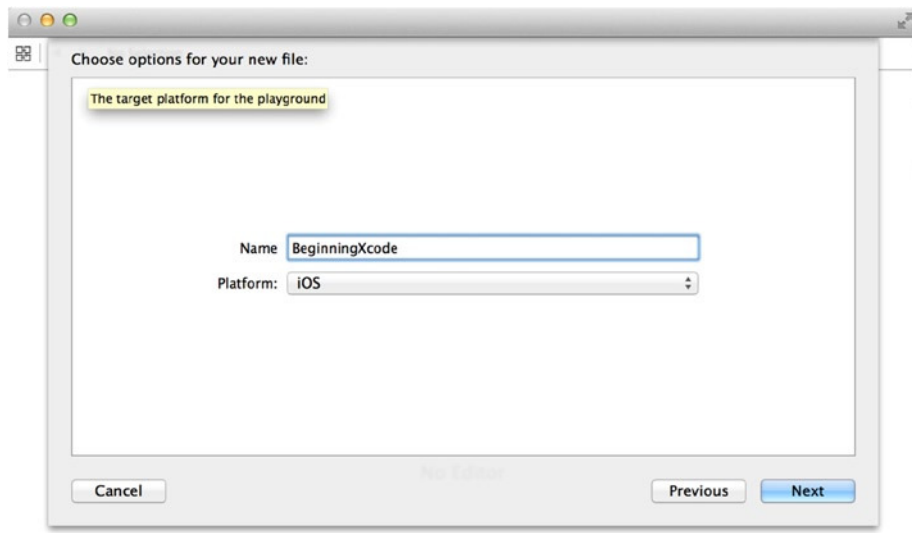


Figure 11-39. Setting the options for the playground

After saving your playground, you arrive at a screen with a couple of lines of code, as shown in Figure 11-40. This is the playground: on the left is a code editor, and on the right is a light grey pane that shows relevant details of the code, such as the stored value or the number of times a loop has run.

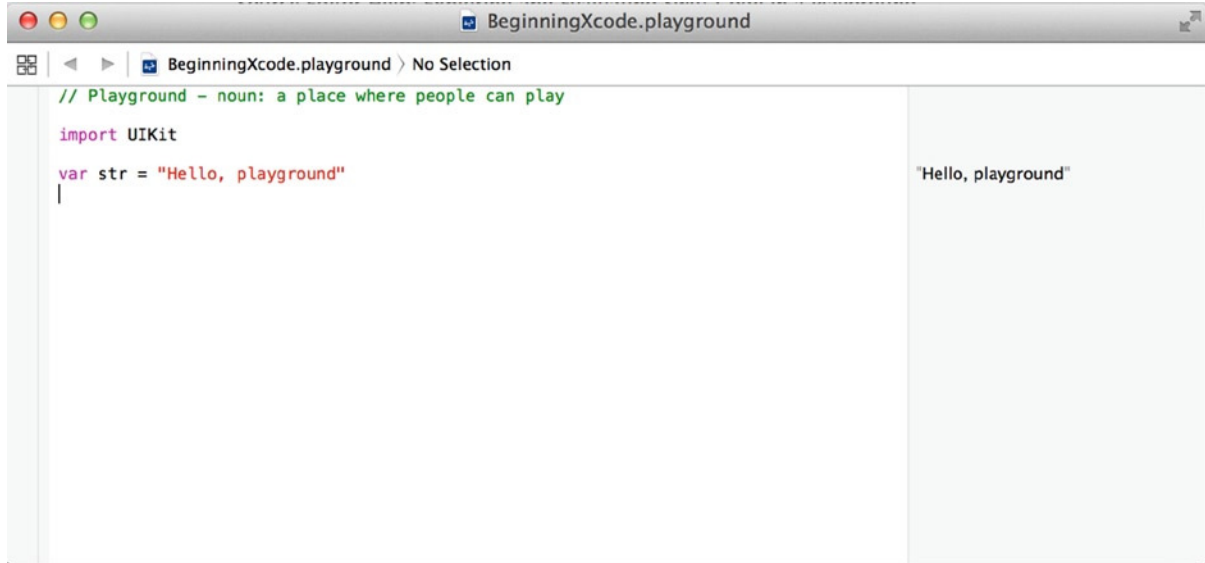


Figure 11-40. The default contents of the playground

Okay, so the playground is open. What do you do next? Let's start by taking a look at how the playground reacts to a few simple functions.

Drop down a few lines from the default code, and type the following:

```
var sum = 4
for var i = 0; i < 6; i++ {
    sum += sum * i
}
sum
```

As you type, notice that familiar things like code completion work exactly as they do in the regular Xcode code editor. Notice as well that the grey bar to the right of the code starts to fill up.

The grey bar shows that the starting value of `sum` is 4, that the loop executes 6 times, and that the finishing value of `sum` is 2,880. Notice that you didn't have to tell the playground to print to the console—you just wrote the variable name.

You're probably starting to see how this can be extremely useful; playgrounds are incredibly powerful. Let's move on and add some far more complicated code and see what happens. Suppose you want to draw a circle in your view for some reason. Below the last block of code, add the following code to your playground:

```
var bounds = CGRect(x: 0,y: 0,width: 200,height: 200)
var center = CGPoint(x: 100, y: 100)
var radius = CGFloat(100.0)

var path:UIBezierPath = UIBezierPath()
path.addArcWithCenter(center,
    radius: radius,
    startAngle: CGFloat(0),
    endAngle: (CGFloat(M_PI) * 2),
    clockwise: true)
path.stroke()
```

This code creates a very simple circle with a radius of 100 in the center of a 200 × 200 invisible area. The playground lets you go beyond seeing the values in variables: you can visualize objects and even complex animations.

Hover over the last line in the grey bar on the right which should say 5 path elements, and notice that to the right, two icons appear: an eye and a hollow circle. Click the eye icon; as shown in Figure 11-41, you can use Quick Look to see the result of your code! Pretty neat, but there is much more you can do.

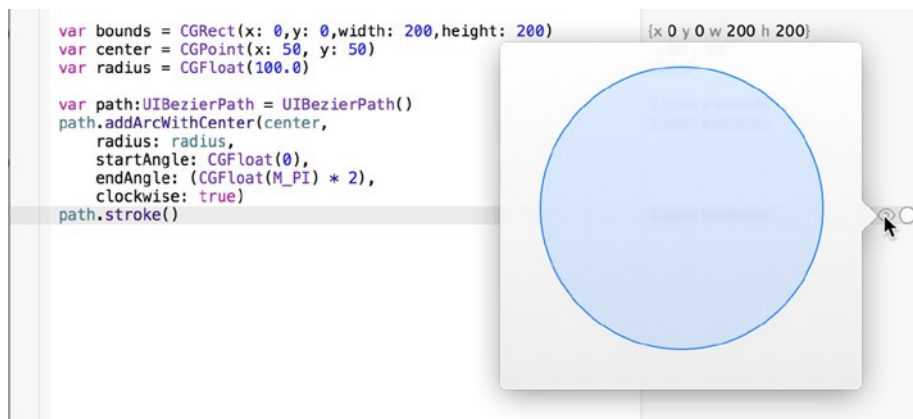


Figure 11-41. Using Quick Look to preview the circle

Let's say you want to modify values, add color information, or animate the circle drawing. Clicking the Quick Look icon each time would quickly become a pain. Move the mouse back to 5 path elements, and this time mouse over the empty circle, which changes to a plus symbol. Click it.

An Assistant Editor pane appears to the right of your code with your perfect blue circle in it, as shown in Figure 11-42. If a massive console element appears first, dismiss it by clicking the X in the top-left corner so that you can focus on your circle.

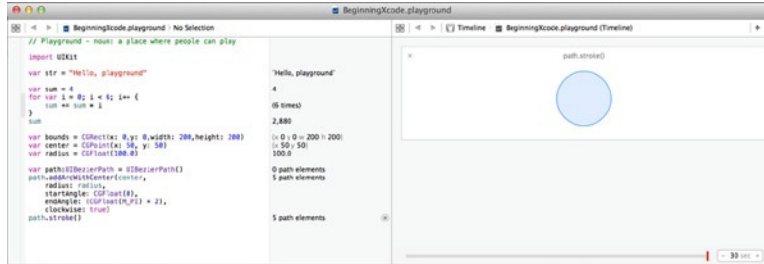


Figure 11-42. The circle shown in the Assistant Editor

Try removing the `* 2` from the line `endAngle: (CGFloat(M_PI) * 2)`, and see that immediately the circle changes to a half circle.

Hover over the line in the for loop that appears as (6 times) in the grey bar, and again click the plus symbol. Notice that the playground renders a nice graph showing the value of `sum` during each loop, as shown in Figure 11-43. This is incredibly useful when you're trying to animate something and you want it to accelerate rather than move at a constant speed.

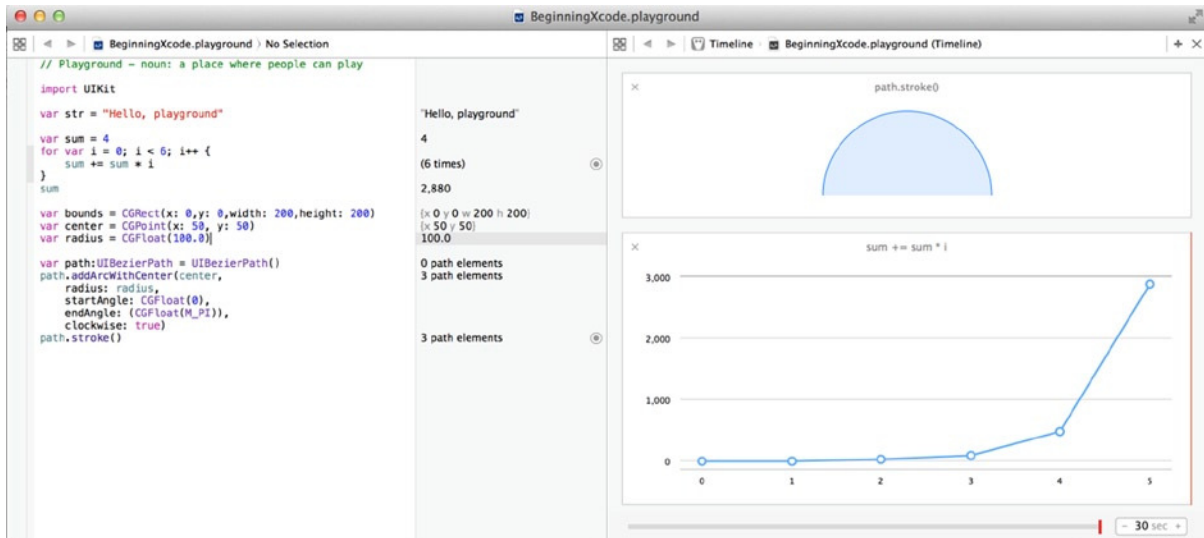


Figure 11-43. Viewing the values created in the for loop as well as the circle

That's it for playgrounds in *Beginning Xcode*. If you're moving on to a Swift programming language book or learning from the free Swift programming book that Apple released, go straight to the playground and start tapping out the examples: tweak them, break them, and see what happens.

Summary

Until IDEs develop artificial intelligence and can accurately predict what programmers intend to do in our code and automatically resolve any errors, there will be a need to debug code. It's rare to write something that works fine the first time, so knowing how to debug an application in Xcode is an essential skill.

This chapter presented a detailed look at how Xcode can be used to resolve various issues with applications. You also learned how to use Simulators to put the functionality of the application to the test when physically moving or when owning additional hardware would otherwise be required.

Specifically, in this chapter, you learned the following:

- About the types of errors that can occur
- How to use breakpoints effectively to investigate logic and runtime issues
- How the call stack can be interpreted to lead to solutions for runtime errors
- How the Breakpoint and Issue Navigators can help you efficiently debug an application
- How to use the Location and Printer Simulator features to assist with testing and debugging an application
- Using playground to try things with Swift code outside of a specific application

You now move into the final part of the book. Chapter 12 looks at the fine level of integration Xcode has with the popular version-control software called Git. You find out how it can help you work better in a team of developers and give you the ability to roll back changes.

Part **3**

Final Preparations and Releasing

Version Control with Git

Chapter 11 focused on errors and exceptions and how you can use the tools in Xcode to root them out. You looked at the three most common types of errors and exceptions: logic, runtime, and compile-time errors. You also looked at how to make sense of the call stack when your application crashes.

The focus of this chapter is version control, and specifically how Xcode integrates with the Git source code management system. Xcode stands out from other IDEs in this department. Its integration is so fine and complete that it's a joy to use and so intuitive you'll wonder why you haven't used it before.

As you work through the intricacies of version control, you'll create a voice-recorder application called *HearMeNow*. This application has Record and Play buttons initially, but later you'll branch the project and modify it to play back the voice at half speed—perfect when trying to listen to people like me who talk too quickly. It's important to note that although the application will run without error in the Simulator, you need a physical device with a microphone to see it working with your recordings.

Why Use Version Control?

Actually, the question when it comes to version control is, why not? Have you ever developed a solution, taken a vacation, and, when you returned, discovered that while you were away, a member of your team made a small change and suddenly there were a dozen bugs as a consequence? If so, version control would have saved the day.

Version-control software, also known as *source control* or *revision control*, allows you to track and manage changes made to code over time. If something suddenly stops working, you can compare it to an older version to see what's changed and hopefully get to the bottom of what's gone wrong. Version control makes it easy to see what's changed between releases. Changes to a file are highlighted and logged against a specific user, and can even hold comments.

A wide variety of version-control systems are on the market, and unlike many software markets, some of the best systems are open source. Most software-development houses use one of several tools for version control, depending on the language they write in or their preference. The most popular systems are Git, Subversion (SVN), and Microsoft's Team Foundation Server (TFS) or Visual Studio Online. Xcode uses the extremely popular Git system to provide version control.

What Is Git?

If you're British, a *git* is that colleague who steals your lunch, even though it had your name written on it in capital letters with black marker. The Git source-control management system actually has a little to do with this vernacular in its origins. It's the brainchild of the principal developer of the Linux kernel, Linus Torvalds. When none of the version-control packages available at the time supported his vision of robust distributed development of the Linux kernel, Torvalds wrote his own system: Git. He names all of his software after himself and has quipped that this one was no different, so read into that what you will. The software is quite amazing: Torvalds designed Git to be fast, efficient, and robust, and it excels at all three.

Git is notable as a system because it can be used either locally for version control or with a server to allow global collaboration on a piece of software. Online systems such as GitHub and Bitbucket provide free Git repositories so you can back up your project online and invite people to take a copy by cloning, branching, or forking the project. Another notable feature that separates Git from other, similar systems is that when you make a change, Git snapshots the project, giving you a true point-in-time view; other systems simply track changes on individual files.

When talking about Git, many terms may sound strange or complicated. Let's look at these terms to prove there's nothing to be afraid of:

- *Repository*: Also known as a *repo*, a repository encapsulates your project, storing the different versions of the files and folders and tracking the changes.
- *Commit*: When you've made changes to a file and want to put them into the repository, you commit those changes.
- *Branch*: Branching a project allows you to work on a duplicate of that project in the repository without altering the original. Typically this is done when you want to add a new feature: you branch off from the original project, make the changes, and then merge the branch back into the master branch or trunk.
- *Fork*: If you want to work on a project but have read-only access, or you want to send a project in a brave new direction, forking is a good idea. This allows you to duplicate the repository, but it's reserved for online services such as GitHub or Bitbucket.

Creating the Project

Because it's suitable for the vast majority of iOS projects, you yet again use the Single View Application template for this chapter's project. The project focuses on using the built-in microphone of an iOS device to record a voice or sound and play it back. Here are the steps:

1. Open Xcode, and create a new project by going to File ► New ► New Project (⌘+Shift+N) or choosing Create A New Xcode Project on the Welcome screen (⌘+Shift+1).
2. Select the Single View Application Template, and click Next.
3. Name the project **HearMeNow**, substitute your personal information, ensure that Devices is set to iPhone, and leave the other options set to their defaults, as shown in Figure 12-1. Click Next.

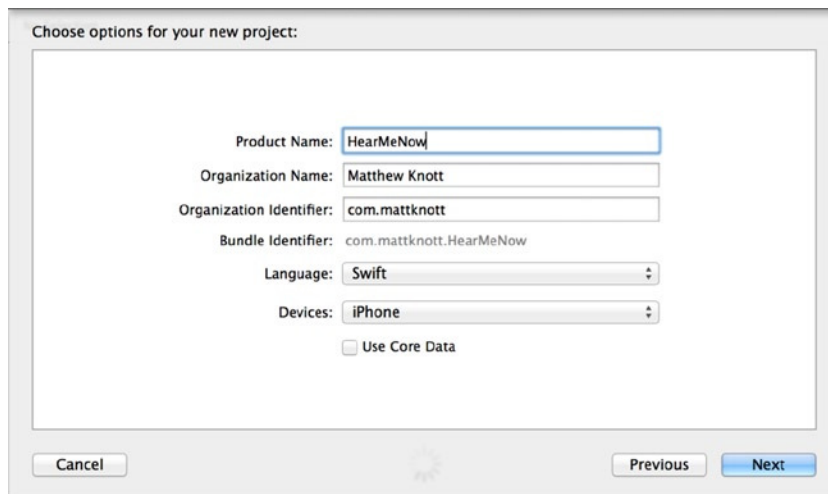


Figure 12-1. Setting up the initial options for the project

4. Click the Source Control check box at the bottom of the Save dialog. This specifies that the project should use source control. Next, specify that you want to create a Git repository locally by choosing My Mac from the drop-down list, unless you have a server set up to house Git repositories. Be sure your settings match those shown in Figure 12-2, and click Create. (Don't worry if you don't see the Add To option—it comes and goes depending on whether you have other workspaces open).

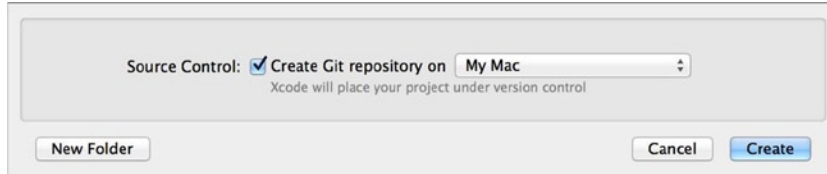


Figure 12-2. Choosing to create a Git repository locally on the Mac

Note For this chapter, you create a local Git repository on the Mac. Many development teams choose to use a dedicated server or an online solution, because doing so gives everyone access to all the team’s projects and greatly simplifies backups. Online solutions are examined later in this chapter.

- The project and a local Git repository have now been created for the project. Take a moment to select Source Control from the menu bar. This menu is where you perform the different actions covered in this chapter. As you can see at the top of the menu in Figure 12-3, you’re working on the master branch, which is fine because you just started the project.

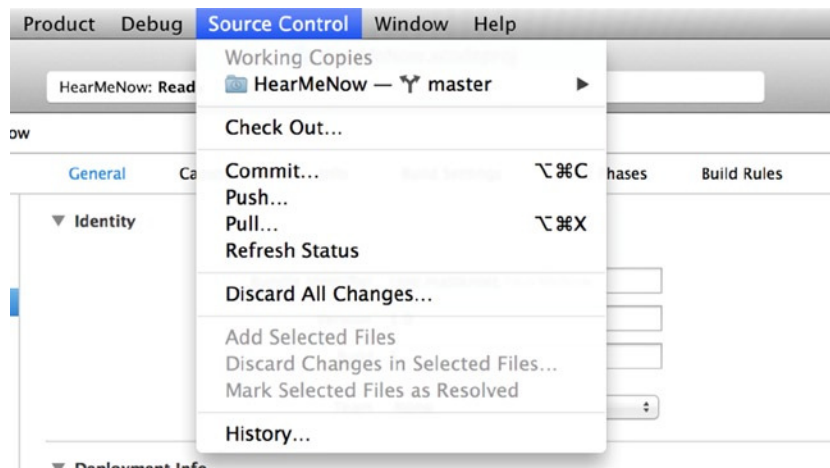


Figure 12-3. The Source Control menu

- Select History from the Source Control menu. Figure 12-4 shows the history of the project in a source-control context. When you created the project, a snapshot of that start point was automatically created: it’s called the *initial commit*. Click Done to close the dialog.

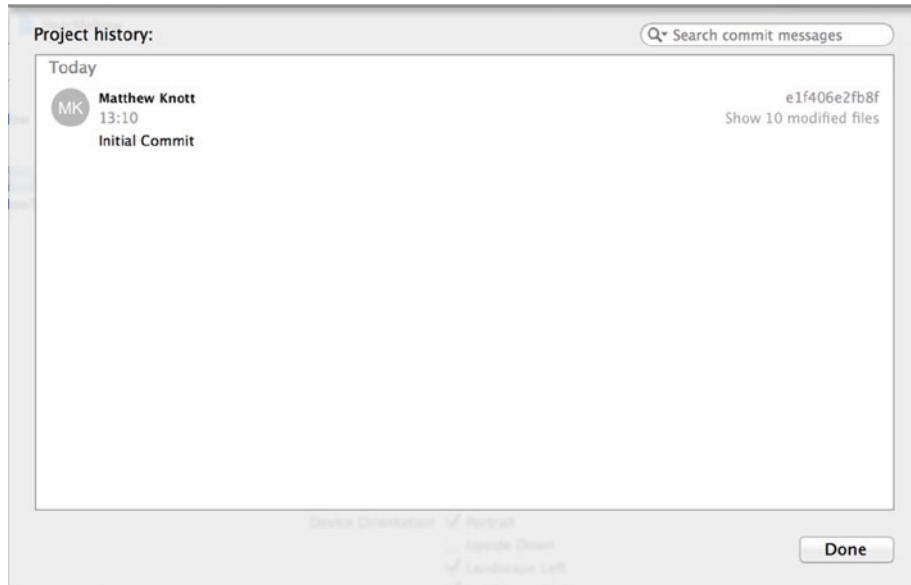


Figure 12-4. The dialog showing the history of the project under Source Control

Note Depending on the version of OS X, when you open the History dialog you may be prompted to allow access to contacts. This is so that Xcode can try to identify who made changes to the project and the repository.

The project has been created and is under source control. You're ready to begin building the interface for the project and writing the code, which uses of a framework called AVFoundation.

The AVFoundation Framework

Before you begin an interface for the project, let's quickly take a look at what the AVFoundation framework is and what it lets you do. Although there are dedicated audio and video frameworks, AVFoundation, as its name might suggest, provides a powerful set of classes that underpin these other frameworks.

AVFoundation can be used in slightly varying forms in both iOS and OS X development. Its main purpose is to support the use of time-based audio and video functions, such as recording and playback of various media types.

In this application, you set up the view controller as an `AVAudioPlayerDelegate` and also `AVAudioRecorderDelegate` so that you can take advantage of the `AVAudioPlayer` and `AVAudioRecorder` classes, which are named to clearly indicate their purpose.

AVFoundation can be added to the project simply by using the `import` statement in the view controller. You don't need to physically add it to the project. Let's start building the interface.

Creating the Interface

The interface for the application is extremely simple. It consists of a label and two buttons, one to record or stop recording and another to play back or stop playing back:

1. Open `Main.storyboard` from the Project Navigator.
2. You created this project from the Single View Application template, so there is a single view on the storyboard. Drag in a label from the Object Library, place it on the view, move it to the top of the view so that the blue guidelines appear, and then release it. Use the handles on either side of the label to resize its width until it again reaches the left and right guidelines. Your label should resemble that shown in Figure 12-5.

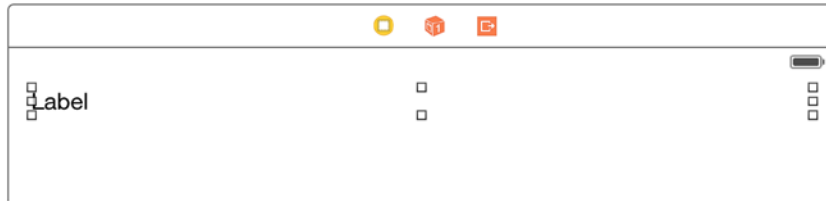


Figure 12-5. Positioning the label in the view

3. You need to set the label's attributes. Open the Attributes Inspector, and change the Text attribute from Label to **Hear me now...** and Alignment to the center position, as shown in Figure 12-6.

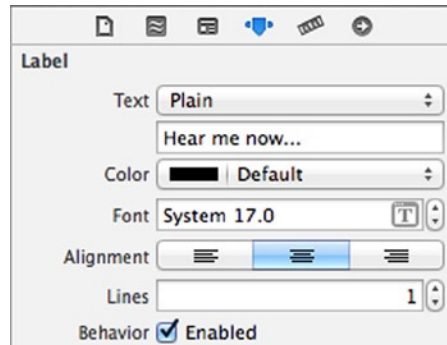


Figure 12-6. Setting the attributes of the label

Now that the label is configured, you can turn your attention to the two buttons. Drag in two buttons from the Object Library, positioning them one beneath the other, as shown in Figure 12-7.

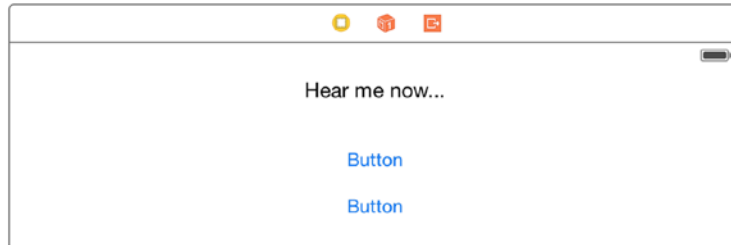


Figure 12-7. Adding two buttons to the view

4. Resize both buttons the same way you did the label, dragging the left and right sides until the guidelines display. Change the attributes for the buttons in the Attributes Inspector, naming the top button **Record** and the bottom button **Play**.
5. Realign both buttons so they're once again dead center. Your finished interface should resemble that shown in Figure 12-8.

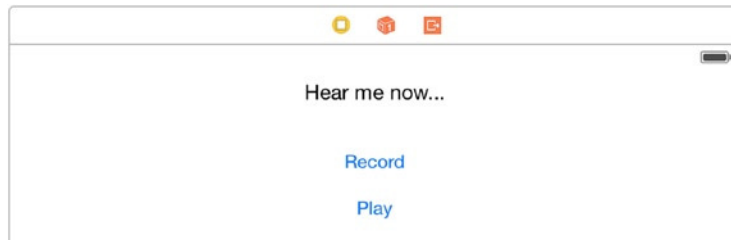


Figure 12-8. The completed interface for the application

6. Let's fix the elements in place using the Resolve Auto Layout Issues button. Click the button, and then click Add Missing Constraints under All Views in the view controller heading, as shown in Figure 12-9.

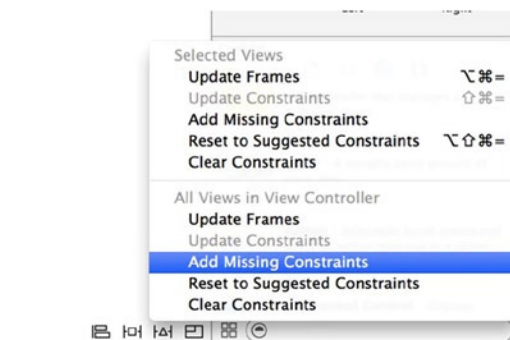


Figure 12-9. Adding in constraints for the controls in the view

7. The next task is to create an outlet and an action for each of the buttons. Open the Assistant Editor, and ensure that `ViewController.swift` is selected alongside your storyboard.
8. Control-drag a connection from the Record button. Create an outlet called `recordButton`; then create another outlet for the Play button, and call it `playButton`.
9. Repeat the control-drag process for each button, but this time create an action for the Record button called `recordPressed` and one for the Play button called `playPressed`. The beginning of your view controller should resemble that shown in Figure 12-10.

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var recordButton: UIButton!
    @IBOutlet weak var playButton: UIButton!
    @IBAction func recordPressed(sender: AnyObject) {
    }
    @IBAction func playPressed(sender: AnyObject) {
    }
}
```

Figure 12-10. The outlets and actions created in the header file

10. With the outlets and actions created, it's time to add the final touches. Switch back to the Standard Editor, and open `ViewController.swift` from the Project Navigator. Earlier, when I mentioned the AVFoundation framework, I mentioned two protocols you need to add to the view controller: `AVAudioPlayerDelegate` and `AVAudioRecorderDelegate`. Add the highlighted code to the header to import the AVFoundation framework and apply the two protocols:

```
import UIKit
import AVFoundation

class ViewController: UIViewController, AVAudioPlayerDelegate, AVAudioRecorderDelegate {

    @IBOutlet weak var recordButton: UIButton!
    @IBOutlet weak var playButton: UIButton!
    @IBAction func recordPressed(sender: AnyObject) {
    }
    @IBAction func playPressed(sender: AnyObject) {
    }
}
```

11. In addition to adding code to the `viewDidLoad` method and the two actions, you also need two delegate methods and a number of instance variables. Let's add the instance variables, because they're essential for all the methods in the application. At the top of the file, add the variables highlighted next, and then add the five instance variables:

```
class ViewController: UIViewController, AVAudioPlayerDelegate, AVAudioRecorderDelegate {

    var hasRecording = false
    var soundPlayer : AVAudioPlayer?
    var soundRecorder : AVAudioRecorder?
    var session : AVAudioSession?
    var soundPath : String?

    @IBOutlet weak var recordButton: UIButton!
    @IBOutlet weak var playButton: UIButton!
```

Let's look at what these instance variables do:

- `Bool` - `hasRecording`: Determines whether a recording has been made
 - `AVAudioPlayer` - `soundPlayer`: Handles all audio playback
 - `AVAudioRecorder` - `soundRecorder`: Handles recording from the microphone
 - `AVAudioSession` - `session`: Activates and deactivates the audio session
 - `String` - `soundPath`: Holds the path for the recorded file
12. Scroll down to the `viewDidLoad` method. In this method, you initialize the `session`, `soundPath`, and `soundRecorder` objects. As in previous chapters, I won't always go into detail about the code, because that isn't the focus of this book. Add the following highlighted code to your `viewDidLoad` method:

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Do any additional setup after loading the view, typically from a nib.

    soundPath = "\(NSTemporaryDirectory())hearmenow.wav"

    let url = NSURL(fileURLWithPath: soundPath!)

    session = AVAudioSession.sharedInstance()
    session?.setActive(true, error: nil)

    var error : NSError?

    session?.setCategory(AVAudioSessionCategoryPlayAndRecord, error: &error)

    soundRecorder = AVAudioRecorder(URL: url, settings: nil, error: &error)
```

```

    if(error != nil)
    {
        println("Error initializing the recorder: \(error)")
    }

    soundRecorder?.delegate = self
    soundRecorder?.prepareToRecord()
}

```

13. With the `soundRecorder` object initialized, it's time to write the code for the `recordPressed` action. This is called when the Record button is tapped. If the application is currently recording, it stops the recording process and sets the button title back to Record; otherwise, it uses the session object to check recording permissions and then either start recording, if granted, or use `println` to display that the request was denied. Add the following highlighted code to the action:

```

@IBAction func recordPressed(sender: AnyObject) {
    if(soundRecorder?.recording == true)
    {
        soundRecorder?.stop()
        recordButton.setTitle("Record", forState: UIControlState.Normal)
        hasRecording = true
    }
    else
    {
        session?.requestRecordPermission(){
            granted in
            if(granted == true)
            {
                self.soundRecorder?.record()
                self.recordButton.setTitle("Stop", forState: UIControlState.Normal)
            }
            else
            {
                println("Unable to record")
            }
        }
    }
}
}

```

14. You've written the action for the Record button. Now you need to write one for the Play button, which calls the `playPressed` action. In this method, the action is determined from three possible states: if the `soundPlayer` object is currently playing back the audio file, it pauses; if the `hasRecording` object is set to Yes or true, the application plays the recorded file; and if neither of the other two states is met, the method checks to see if the player is initialized,

which means it's being asked to resume from a paused state. Add the following highlighted code to the `playPressed` method:

```
@IBAction func playPressed(sender: AnyObject) {
    if(soundPlayer?.playing == true)
    {
        soundPlayer?.pause()
        playButton.setTitle("Play", forState: UIControlState.Normal)
    }
    else if (hasRecording == true)
    {
        let url = NSURL(fileURLWithPath: soundPath!)
        var error : NSError?

        soundPlayer = AVAudioPlayer(contentsOfURL: url, error: &error)

        if(error == nil)
        {
            soundPlayer?.delegate = self
            soundPlayer?.play()
        }
        else
        {
            println("Error initializing player \(error)")
        }
        playButton.setTitle("Pause", forState: UIControlState.Normal)
        hasRecording = false
    }
    else if (soundPlayer != nil)
    {
        soundPlayer?.play()
        playButton.setTitle("Pause", forState: UIControlState.Normal)
    }
}
```

15. The final task in the implementation of this application is to write two delegate methods that are called when the recording or playing is finished, to change the text of the relevant button and, in the case of playing the audio back, to set the `hasRecording` object to `no` or `false`. Add the following two methods just after the `playPressed` method:

```
func audioRecorderDidFinishRecording(recorder: AVAudioRecorder!, successfully flag: Bool)
{
    recordButton.setTitle("Record", forState: UIControlState.Normal)
}

func audioPlayerDidFinishPlaying(player: AVAudioPlayer!, successfully flag: Bool) {
    playButton.setTitle("Play", forState: UIControlState.Normal)
}
```

With those last two methods in place, the application is ready to be run. As I mentioned at the beginning of this chapter, you need a physical iOS device to test the application's full functionality, although you can test for compile-time errors and runtime errors to some extent without one. To select a physical device vs. the currently selected Simulator, click the Simulator name next to the scheme, as shown in Figure 12-11, and select your physical device if you have one. Physical devices are separated from the various Simulators and are represented by the device's icon. In order for your device to appear in the list, it must be connected to your Mac via the USB cable and must be prepared for development, which is something I cover in Chapter 14.



Figure 12-11. Selecting a physical device from the list of available devices

Run the application, and you should find that you have a plain but effective sound recorder. Tap the Record button and speak. When you click Stop, the sound is saved onto the device and is available for playback. Record as many times as you like, but the application uses a single filename and therefore overwrites the file every time you click Record. Figure 12-12 shows the application running on a physical device.

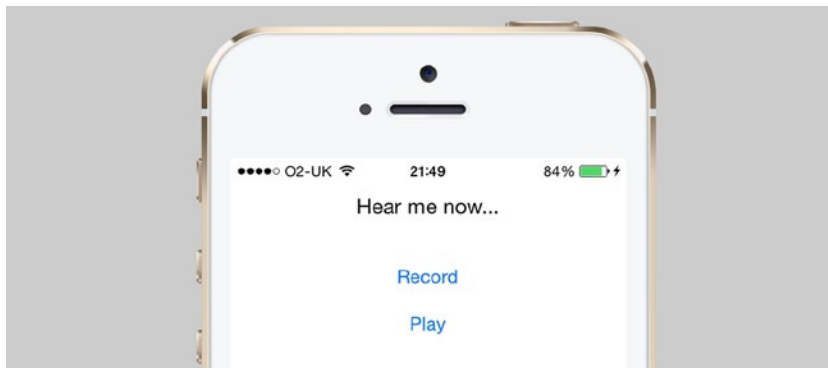


Figure 12-12. The application running on a physical device

Once you've played with this great little application you've just created, stop it and return to Xcode. Take a look at the Project Navigator: every item that has been changed since the project was created (the storyboard and the view controller) has an M symbol next to it, as shown in Figure 12-13. This icon indicates that the project item has been modified since the last time the project changes were committed to the Git repository.

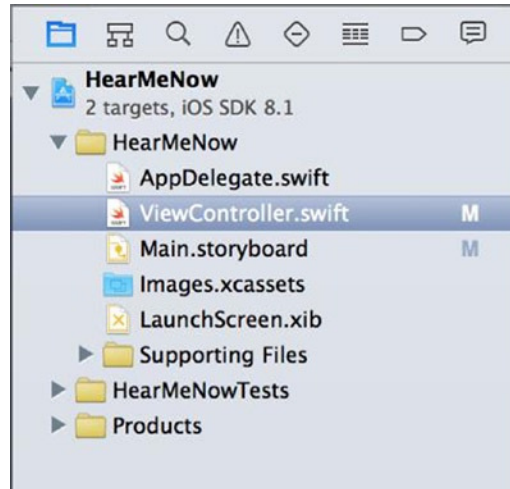


Figure 12-13. The Project Navigator places an M next to each project item modified since the last commit

Committing Changes

When making changes to a project under source control, just as with any other project you've created in this book, the changes take effect locally as you save the files or run the application. So why is Xcode pointing out that some of the files and settings in the project have been modified? The reason is that although the changes are being applied, they aren't saved into the Git repository until you perform an action called a *commit*. A commit action creates an updated snapshot of the project in the Git repository: you're creating a point-in-time reference for your project that you can go back to at any time.

Because you've created the first goal for the project, this is a good time to commit the project and update the Git repository. Select **Source Control** ► **Commit** (⌘+⇧+C). You're presented with a wealth of information about the files that have changed, as shown in [Figure 12-14](#).

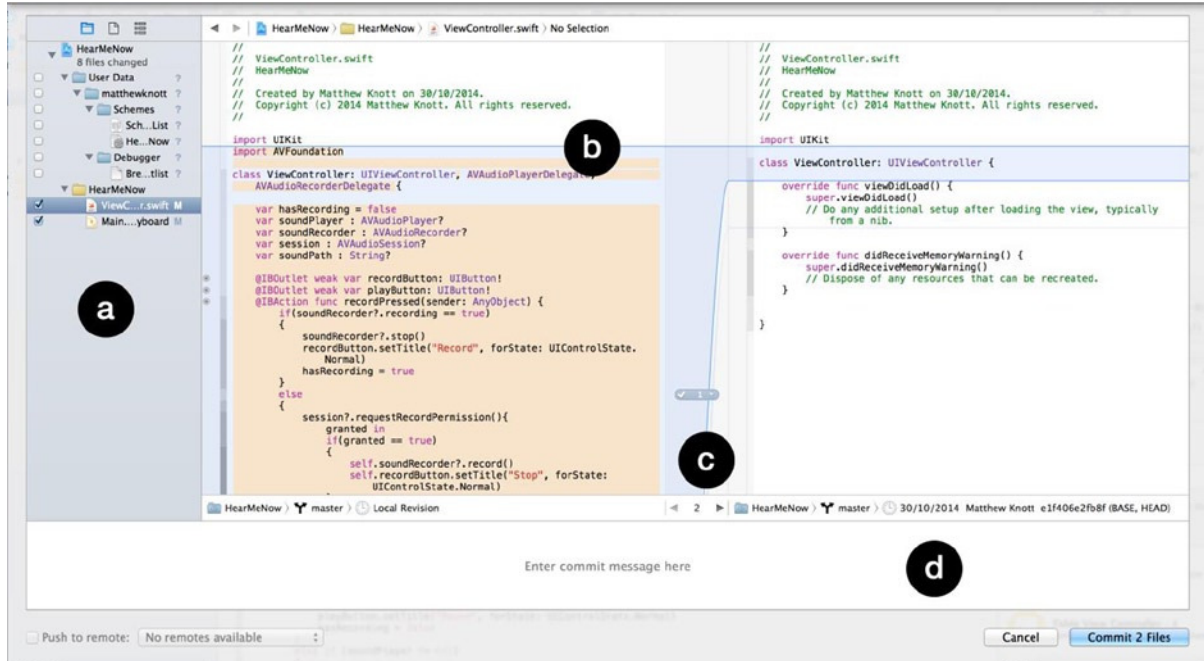


Figure 12-14. The Commit screen, detailing all changes made in the project since the last commit

In this instance, I've selected `ViewController.swift`, because it received the greatest modification. As indicated in Figure 12-14, there are four key points of interest.

- **List of modified items:** This area lists all items in the project that have changed. You have three options for viewing this list: Project View, which is the default; File View, which shows modified files in the context of their folder structure in Finder; and Flat View, which simply lists the items with no structure around them. It's important to note that each item has a check box next to it that can be deselected if you wish to exclude it.
- **Highlighted changes:** When selecting an item, you're shown the changes that have occurred in that item since the initial commit, unless it's a new item. Each change is highlighted, and you can see the point at which the original file was modified.
- **Change numbering:** In each modified item, Xcode dynamically identifies and groups changes to make it easier for you to ignore specific changes at the time of the commit or to discarding them entirely. Clicking the numbered item in this area presents these options, as shown in Figure 12-15. At the bottom of the list is a number that represents the number of changes that can be affected in isolation along with stepper controls that allow you to jump between the changes in sequence.

- **Commit message:** When committing the modifications to Git, it's important to add a message explaining what has been modified. That way, other users can see what's physically changed. Here you can explain why you made the changes and what your thinking was behind each change modification.

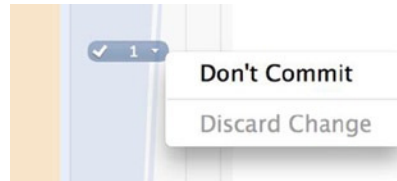


Figure 12-15. When you click one of the numbered changes, you can choose to ignore or discard that change

Add a comment to the commit message box to explain what's changed, and click the button in the bottom-right corner that says Commit 2 Files. After a brief pause, you're returned to Xcode. Note that there are no longer any M icons in the Project Navigator, meaning the project hasn't been modified from the version currently stored in the Git repository.

For an alternative view of what you've just done, go back to Source Control ► History and note that you now have two entries: the initial commit and the commit you just made, as shown in Figure 12-16.

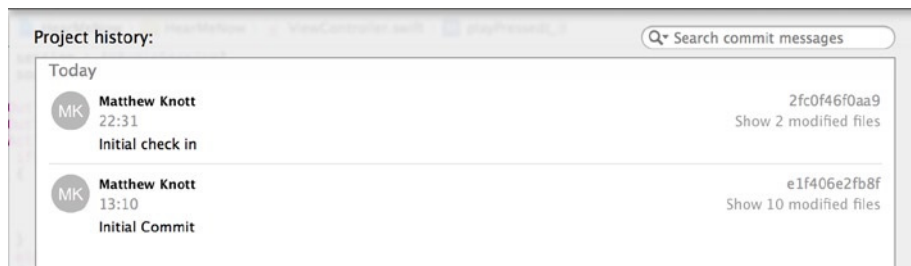


Figure 12-16. The project history, showing the initial commit and the one just performed

The details of the last commit appear at the top, with previous changes appearing below in reverse chronological order. Hopefully you can see the value of adding comments: they help provide a rich picture of how and why the application has changed, which is important especially if you're working as part of a team, where understanding who did what is critical. Although this view gives you a good deal of information about the changes made with each commit, Xcode has a number of ways you can compare multiple version of a file and the changes in that file.

Examining Changes with the Version Editor

One of the most enticing aspects of version control is that it gives you a safety net in which you can have multiple people working on a single project. When something stops working between commits, it's important to be able to look back at what's changed and even determine who is to blame for breaking the code. In addition to the Standard and Assistant Editors, Xcode provides a Version Editor. As you may remember from Chapter 3, the Version Editor is the third icon in the group of editors. You may also have noted the small downward-pointing arrow, which indicates that the editor has multiple views: Comparison (the default), Blame, and Log, as shown in Figure 12-17.



Figure 12-17. The Version Editor is the third icon in the group of editors

The Comparison View

To begin, select `ViewController.swift` from the Project Navigator, and then select the Version Editor icon in Xcode. The comparison view is the default view that is presented. It will be immediately familiar, because it forms the major part of the commit process, except that in the Version Editor it has far more flexibility. The difference in the view can be found on the bar at the bottom of the editor, as shown in Figure 12-18.



Figure 12-18. The bar beneath the editor provides the core functionality in the comparison view

You can configure either the left or right pane to show any available version of the file you've selected in the Project Navigator. Because the left and right sections have identical functionality, I'll focus on the left side for a moment. There are three segments: in this case, they're `HearMeNow`, which represents the projects in the workspace; `Master`, which represents the selected branch; and `Local Revision`, which represents the selected version of the file to be displayed. On the right, almost the exact same information is displayed, except that it's the version from the last time the file was committed.

From the pane on the right, select the `Master` branch item in the bar. You see the available versions of the file that can be compared, as shown in Figure 12-19. To review again what has changed since the initial commit, select the bottom item, which is the oldest one available.



Figure 12-19. The list of available revisions in the selected branch

When you select the older version, the code comparisons appear, highlighting what has changed between the two selected versions, as shown in Figure 12-20. I've already discussed the functionality available when selecting a numbered change in the previous discussion of the commit process, so I won't go into that again. However, it's worth noting that unlike when you commit, you don't have the option to exclude code from the commit, because it's already committed (which makes sense).

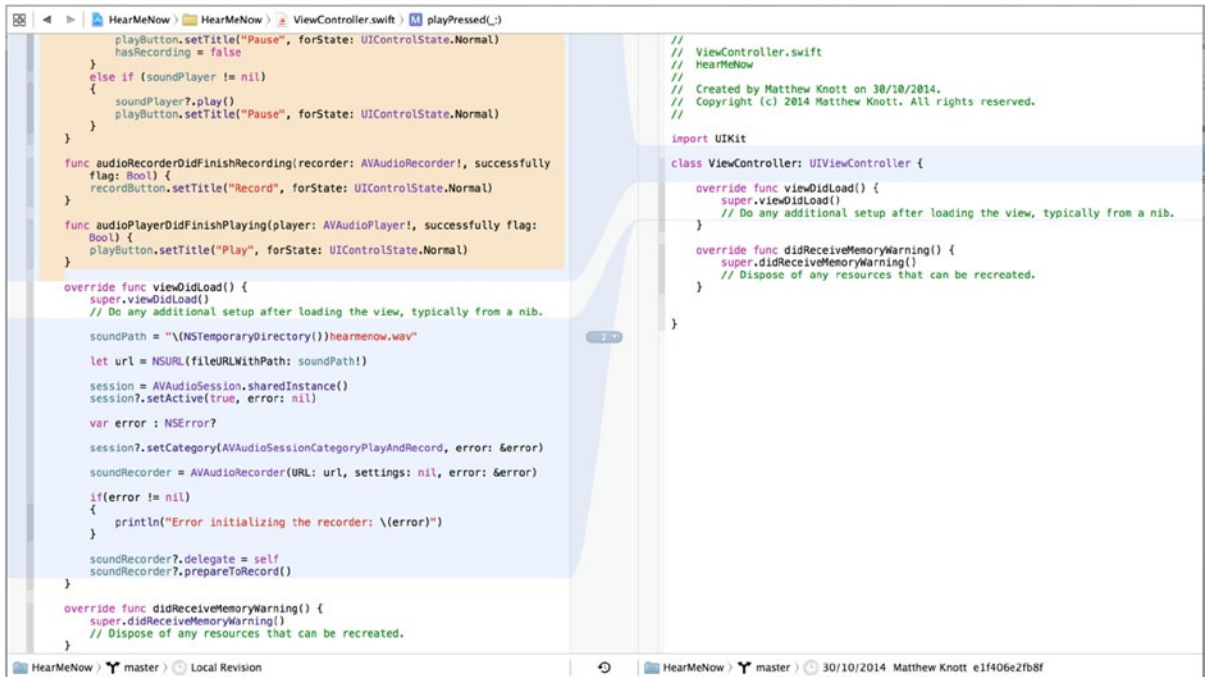


Figure 12-20. Comparing the current local revision with the oldest version available

One neat feature that Xcode provides to allow you to compare different version of a selected file is the version timeline. If you've ever used Time Machine on your Mac to restore a file, you'll find this very familiar. Click the timeline viewer icon, which looks like a clock in the center of the bar in Figure 12-18. The central area between the two panes is then replaced with the version timeline, as shown in Figure 12-21.



Figure 12-21. The timeline viewer icon opens the version timeline

At the bottom of the timeline are two default options: Local and Base. Local shows you the current version of the file, and Base shows the last committed version. Comparing base to local versions can be useful if you started making changes and made a mess of things. You can identify the offending code and revert it back to how it was in the previous version.

Above Base is a reverse chronological order of the previous commits to the repository. As Figure 12-22 shows, as you move the cursor over an entry in the timeline, it shows the key details about the commit, such as date and time, the comment, and the name of the person who performed the commit.

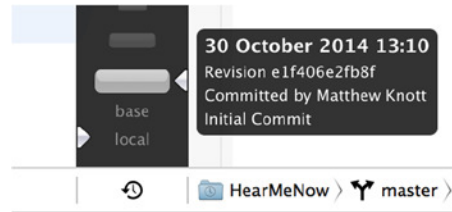


Figure 12-22. Each entry in the timeline shows the commit comment, date and time, and the name of the person who committed the project

Arrows on the left and right of the timeline indicate what is being displayed in the corresponding pane, giving you extremely fine control over which revisions are being compared.

The Blame View

All the Xcode integration with Git is done so that you don't have to worry about what happens behind the scenes. As with many of Apple's products, it just works; but in the background, Xcode is firing off command-line messages to the Git system to perform the action you've just asked Xcode to perform. There is a host of commands, including `git blame`, which is what the blame view bases its display on.

Blame is an unfair term, because often you aren't looking to actually blame anyone for anything—you just want to know who changed what in a project so you can ask them a bit more about it. It's also unfair because it adds a negative term to one of the coolest views when using Xcode's source control.

To turn on the blame view, click the Version Editor button again, and a menu will appear. Select Blame; refer back to Figure 12-17 to see the icon appropriate for your operating system.

Once you've opened the blame view, you can see why it's so powerful in a team environment. Figure 12-23 shows the detailed analysis of every change in the selected file, when it was made, and by whom. If a colleague wrote a method that you're not quite sure about, you know who to talk to about it.



Figure 12-25. The log view in the Version Editor

Branching in a Repository

When you're developing a project that is under version control with Git, the master branch or trunk should always be used to hold the version of your project that is ready for release. When you want to add new features, you could just continue working on the master branch—but then if you need to modify the code to fix a bug, you have no good version to update, and the development process starts falling apart. In this situation, you can create a branch from the master. This allows you to work in total isolation from the master branch. Once you're happy that your branch is complete, and you're ready to add the changes into the release version, you can merge your branch back into the master.

To add some new functionality to the HearMeNow application, let's create a new branch called `SlowDown`, as shown by Figure 12-26. The master branch will remain untouched, and all of your work will be done on the new branch. In this new branch, you're going to modify the playback routine to slow down playback to half of normal speed, which is a useful feature when you're typing up dictation or trying to understand someone who talks quickly or in a language you're trying to learning.

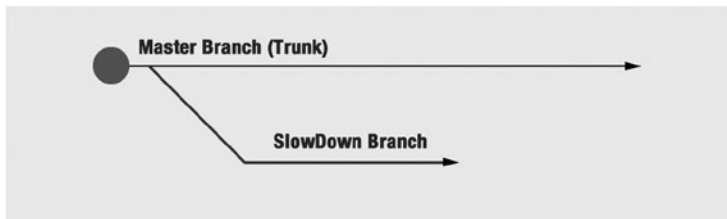


Figure 12-26. A branch allows you to work in isolation from the master branch

To create a new branch, go to Source Control ► HearMeNow ► Master ► New Branch. You're presented with a dialog asking you to name the branch, as shown in Figure 12-27. Name it `SlowDown` or a name of your own choosing, and click Create.

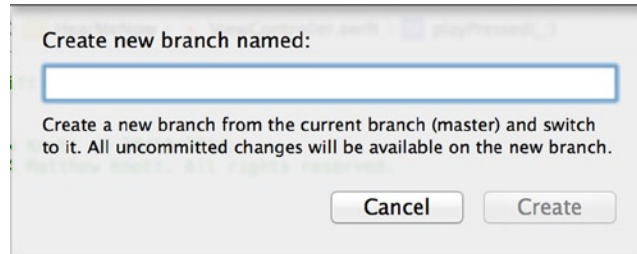


Figure 12-27. The new branch dialog, asking you to name the branch

Xcode creates the new branch and automatically switches you to it. Open `ViewController.swift` from the Project Navigator, and ensure that you're still in the log view. Notice that as Figure 12-28 shows, you're now working on the `SlowDown` branch instead of the master.



Figure 12-28. The branch has changed from the master to the new `SlowDown` branch

Now that there is a separate branch, you're free to alter the code without fear of breaking the original and working project, which safely remains in the master branch. You can add the code needed to reduce the playback speed, which can be achieved with only two lines.

If it isn't already open, select the Standard Editor and open `ViewController.swift` from the Project Navigator. Scroll down to the `playPressed` action; you need to change the `rate` property used by the `soundPlayer` object. Add the highlighted code to the second `if` statement just after you set the player delegate:

```
soundPlayer = AVAudioPlayer(contentsOfURL: url, error: &error)

if(error == nil)
{
    soundPlayer?.delegate = self
    soundPlayer?.enableRate = true
    soundPlayer?.rate = 0.5
    soundPlayer?.play()
}
else
{
    println("Error initializing player \(error)")
}
```

Before you commit the changes to the current branch, run the application to test the new functionality. When the application runs, record yourself saying something, and play it back. If all goes well, you should come across as sounding slightly tired or a little drunk, but most important the sound is half of normal speed.

When you're satisfied with the change, commit the changes the same way as you did earlier in the chapter, and remember to add a suitable comment. You're well on your way to mastering Git—but now that you've changed this branch of the application, how do you apply those changes to the master branch? With unbelievable ease, as it happens.

Merging Branches

You've been able to develop your changes without affecting the working solution that you created in the master branch. But having successfully made the changes to the SlowDown branch, it's time to merge this arbitrary branch with the master branch and release another version of the application.

Merging is incredibly simple to do. Before you merge the two branches, take a moment to compare the `ViewController.swift` file between the two branches. Open the Version Editor and the default compare view, and then select `ViewController.swift` from the Project Navigator.

By default, Xcode present you with the local revision on the left and the last committed version on the right. This tells you nothing, because they're effectively the same at this point. At the bottom of the pane on the right, click `SlowDown`, and a menu appears. Hover over `Master`, and then select the most recent version, as shown in Figure 12-29. Doing so loads the version of `ViewController.swift` that you committed the first time.

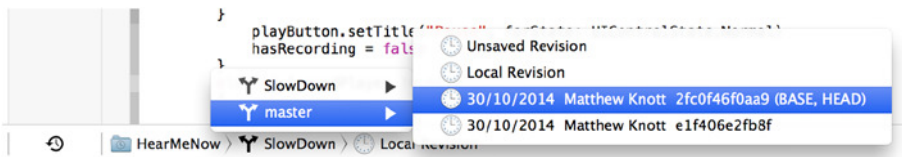


Figure 12-29. Selecting the version of `ViewController.swift` from the master branch for comparison

When the other version of the file appears, scroll down to where you modified the `playPressed` method. As Figure 12-30 shows, Xcode has singled out the change you made, so you can clearly see that the code you just wrote doesn't exist in the master branch.



Figure 12-30. Xcode showing the difference between the `ViewController.swift` file in both branches

Now that you're certain there is a difference between the files, you should be able to merge the branches and see the code added into the master branch. Open the Source Code menu again, and hover over the HearMeNow – SlowDown item. As you can see in Figure 12-31, Xcode gives you two options for merging branches: Merge From Branch and Merge Into Branch. Which branch you're currently working on will influence your choice here:

- Use Merge From Branch if you're currently working on the target branch—the one you want to add the changes to. You can then select a source branch that has the changes you wish to merge from.
- Use Merge Into Branch if you're currently working on the source branch—the one that holds the changes—as you currently should be. You can then select the branch you want to use as the target to merge into.

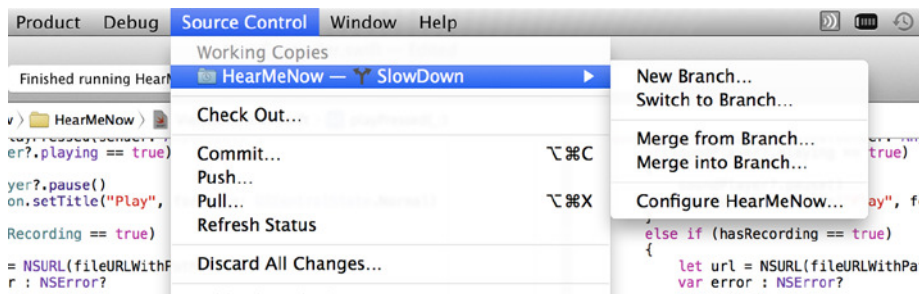


Figure 12-31. Examining the options for merging branches

Based on these two cases, the latter is needed in this scenario. Choose Merge Into Branch, and, as shown in Figure 12-32, you're presented with a screen to select a target for merging into.

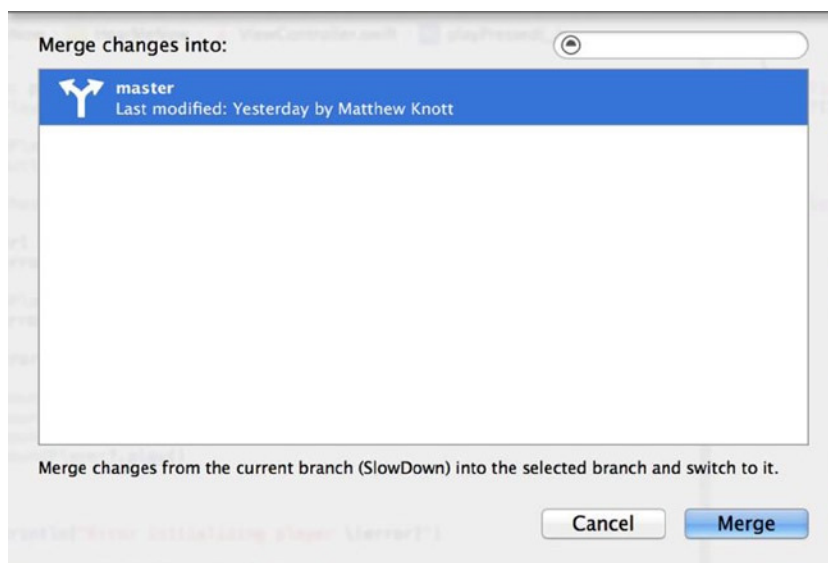


Figure 12-32. Selecting a target branch

Note If you're prompted that your changes have been made but not committed, be sure to commit the changes from the Source Control menu before continuing.

Select the master branch, and click Merge. You're again presented with the code comparison view, as shown in Figure 12-33, just as when you began to commit the changes; however, this time the options are different. Notice that between the left and right panes is a switch instead of the numbered option that appears when you commit changes. This is the direction slider that determines how you merge the files. Because you've been disciplined and haven't modified the master branch, only the SlowDown branch, you need to merge the changes from right to left, as is currently selected.

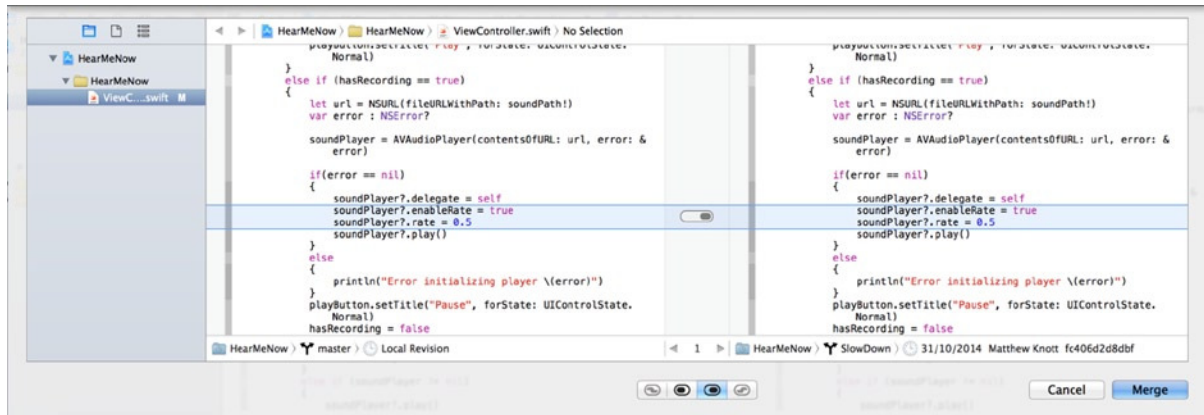


Figure 12-33. The code comparison view, shown before merging

You could very realistically have a scenario where both you and a colleague have separate branches where you're adding new features independently of each other. In this scenario, you may want to merge the two branches before merging them into the master branch. Alternatively, you may wish to merge one of the branches with the master, create a new master branch, and have your colleague merge their branch with the new branch for testing purposes before ultimately merging that branch into the master.

When you're happy that the dark part of the switch is pointing to the newer SlowDown branch as the source, click Merge. At this point, there is a good chance you'll be prompted to enable or disable snapshots. I tend to disable snapshots. I have Time Machine configured on my Mac, and because I'm committing code to the Git repository, I'm fairly comfortable that I don't need snapshots; however, the choice is yours to make.

Once you've made your choice, the merge operation is complete, and you return to Xcode and the ViewController.swift file in the master branch. Run the application: the application being built from the master branch has the half-speed playback you added in the SlowDown branch. Neat!

Removing a Branch

Now that you've successfully merged your branches, it's considered good housekeeping to remove the SlowDown branch because it's no longer needed. Go to Source Control ► HearMeNow ► Master ► Configure HearMeNow, and select the Branches tab, as shown in Figure 12-34.

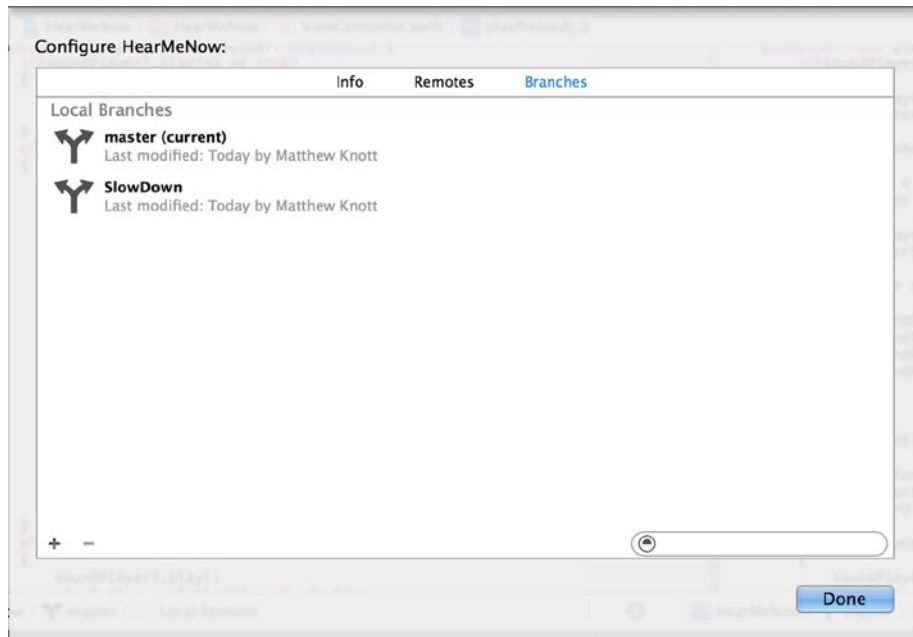


Figure 12-34. The Branches tab in the project repository configuration view

Select the SlowDown branch from the list, and then click the - icon in the bottom-left corner. You're prompted to confirm the removal, as shown in Figure 12-35 click Delete Branch. The branch is permanently removed, leaving you with a neat and tidy repository.



Figure 12-35. Xcode prompting you to delete the branch

Using a Remote Repository

Up until this point, this chapter has concentrated on using a local Git repository; but as I've mentioned, it isn't the only option. A number of online solutions are available to you for hosting or backing up your Git repository online, such as the massively popular GitHub, Bitbucket, and a number of others. Strangely enough, one of the best Git solutions for teams is Microsoft's Visual Studio Online, because of the tools built in for teams working using the Scrum agile methodology.

With these services, you get all the benefits of a local Git repository, with the added benefits of it being available for collaboration with the rest of the online world. If you don't have all the expertise you need to finish your application or game, you can push your repository online and enlist your friends' help in adding those killer features.

In this era of open source software, more and more people are turning to online Git repositories to share their source code with the world. Making your software publically available can be incredibly rewarding when you see people making interesting new applications based on your code. Being online doesn't have to mean being open to everyone, however. You can create a private repository and restrict access to it based on your preferences.

To finish this chapter, I take you through signing up for a GitHub account and sharing your repository online. Doing so is unbelievably simple.

Registering for GitHub and Creating a Repository

Registering for a GitHub account is made extremely simple because of the clever way the web site has been designed. Head to <http://github.com>: as you can see in Figure 12-36, the registration form is right on the front page. Fill in the username, e-mail, and password boxes, and click the Sign Up For GitHub button. If you already have an account, click the Sign In button in the top-right corner.

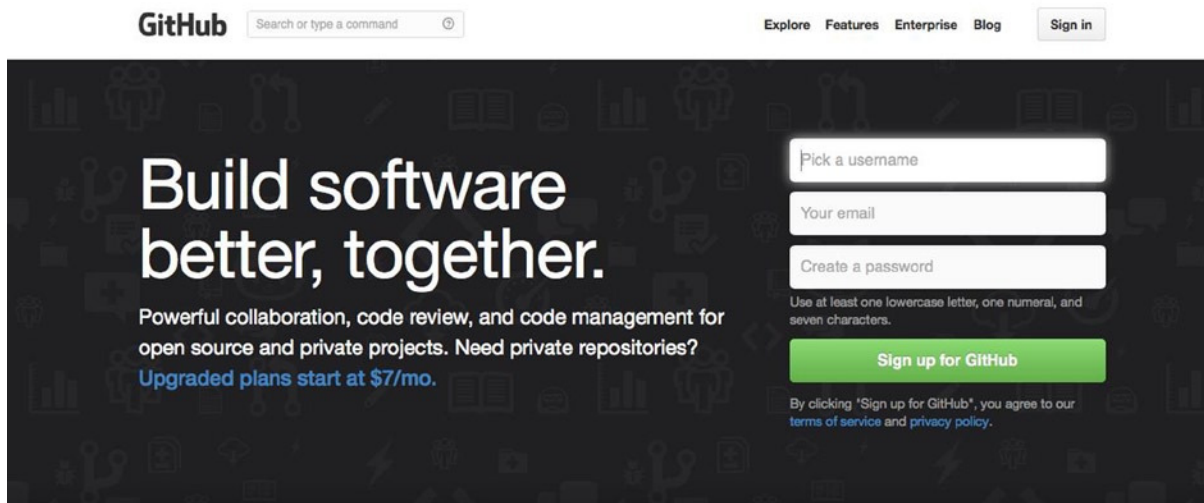


Figure 12-36. The GitHub home page

Once you've signed in to GitHub, you're presented with the launch page shown in Figure 12-37. The first thing you need to do is create a repository to add to Xcode. I've highlighted the New Repository button: click it.

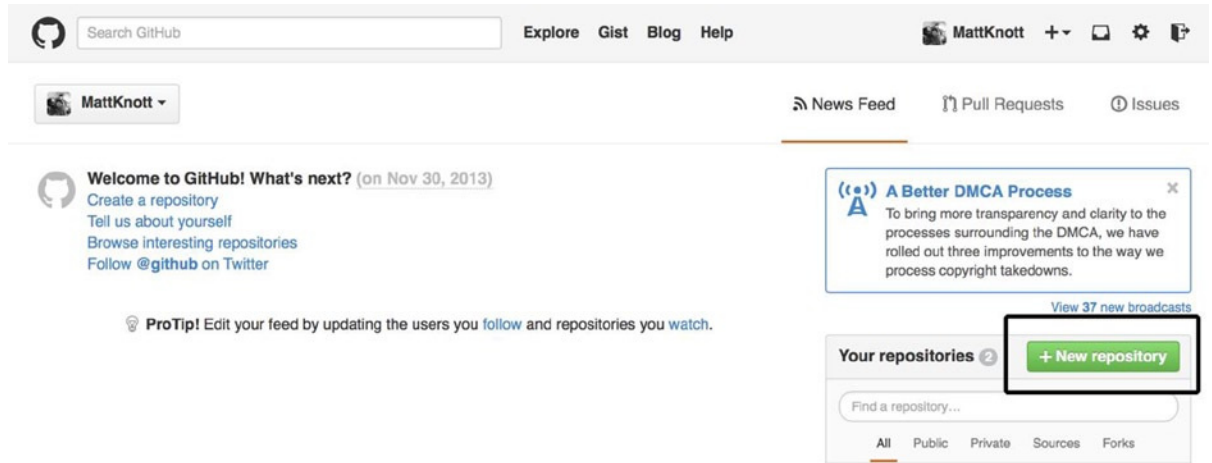


Figure 12-37. The GitHub launch page

Creating a new repository in GitHub is super easy. Just give the repository a name, as shown in Figure 12-38, and choose whether you want the repository to be public or private. I named my repository `HearMeNowSwift` after the project name, but you can name it anything within reason. Additionally, although it's optional, if you're creating a public repository then a description is very helpful. After you've added a name, click the Create Repository button.

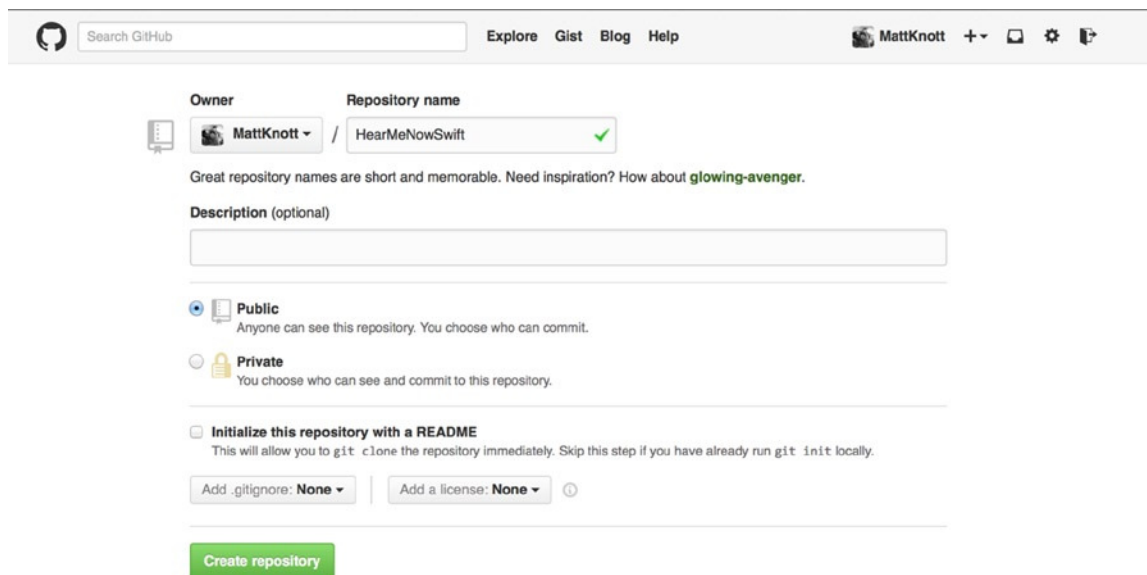


Figure 12-38. Creating a new GitHub repository

Once you've created the repository, you're taken to the repository page, which is effectively empty at this point. But there are some key details you need to make note of. Figure 12-39 shows the Quick Setup box, which holds the HTTP address for the repository. At the end of the address is a Copy To Clipboard button. Click it to copy the repository address so it's ready to add to Xcode.



Figure 12-39. The Quick Setup box on the GitHub repository page

Adding a GitHub Repository to Xcode

Whether you've worked with Git before or you're trying it for the first time, hopefully one thing that's apparent is how easy Apple has made its interface into this fantastic technology. Linking with the remote repository you just created is no exception, and Apple gives you two ways to do it: in the repository configuration or with Xcode preferences.

Adding a Remote Repository in Repository Configuration

You can quickly add a link to a remote repository by going to Source Control ► HearMeNow ► Master ► Configure HearMeNow and selecting the Remotes tab. Click the + symbol in the bottom-left corner, and click Add Remote, as shown in Figure 12-40.

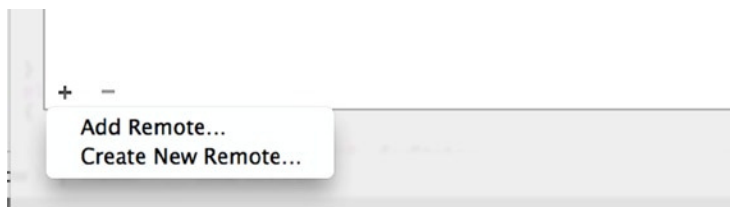


Figure 12-40. Adding a remote repository in the repository configuration's Remotes tab

Give the remote repository a reference name, and then paste in the address you copied from the Quick Setup box in GitHub. Your completed configuration should resemble that shown in Figure 12-41. Click Add Remote to add the reference to Xcode.

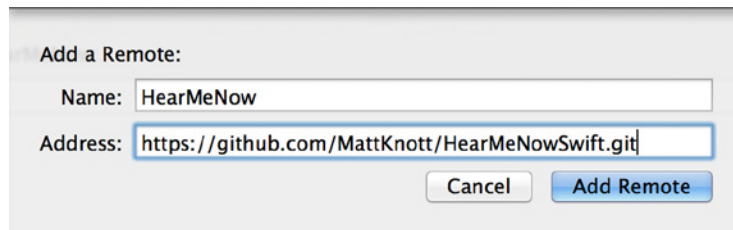


Figure 12-41. Configuring a reference to a remote repository

One of the limitations with this method is that it doesn't let you add the GitHub username and password that are required to write to the repository, which brings me nicely to the second way of adding a remote repository.

Adding a Remote Repository with Xcode Preferences

The second method for adding a remote repository is my preferred way and often the quickest way to get your repository ready for action. For this method, you need to access Xcode's preferences: the area of Xcode you went to when you customized the Xcode interface in Chapter 10. To access the preferences, go to Xcode ► Preferences (⌘+),). When the preferences appear, select the Accounts tab, as shown in Figure 12-42. In the left column, select the reference to the GitHub repository you just added.

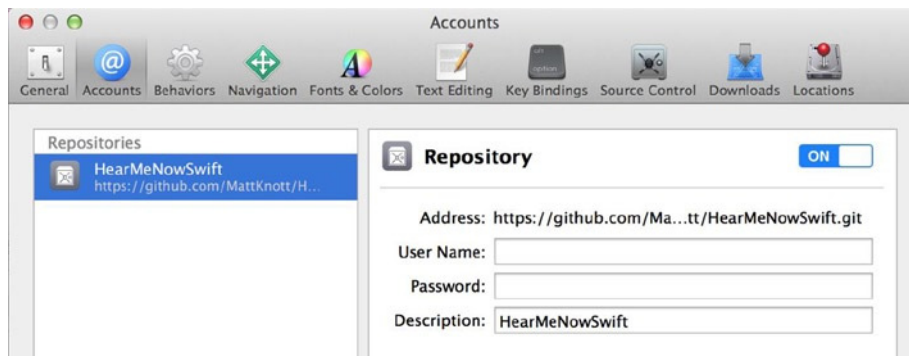


Figure 12-42. The incomplete reference for the GitHub repository

There are fields for User Name and Password: these refer to your GitHub credentials. Take this opportunity to enter them now. When you enter the username and click into the password box, Xcode tries to validate the details online. Once you've entered the details, click the red dot in the top-left corner to close the preferences.

Note A word of caution when entering your credentials: if you make a mistake with the User Name field, you have to remove the repository reference and then add it again. It's not clear why Xcode locks the field, but it does, and it can't be edited.

You have successfully added a remote repository to Xcode. Next I show you how to push your entire repository to the cloud in your GitHub repository.

Pushing to a Remote Repository

You've set up a remote repository and linked it into Xcode; all that remains is to somehow duplicate your repository onto GitHub. Because Git was developed with this type of working in mind, it has two features designed specifically for working with remote repositories: Push and Pull.

Issuing a Push command copies the entire repository over to your GitHub repository, and it's incredibly easy to do now that you've added the repository to Xcode. To push the local repository to GitHub, go to Source Control ► Push. You're presented with the dialog shown in Figure 12-43, where you can choose which remote repository you want to push to.

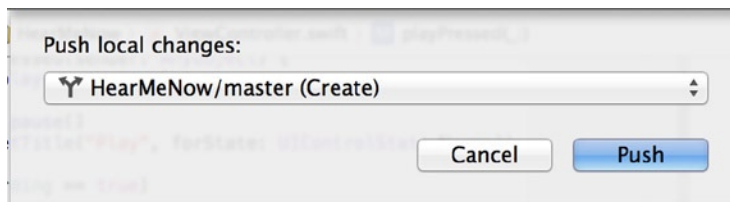


Figure 12-43. Selecting the remote repository

You only have a single repository available, so the choice here is simple: click the Push button. Xcode takes a few moments (depending on your connection speed) as it sends the entire repository to GitHub. When the process is complete, the dialog automatically closes.

Now for the exciting part! Head back to your web browser and the GitHub web site and refresh the page on your repository, and the contents of your project appear, as shown in Figure 12-44.

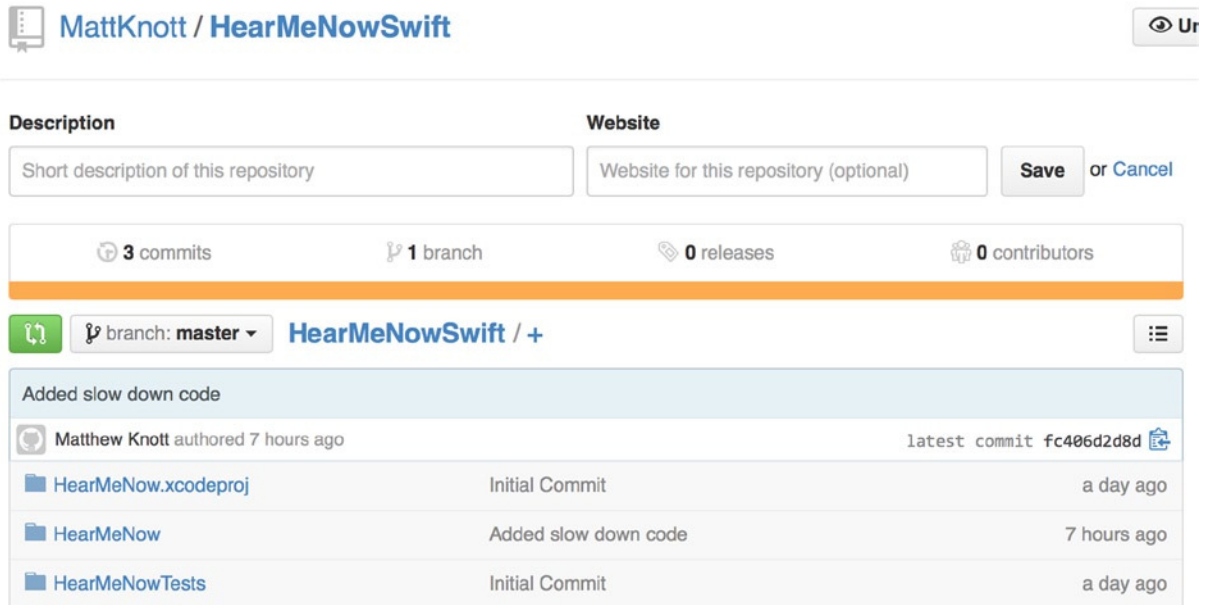


Figure 12-44. The files from the project are now also hosted on GitHub

In just a few quick steps, you've taken a piece of work that was stuck in the confines of your Mac and shared it with the world. Anyone can look at your code if you let them. They can copy it to their own Git repository or offer to update yours, and you'll come to learn the collaborative joy that is Git version control.

Updating the Remote Repository

Now that you've pushed your code to a remote repository, you need to think about maintaining it as your project continues to develop locally. Xcode makes linking your local and remote updates a piece of cake.

To demonstrate this, you need to make a small modification to one of the files in the project. Open `ViewController.swift` from the Project Navigator, and ensure that you're using the Standard Editor.

There is no need to make a drastic change; the only goal is to differentiate the file somehow from the last committed version. To do this, I added a line to the comments at the top of the file to say that this is the view controller. Modify your file in a way similar to this highlighted code:

```
// ViewController.swift
// HearMeNow
//
// Created by Matthew Knott on 30/10/2014.
// Copyright (c) 2014 Matthew Knott. All rights reserved.
// This is the View Controller
```

After you do that, commit the change to the repository by going to Source Contro ► Commit. Before you click the button to commit the files, look in the bottom-left corner of the commit view and note the Push To Remote check box. Check it, as shown in Figure 12-45, and your GitHub repository automatically appears.



Figure 12-45. Electing to also push to the remote repository

You may notice that the Commit button changes to read Commit 1 File And Push. Add a commit message, as the button suggests, and then click it to see what happens. After a brief pause, the view closes and the operation is complete.

To verify what happened in the remote repository, go back to your web browser and the page for the GitHub repository you created. When you refresh the page, note that, as shown in Figure 12-46, the number of commits is now listed as 4.



Figure 12-46. The number of commits on the repository now shows as 4

To see the level of replication between the local and remote repositories, click the number of commits. You're taken to a page showing the commit history, as shown in Figure 12-47. As you see, it has all the previous commits and their comments exactly as the local repository does.

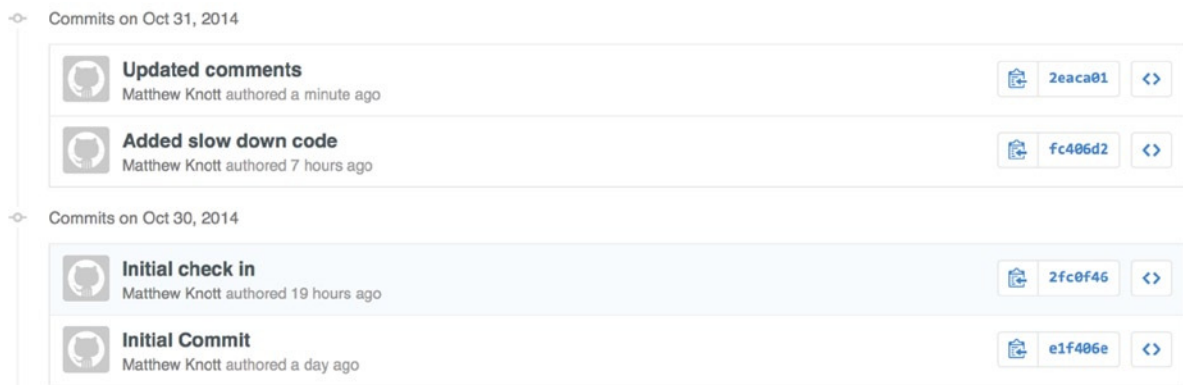


Figure 12-47. The GitHub commit history matches the local repository

Summary

In this chapter, you've learned the ins and outs of software version control in Xcode using Git while writing a basic voice-recorder application. Specifically, you have done the following:

- Created a local repository
- Learned about committing changes to the repository
- Looked at the Version Editor and seen how it allows you to compare files and roll back changes
- Made and merged branches
- Added and used a remote repository

Whatever your hopes for Git, you now have an excellent grounding in all the skills needed to make the most out of it. In the next chapter, you learn about localization with Xcode and how to make your application support multiple languages in a single version.

Localization

Chapter 12 looked at how Xcode uses Git to create some of the finest version control ever seen in an IDE. You created a simple voice recorder under Git source control, branched the repository to add extra functionality without altering the original, and then merged the two branches together.

This chapter looks at the localization of an application. *Localization* is the term given to the ability of a single application to appear in multiple languages. Making your application available in various languages is a key step to maximizing your success on the App Store, because you can offer your app in the native language of every country in which you choose to advertise.

If you talk to a man in a language he understands, that goes to his head. If you talk to him in his language, that goes to his heart.

— Nelson Mandela

Localization works best when the user doesn't even have to make a language selection—the application simply detects the language that the user has set for their operating system and loads the correct language files. In the past, with other programming languages and IDEs, this could create massive development overheads; but developing applications for iOS and OS X with Xcode makes this process simple and requires minimal development. Once you've created your application in one language, you'll be amazed how easy it is to add more.

In this chapter, you create an interesting application called SayMyName. This application uses the Address Book UI framework to let you select a contact from your Address Book and then uses the text-to-speech API in the AV Foundation framework to say the person's name through the device's speakers.

You localize the application and make it available in Spanish. Specifically, you learn about three types of localization: storyboards, images, and strings in code. Figure 13-1 shows the finished application running in both English and Spanish.

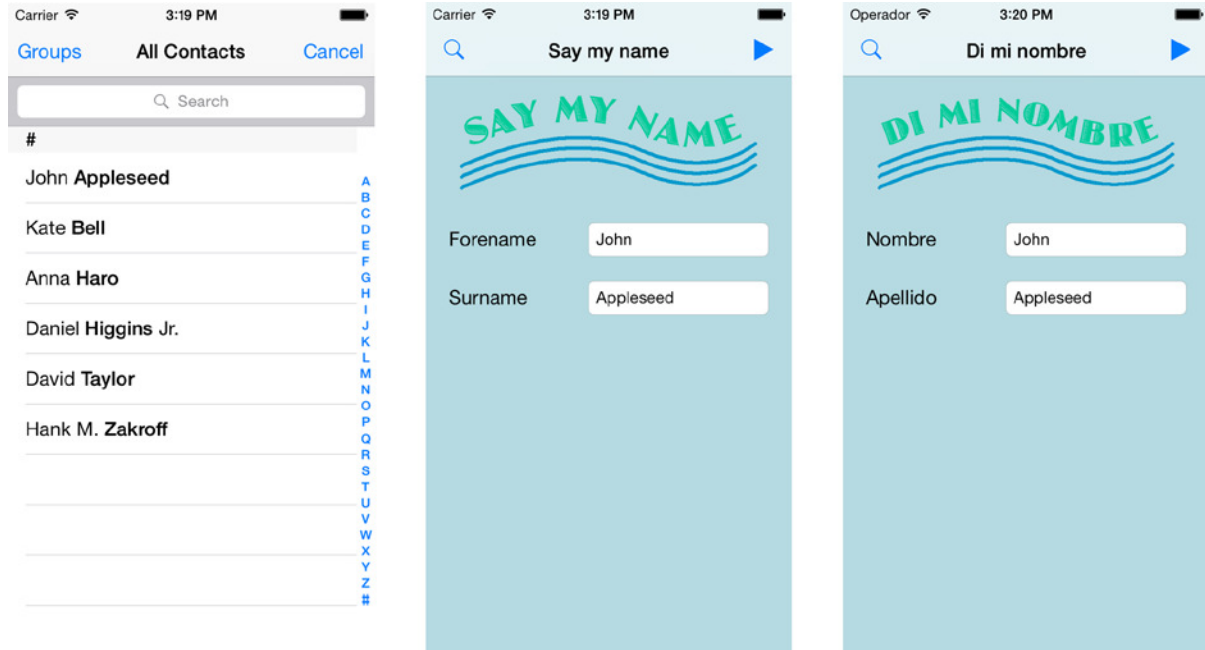


Figure 13-1. The finished application, picking from the Address Book and running in English and Spanish

Another point about the SayMyName application is that it's the last application you create in this book. The final two chapters use this application to explore functionality before you see how to submit it to the App Store, so be sure to keep it handy when you finish this chapter.

Creating the SayMyName Application

One thing that I've always found to be true about localization is that you should do it at the end of the development process. This doesn't necessarily mean you shouldn't write your application with localization in mind, such as by using the `NSLocalizedString` macro for all user-facing strings, but it's not essential. When you're starting out with Xcode, focus on getting your code right and your application working the way you want it. This is the approach you take with the SayMyName application. You write the code in a familiar way and then, when the application is finished and working, you start to localize the strings in your code, images, and storyboard:

1. Open Xcode, and create a new project by going to `File > New > New Project` (`⌘+Shift+N`) or, alternatively, choosing `Create A New Xcode Project` on the Welcome screen (`⌘+Shift+1`).
2. The SayMyName application operates on a single view, so select the `Single View Application` template, and then click `Next`.
3. Name the product `SayMyName`, and ensure that the `Devices` option is set to `iPhone`. Substitute your personal information, leaving the other settings as shown in [Figure 13-2](#), and then click `Next`.

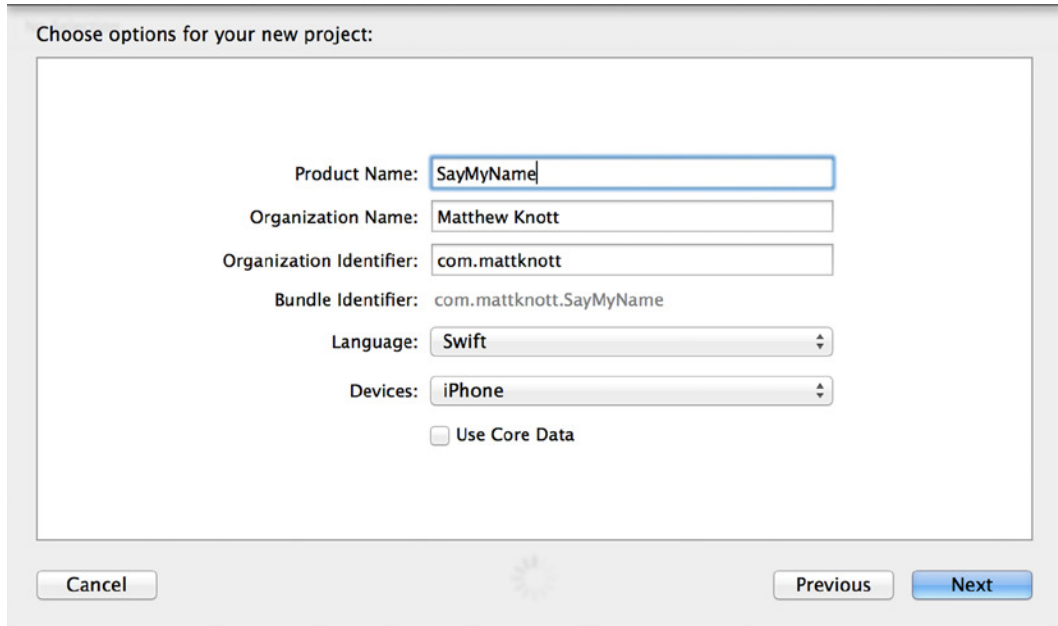


Figure 13-2. Specifying the settings for the new project

4. On the next screen, ensure that the Source Control option is selected, as shown in Figure 13-3, and click Create. Using source control, you can commit the changes once you have a working application and before embarking on the localization of the application, during which it can be easy to break the application or lose some values.

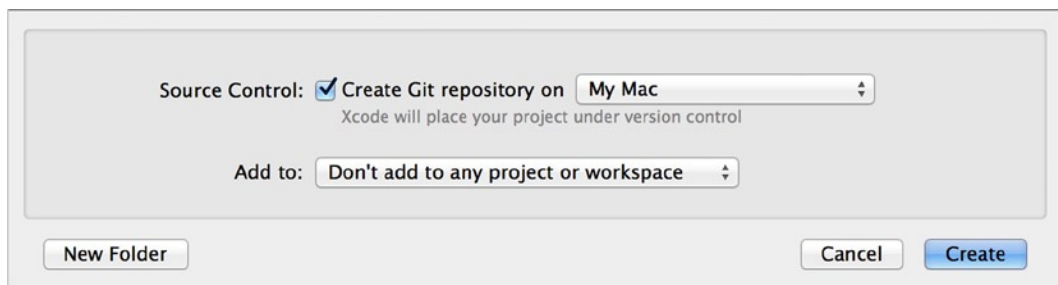


Figure 13-3. Choosing to create a Git repository

5. You're taken to your new project and its settings. Look for the Deployment Info section. Because this application only needs to operate in portrait mode, uncheck the Landscape Left and Landscape Right options, as highlighted in Figure 13-4.

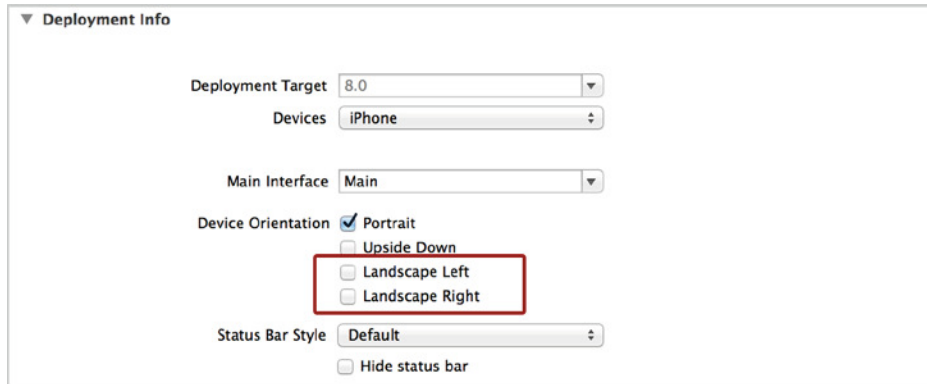


Figure 13-4. Unselect the Landscape options in the project settings

Great! The project is ready to be assembled. But before you do that, there are some pre-prepared resources you need to download and add to the project.

Adding the Resources

Back in Chapter 1, I explained how to download the resources for this book from the Apress web site. For this chapter, I have created a logo in two languages; so, unless you want to create your own, head to www.apress.com/9781430250043. Remember that Chapter 1 shows the process of downloading resources in greater detail:

1. Locate the resources for the chapter in Finder, find `logo.png`, and drag the file into the Supporting Files group, as shown in Figure 13-5.

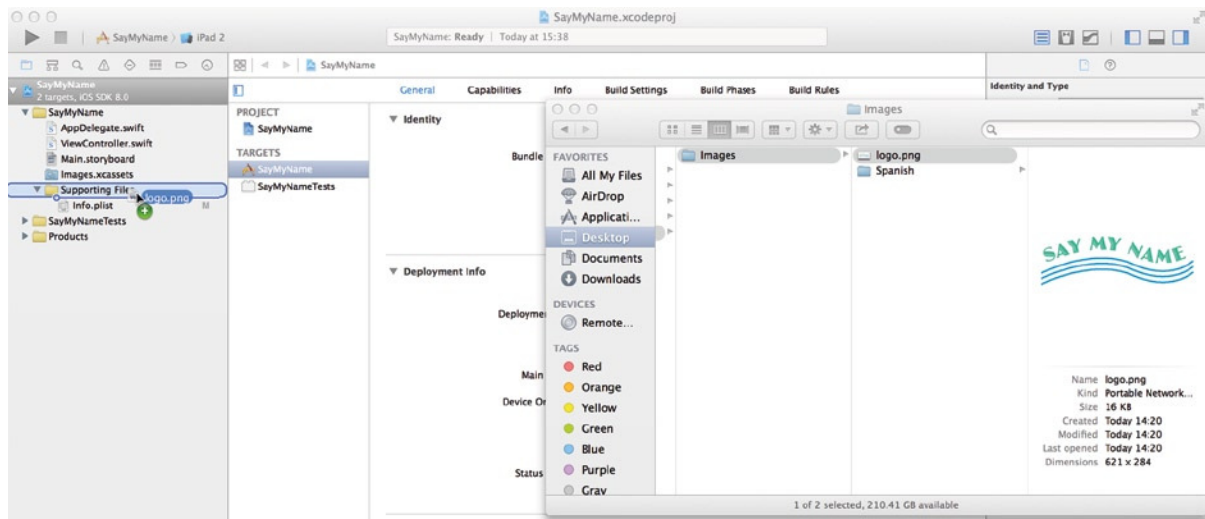


Figure 13-5. Adding the `logo.png` file to the Supporting Files group

2. When you release the image into the Supporting Files group, be sure to choose Copy Items If Needed, as shown in Figure 13-6.

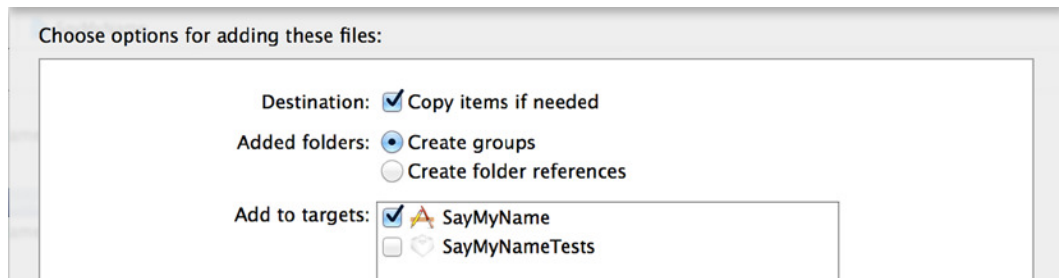


Figure 13-6. Importing the English language logo

That's it: the logo has been added to the project, you're ready to begin building the user interface.

Creating the Application Interface

Now that your project is configured and has all the required resources, all that remains is for you to create the interface and write the code for this application. One of the most important attributes of this application isn't just the experience of making an app that works seamlessly in different languages, but also the way you use the Address Book API to display your Address Book and retrieve information from it. Notably, you don't have to create any view controllers or other interface elements—it's all done with a few lines of code. A great thing about developing for the iOS ecosystem is that Apple provides different hubs of data that are easy to interface with, such as the Address Book, Photo Library, and Calendars.

Laying Out the Views

In Figure 13-1, I gave you a preview of the application's user interface. It consists of a navigation bar, two bar buttons, an image view for the logo, two labels, and two text fields. Select `Main.storyboard` from the Project Navigator to start building the interface:

1. Zoom out your view, and drag a navigation controller onto the design area as shown in Figure 13-7. Positioning isn't too important at this stage.

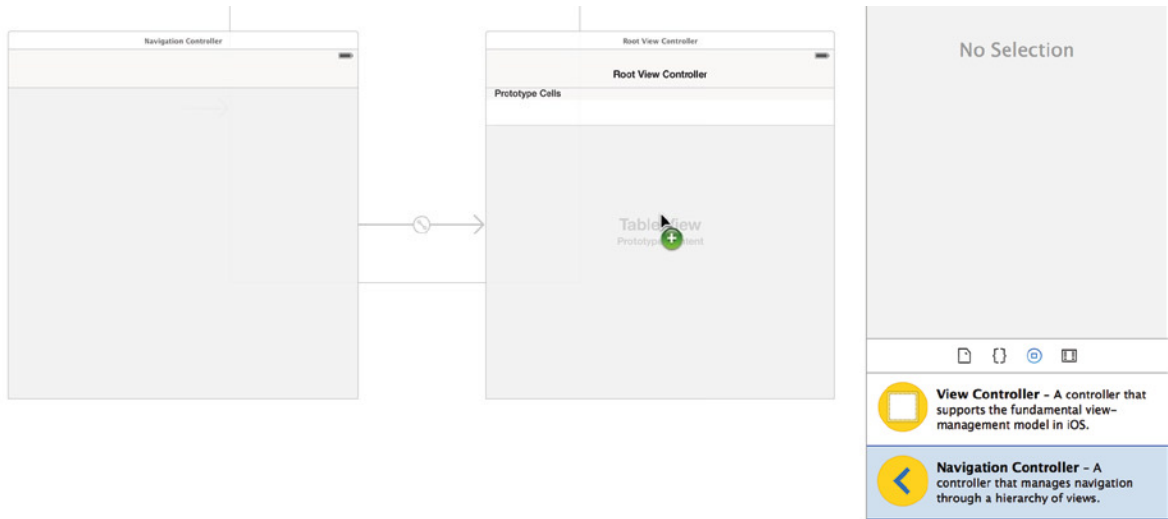


Figure 13-7. Dragging a navigation controller onto the design area

2. Select the root view controller that came with the navigation controller, and delete it using the Backspace key.
3. Position the navigation controller to the left of the view controller in the design area, as shown in Figure 13-8.

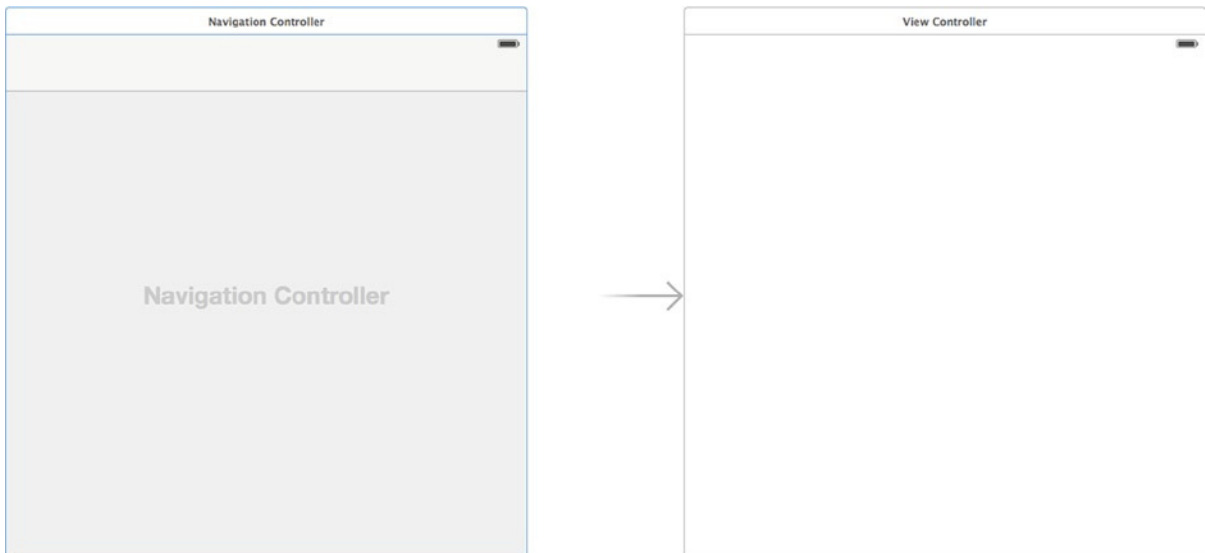


Figure 13-8. The navigation controller beside the view controller

- Control-drag a connection from the navigation controller to the view controller, to link them. When you release the mouse, select Root View Controller under Relationship Segue (see Figure 13-9).

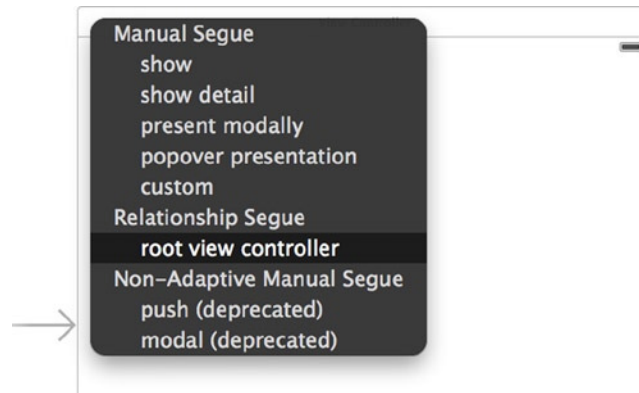


Figure 13-9. Creating a root view controller relationship with the navigation controller

- You need to move the arrow that indicates the initial view controller when the project runs, shown back in Figure 13-8. Click the arrow, and drag it on to the navigation controller. Your design area should now resemble Figure 13-10.

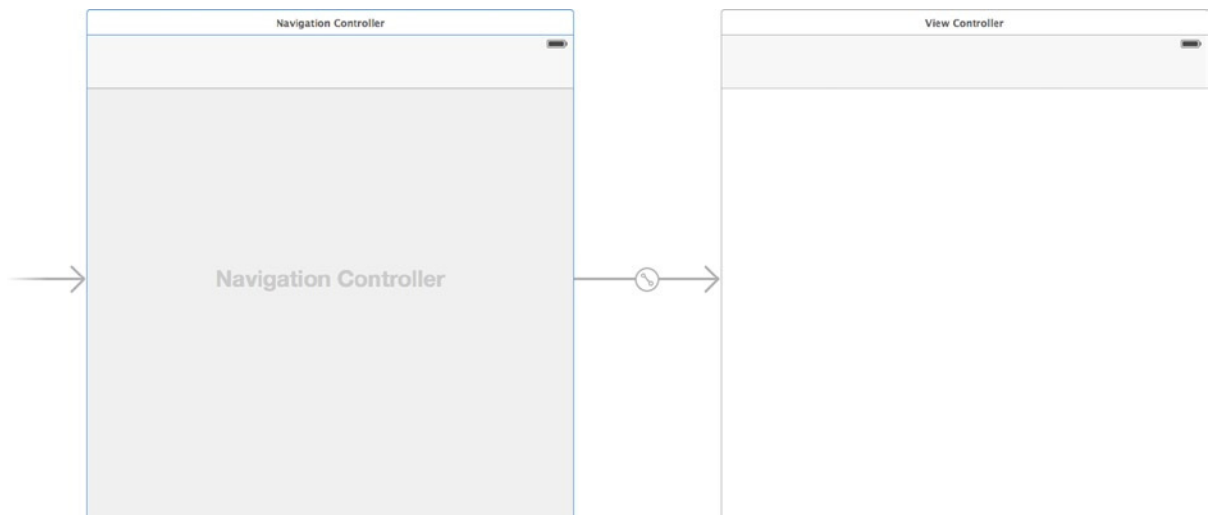


Figure 13-10. The views laid out, ready for the interface to be built

Building the Interface

The interface for this application is straightforward. Just as in Chapters 7 and 8, you use bar button items to control the application. Xcode provides a number of great icons for use in bar buttons; in this case, you use the search icon to trigger the Address Book browse and the play button to make the application say the name of the chosen contact. You also add two text fields to show the selected contact's first and last names, and two labels for each of the text fields:

1. Locate Bar Button Item in the Object Library, and drag one to each side of the navigation controller. Your view controller should resemble Figure 13-11.

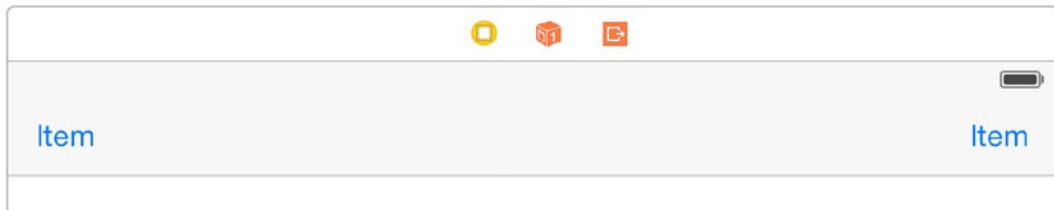


Figure 13-11. The navigation bar with two bar buttons added

2. Select the left button; this one will trigger the display of the Address Book. Open the Attributes Inspector, and change the Identifier attribute to Search.
3. Select the bar button on the right; this one will cause the selected name to be spoken by the iOS device. In the Attributes Inspector, change the Identifier attribute to Play. Your interface is developing nicely and should resemble Figure 13-12.

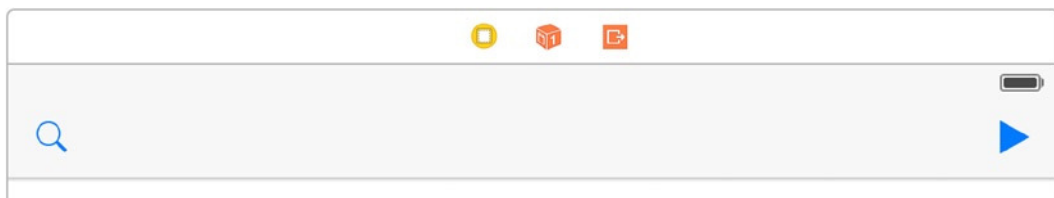


Figure 13-12. The bar button items changed to their respective icons

4. Drag in an Image View item from the Object Library to hold the application logo. Place it just below the navigation bar.
5. Resize the image view so it takes up the full width of the view but isn't too tall, as shown in Figure 13-13.

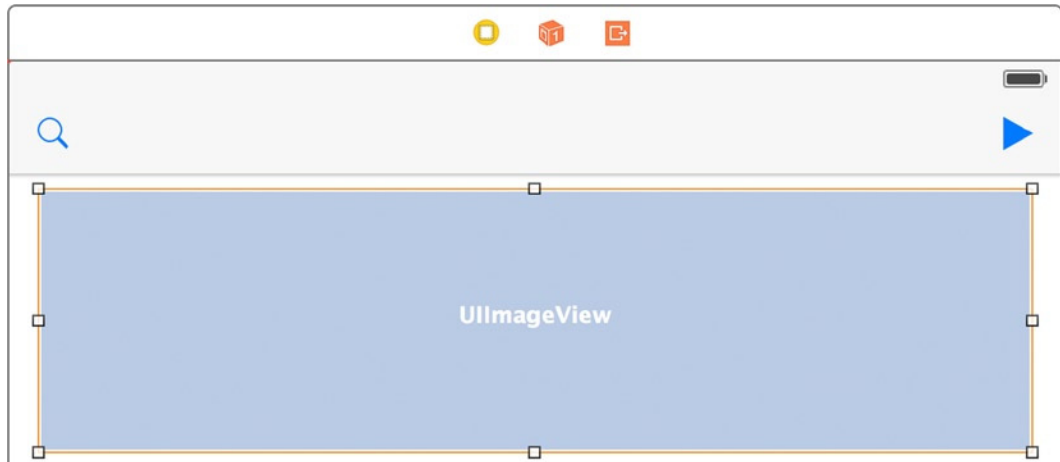


Figure 13-13. Adding the image view to the view

6. In the Attributes Inspector, change the image to `logo.png`.
7. Change Mode to Aspect Fit. Because the logo has text, you want the image to remain fully visible, whatever shape the view takes on; but it should also maintain its shape. This is what Aspect Fit does. Your evolving view should now resemble Figure 13-14.

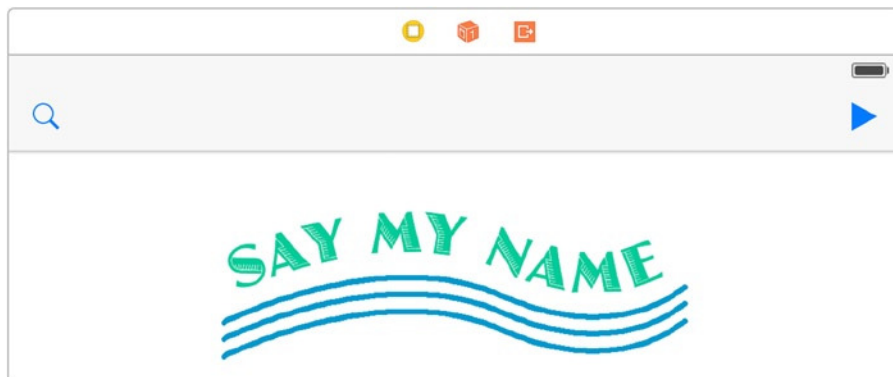


Figure 13-14. The image view positioned and set to Aspect Fit mode

8. You need to add the two labels and two text fields. Refer back to Figure 13-1, and you see that they're positioned side by side. Drag a label onto the view, position it on the left so it fills about one third the width of the view, drag a text view beside it, and expand it to occupy the other two thirds, as shown in Figure 13-15.



Figure 13-15. Positioning the first label and text field

9. Once both elements are positioned, change the label's text to read *Forename*.
10. Repeat the previous two steps, creating the same two elements underneath the Forename row, but this time changing the label text to *Surname*. Your completed views should resemble Figure 13-16.

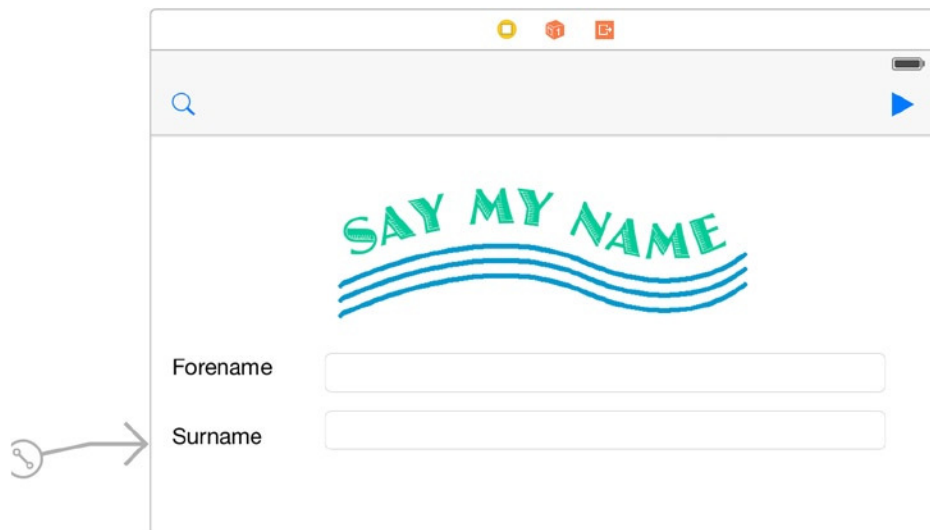


Figure 13-16. The completed view

11. At this point, take a moment to set the constraints for this layout. Click a blank area of the view, and then click the Resolve Auto Layout Issues button. Click Reset To Suggested Constraints under the All Views In View Controller heading. Your interface elements have been created; take a moment to run the application and preview the layout so far. Your running app should resemble Figure 13-17.

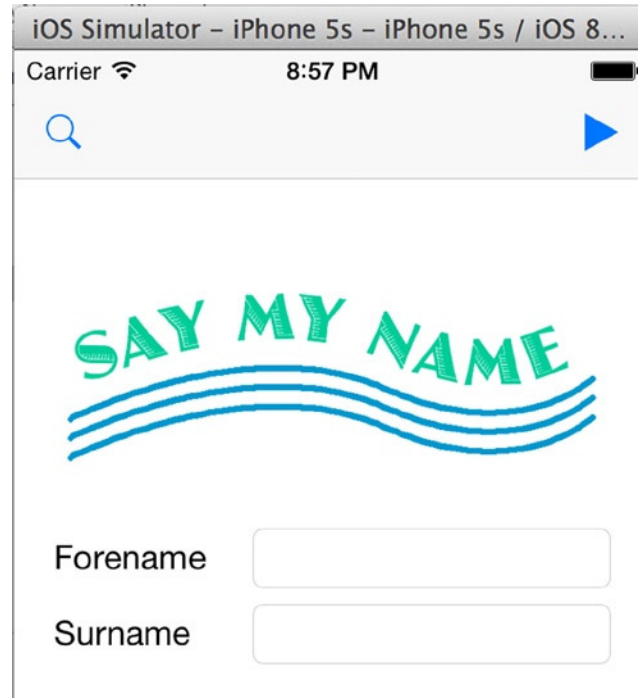


Figure 13-17. The *SayMyName* application running in the simulator

You have completed the interface. It's time to create the outlets and actions for this application before moving on to the code:

1. Switch to the Assistant Editor, and ensure that `ViewController.swift` is being displayed in the Code Editor.
2. Control-drag a connection from the first text field to just below the line that says `class ViewController: UIViewController`. Create an outlet called `forenameField`.
3. Create an outlet for the second text field just below the previous outlet, and call it `surnameField`.
4. Control-drag a connection from the magnifying-glass icon in the navigation bar to just below the last outlet. This time, create an action called `getContact`.
5. Create another action, this time for the play button. Position it just below the `getContact` action, and call it `sayContact`.

Congratulations—you’ve created all the actions and outlets for this application. Before switching back to the Standard Editor, ensure that the start of your view controller’s code resembles the following:

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var forenameField: UITextField!
    @IBOutlet weak var surnameField: UITextField!

    @IBAction func getContact(sender: AnyObject) {
    }
    @IBAction func sayContact(sender: AnyObject) {
    }
}
```

Writing the Code for the SayMyName Application

For this chapter’s application, I’ve already mentioned that you’re using the Address Book UI framework. There are actually two Address Book frameworks: `AddressBook.framework` and `AddressBookUI.framework`. The former framework is used when you want total control over the user experience when presenting Address Book data, or perhaps you want to fetch all of the e-mail addresses in a user’s Address Book to see if any of their friends are members of your social network or web site. The Address Book UI framework gives you access to standardized views and classes for browsing, editing, and retrieving Address Book entries.

Just as in Chapter 5’s project, using a standardized view to perform a task as basic as selecting a contact from an Address Book gives you the benefit of having a view that is common to other applications. This makes it more familiar to the user while also removing the development overhead of updating the view for different devices and, in many cases, across multiple versions of iOS.

In this chapter’s application, you once again use the AV Foundation framework, which you first used in the previous chapter. AV Foundation contains the `AVSpeechSynthesizer` class, which provides the text-to-speech function. Because it’s good practice, you write the code in a separate custom class that you can reuse in any applications where you want to handle spoken text.

Now that you have a brief outline of how to approach the code, let’s get started by implementing the Address Book lookup in the `getContact` action.

Retrieving a Contact from the Address Book

In iOS, whenever you see the Address Book picker that lets you browse your contacts, you’re looking at an instance of the `ABPeoplePickerNavigationController` class. Presenting this view requires several stages. First, because access to the Address Book requires user authorization, you must confirm the status of the application’s access to the Address Book before trying to access it. Second, rather than repeat the code to display the Address Book view, you write a function to do it.

And finally, you implement two delegate functions: one for handling the user's selection, and a second for when the view is cancelled. Here are the steps:

1. Be sure you have the Standard Editor enabled, and then open `ViewController.swift` from the Project Navigator.
2. The first thing you need to do is to make all the Address Book classes available by importing the Address Book UI framework. Add the highlighted code to your view controller:

```
import UIKit
import AddressBookUI
```

3. I've already mentioned that you're using two delegate functions relating to the `ABPeoplePickerNavigationController` class, so the next logical step is to add the `ABPeoplePickerNavigationControllerDelegate` protocol to the `ViewController` class. Do this by adding the highlighted code shown next:

```
class ViewController: UIViewController, ABPeoplePickerNavigationControllerDelegate {
```

4. As mentioned at the start of this section, you're writing a function to display the Address Book people picker. The function is called `showPeoplePicker`: very simply, it creates a new instance of the `ABPeoplePickerNavigationController`, specifies its delegate, and then presents it to the user. Effectively, you're presenting a complete view controller with just three lines of code! After the last action, write the highlighted function as shown next:

```
@IBAction func getContact(sender: AnyObject) {
}
@IBAction func sayContact(sender: AnyObject) {
}

func showPeoplePicker() {

    var picker : ABPeoplePickerNavigationController =
ABPeoplePickerNavigationController()
    picker.peoplePickerDelegate = self

    self.presentViewController(picker, animated: true, completion: nil)
}
```

5. Below this function, you need to create a helper function to obtain a valid Address Book object. Add the highlighted code:

```
func showPeoplePicker() {

    var picker : ABPeoplePickerNavigationController =
ABPeoplePickerNavigationController()
    picker.peoplePickerDelegate = self

    self.presentViewController(picker, animated: true, completion: nil)
}

func obtainAddressbook(addressbookRef: Unmanaged<ABAddressBookRef>!) ->
ABAddressBookRef? {
    if let addressbook = addressbookRef {
        return Unmanaged<NSObject>.fromOpaque(addressbook.toOpaque()).
takeUnretainedValue()
    }
    return nil
}
```

6. Now you're ready to write the slightly more complicated getContact action that calls your new function. The bulk of the code in the action is based around using if statements to determine the current ABAuthorizationStatus of the application. First you check to see whether access has been denied or restricted; if it has, you print a message to the console and continue. Second, you check whether it's authorized, and if so call the new presentPeoplePicker function. And finally, if it's in an undetermined state, you request access to the Address Book and then present the people picker. Add the highlighted code in the getContact action:

```
@IBAction func getContact(sender: AnyObject) {
    if (ABAddressBookGetAuthorizationStatus() == ABAuthorizationStatus.Denied ||
ABAddressBookGetAuthorizationStatus() == ABAuthorizationStatus.
Restricted){
        println("Denied");
    }else if (ABAddressBookGetAuthorizationStatus() == ABAuthorizationStatus.
Authorized){
        self.showPeoplePicker()
    } else { //Undetermined

        var emptyDictionary: CFDictionaryRef?
        var addressBook: ABAddressBookRef?

        println("requesting access...")
        addressBook = obtainAddressbook(ABAddressBookCreateWithOptions
(emptyDictionary,nil))
```

```

ABAddressBookRequestAccessWithCompletion(addressBook, { success, error in
    if success {
        self.showPeoplePicker()
    }
    else
    {
        println("Denied")
    }
    })
}
}

```

Note Rather than returning a result, the `ABAddressBookRequestAccessWithCompletion` method call uses a *Swift closure* to handle the outcome of the request. This is signified with the open brace and the `in` keyword. In Objective-C, this is known as a *block*. If you're referring to the Apple Swift Programming Guide mentioned in Chapter 1, there is a section titled "Closures" that can provide more information.

Take this opportunity to run the application. When it launches in the simulator, click the search icon; immediately you should be prompted to allow access to the device's contacts, as shown in Figure 13-18. When you grant permission, the people picker appears, showing an address book of fictional contacts. You can select any of the contacts and cancel the view controller, which at the moment is the only way you can return to your application. Next you need to make it possible to select one of the contacts and return to the application with that information; this is done using a delegate function.

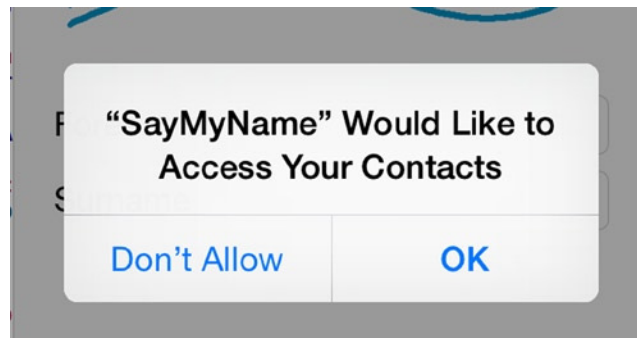


Figure 13-18. Allowing access to the device's contacts

Whenever you tap a contact in the people picker, a call is made to see whether you have implemented the `peoplePickerNavigationController:didSelectPerson:` function. When you implement this function, the Address Book no longer shows you the user's details, but rather returns to your application with the person's information.

Let's implement that function now:

1. Drop down a couple of lines, and type **people**, as shown in Figure 13-19. Code completion appears, with a number of options; highlight the first one, and press the Tab key twice to create the function stub.

```
peoplePickerNavigationController(peoplePicker: ABPeoplePickerNavigationController!, didSelectPerson person
M peoplePickerNavigationController(peoplePicker: ABPeoplePickerNavigationController!, didSelectPerson pers...
M peoplePickerNavigationController(peoplePicker: ABPeoplePickerNavigationController!, didSelectPerson pers...
M peoplePickerNavigationController(peoplePicker: ABPeoplePickerNavigationController!, shouldContinueAfterS...
M peoplePickerNavigationController(peoplePicker: ABPeoplePickerNavigationController!, shouldContinueAfterS...
M peoplePickerNavigationControllerDidCancel(peoplePicker: ABPeoplePickerNavigationController!)
```

Figure 13-19. Using code completion to start the function

One of the parameters passed to the function you just created is an `ABRecordRef` object; this contains all the information about the person the user selected. I won't go into the specifics, but ordinarily you would use the `ABRecordCopyValue` method to return a specific property. Because of the aforementioned issues with unmanaged objects, this is slightly more protracted.

2. Add the following highlighted code to retrieve the first and last names from the supplied person object and then assign those values to their respective text fields:

```
func peoplePickerNavigationController(peoplePicker:
ABPeoplePickerNavigationController!,
didSelectPerson person: ABRecordRef!) {

    if let forename = ABRecordCopyValue(person, kABPersonFirstNameProperty).
takeRetainedValue() as? NSString {
        forenameField.text = forename
    }

    if let surname = ABRecordCopyValue(person, kABPersonLastNameProperty).
takeRetainedValue() as? NSString {
        surnameField.text = surname
    }
}
```

- At the start of this section, I mentioned that you would write two delegate functions. The next one has only a very limited implementation in this application, but it's important that you remember it for your own applications. This function is called when the user cancels the people picker; in this application, that doesn't really matter—the text fields simply remains empty. But in other applications, this could break an entire process, and that possibility needs to be handled. Beneath the previous delegate function, add the following:

```
func peoplePickerNavigationControllerDidCancel(peoplePicker:
ABPeoplePickerNavigationController!) {
    println("Cancelled")
}
```

- Run your application. Click the search button, and select a name from the simulator's Address Book. When you do, the view dismisses itself, and the selected contact's details appear in the text fields, as shown in Figure 13-20.

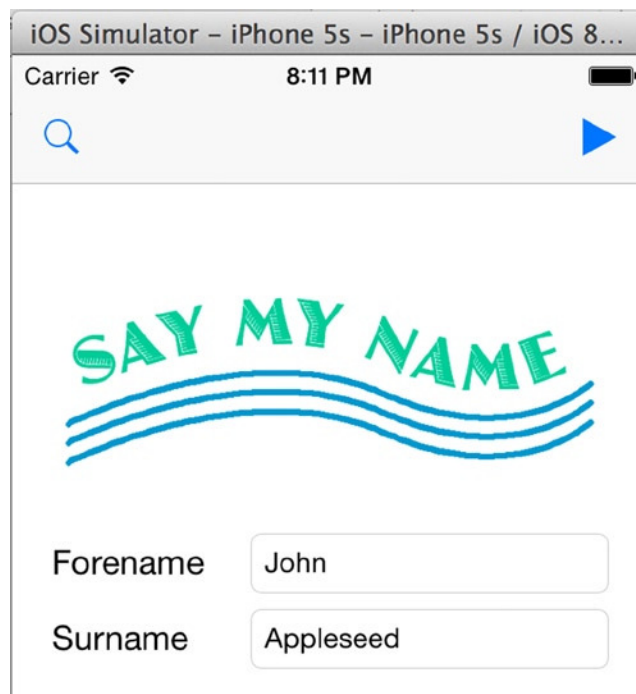


Figure 13-20. The selected name is added to the Forename and Surname fields in the application

Converting Text to Speech

You've already done something amazing by interacting with data held in another iOS application. Now you take it further and add the wow factor by making the device say the name of the person the user chose. This is possible thanks to the `AVSpeechSynthesizer` class in the AV Foundation framework. To make the code you write more reusable, you create a separate class called `TextToSpeech` to encapsulate the type method you write in it. As mentioned in Chapter 5, a type method in Swift is a function that exists in a specific class and that can be called without instantiating the parent class. Here are the steps:

1. Create a new file for the project ($\mathfrak{K}+N$).
2. Select Source under iOS at left, and then choose Cocoa Touch Class. Click Next.
3. The class name is `TextToSpeech`, and you're subclassing the generic `NSObject`. Ensure that your values match those shown in Figure 13-21, and click Next.

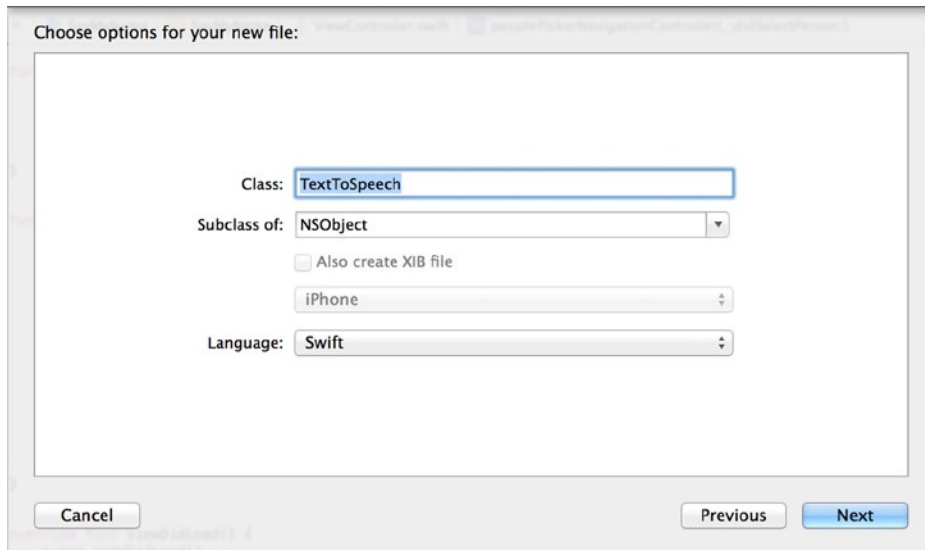


Figure 13-21. Creating the new class, `TextToSpeech`

4. Let Xcode create the new class in the project structure, and click Finish. You're presented with a nice, fresh class.

5. The first step as always is to import any frameworks; and as you know, you need the AV Foundation framework to access the required classes. Import the framework as shown in the highlighted code:

```
import UIKit
import AVFoundation

class TextToSpeech: NSObject {

}
```

6. In this class, you create the type method SayText that takes a single input string called input. A type method is a function prefixed with class. Add the following highlighted code to your class:

```
class TextToSpeech : NSObject {

    class func SayText(input : String) {

    }

}
```

7. To write the code in the function, you create an instance of AVSpeechSynthesizer and an instance of AVSpeechUtterance. The speech utterance object is configured with the input string as a source, and then the pitch and rate of the voice are set before being passed back to the speech synthesizer to be “spoken” by the device. Add the highlighted code to complete this class:

```
class TextToSpeech : NSObject {
    class func SayText(input : String) {
        var synth : AVSpeechSynthesizer = AVSpeechSynthesizer()
        var utterance : AVSpeechUtterance = AVSpeechUtterance(string: input)
        utterance.rate = (AVSpeechUtteranceMinimumSpeechRate) * 0.25
        utterance.volume = 1
        utterance.pitchMultiplier = 1
        synth.speakUtterance(utterance)
    }
}
```


That's it—you've completed the `TextToSpeech` class, and it's ready for implementation. It's not much at the moment, but you can add your own functions and type methods to this class over time to create a useful, reusable library of text-to-speech functions; or you can overload the `SayText` method with other parameters. For now, you call this new method from the `sayPerson` action back in the view controller:

1. Reopen `ViewController.swift` from the Project Organizer.
2. Locate the `sayContact` action. You need to build a string with the person's first and last names in it based on the values of the two text fields; you can then pass the string to the `SayText` method you just wrote. Add the highlighted code to do this:

```
@IBAction func sayContact(sender: AnyObject) {
    var personName = "You have chosen \(forenameField.text) \(surnameField.text)"
}
```

3. You want to pass the `personName` variable to the `SayText` type method in the `TextToSpeech` class. Do this with the following line of highlighted code:

```
@IBAction func sayContact(sender: AnyObject) {
    var personName = "You have chosen \(forenameField.text) \(surnameField.text)"
    TextToSpeech.SayText(personName)
}
```

Note Unlike in other programming languages (including Objective-C), you don't have to import or include the `TextToSpeech` class in order to use it. Swift works hard in the background to simplify and streamline the development process.

At this point, you've completed the base application for this project. Run the application in the simulator, and enjoy the neat little application you've written using only a small amount of code. Next, it's time to get down to the primary purpose of this chapter: localization.

Note At the time of writing, a bug in the simulator prevents this application from working correctly; you see an error in the debug console that says "Speech initialization error." Hopefully Apple will have this resolved shortly; if you experience this problem, you can deploy the application onto a physical device to enjoy it.

Localizing the Application

So, you may be wondering what's involved in taking your single-language application and turning it into a multilanguage super app. It's easy to have a sense of foreboding at this point, but the process is surprisingly simple. With Xcode 6 and iOS 8, Apple has further refined the steps you need to go through to localize an application, making life much easier for you, the developer. But before you dive in, take a moment to think about all the elements in this application that are user facing and need to be translated for each language you localize to:

- Logo image
- Label text
- String passed to the SayText method

Fortunately this application is small, so it won't take a huge amount of effort to localize, but don't think you're missing out on anything. I purposely selected this variety of elements because each one needs to be handled in a completely different way.

Before going any further, I hope you kept Source Control enabled as requested when you created the application. Commit the changes into the repository through Source Code ► Commit (⌘+⌘+C). Add an appropriate comment, and click the Commit Files button.

Enabling Localization

Just as in a number of other development platforms, when you're developing apps for iOS 8 in Xcode 6, localization is already enabled for the default language, which in this case is English. Localization isn't something you strictly enable in Xcode, your application is ready and waiting to be localized; all you need to do is specify which languages you want to make it available in and then add the relevant content.

To understand this a bit better, as shown in Figure 13-22, select the SayMyName project from the Project Navigator in Xcode (step 1), and ensure that the projects and targets list is enabled (step 2) before selecting the SayMyName project from that menu (step 3) instead of the SayMyName target heading.

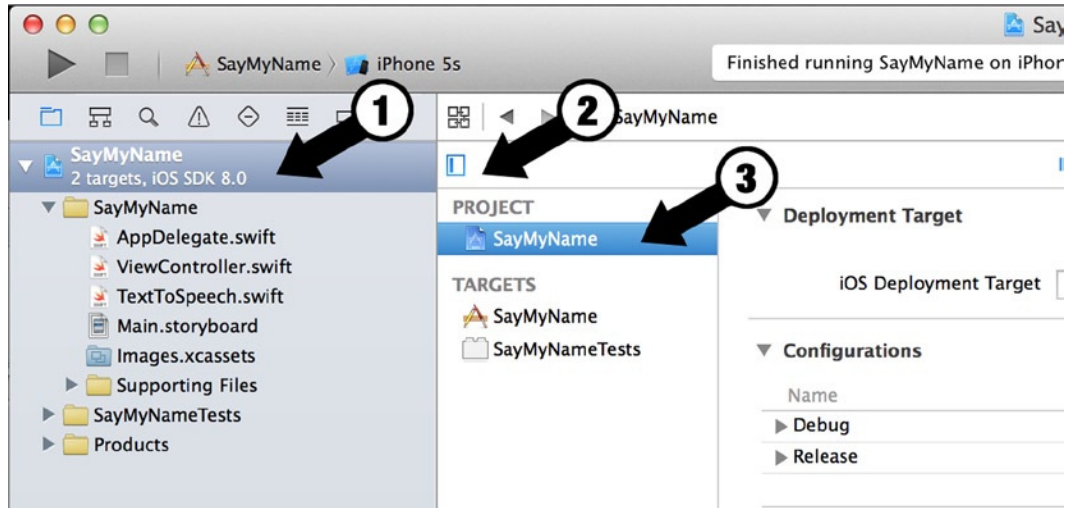


Figure 13-22. Accessing the project settings

In the project settings, and specifically the default Info tab, you find all the details about the different languages that have been added to the project in the Localizations section, as shown in Figure 13-23.

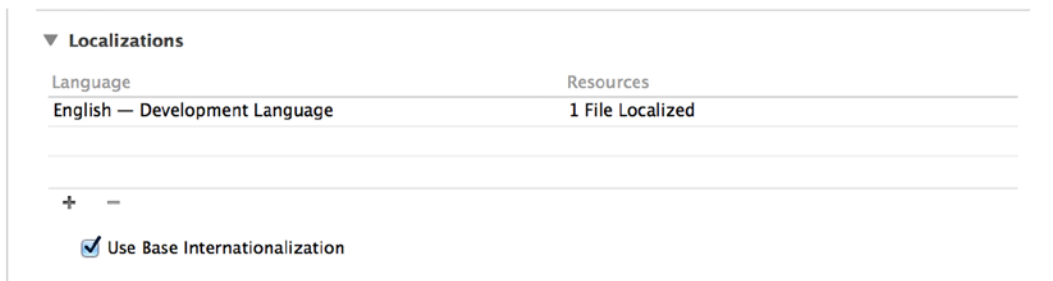


Figure 13-23. The default localization settings

There are two things to note here. First, there is a single language: English. Second, Base Internationalization is enabled. The base internationalization feature isn't unique to Xcode, but it's a great concept. Basically, a base language is your primary language and a fallback for the application; in a situation where a value hasn't been translated to one of the other localizations, the application falls back on the base language to determine the value to display. This obviously isn't ideal but is better than an exception.

When developing an application that will be localized, you must ensure that every string and value has been set at a base level, which is why I encourage you to write the application first before starting translations. Once you're happy with your base localizations, that's when you move on to adding other languages to the project.

For the sake of building progressively toward the end goal of a localized application with more than one language, in this example you create the strings for the base language at the same time as the Spanish translation in some sections. Ordinarily you would get your base translation right before adding any additional languages.

Adding Another Language

Now that you understand what the base language is, it's time to add another language to the project. As I've already stated, the second language is Spanish:

1. To add a new language, click the + symbol at the bottom of the Localizations section shown in Figure 13-23. When you do, a list of available languages appears, as shown in Figure 13-24.

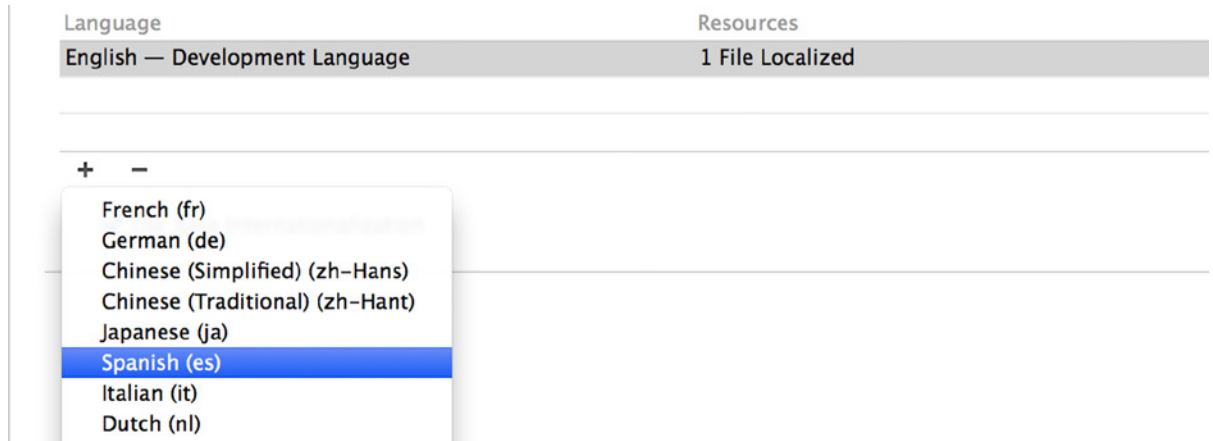


Figure 13-24. Selecting the Spanish language to add to the project

2. Select Spanish from this list. A pop-over appears, as shown in Figure 13-25, listing all the files that will be localized, their reference language, and, if applicable, the destination file type.

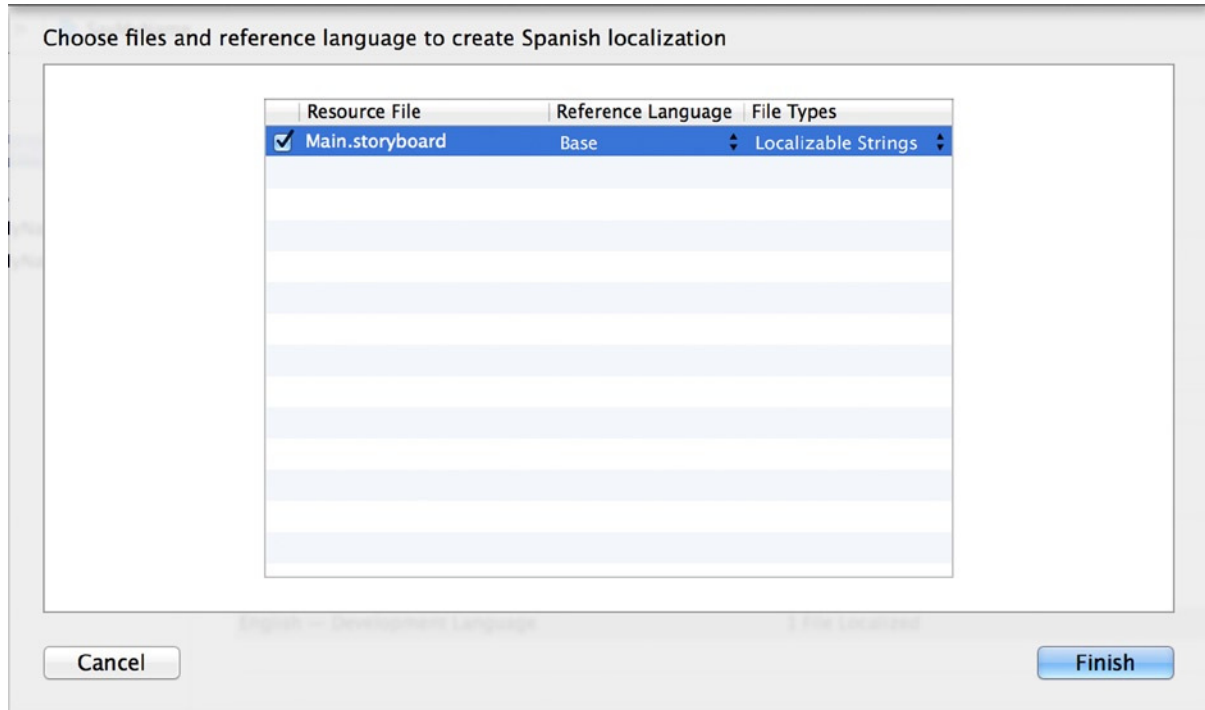


Figure 13-25. Choosing the files to localize to Spanish

This is the most important part when adding a new language. You must ensure that it's using the correct reference language for each resource file. In this case, the files are sourced from the only languages they're available in; but if you're adding multiple languages, you have a degree of variability, so be sure the settings are correct.

The `Main.storyboard` row has a destination File Types option that can be changed from the default Localizable Strings option to Interface Builder Storyboard. The following descriptions will help you understand the differences:

- **Localizable Strings:** Selecting this option means you only have a single base storyboard. Each additional language uses a strings file that holds the translation for every element in the storyboard rather than a completely separate storyboard, which can make changing the storyboard needlessly complicated.
- **Interface Builder Touch Storyboard:** This option creates an entirely separate storyboard that is unique to this locale. Ordinarily you wouldn't do this because of the administrative headache you would be creating for yourself. There are occasions where this option might be beneficial: for example, if you needed the application to be pieced together in a drastically different fashion for the new language area.

Based on these explanations, you definitely want to leave the file type as Localizable Strings, which is the first area you're translating once the Spanish localizations are added.

3. The default options are fine, so click the Finish button. You're returned to the project settings, and two languages appear in the list, as shown in Figure 13-26.



Figure 13-26. The list of localizations now features the Spanish language

Storyboards and Localization

The additional language that has appeared in your project settings wasn't the only change that happened when you added the Spanish language. If you look at the Project Navigator, notice that `Main.storyboard` now has a disclosure indicator next to it. Click the arrow, and you see that it's hiding `Main.storyboard (Base)` and `Main.strings (Spanish)`, as shown in Figure 13-27: the base storyboard is a reference to the true storyboard, and the `Main.strings` file contains all the strings in the storyboard. The more languages you add, the more `.strings` files appear here.

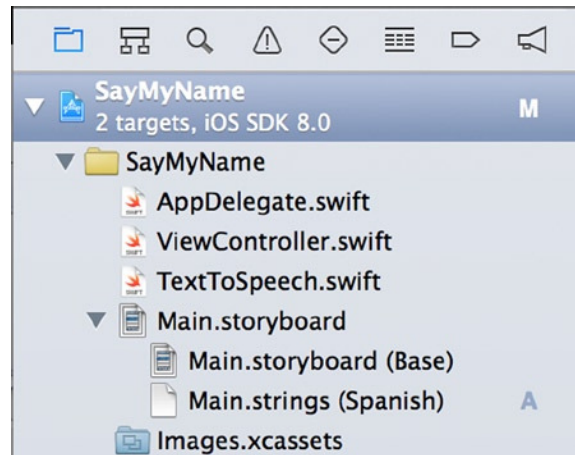


Figure 13-27. Expanding the disclosure indicator next to `Main.storyboard` exposes the localization strings

Click `Main.strings` to examine its contents. Your file should contain several lines similar to the following:

```
/* Class = "IBUILabel"; text = "Forename"; ObjectID = "Bvm-ex-Xm9"; */
"Bvm-ex-Xm9.text" = "Forename";

/* Class = "IBUILabel"; text = "Surname"; ObjectID = "vLt-FM-ItN"; */
"vLt-FM-ItN.text" = "Surname";
```

The first line is a comment, and the second is a key/value pair that links the translation to the label's text. The first part of the key `Bvm-ex-Xm9` points to the label's unique object ID, which is automatically assigned when the label is added to the storyboard. The second part, `text`, indicates that the string is the text attribute for the label.

Note Your label object IDs will be different from mine because the value is randomly generated when the object is added to the storyboard.

The value of `Forename` is taken directly from the base translation and is the first part you need to change. Change `Forename` to `Nombre` and `Surname` to `Apellidos`, as in the following code:

```
/* Class = "IBUILabel"; text = "Forename"; ObjectID = "Bvm-ex-Xm9"; */
"Bvm-ex-Xm9.text" = "Nombre";

/* Class = "IBUILabel"; text = "Surname"; ObjectID = "vLt-FM-ItN"; */
"vLt-FM-ItN.text" = "Apellidos";
```

You have translated two values in the user interface, but you won't preview these changes until all the other elements in the application have been translated. Next you localize the application's logo.

Localizing Images

Hopefully you're starting to sense that localization isn't as daunting as you may have feared. You've successfully added a new language and a couple of translated strings into the storyboard strings files to change the text on your button. It's time to localize the `logo.png` file, which is straightforward because you *didn't* use an asset catalog to store the image.

Asset catalogs are great, but they haven't been integrated into the localization system as well as other elements in Xcode. Although asset catalogs can be duplicated, it isn't worth it when you have a single image in your catalog that needs a localized version. This is because the process to localize the catalog involves duplicating it in its entirety, which isn't practical. The easiest way to localize an image in Xcode 6 is to add it to the Supporting Files group, as you've already done:

1. Expand the Supporting Files group in the Project Navigator, and select `logo.png`.
2. Open the File Inspector by going to `View > Utilities > Show File Inspector` (`⌘+⌘+1`). Contained in the File Inspector sidebar is a button called `Localize`, as shown in Figure 13-28.

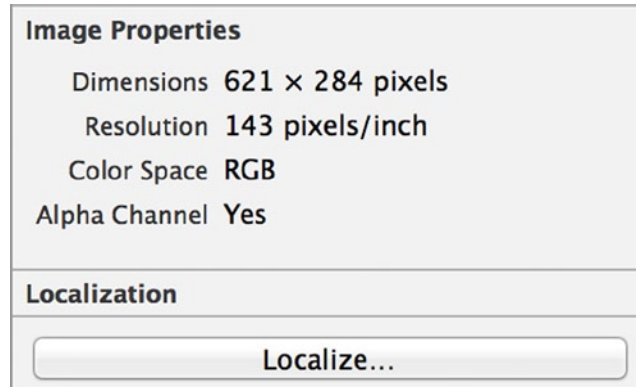


Figure 13-28. The Localize button in the File Inspector

3. Click the Localize button, and you're prompted for which language you wish to localize to, as shown in Figure 13-29. Leave the selection as the default base language, and click Localize.

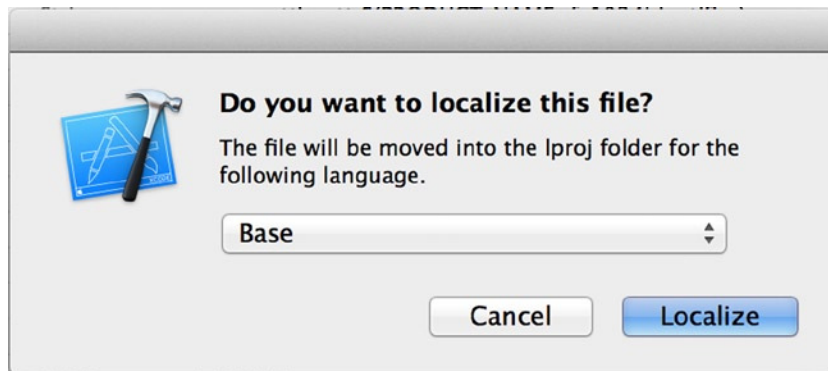


Figure 13-29. Selecting the Base language as the target for localization

4. When the dialog has closed, the area that contained the Localize button in the File Inspector changes to reflect the languages available for localization. Select Spanish, as shown in Figure 13-30. There is no need to select the English box, because English is also the base language, so you would be doubling up needlessly.

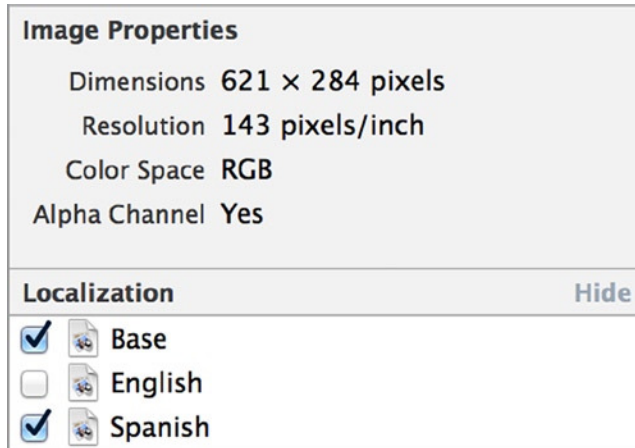


Figure 13-30. *logo.png* can be localized to any available language

5. If you look at `logo.png` in the Project Navigator, just as with the storyboard, there is now a disclosure indicator alongside the image. Expand it. As shown in Figure 13-31, you now have two versions of the image: the base version and a Spanish one.

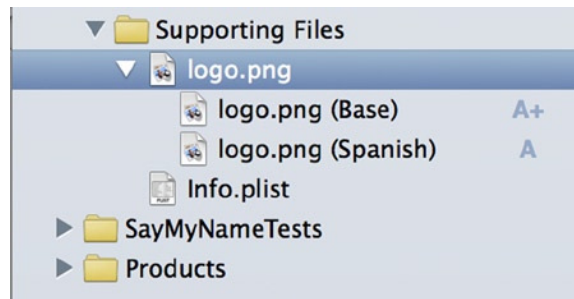


Figure 13-31. The *logo.png* file now has two language versions

A Peek Behind the Scenes

As with many things, Xcode does a great job of creating the appearance of simplicity on top of what is going on behind the scenes in your project, presenting you with a nice, easy-to-use interface. To see what Xcode is really doing, right-click the project name in the Project Navigator in Xcode, and select Show In Finder.

As you can see in Figure 13-32, if you select the SayMyName folder, there is an `es.lproj` folder in addition to the `base.lproj` and `en.lproj` folders that you normally find in an iOS 8 project folder. In this folder are all the files that are localized to Spanish, including `logo.png`. It's here that you need to replace the overlay image, currently written in the English language, overwriting it with a Spanish version:

1. Open an additional Finder window. To do this, in Finder, go to File ► New Finder Window (⌘+N).
2. In the new Finder window, navigate to the resources for this chapter and open the Spanish folder, which contains a single file: `logo.png`. Drag this file over to the `es.lproj` folder, as shown in Figure 13-33.

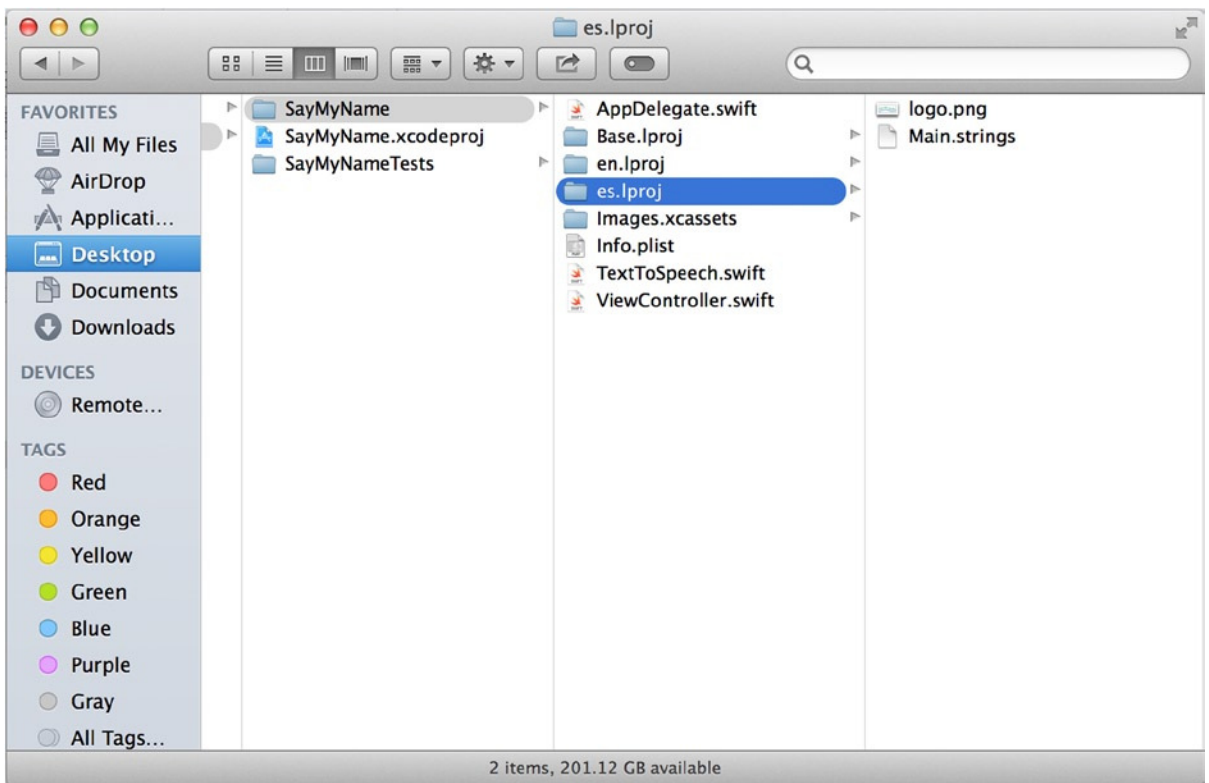


Figure 13-32. Behind the scenes, the SayMyName project folder has a folder for each localization

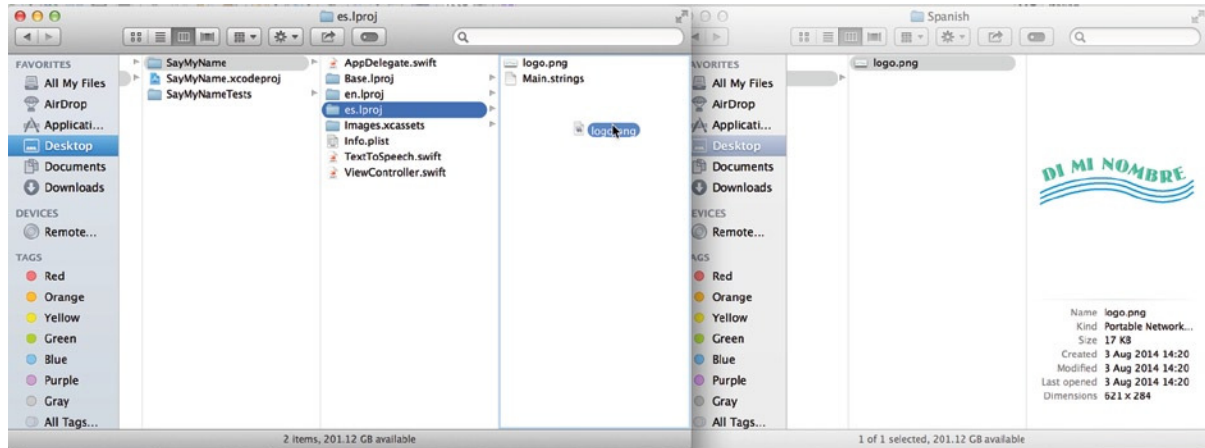


Figure 13-33. Dragging the Spanish `logo.png` over to the `es.lproj` folder

3. When you release the file, you're told that the file already exists and asked how you want to handle the duplicate files. Choose Replace.

You have now learned how to localize a storyboard and a project resource, in this case an image file. All that remains is to look at localizing the string that is passed to the `SayText` method; then you'll be ready to explore previewing and texting localization.

Localizing Code with Localizable.strings

The last area of application localization that I cover in this chapter is code localization. When localizing code, you don't have to translate all of it—just the strings used in the code. Here I have highlighted the string you construct in the `sayContact` action:

```
@IBAction func sayContact(sender: AnyObject) {
    var personName = "You have chosen \(\forenameField.text) \(\surnameField.text)"
    TextToSpeech.SayText(personName)
}
```

As you can see, three strings need to be translated. There are a number of ways to localize these strings, but all of them use the same method of retrieving the localized string: `NSLocalizedString`. In this instance, you're using `NSLocalizedString` to retrieve a localized string value from a file you haven't created yet: `Localizable.strings`.

Creating Localizable.strings

The Localizable.strings file, like the Main.strings file for the storyboard, holds localized strings in a key/value format. Unlike Main.strings, Xcode hasn't created this file for you automatically. Creating a strings file is a quick and easy task, so let's take a moment to create the file:

1. Select the Supporting Files folder in the Project Navigator, and then go to File ► New ► File (⌘+N).
2. When the New File pop-over appears, select the Resource category under iOS on the left, and then choose the Strings File template on the right, as shown in Figure 13-34. Click Next to continue. (You may have to scroll down to see the Strings File template.)

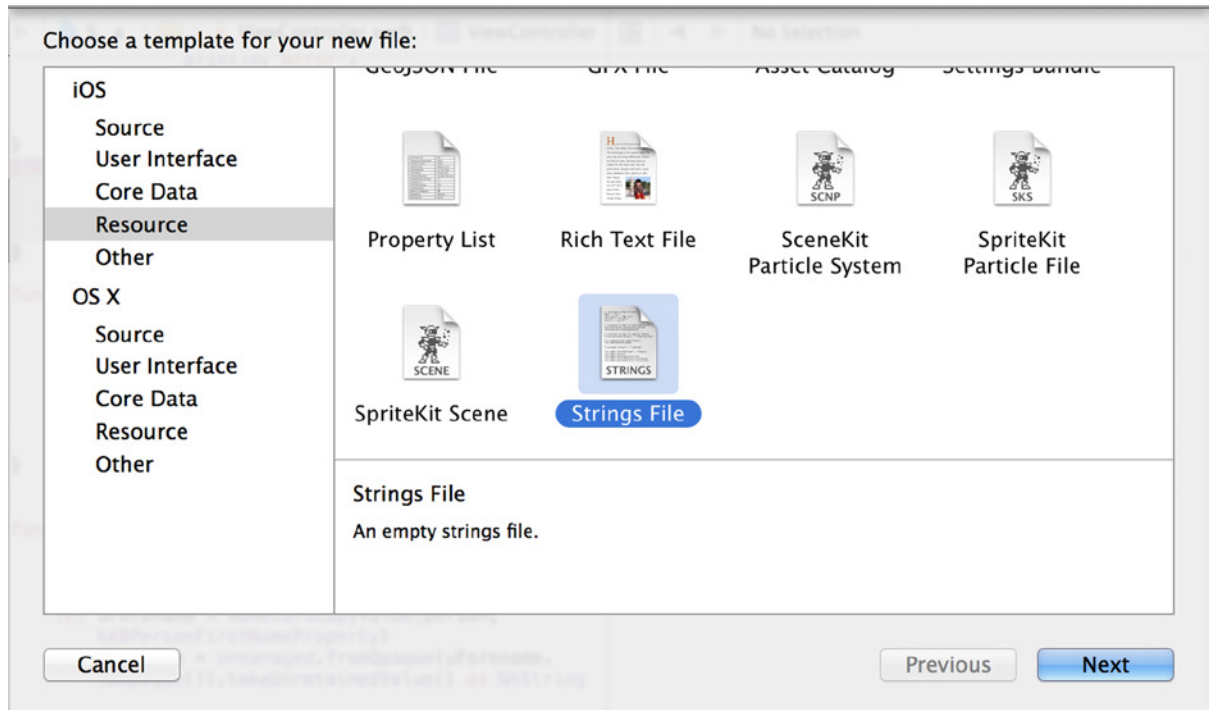


Figure 13-34. Selecting the Strings File template from the Resources category

3. You need to name the file correctly in the Save As box. Be sure you name it Localizable, as shown in Figure 13-35. Click Create to add the new strings file to the project.

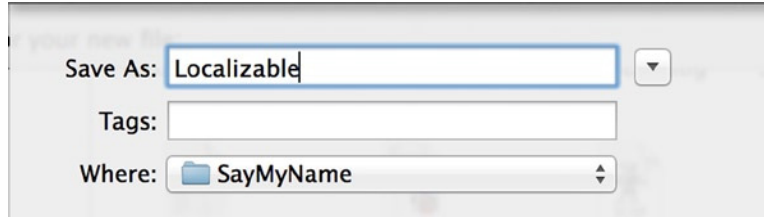


Figure 13-35. Naming the new strings file

4. In your Project Navigator, you should see a `Localizable.strings` file. Select the file, and it appears in the code editor. Currently the file is empty, apart from the opening comments. Beneath the comments, add the following code:

```
"SELECTED" = "You have chosen %@ %@";
```

The first item in quotes, `SELECTED`, is the *key*. This is the value you reference in your code to retrieve the *value*, which is the item on the right of the equals symbol. Although there isn't a specific convention for naming keys, it's good to think semantically and name the key as either a whole word or a part of the sentence. The `%@` symbols are placeholders that you replace with the first and last names later.

Note It's important to remember that `.strings` files aren't Swift code, and as such they require you to end each line with a semicolon. Without a semicolon, the application won't compile.

You've successfully created a string for the base translation. Now you need to localize this file and translate it.

Localizing `Localizable.strings`

When you created `Localizable.strings`, it wasn't associated with any language. In order to have a language-specific version of the file, you need to localize the file just as you did with the overlay image:

1. Select the `Localizable.strings` file in the Project Navigator, and then open the File Inspector by going to `View > Utilities > Show File Inspector` (`⌘+⇧+1`).
2. Look for the Localize button in the File Inspector (refer to Figure 13-28 if you can't find it), and click it. As before, leave the language as Base, and click Localize.

3. Again, the Localize button is replaced with a list of available languages with check boxes. Select the Spanish check box, as was shown in Figure 13-30. This creates a base version and a Spanish version of the file.
4. Expand the disclosure indicator next to Localizable.strings in the Project Navigator, as shown in Figure 13-36, and select the Spanish version.

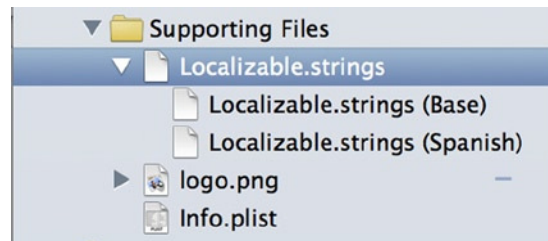


Figure 13-36. Displaying the base and Spanish versions of Localizable.strings

5. In the code editor, you should find that the contents are exactly the same as the base version of the file. Change the values highlighted in the following code to their Spanish counterparts:

```
"SELECTED" = "Has elegido %@ %@",
```

Great! You now have base and Spanish versions of your Localizable.strings file, fully translated and ready to go. Next you need to work the contents into your application's code.

Retrieving Localized Strings with NSLocalizedString

With the localized strings in place for both languages, all that remains is to retrieve the values from the Localizable.strings file using the NSLocalizedString macro. To do this, you replace the static strings that you've already written with the NSLocalizedString macro. NSLocalizedString does the hard work in retrieving the correct strings for the selected language, meaning all you as the developer need to do is supply the correct key.

NSLocalizedString takes a number of arguments, but it requires you to supply it with a minimum of two specific arguments: the key that corresponds to the string you want to display, and a comment that allows you to add context but is completely optional. I provide a string, but you can just as easily specify nil.

Open ViewController.swift from the Project Navigator, and scroll down to the sayContact action. Replace the highlighted code as shown here:

```
@IBAction func sayContact(sender: AnyObject) {
    var personName = String(format: NSLocalizedString("SELECTED", comment: "Selected Person"),
        forenameField.text,
        surnameField.text)
    TextToSpeech.SayText(personName)
}
```

That was easy. You've replaced the string that was there with a call to the `NSLocalizedString` macro, so whenever your application runs, the value from the `Localizable.strings` file will be displayed. It's also worth noting the significance of the `%@` symbols in the strings file and the string formatting. The `%@` symbols are placeholders, and when wrapped with the string formatter, they're replaced.

Let's see how this would look if `NSLocalizedString` hadn't been used:

```
String(format: "You have selected %@ %@", forenameField.text, surnameField.text)
```

Essentially, when the formatter encounters the first string placeholder, it looks to see what value to insert: in this case, the value of `forenameField.text`. When it encounters the second placeholder, it moves on for the second string, `surnameField.text`. Admittedly this isn't as neat as the Swift statement that was there before, but it's the only way to add values to the placeholders at the correct point depending on the language.

Now you're ready to test the application. To do that, you have a couple of tools at your disposal.

Testing Localizations

Once you begin to localize an application, it's a good idea to test it regularly to make sure everything is still as it should be. You may be dealing with a large number of resources and strings, so don't underestimate the potential for making a mistake. Xcode provides a couple of ways you can test your application, depending on the type of localization you want to test.

Testing Localization with Xcode 6

Xcode allows you to test your interface localizations in real time as you build your interface by using the preview option in the Assistant Editor. This is a quick way to ensure that your interface stands up to changing string lengths without constantly recompiling the application:

1. Select `Main.storyboard` from the Project Navigator.
2. Select the Assistant Editor.
3. Click Automatic on the jump bar, and move down to Preview. Select `Main.storyboard`, as shown in [Figure 13-37](#).

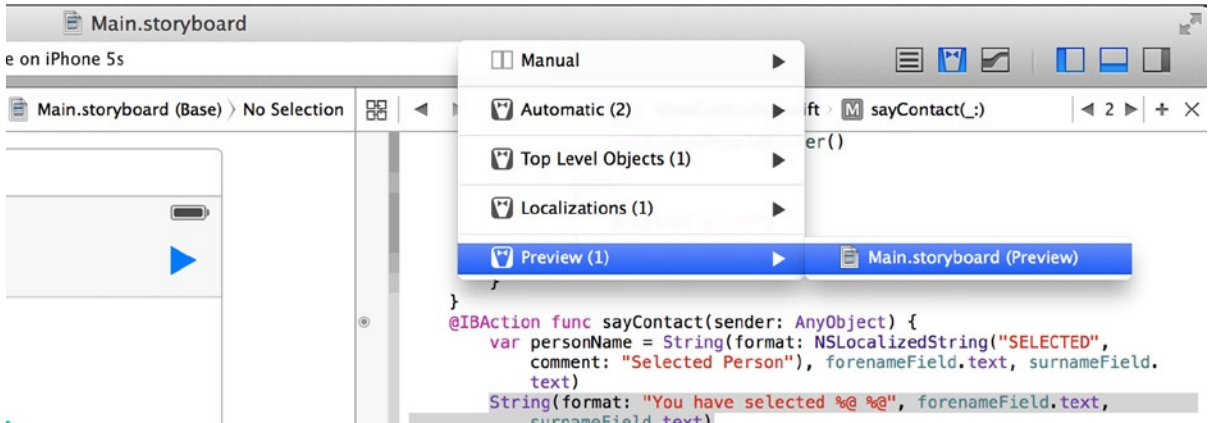


Figure 13-37. Selecting the preview for the storyboard

4. Select the view controller in the design area. The preview shows your interface as it would appear on a 4-inch iPhone, as shown in Figure 13-38.

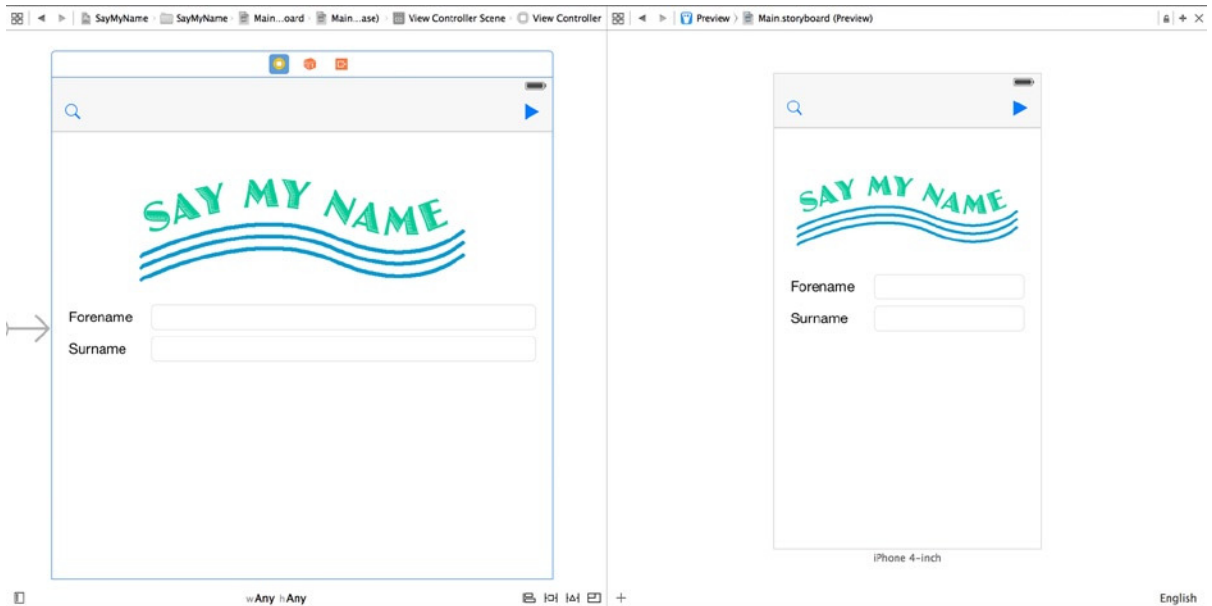


Figure 13-38. Previewing the SayMyName application

5. In the lower-right corner of the preview area is a language selection, currently set to English. Click it, and a menu appears, as shown in Figure 13-39.

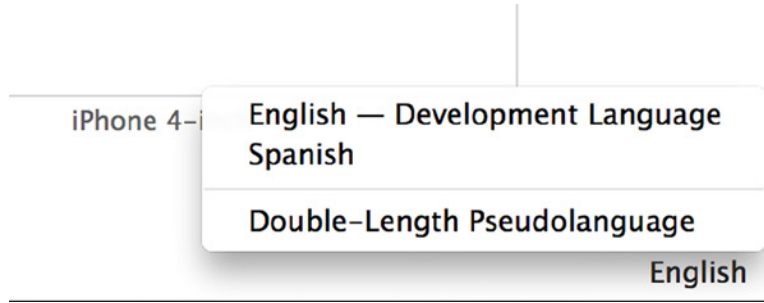


Figure 13-39. Selecting a different language for the preview

6. Choose Spanish from the language selection. As shown in Figure 13-40, the strings are translated in the preview area.

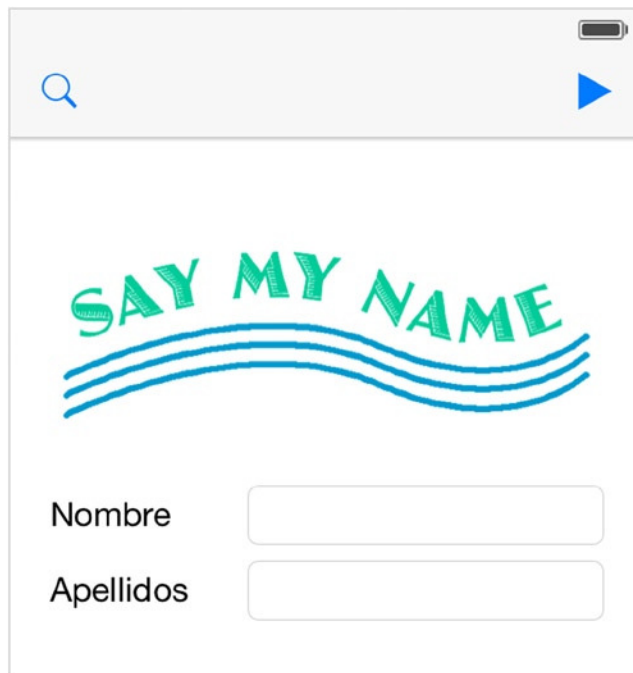


Figure 13-40. Translated strings in the preview area

You're lucky that the translations you're using fit nicely into the labels you created. To test how they might behave with longer strings, the language menu has another option, Double-Length Pseudolanguage, that doubles the length of the base language string for testing purposes.

7. Bring up the language selection as you did in Figure 13-39, and select the Double-Length Pseudolanguage option. As you can see in Figure 13-41, the label's text is truncated.

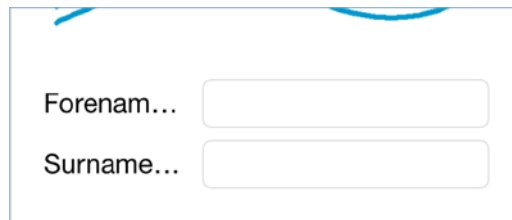


Figure 13-41. The strings truncate when switched to the Double-Length Pseudolanguage option

8. There are a couple of ways to fix this, but the quickest is to enable autoshrink for the field. Click the Forename label in the design area, and open the Attributes Inspector.
9. Locate the Autoshrink attribute, and change it from Fixed Font Size to Minimum Font Size, as shown in Figure 13-42.

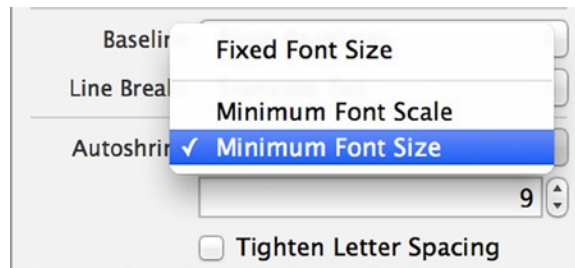


Figure 13-42. Setting a minimum font size

10. Click back into the preview. Now the label text fits perfectly. Repeat this step for the Surname field.

When autoshrink is enabled, iOS shrinks the font size no smaller than the specified minimum until it can fit into the label. If the string is still too long, it's truncated. At times like that, you might want to set up a two-line label as you did when preparing the prototype cells for SocialApp back in Chapter 8.

In Preview, you can't see localized images, and you can't experience the localized string passed to the `sayText` method. For this, you need the simulator.

Testing Localization in the Simulator

In order to fully test your translations, you need to change the language in the simulator or on a physical device to the one you're localizing to. This is done through the Settings application:

1. Stop your application if it's running, and switch back to the simulator. To get to the home screen, go to Hardware ► Home (⌘+Shift+H), and then select the Settings icon, as shown in Figure 13-43.



Figure 13-43. The icon for the Settings application

2. To access the language selection, choose General ► Language & Region ► iPhone Language, as illustrated in Figure 13-44. For the language selection, choose Spanish. You see a message in Spanish and are taken back to the home screen.

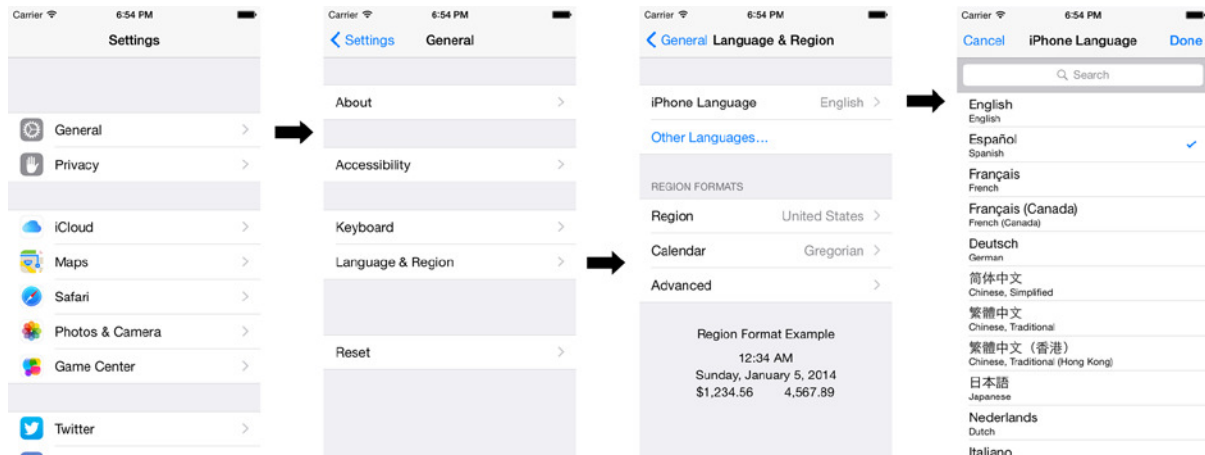


Figure 13-44. Accessing language selections by choosing General ► Language & Region ► iPhone Language

3. Go back into Xcode, and run your application. The entire interface is in Spanish, as shown in Figure 13-45!

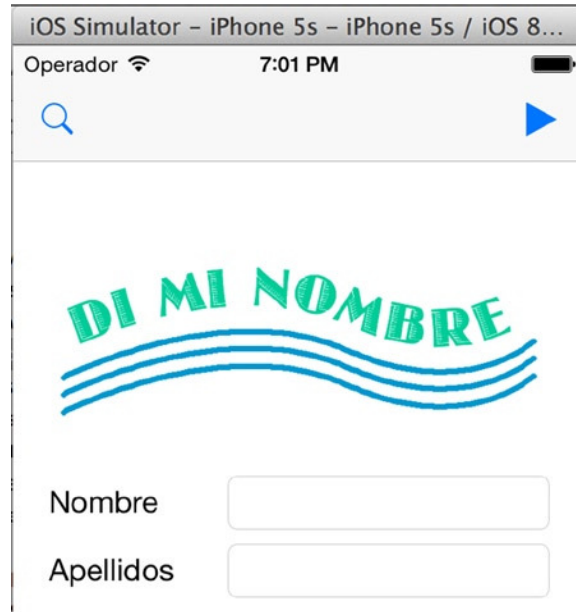


Figure 13-45. Your application, now in Spanish!

Run through the application, select a contact, and click the play button. Not only is your spoken string the Spanish one you created using placeholders, but iOS speaks it in a Spanish accent! This shows the fantastic level of localization that can be achieved with an application in iOS—and writing with Swift makes it even easier.

Setting the Application Language in the Scheme

You’ve just seen how to change the language of the device through the Settings app. There’s actually a much easier way to do this; in Xcode 6, Apple added a new feature that allows you to change the application’s language by editing the active scheme. Here’s how:

1. Click the SayMyName scheme, and click Edit Scheme, as shown in Figure 13-46.

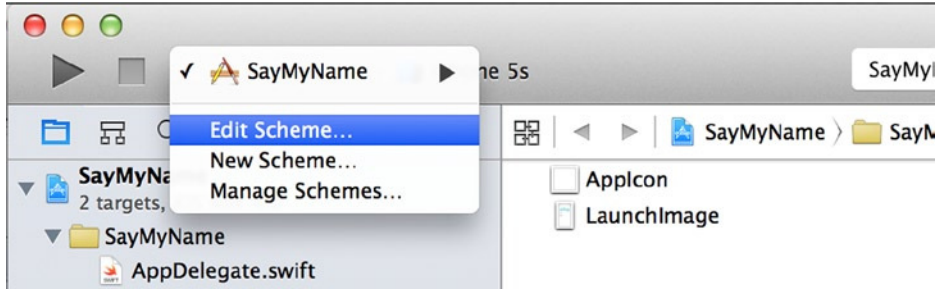


Figure 13-46. Editing the application's scheme

2. Select Run from the column on the left, and then click the Options tab.
3. In this tab, you see an option called Application Language. When you click the drop-down list for this option, you can specify a language to run with, which overrides the settings of the operating system (see Figure 13-47).

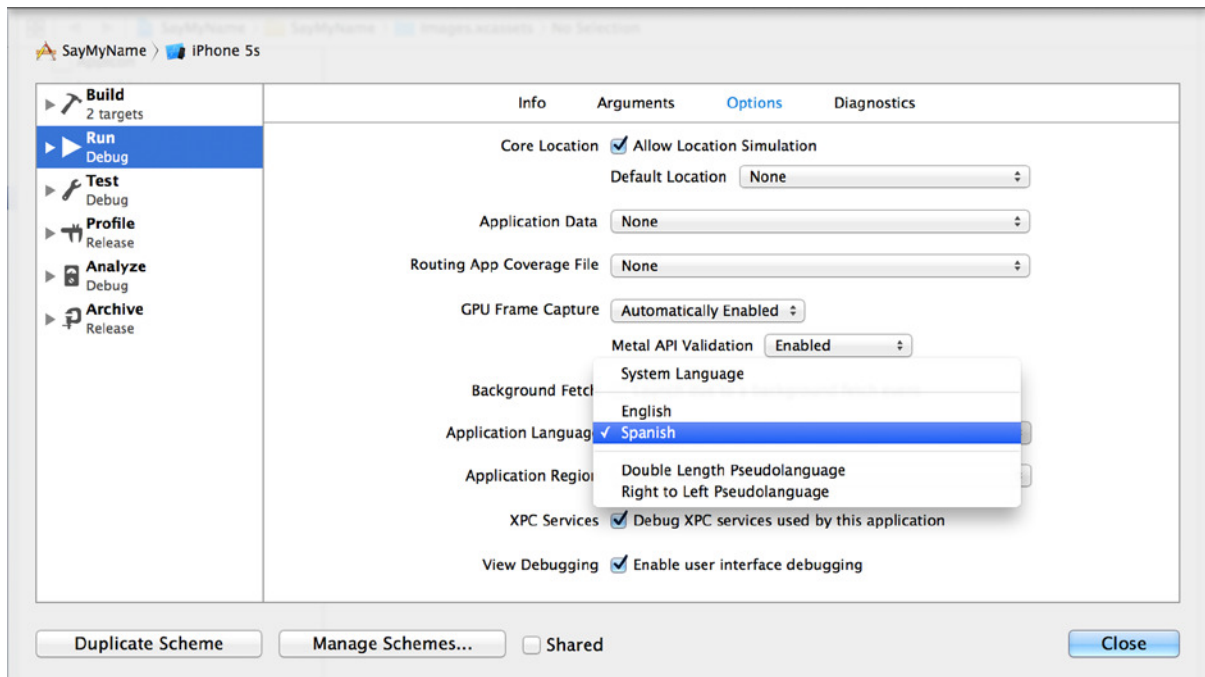


Figure 13-47. Changing Application Language in the Scheme options

By changing the language this way, you never have to face the necessity of remembering how to change the language of the operating system back into your native language, when you don't understand the language that you changed it to!

Summary

That's it! You now know how to make your application appear in any of the supported languages on an iOS device. You started by creating a single-language application that tapped into a device's contacts list and spoke the contact's name, and you ended with the excellent multilanguage application you set out to make. You can test your skills by taking the project further and adding another language to the application.

Specifically, in this chapter you did the following:

- Learned about the Address Book UI framework and took a further look at the AV Foundation framework
- Used a string formatter and placeholders
- Created a custom Swift class
- Added an additional language to a project
- Localized the storyboard and modified its strings file
- Learned about the difficulties of localizing an asset catalog
- Localized an image file
- Saw what happens to the project behind the scenes
- Created a `Localizable.strings` file and populated it with string for both languages
- Implemented `NSLocalizedString` to pull in the string values

You've achieved a lot in this chapter and learned some important skills that can cement your application's success in the App Store. Before you move on, here's a hint: keep the `SayMyName` application handy, because you use it in the next two chapters as you learn about the Organizer and then publish an application in the App Store!

Devices and the Organizer

Chapter 13 taught you about localization and how you can enable your application to appear in multiple languages with minimal effort. To learn about localization, you built the SayMyName application, where you used the Address Book UI framework to pick a contact and the AV Foundation framework to say it.

The SayMyName application, which is really cool, is used in both this chapter and the next as you step away from coding and focus on Xcode's functionality. This chapter explains the Organizer, a part of Xcode that is wasting away to almost nothing; and Devices, the latest part to be culled from the Organizer, but one that you need to use when you want to manage a physical device and prepare it for having applications deployed to it.

Chapter 1 showed you how to register for an Apple Developer Account. This chapter explains how to upgrade this to a fully paid Developer Account, allowing you to deploy applications to a physical device; then, in Chapter 15, you publish to the App Store. This chapter delves into the real nitty-gritty of Xcode's Organizer and Devices areas and explains many advanced activities such as obtaining crash logs, capturing screenshots to your computer from a device, and learning about derived data and snapshots.

Note To make the most of this chapter, you need a physical iOS device such as an iPhone, iPad, or iPod Touch running iOS 8. If you don't have one of these devices, then it will be hard for you to follow some of what is shown in this chapter.

The Role of the Organizer in Xcode 6

The Organizer in Xcode 6 has been streamlined yet again after Apple began reducing its functionality in Xcode 5. Last time the Documentation element of the Organizer and the ability to view repositories used for source control were removed, and now the device-management element has been removed into a separate area called Devices. Seeing the decline in functionality makes me feel that this will be the last version of Xcode that includes the Organizer; but in Xcode 6 it's still a key area of Xcode with which you should be familiar.

The Organizer itself doesn't do anything. Rather, it groups two useful functions into a single location:

- *Projects Organizer*: Not just a list of projects, but rather a place where you can delve into any snapshots that have been created for a project, with the ability to remove any associated or derived data
- *Archives Organizer*: One of the ways you can publish an application to the App Store

You access the Organizer from the menu bar by going to Window ► Organizer, which loads the Projects Organizer by default, as shown in Figure 14-1. Because so much of this chapter revolves around the Devices area, it makes sense to go through the Developer Program enrollment process before looking at the Organizer further.

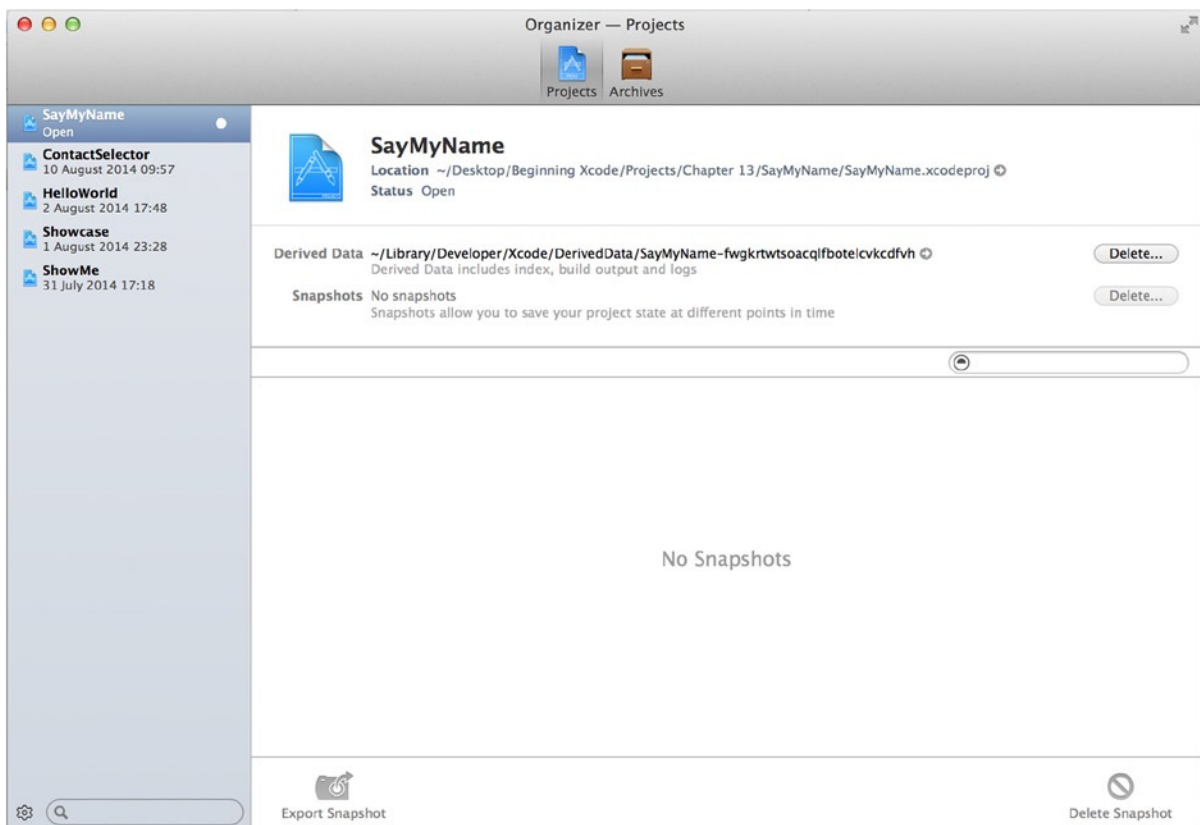


Figure 14-1. The Organizer: specifically, the Projects Organizer

Preparing Xcode for Deploying to a Device

Deploying an application to a physical device allows you to take your innovative applications with you and test them in the real world. This and many of the topics in this chapter and Chapter 15 require that you be a paid-up member of the iOS Developer Program and that your Developer Account be fully integrated with Xcode. So, the first thing you need to do is to enroll in the iOS Developer Program and become a fully fledged iOS Developer.

Enrolling in the iOS Developer Program

If you want to publish to the App Store or test your software on a physical iOS device, then you must be enrolled in the iOS Developer Program. Although everything that has been covered in the book to this point could be done with the free account, this chapter requires enrollment in the Developer Program, which costs \$99 per year (at the time of writing). If you're not ready to enroll in the Developer Program just yet, I recommend that you still read this chapter to get a feel for things, especially in relation to the Organizer.

There's nothing wrong with holding back on enrollment into the program. When I started developing for iOS, I signed up immediately and then didn't make full use of my account until my second year. It might make financial sense for you to hold back until you've got an application that you want to test on a physical device before releasing it to the App Store.

Note that there's much more to the Developer Program than just being able to publish apps and test on a device. You get access to beta versions of iOS, Xcode, and Apple TV firmware, which gives you a head start on adapting any existing apps for the new versions of iOS. You can also show off all the new features Apple brings to its devices months before Joe Public gets a look.

Note If you're already a paid-up iOS developer, skip ahead to the section "Adding Your Developer Account to Xcode."

To begin the process, follow these steps:

1. Go to the iOS Dev Center in your web browser at <https://developer.apple.com/devcenter/ios>. Click the Log In link in the upper-right corner, as shown in Figure 14-2, and enter your Apple Developer Account details.

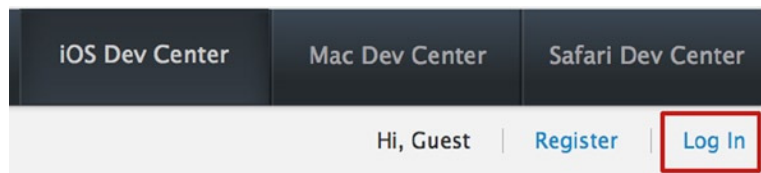


Figure 14-2. Click the Log In link, and enter your Developer Account details

2. After you've entered your details and been returned to the iOS Dev Center, notice the box at right promoting the iOS Developer Program, as shown in Figure 14-3. Either click the Learn More link or visit <https://developer.apple.com/programs/ios>.



Figure 14-3. An advertisement for the iOS Developer Program

3. The landing page for the iOS Developer Program is shown in Figure 14-4. This page holds lots of information about what the program enables you to do, and this is where you start the enrollment process. Click the Enroll Now button when you're ready.

Developer Technologies Resources Programs Support Member Center

iOS Developer Program

The fastest path from code to customer.

[Enroll Now](#) \$99/year

1. Develop
Develop your application with the iOS SDK and a wealth of technical resources in the iOS Dev Center.
[Learn more >](#)

2. Test
Test and debug your code on iPad, iPhone and iPod touch to finalize your applications.
[Learn more >](#)

3. Distribute
Distribute your apps on the App Store and reach millions of iPad, iPhone, and iPod touch users.
[Learn more >](#)

Figure 14-4. The landing page for the iOS Developer Program

4. You're provided with information about how the enrollment process works, as shown in Figure 14-5. I'll take you through registration as an individual; if you're registering as a business in the United States, you need to supply your D-U-N-S number; or if you're in another part of the world, you're asked for alternative information for validation. Once you have read this information, click the Continue button near the bottom of the page.

Enrolling in Apple Developer Programs

Get everything you need to develop and distribute apps for iOS and OS X.

It's easy to get started.

✓ Choose an enrollment type.

Individual: choose this option if you are an individual or sole proprietor/single person business.

Company/Organization: choose this option if you are a company, non-profit organization, joint venture, partnership, or government organization.

✓ Submit your information.

Provide basic personal information, including your legal name and address. If you're enrolling as a company/organization, we'll need a few more things, like your legal entity name and D-U-N-S® Number, as part of our verification process.

✓ Purchase and activate your program.

Once we verify your information, you can purchase your program on the Apple Online Store. After you have completed your purchase, we'll send you an email within 24 hours on how to activate your membership.

Continue



Figure 14-5. The enrollment process, explained

5. You need to choose whether to use the account you signed in with or to create a new account, as shown in Figure 14-6. This is obviously a choice that you must make. The process of registering for an Apple Developer Account was covered in Chapter 1; if you want to register for a new account at this point, feel free to refer back in the book for guidance and then come back here to continue. If you're logged in with the account you want to use for the program, click Continue.

Sign in or create an Apple ID.

You can enroll in the iOS Developer Program or Mac Developer Program with the same Apple ID you use for other services like iCloud and the Apple Online Store. However, if you have an iTunes Connect account for distributing another media type (music, TV, movies, or books) or are enrolled in the iOS Developer Enterprise Program, you need to use a different Apple ID for your enrollment.

Existing Apple ID

Enroll in an Apple Developer Program with your Apple ID associated with **matthew.knott@me.com**. If you would like to use a different Apple ID, [sign out](#).

Continue

or

New Apple ID

Create a new Apple ID if you have an existing iTunes Connect account, participate in the Volume Purchase Program, are enrolled in the iOS Developer Enterprise Program, or prefer to have an Apple ID dedicated to your business transactions.

Create Apple ID

[Forgot your Apple ID or Password?](#)

Figure 14-6. Selecting whether to use an existing account or create a new one

6. On the next screen, select whether you want to register as an individual or as a company. As already mentioned, I take you through the process for an individual, so scroll to the bottom of this page and click the Individual button.
7. You're presented with options for each of the available programs: iOS, Mac, and Safari. Check the box next to the iOS option, as shown in Figure 14-7, before scrolling to the bottom of the page and the clicking Continue.

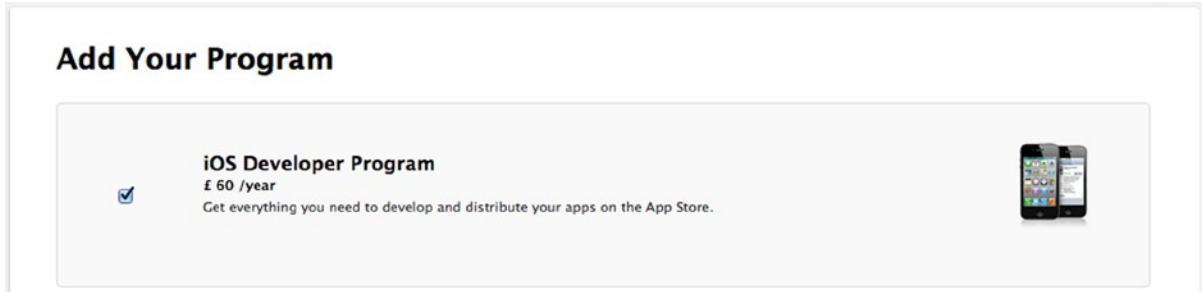


Figure 14-7. Selecting the iOS Developer Program

8. With your options for enrollment set, you see a summary of the proposed purchase, as shown in Figure 14-8. At this point the price of enrollment adjusts to your local currency. If you're ready to do so, click Continue.

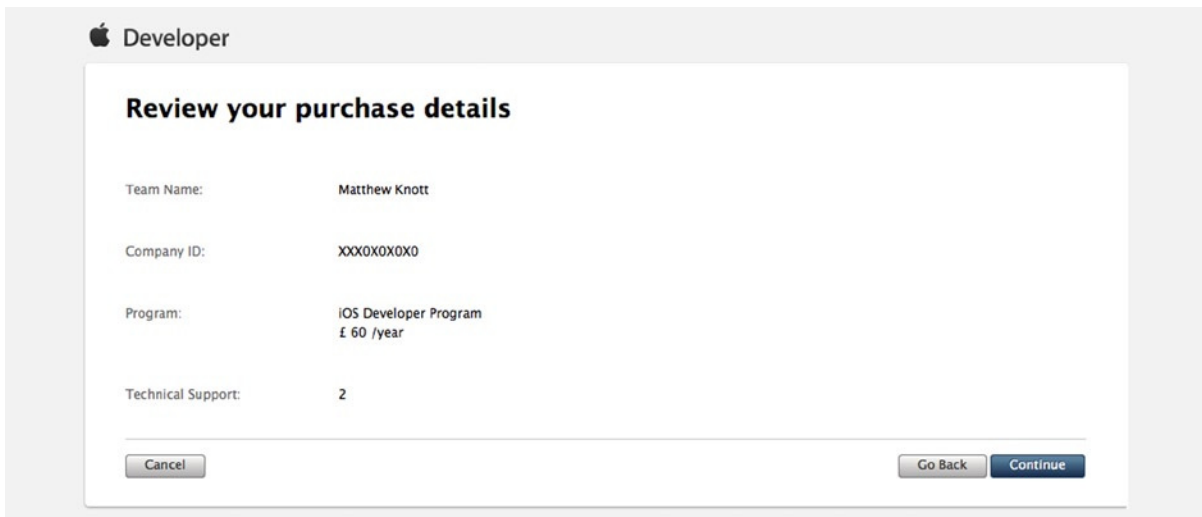


Figure 14-8. Reviewing the details of your enrollment

9. You're almost at the end of the process. Apple presents you with the license agreement, which is legally binding. If you're planning to take iOS app development seriously, then, contrary to how you may usually approach such things, I recommend that you read it and save a copy. There is a link to a PDF version of the agreement, which I have highlighted with a box in Figure 14-9. This document protects you as much as it protects Apple, so keep a copy for your records. When you're ready, check the disclaimer box below the license agreement and click the I Agree button.

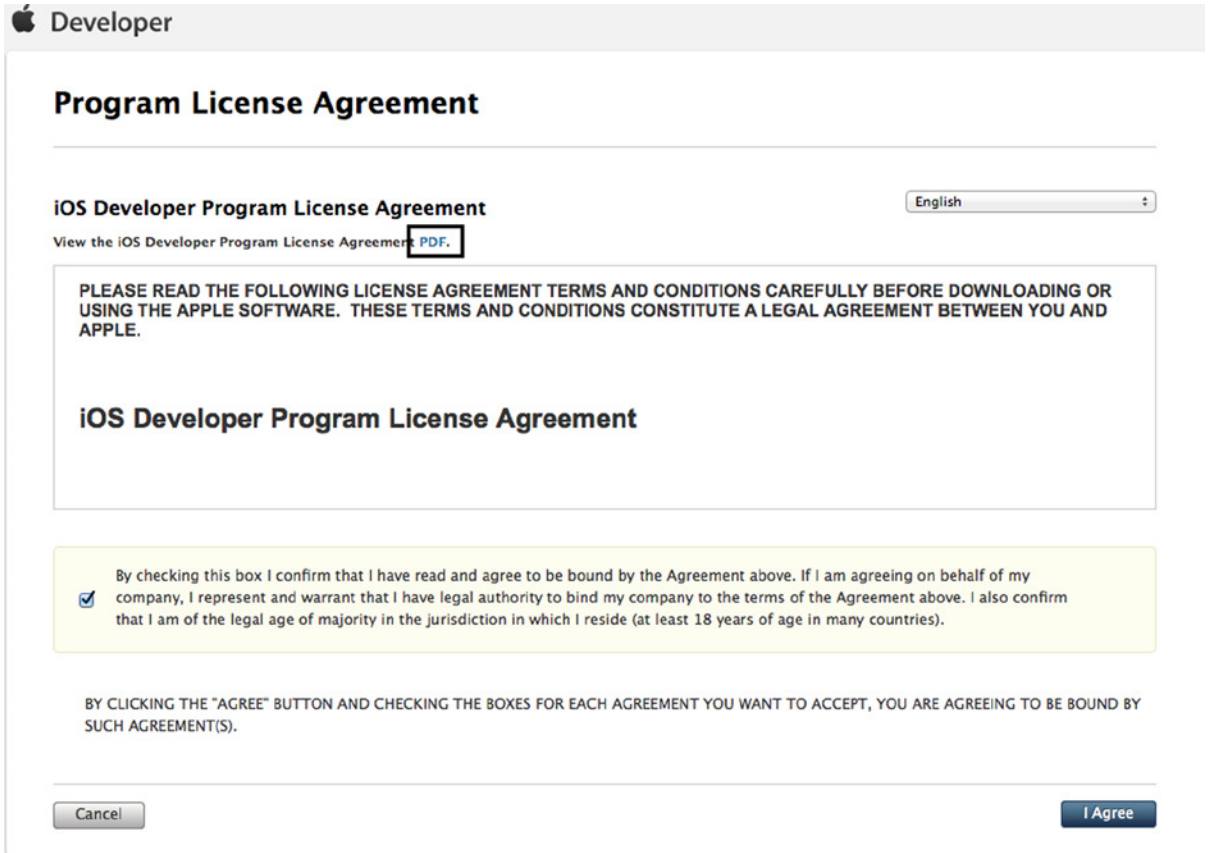


Figure 14-9. The license agreement for the iOS Developer Program

10. The final part of the enrollment process comes when you're prompted to add your personalized iOS Developer Program item to the cart for the Apple Store, as shown in Figure 14-10. Click the Add to Cart button, and you're taken to the Apple Store for your region, where you can go through the checkout process and pay for your purchase.

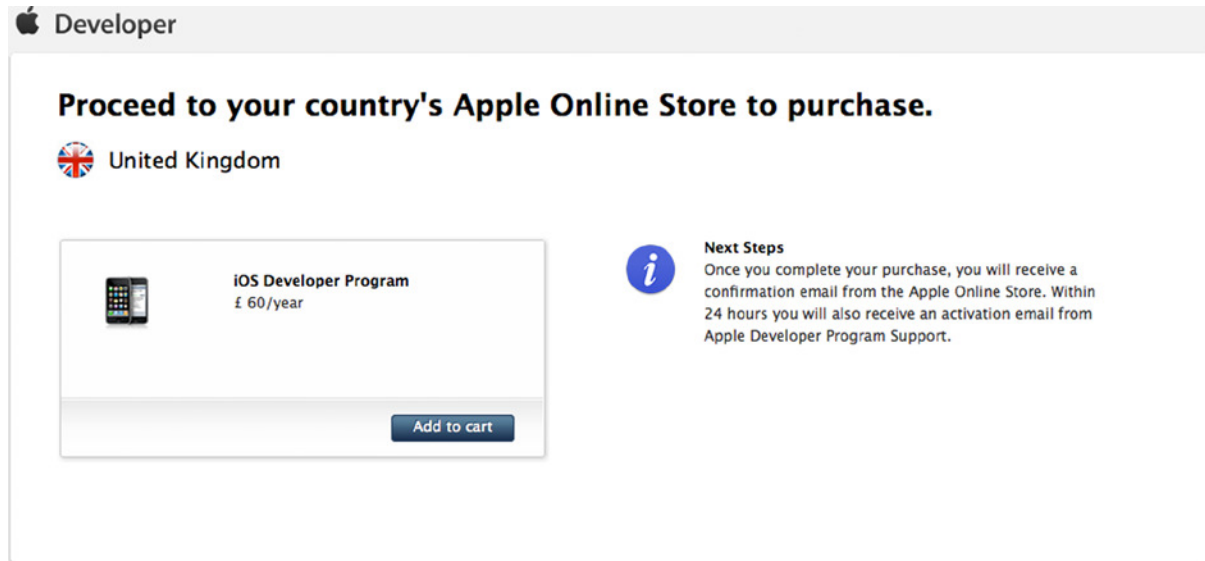


Figure 14-10. Getting ready to purchase your iOS Development Program

I won't step you through the specifics of the checkout and payment process. Once you've completed the purchase and your enrollment is approved, you receive an e-mail confirming that you're a part of the Developer Program!

Note If you live in a country that has no online Apple Store, the process for paying is very different from that shown in this chapter. You can still sign up; it just takes a little longer.

Adding Your Developer Account to Xcode

Being enrolled in the iOS Developer Program is one thing, but if Xcode doesn't know you have a Developer Account, you can't take advantage of any of the extra functionality it gives you. You may have already added your developer account details through your own experiences with Xcode; but if not, this section guides you through adding the account details. Then it's on to the Devices. Follow these steps:

1. Open Xcode. From the menus, select Xcode ► Preferences (⌘+). Once you're in Preferences, select the Accounts tab, as shown in Figure 14-11.

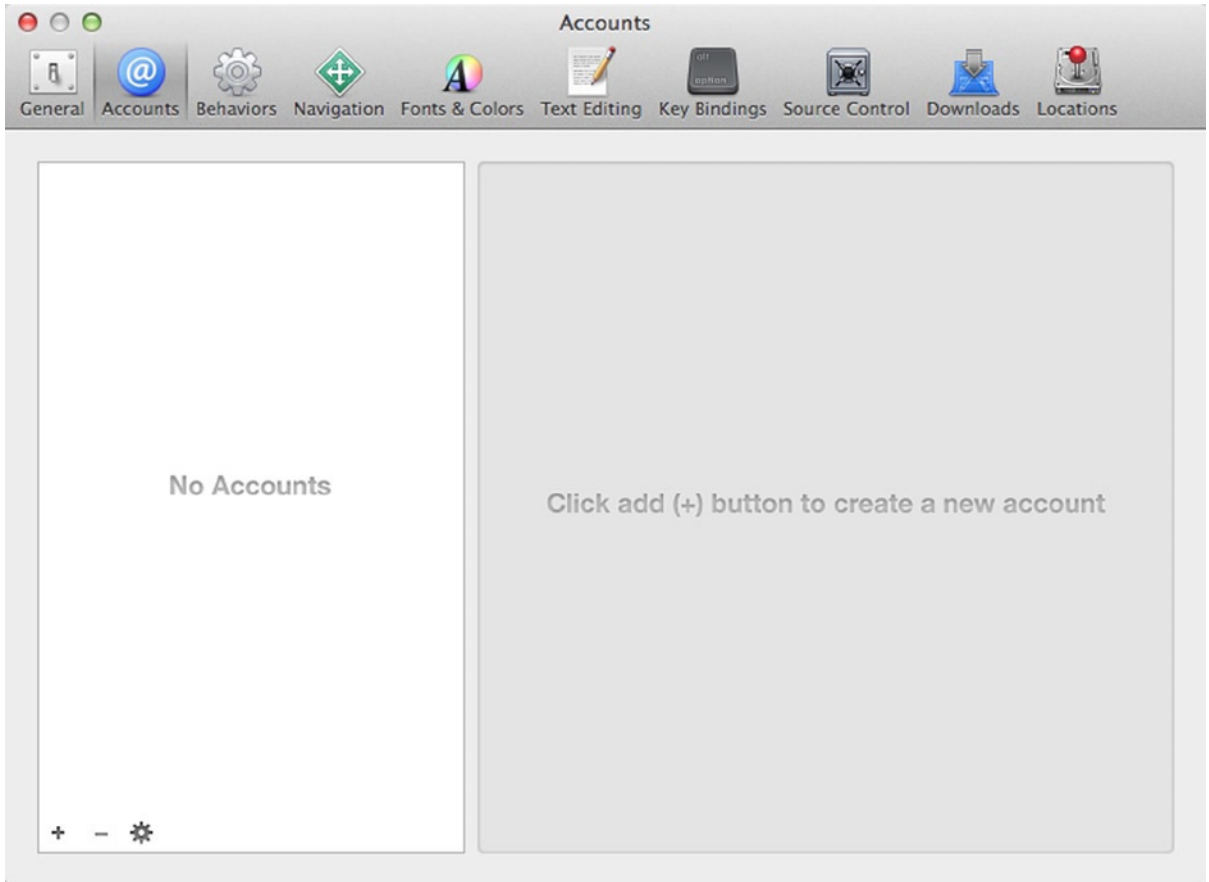


Figure 14-11. The Accounts tab in Xcode

2. This is the same area you came to in Chapter 12 when you added the remote Git repository, but this time you're going to add an Apple ID. (As you see in Figure 14-11, I cleared out all my accounts, leaving the area blank.) Xcode tells you how to add a new account: just as it says, click the + symbol at the bottom of the left page and select Add Apple ID, as shown in Figure 14-12.

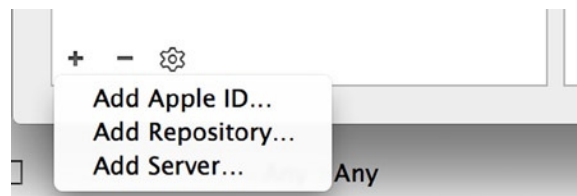


Figure 14-12. Selecting the option to add an Apple ID to Xcode

3. A pop-over appears, asking for your Apple ID and password, as shown in Figure 14-13. Enter your credentials, and click the Add button to link this account with Xcode. Notice the Join a Program button: this is an alternative way to start your enrollment into any of the Apple developer programs.



Figure 14-13. Enter your Apple ID and password to integrate the account with Xcode

When you return to Preferences, your developer account appears in the left pane of the account preferences. The account has been added to Xcode, but there are a few more steps you need to go through before you can deploy your application onto a device. This is a great opportunity to get familiar with the Devices area, which allows you to finish preparing your device for deployment.

Preparing a Device for Deployment

Deploying any of your applications to a physical device is a process that is becoming increasingly simple, especially in this latest release. In order to deploy applications onto a device, you must register the device with Apple on your developer profile.

Prior to Xcode 5, if you wanted to use a physical device for development, you had to obtain the device's unique identifier through the Devices Organizer and then add it manually to the Certificates, Identifiers, and Profiles area of the Apple Dev Center. This was a long-winded and tedious process.

In Xcode 5, Apple simplified the process by allowing you to add a device from the Devices Organizer with a single click of a button. In Xcode 6, Apple has further automated the deployment process by integrating it into the overall process of deploying an app to a physical device. As such, you no longer need to manually add a device to the portal, which is why Apple has removed the ability to do this from Xcode.

To demonstrate this functionality, follow these steps:

1. Ensure that you have the SayMyName project open.
2. Connect your physical device running iOS 8 to the Mac via USB cable. Other applications, such as iPhoto, may open at this time; if so, dismiss them. Click the list of devices, as shown in Figure 14-14, and select your device. In this instance I've selected my wife's unregistered iPhone, Lisa's iPhone.

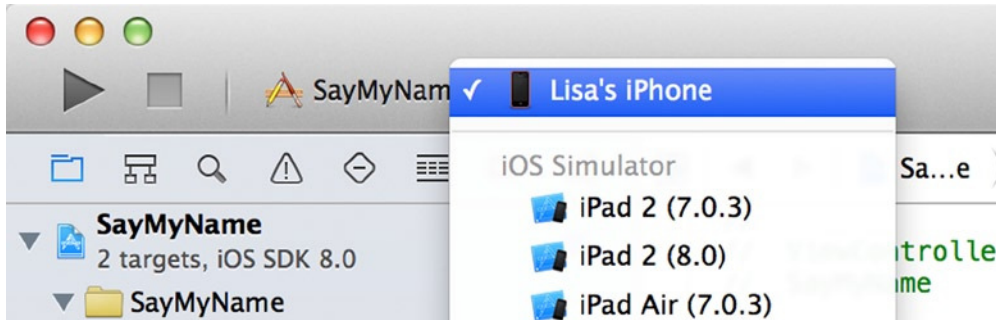


Figure 14-14. Select your physical device from the list of available devices

3. Ensure that your device is unlocked and on the home screen. If prompted by the device, trust the computer you're connected to.
4. Run the application from Xcode, and warning messages start to appear. But as shown in Figure 14-15, Xcode offers to help. Click Fix Issue.

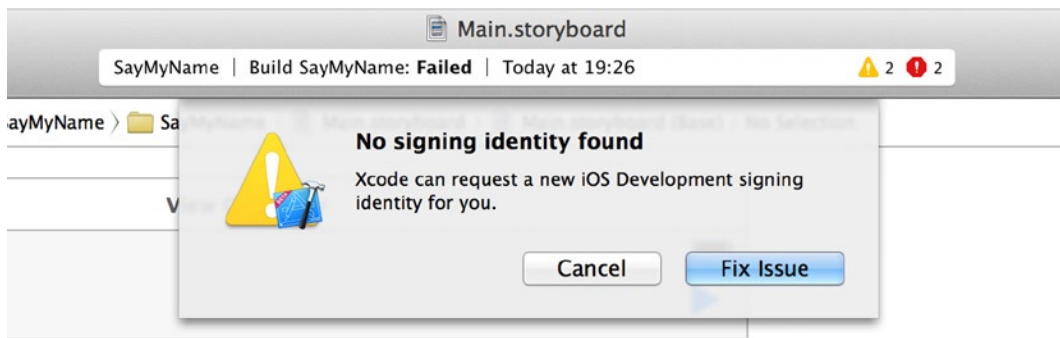


Figure 14-15. Xcode warns about missing signing identities, but it offers to fix the problem for you

5. You're prompted to select a development team to create the signing identity. Choose the team associated with your developer account, as shown in Figure 14-16, and click Choose.



Figure 14-16. Selecting the development team to use

At this point, Xcode does a number of things in the background: it creates a developer certificate on your Mac, it registers your new device on the developer portal, it creates a code-signing identity that is used to secure and encrypt the binary during the build phase, and it creates a provisioning profile on the device itself. All this before it deploys the app to your device!

Note Make sure that if you have a PIN lock on your device, the device is unlocked and on the home screen. Otherwise, Xcode may give you an error message.

Eventually, the app runs on your device. You can now play around by selecting names from your address book and getting the device to speak their names. Deploying an application to physical Apple hardware for the first time is an incredible feeling.

It's hard to get across what a nightmare this process used to be. Apple has made some huge strides in creating a process that by and large sorts itself out. Next, let's take a look at how you can use the Devices area of Xcode to manage and control your devices, as well as gather useful information about your application.

Managing Devices in Xcode

Now that the preparatory work has been done, you're all set to get your device ready to be used for development. If you haven't already, close Xcode's Preferences and open Devices by going to Window ► Devices (⌘+Shift+2).

The Devices area not only allows you to manage your physical hardware in great detail, but also, in Xcode 6, gives you access to information about the different iOS simulators for the first time. With each physical device, you can access crash logs, take screenshots, add provisioning profiles, and view the device's console data.

Assuming that your device is still connected to the Mac via the USB cable, it appears in the sidebar. Select it, and you're presented with an overview of all the information that is relevant to you as a developer. Figure 14-17 highlights the key areas you can see:

- **Sidebar:** In the sidebar you're able to select your Mac, any connected physical devices, and any of the simulators. As you can see in Figure 14-17, I have downloaded the iOS 7 simulators, giving me a pretty full sidebar. To find devices, you can use the filter box at the bottom of the sidebar to display only devices with "iPad" in the title, for example. The plus symbol allows you to create new simulators; the cog lets you rename a device, remove a simulator, see provisioning profiles, or remove the device from the Run Destinations list.

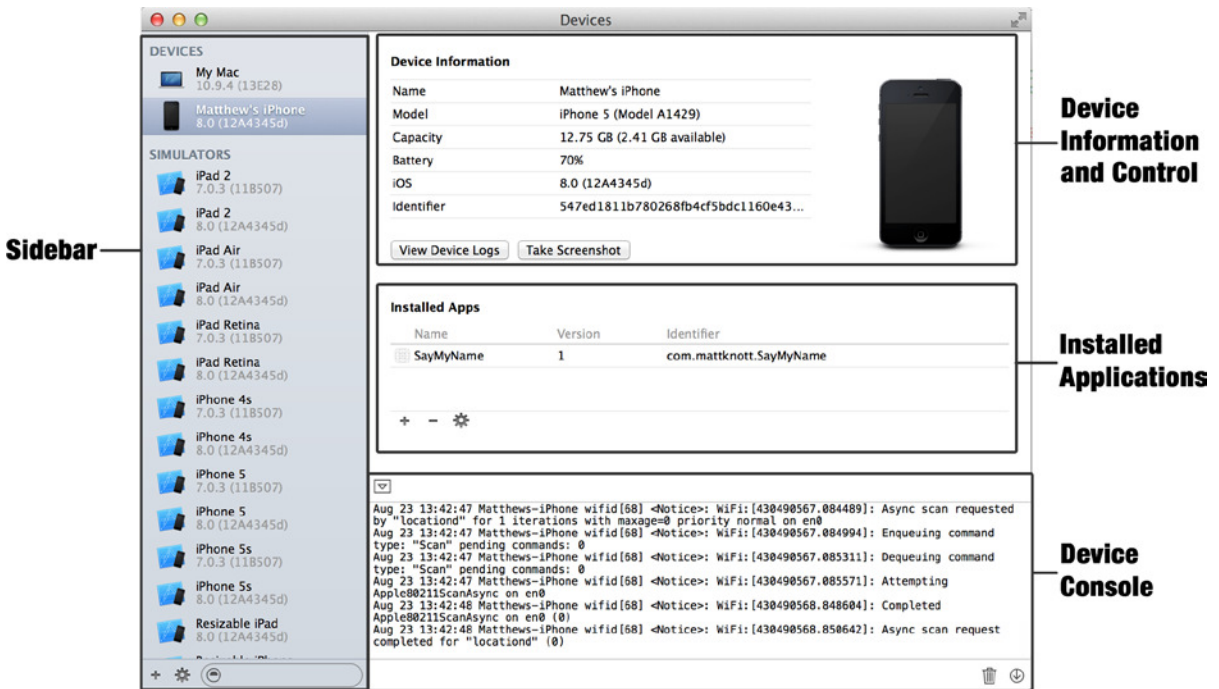


Figure 14-17. The Devices area, with a device selected

- **Device Information and Control:** In this area you can view important information that is unique to your device, such as the serial number and the unique identifier. You also have the ability to view the log files created by applications on the device and to capture a screenshot to the desktop.
- **Installed Applications:** Any applications that you deploy to the device can be found here. You can interact with the applications by viewing their container or by downloading and replacing it. You can also remove the application or add one from an .ipa file.
- **Device Console:** Here you can see in real time the activity being captured to your device's console. You probably see everything from Wi-Fi and iOS errors to individual applications adding comments to the console. This a great tool to use when debugging applications and OS behaviors.

Capturing a Screenshot from a Running Application

As you learn in Chapter 15, capturing screenshots is not only good for posterity—it's a prerequisite for submitting your application to the App Store. You can take a screenshot of your application from the simulator by going to File ► Save Screen Shot (⌘+S). You may already know that you can take a screenshot on a physical device by pressing the Home and power buttons simultaneously, but then you have the hassle of organizing and copying files. Through Devices, Xcode gives you the ability to capture full-resolution screenshots directly onto your Mac, as follows:

1. In the sidebar, make sure you've selected your device.
2. Be sure the SayMyName application is running on your device. Then click Take Screenshot, which is found below the Device Information section as shown in Figure 14-18.

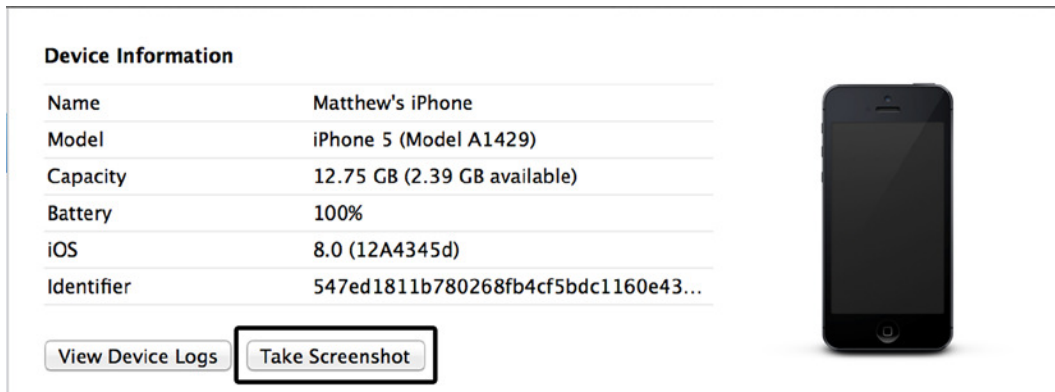


Figure 14-18. The button for taking a new screenshot is found below the device information

It may appear as though nothing has happened, but in fact a screenshot has been captured and saved to your desktop in the .png format.

Taking a screenshot may not seem like an important function. But as I mentioned previously, it's essential for when you submit your application to the App Store, because Apple requests that you supply screenshots of the application in action.

If you're familiar with Xcode 5, then you may notice that in this new release, Apple has removed the ability to manage screenshots from within Xcode. One of the useful functions available from this area was the ability to set a screenshot as a launch image. The reason for the change was that in Xcode 6's final release, Apple introduced a new default approach for the launch screen: you use a .xib file, as I cover in Chapter 15.

The Projects Organizer

You've taken a detailed look at Xcode 6's simplified device-management tool, Devices. Now it's time to open the Organizer and see what functionality is left. Start by opening the Organizer by going to Window ► Organizer.

Note The Organizer used to have the shortcut key $\text{⌘}+\text{Shift}+2$, but in Xcode 6 this shortcut has been assigned to Devices. You can change this assignment in Xcode's preferences by going to the Key Bindings tab and scrolling down until you find Organizer.

In the Organizer, the initial tab that opens is the Projects Organizer. The main purpose of the Projects Organizer is to list all the projects and workspaces you have created or opened in Xcode and provide you with a number of actions you can perform for each project or workspace.

If you want to create snapshots of your projects using Xcode's integrated snapshot facility as an additional backup, then you'll likely find the Projects Organizer invaluable due to the amount of control it gives you over project snapshots. A *snapshot* is, as the name suggests, a complete snapshot of your project at a specific point in time. Don't be fooled into thinking that if you're using Xcode's Git-based source control, you don't need snapshots! They offer valuable protection from mistakes between commits.

If you don't want to use the snapshot facility, at some point you'll appreciate the ability to hunt down a project you've opened and view its location. Figure 14-19 gives you an overview of the key areas of the Projects Organizer.

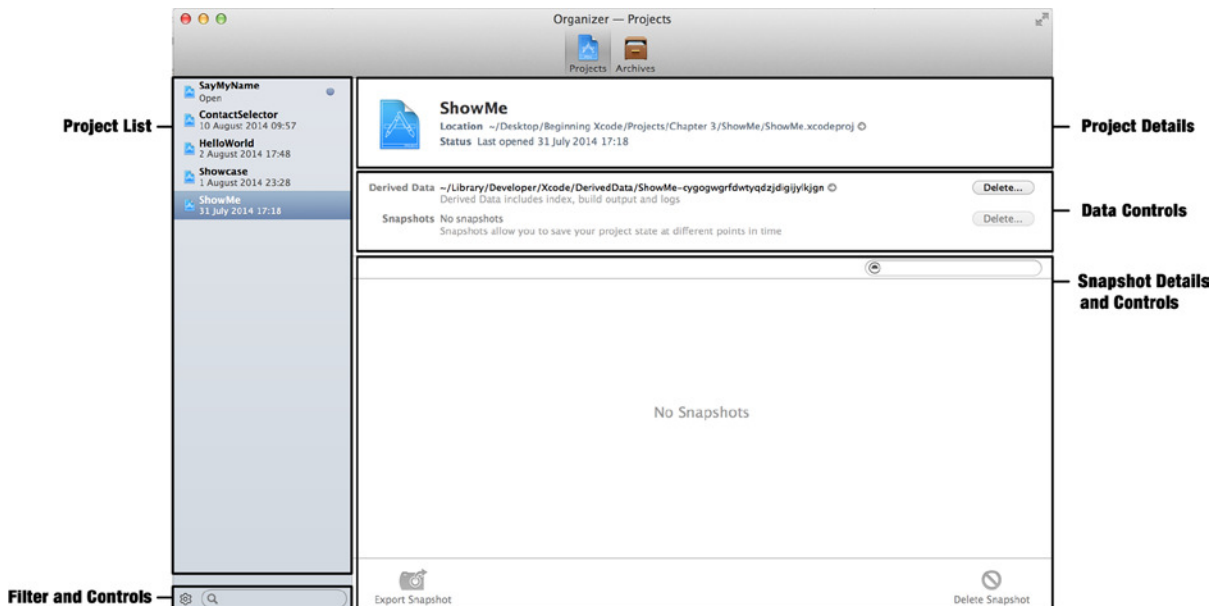


Figure 14-19. An overview of the key areas of the Projects Organizer

Using the Project and Workspace List

The sidebar in the Projects Organizer is dedicated to listing all the projects and workspaces that you have ever opened on your Mac. Each project is sorted in reverse chronological order based on the last time you opened the project, which is why SayMyName appears at the top of the list in Figure 14-19. Any projects listed in red are projects that you have opened but that have since been removed from the Mac.

To make sense of the list, you can filter its contents using the filter box at the bottom of the sidebar, shown in the Filter and Controls section in Figure 14-19. Typing “say” would quickly narrow the listed items to show the SayMyName project, for example.

To the left of the filter bar in Figure 14-19 is a small cog icon that allows you to perform a set number of actions for the selected project or workspace. The options displayed are also available by right-clicking a project or workspace, as shown in Figure 14-20, with the added benefit of Xcode’s contextual help menu being available:

- *Open*: This option opens the project or workspace in Xcode.
- *Reveal in Finder*: Use this option to track down the projects folder in the file system.
- *Remove from Organizer*: This removes the project from the Projects Organizer and deletes any derived data, such as snapshots and indexes, but leaves the project itself intact in its folder.

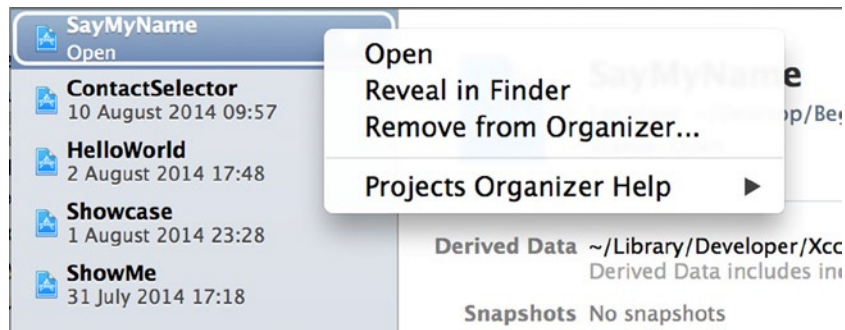


Figure 14-20. Right-clicking a project reveals some useful options

Using Snapshots

Snapshots in Xcode are a great feature that lets you save and archive not just project files and folders, but also workspace settings and configuration. A snapshot, as the name implies, is like a photograph: it’s an image at a point in time that allows you to roll back your project to the time of the snapshot in the event of a major issue.

Automatically Creating Snapshots

One of the most effective ways you can use snapshots is by configuring a behavior in Xcode so that a snapshot is created when you perform a set task, such as a successful build of your application. If you've followed this book to the letter so far, then in Chapter 4 you may recall that you were prompted to set up snapshots; I advised against doing so at that point so you could look at the end-to-end snapshot process now:

1. Access Xcode's Preferences by selecting Xcode ► Preferences (⌘+,) from the menu. Choose the third tab, Behaviors.
2. The left column of the Behaviors tab lists all the events that can have their behaviors configured. The aim here is to configure Xcode so that it creates a snapshot every time you successfully build the project. The first group in the left column is labelled Build, and the third option in that group is Succeeds. Choose Succeeds, as shown in Figure 14-21.

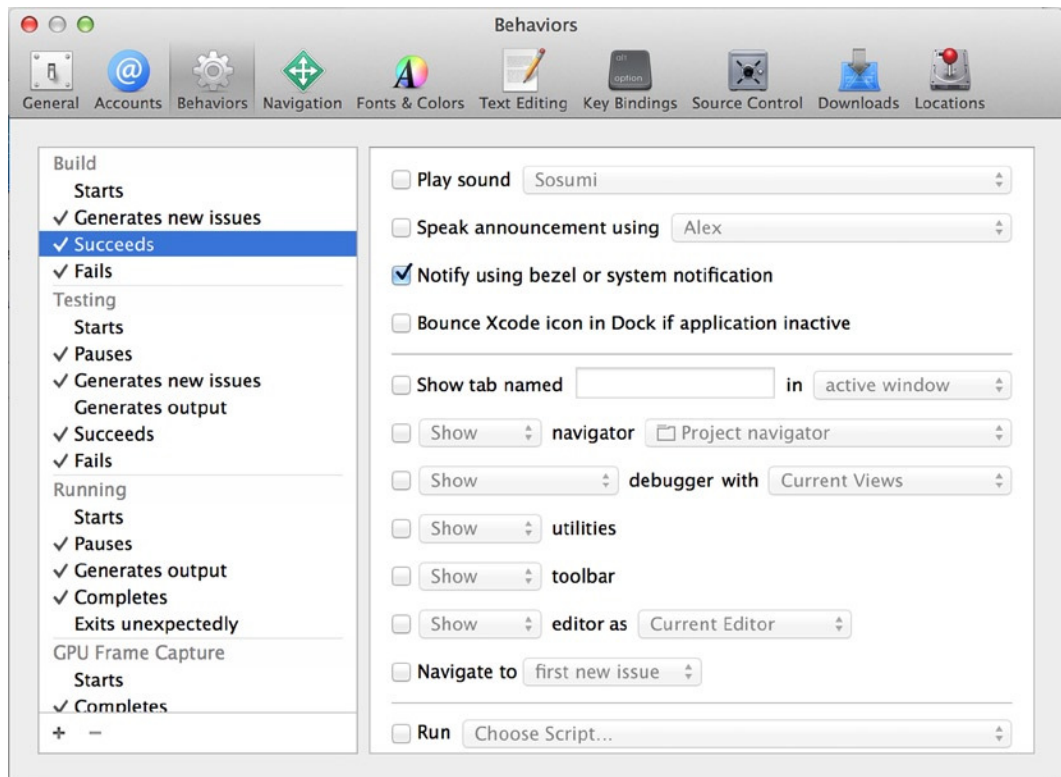


Figure 14-21. The Behaviors tab of Xcode's preferences

3. The right pane displays all the behaviors that can be configured. Scroll to the bottom of the list, and select the Create Snapshot box.
4. Close the preferences, and return to Xcode. Run your application once more, either on your device or in the simulator. Or alternatively, because the criterion for an automatic snapshot is that your project builds successfully, you can go to Product ► Build (⌘+B).

Once the application is running or your build completes, assuming it was successful, you now have a snapshot. Before you confirm this, it's important to note that in addition to automatic snapshots, you can also create manual snapshots.

Manually Creating a Snapshot

Having snapshots automatically created when your build event is successful is really useful because if it all goes wrong, you know you can go back to the last known good configuration and start again without too much heartache. Sometimes, though, you may want to capture an interim snapshot if you feel you're about to attempt something that may cause issues, such as refactoring large amounts of code. In this instance, the best thing to do is to create a quick snapshot and annotate it to explain why you created it and what you were about to do.

To create a snapshot manually, go to File ► Create Snapshot (⌘+Control+S). You're presented with a pop-over, as shown in Figure 14-22, where you can add a title for the snapshot and provide some descriptive text. Fill out both text fields, and click Create Snapshot. It's a testament to the way Xcode has been developed that the snapshots are created so quickly and subtly!

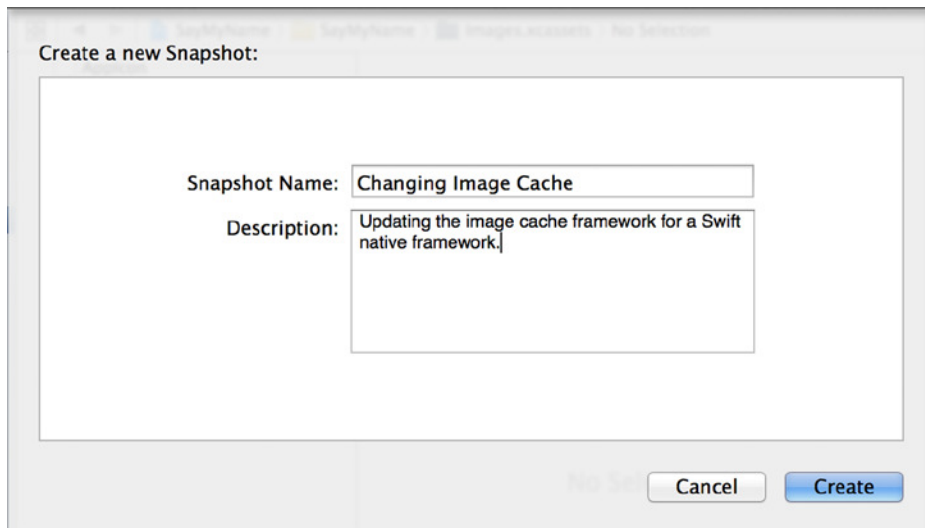


Figure 14-22. Adding a title and description for the manually created snapshot

Now it's time to go back to the Projects Organizer to see what's been happening behind the scenes.

Managing Snapshots

Now that you've created a couple of snapshots, you can see more of what the Projects Organizer can offer you. Reopen the Organizer, and reselect the Projects Organizer tab. With the SayMyName project selected, you see your two snapshots listed, as shown in Figure 14-23.

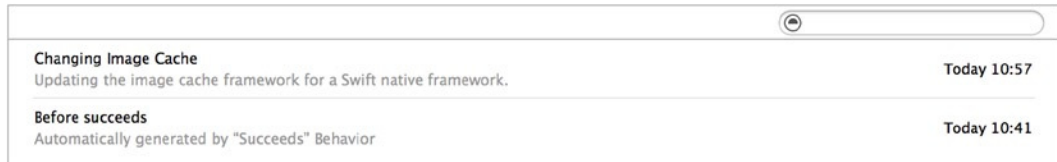


Figure 14-23. The two snapshots listed in reverse chronological order in the Projects Organizer

Although it isn't completely apparent, snapshots are stored in a single archive on your Mac. Just above the list of snapshots is a line titled Snapshots that indicates the full path to the snapshot archive for this project. Next to the path, and highlighted in Figure 14-24, is an arrow that will take you straight to the archive path. Click that arrow to see your snapshot archive in the Finder.

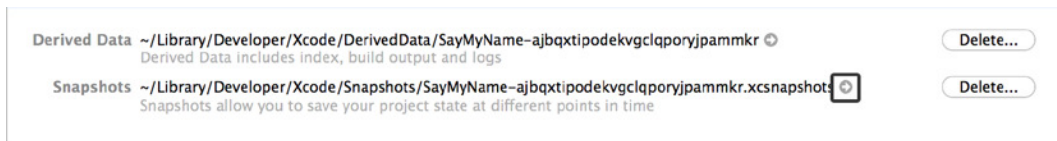


Figure 14-24. You can navigate to the folder containing the snapshot archives to by clicking the highlighted arrow

When you navigate to the folder containing the snapshot archive, note the folder's size. On my system, it sits at just 136 KB. This will rapidly increase as you continue to develop and build your application, which will have an impact on available storage and backup times. So, managing your snapshots and disposing of older snapshots are important maintenance tasks.

Deleting Snapshots

Deleting a snapshot is very straightforward, but be careful how you do it. In Figure 14-24, notice that next to the highlighted arrow is a Delete button. Clicking this button deletes *all* snapshots on the system, not just one, so be careful:

1. To delete a single snapshot, highlight the bottom snapshot, titled Before Succeeds, as shown in Figure 14-25.

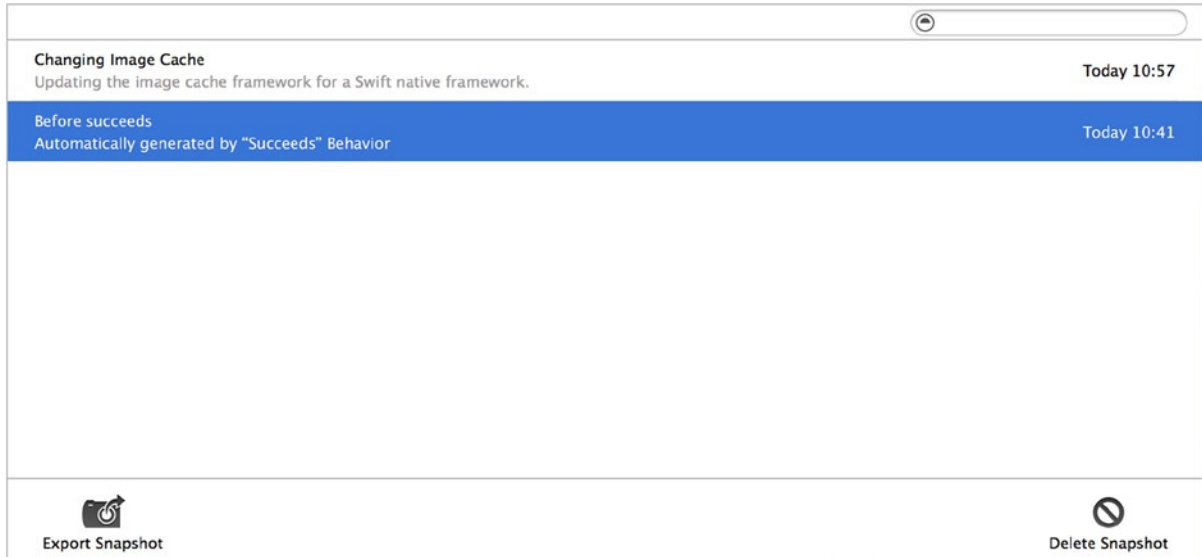


Figure 14-25. Highlighting the oldest snapshot

2. Click the Delete Snapshot button in the lower-right corner, also shown in Figure 14-25.
3. You're prompted whether you wish to delete the snapshot. Click Delete, and the snapshot is removed, leaving you with a solitary snapshot in the list.

Exporting Snapshots

An alternative to restoring your project from a snapshot is to use the Projects Organizer's Export functionality. Exporting a snapshot doesn't export the archive; it allows you to extract the full project from the archive to a separate folder on your system:

4. Select the single remaining snapshot in your list of snapshots, and click the Export Snapshot button at left side on the toolbar, as shown back in Figure 14-25.
5. You're presented with the standard Mac OS Save dialog, as shown in Figure 14-26. Locate the folder you want to export to; I created a folder specifically for snapshot exports. Click the Export button.

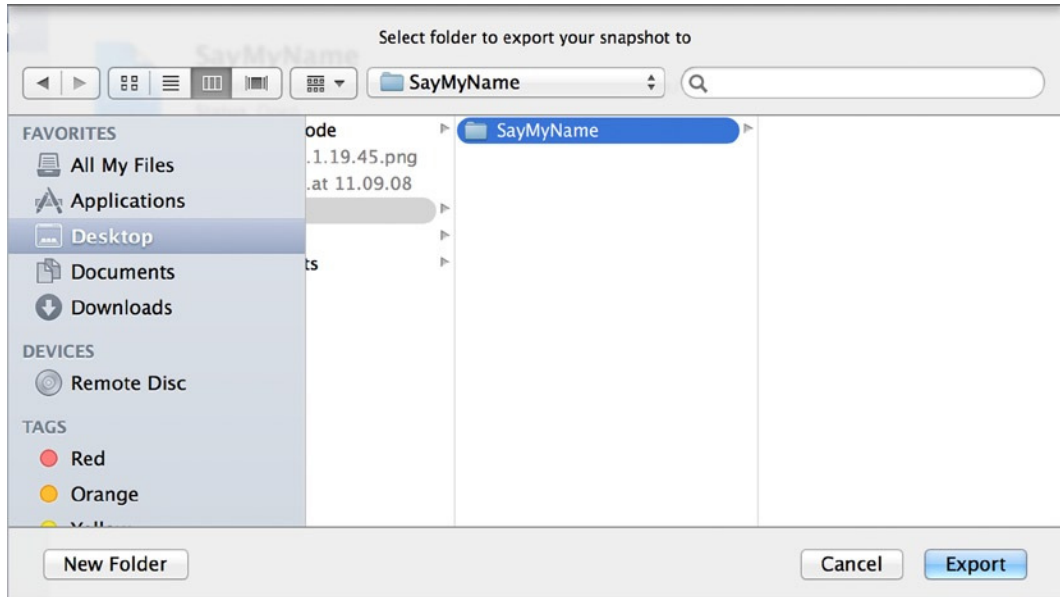


Figure 14-26. Choosing a location for export

Once the export is complete, you receive a notification advising you that it has finished. Click OK, and you're returned to the Projects Organizer, at which point you can navigate to your exported project and run that instance in parallel with the current one.

Restoring from a Snapshot

The main purpose of creating snapshots is that you have a full image of your project to fall back on if something goes horribly wrong. Therefore, it follows that at some point you may need to restore your project from a snapshot:

1. Because the Organizer is effectively operating independently of any workspaces you may have open, you can't trigger a restore from the Projects Organizer. Instead, return to your project in Xcode, and go to **File** ► **Restore Snapshot**.
2. A pop-over appears, as shown in Figure 14-27, asking you to select the snapshot to restore. In this case, your choice is limited to a single snapshot; select it, and click the Restore button.

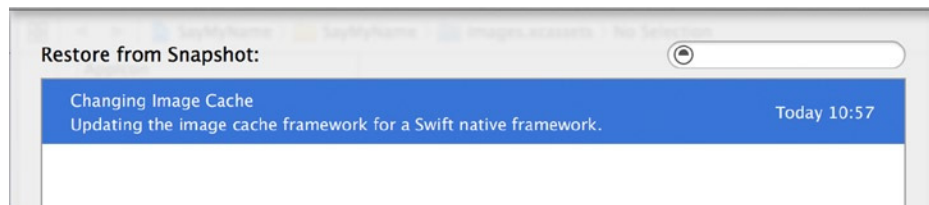


Figure 14-27. Selecting a snapshot to restore

3. If you haven't made any changes to the project since the snapshot was created, you receive a warning advising you that no changes have been made, as shown in Figure 14-28. This is great, because it means Xcode has compared the snapshot with the current project status and detected no changes, instead of simply overwriting the files blindly.

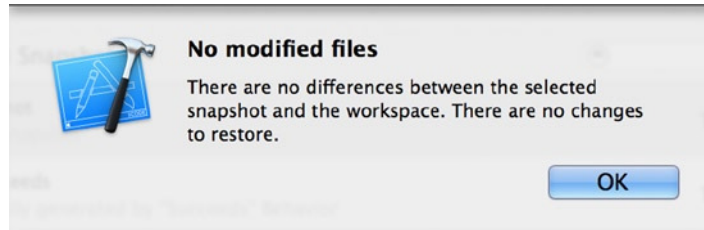


Figure 14-28. Rather than blindly overwriting the files, Xcode compared the snapshot and detected no changes

If you had made changes to the project, they would have been removed, and you would be exactly where you were when the snapshot was created.

You now know everything there is to know about using snapshots with Xcode. With Git version control and snapshots, you should never be in a situation where you lose large chunks of your project.

The Archives Organizer

The Archives Organizer represents one of a couple of ways that you can submit applications to the App Store. Because App Store submission is the topic of Chapter 15, I focus here on how you can create an archive of your application and then use the Archives Organizer to analyze it.

Right now, your Archives Organizer probably looks empty, like the example in Figure 14-29, because it contains no archives. In order to make more sense of the Archives Organizer and its capabilities, the first thing you need to do is archive the SayMyName application.

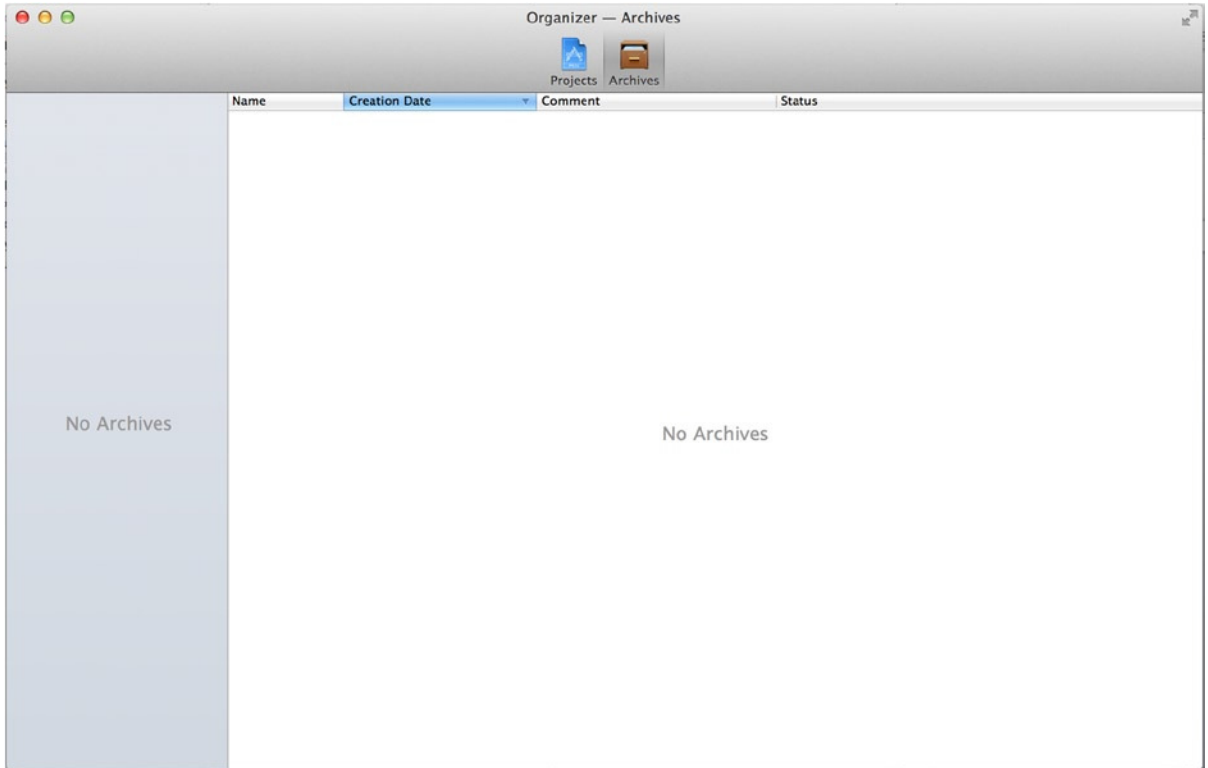


Figure 14-29. The Archives Organizer, minus any archives

Archiving Your Application

When you archive your application, it's compiled, ready for release, and then compressed into an Xcode archive file (.xcarchive) on your computer. The actual archiving process takes two clicks of the mouse, so rather than starting with that, I'll take you behind the scenes so you have a better understanding of what happens when you ask Xcode to archive an application.

When you choose to archive your application, you're performing an action that is customizable as part of the currently active scheme. To see how the action is configured, close the Organizer and go back to Xcode. Next, go to Product > Scheme > Edit Scheme (⌘+<), and select the Archive action from left column, as shown in Figure 14-30.

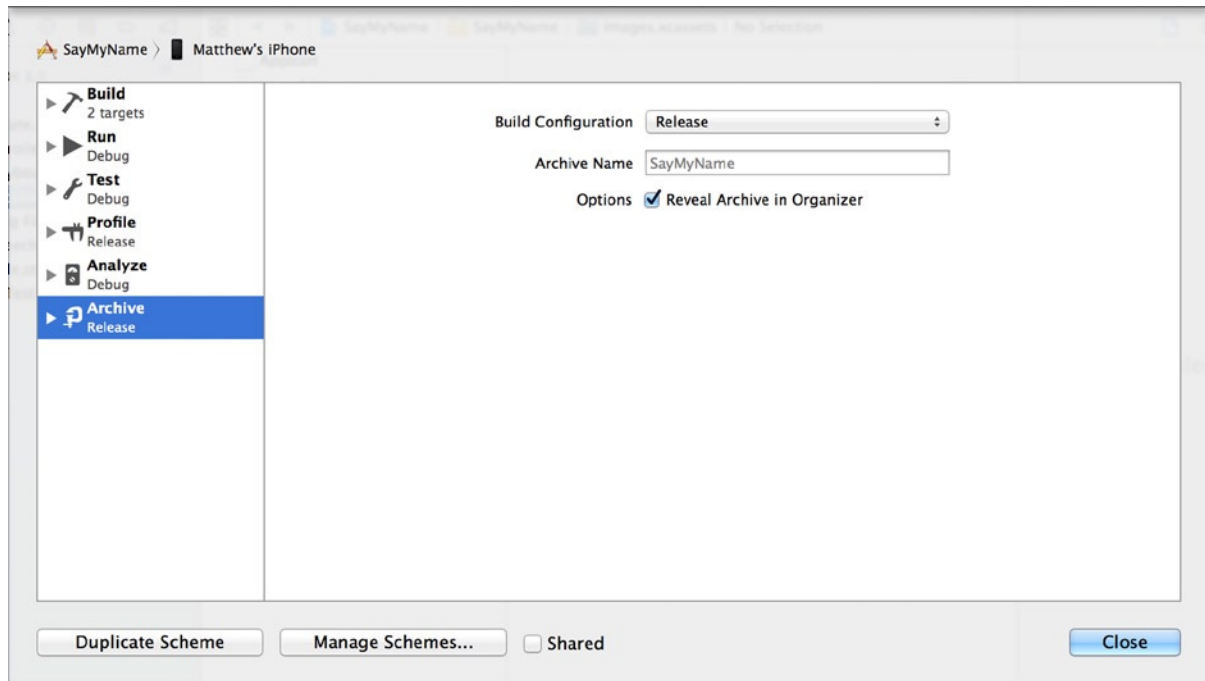


Figure 14-30. The Edit Scheme dialog

You can see that by default, the action is configured to use the release build configuration, that it uses the default name SayMyName, and that upon completion, it will launch the Archives Organizer. You can customize any of these options according to how they would best suit you. Click Close to dismiss this dialog; it's time to try archiving the SayMyName application.

To create an archive of your application, select to Product ► Archive from the menu bar. Xcode goes through a build process before reopening the Archives Organizer, which now contains an archive, as shown in Figure 14-31.

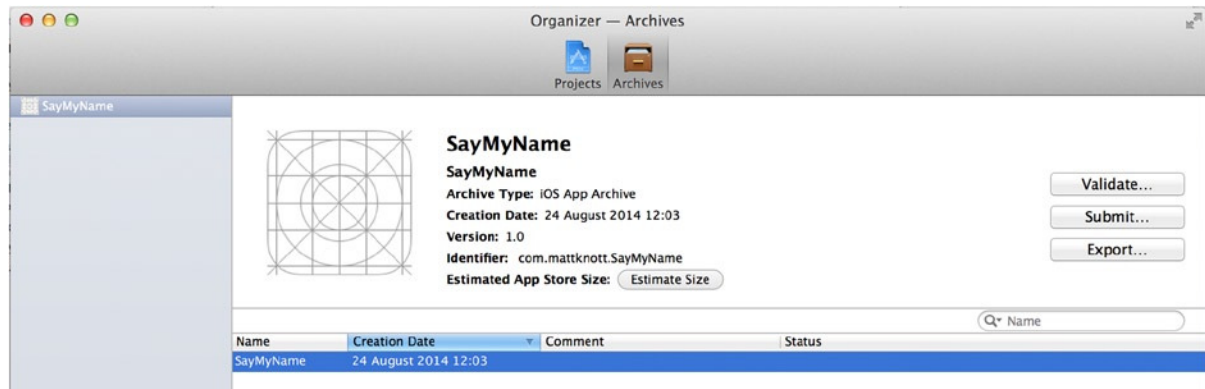


Figure 14-31. The Archives Organizer, looking much more useful with some content

The main information pane shows the application version and bundle identifier, which is useful for differentiating among multiple archives. The Validate, Submit, and Export buttons are covered in Chapter 15, which leaves several small pieces of functionality to discuss here.

First, application file size is something you should try hard to reduce. Notice that next to Estimated App Store Size in the information pane is a button labeled Estimate Size. Clicking this shows you the expected size of your finished application. Think about the end user when adding resources to your project; the smaller the application, the more likely it is that people will download your application over their mobile data connection. Optimize your assets as much as possible by using a photo editor to size and encode your images appropriately for your application.

Second, if you right-click the archive, as shown in Figure 14-32, you can perform actions and access contextually relevant help files. Choosing Show in Finder launches a Finder window in the folder containing the selected archive. This archive can then be distributed to other users on your development team.

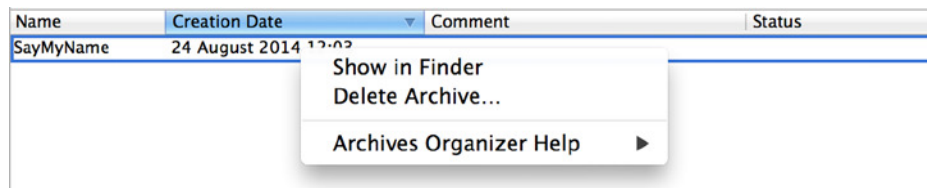


Figure 14-32. Right-clicking the archive reveals several options

Finally, choosing Delete Archive removes the archive from the file system. Before removal, you're prompted whether you wish to delete the archive; in this instance, you would simply confirm the action by clicking the Delete button.

That's as far as I can take you through the Archives Organizer without encroaching on the next chapter. There I show you how to validate your application and submit it to the App Store from this very window.

Summary

It may not be the most exciting facet of Xcode, but if you don't know about Devices or the Organizer, you will be seriously limited in what you can achieve with Xcode. As it stands, you've looked in detail at the key areas of Devices and the Organizer and seen how it can be used to take your applications to the next level.

Specifically, in this chapter you have done the following:

- Learned how to sign up as a paid member of the iOS Development Program
- Registered a physical device, prepared it for development, and deployed your application to it
- Captured an application screenshot
- Discovered how to create snapshots of your application
- Archived your application so it's ready for distribution on the App Store

In the next and final chapter, you complete your knowledge by publishing an application to the App Store!

Building, Sharing, and Distributing Applications

Chapter 14 looked at Devices and the Organizer. The Organizer lets you perform a vast array of actions, from managing iOS devices and project snapshots to preparing archives for submission to the App Store. This chapter continues that theme as you learn how to take your finished application and submit it for hosting on Apple's App Store, the single storefront for both free and paid applications for iOS.

You make the final touches to the SayMyName application that you first created in Chapter 13, by adding an icon to the project. You then move online, where I introduce you to the iTunes Connect portal, a one-stop shop for publishing to the App Store and reporting on download numbers and revenue. You use the iTunes Connect portal to prepare for publishing by creating a profile for your application with all the text and images required for submission.

Finally, this chapter looks at two ways of uploading your application to iTunes Connect, before submitting the application and all the required files to Apple for approval. I'll give you some hints and tips along the way to improve the likelihood of first-time acceptance. You also learn how you can distribute your application. And that's it! Once you've completed this chapter, you'll be ready to dive into writing and sharing your own apps and games!

It's worth noting that being enrolled in the paid iOS Developer Program is essential before you start on this chapter. I covered enrollment in Chapter 14, so if you need to register, now is a good time.

Final Checks Before Publishing Your Application

Before I take you through the process of submitting the SayMyName application to the App Store, you need to add a final touch of polish to complete the application. One of the focal points of this chapter is ensuring that your application has everything it needs to get through the Apple review process and be published to the App Store the first time. One key element that is currently missing from the SayMyName application and that is an absolute prerequisite is an icon.

If you haven't already downloaded the collective resources for the book, then refer back to Chapter 1, where I cover how to access this book's page on the www.apress.com web site and download the resources file. In the Chapter 15 folder, you'll find all the icons needed for both the application and the iTunes Connect portal. The icon provided for the SayMyName application is shown in Figure 15-1.



Figure 15-1. The icon you use for the SayMyName application

I covered the process of setting an application icon in Xcode; but that was back in Chapter 2, so I take you through it again because here you need to set three icons, not just one. Each icon file is named according to its resolution: the main application icon, for example, is named `icon_120.png` because it is 120 pixels × 120 pixels. Here are the steps:

1. Open the SayMyName project in Xcode.
2. Open `Images.xcassets` from the Project Navigator.
3. Select the `AppIcon` image set, as shown in Figure 15-2. Notice that there are six image wells, all of which will hold icons used in different locations in iOS 8.



Figure 15-2. Select the `AppIcon` image set from the `Images` asset catalog to see the six image wells in the set

Note Only one of these sets is a requirement, but you'll set them all for the sake of completeness. Another thing to note is that because this is an iOS 8–only application, and only for iPhone or iPod Touch, all of the image wells are subtitled with 2x or 3x, indicating that they're all retina images. The 3x images are specifically for the new iPhone 6. This is because all the devices in the iPhone form factor that run iOS 8 have retina screens. If you were creating an application that was backward compatible with iOS 6.1, or a universal application, then there would be far more icons to set.

4. Open a Finder window, and navigate to the resources for this book. In the Chapter 15 folder, all the icons are listed. Above iPhone Spotlight iOS 5,6 Settings – iOS 5-8, drag the file named `icon_58.png` into the 2x image well and `icon_87.png` into the 3x image well, as shown in Figure 15-3.

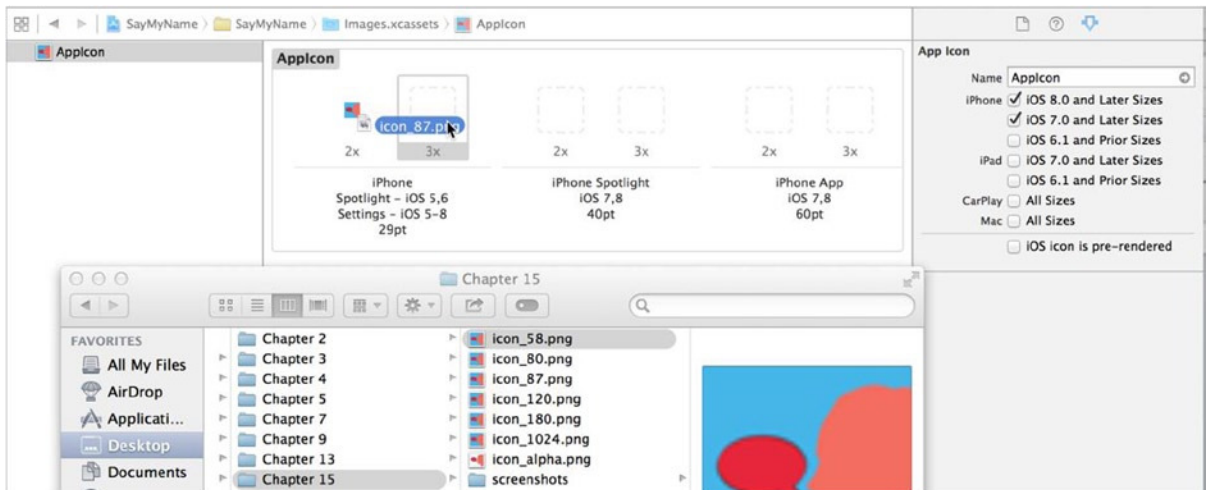


Figure 15-3. Dragging the first of three icons into the `AppIcon` image set

5. Repeat step 4, dragging `icon_80.png` and `icon_120.png` into the 2x and 3x wells in the iPhone Spotlight iOS 7,8 set and `icon_120.png` and `icon_180.png` into the iPhone App iOS 7,8 set.

With your three icons in place, run the application either on a device or in the simulator to ensure that there are no warnings regarding the icons. If you add an icon with the wrong resolution to an image well, then when you build and run the application, you receive a warning similar to that shown in Figure 15-4. If you receive this warning, be sure you added the correct image to the correct well.

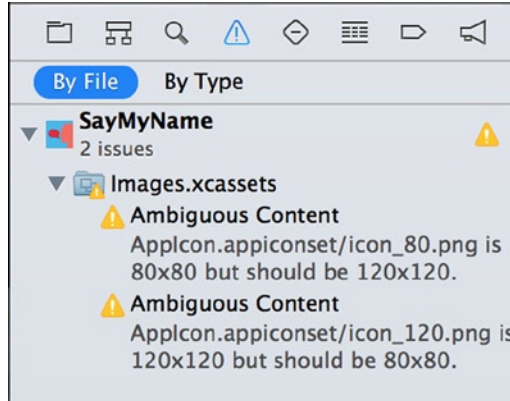


Figure 15-4. Xcode warns you if your icons are the wrong resolution

If you're running your application in the simulator, go to Hardware ► Home (⌘+Shift+H); or, if you're using a physical device, press the Home button to return to the home screen with all the application icons. You should see that the icon for the SayMyName application has been set. Also, searching in Spotlight in iOS uses the 80 × 80 pixel image for the application, as shown in Figure 15-5.

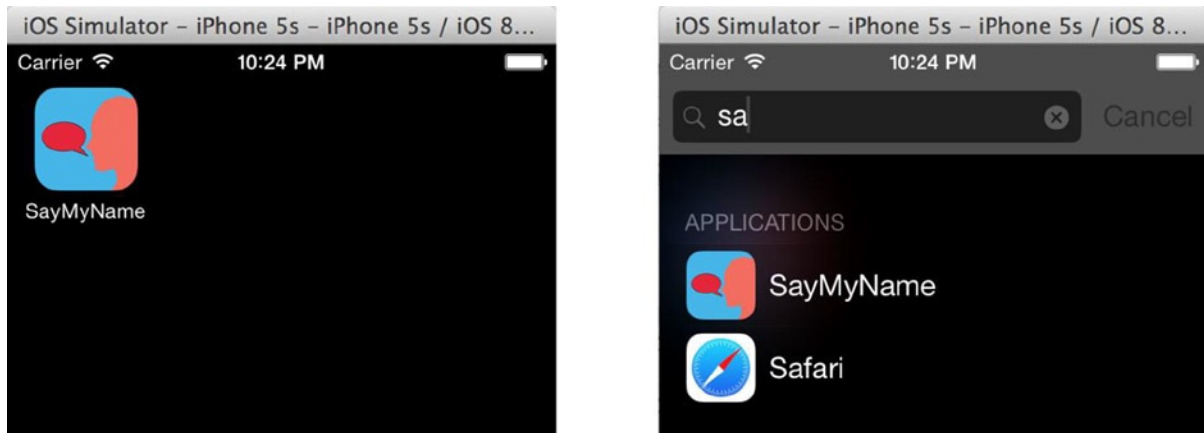


Figure 15-5. The SayMyName application, completed with an icon

Now that you've added the first layer of polish by setting up the icons, it's time to complete the application by setting up a launch screen.

Building a Launch Screen

Prior to Xcode 6, the preferred, default approach to display an image while your app was loading was to use a launch image. This harkens back to when iOS devices were far slower and a screenshot of the app was used to give the impression that the app had loaded before it actually became responsive. In Xcode 6, Apple has changed tactics for the first time since iOS was introduced: the new preferred approach is to use a `.xib` file, something you became familiar with early in this book.

Many app developers use the launch screen to display product branding or partner information. For this app, you create a launch screen that displays the application’s logo in the center:

1. If you’re not still looking at the Images.xcassets asset catalog, select it from the Project Navigator.
2. Open a Finder window, and navigate back to the resources for this chapter. Drag the `icon_alpha.png` icon into the asset catalog sidebar, as shown in Figure 15-6, to create a new image set.

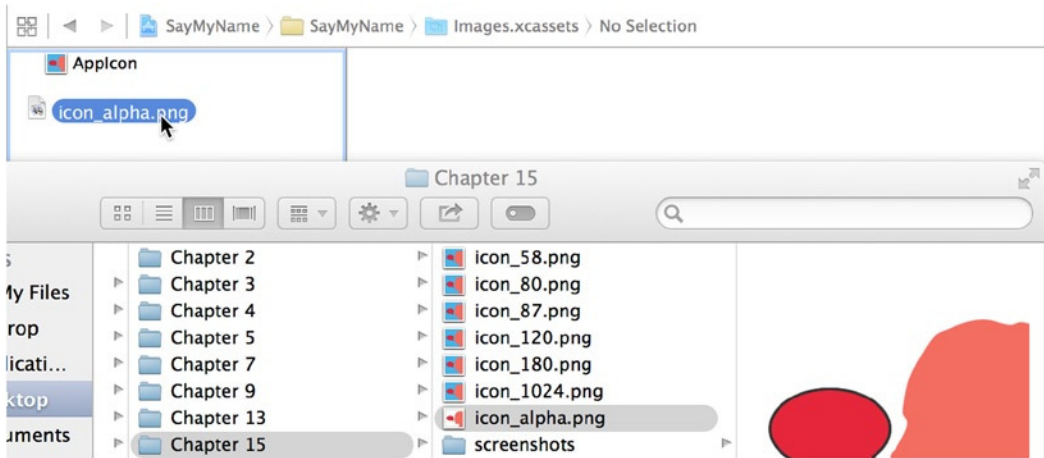


Figure 15-6. Adding `icon_alpha.png` to the asset catalog

3. Open `LaunchScreen.xib` from the Project Navigator. The default launch screen contains the application title and some copyright information, as shown in Figure 15-7.



Figure 15-7. The default `LaunchScreen.xib` created by Xcode

4. Delete the two labels by selecting each one in turn and pressing the Backspace key, so that you have a blank canvas.
5. Drag an image view onto the canvas. By default it fills the canvas, which is fine for now.
6. Go to the Attributes Inspector with the image view selected, and set the Image attribute to `icon_alpha`.
7. Go to the Size Inspector, and set both Width and Height for the image view to 120.
8. Click the pin icon, fix the Height and Width constraints by selecting them, and then click Add 2 Constraints (see Figure 15-8).

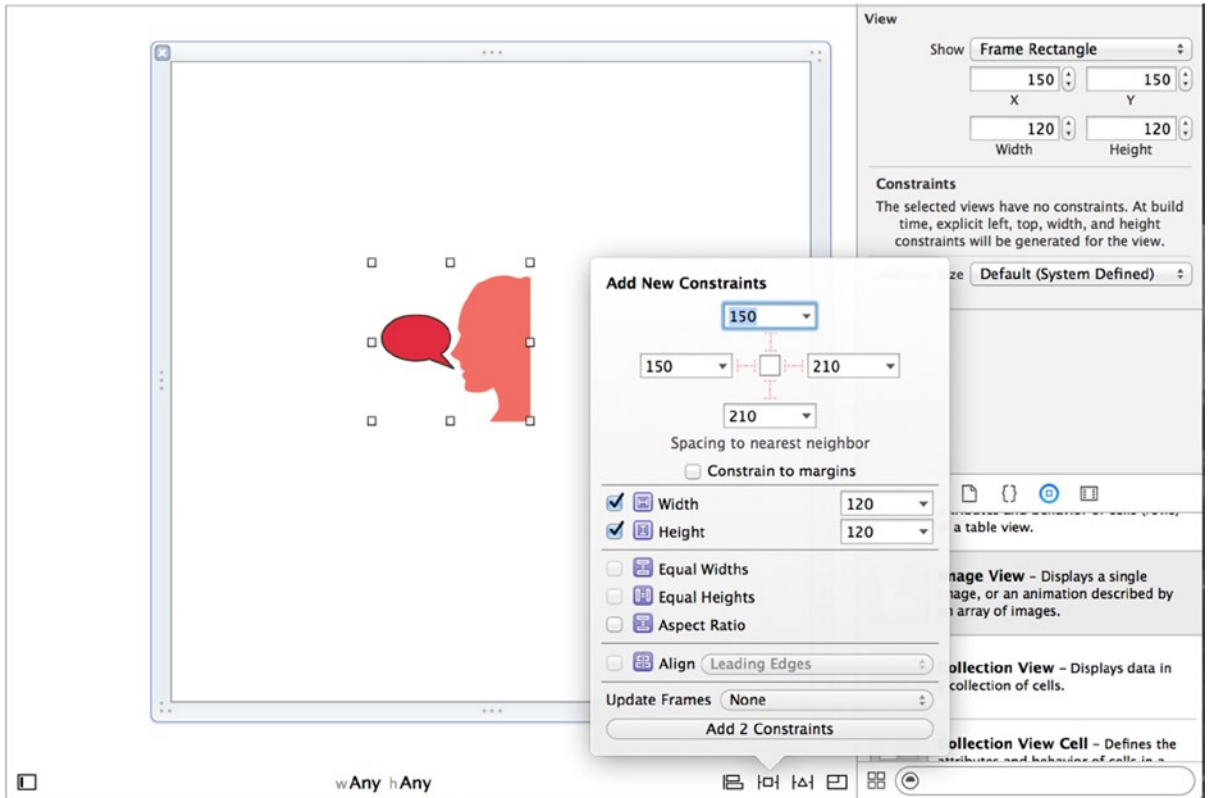


Figure 15-8. Adding Width and Height constraints

9. Select the image view, and go to Editor ► Align ► Horizontal Center in Container.
10. Set the vertical constraint by going to Editor ► Align ► Vertical Center in Container.
11. To see the constraints in effect, click Resolve Auto Layout Issues, and then click Update Frames under the All Views in View heading.
12. With the image view fully configured and positioned, let's change the color of the view to something more interesting. Select the canvas, and then, in the Attributes Inspector, change the background color to anything you want, as shown in Figure 15-9. Although you can't tell in the screenshot, I chose a light blue color.

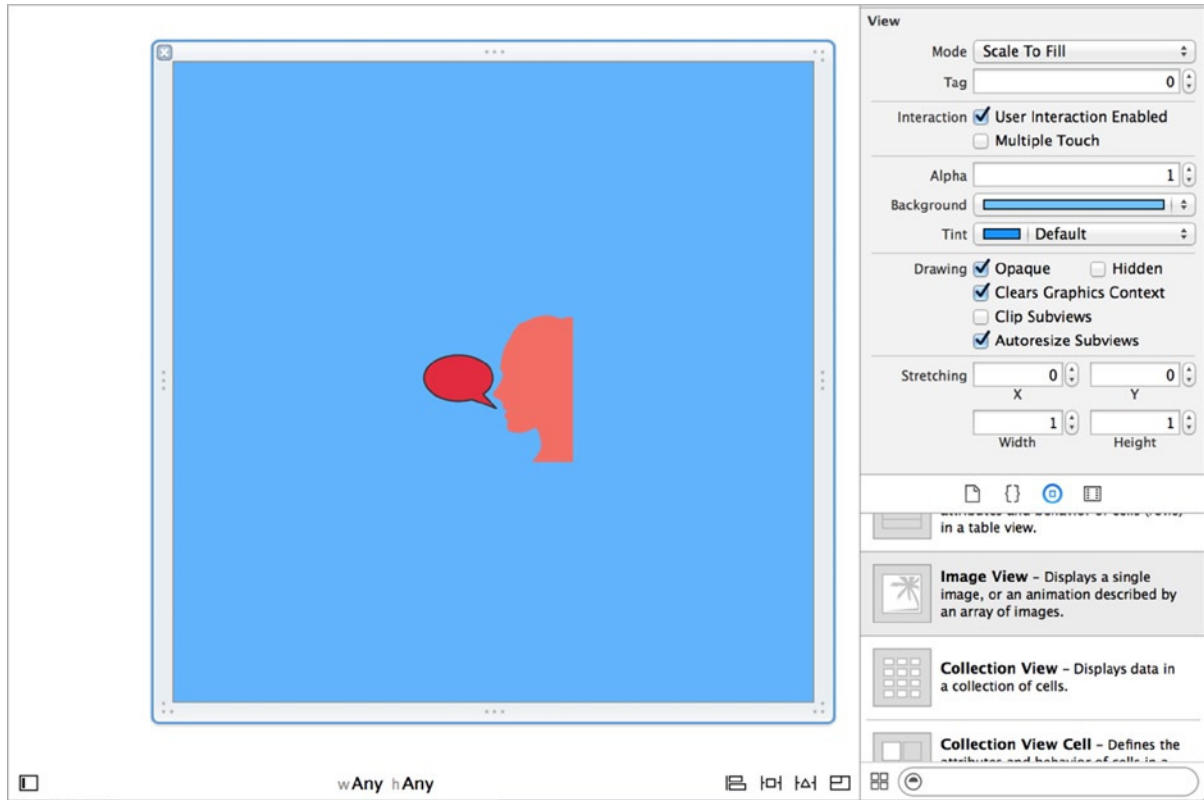


Figure 15-9. The completed launch screen that will greet users every time the app runs

Congratulations: you've just built and configured your first launch screen. If you want to see it in action, run the application. It's time to get down to the nitty-gritty of this chapter and load your application into the App Store.

Discovering iTunes Connect

You now leave Xcode behind for a moment and focus on the functionality available through the iTunes Connect portal. Here you create a profile for your application and set some initial details about the application and its audience.

Note A large portion of this chapter relies on you having an Internet connection and a web browser. I use the default Safari browser, but you should be able to use almost any modern web browser and achieve the same results.

To access iTunes Connect, you can go directly to <https://itunesconnect.apple.com> or, alternatively, reach it from the iOS Dev Center you were introduced to in Chapter 14. The iOS Developer Program is at right on the iOS Dev Center home page, as shown in Figure 15-10.



Figure 15-10. There is a link to the iTunes Connect tool from the iOS Developer portal

When you arrive at the iTunes Connect portal, you're asked to sign in. Because of the highly sensitive nature of the content in iTunes Connect, your connection will time out if left unattended for a short while, so you'll be seeing a lot of the screen shown in Figure 15-11. Enter your iOS Developer ID details, and click the Sign In button.

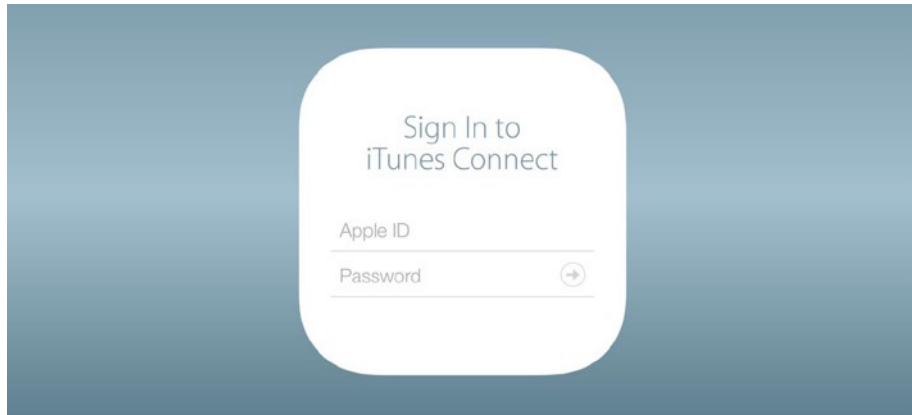


Figure 15-11. Accessing the iTunes Connect portal for the first time

If this is your first time accessing the iTunes Connect portal, you're asked to review and agree to a separate set of terms and conditions, as shown in Figure 15-12. As with other terms and conditions presented to you, it's a good idea to review them before accepting them.

Apple iTunes Connect

knotty1981@googlemail.com ▼

iTunes Connect

TERMS OF SERVICE

THESE TERMS OF SERVICE CONSTITUTE A LEGAL AGREEMENT BETWEEN YOU AND APPLE INC. ("APPLE") STATING THE TERMS THAT GOVERN YOUR USE OF THE ITUNES CONNECT SITE AND THE PRODUCTS AND SERVICES OFFERED THROUGH IT ("ITUNES CONNECT"). TO AGREE TO THESE TERMS OF SERVICE, CLICK "AGREE." IF YOU DO NOT AGREE TO THESE TERMS OF SERVICE, DO NOT CLICK "AGREE," AND DO NOT USE ITUNES CONNECT. YOU MUST ACCEPT AND ABIDE BY THESE TERMS OF SERVICE AS PRESENTED TO YOU: CHANGES, ADDITIONS, OR DELETIONS ARE NOT ACCEPTABLE, AND APPLE MAY REFUSE ACCESS TO ITUNES CONNECT FOR NONCOMPLIANCE WITH ANY PART OF THESE TERMS OF SERVICE.

1. Access. Access to iTunes Connect is provided solely as an accommodation and at Apple's sole discretion, and is available only to (a) authorized representatives of an entity (a "Content Provider") that has one or more valid agreements with Apple or Apple affiliates relating to the offering of the Content Provider's materials on the iTunes Store, App Store, Mac App Store, and iBooks Store, as applicable (each, an "Agreement") (such representatives, "Provider Representative(s)") and (b) artists or authorized representatives of artists whose materials are offered on the iTunes Store by the Content Provider ("Artist Representative(s)"), which Artist Representatives have been granted access to iTunes Connect by the Content Provider or Apple solely for purposes of uploading and managing Artist Images. In addition, limited access to iTunes Connect for certain purposes (including delivery of materials to be offered for potential

I have read and agree to the above Terms of Service

Figure 15-12. iTunes Connect has separate terms and conditions that must be agreed to

Once you have agreed to the terms and conditions, you arrive at the iTunes Connect dashboard, shown in Figure 15-13.



Figure 15-13. The dashboard of the iTunes Connect portal

As you can see, numerous sections are available from this dashboard:

- **My Apps:** As far as this chapter is concerned, this is where it all happens. Here you create and manage your application profiles and control how they appear on the App Store.
- **Sales and Trends:** Get reports on the number of downloads for a particular timeframe.
- **Payments and Financial Reports:** Get a detailed breakdown of payments, and also view trends in your app-generated earnings.

- *iAd*: Start here if you want to make iAds for your products or if you want to include them in your application.
- *Users and Roles*: You can give others access to the iTunes Connect portal. You can control permissions—restricting users from seeing the financial aspects, for example.
- *Agreements, Tax, and Banking*: Here you can view your tax details and request contracts in relation to the developer program. You can also view any transfer agreements you may have in place.
- *Resources and Help*: You can access various support mechanisms for issues when submitting your apps, including FAQs, a contact form, and a link to the developer forums, where you can crowd-source a solution from iOS developers the world over.

Clearly there is a lot you can do with iTunes Connect. For now, click My Apps so you can begin creating an application profile for the SayMyName application:

1. You're taken to an empty list (assuming you haven't created any applications already). At the top of the page is a + icon. Click it, as shown in Figure 15-14, and you're presented with two options: New iOS App and Create Bundle.

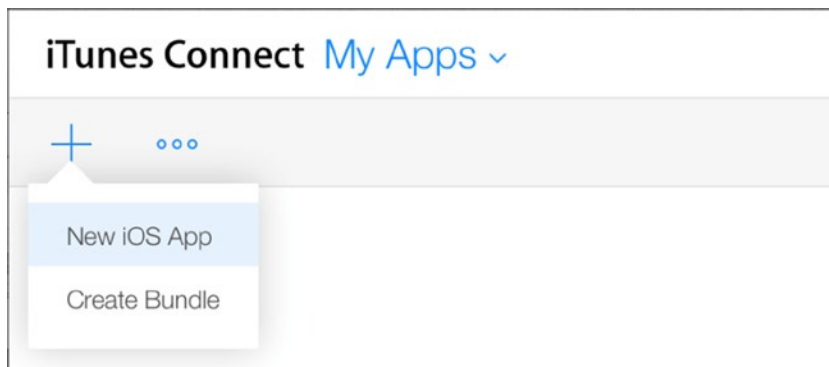


Figure 15-14. The Add dialog on the My Apps page in iTunes Connect

Note If you have a Mac developer account, you see a New Mac App option as well.

2. Click the New iOS App link. A dialog appears in which you need to specify some basic information about your application, as shown in Figure 15-15:
 - a. The Name field should contain the name of the application as you want it to appear on the App Store; I chose “Say My Name!”.
 - b. Set Language to English and the Version value to 1.0.
 - c. The SKU field is for you to specify an identifier that is unique for you that will help you differentiate this app from others you may publish; I chose SayMyName.

The screenshot shows a 'New iOS App' configuration window. It contains the following fields and controls:

- Name**: Text input field containing 'Say My Name!'.
- Version**: Text input field containing '1.0'.
- Primary Language**: Dropdown menu with 'English' selected.
- SKU**: Text input field containing 'SayMyName'.
- Bundle ID**: Dropdown menu with 'Choose' selected.

Below the Bundle ID field, there is a text label: 'Register a new bundle ID on the [Developer Portal](#).' At the bottom right of the form are two buttons: 'Cancel' and 'Create'.

Figure 15-15. Setting the basic app information

3. The Bundle ID select list shows a wildcard app ID when you select it. Depending on what you want your application to do now and in the future, it's worthwhile thinking about this step very carefully. For now, below the select list is the text Register a New Bundle ID on the Developer Portal; click Developer Portal. A new window or tab opens, showing the New App ID page.

Creating an App ID

An App ID links your application into Apple services such as iCloud, Game Center, and Pass Book; it also allows you to enable push notifications. There are two types of App ID, and the services you want your application to access both now and in the future will influence the type you use:

1. Give the App ID the name SayMyName, as shown in Figure 15-16.

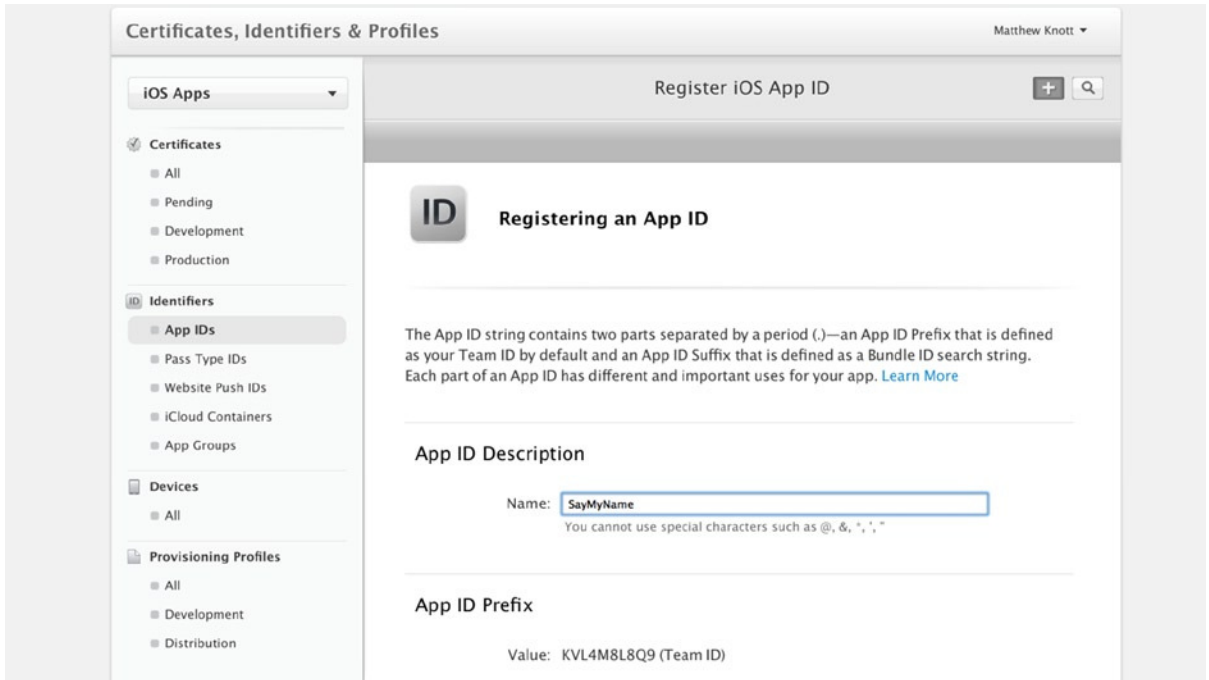


Figure 15-16. Registering a new App ID

2. You need to specify the type of App ID you want to register. I briefly explain each type next:
 - *Wildcard App ID:* Can be shared between multiple applications. You can enable Data Protection, iCloud, Inter-App Audio, and Passbook services with a Wildcard App ID. This type of App ID is useful if you have a series of really basic applications that don't need access to the full range of Apple services.
 - *Explicit App ID:* Gives you access to all Apple services that are available to the Wildcard App ID, with the addition of Game Center, In-App Purchase, and Push Notifications. If you aren't sure what you want to do with your application in the future, choose an Explicit App ID to reduce future hassles.

In this instance, choose Explicit App ID. Enter the Bundle ID that was set for the SayMyName application when you created it in Xcode in Chapter 13 (see Figure 13-2), which in my case was `com.mattknott.SayMyName`, as shown in Figure 15-17.

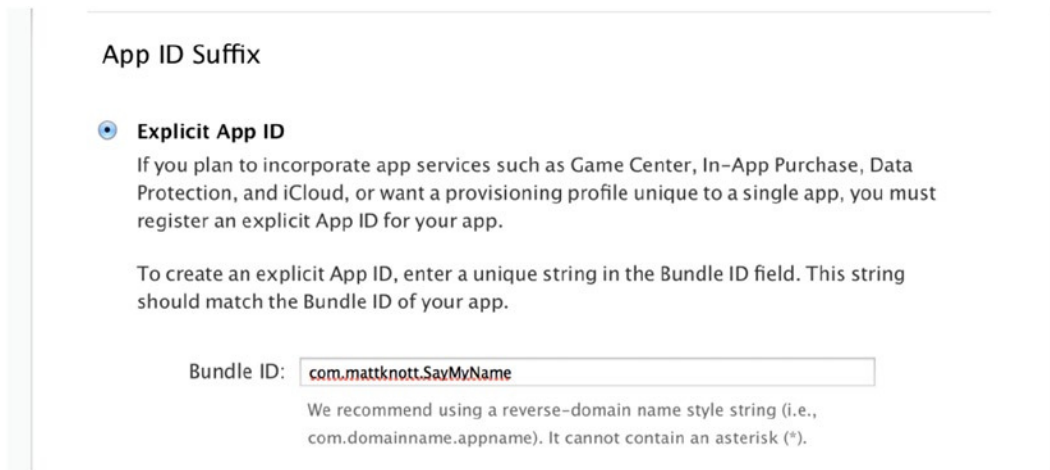


Figure 15-17. Setting the Bundle ID value for the Explicit App ID

If you can't remember the Bundle ID you specified, you can find out quickly by going back into Xcode and selecting the SayMyName project in the Project Navigator, choosing the SayMyName target, and clicking the General tab. Figure 15-18 shows that following this path, you see the Bundle ID value that was specified when you created the application. Enter this value as the Bundle ID in your web browser with the product name instead of the placeholder.

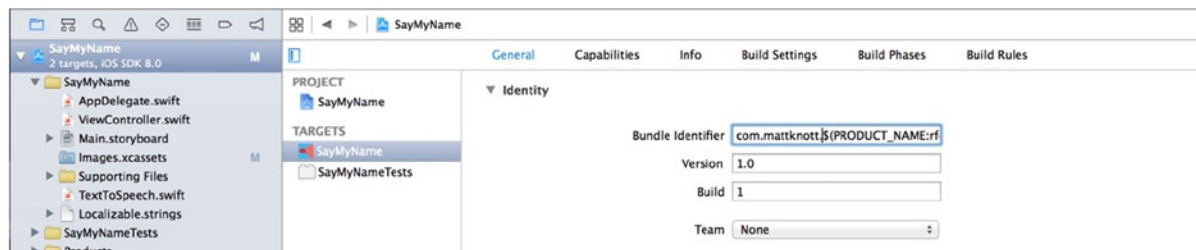


Figure 15-18. Checking the Bundle ID is quick and painless

Note In Xcode 6, the Bundle ID is partly dynamic: the last part of the identifier is the product name. This is visible in the Build Settings tab of the Target Settings area.

- There is no need to enable any additional services, so scroll to the bottom of the page and click the Continue button.
- On the next screen, you're shown a summary of the App ID details, including which services are enabled (see Figure 15-19). Don't worry that Game Center and In-App Purchase are enabled; this is the case for every Explicit App ID, and you don't have to use them. When you're happy with everything, click the Submit button at the bottom of the page.

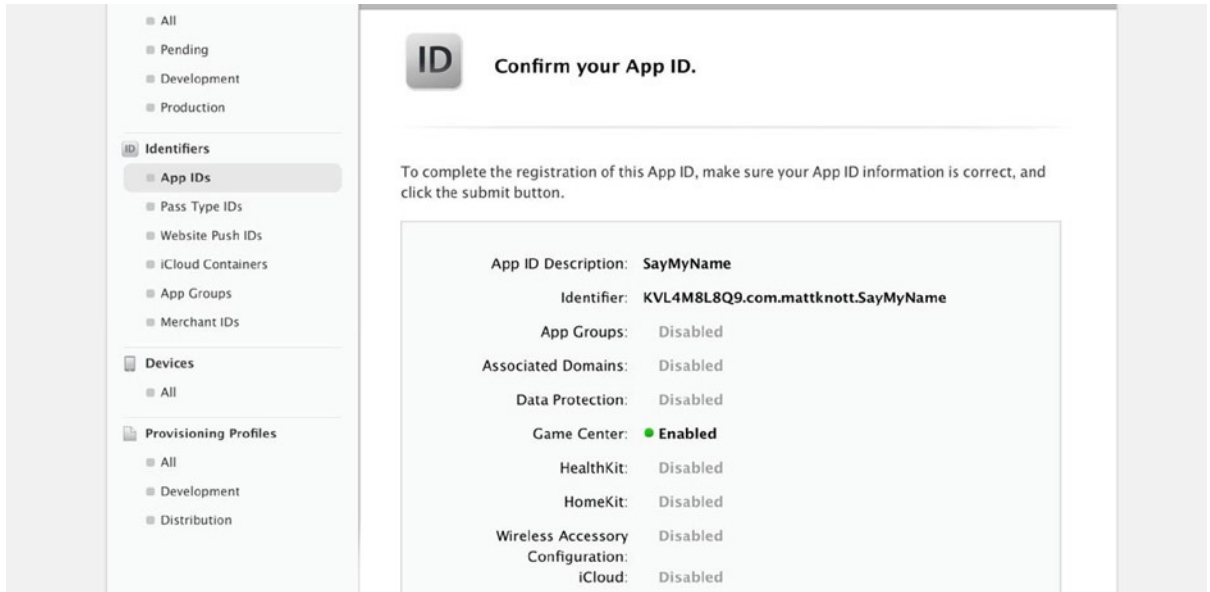


Figure 15-19. The App ID summary page

5. You're taken to a repeat summary page. The App ID has been created. Click Done to return to the list of App IDs, which now contains your Explicit App ID.

That's it: you have everything you need to create an application profile.

Creating an Application Profile

It's frustrating when you have to rip something up and start again as you just did when you abandoned the Application Profile, but in this case, at least you only typed a couple of words:

1. Go back to the iTunes Connect window or tab you were using in your browser, click the Back button, and then once again click the Add New App button that was shown in Figure 15-14.
2. Again, specify English as the default language, enter the app name (mine is "Say My Name!"), and enter the SKU value SayMyName.
3. This time, when you click the Bundle ID select list, your new App ID is in the list; select it, as shown in Figure 15-20, and click Create.

New iOS App

Name ?
Say My Name!

Version ?
1.0

Primary Language ?
English

SKU ?
SayMyName

Choose
Xcode iOS Wildcard App ID - *
✓ SayMyName - com.mattknott.SayMyName

Register a new bundle ID on the [Developer Portal](#).

Cancel Create

Figure 15-20. Selecting the Explicit App ID you created

You're taken to the Versions tab of the Say My Name! app-management page, as shown in Figure 15-21. You need to specify all the information required to submit your application to the App Store. This page is split into sections that you navigate and configure in order, as explained next.

iTunes Connect My Apps ▾

Matthew Knott ▾
Matthew Knott|9228503

< My Apps

Say My Name! iOS
● 1.0 Prepare for Submission

Versions Prerelease Pricing In-App Purchases Game Center Reviews Newsstand More ▾

1.0 Save Submit for Review

Figure 15-21. The Say My Name! application-management page

Adding Screenshots and Video Previews

As you scroll down the page, the first section to be configured is for App Video Preview and Screenshots. There are five different formats for screenshots: 4.7-inch (iPhone 6), 5.5-inch (iPhone 6 Plus), 4-inch (such as iPhone 5s), 3.5-inch (such as iPhone 4s), and iPad. This is not an iPad app, so you only need to set the first four screens. To save time, the resources for this chapter contain four different sizes of screenshots, ready for upload, in the screenshots folder.

Note One of the new features of the App Store with iOS 8 is that you can upload videos to accompany your screenshots so customers can see your app or game in action before buying. Although you only add screenshots for this project, videos are a great way to demonstrate how slick your application is.

Start by making sure the 4.7-Inch tab is selected, and then drag in the `screen47.png` file from the Finder, as shown in Figure 15-22. Repeat this step for the remaining tabs until one screenshot is uploaded for each applicable screen size.

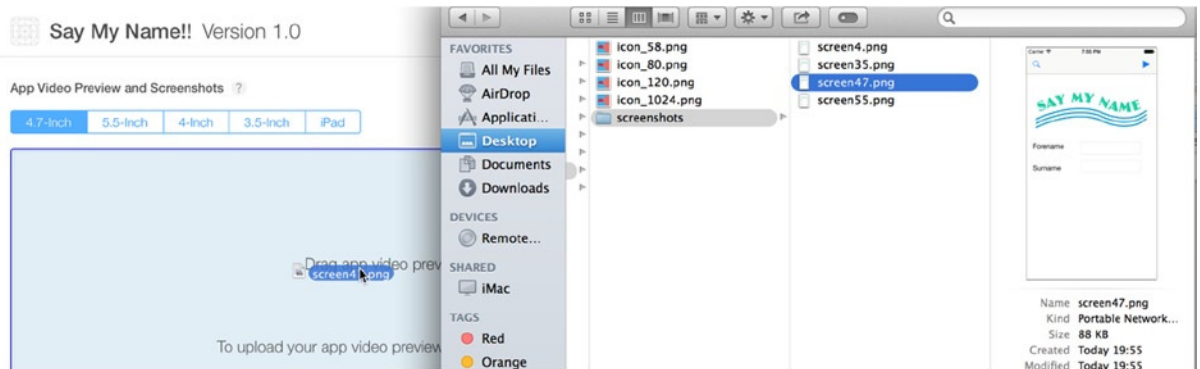


Figure 15-22. Dragging in a screenshot for upload

Adding Application Metadata

Scroll to the next section, traditionally referred to as the metadata area, as shown in Figure 15-23. In this segment you can specify six pieces of information, two of which are optional:

- **Name:** The application name as it will appear in the App Store.
- **Description:** This will appear on the App Store page for your application and should focus on the app's purpose and key features.
- **Keywords:** Words that help your application appear when users search the App Store. You can have 100 characters' worth of keywords, which you separate with commas.
- **Support URL:** The primary support URL for your application. You must have a web site set up to support users with your application. The site doesn't have to be specifically about your application; it can be your personal or business web site.
- **Marketing URL:** This is optional, but a marketing URL is displayed to users in the App Store so they can discover more about your app before they purchase or download it.
- **Privacy Policy URL:** If your application collects user data in any way, you should help your users feel confident that you won't abuse their information by publishing a link to your privacy policy page on the Internet.

Say My Name!! Version 1.0 Save Submit for Review

Name ?
Say My Name!

Description ?
Pick a contact from your phone book and hear the name spoken in your regional dialect (where available)

Keywords ?
Utility, Text to Speech, Pointless

Support URL ?
http://www.mattknott.com

Marketing URL ?
http://example.com (optional)

Privacy Policy URL ?
http://example.com (optional)

Figure 15-23. The metadata area contains searchable information and key URLs for the SayMyName application

Enter the values you feel are relevant. You must provide a description, at least one keyword, and a support URL.

Adding General Application Information

It's good to click Save periodically, so go ahead and do this now. Then scroll down to the next section: General App Information. The section is split into two columns. The right column is automatically populated with your personal address information, so let's start with the left column.

First, you need to set the icon for the application that will appear in the App Store. This is straightforward: click the icon, and a file browse dialog appears, as shown in Figure 15-24. Navigate to the resources for this application, and select the file named `icon_1024.png`.

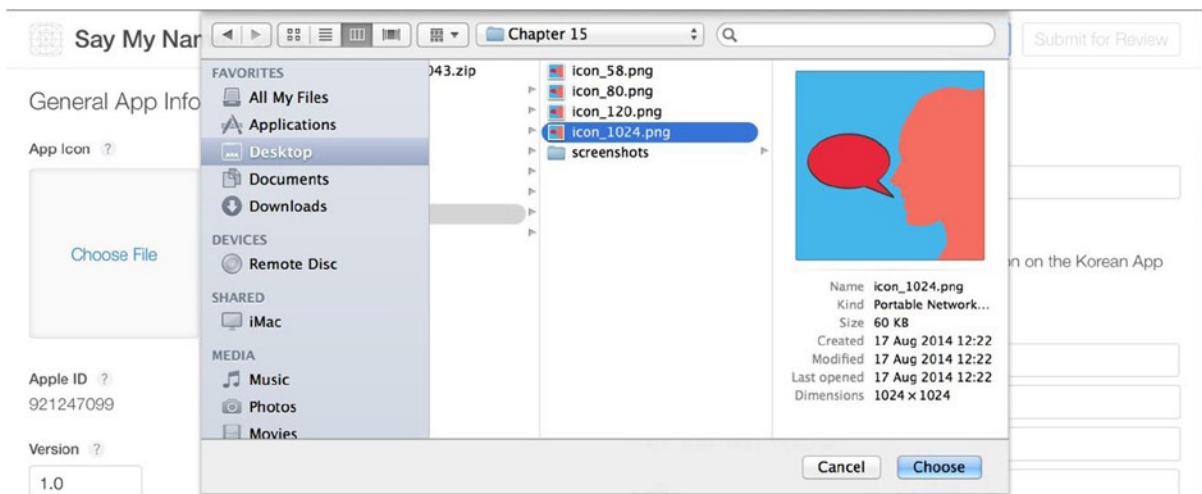


Figure 15-24. Selecting the large-format icon for the App Store

The next value, Version, is already set at 1.0. The next segment in this column lets you set primary and secondary categories in which the app will appear. Looking at the options in the Primary category list, you see that they reflect those available in the App Store. You need to set a primary category; I went with Utilities, as shown in Figure 15-25, because it's most appropriate for this application. Choosing a second category means your app will have greater visibility to users browsing the apps in the App Store, however, in this instance, one category will suffice.

The screenshot shows the 'Store' configuration page in App Store Connect. On the left, a dropdown menu for 'Primary' categories is open, listing various categories such as Books, Business, Catalogs, Education, Entertainment, Finance, Food & Drink, Games, Health & Fitness, Lifestyle, Medical, Music, Navigation, News, Photo & Video, Productivity, Reference, Social Networking, Sports, Travel, Utilities, and Weather. The 'Utilities' category is selected and highlighted in blue. On the right, the 'Store' form is filled out with the name 'Matthew Knott', address, city, state, postal code, phone number, and email. A 'Routing App Coverage File' section is also visible with a 'Choose File' button.

Figure 15-25. Selecting a primary category for the application

The next item to set in the left column is the rating, which currently says No Rating. Next to the word *Rating* is an edit link; click it, and a dialog appears as shown in Figure 15-26. Here you must specify the types of content that may appear in your application so that an appropriate content warning may be given to users downloading the application. The Say My Name! app doesn't have any of the listed content types, so I selected the None value for all of them. At the bottom of the list are new values for Unrestricted Web Access and Gambling and Contests; I selected No for both of these. Complete this form, and click Done. When you return to the application information screen, you have a rating of 4+, which is as low a rating as can be achieved.

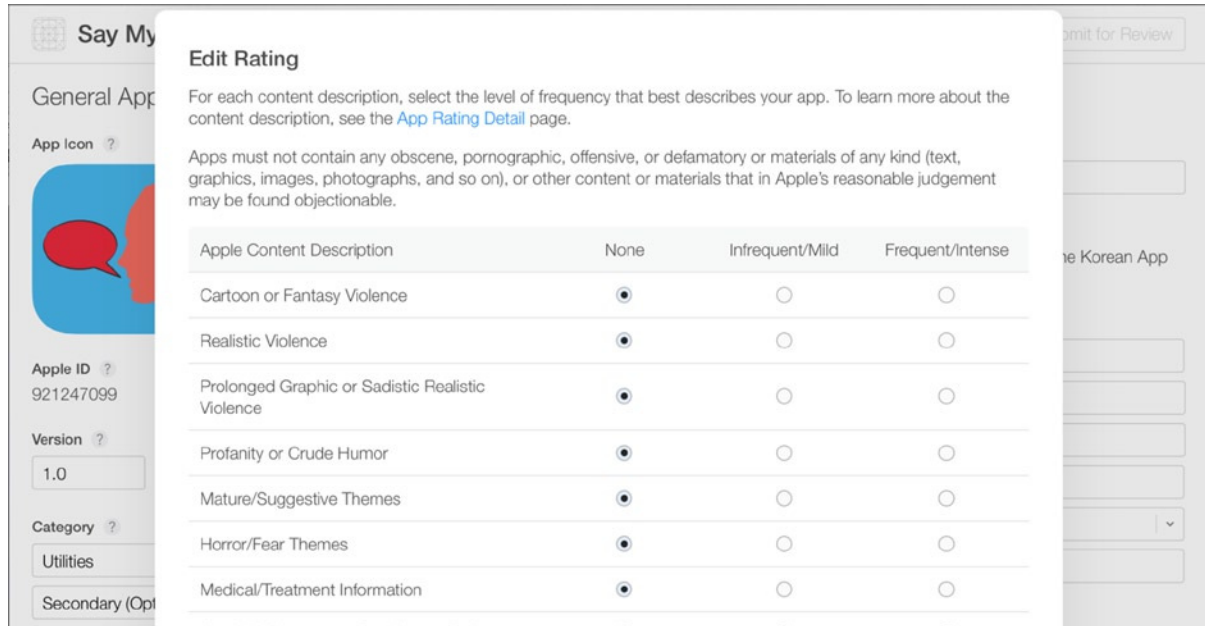


Figure 15-26. Obtaining the application's rating by specifying its content

Finally for the left column, you aren't setting a custom license agreement, so you can skip this option.

In the right column, for the Copyright field, you usually enter the year and your name or company. I entered 2014 Matthew Knott.

The Address field should be filled in, but if it isn't, specify the required values.

The final segment, Routing File, was introduced with iOS 7 and Xcode 5, which allow you to develop applications that can be used when planning routes in the iOS Maps application. This is where you upload the associated coverage file that indicates the geographic areas your application covers.

Adding Release Information

The final few sections largely relate to the release and review of the application. The Build section is explained later in the chapter and is relevant only after you've uploaded your application to iTunes Connect.

The Newsstand section is only applicable for Newsstand-integrated applications. This isn't one, so move to the next section.

In the App Review Information section, you specify contact information for the person tasked with overseeing this application submission. In this instance it's you, but in a large organization a specific program manager may be assigned this task. The Demo Account Information segment is only used if your application is secured with a username and password and you have a test account set up. You enter the details here so that the person testing your application submission can test all of the functionality.

The final field, Notes, is optional; but many people, including me, have found that adding a kind word for the person reviewing your application can have a hugely positive impact on your application being accepted the first time. I always thank them for their work and tell them I hope they have a great day, and so far all of my app submissions have gone through the first time. So, I recommend that you take the time and write your own positive message, as shown in Figure 15-27.

Figure 15-27. Setting the App Review Information

The last configurable section on this page is the Version Release section. Here you simply specify whether you want the application to be released the minute it's approved, or whether you have a more specific date in mind. In this instance, select Automatically Release This Version, as shown in Figure 15-28.

Figure 15-28. Setting the Version Release information

Finish by clicking the Save button at the top of the page, and then address any validation errors.

Setting Rights and Pricing Information

Now that you've set the bulk of the information for your application, it's time to switch to the Pricing tab and specify how much you're charging for the app. One of the first things that surprised me about the Pricing page is that, as you can see in Figure 15-29, Apple hasn't modified it to fit in with the style of the rest of iTunes Connect. This leads me to believe that they will change it in the near future, so please accept my apologies if you're looking at a different screen.

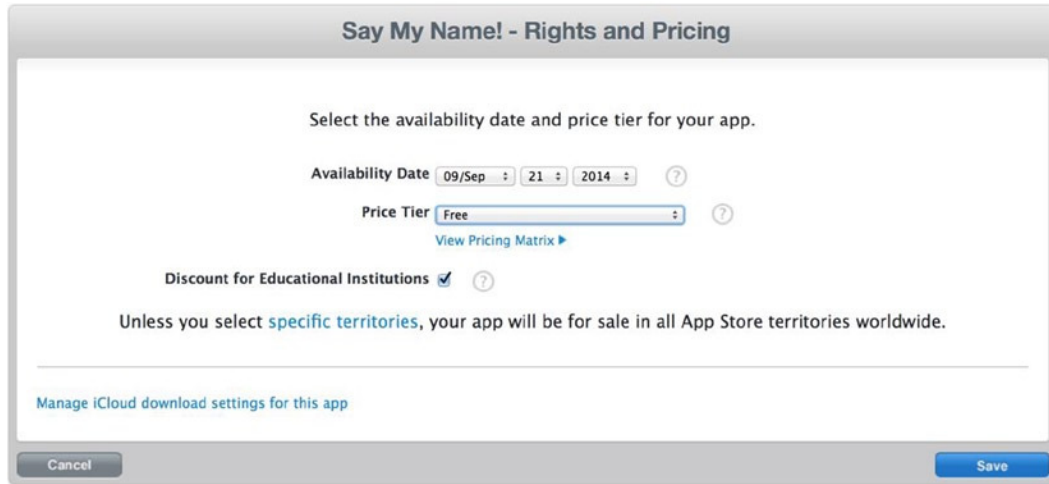


Figure 15-29. Setting the pricing and availability of your application

The availability date you set here can determine the earliest date this application can be available from; this is largely aspirational to some extent, it will default to the current date, but it could take some weeks to be approved, so if you want it to be released once approved, leave the current date selected.

The Price Tier choice is up to you. I selected Free, but with your own applications, this is obviously a choice you need to make. If you opt for one of the pricing tiers, it's good to click the View Pricing Matrix link below the Price Tier select box so you can see the retail price of the app and the proceeds for each territory in which the App Store is available.

Note If you choose to release a paid application, you're asked to supply your bank account details and also to accept the iOS Paid Applications Agreement, which sets out your rights and entitlements. The temptation is to skip through these agreements, but you should always save anything you agree to, because it's legally binding.

Finally, you need to specify whether you want your application to be discounted for educational institutions. As someone who works in the education sector, I can tell you that the cheaper you can make your application for schools and colleges, the more likely it is that they will buy in bulk if they like what they see. The details of the discount may change, so check the specifics of the discount in the Paid Application Agreement that you accepted when you joined the iOS Developer Program to make sure you're happy with what you're agreeing to. Once you've finished making your selections, click the Save button at the bottom of the page.

The page reloads and some additional fields appear, as shown in Figure 15-30. Price Tier Effective Date and End Date allow you to set exceptions from the pricing tier, meaning you can temporarily discount your application. There is no need to change this now, so scroll down and click Cancel to return to the application profile.

Price Tier Effective Date

Price Tier End Date

Price Tier Schedule		
Price Tier	Price Effective Date	Price End Date
Free	Existing	None

Figure 15-30. Additional fields appear after saving

Congratulations: you’ve just completed your first application profile in iTunes Connect! You’re ready to move on to uploading your application.

Uploading an Application to iTunes Connect

You’ve completed your application’s profile, but there are still some key steps to go through in order to submit your application to the App Store. In a sense, all the information you just entered is metadata; it’s there to support the real thing you want to share, which is the binary file for your application.

Creating a Distribution Certificate and Profile

In order to prove that you’re the owner of the binary file you’re uploading, you need to build and sign the binary using an iOS Distribution certificate. This is something a paid-up member of the iOS Developer Program is able to do, and the creation process is quick and relatively painless. In Xcode 6, this should have been created automatically for you; but because of its importance, you should double-check:

1. Go to Xcode. From the top menu, go to Xcode ► Preferences (⌘+,), and then select the Accounts tab.
2. Select your Developer Account from the left column. Click the View Details button in the lower-right corner to display a pop-over containing details about your Developer Account and any linked signing identities and provisioning profiles. If there isn’t an iOS Distribution signing identity, as shown highlighted in Figure 15-31, Xcode should prompt you to create one.



Figure 15-31. With your Developer Account selected, click the View Details button to see the available identities and profiles

3. If you need to manually create an iOS Distribution signing identity, then beneath the list of signing identities, click the + icon. Choose iOS Distribution from the subsequent menu, as shown in Figure 15-32.

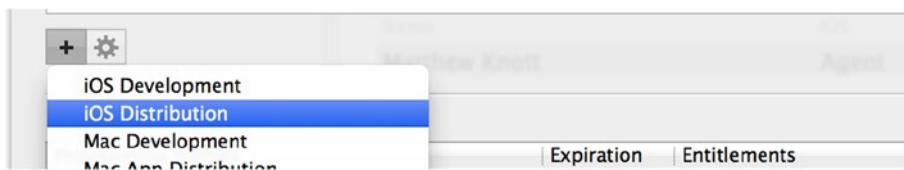


Figure 15-32. Creating an iOS Distribution signing identity

4. At this point, Xcode communicates with Apple while it generates the relevant certificates on Apple's servers. Xcode then confirms that the identity has been created. Click the OK button.
5. You're returned to the Accounts details area, which updates to include a distribution identity alongside the development identity. Click Done to dismiss the pop-over and close the Preferences.

Great—you're finished with Xcode for a moment! It's time to go back online and set up a distribution provisioning profile with the certificate you just created. If you think this is a lot of hassle, you're probably right; but the good news is that setting up the signing identity for distribution only has to be done once. However, you need to create a provisioning profile for each application you submit to the App Store:

1. In your web browser, go back to the iOS Dev Center at <http://developer.apple.com/ios>. Once you've signed in, choose the Certificates, Identifiers & Profiles link from the navigation block on the right; refer to Figure 15-10 if you're not sure what you're looking for.
2. You're taken to a page with three columns: iOS Apps, Mac Apps, and Safari Extensions. Under the iOS Apps heading, click Provisioning Profiles. This takes you to a list of all the provisioning profiles created for your account, as shown in Figure 15-33.

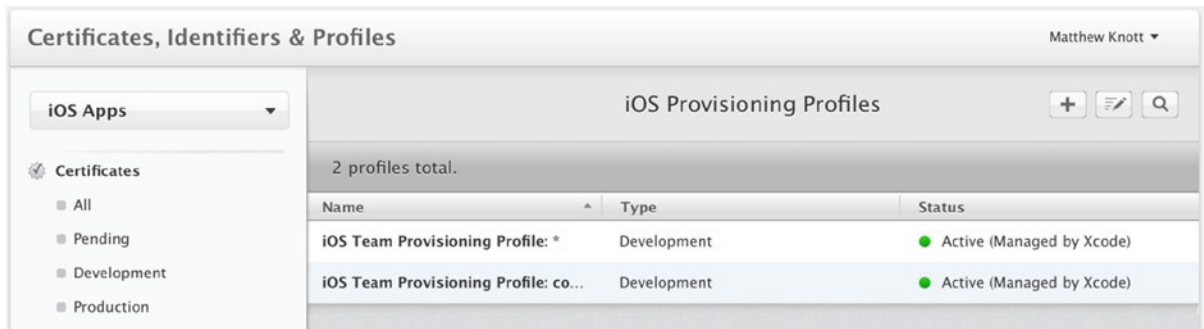


Figure 15-33. Listing all provisioning profiles

3. To the right of the title iOS Provisioning Profiles is a + icon, as shown in Figure 15-33. Click this icon, and you're presented with a form for creating a new provisioning profile. Select the App Store option, as shown in Figure 15-34, and click Continue.

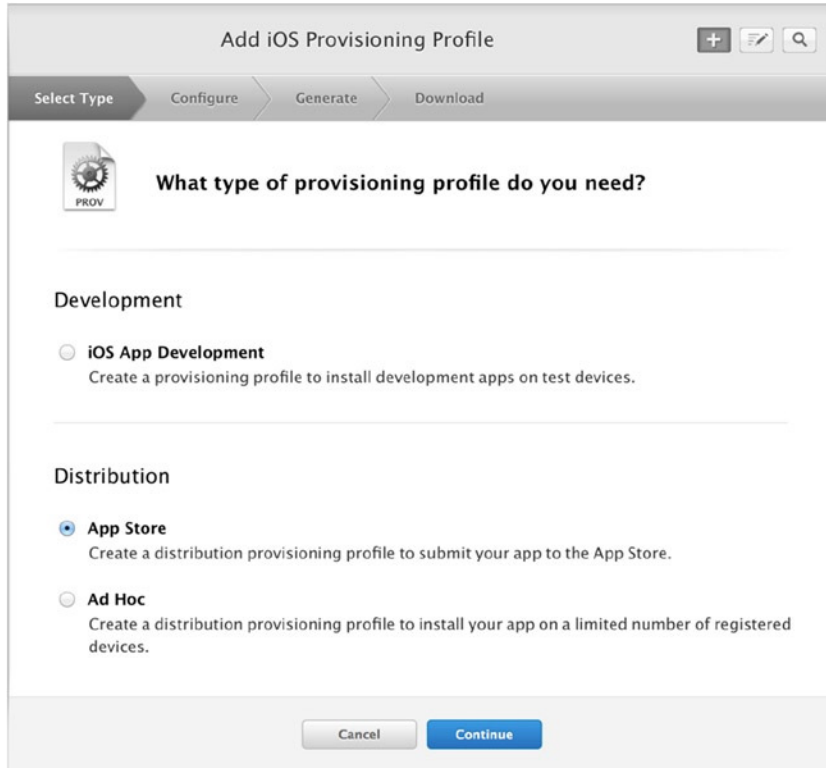


Figure 15-34. Opting to create a new App Store provisioning profile

4. You're asked to select an App ID. If you've followed the steps so far, then SayMyName should appear, as shown in Figure 15-35. If not, select it from the list. When you're happy with the selection, click Continue.

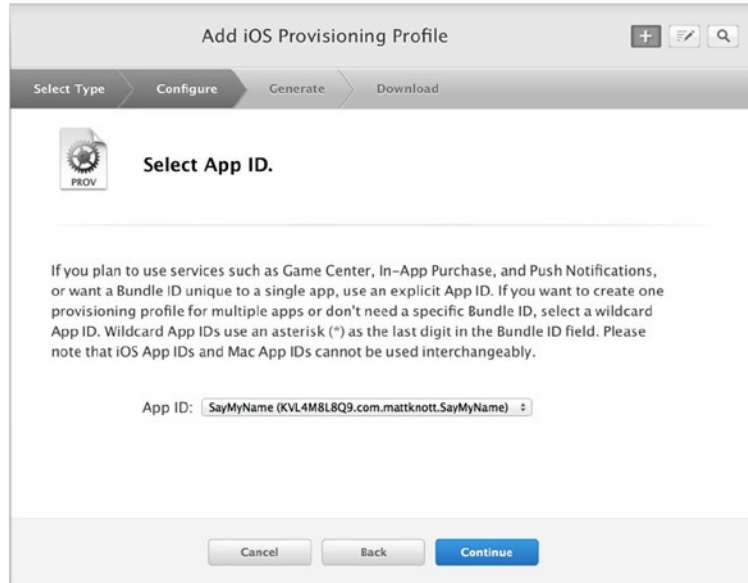


Figure 15-35. Selecting the App ID that the provisioning profile is for

5. You're prompted to select a certificate. Listed here should be the iOS Distribution certificate you just created in Xcode. It's not selected by default, so click the radio button next to it to highlight it, as shown in Figure 15-36, and click Continue.

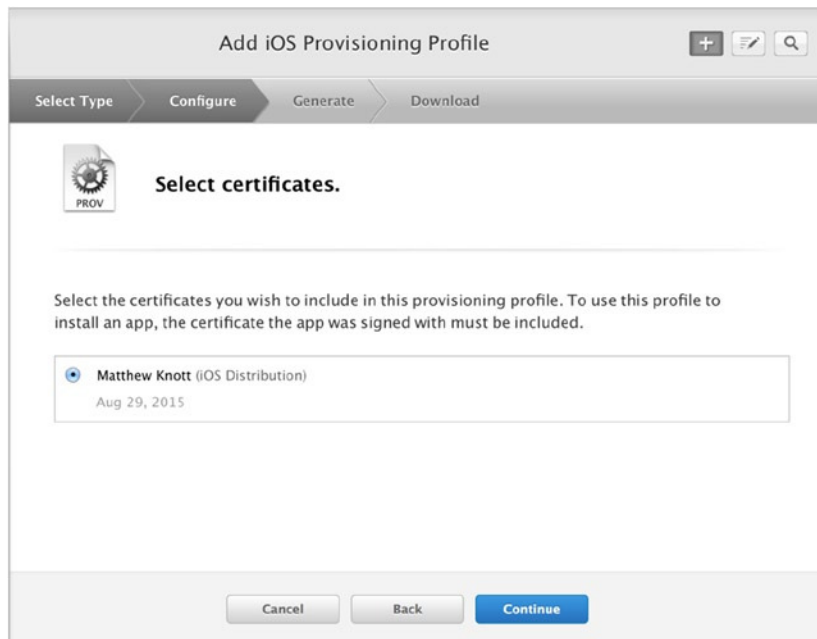


Figure 15-36. Selecting the certificate to use with this provisioning profile

- The last step in generating a provisioning profile is to give it a name. Because this is specific to the application, type **SayMyName** in the Profile Name field, as shown in Figure 15-37, and then click Generate.

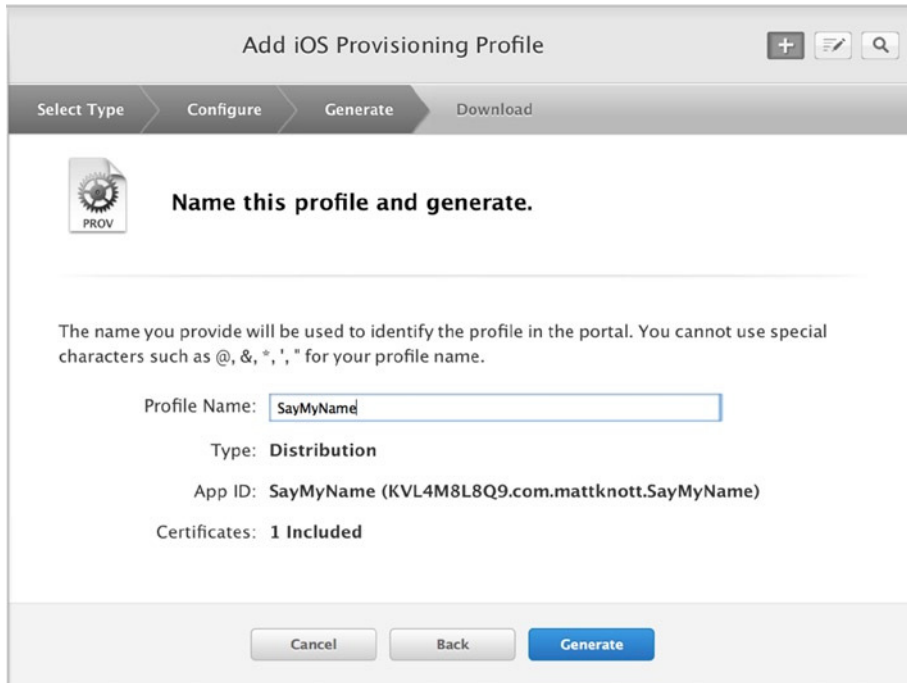


Figure 15-37. Naming the profile SayMyName is the last step before generating the profile

- It will take a moment for your profile to be generated, but eventually you're taken to a confirmation page, as shown in Figure 15-38. Click Done at the bottom of the page, and return to Xcode.

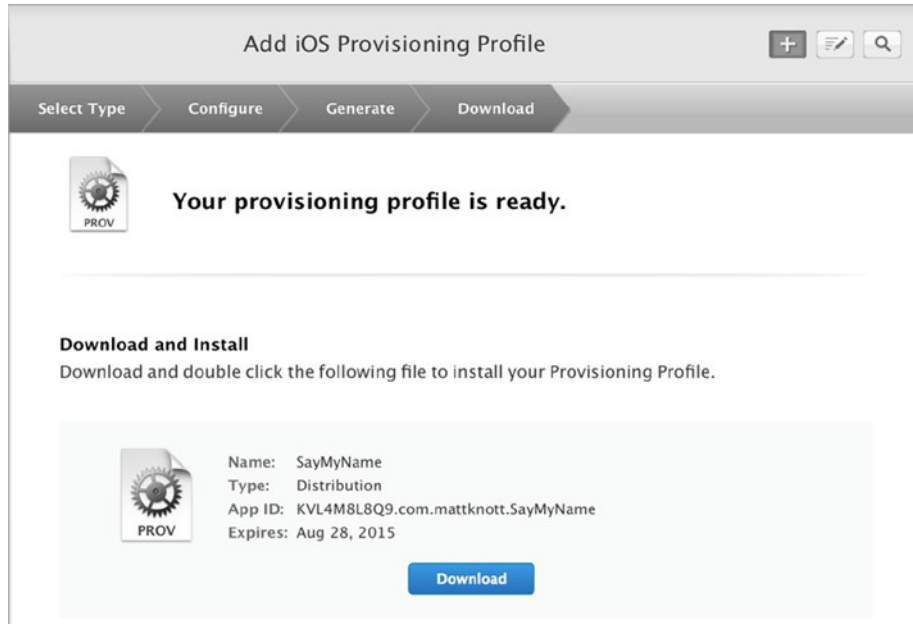


Figure 15-38. Your profile has been created!

Note You could download and install the profile at this point by clicking the Download button and then running the downloaded `SayMyName.mobileprovision` file. However, it's a good demonstration of Xcode's flexibility in fetching missing information to let it handle the whole thing.

Validating Your Application

You may remember that in Chapter 14 I discussed the Archives Organizer in some detail, but I didn't cover the validation or distribution features available for an archived application because they form an essential part of the uploading process. By validating your application in Xcode before submission, you can identify any major issues with the app and rectify them, creating a smoother submission process for yourself. Validating your application doesn't take long at all:

1. In the Archives Organizer, select the newest archive for SayMyName, as shown in Figure 15-39, and click the Validate button.



Figure 15-39. The Archives Organizer has a validation feature built in

2. You're prompted to select a development team. There should only be one entry for you to choose from; select it, and click Choose.
3. Xcode presents an overview of the libraries and components that make up the SayMyName application, as shown in Figure 15-40. Click Validate.



Figure 15-40. Xcode gives you a list of elements to be validated

4. After about 30 seconds, your application should either succeed or fail. If it succeeds, as it should, you're presented with the confirmation shown in Figure 15-41. Click Done to dismiss the validation.

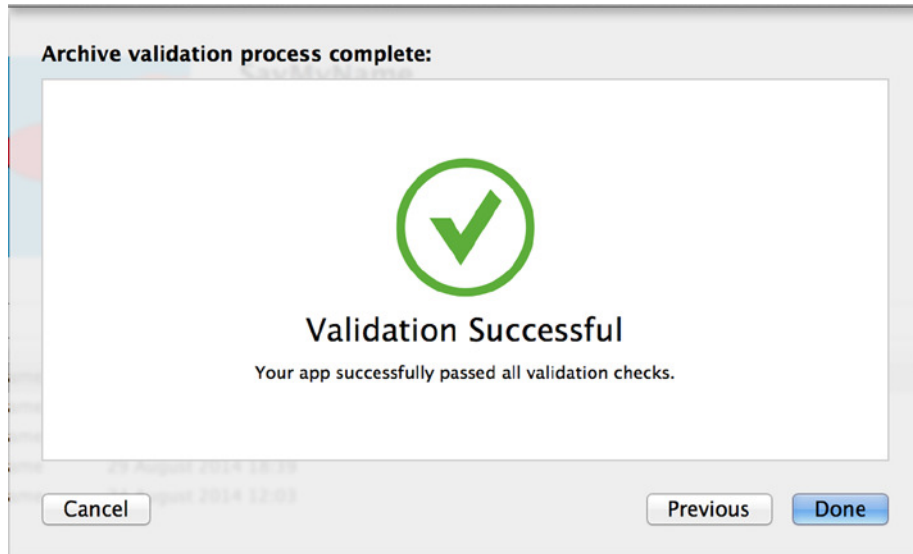


Figure 15-41. The *SayMyName* application has been successfully validated

Now that you know your application has a clean bill of health, it's time to submit it to the App Store! If there are any issues with your submission and you aren't clear on the reason for rejection, head to the Resources and Help section of iTunes Connect that was shown earlier in the chapter, where you can find all the information you need to set about correcting the issue.

Submitting Your Application to the App Store

After all the buildup, it's finally the moment of truth, the culmination of 15 chapters of learning and growing as a developer: you're ready to submit your application to the App Store. I actually show you two ways of submitting your application to the App Store, and I explain why later in the chapter. For now, let's take a look at the first submission method.

Submitting Applications Using the Archives Organizer

You should be quite familiar with the Archives Organizer by now. Let's wring the last bit of functionality out of it by using the Submit feature to complete your submission to the App Store:

1. You're using the same archived application you just validated. In the Archives Organizer, with the archive selected, click the Submit button below the Validate button.
2. You're again prompted to choose your development team. As in the validation step, you probably have just one account to choose from: select it, and click Choose.
3. You see a summary of the application as you did back in Figure 15-36. Click Submit. The archive is validated and then uploaded to Apple, as shown in Figure 15-42.

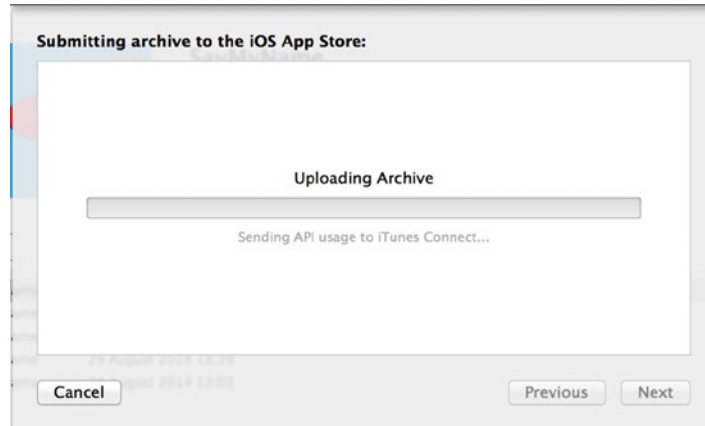


Figure 15-42. The application being uploaded to Apple

Your application will take a while to upload. Then Xcode will present you with the message shown in Figure 15-43 that confirms your submission has been made. In Xcode 5, the moment your application has been uploaded, the Apple review process begins automatically. This is no longer the case, which is helpful, because next you learn the second way Xcode can submit your application to the App Store.

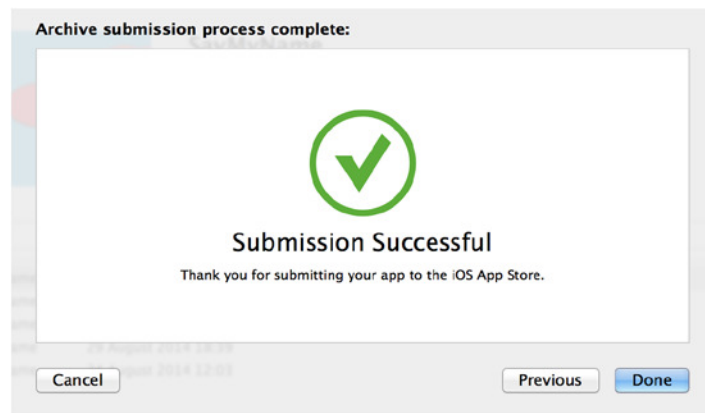


Figure 15-43. Your application has been successfully uploaded to iTunes Connect and the App Store

Submitting Applications Using the Application Loader

Next I'll show you the Application Loader. You may wonder why you need to use it after what should have been a fairly painless upload process using the Archives Organizer. Well, in truth, using the Archives Organizer isn't always a painless process. When I submitted my very first application to the App Store, my elation at finally being ready to submit turned to despair when for some reason the upload process refused to work behind the corporate firewall. Nothing I tried would get around it, which is when I turned to the Application Loader.

The Application Loader is distinctively different visually from other parts of Xcode. Although it can be downloaded separately from the Developer Portal, it's bundled in the Xcode package. You can use it to upload applications to the App Store and also to create in-app purchase packages. Rather than uploading the binary file from an archive, the Application Loader uploads a presigned `.ipa` file to the App Store.

Some organizations have a specific person who is responsible for App Store submissions. The fact that you can use the Application Loader as a stand-alone application makes that process far simpler, because this person only needs to contend with one simple application rather than working through different Xcode screens and project files.

Changing Build Numbers

Before you can use the Application Loader, you need to change the build number and create a signed copy of the application in the `.ipa` file format. You do so using the Archives Organizer with the Export feature.

Let's start by creating a new build number. It's important to know that a build number is not the same as a version number. The SayMyName application is currently on version 1.0 and build 1. If you try to use the Application Loader to upload the binary with the same version and build numbers, you'll receive an error because even if the code has changed, Apple will view these as one and the same and therefore prevent your upload. Changing build numbers is a piece of cake:

1. Open the settings for the SayMyName target, as shown in Figure 15-44.

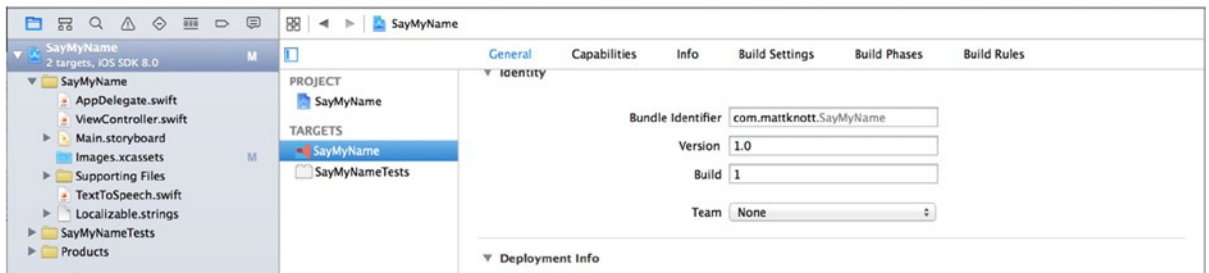


Figure 15-44. Opening the target settings

2. Change the Build value to 2.
3. Create a new archive by selecting Product ► Archive.

Note Remember, if the Archive option is unavailable, the target must be an iOS device that doesn't even need to be connected, rather than a simulator.

Now that you've created an archive with a newer build version, it's time to create the `.ipa` file and upload it to the App Store.

Creating an .ipa File

If you've not come across the extension before, an .ipa file is an application archive: that is, your entire application contained in a single, signed file that can be distributed in a number of ways, including among members of your organization or to the App Store. You can create an .ipa file by using the Export function in the Archives Organizer:

1. Open the Archives Organizer, and select the latest SayMyName archive.
2. Click the Export button. You're presented with three options, as shown in Figure 15-45; choose the first option, Save for iOS App Store Deployment, and click Next.

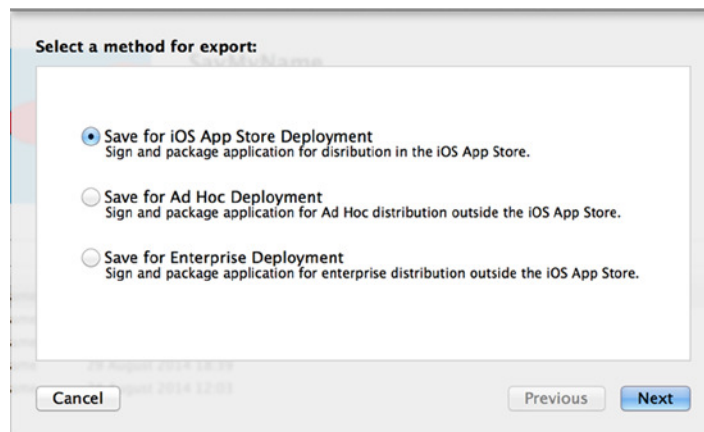


Figure 15-45. Exporting the SayMyName application to an .ipa file

3. As you have done several times, select the development team, and click Export. Once again you see a summary of the application's contents; click Export.
4. You're prompted to save your file somewhere; remember which location you select, and click Save. You have now created the .ipa file, ready for the Application Loader.
5. To launch the Application Loader, go to the menu bar and select Xcode ► Open Developer Tool ► Application Loader.
6. When the Application Loader first loads, you're presented with a terms and conditions page and then asked to log in. Enter your credentials, and click Next.
7. Once your information has been verified, click Done. You see the screen shown in Figure 15-46; this is the Application Loader.

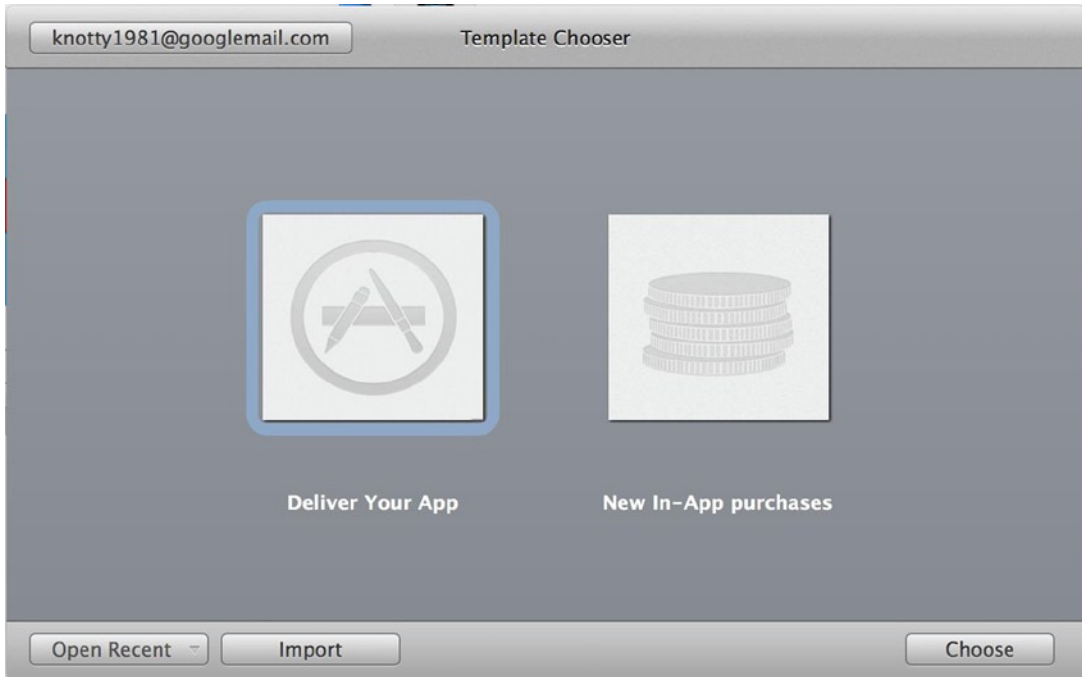


Figure 15-46. The Application Loader main screen

8. Click the Deliver Your App button, and then click Choose. Locate and select your .ipa file, as shown in Figure 15-47.

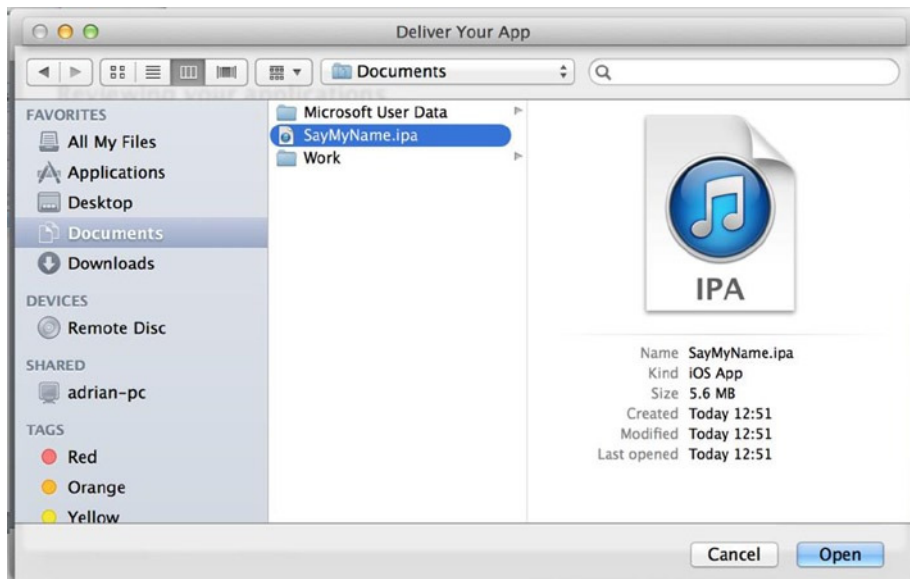


Figure 15-47. Selecting the signed .ipa file to submit it to the App Store

9. After a moment, you're presented with a summary of the Say My Name! application, as shown in Figure 15-48. Click Next.



Figure 15-48. Summary of the Say My Name! app

10. You're shown a screen titled Adding Application. The .ipa file is being uploaded to Apple.
11. When it has finished uploading, you're presented with a thank-you screen; click Done.

You've submitted two builds of your application to Apple; all that remains is to complete the submission process. In the past this would have started the moment you uploaded the binary file, but Apple has changed this process to make it more flexible to users' needs, as described next.

Submitting an Application for Approval

You've written the application and uploaded your build, and it's time to complete the process and submit your application to Apple for approval and addition to the App Store. For this stage, you need to return to iTunes Connect, so open your browser and navigate back to the iTunes Connect web site:

1. On the iTunes Connect web site, click My Apps, and then click the Say My Name! application.
2. Scroll down until you see the Build section, as shown in Figure 15-49.

Build 

Click + to add a build before you submit your app.

Submit your builds using [Xcode 5.1.1](#) or later, or [Application Loader 2.9.1](#) or later.

Figure 15-49. The Build section of the application version information screen

3. Click the + symbol shown next to Build in Figure 15-49. A modal dialog appears, listing the two uploaded builds. Select the newest build, as shown in Figure 15-50, and click Done.

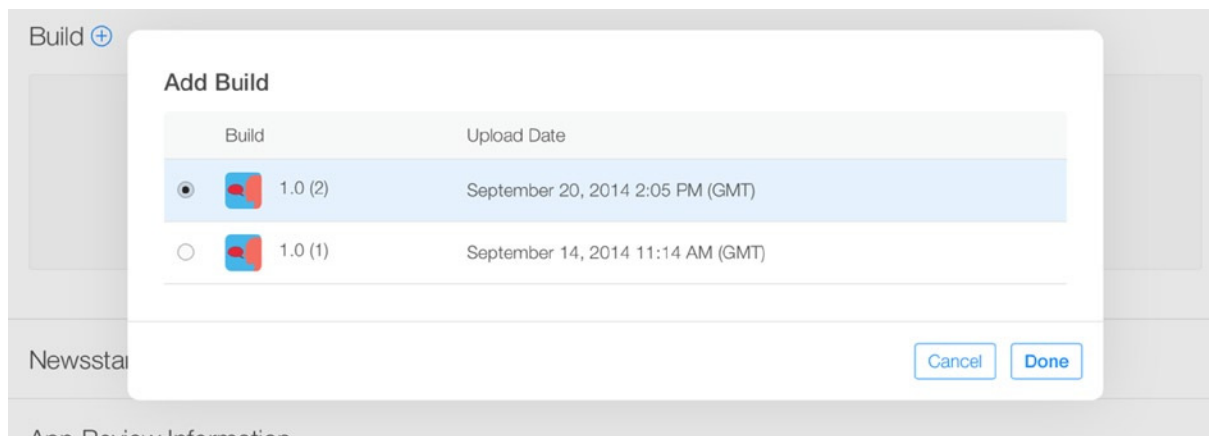


Figure 15-50. Selecting the latest build that has been uploaded to iTunes Connect

4. You're ready to submit your application for review. At the top of the page, click the Save button; this enables the Submit for Review button, as shown in Figure 15-51.

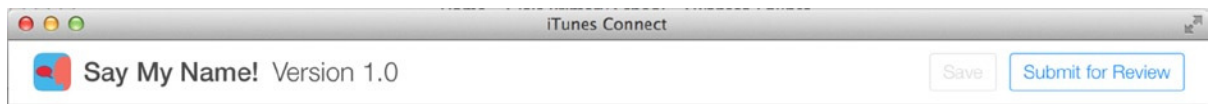


Figure 15-51. The Submit for Review button becomes active after you click Save

5. Click the Submit for Review button. You're presented with several checks to confirm; select No for each of these, as shown in Figure 15-52, and then click Submit.

iTunes Connect My Apps Matthew Knott
Matthew Knott|9228503

Submit for Review

[Cancel](#) [Submit](#)

Export Compliance

Is your app designed to use cryptography or does it contain or incorporate cryptography? (Select Yes even if your app is only utilizing the encryption available in iOS or OS X.) Yes No

Content Rights

Does your app contain, display, or access third-party content? Yes No

Advertising Identifier

Does this app use the Advertising Identifier (IDFA)? Yes No

The [Advertising Identifier \(IDFA\)](#) is a unique ID for each iOS device and is the only way to offer targeted ads. Users can choose to limit ad targeting on their iOS device.

Figure 15-52. Final checks before submitting for review

You return to the application version information screen, as shown in Figure 15-53, but the status has changed to Waiting for Review. Congratulations: you have a wait of roughly two weeks ahead, during which your application will be reviewed by Apple engineers. The first time you submit an application, the wait is horrendous—you will probably check daily for updates even though Apple e-mails you with each status change.

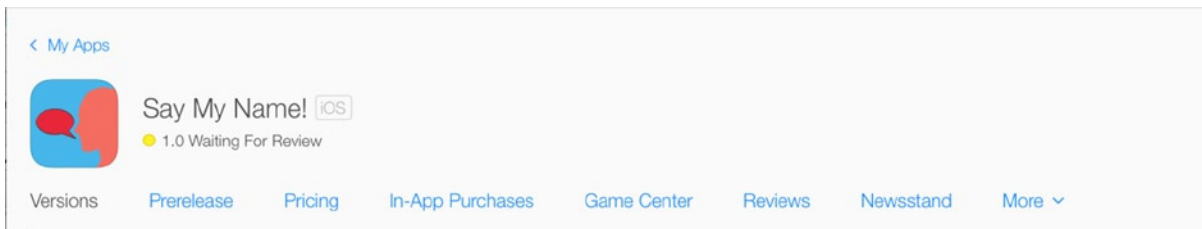


Figure 15-53. The application is now waiting for review

Using the iTunes Connect App

As a footnote to this chapter and indeed the entire book, I want to take a moment to tell you about the iTunes Connect app available for iPhone. Although it has nothing to do with Xcode, as an iOS developer you'll want to check on the profitability and popularity of your apps wherever you are, and the iTunes Connect app is a great way to do this. On your iPhone, open the App Store and search for iTunes Connect, as shown in Figure 15-54.



Figure 15-54. Searching for iTunes Connect in the iOS App Store

Once you've downloaded and installed the application, open it and sign in. Click the Projects tab, and you see your applications listed. Select Say My Name!, and you see that its status is displayed as well as the ability to reject the binary if you realize there is a problem with the build you submitted (see Figure 15-55).

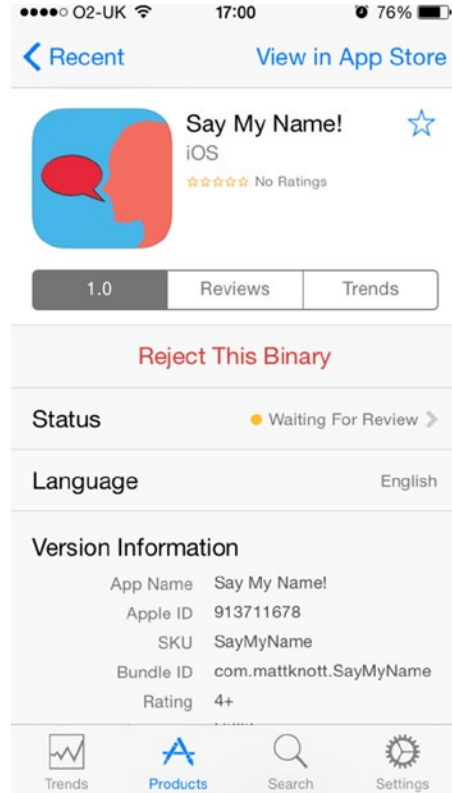


Figure 15-55. Examining the *Say My Name* application using the iTunes Connect application

The iTunes Connect application has a number of the web site's features, including sales reporting, and is certainly something you should be aware of.

Summary

In this chapter, you've added the last feather to your cap, and you can now use Xcode to a high standard. Where you go from here is up to you, but you have all the skills necessary to become a professional iOS developer.

Specifically, in this chapter you've done the following:

- Finished the *SayMyName* application by adding icons
- Configured a loading screen using the `LaunchScreen.xib` file
- Set up an application profile in iTunes Connect
- Created a provisioning profile in the Dev Center

- Learned how to verify your application separately from the upload process
- Uploaded your application two different ways
- Learned how to submit your application for review
- Discovered the iTunes Connect iOS app

Next Steps

You're now armed with an excellent understanding of how to use Xcode, and you've used the new Swift programming language to create a number of really neat applications. It's time to take what you've learned and put it into practice by building your own apps and games.

If you're wondering where to go from here, Apress has published a wide array of titles that can help you get to the next level, whether that's iOS, OS X, or game development. As I mentioned at the start of the book, iTunes U is also a great source of material to help and inspire you. A great deal of help is available online; start small and build up to grander projects, and you'll be more likely to succeed.

My sincerest hope is that you've found this book useful and that it's given you the confidence to try something yourself. If it has, I'd love to hear from you; or, if you have any questions or suggestions about how I might improve a future edition, please send me an e-mail at matthewknott@me.com. I may not be able to reply right away, but I will certainly do my best.

Good luck!

Index

A

- Accounts view controller
 - Accounts scene, 229–230
 - attributes, 230
 - retrieving and displaying Twitter
 - ACAccountStore property, 236
 - accountsWithAccountType method, 238
 - alert view, 239
 - cellForRowAtIndexPath method, 241
 - FeedViewController scene, 244–245
 - indexPath, 243
 - NSIndexPath object, 246
 - numberOfRowsInSection method, 241
 - numberOfSectionsInTableView method, 241
 - prepareForSegue method, 245
 - reloadData method, 239
 - requestAccessToAccountsWithType method, 237
 - security, 242
 - textLabel's Text property, 244
 - twitterAccounts count property, 238
 - Twitter settings, 242
 - UITableView class, 240
 - viewDidLoad method, 237
 - SocialApp project, 228–229
 - social framework, 235
 - static manipulation
 - content set, 234–235
 - dragging, 234
 - dynamic prototype system, 232
 - section attributes, 233
 - style cell, 233–234
- accountsWithAccountType method, 238
- Apple resources
 - forums, 15
 - iOS Dev Center, 11
 - iOS Developer Library, 13
 - mailing lists, 16
 - professional profiles, 12
 - search engines, 16
 - videos, 16
 - Xcode overview, 16
- Apple review process, 483
- Apple's web site, 131
- Application Loader
 - build number changing, 515
 - .ipa file creation, 516
 - presigned .ipa file
 - uploading, 515
- Archives Organizer
 - application file size, 481
 - definition, 456
 - Delete button, 481
 - Edit Scheme dialog, 480
 - Finder window, 481
 - information pane, 481
 - SayMyName application, 478, 480
 - Xcode archive file, 479
- Assistant editor, 55
- Attributes Inspector, 61
- Authentication view
 - design considerations
 - Document Outline button, 146
 - Label and Round Style
 - Text Field, 148
 - label attributes, 149
 - misaligned view, 149–150
 - resizing Password label, 149
 - setting RGB color, 147
 - skeleton of LoginApp, 148–149
 - text field snaps, 147–148
 - View item, 146
 - LoginApp, 144
 - new projects creation, 145

B

Breakpoint Navigator

- conditional breakpoints, 357–358
- creation, 355
- editing, 356–357
- exception, 355
- OpenGL ES error, 355
- properties, 356
- right-click, 356
- symbolic breakpoint, 355
- testing, 355

C

cellForRowAtIndexPath method, 246, 260

Code completion

- application, 141
- Connections Inspector, 139
- delegate function, 134
- Interface Builder, 132, 138
- MFMailComposeViewController, 132, 135
- MFMailComposeViewControllerDelegate, 133
- MFMessageComposeViewController
 - Delegate protocol, 133
- openWebsite action, 135
- SMS, 137
- Source editor, 132
- type methods, 135
- viewDidLoad() line, 134

Code Snippet library, 62

Collection view

- Attributes Inspector, 279
- controller, 275
- displaying items
 - addSubview method, 285
 - cellForItemAtIndexPath method, 284
 - NSCache object, 282
 - NSDictionary object, 284
 - performRequestWithHandler method, 283
 - retrieveUsers method, 282
 - SLRequest object, 283
 - UICollectionViewController class, 281
 - user avatars, 285–286
 - viewDidLoad method, 282

Document Outline, 279

FollowingViewController class, 280–281

NSUserDefaults class

- didSelectRowAtIndexPath method, 272
- indexPath object, 272
- navigation bar, 274
- NSKeyedArchiver class, 273
- objectForKey method, 274
- objects, 271
- performSegueWithIdentifier method, 273
- respondToSelector method, 275
- UITableView method, 272
- viewDidLoad method, 271–272

prototype cell, 278

Size Inspector, 279–280

tab bar controller, 269

Committing changes

- commit message, 393
- files modification, 392
- Git repository, 391
- highlighted changes, 392
- initial commit, 393
- numbering changes, 392

Version editor

- blame view, 396–397
- comparison view, 394–396
- log view, 397

ViewController.m, 392

Connections Inspector, 61

Constraints

- authentication view (see Authentication view)
- Auto Layout, 143–144
- debugging views (see Debugging views)
- layout preview, 161
- size classes
 - any width and any height, 163
 - Assistant editor, 165
 - centering four elements, 168
 - customizing Installed attribute, 166
 - disabling, 169
 - disabling selected constraints, 166–167
 - Email Address and Password fields, 164
 - labels and text fields width, 168
 - Pin button, 168
 - Preview tool, 163–164, 169
 - regular width and any height, 164
 - selecting iPhone-specific constraints, 165–166
 - selecting labels and text fields, 167

- text fields customization
 - E-mail Addresses field configuration, 171
 - hiding passwords, 170
 - navigation, 173

Custom segue, 208

D, E

Data Universal Numbering System (DUNS), 15

Debugging

- application, setup, 344
- cellForRowAtIndexPath method, 349
- compile-time errors
 - Activity Bar, 363
 - Issue Navigator, 363–364
 - viewDidLoad method, 362
 - Xcode indication, 362–363
- initial view controller, 347
- initStates method, 347
- logical error
 - Breakpoint Navigator (see Breakpoint Navigator)
 - debug area, 352–353
 - Debug Navigator, 351–352
 - numberOfRowsInSection method, 351
- NSMutableArray, 347
- numberOfRowsInSection method, 349
- numberOfSectionsInTableView method, 348
- playground
 - Assistant editor, 373–374
 - default code, 372–373
 - default contents, 372
 - feature, 371
 - for loop, 374
 - Quick Look, 373
 - setting, 371
 - start tapping, 374
 - variables, 373
- runtime errors (see Runtime errors)
- simulator, 349–350
- StatesViewController, 346
- storyboard view, 345
- table view controller, 345–346
- tools
 - location, 366–367
 - maps application, 365–366

Printer Simulator (see Printer Simulator)

ViewController.h file, 345

Debugging views

- accessing View Debugger, 150
- automatically adding constraints
 - adding missing constraints, 159
 - Issues Navigator and Document Outline, 158
 - updating, 160
- elements outside bounds, 151
- manually adding constraints
 - Align menu, 154
 - attributes, 154
 - constraint selection, 153–154
 - contextual dialog, 151–152
 - Email Address text field, 152
 - Log In button, 152
 - Password text field, 152
 - Pin menu, 156
 - text field position, 152
 - vertical and horizontal positioning, 153
 - vertical spacing, 153
 - warnings, 153
- portrait mode, 150
- Debug Navigator, 52
- Devices Organizer
 - deployment
 - application, 468–469
 - Fix Issue, 467
 - listing, 466–467
 - signing identity, 467–468
 - Xcode 5, 466
 - device's console, 469
 - information and control, 469
 - screenshots, 470
 - sidebar, 469
- Documentation Viewer
 - navigator
 - access, 127
 - Add Bookmark option, 127–128
 - Apple's online documentation library, 128
 - method, 126
 - Table of Contents, 128
 - resources and different document, 124
 - SDK references, 124
 - toolbar, 125

F

File Inspector, 57
 File Templates Library, 62
 FirstViewController, 47

G, H

Graphical user interface (GUI) design tool, 24

I, J, K

Identity Inspector, 61
 initState function, 348

Interfaces

Compose view controller
 activity indicator, 221–222
 Add Missing Constraints, 221
 Attributes Inspector, 219
 composed scene, 221
 ComposeViewController.swift, 222
 Group Table View Background Color, 219
 label and button, 220
 text view, 220

elements, 201–202

Feed view controller

Attributes Inspector, 211
 bar button item, 211
 Compose identifier, 212
 connection line, 215
 Document Outline, 212–214
 Feed Scene section, 214
 modal segue, 214
 Object Library, 211

Tweet view controller

Add Missing Constraints, 218
 Attributes Inspector, 216
 button's appearance, 216
 dismissView, 219
 Font attribute, 217
 square image view, 217
 text view, 218

Interface Builder

asset catalog
 Applcon image set, 82
 benefits, 81
 Images.xcassets library, 80
 new image set creation, 80–81

CoreLocation

actionControl outlet, 110
 action sheets, 107–108
 ActionViewController.swift file, 111
 alert tab, 114
 alert views, 107–108
 buttons and iOS 8, 115–117
 changeRed code, 104–105
 changeToggle action, 94–95
 color label, 101
 didFailWithError function, 97
 didReceiveMemoryWarning
 function, 97
 element positioning, 102
 info.plist file, 98–99
 locationManager object, 95–97
 Object Library, 101
 performAction, 111–113
 positional tracking, 89–90
 privacy message, 99
 segmented control, 108
 simulator, 100
 SliderViewController.swift file, 103
 text field, 102
 Title attribute, 109–110
 TrackViewController class file, 92
 TrackViewController.swift file, 93–94
 UISwitch control, 91
 UITextView control, 91
 UITextViewDelegate implementation,
 105–106
 updateColor function, 104
 viewDidLoad function, 104

design area, 84–85
 segue selection dialog, 86–87
 Showcase tabbed application
 ActionViewController, 79
 configuration, 76
 creation, 76
 Git repository, 77
 UINavigationController, 77–78

storyboards, 82–83
 tab bar controller, 85–86
 tab bar icon
 custom view controller class, 89
 implementation, 87
 .xib files, 88

- tabs, 76
 - view controllers, 84
 - Interface builder touch storyboard, 436
 - InTouch application
 - additional documentation, 121
 - Apple's web site, 131
 - code completion (see Code completion)
 - Documentation Viewer (see Documentation Viewer)
 - initial settings, 120
 - Quick Help
 - AppDelegate.swift, 123
 - code editor, 123
 - entity, 122
 - UIResponder class, 122
 - UIWindow class, 123
 - iOS Dev Center, 457
 - iOS simulator, 32
 - Issue Navigator, 52
 - iTunes Connect App, 521
 - iTunes Connect portal
 - accessing, 491
 - App ID creation
 - bundle ID checking, 496
 - bundle ID value setting, 496
 - Explicit App ID, 495
 - ID registration, 495
 - summary page, 496–497
 - Wildcard App ID, 495
 - application profile creation, 493
 - default language, 497
 - Explicit App ID selection, 498
 - General App Information addition, 500
 - metadata addition, 499
 - Release Information addition, 502
 - rights and pricing Information setting, 503
 - Say My Name! application-management page, 498
 - screenshots, 498
 - dashboard, 492
 - distribution provisioning profile
 - App ID selection, 508–509
 - certificate selection, 509
 - confirmation page, 510–511
 - list of, 507
 - naming, 510
 - new profile creation, 507–508
 - iOS Distribution signing identity, 505–506
 - link, 491
 - terms and conditions, 492
- ## L
- Launch screen creation
 - background color changing, 489
 - default LaunchScreen.xib, 487–488
 - final screen, 490
 - icon_alpha.png icon adding, 487
 - image attribute setting, 488
 - image view selection, 489
 - labels deletion, 488
 - vertical constraint setting, 489
 - width and height setting, 488–489
 - Localizable strings, 436
 - application language setting, 451
 - creation, 443
 - localization, 444
 - NSLocalizedString, 445
 - testing
 - simulator, 450
 - Xcode 6, 446
 - Localization
 - application interface
 - bar button items, 420
 - completed view, 422
 - Image View item, 420
 - labels and text fields positioning, 421
 - outlets and actions, 423
 - running app, 422–423
 - view controller's code, 424
 - views layout, 417
 - base internationalization feature, 434
 - default settings, 434
 - definition, 413
 - image localization
 - asset catalogs, 438
 - base language selection, 439
 - Localize button, 438–439
 - logo.png, 440
 - project folder, 441
 - label text, 433
 - language detection, 413
 - localizable.strings
 - application language setting, 451
 - creation, 443

Localization (*cont.*)

- localization, 444
- NSLocalizedString, 445
- testing, 446, 450

- logo image, 433
- project settings, 434
- SayMyName (see SayMyName application)
- SayText method, 433
- Spanish language addition, 435
- storyboards, 437

Log Navigator, 53

M

Map Kit framework

- encapsulation, 288
- MapPin application displays, 295
- MapPin project
 - annotations, 300
 - bookmarking, 294
 - debug console, 311
 - Documentation Viewer, 294
 - info.plist setting, 308
 - initial settings, 289
 - iOS 8 frameworks, 291
 - iPhone-specific target, 306
 - linked frameworks and libraries, 292
 - MapPinSatellite scheme, 311
 - MapPinSatellite target, 307
 - MapPin scheme, 312
 - MapPin target, 306
 - map view outlet, 296
 - MKMapView class, 290
 - MKPointAnnotation class, 302
 - new target, 307
 - Project Navigator, 292, 309
 - super.viewDidLoad(), 310
 - view controller, dragging, 290
 - viewDidLoad method, 300
 - Xcode's Manage Schemes, 310
- Swift compiler, 304

MapPinSatellite, 305

Master Detail View template, 45

Matt's Midnight, 323

Midnight theme, 322

MKCoordinateRegion, 298

MKMapView class, 290

Modal segue, 206–207

Model-View-Controller (MVC), 189

N

NSLocalizedString macro, 414

numberOfRowsInSection method, 241

numberOfSectionsInTableView method, 240–241

O

Object Library, 62

OpenGL ES, 48

Organizer

Archives Organizer

- application file size, 481
- Delete button, 481
- Edit Scheme dialog, 479
- Finder window, 481
- information pane, 481
- SayMyName application, 478, 480
- Xcode archive file, 479

Devices Organizer (see Devices Organizer)

iOS developer program

- account addition, 464
- advertisement, 458
- checkout and payment process, 464
- enrollment process, 459–460, 463
- existing/new account selection, 460–461
- iOS Dev Center, 457
- landing page, 458–459
- license agreement, 463
- Log In link, 457
- purchase summary, 462
- selection, 461–462

Projects Organizer (see Projects Organizer)

Window Organizer, 456

P

Page-based template, 46

PageViewController, 46

performRequestWithHandler method, 255

playPressed method, 389, 399

Popovers segues, 208

prepareForSegue method, 246

Printer Simulator

- Apple web site, 367
- execution, 368

- hardware IO tools, 367
- options, 369
- output, 370
- Photos application, 368–369
- sharing action sheet, 369
- Project Navigator, 23, 52
- Projects Organizer
 - definition, 456
 - key areas, 471
 - snapshots
 - automatic creation, 473
 - deletion, 475
 - exporting, 476
 - facility, 471
 - manual creation, 474
 - restoring, 477
 - reverse chronological order, 475
 - workspace list, 472
- Project templates
 - default location, 44
 - game templates
 - Metal, 48
 - OpenGL ES, 48
 - Scene Kit, 48
 - Sprite Kit, 48
 - Master Detail View template, 45
 - Page-based template, 46
 - selection, 44, 49
 - Single View project template, 46
 - Tabbed template, 47

Q

- Quick Help
 - AppDelegate.swift, 123
 - entity, 122
 - code editor, 123
 - UIResponder class, 122
 - UIWindow class, 123
- Quick Help Inspector, 57

R

- recordPressed method, 388
- reloadData method, 239
- Repository branches
 - merging branches
 - code comparison view, 402
 - enable/disable snapshots, 402

- options, 401
- playPressed method, 400
- target branch, 401
- ViewController.m file, 400
- playPressed action, 399
- removing branch, 403
- SlowDown branch, 398–399
- requestAccessToAccountsWithType
 - method, 237
- Runtime errors
 - call stack, 359
 - debug area, 358
 - exception breakpoint
 - addition, 359
 - Breakpoint Navigator, 359–360
 - execution, 361–362
 - Identifier attribute, 361
 - StateCell, 360
 - table view, 361
 - toggle button, 360
 - NSError, 359

S

- SayMyName application
 - Address Book lookup
 - ABPeoplePickerNavigationController
 - class, 424–425
 - ABPeoplePickerNavigation
 - ControllerDelegate protocol, 425
 - ABRecordCopyValue method, 428
 - ABRecordRef object, 428
 - code completion, 428
 - device contact access, 427
 - didSelectPerson function, 428
 - first and last names retrieval, 428
 - getContact action, 426
 - helper function, 426
 - presentPeoplePicker function, 426
 - selected name addition, 429
 - showPeoplePicker function, 425
 - user authorization, 424
 - user selection handling, 425
 - view controller, 425
 - App Store submission
 - Application Loader (see Application Loader)
 - approval, 518
 - Archives Organizer, 513

- SayMyName application (*cont.*)
 - AVSpeechSynthesizer class, 424
 - icon_120.png, 484
 - icon setting process, 484
 - iTunes Connect portal (see iTunes Connect portal)
 - Landscape options, 415–416
 - launch screen (see Launch screen creation)
 - NSLocalizedString macro, 414
 - resources addition, 416
 - settings, 414–415
 - Source Control option, 415
 - text to speech conversion, 430
 - validation, 511
 - Scene Kit, 48
 - Search Navigator, 52
 - SecondViewController, 47
 - Segues
 - Accounts view controller, 203–204
 - contextual dialog, 204–205
 - custom, 208
 - definition, 202
 - Feed view controller, 203
 - Identity Inspector, 205
 - modal, 206
 - popover presentation, 207
 - Push segue, 206
 - replace style, 208
 - table cell and Feed view controller, 204
 - Short Message Service (SMS), 137
 - Single View project template, 46
 - Size Inspector, 61
 - SlowDown branch, 399
 - Sprite Kit, 48
 - Sprite-Kit based application
 - changing color scheme
 - font configuration, 320
 - Fonts&Colors tab, 319–320
 - font-selection dialog, 320–321
 - new theme creation, 321
 - productivity, 319
 - sharing/importing theme, 322
 - characters, 317
 - code folding
 - class method, 335
 - code compression, 333
 - createBug method, 335
 - “folded” didMoveToView method, 334
 - fold toggle, 333
 - implementation file, 334–335
 - Xcode’s Editor tab, 334
 - Code Snippet library
 - AlienDev application, 341–342
 - dragging, 339–340
 - finished method, 338
 - highlighted code, 339
 - if statement, 337
 - My Code Snippet, 340
 - paradox, creating code, 339
 - preview, 340
 - properties, 341
 - update method, 341
 - utilities bar, 336
 - images asset catalog, 318
 - initial project setup, 316
 - landscape device orientation, 317
 - new project creation, 316
 - organizing and navigating code
 - code reminder creation, 326
 - GameViewController.swift, 324
 - jump bar, 327
 - pragma marks, 328
 - viewDidLoad method, 324–325
 - scene building
 - AlienDev app, 330
 - createBug function, 332
 - didMoveToView method, 331
 - GameScene.swift, 329
 - labelNode object, 331
 - SKLabelNode class, 330
 - SKScene object, 329
 - SKScene update method, 332
 - SKSpriteNode, 332
 - two custom classes, 328
 - supporting files group, 317–318
 - Xcode, 315
- Standard editor, 55
- Storyboards
 - ComposeViewController, 201
 - Identity Inspector, 200
 - interface (see Interfaces)
 - navigation controller, 208
 - segues (see Segues)
 - SocialApp

- account selection, 183
- Compose Tweet, 184
- default view controller, 188
- Git repository, 185
- Identity Inspector, 198
- inheritance, 199
- key screens, 182
- Project Navigator, 188
- project options, 185
- Single View Application template, 184
- size classes, 185
- starting arrow, 186–187
- table view controller, 186
- Tweet Detail scene, 184
- tweet list, 183
- Twitter application, 182
- view controllers creation (see View controllers)
- table view controllers, 200
- TweetViewController, 200
- Walt Disney studios, 182

Swift, 4

Symbol Navigator, 52

T

Tabbed template, 47

Table view

- Accounts view (see Accounts view controller)
- compose view
 - dismissView and postToTwitter methods, 268
 - import statements, 264
 - performRequestWithHandler method, 266–267
 - postContent method, 265
 - selectedAccount object, 265
 - shouldChangeTextInRange method, 268
 - SLRequest object, 266
 - startAnimating message, 266
 - UITableView delegate method, 264

composition, 227

Feed view

- ACAccount object, 255
- cellForRowAtIndexPath method, 257, 260
- custom table cell, 253
- dragging, 250
- image view, 250–251
- NSDictionary objects, 258

- NSOperationQueue object, 259

- processing, 247

- Project Navigator, 248

- reloadData method, 255

- retrieveTweets method, 256

- Size Inspector, 249–250

- storyboard file, 249

- TweetCell.swift, 252

- tweet content label, 252

- tweetUserAvatar, 253

- User Name label, 251

- viewDidLoad method, 254

iOS apps, 226

plain and group styles, 227

Tweet view

- compose view controller, 263–264

- NSDictionary object, 262

- UIViewController, 263

- viewDidLoad method, 263

Test Navigator, 52

U

UINavigationController, 49

UIPageViewControllerDelegate protocol, 46

UITabBarController, 49

V

Version control

- committing Changes (see Committing changes)

- definition, 379

- HearMeNow, 379

- project creation

- adding two buttons, 385

- AVFoundation framework, 383

- Git repository, 380–382

- history, 382–383

- instance variables, 387

- interface completion, 385

- label attributes, 384

- label positioning, 384

- outlets and actions, 386

- physical devices, 390

- playPressed method, 389

- Project Navigator, 391

- recordPressed method, 388

- set up, 381

Version control (*cont.*)

- Source Control menu, 382
- view controller, 385
- viewDidLoad method, 387–388

remote repository

- benefits, 404
 - configuration remotes tab, 406–407
 - GitHub home page, 404
 - GitHub launch page, 405
 - GitHub repository page, 405
 - push and pull command, 408–409
 - Quick Setup box, 406
 - updating, 409–410
 - Xcode preferences, 407–408
- repository branches (see Repository branches)

Version editor, 56

View controllers

- MVC design pattern, 189
- Subclassing UITableViewController
 - AccountsViewController, 196
 - adding file, 196
 - FeedViewController, 196–197
 - Project Navigator, 197
- Subclassing UIViewController
 - adding file, 190
 - Cocoa Touch Class file template, 190–191
 - group and target selection, 192–193, 195
 - Project Navigator, 193–194
 - TweetViewController file, 191–192

viewDidLoad() line, 134

viewDidLoad method, 32, 237, 254, 387–388

W

Window Organizer, 456

X, Y, Z

Xcode

adding files

- asset catalogs, 34
- constraints, 39
- Copy Items, 36
- Document Outline, 40
- HelloWorld, 35
- image view and resizing, 38
- iOS simulator, 35
- product finishing, 41

application creation, 3

- Application Loader, 6
- application testing, 32
- Auto Layout system, 6
- breakpoints, 6
- code-completion feature, 6
- code editor, 6
- content downloading, 7
- developer account, 14
- developer assets, 6
- features, 4
- file structure, 37
- Git, 6
- installation, 8
- interface, 5

Assistant editor, 29

Attributes Inspector, 25

dragging label, 27

label constraints, 28

Object Library, 26

ViewController.swift file, 31

viewDidLoad function, 32

libraries, 6

Mac App Store, 7

multiple languages, 6

OS X/iOS development, 4–5

project creation, 5

bundle identifier, 20

Core Data, 20

devices, 20

initial application creation, 22

Organization Identifier, 19

Organization Name, 19

Product Name, 19

Project Navigator, 23

specification, 20–21

Swift programming language, 20

template screen, 18

project templates, 5

source code download page, 15

storyboards, 6

Swift, 4

table and collection views, 6

Welcome splash screen, 9

Xcode Downloads tab, 121

Xcode's interface

Assistant editor, 55

debugging, 73

- main parts, 51
- navigators, 51
- second view controller
 - finished application, 73
 - MainViewController class, 71
 - MessageViewController class, 70
 - segue linkage, 68–69
- Standard editor, 55
- toolbar, 53
- utilities
 - adding missing constraints, 66
 - Attributes Inspector, 61, 64
 - changing view dimensions, 64
 - Class value, 60
 - Code Snippet library, 62
 - Connections Inspector, 61
 - File Inspector, 57
 - File Templates library, 62
 - Headline text style, 65
 - Identity Inspector, 61
 - Interface Builder, 58, 61
 - linking objects, 67
 - Media Files, 62
 - navigation controller, 62–63
 - new file template selection, 59
 - Object Library, 62
 - Quick Help Inspector, 57
 - running application,
 - simulator, 67–68
 - Size Inspector, 61
 - Subclass value, 60
 - tabs, 57, 61
 - view controller, 58–59
- Version editor, 56

Beginning Xcode

Swift Edition



Matthew Knott

Apress®

Beginning Xcode: Swift Edition

Copyright © 2014 by Matthew Knott

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

ISBN-13 (pbk): 978-1-4842-0539-6

ISBN-13 (electronic): 978-1-4842-0538-9

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director: Welmoed Spahr

Lead Editor: Michelle Lowman

Development Editor: Douglas Pundick

Technical Reviewer: Felipe Laso Marsetti

Editorial Board: Steve Anglin, Gary Cornell, Louise Corrigan, James T. DeWolf, Jonathan Gennick,

Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper,

Douglas Pundick, Ben Renow-Clarke, Gwenan Spearing, Matt Wade, Steve Weiss

Coordinating Editor: Melissa Maldonado

Copy Editor: Tiffany Taylor

Compositor: SPi Global

Indexer: SPi Global

Artist: SPi Global

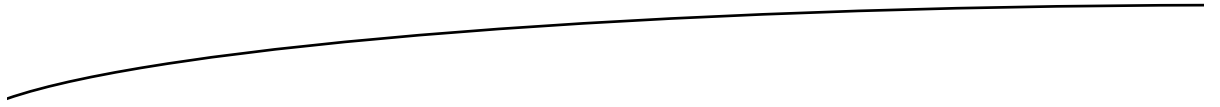
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.



To Mikey. Never stop dreaming.

Contents

About the Author	xv
About the Technical Reviewer	xvii
Acknowledgments	xix
Introduction	xxi
■ Part 1: Getting Acquainted	1
■ Chapter 1: Welcome to Xcode.....	3
What Is Xcode?.....	3
What Is Swift?	4
Why Choose Xcode?	4
Prior Assumptions	4
What's Covered in This Book	5
Part 1: Getting Acquainted	5
Part 2: Diving Deeper	6
Part 3: Final Preparations and Releasing.....	6
Getting and Installing Xcode.....	7
Firing Up Xcode	9
Apple's Resources for Developers.....	10

The Dev Center	13
Your Developer Account	14
Source Code	15
Additional Resources.....	15
Summary.....	16
■ Chapter 2: Diving Right In	17
Creating Your First Xcode Project.....	18
The Project	23
Designing the Interface	25
Making Connections.....	29
Running and Testing.....	32
Adding Files.....	34
Organizing Files in Xcode.....	37
Summary.....	42
■ Chapter 3: Project Templates and Getting Around	43
Project Templates.....	44
Master Detail View.....	45
Page-Based Applications.....	46
Single View Applications.....	46
Tabbed Applications.....	47
Games.....	48
Template Selection.....	49
Getting Around.....	51
Navigators	51
Toolbar.....	53
Editor	54
Utilities.....	57
Configuring the Second View Controller.....	68
Debugging Area	73
Summary.....	74

■ Chapter 4: Building Interfaces	75
Getting Ready	76
Adding Tab Bar Icons to an Asset Catalog	80
Before You Start	82
Building the Interface	84
Setting the Tab Icons	87
Tracking Location with the Track It Tab	89
Mixing Colors with the Slide It Tab	100
Adding “Off the Menu” Controls	107
Changing the Interface with Code	114
Summary	117
■ Chapter 5: Getting Help and Code Completion	119
Getting Help.....	120
Creating the Project.....	120
Downloading Additional Documentation.....	121
Quick Help	121
Documentation Viewer.....	124
Apple’s Web Site.....	131
Code Completion	132
Opening Web Sites in Safari	135
Sending an E-mail with MFMailComposeViewController	135
Sending a Text Message.....	137
Building the Interface	138
Running the Application.....	141
Summary.....	142
■ Chapter 6: Constraints	143
Understanding Auto Layout	143
Building an Authentication View	144
Design Considerations.....	146

Debugging Views in Xcode	150
Manually Adding Constraints	151
Automatically Adding Constraints.....	158
Previewing Your Layout	161
Size Classes	163
Finishing Touches.....	169
Customizing Text Fields	170
Summary.....	177
■ Part 2: Diving Deeper.....	179
■ Chapter 7: Storyboards.....	181
A Brief History of Storyboards.....	182
Creating a New Project Called SocialApp.....	184
Creating View Controllers	189
Pairing the View to the Controller	198
Building Up the Storyboard	200
Linking Scenes and Building Interfaces	202
What Are Segues?	202
Adding a Navigation Controller	208
Creating an Interface for the Feed View Controller.....	210
Creating an Interface for Tweet View Controller	216
Creating an Interface for the Compose View Controller.....	219
Summary.....	223
■ Chapter 8: Table and Collection Views	225
What Is a Table View?.....	226
Table View Composition.....	227
Table View Styles.....	227
Configuring the Accounts View.....	228
The Key Attributes of Table Views.....	230
Manipulating Static Table Views	232

The Accounts and Social Framework	235
Retrieving and Displaying Twitter Accounts	236
Configuring the Feed View	247
Configuring the Tweet View	262
Configuring the Compose View	264
Discovering the Collection View	268
Embedding a Tab Bar Controller	269
Persisting User Preferences with UserDefaults	271
Adding a Collection View Controller	275
Configuring a Collection View	278
Displaying Items in a Collection View	281
Summary	286
■ Chapter 9: Frameworks, Libraries, and Targets	287
Understanding Frameworks	288
Creating the Project	289
Static Libraries, Frameworks, and Swift	304
Working with Multiple Targets	305
Summary	313
■ Chapter 10: Advanced Editing	315
Getting Started	315
Efficient Editing	318
Changing Color Schemes	319
Organizing and Navigating Code	324
Building the Scene	328
Folding Code	333
The Code Snippet Library	336
Summary	342

■ Chapter 11: Debugging and Analysis	343
Building the Application	344
Using Breakpoints to Resolve Logical Errors.....	350
Setting a Breakpoint.....	351
The Debug Navigator	352
The Debug Area	353
The Breakpoint Navigator	355
Runtime Errors	358
Using Exception Breakpoints.....	359
Compile-Time Errors	362
The Issue Navigator.....	363
Tools to Help with Debugging.....	364
Debugging a Location.....	366
Print Debugging with the Printer Simulator.....	367
Playground	371
Summary.....	375
■ Part 3: Final Preparations and Releasing	377
■ Chapter 12: Version Control with Git	379
Why Use Version Control?	379
What Is Git?	380
Creating the Project.....	381
The AVFoundation Framework.....	383
Creating the Interface.....	384
Committing Changes	391
Examining Changes with the Version Editor	393
Branching in a Repository	398
Merging Branches	400
Removing a Branch	403

Using a Remote Repository	404
Registering for GitHub and Creating a Repository	404
Adding a GitHub Repository to Xcode	406
Pushing to a Remote Repository	408
Updating the Remote Repository	409
Summary	411
Chapter 13: Localization	413
Creating the SayMyName Application	414
Adding the Resources	416
Creating the Application Interface	417
Laying Out the Views	417
Building the Interface	420
Writing the Code for the SayMyName Application	424
Retrieving a Contact from the Address Book	424
Converting Text to Speech	430
Localizing the Application	433
Enabling Localization	433
Adding Another Language	435
Storyboards and Localization	437
Localizing Images	438
Localizing Code with Localizable.strings	442
Testing Localizations	446
Testing Localization with Xcode 6	446
Testing Localization in the Simulator	450
Setting the Application Language in the Scheme	451
Summary	453
Chapter 14: Devices and the Organizer	455
The Role of the Organizer in Xcode 6	455
Preparing Xcode for Deploying to a Device	457
Enrolling in the iOS Developer Program	457
Adding Your Developer Account to Xcode	464

Preparing a Device for Deployment.....	466
Managing Devices in Xcode	468
Capturing a Screenshot from a Running Application.....	470
The Projects Organizer	470
Using the Project and Workspace List	472
Using Snapshots.....	472
The Archives Organizer	478
Archiving Your Application	479
Summary.....	481
■ Chapter 15: Building, Sharing, and Distributing Applications	483
Final Checks Before Publishing Your Application	483
Building a Launch Screen	486
Discovering iTunes Connect	490
Creating an App ID	494
Creating an Application Profile	497
Uploading an Application to iTunes Connect	505
Creating a Distribution Certificate and Profile	505
Validating Your Application	511
Submitting Your Application to the App Store.....	513
Submitting Applications Using the Archives Organizer.....	513
Submitting Applications Using the Application Loader.....	514
Changing Build Numbers.....	515
Creating an .ipa File.....	516
Submitting an Application for Approval	518
Using the iTunes Connect App.....	521
Summary.....	522
Next Steps.....	523
Index.....	525

About the Author



Matthew Knott has been writing code for as long as he can remember, from marveling at moving pixels on a BBC Micro to writing ridiculous text adventures for his mother on an overheating ZX Spectrum 48K. Matthew has been a professional software developer for the past 12 years, 6 of which have been spent in the education sector, where he now leads two teams as a solutions architect. Matthew's work and hobby are basically the same things, but when he's not working, he is loving life in a beautiful part of Wales with his wife, Lisa, and two kids, Mikey and Charlotte.

About the Technical Reviewer



Felipe Laso Marsetti is a senior systems engineer working at Lextech Global Services. In his spare time, Felipe enjoys learning new languages and frameworks, playing violin or guitar, cooking, and video games. You can follow him on Twitter as [@iFeliLM](https://twitter.com/iFeliLM) or his blog at <http://iFeli.me>.

Acknowledgments

I wasn't sure if I was going to write a second book—the physical demands of having a full-time job while trying to write a book like *Beginning Xcode* should not be underestimated—but I'm glad I did. Once again, without the patience and support of my family, this book wouldn't have been possible.

My thanks go to everyone at Apress who helped me get the book finished, including Melissa Maldonado, Douglas Pundick, and Tiffany Taylor. Special thanks to Michelle Lowman at Apress for working with me to make *Beginning Xcode: Swift Edition* a reality. Thanks also to Felipe Laso Marsetti for the constructive feedback, and good luck with your own title.

Finally, thanks to everyone who bought the previous edition. Your many positive messages are what encouraged me to write this book.