iOS 7 IN ACTION

Brendan G. Lim
Martin Conte Mac Donell

MANNING

iOS 7 in Action

BRENDAN G. LIM
MARTIN CONTE MAC DONELL

MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964

Development editor: Jennifer Stout
Copyeditor: Linda Recktenwald
Proofreader: Alyson Brener
Typesetter: Dennis Dalinnik
Cover designer: Marija Tudor

ISBN: 9781617291425
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – MAL – 19 18 17 16 15 14

brief contents

contents

preface

We wrote this book as a guide that you can count on and refer to as you develop your own apps for iOS using the iOS 7 SDK. We tried to cover topics in a simple and immersive way—a way that allows you to learn by getting your hands dirty. It's always easier to learn something new by *doing*, and that's exactly what you'll find in this book, and that's what defines books in the *In Action* series. The book will allow you to learn at your own pace by building real-world applications for each of the topics covered in each of the chapters.

We assume that you're already motivated to write your own iOS apps and want to get started right away, so we won't spend much time convincing you. If you've never created an app before, rest assured that you will have created your very first one after the first chapter. This book will act as your trusted guide whether you want to dive into iOS development, or only want to learn how to use the new features available in iOS.

You'll learn what makes up an iOS application and thus gain a deep understanding of its different components. These many components have to come together to make an app truly great. As you go along, the topics you'll learn will give you the knowledge you need to build more impressive apps on your own. And then we will have succeeded in what we set out to do!

acknowledgments

Many people helped bring this book to fruition—mentors, colleagues, reviewers, editors, friends, and family. We thank you all.

The reviewers who read the manuscript in various stages of its development and provided invaluable feedback: Albert Choy, Andreas Walsh, Brent Stains, Chris Catalfo, Daniel Zajork, David Cabrero, Ecil Teodor, Gavin Whyte, John D. Lewis, Jonathan Twaddell, Mayur Patil, Moses Yeung, Richard Lebel, Stephen Wakely, Steve Tibbett, Yousef Ourabi, and Zorodzayi Mukuya.

The readers of Manning's Early Access Program (MEAP) for their comments and their corrections to our chapters as they were being written. You helped make this a better book.

Our technical proofreader, Joe Smith, who reviewed the manuscript one last time shortly before it went into production.

Finally, the team at Manning who worked with us and supported us, and allowed one of us (Brendan) to do this for a second time: Marjan Bace, Scott Meyers, Jennifer Stout, Kevin Sullivan, Linda Recktenwald, Alyson Brener, and the many others who helped along the way.

BRENDAN LIM

I'd like to dedicate this book to my extremely loving and supportive wife, Edelweiss. Knowing what the experience would be like from the first book I wrote, she still had the patience to encourage me to finish my second. To my father, Chhorn, who has always pushed me to work hard and has been the best role model anybody could ask

for: I can only hope to have a few of the many accomplishments you have achieved. To my mother, Brenda, who is the nicest and most caring and loving person I'll ever know: I strive to be as loving and caring as you are, and to carry myself with the same smile that you always have on your face. Without the two of you, I wouldn't be in this world, and I owe everything to you both. To my two brothers, Chhorn and Chhun, who have always been so supportive of me. To my niece, Madelyn and my nephew, Bryent and to the other members of my family: Edwin, Leticia, Mark, Beth, and Lisa. To all of my friends who have contributed directly and indirectly to the book.

MARTIN CONTE MAC DONELL

The following (and not limited to this book) is dedicated to the memory of my little mentor, the one who taught me how to fight the unbearable and taught me The Meaning. To you and your life: you're still teaching me how to be a better man. Without a word. As it should be. I'd also like to thank Victoria, who opened the gate to the garden and whom I admire and love profoundly. To my dear father, Juan José, my lovely mother, Maria Teresa, my wonderful sister, Lucia, and to my dearest friend, Ezequiel. These four incredible human beings have shaped me to be who I am today: thank you very much.

about this book

If you're interested in developing apps for iOS, then this book is for you.

There are a few prerequisites to be able to use the book effectively. First, you need to be interested in developing apps for iOS. You should have a Mac or at least a computer that's running OS X. Also, although object-oriented methodologies and Objective-C are covered in the appendix, it's helpful to have an understanding of both.

With the prerequisites out of the way, this book is beneficial for developers new to iOS or those who are experienced iOS developers who want to learn more about creating apps for iOS. The book is structured so that you can skip a chapter if you already have a good understanding of the topic. Most of the chapters and the apps we create in them are atomic to allow you to read just the ones you need if you're already experienced.

Roadmap

This book has 14 chapters and is divided into 3 parts.

Chapter 1 gets your development environment up and running, teaches you about iOS fundamentals, and lets you build your first application.

Chapter 2 gives you an in-depth look at views, controls, and the view coordinate system. You also take a look at view controllers and how to support multiple orientations. This is done while enhancing the application that you built in the first chapter.

Chapter 3 teaches you how to use storyboarding to organize the view controllers in your application. We'll use different scenes and show you how to transition and pass data between them by creating a task management app.

Chapter 4 introduces you to table views, table view controllers, and prototype cells so that you can organize and present data as lists. You'll use a table view of albums in the Photos application.

Chapter 5 looks at collection views and custom collection view cells. You'll also use custom collection view flow layouts to organize photos in an application you create to display your photos.

Chapter 6 goes into retrieving remote data using iOS and custom third-party libraries. You'll learn how to use web views to display web pages within an application.

Chapter 7 takes an in-depth look at the Assets Library framework, which allows you to access all of the media on your device. You'll learn how to retrieve assets, display them, and capture photos and videos with the image picker. By the end of the chapter you'll have an application that can display the metadata for a photo.

Chapter 8 introduces you to the Accounts and Social frameworks by creating an application for access to Twitter and Facebook feeds.

Chapter 9 explores advanced view customization by going beyond Interface Builder. You'll learn how to create custom views and animations by creating your own animated clock application.

Chapter 10 gives you an introduction to Core Location and MapKit. Using these two frameworks, you'll learn how to retrieve your current location and heading and how to geocode location data. By the end of the chapter you'll build an app that shows your current speed and location.

Chapter 11 looks at persistence and object management by utilizing Core Data. You'll find out the differences between Core Data and traditional databases and use this knowledge to build a Core Data–backed task management application.

Chapter 12 teaches you how to use AirPlay for streaming media and to display content on external screens. You'll learn how to create your own music application that streams and displays song information through an Apple TV.

Chapter 13 explores how to notify users of your app by sending them push notifications. This chapter goes in depth on how to configure your app to send and receive remote push notifications and how to schedule local notifications.

Chapter 14 explores adding the parallax effect and realistic animations such as gravity, bouncing, elasticity, and friction to views in your applications. You'll see how easy it is to add these effects using iOS 7's APIs for motion and UIKit Dynamics.

Code conventions and downloads

There are many code examples throughout this book. These examples always appear in a `fixed-width code font like this`. If we want you to pay special attention to a part of an example, it appears in a **`bolded code font`**. Any class name or method within the normal text of the book appears in code font as well.

Some of the lines of code are long and break due to the limitations of the printed page. Because of this, line-continuation markers (➥) may be included in code listings

when necessary. Code annotations accompany some of the code listings, highlighting important concepts.

Not all code examples in this book are complete. Often we show only a method or two from a class to focus on a particular topic. Complete source code for the applications found throughout the book can be downloaded from the publisher's website at www.manning.com/iOS7inAction.

An Intel-based Macintosh running OS X 10.7 or higher is required to develop iOS 7 applications. You also need to download the iOS SDK, but this is freely downloadable as soon as you sign up with Apple.

Author Online

Purchase of *iOS 7 in Action* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the authors and from other users. To access the forum and subscribe to it, point your web browser to www.manning.com/iOS7inAction. This page provides information on how to get on the forum once you're registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialog between individual readers and between readers and the authors can take place. It's not a commitment to any specific amount of participation on the part of the authors, whose contribution to the AO remains voluntary (and unpaid). We suggest you try asking the authors some challenging questions lest their interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the cover illustration

The figure on the cover of *iOS 7 in Action* is captioned "Morning Habit of a Russian Lady in 1764." The illustration is taken from Thomas Jefferys's *A Collection of the Dresses of Different Nations, Ancient and Modern* (4 volumes), London, published between 1757 and 1772. The title page states that these are hand-colored copper-plate engravings, heightened with gum arabic. Thomas Jefferys (1719–1771) was called "Geographer to King George III." He was an English cartographer who was the leading map supplier of his day. He engraved and printed maps for government and other official bodies and produced a wide range of commercial maps and atlases, especially of North America. His work as a map maker sparked an interest in local dress customs of the lands he surveyed and mapped; they are brilliantly displayed in this four-volume collection.

Fascination with faraway lands and travel for pleasure were relatively new phenomena in the eighteenth century and collections such as this one were popular, introducing both the tourist as well as the armchair traveler to the inhabitants of other countries. The diversity of the drawings in Jeffreys's volumes speaks vividly of the uniqueness and individuality of the world's nations centuries ago. Dress codes have changed, and the diversity by region and country, so rich at one time, has faded away. It is now often hard to tell the inhabitant of one continent from another. Perhaps, trying to view it optimistically, we have traded a cultural and visual diversity for a more varied personal life—or a more varied and interesting intellectual and technical life.

At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of national costumes two centuries ago, brought back to life by Jeffreys's pictures.

Part 1

Basics and necessities

When the water's cold, it's better to jump in without hesitation. You'll be doing just that as you learn the important principles necessary for iOS development. These are many of the core principles and tools you'll be utilizing when you start creating more advanced applications.

In chapter 1 you'll be introduced to iOS, the development environment, and will even create your own Hello World application called Hello Time.

Chapter 2 takes an in-depth look at the user interface layer of an iOS app. You'll learn about views, controls, and view controllers. Chapter 3 expands on views and view controllers by going into storyboarding and scenes. By using storyboards you'll be able to organize and transition among multiple view controllers in your application.

In chapter 4 you'll tackle the common problem of organizing data into a list. You'll do this by using table views. We'll then segue into chapter 5, where you'll learn how to organize data using collection views.

1 Introduction to iOS development

This chapter covers

- Introduction to iOS development
- Designing applications for the mobile paradigm
- Building and running your first iOS application
- Objective-C and MVC primer
- Overview of Apple's development tools

Developing iOS apps is something that many people wish they knew how to do. How many times have you heard people say, "If only there was an app for..."? By the end of this book you'll be able to create those apps and possibly create one that could be downloaded by millions of people around the world. Even by the end of this chapter, you'll be able to call yourself an iOS developer after we create our first iOS application together.

Many people who want to develop for iOS get scared away by the perceived complexity of the platform. You'll soon learn that once you focus on just the essentials, you won't feel overwhelmed as most people do with other iOS books. It's also crucial to be able to apply what you've learned by using that knowledge to create something tangible. The best way to learn is by doing, and that's just what you're going to do.

Instead of just reading about these topics, you'll be building useable applications so that you can see firsthand how they work and how you can use them in real-world applications. Throughout this book we'll be covering core iOS topics and many of the great new things in iOS like UIKit Dynamics, AirPlay, Social framework, table and collection views, auto layout, animation, Core Data, and much more. By creating focused applications based on each topic, you'll have a better understanding of what you've just learned. Within this chapter is a quick overview of iOS and then you'll quickly jump into making your first iOS app, as shown in figure 1.1.

You'll be creating an app called Hello Time, which is a fully functioning clock application that tells the current time. While creating Hello Time you'll become familiar with the ins and outs of iOS development. You'll then review exactly what you did while creating the app and learn more iOS development fundamentals.

Figure 1.1 Hello Time, a fully functioning clock application that tells the time, which we'll build together by the end of this chapter

1.1 Developing for iOS

iOS 7 is the seventh major release of Apple's iOS Software Development Kit. The SDK provides many frameworks and tools used to create applications for iPhone, iPad, and iPod touch devices that you can release in Apple's App Store. As you go through this book, you'll learn why developing for iOS is different than developing for the web or desktop, and you'll go through the steps of setting up your development environment to create your own iOS apps.

1.1.1 Different kind of design interaction

The iPhone's release brought a new type of device into the mainstream that relied on fingertips for input with capacitive screens. It also allowed us to use natural multitouch gestures with our fingers that mimicked those once only found in the movies. It's this type of interactive design that makes developing for iOS quite different from developing for desktop and web applications. It's also this amazing level of interaction and ease of use that allows toddlers and young children to interact with iOS apps.

On iOS devices, when browsing the web through Safari, you *flick* the screen upward with the tips of your fingers to scroll down. To go to the next photo in the Photos app, you *flick* to the left. When you use the Maps app, you *pinch* the screen to zoom outward. To zoom in, you could *pinch* outward or *double-tap* with one finger. If you want to "click" a button, you *tap* it. Other gestures allow you to interact with apps to reveal

options for a particular item. For example, the Mail app displays a context menu after *swiping* to the left on an email.

App developers also have to take into account that everything needs to be displayed on a small 3.5"–4" device. You're limited with screen real estate, which requires you to present information to your users in a reasonable manner. You also need to take into account expected usage patterns and interactions. Almost everybody who uses apps on their phone uses them for short periods of time. You not only have to limit what's presented on a screen of this size but also limit the number of interactions required to accomplish a particular task. It's difficult to make something simple, but this type of design interaction can make your apps more successful than those of your competitors.

1.1.2 Getting ready to develop for iOS

To develop for iOS you'll need to have an Intel-based Mac running at least Mac OS X v10.8.4 (Mountain Lion). You'll have to install Xcode 5, Apple's integrated development environment (IDE), to create iOS applications. Xcode is available for free, and you can find it by searching for it in the Apple App Store or by going to http://developer.apple.com/xcode/. Once you've downloaded and installed it, you'll be ready to start creating your first application.

1.2 Creating your first iOS application

Ready to create your first iOS app? Instead of a basic Hello World application, you'll create something with more functionality that can serve as the base of a real-world application. You could even submit it to the App Store if you decided to spend a little more time on it. You're going to create an application called Hello Time, which will be a fully functional clock that will show you the current time.

1.2.1 Creating the Hello Time application in Xcode

Before continuing, make sure that Xcode has finished installing. Once it's installed, open it by choosing Applications > Xcode. Then you can start creating a new project by going to the application menu and choosing File > New > Project. You'll then be presented with many different application templates to choose from. Choose Single View Application and click Next, as shown in figure 1.2.

You'll then be prompted to fill out the name of the project, organization, company identifier, and class prefix. The name of the project should be Hello Time. The organization name and company identifier as well as the class prefix are for you to decide. We'll be using the prefix IA throughout the rest of the book to stand for "In Action." This will help you identify your own files that are related to your project, which is important when you import other libraries into your projects. This is shown in figure 1.3.

After clicking Next, you'll be prompted to save the project on your computer. Consider creating a new folder on your computer that holds all of your iOS applications. This will keep your projects organized and make them easy to find in the future. Once

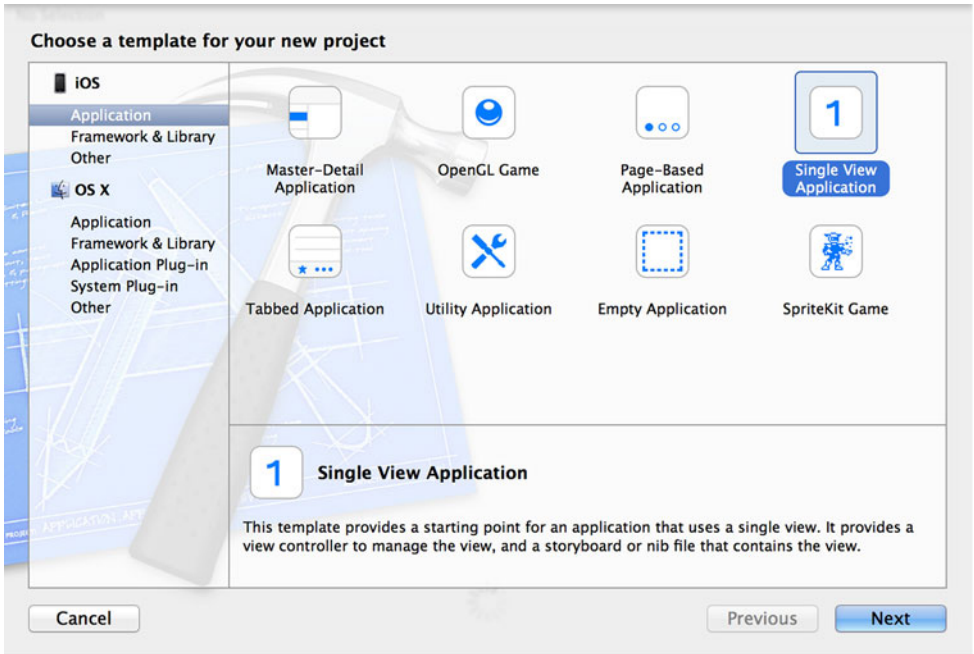


Figure 1.2 Choosing Single View Application as the template for your Hello Time project

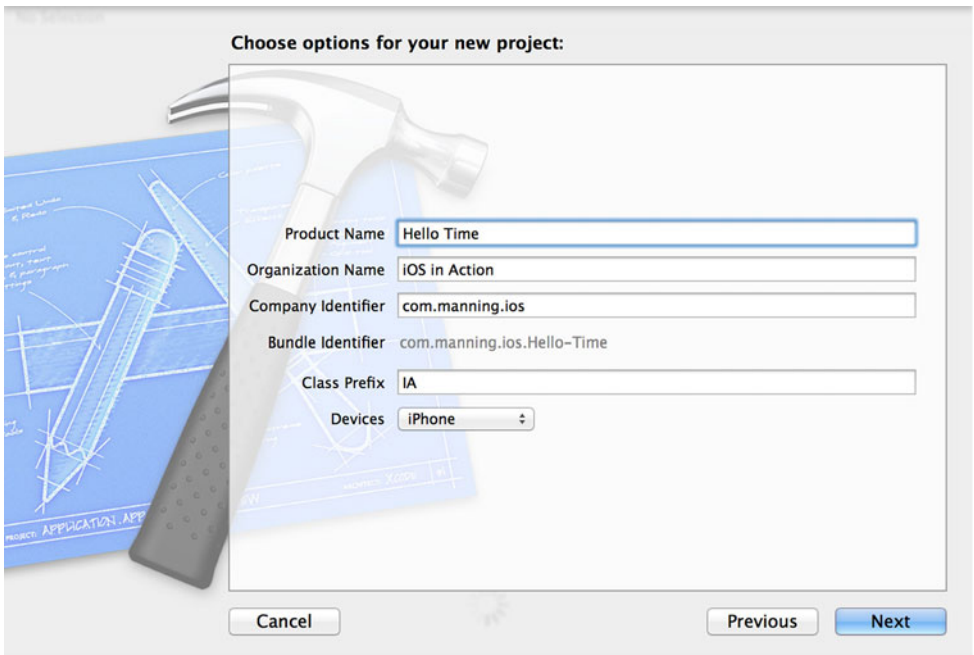


Figure 1.3 Options you need to specify when creating your new project

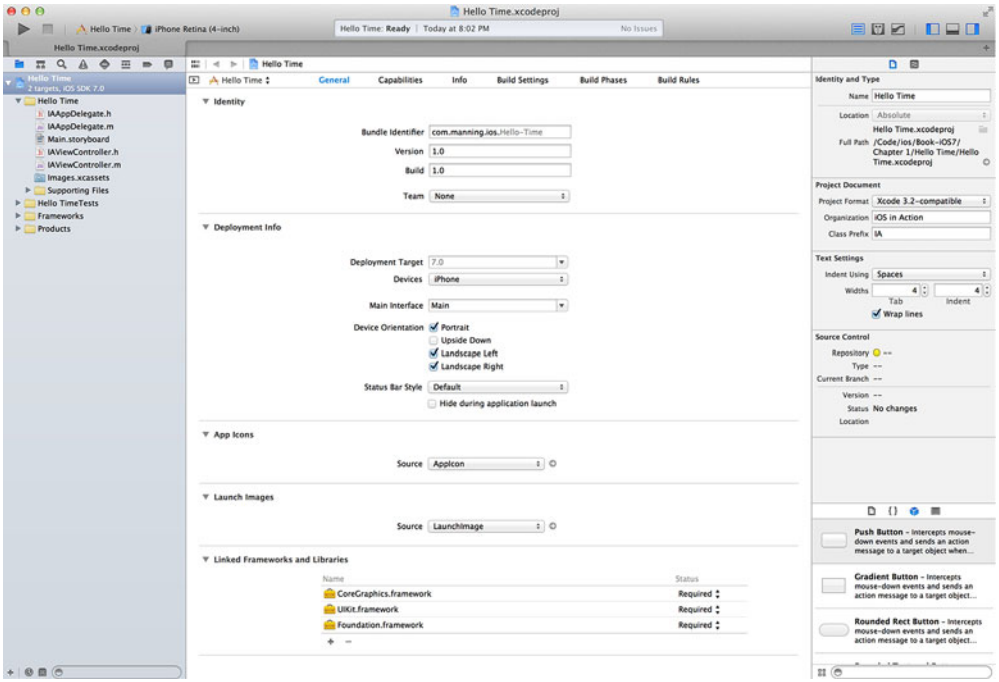


Figure 1.4 Our newly created Hello Time project within Xcode

you've created your project, you'll be taken to the main project window within Xcode. You can see all of the different files that were automatically created for you on the left side of the window in figure 1.4.

Let's get started piecing together the application, beginning with the interface.

1.2.2 Creating the application interface

On the left side of the window, click Main.storyboard to bring up Xcode's interface editor. Your interface will be fairly simple and straightforward because you're showing only one piece of information on the screen, which is the current time. You're going to add a label to the view in the scene that was created for you in your app's storyboard. This label will be used to display the current time.

On the bottom right of the window you'll see the Object Library. To make sure you're able to see this, you can also manually show it by selecting View > Utilities > Show Object Library within the application menu (Control-Option-Command-3). Once you have the Object Library showing, find the `Label` object. You can do this by scrolling down through the list or by searching for it in the search field underneath, as shown in figure 1.5.

Note that you may be in the icon view instead of the list view. You can change this by clicking the icon to the left of the search field. Drag this label into the blank view

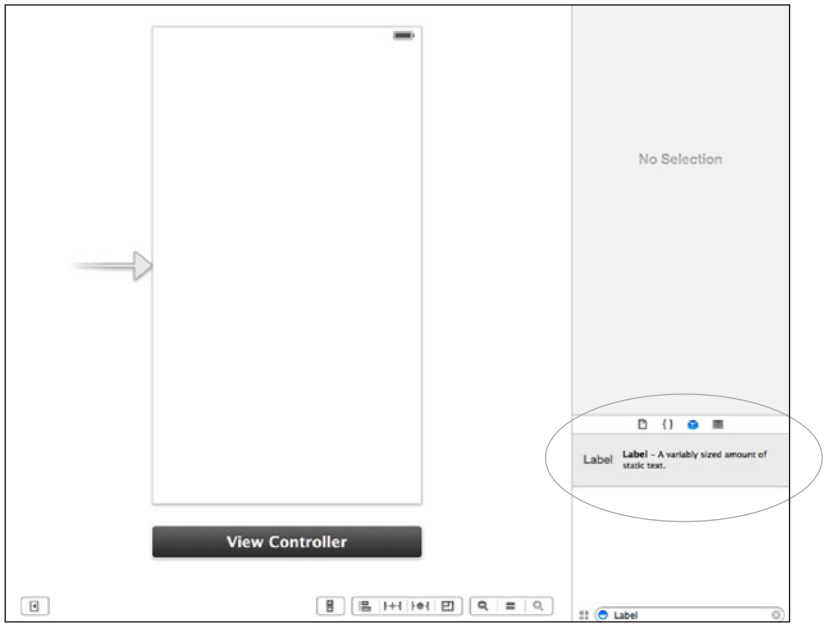


Figure 1.5 Find the `Label` object in the bottom-right corner of the interface editor.

with the white background. Try to arrange it so that it is aligned in the center of the view, as shown in figure 1.6.

You're going to change the appearance of this text to make it look much better than the default label styling. With the label still selected within your view, go to the application menu and choose View > Utilities > Show Attributes Inspector (Option-Command-4). This will load a new window on the right that you can use to edit the object that's currently selected. You can see this in figure 1.7.

Change the font to System Bold with a size of 30.0. Also change the alignment so that the text is centered. After doing this your label might not appear correctly because the width of the label is too small to hold the default text, as shown in figure 1.8.

Change the width of the label by clicking and dragging the two center anchors on the left and right sides of the label. Drag them so that the label's width is the exact same width as the view it's contained within. The height also needs some adjustment. Drag the middle anchor on the top of the label to make it taller. It should look similar to what's shown in figure 1.9.

You're almost finished with your view. The most important aspect is the ability to have your code communicate with different parts of your view.

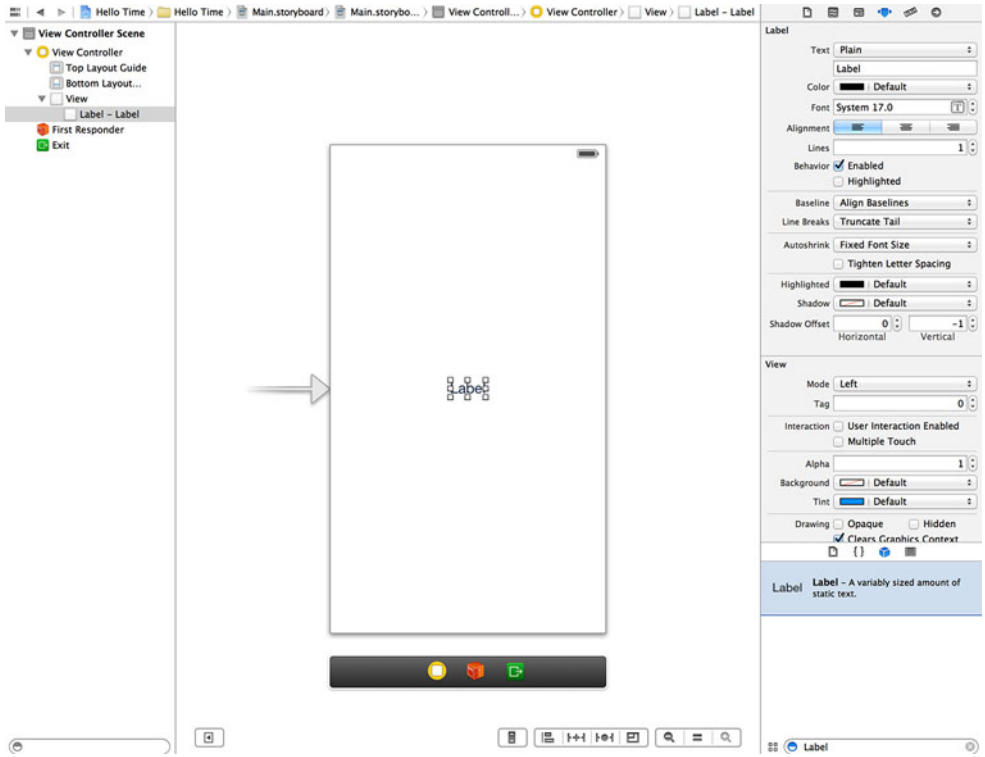


Figure 1.6 Drag the label into the view and align it so that it is centered horizontally and vertically.

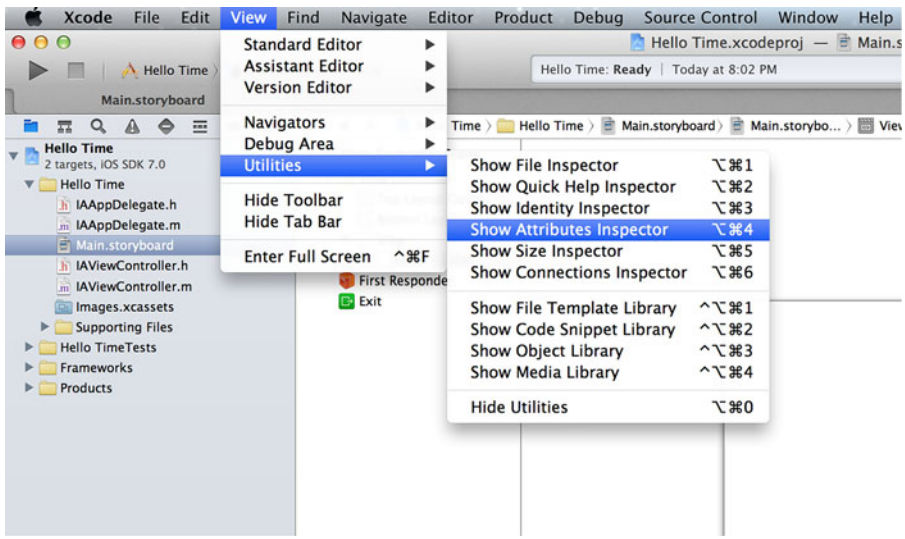


Figure 1.7 The attributes inspector after dragging the label into our view

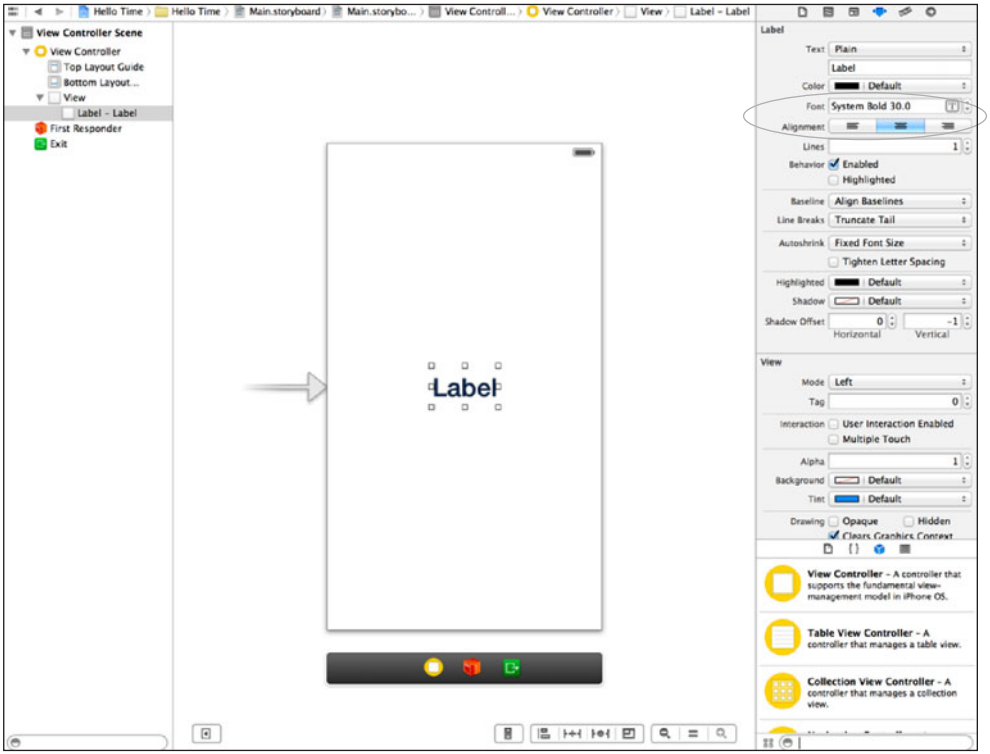


Figure 1.8 Changing the font and alignment of your label within the attributes inspector

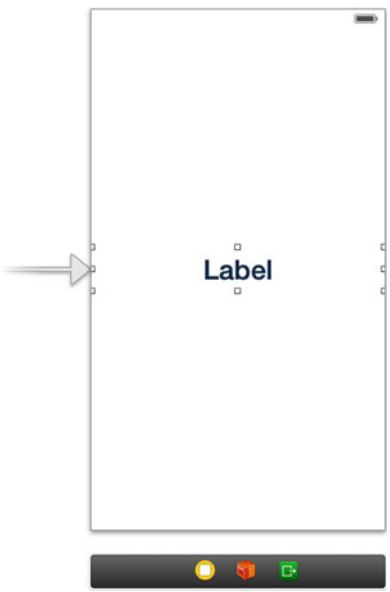


Figure 1.9 Adjust the width and height of the label to ensure that the text fits after changing the font size.

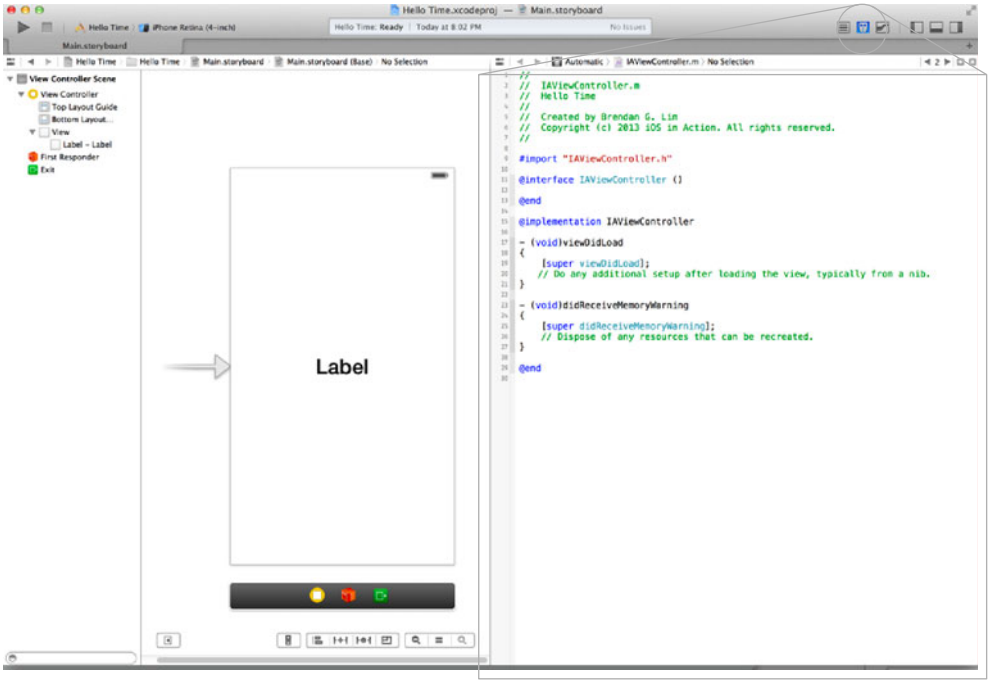


Figure 1.10 Opening the assistant editor while working on the view will bring up the associated view controller.

1.2.3 Connecting your user interface to your code

You're going to create a connection between your interface and your code. This connection will enable your code to communicate with the label you've just created. We'll spend more time on this in the next chapter, but it's important to note that this is something that you'll be using with every application we create together.

Open the assistant editor in Xcode by going to the top right of the window and choosing the middle tab within Editors. You can also open it by choosing View > Assistant Editors > Show Assistant Editor (Option-Command-4). The assistant editor automatically opens IAViewController.m for you because you were working on its view. Change this to show IAViewController.h, as shown in figure 1.10.

You'll now create the connection between your label and your code, which is referred to as an *outlet*. You'll be learning more about outlets and their importance in the next chapter. Create an outlet for your label by holding down Control on your keyboard, clicking your label, and dragging a connection to `IAViewController` in the assistant editor, as shown in figure 1.11.

Once you let go, you'll see a popup that will ask you what you want to name your label. Call it `timeLabel` and click Connect. This will create a property for you within IAViewController.h that you can use within your code to make changes to your label—

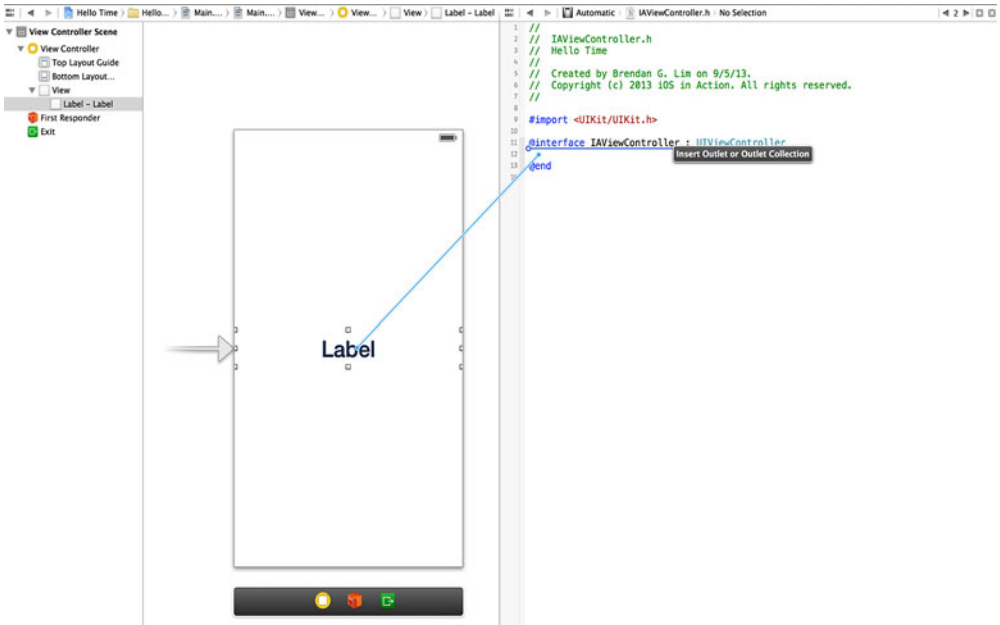


Figure 1.11 Clicking and dragging a connection from our label to our view controller to create an outlet

in this case, to display the current time. The code that will be inserted into your code should look like the following:

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

Now that your view is connected to your view controller, you can start working on implementing the clock functionality.

1.2.4 Implementing the clock functionality

You'll now implement the functionality needed for a basic working clock. Find IAViewController.m to open the implementation of your view controller. Add the following method within Xcode's code editor:

```
- (void)checkTime:(id)sender
{
    NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
    [formatter setDateFormat:@"h:mm:ss a"];
    [self.timeLabel setText:[formatter stringFromDate:[NSDate date]]];

    [self performSelector:@selector(checkTime:) withObject:self
    ➥ afterDelay:1.0];
}
```

This code can be seen added to `IAViewController` in figure 1.12.

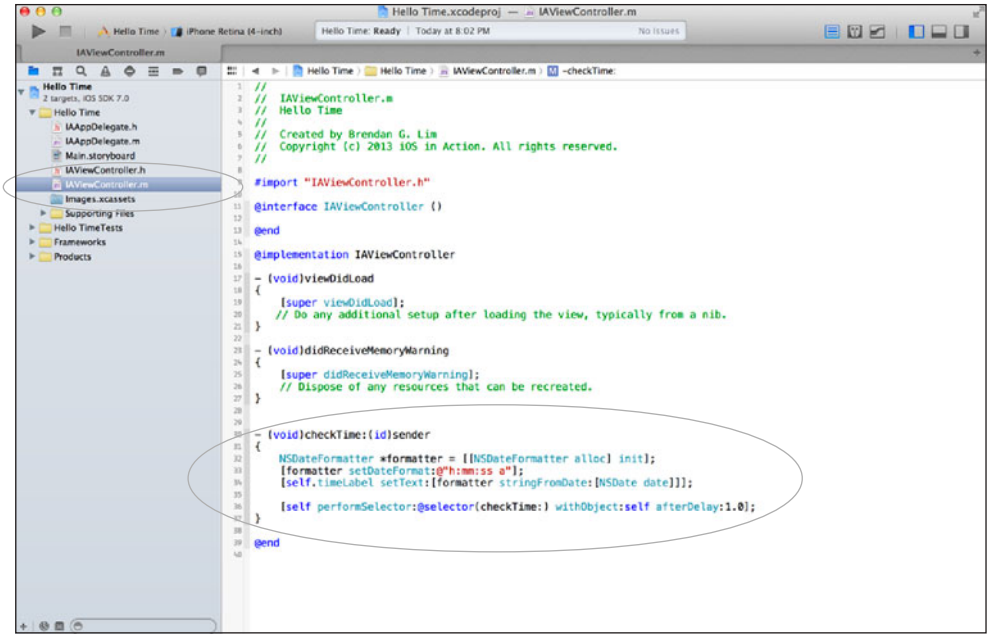


Figure 1.12 The `checkTime:` function added to `IAViewController`'s implementation

This method will get the current date and format it so that you're displaying only the time. It will then change the text of `timeLabel` by calling `setText:`. The last line tells your controller to call this method again every second. This will cause the time to be updated on your label every second, just like a regular clock.

The last thing you need to do is have this method called when your view loads. Luckily there's a method called `viewDidLoad` that was already prepopulated for you when you created your project. This method is triggered after your view has finished loading and is the perfect place for you to trigger the `checkTime:` method. Add the following to the bottom of the `viewDidLoad` method:

```
[self checkTime:self];
```

Adding that line to call the `checkTime:` method will kick-start your clock, which will have it updating the time every second. After doing this, you won't have to do anything else. Why's that? Well, because you've already finished creating your first iOS application! Your interface has been set up and connected to your code. Your code will update the label in your view with the current time every second. You've created a fully functional clock application that tells the time.

1.2.5 Building and running your application

You can finally build and run your application to see just what we've built together. This is extremely simple to do. If you look at the top left of the Xcode window, you

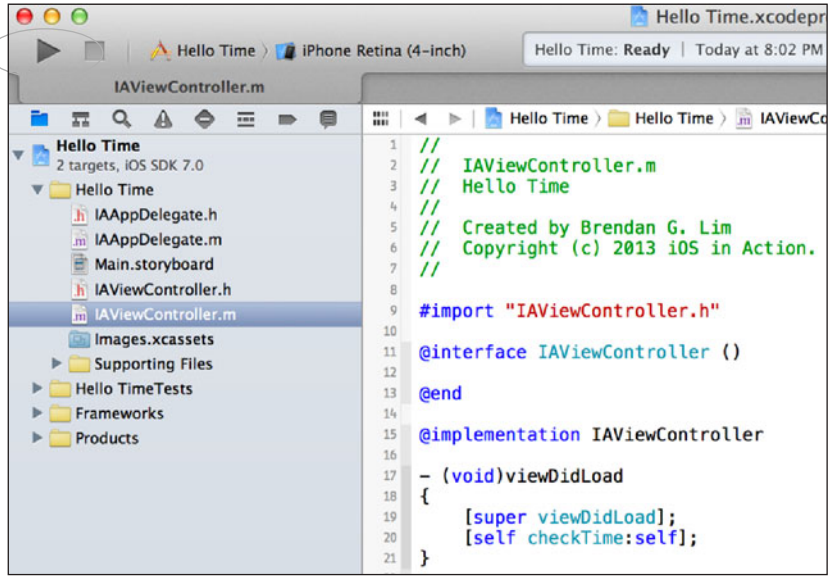


Figure 1.13 Click the Run button to build and launch your application.

should see a button with a "play" icon on it and with the label Run underneath. Once you click it, Xcode will automatically build your application and then launch it within the iOS Simulator. If there is a problem building your application, the compiler will inform you of any errors it encountered.

Click the Run button to build and run your application, as shown in figure 1.13.

Once your application launches, you should see it running in the iOS Simulator. It will show the current time and update itself every second, as shown in figure 1.14.

It's amazing that you've already created your first iOS application. You've created the interface, connected an outlet between your label and your code, and added functionality to set the current time. How about we dig deeper into iOS development by exploring its fundamentals?

1.3 iOS development fundamentals

When creating your application you didn't need to write much Objective-C. Knowing Objective-C is just one piece of iOS development. It's the same as knowing all the words of a particular language. If you know the meaning of all of the words in the English dictionary but don't know how to form sentences, it's like knowing Objective-C but not knowing how to use Apple's APIs. Cocoa Touch is Apple's API for creating iOS apps.

We'll give you a quick introduction to object-oriented programming, Objective-C syntax, MVC, and frameworks. There are whole books written on these topics, so we'll just touch on a few important things that you should be familiar with before moving forward. If you're interested in learning about these items in more detail, please take a look at the appendix.

1.3.1 Object-oriented programming

In a nutshell, object-oriented programming (OOP) is a concept in programming in which objects and their structure are generally more important than the logic needed to manipulate the objects. Before OOP was introduced, programs were seen as recipes or procedures with a set of instructions that you could follow from start to finish. As soon as programs started to become more complex, a new method was needed to reduce this complexity. Object-oriented programming helps solve this complexity by allowing you to break down and flesh out your logic in a more natural way.

Figure 1.14 Our Hello Time application running within the iOS Simulator

When things are broken out into objects, they are easier to understand; we relate to them because they are similar to the world around us. Objects can be named anything to represent something that would hold data within your program. An object could be a person defined by different properties such as name, age, sex, gender, and so on. It could also be a home with properties like address, city, state, price, bedrooms, bathrooms, and the like. Your Hello Time application could be changed to have a clock object that has the time. Instead, your Hello Time app has a `Label` object in its view that has a text property. You modified this text property on this object to display the current time.

Every object in Objective-C is a child, or subclass, of `NSObject`. Objects are defined as classes, and they can have children and parents. Much like in the real world, subclasses, or children, inherit the properties of their parents. For instance, you could have a class called `Automobile` with properties `make` and `model`. You could then create a subclass of `Automobile` called `Truck` that would represent a different type of automobile. You wouldn't need to re-create or redefine the `make` and `model` properties because it would inherit them from its parent. You could, however, add properties that are specific to a truck like the bed size, whether or not it has four-wheel drive, and so on.

The concepts found in object-oriented programming are easily transferred from one language to the next. Even though we haven't gone through many specifics and intricacies of OOP, you should have an understanding of what to expect and enough background to be able to move forward.

1.3.2 Objective-C syntax and message passing

The common response for people new to Objective-C is that the syntax makes the language look daunting and confusing. They are often thrown off by the use of brackets

everywhere. Once you understand why and just how the syntax works, it will all make sense and become easy to read.

Messages are passed to a particular object. Generally, whatever is declared on the left side within a set of brackets is the object, and whatever is on the right side is the message you're passing to it.

```
[object message];
```

The message you're passing to it has to be a predefined method or function that has been defined on that particular object. If one hasn't been defined, you'll get an error telling you that the object doesn't know how to respond.

You can even pass in an argument or a parameter with a message:

```
[object message:parameter];
```

Many people get confused when they see multiple parameters being passed in a method. The previous example has one parameter. Imagine if you had a method definition that looked like the following:

```
-(void) message:(id)parameterOne secondParam:(id)parameterTwo;
```

You would call this method by passing in the parameters as such:

```
[object message:paramOne secondParam:paramTwo];
```

As you can see, the whole name of the method, or the message you're calling, is `message:secondParam:`. The parameters are declared after each colon defined in the method name.

Let's now take a look at creating a new instance of a real object in Objective-C. Here you're creating a new instance of an `NSString` object.

```
[[NSString alloc] init];
```

This is the standard way of creating a new object instance. Don't be confused by the multiple brackets. You can break this down to see how these messages are being passed. You're first calling `[NSString alloc]` in the first set of brackets. Then you're calling `init`. You could break this into two lines:

```
NSString *string = [NSString alloc];
[string init];
```

This is perfectly fine except that it's much easier to do `[[NSString alloc] init]` instead of breaking it into two separate lines. The takeaway here is that whatever is returned within the inner set of brackets is then passed a message by the outer brackets.

From the previous example you can also see that you're returning a reference to a new `NSString` object and storing it as an instance variable called `string`. You did the same thing within the `checkTime:` method that you created in your Hello Time app when you needed to create something to format the time for you.

```
NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
[formatter setDateFormat:@"h:mm:ss a"];
```

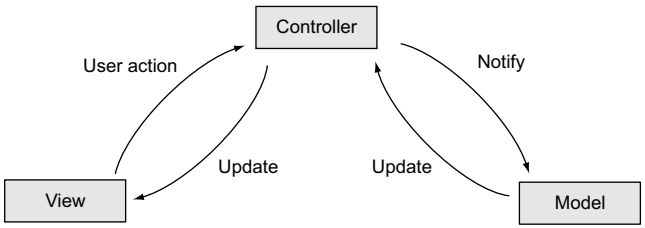


Figure 1.15 Communication between models, views, and controllers within MVC

You created an instance of `NSDateFormatter` and stored a reference to it in the variable named `formatter`. You then passed it a message to set the date format property by passing in a string value as the parameter.

1.3.3 The Model-View-Controller pattern

The Model-View-Controller (MVC) pattern is a design pattern that assigns objects in an application to one of three different roles. These roles are a model, a view, and a controller, as shown in figure 1.15. You may already be familiar with this pattern, because many other frameworks often implement MVC.

Using MVC, models encapsulate data and logic specific to an application. You didn't have any models in your Hello Time application; it wasn't necessary because of its lack of complexity. A view object is something whose main purpose is to visually display information to the user. View objects learn through controllers about the model, which contains the data that they can display. You created your view by modifying the storyboard within the interface editor. Controllers act as the intermediary between view and model objects. You connected your view to your controller, and your controller made the changes to your label to display the current time every second.

1.3.4 Frameworks introduction

Frameworks are compiled libraries that you can use to add functionality to your applications. By default, when you create a new iOS project in Xcode, it automatically includes the frameworks UIKit, Foundation, and CoreGraphics. UIKit provides classes to create and manage user interfaces. Foundation provides the base layer of Objective-C classes. Core Graphics provides functionality based on the Quartz drawing engine. It aids with image manipulation, color management, gradients, shadings, and the like. These three frameworks give you the basic functionality to create iOS applications.

If you wanted to access all of the photos and videos on your iOS device, you could use the Assets Library framework, which contains classes that allow you to do just that. If you were to build a web browser, you'd use the WebKit framework. You'll be using various frameworks to add in functionality to the apps that you build in the book. You've now had a general overview and introduction to OOP, Objective-C syntax, message passing, MVC, and frameworks. Because you'll be spending most of your time in Xcode, let's go over it in more detail so that you become more comfortable with your development environment.

1.4 Overview of Apple's development tools

You'll be spending almost all of your development time within Xcode and the iOS Simulator. If you're not already familiar with these tools, this quick introduction should help you get started. Apple has a few other tools that you can use for development, but these are the two that you'll be using 99% of the time. Let's get started by getting familiar with Xcode and creating our first app together.

1.4.1 Creating different types of projects in Xcode

When we created the Hello Time application, we chose to use a Single View Application project template. There are many other project templates you can choose from when creating a new project in Xcode. We chose the one that best suited our needs because there's only one view in the application.

Take a look at figure 1.16, which shows all of the different application project templates available when you create a new project within Xcode.

Each project template serves a purpose for any applications that you could be building. Table 1.1 shows a run-down of the default templates available for iOS. Note that this differs depending on which version of the iOS SDK you're using.

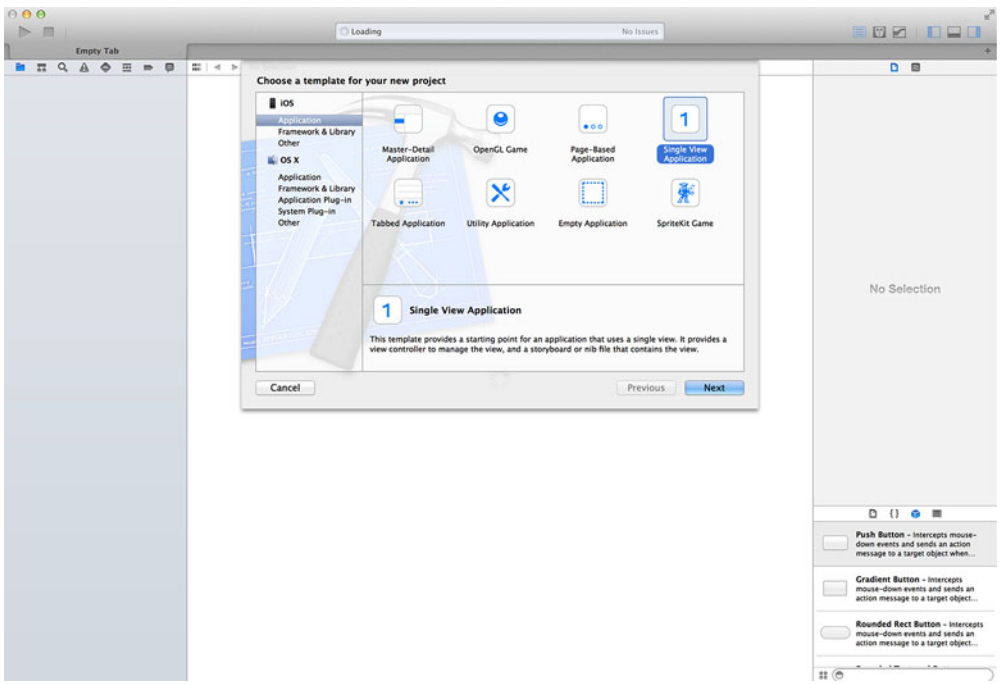


Figure 1.16 Drop-down window shown in Xcode that lets you choose from multiple application templates when creating a new project

Table 1.1 Default iOS application templates for new projects in Xcode

Template	Description
Empty Application	The most basic application template. Provides just an application delegate and a window.
Master-Detail Application	Provides a split-view–based application template for the iPad with a navigation controller.
OpenGL Game	Provides a starting point for an OpenGL ES-based game. Comes with a view to render an OpenGL ES scene and a timer to animate the view.
Page-Based Application	Provides an application setup to use a page-view controller.
Single View Application	Provides a single view and a view controller to manage the view.
Tabbed Application	For applications that need to use a tab bar. Provides a tab bar controller and view controllers for the tab bar items.
Utility Application	Provides a main view and an alternate view. For iPhone an Info button is set up to flip the main view to the alternate view. For iPad it uses a popover to show the alternate view.

If you don't want Xcode to create anything for you beyond a basic application with no pregenerated views, you could choose the Empty Application template.

After choosing and creating the project, you're dropped into Xcode's main workspace window. You'll soon become more familiar with the different sections contained within this window.

1.4.2 Getting familiar with Xcode's workspace

Xcode's workspace window consists of four different sections. These sections include the editor, the navigator, the utilities, and the debug area. In figure 1.17 you can see these sections separated and labeled.

The left side contains the navigator area, which lets you view and access all of the files contained within your project. The editor area is where you'll be diving into code as well as working with visual interfaces for your apps. You can have multiple panes showing different files within the editor area. The bottom center of the screen is the debug area, where you can view log output from your app as well as perform deep application debugging. The right side is the utility area, which is made up of the library and the inspector pane.

When you were creating the interface for the Hello Time app, you clicked Main.storyboard to launch the interface editor. You can see an example of a view being edited inside a storyboard in figure 1.18.

This storyboard handles all of the scenes within your application and how they interact with one another. Xcode's interface editor is a true drag-and-drop interface. You can drag and drop UI elements from the library pane within the utility area directly onto your views. Modifying these UI elements can be done within the inspector

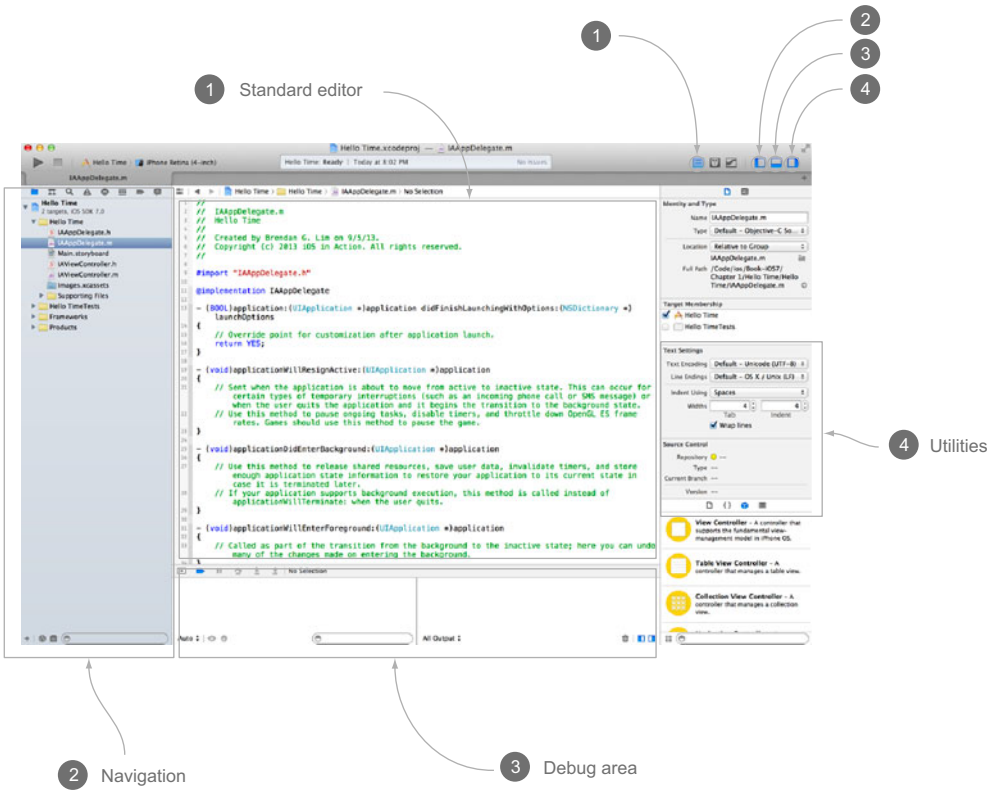


Figure 1.17 The different sections of Xcode's workspace window

pane, which is located within the utility area on the right side of the window. Xcode's interface editor also allows you to edit XIBs (pronounced as "nibs" because the extension used to be .nib instead of .xib). We'll be going deeper into views and storyboards in the chapters to come.

When you ran your application, you used it within the iOS Simulator. You'll be using the Simulator to run and build every application that we build together.

1.4.3 iOS Simulator

Another big part of the iOS SDK is the iOS Simulator. The Simulator allows you to run iOS apps on your computer without the need for a physical device. You have the option of running the Simulator as an iPhone or iPad with or without retina displays. This makes it easy for you to simulate your applications on multiple devices.

The quickest way for you to open the Simulator is to build and run your application. When you built and ran Hello Time, you clicked the Run button on the top left of Xcode's workspace window, as shown in figure 1.19.

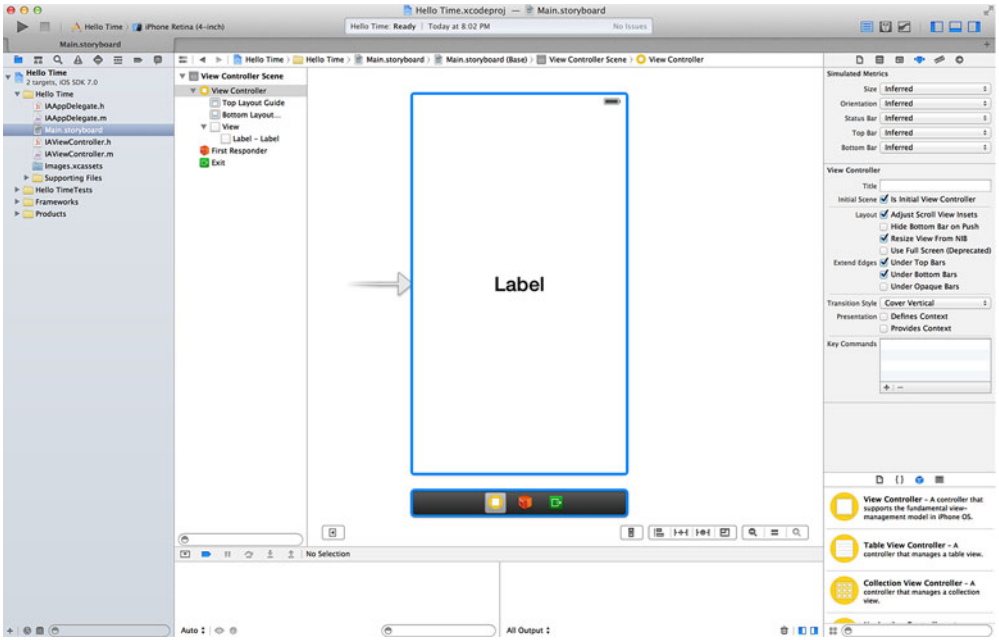


Figure 1.18 Editing a view from a storyboard within Xcode's interface editor

When your application is launched, you are presented with a window that looks exactly like an iPhone. You can click the Home button on the bottom of the Simulator just like on a real device. Once you do this, you'll have access to many preinstalled apps. You can also see that the Simulator has different hardware choices on its application menu, as shown in figure 1.20.

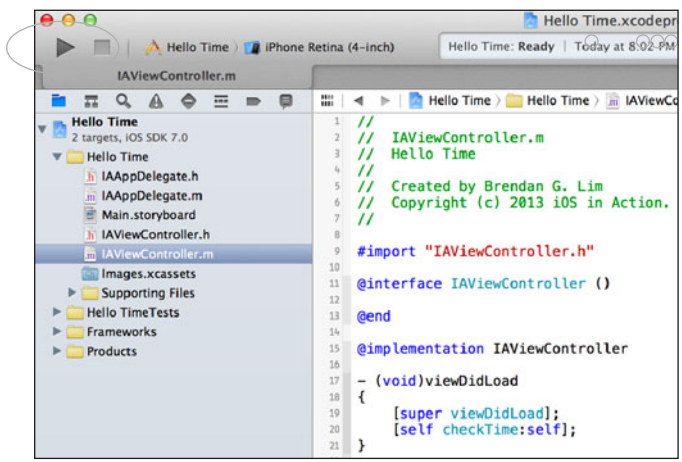


Figure 1.19 Building and launching the application will launch the iOS Simulator.

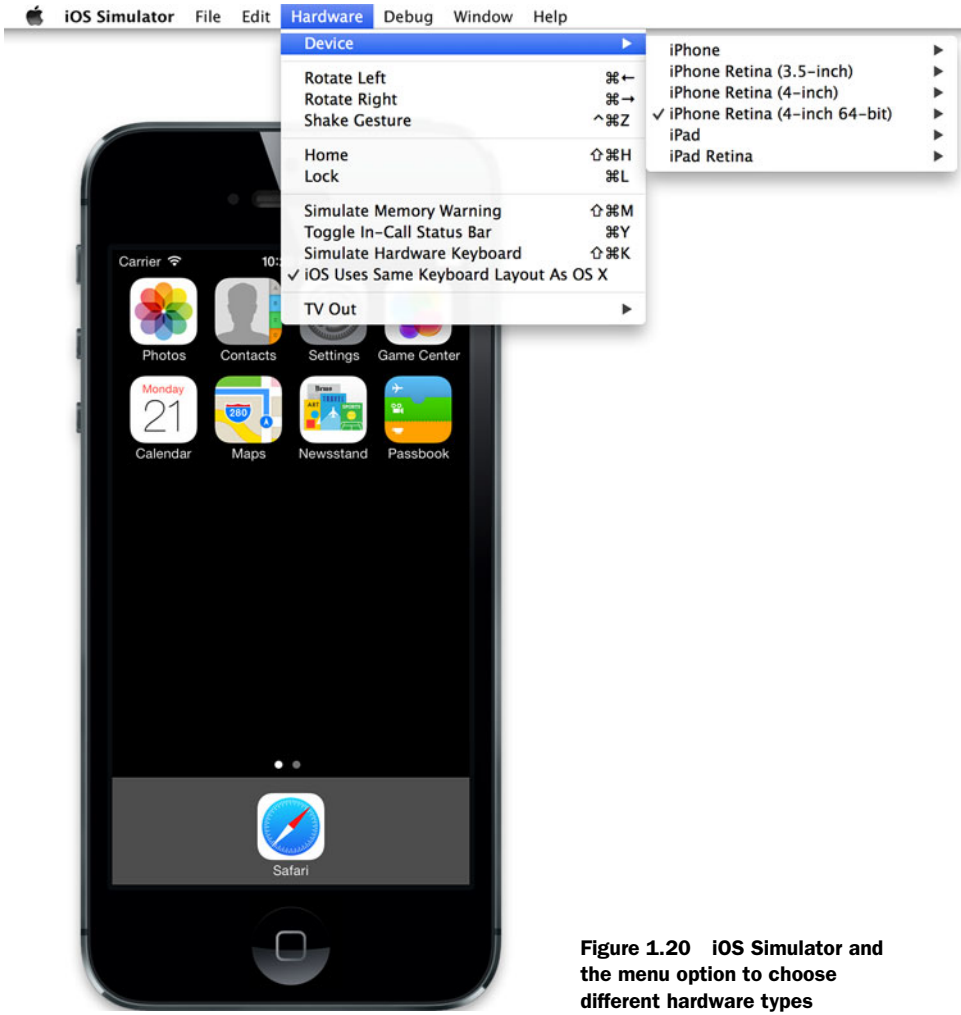


Figure 1.20 iOS Simulator and the menu option to choose different hardware types

These different hardware choices allow you to see how your application would appear if run on different-sized iPads, iPhones, and even retina and non-retina devices.

Instead of having to build and run an application to get the Simulator to run, you can also launch it manually. To do this, you can find the application bundle for the Simulator at the following path in Finder or within the Terminal application.

```
<Xcode Path>/Platforms/iPhoneSimulator.platform/Developer/Applications
```

While using the Simulator, you can mimic different scenarios that would occur on a real physical device. The Simulator can be rotated in any direction; calls can be simulated, as well as your current location. There is also an option to trigger a memory warning to see how your app would respond under such circumstances.

Now that we've taken a look at the tools you'll be using, you should be more comfortable working with them. You'll be using Xcode and the iOS Simulator throughout the rest of the book. As you go along, you'll learn more about Xcode and how you can use it to create complex iOS applications.

1.5 Summary

Within this chapter you've had an overview of iOS and the differences when creating touch-based applications. You also jumped right into creating your first iOS application, Hello Time, a functioning clock app. You also got a primer on Objective-C, message passing, and MVC. Finally, you took a look at Xcode and Apple's development tools. Now you're ready to tackle the rest of the book and jump into the many different parts of iOS development.

- Designing mobile applications requires a shift in thinking compared to other paradigms, such as web or desktop applications.
- Interacting with an application through the use of gestures increases the number of ways someone can interact with your applications.
- Limited real-estate screen space means you should be conscious about how you present information.
- You'll spend all of your application development time within Xcode.
- You can write code as well as create an app's interface within Xcode.
- You can make connections, such as outlets, between your interface and your code.
- Your apps can be run on the iOS Simulator, which comes bundled with Xcode.
- The iOS SDK has many frameworks that can be used to add functionality.
- Xcode has various templates you can use to generate a new project.
- You can create a basic but functional iOS application, such as Hello Time, quickly with a minimal amount of code.

2 Views and view controller basics

This chapter covers

- Enhancing your Hello Time app
- Views and the view coordinate system
- User interface controls and responding to events
- View controller lifecycle and creating views programmatically
- Supporting multiple orientations

You've all used apps on your mobile devices. The way you interact with them is much different than the way you use apps on your laptops or desktop computers. Why is that? The amount of space available to present information on mobile screens is much smaller than on desktop applications. Only a limited amount of data can be displayed on the screen at once to ensure a good visual experience.

You also interact with your apps differently. Instead of using peripherals such as a keyboard and mouse as input devices, you use your fingers. The appearance and design should feel natural to users. You should take all of this into account when creating and designing the views that make up your apps.

In this chapter we'll go over the different parts of what you see in an app—its windows, its views, and the view controllers that manage them. You'll learn about

Figure 2.1 The updated Hello Time application after adding more functionality to it

the different controls you can use within each view of your app, such as buttons, labels, and text fields. You'll then learn how to visually arrange and organize the views within your application, better known as storyboarding. First, we'll revisit the Hello Time app you created in chapter 1 and add more functionality to it.

2.1 Enhancing Hello Time

When you created Hello Time in the previous chapter, the goal was to have a simple application that would act as a clock. In this chapter, you'll be giving it a few enhancements, as shown in figure 2.1.

First, you'll be adding a button to the view that will enable a night mode. This mode should make Hello Time easier on the eyes when you're using it at night. You'll also need a way to switch back to day mode. Next, you'll ensure that Hello Time appears properly when the device it's running on is rotated into landscape mode.

2.1.1 Switching between night and day modes

To begin, launch Xcode and open the Hello Time project. Because you're adding a night mode, you need a way for someone to turn it on. You'll be adding a button below the label that you're using to display the time. In the project navigator (View > Navigators > Show Project Navigator, or press Command-1), choose Main.storyboard to bring up the interface editor. You should see everything just as you left it, as shown in figure 2.2.

Locate the Object Library on the bottom right of your workspace window. Search for "button" to locate Button and drag it underneath the label you've just added, as shown in figure 2.3.

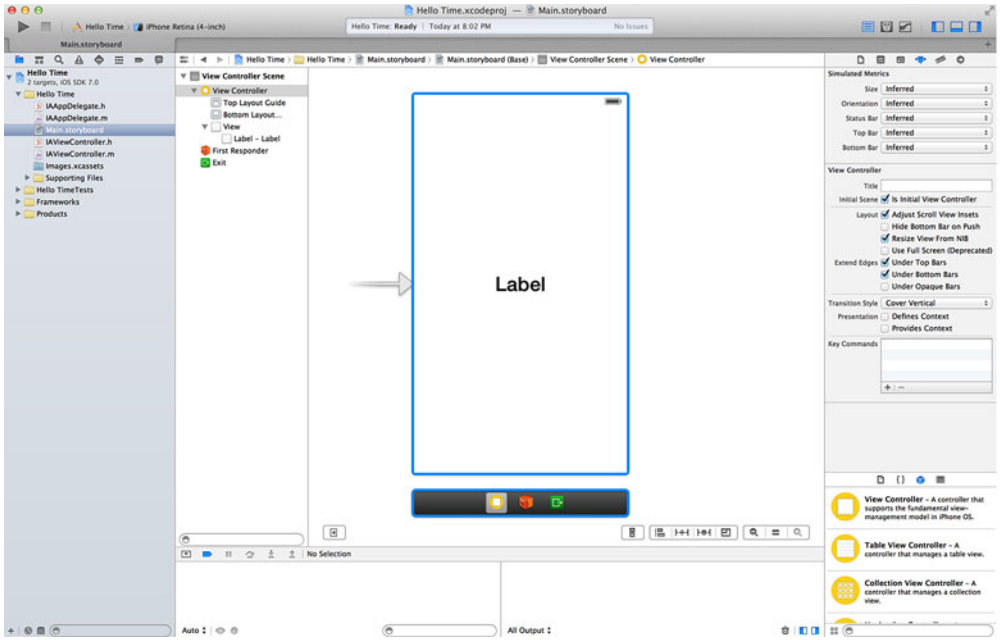


Figure 2.2 The interface for Hello Time just as you left it in the previous chapter

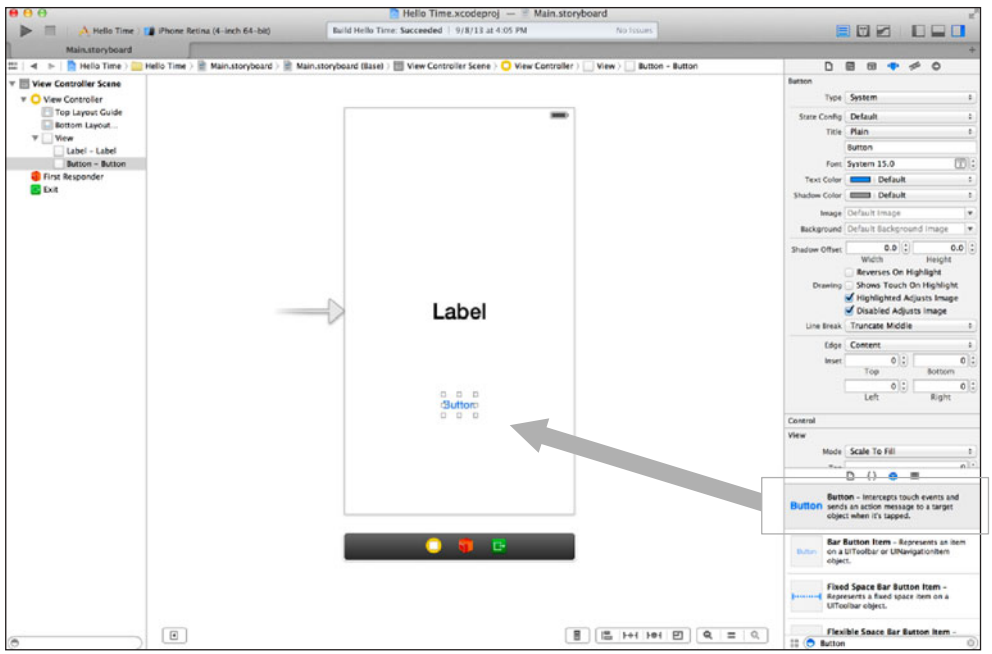


Figure 2.3 Dragging a button from the Object Library onto the bottom of your view

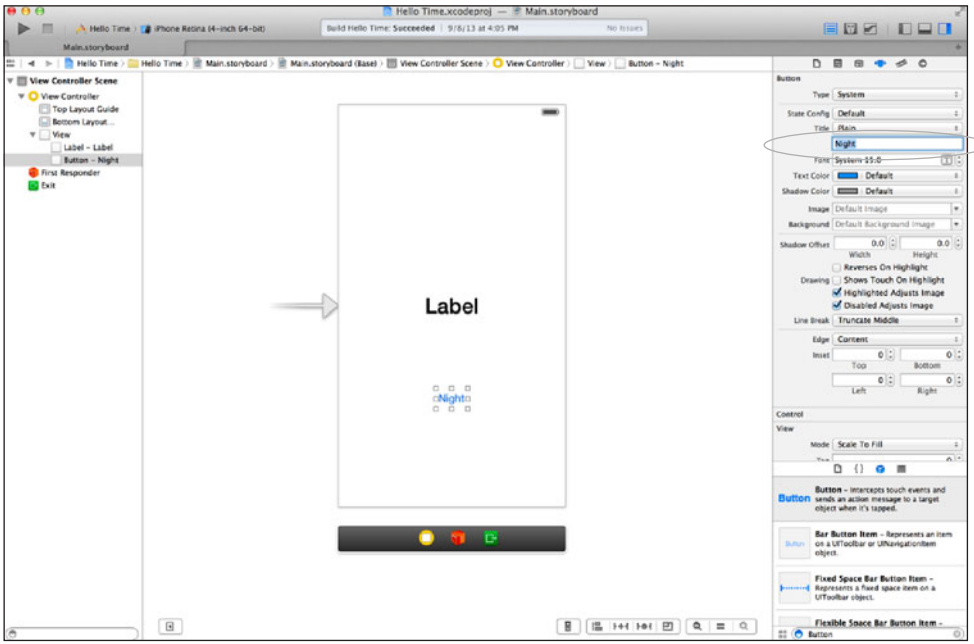


Figure 2.4 Change the title of the button to "Night" by editing the Title property in the inspector.

Next, you can change the text of this button from "Button" to "Night." On the right side of your workspace window, with your button selected, look for the Title property and change its entry to "Night." You can see this in figure 2.4.

You've now finished adding the button to your interface. The next step is to connect the button so that the code you're going to write can interact with it. First, open the assistant editor by choosing View > Assistant Editor > Show Assistant Editor (Option-Command-Return) in the application menu. This should bring up IAView-Controller.h on the right side of your workspace. Hold down the Control key on your keyboard and drag an outlet into `IAViewController`, as shown in figure 2.5.

When you let go, you'll be prompted to set a name for this new connection. Call it `modeButton`. Once you make the connection, the following code will be inserted:

```
@property(weak, nonatomic) IBOutlet UIButton *modeButton;
```

This will allow you to make changes to the button such as modifying the label you're using to display the current time. What about when someone taps the button? How do you know when that happens? Let's create something to handle that event.

In the same way that you dragged a connection to create an outlet, hold down Control and drag the connection into the assistant editor. In the pop-up that appears, change the connection to Action, change the event to Touch Up Inside, and set the name to `toggleMode`, as shown in figure 2.6.

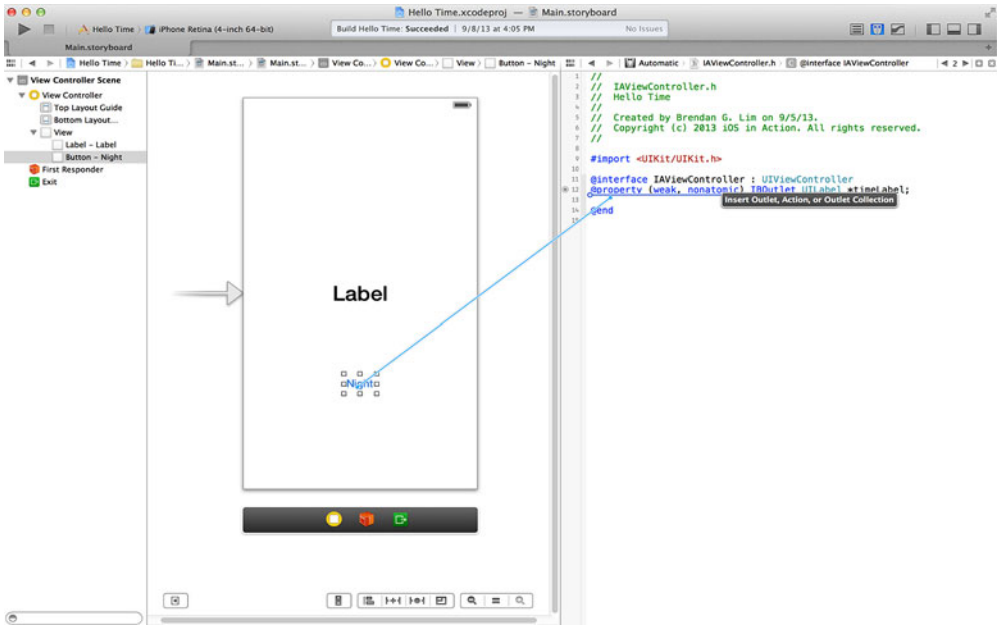


Figure 2.5 Drag a connection from the button in your view to `IAViewController`'s interface.

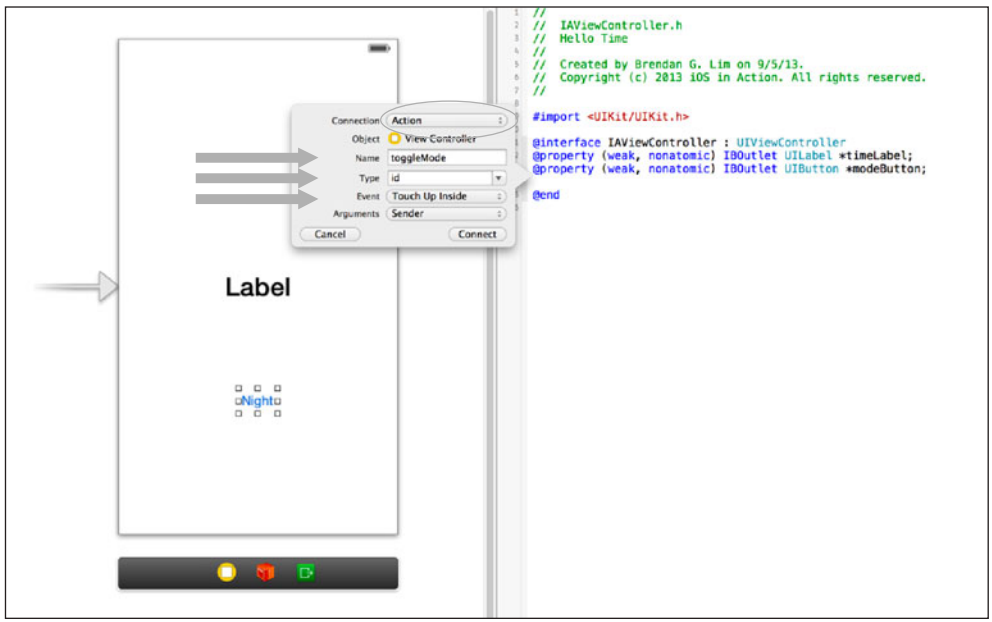


Figure 2.6 Create an action for the Touch Up Inside event called `toggleMode`.

Once the connection is made, Xcode will generate a `toggleMode:` method that will serve as an action that will be triggered when your button is touched. Let's add the code for this action that will enable or disable night mode.

Go back to the standard editor by selecting the application menu and choosing View > Standard Editor > Show Standard Editor (Command-Return). Next, select the project navigator and choose IAViewController.m. Inside the editor you'll see a blank `toggleMode:` method that Xcode generated. Replace it with the code shown in the following listing.

Listing 2.1 Toggling between night and day modes

```
- (IBAction)toggleMode:(id)sender {
    if ([self.modeButton.titleLabel.text isEqualToString:@"Night"]) {
        self.view.backgroundColor = [UIColor blackColor];
        self.timeLabel.textColor = [UIColor whiteColor];
        [self.modeButton setTitle:@"Day" forState:UIControlStateNormal];
    } else {
        self.view.backgroundColor = [UIColor whiteColor];
        self.timeLabel.textColor = [UIColor blackColor];
        [self.modeButton setTitle:@"Night" forState:UIControlStateNormal];
    }
}
```

If you take a look at figure 2.7, you'll see this code added to `IAViewController`.

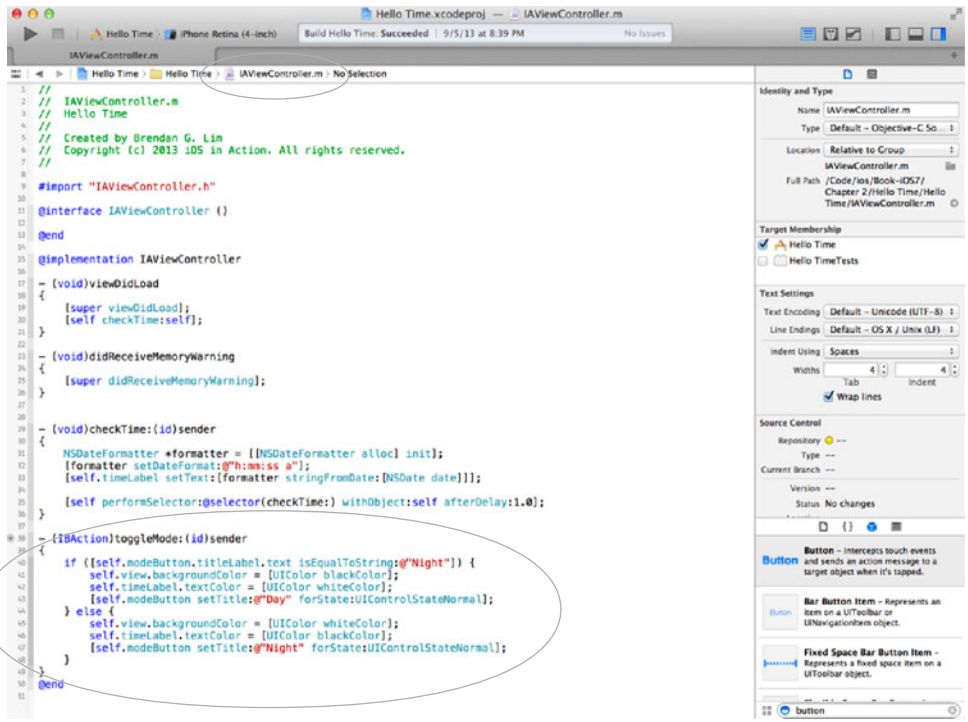


Figure 2.7 Your code added to the `toggleMode:` action within `IAViewController`

Figure 2.8 The Hello Time app currently doesn't support landscape orientation. You're going to fix this.

This will take care of switching between night and day modes. The next thing you're going to add to Hello Time is support for landscape mode.

2.1.2 Adding support for landscape mode

If you were to view your application in landscape mode right now, it would look pretty strange. The time label and the button don't reposition themselves properly when the orientation changes. You can see this in figure 2.8.

You'll alleviate this by adding support for landscape mode. You'll make the label appear in the center of the view and make the button disappear. This will make toggling between night and day modes possible only within portrait mode.

Jump back into Xcode and into IAViewController.m. You're going to add two new methods. One of them will declare the orientations that you'll support. The other will adjust your view so that it appears different in landscape versus portrait mode. Add the two methods in the following listing to the bottom of IAViewController.m.

Listing 2.2 Changing the view depending on orientation

```
- (void) willAnimateRotationToInterfaceOrientation:
(UIInterfaceOrientation)toInterfaceOrientation
 duration:(NSTimeInterval)duration
{
    CGRect timeFrame = self.timeLabel.frame;
    float viewHeight = self.view.frame.size.height;
    float viewWidth = self.view.frame.size.width;
    float fontSize = 30.0f;
    BOOL hideButton = YES;

    if (UIInterfaceOrientationIsLandscape(self.interfaceOrientation)) {
        fontSize = 40.0f;
        timeFrame.origin.y = (viewWidth / 2) - timeFrame.size.height;
        timeFrame.size.width = viewHeight;
    } else {
        hideButton = NO;
        timeFrame.origin.y = (viewHeight / 2) - timeFrame.size.height;
        timeFrame.size.width = viewWidth;
    }
```

```
    [self.modeButton setHidden:hideButton];
    [self.timeLabel setFont:[UIFont boldSystemFontOfSize:fontSize]];
    [self.timeLabel setFrame:timeFrame];
}

- (NSUInteger) supportedInterfaceOrientations
{
    return UIInterfaceOrientationMaskPortrait |
➥ UIInterfaceOrientationMaskLandscape;
}
```

You can see these two methods added to `IAViewController` by taking a look at figure 2.9.

That's it! You've finished adding enhancements to your Hello Time app. You'll be referencing what you've done throughout the chapter. But first, we should take a closer look at what views are and how they are used within an app.

2.2 Introducing views

It's important to familiarize yourself with the basic components that make up the visual layer of your application. These include screens, windows, views, and controls. You'll learn what you're actually seeing in an iOS app and how apps work inside and out.

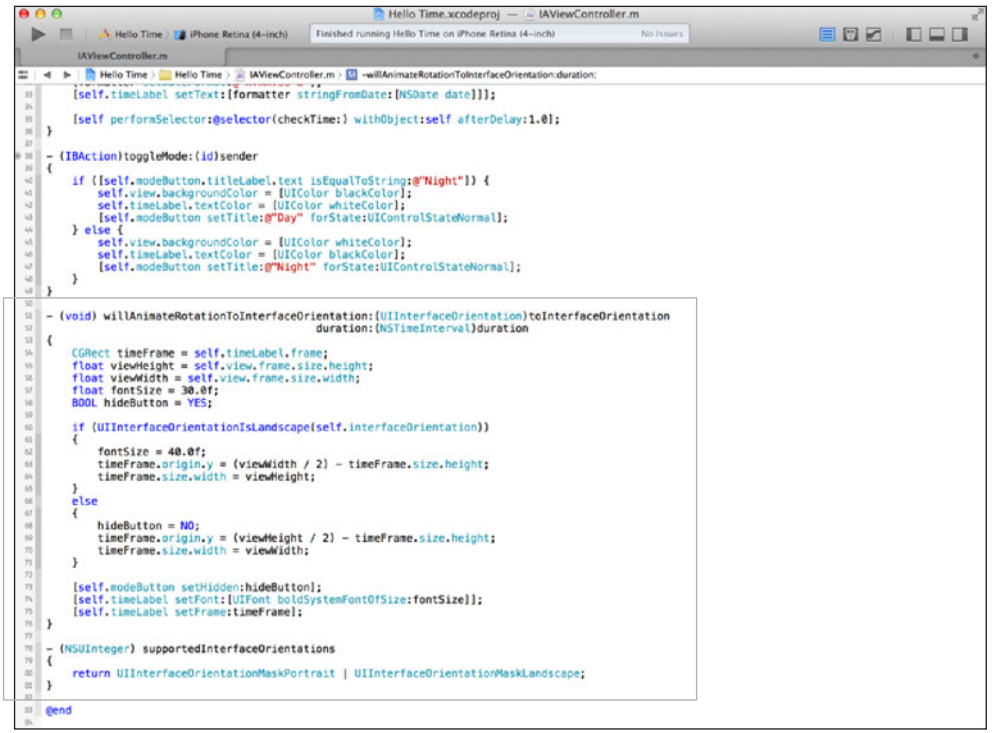


Figure 2.9 Adding two methods to `IAViewController` to support two different orientations

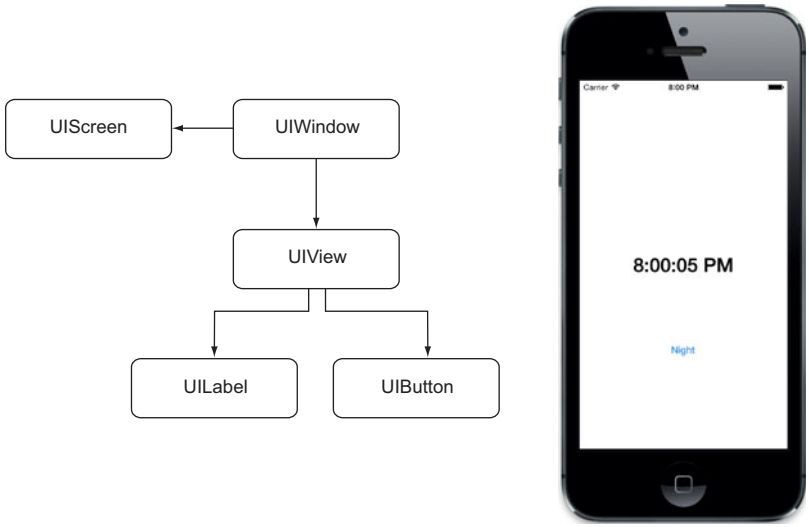


Figure 2.10 The Hello Time app, which has a label and button within a view, which is contained within a window inside a screen

2.2.1 Screens, windows, and views

Everything you see in an app is primarily made up of views. These views are displayed within a window, and that window is contained within a screen. Take a look at the anatomy of a view you'd see within your Hello Time application by focusing on figure 2.10.

Two items are shown on the screen of the iPhone in figure 2.10—a label and a button. Labels (`UILabel`) and buttons (`UIButton`) are known as controls—as are text fields (`UITextField`), images (`UIImageView`), and the like. These controls are all subclasses of `UIView`.

Notice how these two controls are contained within a white area? This white area is a view (`UIView`) that has a property that specifies its background color. We changed this background property when switching to night mode using the following code.

```
self.view.backgroundColor = [UIColor blackColor];
```

This view is contained within a single window (`UIWindow`), which is contained within a screen (`UIScreen`). You generally have only one screen or window when working with an application. You may have more screens and windows if you're using video-out or AirPlay to display something on a different screen or device. Here's an overview of these three types of classes:

- `UIScreen`—Represents the physical screen of the device
- `UIWindow`—Provides drawing support for displaying views on a `UIScreen`
- `UIView`—Represents a user interface element within a rectangular area

As you learned in the previous chapter, every class is a subclass of an `NSObject`. A `UIView` also inherits from `UIResponder`, which means it's capable of responding to actionable

events. A `UIButton`'s direct parent is `UIControl`, which is a subclass of `UIView`. `UIControl` subclasses are views that you directly interact with, like buttons, labels, or text fields. Views can also be nested or layered within a hierarchy. View controls, like a label or button that you're using in Hello Time, are nested as subviews of a parent view.

Views can also be animated, which is used heavily in iOS. Almost every time you transition from one view to another in an app, the current view slides out to the left while the new view slides in from the right. When a modal-style view appears, it comes in from the bottom of the screen and slides upward until it's fully in view. When you delete an item from the Mail app, the row that contains the email you're deleting slides out. When you delete a photo from the Photos app, the `UIImageView` that contains your photo gets sucked into the trashcan button. Animations can be applied as changes to properties of a particular view for a specific duration. You'll learn more about animations in a later chapter. Next, you'll learn about the coordinate system and how views are represented within a window.

2.2.2 Views and the coordinate system

When you lay out your views, you do so by providing X and Y coordinates as well as a width and height. These coordinates are placed within a view coordinate system with the point (0,0) at the top left of the screen. The X coordinate, Y coordinate, width, and height are contained within a `CGRect` and are referred to as a view's frame. You can provide a view with a frame either programmatically or by using Interface Builder.

Most initializer methods for a `UIView` ask for a frame to be provided. For example, let's say you want to create a `UIView` that is 200 x 200 and positioned at the X coordinate 10 and the Y coordinate 10. You would use the `initWithFrame:` method and supply a `CGRect` as your parameter by using the `CGRectMake()` function. This function takes four parameters: `x`, `y`, `width`, and `height`:

```
CGRect frame = CGRectMake(10,10,200,200);
UIView *foo = [[UIView alloc] initWithFrame:frame];
```

Let's give your view a red background to make it stand out. A white view on top of another white view would make it impossible to see.

```
[foo setBackgroundColor:[UIColor redColor]];
```

Your view will appear on the top-left corner of its parent view. You can see this in figure 2.11.

This is because the coordinate system for iOS assumes (0,0) to be at the top-left corner. The coordinate system is also *relative* to the parent of the view. For instance, if you were to add a view within the red `UIView` shown in figure 2.11 at (10,10), it would appear 10 points to the right and to the bottom of the red view's top-left corner. This is because (0,0) of the red view is at its absolute top-left corner. To visualize this, take a look at figure 2.12.

Here we've added a white view at (10,10) with a size of 150 x 150 as a subview of our red view. Instead of appearing at the exact same position as the original red view

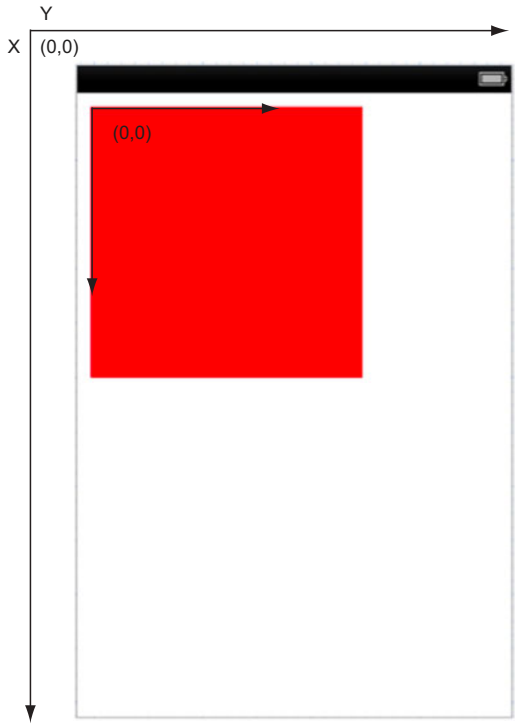


Figure 2.11 A view at coordinates (10,10) and measuring 200 x 200 will appear at the top left of its parent view.

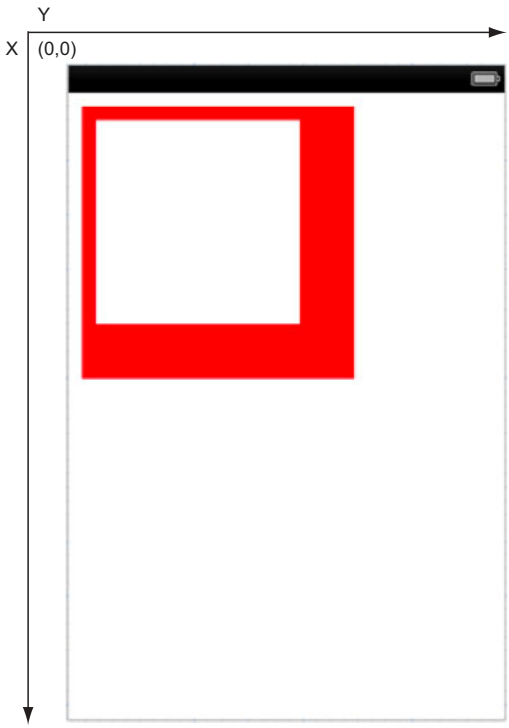


Figure 2.12 A view added as a subview to our red view at (10,10) with a size of 150 x 150

shown in figure 2.11, it assumes a new coordinate system relative to its parent. This means that the origin point (0,0) is at the very top left of the red view.

When you rotate the Hello Time app, you're retrieving the frame of `timeLabel` and editing its Y origin. The new Y origin point was calculated by referencing the height of its parent view. By editing its frame you're able to change its vertical positioning to center it within the view depending on the orientation.

Now that you've learned about the view coordinate system, you can learn more about different UI controls.

2.2.3 User interface controls

User interface controls (or simply *controls*) are represented by the `UIControl` class. Controls are interface elements that a user can view and interact with. You've interacted with many different types of controls when using iOS apps. Examples of different types of controls include labels, buttons, sliders, text fields, selectors, activity indicators, and search bars. Various controls as listed within Xcode are shown in figure 2.13.

To visualize the hierarchy of a control, take a look at figure 2.14.

When you created the Hello Time app, you used the Single View Application template. This created a single view controller that had a view attached to it. You've been editing this view within Main.storyboard. When you were dragging in your label and your button to this view, you were adding them as subviews.

By doing this programmatically you can see what goes into adding a subview. You'd need to pass in a frame to the `alloc:initWithFrame:` method, which is found on most `UIView` subclasses:

```
CGRect frame = CGRectMake(0,0,100,20);
UILabel *label = [[UILabel alloc] initWithFrame:frame];
```

To add this label to its parent view within a view controller, all you have to do is use the `addSubView:` function.

```
[self.view addSubView:label];
```

This will add the label as a subview and position it at (0,0) with a width of 100 and height of 20. You can make changes to this frame as you did in Hello Time by retrieving the frame, making a change to its size or origin, and then resetting its frame property:

```
CGRect frame = label.frame;
frame.origin.x = 10;
frame.size.width = 200;
[label setFrame:frame];
```

In this example, you're retrieving the frame, and setting it's X origin to 10 and its width to 200. You're then resetting the label's `frame` property with the newly modified frame.

2.2.4 Responding to actions and events

In Hello Time you created an action for the button that you used to toggle between night and day modes. This button triggered an action you defined, called `toggleMode:`,

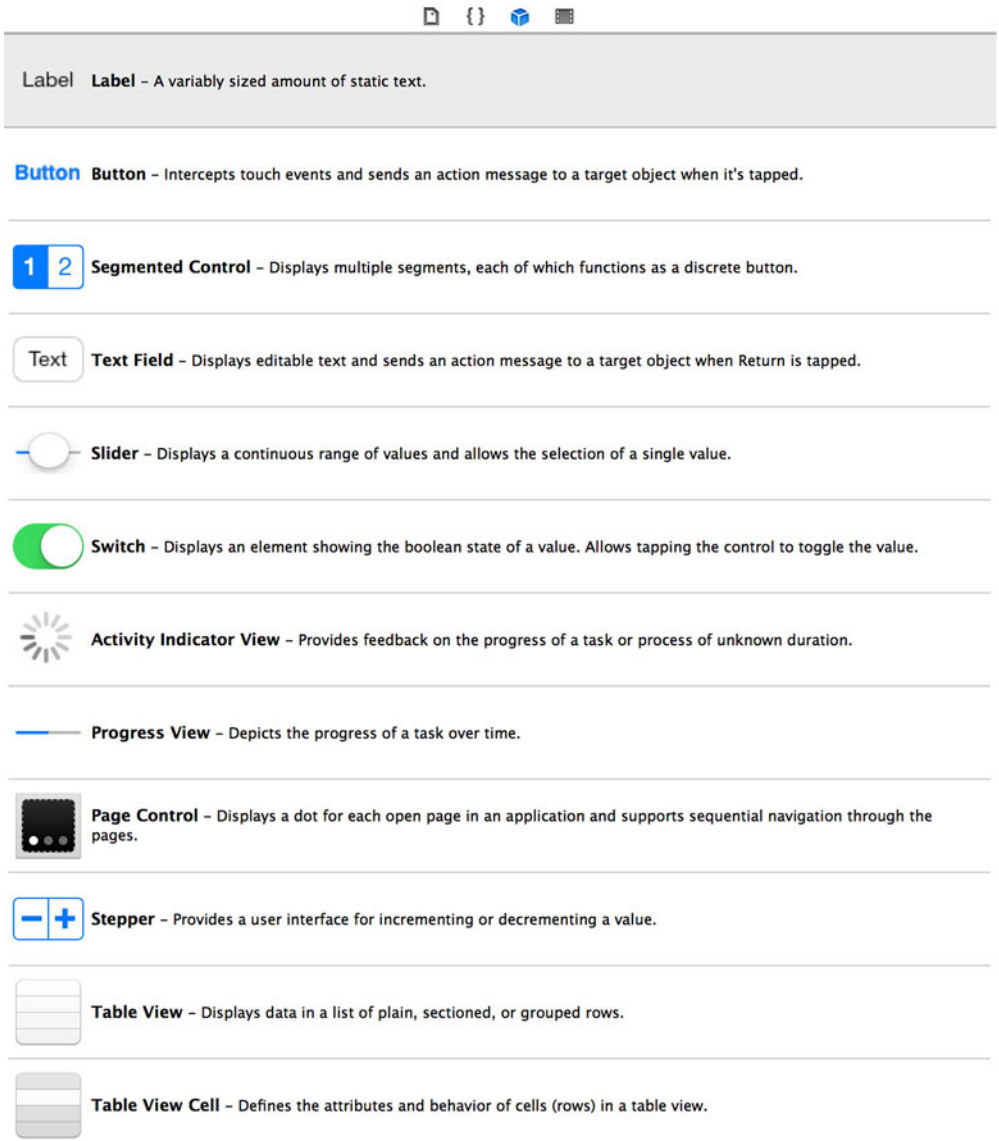


Figure 2.13 List of different user interface controls as shown in Xcode's interface builder

whenever the button was touched. The specific event you used was Touch Up Inside, which is the default event to respond to when someone touches a button. You used Xcode's interface tools by dragging an action connection from the button to your view controller.

Take a second look at the `toggleMode:` action that you have within `IAViewController`. In the method definition it's specified as an `IBAction`.

```
- (IBAction) toggleMode:(id) sender
```

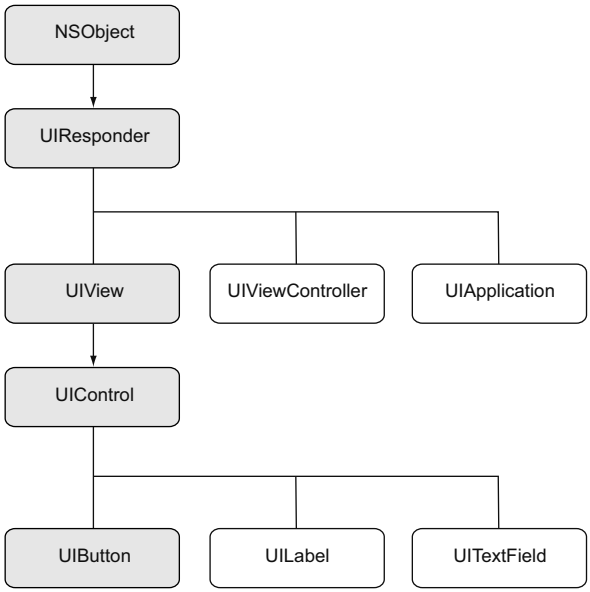


Figure 2.14 Class hierarchy of a `UIResponder` as well as subclasses of a `UIView` and `UIControl`

It also takes in one parameter named `sender`. The object that triggered the action usually fills the `sender` parameter. For instance, your button, `modeButton`, would be accessible through the `sender` parameter within that action because it triggered its execution. If you wanted it to trigger another action for a specific event, you could create another connection through the interface tools, or you could do it programmatically.

By doing it programmatically you can see what's being done to trigger an action for a specific event. Say, for instance, you have another action called `turnRed:`, which changes the color of your label to red. If you want this to happen when your button is touched, you could add the following code:

```
[self.modeButton addTarget:self action:@selector(turnRed:)
➥ forControlEvents:UIControlEventTouchUpInside];
```

This will call the `turnRed:` method on your current class when the Touch Up Inside event is triggered. You'll also notice that for the `action` parameter, you're using `@selector(turnRed:)` instead of just `turnRed:`. The `@selector()` function returns a `SEL`, or selector, which allows you to reference a particular method or action. A selector is essentially a pointer to a method. For example, the previous code example could be written as follows:

```
SEL turnRedSelector = @selector(turnRed:)
[self.modeButton addTarget:self action:turnRedSelector
➥ forControlEvents:UIControlEventTouchUpInside];
```

As we go along, you'll be using controls other than buttons and responding to different types of actions.

2.2.5 Custom tint colors

iOS 7 introduced tint colors to `UIViews`, which are used to define key colors that are used to represent interactivity for user interface elements. Adding a tint color to a view also changes the tint color for all of its subviews. Without manually setting a tint color, the default color will be blue, as you can see in the Hello Time application. The controls that you've added are all shown using the default tint color.

To change the tint color of a view, you just have to modify its `tintColor` property. Shown here is how you'd set the tint color of a particular view and its subviews to red:

```
view.tintColor = [UIColor redColor];
```

If you wanted to change the tint color of an entire application, you could just update the tint color in its `UIWindow`. This is because the `UIWindow` contains all of the subviews within an application. This is shown in the following code example:

```
self.window.tintColor = [UIColor redColor];
```

It's also possible to change the tint color of a view within the interface editor in Xcode. By selecting a view, you can modify its tint color in the attributes inspector. Next, let's take a closer look at view controllers.

2.3 View controller basics

View controllers manage each separate view in your application, which are the screens you see in an iOS application. Although managing views is what a view controller does, that's not its only responsibility. View controllers also handle transitions between other view controllers and transferring data between them. When you tap a button or any other type of control and are transitioned from one view to another, a view controller handles that transition.

2.3.1 Introducing view controllers

Simply put, a view controller manages your views and segues between view controllers. They are a core component of each iOS app and act as the glue between your views and your models. They initialize and set up your models and populate your views. View controllers are also the controller objects in the MVC pattern.

If you go back to the Hello Time project in Xcode, you'll see that two methods were already declared for you: `viewDidLoad` and `didReceiveMemoryWarning`. Both names are self-explanatory. One is for handling when the view has already been loaded, and the other is for when you need to handle a situation where you received a memory warning. These two methods help with the view controller's lifecycle. Let's explore this, and you'll see how you can override certain methods to help manage your controllers.

Naming conventions for view controllers

It's good practice to use very specific names for view controllers throughout your project. This makes maintenance and reusability much easier to manage. It also makes it easy to work with other developers. In your Hello Time project, you're using `IAViewController` for your main view. This is because it was generated for you using the Single View Application template. You can change its name if you want to. If you were building a photo-picker controller for a photo-sharing application, you could name it something like `IAPhotoSelectionViewController`.

2.3.2 The view controller lifecycle

Understanding the lifecycle of a view controller allows you to properly manage the models and views contained within it. It helps you understand when your view will be ready for you to manipulate and when you should have to clean things up when your view is about to be removed. Having a deep knowledge of the view lifecycle is a core part of becoming a great iOS developer because everything revolves around view controllers and using them effectively.

When any part of your application asks a specific view controller for its `view` property, it will kick off a chain of events. If the `view` property for this view controller is not yet loaded into memory, it will call a method called `loadView`. Once this view is fully loaded, it will call the `viewDidLoad` method. This is where you can start initializing any data needed for your views. You can see this flow in figure 2.15.

Looking at the flow for retrieving and setting up a view in a view controller, you can see that you can override the `loadView` function to programmatically create a `UIView` to set as the view controller's `view` property. This is opposed to loading it from a nib or from a storyboard, as you've been doing in Hello Time.

There are a few more methods throughout the lifespan of a view controller that are triggered by various view events. These methods give you more precise control throughout a view controller view's different states. They are listed in table 2.1.

Table 2.1 Methods related to the lifecycle of a `UIViewController`

Method	Description
`loadView`	Creates or returns a view for the view controller.
`viewDidLoad`	View has finished loading.
`viewWillAppear:`	View is about to appear with or without animation.
`viewDidAppear:`	View did appear with or without animation.
`viewWillDisappear:`	View is about to disappear with or without animation.
`viewDidDisappear:`	View did disappear with or without animation.
`viewWillLayoutSubviews`	View is about to lay out its subviews.
`viewDidLayoutSubviews`	View did lay out its subviews.
`didReceiveMemoryWarning`	View detected low memory conditions.

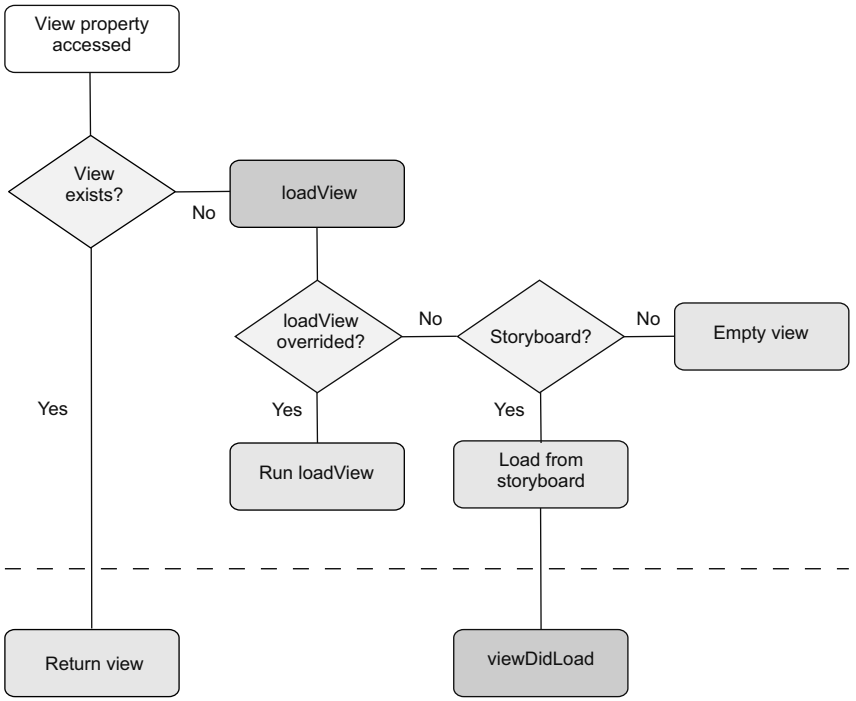


Figure 2.15 The flow for retrieving and setting up a view in a view controller

Because of the verbosity of Cocoa's method names, it's rather simple to understand when each of these methods is called. Within `IAViewController` in your Hello Time project, you added a call to `checkTime:` at the end of the `viewDidLoad` method. This was added here because you want to trigger the time check *after* your view is ready to be used.

Let's go one step further and add something else to the bottom of `viewDidLoad`. You'll add a log statement so you can see when it's being triggered. Add the following code to the bottom, underneath the call you added to check the time:

```
NSLog(@"viewDidLoad called");
```

Now override `viewWillAppear:` and `viewDidAppear:` by adding the following code:

```
- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];
    NSLog(@"viewDidAppear: called");
}

- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
    NSLog(@"viewWillAppear: called");
}
```

Run the application and look at the console within Xcode. You can activate the console by pressing Shift-Command-C or by choosing View > Debug Area > Activate Console. You'll then see the following output:

```
Hello Time [12139:c07] viewDidLoad called
Hello Time [12139:c07] viewWillAppear: called
Hello Time [12139:c07] viewDidAppear: called
```

From this you can see that the first method that's called is `viewDidLoad`, followed by `viewWillAppear:`, and then finally `viewDidAppear:`. Before `viewDidLoad` is triggered, your view controller checks to see if `loadView` has been overridden. In this case it's not because it's being loaded from the storyboard. You could also go one step further and add in calls to see when your view disappears by adding log statements to `viewWillDisapear:` and `viewDidDisappear:`. These would normally be triggered when you decided to load another view controller in your application.

You've used only a basic view controller, but there are many different types to choose from. Apple provides many for you to use for different types of applications.

2.3.3 Different types of view controllers

There are different types of view controllers that you can use without having to write your own from scratch. Some of these view controllers help you manage data by laying it out in a table or grid format. Others manage other view controllers that allow you to display them in a tab-based layout or in a hierarchical structure. All of these view controllers are listed in table 2.2.

Table 2.2 Different out-of-the-box subclasses of `UIViewController` in iOS

UIViewController subclass	Description
`UINavigationController`	Manages navigation of hierarchical view controllers
`UITabBarController`	Represents and manages multiple view controllers as tabs
`UITableViewController`	Presents and manages data represented as a table
`UICollectionViewController`	Presents and manages data represented as a collection

You've encountered most of these while using other iOS apps. Let's take a quick high-level look at each of these to see how they differ, starting with `UINavigationControllers`.

NAVIGATION CONTROLLERS

Navigation controllers (`UINavigationController`) handle the display of data hierarchically using multiple view controllers within a stack. The first view controller within a navigation controller is referred to as the root view controller. When displaying a view, you can push another view onto the stack. The navigation controller allows you to go back by popping the current view off the stack until you reach the

Figure 2.16 A navigation controller in action within the default Settings app

root view. Figure 2.16 shows a navigation controller in action within the default Settings app.

As you can see in the figure, the Settings app uses a navigation controller. By tapping on Twitter, a new view is pushed onto the navigation controller and brought into display. A back button with the label Settings is also added to this newly pushed view. From there you can go one level deeper by clicking a Twitter account, which pushes yet another view.

TAB BAR VIEW CONTROLLERS

Tab bar view controllers (`UITabBarViewController`) offer a simple way to segment different view controllers for your users. The identifying element of a tab bar controller is, of course, the tab bar. The tab bar view controller is used in Apple's App Store application, as shown in figure 2.17.

This tab bar (`UITabBar`) can contain several tab bar items (`UITabBarItem`). These tab bar items contain an icon and a title used to represent a specific view that it will display once activated. Each tab contains a single view controller, which can have its own hierarchy. For example, a view controller contained within a tab can be a `UINavigationController`.

TABLE VIEW CONTROLLERS

Table view controllers (`UITableViewController`) contain a single table view. Table views allow you to display data as a list of rows represented by table view cells. These rows of data can also be visually separated and sectioned into groups.

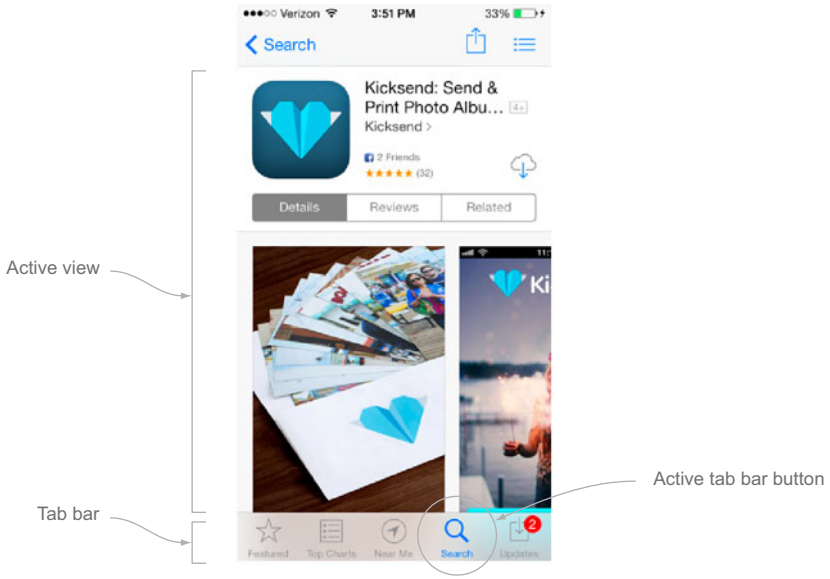


Figure 2.17 `UITabBarViewController` used in Apple's App Store application

A table view controller does much of the table view setup for you and doesn't make you worry about manually plugging in the right methods to feed your table view with data. It's common to use a regular view controller and manually add a table view to it without relying on a table view controller to do it for you. Depending on the kind of data you're displaying or the complexity of your app, you may want more control over how your data is set up.

Figure 2.18 shows how a table view controller would look if it contained a grouped table view with multiple sections.

It's virtually impossible to tell the difference between a table view controller and a regular view controller with a single table view attached because they both display a table view.

So far you've dealt with both portrait and landscape orientations in the Hello Time app. Now you'll learn more about supporting different types of orientations.

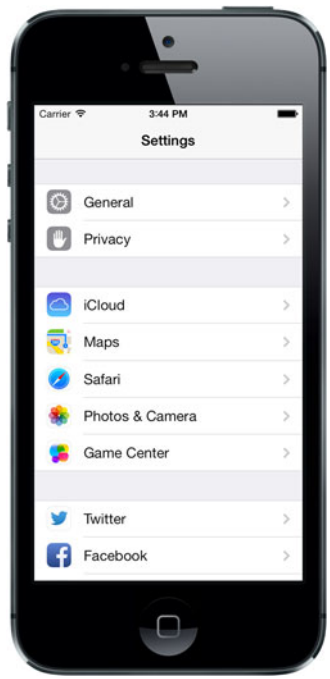


Figure 2.18 Table view shown within a table view controller as seen in the Settings application

2.3.4 Different status bar styles

The status bar includes the time, battery charge, network, and Wi-Fi information. In iOS 6 the status bar

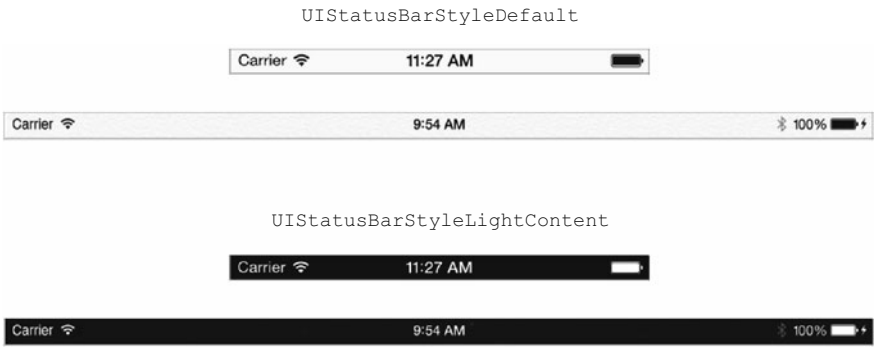


Figure 2.19 Different status bar styles that should be used depending on the type of content shown

had a specific tint that was separate from the view directly underneath it. In iOS 7 this changed because the status bar is transparent and shows the view behind it. Different status bar styles are available that you can specify using a `UIStatusBarStyle` constant. One specifies whether the *content* should be dark (`UIStatusBarStyleDefault`) or light (`UIStatusBarStyleLightContent`). This is shown in figure 2.19.

The status bar style can be globally set in the project's settings in the General tab within Xcode. There are times where you might not want to have your status bar be the same style throughout each view controller in your application. For example, you could have one view where you're showing light content and another where you're showing dark content. To do this you'd first have to set the `UIViewControllerBasedStatusBarAppearance` key to `YES` in your application's `plist` file.

Next, you'd need to implement the `preferredStatusBarStyle` method in your view controller. This method expects to return a `UIStatusBarStyle` constant, which means if you wanted to use a status bar for a screen with light content, you'd return `UIStatusBarStyleLightContent`, as shown here:

```
- (UIStatusBarStyle)preferredStatusBarStyle
{
    return UIStatusBarStyleLightContent;
}
```

To trigger this method, you'll have to call `[self setNeedsStatusBarAppearanceUpdate]` within your view controller. You can do this within `viewDidLoad` or anywhere else that would warrant a new status bar style. Next, we'll explore how to support different types of orientations.

Extended layout support in iOS 7

In iOS 7 content extends from edge to edge of the screen as opposed to iOS 6. Content is also displayed underneath navigation and tab bars so that it fills the screen. It's possible to not have your content go from edge to edge by updating the `edgesForExtendedLayout` property on a `UIViewController`. You could set this to `UIRectEdgeNone` if you don't wish to support extended layout or to `UIRectEdgeAll` if you do.

2.4 Supporting different orientations

An iOS app can be used in either portrait or landscape mode. Portrait mode is the default mode for any new application you create for Xcode. Landscape mode is more popularly used for viewing photos or videos in full screen. Depending on the type of application you're building and the type of device (iPhone or iPad), you could choose to support either orientation or just one. When supporting both orientations you have to add in functionality so that your views react accordingly to the change in width and height of the new orientation.

2.4.1 Enabling support for portrait and landscape

There are four different types of orientations you can choose to support: portrait, upside-down portrait, landscape left, and landscape right. In Hello Time you're supporting three different types of orientations by default. You can see this by going to the project navigator in Xcode and viewing your target's general settings, as shown in figure 2.20.

By default, upside-down portrait is not supported, but you can easily enable it by clicking it inside your target's Summary tab in Xcode. By doing this, all of your views won't automatically resize when the device's orientation changes. You also still need to specify in each specific view controller what orientations are supported. This is because you may have certain view controllers where you would like to support only portrait and another where you may want to play video and support portrait *and* landscape. In Hello Time you specified which orientations you support by adding the following code using the `supportedInterfaceOrientations` method:

```
- (NSUInteger) supportedInterfaceOrientations
{
    return UIInterfaceOrientationMaskPortrait |
➥ UIInterfaceOrientationMaskLandscape;
}
```

You could have used the `UIInterfaceOrientation` enumerable type to specify each specific orientation. All of these are listed in table 2.3.

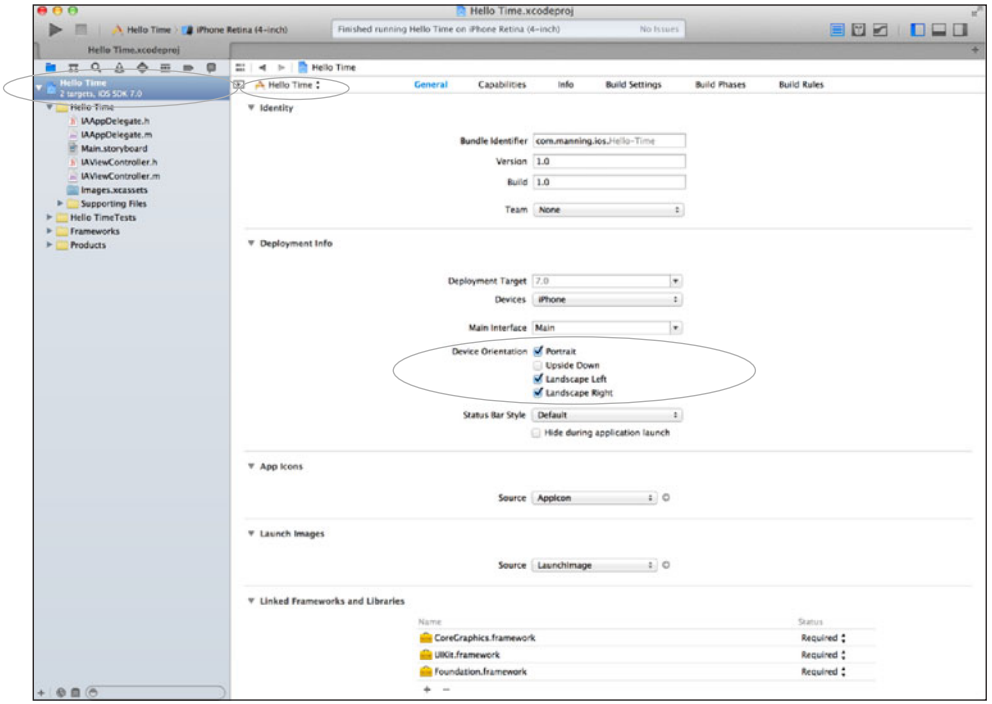


Figure 2.20 The three different orientations supported by Hello Time

Table 2.3 `UIInterfaceOrientation` enumerable for representing different orientations

UIInterfaceOrientation	Orientation description
`UIInterfaceOrientationPortrait`	Portrait
`UIInterfaceOrientationPortraitUpsideDown`	Upside-down portrait
`UIInterfaceOrientationLandscapeLeft`	Left landscape
`UIInterfaceOrientationLandscapeRight`	Right landscape

In Hello Time, you used a mask to represent the supported orientations. This allowed you to combine all portrait and landscape orientations instead of writing them all out. These orientation bit masks are used to represent multiple orientation combinations, as shown in table 2.4.

Table 2.4 `UIInterfaceOrientationMask` enumerable for representing different orientation combinations

UIInterfaceOrientationMask	UIInterfaceOrientations supported
`UIInterfaceOrientationMaskPortrait`	Portrait
`UIInterfaceOrientationMaskPortraitUpsideDown`	Upside-down portrait

Table 2.4 `UIInterfaceOrientationMask` enumerable for representing different orientation combinations *(continued)*

UIInterfaceOrientationMask	UIInterfaceOrientations supported
`UIInterfaceOrientationMaskLandscapeLeft`	Left landscape
`UIInterfaceOrientationMaskLandscapeRight`	Right landscape
`UIInterfaceOrientationMaskPortraitAll`	Portrait, upside-down portrait
`UIInterfaceOrientationMaskLandscapeAll`	Left landscape, right landscape
`UIInterfaceOrientationMaskAll`	Portrait, upside-down portrait, left landscape, right landscape

If you wanted to support just portrait (and not also upside-down portrait) within your view controller, you'd add the following:

```
-(NSUInteger) supportedInterfaceOrientations
{
    return UIInterfaceOrientationPortrait;
}
```

If you wanted to support both portrait and landscape orientations, you could change the `supportedInterfaceOrientations` method to the following:

```
-(NSUInteger) supportedInterfaceOrientations
{
    return UIInterfaceOrientationMaskAll;
}
```

2.4.2 Updating your views for different orientations

When you added support for landscape, your views didn't support the new orientation automatically. When you first positioned the label and button for your views, you did it for the portrait orientation. After adding support for the landscape orientation, you saw how they looked when the device was rotated, as shown in figure 2.21.

You then added a method to your view controller that allowed you to know when the orientation of your view controller changed. Adding the `willAnimateRotationToInterfaceOrientation:toInterfaceOrientation:duration:` method gave you the opportunity to make adjustments to the positioning of your views:

Figure 2.21 The views in the Hello Time application didn't automatically support the landscape orientation.

```
- (void) willAnimateRotationToInterfaceOrientation:
➥ (UIInterfaceOrientation)toInterfaceOrientation
➥ duration:(NSTimeInterval)duration
{
    CGRect timeFrame = self.timeLabel.frame;
    float viewHeight = self.view.frame.size.height;
    float viewWidth = self.view.frame.size.width;
    float fontSize = 30.0f;
    BOOL hideButton = YES;

    if (UIInterfaceOrientationIsLandscape(self.interfaceOrientation)) {
        fontSize = 40.0f;
        timeFrame.origin.y = (viewWidth / 2) - timeFrame.size.height;
    } else {
        hideButton = NO;
        timeFrame.origin.y = (viewHeight / 2) - timeFrame.size.height;
    }

    [self.modeButton setHidden:hideButton];
    [self.timeLabel setFont:[UIFont boldSystemFontOfSize:fontSize]];
    [self.timeLabel setFrame:timeFrame];
}
```

This method is automatically called in your view controller when the device is about to be rotated. In this method you checked to see what orientation your view was in and adjusted your label's frame accordingly. You also took the opportunity to change the background color of your main view and to hide the mode button. Something you could have done to avoid having to adjust the frame of each view as you did was to use auto layout.

Auto layout was introduced in iOS 6 and is helpful with situations like this. It allows you to specify certain constraints on your views so that they can reposition themselves depending on screen size or orientation. There are still times when you'll want to do more to your views when viewing them in a different orientation. You'll learn about auto layout and storyboarding in the next chapter.

2.5 Summary

We've only skimmed the surface of views and view controllers. As we progress, we'll be diving deeper into different ways of working with views and using advanced view controllers to help you make more immersive, rich applications.

- You saw how to position views within the view coordinate system.
- Views can be added and modified programmatically and even nested within one another.
- By knowing your view controller's lifecycle, you know how and when to properly manage the models and views it contains.
- Custom tint colors can be applied to a `UIView`, which will be applied to all of its subviews.
- Different types of view controllers are available to you out of the box, such as the `UITableViewController`, `UITabBarViewController`, and `UINavigationController`.

- You can create actions and connect them to respond to a specific event on a `UIControl` such as a button.
- You can specify different status bar styles per view controller.
- You can support two different types of portrait and landscape orientations.
- It's possible to alter your views depending on the orientation of the device.
- You updated your Hello Time application by adding a separate night mode as well as support for landscape orientation.

3 Using storyboards to organize and visualize your views

This chapter covers

- Creating a task management app
- Overview of Xcode's interface tools
- Using storyboarding in your applications
- Transitioning between scenes using segues
- Passing data between view controllers

Until recently, every view controller in iOS needed its own separate file for its interface (known and phonetically pronounced as NIBs but having the extension .xib). NIBs are individual interface files that allow you to create interfaces graphically instead of programmatically. You would need to edit these independently of other view controllers within the same application. Some developers would even choose to skip NIBs altogether and create their views programmatically. Those who did this often felt that they needed more control or were uncomfortable using Xcode's built-in interface tools. Even though these two approaches to creating interfaces are radically different, they do have a few things in common. For one, with either solution, no single file encompasses all of the interfaces for the application. They also lack the ability to visually see how each view controller interacts with others within their application.

Since the release of iOS 5, Apple has included an entirely new way to create, organize, and connect a collection of views within an app into a single file: storyboarding. Storyboarding also lets you manage the way views segue between each other. Above all, storyboards can help you cut down on the amount of code you have to write and make it easier for you to create your apps. In this chapter you'll become familiar with Xcode's interface tools, learn to use storyboarding, and even work with table views. Within this chapter you'll be building your very own task management app, as shown in figure 3.1.

To begin you'll create your app, which is aptly named Tasks. After you've finished creating your application, we'll go over the process.

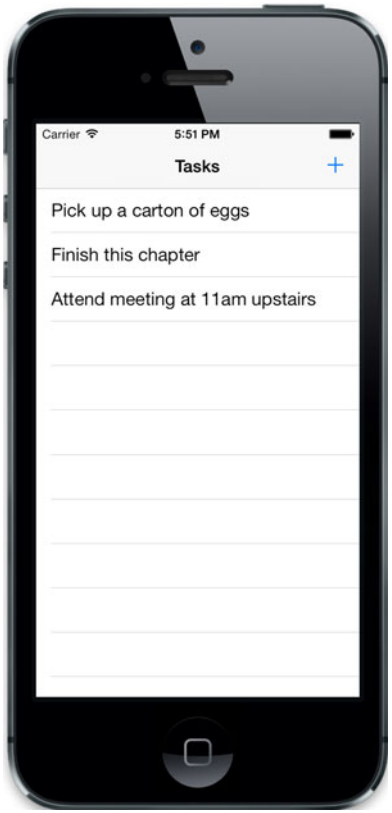


Figure 3.1 The task management app we'll be building together using storyboarding

3.1 Building a task management app

When you created Hello Time in chapter 1, you used the Single View Application template. This served you well, and you had only one view throughout the application. You'll be using the same template for your Tasks application, but this time you'll be creating an application with more views and have them interact with one another. Follow along, and we'll come back and explain the process after you've finished creating the Tasks app.

3.1.1 Creating the Tasks app project in Xcode

To create a new project for your Tasks app, start by opening Xcode. Choose File > New Project and choose the Single View Application project template. Name the application Tasks, as shown in figure 3.2.

Once you've created your project, you should have a file called Main.storyboard in the project navigator. Click this file to open the interface editor.

3.1.2 Creating the interface for listing tasks

Let's add a table view that will be used to list the tasks that you want to display. With Main.storyboard selected and the interface editor open, look through the Object Library on the bottom right of the workspace window to find the table view, as shown in figure 3.3.

Using the mouse, click and drag the table view from the Object Library onto the view in the editor area. As the table view is being dragged in, the sides of the table view

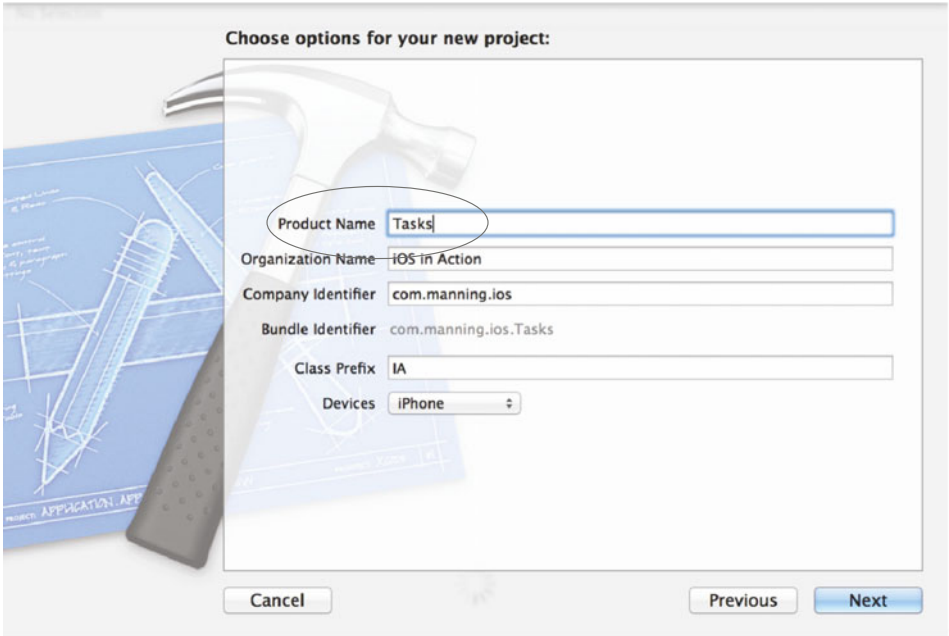


Figure 3.2 Creating a new project in Xcode named Tasks targeted for the iPhone

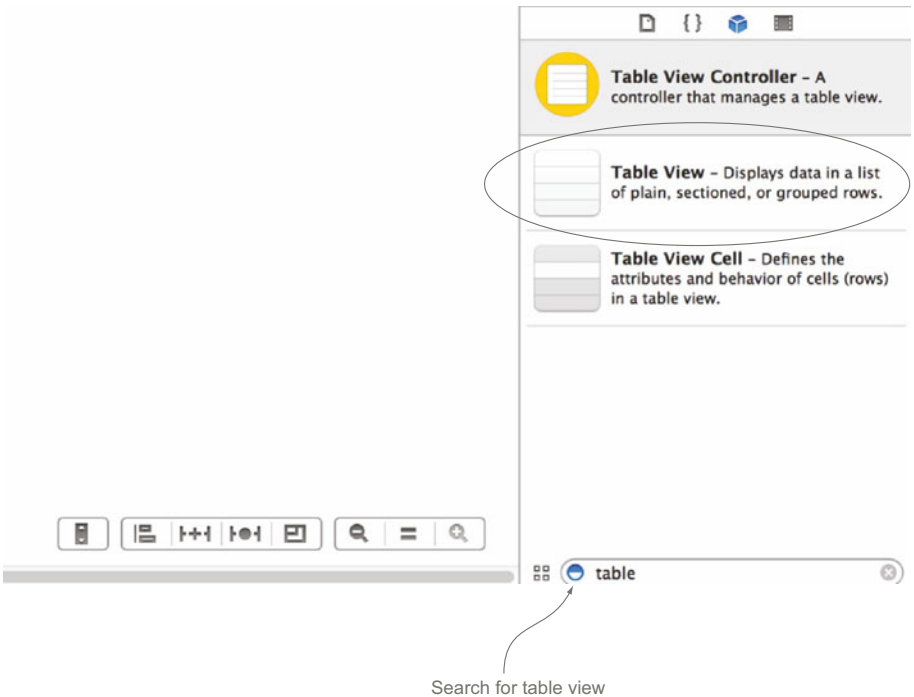


Figure 3.3 Searching for the table view in the Object Library to add to the view

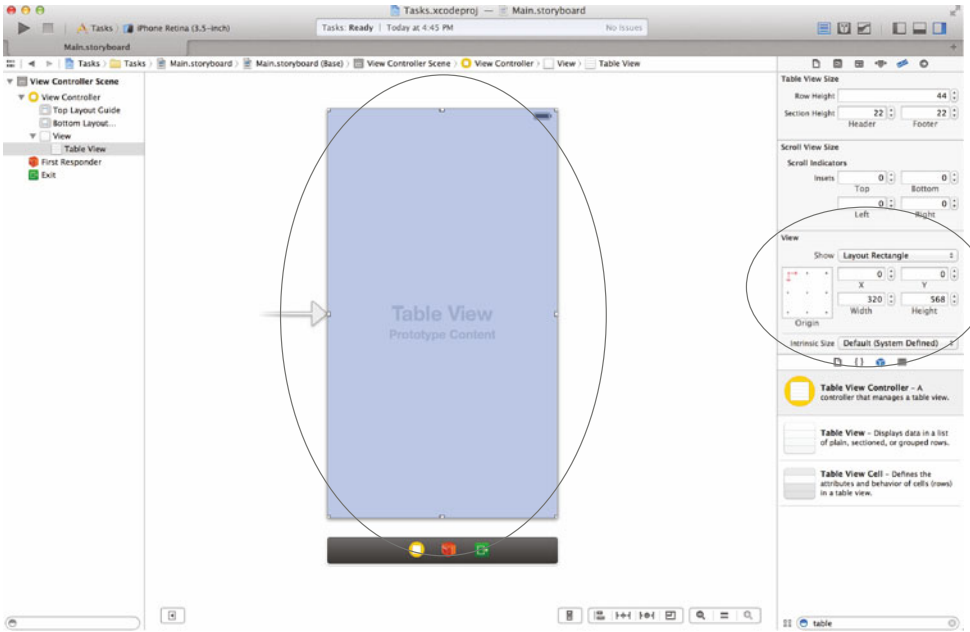


Figure 3.4 Dragging a table view into the view and positioning it so that the X and Y coordinates are set to 0, as shown in the size inspector

will snap to the bounds of the view you're dragging it into. Once you've dragged the table view into your view and selected it, open the size inspector by choosing View > Utilities > Show Size Inspector. From here you can check the X and Y origin points for the table view. The origins should be set to 0 for both, as shown in figure 3.4.

You can now make an outlet from the user interface to the `IAViewController` class. First, open the assistant editor by choosing View > Assistant Editor > Show Assistant Editor (Command-Option-Return) in the application menu and bring up the header file. With the table view selected in your interface, hold down the Control key while clicking and dragging from the table view to your `IAViewController`'s class definition in the assistant editor. If you don't hold down the Control key, you will end up dragging the element around the screen. Once you've finished dragging and let go, a modal will appear asking you to name the outlet you're setting on your class for this table view. Name it `tableView`, as shown in figure 3.5, and then click Connect.

Once your outlet has been connected, Xcode will add the following property:

```
@property (weak, nonatomic) IBOutlet UITableView *tableView;
```

There are two other quick outlets you need to add for your table view before we move on. A table view needs a delegate and a data source to respond to actions and to supply it with data, and you can set this right from the interface editor. Delegates and data sources are very important topics in Cocoa development. We won't go into detail with them yet, but we will in the next chapter.

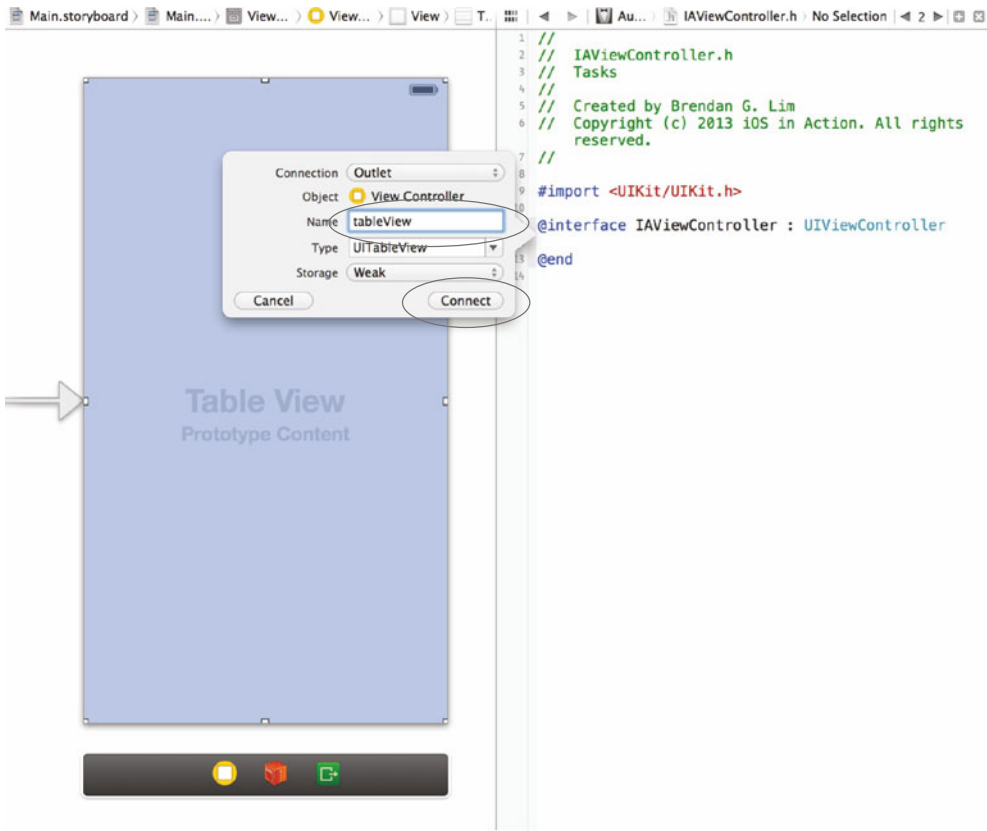


Figure 3.5 Set the name of the outlet for the table view to `tableView` and then click Connect.

Go back into the interface editor and make sure your table view is selected. Open the Connections Inspector tab by choosing View > Utilities > Show Connections Inspector (Command-Option-6) in the application menu. You should see two outlets that have not yet been connected: `dataSource` and `delegate`. Click the circle to the right of both of these outlets and drag to View Controller in the scene list on the left side of the window, as shown in figure 3.6.

After setting both of these, hop back into IAViewController.h by opening the assistant editor again. You need to declare your controller as a class that conforms to a `UITableView`'s data source and delegate protocols. On top of the outlets you've made, your table view will check to see if you've set this in your class definition. Change the top class definition line in IAViewController.h to the following:

```
@interface IAViewController : UIViewController <UITableViewDelegate,
➥ UITableViewDataSource>
```

You should notice the addition of `<UITableViewDelegate, UITableViewDataSource>`. This declares that your view controller conforms to these two protocols. If you've done

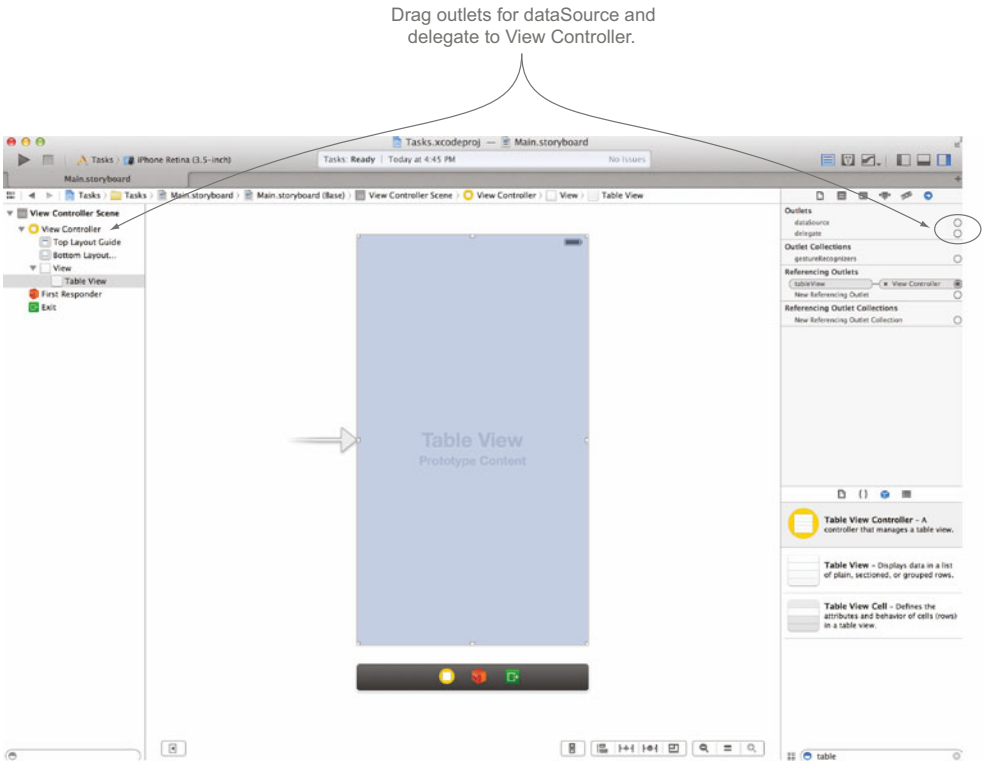


Figure 3.6 Creating outlets to specify the data source and delegate of the table view as the view controller

Java development before, this is similar to specifying that a class implements a specific interface. You'll need to implement the methods that are required by these protocols to be able to use your table view.

Let's add a property that you'll use to represent a mutable array of tasks. Each task will just be an `NSString`, and you'll store these within a `tasks` property. Add it directly underneath the property that you created for the table view:

```
@property (strong, nonatomic) NSMutableArray *tasks;
```

Now go to IAViewController.m, and add the following to the bottom of the `viewDidLoad` method to initialize your tasks array:

```
self.tasks = [[NSMutableArray alloc] init];
```

This allocates and then initializes a new `NSMutableArray` for the `tasks` property, which you'll be using to store each individual task that's created. Next, you need to set up the methods needed for your table view to feed it data and respond to methods that belong to the `UITableViewDelegate` and `UITableViewDataSource` protocols.

Let's start adding two required `UITableViewDataSource` methods by adding a method called `tableView:numberOfRowsInSection:`. This method returns the number

of rows within each section of your table view. For your application, you will always have just one section, so you will just return the total number of tasks in your array:

```
- (NSInteger) tableView:(UITableView *)tableView
➥ numberOfRowsInSection:(NSInteger)section
{
    return [self.tasks count];
}
```

Next, add `tableView:cellForRowAtIndexPath:`, which will return a `UITableViewCell` to represent each task for each row in the table view. You will keep this basic and have it set the title property of `UITableViewCell` to the title of your task, as shown in the following listing.

Listing 3.1 Returning a cell for each row you're displaying

```
- (UITableViewCell *) tableView:(UITableView *)tableView
➥ cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier];
    if(cell == nil)
        cell = [[UITableViewCell alloc]
initWithStyle:UITableViewCellStyleDefault
reuseIdentifier:CellIdentifier];#C

    cell.textLabel.text = self.tasks[indexPath.row];

    return cell;
}
```

1 Create static cell identifier.

2 Attempt to dequeue reusable cell.

3 If cell is nil, create a new cell.

4 Set the text label on the cell.

5 Return the cell.

In this method you set a static cell identifier named `"Cell"` 1. You then retrieve a cell using the `dequeueReusableCellWithIdentifier:` method 2. This method will try to retrieve an existing cell so that it doesn't waste resources and time creating a new one for each row you're displaying. If an unused cell doesn't exist, you create a new one 3. You then set text on the cell to be of the specific task of the row you're displaying 4. Finally, you return the cell 5.

3.1.3 Adding a navigation controller

You'll be using a navigation controller because you want to be able to drill down into a task from your tasks list. To embed a view within a navigation controller, all you have to do is select your view controller, go to the menu bar, and choose Editor > Embed In > Navigation Controller, as shown in figure 3.7.

Your view will now be embedded in a navigation controller. You've also inherited the navigation bar that comes with a navigation controller. It doesn't have a title right now, and you should fix this so that your users know what this view is. To let your users know that this will show them their tasks, click this bar, select the attributes inspector, and set its title to Tasks, as shown in figure 3.8.

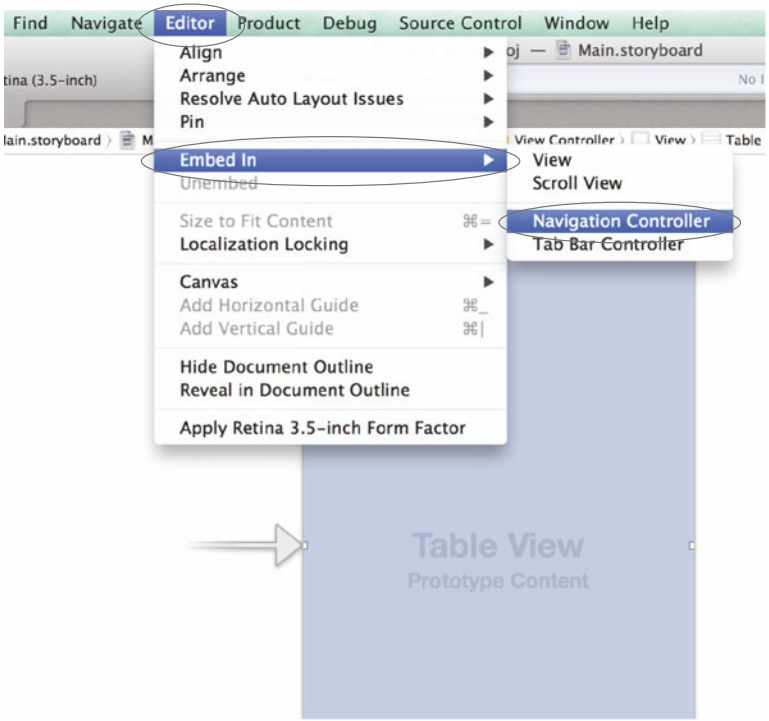


Figure 3.7 Embedding an existing view into a navigation controller is extremely simple when using storyboards.

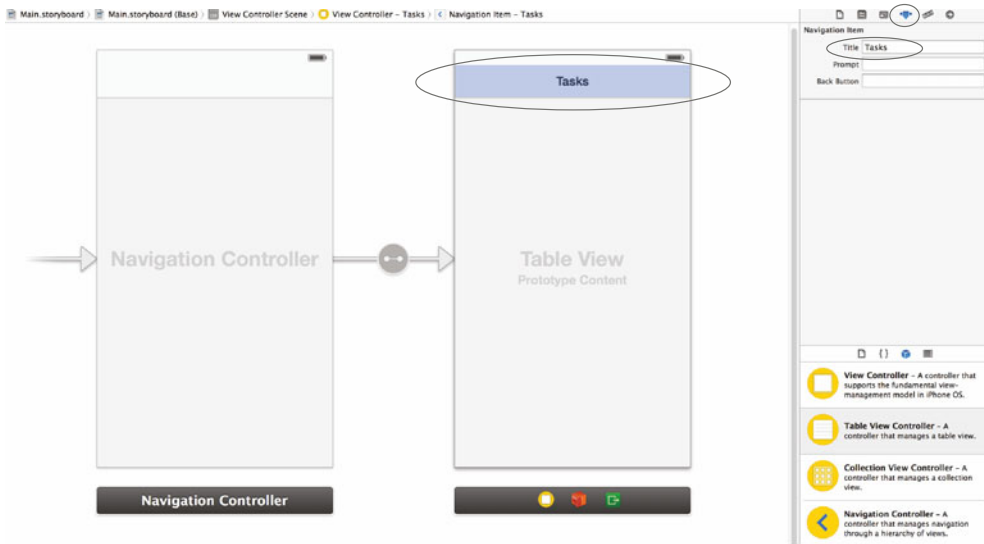


Figure 3.8 Set the title of the navigation bar to Tasks to let your users know that this view will list all of their tasks.

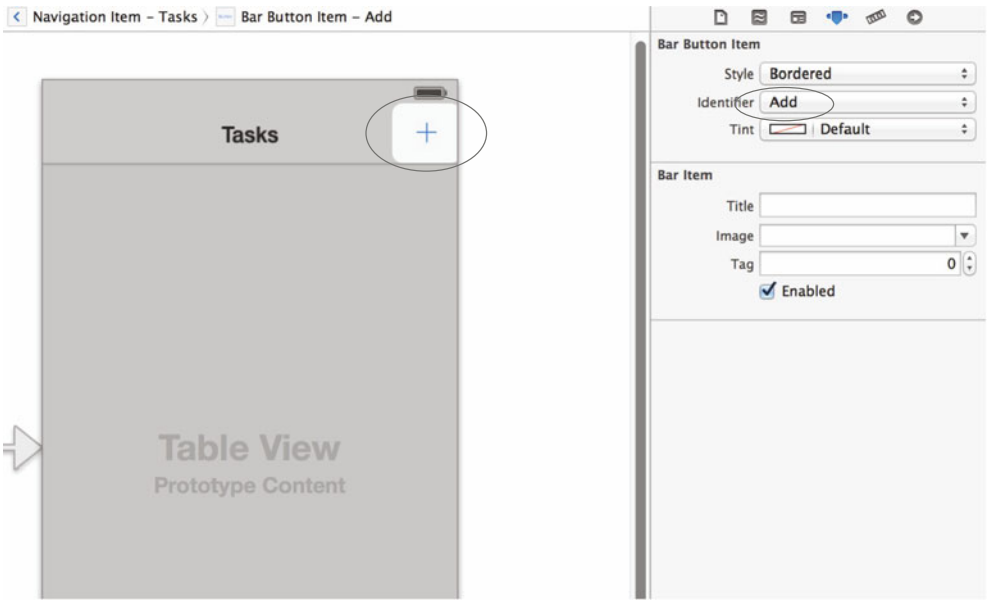


Figure 3.9 Adding a bar button item to your navigation controller that will later be used to create a new task

Now add a button to the top right of this view, within your navigation bar, which will be used to create a new task. In the Object Library on the bottom right, locate Bar Button Item and drag it to the far right side of the navigation bar. Next, go to the attributes inspector and change its identifier to Add, as shown in figure 3.9.

Next up, you're going to add two new view controllers to the project—one for creating a new task and one for viewing a task.

3.1.4 Creating and viewing a task

You'll need to add two new files to the two view controllers for creating and viewing a task. In the project navigator, right-click the Tasks group and choose New File. In the dialog that appears, choose Objective-C class as the file template, and name the first one `IANewTaskViewController`. Ensure that it's a subclass of `UIViewController`, as shown in figure 3.10.

Repeat these steps to create another view controller named `IATaskViewController`. This will serve as the view controller for an individual task. You can now add these two scenes to your storyboard.

Jump back into Main.storyboard by choosing it in the project navigator. Find View Controller in the object library and drag it to the right of the scene you've been working with for listing your tasks. Your storyboard will now contain three scenes, as shown in figure 3.11.

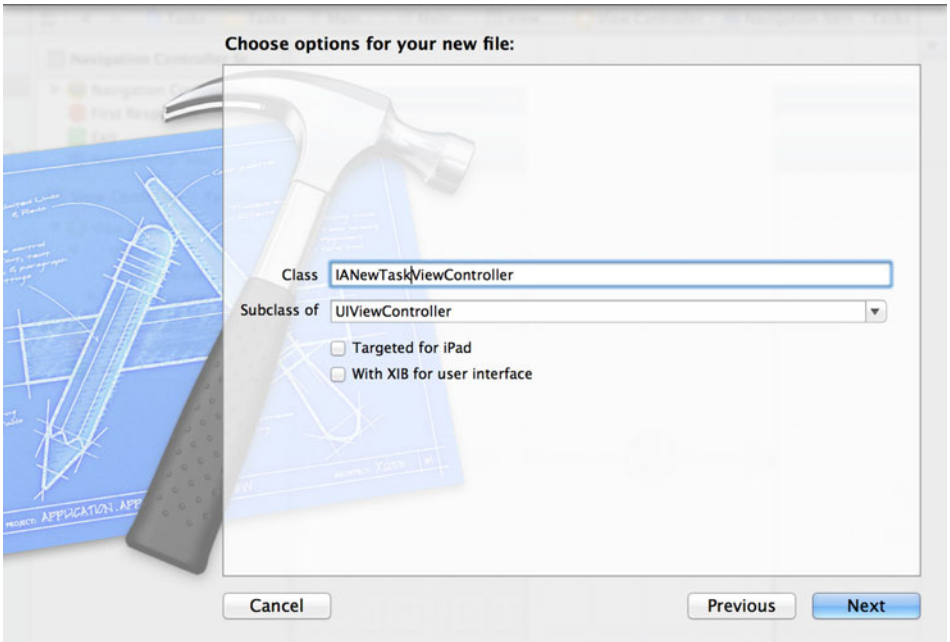


Figure 3.10 Create a new view controller called `IANewTaskViewController` as a subclass of `UIViewController`.

Currently, the view controller isn't tied to one of the classes that you've just created. Open the identity inspector (View > Utilities > Show Identity Inspector) and set its class to `IANewTaskViewController`. You can now fill out the view with a text field and a button.

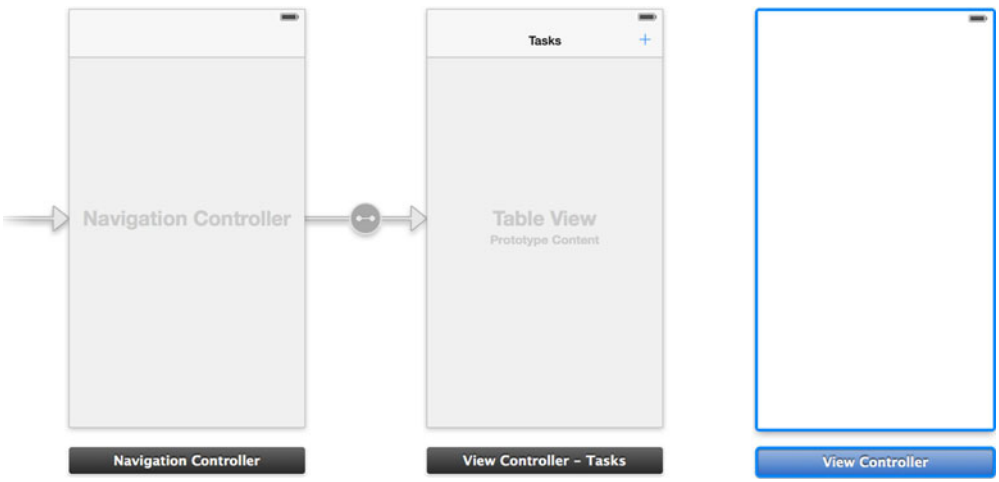


Figure 3.11 The storyboard after dragging in a new view controller

Starting with the text field, find a text field in the Object Library and drag it into your view. Line it up so that it's close to the top and almost the whole width of the view, like in figure 3.12.

Add a button underneath the text field that will allow users to save a new task. In the Object Library find Button and then drag it into your view so that it's underneath the text field. Adjust the width of the button so that it matches the width of the text field. Next, go into the attributes inspector and change the title to Save. Your scene should now look like figure 3.13.

Open the assistant editor and create a new outlet connection for the text field called `textField`. Next, create an action connection for your button called `saveTask`. This will be triggered for the Touch Up Inside event, as shown in figure 3.14.

This action will be triggered when someone taps the Save button. While you still have IANewTaskViewController.h open in the assistant editor, you need to add another

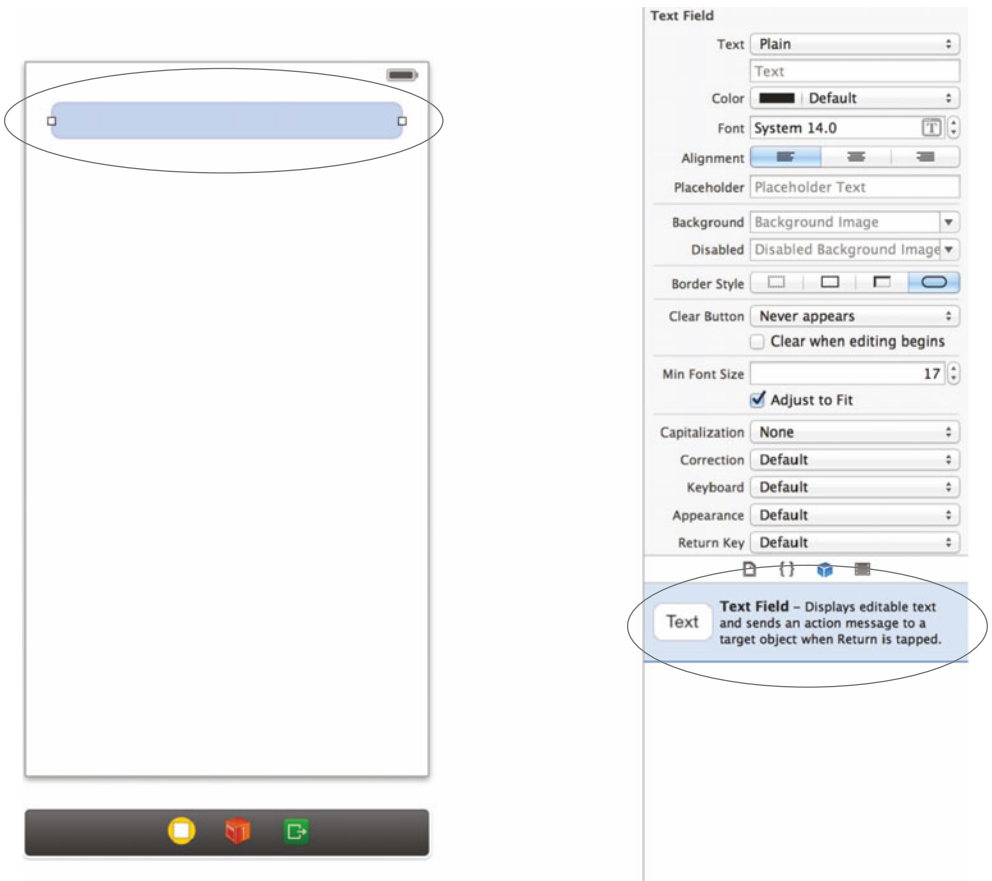


Figure 3.12 Add a text field to your view that's responsible for creating a new task. Position it toward the top and make its width almost fill up the view.

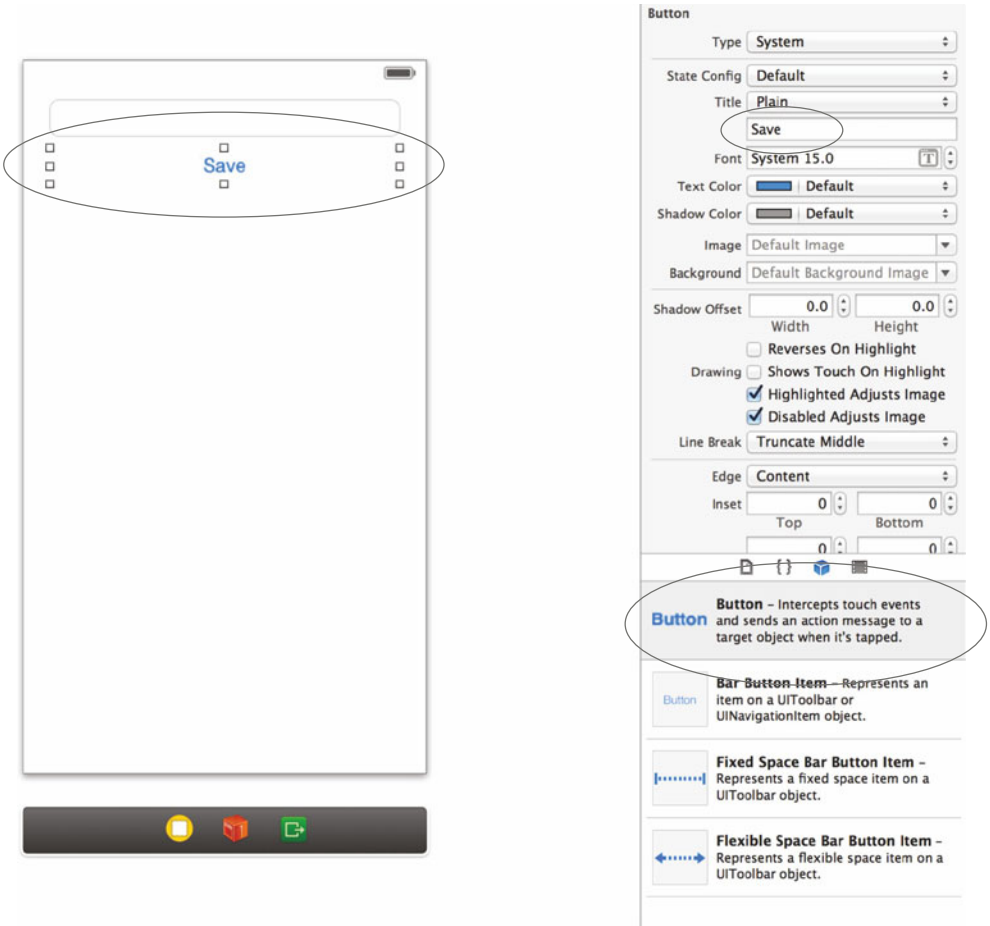


Figure 3.13 Adding a button underneath the text field and setting its title to Save

property. This property will be used to maintain a reference to the controller that displays and holds your tasks. Add the following code:

```
@property (weak, nonatomic) id delegate;
```

After adding this property, close the assistant editor and reopen Main.storyboard. You have one more scene to add that will be used to display a task that you've selected from the tasks list. This scene will contain two controls: a text view to display the task and a button to mark it as completed. First, add a new view controller into your storyboard and set its class to `IATaskViewController`. Grab a label from the Object Library and drag it to the view in the same position as the text field in your previous scene. Next, add a button underneath it titled Completed, as shown in figure 3.15.

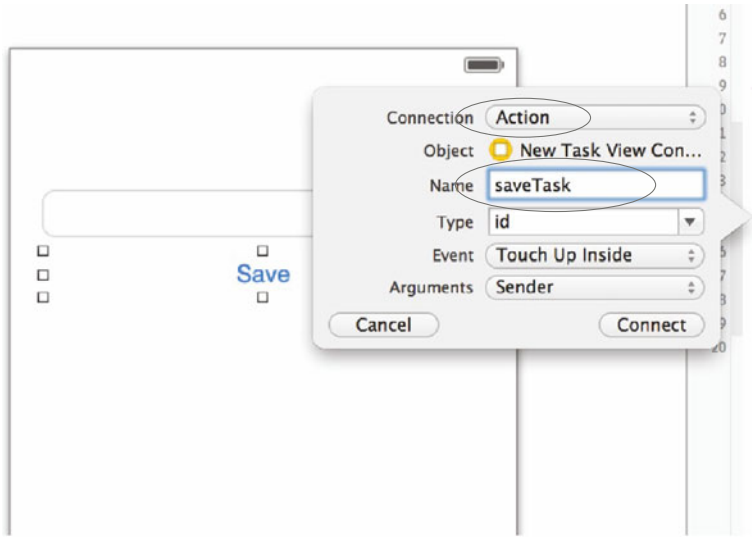


Figure 3.14 Create an action by dragging into the assistant editor, just as if you were creating an outlet, but change the connection type to Action.

Open the assistant editor and create an outlet for the label called `taskLabel`. Next, create an action for your button named `completeTask`. You'll also need to add two custom properties. Add the following code underneath the properties you created for `taskLabel` and `completeTask`:

```
@property (weak, nonatomic) NSString *task;
@property (weak, nonatomic) id delegate;
```

Once completed, IATaskViewController.h should look like figure 3.16.

The `task` property will be used to reference the task that you need to display and the `delegate` property to reference the controller that holds your tasks.

3.1.5 Connecting your views within the storyboard

Jump back into your storyboard and select the + button that you added to the tasks list scene. In the same way that you created outlets, click the button with your mouse, hold down Control on your keyboard, and then drag a connection to the new task scene. Once you let go, you'll see a popover that asks you what type of segue you want to create, as shown in figure 3.17.

Choose modal as the type of segue you want to create. This segue will open the new task scene when you tap the + bar button. Create another segue by Control-dragging from your scene (the Tasks View Controller) to the scene you've created for viewing an individual task. Instead of choosing modal, choose push. You want this to be triggered when someone clicks a row within the table view. To do this

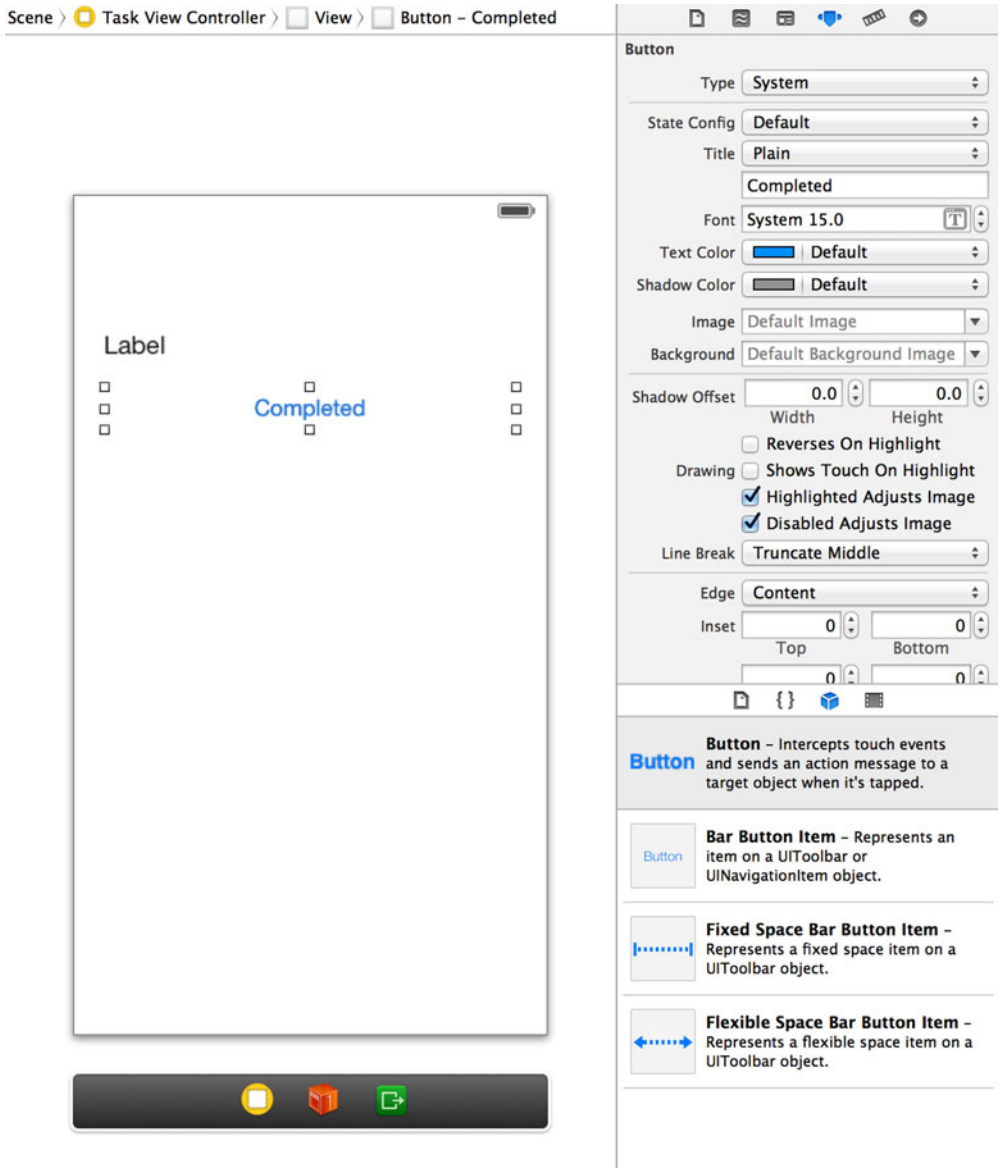


Figure 3.15 Label and button added to a new scene to display each task and mark as completed

you'll need some way to identify this segue and give it a name. Click the arrow between the two scenes, and go to the attributes inspector. Here you can set the identifier to `taskSegue`.

```
Tasks > Tasks > IATaskViewController.h > @interface IATaskViewController
1  //
2  //  IATaskViewController.h
3  //  Tasks
4  //
5  //  Created by Brendan G. Lim
6  //  Copyright (c) 2013 iOS in Action. All rights reserved.
7  //
8
9  #import <UIKit/UIKit.h>
10
11 @interface IATaskViewController : UIViewController
12
13 @property (weak, nonatomic) IBOutlet UILabel *taskLabel;
14 @property (assign, nonatomic) NSString *task;
15 @property (assign, nonatomic) id delegate;
16
17 - (IBAction)completeTask:(id)sender;
18
19 @end
20
```

Figure 3.16 The interface for `IATaskViewController` with properties and actions defined

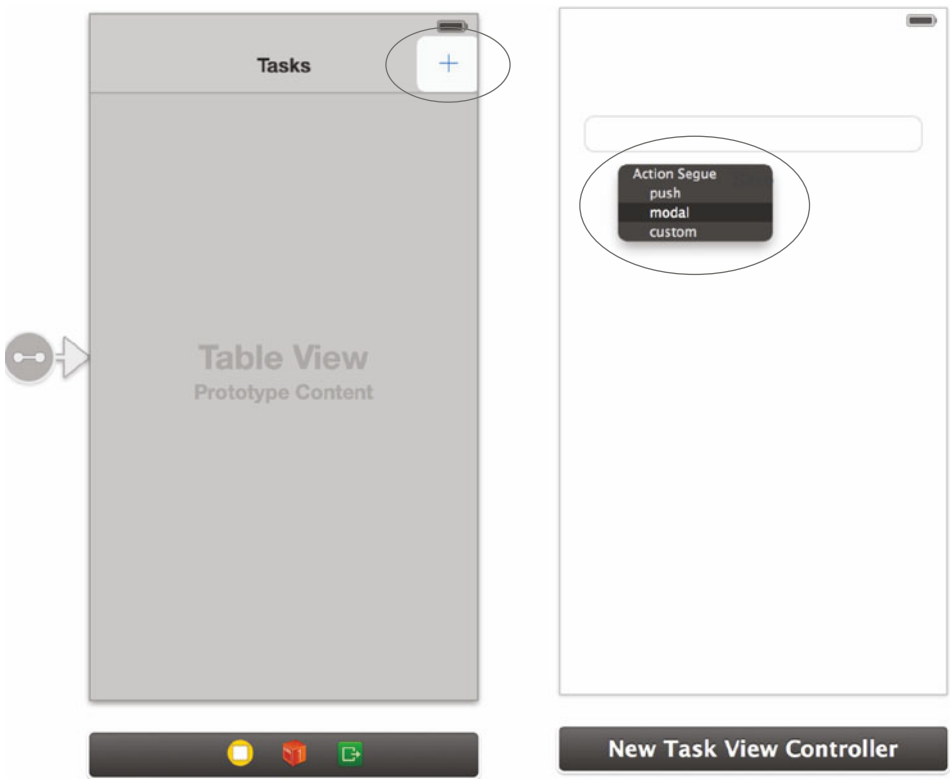


Figure 3.17 Choose a modal segue when creating a segue from the button in the navigation bar to the new task scene.

Next, jump into IAViewController.m and add the two methods shown in the following listing.

Listing 3.2 Performing a segue when a row is selected

```
-(void) tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    [self performSegueWithIdentifier:@"taskSegue"
➥ sender:self.tasks[indexPath.row]];
}

- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    UIViewController *destination = segue.destinationViewController;

    if ([segue.identifier isEqualToString:@"taskSegue"])
        [destination setValue:sender forKeyPath:@"task"];
    else
        destination = [segue.destinationViewController
➥ topViewController];

    [destination setValue:self forKeyPath:@"delegate"];
}
```

Jump back into your storyboard and select the new task view controller. Go to the application menu and choose Editor > Embed In > Navigation Controller to embed it within a `UINavigationController`. The view will now have inherited the navigation bar. Drag a bar button item from the Object Library to the top left of the navigation bar and change the title to Close. Also set the title of the navigation bar to New Task, as shown in figure 3.18.

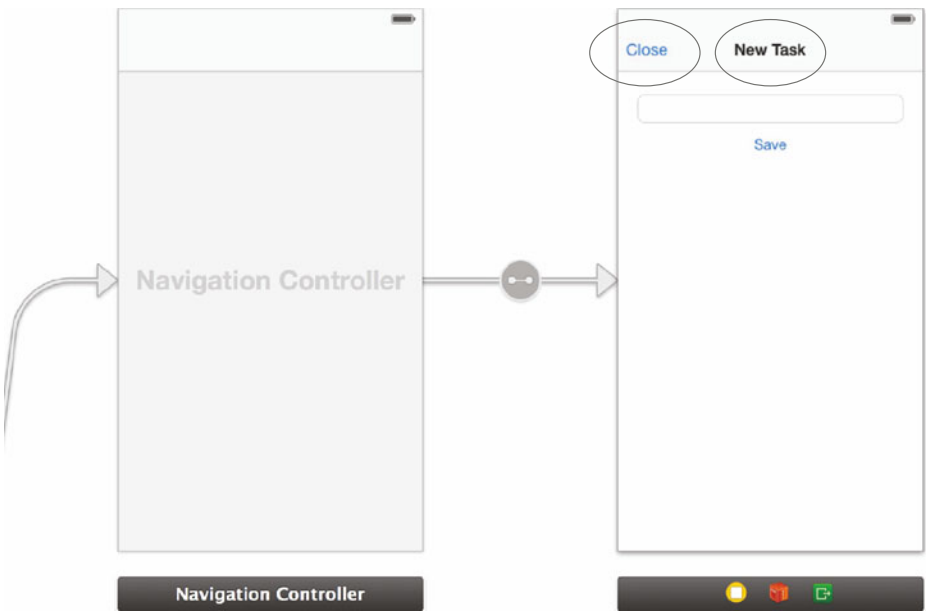


Figure 3.18 Adding a Close button and title to the New Task scene's navigation bar

The Close button that you've added isn't hooked up yet. Open the assistant editor and add an action called `close`. Go into IANewTaskViewController.m and implement the `close:` and `saveTask:` actions that you've created.

First, you need to import the interface from `IAViewController` by adding the following `import` statement to the top of your class. This is a very important step, and without doing this you won't be able to reference it within one of the methods you're going to create.

```
#import "IAViewController.h"
```

Next, add the code from the following listing.

Listing 3.3 Closing or saving a new task

```
- (IBAction)close:(id)sender
{
    [self dismissViewControllerAnimated:YES completion:nil];
}

- (IBAction)saveTask:(id)sender
{
    if ([self.textField.text length] == 0)
        return;

    IAViewController *tasksListView =
➥ (IAViewController *)self.delegate;
    [tasksListView.tasks addObject:self.textField.text];
    [self close:sender];
}
```

Next, open IAViewController.m from the project navigator. You need to make sure that your table view reloads with the newly added task when the view appears. You'll do this by making a little change to the `viewWillAppear:` method.

```
- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
    [self.tableView reloadData];
}
```

This will cause your table view to reload its data once the view appears.

The last thing you need to do is hook up the view that loads when you click a task within the tasks list. Jump into IATaskViewController.m by finding it in the project navigator. Then add the following import to the top of the class:

```
#import "IAViewController.h"
```

Next, replace the code in `viewDidLoad` with the following code:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.taskLabel.text = self.task;
}
```

This will set the label's text to the appropriate task that you need to display. The action to complete a task also hasn't been hooked up yet. You'll need to add the `completeTask:` action to the following code to be able to remove the task from the tasks list:

```
- (IBAction)completeTask:(id)sender
{
    IAViewController *tasksListView = self.delegate;
    [tasksListView.tasks removeObject:self.task];
    [self.navigationController popViewControllerAnimated:YES];
}
```

First, you get a reference to the tasks list view, and then you remove the task from its `tasks` array. Finally, you call `popViewControllerAnimatedYes:` on the navigation controller to go back to the previous view.

Go ahead and run the application, and try to add a task and complete it. Once a task has been marked as completed, it should be removed from the tasks list. Take a look at what we've created together in figure 3.19.

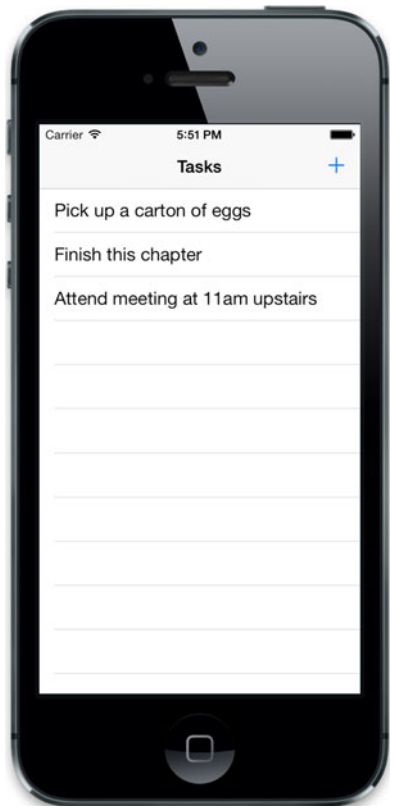


Figure 3.19 The finished Tasks application with a few tasks displayed

Great job! You've just created an application to manage your tasks using storyboards. You've created this application using things you've never used before. You've been using Xcode's interface editor this whole time when creating your views. Let's identify the specific different sections you've come across along the way.

3.2 Exploring Xcode's interface editor

We've spent more time in Xcode's interface editor than its code editor in this chapter. The interface editor has allowed you to make many changes to specific views in the Tasks app without having to dive into the code. This just shows you how powerful a tool it is, but it also means that you need to understand what the different sections are used for. Although you've already used many of these sections to create your app, let's take a closer look at each of them again.

3.2.1 Overview of Xcode's interface editor

The interface editor has a few distinct sections. They can be used for viewing the hierarchy of objects within a single view, adjusting properties on a single view, or for adding

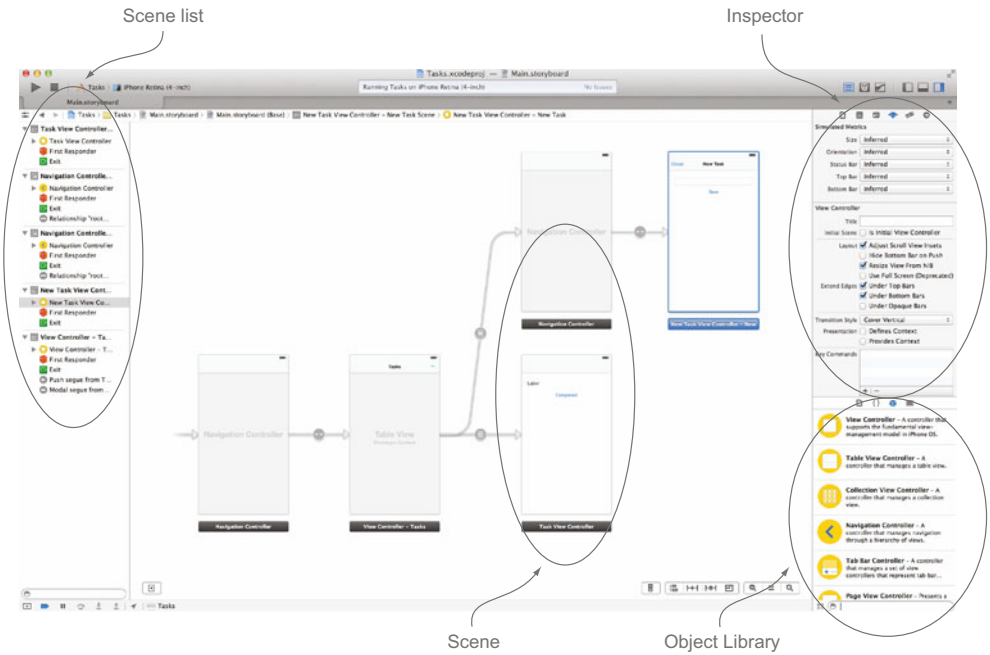


Figure 3.20 The interface editor within Xcode editing a storyboard. All scenes for an app are listed on the left, with an overview of all scenes in the middle editor area. The Object Library is on the bottom right below the attribute inspector.

new objects to an interface. Take a look at figure 3.20 to see an overview of the interface editor when working with storyboards.

The interface editor has a few distinct sections. On the left side is the scene list. Each view controller is referred to as a view controller scene in a storyboard. All of the scenes within your storyboard are listed here.

The Object Library is located on the bottom right of the window in the utility area. The Object Library holds the different types of view controllers or UI controls that you can drag into the editor area. You've used this to find text fields, buttons, table views, and anything else you need when adding something new to your interface. The section right above it is the inspector.

The inspector is used to edit many different types of attributes of a particular selected object. There are many different sections within the inspector that can be used to adjust different types of properties of an object. You've used it to adjust titles, sizes, class representations, and the like. We'll take a closer look at the attributes inspector by identifying its various sections.

3.2.2 The inspector sections

We'll explore the inspector area because you used many different sections of it while creating the Tasks app. This is where you did most of the customization for your interface. Table 3.1 lists the six buttons on the top right of the window within the inspector.

Table 3.1 Sections of the attributes inspector

Section	Description
File helper	Provides information about the currently selected file in the project navigator
Quick help	Provides documentation for the currently selected object in the interface
Identity	Lets you set custom classes and identifiers to represent the selected object
Attributes	Allows you to adjust specific attributes for the selected object
Size	Lets you adjust framing and constraints for the selected object
Connections	Lets you create connections to outlets and actions for the selected object

The first tab in the inspector, file helper, shows you all the file details related to this one view. The second tab, quick help, shows you quick help documentation related to view controllers. The content that is shown here will change depending on what type of object is currently selected. The third tab within the inspector, also known as the identity inspector, lets you see and change the class that represents the view controller for this particular scene. This is shown in figure 3.21.

This tab within the inspector is particularly important because you'll be using it whenever you want to point a UI control to a custom class you've created within your project. For instance, when you added the two view controllers for creating and viewing a new task, you used this section within the inspector to set the class to `IATaskViewController` and `IANewTaskViewController`.

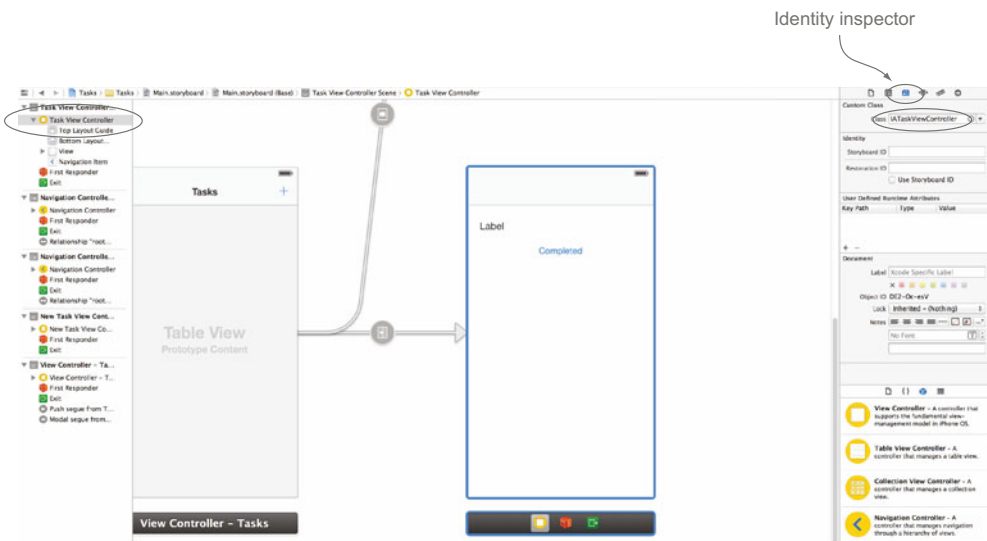


Figure 3.21 The third tab within the inspector, known as the identity inspector, allows you to change the class for the selected object. For view controllers in storyboards, it also allows you to specify a specific storyboard identifier.

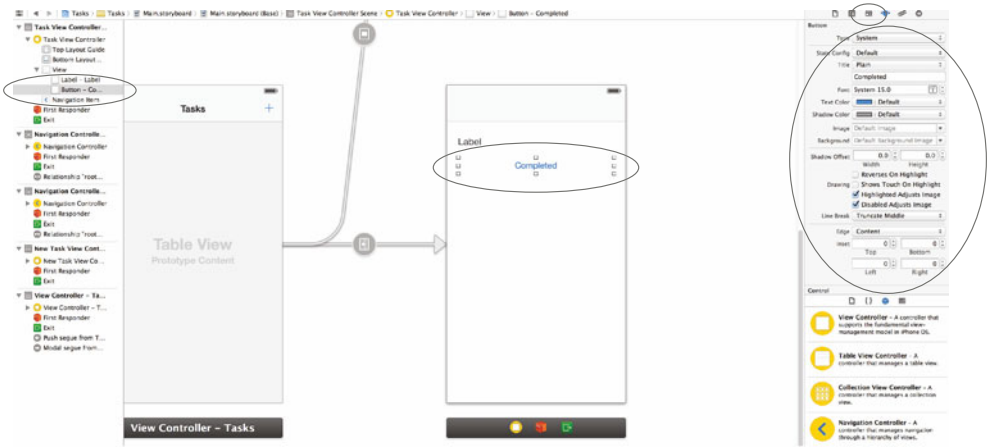


Figure 3.22 Clicking any object will show its relevant attributes within the attributes inspector.

Below the Custom Class section you'll see the Identity settings. These settings allow you to specify a unique ID for this particular view controller that you can later use to retrieve it from the storyboard.

The fourth tab, the attributes inspector, allows you to change any attribute on the currently selected object. The content within this tab changes depending on the object you have selected. You commonly use this section to change titles of buttons and other visible properties on an object. With a `UIButton` selected in figure 3.22, you can see the different editable attributes within the attributes inspector.

The next section is the size inspector. Within the size inspector you can modify all aspects of a view's framing, as shown in figure 3.23.

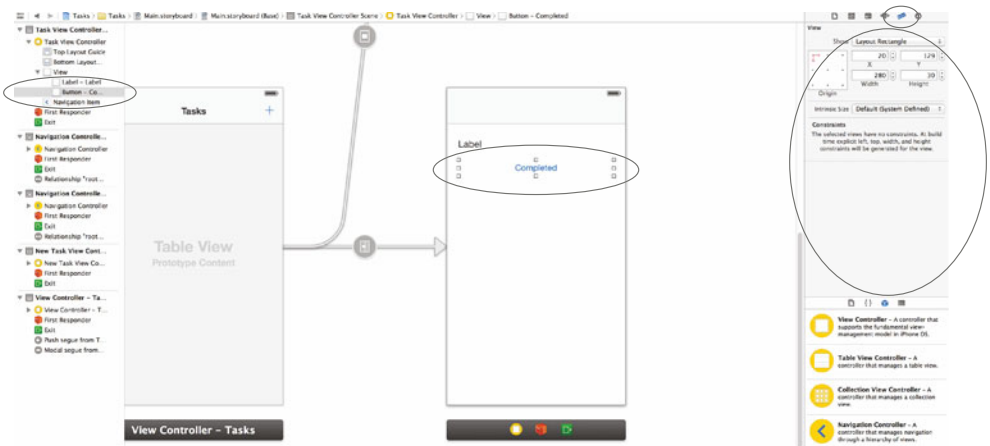


Figure 3.23 The size inspector allows you to modify anything related to a view's framing as well as its constraints.

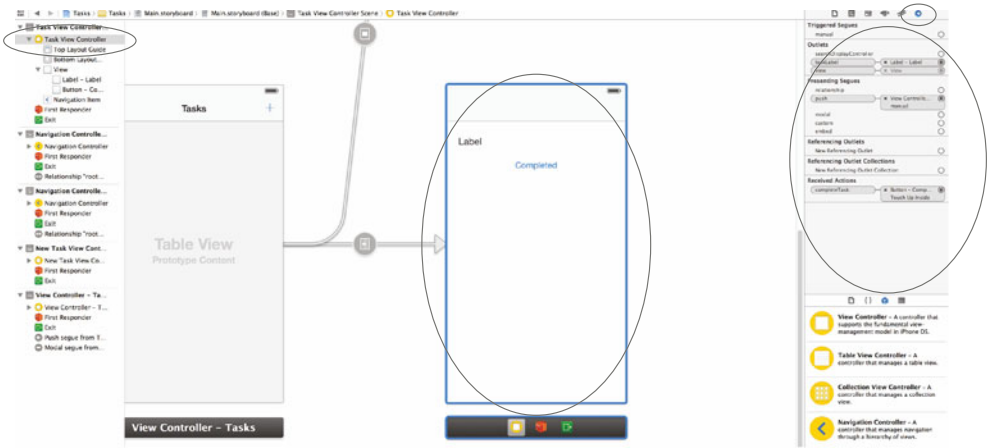


Figure 3.24 The connections inspector allows you to connect views, objects, and actions to the code you've written in your controllers.

Below that is a section called Constraints, which you used to set the auto-layout constraints.

The last tab is the connections inspector section, as shown in figure 3.24.

With certain views or objects that you've created in your controller, you needed to make connections to allow for your code to communicate with your interface. In figure 3.24 you can see the outlet connections we made for the `tableView` property on the view controller as well as the table view's `delegate` and `dataSource` properties.

You used these interface tools to help create the storyboard for your Tasks application. What exactly is storyboarding, though? How does it help you when creating applications, and how do scenes interact with one another?

3.3 Using storyboards to manage your views

You made good use of storyboarding with the Tasks application. You added multiple scenes and connected them with segues. While doing so you worked with only one file because everything was contained within one storyboard.

3.3.1 How does storyboarding benefit you?

By definition, a storyboard is a series of panels or sketches that are used to outline a scene or sequence of actions. A common use of storyboarding is to plan shots for a film. In software, storyboarding is also a very common practice. When designing an application you normally don't dive headfirst and start coding without any idea of what you're about to create. Even if you don't do any formalized planning with paper and pen, you have an idea in your mind of what you want to create and the flow of events to proceed from one action to the next. Take a look at figure 3.25 to see some of the planning we've done at my company for our iPhone application.

Figure 3.25 Real-world planning and storyboarding used to create an iPhone application

Before storyboarding, each of the views for these controllers would have been done in separate interface files. There was no visual representation of how they interact with one another, nor was there a way to visualize the whole application. With storyboarding you can alleviate all of this by being able to edit all of the views for a particular flow within one file and visualize how they interact with one another. Figure 3.26 shows the view of the storyboard we created for the Tasks app.

By looking at the storyboard you can have a clear understanding of the application without even running it. Here you can see the different views we created for the application. You can see how each view is connected to each other. Each view controller we added to the storyboard is represented as a scene.

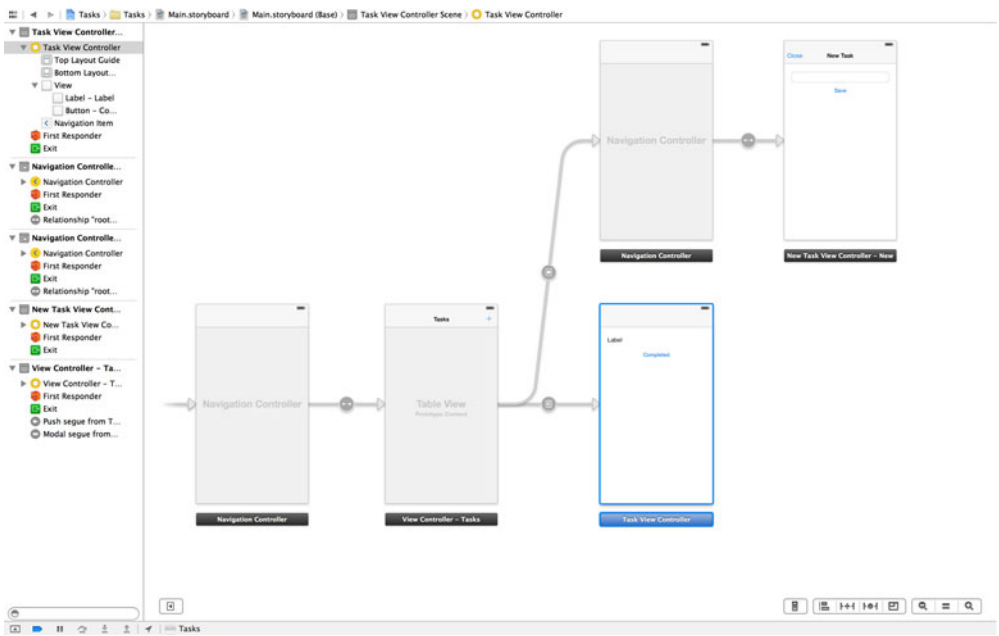


Figure 3.26 The storyboard we created for Tasks shows you how the different scenes are connected to one another.

3.3.2 Scenes within storyboards

Scenes in a storyboard represent content shown within one screen in your application. A scene involves a view controller and the views that make up its interface. There's not much involved with creating a new scene. When you dragged a view controller object onto the storyboard, you were creating a new scene.

There's also no limit as to how many scenes you can have within one storyboard. You used a storyboard for your Hello Time app, and it consisted of only one scene. Your Tasks application has three distinct scenes: one for viewing a list of tasks, one for creating a new task, and another for viewing a task. If you have many scenes that are part of a distinctly different part of your application, you could also separate them out into another storyboard file. Separate storyboards can be loaded programmatically. For instance, if you had a storyboard file named OtherStoryboard.storyboard, you could load it by using the following command:

```
UIStoryboard *newStoryboard = [UIStoryboard
➥ storyboardWithName:@"OtherStoryboard" bundle:nil];
```

With a reference to the OtherStoryboard, you can then retrieve a scene's view controller by referencing its scene identifier using the `instantiateViewControllerWithIdentifier:` method. The scene's identifier is its storyboard ID, which is set within the identity inspector, as shown in figure 3.27.

Scenes within a storyboard are connected by using segues. We'll explore transitioning between different scenes via segues.

3.3.3 Transitioning between scenes with segues

Segues allow you to easily transition from scene to scene. Segues are represented by the `UIStoryboardSegue` class. You created two segues within your Tasks application. One of them was triggered automatically to display the scene to create a new task. The

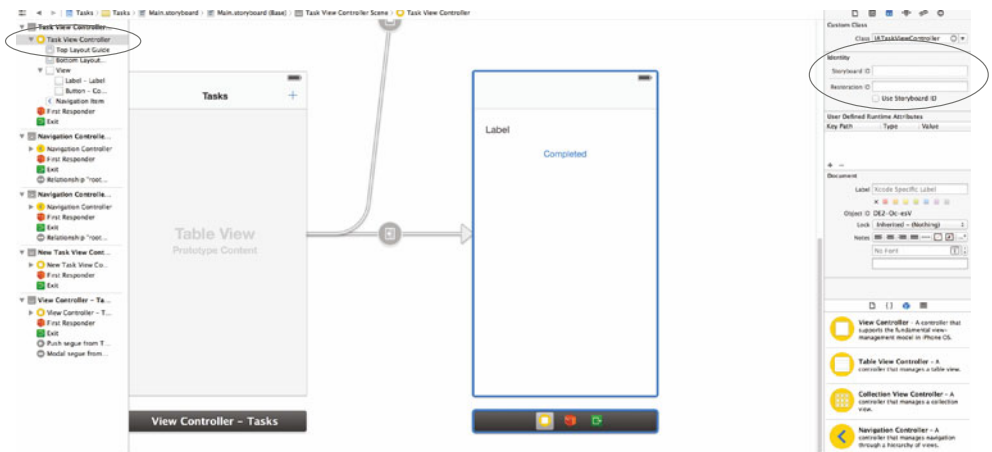


Figure 3.27 You can set the storyboard ID for a particular scene in the identity inspector.

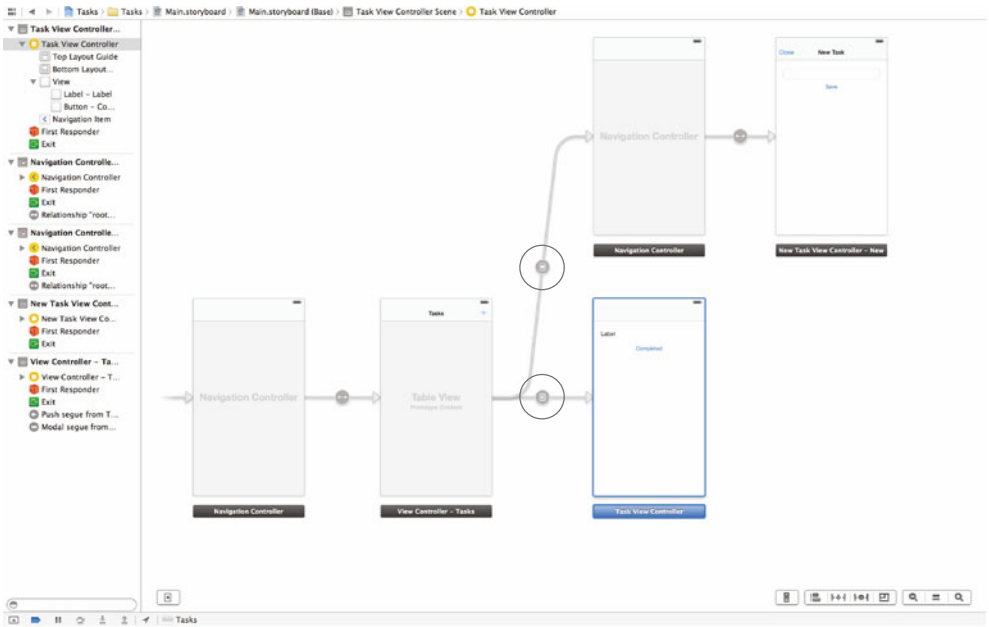


Figure 3.28 The two segues that we created to display two separate scenes within the Tasks application

segue you created for displaying the scene for viewing an individual task was triggered programmatically. Segues can be identified within a storyboard, as shown in figure 3.28.

There are a few built-in segues that you can choose from. When working with an iPhone application you can choose from *push*, *modal*, or *custom*. For the iPad you're given an extra choice of using a *popover* segue.

Modal segues slide a scene from the bottom to the top and appear to be on top of the parent scene. You used a modal segue when showing your scene to create a new task. *Push* segues are used to transition the new scene in from the right. In a push segue the original scene that prompted the segue then goes away by sliding out to the left. You used this type of segue when tapping on a task from the tasks list to show the task view.

Popover segues overlay only part of the parent view within a popover.

You can create custom segues by creating your own `UIStoryboardSegue` class. This gives you full control over the transition and appearance of the new scene.

You create segues in your storyboard by using your mouse to drag a connection from one scene or an actionable object to another, as shown in figure 3.29.

When creating a segue from an actionable object like a button, the segue is automatically triggered when the button is tapped with the Touch Up Inside event.

Segues can be triggered programmatically, as you did with the segue from your tasks list to the individual task scene. You had to set an identifier for it, `taskSegue`, so that you could reference it in your code by calling `performSegueWithIdentifier:`

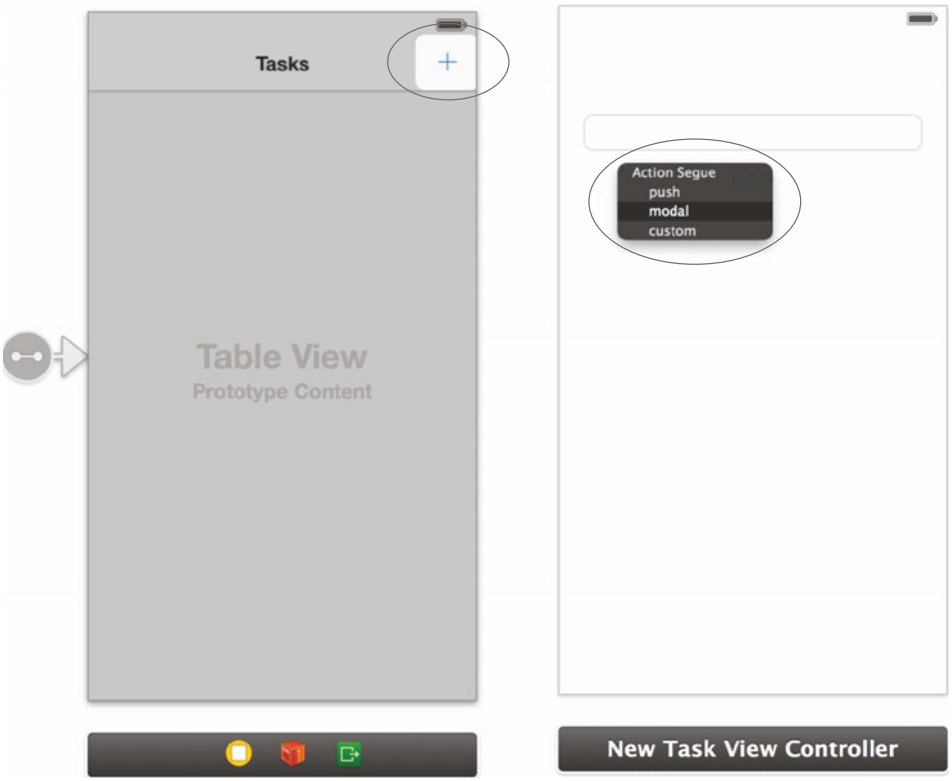


Figure 3.29 You create a segue by dragging from one scene or actionable object to another scene.

from the originating view controller. You chose to trigger it when a row in your table view was selected.

```
-(void) tableView:(UITableView *)tableView
➥ didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    [self performSegueWithIdentifier:@"taskSegue"
➥ sender:self.tasks[indexPath.row]];
}
```

3.3.4 Passing data between view controllers with segues

Segues also allow you to pass data to the next view controller before completing the transition. You do this by overriding the `prepareForSegue:sender:` method from the originating view controller, as shown here:

Get reference to the view controller you're about to pass data to.

Check to see if you're preparing for the correct segue.

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    UIViewController *destination = segue.destinationViewController;
    if ([segue.identifier isEqualToString:@"taskSegue"])
```

```
        [destination setValue:sender forKeyPath:@"task"];
    else
        destination = [segue.destinationViewController
➥ topViewController];

    [destination setValue:self forKeyPath:@"delegate"];
}
```

Setting the sender as the `task` property

Get destination controller if not `taskSegue`.

Set your current view controller as the `delegate` property.

You first retrieve a reference to the view controller that you want to pass data to. You then set the `delegate` property to the instance of the controller you're calling this from and apply this to all segues. You then check to see if the segue identifier is equal to `"taskSegue"` because you want to pass in only the selected task for this particular segue. When you call `performSegueWithIdentifier:sender:`, you also pass in the task as the sender. You set a `task` property on `IATaskViewController` to represent an `NSString`. It's because of this that you're setting `sender` as the task key value for the destination view controller.

By understanding scenes and how to use segues to transition between them, you can utilize storyboards to build your apps more efficiently. There are a few minor downsides to storyboarding to consider, though.

3.3.5 Problems with using storyboarding

Storyboarding allows you to manage all of your scenes, connections, and segues within one file. It also gives you an overview of the flows within your application. With all of the benefits that storyboarding offers, it does come with a few downsides, though.

The biggest problem with storyboarding is that it's extremely difficult to use when working with a team that uses source code revision and management tools like Git, Subversion, or Mercurial. If you open a storyboard file within a text editor, you'll see that it's essentially an XML file that maintains all of your views, outlets, segues, positioning, and so on. When multiple teammates make changes to the same storyboard, it becomes problematic. Source code revision tools won't be able to properly merge the changes, and Xcode won't be able to open the storyboard file until the changes have been merged successfully. You'll be forced to do a deep dive into the XML to fix it yourself.

Another problem is that it forces older developers, who have been comfortable using NIBs and creating and managing views programmatically, to change their ways. People generally don't like change, and when working with a team, everybody will have to be comfortable with storyboarding because it's not a good practice to mix conventions.

Although there are problems, Apple has made it clear that storyboarding is here to stay. All of their sample applications and documentation have been updated to use storyboarding. They've also made a huge effort to support it within their own tools. Many of the problems with storyboarding will become a thing of the past as Apple figures out a way to alleviate these issues.

3.4 Summary

Working with interfaces in Xcode is made simple thanks to the tools that Apple has provided. Storyboarding allows you to streamline all of the views in your application by giving you a single file to manage and visualize how they interact with one another. Although there may be some drawbacks with storyboarding, especially when working on a team, these issues will definitely be mitigated as time goes on.

- Storyboarding allows you to streamline all of the views in your application by giving you a single file to manage and visualize how they interact with one another.
- Segues allow you to transition from one scene to another within a storyboard.
- Different types of segues can be used, depending on the way you want to present the new scene.
- Although there may be some drawbacks with storyboarding, it's better and easier in the long run to use it in your applications.
- Dragging a new view controller into your storyboard will create a new scene.
- By overriding `prepareForSegue:sender:`, you can pass data from one view controller to another when using segues.
- Xcode's interface editor has many tools that allow you to customize and connect your views to your code.
- You created a task-management application using a storyboard with multiple scenes connected by segues.

4 Using and customizing table views

This chapter covers

- Using table views and table view controllers
- Implementing prototype table view cells
- Handling selection of rows within the table view
- Utilizing the Assets Library framework to retrieve photos
- Using table views to list photo albums

It's inevitable that you'll need to organize and display something as either a list or a grid. If you think of the applications you often use, you'll realize that almost all of them have views where information is displayed to you in this manner. Our Hello Time app is one of the very few examples where you don't display anything in a list or grid format. The Photos app shows a grid of all of your photos or videos using a collection view. The Contacts app shows a list of the people in your address book using a table view. The Settings app also uses a table view but lists different options you can choose from. These can be seen in figure 4.1.

Earlier, you had a chance to use a table view when you created your Tasks application when exploring storyboarding. Now you'll be able to spend more time specifically focusing on table views. As you've realized, almost all of the apps

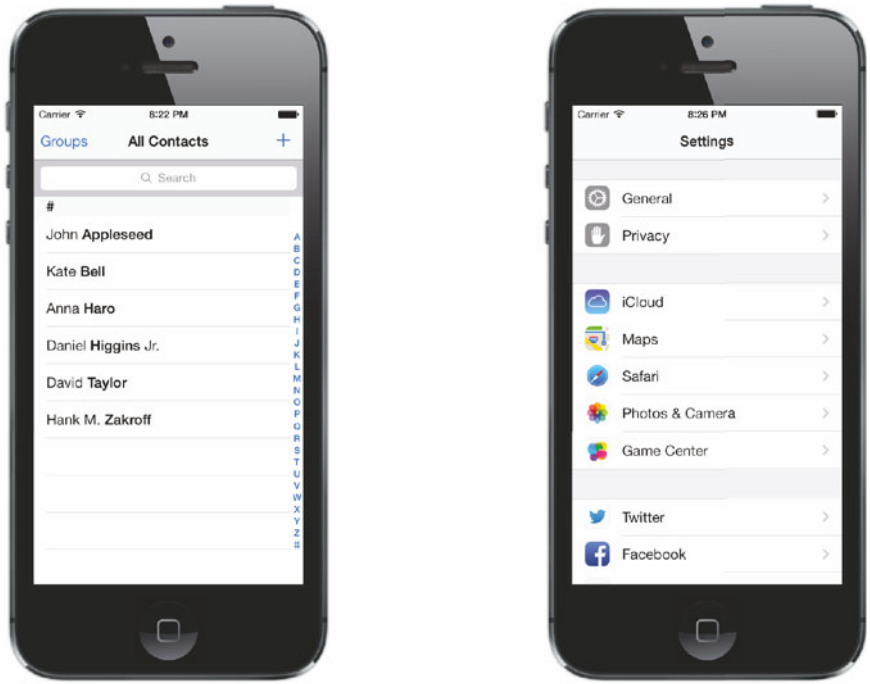


Figure 4.1 The Contacts app and Settings app both use table views to display information as a list.

you use need to display information within a list or grid. By knowing the ins and outs of table views, you'll be able to cover most of your bases. You'll learn how to use table views in this chapter by using them in a single application to display albums and photos.

Another application that does just this is Apple's own Photos application. You'll essentially be making your own version of this app to display the exact same data. While doing this you'll also use your first framework, the Assets Library. Your application will retrieve all of your albums and list them in a table view, as shown in figure 4.2.

This chapter is just the first part of your Albums application, though. You'll add even more functionality to make it feature complete in the next chapter.

4.1 Introduction to table views

This will be your second chance to use table views but your first time to really understand what's going on under the hood. Imagine writing a list of information on a whiteboard or piece of paper. Each line within a list typically has one important item. The items you choose to write are distinctly separated because they're on different lines. This is just like a grocery list or a real-world to-do list.

With the Tasks application you created a basic table view that displayed a list of tasks. For your purposes, a table view was the best way to represent your data. Imagine a task application that did not arrange your tasks into a list. Hard to picture, isn't it? On the other hand, imagine if all of the photos in your Photos app were organized into a table view with only one photo per row. You could argue that this would result in too much wasted space because you could organize your photos in a grid to maximize the space you have available.

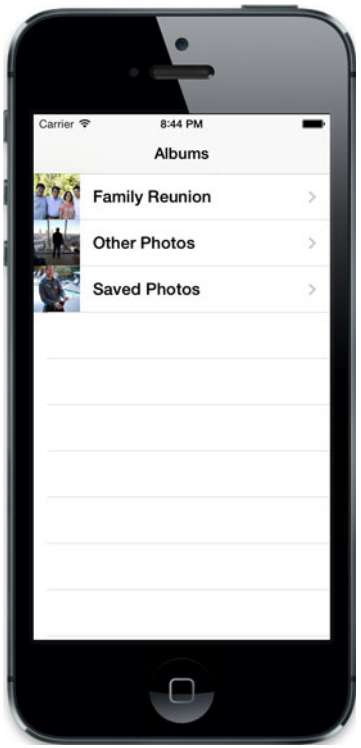


Figure 4.2 Preview of the Albums application that you'll be creating throughout this chapter

4.1.1 Anatomy of a table view

Let's look at a table view and examine its different parts. Table views are represented by the `UITableView` class. The data is displayed by using a list of rows. Each individual row is contained within its own section. A table view can also have one or more different sections. You can see an example of a table view with its different parts labeled in figure 4.3.

A `UITableViewCell` is used to represent each row within a table view. This view may contain any other views you want to use to display your data. In figure 4.3, each cell has a `UILabel` that is used to display the name of a contact. You can easily customize and create your own cells, as you'll see later in this chapter.

Figure 4.3 contains only a single section. Table view cells can belong to only one section. The most common use for sectioning is to visually separate and group rows. A great example of this is the Settings application, which uses different sections to separate the different setting options into relevant groups. Take a look at it in figure 4.4.

As you can see, the table views shown in figures 4.3 and 4.4 look fairly different from one another. This is because two different types of styles can be applied to a table view. Each of these styles changes the way that sections in your table views are visually represented. They are represented as the enumerables `UITableViewStylePlain` and `UITableViewStyleGrouped`. The Contacts application uses `UITableViewStylePlain`, whereas the Settings application uses `UITableViewStyleGrouped`. When you create your table view in Xcode, you'll have the chance to choose either style.

For your application, you'll be using a table view on your first screen to show a list of albums that you have stored on your device. You'll create the application in Xcode, add your table view, and continue from there.

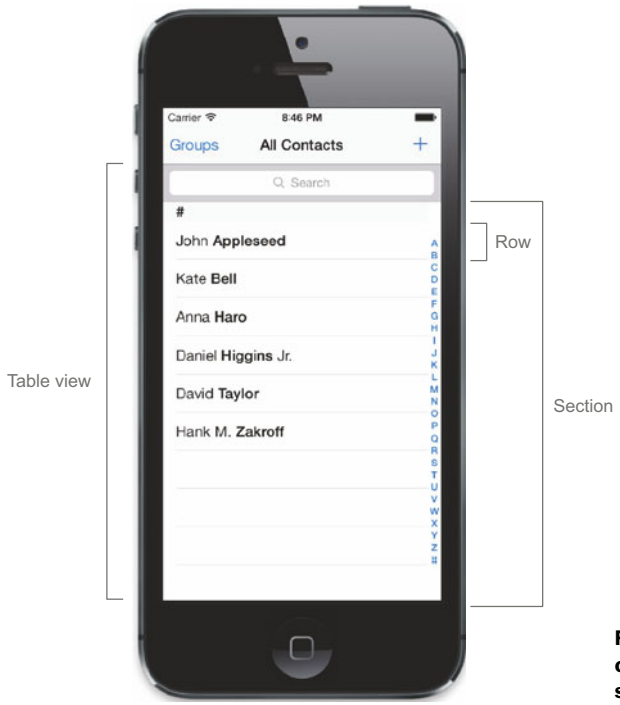


Figure 4.3 A table view with rows contained within one section as shown in the Contacts application

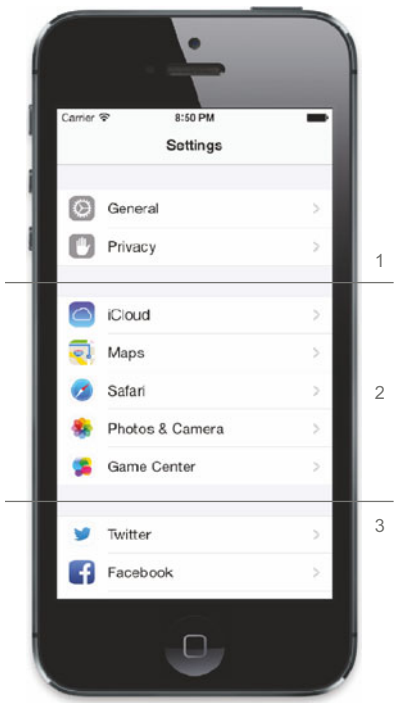


Figure 4.4 The Settings app uses sections to visually separate relevant setting options. Each of the numbers represents a separate section. This figure shows three sections separating the rows within this table view.

4.2 Using table views to display data

Your Albums app will list all of the albums you have within a table view. Once you tap a specific album, you'll see the photos contained within that album. It'll be very similar to Apple's own Photos application, except you'll be building it all by yourself. First, you'll need to do some setup for your new Albums application.

4.2.1 Setting up your Albums application

Hop into Xcode and create a new single-view application project called Albums, as shown in figure 4.5.

You use the Single View Application template because it creates a base view controller for you, as well as a storyboard for your application. You'll make some changes in the view controller that it creates by removing it and adding one based on a `UITableViewController`. Right-click both IAViewController.h and IAViewController.m in the project navigator, and then click Delete to move them to the trash. Next, click Main.storyboard to open the storyboard in your interface editor. There should be one scene in there already. Delete it by selecting the scene and clicking the Delete button. You should end up with an empty storyboard when you've finished doing this.

You can now add your new view controller that will take the spot as the first view in your storyboard. Right-click the top-level Albums group within the project navigator and choose New File. Select Objective-C Class as the new file template. Name this new class `IAAlbumsViewController` and set it as a subclass of `UITableViewController`, as shown in figure 4.6.

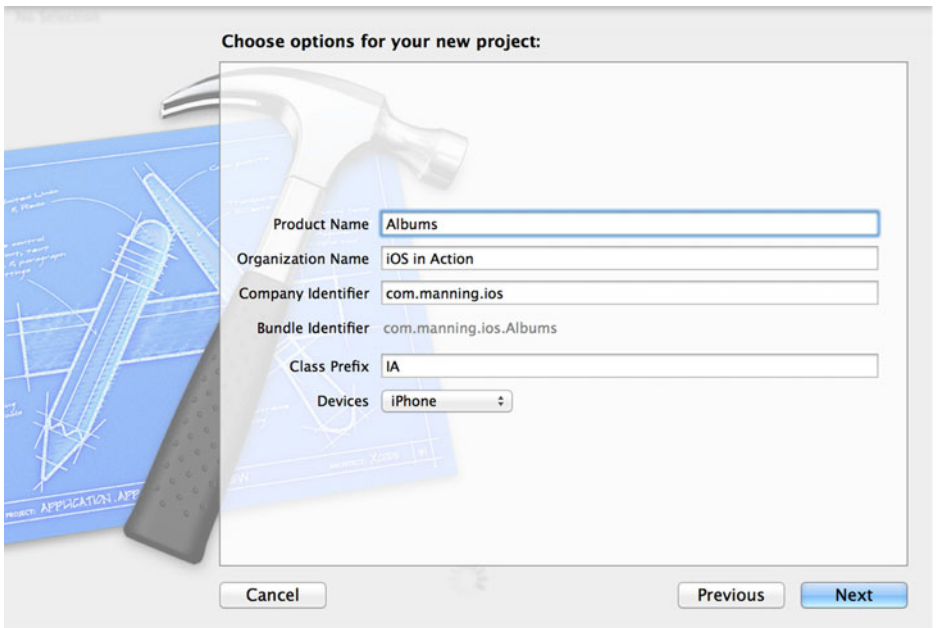


Figure 4.5 Creating a new single-view application project called Albums within Xcode

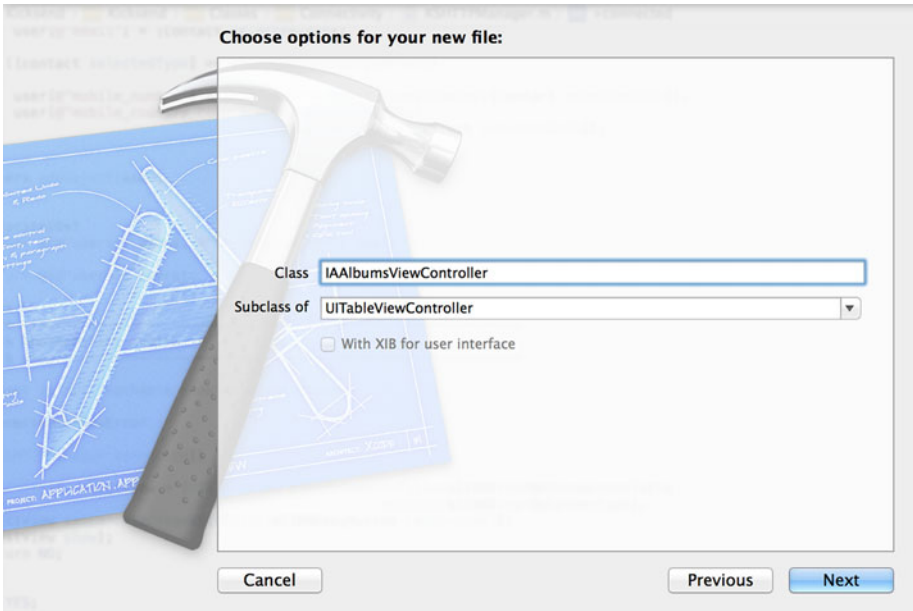


Figure 4.6 Create a new class called `IAAlbumsViewController` as a subclass of `UITableViewController`.

This will generate a new subclass of `UITableViewController` that will provide you with many autogenerated methods. You can now add this to your project's storyboard. Open Main.storyboard, find Table View Controller from the Object Library, and drag it into your storyboard. Next, while it's still selected, go to the Application menu and choose Editor > Embed In > Navigation Controller to wrap it in a navigation controller. Finally, but most important, you need to set the table view controller's class as `IAAlbumsViewController`, as shown in figure 4.7.

Also, to inform your users that what you're displaying in this scene are albums, change the title in the navigation bar of this scene to Albums. You've taken care of much of the boilerplate work for your first view. Next, you have to create a new class in preparation for accessing your photos.

You'll need to add a framework to this project that will help you access the media, or assets, stored on a device. Frameworks are compiled libraries that you can use to add functionality to your applications. By default, all new iOS apps created in Xcode include the frameworks UIKit, Foundation, and CoreGraphics. These three frameworks provide the basic functionality to create iOS applications. You're going to add a new framework that will allow you to access media assets.

The framework you'll need to add is the Assets Library framework. Go to the General tab for your target and scroll down to Linked Frameworks and Libraries, as shown in figure 4.8.

Click the + button and choose AssetsLibrary.framework. Then click the Add button, as shown in figure 4.9.

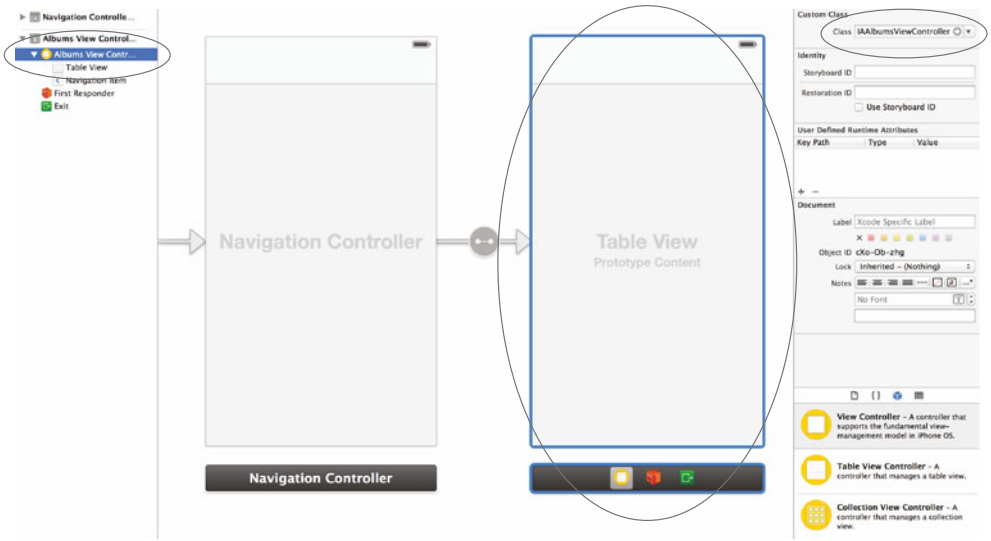


Figure 4.7 Setting the custom class for the table view controller to `IAAlbumsViewController`

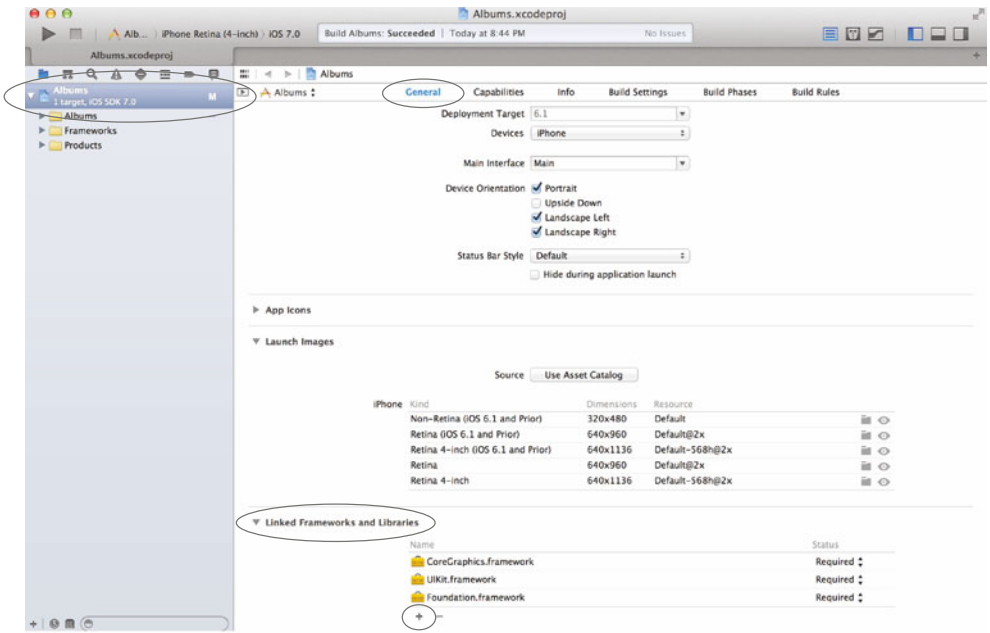


Figure 4.8 Go to the General tab of your target, scroll to Linked Frameworks and Libraries, and click the + button to add a new framework to your project.

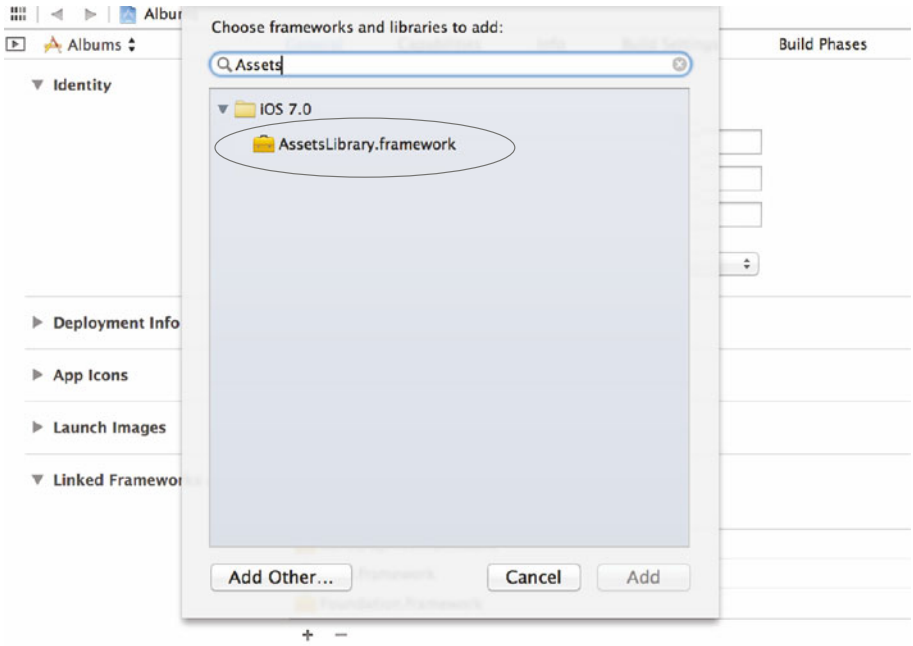


Figure 4.9 Choose Assets Library.framework and click the Add button to add it to your project.

This will give you the ability to utilize this framework in your project to access all of your photos and the albums that they belong to.

You're now going to use the assets library by creating a new class that will act as a singleton instance of an `ALAssetLibrary`. An `ALAsset` represents assets such as photos or videos. Its parent is an asset library (`ALAssetLibrary`). Each instance of an `ALAsset` will keep a reference to its parent library. If this parent goes missing or is different, the `ALAsset` becomes useless and you'll be throwing your hands in the air wondering why your app is crashing. This is why you need one `ALAssetLibrary` singleton instance throughout your application that you can always use to retrieve assets.

Right-click the Albums group in the project navigator and choose New File again. Name this file `IAAssetsLibrary` and make it a subclass of `ALAssetsLibrary`. Once the file is created, click IAAssetsLibrary.h. At the top of the class add the following `import` statement:

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

Within the interface declaration add the following, which will describe the class method you're about to add:

```
+ (IAAssetsLibrary *) defaultInstance;
```

You can immediately jump into the implementation by clicking IAAssetsLibrary.m in the project navigator, and add the following method:

```
+ (IAAssetsLibrary *) defaultInstance
{
    static IAAssetsLibrary *singleton = nil;
    static dispatch_once_t oncePredicate;
    dispatch_once(&oncePredicate, ^
    {
        singleton = [[super alloc] init];
    });

    return singleton;
}
```

This will return a singleton instance of `IAAssetsLibrary` by calling the `defaultInstance` class method you've just created. You'll be using this when you retrieve the albums or photos you want to display. First, let's look at how you can provide a table view with the information it needs by conforming to its delegate and data source protocols.

4.2.2 Providing data through a data source

To supply a table view with data, you need to conform to its `UITableViewDataSource`. These protocols have a set of required and optional methods that you need to implement within the class that supplies your table view with the data it needs. For instance, the table view will need to know how many sections there are and how many rows to display within each section. The purpose of the data source is, you guessed it, to supply your table view with data.

You're using a `UITableViewController` as the parent class of your `IAAlbumsViewController` class. By doing this, you don't need to specify anything special to let your class know that it conforms to the `UITableViewDataSource` protocol. If this wasn't the case, you'd specify that you adhered to this protocol by doing the following in your class's header file:

```
@interface ExampleViewController : UIViewController <UITableViewDataSource>
    ...
@end
```

Let's look at the important required methods in the `UITableViewDataSource` protocol, as shown in table 4.1.

Table 4.1 Required methods for the `UITableViewDataSource` protocol

Method	Description
`tableView:numberOfRowsInSection:`	Total number of rows to display within a section
`tableView:cellForRowAtIndexPath:`	Return a `UITableViewCell` for a row at a specified index path

The first method is `tableView:numberOfRowsInSection:`, which returns the total number of rows that the table view should display. In most cases, you'd store the data that you want to display within an `NSArray`. Say you had an array called `assets` and

you wanted to display it in a table view. When you specify how many rows you want to display, you return the number of items you have in your array:

```
- (NSInteger)tableView:(UITableView *)tableView
➥ numberOfRowsInSection:(NSInteger)section
{
    return [self.albums count];
}
```

By calling `[self.albums count]` you are returning the total number of items in the `assets` array. Also notice that the method passes in a specific `section` parameter. Depending on what you're showing in a table view, you can choose to display a different set of data and can perform the necessary calculations on your side to determine how many rows you should display. In the previous code example, you're assuming that there's only one section in which you want to show all items in the `assets` array.

Depending on the number of items to display, your table view will ask you for a `UITableViewCell` to display for a given index path. An index path is represented by the `NSIndexPath` class, which just contains a `section` and a `row` parameter. This brings you to the next required method, `tableView:cellForRowAtIndexPath:`. This method will be called by your table view and gives you the opportunity to provide it with a `UITableViewCell` that you want to display at that specific index path:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
➥ cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView
➥ dequeueReusableCellWithIdentifier:CellIdentifier];
    if(cell == nil)
        cell = [[UITableViewCell alloc]
➥ initWithStyle:UITableViewCellStyleDefault
➥ reuseIdentifier:CellIdentifier];

    // Customize cell

    return cell;
}
```

Within this method, you specify a cell identifier that you use to pull back a specific type of `UITableViewCell` from your table view. It's up to you to choose the cell identifier name you want to use for the cell you're displaying. You can set this within Xcode's interface tools when preparing a prototype cell for your table view. The purpose of using this cell identifier is to call the method `dequeueReusableCellWithIdentifier:forIndexPath:` on your table view. This is used for performance reasons. Instead of creating a brand-new `UITableViewCell` to display each time, you can pull back one that has already been created and change it to display whatever content is relevant at the specified index path.

You'll put all of this together by updating your Albums app to display all of its albums within its table view. Before you can display albums, you first need to retrieve

them. Hop back into Xcode and open IAAlbumsViewController.h. You're going to add a property that will store all of your albums:

```
@property(nonatomic, strong) NSMutableArray *albums;
```

Next, open IAAlbumsViewController.m and add the following `import` statement to import the `IAAssetsLibrary` class:

```
#import "IAAssetsLibrary.h"
```

Now you can add a method to retrieve all of the albums from the assets library. Add the `loadAlbums` method, as shown in the following listing.

Listing 4.1 Load all albums from the assets library

```
- (void)loadAlbums
{
    IAAssetsLibrary *library = [IAAssetsLibrary defaultInstance];
    [library enumerateGroupsWithTypes:ALAssetsGroupAll
➥ usingBlock:^(ALAssetsGroup *group, BOOL *stop)
    {
        if(group)
        {
            [self.albums addObject:group];
        }
        else
        {
            [self.tableView
➥ performSelectorOnMainThread:@selector(reloadData)
                                                withObject:nil
                                             waitUntilDone:YES];
    }
                         failureBlock:^(NSError *error)
    {
        NSLog(@"Problem loading albums: %@", error);
    }];
}
```

The `loadAlbums` method will iterate through different groups of type `ALAssetsGroupAll`. This retrieves all of the albums in the assets library. You're adding each album to your assets array and, once finished, calling `reloadData` to tell your table view that it needs to load the data and display it.

Once you've added this method, you need to initialize the `albums` array and call `loadAlbums` after your view has finished loading. You can do this by appending the following two lines to the bottom of the `viewDidLoad` method:

```
self.albums = [NSMutableArray array];
[self loadAlbums];
```

Take a look at the code added to the `IAAlbumsViewController` class, as shown in figure 4.10.

You can now implement two data source methods for your table view that will tell it just what to display when `reloadData` is called. First, you'll tell your table view how

Figure 4.10 Initializing your `albums` array and setting up and triggering your `loadAlbums` method

many rows it needs to display by implementing `tableView:numberOfRowsInSection:` and having it return the number of albums within the `albums` array property:

```
- (NSInteger)tableView:(UITableView *)tableView
➥ numberOfRowsInSection:(NSInteger)section
{
    return [self.albums count];
}
```

You're returning the number of items within the `albums` array by calling the `NSMutableArray count` method. Next, you'll implement `tableView:cellForRowAtIndexPath:` to return a `UITableViewCell` with the name of each album in the `albums` array, as shown in the next listing.

Listing 4.2 Create a `UITableViewCell` for each album

```
- (UITableViewCell *)tableView:(UITableView *)tableView
➥ cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView
➥ dequeueReusableCellWithIdentifier:CellIdentifier];
    if(cell == nil)
        cell = [[UITableViewCell alloc]
➥ initWithStyle:UITableViewCellStyleDefault
➥ reuseIdentifier:CellIdentifier];

    ALAssetsGroup *group = self.albums[indexPath.row];
    cell.textLabel.text = [group
➥ valueForProperty:ALAssetsGroupPropertyName];
```

Retrieve `ALAssetsGroup` from `albums` array using the row of `indexPath`.

Set the text label on the cell to the name of the album.

```
    return cell;
}
```

You're using the `row` property on `indexPath`, which is passed into this method, to retrieve the appropriate `ALAssetsGroup` from your `albums` array for this specific row. You're then setting the cell's `textLabel` to the name of your album.

By looking at figure 4.11 you can see these two data source methods added to `IAAlbumsViewController`.

You can now try to run the application to see what you've just accomplished. If you don't have any albums on your device, you can create them by going to Apple's own Photos application and clicking the + button, as shown in figure 4.12.

With a few albums in place, your Albums application will look like what's shown in figure 4.13.

For your table view cells, you're just setting the `titleLabel` to the name of the album. You can make this look much better by using a custom cell of your own using a prototype cell.

4.2.3 Custom table view cells with prototype cells

Prototype cells allow you to easily create custom table view cells right within Xcode's interface editor. This allows you to create new table view cells with custom layouts from within your storyboard. You're going to customize the cell used to display an album in your Albums app by using prototype cells. Back when you added your table view controller to your storyboard, you may have noticed it within the table view itself. It can be seen in figure 4.14.

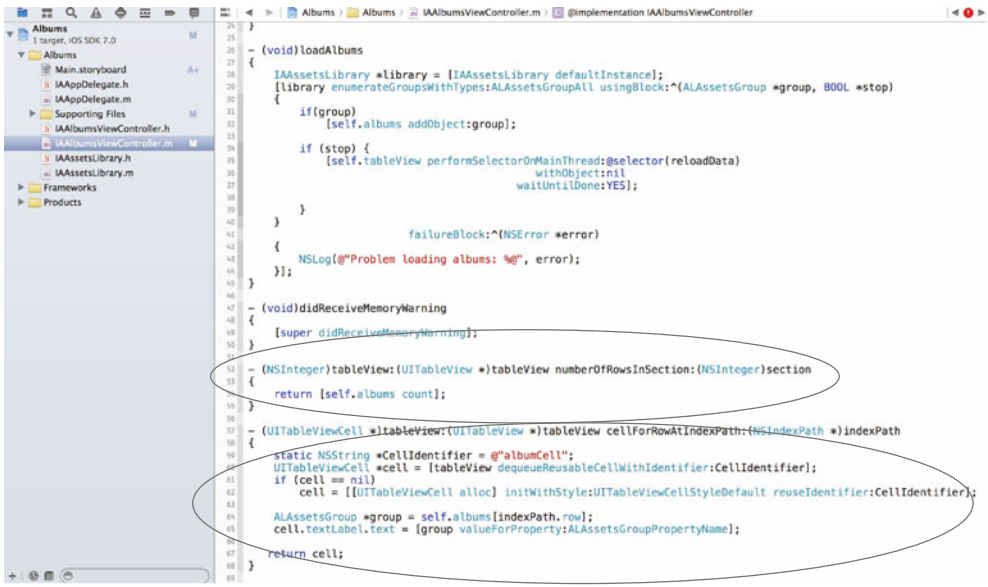


Figure 4.11 Table view data source methods added to `IAAlbumsViewController`

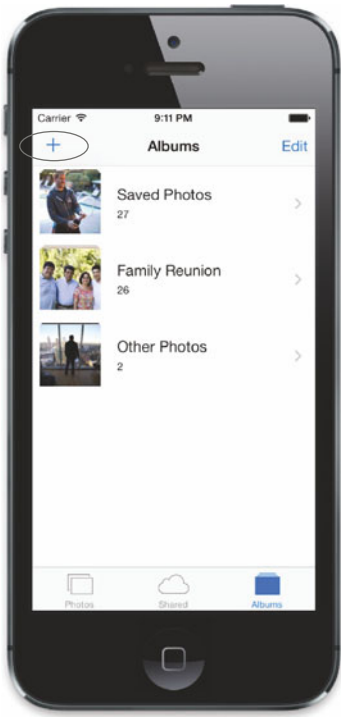


Figure 4.12 Go to Apple's Photos application and click the + button to add a few albums of your own.

Figure 4.13 Our Albums application showing the albums on our device

You're going to be creating your own custom table view cell using the prototype cell within the interface editor. Hop back into Xcode, open Main.storyboard, and locate the prototype cell within the scene that contains your table view.

The first thing you're going to do is change the height of the cell. With the cell selected, go to the size inspector and change the row height to 50. Next, go to the Object Library and locate an Image View. Drag it to the very left corner (0,0) of the cell and make its size 50 x 50. This will be used to show an image preview of each album. Now go back to the Object Library, find a Label, and drag it to the right of your image view. Change the font within the attributes inspector by making it bold and of size 18. This label will be used to display the album name.

With the prototype table view cell selected, you need to change the field named Identifier within the attributes inspector. Set this to `albumCell`. You'll be referencing this special identifier for this prototype cell when you revise the way you create the table view cell in your controller. It's crucial to be able to set this correctly, because you'll be using this identifier to specify that this is the view that you want to use for your table view cell. Take a look at the identifier being set within figure 4.15.

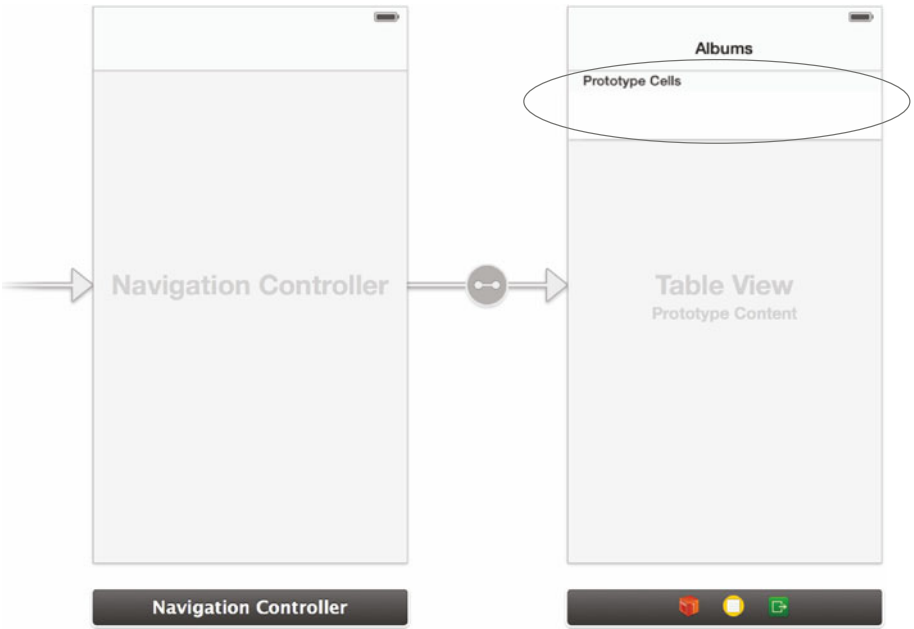


Figure 4.14 Prototype cells can be edited from within a table view inside a storyboard.

Lastly, select the table view itself and go to the size inspector. From here you should set the row height to 50 because you changed your cell to be of the same height, as shown in figure 4.16.

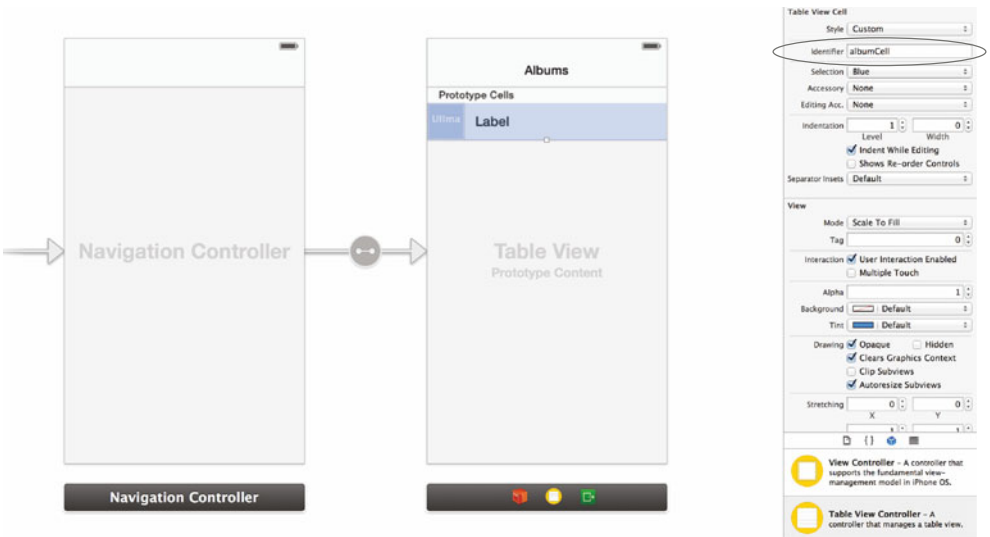


Figure 4.15 Our customized prototype cell with image view and label to better display an album. The identifier is also set to `albumCell`.

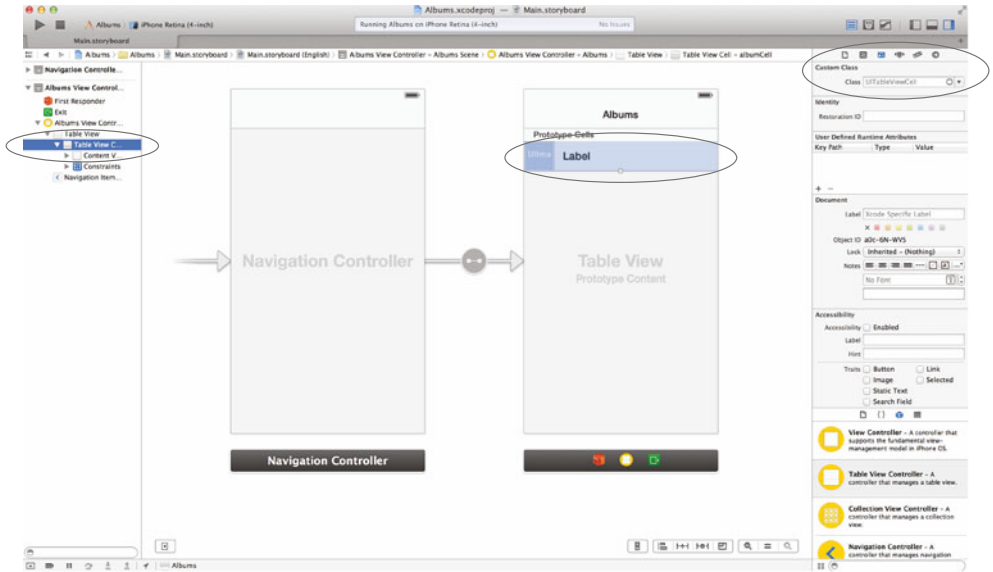


Figure 4.16 Setting the cell's custom class to `IAAlbumTableViewCell`

To be able to customize this cell, you'll create a new `UITableViewCell` subclass for your prototype cell. Hop into the project navigator, right-click the Albums group folder, and choose New File. Choose Objective-C Class, click Next, and then name it `IAAlbumTableViewCell` and specify `UITableViewCell` as the subclass.

Go back into your storyboard and set the prototype cell's class to `IAAlbumTableViewCell` in the identity inspector. Once you've done this, open the assistant editor so you can create two outlets for the image view and label that you have in this cell. Make sure that IAAlbumTableViewCell.h is selected in the assistant editor before making the connections. Start with the image view by dragging out an outlet connection and name it `albumImageView`. Next, drag an outlet connection for the label and name it `albumTitleLabel`.

You'll need to add the following to import the Assets Library framework so that your `IAAlbumTableViewCell` class knows what an `ALAssetsGroup` is:

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

Also, within the assistant editor for `IAAlbumTableViewCell`, declare the following method, which you'll set up shortly:

```
- (void)setFromAlbum:(ALAssetsGroup *)album;
```

Once finished, the interface for `IAAlbumTableViewCell` should look like what's shown in figure 4.17.

The method `setFromAlbum:` that you declared in your class's interface will be used to populate the image view and label using a specified album (`ALAssetsGroup`). Add

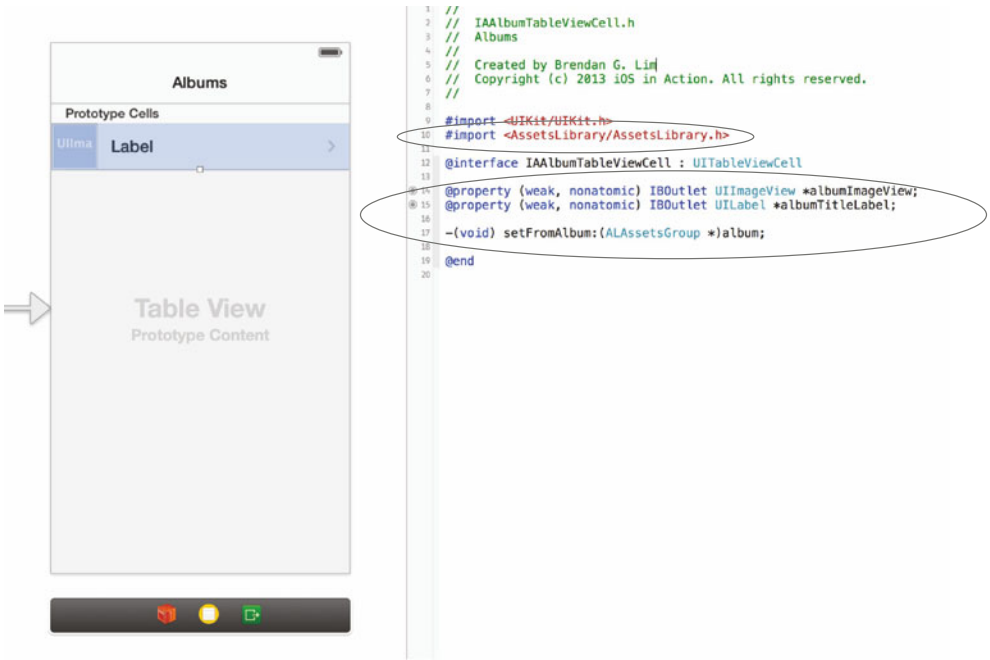


Figure 4.17 Creating outlets and declaring a method in `IAAlbumTableViewCell`'s interface

this method into your implementation by opening IAAlbumTableViewCell.m in the project navigator and inserting the following method:

```
- (void) setFromAlbum:(ALAssetsGroup *)album
{
    self.albumImageView.image = [UIImage
➥ imageWithCGImage:album.posterImage];
    self.albumTitleLabel.text = [album
➥ valueForProperty:ALAssetsGroupPropertyName];
}
```

Next, you'll update `IAAlbumsViewController` so that it supports your new custom table view cell. Open IAAlbumsViewController.m from the project navigator and add the following `import` statement to give you access to the new `IAAlbumTableViewCell` class:

```
#import "IAAlbumTableViewCell.h"
```

You can now replace your `tableView:cellForRowAtIndexPath:` method with a newly revised version that supports your custom table view cell:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
➥ cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"albumCell";
```

```
    IAAlbumTableViewCell *cell = [tableView
➥ dequeueReusableCellWithIdentifier:CellIdentifier];
    [cell setFromAlbum:self.albums[indexPath.row]];

    return cell;
}
```

Notice that it's shortened quite a bit from what you had previously. You're taking advantage of the identifier that you specified as `albumCell` and are using the `setFromAlbum:` method to populate your views. You can now run your application and see your new custom cell in action, as shown in figure 4.18.

Adding photos to the iOS Simulator

By default you won't have any photos in the iOS Simulator. A quick way to add photos is to use the Safari application on the Simulator itself. Navigate to a website that has photos, and tap and hold on one to save it.

Table view cells also have an accessory view. The accessory view for a table view cell can act as a visual cue to a user. If there is something more to see by tapping a row, you can use an accessory view to convey this. If a row has a selected state, you could even show a checkmark as the accessory view. Right now, you don't have an accessory view for the cells representing each of your albums. It would let users know that they can click a cell in a row to progress further. A few different types of accessory views are available, as shown in figure 4.19.

The disclosure indicator (`UITableViewAccessoryDisclosureIndicator`) accessory view is the most commonly used accessory view, which you've seen in many other apps. It's normally used to tell users that after clicking this row, they'll be presented with another view that contains another table or collection view.

The detail button (`UITableViewAccessoryDetailButton`) accessory view tells users that they'll be presented with a detail about the cell's contents. This control itself is clickable. There's also a detail disclosure button (`UITableViewAccessoryDetailDisclosureButton`), which contains the same button but with a disclosure indicator on the right.

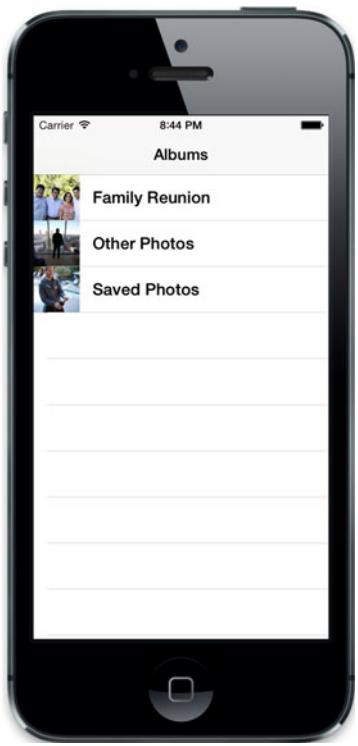


Figure 4.18 Albums application with our custom table view cell being used to display each album

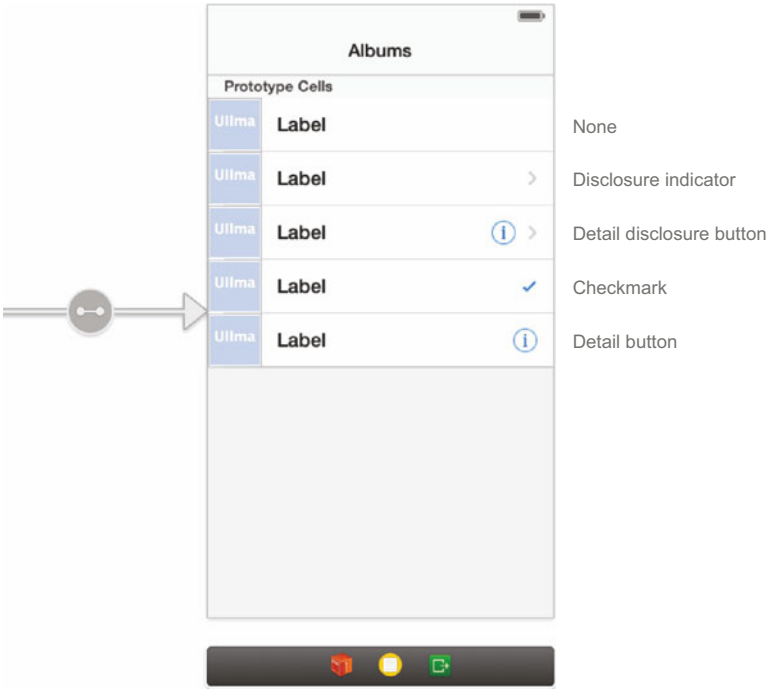


Figure 4.19 Different types of accessory views that are available

If you were to create an interface where you could select one or multiple items before continuing to another view, you'd use the checkmark (`UITableViewAccessoryCheckmark`) accessory type.

For your purposes, you'll be using the disclosure indicator since you'll be presenting another view in which a user will be able to eventually dig down further. Jump into Main.storyboard, select your table view cell, and go to the attributes inspector. Under the Accessory field, choose Disclosure Indicator, as shown in figure 4.20.

Since you've added this accessory view to your cell, your albums should display the disclosure indicator to the right of the album title. This subtle change will make a big difference because it will convey to your users that there is more to see once they click an album within a row. You can see our accessory view in action in figure 4.21.

You've supplied your table view with the data it needs to display the albums within your assets library. What you haven't done yet is handle the different events when a user performs an action on your table view. How do you know when a row is tapped and selected or deselected?

4.3 Managing selection and deletion within a table view

Different actions can take place on a row within a table view; the most common ones are selection and deselection. When you tap a row, you expect something to happen, especially if there's an accessory view that tells you there's more to be seen. Luckily, you have

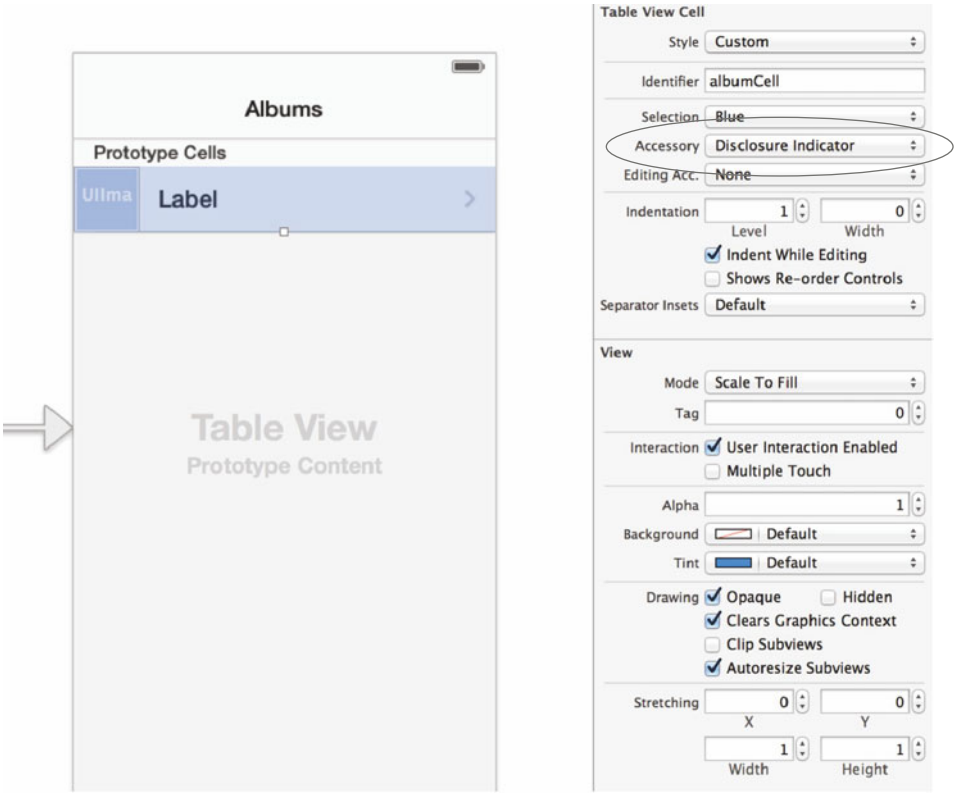


Figure 4.20 Select the table view cell and set its accessory view to Disclosure Indicator within the attributes inspector.

control over this if you implement the `UITableViewDelegate` protocol. Within the protocol are various methods that are triggered when a variety of these events occur.

You've already used the `UITableViewDataSource` protocol to provide your table view with the data it needed. Much like how you implemented various methods to supply it with data, your controller can implement various methods to respond to actions. Unlike the data source protocol, there are no methods that you are *required* to implement. Everything is purely optional and up to you. You decide what you want to respond to and how you want to do it.

4.3.1 Deleting rows within a table view

There are times when you'd want someone to be able to slide to delete a row within a table view. For instance, if you wanted a quick way to delete tasks that you've entered in the Tasks app you previously created, you could allow users to swipe to delete. In your Albums app, you're only allowing users to view what they have in their assets library. If you wanted to implement deletion, you could do so by implementing a few methods in your view controller.

Let's go ahead and implement it without actually deleting anything permanently. You'll just be visually removing the rows and not actually deleting any photo albums that you've created. So don't worry—none of the photos you have on your device will actually be removed. First, you need to enable deletion on your table view by adding one specific `UITableViewDataSource` method. Open Xcode, choose IAAlbumsViewController.m, and add the following method:

```
- (BOOL)tableView:(UITableView *)tableView
➥ canEditRowAtIndexPath:(NSIndexPath *)indexPath
{
    return YES;
}
```

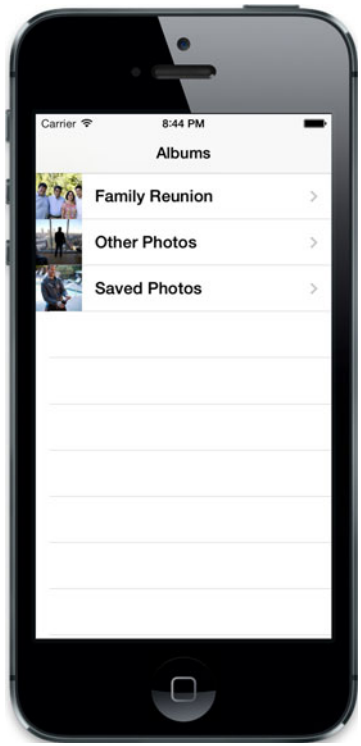


Figure 4.21 The disclosure indicator shown to the right of each album name within our Albums application

This method tells your table view that any row can be edited because you're returning `YES` no matter what index path is passed in. If you wanted only a few rows to be editable, you could do so by conditionally returning `YES` or `NO` depending on the index path. Next, you'll have to implement the method `tableView:editingStyleForRowAtIndexPath:`. This method expects a return type of `UITableViewCellEditingStyle`. You can choose to have no editing style (`UITableViewCellEditingStyleNone`), deletion (`UITableViewCellEditingStyleDelete`), or insertion (`UITableViewCellEditingStyleInsert`). For your purposes you'll support only deletion. Add the following delegate method to `IAAlbumsViewController`:

```
- (UITableViewCellEditingStyle)tableView:(UITableView *)tableView
➥ editingStyleForRowAtIndexPath:(NSIndexPath *)indexPath
{
    return UITableViewCellEditingStyleDelete;
}
```

Next, you'll need to perform an action when a cell is deleted. You guessed it: there's a delegate method for this as well—although this method is actually part of the `UITableViewDataSource` protocol. Add the following code to your view controller.

Listing 4.3 Deleting a row from a table view

```
- (void)tableView:(UITableView *)tableView
➥ commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
➥ forRowAtIndexPath:(NSIndexPath *)indexPath
{
    if (editingStyle == UITableViewCellEditingStyleDelete)
```

Check if editing style is for deletion.

```
    {
        [self.albums removeObjectAtIndex:indexPath.row];
        [tableView deleteRowsAtIndexPaths:@[indexPath]
➥ withRowAnimation:UITableViewRowAnimationFade];
    }
}
```

Remove object from `albums` array.

Remove row from table view.

Within this method you're first checking to see if the editing style is for deletion, because you know that a table view can support multiple editing styles. You're then removing the object from your `albums` array and removing the row from the table view. You're removing the object from the array because your table view will still think that there are X number of rows to display, even after your row has been deleted. Because you are returning the number of rows to display depending on the size of your `albums` array, you need to update your array to reflect the change.

Once finished, your code controller should contain these three new methods, as shown in figure 4.22.

Try to run the application and see how it works. Once your albums are showing, swipe from left to right on a specific row. A Delete button should appear on the right side of the table view cell. If you tap the Delete button, the cell should be removed from your table view, as shown in figure 4.23.

One remarkable thing is that you didn't need to write code to detect the left-to-right swipe gesture. You also didn't need to implement your own Delete button or even the animation for removing a row in a table view. This is all handled for you with minimal effort.

How about when you want to select a row? Let's see what delegate method you need to implement to be able to respond to this.

Figure 4.22 Three new methods to support deletion added to `IAAlbumsViewController`

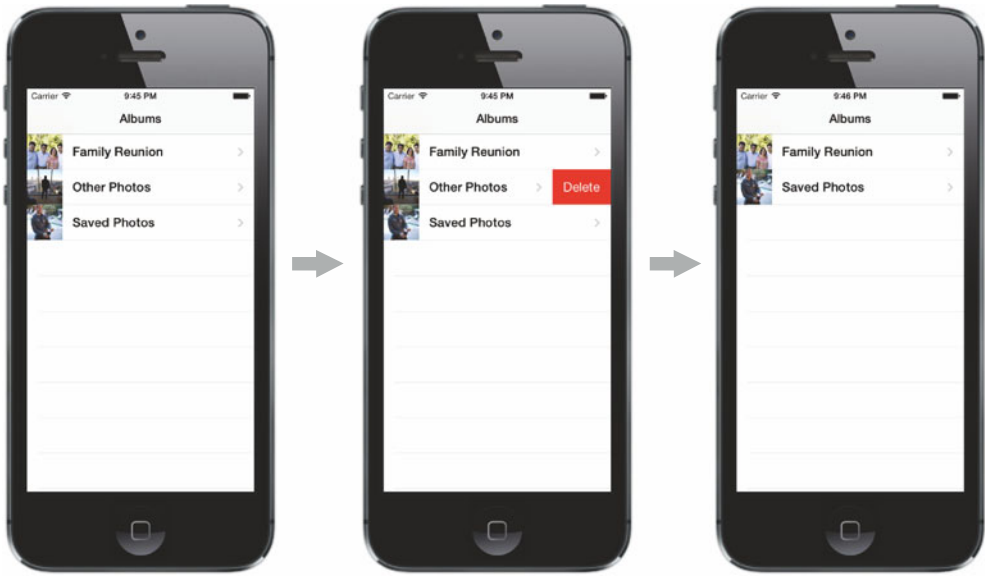


Figure 4.23 Swipe to delete a specific row in the table view within the Albums application.

4.3.2 Handling the selection and deselection of rows

Probably the most important method to implement is the one that is called by your table view when a row is selected or deselected. You want to be able to respond when someone taps a row within your table view. The method within the `UITableViewDelegate` protocol that you'd use to handle this action is `tableView:didSelectRowAtIndexPath:`. Let's see what it looks like:

```
- (void)tableView:(UITableView *)tableView
➥ didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Perform an action
}
```

The second parameter that's passed in is an `NSIndexPath`, which lets you know the section and the row for the cell that's been selected. In this case, this lets you know which album was selected by using the position of the row to locate the album within the `albums` array.

```
ALAssetsGroup *group = self.albums[indexPath.row];
```

Within this method you can programmatically trigger a segue to load a new scene for this album. You did something similar in the Tasks app that we built together in the previous chapter. When a row was selected, you loaded another scene that showed a specific task.

In the next chapter you'll expand on this application and add a view that you can display when a row is selected. For now you'll add a log statement so that you know

when a row has been selected. While you're at it, you can also deselect the row so that it doesn't stay highlighted after you've finished tapping it. Add the following code to the `IAAlbumsViewController` implementation.

Listing 4.4 Handling selection of a row within a table view

```
- (void)tableView:(UITableView *)tableView
➥ didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  IAAlbumTableViewCell *cell = (IAAlbumTableViewCell *)
➥ [tableView cellForRowAtIndexPath:indexPath];
  NSLog(@"Selected %@", cell.albumTitleLabel.text);
  [self.tableView deselectRowAtIndexPath:indexPath
➥ animated:YES];
}
```

Retrieve the cell that was selected from the index path.

Log that you've selected a specific album.

Deselect the selected row with a fade animation.

Within this method you first retrieve the cell that was selected by using the index path passed in. You then log that you've selected a specific album by specifying the album title to be shown within the log. Last, you deselect this cell and specify that you want the deselection to be animated.

If you run the application and select a row for an album named Camera Roll, you should see the following log statement printed to the debug area (View > Debug Area > Show Debug Area, or press Shift-Command-Y) within Xcode. The selected row should also deselect itself after this log statement is printed:

```
Selected Camera Roll
```

You're now able to respond to taps on a specific row within a table view. Don't worry; there's much more to come with this. You'll expand on row selection within your Albums application when you get to the next chapter.

4.4 Summary

Table views are used throughout iOS to display data within an organized list. You've learned how to add a table view first-hand by adding one to your own application. You provided the table view with data using the `UITableViewDataSource` protocol. Also you created your own custom table view cells and learned how to remove rows and how to respond to row selection. Along the way you also created the first part of an application that lets you list the albums that you have on your iOS device.

- You can use table views to present information in a list using cells that can be contained within different sections.
- Content within each row of a table view is contained within a `UITableViewCell`.
- To improve performance, you can give prototype cells cell identifiers that can allow them to be reused when drawing a table view.
- You can choose different accessory views for a table view cell that can be used to provide users with visual triggers depending on the appropriate action you want them to take.

- Table views rely on a data source to provide them with the data that they should display.
- You use the delegate of a table view to respond to specific actions that take place within the table view.
- You created an application that lists all of the photo albums you have on your device within a table view.

5 Using collection views

This chapter covers

- Using a collection view to display photos
- Implementing custom collection view cells
- Customizing collection view flow layouts
- Adding a collection view to your Albums app

When you created a table view for your Albums application, you used it to display a list of albums represented as rows. This worked well because you needed to display only an album's thumbnail and its title beside it. One vital piece that it was missing was the ability to click an album and view all of the photos contained within it. The one thing you'll need to display for each photo in the album is its thumbnail. You'll use collection views to display all of the photos within a specific album. By using collection views, you'll be able to put the thumbnails of each photo side by side using rows *and* columns.

Although you'll be using a collection view to display photos in a grid, you'll learn that they allow quite a bit of flexibility. You'll be starting where you left off in the previous chapter so that you can finish your Albums application. Once you've finished, you'll have a nice app that can allow you to view the albums and the photos you've stored on your phone, as shown in figure 5.1.

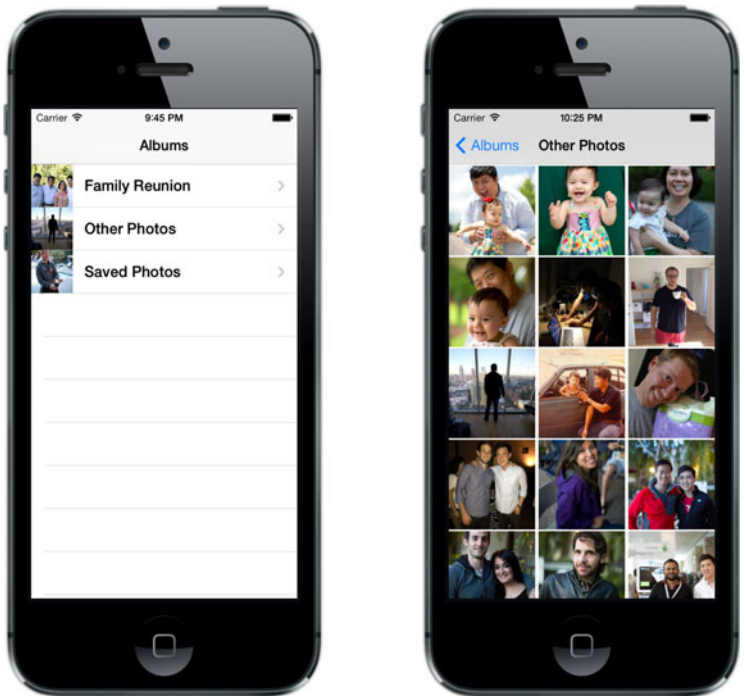


Figure 5.1 Our finished Albums application after we added a collection view to view photos within an album

Let's get started by learning what makes up a collection view.

5.1 Introducing collection views

In many ways collection views are similar to the table views that you've already used. They both use delegates, a data source, and cells to display your data. There's also a special view controller provided in `UIKit` that makes it easy for you to integrate a collection view in your application, much like the `UITableViewController` class. Things start getting pretty different when you explore how to visually lay out the cells within your collection view, as you'll soon see. You can start by breaking down the different parts of a collection view and then integrating them into your Albums application.

You've seen collection views used in a variety of apps within iOS. They're most commonly used for displaying data in a grid. This can be seen in figure 5.2, which shows a collection view used to display photos within the Photos application.

Within the view shown in figure 5.2, there's one collection view but many cells used to represent photos. Note that each photo is a separate collection view cell. There are multiple columns and rows within this view. This is very similar to what you're going to be creating.

The `UICollectionView` class is used to represent a collection view. Each collection view has a few properties that are required to supply and respond to actions, such as

Figure 5.2 Collection view used within Apple's Photos application

`delegate` and `dataSource`, just like a table view. These two protocols are conveniently named `UICollectionViewDelegate` and `UICollectionViewDataSource`. The content displayed within a collection view is also referred to as a cell (`UICollectionViewCell`). The way that these cells are presented within a collection view is dramatically different from a table view, though.

When presenting content within a collection view, *layout* objects are used. These layout objects allow you to dynamically position items within the collection view. With table views, rows are presented one after the other. Each row within the table view can have a different height, but they all have the same width. Because collection views allow you to display content using grids and rows, a wider variety of customization is needed, especially because cells can be a different height *and* width. Take a look at figure 5.3, which shows the exact same photos within the Photos application using the iPad Simulator.

On the iPad, the Photos app displays cells with varying height and width depending on the proportion of the images. In the Photos app for iPhone, the thumbnails are shown as squares with the same height and width for each cell. Collection views give you a large amount of flexibility with layout objects. You'll be controlling the layout with the collection view used in your Albums app later in this chapter. But for now, let's focus on how to start using collection views by adding one to your app.

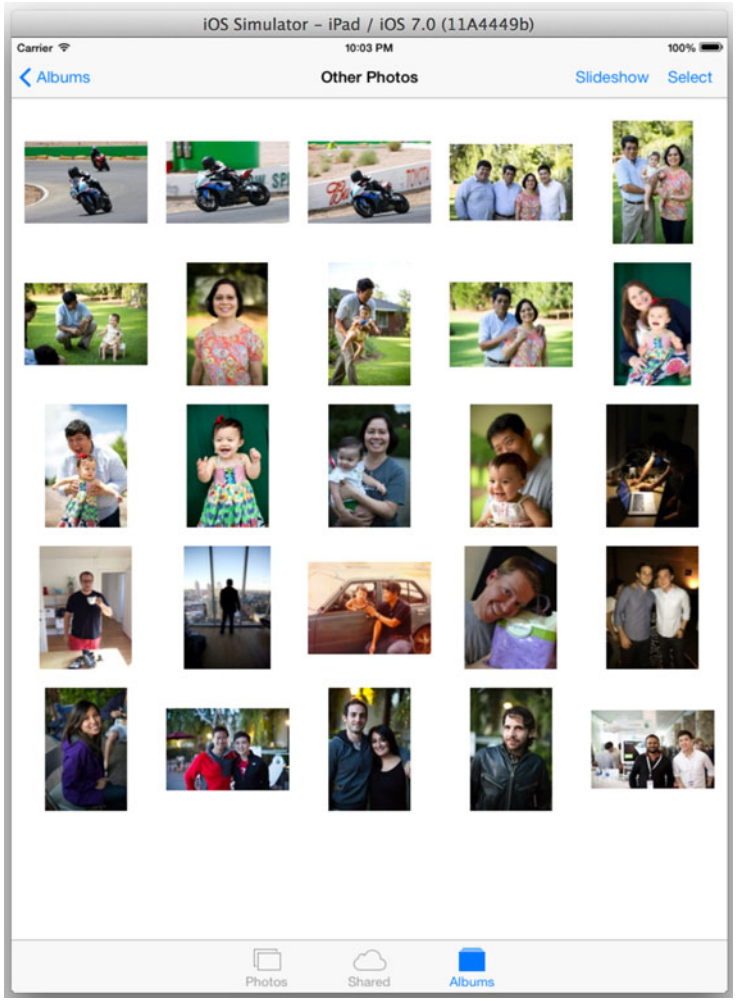


Figure 5.3 Photos shown in a collection view within the iPad Simulator. The cells are of varying height and width compared to the collection view in the Photos app for iPhone.

5.2 Using collection views to display data

To be able to display anything within a collection view, the view needs to be told what it should display. You're going to be adding a new scene to the storyboard within your Albums application that will be shown when someone taps an album they want to view. This scene will contain the collection view you'll be using to display photos within a particular album. After setting up this scene within your Albums application using a `UICollectionViewController`, you'll populate it with the data it should display.

5.2.1 Adding a UICollectionViewController as a new scene

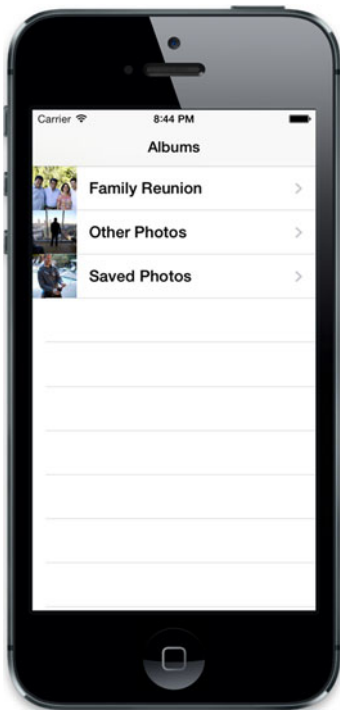


Figure 5.4 Our Albums application, with its single scene to display albums within a table view as we left it in the previous chapter

You're going to be starting right where you left off in the previous chapter with your Albums application. The most that you were able to do was list all of the albums within a table view. You can see these albums listed within figure 5.4.

When you tap an album you'll need to segue to a new scene that shows all of its photos. You'll be creating a `UICollectionViewController` subclass to handle this for you.

To get started, jump back into Xcode and open the Albums project. Within the project navigator, create a new file within the Albums group. Choose Objective-C class as the file template, and then click Next. Let's call this new class `IAAlbumPhotosViewController` and make it a subclass of `UICollectionViewController`, as shown in figure 5.5.

Once you've created your new class, jump into your storyboard by selecting Main.storyboard from the project navigator. You're going to add a new scene for your newly created class. Go to the Object Library and find Collection View Controller, and drag it into the storyboard. Once you've dragged it in, you'll set its class to `IAAlbumPhotosViewController` by changing it within the identity inspector. You can see this in figure 5.6.

To be able to transition to this scene when an album is selected you'll trigger a segue when a row is tapped in your album table view. Create a push segue by selecting and then dragging from your album table view cell to your newly created collection view scene. Once the segue has been created, select it and go to the attributes inspector. From there you can set the name of its identifier to `albumPhotosSegue`, as shown in figure 5.7.

You now have your segue set up to transition to your new scene. Before you go further, click the collection view and set its background color to white within the attributes inspector. Now that it's set, you need to supply your collection view with the data it needs before you proceed further.

5.2.2 Supplying a collection view with data

Just like a table view, a collection view requires a data source to provide it with data to display. The data source has to implement a few required methods so that it knows the number of sections, total number of items within each section, and what cell to display

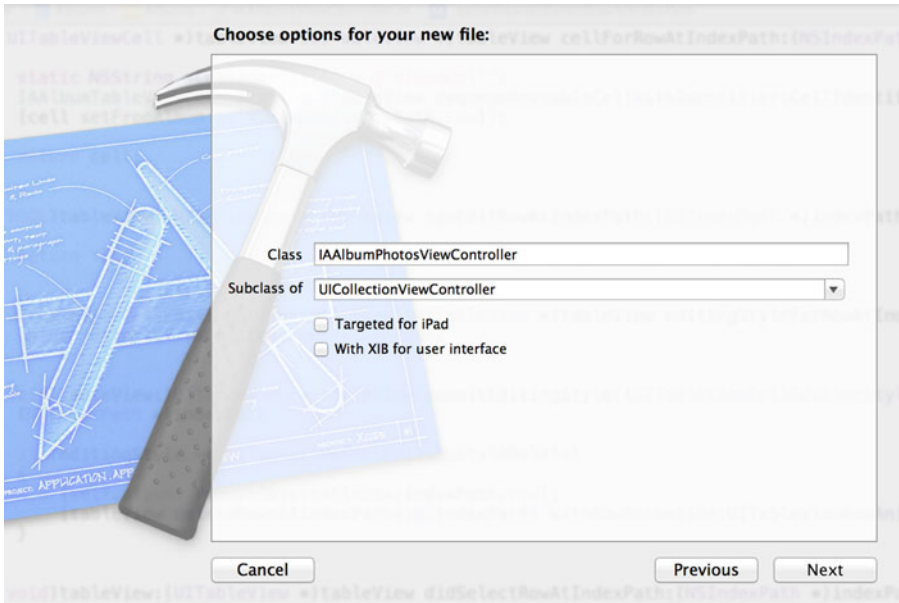


Figure 5.5 Create a new Objective-C class called `IAAlbumPhotosViewController` as a subclass of `UICollectionViewController`.

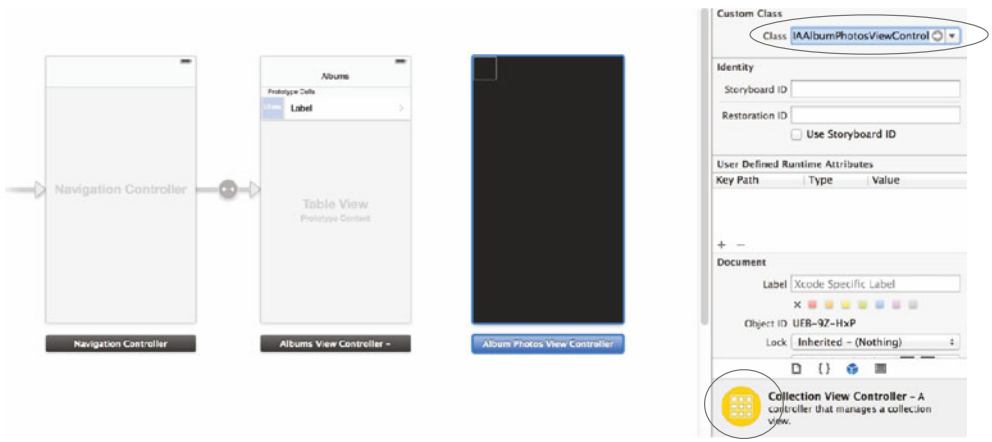


Figure 5.6 Add a Collection View Controller as a new scene in your storyboard and set its class to `IAAlbumPhotosViewController`.

at a particular index path. These methods are defined within the `UICollectionViewDataSource` protocol. Table 5.1 lists the methods you'll be using.

Before you add these methods to your controller, you'll lay some groundwork so that you can retrieve photos from an album to display within a table view. You need to add two properties: one that will represent the album you're going to display as well as an array to store all of the album's photos.

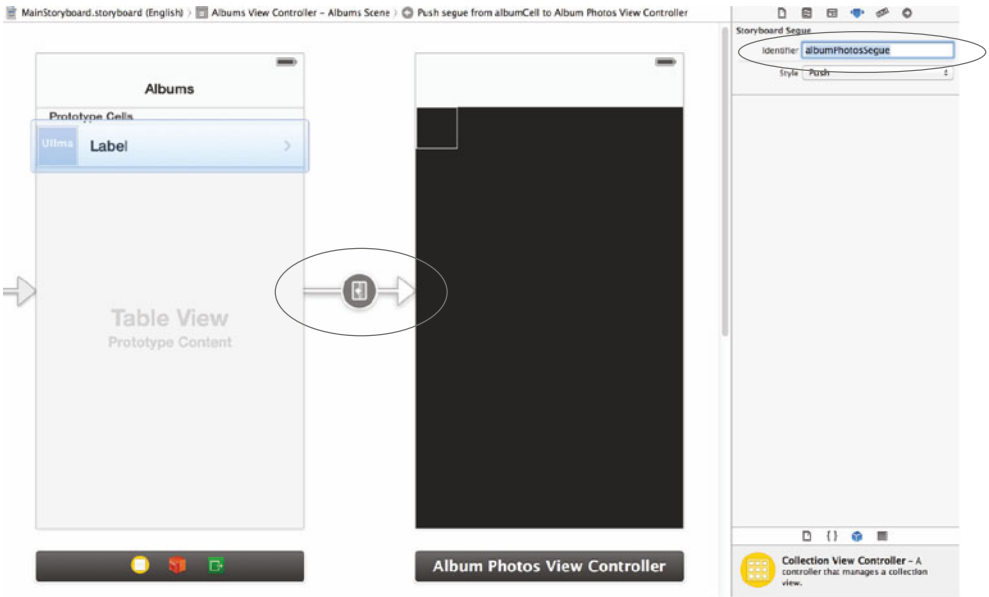


Figure 5.7 Create a segue from your album table view cell to your new collection view scene. Set the name of the identifier to `albumPhotosSegue` in the attributes inspector.

Table 5.1 Three `UICollectionViewDataSource` protocol methods you'll be implementing

Section	Description
`numberOfSectionsInCollectionView:`	Number of sections within the collection view
`collectionView:numberOfItemsInSection:`	Number of items to display within a particular section
`collectionView:cellForItemAtIndexPath:`	`UICollectionViewCell` for a particular index path

You'll be adding an `NSMutableArray` property to the `IAAlbumPhotosViewController` class to store the photos contained within an album. To be able to reference the album that you're viewing, you'll create a property that holds an instance of an `ALAssetsGroup`. You'll set these up first within the interface for the `IAAlbumPhotosViewController` class by opening IAAlbumPhotosViewController.h. First, you should import the Assets Library framework because the `ALAssetsGroup` class is part of that framework. To do this, add the following `import` statement:

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

Next, add the following two properties that will be used to store a reference to your photos and the album that they belong to:

```
@property(nonatomic, strong) ALAssetsGroup *album;
@property(nonatomic, strong) NSMutableArray *photos;
```

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

@interface IAAlbumPhotosViewController : UICollectionViewController

@property(nonatomic, strong) ALAssetsGroup *album;
@property(nonatomic, strong) NSMutableArray *photos;

@end
```

Figure 5.8 Adding two properties to the interface of your `IAAlbumPhotosViewController` class to store a reference to your photos and the album that they belong to

Once you've finished, your code should look like that shown in figure 5.8.

To be able to supply your collection view with the album it needs to use to fetch photos, you need to add the `prepareForSegue:sender:` delegate method to `IAAlbumsViewController`. This method will allow you to pass necessary data to the view controller that's about to load. Within this method you'll set the album property before the segue takes place.

Open IAAlbumsViewController.m from the project navigator to get started. First, import IAAlbumPhotosViewController.h by adding the following line, because you'll be referencing it in your code shortly:

```
#import "IAAlbumPhotosViewController.h"
```

Next, remove the `tableView:didSelectRowAtIndexPath:` method that you added in the previous chapter: you're not going to need this anymore because you're automatically triggering the segue when your table view cell is tapped. Also, in that method you were deselecting the selected table view row. You'll need this to remain selected so that you can use it to pass the appropriate album to the `IAAlbumPhotosViewController`.

Next, add the `prepareForSegue:sender:` method, as shown in the following listing.

Listing 5.1 Passing a reference to the selected album using `prepareForSegue:sender:`

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    NSIndexPath *selectedIndex = [self.tableView
➥ indexPathForSelectedRow];
    ALAssetsGroup *album = self.albums[selectedIndex.row];
    [(IAAlbumPhotosViewController *)[segue destinationViewController]
➥ setAlbum:album];
  [self.tableView deselectRowAtIndexPath:selectedIndex animated:YES];
}
```

❶ Retrieving index path of selected row in table view

❷ Retrieving the album from `albums` array

❸ Setting the `album` property for `IAAlbumPhotosViewController`

❹ Deselecting the currently selected row in table view

Within this method you're first retrieving the `NSIndexPath` of the currently selected row in your table view ❶. The `row` property within `selectedIndex` will give you the correct index to use to retrieve the album within your `albums` array ❷. You're then

setting the `album` property on the `IAAlbumPhotosViewController` ❸. Last, you're deselecting the currently selected view ❹.

Let's jump into IAAlbumsPhotoViewController.m and retrieve the photos that belong to the album that you're passing in when you perform the segue. First, add the following `import` statement because you'll be referencing your `IAAssetsLibrary` singleton that you created in the previous chapter.

```
#import "IAAssetsLibrary.h"
```

Next, within the `viewDidLoad` method, add the following three lines of code:

```
self.title = [self.album valueForProperty:ALAssetsGroupPropertyName];
self.photos = [NSMutableArray new];
[self loadPhotos];
```

The first line is to set the title of your view to the name of the album you're viewing. The second line is added to initialize the `NSMutableArray` that you'll be using to store the photos contained within an album. The third line is used to call a method called `load_Photos` that you're about to add. The goal of this method is to retrieve all of the photos that belong to the `ALAssetsGroup` stored within your `album` property. Add the `loadPhotos` method shown in the following listing.

Listing 5.2 Adding `loadPhotos` to retrieve photos within an album

```
- (void)loadPhotos
{
    [self.album enumerateAssetsUsingBlock:^(ALAsset *result,
➥ NSUInteger index, BOOL *stop)
    {
        if ([result valueForProperty:ALAssetPropertyType] ==
➥ ALAssetTypePhoto)
            [self.photos addObject:result];
    }];

    [self.collectionView reloadData];
}
```

❶ **`ALAssetsGroup`'s `enumerateAssetsUsingBlock:` method to retrieve photos**

❷ **Check if the `ALAsset` is a photo.**

❸ **Add the `ALAsset` to the photos array property.**

❹ **Reload the collection view.**

To be able to load all of the assets within an album (`ALAssetsGroup`) you use the `enumerateAssetsUsingBlock:` method ❶. Within the block you're passing in, you check to see if the `ALAsset` named `result` is of type `ALAssetTypePhoto` ❷. If it is, you add it to the `photos` array ❸. The last thing you do is reload the collection view by calling `reloadData` ❹.

The data source methods within the `UICollectionViewDataSource` protocol are fairly similar to those of the `UITableViewDataSource` protocol. When you wanted to specify the number of sections within a table view, you'd use `numberOfSectionsWithinTableView:`. For a collection view, you'd use `numberOfSectionsWithinCollectionView:`. When specifying the number of rows within a section for a table view, you'd use `tableView:numberOfItemsInSection:`. What do you use for collection views? You guessed it—`collectionView:numberOfItemsInSection:`.

The number of items within each section will inform the collection view how many items it needs to display. For your Albums app, you'll have only one section, and the number of items in this section will be the total count of the `photos` array. Add the code shown in the following listing.

Listing 5.3 Specifying the number of sections and items in each section

```
- (NSInteger) numberOfSectionsInCollectionView:(UICollectionView
*)collectionView
{
    return 1;
}
- (NSInteger) collectionView:(UICollectionView *)collectionView
     numberOfItemsInSection:(NSInteger)section
{
    return [self.photos count];
}
```

Once you've added this to IAAlbumPhotosViewController.m, you should see code similar to what's shown in figure 5.9.

```
#import "IAAlbumPhotosViewController.h"
#import "IAAssetsLibrary.h"

@interface IAAlbumPhotosViewController ()

@end

@implementation IAAlbumPhotosViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    self.title = [self.album valueForProperty:ALAssetsGroupPropertyName];
    self.photos = [NSMutableArray new];
    [self loadPhotos];
}

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

- (void)loadPhotos
{
    [self.album enumerateAssetsUsingBlock:^(ALAsset *result, NSUInteger index, BOOL *stop)
    {
        if ([[result valueForProperty:ALAssetPropertyType] isEqualToString:ALAssetTypePhoto])
            [self.photos addObject:result];

    }];

    [self.collectionView reloadData];
}

- (NSInteger) numberOfSectionsInCollectionView:(UICollectionView *)collectionView
{
    return 1;
}

- (NSInteger) collectionView:(UICollectionView *)collectionView numberOfItemsInSection:(NSInteger)section
{
    return [self.photos count];
}
```

Figure 5.9 `IAAlbumPhotosViewController` with methods in place for you to supply your collection view with the data it needs

Now that you have some of this groundwork set up, you'll get ready to display the photos that you have within an album by creating a custom `UICollectionViewCell`.

5.2.3 Creating a custom collection view cell

If you remember from the previous chapter, you used a prototype cell to create a custom cell for your table view. You'll be doing something very similar for your collection view that will be used to display a thumbnail for each photo. This will involve creating a new subclass of `UICollectionViewCell` that contains a single `UIImageView`. You'll then create this view *within* your collection view in your storyboard.

Start by creating a new Objective-C class within your project called `IAPhotoCollectionViewCell` that's a subclass of `UICollectionViewCell`. This is shown in figure 5.10.

Once you have this class created, you can hop right into your storyboard to create its view. Open Main.storyboard and locate your collection view. You should see a single collection view cell on the top right. If you can't locate it, take a look at figure 5.11 to see it.

This `UICollectionReusableView`, which is a subclass of `UICollectionViewCell`, will act as a prototype cell for you. Once you've finished you'll set its class to `IAPhotoCollectionViewCell`. First, make it bigger by going to the size inspector and changing its size from Default to Custom. You can then set its width to 104 and its height to 104 as well, as shown in figure 5.12.

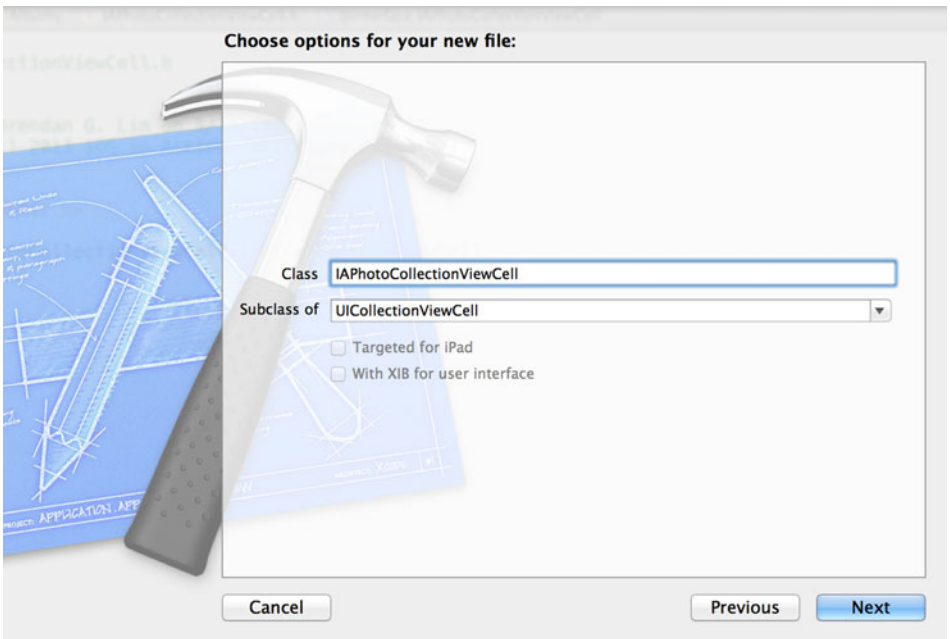


Figure 5.10 Creating a new `IAPhotoCollectionViewCell` class as a subclass of `UICollectionViewCell`

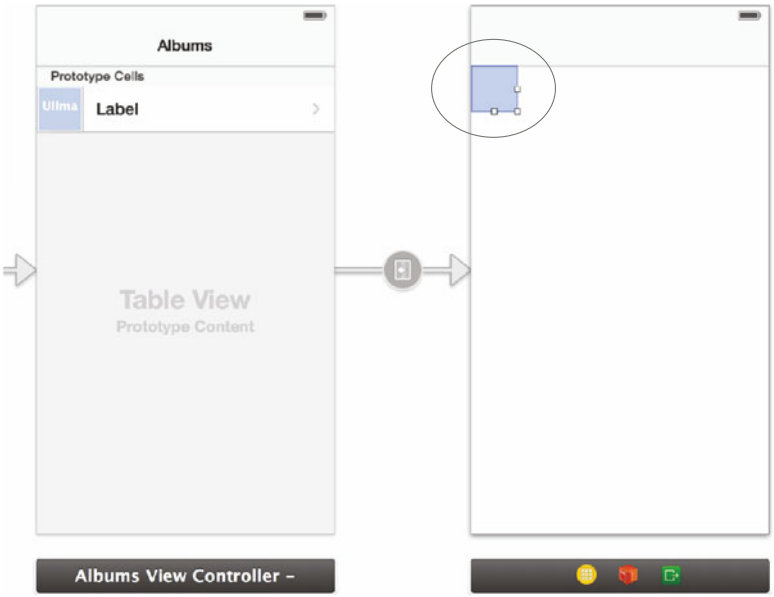


Figure 5.11 The collection reusable view that you'll be using to customize your collection view cell

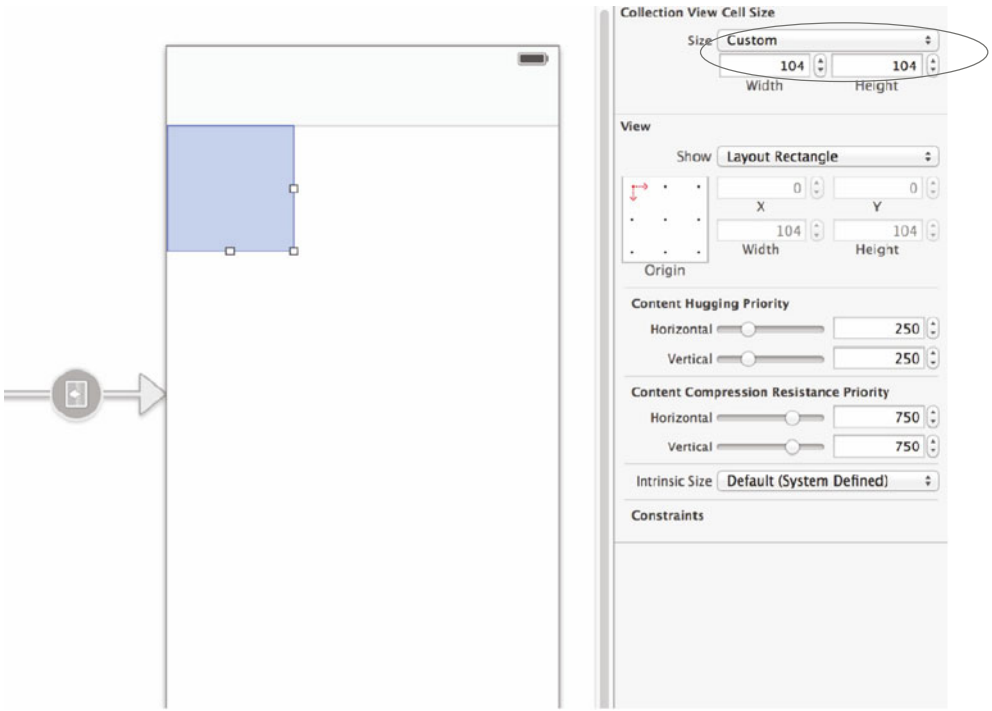


Figure 5.12 Setting the width and height of the collection view to a custom size of 104 x 104

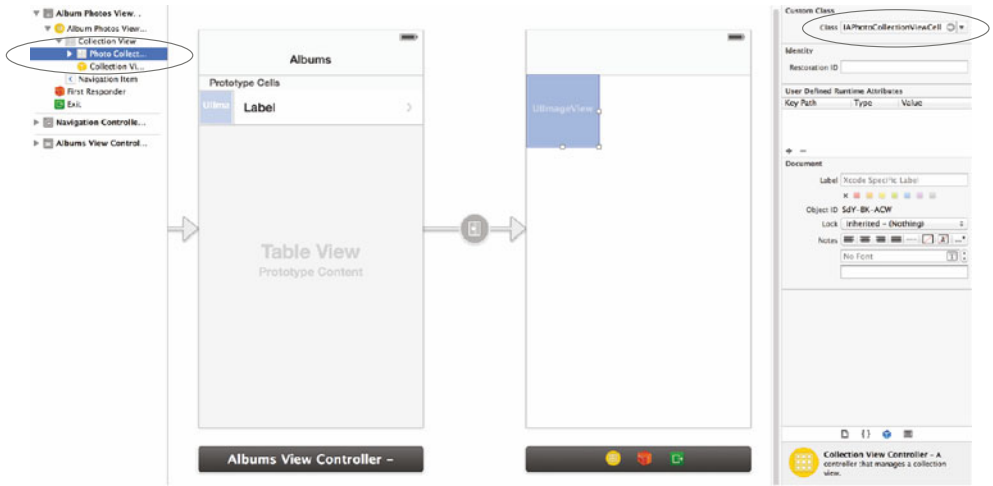


Figure 5.13 Setting the class of the collection view cell to `IAPhotoCollectionViewCell`

Next, drag a `UIImageView` into this cell by first locating one from the Object Library on the bottom-right of your window. Ensure that the `UIImageView` is positioned exactly in the center of this cell and has the dimensions 104 x 104 as well. Now jump into the identity inspector of the collection view cell and set its class to `IAPhotoCollectionViewCell`, as in figure 5.13.

You can make a new outlet from the `UIImageView` by first opening the assistant editor. Open IAPhotoCollectionViewCell.h within the assistant editor. Next, drag a connection from the `UIImageView` within the collection view cell to create an outlet called `imageView`.

Next, make sure the collection view cell is selected; then jump into the attribute inspector and change its identifier to `photoCell`. It's important to spell this correctly because you'll be using this identifier to retrieve this cell from your collection view. That's it for your custom collection view cell. Because you're just showing the thumbnail of a single photo, there's no need for anything special other than a `UIImageView`.

You can now use the `IAPhotoCollectionViewCell` that you've created. You'll use the `collectionView:cellForRowAtIndexPath:` method to return an instance of your custom collection view cell. Go into the project navigator and open IAAlbumPhotosViewController.m. Then import your `IAPhotoCollectionViewCell` class:

```
#import "IAPhotoCollectionViewCell.h"
```

Next, implement `collectionView:cellForRowAtIndexPath:` by adding what's shown in the following listing.

Listing 5.4 Returning a new `IAPhotoCollectionViewCell` for an index path

```
- (UICollectionViewCell *)collectionView:(UICollectionView *)collectionView
                  cellForItemAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"photoCell";
    IAPhotoCollectionViewCell *cell = [self.collectionView
➥ dequeueReusableCellWithReuseIdentifier:CellIdentifier
➥ forIndexPath:indexPath];

    ALAsset *asset = self.photos[indexPath.row];
    cell.imageView.image = [UIImage imageWithCGImage:asset.thumbnail];

    return cell;
}
```

❶ Static `NSString` "photoCell" to serve as the cell's identifier

❷ Dequeue a new cell using the cell identifier.

❸ Retrieve an `ALAsset` using the index path from the `photos` array.

❹ Set image for cell's image view.

You first create a static `NSString` that contains `photoCell`, which is the identifier you set for your collection view cell ❶. Next, you retrieve an instance of your cell by using the `dequeueReusableCellWithReuseIdentifier:forIndexPath:` method ❷. Once you have your collection view cell, you're ready to populate its image view with a `UIImage`. First, you get the appropriate photo represented by an `ALAsset` object by returning the appropriate instance based on the index path you're displaying ❸. Then you set the image for the `imageView` property on your cell to a new `UIImage` based on the `thumbnail` property of your `ALAsset` ❹.

Great job! Take a look at what you've done by running your Albums application and tapping an album that you want to view. Your collection view will look similar to what's shown in figure 5.14.

The problem you're faced with now is that the cells are too small and not spaced properly. You'll fix this by customizing the layout of your collection view.

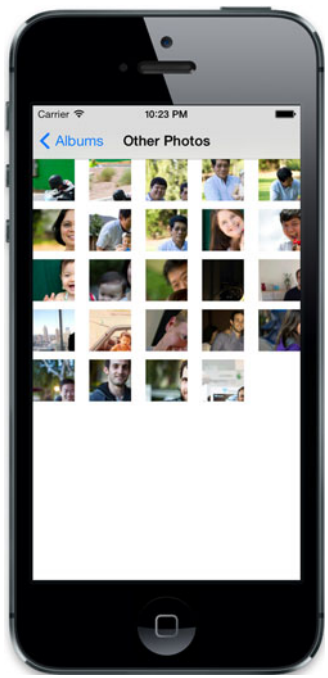


Figure 5.14 Seeing the `IAAlbumPhotosCollectionView` in action within our Albums application

5.3 Customizing a collection view layout

Collection views have layout objects that are used to determine the way their cells should be laid out and displayed. This gives you a great amount of control over the appearance of your cells depending on what content you want to display. Because the photos you're displaying are squares with equal width and height, margins between each photo should be the same throughout the collection view. This makes your layout customization simple. If you were displaying items of varying dimensions, you'd be able to adapt your collection view easily because you're able to customize the way the cells are laid out.

Because you're displaying photos, you want to show as much of the photos as you can, while still allowing

your users to be able to quickly glance and see others. You'll see how you can change this by using the `UICollectionViewDelegateFlowLayout` protocol.

5.3.1 Collection view flow layouts

The flow layout controls how a particular collection view's contents are laid out and visually arranged. You can choose to subclass a `UICollectionViewFlowLayout` and change its properties to meet the design needs of your collection view if you don't need to do much customization. You can also choose to implement the `UICollectionViewDelegateFlowLayout` protocol within your collection view delegate. This gives you more control because you can dynamically change layout properties when a delegate method is triggered. For your purposes, you'll be using the delegate approach in your Albums application.

Let's first take a look at the `UICollectionViewFlowLayout` class to see what you can customize. Table 5.2 shows the different properties and a description of what they do.

Table 5.2 Important `UICollectionViewFlowLayout` properties

Property	Description
`scrollDirection`	Scroll either vertically or horizontally
`minimumLineSpacing`	Line spacing between sections
`minimumInteritemSpacing`	Line spacing between items in sections
`itemSize`	Size of each cell within the collection view
`sectionInset`	Edge inset for each section
`headerReferenceSize`	Header size
`footerReferenceSize`	Footer size

By default, your scroll direction will be vertical, but you can set it to horizontal by updating the `scrollDirection` property. There are various properties for controlling the amount of space between items as well as in each section. Also, if you have a view for a header or footer within your collection view, you could specify that size as well.

When implementing the `UICollectionViewDelegateFlowLayout` protocol, you're able to update these properties dynamically. The delegate methods will be called by the collection view when they're needed, which can give you the opportunity to change it depending on your situation. For instance, you might want the line spacing to be different if your application was in portrait mode versus landscape mode. Let's examine the different delegate methods you can use within the `UICollectionViewDelegateFlowLayout` protocol.

This `collectionView:layout:sizeForItemAtIndexPath:` method will return a `CGSize` that contains the width and height of a cell at a given index path. To deal with

the space between sections, you implement `collectionView:layout:insetForSectionAtIndex:`. This method returns a `UIEdgeInsert`, which has properties for spacing for the top, left, and bottom. To determine the minimum amount of space that should be used between each row within the table view, you'd implement `collectionView:layout:minimumLineSpacingForSectionAtIndex:`. This method returns a `CGFloat`. For the minimum amount of space between each item in a section, you'd implement `collectionView:layout:minimumInteritemSpacingForSectionAtIndex:`. If you had a header or footer, you could specify its size by implementing `collectionView:layout:referenceSizeForHeaderInSection:` or `collectionView:layout:referenceSizeForFooterInSection:`.

Let's use a few of these delegate methods to change the way your photos are laid out in your collection view.

5.3.2 Using the flow layout delegate protocol

You want your photos to take up most of the space within your view. You also want separation between each photo so that it's easy to see where one photo starts and ends. It'll make it easier to see but at the same time you'll still be maximizing the amount of space you have at your disposal.

First, open IAAlbumPhotosViewController.m. To make sure that your cells are displayed with the correct size, add the following code to the bottom of your class:

```
- (CGSize) collectionView:(UICollectionView *)collectionView
                  layout:(UICollectionViewLayout *)collectionViewLayout
   sizeForItemAtIndexPath:(NSIndexPath *)indexPath
{
    return CGSizeMake(104.0f, 104.0f);
}
```

Here you're returning a `CGSize` using the `CGSizeMake` function to specify that your cells are 104 x 104. Next, you'll set the minimum spacing between each item to 2 points.

```
- (CGFloat) collectionView:(UICollectionView *)collectionView
                   layout:(UICollectionViewLayout *)collectionViewLayout
minimumInteritemSpacingForSectionAtIndex:(NSInteger)section
{
    return 2.0f;
}
```

For each row, you should apply the same spacing to keep things uniform. Add the following method:

```
- (CGFloat) collectionView:(UICollectionView *)collectionView
                   layout:(UICollectionViewLayout *)collectionViewLayout
minimumLineSpacingForSectionAtIndex:(NSInteger)section
{
    return 2.0f;
}
```

Finally, you'll adjust the inset of your collection view so that it has a little space at the top and the bottom to make the spacing you've specified between each item and each row:

```
- (UIEdgeInsets) collectionView:(UICollectionView *)collectionView
                         layout:(UICollectionViewLayout
*)collectionViewLayout
         insetForSectionAtIndex:(NSInteger)section
{
    return UIEdgeInsetsMake(2.0f, 0.0f, 2.0f, 0.0f);
}
```

Here you're returning a `UIEdgeInset` by using the `UIEdgeInsetsMake` function. You're specifying that the inset should be 2 on the top of the collection view and 2 on the bottom. Once you've finished, your code should look like what's shown in figure 5.15.

Now you can run your application and see the result! Build and run the application and choose an album with many photos. You should see your nicely laid-out collection view just like the one in figure 5.16.

Give yourself a nice pat on the back! You've created an application that can view the albums and photos on your phone. If you want to go even further, you could change the layout to display the correct proportion of each photo instead of showing a square thumbnail.

```
66
67  - (CGSize) collectionView:(UICollectionView *)collectionView
68                     layout:(UICollectionViewLayout *)collectionViewLayout
69     sizeForItemAtIndexPath:(NSIndexPath *)indexPath
70  {
71      return CGSizeMake(104.0f, 104.0f);
72  }
73
74  - (CGFloat) collectionView:(UICollectionView *)collectionView
75                     layout:(UICollectionViewLayout *)collectionViewLayout
76  minimumInteritemSpacingForSectionAtIndex:(NSInteger)section
77  {
78      return 2.0f;
79  }
80
81  - (CGFloat) collectionView:(UICollectionView *)collectionView
82                     layout:(UICollectionViewLayout *)collectionViewLayout
83  minimumLineSpacingForSectionAtIndex:(NSInteger)section
84  {
85      return 2.0f;
86  }
87
88  - (UIEdgeInsets) collectionView:(UICollectionView *)collectionView
89                           layout:(UICollectionViewLayout *)collectionViewLayout
90           insetForSectionAtIndex:(NSInteger)section
91  {
92      return UIEdgeInsetsMake(2.0f, 0.0f, 2.0f, 0.0f);
93  }
94
```

Figure 5.15 Adding a few `UICollectionViewDelegateFlowLayout` protocol methods to `IAAlbumPhotosViewController`

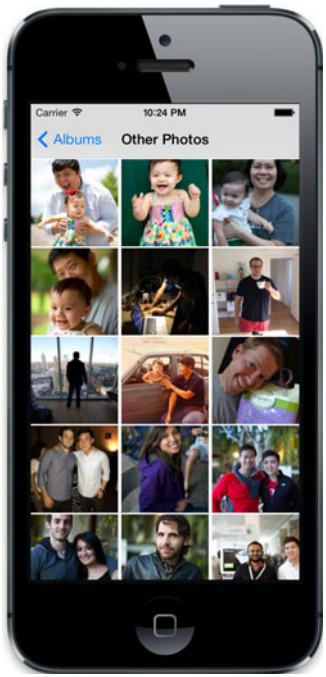


Figure 5.16 Our finished collection view with photos laid out properly in the Albums application

5.4 Summary

You've seen that collection views, in many ways, are similar to table views. Their protocol implementations are very similar when adding data and interacting with them. Their similarities end when you realize that you have much more control handling the way cells are laid out within a collection view. Throughout this chapter you saw all of these things by adding on to your Albums application, displaying photos from an album using the Assets Library framework.

- A collection view can be used when you need to display data in something more than a list of rows.
- Table views and collection views are similar in the way that you supply them with data and respond to actions.
- You have considerable control when laying out cells within a collection view when using a custom flow layout.
- Custom views for each item within a collection view are contained within a collection view cell.
- Prototype collection view cells can be reused for increased performance in your applications.
- By using collection views, you can add the ability to show photos contained within an album in your Albums application.

Part 2

Building real-world applications

After you've become familiar with the basics of iOS development, you'll be ready to start creating more advanced applications. This part of the book will arm you with the knowledge you need to start creating real-world apps. Also, in each chapter you'll be learning by creating basic versions of real-world apps.

In chapter 6 you'll learn how to retrieve and interact with remote data, which opens up a whole world of possibilities. Chapter 7 will teach you how to use the photos and videos on your iOS device to make the next big photo app.

Chapter 8 shows how to integrate with social networks such as Twitter and Facebook using what's already baked into iOS 7. Chapter 9 explores making advanced views and custom animations to make your apps really stand out.

In chapter 10 you'll learn how to use location information and to work with maps. Then in chapter 11 you'll learn about object management with Core Data to add a layer of persistence to your apps.

6 Retrieving remote data

This chapter covers

- Retrieving remote data
- Understanding data serialization and interacting with external services such as Twitter
- Sending advanced requests using iOS libraries
- Using web views to display web pages
- Using popular open source networking libraries

Mobile applications are by their mobile nature connected somehow with remote computers. Even apparently static applications need remote communications to send stats, retrieve updates, or just show the user a message. Take, for example, our Hello Time application, where we were relying exclusively on device time: what if the user moves through different time zones or the device's clock is off? In that case, we would have to communicate with a time server connected to a very accurate clock, with communication taking the form of the client (our application) sending a message using a specific protocol over the physical network to the server, waiting for the response, and, once the response is available, reading and processing it.

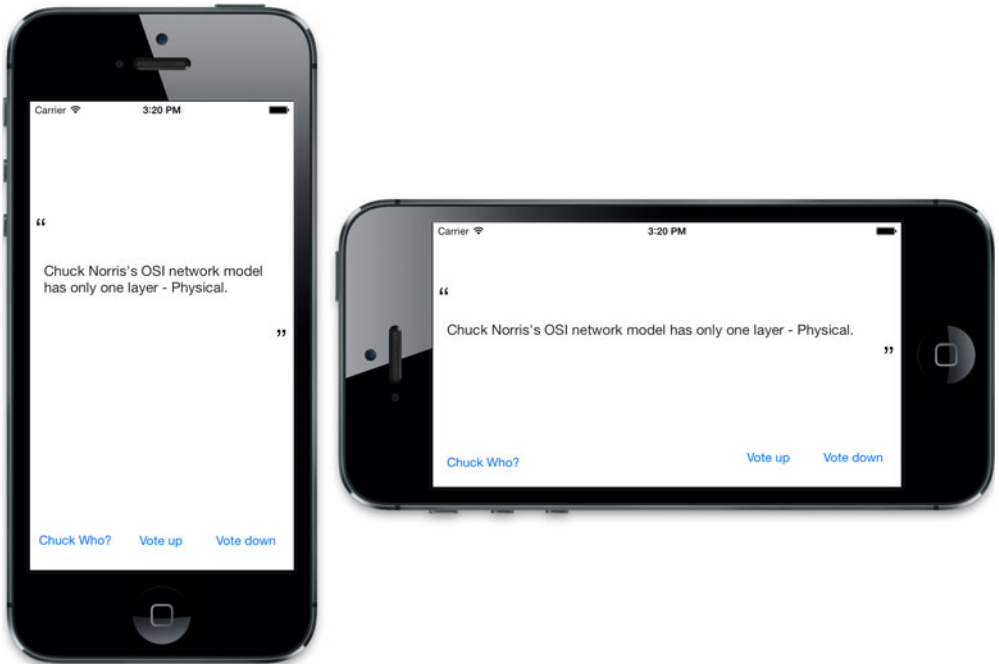


Figure 6.1 Our ChuckNorrisRater application, showing a joke, along with Vote up and Vote down buttons for liking/not liking

In this chapter, we'll cover exactly how this communication takes place. You'll learn the most common way to phrase messages (with data serialization) and how to read, parse, and process those messages. You'll learn how to present full, rich HTML pages in a snap using the native `UIWebView` object, which will allow you to easily display websites in your application. We'll then dig deeper into open source networking libraries used by most popular apps to help you implement the most common communications in a simplified way.

To demonstrate how these skills come together, we'll create an app that retrieves random jokes about Chuck Norris, using a public API, and lets its users like or unlike each joke; see figure 6.1.

6.1 Retrieving data using NSURLSession

Before we start to examine how to retrieve data using iOS native libraries in detail, it's worth devoting some time to understanding the details of how an HTTP request is made or, to put this more generally, how the HTTP communication is done.

Let's start with an example. Suppose you type `http://google.com/?q=Hello&safe=off` into your browser (this URL will search the keyword Hello on Google); it'll separate the URL as follows:

- `http` is the protocol, which tells the browser to follow the HTTP standard for the request.
- `://` separates the protocol from the domain.
- `google.com` is the domain you are retrieving the data from.
- `/` is the path of the request. It indicates the location of the resource you're trying to retrieve.
- `?` is used to separate the path from the parameters.
- `q=Hello&safe=off` are the parameters. There are two (key/value) pair parameters separated by `&`. Key `q` has `Hello` as the value, and key `safe` has `off` as the value.

In addition, when making an HTTP request, you always specify a method. The method indicates what the server should do with the information you're sending to the path defined on the URL. You'll learn more about those methods along with the full methods list in the next section. You just need to know for now that the method for retrieving information is called `GET` and that the browser will use it to retrieve your example URL.

In a nutshell, the browser will connect to google.com and it'll make a `GET` request following the HTTP protocol to `/`, passing `q=Hello&safe=off` as parameters.

Luckily for us, HTTP is an ASCII protocol, which means that we can see the request in plain text. This is what our previous example will look like after the browser finishes the parsing process we described previously:

```
GET /?q=Hello&safe=off HTTP/1.1
Host: google.com
Content-Length: 133
(…)
```

A request consists of one or more lines of ASCII text. The first word in the first line is the name of the method, followed by the path along with zero or more parameters, followed by the HTTP version. The next lines in the request are headers, which will include information about the browser capabilities and other details such as the requested host (google.com in our example) and the length of the request (`Content-Length`). Headers are separated from the body by two line breaks. In `GET` requests the body is empty.

Figure 6.2 shows the flow of an HTTP request, starting from creating a request (following the rules you just learned), then creating a connection to the remote server, and finally sending the request as plain text. Depending on the size of the response and the quality of the network, the response could take seconds, hours, or even days to arrive.

Before we go any further, let's set up the application you'll be creating throughout the chapter. Go to Xcode and create a new single-view application named ChuckNorrisRater. As you did in the previous chapters, you'll set the class prefix to `IA`. Once

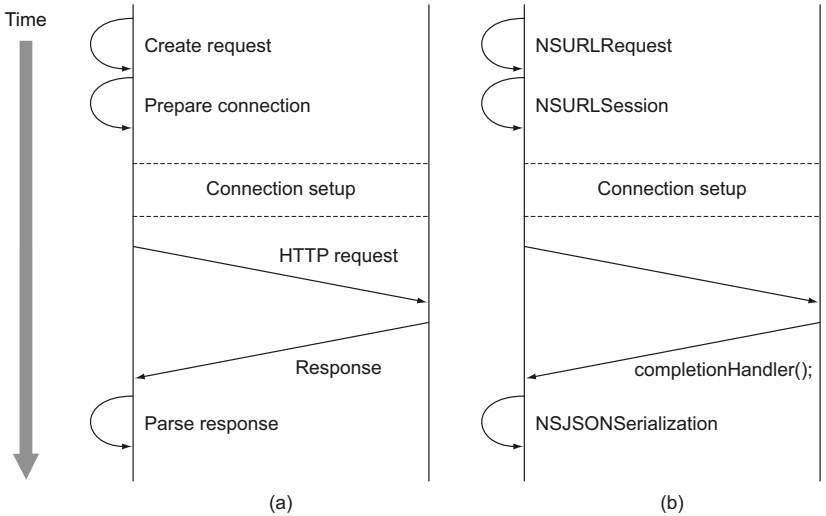


Figure 6.2 The steps in a common HTTP request: (a) a generic HTTP communication and (b) the equivalents between each step from an Objective-C standpoint

the project is created, Xcode should automatically add the files shown in figure 6.3 to your project.

Next, you'll create a new class named `IAHTTPCommunication` that you'll be using to make the HTTP communication, as shown in figure 6.4.

The next thing you'll do is create your main user interface. For this, you'll open the autogenerated storyboard file named Main.storyboard. Then you'll add a label for the title, a label for the joke, and two buttons for voting up and down. You'll do that by dragging and dropping the objects from the Object Library. Figure 6.5 shows what the interface will look like.

1. The joke is contained inside a `UILabel`. We are going to set the `Lines` attribute of this label to 0 and the height to 5 lines tall.
2. Vote up and Vote down are `UIButtons` right next to each other.
3. Two `UILabels` contain quotation marks.

Do you remember that we said that a request could take a long time to complete? For that reason you need a way to continue the execution of the main thread while the

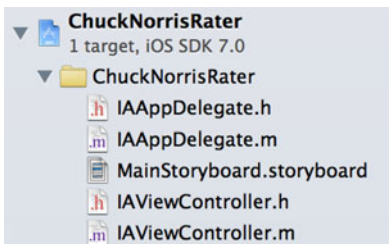


Figure 6.3 Autogenerated files when creating the new project called ChuckNorrisRater

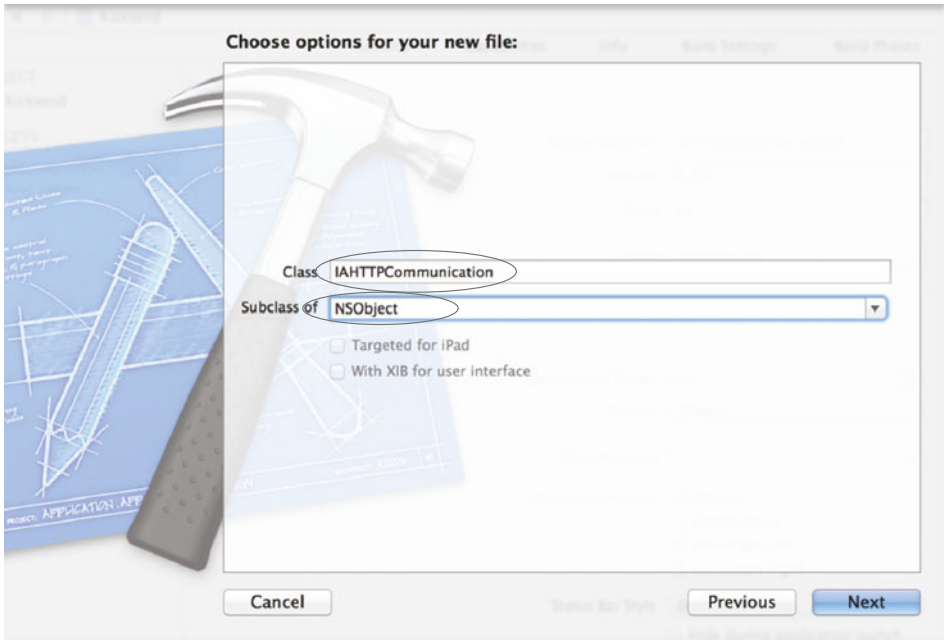


Figure 6.4 Adding the `IAHTTPCommunication` class to your project

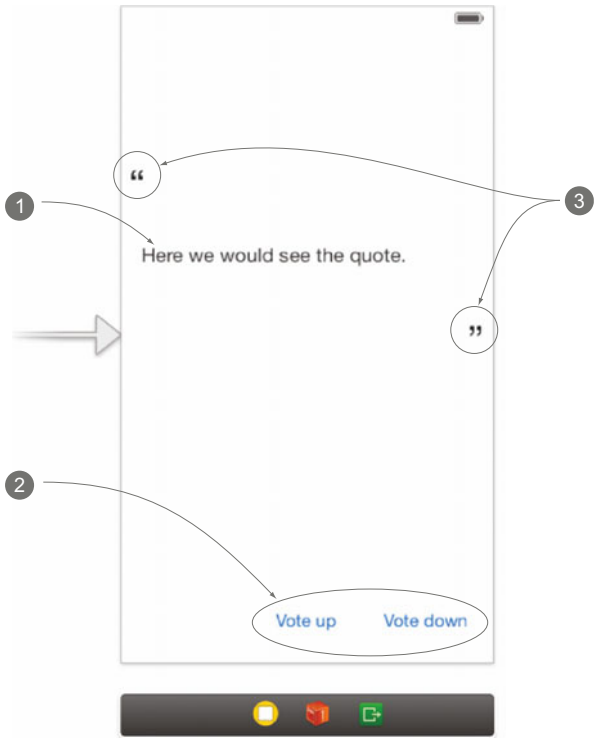


Figure 6.5 Configuring the main controller located in the application's storyboard

request is still working. This is crucial because all UI operations on iOS applications are done in the main thread, which means that if you block the main thread, you'll be blocking all touch events, graphic drawing, animations, sounds, and the like, ending up with an unresponsive application. That's why you can't just interrupt the application and wait for the response. Two techniques can resolve this issue: creating a new thread of execution to manage two simultaneous operations or configuring an instance as a delegate, as you learned in chapter 1, and implementing the methods defined in the delegate protocol.

Multithreading and concurrent operations are easily the most complex things to implement and maintain. They're useful, but if implemented incorrectly, things can go very wrong very fast. You can learn more about threads and dispatch queues (an easy and seamless way of using threads on iOS applications) in appendix A. For now, keep this in mind: when in doubt, don't use threads.

Most I/O methods in Cocoa frameworks were initially designed to use the delegation pattern to make time-consuming operations asynchronous. This means that the main thread will continue until the operation is done, at which point a specific method will be called.

You learned in chapter 1 that starting from iOS 5, Objective-C supports blocks, and now you're going to use them in your ChuckNorrisRater application, most specifically in your recently created `IAHTTPCommunication` class.

iOS 7 shortcuts

We're going to use the delegate protocol in this example so that you'll gain a deep understanding of how networking operations work. But keep in mind that iOS 7 includes a set of shortcuts that use blocks to simplify the task of doing HTTP requests.

So let's get down to business. First of all, you need to declare the instance variables you're going to use along with your `IAHTTPCommunication` class. For that, you'll use the `@interface` operator. Open the IAHTTPCommunication.m file, and on the top of the file, right after the `#imports`, add the code in the following listing.

Listing 6.1 Defining instance variables in your `IAHTTPCommunication` class

```
@interface IAHTTPCommunication ()

@property (nonatomic, copy) void (^successBlock)(NSData *);

@end
```

`successBlock` will contain the block you're going to call when the request is completed.

Next, you'll create the method in charge of the HTTP communication, as shown in the following listing. This method will retrieve jokes from a public API called `icndb` (http://icndb.com) and will return the information asynchronously using blocks.

Listing 6.2 Craft and send request using `NSURLRequest` and `NSURLConnection`

```
- (void)retrieveURL:(NSURL *)url successBlock:(void (^)(NSData
➥ *))successBlock
{
    self.successBlock = successBlock;
    NSURLRequest *request = [[NSURLRequest alloc] initWithURL:url];

    NSURLSessionConfiguration *conf = [NSURLSessionConfiguration
➥ defaultSessionConfiguration];
    NSURLSession *session = [NSURLSession sessionWithConfiguration:conf
➥ delegate:self delegateQueue:nil];
    NSURLSessionDownloadTask *task = [session
➥ downloadTaskWithRequest:request];
    [task resume];
}
```

❷ **Creating the request using the given URL**

❶ **Persisting given `successBlock` for calling it later on**

❸ **Creating a session using the default configuration and setting our instance as delegate**

❹ **Preparing the download task**

❺ **Establishing the HTTP communication**

This method takes two parameters: the URL from where you'll retrieve the content (in your application you'll use the `icndb` API URL for retrieving random jokes) and a block that you'll call once the request is completed. The first thing you need to do is store the given block for calling it later when the request is finished ❶. The next step is to create an `NSURLRequest` instance for the given URL ❷ and use this request to establish the HTTP communication ❸, ❹, and ❺. As you can imagine, `[task resume]` won't block the execution. You'll need to implement the `NSURLSessionDownloadDelegate` protocol in order to catch some of the communication events, such as when you get a new response.

The `NSURLSessionDownloadDelegate` protocol defines a series of methods that the `NSURLConnection` instance can call along the HTTP communication. In your application you'll use the most important one:

```
URLSession:downloadTask:didFinishDownloadingToURL:
```

There are two more methods that you'll probably need to implement for more complex cases such as tracking download progress or being able to resume a request. As you can see, those event names are self-explanatory:

```
URLSession:downloadTask:didResumeAtOffset:expectedTotalBytes:
URLSession:downloadTask:didWriteData:totalBytesWritten:totalBytesExpectedToWrite:
```

But these aren't the only ones. In addition the `NSURLSession` API provides three protocols that define delegate methods your app can implement to have more control over sessions. These protocols are

- `NSURLSessionDelegate`—This protocol defines delegate methods to handle session-level events such as session invalidation and credentials.
- `NSURLSessionTaskDelegate`—This protocol defines delegate methods to handle communication events such as redirects, errors, or data sending.
- `NSURLSessionDataDelegate`—This protocol defines delegate methods to handle task-level events specific to data and upload tasks.

Go ahead and implement the method that will allow you to get the data from the response in your IAHTTPCommunication.m file. `NSURLSession` will call this method once the data is available and the task has finished downloading.

Listing 6.3 Task has finished downloading delegate method

```
- (void)URLSession:(NSURLSession *)session
     downloadTask:(NSURLSessionDownloadTask *)downloadTask
     didFinishDownloadingToURL:(NSURL *)location
{
    NSData *data = [NSData dataWithContentsOfURL:location];
    dispatch_async(dispatch_get_main_queue(), ^{
        self.successBlock(data);
    });
}
```

❶ Getting the downloaded data from local storage

❷ Ensuring that you call the `successBlock` from the main thread by using dispatch queues

❸ Calling the block you stored before as a callback

This piece of code is the last step of the communication. You received the full response, and now you will call the block that you stored before. The first thing we'll do is get the locally stored data that we got from the server, as shown at ❶. Note that the location is represented as an `NSURL` instance, but at this point, the URL is just the path of a file that holds the response data and not a remote URL.

You need to be sure to call the `successBlock` callback from the main thread. This is usually a good practice because chances are that the method implementing your class is doing main-thread-specific tasks such as UI operations. You'll learn more about dispatch queues and threads in the appendix, but for now just keep in mind that line ❷ is forcing your `self.successBlock` invocation ❸ to occur on the main thread.

In some cases when retrieving remote information, the request could jump to different servers before reaching the desired destination. See, for example, the case in figure 6.6. We tried to retrieve an image located at http://t.co/, but the first response is a redirect to the server that contains the image. We need only the latest response (the actual image). Even though you can have fine control over redirects by implementing `NSURLSessionTaskDelegate`, you can let `NSURLSession` handle all these details for you, which is the default behavior.

Before moving to the next section, you'll need to expose the method `retrieveURL:successBlock:` you just created in order to be able to use it from your main

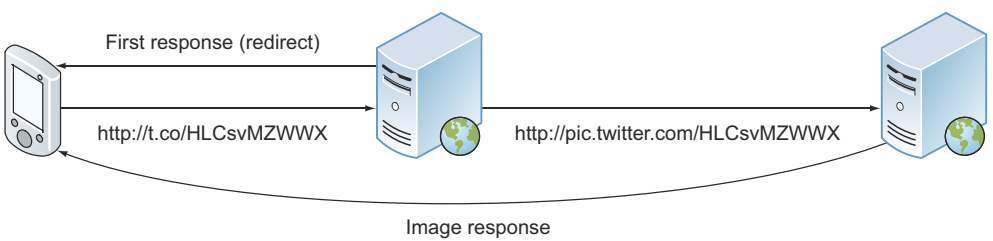


Figure 6.6 Retrieving an image from a shortened Twitter URL generates a redirect.

controller in the next chapter. For that, you'll open the file IAHTTPCommunication.h and add your method declaration, as shown in the following listing.

Listing 6.4 Exposing your `retrieveURL:successBlock:` method

```
@interface IAHTTPCommunication : NSObject <NSURLSessionDownloadDelegate>

- (void)retrieveURL:(NSURL *)url successBlock:(void (^)(NSData
➥ *))successBlock;

@end
```

Declaring your method in the .h file makes it public. The method returns `void` and takes two parameters: a URL and a block.

6.2 Understanding data serialization and interacting with external services

In the previous section, you set up your ChuckNorrisRater application and you wrote the logic to retrieve data from remote computers. You're going to retrieve random Chuck Norris jokes using an API provided by icndb.com. The icndb's API, as well as all services that provide a way for third parties to interact, has a normalized way of formatting information. Thus, you need to transform this format into something easy to use and manipulate. In other words, you need a way to convert formatted data into Objective-C objects.

Figure 6.7 illustrates how the serialization process works. The example shows the sender part (`icndb` server) on the left and the receiver part (your client) on the right. First, a joke is generated, which `icndb` saves as binary (it could be saved to a

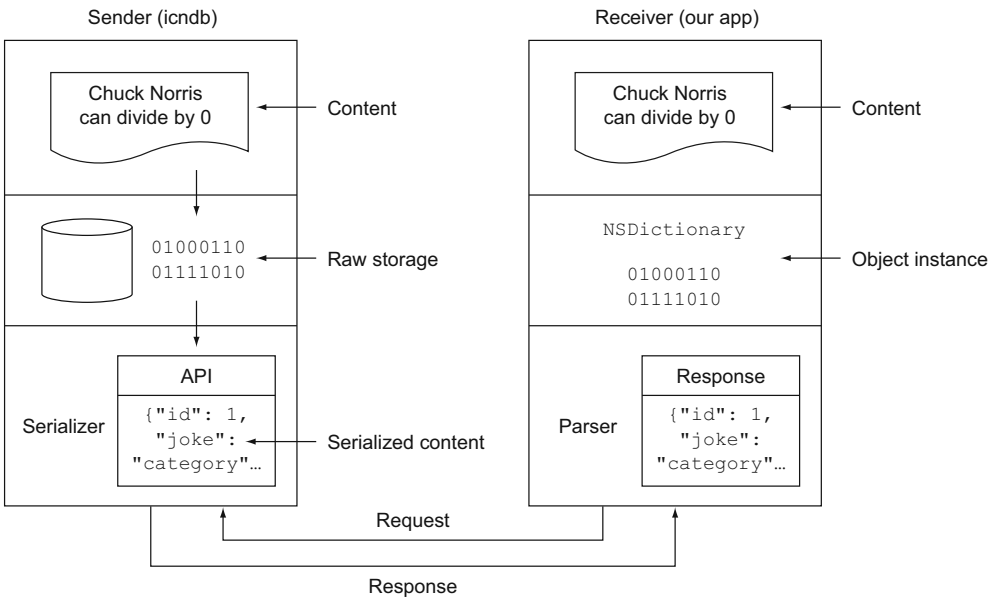


Figure 6.7 Message serialization and deserialization architecture

database, memory, file system, or any kind of storage). When a request from your application takes place, the joke information is serialized and sent to you (the receiver). Your application parses the information and converts the received data to native Objective-C objects.

There are different standard ways of exchanging information but we'll focus on one of the most commonly used serialization formats: JavaScript Object Notation (JSON).

JSON is a standard way of representing different kinds of data structures in a text-based manner. As the name implies, it's derived from the JavaScript language's syntax. JSON defines a small set of rules to represent strings, numbers, and Booleans. Even though the standard comes from JavaScript, it's used across multiple languages, and along with XML, it's one of the most used serialization methods today. Let's look at an example of JSON in action:

```
{
    "name": "Martin Conte Mac Donell",
    "age": 29,
    "username": "fz"
}
```

This fragment represents an associative array (or dictionary), which is surrounded by {} and composed of key/value pairs. Keys can't be repeated in the associative array. In our example, `name`, `age`, and `username` are keys, and `Martin Conte Mac Donell`, `29`, and `fz` are values.

Now that you know how the JSON format is defined and how the serialization process takes place, let's go back to your application. You'll implement the code for retrieving Chuck Norris jokes. First, you need to import the `IAHTTPCommunication` class you created previously and declare the instance variables you're going to use in the class. For that you'll open the file IAViewController.m, and on top of the file, right after the `#import`, you'll add the code in the following listing.

Listing 6.5 `IAViewController` instance variables declaration

```
#import "IAHTTPCommunication.h"

@interface IAViewController () {
    NSNumber *jokeID;
}
@end
```

`jokeID` is declared as `NSNumber` and it'll contain the ID of the joke you are showing.

You'll then implement the method that will retrieve the jokes, as shown in the next listing.

Listing 6.6 Received data chunk delegate method

```
- (IBAction)retrieveRandomJokes:(id)sender
{
    IAHTTPCommunication *http = [[IAHTTPCommunication alloc] init];
    NSURL *url = [NSURL URLWithString:@"http://api.icndb.com/jokes/random"];
```

1 Creating an instance of your recently created `IAHTTPCommunication` class

```
[http retrieveURL:url successBlock:^(NSData *response)
{
    NSError *error = nil;
    NSDictionary *data = [NSJSONSerialization
 JSONObjectWithData:response options:0 error:&error];
    if (!error)
    {
        NSDictionary *value = data[@"value"];
        if (value && value[@"joke"])
        {
            jokeID = value[@"id"];
            [self.jokeLabel setText:value[@"joke"]];
        }
    }
}];
}
```

Retrieving jokes using the `IAHTTP-Communication` instance ❷

Deserializing the information you got from the API ❸

Getting the information you need from the normalized response ❹

Updating the UI with the joke ❺

In listing 6.6, you define a method called `retrieveRandomJokes`. By implementing this method, you'll see how the serialization takes place from a code perspective. In that method you're using the class you created earlier (`IAHTTPCommunication`) to retrieve data from icndb.com (in this case, this data happens to be Chuck Norris jokes). That's why you're first creating an instance of `IAHTTPCommunication` ❶ and then calling the method `retrieveURL:successBlock:`, which is in charge of retrieving the data. As soon as `IAHTTPCommunication` gets a response from icndb.com, it calls the code inside the block you passed as a parameter ❷. At that point you'll have the data available and ready to be parsed.

Once the information is retrieved, you have to understand it. You need a way to convert the text you just downloaded into something that you can easily manipulate. You'll need to extract the joke and its id from the response. The process of converting serialized data (JSON in this case) to data structures is called deserialization. Luckily, starting from iOS 5, the Cocoa framework includes a class for parsing JSON. The name of the class is `NSJSONSerialization`, and as you can see in listing 6.6, parsing the response data is the first thing you do once the block is called ❸.

The response from the `icndb` API is an associative array represented using JSON as follows:

```
{
  "type": "success",
  "value": {
    "id": 201,
    "joke": "Chuck Norris was what Willis was talkin' about"
  }
}
```

Looking at this JSON, you'll see that the response represents an associative array and the `value` key contains another associative array. Once `NSJSONSerialization` does the deserialization, JSON associative arrays will be converted into Objective-C `NSDictionaries`, arrays will be converted into `NSArrays`, numbers into `NSNumbers`, and

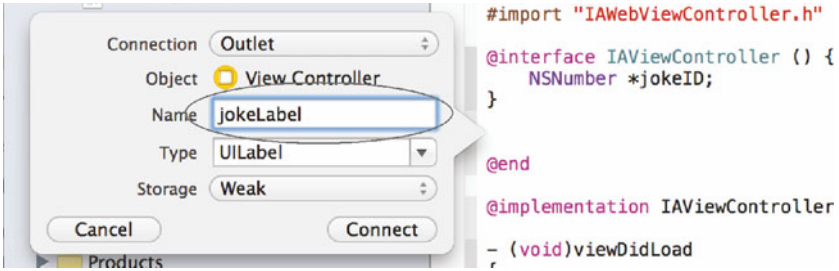


Figure 6.8 Set the name of your outlet for your joke label to `jokeLabel` and then click the Connect button.

strings into `NSStrings`. All of this leaves you with objects you can use and manipulate in your application.

Getting back to listing 6.6, after doing the deserialization, you assign the associative array located on the value key of the deserialized response to an `NSDictionary` ❹. The last step of your `retrieveRandomJokes:` method is to place the joke itself into the label ❺, for which you need to link those instance variables in Interface Builder, as shown in figure 6.8. First, open the assistant editor by going to View > Assistant Editor > Show Assistant Editor in the application menu bar. With the joke label selected in the interface, hold down the Control key while clicking and dragging from the label to your `IAViewController`'s class definition in the assistant editor. Once you've finished dragging and let go, a modal will appear asking you to name the outlet you're setting on your class for this label. Name it `jokeLabel`, as shown in figure 6.8; then click Connect.

The only thing left to do is to call the `retrieveRandomJokes:` method once the view is loaded, as follows:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    [self retrieveRandomJokes:self];
}
```

And that's it! Launch the application to see how different jokes appear each time you open the app.

6.3 Advanced HTTP requests

So far, your application has been using a method called `GET`, but other HTTP methods are available:

- `POST`
- `PUT`
- `DELETE`
- `OPTIONS`
- `HEAD`

- TRACE
- CONNECT

We'll focus on only the two most common methods: GET and POST.

GET is the simplest type of HTTP request method, and it's the one that browsers use to fetch a web page. It's used for requesting the content located at a specific URL. The content can be, for example, a web page, an image, or a song. As a convention, GET requests are read-only and according to W3C standards shouldn't be used for operations that cause changes on the server side. For example, you wouldn't use a GET request to send a form or to send a photo because those operations would need some change on the server side (as you'll see shortly, those cases will use POST).

POST submits data to be processed to the identified resource (URL). Parameters are included in the body of the request using the same format as the GET. For instance, if you wanted to post a form containing two fields, name and age, you'd send something similar to name=Martin&age=29 in the body of the request.

This way of sending parameters is widely used in web pages. The most used cases are forms. When you complete a form on a website and click Submit, chances are the request is going to be a POST.

Let's jump back to the application and use some of this knowledge. Specifically, let's use POST to rate jokes. You'll be sending votes (either +1 or -1) to a remote server.

First, you need to implement the functionality to do POST requests to the class in charge of all your HTTP operations: your IAHTTPCommunication class. To do this, in the following listing you'll add a new method, postURL:params:successBlock, which, as you'll see, is quite similar to the previous retrieveURL:successBlock method.

Listing 6.7 Method to perform POST requests

```
- (void)postURL:(NSURL *)url params:(NSDictionary *)params
➥ successBlock:(void (^)(NSData *))successBlock
{
    self.successBlock = successBlock;

    NSMutableArray *parametersArray = [NSMutableArray
➥ arrayWithCapacity:[params count]];
    for (NSString *key in params)
    {
        [parametersArray addObject:[NSString stringWithFormat:@"%@=%@",
➥ key, params[key]]];
    }

    NSString *postBodyString = [parametersArray
➥ componentsJoinedByString:@"&"];
    NSData *postBodyData = [NSData dataWithBytes:[postBodyString UTF8String]
                                          length:[postBodyString length]];
    NSMutableURLRequest *request = [[NSMutableURLRequest alloc]
➥ initWithURL:url];
    [request setHTTPMethod: @"POST"];
    [request setValue:@"application/x-www-form-urlencoded"
```

1 Creating a temporary array that will hold your POST parameters

2 Adding parameters to the temporary array as key=value strings

3 Creating a string from the parameters array containing all the parameters separated by &

4 Converting NSString to the NSData instance you'll use for the request

5 Setting the request method as POST

```
forHTTPHeaderField:@"content-type"];                    ⑥ Setting the request content-type as form encoded
    [request setHTTPBody:postBodyData];

    NSURLSessionConfiguration *conf = [NSURLSessionConfiguration
➥ defaultSessionConfiguration];
    NSURLSession *session = [NSURLSession sessionWithConfiguration:conf
➥ delegate:self delegateQueue:nil];
    NSURLSessionDownloadTask *task = [session
➥ downloadTaskWithRequest:request];
    [task resume];
}
```

⑦ Adding the POST body you created before into the request

POST request data can be structured using different formats. As you've learned, parameters are usually formatted following the form-urlencoded standard (according to the W3C's HTML standard). This format is the default and it's widely used in all web browsers. Your method takes an NSDictionary as the argument, but you can't send NSDictionaries over HTTP because that's an internal Objective-C type. To be able to send it over your HTTP connection, you need to create a comprehensible representation of it. You can think of this process as if you were trying to communicate with someone who speaks a different language. For that matter, you translate your message to a universal language, and the recipient translates the message back to their native language, as figure 6.9 illustrates. The universal language in HTTP is the W3C's standard, your language is Objective-C, and the recipient's language is unknown to you.

As we mentioned, W3C's standard indicates some rules to define what *comprehensible* means for each case. For this case, you need to represent parameters following the form-urlencoded part of the standard (for example, `param1=var1&param2=var2`).

Let's jump back to your method and see how to translate this to your code. First, you create an array containing all the (key/value) pairs ❶ and ❷, which you'll join afterward by the character &, as you can see on line ❸. The resulting string converted to an NSData instance ❹ is what you will post to the server in charge of storing the votes ❼. The way to perform a POST request using your NSURLRequest instance is by setting HTTPMethod to "POST" as well as the content-type, as you can see at ❺ and ❻.

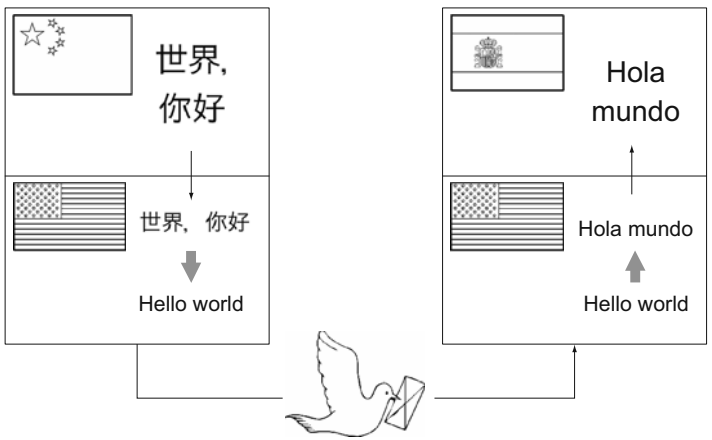


Figure 6.9 Message translation metaphor

You'll need to expose this method in order to be able to use it from your `IAHTTPCommunication` class. For that matter, you'll open the file IAHTTPCommunication.h, and right after the `retrieveURL` declaration, you'll add your new method, as shown in the following listing.

Listing 6.8 Exposing the method `postURL:params:successBlock:`

```
@interface IAHTTPCommunication : NSObject

- (void)retrieveURL:(NSURL *)url successBlock:(void (^)(NSData*))
➥ successBlock;
- (void)postURL:(NSURL *)url params:(NSDictionary *)params
➥ successBlock:(void (^)(NSData *))successBlock;

@end
```

Now that your `IAHTTPCommunication` class includes the method to perform `POST` requests, you'll need to call it from your main `IAViewController`. It's time to add the thumbs up/thumbs down touches, as shown in the next listing.

Listing 6.9 Link button touches to your `UIViewController`

```
- (IBAction)thumbUp:(id)sender
{
    NSURL *url = [NSURL URLWithString:@"http://example.com/rater/vote"];
    IAHTTPCommunication *http = [[IAHTTPCommunication alloc] init];
    NSDictionary *params = @{@"joke_id": jokeID, @"vote": @(1)};
    [http postURL:url params:params successBlock:^(NSData *response) {
        NSLog(@"Voted Up!");
    }];
}

- (IBAction)thumbDown:(id)sender
{
    NSURL *url = [NSURL URLWithString:@"http://example.com/rater/vote"];
    IAHTTPCommunication *http = [[IAHTTPCommunication alloc] init];
    NSDictionary *params = @{@"joke_id": jokeID, @"vote": @(-1)};
    [http postURL:url params:params successBlock:^(NSData *response) {
        NSLog(@"Voted Down!");
    }];
}
```

1 Creating an NSURL instance with the full server URL that you'll use for the request

2 Creating an instance of your `IAHTTPCommunication` class

3 Defining the parameters you're going to use on the `POST` request

4 Making the `POST` request and setting the `successBlock` callback block

These functions are very similar to each other. You first define the full URL ❶ that you'll use for the request, and then you create an instance of your `IAHTTPCommunication` class ❷. So far you shouldn't be surprised; that's exactly what you did before with the `GET` request. Starting from ❸ things change a little. You create an `NSDictionary` that will hold your parameters. Those parameters need to be transformed to the format you already learned (for example, `joke_id=<jokeID>&vote=1`) in order to be able to include them in the `POST` request. As you previously saw, the method in charge of doing that transformation is `postURL:params:successBlock` from your `IAHTTPCommunication` instance. The only thing left is to make the request by calling that method ❹.

Figure 6.10 Linking the "thumbs up" button touch to the `thumbUp` method

You'll link those functions to your interface. First, open the assistant editor by going to View > Assistant Editor > Show Assistant Editor in the application menu bar. Hold down the Control key while clicking and dragging from the Vote up button to your `thumbUp` method in the assistant editor, as shown in figure 6.10. Once you've finished dragging and let go, the connection will be complete.

When you've finished, repeat the process but from the Vote down button to the `-(IBAction)thumbDown:(id)sender` method.

You're all set! So far you wrote an application to retrieve jokes using the `icndb` API and the `GET` HTTP verb. You were able to show those jokes on a `UIView`, and each joke could be voted up or down. This action sends a `POST` request to a remote server that should save the vote.

6.4 Using web views to display remote pages

The previous section taught you how to make raw requests to a remote computer. That's exactly what browsers do before displaying a web page. The only difference lies in the contents of the response. Web pages are formatted using the HTML standard, which defines a bunch of rules on how to graphically represent different markup tags. Those rules seem simple, but displaying an entire web page following the complete WC3 standard is a complex task. Luckily for us, iOS comes with a component called `UIWebView`, which using the well-known WebKit engine does exactly that: it interprets HTML/CSS/JavaScript and displays full web pages inside a `UIView`.

Let's go straight to your ChuckNorrisRater application. You're going to add a feature to show the Chuck Norris Wikipedia page. It'll be triggered when a button is touched. As you learned in chapter 3, storyboards are perfect for defining navigations between two or more view controllers. Storyboards save you a lot of time and repetitive code. For that, you're going to use storyboards to navigate between the jokes view controller and the controller holding the `UIWebView` that you'll create in this chapter.

First, go to Xcode, create a new file, name the class `IAWebViewController`, and define the class as a subclass of `UIViewController`. Once the class is created, open

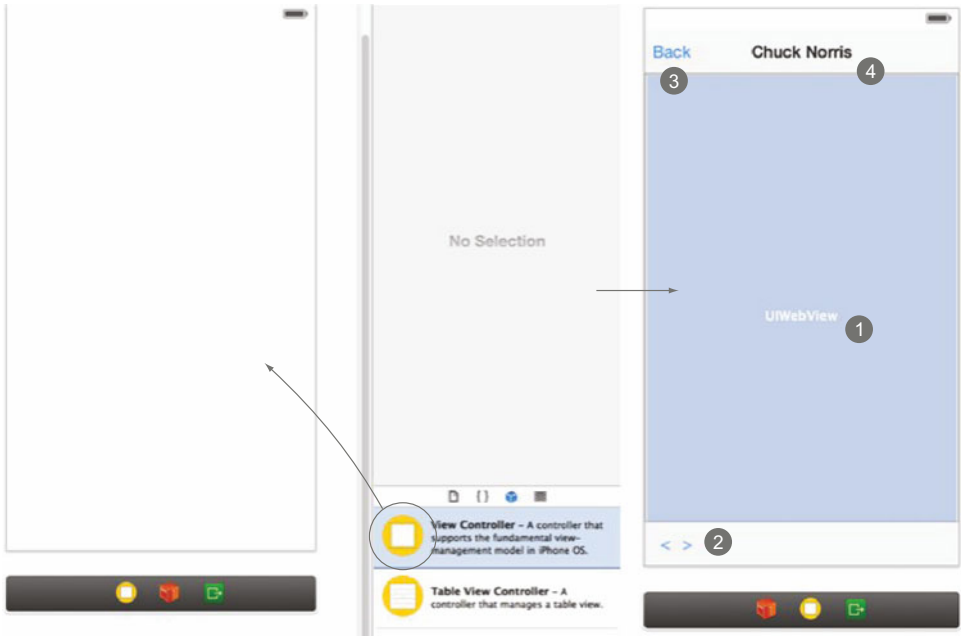


Figure 6.11 Creating the controller that will hold your `UIWebView`

the storyboard using Interface Builder, and add a new `UIViewController` to your main storyboard by dragging and dropping the object from the objects panel to the main window, as the left part of figure 6.11 shows. Then drag and drop the rest of the objects by positioning them as the right part of figure 6.11 shows.

1. Add a `UIWebView` right in the center of the view controller.
2. Add two buttons that you'll use as forward/backward link navigation controls.
3. Add a back button on top that you'll use to close the modal and come back to your main view controller.
4. Include a `UILabel` to show the title.

Now that you have your `UIWebViewController` prepared, you should pair the class you just created with the `UIViewController` you just dropped. In order to do that, select the view controller on Interface Builder and change the Custom Class property from the identity inspector, as shown in figure 6.12.

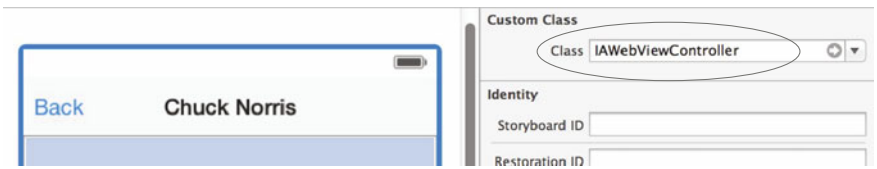


Figure 6.12 Setting the `UIViewController` as an instance of `IAWebViewController`

You now have a new view controller on your storyboard, but it's not connected to your main view. In order to address that, the next thing you'll do is add a button to your main interface that will work as the trigger to show the web page and create a segue from that button to the controller holding your `UIWebView` by following the steps illustrated in figure 6.13.

1. Hold down the Control key while clicking and dragging from the button to the new `UIViewController`. Once you've finished dragging and let go, a popup with three options will appear.
2. Select modal on the popup.

In order to control your `UIWebView` you'll need to connect it from the interface to your code. For that, open the assistant editor by going to View > Assistant Editor > Show Assistant Editor in the application menu bar. With the `UIWebView` control selected in your interface, hold down the Control key while clicking and dragging from the control to your `IAWebViewController`'s class definition in the assistant editor. Once you've finished dragging and let go, a modal will appear asking you to name the outlet you're setting on your class for this label. Name it `webView` and then click Connect.

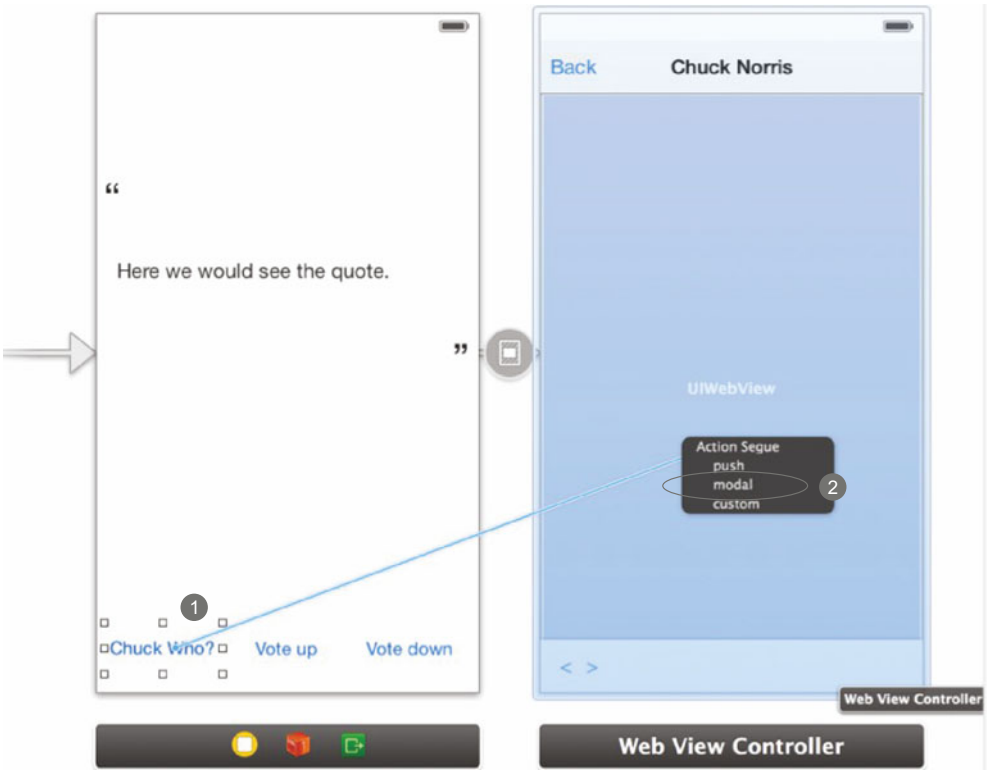


Figure 6.13 Setting a button as the trigger for showing your new `UIWebViewController`

You have everything hooked up from a user-interface standpoint; next, you'll implement the code to show Chuck Norris's Wikipedia page in the view controller you just created. For that you need to make two changes on the `IAViewController` class you created previously.

When the user touches the button, the storyboard will perform the segue you created and the Wikipedia page will be shown. You'll show the content (technically, it's the graphical representation of the content) in your `UIWebView`, and for that you'll implement the `-(void)viewDidLoad` method, which will be called as soon as the `UIViewController` finishes loading the main view, as shown in the following listing.

Listing 6.10 Load and display the web page inside your `UIWebView`

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    NSURLRequest *request = [NSURLRequest requestWithURL:
➥ [NSURL URLWithString:@"http://en.wikipedia.org/wiki/Chuck_Norris"]];
    [self.webView loadRequest:request];
}
```

❶ Creating an `NSURLRequest` object with the Chuck Norris Wikipedia URL

❷ Performing the request by using the `webView` instance

You create the request as you did before ❶, but instead of sending the request using `NSURLSession`, you use the `UIWebView`'s method `loadRequest:` ❷, which will do all the work for you.

Finally, you need to implement the Back, Forward, and Close buttons. Because the three methods are very straightforward, you'll go ahead and implement them all in the next listing.

Listing 6.11 Close method implementation. Just dismiss the view controller.

```
- (IBAction)close:(id)sender
{
    [self dismissViewControllerAnimated:YES completion:nil];
}

- (IBAction)back:(id)sender
{
    [self.webView goBack];
}

- (IBAction)forward:(id)sender
{
    [self.webView goForward];
}
```

❶ Dismissing view controller from screen when `close` method is called

❷ Loading the previous location in the back-forward list when Back button is touched

❸ Loading the next location in the back-forward list when Forward button is touched

The first method ❶ just dismisses the modal (remember that you connected both controllers in your storyboard by selecting the modal option). The second and third methods ❷ and ❸ are used to navigate the browsing history backward and forward.

Now that you're all set, it's important to know a few other things about `UIWebView`. There are some cases where you'll want to control the flow of the navigation. For example, you might want to know when specific content or a specific URL is loaded.

Figure 6.14 Web view inside our view controller

Or perhaps you're implementing a child-safe browser; you might want to block the user from loading pages matching some criteria such as sex or drugs. For all those kinds of customizations, you'll set an instance that implements the `UIWebViewDelegate` protocol as the delegate of your `UIWebView`. The methods you could implement are these:

- `webView:shouldStartLoadWithRequest:navigationType:`
- `webViewDidStartLoad:`
- `webViewDidFinishLoad:`
- `webView:didFailLoadWithError:`

With the first method you can control the navigation flow by allowing or blocking specific requests. The other three methods are informative events (the names of the methods will give you a good idea of the event).

You're all set! Figure 6.14 gives a sneak peak of what the web view should look like in your app.

6.5 Popular open source networking libraries

Lots of open source libraries make the process of retrieving data and interfacing with external services such as Twitter a very easy task. We'll mention a couple of those along with examples.

6.5.1 AFNetworking

Sometimes NSURLSession could, at first glance, be a little confusing. AFNetworking is a framework for iOS that helps you in the process of creating applications that use remote communications by making networking in iOS a joy. It's built on top of NSURLSession and NSOperations, it supports blocks, and it serializes the response automatically. But more important, the community is very active. Big companies such as Parse, Heroku, Pinterest, and Simple use AFNetworking in their apps.

Take, for example, the code in the following listing. It will retrieve your external IP by making a GET request to a remote service that returns the IP formatted as JSON.

Listing 6.12 AFNetworking GET example

```
NSURL *url = [NSURL URLWithString:@"http://httpbin.org/ip"];
AFHTTPSessionManager *manager = [[AFHTTPSessionManager alloc]
➥ initWithBaseURL:URL];
manager.responseSerializer = [AFJSONSerializer serializer];
[manager GET:@"/resources" parameters:nil success:^(NSURLSessionDataTask
➥ *task, id responseObject) {
    NSLog(@"IP Address: %@", [JSON valueForKeyPath:@"origin"]);
} failure:nil];
```

You first create the request object the same way you did on your IAHTTPCommunication class. ❶

You configure the manager response serializer as JSON. ❷

Using the request object and a block, you create a request operation. When the request is finished, your block will be called, passing the deserialized response. ❸

Authors:
Mattt Thompson @mattt
Scott Raymond @sco
Project page: http://afnetworking.com
License: MIT license

6.5.2 RestKit

When we refer to RESTful services, we're referring to API services created in such a way that instead of having a dozen methods (for example, http://example.com/user/1/create, http://example.com/user/1/delete, http://example.com/user/1/update…), they have four standardized methods (create, retrieve, update, destroy) for the same resource (for example, http://example.com/user/) using the following HTTP verbs:

- GET
- PUT
- POST
- DELETE

REST is becoming very popular, and big companies such as Facebook, Twitter, and GitHub are implementing their APIs following REST principles.

RestKit aims to make interacting with RESTful web services simple. It's built on top of NSURLConnection and makes it very easy to implement an entire RESTful service. It has

serialization logic built in, which means that you'll get responses in native Objective-C objects. It also supports Core Data for caching responses. The following listing shows an example using `GET`.

Listing 6.13 RestKit `GET` example

```
[objectManager getObjectsAtPath:@"/status/user_timeline/RestKit"
                     parameters:nil
                        success:^(RKObjectRequestOperation *operation,
 ➥ RKMappingResult *mappingResult)
{
    NSArray* statuses = [mappingResult array];
    NSLog(@"Loaded statuses: %@", statuses);
}
                        failure:^(RKObjectRequestOperation *operation,
 ➥ NSError *error)
{
    NSLog(@"Hit error: %@", error);
}];
```

Author: Blake Watters @blakewatters
Project page: http://restkit.org/
License: Apache License, version 2.0

6.6 Summary

Throughout this chapter, we've created an application that shows Chuck Norris facts retrieved using the icndb.com API. The application is able to send votes to a remote server using `POST` requests, and it displays the Chuck Norris Wikipedia page using a `UIWebView`. With this understanding of data retrieval you're now capable of implementing an application that receives (and sends) information from (and to) a remote server. Some key topics we covered are these:

- Message serialization converts formatted text into Objective-C instances.
- JSON is a standard way of representing different kinds of data structures in a text-based manner.
- Cocoa frameworks include classes such as `NSURLRequest` and `NSURLSession` that support simple and complex HTTP requests.
- Including web pages inside your applications is very easy thanks to the `UIWebViewController` class.
- Open source libraries can help make HTTP communications very easy and require fewer lines of code.

7 Photos and videos and the Assets Library

This chapter covers

- Using the image picker controller
- Retrieving assets from the Assets Library
- Capturing photos and videos with the image picker
- Retrieving and displaying asset metadata

You created an application using the Assets Library framework when you created the Albums app. This application allowed you to view the photos contained within albums managed by the Photos app on iOS. Truth is, you only scratched the surface of what you can do with the Assets Library in the Albums app. You only retrieved albums and photos so that you could use them to be displayed within a table and collection view.

By the end of this chapter you'll be able to capture photos or videos using the camera, use the built-in photo picker, and access the detailed bits of information contained within each photo or video. Some of this information includes EXIF information for photos and duration and location information for videos. Together we'll be using this knowledge to put together an app called Media Info that will allow you to do all of these things; you can see it in action in figure 7.1.

Figure 7.1 Your finished Media Info app will allow you to choose and capture media and view detailed asset information.

We'll look at an extended overview of the Assets Library, and then we'll set up the foundation for your new application.

7.1 Overview of the Assets Library framework

The Assets Library framework allows you to access and modify the photos and videos that are managed by the Photos application in iOS. Any app that you've ever used that accesses or creates photos or videos on your iOS device is doing so by using the Assets Library framework. Within this framework, one object represents your whole Assets Library. Thankfully, it's named quite conveniently as `ALAssetsLibrary`. A single instance of this `ALAssetsLibrary` object may contain many albums and also may contain photos or videos. Albums are represented by an `ALAssetsGroup` instance. Photos or videos are represented by an `ALAsset` instance. Take a look at figure 7.2, which shows this relationship.

Let's examine these three different classes a little more closely. We'll look at how they relate to one another and how you use them to find just what you're looking for.

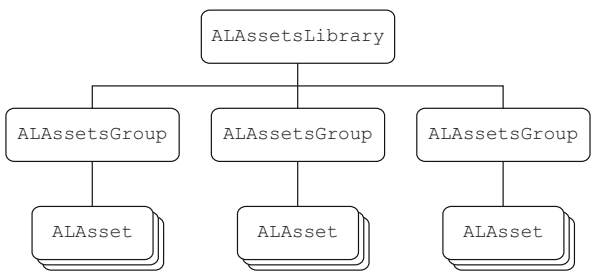


Figure 7.2 An `ALAssetsLibrary` instance may contain multiple `ALAssetsGroup` instances, which may also contain instances of `ALAssets`.

7.1.1 The Assets Library, groups, and individual assets

There are more than just three classes within the Assets Library framework, but these are the most important ones that you'll be using most often. We'll do a quick overview of each of them but won't go too far because you'll continue to learn more about them when you build your application. We'll start with the gatekeeper to your assets, the `ALAssetsLibrary` class.

ALAssetsLibrary

In chapter 4 you created a singleton instance of the `ALAssetsLibrary` that was used throughout the Albums application. This is because each `ALAssetsGroup` and `ALAsset` that you use has a parent `ALAssetsLibrary` instance. If this `ALAssetsLibrary` instance is somehow deallocated or unavailable, you'll get an exception when you try to access any of its children.

The `ALAssetsLibrary` represents the parent of all of your photos and videos and the albums that they belong to. To be able to retrieve your photos or videos you must first iterate through all of the albums represented as `ALAssetsGroup` instances. You can do this using the `enumerateGroupsWithTypes:usingBlock:failureBlock:` method shown in the following listing.

Listing 7.1 Enumerating through all groups within the Assets Library

```
[assetsLibraryInstance enumerateGroupsWithTypes:ALAssetsGroupAll
                                     usingBlock:^(ALAssetsGroup *group,
➥ BOOL *stop)
{
    if(stop)
      // Finished enumerating through groups
    else
      // Do something with 'group'
}
                                   failureBlock:^(NSError *error)
{
    // Handle error
}];
```

Notice that you are choosing to enumerate through groups with type `ALAssetsGroupAll`. You can change this to specify different types of albums represented by the `ALAssetsGroupType` enumerable type, which are listed in table 7.1.

Table 7.1 Different types of `ALAssetsGroupType` you can filter on

Type	Description
`ALAssetsGroupLibrary`	Library group that contains all assets synced from iTunes
`ALAssetsGroupAlbum`	All albums created on device or synced from iTunes
`ALAssetsGroupEvent`	All albums created as events from importing photos
`ALAssetsGroupFaces`	All albums with detected faces synced from iTunes

Table 7.1 Different types of `ALAssetsGroupType` you can filter on *(continued)*

Type	Description
`ALAssetsGroupSavedPhotos`	All photos saved to the camera roll
`ALAssetsGroupPhotoStream`	All assets available through Apple's Photo Stream
`ALAssetsGroupAll`	All albums; combination of all of the above

The first time you try to access a user's assets, the user will be prompted with an alert that asks them for their permission. Without their permission you will be unable to access any photos or videos. For example, an error will be returned within the `failureBlock` parameter in the function defined previously. Once permission has been denied, you won't be able to prompt them again unless they uninstall and then reinstall the application. The only way to enable access from this point forward is to do so within the Privacy section of the Settings application.

The `ALAssetsLibrary` class is also used to save photos and videos to the Assets Library. These assets can also be saved to any album that you choose. You'll be doing this later when you save new photos to your camera roll. Next, let's take a look at what represents each album in your Assets Library.

ALAssetsGroup

An assets group represents an ordered set of assets within the Assets Library. They're most commonly referred to as albums, which is how they're represented within the Photos application.

You can access different properties, such as the name, group type, and URL of a group, which can be retrieved using the `valueForProperty:` method. A set of convenient `NSString` constants, property keys, can be passed into this parameter to retrieve the property you want. These are shown in table 7.2.

Table 7.2 Property keys representing different properties for an `ALAssetsGroup`

Property	Description
`ALAssetsGroupPropertyName`	The name of the asset group
`ALAssetsGroupPropertyType`	The `ALAssetGroupType` for a specific group
`ALAssetsGroupPropertyPersistentID`	An ID that identifies a group
`ALAssetsGroupPropertyURL`	A URL that uniquely identifies a group

Let's see how you'd use this to retrieve the name of an assets group. Assuming you have an assets group instance variable named `assetsGroupInstance`, you'd do the following:

```
NSString *name = [assetsGroupInstance
➥ valueForProperty:ALAssetsGroupPropertyName];
```

This would return the name of the assets group as an `NSString`. You could do this using the constants mentioned earlier to retrieve the other properties. You can also determine if there are any assets within a group by retrieving the number of assets it contains by using the `numberOfAssets` instance method.

How about retrieving photos and videos within this group? You can retrieve all of its assets, the photos and videos, by using the `enumerateAssetsUsingBlock:` method. In your Albums application you used this method to retrieve all of the assets from each album that you chose to display. You can see an example of this in the next listing.

Listing 7.2 Enumerating assets within an assets group

```
[assetsGroupInstance enumerateAssetsUsingBlock:^(ALAsset *result,
➥ NSUInteger index, BOOL *stop)
{
    // Do something with 'result'
}];
```

Within this block you can choose to store the assets within an array and use them however you like. Let's look at these assets, represented by the `ALAsset` class.

ALAsset

The `ALAsset` class is used to represent the individual photos and videos within the Photos application. This class allows you to retrieve the original asset for use in your application as well as multiple representations of photos. For instance, photos can have the original representation, an edited version, thumbnails, and more.

To retrieve a square thumbnail you can use the `thumbnail` property, which returns a `CGImageRef`. By using this method you can then generate a `UIImage` using the `imageWithCGImage:` class method as shown here:

```
[UIImage imageWithCGImage:asset.thumbnail];
```

If you wanted to return a thumbnail that retained the same aspect ratio of the original asset, you could use the `aspectRatioThumbnail` method instead. You can also retrieve different individual properties using the `valueForProperty:` method. Table 7.3 shows the different properties you can access on an `ALAsset`.

Table 7.3 Property keys representing different properties for an `ALAsset`

Property	Description
`ALAssetPropertyType`	Type of asset: a photo, video, or unknown
`ALAssetPropertyLocation`	Location representation with latitude and longitude
`ALAssetPropertyDuration`	Total play time for a video asset
`ALAssetPropertyOrientation`	Number representing the orientation of a photo
`ALAssetPropertyDate`	Creation date of the asset

Table 7.3 Property keys representing different properties for an `ALAsset` *(continued)*

Property	Description
`ALAssetPropertyRepresentations`	Array of different types of representations of an asset
`ALAssetPropertyURLs`	Array of URLs for each representation
`ALAssetPropertyAssetURL`	Unique identifier URL for asset

Using the `ALAssetPropertyType` property key you can determine whether an asset is a photo or a video. You can check this by doing what's shown in the following code example:

```
NSString *assetType = [result valueForProperty:ALAssetPropertyType];
if ([assetType isEqualToString:ALAssetTypePhoto])
    // Photo
else if ([assetType isEqualToString:ALAssetTypeVideo])
    // Video
else
    // Unknown
```

By retrieving the property type, you can compare to see if it's equal to `ALAssetTypePhoto` or `ALAssetTypeVideo`. These two constants are `NSString`s, so you compare them using `isEqualToString:`.

With this overview of the Assets Library, groups, and assets, you're ready to get started setting up your new application.

7.1.2 Setting up the Media Info project

Together we'll be building a new app called Media Info that will allow you to examine assets and make extensive use of the Assets Library framework. Fire up Xcode and create a new single-view application named Media Info, as shown in figure 7.3.

Once you've created the project, head over to the project settings and click the General tab. You're going to add the Assets Library framework to your project so you can use it in your application. Within this tab find the Linked Frameworks and Libraries section and expand it so that you can add a new framework. Click the + button shown in figure 7.4 to open the dialog that will list available frameworks.

Once the dialog appears, search for "AssetsLibrary.framework." Once you've found it, select it and then click the Add button to add it to your project. This is shown in figure 7.5.

While you're still on this screen also add MobileCoreServices.framework. You'll be using this framework to identify certain media types later in the chapter.

Now that you have the framework hooked into your application, you'll add a new view controller. Your application will consist of two view controllers, one of which has already been added for you when you created the Xcode project. This second view controller will be used for displaying one asset and its detailed information.

Create a new Objective-C class within the Media Info folder in your project. Name it `IAAssetInfoViewController` and make it a subclass of `UIViewController`. Now add

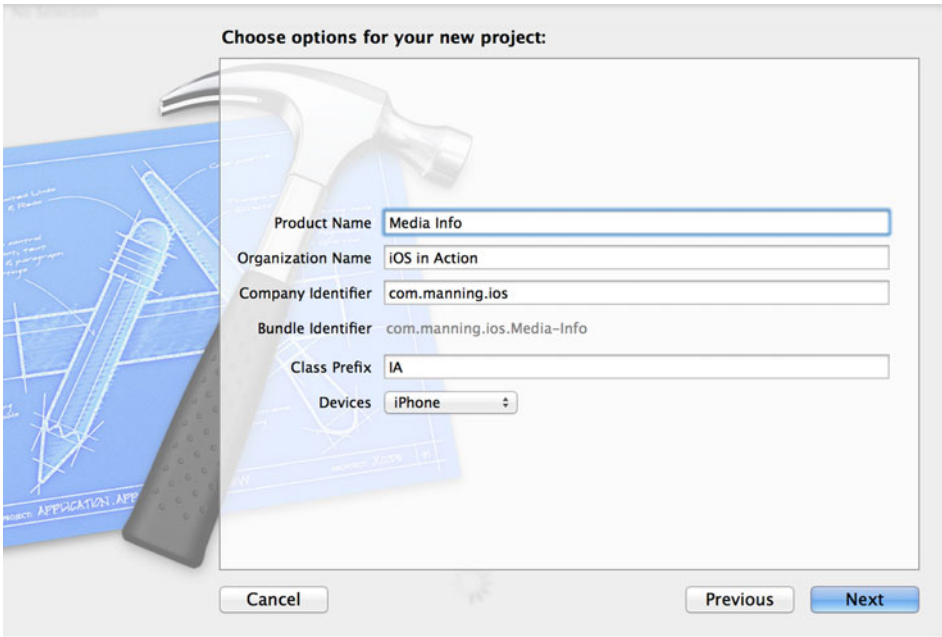


Figure 7.3 Creating the Media Info project using the Single View Application template

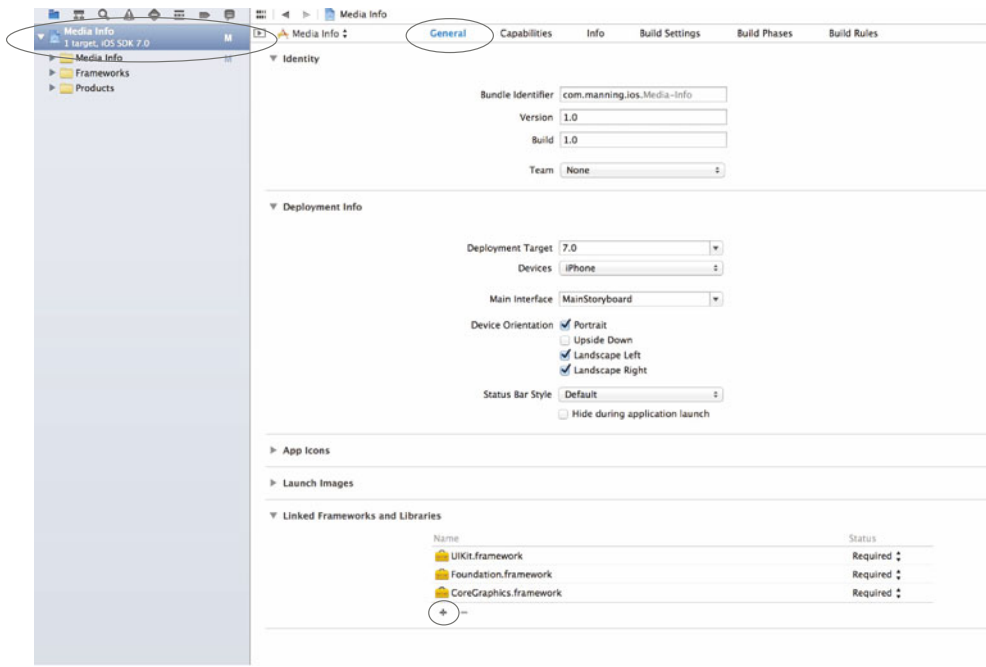


Figure 7.4 Go to the General tab within the project settings to add the Assets Library framework to your project.

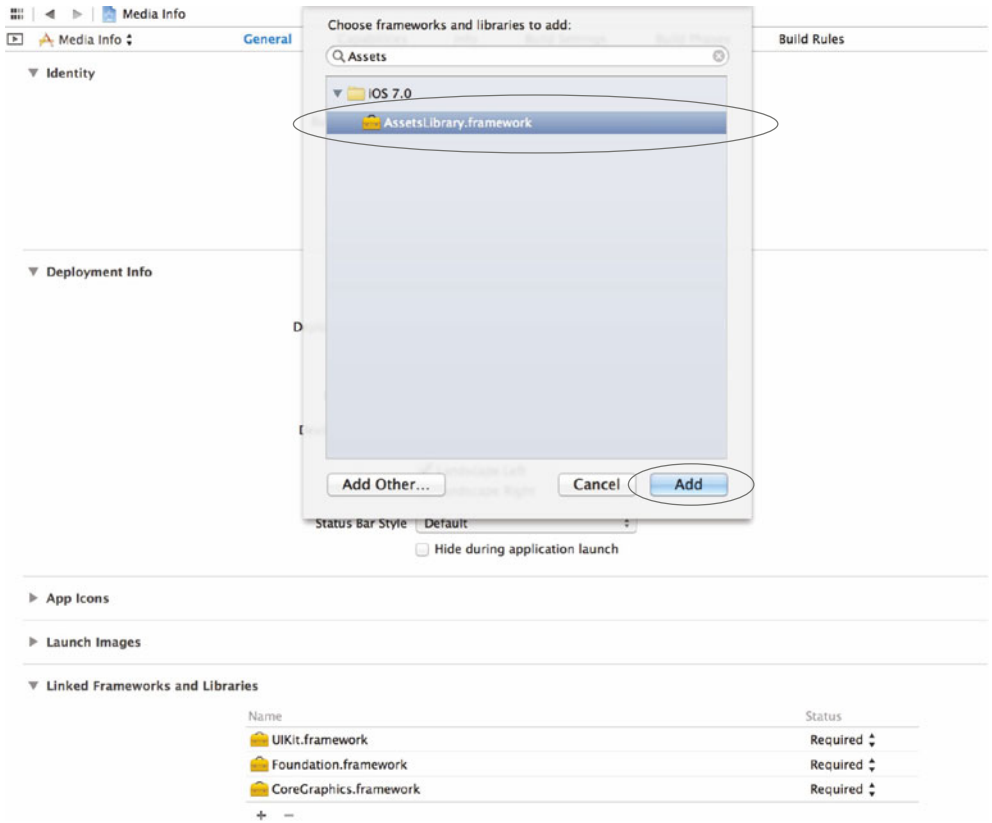


Figure 7.5 Search for "AssetsLibrary.framework" and add it to your project.

it to your storyboard by first opening Main.storyboard. Next, go to the Object Library and find a View Controller, and drag it to the right of your existing scene, as shown in figure 7.6.

One very important step is to set the identity of this view controller to `IAAsset-InfoViewController` by going to the identity inspector and populating the Class field. Next, select your first view controller scene. You'll embed this in a navigation controller because you want to be able to transition back and forth from this scene to the one that you just added. With the first scene selected, go to the application menu and choose Editor > Embed In > Navigation Controller; the result is shown in figure 7.7.

The last piece of groundwork you'll need to do before proceeding further is to add a new push segue from your first scene to your newly added one. Control-click from your first scene and drag a connection to your second scene. Once the segue has been created, select it and hop into the attributes inspector. Name this segue `assetView`, as shown in figure 7.8.

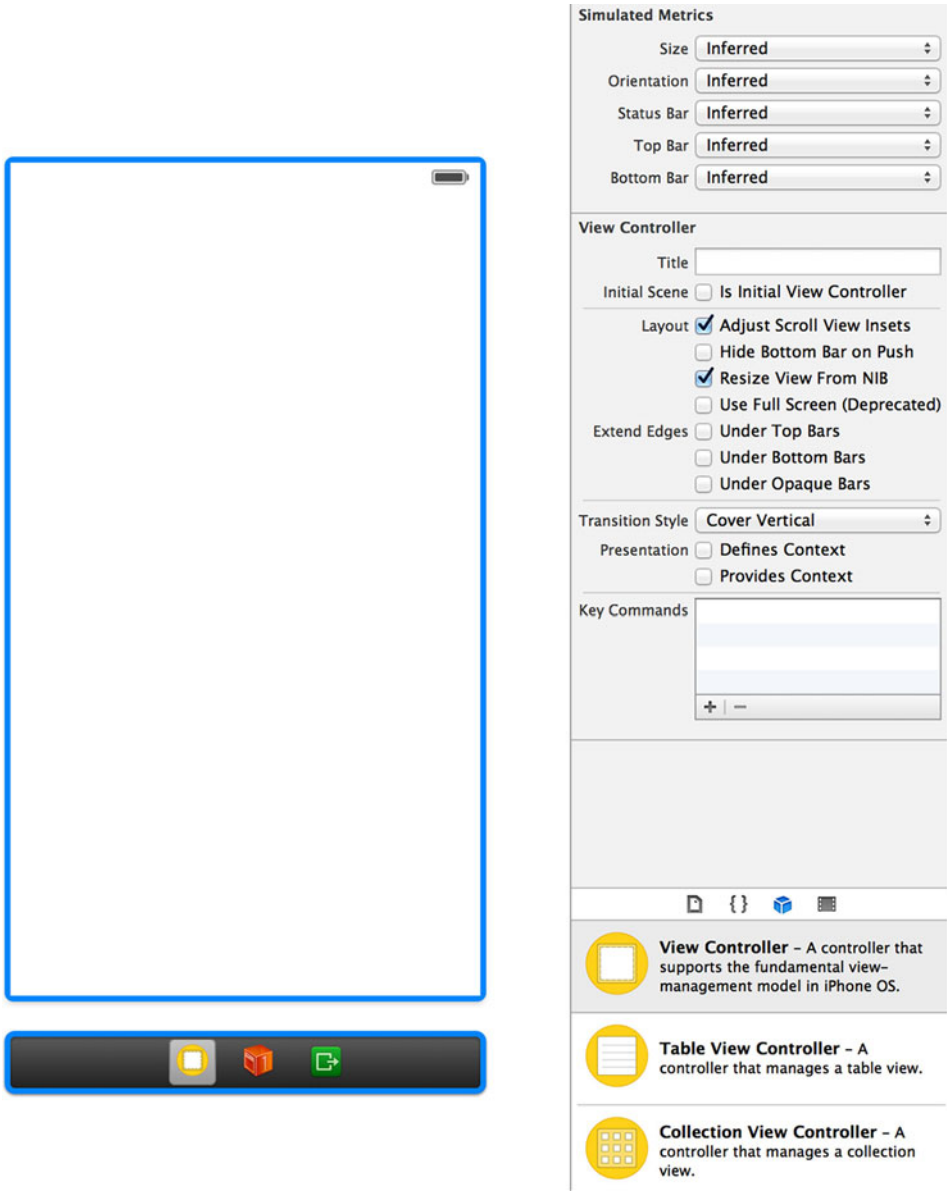


Figure 7.6 Add a new scene to your storyboard to represent the `IAAssetInfoViewController`.

Last, you'll create a new class called `IAAssetsLibrary`, which will act as your Assets Library singleton instance. This will be the same as what you used when you created your Albums application. Go to the application menu and choose File > New File. Select Objective-C class and create the class named `IAAssetsLibrary` as a subclass of

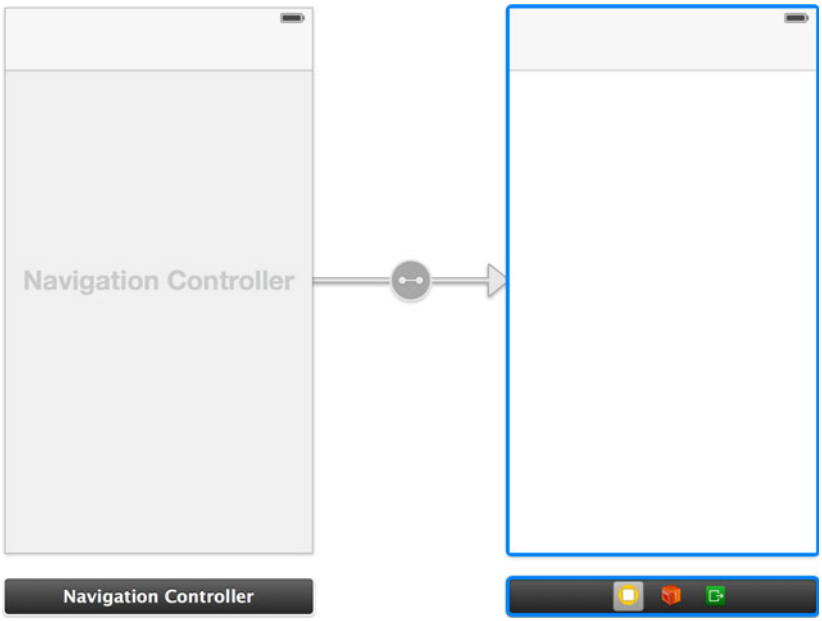


Figure 7.7 Embed the first view controller in a navigation controller.

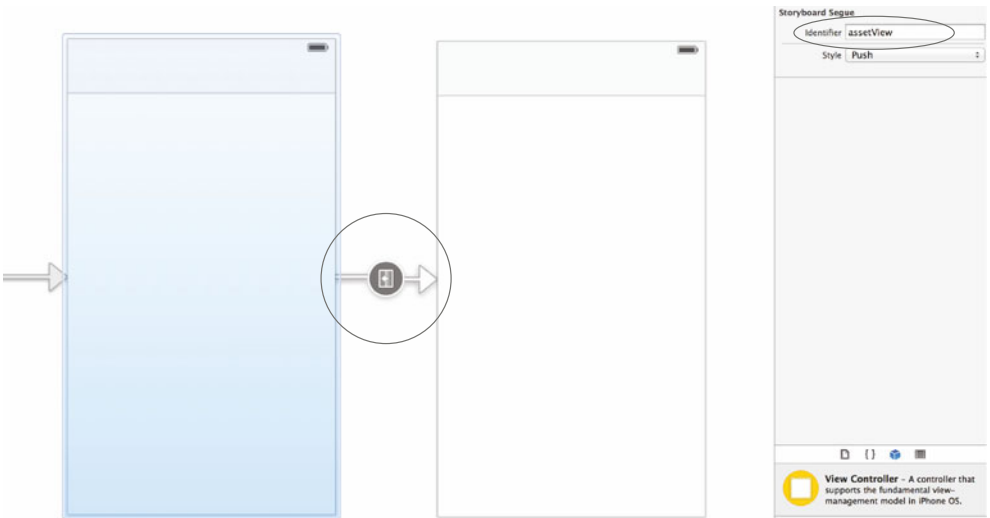


Figure 7.8 Add a new push segue from your first scene to your second. Name the segue identifier `assetView` within the attributes inspector.

ALAssetsLibrary. Replace the contents of IAAssetsLibrary.h with the code shown in the following listing.

Listing 7.3 IAAssetsLibrary.h

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

@interface IAAssetsLibrary : ALAssetsLibrary

+ (IAAssetsLibrary *) defaultInstance;

@end
```

Now open IAAssetsLibrary.m and replace its contents with this code.

Listing 7.4 IAAssetsLibrary.m

```
#import "IAAssetsLibrary.h"

@implementation IAAssetsLibrary

+ (IAAssetsLibrary *) defaultInstance
{
    static IAAssetsLibrary *singleton = nil;
    static dispatch_once_t oncePredicate;
    dispatch_once(&oncePredicate, ^
    {
        singleton = [[super alloc] init];
    });

    return singleton;
}

@end
```

You're now finished with the initial setup of your Media Info application and can get started using different pieces of the Assets Library framework to add in the features you need. You'll start by retrieving existing photos and videos using the image picker controller.

7.2 Retrieving photos and videos with the image picker

Most applications use the default image picker controller (UIImagePickerController) to let users pick photos or videos that they want to use. A handful of others go the other route by creating their own picker. The reason for this is that the default image picker doesn't allow for much customization. You can select only one photo or video at a time, and you can't do much to change the way it looks. In figure 7.9 you can see how the image picker is used when adding a new photo to a contact in the Contacts application.

You'll be sticking with the default picker because you'll be using it similarly to how it's shown in figure 7.9. You'll prompt a user to choose a single photo or video and

Figure 7.9 How the image picker is used to add a photo to a contact within the Contacts application

that's all, although you'll soon learn all of the different properties you can set to add some customization to how a photo or video is chosen.

7.2.1 Preparing and presenting the image picker controller

The view controller that is shown when you launch your application is the `IAViewController` class that was created for you when you created your project. You'll continue to use this as the main view that will present your users with two different actions. The first action is the one you'll be focusing on in this section. This will be a button with the label Choose Photo or Video, which will launch an image picker controller.

Add this button to the scene within your storyboard by opening Main.storyboard. Go to the Object Library and find a button, drag it to the top of `IAViewController`'s view, and change its title to Choose Photo or Video. Take a look at it in figure 7.10.

While doing this, go ahead and change the title in the navigation bar to Media Info, which is also shown in figure 7.10.

With the button positioned in your view, you can create an action that will be triggered to launch the image picker when touched. Open the assistant editor to bring up IAViewController.h and drag a connection to create a new action called `launchImagePicker`, as shown in figure 7.11.

Your work with the storyboard is finished for now. Hop into IAViewController.h to bring up your class's interface.

CONFORMING TO PROTOCOLS

You have to specify that your view will act as a delegate to the `UIImagePickerController`. The image picker will inform the delegate when a photo is picked or when it has been canceled. It's your job to specify that your class conforms to the

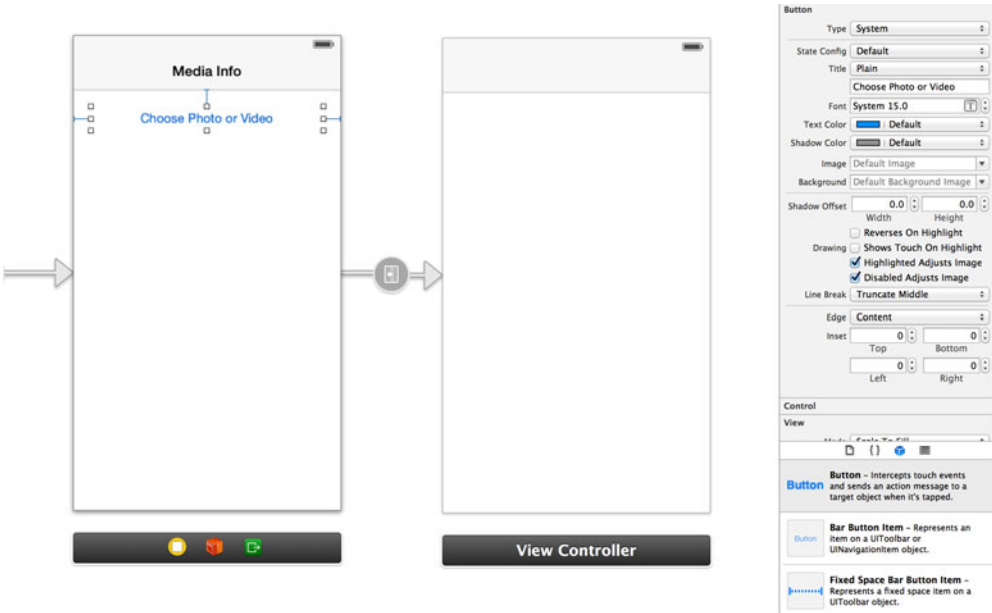


Figure 7.10 Add a button to `IAViewController` with the label Choose Photo or Video.

`UIImagePickerControllerDelegate` protocol by adding it to your interface definition shown in the code snippet at the top of the following page. Also, the image picker requires that you conform to the `UINavigationControllerDelegate` protocol. You have to specify it, although you won't be digging into that within this application.

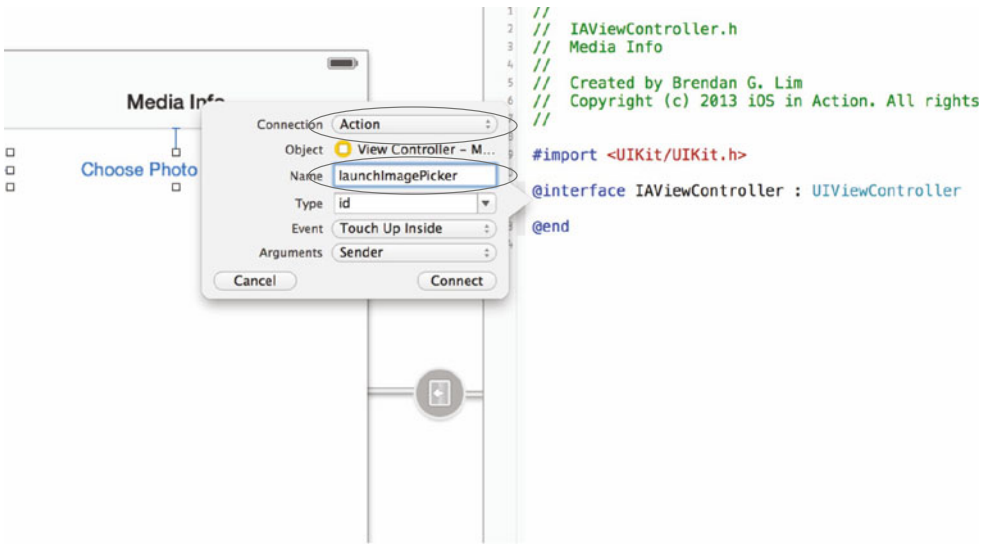


Figure 7.11 Create an action for the button named `launchImagePicker` that's triggered when the button is touched.

```
@interface IAViewController : UIViewController
➥ <UIImagePickerControllerDelegate, UINavigationControllerDelegate>
```

You didn't need to import anything to be able to use the `UIImagePickerController` in your class. This is because it's part of the base UIKit framework. After you've added this to IAViewController.h, you can move over to its implementation by opening IAViewController.m.

Find the `launchImagePicker:` action that you created earlier. Here you'll be setting up your image picker. You'll start by initializing a new `UIImagePickerController` within this method. Add the following code to your action:

```
UIImagePickerController *picker = [[UIImagePickerController alloc] init];
picker.delegate = self;
```

SETTING THE IMAGE PICKER SOURCE

There are different sources from which an asset can be retrieved when using the image picker. You can change this source by specifying the `sourceType` parameter for an image picker. One source type is for capturing content using the device's camera. You'll learn about that later. There are two source types that apply to choosing a photo or a video. The source type `UIImagePickerControllerSourceTypePhotoLibrary` tells the image picker that it should let the user choose from the entire Photo Library. The other option, `UIImagePickerControllerSourceTypeSavedPhotosAlbum`, takes the user directly to the saved photos album, skipping the choice for any other album that they have.

To give your users the choice to pick any photo from any album, you'll be using the entire Photo Library as the source. Add the following to the `launchImagePicker:` action:

```
picker.sourceType = UIImagePickerControllerSourceTypePhotoLibrary;
```

Next, you need to let the picker know whether it should show photos, videos, or both.

SPECIFYING SUPPORTED MEDIA TYPES

You can specify different media types depending on how you want to use the image picker in your application. You can change what type of media is supported by setting the `mediaTypes` property and passing in an array that contains the values `kUTTypeImage`, `kUTTypeMovie`, or both. For example, if you wanted to limit your image picker to just photos, you'd do the following:

```
picker.mediaTypes = @[kUTTypeImage];
```

There's also a class method that returns to you all of the supported media types for a specified source type, `availableMediaTypesForSource:`. This ensures that you always support all media types that can be supported. You want to do this for your application because you want to support all media types that are available. Add the following line to the `launchImagePicker:` action:

```
picker.mediaTypes = [UIImagePickerController
➥ availableMediaTypesForSourceType:picker.sourceType];
```

PRESENTING THE IMAGE PICKER

You've set up the image picker just how you'd want to use it in your application. It's time to use the picker by showing it. This is done by calling the `UIViewController` method `presentViewController:animated:completion:`. Add the following line to the bottom of your action:

```
[self presentViewController:picker animated:YES completion:nil];
```

The `launchImagePicker:` method is shown in the following listing in its entirety.

Listing 7.5 Launching the image picker controller from the `launchImagePicker:` method

```
- (IBAction)launchImagePicker:(id)sender
{
    UIImagePickerController *picker = [[UIImagePickerController alloc]
➥ init];
    picker.delegate = self;
    picker.sourceType = UIImagePickerControllerSourceTypePhotoLibrary;
    picker.mediaTypes = [UIImagePickerController
➥ availableMediaTypesForSourceType:picker.sourceType];
    [self presentViewController:picker animated:YES completion:nil];
}
```

You can now run the application and see it in action. It should be possible to pick any asset from any of the albums on your iOS device. Once the asset is picked, nothing currently happens. You'll remedy this by implementing the `UIImagePickerControllerDelegate` protocol.

7.2.2 Selecting assets from the image picker

Now that your image picker is visible and is allowing you to choose photos or videos, you need to do something with what's selected. Currently you're not doing anything because you haven't implemented a key method from the `UIImagePickerControllerDelegate` protocol. There are two methods specified in this protocol, with one being `imagePickerControllerDidCancel:`. This is called when no asset has been selected and the picker has been canceled.

The one that you'll be implementing in your app is `imagePickerController:didFinishPickingMediaWithInfo:`, which is called when a photo or video has been selected. Look at the method definition here:

```
- (void)imagePickerController:(UIImagePickerController *)picker
➥ didFinishPickingMediaWithInfo:(NSDictionary *)info;
```

This gives you access to the instance of the picker that was used as well as an `NSDictionary`. You'll be using this dictionary to retrieve the `NSURL` of the asset that was selected. The keys of the items in the dictionary, referred to as editing information keys, are shown in table 7.4.

Table 7.4 Editing information keys returned when selecting an asset from the image picker

Key	Description
`UIImagePickerControllerMediaType`	Type of media selected: photo, video, or unknown
`UIImagePickerControllerOriginalImage`	`UIImage` of the original image
`UIImagePickerControllerEditedImage`	`UIImage` of an edited version of the image
`UIImagePickerControllerCropRect`	Crop rectangle for a cropped edited version of the image
`UIImagePickerControllerMediaURL`	File system URL for a newly created movie
`UIImagePickerControllerReferenceURL`	Reference URL for retrieving an `ALAsset` representation
`UIImagePickerControllerMetaData`	Metadata for a newly created photo

Some of these editing information keys apply to photos or videos that have been edited within the picker or newly captured using the camera. You'll be using the `UIImagePickerControllerReferenceURL` key to get an NSURL that you can use to retrieve the asset from the Assets Library.

Implement the method shown in the following listing into IAViewController.m.

Listing 7.6 Handling selection of a photo or video from the image picker

```
- (void)imagePickerController:(UIImagePickerController *)picker
➥ didFinishPickingMediaWithInfo:(NSDictionary *)info
{
    NSURL *assetURL = info[UIImagePickerControllerReferenceURL];
    [picker dismissViewControllerAnimated:YES completion:nil];
    [self performSegueWithIdentifier:@"assetView" sender:assetURL];
}
```

❶ **Retrieve the NSURL for the selected asset.**

❷ **Dismiss the image picker.**

❸ **Prepare for the `assetView` segue.**

Here you first retrieve the NSURL for the selected asset ❶. Next you dismiss the image picker because it doesn't automatically do this for you when you implement this method ❷. Lastly you perform the `assetView` segue ❸ you created earlier and pass the `assetURL` as the sender so that you can use it when you override the `prepareForSegue:` method.

Now you'll add a property to IAAssetInfoViewController.h that will store this asset URL. Open it in your editor and add the following line:

```
@property (nonatomic, strong) NSURL *assetURL;
```

Next, go back into IAViewController.m and add the following `import` statement so that you can access the `IAAssetInfoViewController` class.

```
#import "IAAssetInfoViewController.h"
```

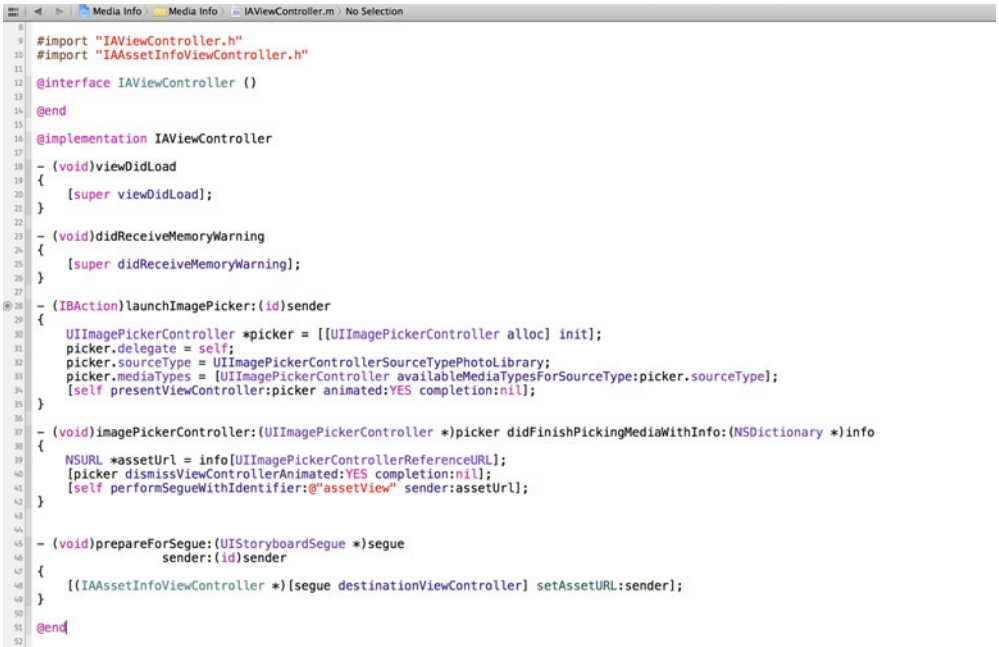


Figure 7.12 What your `IAViewController` implementation should look like at this point.

Override the `prepareForSegue:sender:` method to pass your asset's URL to the `IAAssetInfoViewController` triggered by your segue identifier.

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
                 sender:(id)sender
{
    [(IAAssetInfoViewController *)[segue destinationViewController]
➥ setAssetURL:sender];
}
```

At this point your IAViewController.m file should look like what's shown in figure 7.12.

With this in place you can run the application and choose a photo or video. Your segue will be triggered, which will cause your other view to appear. There will be nothing there yet because you haven't hooked anything up. Don't worry; you will very soon! Before you do this, you'll add another method of choosing photos or videos—capturing new ones using the camera.

7.3 Capturing photos and videos with the camera

It might sound complicated to use the camera to take photos and videos, but that's not the case. The APIs made available to you in UIKit have extracted much of the complexity. Specifically, the task of capturing media falls upon the image picker you've already been using. Capturing photos and videos is just another source type that you can specify. You'll soon learn about a few other options that you can use.

7.3.1 Checking for camera availability

If you're using the Simulator to run the application, you'll find out that you won't be able to use the camera because there isn't one. The only way you'd be able to truly test this is if you were to run the application on a real device. Also some iPods and iPads don't have a camera. Luckily, there's a way to check if one exists before proceeding further.

The camera source type you can specify for the `sourceType` property on a `UIImagePickerController` is `UIImagePickerControllerSourceTypeCamera`. To be able to check if a camera is available, you can use the `isSourceTypeAvailable:` class method as shown in the following code:

```
BOOL hasCamera = [UIImagePickerController
➥ isSourceTypeAvailable:UIImagePickerControllerSourceTypeCamera];
```

If it is available, you can choose to show the camera. If it isn't, you should show an alert saying that there is no camera. Let's do this in your app so that you can examine photos or videos that you capture with the camera if a camera is an available source.

Open Main.storyboard and add a button underneath your Choose Photo or Video button with the label Capture Photo or Video, as shown in figure 7.13.

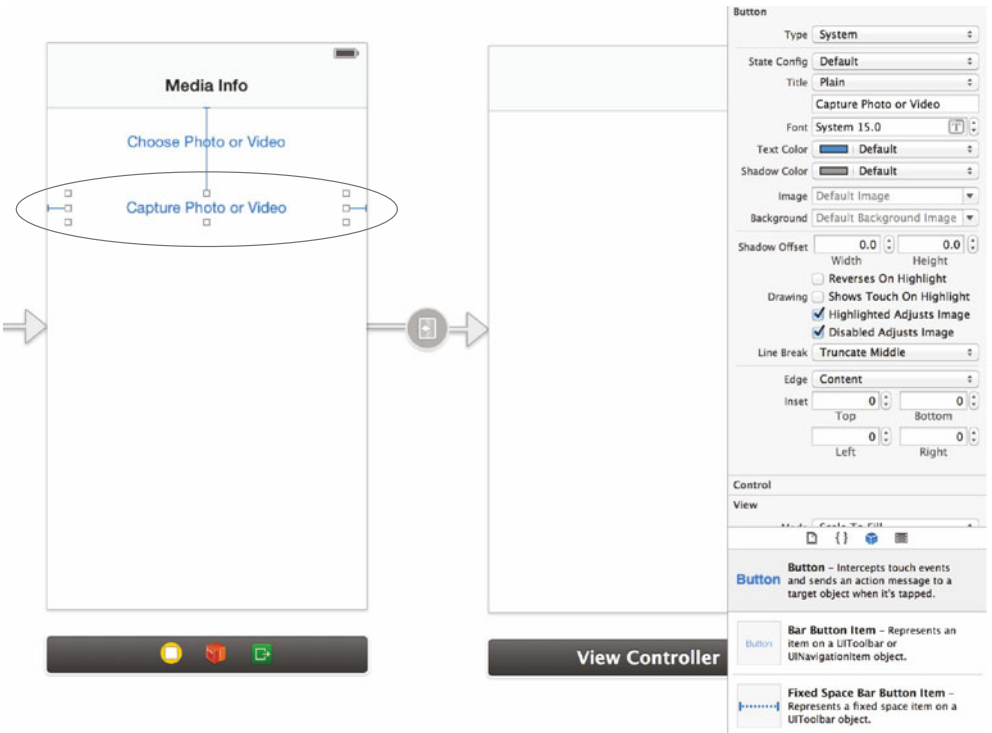


Figure 7.13 Add a button with the label Capture Photo or Video directly underneath your button to choose a photo or video.

Next, connect it to an action called `launchCamera` in `IAViewController` that's triggered when the button is touched. After the connection to the action has been made, jump back into IAViewController.m so you can add the code to detect if a camera is available.

You'll check to see if the `UIImagePickerControllerSourceTypeCamera` source is available. If it isn't, you'll show an alert informing the user. Replace the generated `launchCamera:` action with the following code.

Listing 7.7 Checking if a camera is available within the `launchCamera:` action

```
- (IBAction)launchCamera:(id)sender
{
    if (![UIImagePickerController
➥ isSourceTypeAvailable:UIImagePickerControllerSourceTypeCamera])
    {
        UIAlertView *alert = [[UIAlertView alloc] initWithTitle:
➥ @"Sorry!"
                                                        message:@"No camera
➥ available"
                                                       delegate:nil
                                              cancelButtonTitle:@"Okay"
                                              otherButtonTitles:nil, nil];
        [alert show];
    }
    else
    {
        // Camera is available
    }
}
```

❶ Check if the camera source type is available.

❷ Create an alert if not available.

❸ Show the alert.

You first check to see if a camera source type is available ❶. If it isn't, you create a `UIAlertView` ❷ and then show it ❸. If you run this in the Simulator, you'll see that this condition is executed because the `UIImagePickerControllerSourceTypeCamera` source isn't available. This is shown in figure 7.14.

If you wanted to, you could even go more fine-grained and check the availability of either a front- or rear-facing camera. This is particularly useful for applications that have functionality that relies on either of those cameras, such as videoconferencing apps. This is done by using the `isCameraDeviceAvailable:` class method, which can check against `UIImagePickerControllerCameraDevice` type. To check for the front-facing camera you could use the following:

```
BOOL hasFrontCamera = [UIImagePickerController
➥ isCameraDeviceAvailable:UIImagePickerControllerCameraDeviceFront];
```

To check for the rear-facing camera you could use this code:

```
BOOL hasRearCamera = [UIImagePickerController
➥ isCameraDeviceAvailable:UIImagePickerControllerCameraDeviceRear];
```

You also have the ability to check if the device has a flash for a particular camera by using the `isFlashAvailableForCameraDevice:` method.

```
BOOL hasFlashInRear = [UIImagePickerController
➥ isFlashAvailableForCameraDevice:
➥ UIImagePickerControllerCameraDeviceRear];
```

Now that you've implemented checking for a camera, you can see how to customize the media-capturing experience with the `UIImagePickerController` and use your newly captured photos and videos in your app.

7.3.2 Taking photos and videos with the camera

Preparing the `UIImagePickerController` for capturing from the camera is very similar to what you've already done. You'll need to first create a new instance of it, set its delegate, set the camera source type, and then specify a few parameters to customize its functionality. You'll start off by setting it up with the delegate and source in place:

```
UIImagePickerController *picker =
➥ [[UIImagePickerController alloc] init];
picker.delegate = self;
picker.sourceType =
➥ UIImagePickerControllerSourceTypeCamera;
```

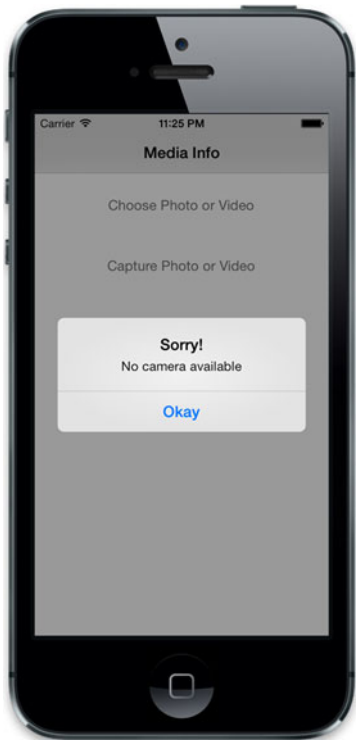


Figure 7.14 An alert is being displayed within the Simulator when trying to launch the camera because the camera source type is not available.

Also, just like when choosing a photo or video from the image picker, you can specify which media type you'd like to use. You can limit capturing to just photos or videos or both. For your app you'll allow for whatever's available:

```
picker.mediaTypes = [UIImagePickerController
➥ availableMediaTypesForSourceType:picker.sourceType];
```

> **Customizing the camera view**
>
> There are options that allow you to customize the view of the camera screen. By default there is a `BOOL` property named `showCameraControls`, which is `YES` by default. If you set this to `NO`, you *must* provide your own custom `UIView` and specify the `cameraOverlayView` property. By providing your own camera overlay view, you also have to implement the functionality to take a photo or capture a video. To capture a photo you'd use `takePicture`, and to capture a video you'd use `startVideoCapture` and `stopVideoCapture`.

With video capture you can limit the duration as well as the quality of the video. The duration of the video is determined by setting the `videoMaximumDuration` property.

By default this is an `NSTimeInterval` of 600 seconds (10 minutes). The following shows how you'd set it to 5 minutes:

```
picker.videoMaximumDuration = 300;
```

When setting the video quality, you'd set the `videoQuality` property using the `UIImagePickerControllerQualityType` enumeration. Table 7.5 shows the different options you're given.

Table 7.5 Different video quality options with the `UIImagePickerControllerQualityType` enumeration

Quality type	Description
`UIImagePickerControllerQualityTypeHigh`	Highest-quality video possible
`UIImagePickerControllerQualityType640x480`	VGA-quality video (640 x 480)
`UIImagePickerControllerQualityTypeMedium`	Medium-quality video
`UIImagePickerControllerQualityTypeLow`	Low-quality video
`UIImagePickerControllerQualityTypeIFrame1280x720`	iFrame video format (1280 x 720)
`UIImagePickerControllerQualityTypeIFrame960x540`	iFrame video format (960 x 540)

To get the highest-quality video possible, you'll set the `videoQuality` property on your picker to `UIImagePickerControllerQualityTypeHigh`:

```
picker.videoQuality = UIImagePickerControllerQualityTypeHigh;
```

You can put this together and show the camera using the image picker if you have a capable device. Change the `launchCamera:` action to the code in the following listing.

Listing 7.8 Updating `launchCamera:` to show the camera if available

```
- (IBAction)launchCamera:(id)sender
{
    if (![UIImagePickerController
➥ isSourceTypeAvailable:UIImagePickerControllerSourceTypeCamera])
    {
        UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Sorry!"
                                                        message:@"No camera
➥ available"
                                                       delegate:nil
                                              cancelButtonTitle:@"Okay"
                                              otherButtonTitles:nil, nil];
        [alert show];
    }
    else
    {
        UIImagePickerController *picker = [[UIImagePickerController alloc]
➥ init];
        picker.sourceType = UIImagePickerControllerSourceTypeCamera;
```

```
        picker.mediaTypes = [UIImagePickerController
➥ availableMediaTypesForSourceType:picker.sourceType];
        picker.videoQuality = UIImagePickerControllerQualityTypeHigh;
        picker.videoMaximumDuration = 300;
        picker.delegate = self;

        [self presentViewController:picker animated:YES completion:nil];
    }
}
```

This will allow you to take a photo or a video. You're now left with one other problem, though. When you've finished using the camera, the photo or video that you've captured isn't actually saved to your Assets Library.

7.3.3 Saving newly captured photos and videos to the Assets Library

The Assets Library allows you to save new photos or videos using a few convenient methods. You can save a new image by passing in an `NSData` representation or by using a `CGImageRef`. To be able to save a photo with `NSData` you could use the method shown here:

```
- (void)writeImageDataToSavedPhotosAlbum:(NSData *)imageData
                                metadata:(NSDictionary *)metadata
                         completionBlock:
➥ (ALAssetsLibraryWriteImageCompletionBlock)completionBlock;
```

When using the image picker you can retrieve a `CGImageRef` much more easily than the `NSData`. You'll be using this method when saving your photos:

```
- (void)writeImageToSavedPhotosAlbum:(CGImageRef)imageRef
                         orientation:(ALAssetOrientation)orientation
                     completionBlock:
➥ (ALAssetsLibraryWriteImageCompletionBlock)completionBlock;
```

You can retrieve the `UIImage` for the photo that was just taken by using the `UIImagePickerControllerOriginalImage` key from the dictionary given to you in the delegate method. The `UIImage` then gives you access to the `CGImageRef` by using the `CGImage` property. Using this you can save the photo to your Assets Library, as shown here:

```
UIImage *image = info[UIImagePickerControllerOriginalImage];
[[IAAssetsLibrary defaultInstance]
➥ writeImageToSavedPhotosAlbum:image.CGImage
                                                    orientation:
➥ (ALAssetOrientation)image.imageOrientation
                                          completionBlock:^(NSURL
➥  *assetURL, NSError *error)
{
    // Do something after completion
}];
```

The completion block returns the URL of the newly saved asset or an error if it couldn't be saved.

Saving a video is fairly similar to saving a photo. Instead of passing in a `CGImageRef`, you can pass in an NSURL for your newly created video. You can get this with the `UIImagePickerControllerMediaURL` key. Here's how you'd save a video to your Assets Library:

```
NSURL *mediaURL = info[UIImagePickerControllerMediaURL];
[[IAAssetsLibrary defaultInstance]
➥ writeVideoAtPathToSavedPhotosAlbum:mediaURL
➥ completionBlock:^(NSURL *assetURL, NSError *error)
{
    // Do something after completion
}];
```

As you can see, saving a photo and saving a video to the Assets Library are very similar. Let's make changes to your code to save newly captured photos and videos.

Within the delegate method that you've already added in `IAViewController`, you're going to first have to check the `sourceType` of the picker to see if something was captured or chosen from your Assets Library. This is because the same delegate method is called when an image or video is picked or when one is captured from the camera. From there you'll have to either save the photo or video and retrieve the asset URL or retrieve the asset URL from the existing asset and then trigger your segue.

Let's start by changing the delegate method `imagePicker:didFinishPickingMediaWithInfo:` to see if it was from the camera or another source. The next listing shows how this method has been updated.

Listing 7.9 Checking the source type from the image picker's delegate method

```
- (void)imagePickerController:(UIImagePickerController *)picker
➥ didFinishPickingMediaWithInfo:(NSDictionary *)info
{
    [picker dismissViewControllerAnimated:YES completion:nil];
    if (picker.sourceType == UIImagePickerControllerSourceTypeCamera)
    {
        [self saveMediaThenViewAsset:info];
    }
    else
    {
        NSURL *assetURL = info[UIImagePickerControllerReferenceURL];
        [self viewAssetFromURL:assetURL];
    }
}
```

Checking the source type to see if it was from the camera

New method to save the photo or video and then show the next view

New method to show the next view

You need to add two new methods, starting with `saveMediaThenViewAsset:`, which will save the photo or video depending on the media type. First, add the following `import` statement at the top of the file.

```
#import <MobileCoreServices/UTCoreTypes.h>
```

Next, add the method as shown in the following listing.

Listing 7.10 Saving a photo or video to the Assets Library depending on the media type

```
- (void)saveMediaThenViewAsset:(NSDictionary *)info
{
    NSString *mediaType = info[UIImagePickerControllerMediaType];

    if ([mediaType isEqualToString:(NSString *)kUTTypeImage])
    {
        UIImage *image = info[UIImagePickerControllerOriginalImage];
        [[IAAssetsLibrary defaultInstance]
➥ writeImageToSavedPhotosAlbum:image.CGImage

➥ orientation:(ALAssetOrientation)image.imageOrientation

➥ completionBlock:^(NSURL *assetURL, NSError *error)
        {
            [self viewAssetFromURL:assetURL];
        }];
    }
    else if ([mediaType isEqualToString:(NSString *)kUTTypeVideo])
    {
        NSURL *mediaURL = info[UIImagePickerControllerMediaURL];
        [[IAAssetsLibrary defaultInstance]
➥ writeVideoAtPathToSavedPhotosAlbum:mediaURL

➥ completionBlock:^(NSURL *assetURL, NSError *error)
        {
            [self viewAssetFromURL:assetURL];
        }];
    }
}
```

1 Retrieve media type from dictionary.

2 Check if it's a photo.

3 Proceed to view asset passing in the new asset URL.

4 Check if it's a video.

You're first grabbing the media type 1, and then checking to see if it's a photo 2. If it is, you proceed with saving the photo to the saved photos album. Once it's saved, you call the method `viewAssetFromURL:` (which you're about to add), passing in the new asset URL 3. If it's not a photo, you check to see if it's a video 4, and then save it to your saved photos album if it is.

You'll now add the `viewAssetFromURL:` method, which just contains your call to prepare for the `assetView` segue and passes in the NSURL of the asset you want to display. Add the method shown in the following listing.

Listing 7.11 `viewAssetFromURL:` to prepare for segue

```
- (void)viewAssetFromURL:(NSURL *)assetURL
{
    [self performSegueWithIdentifier:@"assetView" sender:assetURL];
}
```

Great job so far! You've added the ability to capture photos and videos and have them saved to your Assets Library. Next, you'll move on to the final view of your application, which will let you view the different properties of an asset that you've chosen.

7.4 Retrieving assets and accessing metadata

There's quite a bit of data to be found within each photo or video in your Assets Library. When working with photos you can access the EXIF (exchangeable image file format) information, which can reveal to you the type of camera used, lens, aperture, shutter speed, copyright, description, location, date captured, and so on. Sadly there's no EXIF available for videos, but you can still find out many details for the video, including its length, quality, date, and more.

7.4.1 Setting up your view to display the metadata

Open Main.storyboard and find your scene that represents the `IAAssetInfoViewController`. You're going to add an image to show a thumbnail of the asset you've chosen and a table view underneath that will display its metadata. Go to the Object Library and look for a `UIImageView`. Drag it to your view and set its dimensions in the size inspector to 320 x 190, as shown in figure 7.15.

After the size has been set, go back to the attributes inspector and set the image view's mode to Aspect Fit and the background color to black.

Next, open the assistant editor and create a new outlet from the `UIImageView` to `IAAssetInfoViewController`'s interface class. Name this property `imageView`.

You can now close the assistant editor and set up the table view that will display your asset's metadata. Go to the Object Library and find a table view, and drag it to take up the rest of the screen, as shown in figure 7.16.

Next you're going to set the table view's `delegate` and `datasource` outlets to `IAAssetInfoViewController`. With the table view selected, open the connections inspector and drag a connection for the `delegate` and `datasource` properties to the `IAAssetInfoViewController`. You'll also need to add an outlet for the table view itself. Open the assistant editor to bring up IAAssetInfoViewController.h and drag an outlet to create a connection for the table view named `tableView`.

When you've finished, close the assistant editor and open IAAssetInfoViewController.h in the main editor window. Add an import for the Assets Library:

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

Specify that you're conforming to the `UITableViewDelegate` and `UITableViewDataSource` protocols by changing the interface definition to the following:

```
@interface IAAssetInfoViewController : UIViewController
➥ <UITableViewDataSource, UITableViewDelegate>
```

Next, you'll add two new properties. One is for storing an `ALAsset` after you've retrieved it by using the asset URL that you passed in. The other is an array that will hold the metadata that you'll display within the table view.

```
@property (nonatomic, strong) ALAsset *asset;
@property (nonatomic, strong) NSMutableArray *metadata;
```

After these properties have been added, your class's interface should look similar to what's shown in figure 7.17.

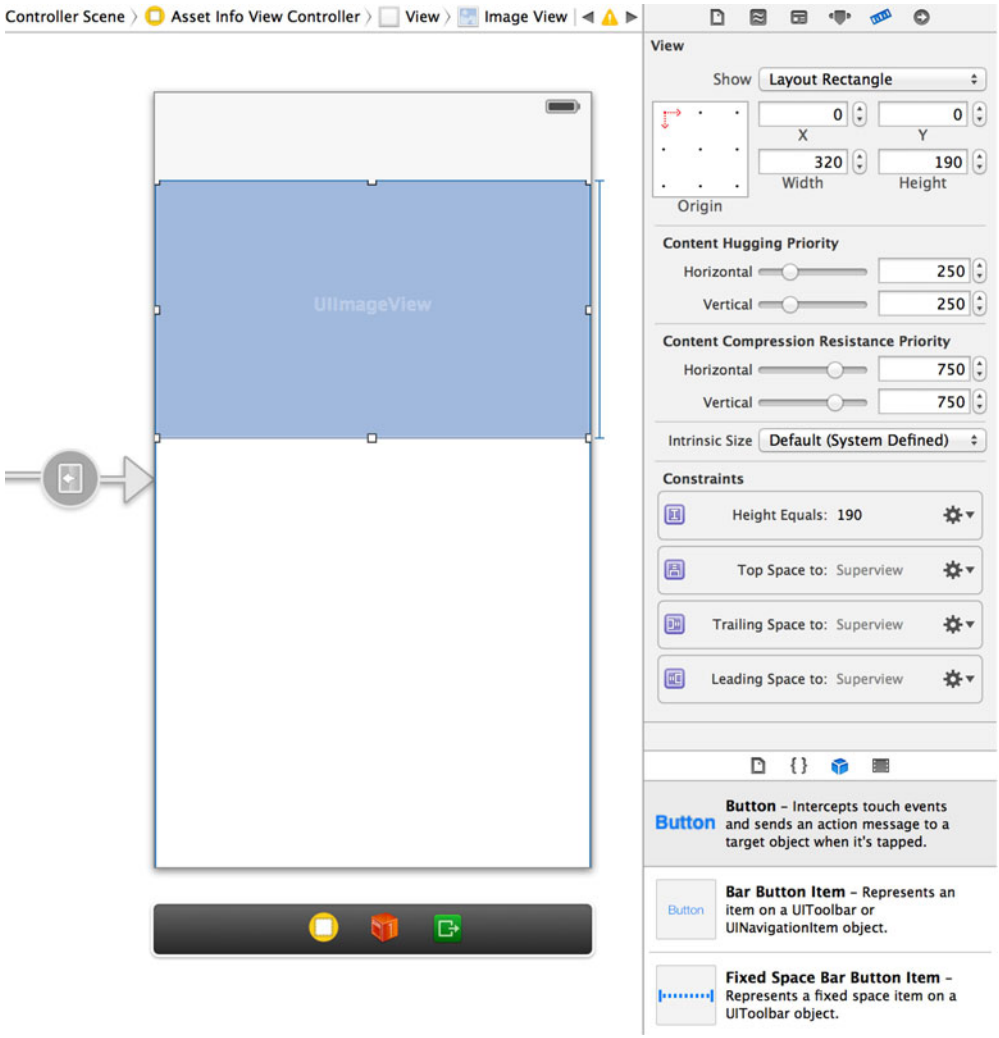


Figure 7.15 Add a `UIImageView` to `IAAssetInfoViewController`'s view and set its dimensions to 320 x 190.

Now open the implementation at IAAssetInfoViewController.m. You need to import your `IAAssetsLibrary` class:

```
#import "IAAssetsLibrary.h"
```

There will also be a few strings that you'll be using in a few of the methods in this class. To reduce the number of times you have to repeat yourself, let's define a few constants. Add this code directly above `@implementation IAAssetInfoViewController`:

```
#define kExif @"{Exif}"
#define kTitle @"title"
#define kValue @"value"
```

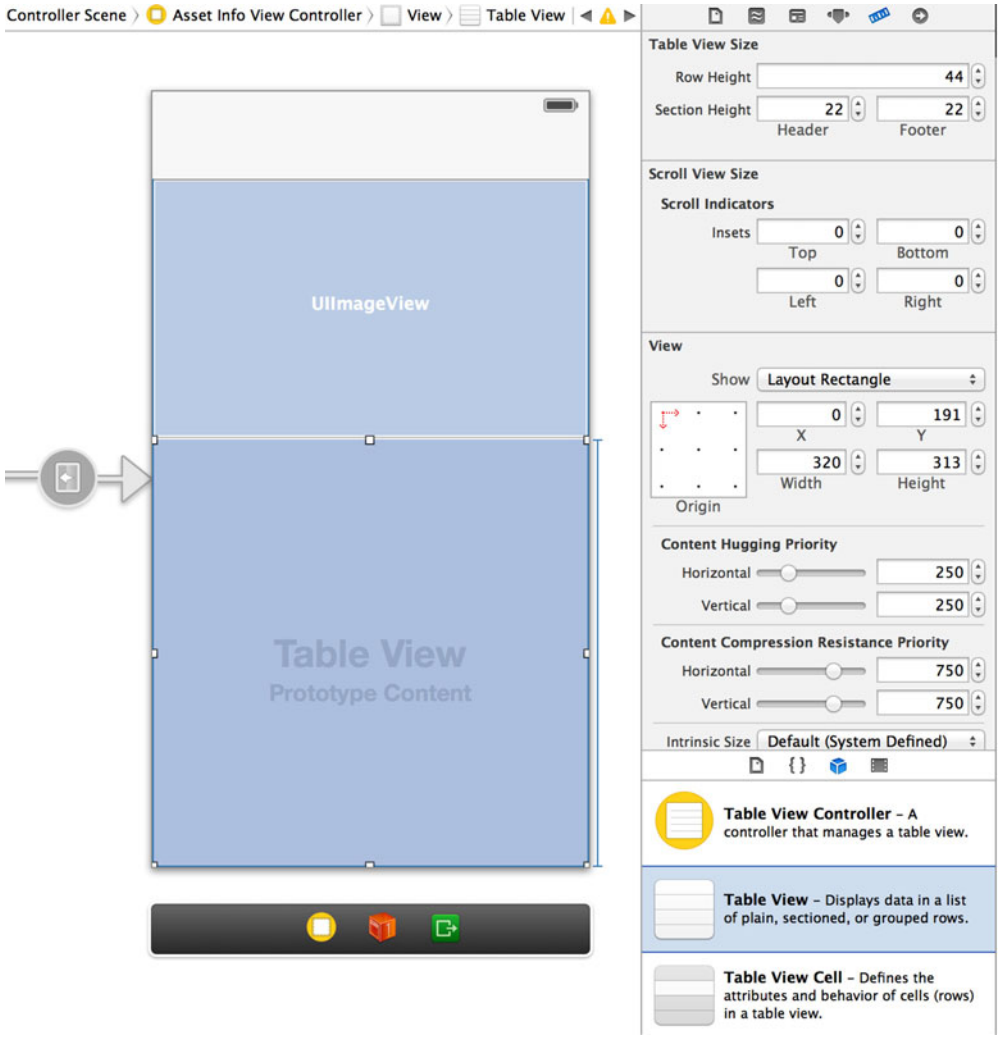


Figure 7.16 Add a table view to fit the rest of the screen underneath the image view that you've just added.

Once it's added, your code should look like what's shown in figure 7.18.

This will be all of the setup you'll need to do for the rest of the app. All you need to do now is plug in code to retrieve an asset from the Assets Library, filter through its metadata, and display it in your new view.

7.4.2 Retrieving an asset from the Assets Library

You can retrieve a specific `ALAsset` from the Assets Library if you have its unique URL. Luckily you happen to have one for each asset that you want to display. Using your

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

@interface IAAssetInfoViewController : UIViewController <UITableViewDataSource, UITableViewDelegate>

@property (nonatomic, strong) NSURL *assetURL;
@property (nonatomic, weak) IBOutlet UIImageView *imageView;
@property (nonatomic, weak) IBOutlet UITableView *tableView;
@property (nonatomic, strong) ALAsset *asset;
@property (nonatomic, strong) NSMutableArray *metadata;

@end
```

Figure 7.17 The interface for `IAAssetInfoViewController` after you've finished adding properties

```
#import "IAAssetInfoViewController.h"
#import "IAAssetsLibrary.h"

@interface IAAssetInfoViewController ()

@end

#define kExif @"{Exif}"
#define kTitle @"title"
#define kValue @"value"

@implementation IAAssetInfoViewController

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

Figure 7.18 `IAAssetInfoViewController`'s implementation should look like this before you move forward.

instance of the `ALAssetsLibrary`, you'll call the `assetForURL:resultBlock:failureBlock:` method like so:

```
[[IAAssetsLibrary defaultInstance] assetForURL:self.assetURL
                                   resultBlock:^(ALAsset *asset)
{
  // Successfully retrieved asset
}
                                  failureBlock:^(NSError *error)
{
  // Failed retrieving asset
}];
```

This method will retrieve an `ALAsset` from the NSURL that you provide it. If it succeeds in finding the asset, it returns it to you in the result block. If it fails, it gives you an error within the failure block.

Let's add the code to retrieve an asset when your view is loaded. In IAAssetInfoViewController.m add this one line to the bottom of the `viewDidLoad` method to call the `retrieveAsset` method that you're about to write:

```
[self retrieveAsset];
```

Next, let's add the `retrieveAsset` method to load the `ALAsset` from the Assets Library using the `assetURL` property, as shown in the following listing.

Listing 7.12 `retrieveAsset`—Retrieve asset from the Assets Library

```
- (void)retrieveAsset
{
    [[IAAssetsLibrary defaultInstance] assetForURL:self.assetURL
                                       resultBlock:^(ALAsset *asset)
    {
        [self setupViewFromAsset:asset];
    }
                                      failureBlock:^(NSError *error)
    {
        UIAlertView *alert = [[UIAlertView alloc] initWithTitle:
➥ @"Oops!"
                                                        message:@"You
➥ couldn't load that asset"
                                                       delegate:nil
                                              cancelButtonTitle:@"Okay"
                                              otherButtonTitles:nil, nil];
        [alert show];
    }];
}
```

❶ Set up view using the asset you've retrieved.

❷ Show an alert if asset retrieval failed.

In `retrieveAsset` you're calling a new method, `setupViewFromAsset:`, if asset retrieval succeeds ❶. If there's an error retrieving the asset, you show an alert ❷.

Now you'll add the `setupViewFromAsset:` method. Its purpose will be to set the title of the view with the asset's filename, populate your image view, and retrieve the metadata. Add the method found in the following listing.

Listing 7.13 `setupViewFromAsset:`—Populate your views from the retrieved asset

```
- (void)setupViewFromAsset:(ALAsset *)asset
{
    self.asset = asset;
    self.title = self.asset.defaultRepresentation.filename;
    UIImage *image = [UIImage
➥ imageWithCGImage:self.asset.defaultRepresentation.fullScreenImage];

    [self.imageView setImage:image];
    [self retrieveMetadata];
}
```

❶ Set the `asset` property.

❷ Set the view's title to the asset's filename.

❸ Populate your image view with the asset's image.

❹ Start metadata retrieval.

The first thing you do is set the `asset` property ❶ and then the title of your view using the asset's filename ❷. You then retrieve the full-screen image representation of the asset and use that to populate your image view ❸. At the end of the method you make a call to a new method that will retrieve and populate your metadata depending on whether it's a photo or a video ❹.

7.4.3 Accessing metadata for photos and videos

You've retrieved an asset from the Assets Library, and now you need to add a method to retrieve its metadata. Depending on whether it's a photo or a video, you'll display

different metadata. If it's a photo, you'll show the available EXIF information. If it's a video, you'll show a few properties on the `ALAsset` using the `valueForProperty:` method.

Because the retrieval for the data to display is different depending on the type of asset you're displaying, you're going to add separate methods for each type. The last method you added required a `retrieveMetadata` method to be defined. Add this to IAAssetInfoViewController.m, as shown in the next listing.

Listing 7.14 `retrieveMetadata`—Retrieve metadata depending on photo or video

```
- (void)retrieveMetadata
{
    self.metadata = [NSMutableArray new];

    if ([[self.asset valueForProperty:ALAssetPropertyType]
➥ isEqualToString:ALAssetTypePhoto])
        [self retrievePhotoMetadata];
    else
        [self retrieveVideoMetadata];
}
```

❶ Call `retrievePhotoMetadata` if a photo.

❷ Call `retrieveVideoMetadata` if a video.

In `retrieveMetadata` you check the type of the asset to see if you should call `retrievePhotoMetadata` if it is a photo ❶ and `retrieveVideoMetadata` if it's a video ❷.

When retrieving EXIF information on a photo, you first have to retrieve the `metadata` property on its `defaultRepresentation`, as shown here:

```
NSDictionary *meta = [[self.asset defaultRepresentation] metadata];
```

Within the dictionary that's returned, the EXIF information is available under the `{Exif}` key. This returns another `NSDictionary` that contains the EXIF. This is why you defined the `kExif` string when you were setting up this view controller. You're going to be enumerating through this dictionary to find displayable information. Each displayable bit of EXIF information will be stored in your class's `metadata` array as a new `NSDictionary`.

Add the `retrievePhotoMetadata` method shown here.

Listing 7.15 `retrievePhotoMetadata`—Retrieves and stores photo EXIF data

```
- (void)retrievePhotoMetadata
{
    NSDictionary *meta = [[self.asset defaultRepresentation] metadata];
    NSDictionary *exif = meta[kExif];
    for (id key in exif)
    {
        id value = exif[key];
        if (value && ![value isKindOfClass:[NSArray class]] && ![value
➥ isKindOfClass:[NSDictionary class]])
            [self.metadata addObject:@{kTitle : key,
                                       kValue : [NSString stringWithFormat:@"%@",
➥ value]}];
    }
```

❶ Retrieve the asset metadata dictionary.

❷ Retrieve EXIF from the metadata.

❸ Enumerate through EXIF dictionary.

❹ Filter for displayable EXIF data.

❺ Add displayable data to your metadata array.

```
    [self.tableView reloadData];
}
```

❻ Reload the table view.

You first load the metadata dictionary for the asset ❶ and then retrieve the EXIF information ❷. You then enumerate through the EXIF data ❸ and check for data that you want to display ❹. With this displayable data you create a new dictionary holding the key and the value and add that to your class's metadata array ❺. Last, you reload the table view ❻. When you run this using different photos, you may notice that some photos show more information than others. Some photos may have location information and some may not.

Finally, you need to retrieve the data you want to show for videos. Add the `retrieveVideoMetadata` method in the following listing.

Listing 7.16 `retrieveVideoMetadata`—Retrieves properties to display for a video

```
- (void) retrieveVideoMetadata
{
    [self.metadata addObject:@{kTitle: ALAssetPropertyDate,
                        kValue: [self.asset
➥ valueForProperty:ALAssetPropertyDate]}];
    [self.metadata addObject:@{kTitle: ALAssetPropertyDuration,
                        kValue: [self.asset
➥ valueForProperty:ALAssetPropertyDuration]}];
    [self.metadata addObject:@{kTitle: ALAssetPropertyLocation,
                        kValue: [self.asset
➥ valueForProperty:ALAssetPropertyLocation]}];

    [self.tableView reloadData];
}
```

❶ Date video was created

❷ Duration of the video

❸ Location information

❹ Reload table view

In this method you manually retrieve and add three properties for each video using the `valueForProperty:` method on the asset. You first add the video's creation date ❶, then the duration of the video ❷, and then the video's location ❸. After you've added these properties to your array, you reload the table view ❹.

The last thing for you to do is to add two methods for your table view to display the metadata that you've pulled back, as shown in the next listing.

Listing 7.17 Table view methods to specify row count and a cell with metadata for each row

```
- (NSInteger) tableView:(UITableView *)tableView
➥ numberOfRowsInSection:(NSInteger)section
{
    return [self.metadata count];
}

   - (UITableViewCell *) tableView:(UITableView *)tableView
➥ cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView
➥ dequeueReusableCellWithIdentifier:CellIdentifier];
```

```
    if (cell == nil)
        cell = [[UITableViewCell alloc]
➥ initWithStyle:UITableViewCellStyleSubtitle
reuseIdentifier:CellIdentifier];

    NSDictionary *data = self.metadata[indexPath.row];
    cell.textLabel.text = data[kTitle];
    cell.detailTextLabel.text = data[kValue];

    return cell;
}
```

You've finished setting up your Media Info app! Now run it and see what happens when you choose a photo. Take a look at figure 7.19 to see your app in action.

You can also try the app on your device to inspect the data for a photo or a video that you've captured using the camera. There are a handful of *paid* apps on the App Store that show you exactly what you've built yourself.

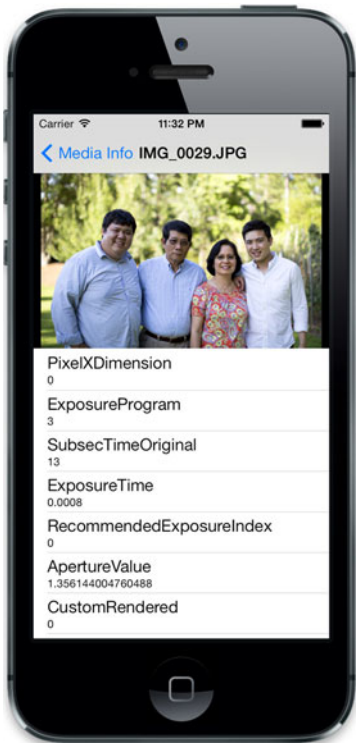


Figure 7.19 Showing the EXIF information for a photo that was chosen from the Assets Library

7.5 Summary

In this chapter you learned how to use the camera to capture and retrieve photos and videos using the image picker controller. You also learned about the assets library's key components and how to save and retrieve assets. Finally, you built an app that allows you to display detailed information for any asset you've chosen or captured with the camera.

- The Assets Library framework allows you to access and modify photos and videos managed by the Photos app.
- The `ALAssetsLibrary` instance acts as the gatekeeper to all of your assets.
- Groups, or albums, are known as an `ALAssetsGroup`. You can retrieve groups from iTunes, albums, events, detected faces, saved photos, and Photo Stream.
- The `ALAsset` class is used to represent photos and videos that are contained within an `ALAssetsGroup`.
- Applications must be given permission to access the photos stored on a device.
- The `UIImagePickerController` provides functionality that allows you to choose or capture photos and videos. You can even crop photos and trim videos. All of this is provided to you for free.

- You're able to control the level of quality when capturing video with the `UIImagePickerController`.
- You can access many properties on an `ALAsset` such as the EXIF information stored within photos.
- Using the Assets Library and the `UIImagePickerController`, you can choose photos and examine their metadata in your Media Info application.

8 Social integration with Twitter and Facebook

This chapter covers

- Accessing accounts with the Accounts framework
- Retrieving and displaying a Twitter or Facebook feed
- Integrating with Twitter's API and Facebook's OpenGraph
- Posting content to Twitter or Facebook

Facebook and Twitter are two social networking services that have invaded the daily lives of so many of us—so much so that Apple has decided to integrate the two directly into iOS. People can share photos, videos, links, and other types of content using either service directly from iOS. Previously, developers would have to supply their own integrations or resort to using third-party libraries to include this functionality within their applications. Now it's possible to use both of these services within your apps using libraries that are part of the iOS SDK.

Because there is now a centralized place within iOS to store these accounts, you can give apps permission to securely use their credentials to interact and share content with Twitter, Facebook, and Weibo. Within this chapter you'll learn how to

Figure 8.1 An overview of some of the screens that you'll be creating when you build TweetBook throughout this chapter

interact with Twitter and Facebook by using the Accounts and Social frameworks. Together we'll build an app called TweetBook that will allow you to use Twitter and Facebook within the same application.

TweetBook will use the Twitter and Facebook accounts stored within iOS to retrieve their latest updates and post new content. Let's take a quick look at what we'll be building together by looking at a few of its screens, as shown in figure 8.1.

By the end of this chapter you'll create an app that lets you use both Twitter and Facebook. You'll see how to request permission for accounts by using the Accounts framework. After having access to these accounts you'll use each one specifically to create and pull back new updates using the Social framework.

The app isn't going to be full featured like the Twitter or Facebook applications, but you'll still be creating quite a bit of functionality using a relatively small amount of code. It also gives you the groundwork to expand on TweetBook and create your own full-fledged social network application.

8.1 Accessing accounts with the Accounts framework

Facebook and Twitter accounts added within the Settings app can be accessed programmatically through the Accounts framework. A centralized accounts system within iOS gives people a single place to store their Twitter and Facebook credentials. This means that if all iOS apps used the Accounts framework, you would never have to enter your credentials more than once for either service.

The Accounts framework gives you secure access to authorized accounts, as long as you're granted permission. All of this is handled for you within the framework. In this section you'll learn how to find and retrieve accounts as well as access different properties of an account and use them within your own application.

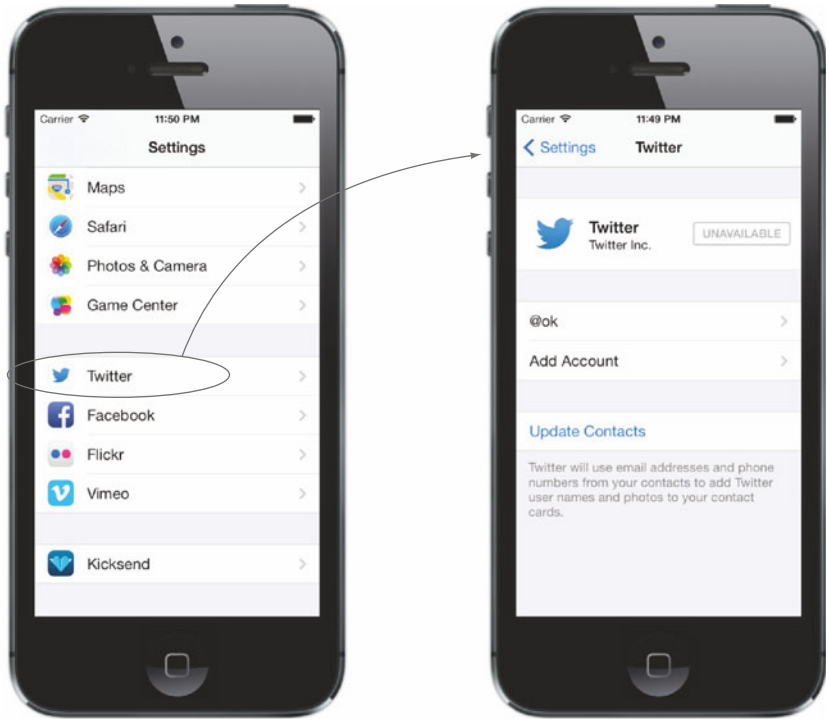


Figure 8.2 The Twitter accounts section within the Settings application allows you to store multiple account credentials.

8.1.1 Accessing Twitter accounts and account properties

Accounts for different social networks are added within the Settings app. Since you'll be accessing Twitter accounts, go to the Settings app and make sure that you've added your own credentials. You can see the Twitter accounts section in figure 8.2.

SETTING UP TWEETBOOK WITHIN XCODE

Before we go any further, let's set up the base TweetBook project. Open Xcode and create a new Single View Application named TweetBook. Once the project's been set up, you'll add the Accounts framework in the Link Binary With Libraries build phase within the project settings. Take a look at figure 8.3 to see how to add `Accounts.framework` to your project.

Next you'll create a new file that you'll be using to retrieve and display all accounts a user gives you permission to. Name it `IAAccountsViewController` and make sure that it's a subclass of `UIViewController`, as shown in figure 8.4.

Next, open the interface for `IAAccountsViewController` by clicking IAAccountsViewController.h and include the Accounts framework by adding the following `import` statement:

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

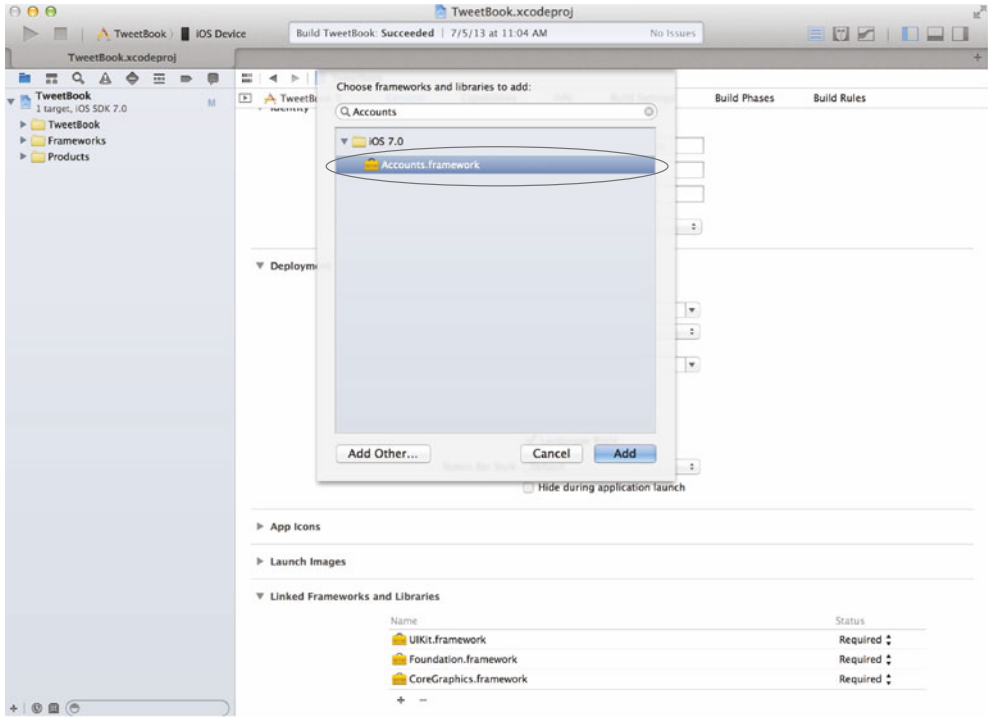


Figure 8.3 Add `Accounts.framework` in the Link Binary With Libraries build phase in TweetBook.

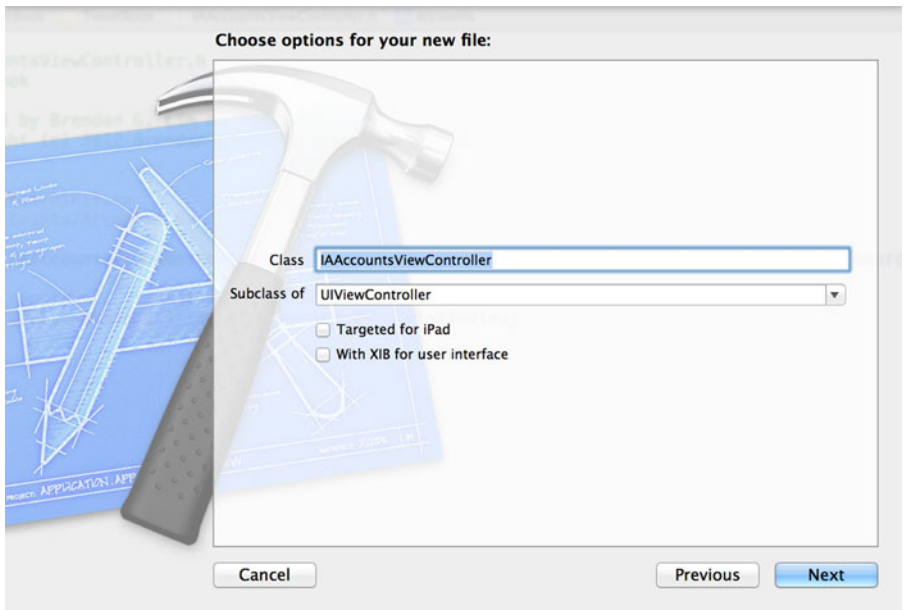


Figure 8.4 Adding `IAAccountsViewController` to our project to retrieve and display all user accounts

You're also going to need an array that stores all of the different accounts that you can retrieve. Add the following `NSMutableArray` property to the `IAAccountsViewController` interface.

```
@property (nonatomic, strong) NSMutableArray *accounts;
```

Now go to the implementation of `IAAccountsViewController`. You need to initialize this array when the view loads so that you can add accounts to it. Add it to the bottom of your autogenerated `viewDidLoad` method, as shown here:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.accounts = [[NSMutableArray alloc] init];
}
```

You can now proceed with retrieving accounts.

REQUESTING PERMISSION TO ACCESS TWITTER ACCOUNTS

Since you'll need permission to access Twitter accounts, you need a way to request this from the user. You can ask for permission to retrieve a list of accounts by using the `ACAccountStore` class. This allows you to request accounts by specifying a specific account type. You'll use the `ACAccountType` class to represent different account types. Currently there are only three different account types you can specify: Twitter, Facebook, and Weibo. You'll be using Twitter and Facebook in this chapter.

Once you make a request to fetch accounts and are granted permission, you can retrieve an array of accounts associated with an account type. Each object in the array that represents an account is an instance of an `ACAccount`.

Each `ACAccount` stores various properties associated with that account for a particular service. Following are some of the important properties on an `ACAccount`.

- `accountType`—`ACAccountType` for the type of service associated with the account
- `credential`—`ACAccountCredential`, which is used to make authenticated requests on behalf of the user
- `identifier`—`NSString` that represents a unique identifier for the account
- `username`—`NSString` of the user's username for the account

Let's retrieve a list of Twitter accounts a user has on their device. In the `IAAccountsViewController` implementation, you're going to declare a new method called `retrieveAccounts:options:` that will do this for you. The following listing shows how this is implemented.

Listing 8.1 Request permission and store a user's Twitter accounts into an array

```
-(void)retrieveAccounts:(NSString *)identifier options:
(NSDictionary *)options
{
    ACAccountStore *accountStore = [[ACAccountStore alloc] init];
```

Initializing an `ACAccountStore` ❶

```
    ACAccountType *accountType = [accountStore
➥ accountTypeWithAccountTypeIdentifier:identifier];

    [accountStore requestAccessToAccountsWithType:accountType
➥ options:options completion:^(BOOL granted, NSError *error)
    {
        if(granted)
        {
            [self.accounts addObjectsFromArray:[accountStore
➥ accountsWithAccountType:accountType]];
            dispatch_async(dispatch_get_main_queue(), ^{
                [self.tableView reloadData];
            });
        }
    }];
}
```

❷ **Creating an `ACAccountType` based on an identifier**

❸ **Requesting permission to access accounts**

❹ **Check if permission is granted.**

❺ **Retrieve and store all accounts of this type.**

❻ **Reload table view once finished.**

You're first allocating and initializing a new `ACAccountStore` object, `accountStore` ❶. Next, you're calling the method `accountTypeWithAccountTypeIdentifier:` on `accountStore` and passing in an `NSString` that represents an account type identifier to return an `ACAccountType` for Twitter ❷. You're then requesting permission from the user to access their Twitter accounts by calling the `requestAccessToAccounts-WithType:options:withCompletionHandler:` method ❸. Within the completion handler a `BOOL` is returned that lets you know if access to these accounts has been granted and returns an `NSError` if there's a problem. If access is granted by the user ❹, you can then retrieve an array of their accounts by calling `accountsWithType:` on `account-Store` ❺. Lastly you're making a call on the main queue to reload a table view ❻. You don't currently have a table view set up, but you will shortly.

You're going to retrieve the accounts when your view loads. Add a call to `retrieveAccounts:options:` with the Twitter accounts type identifier at the end of your `viewDidLoad` method. There's no need to pass in options yet; you'll be using this shortly when you retrieve Facebook accounts. Change the `viewDidLoad` method to what's shown here:

```
-(void)viewDidLoad
{
    [super viewDidLoad];
    self.accounts = [[NSMutableArray alloc] init];
    [self retrieveAccounts:ACAccountTypeIdentifierTwitter options:nil];
}
```

By adding this, the accounts will be retrieved after your view loads.

DISPLAYING ACCOUNTS IN A TABLE VIEW

You need to create a view for the account's view controller within your storyboard. In the project navigator find and choose `MainStoryboard.storyboard` to get started on your app's interface.

You should already have a scene that was generated when you created the project in Xcode. Choose the class that represents the view controller within the interface

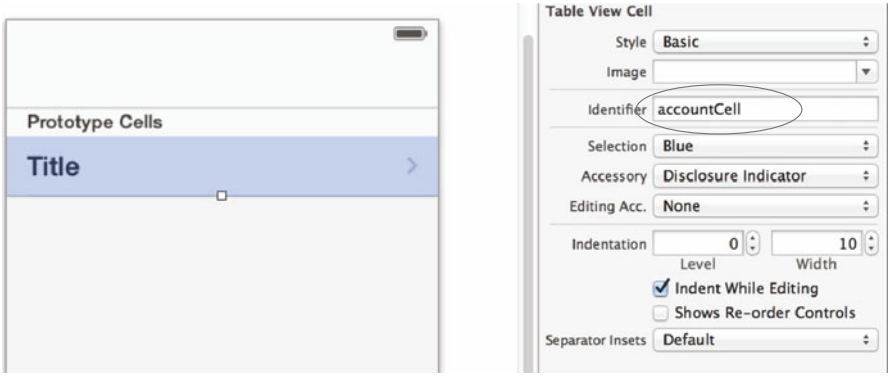


Figure 8.5 Add a table view cell as a new prototype cell to the table view, setting its style to Basic and its identifier to `accountCell`.

editor. Then open the inspector on the right side of the window, choose the Identity Inspector tab, and change the class property to `IAAccountsViewController`.

First, you'll embed this within a navigation controller. With the `IAAccountsViewController` scene selected, go to Xcode's application menu and choose Editor > Embed In > Navigation Controller. Now you'll add a table view to this view that will be used to list all of the accounts that you've retrieved using the Accounts framework. On the bottom right of your editor, go to the Objects Library and choose Table View. Drag it into the view, and make sure that it fits the whole view. Next, you'll add a table view cell to represent each account within the table view. In the Object Library choose Table View Cell and drag it into the table view. Once it's added, go to the attributes inspector in the utility area and set its identifier to `accountCell`, as shown in figure 8.5.

It's crucial that you set the cell's identifier correctly, so make sure that `accountCell` is spelled and specified correctly.

While you're still in the accounts scene, click the navigation bar and set the title to Accounts in the attributes inspector, as shown in figure 8.6.

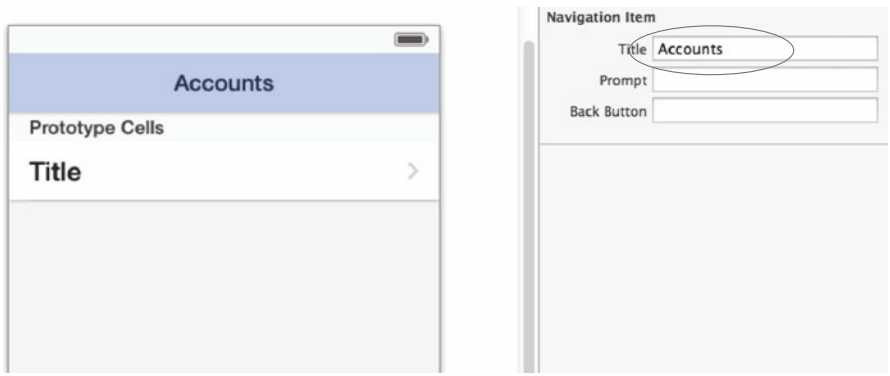


Figure 8.6 Set the title of the navigation item of the accounts view controller to Accounts.

Right now, your controller doesn't know your table view exists because you haven't created an outlet for it yet. Open the assistant editor to bring up IAAccountsViewController.h. Select the table view from your scene and then create a new outlet named `tableView`, which will add a new property to the header file, as shown here:

```
@property (weak, nonatomic) IBOutlet UITableView *tableView;
```

The table view needs to know where to look when it calls its delegate and data source methods to populate it with data and respond to touch events. Right-click the table view in the document outline to bring up its outlets. Drag the `delegate` and `dataSource` outlets to the account's view controller.

With the assistant editor still open and showing, you need to specify that `IAAccountsViewController` conforms to the `UITableViewDelegate` and `UITableViewDatasource` protocols. Change the top line to the following:

```
@interface IAAccountsViewController : UIViewController
➥ <UITableViewDelegate, UITableViewDataSource>
```

You can now close the assistant editor and switch back to the standard editor. Let's add some code so that you can fill this view with your accounts.

You now need to implement a few methods that are required since you added these two protocols. These methods will let your table view know how many rows to display and will return a `UITableViewCell` that will represent each account.

You will have one row for each account that you're displaying. The number of accounts you have should represent the number of rows that you'll have in your table view. Go into the implementation and add the following method to return the number of rows you will display in your table view:

```
- (NSInteger)tableView:(UITableView *)tableView
➥  numberOfRowsInSection:(NSInteger)section
{
    return [self.accounts count];
}
```

Each row will need to be populated with information from the `ACAccount` it's displaying. Create the method that returns a `UITableViewCell` for a particular row, as shown in the next listing.

Listing 8.2 Return a populated `UITableViewCell` with information from an `ACAccount`

```
- (UITableViewCell *)tableView:(UITableView *)tableView
     cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"accountCell";
    UITableViewCell *cell = [tableView
➥ dequeueReusableCellWithIdentifier:CellIdentifier];

    ACAccount *account = self.accounts[indexPath.row];
    cell.textLabel.text = account.accountDescription;

    return cell;
}
```

1. Create a static cell identifier.
2. Dequeue cell with identifier.
3. Retrieve account for row in index path.
4. Set title of cell to the account's description.

You first create a static `NSString` to act as a cell identifier ❶ that can be used to dequeue and reuse a particular cell ❷. To populate the cell with the account you're supposed to be displaying, you retrieve the account from the accounts array by using the row of the index path passed into this method ❸. For the title of the cell you're using the account's `accountDescription` property ❹.

By using the `accountDescription` property, you get a human-readable version of your account's username. For example, if you were to just use the username property, you would get back a string like `"ok"`. By using the `accountDescription` property, you get a more Twitter-esque version that people are accustomed to seeing: `"@ok"`.

Now the moment you've been waiting for. Run the application and check to see if everything's been set up properly. After being prompted for permission to access your accounts, you should see a list of all of your accounts shown in the Simulator. If you have no accounts set up, you'll see an empty table view. Make sure to double-check the Twitter section within the Settings app to ensure that you do have at least one account set up and that everything's been set up properly. Figure 8.7 shows our accounts view populated with one Twitter account.

Figure 8.7 Our application running with one Twitter account listed in the `IAAccountsViewController`

Next, you'll see how to retrieve and list your Facebook accounts.

8.1.2 Accessing Facebook accounts

Returning a list of all Facebook accounts is almost as easy as retrieving Twitter accounts. In your `viewDidLoad` method, you call `retrieveAccounts:options:` and pass in `ACAccountTypeIdentifierFacebook` with no options specified.

Finally, you'll need to pass in a few options in the form of an `NSDictionary`. Within this dictionary you'll specify a new Facebook app ID and permission you're requesting from each Facebook user.

CREATING A NEW FACEBOOK APP

Facebook requires that you create a new application for each app that interacts with Facebook. By doing this you also get back a Facebook app ID that's required to be passed in when asking for permission to access a user's Facebook account. Go to Facebook's developer portal at http://developers.facebook.com and create a new app. Application names are unique, so you may need to get creative with the name you specify, as shown in figure 8.8.

Once the application is created, you'll be taken to its settings page. Here you'll find your application ID and will also be required to fill out a few details about your

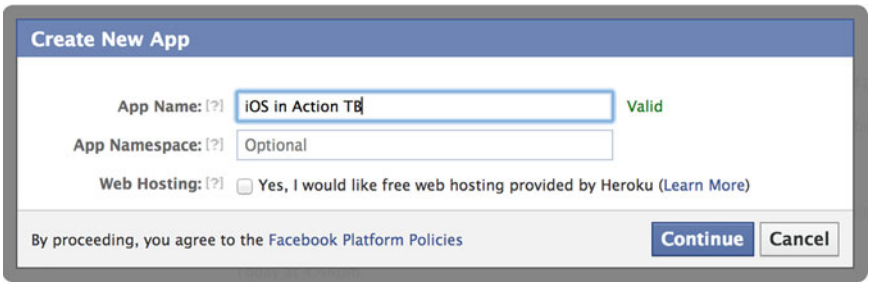


Figure 8.8 Creating a new Facebook app for TweetBook in Facebook's developer portal

iOS application. Take a look at figure 8.9 to see the app ID for the Facebook app you've just created and the iOS app settings you should fill out.

Your Facebook app ID is shown on the top of the Settings page. On the bottom of the page, you'll need to add the application's bundle identifier in the Native iOS App section. After that, you'll need to set the Facebook Login option to Enabled.

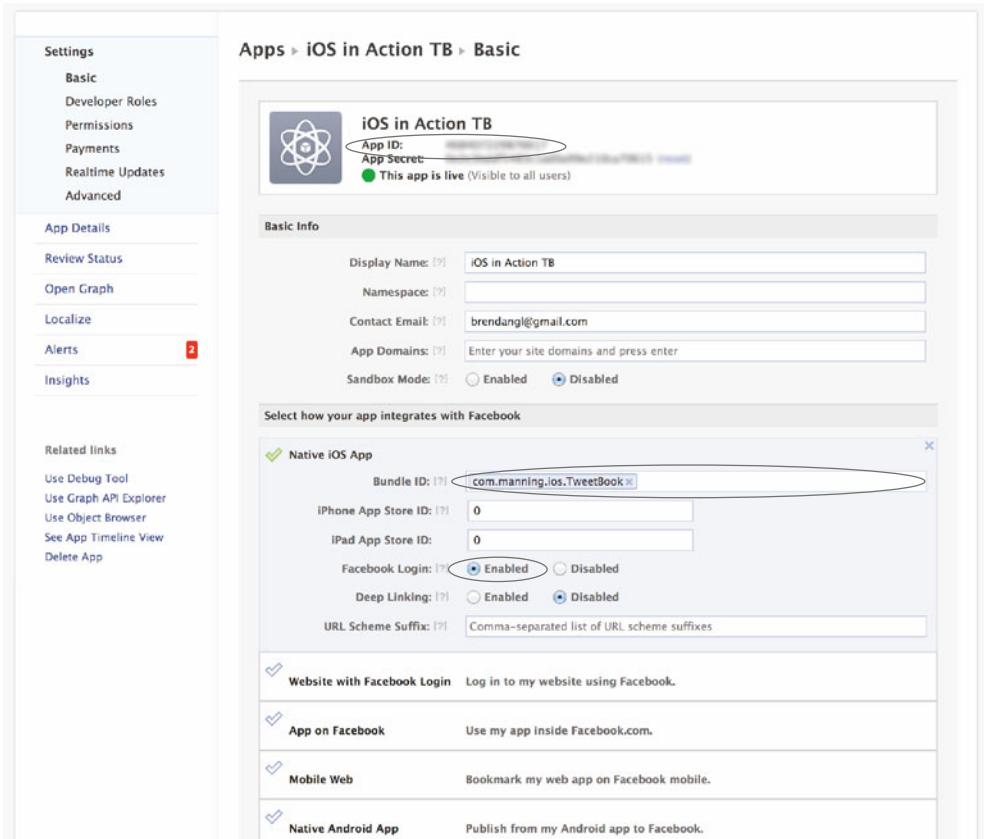


Figure 8.9 Application Settings page where you can see your app ID as well as specify native iOS app settings

REQUESTING PERMISSION FOR FACEBOOK ACCOUNTS

You can now go back to your code and prepare your options. You'll need to pass in three things: `ACFacebookAppIDKey`, `ACFacebookPermissionsKey`, and `ACFacebookAudienceKey`. The value for `ACFacebookAppIDKey` will be your newly created Facebook app's app ID. The permissions key, `ACFacebookPermissionsKey`, is for specifying which permissions you want the user to grant. These permissions can be used to retrieve their information and friends lists, read their stream, publish to their stream, and much more. You can find a list of all permissions you can specify by going to Facebook's permission documentation at the following address: https://developers.facebook.com/docs/reference/login/#permissions. It's *extremely* important to know that the first request that your application makes to access a user's Facebook account can *only be read permission*. You can't initially ask to publish to their news stream. You'll be asking for two read permissions, email and user_about_me, initially.

Go back to your `viewDidLoad` method in IAAccountsViewController.m. You'll declare a new `NSDictionary` named `fbOptions` that has two keys, `ACFacebookAppIDKey` and `ACFacebookPermissionsKey`, specified with their corresponding values:

```
NSDictionary *fbOptions = @{ ACFacebookAppIdKey: @"YOUR-FB-APP-ID",
➥  ACFacebookPermissionsKey: @[@"email",@"user_about_me"] };
```

Last you'll add a call to retrieve accounts using your `retrieveAccounts:options:` method but pass in `ACAccountTypeIdentifierFacebook` as the identifier and `fbOptions` as the options parameter. This method will perform the same action as the method used earlier but will retrieve Facebook accounts relative to the account identifier you're specifying. Your `viewDidLoad` should look like the following:

```
-(void)viewDidLoad
{
    [super viewDidLoad];
    self.accounts = [[NSMutableArray alloc] init];

    [self retrieveAccounts:ACAccountTypeIdentifierTwitter options:nil];

    NSDictionary *fbOptions = @{ ACFacebookAppIdKey: @"YOUR-FB-APP-ID",
    ➥  ACFacebookPermissionsKey: @[@"email",@"user_about_me"] };
}
```

Now when you run your application, you'll be prompted to give access to your Facebook accounts, as shown in figure 8.10.

If you hit OK, you'll see your Facebook account(s) listed underneath your Twitter accounts. Notice that our Facebook account is simply labeled Facebook. This needs to be way more specific, especially if someone has more than one Facebook account. Within `tableView:cellForRowAtIndexPath:` you'll change what's used for your cell's text label depending on the account type. The following is how you're currently setting the title for an account:

```
cell.textLabel.text = account.accountDescription;
```

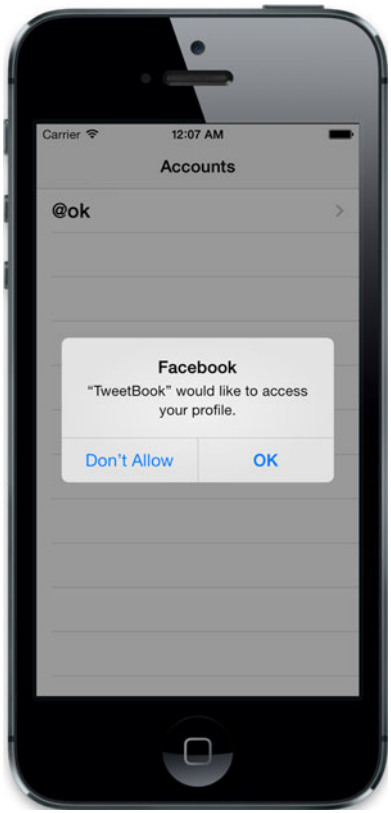


Figure 8.10 You will be prompted to give access to your Facebook accounts after you run your application.

For Twitter accounts the `accountDescription` property serves you exactly what you need. For Facebook you should use the `username` property because this will return the user's email address. To determine the account type and what you should use, you can examine the account type's `identifier` property. Replace the previous line with the following code:

```
if ([account.accountType.identifier
➥ isEqualToString:ACAccountTypeIdentifierTwitter])
    cell.textLabel.text = account.accountDescription;
else
    cell.textLabel.text = account.username;
```

The table view should now show the username of your Facebook account. So far you've retrieved and listed different accounts. It's now time to start posting to Twitter or Facebook using these accounts.

8.2 Using the Social framework to post content

With access to these accounts, you can actually use them to post content to Twitter or Facebook, depending on the account type. The Social framework allows you to do this by giving you a simple interface and API to post status updates, photos, and videos.

Here you'll learn how you can create content using an account with the Social framework within TweetBook.

8.2.1 Posting to Twitter using the Tweet Composer view

Before you start tweeting what you ate for lunch or posting random cat photos to Twitter, you need to create a new view controller in your TweetBook application. This new view controller will be used to view a user's stream and will also serve as the starting point from which you'll be able to post new content.

CREATING THE STREAM VIEW CONTROLLER

First, you'll need to add the Social framework to your project in the same way that you added the Accounts framework earlier. Hop back into Xcode and go to the General tab within your project settings. Then add `Social.framework` into the Link Binary with Libraries section, as shown in figure 8.11.

Next, go to the project navigator in Xcode and add a new class, `IAStreamViewController`, and make it a subclass of `UIViewController`. In `IAStreamViewController`'s interface, you want to go ahead and include the Social and Accounts frameworks by adding the following:

```
#import <Social/Social.h>
#import <Accounts/Accounts.h>
```

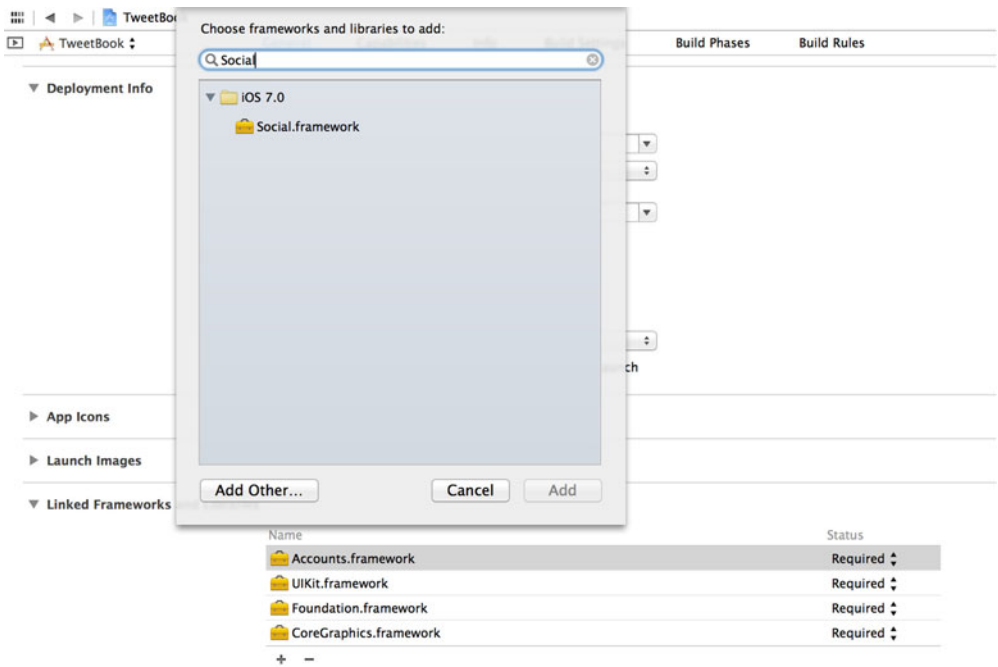


Figure 8.11 Add `Social.framework` to the Link Binary with Libraries section under Build Phases in your project settings.

Then set `IAStreamViewController` as a delegate and data source for the `UITableView`. You'll need to specify an outlet for your `UITableView` and an `IBAction` named `postToStream:` that you can use to post to a user's stream. Also, you need to add a property for an `ACAccount` that will represent the account the user has selected from the account view controller. All of this is shown in the following listing.

Listing 8.3 Interface for `IAStreamViewController`

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

@interface IAStreamViewController : UIViewController
➥ <UITableViewDataSource, UITableViewDelegate>

@property (weak, nonatomic) IBOutlet UITableView *tableView;
@property (strong, nonatomic) ACAccount *account;

- (IBAction)postToStream:(id)sender;

@end
```

Jump into your storyboard by clicking `MainStoryboard.storyboard` in your project navigator. You'll be adding a new `UIViewController` to your storyboard to represent your newly created `IAStreamViewController`. Go to the Object Library on the bottom right, choose a `UIViewController`, and drag it into your storyboard directly to the right of your existing `IAAccountsViewController`. Then go to the inspector and set the class name for this view to match your `IAStreamViewController`, as shown in figure 8.12.

Go back to the Object Library to find a `UITableView` and drag it into your new view. Set its `delegate` and `dataSource` outlets to the stream view controller. Next, right-click the table view and drag a new outlet from the New Referencing Outlet to the stream view controller and select `tableView`.

You want the stream view to be loaded when you click a `UITableViewCell` that represents one of your accounts. Control-click from the table view cell you've set up in the Accounts view and drag it to the stream view to create a new push segue, as shown in figure 8.13.

After you create the segue, a navigation bar will appear in the scene that contains your stream view. Add a bar button item to the right side of the navigation bar that will serve as the button users will use to post to their streams. Choose `UIBarButtonItem` from the Object Library and drag it to the right of the navigation bar. Set its identifier to Compose to get the button to show the standard iOS compose icon (+). Finally, right-click the button, drag the selector outlet to your stream view controller, and choose the action you defined in your header file, `postToStream:`.

Because you need to carry over account information to the stream view when a user clicks an account in the accounts view, you'll need to first include the header file for your stream view at the top of IAAccountsViewController.m.

```
#import "IAStreamViewController.h"
```

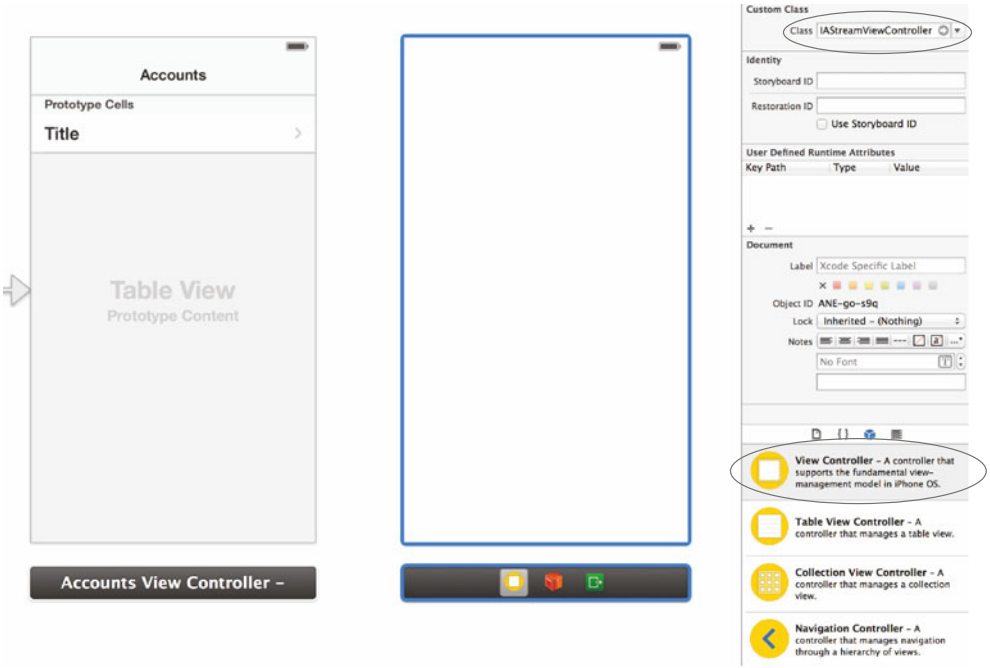


Figure 8.12 Add a new view controller to the right of the accounts view controller; then set its class to `IAStreamViewController`.

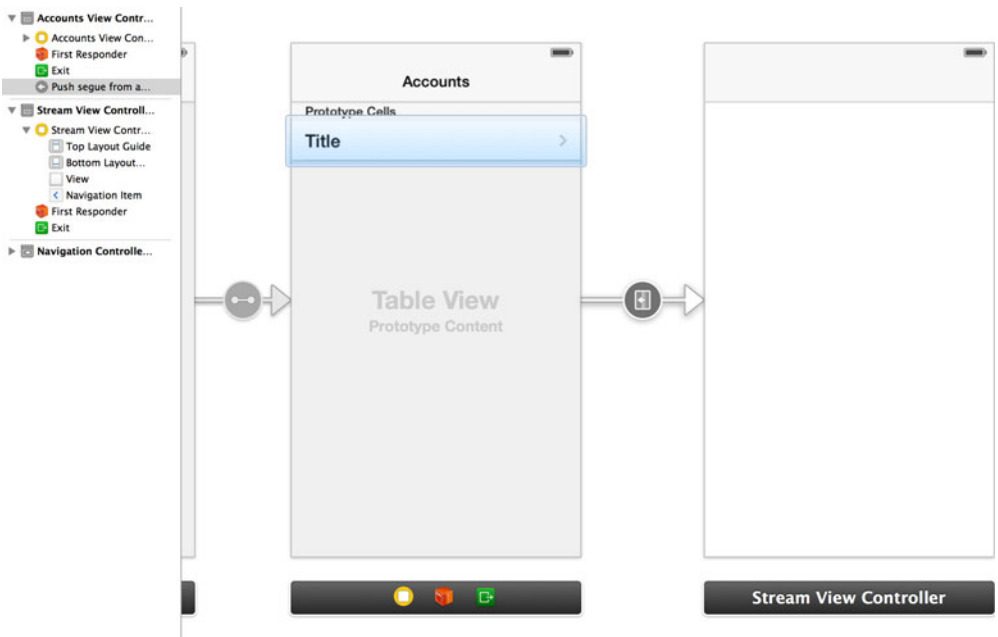


Figure 8.13 Create a new segue from the Accounts view's table view cell to the new stream view.

You'll next need to intercept the segue that you created and pass the selected account information to the stream view. You'll do this within `prepareForSegue:sender:` in the accounts view. Jump back into the accounts view and add the following method to your implementation:

```
- (void) prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    NSIndexPath *selectedIndex = self.tableView.indexPathForSelectedRow;
    ACAccount *account = [self.accounts
➥ objectAtIndex:selectedIndex.row];

    IAStreamViewController *view = [segue destinationViewController];
    view.title = [self.tableView
➥ cellForRowAtIndexPath:selectedIndex].textLabel.text;
    view.account = account;
}
```

❶ Retrieve the selected index path.

❷ Get the account for the selected row.

❸ Retrieve the destination view controller.

❹ Set the view's text property.

❺ Set the view's account property.

Within this method you first grab the selected index path from your table view ❶. You use the index path's row to retrieve the selected account from the accounts array ❷. Next, you get a reference to the stream view controller ❸, and then you set the title of the view to the text used in the selected table view cell ❹, which is the user's account name. Last, you set the account property on the stream view to the account that was selected ❺.

Before you run your app, you'll need to add a few methods so that you comply with the `UITableViewDatasource` delegate protocol. Without doing this your application will crash when the `IAStreamViewController` is loaded. You'll be replacing these methods later in the chapter when you finish this view.

```
- (NSInteger)tableView:(UITableView *)tableView
➥ numberOfRowsInSection:(NSInteger)section
{
    return 0;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
➥ cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    return nil;
}
```

Now when you run the application and tap on an account, you'll be taken directly to the stream view, as shown in figure 8.14.

You want to be able to click the bar button item on the top right to be able to post to your stream. If you click it right now, nothing will happen because you haven't defined anything in the `postToStream:` method yet. Let's do that right now.

Posting to Twitter

The Social framework gives you a really handy view called the `SLComposeViewController`. If you've ever shared a web page that you were browsing to Twitter while

Figure 8.14 After clicking an account, you're taken to the stream view.

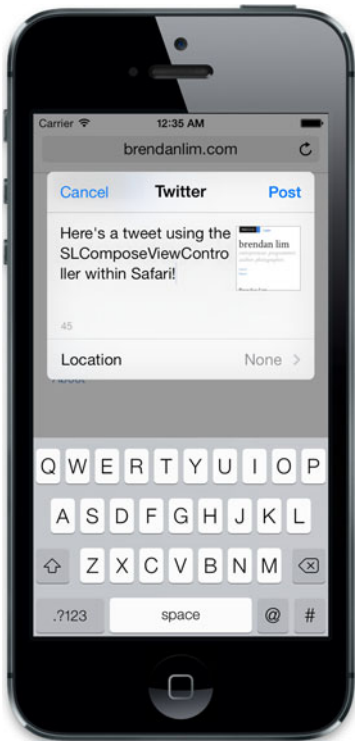


Figure 2.15 Using the `SLCompose-ViewController` within Safari to share a web page on Twitter

using Safari, you've most likely used this view. For example, take a look at figure 8.15 to see how content is posted to Twitter within Safari.

The `SLComposeViewController` gives you many things out of the box that would normally take a while to implement on your own. This saves you quite a bit of time because you don't need to worry about interfacing directly with the Twitter API when creating a custom-styled view, validations, and much more.

Add this to your stream view controller by presenting a compose sheet when the button on the top right is tapped. Within `postToStream:` you'll first check what type of account you're working with because you'll need to request additional permissions for Facebook. Because you're working with Twitter right now, just do a quick check against the account type identifier and call a new method called `postToTwitter`, as follows:

```
- (IBAction)postToStream:(id)sender
{
    if ([self.account.accountType.identifier
➥ isEqualToString:ACAccountTypeIdentifierTwitter])
        [self postToTwitter];
}
```

If you try to run the application now, you'll get a compile error because you haven't added a `postToTwitter` method yet. You'll define this method, and within it you'll add the code necessary to launch an `SLComposeViewController`. Look at the following listing, and we'll go over it together.

Listing 8.4 Launching an `SLComposeViewController` for a Twitter account

```
- (void)postToTwitter
{
    if([SLComposeViewController
➥ isAvailableForServiceType:SLServiceTypeTwitter])
    {
        SLComposeViewController *view = [SLComposeViewController
➥ composeViewControllerForServiceType:SLServiceTypeTwitter];
        [self presentViewController:view animated:YES completion:^{}];
    }
}
```

❶ Check if Twitter is available for a compose view.

❷ Initialize a compose view for Twitter.

❸ Show the compose view.

In only a few lines you're able to perform this functionality. You first check to see if you're allowed to use the compose view for the Twitter service (`SLServiceTypeTwitter`) ❶. If you're allowed, you initialize a new compose view for the service type `SLServiceTypeTwitter` ❷. To show the view, you use your view controller's `presentViewController:animated:completion:` method ❸. This method gives you the option to specify whether the view is animated when it appears and also a callback for when the compose view closes.

You're also able to prefill the compose view with text, images, and a URL. For example, right before you call `presentViewController:animated:completion:`, you could prefill the compose view with the following information:

```
[view setInitialText:@"Boom! Tweeting this from
    an app I created while
➥ reading iOS in Action! "];
[view addURL:[NSURL URLWithString:@"http://
    manning.com"]];
```

Now if you launch the compose view by clicking the Compose button in the stream view, you'll see it prefilled with this information, as shown in figure 8.16.

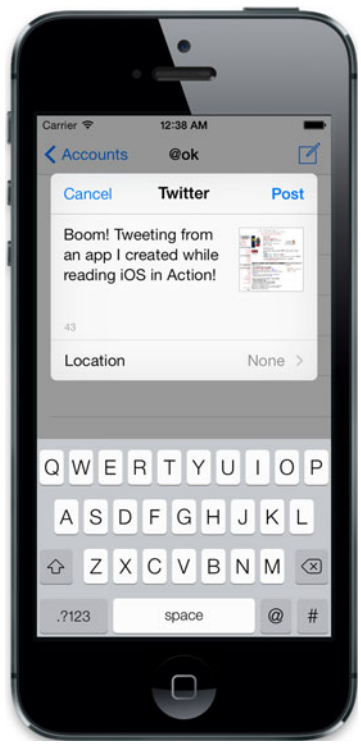


Figure 8.16 Prefilling the compose view with initial text and a URL

Wasn't that easy? It's a pretty simple experience for Facebook as well.

8.2.2 Posting to Facebook

When posting to Facebook you only need to change the service type from `SLServiceTypeTwitter` to `SLServiceTypeFacebook`. Change the `postToStream: IBAction` to call `postToFacebook` if the account isn't from Twitter. Your completed `postToStream:` method should look like the following:

```
- (IBAction)postToStream:(id)sender
{
    if ([self.account.accountType.identifier
➥ isEqualToString:ACAccountTypeIdentifierTwitter])
        [self postToTwitter];
    else
        [self postToFacebook];
}
```

Now create a new method called `postToFacebook` that will launch a new `SLComposeViewController`, as shown in the following listing.

Listing 8.5 Launching an `SLComposeViewController` for a Facebook account

```
- (void)postToFacebook
{
    if([SLComposeViewController
➥ isAvailableForServiceType:SLServiceTypeFacebook])
    {
        SLComposeViewController *view = [SLComposeViewController
➥ composeViewControllerForServiceType:SLServiceTypeFacebook];
        [view setInitialText:@"Boom! Posting from an app I created while
➥ reading iOS in Action!"];
        [view addURL:[NSURL URLWithString:@"http://manning.com/lim2"]];
        [self presentViewController:view animated:YES completion:^{}];
    }
}
```

Notice that all that's changed is that the service type has changed from `SLServiceTypeTwitter` to `SLServiceTypeFacebook`. You could further simplify this method to accept Twitter or Facebook accounts, but this leaves you with room to customize interaction based on the service type.

If everything's hooked up properly, you should see the compose view appear for a Facebook account when you click the Compose button, as shown in figure 8.17.

You now know how to easily post to your Twitter or Facebook stream. The last thing you'll need to do within TweetBook is retrieve a user's stream depending on which service type they've chosen.

8.3 Making API requests with the Social framework

The Social framework gives you the ability to manually interact with the API of all the services that it supports as long as you have access to an account. All of this is accomplished by using the `SLRequest` class. This allows you to post to someone's stream, retrieve their stream or timeline, friend, unfriend, follow, unfollow, and so on. It all depends on the service and what you're allowed to do based on the limits of their API.

The `SLRequest` class encapsulates various properties of a standard HTTP request. We covered the different types of HTTP requests and REST in chapter 6.

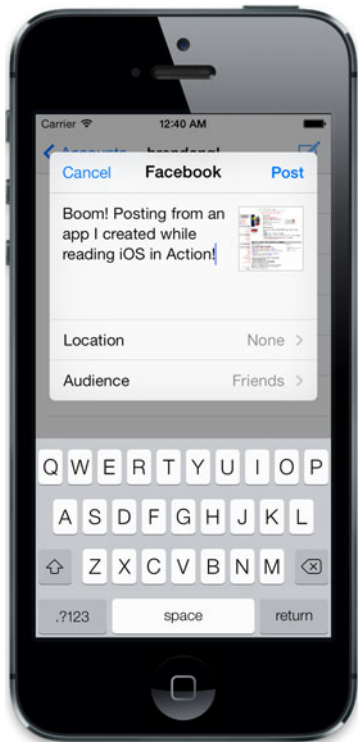


Figure 8.17 The compose view shown for Facebook accounts

8.3.1 Retrieving a Twitter stream using an SLRequest

All of your tweets can be retrieved through your public timeline. You can retrieve this via Twitter's API using an `SLRequest`. If you look at Twitter's developer documentation (http://dev.twitter.com/docs/api), you'll see exactly what you need to send in your request to retrieve a timeline:

- URL—http://api.twitter.com/1.1/statuses/user_timeline.json
- HTTP method—`GET`
- Required parameter—`user_id` or `screen_name`
- Optional parameters—`since_id`, `count`, `max_id`, `page`, `trim_user`, `include_rts`, `include_entities`, `exclude_replies`, `contributor_details`

Let's use `SLRequest`'s `requestWithServiceType:requestMethod:URL:params` method to make this request. Because you might not know the `user_id`, you can specify the `screen_name` parameter, which will be the `username` property of the selected account.

First, you need to add a property called `updates` for an `NSMutableArray` to your interface that will be used to store every update. Add this right below the property you created for `account`:

```
@property (strong, nonatomic) NSMutableArray *updates;
```

Within the `viewDidLoad` method in your stream view controller, you'll need to initialize the `updates` array and check what account type you're displaying. If the account is a Twitter account, you can call the `retrieveTwitterStream` method. Your `viewDidLoad` should look like what's shown here:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.updates = [[NSMutableArray alloc] init];

    if ([self.account.accountType.identifier
➥   isEqualToString:ACAccountTypeIdentifierTwitter])
        [self retrieveTwitterStream];
}
```

Next, you can define the `retrieveTwitterStream` method. This method will make a request using `SLRequest` and store all items in the `updates` array. Take a look at the following listing to see how to make your request to retrieve a user's Twitter stream.

Listing 8.6 Retrieving a user's Twitter stream with `SLRequest`

```
- (void)retrieveTwitterStream
{
    NSURL *url = [NSURL URLWithString:
➥ @"https://api.twitter.com/1.1/statuses/user_timeline.json"];
    NSDictionary *params = @{@"screen_name" : self.account.username};

    SLRequest *request = [SLRequest
➥ requestForServiceType:SLServiceTypeTwitter
➥ requestMethod:SLRequestMethodGET URL:url parameters:params];

    [request setAccount:self.account];
    [request performRequestWithHandler:^(NSData *responseData,
➥ NSHTTPURLResponse *response, NSError *error)
     {
         if (response.statusCode == 200)
         {
             NSError *parsingError = nil;
             self.updates = [NSJSONSerialization
➥ JSONObjectWithData:responseData options:0 error:&parsingError];

             dispatch_sync(dispatch_get_main_queue(),^{
                 [self.tableView reloadData];
             });
         }
     }];
}
```

1. URL for retrieving a timeline
2. Specify your parameters for the request.
3. Create the `SLRequest` object.
4. Specify the account for the request.
5. Perform the request and set up the handler.
6. Check the response status code.
7. Parse the request's response.
8. Reload the table view once you've retrieved updates that you want to display.

You're setting a `url` parameter with the URL for retrieving a user's timeline according to Twitter's API docs ❶. You're then preparing the one parameter you're passing in for this request, which is the select account's username ❷. Next, you create a new `SLRequest` by specifying your service type, your request method, URL, and parameters for the request ❸. After your request has been initialized, you first specify the authenticated account that will be making the request ❹. You can then perform the request and specify a request handler for when the request is finished ❺. It's within the request handler where you will inspect what Twitter returns. You first check to see if the status code for the response is 200 to ensure that the request went through smoothly ❻. You then parse the response data using the `NSJSONSerialization` class, which returns an array of dictionaries that represents each tweet ❼. Lastly you make a call to reload your table view within the main queue ❽.

DISPLAYING A USER'S TWITTER STREAM

Although you've retrieved a user's tweets, you can't see them yet. You haven't added a `UITableViewCell` to your view or even implemented the delegate or data source methods for the table view used in your stream view controller.

Click `MainStoryboard.storyboard` within your project navigator to hop into Interface Builder. You'll first drag a new `UITableViewCell` to your stream view's table view. In the inspector on the right side of your window, set its style to Subtitle and its identifier as `updateCell`.

Now jump back into the implementation of your stream view controller and add two methods. One of the methods you'll add will tell your table view how many rows to display, which will be a count of how many updates you have in your array. You currently have a placeholder method in its place that's returning `0`. Change this to return the real value of updates you're going to display.

```
- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section
{
    return [self.updates count];
}
```

Next, you'll add the `tableView:cellForRowAtIndexPath:` method to return a populated `UITableViewCell` for each row. The code shown in the next listing will replace what you previously added, which was just returning `nil`.

Listing 8.7 Return a table view cell for each tweet

```
- (UITableViewCell *)tableView:(UITableView *)tableView
➥ cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"updateCell";
    UITableViewCell *cell = [tableView
➥ dequeueReusableCellWithIdentifier:CellIdentifier];

    NSDictionary *update = self.updates[indexPath.row];

    if ([self.account.accountType.identifier
➥ isEqualToString:ACAccountTypeIdentifierTwitter])
    {
        cell.textLabel.text = [update objectForKey:@"text"];
        cell.detailTextLabel.text = [update
➥ valueForKeyPath:@"user.name"];
    }

    return cell;
}
```

❶ Checking if this is a Twitter account

Here you're checking to see if the account is for Twitter ❶ and then populating each cell using the dictionary for each update. For the tweet you're using the text value within the dictionary. For the user's name you're using the `user.name` key path.

If you run your application, you should see the selected user's latest updates, as shown in figure 8.18.

You can continue with this to provide a stream that contains all of the tweets of the users that the selected user follows and add more things like photos, pagination, and dynamic row heights. For now you're going to move on to retrieving your Facebook stream.

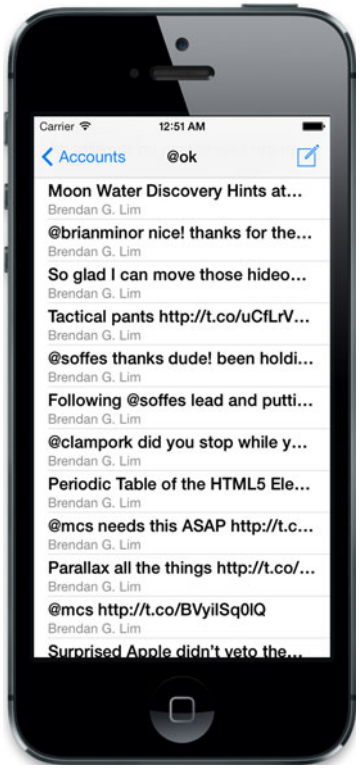


Figure 8.18 A user's Twitter stream shown in the stream view controller

8.3.2 Retrieving a Facebook news feed

Just like with retrieving a user's Twitter stream, what you have to do to retrieve a user's Facebook stream depends on the API that they expose. Facebook's Open Graph API will allow you to retrieve their news feed. You'll also need to request permission to retrieve their news feed.

Go to the `viewDidLoad` method and add a call to `retrieveFacebookStream` if the account is not a Twitter account. Your `viewDidLoad` method should look like the following:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.updates = [[NSMutableArray alloc] init];

    if ([self.account.accountType.identifier
➥ isEqualToString:ACAccountTypeIdentifierTwitter])
        [self retrieveTwitterStream];
    else
        [self retrieveFacebookStream];
}
```

You can now define your `retrieveFacebookStream`. What you'll have to do, which you didn't do for Twitter, is request additional permissions to read a user's stream. Earlier you requested permission to access a user's Facebook accounts. You passed in `email`

and `user_about_me` as the permissions you requested for the `ACFacebookPermissionsKey` parameter. You'll make almost the same request as before to gain access to an account but use different permissions.

In `retrieveFacebookStream` add the following to re-request permission to access Facebook accounts with the `read_stream` permission:

```
NSDictionary *options = @{ ACFacebookAppIdKey: @"YOUR-FB-APP-ID",
➥ ACFacebookPermissionsKey: @[@"read_stream"] };

ACAccountStore *accountStore = [[ACAccountStore alloc] init];
ACAccountType *accountType = [accountStore
➥ accountTypeWithAccountTypeIdentifier:ACAccountTypeIdentifierFacebook];

[accountStore requestAccessToAccountsWithType:accountType options:options
➥ completion:^(BOOL granted, NSError *error)
     {
         if(granted)
         {
             // Replace account and make an SLRequest
         }
     }
];
```

Here you're also retrieving new `ACAccount` instances with this new permission. You'll have to iterate through the array of accounts, and if the username matches the account currently set on your stream view, you should replace it.

Right underneath `if(granted)` add the following code to match and replace the user's account:

```
for (ACAccount *account in [accountStore
➥ accountsWithAccountType:accountType])
{
 if ([account.username isEqual:self.account.username])
     self.account = account;
}
```

Once you have the right account, you can create an `SLRequest` to retrieve a user's stream. Right after you set the account you'll make the request. Take a look at the next listing to see how you can retrieve a user's Facebook stream.

Listing 8.8 Retrieve a user's Facebook stream with an `SLRequest`

```
NSURL *url = [NSURL URLWithString:
➥ @"https://graph.facebook.com/me/feed"];
SLRequest *request = [SLRequest requestForServiceType:SLServiceTypeFacebook
                                        requestMethod:SLRequestMethodGET
                                                  URL:url
                                           parameters:nil];

[request setAccount:self.account];

[request performRequestWithHandler:^(NSData *responseData,
➥ NSHTTPURLResponse *response, NSError *error)
{
    if (response.statusCode == 200)
```

1. **URL for retrieving news feed**
2. **Create the `SLRequest`.**
3. **Specify the account for the request.**

```
    {
        NSError *parsingError = nil;
        self.updates = [[NSJSONSerialization
➥ JSONObjectWithData:responseData options:0 error:&parsingError]
➥ objectForKey:@"data"];

        dispatch_sync(dispatch_get_main_queue(),^{
            [self.tableView reloadData];
        });
    }
}];
```

4 Parse the request's response.

5 Reload the table view.

Within this method, you're specifying the open graph URL to retrieve a user's feed ❶. You then set up the `SLRequest` ❷ and then attach the account to this request ❸. This is so that the request is fully authenticated. Once you retrieve the user's news feed, you parse it using `NSJSONSerialization` ❹ and retrieve the updates by pulling back the data key. Then you reload your table view ❺.

Finally, you need to change the `tableView:cellForRowAtIndexPath:` to accommodate the different structure of data from Facebook's news feed. Within this method, add an `else` statement for when the account type is not from Twitter. You'll use the `story` property for the main text for the cell and the `from.name` key path to display the name of the user. Your `tableView:cellForRowAtIndexPath:` should look like the following listing.

Listing 8.9 Finalized `tableView:cellForRowAtIndexPath:` method to display updates

```
- (UITableViewCell *)tableView:(UITableView *)tableView
➥ cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"updateCell";
    UITableViewCell *cell = [tableView
➥ dequeueReusableCellWithIdentifier:CellIdentifier];

    if (!cell)
        cell = [[UITableViewCell alloc]
➥ initWithStyle:UITableViewCellStyleSubtitle
➥ reuseIdentifier:CellIdentifier];

    NSDictionary *update = self.updates[indexPath.row];

    if ([self.account.accountType.identifier
➥ isEqualToString:ACAccountTypeIdentifierTwitter])
    {
        cell.textLabel.text = [update objectForKey:@"text"];
        cell.detailTextLabel.text = [update valueForKeyPath:@"user.name"];
    }
    else
    {
        cell.textLabel.text = [update objectForKey:@"story"];
        cell.detailTextLabel.text = [update valueForKeyPath:@"from.name"];
    }

    return cell;
}
```

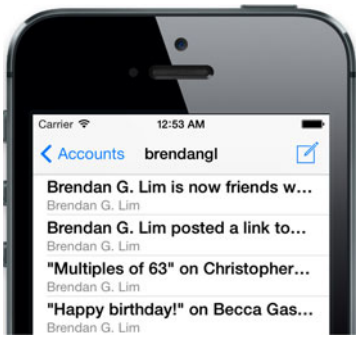


Figure 8.19 A user's Facebook stream populating our table view within the stream view controller

You can now run your application and see your Facebook updates. If everything has been set up properly, your stream view will look like what's shown in figure 8.19.

As with the Twitter stream you displayed, you can continue to modify this to provide a much richer experience for your users. You can examine the different types of updates to a user's stream and use different table view cells to display it with photos, links, comments, and the like.

8.4 Summary

You learned how to use the Accounts framework to gain permission to the accounts stored in iOS for Twitter and Facebook. You also learned how to post content using the Social framework. While using the Social framework you also saw how to retrieve a user's stream for both Twitter and Facebook. There are many places where you can expand on TweetBook to make it a really wonderful full-fledged Twitter and Facebook experience.

- The Accounts framework lets you access Twitter, Facebook, and Weibo accounts that you've been given permission to use.
- By using the `SLComposeViewController`, you can easily post updates to each of these networks.
- The `SLComposeViewController` gives you the ability to post text, images, links, location, and much more right out of the box.
- The Social framework helps you to interface with the APIs made available to you by Twitter, Facebook, and Weibo.
- You can use an account that you've been given permission to use to authenticate requests made with an `SLRequest`.
- By using both the Accounts and Social frameworks, you were able to create a simple application that allows you to list accounts, view your streams, and post content to each service.

9 Advanced view customization

This chapter covers

- Going beyond the Interface Builder with custom views
- Creating basic animations
- Using advanced animation techniques

iOS includes a large set of view classes for interacting with the user and displaying all kinds of controls such as buttons, tables, images, and scroll views. But sometimes your app has unique needs that aren't covered by default views. In an application, we all want the "wow" effect. Throughout this chapter you'll learn how to achieve it by customizing your views beyond the default properties. You'll be able to create both basic and advanced animations to enrich user interactions. This chapter will give you the tools to make the difference by creating unique experiences. To illustrate, let's see what a stunning app like Path would look like without view customizations, as shown in figure 9.1.

In order to learn about animations and customizing views, in this chapter you'll be creating an app called AnimatedClock, as shown in figure 9.2. AnimatedClock is a fully functioning clock application. You're going to visually represent time by an animated pendulum and an analog clock with animated clock hands. While creating

Figure 9.1 Difference between what Path would look like using only default views (left) and what Path actually looks like using customized views and effects (right)

your app you'll learn how to customize default iOS views by subclassing the parent class for display objects, the `UIView`.

9.1 Going beyond the Interface Builder with custom views

You learned in the previous chapters how to create your application interfaces by dragging and dropping controls from Interface Builder. You also learned that you can assign a custom class to these dropped controls by changing the `Class` attribute on the identity inspector, as shown in figure 9.3. Previously you've done that only to customize view controllers. In this chapter you're going to use this method to create custom views and associate them with your Interface Builder's components.

Interface Builder is a great tool for configuring a wide set of component parameters, and it's usually enough to create simple applications. But if you want to go the extra mile and implement stunning interfaces, chances are you'll need to create a couple of custom views for your applications as you're going to do with your Animated-Clock application.

When we say "view," what we're actually referring to is `UIView`. `UIView` is the parent class for all display objects. Every display component that we've used in this book, such as `UIButton`, `UITableView`, `UILabel`, and `UIWebView`, is a subclass of `UIView`. That means that knowing how `UIView` works will help you not only for `UIViews` but also for all other display objects you customize. Let's start by seeing what happens behind the scenes when a view is placed on the screen.

Figure 9.2 AnimatedClock, a fully functioning clock application with stunning animations and a customized interface that we'll build together by the end of this chapter

In order to place a view on the screen you must create the view. If you're doing it programmatically, the default initialization method is `initWithFrame:`. This method defines the position of your new view, which is relative to its (future) parent view and the size in points. Once the view is created, you need to place it on the screen by adding it as a subview of another view. `UIView` defines the following methods for adding subviews:

- `(void)addSubView:`
- `(void)insertSubView:aboveSubview:`
- `(void)insertSubView:belowSubview:`
- `(void)insertSubView:atIndex:`

The first one is the most commonly used, and it inserts a view following the painter's algorithm; the name of the algorithm refers to the technique painters employ of painting each element on top of the last one, as figure 9.4 shows.

The other three methods are used to add a subview and manually manage the view hierarchy. You can use them to add a view on top of or below an existing view. Managing

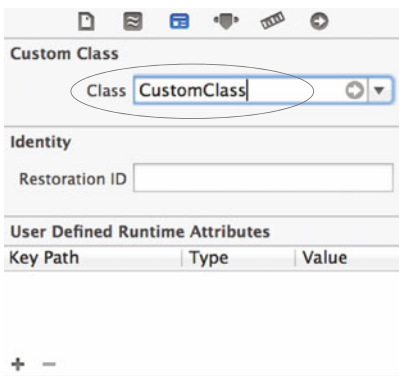


Figure 9.3 As you learned in previous chapters, you can customize classes from Interface Builder by changing the `Class` attribute.

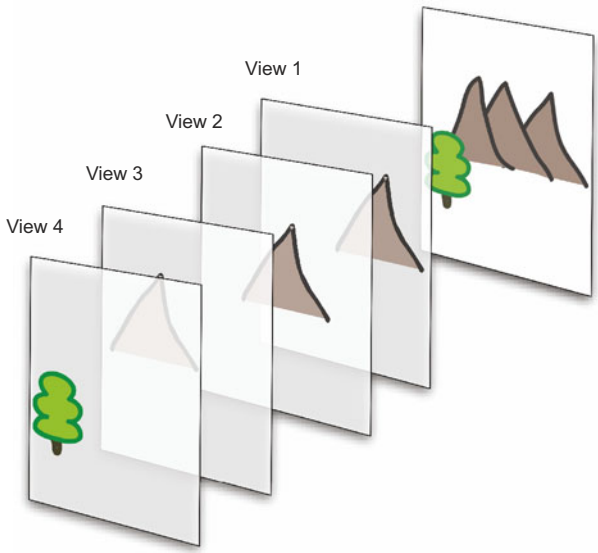


Figure 9.4 The painter's algorithm shows how each component represents layers one on top of each other.

view hierarchies is a crucial part of your applications because it not only influences the visual appearance of your application, but also it defines how the views react to touch events. You'll learn more about this in this section.

As we said, Interface Builder is the most convenient way to build view hierarchies because you can create interfaces graphically by dragging and dropping views. And even when creating custom views you'll still use Interface Builder to position them. When a view is created using Interface Builder, behind the scenes Cocoa will initialize it by calling the method `initWithCoder:`.

Once the view is initialized and positioned on a parent view, the view will draw its content and call its `drawRect:` method internally.

Let's summarize all this so that you understand how you can customize views:

1. When initializing a view, you call either `initWithFrame:` or `initWithCoder:`. This is important when creating custom views or when subclassing display elements because you're going to write all the initialization code such as gestures, instance variable initializations, and the like there.
2. You call `drawRect:` each time the view is drawn. This is useful when, for example, you want to define a tile background or create custom shapes.
3. You add subviews one on top of the other.

It's time to create the AnimatedClock application you're going to use throughout the chapter. Go to Xcode and create a new, single-view application named AnimatedClock. As you did in the previous chapters, you'll set the class prefix to `IA`, as shown in figure 9.5.

For this application you'll use a set of images to enrich your interface. Figure 9.6 shows all the images you're going to use and their filenames.

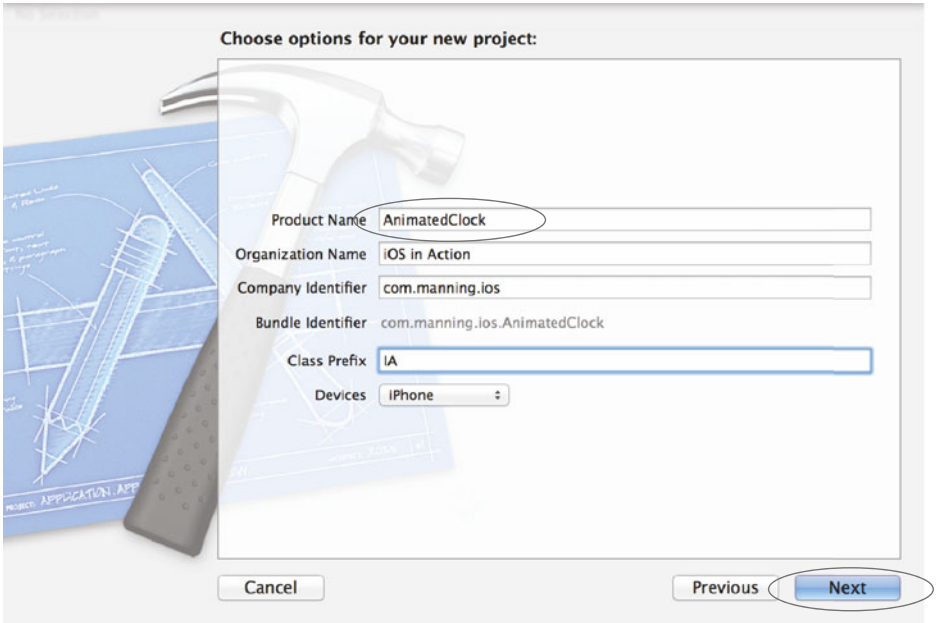


Figure 9.5 Creating a new project named AnimatedClock

In order to include those images in your project, you'll create a new group called Images. Then you'll select all the files from the Finder and drag and drop them into your recently created Images group, as shown in figure 9.7. Once you've finished dragging

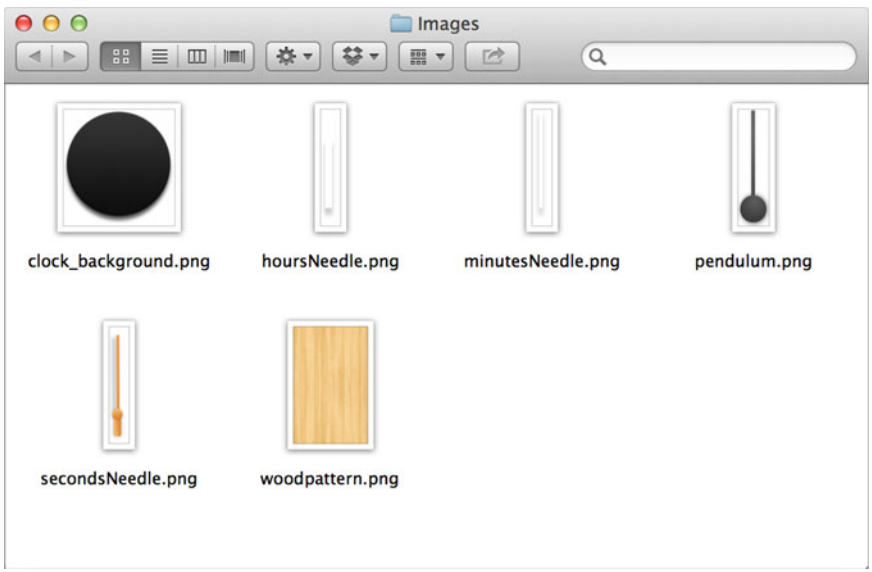


Figure 9.6 Images used in the AnimatedClock application

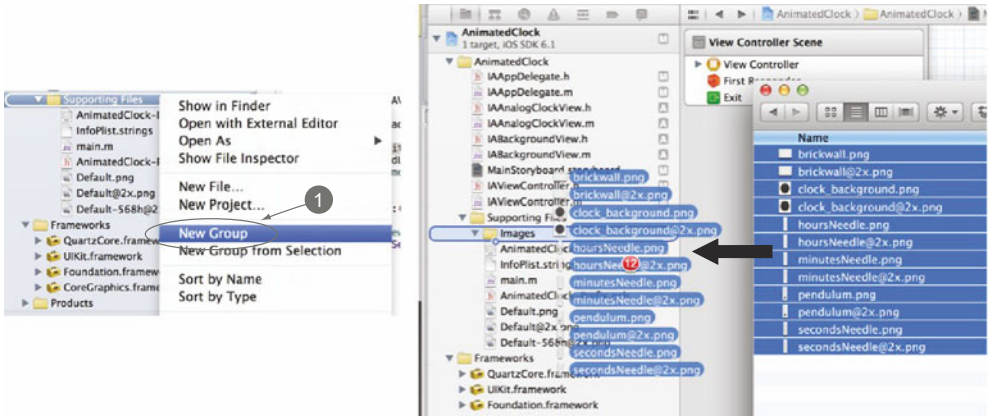


Figure 9.7 Include the images in your project by (1) creating a new group and then dragging and dropping files from the Finder.

and let go, a modal will appear asking you if you want to save these items into your project tree. Select Copy Items into Destination Group's Folder and click Finish.

Now that you have the assets you need to build the interface for your project, you're going to open the autogenerated storyboard Main.storyboard using Interface Builder and create your interface. For that you'll add the view you're going to customize in this section. This view will represent your pendulum clock. In order to do this, you'll drag and drop a `UIView` from the Object Library and place it in your main view controller, as shown in figure 9.8. Once you've finished dragging and let go, set the view's background color to Clear Color.

Then you will include the image views that you'll use as the clock's hands, the pendulum, and the clock base. For that, you'll drag and drop five `UIImageViews`, place them to form the clock, and set the Image parameter of each view according to the corresponding filename. For example, the clock base is clock_background.png. When positioning the views, make sure the three clock hand views have the same dimensions and are in the exact same x and y positions, as shown in figure 9.9. Also make sure to set the Mode as Center for all `UIImageViews`. After doing that you'll add four labels that you'll use as your clock hours. The labels are "12," "3," "6," and "9." The reason why you're including files with an @2x suffix is because some devices include a higher definition (Retina) display. So what you need to do is provide another version for all of your images: one that's double the size. If you name your image with double the size with an @2x extension, whenever you try to load an image with `[UIImage imageNamed:...]` or similar APIs, it will automatically load the @2x image instead on the Retina display. Note that you don't need to specify the @2x Retina version when selecting images in Xcode because they're called automatically when the application is running.

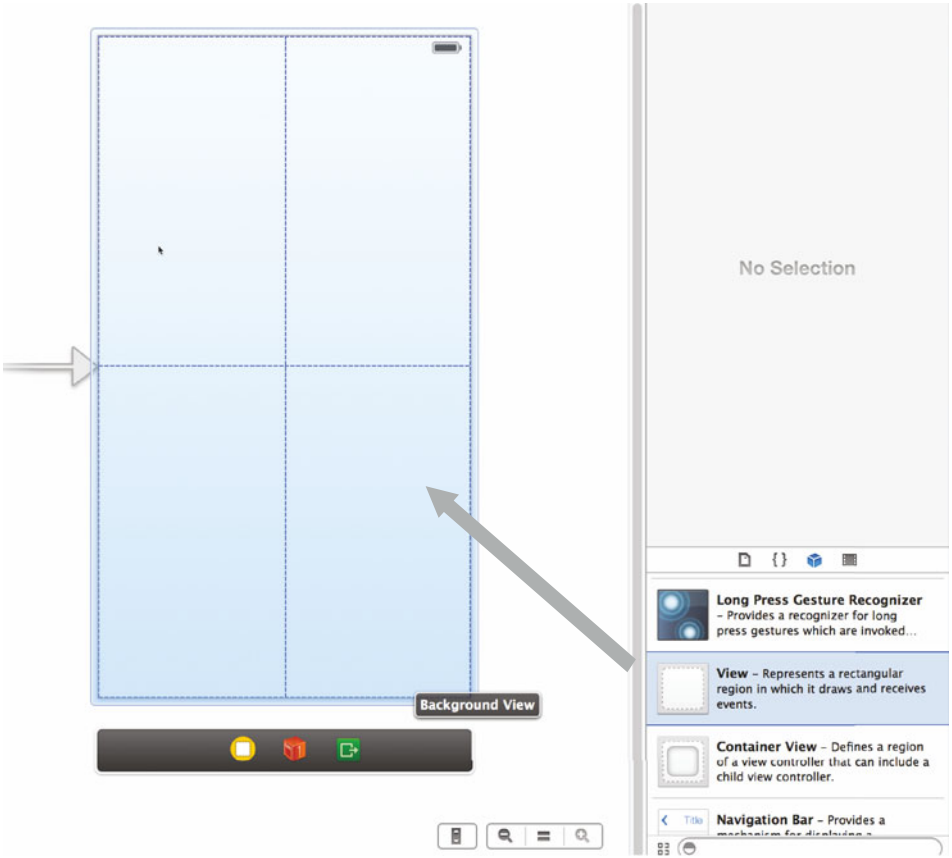


Figure 9.8 Adding the view you're going to use as your pendulum clock

The image views shown in figure 9.9

1. The pendulum is the bottom view and therefore is the first one you have to add.
2. Clock base is just an image, and there are four labels on top of it.
3. Three clock hands in the exact same position represent seconds, minutes, and hours.

Your interface is ready! Go ahead and run the application using the 4-inch Simulator. Figure 9.10 shows what it looks like so far.

As you can see, the background is white and the clock is not moving whatsoever. You'll fix the animation soon, but first you'll set a nicer tile background. Tile backgrounds are created by placing a small pattern (tile) repeatedly across the x and y axes. The benefit of this is that you can create backgrounds of any size by using only one (usually) small image. You're going to use a wood image as a tile, and the background will look like a wall afterward.

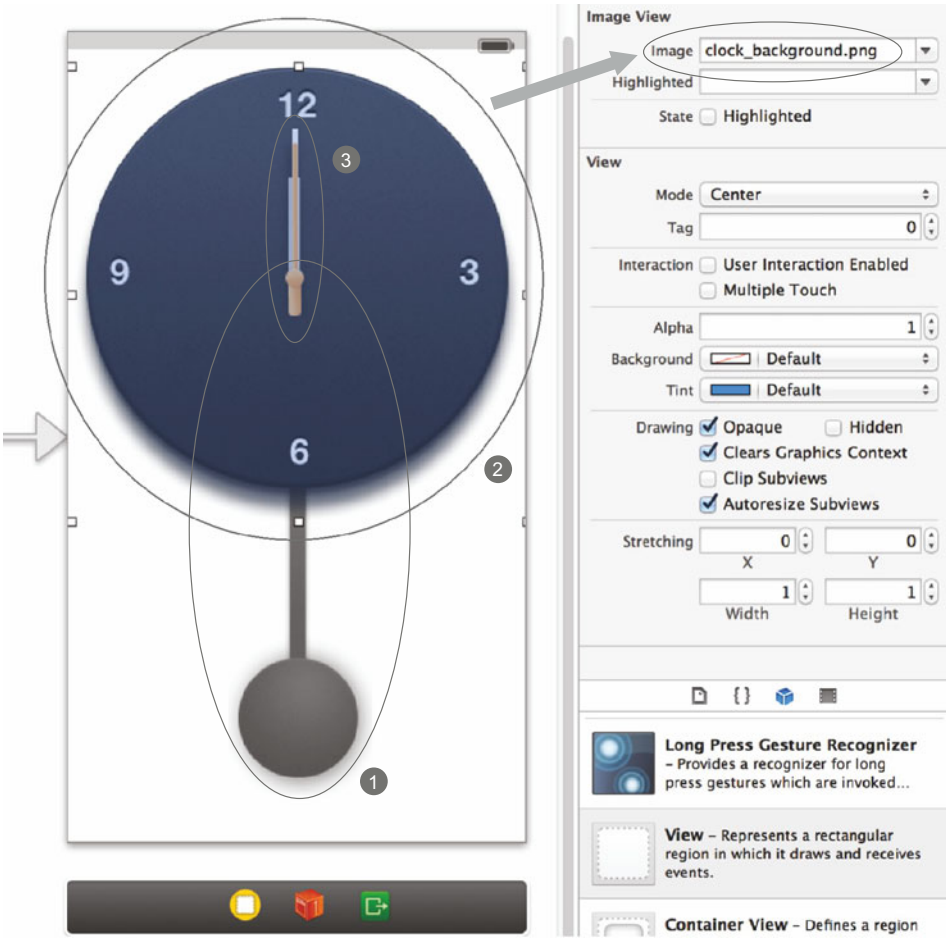


Figure 9.9 Creating the clock interface by placing five image views

For that, you'll create a new class called `IABackgroundView` and make it a subclass of `UIView`. To be able to draw the background inside your view, you need to override the method that performs the drawing. Any idea what this method would be? You guessed right! The method is `drawRect:`. This method is called internally when the view needs to draw (or redraw) its content. It's called when you first display the view and when the view is altered. In the lifecycle of the view it could be called thousands of times. Any visual change on the view or on a subview will trigger a redraw cycle and therefore call `drawRect:`.

Let's write the logic to create your tile background, shown in the following listing. Open the class you just created by going to Xcode and locating the IABackground-View.m file.

Listing 9.1 Overriding `drawRect` and drawing your tile background

```
- (void)drawRect:(CGRect)rect
{
    CGContextRef context = UIGraphicsGetCurrentContext();
    UIImage *tile = [UIImage imageNamed:@"woodpattern"];
    [[UIColor colorWithPatternImage:tile] set];
    CGContextFillRect(context, self.bounds);
}
```

❷ Setting a color generated with your tile image as a pattern

❶ Getting the current `UIView` graphic context

❸ Filling the pattern across the entire `UIView` bounds

On the first line ❶ of the method you obtain the `UIView` graphic context. A graphic context represents a drawing destination, and it contains parameters that the iOS drawing system needs to perform subsequent drawing commands such as stroke and fill colors, clipping areas, styles, and others. One of the parameters you're changing on the context you just obtained is the fill color ❷. You configure the fill color as an image generated by a pattern. As a side note, you'll call it "color" even if it's not strictly a "color" to be consistent with iOS nomenclatures. Line ❸ fills a rectangle (in this case you're using `self.bounds`, which is your entire view) using the pattern image you just set.

Now that you have the code in place, you'll define the custom class of the root view (note: this is the first view you find on the view hierarchy, not the container view you added on top) and use the file you just created. For this, you'll open Interface Builder, select the view, and change the Class parameter from the identity inspector, as figure 9.11 shows.

Now compile and run the app again to see how your new background looks. As you can see in figure 9.12, the tile is repeated, generating a full background.

Figure 9.10 Your application looks like this after adding a couple of views using Interface Builder.

9.2 Creating basic animations

Animations are used extensively in iOS applications to enrich the user interface. They convey feedback to the user about application changes and input events. iOS offers several animation mechanisms that let you create sophisticated animations very easily with only a few lines of code. All animation techniques you will learn in this chapter use a built-in infrastructure called Core Animation. Core Animation is an essential part of iOS applications, and you've been using it since chapter 1 without knowing it. Internally every transition between view controllers, every navigation

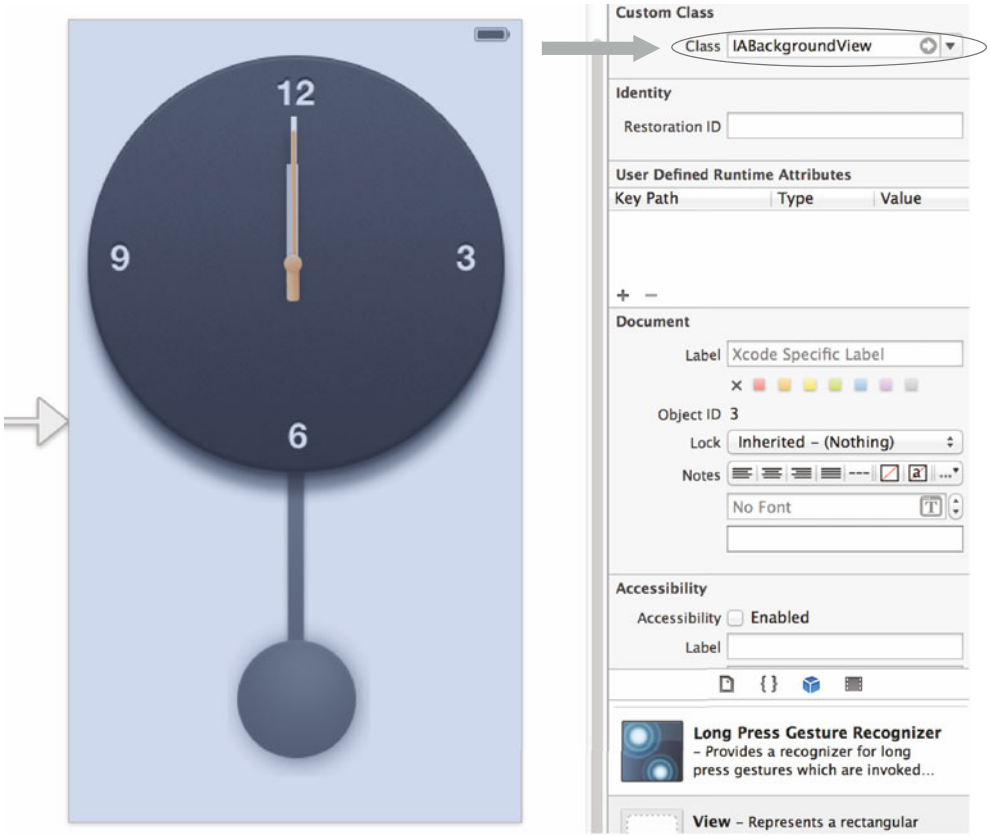


Figure 9.11 Changing the main view class to `IABackgroundView`

bar effect, and the like use Core Animation. There are two different ways of animating views on iOS:

- Basic `UIView` animations using UIKit abstractions. (UIKit is the framework iOS uses to manage user interfaces. Basically every view you've used before such as buttons, tables, and the like is part of UIKit.) You'll use this kind of animation for animating property changes, for example, changing the position, the rotation, or the transparency of your view.
- For more complex and sophisticated animations you'll use Core Animation directly (section 9.3).

Creating basic animations is very straightforward. You only need to create an animation block by calling the method `animateWithDuration:animations:` of the `UIView` class, setting the duration of the animation as follows:

```
[UIView animateWithDuration:1.0 animations:^{
    // Here we have to modify our view(s) properties.
}];
```

As you can see, the animation block is just a regular block, and inside it you'll set the properties of the view that you want to animate. For example, you can set the alpha (transparency) property to 0. This will gradually change the alpha from 1 (the default alpha value of every view) to 0 in a second. By doing that, you'll see a fade-out effect. Using the previous example,

```
[UIView animateWithDuration:1.0 animations:^{
    [ourViewInstance setAlpha:0.0];
}];
```

You can change more than one parameter inside the block, and all the animations will start in parallel.

Figure 9.12 The background is generated by repeating a pattern image. You can see the actual image (tile) inside the rectangle.

Let's go back to the AnimatedClock application. So far you created your clock's interface and it looks great, but you didn't write the code to show the current time, as a clock would do. Next, you'll create a customized view, responsible for moving the clock's hands. For that, you're going to open Xcode and create a new class named `IAAnalogClockView` and make it a subclass of `UIView`, as shown in figure 9.13.

This class will be in charge of rotating the three clock hands (seconds, minutes, hours) according to the current time and oscillating the pendulum. Before going any further, you need to understand how rotation works. iOS graphic frameworks use transform views by creating a transformation matrix. To understand this, imagine you have a drawing on a paper. The paper represents a two-dimensional space, and so you can draw a Cartesian plane with x and y axes, as shown in figure 9.14. Let's say that you draw the axes on the table such that you move the paper. The axes remain in the same place. When the drawing is rotated you'll see that each point of your previous drawing is located in a different x and y position.

Transformation matrixes are a mathematical way of translating coordinates from the initial state (left part of the image) to the new transformed state (right). Those matrixes are used to rotate, scale, and skew objects you draw in a view. Creating these matrixes is not trivial and would require some knowledge of mathematics. Take, for example, the clockwise rotation matrix, shown in figure 9.15.

Luckily, you don't typically need to create matrixes directly. For example, in order to create the rotation matrix you'll call the function `CGAffineTransformMakeRotation` with an angle as the parameter, and it will create the matrix for you.

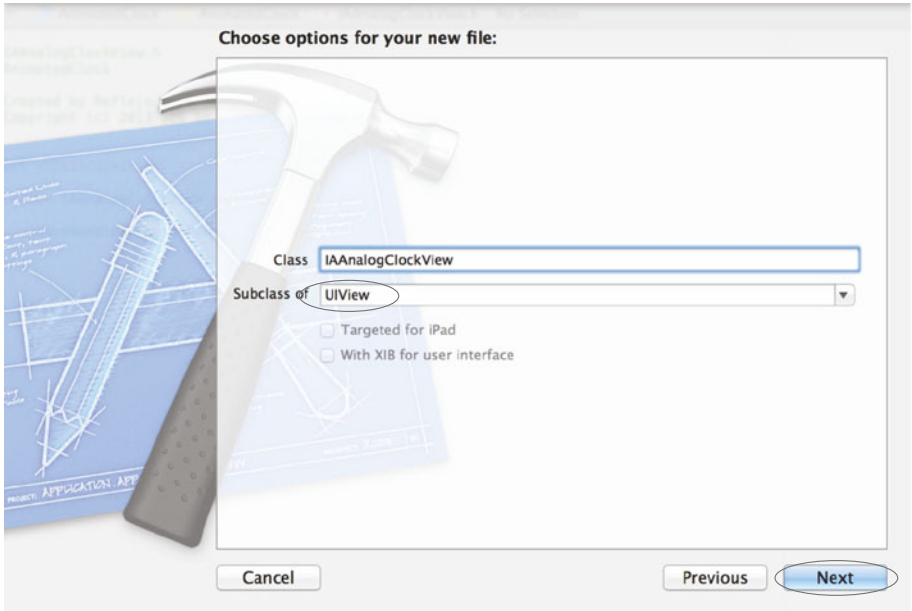


Figure 9.13 Creating a class named `IAAnalogClockView`, which is a subclass of `UIView`

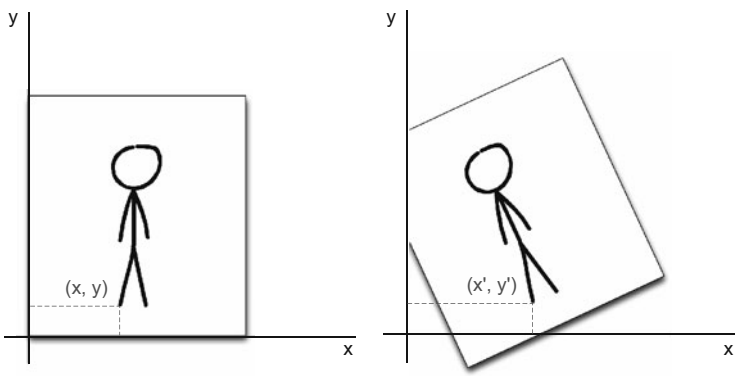


Figure 9.14 Example of rotation using a sheet of paper and a fixed coordinate system

There's one more thing you need to know about transformations before going back to your application. When you rotated the paper, you did it in such a way that the center of the paper remained in the middle. Try it yourself: put your finger in the middle of the paper and rotate the sheet. Now put your finger on top of the paper and rotate

$$\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 9.15 The clockwise rotation matrix

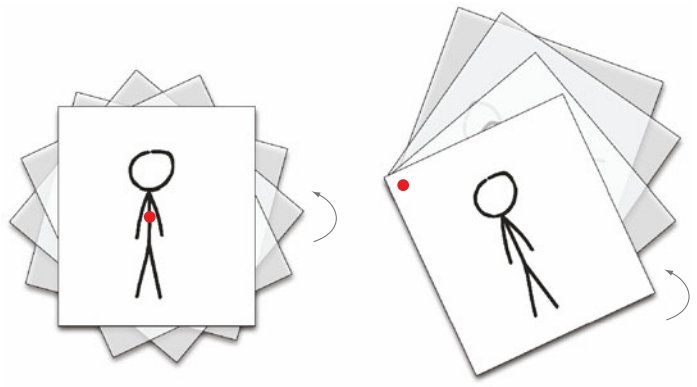


Figure 9.16 How rotations change when you move the anchorPoint

the sheet again, as shown in figure 9.16. The point where your finger is positioned is called the anchorPoint. The anchorPoint values range from 0 to 1 such that (0, 0) is the top left and (1, 1) is the bottom right. By default the anchorPoint is (0.5, 0.5), which is the middle of the view.

You're ready to go back to your application and create the logic to rotate the clock's hands. First, you'll need to link your custom clock view, clock hands, and pendulum from Interface Builder to your code in order to control them. For that you'll change the class of the view you created before, the one that holds the clock's hands, pendulum, and base, by selecting it and changing the Class property from the identity inspector to `IAAnalogClockView`. Next, you'll link the pendulum and clock's hands to this view. For that you'll open the assistant editor in Interface Builder by selecting View > Assistant Editor > Show Assistant Editor in the application menu. With the hours hand selected in your interface, hold down the Control key while clicking and dragging from the view to your `IAAnalogClockView`'s class definition in the assistant editor. Once you've finished dragging and let go, a modal will appear asking you to name the outlet you're setting on your class for this view. Name it `hoursHand` and then click Connect. You'll do the exact same procedure with the seconds hand, minutes hand, and pendulum. The names of these variables will be `secondsHand`, `minutesHand`, and `pendulum`, respectively.

In the following listing you'll write the code in charge of rotating the clock's hands.

Listing 9.2 Method used to rotate the clock's hands

```
- (void)moveHand:(UIView *)hand toAngle:(CGFloat)angle
{
    [[hand layer] setAnchorPoint:CGPointMake(0.4f, 0.75f)];
    [[hand layer] setPosition:CGPointMake(165.0f, 155.0f)];
    [UIView animateWithDuration:0.5 animations:^
    {
```

1 **Anchor point is set as the bottom of the clock's hands inside the image.**

2 **After changing the anchor point, you reposition the view to the center of the clock's base.**

3 **Create the animation block with duration of half a second.**

```
        CGAffineTransform matrix = CGAffineTransformMakeRotation(angle *
➥ M_PI / 180);
            [[hand layer] setAffineTransform:matrix];                    ❹
    }];
}
```

❹ Rotate the clock's hand to a given angle.

The first thing you'll notice in the code is that you're moving the anchor point to a specific place ❶. You created your clock hands' images in such a way that the same anchor point position works for all the hands, but this number makes sense only after seeing how the images look. Take a look at figure 9.17 to see why the point is (0.4, 0.75). After setting the anchor point, the position of your view will change; that's why you're repositioning it ❷ to the center of your clock's base, which is the point (165, 155). Next, you define an animation block ❸ and inside it you set your view transformation as a matrix containing the rotation as you learned before. By doing this, you'll create an animation of half a second length that will rotate the hand. Note that `CGAffineTransforMakeRotation` takes the angle as a parameter in radians. The formula to convert from degrees to radians is angle × pi / 180, and that's what you're using to rotate the view in ❹. If you compile and run the application at this point, you'll get a strange error about CALayer being a forward declaration. But don't worry! You're about to fix that.

In order to apply the transformations to the layer of your view as you're doing in listing 9.2, you need to include the QuartzCore framework to your project. In order to do that, you'll add a Linked Framework from the Summary tab on the right, as shown in figure 9.18. The framework you're going to add is QuartzCore.framework. Once you finish doing that, you'll open the file IAAnalogClock.m again and add the following line to the very top:

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

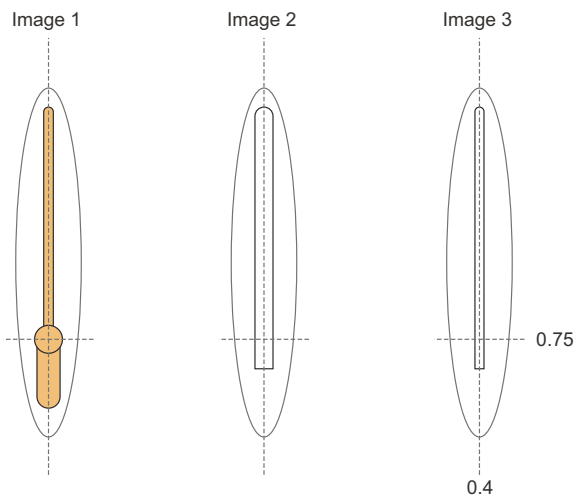


Figure 9.17 All your clock hands are designed to have the same anchor point: (0.4, 0.75).

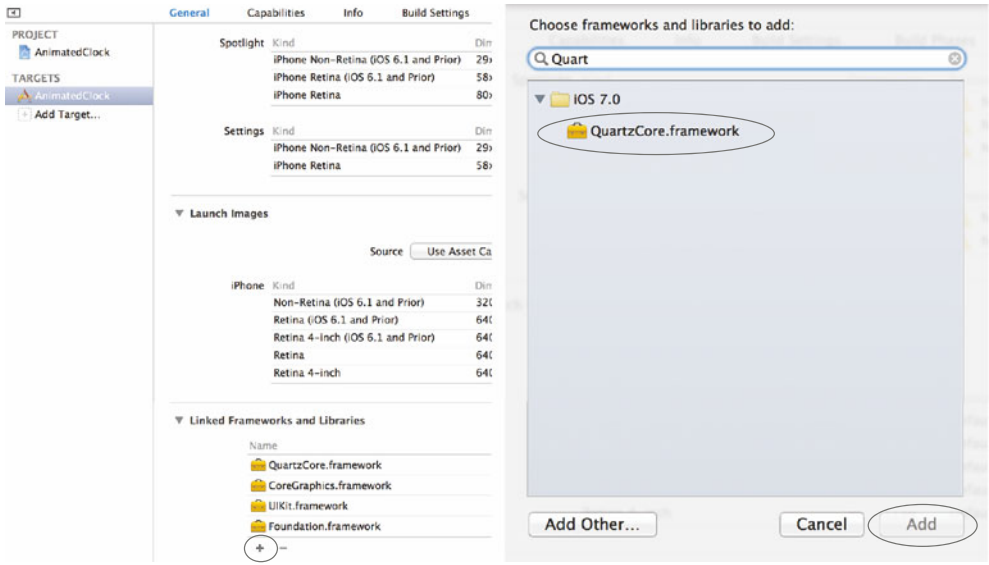


Figure 9.18 Adding QuartzCore.framework to your project

The next thing you'll add to this file is the method that will be called once per second, as shown in listing 9.3. This method will be responsible for moving the clock's hands.

Listing 9.3 Method in charge of moving the clock's hands to the correct angle

```
- (void)moveHandsToLocalTime
{
    NSInteger comp = (NSHourCalendarUnit | NSMinuteCalendarUnit |
     NSSecondCalendarUnit);
    NSDateComponents *components = [[NSCalendar currentCalendar]
     components:comp fromDate:[NSDate date]];

    [self moveHand:self.hoursHand toAngle:([components hour] % 12) * 360.0f
    / 12.0f];
    [self moveHand:self.minutesHand toAngle:[components minute] * 360.0f /
    60.0f];
    [self moveHand:self.secondsHand toAngle:[components second] * 360.0f /
    60.0f];
}
```

❶ **The calendar object contains methods that make it easy for you to separate hours, minutes, and seconds from the current time.**

❷ **Animating the clock's hands using the method you created in listing 9.2**

In order to move the clock's hands you need a way to extract hours, minutes, and seconds from the current date. For that, you use a class named `NSDateComponents` ❶. This class encapsulates the date and time fragments in an object-oriented manner, allowing you to retrieve the hours, minutes, and seconds very easily by calling the methods `hour`, `minute`, and `second`. Now you need to calculate the angle your clock's hand should take each second. Let's think this through: the clock's hand will have an angle of 360° once it reaches its last value. So, the hours hand will have an angle of 360° when the hour is 12, and the minutes and seconds hands will have an angle of 360°

once they equal 60. Knowing that, let's do cross multiplication. The calculation for the hours hand would be:

If you apply the same logic to minutes and seconds, you'll end up with these three formulas:

```
hour_hand_angle = hour * 360 / 12
minute_hand_angle = minute * 360 / 60
second_hand_angle = second * 360 / 60
```

These formulas give you the angles you're using in listing 9.3 as parameters of the calls to `moveHand:toAngle:` ❷.

If you try to compile and run the application, you'll see that the clock still doesn't move. That's because you have all the logic in place for rotating the clock's hands, but you're not calling it every second yet. You'll fix that in the next listing.

Listing 9.4 Triggering the `moveHandsToLocalTime:` method every second

```
- (void)didMoveToWindow
{
    [NSTimer scheduledTimerWithTimeInterval:1.0 target:self
➥ selector:@selector(moveHandsToLocalTime) userInfo:nil repeats:YES]; ❶
}
```

❶ **Creating a timer that will call your method every second**

`didMoveToWindow` is another method defined in the `UIView` class. This method is called internally by iOS when the view changes its window property. On iOS that's similar to saying "when the view is placed on the screen." Here you're initializing a timer that will call the method you created in listing 9.3 every second ❶.

You're finished! Compile and run the application. The clock should show the current time, and you should be able to see the clock's hands animations. But hold on. Did you notice something strange? The pendulum isn't moving! We're going to address this in the next section.

9.3 Using advanced animation techniques

In the previous section you learned how to create animations very quickly by changing the view properties inside an animation block. That's usually enough to create basic interface animations, but sometimes you'll need to create more complex transitions. Take, for example, the animation shown in figure 9.19.

Whereas a property-based animation, like the one you've been using for the clock's hands, changes a property from a start value to an end value gradually in a specific timeframe, there's another way of defining animations: by using `CAAnimation` objects directly. This method gives you total control of your animations, timing, and positions of your views at any given moment of the transition. The disadvantage of using `CAAnimation` directly is that your code will get more complex.

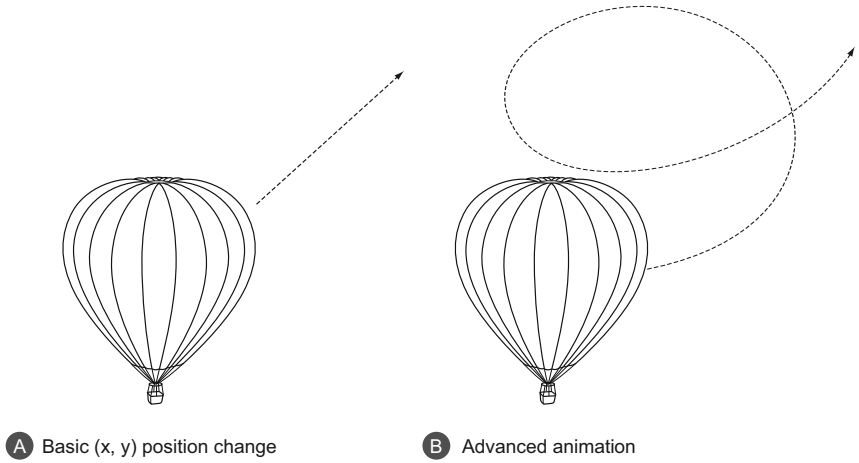


Figure 9.19 Figure A shows a basic animation changing the frame position. Figure B shows an animation with the same end point but a more complex trajectory.

Let's go back to your AnimationClock application and write the code to animate your missing piece: the pendulum. In order to animate the pendulum, you're going to use the `CAKeyframeAnimation` object, which lets you create animations in segments. This is useful when instead of moving (or rotating) an object from point A to point B, you need to define a custom path, as the previous hot-air balloon image shows (figure 9.19).

For your pendulum animation you want to mimic how it would move in real life. For that you have three states: the first one is when the pendulum is at equilibrium position (0°) and the other two are both extremes (15° and -15°, respectively). As the pendulum moves from the equilibrium position to the extremes, the velocity decreases. The farther the pendulum has moved from the equilibrium position, the slower it moves; the closer the pendulum is to the equilibrium position, the faster it moves. For that you'll split your animation into four segments, as shown in figure 9.20.

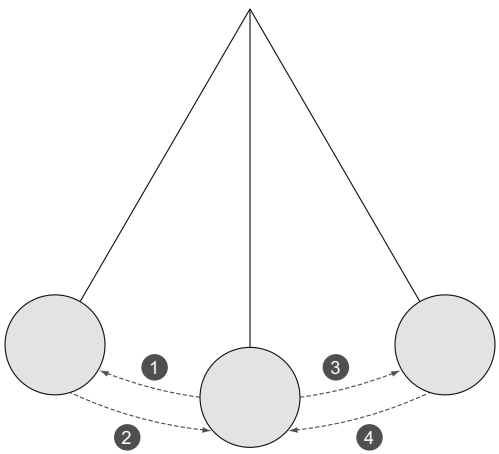


Figure 9.20 Pendulum animation used in the AnimatedClock application, separated into four segments

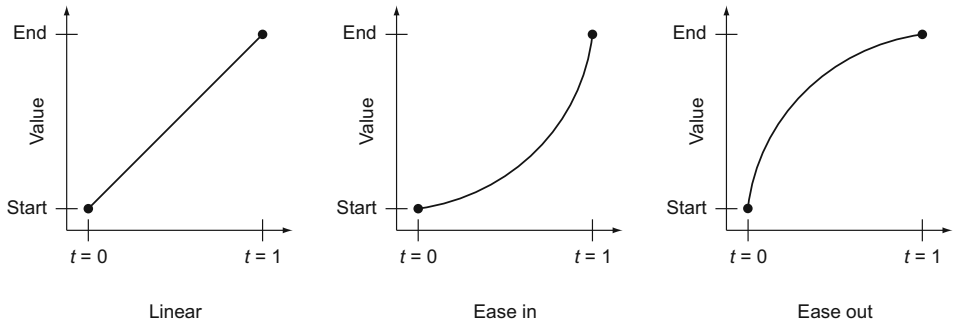


Figure 9.21 Most common timing functions. By default iOS uses the Linear function.

The initial position of your pendulum is the equilibrium point, and as such, the segments are defined as follows:

1. 0° to 15°—Goes from the equilibrium point to the left extreme. As the pendulum moves, the velocity decreases.
2. 15° to 0°—Goes from the left extreme to the equilibrium point. As the pendulum moves, the velocity increases.
3. 0° to -15°—Goes from the equilibrium point to the right extreme. As the pendulum moves, the velocity decreases.
4. -15° to 0°—Goes from the right extreme to the equilibrium point. As the pendulum moves, the velocity increases.

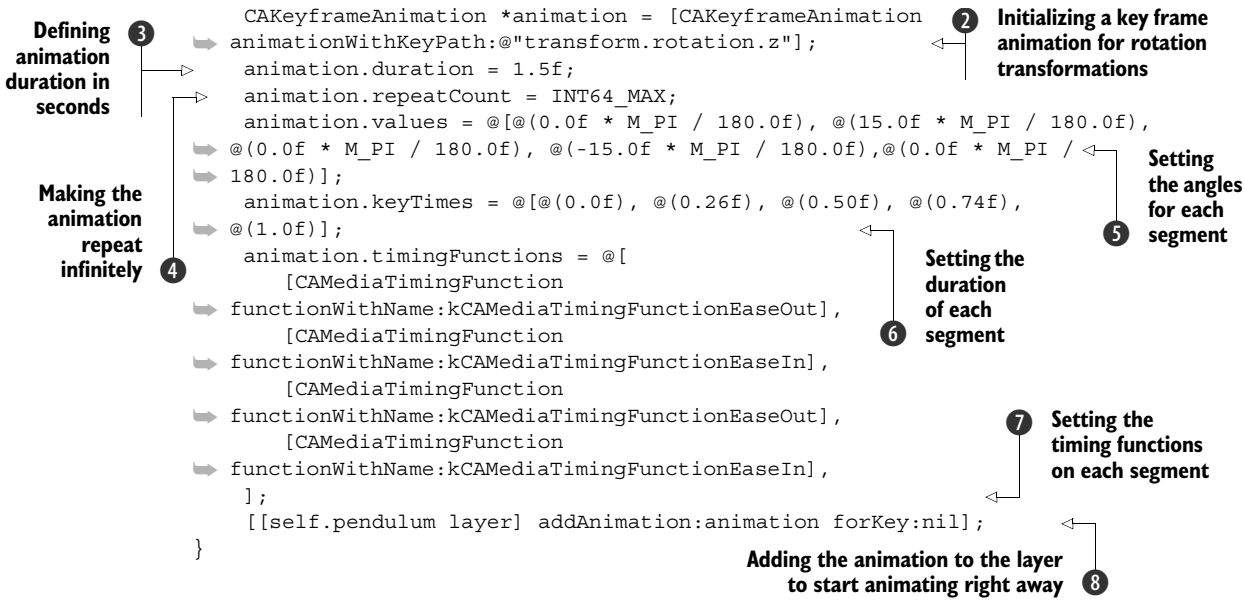
When you're creating an animation, what happens to your animated properties while your animation is running is almost as important as what happens at the end. To illustrate this, let's say you want to move a view from point A to point B in one second. By default, what you'll see is an animation that seems to neither speed up nor slow down. The view will move at a constant rate. In other words, your view position will change *linearly* with time. But when defining the pendulum's segments the velocity should decrease or increase according to the pendulum position. In order to achieve those velocity changes, you use timing functions. Timing functions define the rate at which a property value changes from one value to another over time. In figure 9.21 you'll see some of the most common timing functions. The ease-in function causes the animation to begin slowly and then speed up as it progresses. Ease-out causes the animation to begin quickly and then slow as it completes.

Let's apply everything you've learned to your AnimatedClock application. You'll create the segmented animation on your pendulum using `CAKeyframeAnimation`. Open the file IAAnalogClockView.m using Xcode and add the method in the following listing.

Listing 9.5 Method in charge of pendulum oscillation

```
- (void)oscillatePendulum
{
    [[self.pendulum layer] setAnchorPoint:CGPointMake(0.5f, 0.0f)];
    [[self.pendulum layer] setPosition:CGPointMake(165.0f, 155.0f)];
```

1 **Defining the anchor point and repositioning the view afterward**




```
    CAKeyframeAnimation *animation = [CAKeyframeAnimation
➥ animationWithKeyPath:@"transform.rotation.z"];
    animation.duration = 1.5f;
    animation.repeatCount = INT64_MAX;
    animation.values = @[@(0.0f * M_PI / 180.0f), @(15.0f * M_PI / 180.0f),
➥ @(0.0f * M_PI / 180.0f), @(-15.0f * M_PI / 180.0f),@(0.0f * M_PI /
➥ 180.0f)];
    animation.keyTimes = @[@(0.0f), @(0.26f), @(0.50f), @(0.74f),
➥ @(1.0f)];
    animation.timingFunctions = @[
        [CAMediaTimingFunction
➥ functionWithName:kCAMediaTimingFunctionEaseOut],
        [CAMediaTimingFunction
➥ functionWithName:kCAMediaTimingFunctionEaseIn],
        [CAMediaTimingFunction
➥ functionWithName:kCAMediaTimingFunctionEaseOut],
        [CAMediaTimingFunction
➥ functionWithName:kCAMediaTimingFunctionEaseIn],
    ];
    [[self.pendulum layer] addAnimation:animation forKey:nil];
}
```

The first thing you have to set, as you did before, is the anchor point. You use (0.5, 0.0) as the anchor point ❶ because your pendulum anchor point is located at the top center of the image. Then you need to initialize the `CAKeyframeAnimation` instance that you'll use to animate your pendulum in segments. When initializing `CAKeyframeAnimation`, you have to define which property of the view you're going to animate. In this case you want the pendulum to rotate, and so you set the animation key as `transform.rotation.z` ❷, which is the rotation property of the transformation, as you learned previously. Next, you configure your animation's properties such as total duration ❸ and how many times you want the animation to repeat. You assign a very (very) big number to `repeatCount` in order to repeat the animation infinitely ❹.

Now it's time to create the animation segments you defined. Each segment is defined by using three properties: end value ❺, duration ❻, and timing function ❼. You define those events by setting the arrays `values`, `keyTimes`, and `timingFunctions`. Figure 9.22 shows how those arrays are constructed.

One thing to notice is that `keyTimes` values are not seconds. Each value in the `keyTimes` array is a number between 0.0 and 1.0 that defines the time point (specified as a fraction of the animation's total duration) at which to apply the corresponding key frame value. The animation is finally applied to the pendulum layer by calling `addAnimation:forKey:` method ❽.

Finally, you call the method you just created when the clock is initialized. For that you'll open the file IAAnalogClockView.m, locate the method `didMoveToWindow` you created in the previous section, and add the `oscillatePendulum` call to the end of this method as follows:

```
    [self oscillatePendulum];
```

	values	keyTimes	timingFunctions
0	0°	0.0	-
1	15°	0.26	Ease In
2	0°	0.50	Ease Out
3	-15°	0.74	Ease In
4	0°	1.0	Ease Out

Figure 9.22 Animation segments you defined previously but now represented as arrays

Now it's show time! Compile and run the application to see the pendulum moving along with the three clock hands.

9.4 Summary

Throughout this chapter, we dove deeper into different ways of working with views to help you make more immersive, rich applications. We created an application that shows the current time represented by a pendulum clock. This clock has the particularity that its hands as well as the pendulum move in real time by using basic animations for the clock's hands and a customized animation to make the pendulum transitions mimic real life. Following are some key topics we covered:

- Stunning interfaces always include some kind of view customization; for that you subclass `UIView` classes.
- View hierarchies define how the views react to touch events.
- Interface Builder is the most convenient way to build view hierarchies.
- A graphic context represents a drawing destination, and it contains parameters to perform subsequent drawing commands such as stroke and fill colors, clipping areas, styles, and others.
- For very basic animations you use `UIView` class methods.
- For sophisticated animations you use Core Animation.

10 Location and mapping with Core Location and MapKit

This chapter covers

- Introducing the Core Location and MapKit frameworks
- Retrieving your current location, heading, and speed
- Geocoding locations to display user-friendly locations
- Displaying a map of your location

Whenever I can't figure out where to eat, I turn to my favorite restaurant review app to make the difficult decision for me. It shows me all of the nearby restaurants and lists them according to their rating. I choose a place that I believe will be able to satisfy my enormous appetite. Not knowing where this place is, I load the directions into my favorite maps app to give me walking directions. As I turn, the map turns with me, orienting itself so that I know where to go. After a few minutes of walking, I'm ready to stuff my face.

There's no need for me to manually enter my current street address when searching for a restaurant. It just knows where I am. There's no need for me to tell the maps app that I'm facing a certain direction and to change its perspective. It just knows the direction I'm facing. These apps use Core Location to retrieve my

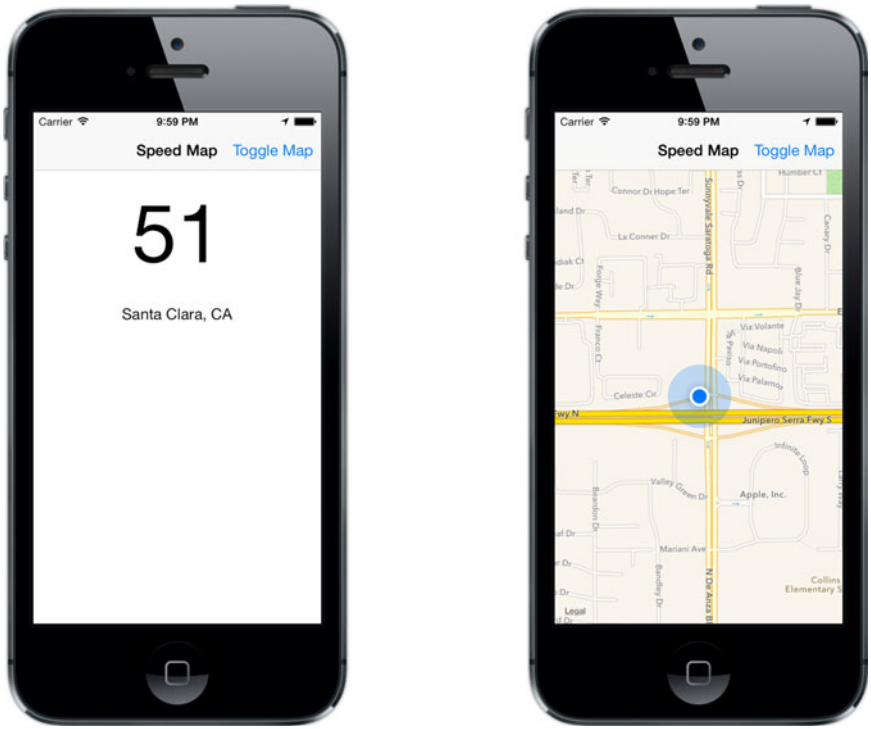


Figure 10.1 The application we'll be building together will show your current location and the speed you're traveling, and it will even track your location in real time within a map.

current location as well as my heading. You'll be learning how to use Core Location and MapKit in this chapter by creating an application called Speed Map, as shown in figure 10.1.

This app will show you your current location in user-friendly, readable text. It also has a map that will show and track your location in real time and rotate depending on which direction you're facing. As you go along you'll also learn about the many different things you can do with these two frameworks.

10.1 Introduction to the Core Location framework

Core Location is a framework that's included in the iOS SDK that can be used to determine location and heading with a device. Location is found using GPS or assisted GPS on the device. Assisted GPS helps retrieve your location quicker by using the cellular or Wi-Fi network to triangulate your location. Heading is determined by the phone's compass. You also get a geocoder, which can be used to retrieve placemarks and names of the city and state when given a pair of coordinates. You'll learn how to use all of these in this section.

10.1.1 Representing a location with CLLocation

The `CLLocation` class is one that you'll become very familiar with because it's used to represent location data. A single instance contains geographical coordinates, altitude, values indicating the accuracy of these measurements, and the time at which they were captured. There's also information regarding the speed and heading of the device at the time of measurement.

You can initialize a new `CLLocation` instance by using the `initWithLatitude:longitude` method. You'll use the coordinates for Mountain View, California, for this example. You can specify the latitude 37.3861 and the longitude 122.0828 as follows:

```
CLLocation *location = [[CLLocation alloc] initWithLatitude:37.3861
➥ longitude:122.0828]];
```

The `location` instance variable that you've created can let you know the latitude and longitude you've set by retrieving the `coordinate` property. The `coordinate` property returns a `CLLocationCoordinate2D` structured type, which contains two fields, `latitude` and `longitude`. Following is how you'd check the latitude and longitude of the `CLLocation` that you created:

```
location.coordinate.latitude      // 37.3861
location.coordinate.longitude     // 122.0828
```

If you were to retrieve a `CLLocation` from your device that represented your current location, you'd have properties that you could check to determine the accuracy. Accuracy is determined horizontally and vertically using the `horizontalAccuracy` and `verticalAccuracy` properties. These values represent the level of uncertainty, measured in meters. For example, if you had a horizontal accuracy of 25, that would mean that the location retrieved was uncertain up to a 25-meter radius. The lower the accuracy number, the better. Take a look at figure 10.2 for a real-world example using the Google Maps application.

What you're seeing is the Google Maps visual representation of a `CLLocation` instance that represents our current location. The left side of the figure shows a bad accuracy example. This was captured right when the application was opened. It retrieved a `CLLocation` but the accuracy wasn't great yet. After a few seconds, the accuracy became better, which is what's shown on the right side of the figure.

Since you created the `CLLocation` instance without retrieving it from the device, you won't have a horizontal or vertical accuracy measurement. You still could set the accuracy properties by doing the following, using any value you like:

```
location.horizontalAccuracy = 25.0f;
location.verticalAccuracy = 50.0f;
```

If you retrieved it from a device, you'd also be able to retrieve the altitude as well as the speed at which you were traveling. You can also set these in the same way as shown previously.

There's also a way to initialize a new `CLLocation` with altitude, latitude, longitude, accuracy, as well as timestamp. You can do this by using the `initWithCoordinate:`

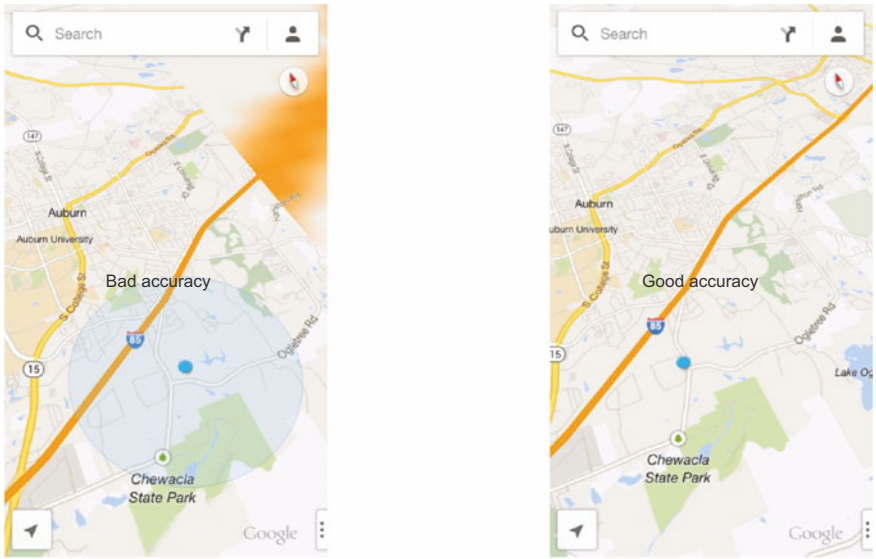


Figure 10.2 Bad accuracy on the left and good accuracy on the right. The larger blue circle equals the distance of uncertainty.

`altitude:horizontalAccuracy:verticalAccuracy:timestamp:` initializer. Unlike the initializer that you previously used, this one requires you to pass in the coordinates as a single parameter of type `CLLocationCoordinate2D`. Let's create one by using the same values you used earlier:

```
CLLocationCoordinate2D coordinates = CLLocationCoordinate2DMake(37.3861,
➥ 122.0828);
```

You can then pass in coordinates along with any altitude, accuracy, and timestamp, as shown here:

```
CLLocation *location = [[CLLocation alloc] initWithCoordinate:coordinates
                                                     altitude:1000.0f
                                           horizontalAccuracy:25.0f
                                             verticalAccuracy:50.0f
                                                    timestamp:[NSDate
➥ date]];
```

Because you already know about the `CLLocation` object, you can learn how to retrieve this using `CLLocationManager`.

10.1.2 The location manager

To be able to retrieve the current location of a device you'll have to use `CLLocationManager`. `CLLocationManager` defines the interface for determining how you receive location updates in your app. You can use a `CLLocationManager` instance to start and stop monitoring for changes in location, heading, and when a user enters or leaves a

distinct region or location, and you can also defer location updates when your app is running in the background.

First, you should check to see if it's possible to retrieve what you want. A variety of class methods will help you do just that. Take a look at table 10.1, which lists each method and gives a brief description of what it does.

Table 10.1 `CLLocationManager` class methods to check for availability and permissions

Type	Description
`authorizationStatus`	Has the user given permission to use their current location?
`locationServicesEnabled`	Are location services enabled on the device?
`deferredLocationUpdatesAvailable`	Does the device support deferred location updates?
`significantLocationChange-MonitoringAvailable`	Does the device support significant location change monitoring?
`headingAvailable`	Can the device retrieve heading-related events?
`regionMonitoringAvailable`	Does the device support region monitoring?

All of these methods return a `BOOL`, which you can use to help you decide what you should do in your application.

There are also a few properties that you can set on the location manager. These properties will determine what you retrieve, the frequency, and the accuracy. New readings from the location manager are delivered to you using a delegate, which is specified with the `delegate` property. We'll jump into this shortly, but it's important to know that location isn't requested synchronously.

First, there's the `distanceFilter` property, which determines how far the device must move horizontally in meters before an update event is generated. The distance is measured relative to the last location update that was generated.

Next is `desiredAccuracy`, which lets you specify how accurate the readings will be. There are times when you need something extremely accurate; for example, you could be building a car navigation app, which needs precise location data. But if you were building a weather app that needed only the city you're in, you wouldn't require the same kind of accuracy. You can specify six accuracy constants, which are ordered from most to least accurate: `kCLLocationAccuracyBestForNavigation`, `kCLLocationAccuracyBest`, `kCCLocationAccuracyNearestTenMeters`, `kCCLocation-AccuracyHundredMeters`, `kCCLocationAccuracyKilometer`, and `kCCLocationAccuracy-ThreeKilometers`.

Just like with the `distanceFilter` property, you can filter the change in heading for heading updates using the `headingFilter` property. The heading filter is specified in degrees rather than meters. The heading reading also needs to know the orientation of the device. This is specified by using the `headingOrientation` property.

Be mindful of battery consumption

There are a few things to keep in mind when configuring the location manager. First, some readings, such as heading, require that a hardware compass be available on the device that your app is running on. Also, the location manager can drain a phone's battery relatively quickly. This can happen if you set an accuracy requirement that's too high. For example, if you set the required accuracy reading too high, the location manager would have to use the GPS to retrieve a new location every time you move this distance. If you had this on while you were in a fast-moving vehicle, there's a high chance that it would quickly drain your battery. It's very important to take these things into consideration when dealing with location.

We mentioned the `delegate` property earlier. When using the location manager, you must specify a delegate that conforms to the `CLLocationManagerDelegate` protocol to be able to retrieve updates. Whenever a location or heading update is retrieved, this result is passed to the delegate specified for your `CLLocationManager` instance. There are two methods you have to add if you wish to know when the location manager has *resumed* or has *paused* receiving updates, as shown here:

```
- (void)locationManagerDidResumeLocationUpdates:(CLLocationManager
➥ *)manager
{
    // Location manager has resumed location updates
}

- (void)locationManagerDidPauseLocationUpdates:(CLLocationManager *)manager
{
    // Location manager has paused location updates
}
```

These two methods allow you to act according to the location manager's update retrieval state. For example, if you were building a GPS navigation app, you would have to show an alert if you weren't receiving location updates anymore. Without an alert, the users of your navigation app would be wondering why their location wasn't updating on the screen.

The really important information that you'll be using is passed into the method `locationManager:didUpdateLocations:`. This method is called whenever the location manager has new current location readings. Even though it's optional, consider it required if you're looking to retrieve the current location within your app.

The first parameter passed into this method is the instance of the location manager that's being used to listen for location changes. The second parameter is an array of `CLLocation` objects. You're guaranteed to have at least one object in this array. Each `CLLocation` object in the area is ordered by the time at which it occurred. This means that the newest location would be at the end of the array. For instance, if you always wanted to retrieve the latest location information from this method, you could do the following:

```
- (void)locationManager:(CLLocationManager *)manager
➥ didUpdateLocations:(NSArray *)locations
{
    CLLocation *newestLocation = [locations lastObject];
}
```

There are also delegate methods you can implement for getting the latest heading readings as well. You'll be using the `locationManager:didUpdateLocations:` method as well as other delegate methods when you create your application. When you're ready to retrieve location and heading updates, you can do so by calling either `startUpdatingLocation` or `startUpdatingHeading` on your location manager instance. If you want to stop these updates, you can do so by calling `stopUpdatingLocation` and `stopUpdatingHeading`.

How about you start setting up your application in Xcode before we go further?

10.1.3 Setting up Speed Map in Xcode

Together we'll be building an app called Speed Map that contains a single view. This view will show your current speed and current location in plain text and a map that will track and show the current location in real time. You'll be using the Simulator to simulate location scenarios for this application, but to really enjoy it, we highly suggest using it on a real device if you have a paid developer license.

Let's first create the project and then hook in the Core Location framework. Open Xcode and create a new, single-view application called Speed Map, as shown in figure 10.3.

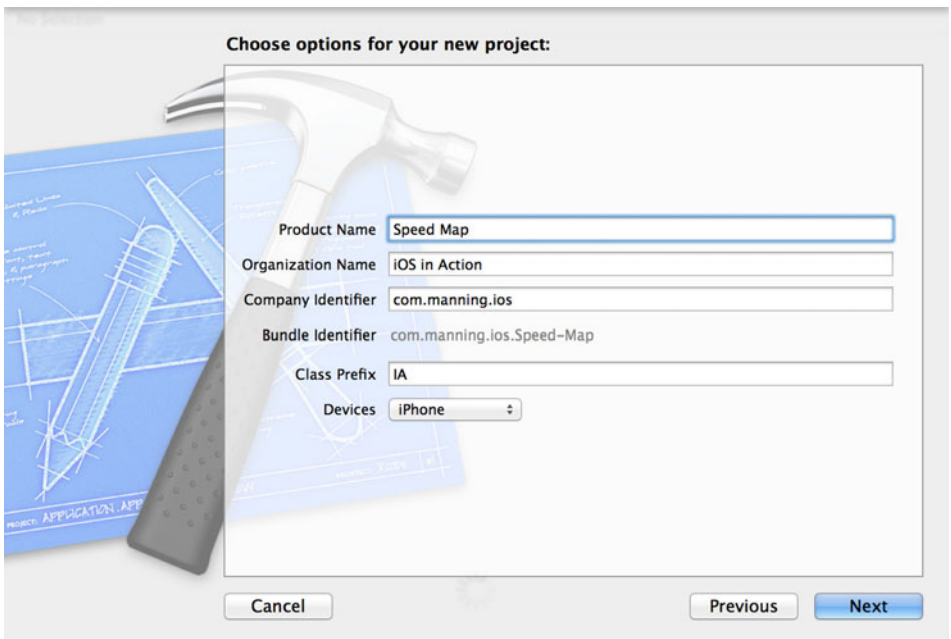


Figure 10.3 Creating Speed Map as a single-view application in Xcode

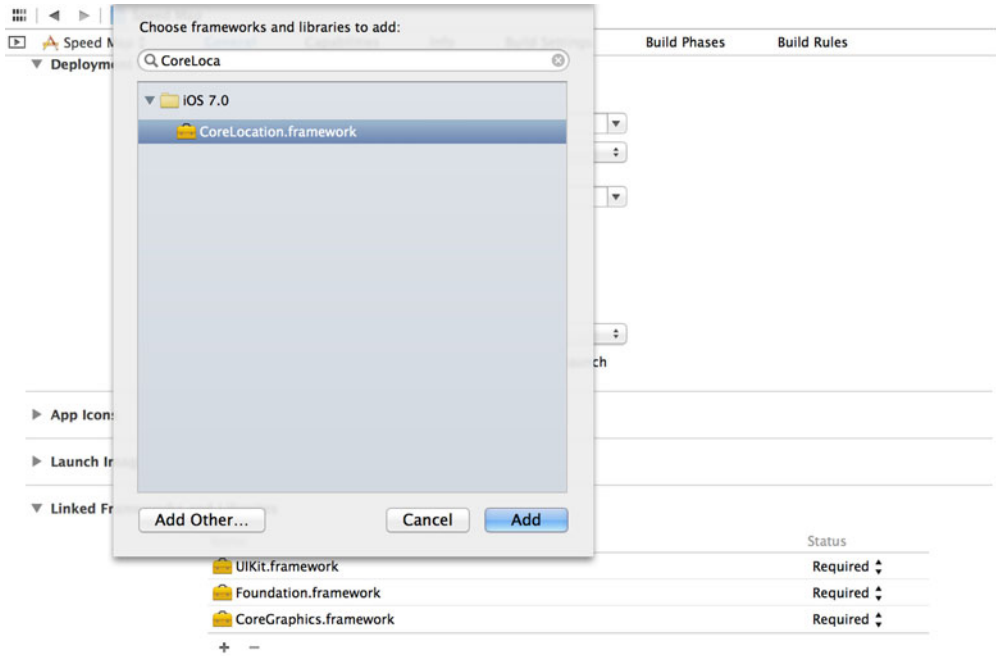


Figure 10.4 Add the Core Location framework in the Link Binary with Libraries section in the project settings.

Once the project's been created, go into the Settings tab and expand the Link Binary with Libraries section. Here you want to add the Core Location framework by looking for and adding CoreLocation.framework, as shown in figure 10.4.

Once you've added the Core Location framework, you should start preparing the views in your storyboard. Open Main.storyboard in the project navigator. You'll put together the first of two screens that you'll have in your application. This first screen will show your current speed as well as your location.

First you'll embed this view in a navigation controller by selecting the view and choosing Editor > Embed In > Navigation Controller from the application menu bar. Next select the navigation bar and set its title to Speed Map within the attributes inspector, as shown in figure 10.5.

You'll now add a label that will be used to show the speed you're traveling in miles per hour. This will be the main part of this view, which means that you should make it stand out. Go to the Object Library, grab a `UILabel`, and drag it into your view. Position it so that it appears at the top of your view. Set the font for this label to Helvetica Light with a font size of 90 and center aligned. Next, set the text in the label to 0 because you'll be starting at 0 MPH. Also, change the view sizing so that the width is 320 and the height is 140. This is shown in figure 10.6.

You can now create a new outlet for this label by first opening the assistant editor. Ensure that IAViewController.h is open within the assistant editor window. Once it is,

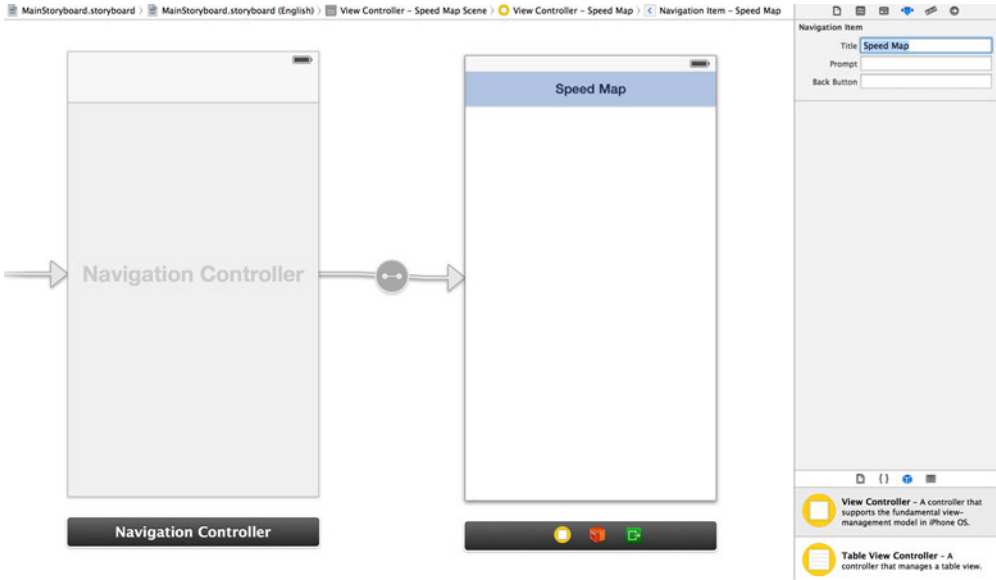


Figure 10.5 Set the title of the navigation bar to Speed Map within the attributes inspector.

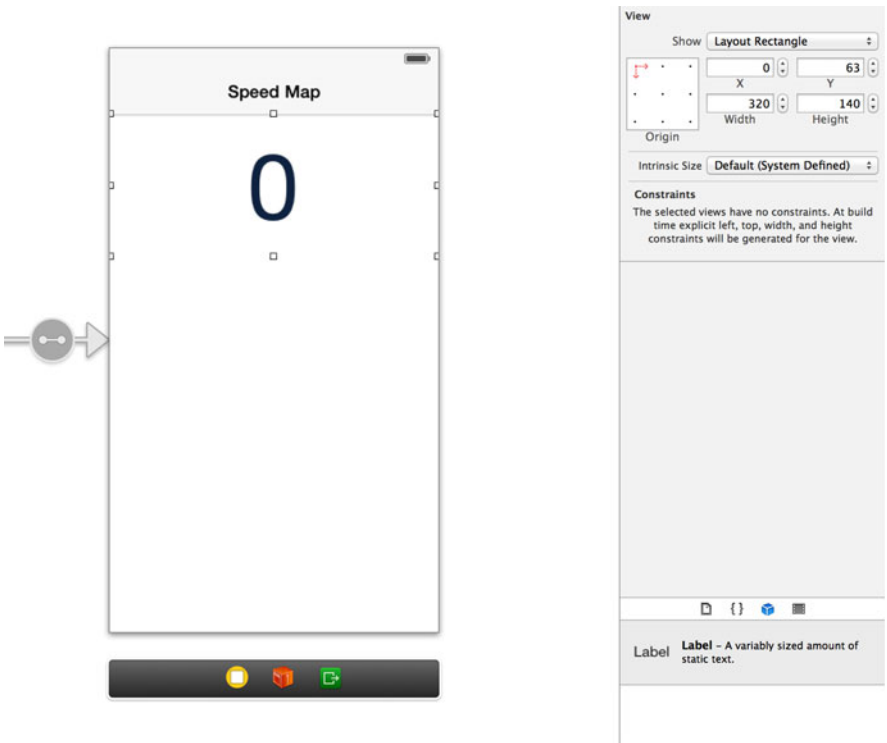


Figure 10.6 Add a new `UILabel` and position it toward the top of your view. Set its font, alignment, and constraints so that it appears correctly.

Figure 10.7 Create a new outlet within IAViewController.h for your new label named `speedLabel`.

drag a connection to create a new outlet for this label called `speedLabel`, as shown in figure 10.7.

Let's focus once again on your view within the storyboard. You'll add one more label that you'll use to display your current location. Place it directly underneath the label you're using to display your speed, and make sure the text is center aligned. Then change its Width attribute to 320, as shown in figure 10.8.

Next, go to the attributes inspector and set its text to be blank. Now open the assistant editor and create a connection for this label with an outlet called `locationLabel`. We can stop here and get right into adding some functionality to this app. You'll learn how to set up a location manager and retrieve your location, speed, and heading information.

10.2 Retrieving location, heading, and speed

Earlier we went over the location manager and how it's used to retrieve information from the GPS and compass. You'll be using the location manager in this section to add the functionality you need in your Speed Map application. First, you'll use it to retrieve your current location.

10.2.1 Retrieving your current location with the location manager

You know that you need to use a location manager and set up delegate methods to listen for location update events. You'll be putting this into action in this section by adding this to `IAViewController`. Open IAViewController.h and add the following `import` line at the top of the controller to give you access to the Core Location framework in this class:

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

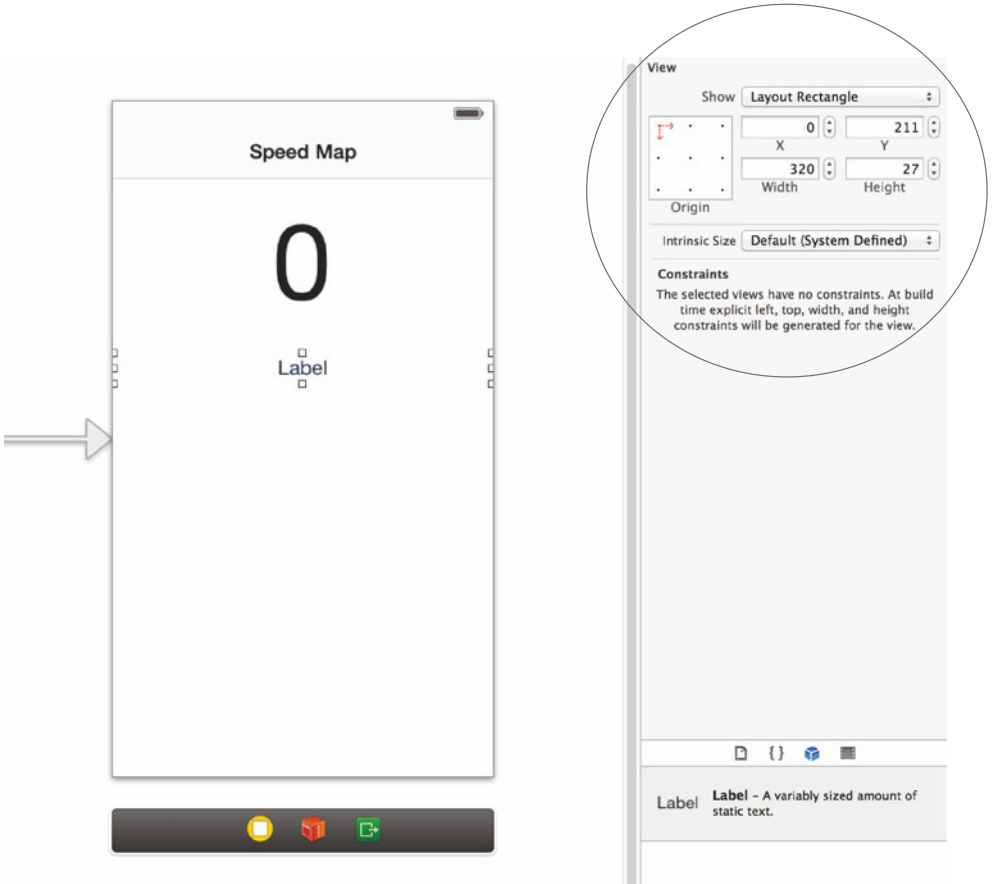


Figure 10.8 Add another label underneath the speed label that will be used to represent your current location.

Next, you need a property to represent the location manager you'll be using. Add the following property:

```
@property(nonatomic, strong) CLLocationManager *locationManager;
```

After this you'll have to specify that this class conforms to the `CLLocationManagerDelegate` protocol. Update the following line to reflect this:

```
@interface IAViewController : UIViewController <CLLocationManagerDelegate>
```

You can now jump into IAViewController.m to work on the implementation of your controller. First, you'll define a constant that you'll use to help you calculate the number of meters that are in a mile. Right above the `@implementation` declaration in your controller, add the following:

```
#define kMetersPerSecondToMilesPerHour 2.2369362920544
```

Next, because you specified that you conform to the `CLLocationManagerDelegate` protocol, you have to add the needed methods. Add the code shown in the following listing to the controller.

Listing 10.1 `CLLocationManagerDelegate` required methods

```
- (void)locationManagerDidPauseLocationUpdates:(CLLocationManager *)manager
{
    self.speedLabel.text = @"";
    self.locationLabel.text = @"Location unavailable";
}

- (void)locationManagerDidResumeLocationUpdates:(CLLocationManager *)manager
{
}
```

When the location manager is paused, you'll update your labels to reflect its current state. Are you wondering why you're not doing anything when location updates resume? The code you're about to add will automatically set those labels when you get new location readings. Because the location manager has resumed, it should send you new readings.

Toward the bottom of the `viewDidLoad` method you should set your `locationManager` property, specify a distance filter and accuracy, and set your controller as the delegate. You can also make a call to start updating the location by calling `startUpdatingLocation`. The updated `viewDidLoad` method is shown here:

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

    self.locationManager = [[CLLocationManager alloc] init];
    self.locationManager.desiredAccuracy = kCLLocationAccuracyBest;
    self.locationManager.distanceFilter = 10.0f;
    self.locationManager.delegate = self;

    [self.locationManager startUpdatingLocation];
}
```

Create location manager instance. ❶
❷ **Set desired accuracy.**
❸ **Set distance filter.**
❹ **Assign controller as delegate.**
Start updating location. ❺

First, you set the `locationManager` property to be a new instance of `CLLocationManager` ❶. You set your desired accuracy to best ❷, mainly because of the speed readings that you'll be displaying. You also set a distance filter to be 10 meters ❸. This means that someone must travel 10 meters for an event update to occur. You then set your controller as the delegate ❹ and make a call to start updating the location ❺.

Even though you're calling `startUpdatingLocation`, nowhere in the controller are you actually retrieving any location events. You should implement another method from the `CLLocationManagerDelegate` protocol in order to update your interface with your speed and location. This method is `locationManager:didUpdateLocations:`, which will give you an array of `CLLocation` objects. Add the code shown in the next listing to `IAViewController`.

Listing 10.2 Updating the interface when you retrieve new location information

```
- (void)locationManager:(CLLocationManager *)manager
didUpdateLocations:(NSArray *)locations
{
    CLLocation *location = [locations lastObject];
    NSString *locationString = [NSString stringWithFormat:@"%f, %f",
➥ location.coordinate.latitude, location.coordinate.longitude];
    NSString *speedString = [NSString stringWithFormat:@"%d",
➥ [@(location.speed * kMetersPerSecondToMilesPerHour)
➥ intValue]];

    self.locationLabel.text = locationString;
    self.speedLabel.text = speedString;
}
```

1. Grab most recent `CLLocation`.
2. Create string with latitude and longitude.
3. Create string with speed in miles.
4. Set `locationLabel` text to `locationString`.
5. Set `speedLabel` text to `speedString`.

In this method you get the last location object in the locations parameter passed in by the location manager ❶. This is because the last object in the array is the newest location reading. You then create a new string using the latitude and longitude ❷ and another one for the speed ❸. When creating the string to represent the speed, you multiply the speed reading by your constant, `kMetersPerSecondToMilesPerHour`. Lastly, you're setting the `locationLabel` text property to `locationString` ❹ and setting the `speedLabel` text property to `speedString` ❺.

How about you put what you've done to the test and run the application? If all is well, you should first be presented with an alert that asks you for permission to use your current location, as shown in figure 10.9.

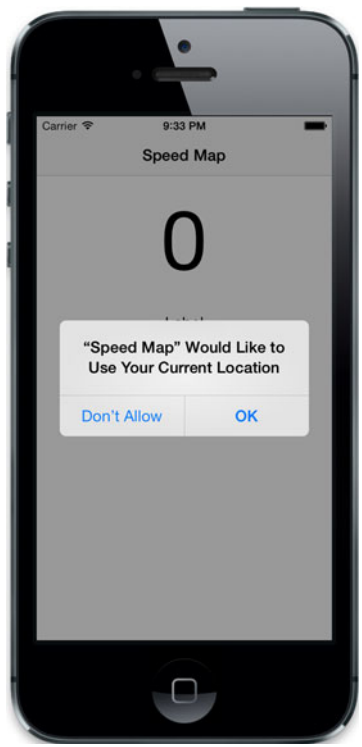


Figure 10.9 The first time you run your application, you'll be asked for permission to use your current location.

Once you tap OK, you'll see that your speed is 0 and that you don't have a location displaying. This is because the Simulator needs to be told exactly where you are for location readings to work. You can do this within Xcode in the debug area shown on the bottom of the window while the application is running. You can open this by going to View > Debug Area > Show Debug Area (Command+Shift+Y). You can choose one of the preset locations by clicking the location arrow, as shown in figure 10.10.

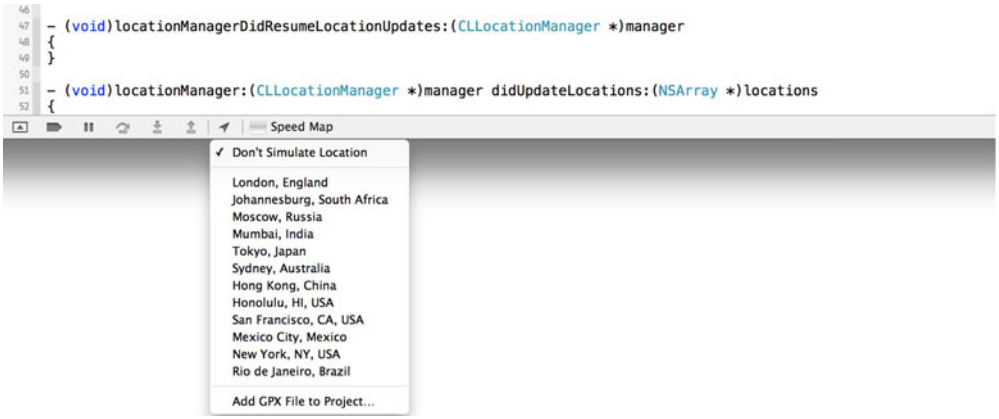


Figure 10.10 When running your app in the Simulator, you can choose a preset location within Xcode while the application is running.

How about you choose Honolulu, HI, USA, because I'm sure you wouldn't mind being there right now. If you're already there and are feeling generous, I wouldn't mind a ticket to come and visit ☺. Anyway, once the location's been set, look back at the Simulator to see the latitude and longitude for sunny Honolulu, as shown in figure 10.11.

One very special feature of the Simulator is the ability to simulate location scenarios. While you have the Simulator open, you can access the Debug > Location menu in the application menu bar. Within this menu you can choose Freeway Drive, which will simulate location updates during a drive along the freeway. Once you've set this, take a look at the Simulator. You'll slowly see the speed increase and your location will start changing, as shown in figure 10.12.

You'll now see how to change the latitude and longitude to something more user friendly by using the built-in geocoder.

Figure 10.11 We set our location to Honolulu, HI, USA, by using a predefined location within Xcode.

10.2.2 Geocoding a location

Displaying the latitude and longitude is beneficial for debugging, plotting points on a map, and other various non-user-friendly scenarios. To turn these numbers into something that non-developers can understand, you can use Core Location's geocoder.

The `CLGeocoder` class represents the built-in geocoder that you can use to reverse-geocode and forward-geocode locations.

Forward geocoding is useful when you have an address, such as Auburn, AL, and you want to find the latitude and longitude. Reverse geocoding is used when you have a latitude and longitude, such as 32.6097, 85.4808, and want to turn that into a readable city, state, and country. You'll need to reverse geocode the latitude and longitude reading that you get from the location manager. You'll need to use the `reverseGeocodeLocation:completionHandler:` method to do just that. This method accepts a `CLLocation` and a block that will be executed once the reverse geocode is finished.

The block is executed once the reverse geocoding request is completed. It will always get called, even when there's an error. Also, geocoding is asynchronous but will run on the main thread. It's important to run only one geocoding operation at a time because otherwise it will delay the main thread and make the user interface appear locked up or frozen. Take a look at the definition of the completion handler:

Figure 10.12 The location is now constantly changing with the speed increasing after setting the Simulator to simulate a freeway drive.

```
typedef void (^CLGeocodeCompletionHandler)
     (NSArray *placemark, NSError
➥ *error);
```

If there's no error, you'll be passed back an array of placemarks. Each placemark is a `CLPlacemark` object, which will contain detailed information about that location. The information you can obtain includes the city, state, country, landmark locations, postal code, whether it's inland or near an ocean, and the like. For your purposes you'll be looking for the city and state. The city is retrieved by calling the `subAdministrativeArea` property, and the state can be retrieved by using the `administrativeArea` property.

To ensure that you're calling the reverse geocoding function only once, you can use the `isGeocoding` property on the `CLGeocoder` instance, which returns a `BOOL` value. You can make this check before you call the `reverseGeocodeLocation:completionHandler:` method.

First, go into IAViewController.h and create a new property for the geocoder you'll be using throughout the controller:

```
@property (strong, nonatomic) CLGeocoder *geocoder;
```

Open IAViewController.m and replace the `locationManager:didUplodateLocations:` method with the code shown in the following listing.

Listing 10.3 Reverse geocoding to retrieve the city and state from a `CLLocation`

```
- (void)locationManager:(CLLocationManager *)manager
➥ didUpdateLocations:(NSArray *)locations
{
    CLLocation *location = [locations lastObject];
    NSString *speedString = [NSString stringWithFormat:@"%d",
                             [@(MAX(location.speed, 0)) intValue]];

    self.speedLabel.text = speedString;

    if (![self.geocoder isGeocoding])
    {
        [self.geocoder reverseGeocodeLocation:location
                            completionHandler:^(NSArray *placemarks,
➥ NSError *error)
         {
             if (placemarks && [placemarks count] > 0 && error == nil)
             {
                 CLPlacemark *placemark = [placemarks lastObject];
                 NSString *locationString = [NSString
➥ stringWithFormat:@"%@, %@",
➥ placemark.subAdministrativeArea, placemark.administrativeArea];

                 self.locationLabel.text = locationString;
             }
         }];

    }
}
```

1. **Continue only if not currently geocoding.**
2. **Call the reverse geocoding method.**
3. **Check to see if placemarks were returned and no errors.**
4. **Grab the last placemark.**
5. **Format a string to represent "City, State".**
6. **Set the `locationLabel` text to the new string.**

You're making a few changes to this method. The speed is being calculated and displayed in the same way but the way you're displaying the location has changed. You now first check to see if you're currently geocoding. If you're not geocoding, you proceed ❶ and call the reverse geocoding method ❷. Within the completion block you check to make sure that you have placemarks and there are no errors ❸. Then you retrieve the latest placemark ❹ and create a new formatted string that contains the city and state ❺ and then set that to the `text` property of `locationLabel` ❻.

You can easily test this by using the predefined locations within Xcode when you run the application. Run the application and set the location to Honolulu, HI. You should see the location label set to Honolulu, Hawaii, as shown in figure 10.13.

If you have a developer license, you can put this on your device and leave it running while you're in a moving vehicle. It should be fun to monitor your speed and your current location as you travel. You can expand this if you want to by adding the heading reading from the compass underneath the current location. You can do this by starting heading updates in the location manager and implementing the appropriate delegate method. For now, though, we'll move on to an overview of the MapKit framework and add a map to your application.

10.3 Introduction to the MapKit framework

The MapKit framework is used to embed and interact with maps within an application. It's a fairly large framework that allows you to do many things. You can include illustrated maps, satellite maps, hybrid maps, annotations, overlays, routes, and much more. To put it into perspective, most of what MapKit provides can allow you to create something close to Apple's very own Maps application.

10.3.1 Using the map view to display a map

One of the biggest pieces of MapKit is the `MKMapView`. You can use this view to embed a map anywhere in your application, and it provides a decent amount of functionality out of the box. For instance, the ability to show your current location is already baked into this view. The main view that you interact with within the Maps application is the `MKMapView`, as shown in figure 10.14.

You can use three different map types: a default map, which provides an illustrated representation; a satellite map, which uses real aerial images to represent the map;

Figure 10.13 By using the predefined Honolulu, HI location in Xcode, you're reverse geocoding the latitude and longitude to display Honolulu, Hawaii in the location label.

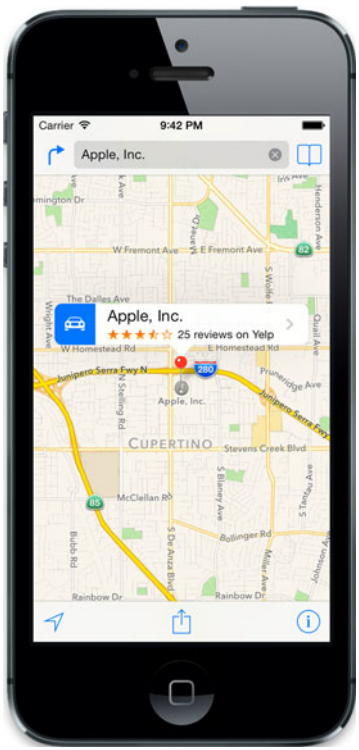


Figure 10.14 The `MKMapView` used within Apple's Maps application

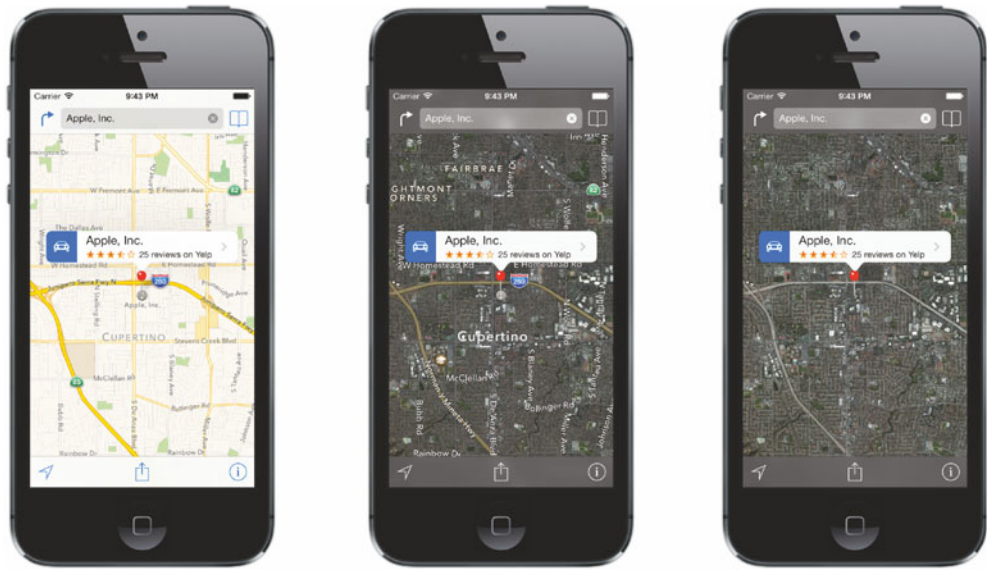


Figure 10.15 The three different map types that you get out of the box with an `MKMapView`

and the hybrid map, which lays illustrated roads and points of interest over the satellite imagery used in a satellite map. You can see these three different map types side by side in figure 10.15.

You can set the type programmatically on an instance of `MKMapView` by specifying the `mapType` property. The three constants that represent these types are `MKMapTypeStandard`, `MKMapTypeSatellite`, and `MKMapTypeHybrid`. Within the interface editor you can just choose among Map, Satellite, and Hybrid within the attributes inspector.

When using the `MKMapView`, you should specify an initial region for the map to display. This is done by setting the region with a center point, defined by latitude and longitude, and a span, which is the distance from the center. The span is used to determine how much the map should zoom out from the center point for it to display within the view. For example, say you wanted to set an instance of `MKMapView`'s region to 37.3861, 122.0828 with a span of 1 kilometer. You could do so as follows:

```
CLLocationCoordinate2D coordinate = CLLocationCoordinate2DMake(37.3861,
➥ 122.0828);
MKCoordinateRegion region = MKCoordinateRegionMakeWithDistance(coordinate,
➥ 1000, 1000);
[mapView setRegion:region];
```

You can use the `CLLocationCoordinate2DMake()` function to create a `CLLocationCoordinate2D` for a specific latitude and longitude. Then you can use the `MKCoordinateRegionMakeWithDistance()` function to create an `MKCoordinateRegion` using the coordinate and passing in the span.

If you just wanted to let the map zoom in on your location, you could set a tracking mode without having to worry about setting an initial region. You can set this mode with the `setUserTrackingMode:animated:` method. There are three different tracking modes you can specify: `MKUserTrackingModeNone`, `MKUserTrackingModeFollow`, and `MKUserTrackingModeFollowWithHeading`. By default, no tracking mode is set. The tracking mode `MKUserTrackingModeFollow` will follow and update the map based on the user's current location. The last option, `MKUserTrackingModeFollowWithHeading` will also update the map based on the user's location but will orient it depending on the user's heading. You'll be using this shortly within your Speed Map application.

10.3.2 Retrieving user location using MapKit

You could retrieve a user's location using MapKit instead of having to use Core Location's location manager. An `MKMapView` can do this by talking to Core Location for you. First, you'll need to set the `showsUserLocation` property to `YES` on the map you're using. You'll then need to implement the `MKMapViewDelegate` protocol and set the `delegate` property on the map view instance you're using.

The `MKMapViewDelegate` protocol has a single required method that's used to let you know when the tracking mode of a user's location has changed. Whether you use this or not, it must be implemented. This method is shown here:

```
- (void)mapView:(MKMapView *)mapView
➥ didChangeUserTrackingMode:(MKUserTrackingMode)mode
➥ animated:(BOOL)animated
{
    // Handle tracking mode change
}
```

The method that will inform you of a user's location updates is the `mapView:didUpdateUserLocation:` optional method. The location object that's returned is an `MKUserLocation`. You can use its `location` property to retrieve the `CLLocation` representation, as follows:

```
- (void)mapView:(MKMapView *)mapView didUpdateUserLocation:
➥ (MKUserLocation *)userLocation
{
    CLLocation *location = userLocation.location;
}
```

Next, you'll see how you can add annotations to a map.

10.3.3 Using annotations in a map

An annotation is a view that's used to mark a single coordinate on a map. You can use annotations to mark specific points of interest. An example of an annotation is shown in figure 10.16 to mark the location of Apple, Inc.

To create an annotation you can use the `MKPointAnnotation` class. You'll have to specify a coordinate, title, and, optionally, a subtitle. If you had an instance variable of a map named `mapView`, you could add an annotation using the `addAnnotation:`

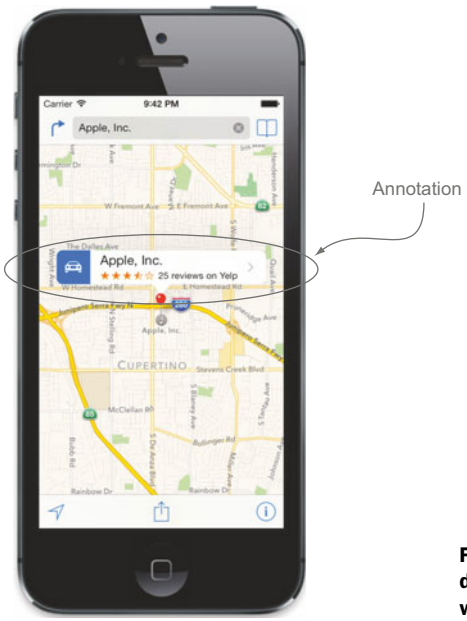


Figure 10.16 An annotation used to display the location of Apple, Inc. within the Maps application

method. Here you can see how you'd add an annotation for Apple's headquarters in Cupertino, California:

```
MKPointAnnotation *annotation = [[MKPointAnnotation alloc] init];
annotation.coordinate = CLLocationCoordinate2DMake(37.332, -122.031);
annotation.title = @"Apple, Inc.";
annotation.subtitle = @"1 Infinite Loop, Cuptertino, CA";

[mapView addAnnotation:annotation];
```

If you wanted to create your own custom annotation, you'd have to create a new class that implements the `MKAnnotation` protocol. Suppose you wanted to create one named `IAAnnotation`. You could use the code shown in the following listing.

Listing 10.4 `IAAnnotation` custom class that implements the `MKAnnotation` protocol

```
@interface IAAnnotation : NSObject<MKAnnotation>

@property (nonatomic, readonly) CLLocationCoordinate2D coordinate;
@property (nonatomic, readonly) NSString *title;
@property (nonatomic, readonly) NSString *subtitle;

@end
```

The object that implements the `MKAnnotation` protocol must have at least a coordinate, a title, and a subtitle. When you decide to use this in your application, you can initialize it, set these properties, and add it to your map. You can also create custom views by subclassing the `MKAnnotationView` class and adding them to the map view.

Now let's add a map to your Speed Map application.

10.3.4 Adding a map to your application

You're going to add a map that will display and track a user's current location with a heading. The first thing you should do is add the MapKit framework to your application. Go to the project's Build Settings tab and add MapKit.framework to the Linked Frameworks and Libraries section, as shown in figure 10.17.

Let's now jump into Main.storyboard and look for Map View within the Object Library on the bottom right of the window. Drag it into your view so that it completely covers what you currently have in place. Also ensure that its width and height fill up the entire view. Next, open the attributes inspector and ensure that Shows User Location is checked. Also mark this view as hidden. This is shown in figure 10.18.

Open the assistant editor and create a new outlet for your map called `mapView` in IAViewController.h. You'll see an error that says "Unknown type name 'MKMapView'." You can fix this by importing the MapKit framework using the following `import` statement:

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

Now jump back into the storyboard and look for a `UIBarButtonItem` in the Object Library. You're going to use this to add a button to the navigation bar to toggle the

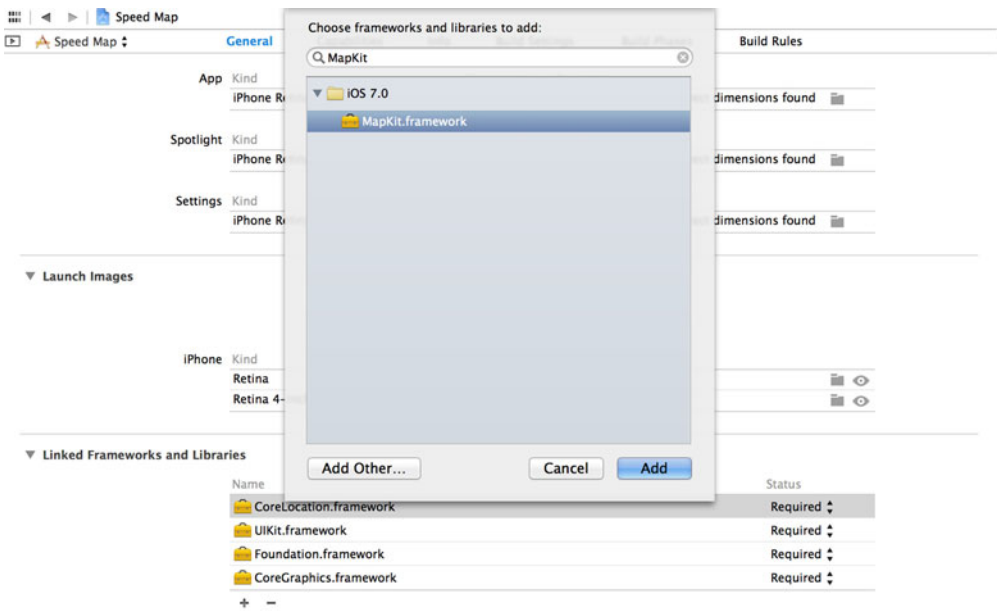


Figure 10.17 Add MapKit.framework to Speed Map within the Link Binary With Libraries section in the Build Settings.

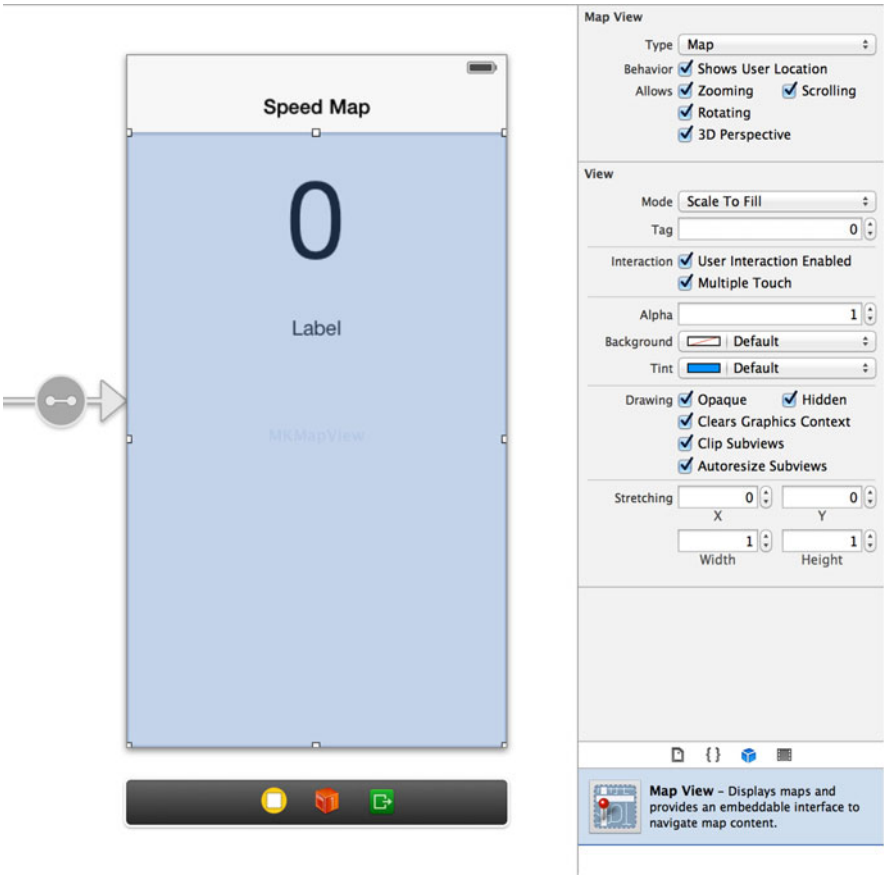


Figure 10.18 Add the map view on top of your view so that it fills up the entire width and height. Ensure that it shows the user location and is hidden.

map view. After you've found the bar button item, drag it to the top right of the navigation bar and set its title to Toggle Map, as shown in figure 10.19.

Once again, open the assistant editor and drag a connection to IAViewController.h to create a new `IBAction` named `toggleMap`, as shown in figure 10.20.

Now go to the implementation of `IAViewController` by opening IAViewController.m from the project navigator. Add the following code to the bottom of the `viewDidLoad` method to set the user tracking mode on the map:

```
[self.mapView setUserTrackingMode:MKUserTrackingModeFollowWithHeading];
```

Finally, add code to the `toggleMap:` action you created. Within the method definition you'll add code to hide or show the map depending on its previous state. The method in its entirety is shown in the following listing.

.

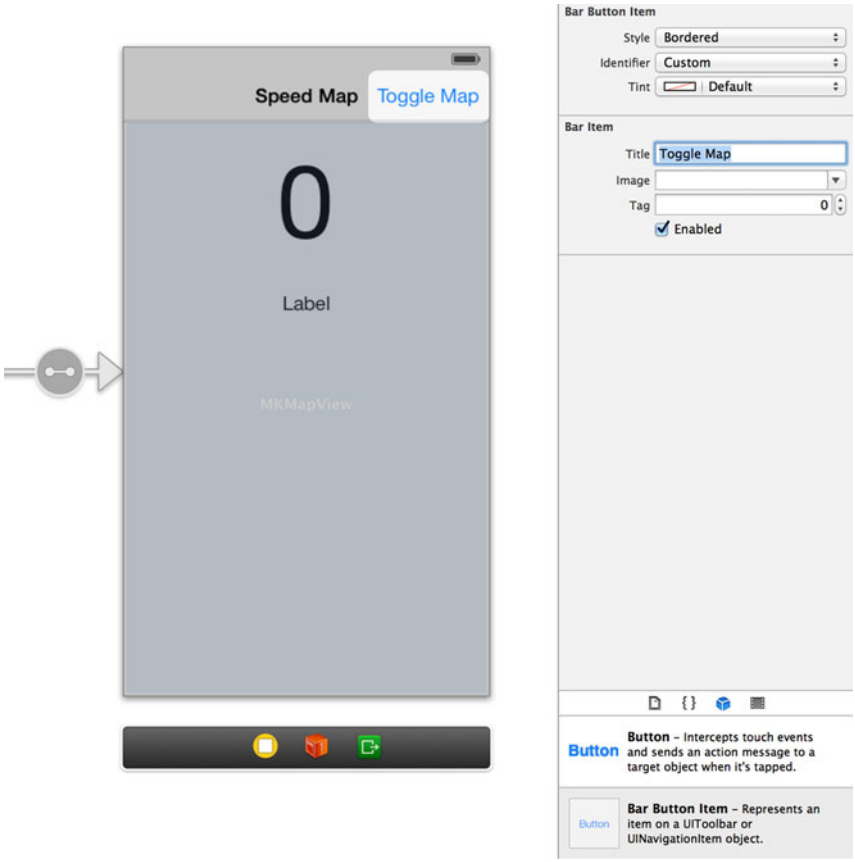


Figure 10.19 Add a bar button item to the navigation bar and set its title to Toggle Map.

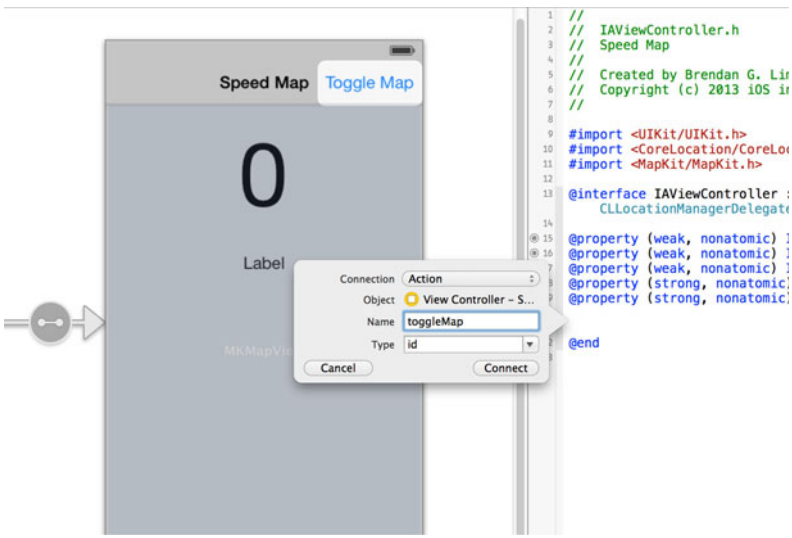


Figure 10.20 Create a new action for the bar button item called `toggleMap` within IAViewController.h.

Listing 10.5 `toggleMap:` action to hide or show the map

```
- (IBAction)toggleMap:(id)sender
{
    [self.mapView setHidden:![self.mapView isHidden]];
}
```

You can now run the application and give it a try. Run the app in the Simulator and set the debug location to Freeway Ride in Debug > Location within the application menu. Your speed should start increasing and your location should change. Tap the Toggle Map button, and you'll see your location being shown and tracked on the map. If you put this on a real device, the map will not only update with your current location but will also rotate depending on your current heading.

10.4 Summary

In this chapter you learned about two frameworks, Core Location and MapKit, that enable you to use location to provide a whole new level of interaction in your apps. You learned how to retrieve location updates and reverse geocode location information using the Core Location framework. You also saw how easy it is to map a user's location using the MapKit framework. Finally, you created an application that tied this all together to show speed and location information as well as plot a user's location on a map. These are the key points to remember:

- Location information is wrapped in a `CLLocation` instance.
- The `CLLocationManager` lets you know of location updates depending on the filters and settings you have in place.
- It's important to be mindful of battery drain when working with location monitoring on the device.
- The iOS Simulator has built-in tools to help you debug real-life location scenarios.
- You can reverse geocode location information using the `CLGeocoder` class.
- The MapKit framework allows you to very quickly add full map integration into your apps.
- The `MKMapView` allows you to view maps in three different visual modes.
- Annotations allow you to add detailed contextual information at a given point on a map.

11 Persistence and object management with Core Data

This chapter covers

- Introduction to Core Data
- Differences between Core Data and traditional databases
- Creating your Core Data model with relationships
- Creating, updating, deleting, and fetching managed objects
- Using a fetched results controller with a table view
- Creating a Core Data–backed task list management app

None of the apps that we've created together so far have used any type of local persistence. The implication of this is that the data you store in your apps will never be saved. If you quit an app and relaunch it, it won't be able to retrieve anything that you've created. Depending on the type of application you're building, you could store and fetch this data remotely on some server. The problem with this would be making your users wait while you make a long request to retrieve

Figure 11.1 Core Tasks, the Core Data–backed task list management application

their data. Also, if they didn't have an internet connection, you wouldn't be able to retrieve it for them.

Thankfully, Apple has provided us with the Core Data framework. Core Data can help us by allowing us to do object management and persistence within our apps. It's such a big and powerful framework that entire books have been written on Core Data alone. We'll cover many of the important parts—the parts that will allow you to write your own Core Data–backed application by the end of the chapter.

The application you'll be creating will be one that will allow you to manage a list of tasks. You could create a task list for work, personal errands, groceries—you name it. Each list would contain its own set of tasks that could be marked as completed and deleted. Figure 11.1 shows what it will look like.

Coincidentally, your application will be named Core Tasks. It's the perfect way to remind you that this task application will be fully backed by Core Data. First you'll get more familiar with Core Data by finding out just what it is and what makes it different.

11.1 Introduction to Core Data

Core Data is an object-graph and persistence framework provided by iOS: essentially, it lets you persist data within your applications. This is especially useful when creating applications that allow users to save data that they'd expect to see when the app is closed and then relaunched later. One example is the task management app you're going to create. It would be a terrible experience for your users if the tasks they created disappeared when they quit the application. You'll learn how Core Data compares to

other persistence and storage solutions that you may be familiar with, and then you'll set up your application.

11.1.1 Differences between Core Data and traditional databases

Chances are you've worked with or have heard of SQL-based database solutions such as MySQL, PostgreSQL, or SQLite. Both Core Data and SQL databases provide a means of persistently storing and searching data. Core Data does this well because it's backed by SQLite. What makes Core Data different is the fact that it acts as a layer above traditional SQL databases that makes it easier for you to work with objects in your application, in some ways similar to an ORM (object-relational mapping) tool.

Imagine you have an application that catalogs people at specific companies. You can have two types of objects, `Employee` and `Company`. The relationship between these two objects would mimic real life by specifying that one company can have many employees. This inherently means that an employee essentially belongs to a company. In a traditional database there would be one table for employees and one for companies. Each table would contain rows of data for each individual record. For instance, the companies table would have rows that contain records for each different company. Each record within each table would have its own unique identifier, which is most commonly represented by a number.

You can think of the proposed database structure for this example as what's shown in figure 11.2.

Each node in this figure would be a single column for each row within each table. For example, if you had an employee named Bob, the row containing his information would look like the following:

```
id: 42
name: Bob Johnson
company_id: 3
```

He works for a company named Punchkeep, which has the unique identifier of `3`. It would look like the following:

```
id: 3
name: Punchkeep
```

Each employee belongs to a single company. This is why Bob has a column called `company_id`, which is set to `3`. The `company_id` column on employees and the `id` column on companies can be used to define the relationship between the two objects.

Figure 11.2 Proposed database structure for two tables: employees and companies. Each node is a column within a single row in each table.

Using SQL you could find the company that Bob worked for by doing the following, knowing only Bob's unique identifier:

```
SELECT companies.* FROM companies, employees
WHERE companies.id = employees.company_id AND employees.id = 42;
```

This would return raw data to represent the company that Bob works for. You would have to go through each column and manually create a new company object to represent Punchkeep. This is a decent amount of grunt work and can become very cumbersome when you're dealing with complex relationships among multiple objects. You also have to worry about changing many things if you decide to add new columns or rename existing ones. Also, what would happen if you were to delete the Punchkeep company from your database? Bob would be left with a `company_id` value pointing to a company that doesn't exist anymore.

This is where Core Data really shines with object management. Retrieving objects is simple and doesn't require a large amount of manual sifting through data to instantiate a new object. For instance, you won't need to retrieve each value for each record and manually set it on a new instance of an object. As you'll later see, Core Data will do this for you. Also, relationships between objects are handled for you out of the box; in the previous case where Punchkeep was deleted, Core Data could automatically remove the orphaned employee entries for you. Saving and updating objects that are backed by Core Data is simple as well. You'll learn how to do all of this as you progress through the app you're building in this chapter.

11.1.2 What Core Data doesn't do well

You should keep a few things in mind when choosing to use Core Data. Out of the box there are some things that it doesn't do well compared to other SQL-based database solutions. The decision depends heavily on the type of application you're trying to build.

Suppose you're building an application that displays news articles from various news sources. This list of articles could grow to a potentially large number. If you had a feature that caused all news articles to be marked as read, it would require you to update every single object in Core Data. For each object you need to update, Core Data will have to load it into memory. This could potentially be an extremely memory-intensive process if you don't use advanced techniques to perform this action. Some SQLite solutions that persist everything to disk don't face this exact problem because they don't need to load large amounts of data into memory.

Another limitation with Core Data is the specific data constraints. If you're used to using SQL databases, you may be familiar with using unique keys and values for specific columns. These would prevent multiple identical values from being stored in a single table. For instance, you may have a users table where each user has a unique username. This is how many websites restrict multiple people from having the same username. With Core Data this isn't handled for you automatically. Core Data expects you to perform this validation on the business-logic side of your integration within

your app. But those limitations aside, Core Data does offer you a very tightly integrated way to store, retrieve, and manage the objects in your applications. Now let's go set it up and start using it.

11.1.3 Setting up your application

To put Core Data to use, you'll start by creating and setting up a new project in Xcode called Core Tasks. You'll create multiple task lists that will contain their own set of tasks. Open up Xcode and create a new project using the Master-Detail Application template. Name the project Core Tasks, and make sure that the Use Core Data option is checked, as shown in figure 11.3.

Because you used the Master-Detail Application template, Xcode automatically created two separate view controllers for you, `IAMasterViewController` and `IADetailViewController`. Xcode has also added a table view to the master view controller and wrapped both of your views within a navigation controller.

Look at your storyboard, and you'll see that your views have already been set up for you. In figure 11.4 you can see the navigation controller, a scene for your master view controller, and one for the detail view controller.

Since you chose to use Core Data, the application was set up to allow you to create a new object from the master view controller and then view its information within the detail view. Even if this isn't exactly what you wanted, it sets up many things for you so

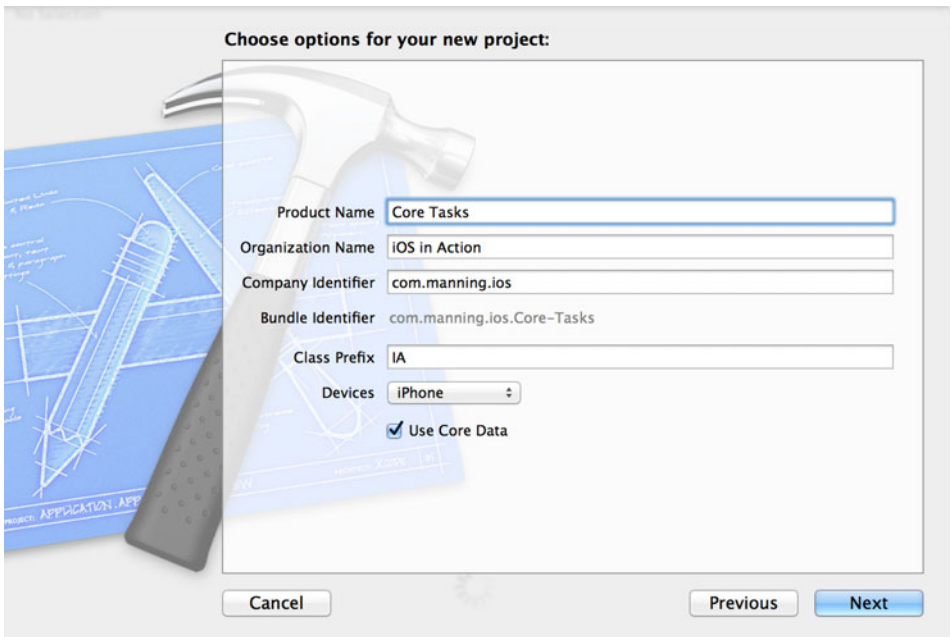


Figure 11.3 Create a new Master-Detail Application project named Core Tasks, and make sure that the Use Core Data option is checked.

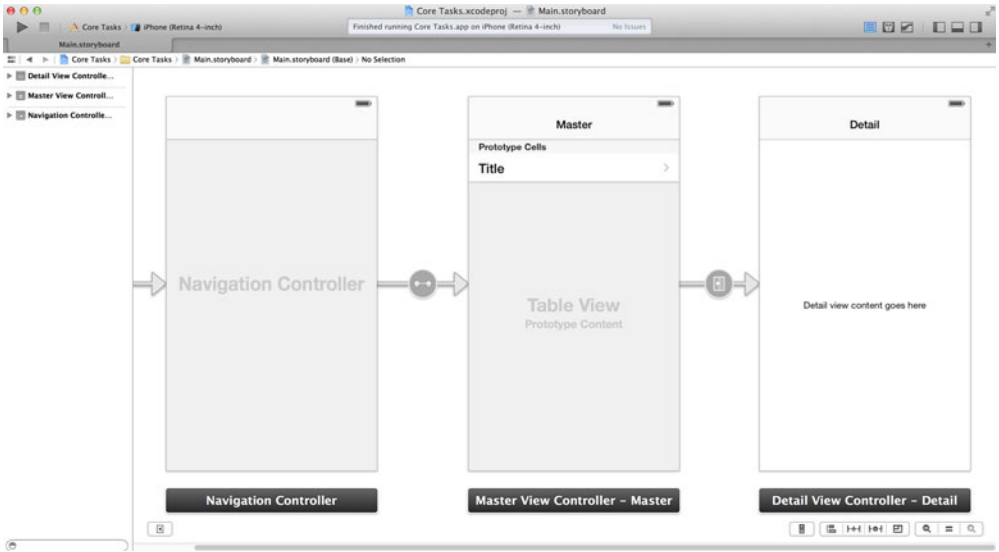


Figure 11.4 Our storyboard already contains a navigation controller and scenes for the master and detail view controllers.

that you only have to change a few items to get started. It also added the Core Data framework for you, which you can see in the project's build settings.

For your needs the current setup of the master view controller will work perfectly. You'll use this view to create a new task list. Once a task list is created, it will be shown within the table view. When you tap the task list, you should be taken to a new view that will show a table view listing all of the tasks that *belong* to that list. Currently the detail view controller contains only a label, not a table view. Let's make that change right now.

First, jump into IADetailViewController.h and add an import for Core Data by adding the following:

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

You'll also notice that it was set up as a subclass of `UIViewController`. You want this to use a table view to list tasks that belong to a list. Change the interface declaration such that it specifies that your class is a subclass of `UITableViewController`, as shown here:

```
@interface IADetailViewController : UITableViewController
```

Also, you'll see a property for an `IBOutlet` that was created for a `UILabel`, as follows:

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

You can remove this line. You won't need this anymore because you'll be using a table view, not a label, to display a list of tasks.

Now jump into the implementation by opening IADetailViewController.m and adding a few methods that are now required because this is a table view controller. To

start off with a good base, replace what's contained within IADetailViewController.m with the code shown in the following listing.

Listing 11.1 IADetailViewController.m

```
#import "IADetailViewController.h"

@interface IADetailViewController ()
- (void)configureView;
@end

@implementation IADetailViewController

#pragma mark - Managing the detail item

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

- (void)configureView
{
    if (self.detailItem) {
    }
}

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

#pragma mark - Table View

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView
➥ numberOfRowsInSection:(NSInteger)section
{
    return 0;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
➥ cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [tableView
➥ dequeueReusableCellWithIdentifier:@"Cell" forIndexPath:indexPath];
    return cell;
}

- (BOOL)tableView:(UITableView *)tableView
➥ canEditRowAtIndexPath:(NSIndexPath *)indexPath
```

```
{
    return YES;
}

- (void)tableView:(UITableView *)tableView
➥ commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
➥ forRowAtIndexPath:(NSIndexPath *)indexPath
{
}

- (BOOL)tableView:(UITableView *)tableView
➥ canMoveRowAtIndexPath:(NSIndexPath *)indexPath
{
    return NO;
}

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

@end
```

You'll be updating most of the table view–related methods soon when you're ready to list, create, update, and delete tasks. Jump back into your storyboard and re-create the scene for your detail view controller.

Once you're in the storyboard, select the existing scene for the detail view controller and remove it completely. Instead of having to add a table view to a `UIViewController`, you're going to use a `UITableViewController`. This way you don't need to manually set up the table view's outlet, data source, and delegate. Go to the Object Library, find a table view controller, and drag it into the storyboard. Once you've added it, go to the inspector and change its class to `IADetailViewController`. Finally, create a new push segue from the prototype cell within the master view controller to the detail view controller that will replace the one that was removed when you first removed the scene. After you've added the segue, select it, and go to the attributes inspector and set its identifier to `showDetail`, as shown in figure 11.5.

Next, change the prototype cell type to Basic instead of Custom. Also set the reuse identifier to `TaskCell`. This will be used when you're displaying each individual task for a list.

While you're still in the storyboard, change the title of the navigation bar in the master view controller's scene. Name the title Lists to convey that this will show the lists you've created.

You've now finished doing most of the initial setup for your project. You're almost ready to start setting up your data model in Core Data. Before you do so, it's important that you understand some key components.

11.2 Managed objects, entities, relationships

In Core Tasks you'll store tasks that belong to a specific list. This involves two separate entities: one for a task and one for a list. You need to be able to define properties for

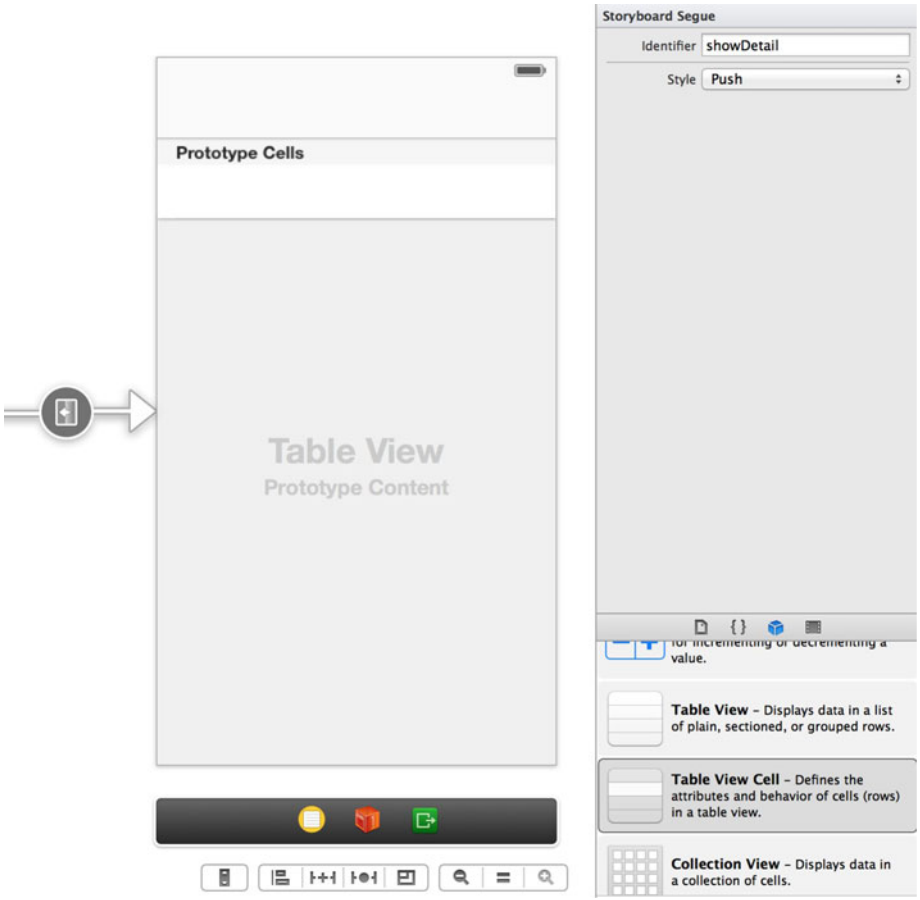


Figure 11.5 Add a new segue from the master view controller to your new detail view controller scene. Once it's been added, change the segue's identifier to `showDetail`.

each of these because you should be able to store the name of a list or a description for a task and whether it's been completed.

How do you go about storing this information within Core Data? When working with a database solution like MySQL, you'd create tables to represent each entity. Core Data is structured a little differently, and you'll soon see that Xcode provides you with a simple but powerful interface to easily configure your data model. In the next section you'll learn how to persist the data crucial to your application by using managed objects and contexts and how to create entities and relationships.

11.2.1 Managed object models and contexts

When thinking about a managed object context (*context* for short) or a managed object model (MOM), try not to get lost in Apple's verbose naming conventions, which make it seem much more complicated than it actually is. All of the data you wish to

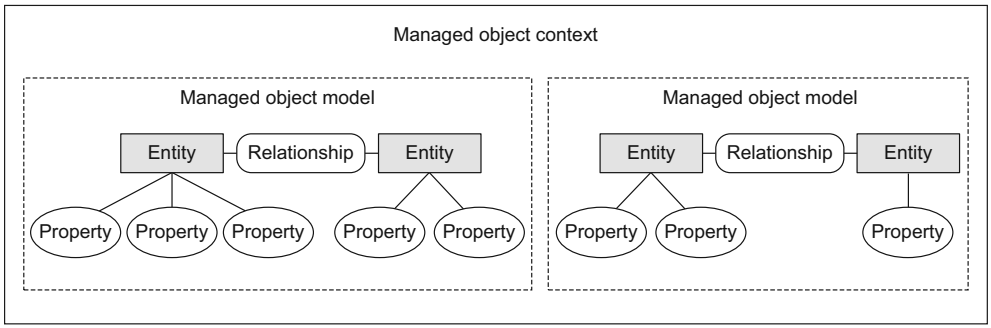


Figure 11.6 Overview of a managed object context and managed object models, which contain entities, their properties, and relationships

store in Core Data needs some place to live, just like people do. The context plays the role of your data's environment by being the gatekeeper to your data and keeping track of its state. Without a context defined in your application, you won't be able to access any of your data because the context is what you'll need to interact with. When you retrieve data from your context, it will attempt to find it within a managed object model.

A managed object model belongs to a context and contains your application's structured data. You can think of a MOM as the city or town that your data lives in; its context is the state or country that your city is within. Also, just as there are many cities within a state, there can be many MOMs within a single context.

Each MOM instance contains *entities*, their properties, and their relationships. You can see an overview of a managed object context and multiple managed object models in figure 11.6.

In your app you'll have two entities representing your data: a list and a task. Each of them will have properties that describe different attributes specific to it.

When a new MOM is created, it's inserted into a context. When you're working with concurrency and using multiple threads, you'll need to have multiple contexts to ensure that everything is thread-safe. Note that it's the context that performs most of the magic for you with Core Data, such as creating, fetching, updating, and saving the data.

The context is responsible for all of your application data at runtime. When you need to fetch, save, delete, update, or even undo changes to your data, you'll be making requests through the managed object context.

When you created your project, Xcode already added code to set up an instance of your managed object context and managed object model. If you open IAAppDelegate.m, you'll see a method called `managedObjectContext`, which returns an instance of an `NSManagedObjectContext`, as shown in the following listing.

Listing 11.2 IAAppDelegate.m returns an instance of a managed object context

```
- (NSManagedObjectContext *) managedObjectContext
{
```

```
    if(_managedObjectContext)
        return _managedObjectContext;

    NSPersistentStoreCoordinator *store = [self
➥ persistentStoreCoordinator];
    if(coordinator)
    {
        _managedObjectContext = [[NSManagedObjectContext alloc] init];
        [_managedObjectContext setPersistentStoreCoordinator:store];
    }

    return _managedObjectContext;
}
```

❶ If `NSManagedObjectContext` already created, return it.

❷ Retrieving reference to persistent store coordinator.

❸ Create a new instance of a `NSManagedObjectContext`.

❹ Add the persistent store to the managed object context.

❺ Return the managed object context.

Running through this method, you can see that if you already have an instance of `_managedObjectContext` you immediately return it ❶. If `_managedObjectContext` is `nil`, you set up a new instance. First, you need a reference to the `NSPersistentStoreCoordinator` from the method we went over previously in this chapter ❷. You then create a new instance of `NSManagedObjectContext` and store it into your instance variable ❸. After that, you set the persistent store to point to `store` ❹, and then you return the newly created managed object context ❺. This code ensures that you have only one instance of a managed object context within your application delegate class.

The managed object context retrieved from this method is also passed to the master view controller after the application is launched, as follows:

```
- (BOOL)application:(UIApplication *)application
➥ didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    UINavigationController *navigationController = (UINavigationController
➥ *)self.window.rootViewController;
    IAMasterViewController *controller = (IAMasterViewController
➥ *)navigationController.topViewController;
    controller.managedObjectContext = self.managedObjectContext;
    return YES;
}
```

❶ Setting the managed object context on the master view controller

Here you can see where the `managedObjectContext` property is set on the master view controller ❶. As we mentioned earlier, the managed object context contains entities. Let's talk about entities, which are managed objects. They're what you'll be using to represent the data you're storing in your app.

11.2.2 Entities and managed objects

Entities are used to describe the objects that you want to store in Core Data. Each entity is a subclass of `NSManagedObject`, which is a generic class that outlines the structure of the data within your entity.

One new type of file was created for you when you created your Core Tasks project. Open the project within Xcode again. In the project navigator you should see a file named Core_Tasks.xcdatamodeld. This data model contains all of the details of each

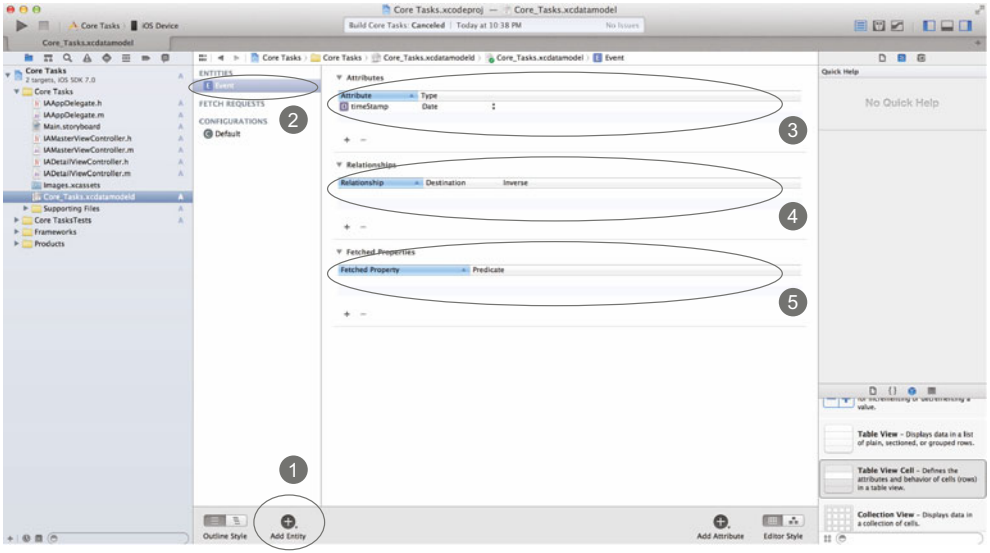


Figure 11.7 Our data model, which shows an entity named Event that was set up for us by the application template used to create the project

entity that you'll be storing using Core Data. The template used to create this project already set up a basic Event entity, as shown in figure 11.7.

Let's examine the different parts of this data model editor by going through each numbered part of figure 11.7.

1. *Add Entity*—Create a new entity in your data model.
2. *Entities*—List of all entities within your data model.
3. *Attributes*—`NSAttributeDescription`—Properties for an entity represented by a specific data type such as an `NSString`, `NSNumber`, `NSDate`, or `NSData`.
4. *Relationships*—`NSRelationshipDescription`—Property that represents a relationship or connection to another entity.
5. *Fetched Properties*—`NSFetchedPropertyDescription`—Properties that are dynamically retrieved through a relationship, search, or a conditional statement.

One important thing to note is that *attributes*, *relationships*, and *fetched properties* are three specific types of items that belong to an entity. You'll be setting up some of these right now. First, remove the Event entity by selecting it and hitting the Delete key on your keyboard.

Now you'll create your first entity for your project to represent a list of tasks. Click Add Entity and name this new entity List. As it stands now, it has no declared attributes, which means that there is nothing you can store for a list.

What attributes will you need to represent each instance of a list? You'll need just one, which will be the name of the list itself, which should be an `NSString`. An attribute can also be declared as optional or required. If an attribute is required, the managed

object context won't allow you to save this list if this attribute is not set. You can also set a default value for an attribute, just in case nothing is specified for it. The attribute you'll be setting for the List entity is shown in table 11.1.

Table 11.1 Attribute properties for the List entity that represents a list of tasks

Name	Type	Optional	Default value
name	String	No	None

Select the List entity and then click the + button within the Attributes section of the editor to add a new attribute, as shown in figure 11.8.

When you add a new attribute, you can set its type to the right of its name. Use the inspector in the right of the data model editor window to expose more advanced options to declare an attribute as required (unchecking Optional).

Let's create another entity to represent a task. As shown in table 11.2, it will have two attributes to represent the task: one will be a description or summary of the task, and one will represent whether or not it has been completed.

Table 11.2 Attribute properties for the Task entity of a summary of a task

Name	Type	Optional	Default Value
summary	String	No	None
completed	Boolean	No	NO
created	Date	No	None

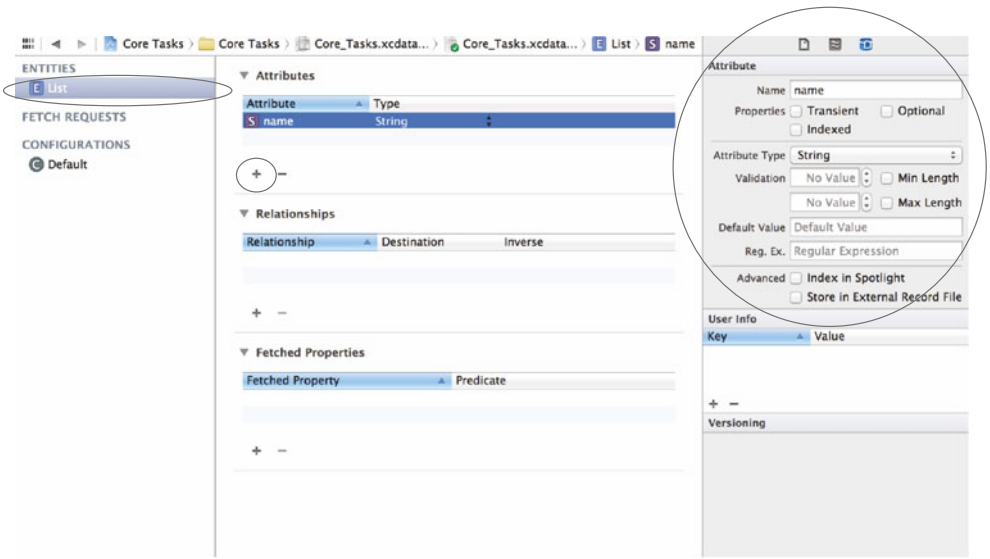


Figure 11.8 Click the + button on the bottom left of the Attributes section of a specific entity to add an attribute.

Create a new entity named Task using these attributes.

After you've added the entity for your tasks and lists, you're still not fully finished. You know that a list will have its own set of tasks. You'll define this by creating a relationship for each entity in your data model.

11.2.3 Relationships between entities

Using Core Data you can also create relationships between objects represented by your entities. Relationships are defined as properties on an entity. You'll have to give each relationship that you set a specific name and information as to how the relationship works so that Core Data knows how to handle your data. These items are described here:

- *Destination entity*—A relationship is between one or two entities. The destination entity is the other entity involved in the relationship you're creating. If you create a task and one of the properties is a list, that list will point to a specific List entity.
- *Inverse relationship*—An existing relationship on the destination entity that can be used to relate back to the entity you're creating the new relationship on. For example, if a list has many tasks, the inverse would be the relationship that defines the specific list that contains a task.
- *Delete rule*—What to do with relationship data after the owner has been deleted. If a list has many tasks and then the list is deleted, should Core Data delete all of its tasks automatically? You can tell Core Data how it should handle this situation for you so that there are no orphaned objects.

You'll start by creating a relationship on your List entity. Create a relationship named `tasks` with its destination entity as Task. In the attributes inspector mark it as a To Many relationship because there are many tasks for one list. Next, set the delete rule to Cascade because you don't wish to keep the tasks if the record for the list that owns them is deleted. This will go through the list's tasks and delete them all. This is shown in figure 11.9.

Right now there is no inverse relationship specified for the `tasks` relationship you just created. If you were to click the drop-down underneath Inverse, you'd see that there is nothing to choose. You'll first need to create a new relationship on the Task entity that has the destination as a List to be able to choose it as the inverse relationship.

Go to the Task entity and create a relationship named `list`, with its destination set to List. It's not a To Many relationship because there's only one list for each task. Also, for the delete rule, specify No Action because you shouldn't delete a list if you delete only one of the tasks. Once you've added this, go back to the List entity and set the inverse relationship for `tasks` to `list`. This allows you to retrieve a list's tasks and go backward and retrieve the list that owns an instance of a particular task.

You can get a better picture of your entities and their relationships by changing the editor style of the data model editor. Figure 11.10 shows a UML (unified modeling language) representation of our entities and their relationships after switching the editor style on the bottom right-hand side of the window.

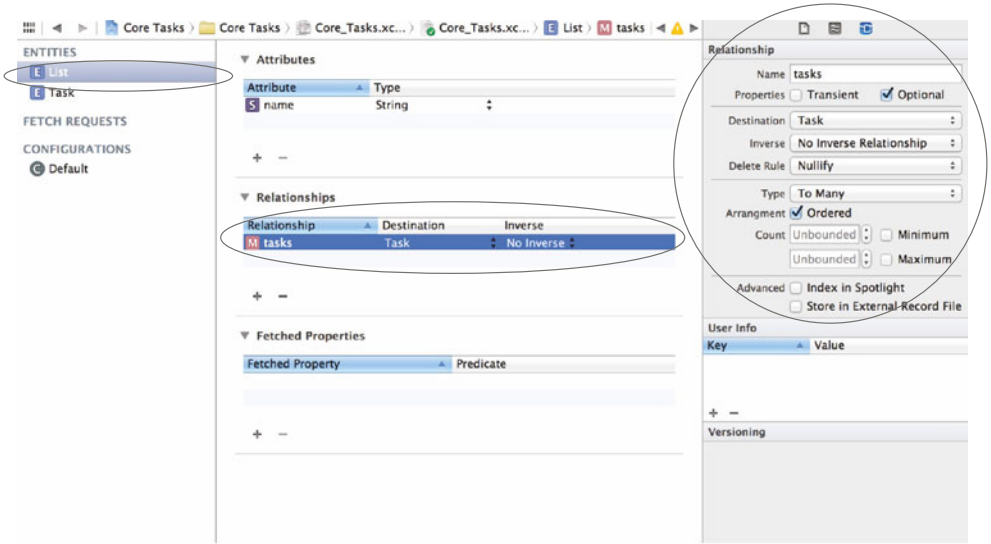


Figure 11.9 Adding a To Many relationship for tasks on the List entity

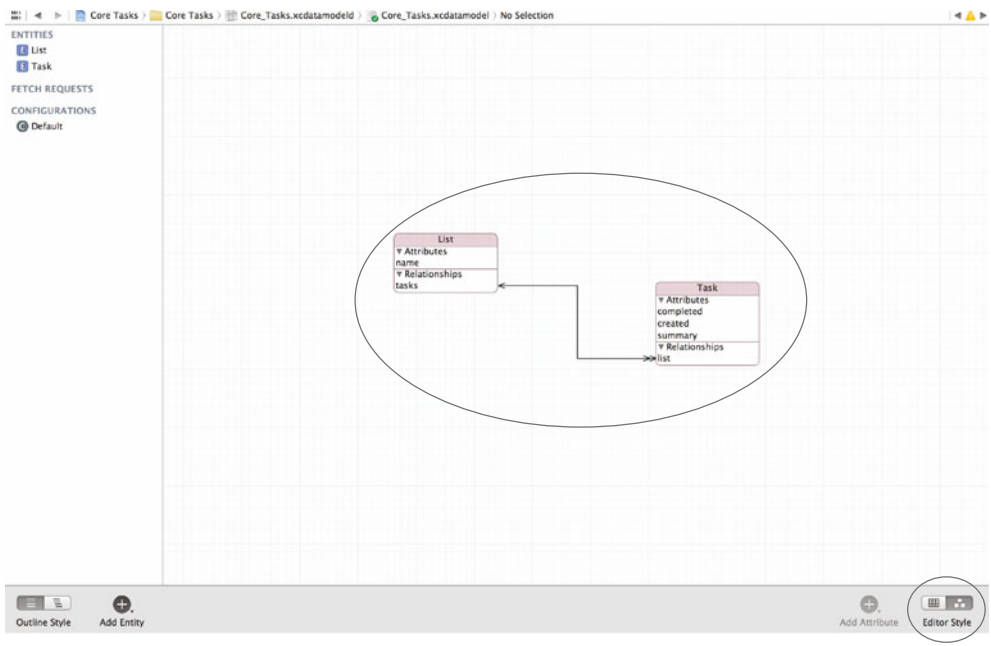


Figure 11.10 The List and Task entities and their relationships shown as a UML representation by switching the editor style in Xcode

This gives you a better visual overview of the relationships between the entities in your data model. By looking at the arrows drawn outward from each relationship, you can tell which one is a To Many versus a Single relationship. The double arrow signifies that a particular relationship has the cardinality of To Many as opposed to a single arrow.

You've now finished setting up attributes and relationships for your entities. Next, you'll see how to easily generate new files based on your entities to add them to your project.

11.2.4 Generating managed object classes for your entities

Xcode can automatically generate managed object subclasses for you based on the entities in your data model and add them to your project. It's required that you do this and add these classes to represent your entities. You'll need to switch your editor style back to a table view instead of the graph that showcased the relationships between your entities on the bottom right-hand corner of the window. Start by selecting all of the entities in your data model. Next, go to the application menu bar and choose Editor > Create NSManagedObject Subclass, as shown in figure 11.11.

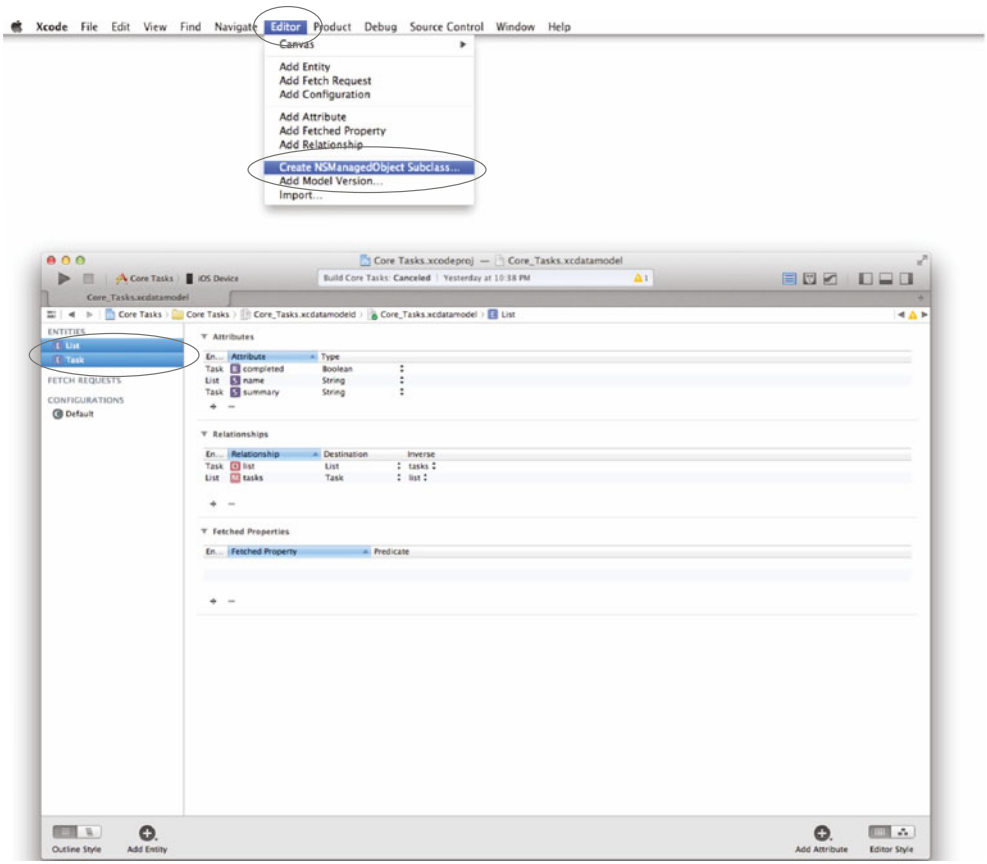


Figure 11.11 Creating `NSManagedObject` subclasses based on your data model's entities

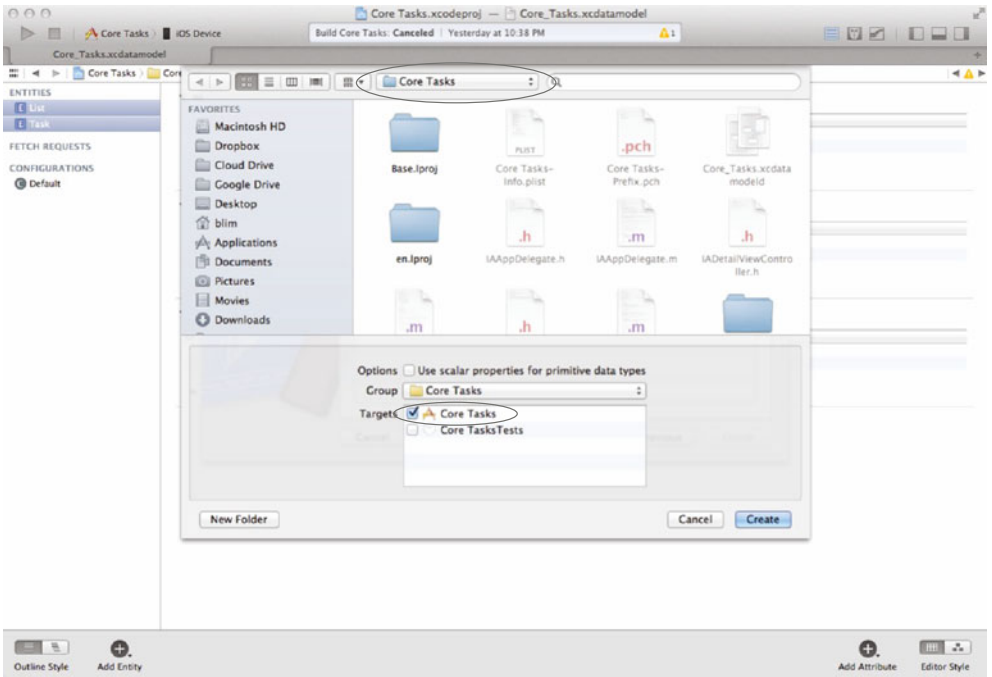


Figure 11.12 Save the generated files within the Core Tasks folder, and make sure that the Core Tasks target is selected.

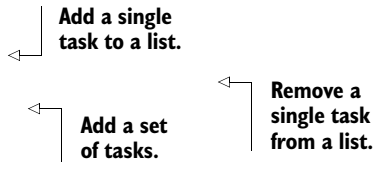
You'll be asked to select the data models with the entities you'd like to manage. There should be only one option, Core_Tasks. Make sure it's selected, and then click Next. You'll then be asked to verify that List and Task are the entities that you want to manage. Because they are, make sure that they're selected and click Next again. The next window that will pop up will ask you where you want them saved. Save them within your Core Tasks project folder, and make sure that the Core Tasks target is checked, as shown in figure 11.12.

When you have finished, you should have two new classes added to your project for lists and tasks. Look at the interface for the `List` class by opening List.h from the project navigator; the interface is shown in the following listing.

Listing 11.3 Interface for generated `List` class

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

@class Task;

@interface List : NSManagedObject

@property (nonatomic, retain) NSString * name;
@property (nonatomic, retain) NSSet *tasks;
@end
```

```
@interface List (CoreDataGeneratedAccessors)

- (void)addTasksObject:(Task *)value;
- (void)removeTasksObject:(Task *)value;
- (void)addTasks:(NSSet *)values;
- (void)removeTasks:(NSSet *)values;

@end
```



Notice in listing 11.3 that there are many predefined methods that have to do with adding and removing tasks that belong to a list. If you were to look at Task.h, you'd see that these methods aren't there. This is because of the `tasks` relationship that you added, which told Core Data that there could be many tasks for one list.

Next, take a look at the implementation by opening List.m. You should see the code shown in the next listing.

Listing 11.4 List.m

```
#import "List.h"
#import "Task.h"

@implementation List

@dynamic name;
@dynamic tasks;

@end
```

What may be shocking is that even though many methods were defined in the interface, the implementation is handled for you by Core Data. Also notice that your properties are using `@dynamic` rather than `@synthesize`. This tells the compiler that the getter and setter methods are not generated by your class but are generated somewhere else instead, specifically by Core Data. For example, to create a new instance of `List` and programmatically set the value of the `name` property, you would normally do the following:

```
List *list = [[List alloc] init];
list.name = @"Groceries";
```

Because you don't have getter and setter methods defined in the class, you can rely on key-value coding to help you set values for properties on an entity. You can do this by using the `setValue:forKey:` method:

```
List *list = [[List alloc] init];
[list setValue:@"Groceries" forKey:@"name"];
```

You're now finished setting up the entities for your Core Tasks app. Let's see how you can use these managed objects that you've created with Core Data. By doing so, you can also continue building your app.

11.3 Working with managed objects

You'll now be able to use the managed objects classes you created for the List and Task entities by creating, updating, and deleting instances of them within Core Data inside

your application. This means you get to dive back into your code and modify what Xcode has already set up for you in your project. First, you'll learn how to create a new managed object instance and save it. You'll then learn how to update it and then remove it.

11.3.1 Creating, updating, and deleting managed objects

Earlier we discussed how managed objects belong to a managed object context. We also showed how a reference to the managed object context from your application delegate was passed to the master view controller. You'll be referencing this managed object context when you work with your data. Remember, the managed object context is essentially the gatekeeper that manages your objects in Core Data.

CREATING MANAGED OBJECTS

Let's quickly see how you'd create a new list in your Core Tasks application. Bring up Xcode once again and open IAMasterViewController.h. You're first going to declare that you conform to the `UIAlertViewDelegate` protocol. You're adding this because you'll be using a `UIAlertView` with a text field to let users create a new list. Once you've added it, hop into IAMasterViewController.m. Add the following `import` statement to bring in the class for your List entity.

```
#import "List.h"
```

Underneath the imports, add the following line, which will be used to tag the alert view that you'll be showing.

```
#define kAlertNewList 1000
```

Next, you need to find the `insertNewObject:` method that was generated for you. This method is called when the bar button item on the top right is tapped to create a new object in Core Data. Replace this method with that shown in the following listing.

Listing 11.5 `UIAlertView` to ask for a new list name in `insertNewObject:`

```
- (void)insertNewObject:(id)sender
{
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Create List"
                                                    message:nil
                                                   delegate:self
                                          cancelButtonTitle:@"Cancel"
                                          otherButtonTitles:@"Save",
    nil];

    [alert setTag:kAlertNewList];
    [alert setAlertViewStyle:UIAlertViewStylePlainTextInput];
    [alert show];
}
```

1 Create a new `UIAlertView` instance.

2 Set a tag on it to reference it by.

3 Set the alert type to have a text view.

4 Show the alert view.

Here you first create a new `UIAlertView` instance that asks your users for a new name for a list they want to create 1. You then set the `tag` property 2 so that you can reference this alert view within its delegate method. Next you set the alert view style 3 to

`UIAlertViewStylePlainTextInput` so that it has a text field to supply a name for the list. Finally, you show the alert view ❹.

What you want to do now is add the delegate method that's called when the `UIAlertView` is dismissed. Add the following method underneath the `insertNewObject:` method.

Listing 11.6 Creating and saving a new List after the alert view is dismissed

```
- (void) alertView:(UIAlertView *)alertView
     didDismissWithButtonIndex:(NSInteger)buttonIndex
{
    if (alertView.tag == kAlertNewList && buttonIndex !=
➥ alertView.cancelButtonIndex)
    {
        UITextField *textField = [alertView textFieldAtIndex:0];
        if ([textField.text length] == 0)
            return;

        List *newList = [NSEntityDescription
➥ insertNewObjectForEntityForName:@"List"
➥ inManagedObjectContext:self.managedObjectContext];

        [newList setValue:textField.text
                ➥ forKey:@"name"];

        NSError *error = nil;
        if (![self.managedObjectContext save:&error])
            NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
    }
}
```

❶ Check alert view by checking the tag and verifying that the user did not hit Cancel.

❷ Retrieve a reference to the text field from the alert view.

❸ Check if user entered in any text; if not return.

❹ Create a new managed object instance for a List.

❺ Set the `name` property on the list.

❻ Save the managed object context.

You first check to see if the alert view you're inspecting is the right one by comparing its tag and making sure that the user did not click the Cancel button ❶. You then retrieve a reference to the text field from the alert view ❷ and check if the user entered anything ❸. After that you instantiate a new instance of a managed object context that represents a new list ❹ in the managed object context. Using the text field text value, you set the `name` on your managed object ❺ and then save it ❻ using the managed object context.

The biggest takeaways from this method are the following lines. Let's first look at when you actually instantiated a new `List` using the `NSEntityDescription` class:

```
List *newList = [NSEntityDescription insertNewObjectForEntityForName:@ "List"
     inManagedObjectContext:self.managedObjectContext];
```

This inserted a new `List` object in the managed object context you passed in that has the entity name List. If this entity name did not exist, you would have gotten an error. It's because you actually have a List entity in your data model that Core Data knew how to handle this.

When you wanted to save the new list, you had to call the `save` method using the `managedObjectContext` property, because you used this managed object context

when you instantiated a new managed object. How about updating them once they've been created?

UPDATING MANAGED OBJECTS

If you wanted to update the new list that you created, it'd be even easier. If you still had a reference to the managed object, you'd only need to use the `setValue:forKey:` function to update the value of an attribute. For instance, after you saved the `newList`, you could add the following to change its `name` attribute:

```
[newList setValue:@"Another name" forKey:@"name"];

NSError *error = nil;
if (![self.managedObjectContext save:&error])
    NSLog(@"Error updating List: %@, %@", error, [error userInfo]);
```

All you had to do was save the managed object context again after the changes were made. Quite simple isn't it? How about deleting?

DELETING MANAGED OBJECTS

To be able to delete a managed object instance, you'd have to tell your managed object context that you want to delete a specific item. Once again, the managed object context acts as the owner of your managed objects. Using the same `newList` instance, you could do the following to delete it:

```
[self.managedObjectContext deleteObject:newList];

NSError *error = nil;
if (![self.managedObjectContext save:&error])
    NSLog(@"Error deleting List: %@, %@", error, [error userInfo]);
```

All you're doing here is calling the `deleteObject:` method on your managed object context and passing in `newList` as the object that you want to remove. You then call the `save:` method again to make sure that this change is persisted.

You've seen how to create, update, and delete managed objects; now you'll learn how to fetch them from Core Data so that you can list them in the master view in your app.

11.3.2 Using fetch requests to retrieve managed objects

What good would Core Data be if you could save objects but couldn't retrieve them? Retrieving saved objects isn't as simple as updating or deleting, but you'll see that it's not that complicated. The simplest way to retrieve objects is by using an `NSFetchRequest`. The following example shows how you can create a fetch request to retrieve all of the lists stored in Core Data:

```
NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
NSEntityDescription *entity = [NSEntityDescription entityForName:
➥ @"List" inManagedObjectContext:self.managedObjectContext];
[fetchRequest setEntity:entity];

NSError *error = nil;
NSArray *lists = [self.managedObjectContext
➥ executeFetchRequest:fetchRequest error:&error];
```

1. Create a new fetch request.
2. Create an entity description for lists.
3. Set the entity on the fetch request.
4. Execute the fetch request.

When fetching objects you first need to instantiate a new `NSFetchRequest` ❶. You then create an entity description for the entity you want to retrieve with a reference to its managed object context ❷. Next, you set the entity on the fetch request ❸. The last thing you need to do is execute the fetch request on the managed object context by calling `executeFetchRequest:error:` ❹. This will return an array that contains all managed object instances for your specified entity. In this case it would return all of your lists.

Chances are you may encounter a situation where you wouldn't want to return all elements stored in Core Data for a specific entity. The simplest thing you can do is limit the number of objects returned in the fetch request. Specifying a fetch limit as follows allows you to do just that.

```
[fetchRequest setFetchLimit:20];
```

By doing this, you'd be limiting the total number of objects returned to 20. If you wanted to only load 20 at a time, you could also set an offset so that the next time you fetch you could return the next 20. This is useful when doing pagination, which would allow the user to load more objects to display as they scroll down. You can do this by typing the following:

```
[fetchRequest setFetchOffset:20];
```

It's also possible to change the ordering of the objects returned from the fetch request. For example, if you were making an address book application, you'd want to be able to sort in alphabetical order. You could do this by using a sort descriptor (`NSSortDescriptor`). To be able to sort lists by their name in descending order, you could add a sort descriptor to your fetch request, like this:

```
NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
➥ initWithKey:@"name" ascending:NO];
[fetchRequest setSortDescriptors:@[sortDescriptor]];
```

You can create a sort descriptor by using the `initWithKey:ascending:` method. The key you're passing in is the attribute on your entity that you want to sort by. In this case, it's `name`, which is the name of your list. You're saying that you want it listed in descending order. When you add it to your fetch request, notice that it expects an array of source descriptors. This means that you can add multiple sort descriptors to your fetch request.

Next, you'll learn how to do advanced filtering using predicates.

11.3.3 Filtering results using predicates

Predicates (`NSPredicate`) can be used to filter arrays unrelated to Core Data. They're useful when you want to filter data stored in an array. It is, however, the easiest way for you to filter the results that you retrieve when making a fetch request. If you have an understanding of SQL, you can think of it as the conditional when making a `SELECT` statement. To illustrate this, you'll create a local array of strings that you'll use to filter using predicates.

We'll start off with an example of an array that contains names of people that we've added to an `NSArray` named `names`. This will give you an easy data set that you can use to filter using predicates:

```
NSArray *names = @[@"Spencer", @"Brent", @"Pradeep", @"Wells", @"Michael"];
```

Using various predicate conditions, you can see just what names would be returned, as shown in table 11.3.

Table 11.3 Different predicates applied to the `names` array to see what names would be returned with each different condition

Predicate	Filtered names
`name == 'Spencer'`	Spencer
`name like[c] '*en*'`	Spencer, Brent
`(name ==[c] 'Pradeep') OR (name like[c] '*s')`	Pradeep, Wells
`(name beginswith 'M') OR (name contains[c] 'r')`	Michael, Spencer, Brent, Pradeep

You can create these by using `NSPredicate`'s `predicateWithFormat:` class method and apply it to an array, as shown here:

```
NSPredicate *predicate = [NSPredicate
➥ predicateWithFormat:@"name == 'Spencer'];
[names filterUsingPredicate:predicate];
```

After the predicate is applied, only the names that match are left in the array. You can use predicates similarly when creating fetch requests by using the `setPredicate:` method on a fetch request instance. There are many different ways of filtering using predicates. It's highly recommended that you refer to Apple's Predicate Programming Guide (http://mng.bz/vAHR) for a complete list of different conditions you can specify.

11.3.4 Using a fetched results controller to manage results in a table view

The job of a fetched results controller (`NSFetchedResultsController`) is to help you efficiently manage the lifecycle of managed objects within a table view—when they're created, updated, or removed. If you take a look at the code in the master view controller, you'll see a couple autogenerated methods referencing a fetched results controller. Fetched results controllers also monitor changes to objects in a managed object context. They report these changes to a delegate, which can then notify a table view to update so that it reflects the new changes. For example, if you delete an item in the table view, it should not only visually disappear but also be removed from Core Data. There's also an option to cache the results of the fetch so that if the same data is fetched multiple times, it doesn't need to be refetched from Core Data.

Typically, a fetched results controller is an instance variable on a table view controller. This is evident in `IAMasterViewController` where a `fetchedResultsController`

property was generated for you. Open Xcode and take a look at its implementation by finding the getter method `fetchedResultsController`. Replace the contents of this method with what's shown in the next listing.

Listing 11.7 `fetchedResultsController` getter method updating for fetching lists

```
if (_fetchedResultsController != nil)
    return _fetchedResultsController;

NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
NSEntityDescription *entity = [NSEntityDescription
➥ entityForName:@"List"
➥ inManagedObjectContext:self.managedObjectContext];
[fetchRequest setEntity:entity];

NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
➥ initWithKey:@"name" ascending:YES];
[fetchRequest setSortDescriptors:@[sortDescriptor]];

NSFetchedResultsController *fetchedController = [[NSFetchedResultsController
     alloc] initWithFetchRequest:fetchRequest
➥ managedObjectContext:self.managedObjectContext
➥ sectionNameKeyPath:nil
➥ cacheName:@"Master"];

fetchedController.delegate = self;
self.fetchedResultsController = fetchedController;

NSError *error = nil;
if (![self.fetchedResultsController performFetch:&error])
{
    NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
    abort();
}

return _fetchedResultsController;
```

1. Check if fetched results controller already exists.
2. Create fetch request and set sort descriptors.
3. Create new fetched results controller.
4. Set your class as the delegate.
5. Point `fetchedResultsController` to `fetchedController`.
6. Perform the fetch.

In this method you first check to see if you've already created a fetch request controller ❶. If so, you just return it instead of re-creating it. You then create a fetch request for your List entity and set it to sort by name in ascending order ❷. Next, you instantiate a new `fetchedController` with the fetch request, supply your managed object context, and specify a cache named Master that will hold your results ❸. You then set your master view controller as the delegate ❹ and set the `fetchedResultsController` property to `fetchedController` ❺. Finally, you perform the fetch on the fetched results controller ❻.

When your controller was generated it was specified to conform to the `NSFetchedResultsControllerDelegate` protocol. When the fetched controller gets a notification of a change on one of its result objects, it will call a few delegate methods to let you know of this. In these methods you can update the table view accordingly. Let's quickly go over a few key delegate methods that have been implemented for you:

- `controllerWillChangeContent:`—Notifies delegate that the fetched results controller is *about to* process a change due to an object being added, removed, moved, or updated.
- `controller:didChangeObject:atIndexPath:forChangeType:newIndexPath:`—Notifies delegate that the fetched results controller *did already* change an object at a specific index path. The change type lets you know if an object's been added, removed, moved, or updated. You're also supplied with a new index path if the index path of the object changes or is newly inserted.
- `controllerDidChangeContent:`—Notifies delegate that the fetched results controller has finished making changes.

Taking a look at the implementation of one of these methods can help you better understand just how it helps you. In IAViewController.m find the `controller:didChangeObject:atIndexPath:forChangeType:newIndexPath:` method. It's shown in the following listing, and we'll explain just what's happening.

Listing 11.8 Fetched results controller notifying you an object has changed

```
- (void)controller:(NSFetchedResultsController *)controller
   didChangeObject:(id)anObject
       atIndexPath:(NSIndexPath *)indexPath
     forChangeType:(NSFetchedResultsChangeType)type
      newIndexPath:(NSIndexPath *)newIndexPath
{
    UITableView *tableView = self.tableView;

    switch(type) {
        case NSFetchedResultsChangeInsert:
            [tableView insertRowsAtIndexPaths:@[newIndexPath]
                             withRowAnimation:UITableViewRowAnimationFade];
            break;

        case NSFetchedResultsChangeDelete:
            [tableView deleteRowsAtIndexPaths:@[indexPath]
                             withRowAnimation:UITableViewRowAnimationFade];
            break;

        case NSFetchedResultsChangeUpdate:
            [self configureCell:[tableView cellForRowAtIndexPath:indexPath]
                    atIndexPath:indexPath];
            break;

        case NSFetchedResultsChangeMove:
            [tableView deleteRowsAtIndexPaths:@[indexPath]
                             withRowAnimation:UITableViewRowAnimationFade];

            [tableView insertRowsAtIndexPaths:@[newIndexPath]
                             withRowAnimation:UITableViewRowAnimationFade];
            break;
    }
}
```

1. **Object added—insert it into table view.**
2. **Object deleted—remove from table view.**
3. **Object changed—reconfigure the cell.**
4. **Object moved—remove old row and add new row.**

Here you can see the different cases for when the results of a fetched results controller are modified. If a new object is added, you insert a new row for it in the table view ❶. If it's been removed, you remove it from the table view ❷. When an object is updated, you reconfigure the cell for that object in the table view ❸. Lastly, if an object is moved to a new position, you remove the old row and insert it at the new index path ❹.

You can see just how it works by making one quick change to your master view controller. You've updated all necessary methods except for `configureCell:atIndexPath:`. This method still contains code specific to the event example that was supplied when you generated the project. Change the content of this method to what's shown here so that you can display the name of the list within its cell:

```
- (void)configureCell:(UITableViewCell *)cell
➥ atIndexPath:(NSIndexPath *)indexPath
{
    List *list = [self.fetchedResultsController
➥ objectAtIndexPath:indexPath];
    cell.textLabel.text = list.name;
}
```

Lastly, let's replace the `prepareForSegue` method with what's shown here. This will be in preparation for the next section.

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    if ([[segue identifier] isEqualToString:@"showDetail"])
    {
        NSIndexPath *indexPath = [self.tableView indexPathForSelectedRow];
        List *list = [[self fetchedResultsController]
➥ objectAtIndexPath:indexPath];

        [[segue destinationViewController] setValue:list
➥ forKey:@"detailItem"];
        [[segue destinationViewController]
➥ setValue:self.managedObjectContext forKey:@"managedObjectContext"];
    }
}
```

Now you can run the app and start creating a few lists of your own. Tap the + button on the top right and add a new list with a name of your choice. In figure 11.13 you can see that you can create a new list named Personal and have it automatically listed in your table view.

You can also trigger the Edit function of the table view to remove a specific list. Also, you can swipe to delete if you'd like. Whenever you make any of these changes, your fetched results controller informs your table view controller how to display the correct content.

Give yourself a big old pat on the back! You've just made a fully functional Core Data–backed application. You're almost finished with it. You just need to modify your detail view controller so that it can create a task and add it to a list using a table view.

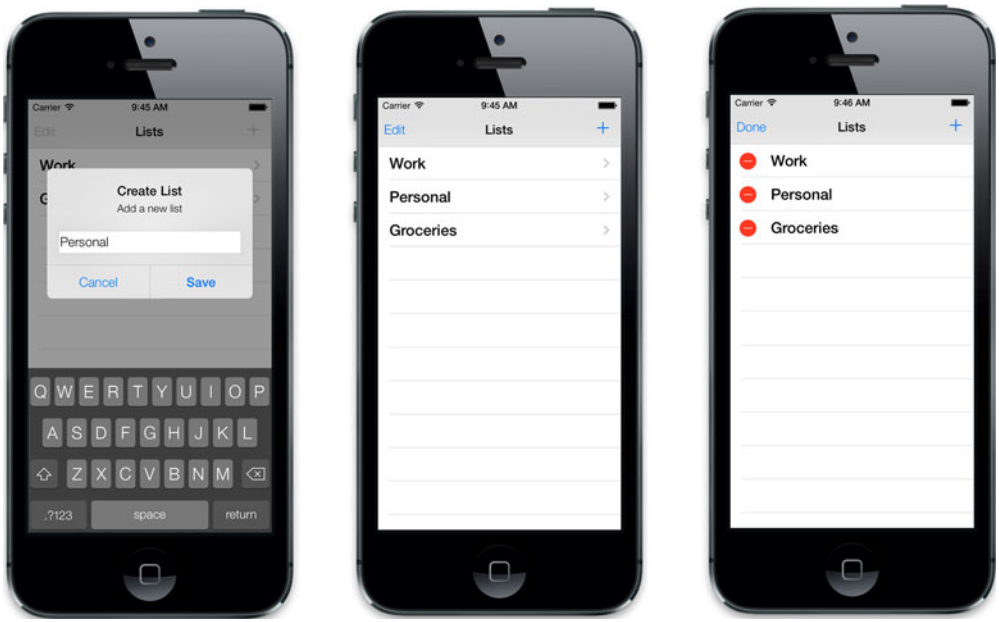


Figure 11.13 When a new list is added, it will automatically appear in your table view. You can also trigger the Edit function of the table view to remove a list.

11.3.5 Adding and removing tasks from a list

Quickly jump into IADetailViewController.h so that you can make the changes you need to start creating tasks. Much of what you'll be adding we've already covered in the previous sections. We'll go over a few key points that cover how to add and remove tasks from a specific list. First, replace the contents of IADetailViewController.h with the code shown in the following listing.

Listing 11.9 IADetailViewController.h

```
#import <UIKit/UIKit.h>
#import <CoreData/CoreData.h>
#import "List.h"

@interface IADetailViewController : UITableViewController
➥ <UIAlertViewDelegate>

@property (strong, nonatomic) List *detailItem;
@property (strong, nonatomic) NSManagedObjectContext *managedObjectContext;

@end
```

First, you import the `List` class to replace the generic `NSManagedObject` property. You also change this class to be a delegate of `UIAlertView` and add a property for a managed object context. The `detailItem` and `managedObjectContext` properties are set from the master view controller's `prepareForSegue:` method.

Next, hop into IADetailViewController.m and replace your code with what's shown in the next listing to give you a clean base to start with. We'll be filling out this class together.

Listing 11.10 IADetailViewController.m

```
#import "IADetailViewController.h"
#import "IAAppDelegate.h"
#import "Task.h"
#import "List.h"

@interface IADetailViewController ()
- (void)configureView;
@end

#define kAlertNewTask 1002

@implementation IADetailViewController

- (void)setDetailItem:(id)newDetailItem {
    if (_detailItem != newDetailItem) {
        _detailItem = newDetailItem;

        [self configureView];
    }
}

- (void)configureView
{
    if (self.detailItem)
    {
        self.title = [self.detailItem valueForKey:@"name"];
        UIBarButtonItem *addButton = [[UIBarButtonItem alloc]
➥ initWithBarButtonSystemItem:UIBarButtonSystemItemAdd
➥ target:self
➥ action:@selector(insertNewObject:)];
        self.navigationItem.rightBarButtonItem = addButton;

    }
}

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

@end
```

What you've just pasted into your implementation of `IADetailViewController` contains three methods that are already filled out for you. The first one is `setDetailItem:`, which is the setter method for the list that owns all of the tasks shown within this view controller. The other method is `configureView`, which is similar to what you have in the master view controller. You're setting the title of the view to the name of the list that was clicked to launch this view. You're then adding a button on the top right that will call a method called `insertNewObject:`.

The `insertNewObject:` method will work just like the one you used in the previous view controller. You'll show an alert view that will prompt for a summary for the new task. Add the following `insertNewObject:` method:

```
- (void)insertNewObject:(id)sender
{
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Create Task"
                                                    message:nil
                                                   delegate:self
                                          cancelButtonTitle:@"Cancel"
                                          otherButtonTitles:@"Save", nil];
    [alert setTag:kAlertNewTask];
    [alert setAlertViewStyle:UIAlertViewStylePlainTextInput];
    [alert show];
}
```

Next, you'll need to add the delegate method for the `UIAlertView` that you're showing. This delegate method is triggered when the alert is dismissed. Add the `alertView:didDismissWithButtonIndex:` method using the code shown in the following listing.

Listing 11.11 Delegate method for the alert view that will create a new task

```
- (void) alertView:(UIAlertView *)alertView
     didDismissWithButtonIndex:(NSInteger)buttonIndex
{
    if (alertView.tag == kAlertNewTask && buttonIndex !=
➥ alertView.cancelButtonIndex)
    {
        UITextField *textField = [alertView textFieldAtIndex:0];
        if ([textField.text length] == 0)
            return;

        Task *newTask = [NSEntityDescription
➥ insertNewObjectForEntityForName:@"Task"
➥ inManagedObjectContext:self.managedObjectContext];

        [newTask setValue:textField.text forKey:@"summary"];
        [newTask setValue:[NSDate date] forKey:@"created"];
        [newTask setValue:@(NO) forKey:@"completed"];

        [self.detailItem addTasksObject:newTask];

        NSError *error = nil;
        if (![self.managedObjectContext save:&error])
            NSLog(@"Unresolved error %@, %@", error, [error userInfo]);

        [self.tableView reloadData];
    }
}
```

1. Create a new task managed object.
2. Set the summary value using the text supplied.
3. Set the created value to the current date.
4. Set the task as not yet completed.
5. Add the task to the list.
6. Save the task in the managed object context.
7. Reload our table view.

You're creating a new task managed object ❶ and setting its `summary` ❷, `created` ❸, and `completed` ❹ attributes. You then add it to your list by calling the `addTasksObject:` method, passing in our new task ❺. The managed object context is then saved ❻ and you reload your table view ❼.

Speaking of table views, it's time to start setting up the data source and delegate methods. First, you're going to set up a method called `allTasks`, which will return an array of ordered tasks using the `tasks` relationship on a list. Add the `allTasks` method using the following code:

```
- (NSArray *)allTasks
{
    NSSortDescriptor *completed = [NSSortDescriptor
➥ sortDescriptorWithKey:@"completed" ascending:YES];
    NSSortDescriptor *created = [NSSortDescriptor
➥ sortDescriptorWithKey:@"created" ascending:NO];

    return [self.detailItem.tasks sortedArrayUsingDescriptors:@[completed,
➥ created]];
}
```

All you're doing here is creating two separate sort comparators. These sort comparators are then passed into the `sortedArrayUsingDescriptors:` method on the `tasks` property of your list. This will cause the array of tasks to be sorted by first showing the incomplete tasks and then ordered by created date in descending order.

Now add the following table view methods to let it know how many sections and rows there are:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView
➥ numberOfRowsInSection:(NSInteger)section
{
    return [[self allTasks] count];
}
```

You'll also allow for this table view to be edited, but it won't allow rows to be manually reordered. Add the following delegate methods:

```
- (BOOL)tableView:(UITableView *)tableView
➥ canEditRowAtIndexPath:(NSIndexPath *)indexPath
{
    return YES;
}

- (BOOL)tableView:(UITableView *)tableView
➥ canMoveRowAtIndexPath:(NSIndexPath *)indexPath
{
    return NO;
}
```

Next, to display each row in the table view, you need to implement the `tableView:cellForRowAtIndexPath:` method. Add the following code to your class:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
     cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [tableView
```

```
    dequeueReusableCellWithIdentifier:@"TaskCell"
    forIndexPath:indexPath];

    [self configureCell:cell atIndexPath:indexPath];

    return cell;
}
```

Here you're using the `TaskCell` reuse identifier you set for your view's prototype cell. You're then calling `configureCell:atIndexPath:` and returning the cell. All of the real work for customizing your cells is done within that method. Add it using the following code:

```
- (void)configureCell:(UITableViewCell *)cell atIndexPath:(NSIndexPath
  *)indexPath
{
    Task *task = [[self allTasks] objectAtIndex:indexPath.row];

    BOOL completed = task.completed;
    cell.textLabel.text = [task valueForKey:@"summary"];

    if (completed)
        [cell setAccessoryType:UITableViewCellAccessoryCheckmark];
    else
        [cell setAccessoryType:UITableViewCellAccessoryNone];
}
```

Retrieve the task within the `allTasks` array. ❶

Retrieve completed value. ❷

Set the text label to be the summary value. ❸

Set accessory checkmark if completed. ❹

Set accessory blank if not completed. ❺

At the beginning of this method you retrieve the appropriate task that needs to be rendered within a cell ❶. You then check to see if it's been completed ❷ and set the cell to display the `summary` value of the task ❸. Lastly, if it's completed, you show a checkmark as the accessory view ❹. If it's not completed, you don't show anything ❺.

How do you mark a task as completed? All you have to do is tap on it to mark it as completed or uncompleted. This is done within the `tableView:didSelectRowAtIndexPath:` method. Add the following to your view controller:

```
- (void) tableView:(UITableView *)tableView
  didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    Task *task = [[self allTasks] objectAtIndex:indexPath.row];
    task.completed = @(!task.completed);

    NSError *error = nil;
    if (![self.managedObjectContext save:&error])
        NSLog(@"Unresolved error %@, %@", error, [error userInfo]);

    [self.tableView deselectRowAtIndexPath:indexPath animated:YES];
    [self.tableView reloadData];
}
```

❶ **Retrieve selected task.**

❷ **Change the completed property.**

❸ **Save change to the managed object context.**

❹ **Deselect the row you've just selected.**

❺ **Reload the table view.**

When a row is selected you first retrieve the task ❶. You then set the completed value to the opposite of its current value ❷. After that's been set, you save it on the managed object context ❸. You then deselect the row ❹ and reload the table view ❺ to reflect your changes.

The last method you need to implement is the one to delete a task if someone slides to delete it from the table view. This method will remove it from the list that owns it and will remove it from the managed object context. Add this method to your class using the following code:

```
- (void)tableView:(UITableView *)tableView
➥ commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
➥ forRowAtIndexPath:(NSIndexPath *)indexPath
{
    if (editingStyle == UITableViewCellEditingStyleDelete)
    {
        Task *task = [[self allTasks] objectAtIndex:indexPath.row];
        [self.detailItem removeTasksObject:task];

        NSError *error = nil;
        if (![self.managedObjectContext save:&error])
            NSLog(@"Unresolved error %@, %@", error, [error userInfo]);

        [self setEditing:NO animated:YES];
        [self.tableView reloadData];
    }
}
```

1 Check if deleting.

2 Retrieve task to delete.

3 Remove from managed object context using method on List.

4 Save managed object context.

5 Exit editing mode.

6 Reload table view.

You first check to see if you're in editing mode and deleting 1. If so, you retrieve the task that you want to remove 2 and delete the object from your managed object context using the `removeTasksObject:` method of List, so your `detailItem` is up to date 3. After saving your changes 4 you exit edit mode 5 and reload your table view 6.

That's it! Cue a round of applause, balloons, and magic unicorns. After you're finished you should be able to run the app and use your Core Tasks app. You should be able to create and remove new tasks within different lists, as shown in figure 11.14.

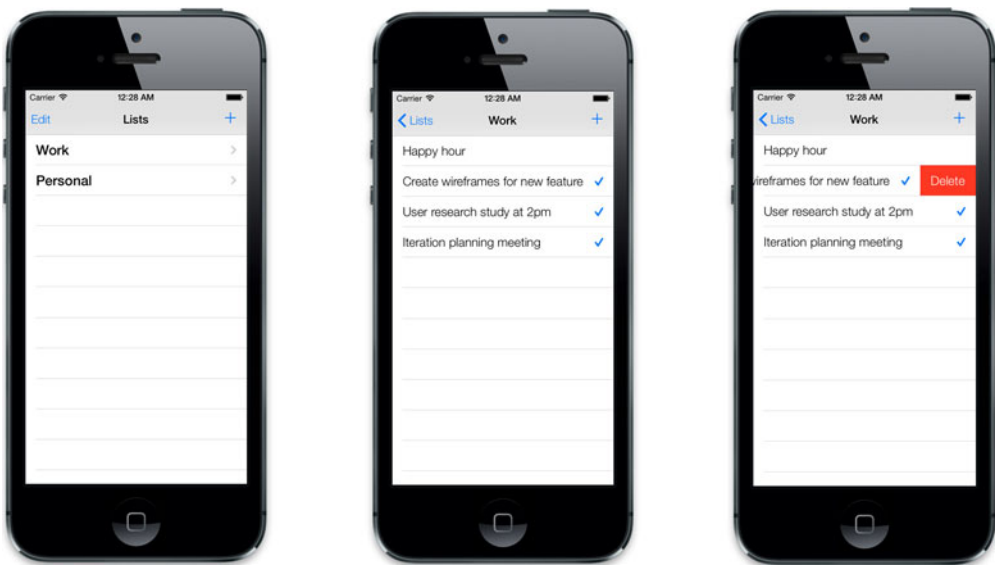


Figure 11.14 Our completed Core Tasks application

You'll be able to create and delete lists, dive into those lists to create tasks, complete tasks, and also delete them. You've made quite a useful application that you can use daily. It's also something that's dependable because your data should still be there the next time you open the app.

11.4 Summary

In this chapter you've covered quite a bit of ground. You've had an overview of Core Data and explored the differences between it and traditional SQL databases. You learned how to structure the data model for Core Data and to create relationships between entities. You also learned how to create, update, delete, and fetch managed objects. When listing out your lists, you used a fetched results controller to manage your managed objects. You used all of this knowledge to create your own task management app that was backed by Core Data. Also, as we mentioned earlier, even though you now have great first-hand experience with Core Data, there's still so much more to learn and so much more that you can do with it.

- Core Data helps you with object management and persistence in your iOS apps.
- The managed object context acts as the gatekeeper to your data and keeps track of its states.
- A managed object model represents an outline of your entities in Core Data.
- Entities represent each object that you store in Core Data.
- Core Data is very tightly integrated with Xcode. You never have to leave Xcode to work with Core Data.
- Relationships are defined as properties on an entity and are used to connect one entity to another.
- Managed objects are instances of entities stored in Core Data.
- You can use fetch requests to retrieve managed objects.
- Filtering results in a fetch request can be done by using predicates.
- A fetched results controller can be used to manage the lifecycle of objects retrieved from fetch requests.

Part 3

Application extras

You've mastered the basics and have armed yourself with the knowledge to build real-world applications. In this part of the book you'll explore a few extra topics that will be sure to help you in the future.

In chapter 12 you'll learn about AirPlay by creating your own streaming music player that uses an Apple TV as an external display. Chapter 13 teaches you about push notifications and how you can set up everything necessary to have them in your own apps. Then in Chapter 14 you'll learn how to implement parallax and realistic animation effects using iOS 7's new APIs for motion and dynamics.

12 Using AirPlay for streaming and external display

This chapter covers

- Introduction to AirPlay
- Streaming content over AirPlay
- Creating a view controller for an external screen
- Displaying content on an external screen
- Creating an AirPlay-powered music player

Being able to wirelessly stream audio to a sound system is one of the most amazing things that some of us take for granted. Not too long ago we were looking for the 3.5 mm stereo cable that we used to play audio from our devices. Actually, most people don't have the ability to stream audio wirelessly in their home or automobile. You can count on this changing soon, with the advent of wireless streaming solutions built into the devices you use in your everyday life—especially in your phones. With Apple's AirPlay you'll soon be able to have one less tangled cable lying around.

AirPlay lets you stream not only audio but also photos, videos, and even content to be used as an external screen, right from your devices. In this chapter you'll learn more about AirPlay and its different applications. You'll be able to then use this knowledge to build your own app, step by step. The app you'll be building is

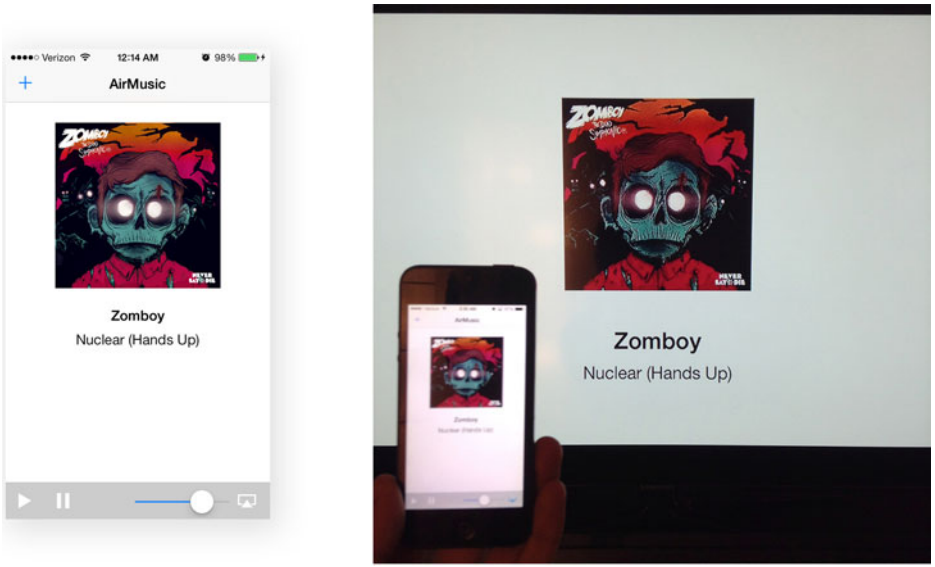


Figure 12.1 You'll be creating your own AirPlay-powered mini music player named AirMusic.

called AirMusic, and it will be your own mini AirPlay-supported music player that can also wirelessly broadcast content on external screens. Take a look at it in figure 12.1.

It's important to note that you can test this application only on a real device. The iOS Simulator does not allow you to connect to any AirPlay devices. It's also important to have an AirPlay device that at least supports audio streaming. Later in the chapter you'll be using AirPlay to display content on an external screen. You'll need an AirPlay device that supports video to be able to try this out.

First, you'll learn more about Apple's AirPlay, what you can do with it, and which devices are supported, and then you'll start setting up the base of your application.

12.1 Introduction to AirPlay

Apple originally had wireless audio streaming through what they called AirTunes. There were AirTunes-supported radios, alarm clocks, and other types of audio devices that allowed users to stream music from their Macs and supported iDevices. Things became interesting when Apple announced AirPlay with iOS 5 and the Apple TV. Using the Apple TV as an AirPlay receiver, people were then able to stream photos and videos, as well as just audio, wirelessly to their TV. It was even possible to mirror the display of a device and use the TV as a second screen. Let's take a quick look at some of the examples of great AirPlay integration.

12.1.1 Examples of AirPlay integration

If you haven't used AirPlay before, you'll find out that it's incredibly seamless to use. Within iOS you can see the AirPlay option within the control center. Tapping it will

Figure 12.2 AirPlay shown within the control center and the Mirroring option shown when selecting an Apple TV device

show all of the AirPlay-enabled devices on your current Wi-Fi network. If one of these devices is an Apple TV, it will allow you to mirror whatever is shown on your device directly on your Apple TV–connected television. This is shown in figure 12.2.

If you were playing music, enabling AirPlay would allow you to stream what you were hearing. The same goes for photos and videos. The Photos application has an AirPlay option within the standard share dialog. Using this would allow you to display a particular photo or video on a TV over AirPlay, as shown in figure 12.3.

Mirroring a display offers some interesting possibilities with AirPlay. When you take over the screen of the external display, you have full control over it. You can use it as an extra screen to display relevant content. Apple's Keynote application will let you stream your presentation over AirPlay while showing slide notes, controls, and more on your device. A few racing games get even more creative with AirPlay. They allow the external screen to act as the main screen for the game, leaving your device as the controller with live track information. Some even allow the screen to be split to show the screens of multiple people playing the same game together.

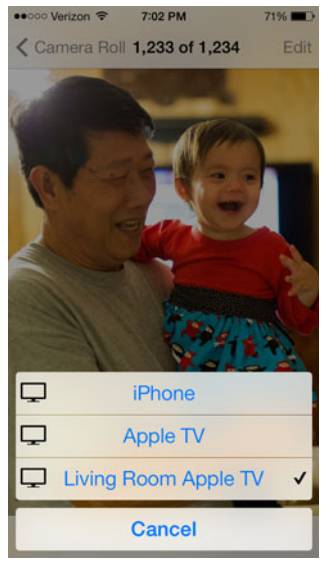


Figure 12.3 The Photos app allows you to display a photo or video using AirPlay.

Let's now start setting up the application we'll be building together in Xcode.

12.1.2 Setting up your application

You'll be using AirPlay to stream audio and display song information on an external screen from your own application. Open Xcode and create a new project named AirMusic using the Single View Application template. Once the project's been created, jump into the Build Phases section, expand Link Binary With Libraries, and add MediaPlayer.framework, as shown in figure 12.4.

You can now start piecing together the main view of your app. Open Main.storyboard and embed a single scene in its own navigation controller. Do this by selecting the scene and then choosing Editor > Embed In > Navigation Controller within Xcode's application menu bar. Once it's embedded, set the title of the navigation item to AirMusic, as shown in figure 12.5.

Next, go into the Object Library, find a `UIImageView`, drag it toward the top of your screen, and set its size to 200 x 200. This will be used to display the album art for a selected song. Go back to the Object Library, grab a `UILabel`, and drag it into your view underneath the `UIImageView`. To make it more visually appealing, set its typeface to bold and text alignment to centered. Next, expand its width to fill up most of the view. You'll use this to display the artist name. After adding this, add another `UILabel` underneath the first one, and make it the same size and center aligned. You'll use this label to display the song name. Your view should look like what's shown in figure 12.6.

To be able to choose a song you'll add a bar button item to the navigation bar. Drag a bar button item to the left side of the navigation bar. Within the attributes inspector set its identifier to Add. You'll add one more button later on in your view controller's implementation to launch the AirPlay picker.

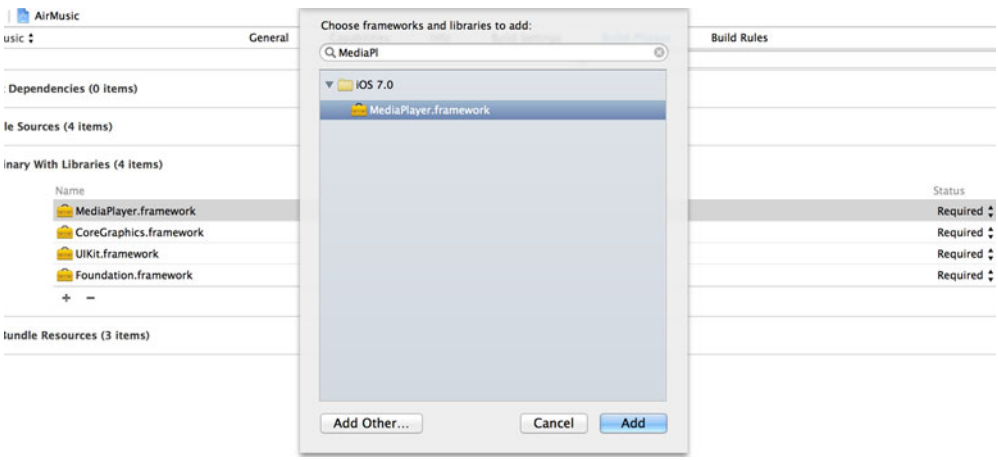


Figure 12.4 Add the MediaPlayer framework to your project within the Link Binary With Libraries section of the Build Phases tab.

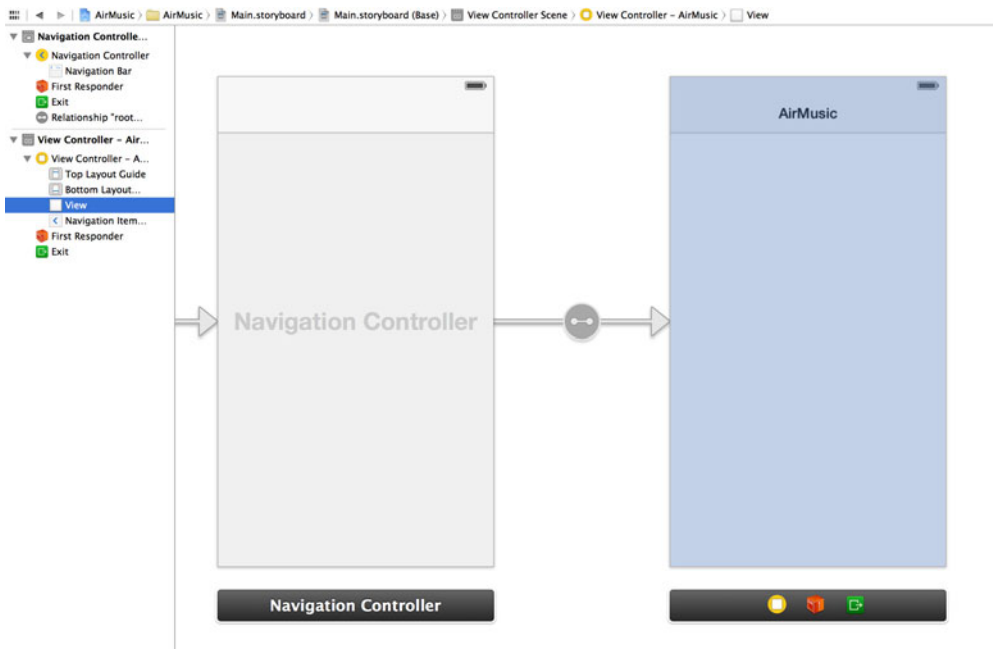


Figure 12.5 Embed your single scene within a navigation controller, and set the navigation item's title to AirMusic.

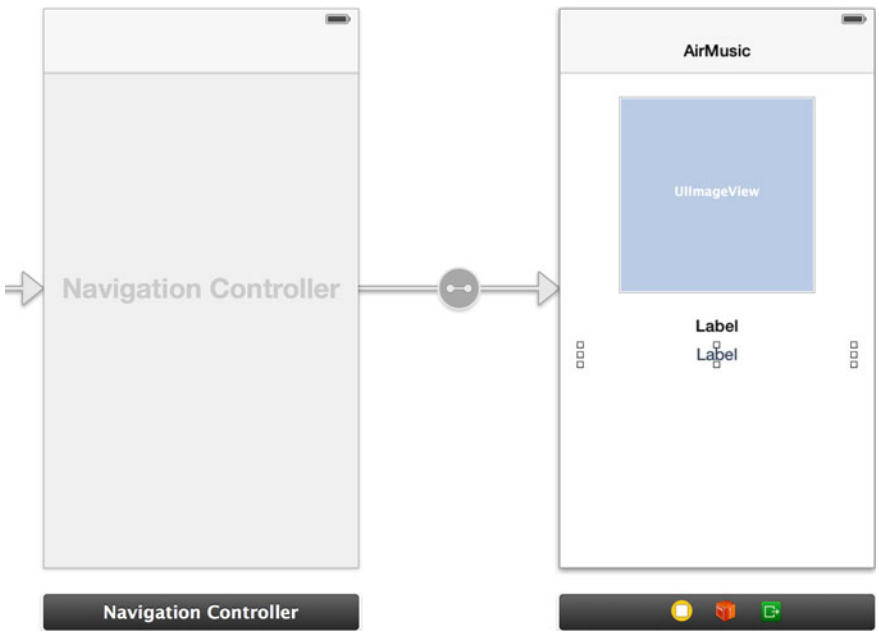


Figure 12.6 Our view after adding an image view and two labels to represent information for a chosen song

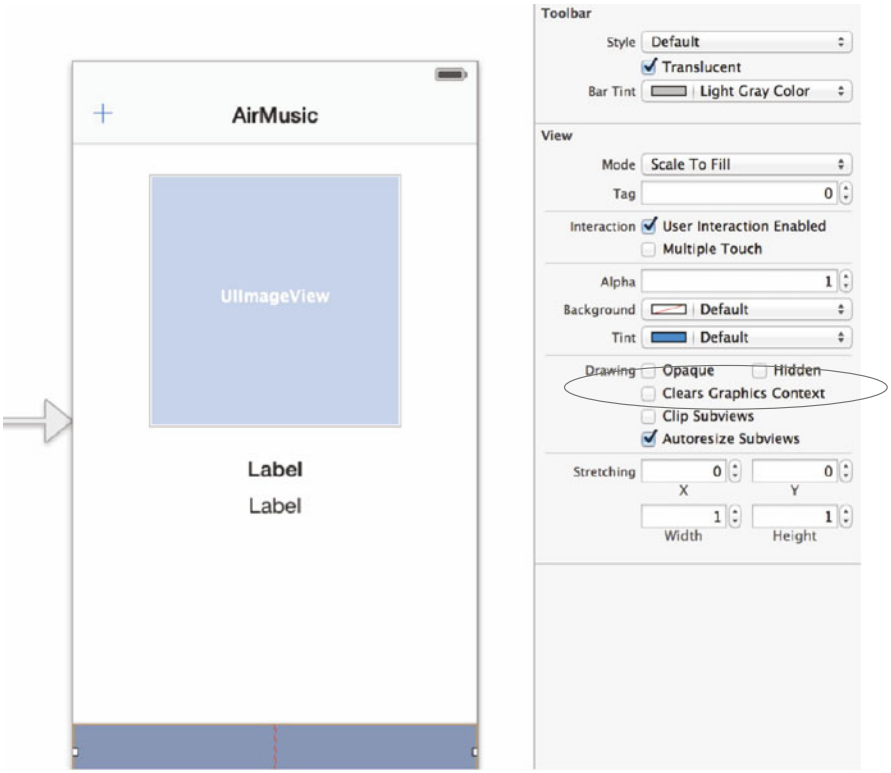


Figure 12.7 With the toolbar selected, go to the attributes inspector and change the Bar Tint attribute to Light Gray Color.

You'll now add a toolbar (`UIToolbar`) to your view that will be used to house the player controls in order to play and pause a song. Go to the Object Library and drag a toolbar to the bottom of your view. Tap it, and then go to the attributes inspector to change its tint to Light Gray Color, as shown in figure 12.7.

The toolbar will have a bar button item already added to it. To avoid confusion, first remove the one that was already in the toolbar when you dragged it into your view. Next, grab another bar button item and add it to the bottom left of the new navigation bar. Set its identifier to Play and tint color to White within the attributes inspector. Now go to the Object Library and find a Flexible Space Bar Button Item; add it to the right of the Play button you've just added. Once it's added, choose another bar button item from the Object Library and place it to the right of the Play button. Set its identifier to Pause and tint color to White. Next, grab a Fixed Space Bar Button Item and drag it in between the Play and Pause buttons. In the attributes inspector change its width to 20.

You're now going to add two auto layout constraints to the `UIImageView` to ensure that the width and the height stay at 200 x 200, as shown in figure 12.8.

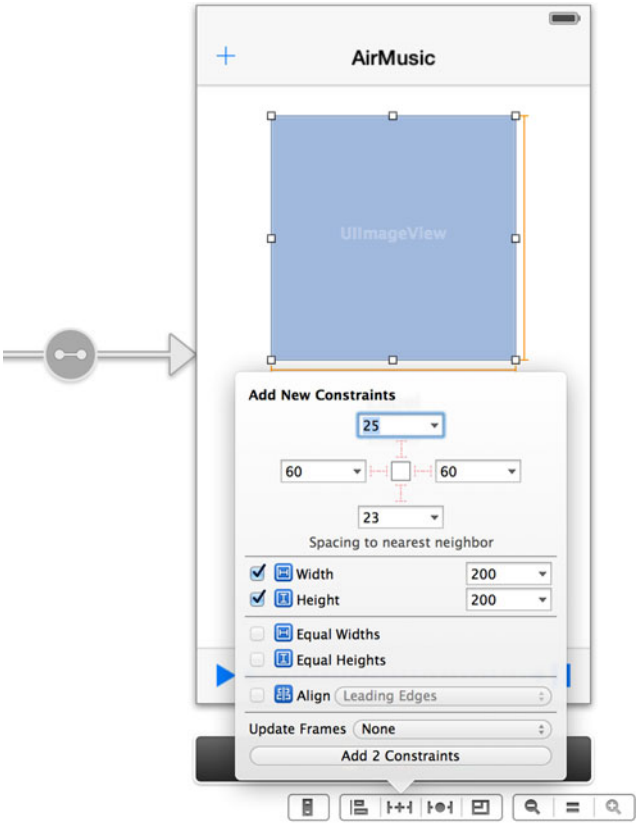


Figure 12.8 Add two auto layout constraints for the width and the height so that your image view's dimensions stay fixed at 200 x 200.

You'll need to add one more set of constraints to the toolbar so that it's always attached to the bottom of your view. Add a constraint to the toolbar's bottom space, as shown in figure 12.9.

Once you've finished, your view should look like what's shown in figure 12.10.

What's left to do is to make a few connections. Let's start with the actions. Open the assistant editor and make a new action for the + button in the top left of your view and name this action `chooseMusic`. For the Play button on the bottom left of your view, create a new action named `playMusic`. For the Pause button create a new action named `pauseMusic`.

You can now make outlets for your image view and two labels. For the image view make a new outlet called `albumImageView`. For the top-most label create an outlet named `artistLabel`. Next, create an outlet for the toolbar called `controlToolbar`. Lastly, for the bottom label create an outlet named `songLabel`.

Now that you've set up the base of your project, you can start adding AirPlay functionality to your application.

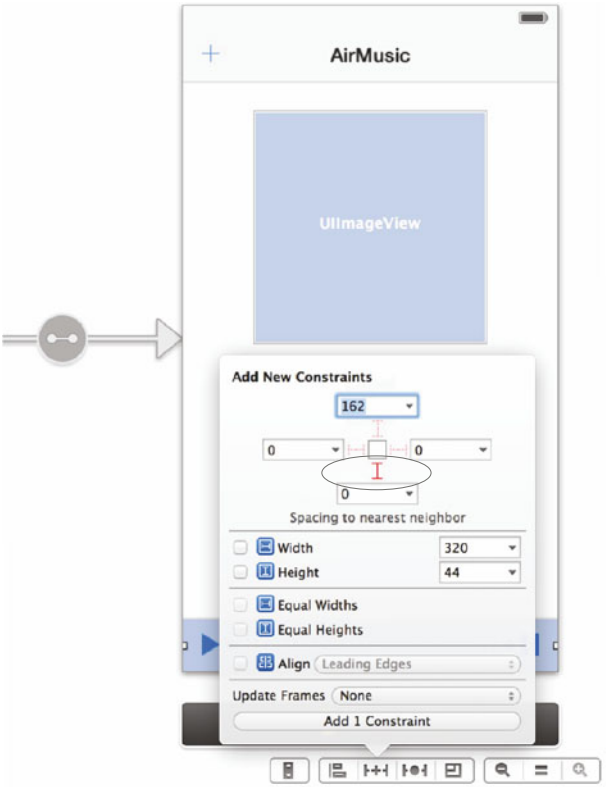


Figure 12.9 Add an auto layout constraint to the toolbar's bottom space so that it always stays locked to the bottom of your view.

12.2 Controlling and enabling AirPlay output

In figure 12.3 we showed a picker displaying different output options for AirPlay. This example was specifically for a photo we wanted to broadcast to an Apple TV. For music there's also the ability to show a volume slider to adjust the volume broadcasted over AirPlay. In this section you'll learn which view allows you to do this and how to add it to your application.

12.2.1 Enabling AirPlay support using built-in media players

There are different built-in methods of playing media within iOS. The `AVPlayer` class, part of the `AVFoundation` framework, gives you more granular control over the entire playback experience. A higher-level media player is the `MPMoviePlayerViewController`. This gives you a prebuilt view that has common playback controls already implemented.

When using the `AVPlayer` you can enable AirPlay by setting the property `allowsExternalPlayback`. For example, you could initialize and enable AirPlay by doing the following:

```
AVPlayer *player = [AVPlayer playerWithUrl:MY_MEDIA_URL];
player.allowsExternalPlayback = YES;
...
```

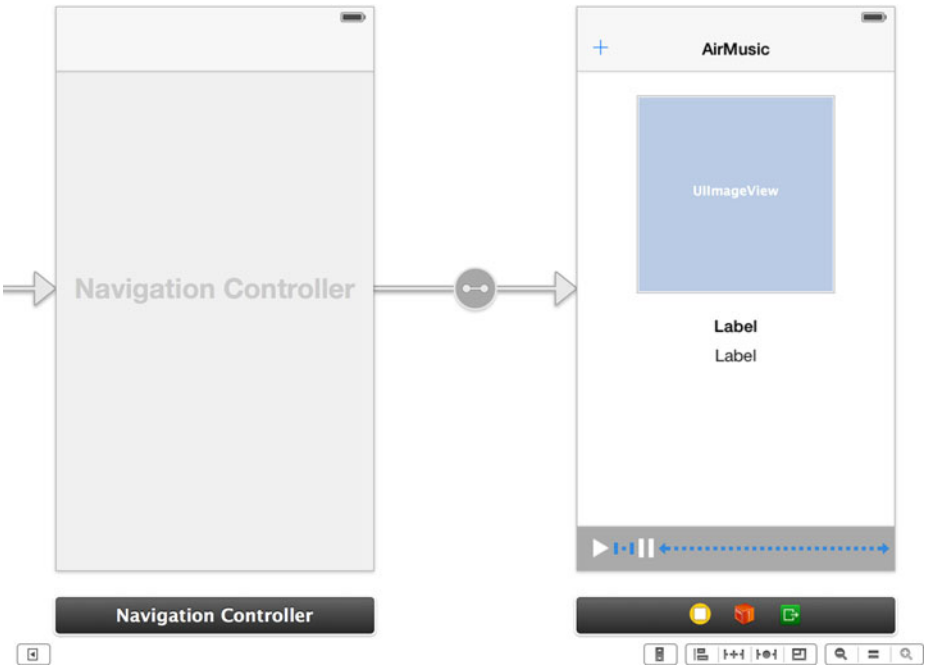


Figure 12.10 The finished view for AirMusic after adding controls to choose a new song and a toolbar to control our music

If you're using the `MPMoviePlayerViewController`, you can set the `allowsAirPlay` property in the same way:

```
MPMoviePlayerViewController *player = [[MPMoviePlayerViewController alloc]
➥ initWithContentURL:MEDIA_URL];
player.allowsAirPlay = YES;
...
```

When using the `MPMoviePlayerViewController` with AirPlay enabled, you'll see an AirPlay button that will allow you to choose which device the audio or video is routed to. This is baked into the `MPMoviePlayerViewController`, but it's actually something you can add to your own view.

12.2.2 Displaying an AirPlay controller to a view

The `MediaPlayer` framework has a class called `MPVolumeView` that allows you to control the system volume on your device using a slider, as well as the output destination for your audio. If you're on Wi-Fi and an AirPlay device is detected, you'll see an AirPlay route button. Both the route button and the volume slider are shown in figure 12.11.

`MPVolumeView` gives you the option to disable either the volume slider or route button from appearing by using the `setShowsVolumeSlider:` or `setShowsRouteButton:` method. When the route button is tapped, a `UIActionSheet` will appear showing the

different output devices you have available. Adding one of these views to an app is simple. All you have to do is make sure you import the `MediaPlayer` framework first:

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

Then do the following.

```
MPVolumeView *volumeView = [[MPVolumeView alloc]
initWithFrame:CGRectMake(0,0,150,20)];

[volumeView setShowsVolumeSlider:YES];
[volumeView setShowsRouteButton:YES];
[parentView addSubview:volumeView];
```

The default size of the route button is 20 points high, but the volume slider can be as wide as you want it to be. The value of 150 points for its width is arbitrary and can be changed to whatever suits your needs.

Next, you'll add this to your own application and get it to function by routing a song of your choice to an AirPlay device.

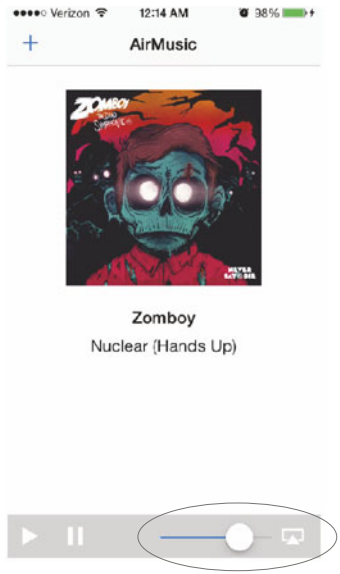


Figure 12.11 The volume slider and the AirPlay route button shown in our very own AirMusic app

12.2.3 Streaming audio to an AirPlay destination in your application

You want to add `MPVolumeView` to the bottom of your view, within the toolbar containing your audio controls. This allows all of the audio controls to appear in a single place. For you to add an `MPVolumeView` within a `UIToolbar`, it must be contained within a `UIBarButtonItem`. Luckily, you can do this with a custom `UIBarButtonItem`.

Open IAViewController.m in Xcode and add the following method named `addAirPlayControls`:

```
- (void)addAirPlayControls
{
    NSMutableArray *items = [self.controlToolbar.items mutableCopy];
    MPVolumeView *airPlayView = [[MPVolumeView alloc]
➥ initWithFrame:CGRectMake(0, 0, 150, 20)];
    [airPlayView setShowsVolumeSlider:YES];
    [airPlayView setShowsRouteButton:YES];
    [airPlayView sizeToFit];

    UIBarButtonItem *airPlayButtonItem = [[UIBarButtonItem alloc]
➥ initWithCustomView:airPlayView];
    self.controlToolbar.items = [items
➥ arrayByAddingObject:airPlayButtonItem];
}
```

1. Getting all items currently in the toolbar
2. Initializing a new `MPVolumeView`
3. Volume slider should be shown.
4. Route button should be shown.
5. Size view to fit within its parent view
6. Create bar button item to act as a wrapper for the `MPVolumeView`.
7. Add the bar button item to the toolbar.

You first retrieve an array of all of the items currently within your toolbar ❶. You then initialize a new `MPVolumeView` ❷ and set its volume ❸ and route ❹ buttons to be

shown. You then tell it to size itself to fit within its parent view ❺. Next, you create a custom `UIBarButtonItem` that will wrap your `MPVolumeView` ❻. Lastly, you add it to your toolbar's array of items ❼.

To ensure that this is called when your view loads, add the following line to your `viewDidLoad` method:

```
[self addAirPlayControls];
```

To make sure that this works as intended, you must finish the rest of your app's music-playing capabilities. Guess what? By the end of this section you'll have created your own basic iPod application. You'll be piggybacking off the existing iPod player within the `MediaPlayer` framework, but why reinvent the wheel? Start by adding functionality to the actions you created earlier. Replace all three of them in IAViewController.m with the code shown in the following listing.

Listing 12.1 Implementing actions to choose, play, and pause a song

```
- (IBAction)chooseMusic:(id)sender                    <-- ❶ Choose music using the MPMediaPickerController.
{
    MPMediaPickerController *picker = [[MPMediaPickerController alloc]
➥ initWithMediaTypes:MPMediaTypeMusic];
    [picker setDelegate:self];

    [self presentViewController:picker animated:YES completion:nil];
}

                                                      <-- ❷ Play music by using the default MPMusicPlayerController.
- (IBAction)playMusic:(id)sender
{
    MPMusicPlayerController *player = [MPMusicPlayerController
➥ iPodMusicPlayer];
    [player setQueueWithItemCollection:self.songQueue];
    [player play];}

                                                      <-- ❸ Pause music by using the default MPMusicPlayerController.
- (IBAction)pauseMusic:(id)sender
{
    MPMusicPlayerController *player = [MPMusicPlayerController
➥ iPodMusicPlayer];
    [player pause];
}
```

In the `chooseSong:` action you use the `MPMediaPickerController` to choose a song to play ❶. You also use the device's default `MPMusicPlayerController` (known as the `iPodMusicPlayer`) to handle your music within the `playMusic:` ❷ and `pauseMusic:` ❸ actions.

Because you're using the `MPMediaPickerController`, you need to conform to the `MPMediaPickerControllerDelegate` protocol. You also need to add a property to your class that will hold a reference to your song queue, which is used to specify which songs to play. It will be set whenever a song is chosen. Go to IAViewController.h and add the following `import` for the `MediaPlayer` framework:

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

Now you'll specify that you conform to the `MPMediaPickerControllerDelegate` by updating your interface declaration, as shown here:

```
@interface IAViewController : UIViewController
➥ <MPMediaPickerControllerDelegate>
```

For your song queue, add the following property for an `MPMediaItemCollection`:

```
@property (strong, nonatomic) MPMediaItemCollection *songQueue;
```

Go back to IAViewController.m and add two delegate methods to properly handle the event when a song is chosen using the media picker and when the media picker is canceled. Add the two methods shown in the following listing.

Listing 12.2 Methods for handling song selection and cancelation

```
- (void)mediaPicker:(MPMediaPickerController *)mediaPicker
➥ didPickMediaItems:(MPMediaItemCollection *)mediaItemCollection
{
    [mediaPicker dismissViewControllerAnimated:YES completion:nil];

    self.songQueue = mediaItemCollection;
    [self populateViewsWithSongQueue];
    [self playMusic:self];
}

- (void)mediaPickerDidCancel:(MPMediaPickerController *)mediaPicker
{
    [mediaPicker dismissViewControllerAnimated:YES completion:nil];
}
```

1. Media items have been selected from the media picker.
2. Dismiss the media picker.
3. Store reference to the selected song.
4. Populate your views based on the selected song.
5. Play the selected song.
6. Media picker was canceled.

When a song is chosen ❶ you first dismiss the media picker ❷. You then hold onto a reference to the `MPMediaItemCollection` using the `songQueue` property ❸, populate your views based on the selected song ❹, and then play it ❺. If the media picker is canceled ❻, you make sure it's dismissed ❷.

The method you call to populate your views with the currently selected song, `populateViewsWithSongQueue`, will retrieve the album artwork as well as the artist and song name. Add this method to your view controller by using the code in the next listing.

Listing 12.3 Populate your views with information from currently selected song

```
- (void)populateViewsWithSongQueue
{
    self.artistLabel.text = @"";
    self.songLabel.text = @"";
    self.albumImageView.image = nil;

    MPMediaItem *mediaItem = self.songQueue.items[0];
    MPMediaItemArtwork *albumArtwork = [mediaItem
➥ valueForProperty:MPMediaItemPropertyArtwork];
    NSString *artistName = [mediaItem
➥ valueForProperty:MPMediaItemPropertyAlbumArtist];
```

1. Clear song-related views.
2. Retrieve the selected song.
3. Retrieve the album artwork.
4. Retrieve the artist name.

```
    NSString *songName = [mediaItem
➥ valueForProperty:MPMediaItemPropertyTitle];

    if (albumArtwork)
        [self.albumImageView setImage:[albumArtwork
➥ imageWithSize:self.albumImageView.bounds.size]];

    if (artistName)
        self.artistLabel.text = artistName;

    if (songName)
        self.songLabel.text = songName;
}
```

❺ Retrieve the song name.

❻ Show the album artwork in the album image view.

❼ Show the artist name in the artist label.

❽ Show the song name in the song label.

In this method you first clear your existing views ❶. You then retrieve the song to play from your queue ❷ and retrieve its artwork ❸, artist name ❹, and song name ❺. You then populate your views by first setting the `albumImageView`'s image to that of the album artwork ❻. Next, you set `artistLabel`'s `text` property to the artist name ❼. Finally, you set `songLabel`'s `text` property to the song name ❽.

One last thing you'll add is the ability to immediately queue up the last-played song from your device's music player. This could be the song you last played in the Music app long before you installed `AirMusic` on your device. It's very simple to do because you're using the default music player. Add the following code to the bottom of `viewDidLoad`:

```
MPMusicPlayerController *player = [MPMusicPlayerController
iPodMusicPlayer];
if ([player nowPlayingItem])
{
    self.songQueue = [[MPMediaItemCollection alloc]
➥ initWithItems:@[player.nowPlayingItem]];
    [self populateViewsWithSongQueue];
}
```

This checks to see if there was a media item within the `nowPlayingItem` property. If so, you create a new `MPMediaItemCollection` and set that to your `songQueue` property. You then populate your view by calling the `populateViewsWithSongQueue` method.

Amazingly, you've just built a functional music application in no time at all! It also allows you to stream and control the volume of the music to an AirPlay-connected device. Again, this application should function fully only on a real device. If you have a paid Apple developer license, you can install this directly from Xcode.

Next, you'll learn how to display this song information using AirPlay's mirroring feature.

12.3 Using external screens with AirPlay

By using AirPlay's mirroring feature you can change the content shown on an external screen. Within an application you can check to see if there are multiple screens available and also observe notifications that let you know when a new screen has

connected or disconnected. When a new screen is available, you can attach your own custom view to it.

12.3.1 Creating a custom view controller for external screens

When working with external screens you cannot rely on Xcode's interface tools, which only help you design for standard-size iOS devices. The size of an external screen is usually unknown, thus requiring you to programmatically create your own views. The size of its subviews should be relative to the size of the screen itself. This allows you to maximize your custom view to fit a very large or small screen.

By using an available external screen you can display content that may be useful for the users of your app. For AirMusic, you'll be using the external screen to show a bigger version of the song information you were already showing. You won't be showing the audio controls because there's currently no way for your users to interact with the external screen directly. They can use your app on their device in the same way as before.

First, you'll create a brand-new view controller called `IAExternalViewController`. It will contain most of the same views that you currently have in `IAViewController` and will also have the same functionality for populating these views. Once you've created this new view controller, open IAExternalViewController.h and replace its content with what's shown in the following listing.

Listing 12.4 IAExternalViewController.h

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

@interface IAExternalViewController : UIViewController

@property (strong, nonatomic) UIImageView *albumImageView;
@property (strong, nonatomic) UILabel *artistLabel;
@property (strong, nonatomic) UILabel *songLabel;

@property (assign, nonatomic) CGRect windowFrame;
@property (strong, nonatomic) MPMediaItemCollection *songQueue;

- (id)initWithWindowFrame:(CGRect)windowFrame
➥ withSongQueue:(MPMediaItemCollection *)songQueue;
- (void)populateViewsWithSongQueue;

@end
```

❶ Frame of the external window's parent window

❷ Custom initializer with frame and song queue

This sets up the image view for the album art, the labels for the artist and song information, and the song queue. You've also added another property for the frame of its parent window (which you'll soon see) ❶, as well as a custom initializer for the `IAExternalViewController` view controller object ❷.

Now open IAExternalViewController.m. You'll be replacing all of the code there with what's shown in the next listing.

Listing 12.5 IAExternalViewController.m

```
#import "IAExternalViewController.h"

@implementation IAExternalViewController

- (id)initWithWindowFrame:(CGRect)windowFrame
➥ withSongQueue:(MPMediaItemCollection *)songQueue
{
    self = [super initWithNibName:nil bundle:nil];
    if (self)
    {
        self.windowFrame = windowFrame;
        self.songQueue = songQueue;
    }

    return self;
}

- (void)loadView
{
    UIView *view = [[UIView alloc] initWithFrame:self.windowFrame];
    [view setBackgroundColor:[UIColor whiteColor]];

    float widthHeight = view.bounds.size.width / 4;
    CGRect imageFrame = CGRectMake((view.bounds.size.width -
➥ widthHeight)/2,
                                   (view.bounds.size.height -
➥ widthHeight)/3,
                                   widthHeight, widthHeight);

    self.albumImageView = [[UIImageView alloc] initWithFrame:imageFrame];
    [view addSubview:self.albumImageView];

    CGRect artistLabelFrame = CGRectMake(0,
                                         imageFrame.origin.y +
➥ imageFrame.size.height + (imageFrame.size.height / 5),
                                         view.bounds.size.width,
                                         45.0f);
    self.artistLabel = [[UILabel alloc] initWithFrame:artistLabelFrame];
    [self.artistLabel setFont:[UIFont boldSystemFontOfSize:40.0f]];
    [self.artistLabel setTextAlignment:NSTextAlignmentCenter];
    [view addSubview:self.artistLabel];

    CGRect songLabelFrame = CGRectMake(0,
                                       artistLabelFrame.origin.y + 55.0f,
                                       artistLabelFrame.size.width,
                                       artistLabelFrame.size.height);
    self.songLabel = [[UILabel alloc] initWithFrame:songLabelFrame];
    [self.songLabel setFont:[UIFont systemFontOfSize:30.0f]];
    [self.songLabel setTextAlignment:NSTextAlignmentCenter];
    [view addSubview:self.songLabel];

    [self setView:view];
}

- (void)populateViewsWithSongQueue
{
    MPMediaItem *mediaItem = self.songQueue.items[0];
```

1 **Custom initializer with window frame and song queue**

2 **Programmatically creating your view**

3 **Populating views with information from the current song**

```
    MPMediaItemArtwork *albumArtwork = [mediaItem
➥ valueForProperty:MPMediaItemPropertyArtwork];
    NSString *artistName = [mediaItem
➥ valueForProperty:MPMediaItemPropertyAlbumArtist];
    NSString *songName = [mediaItem
➥ valueForProperty:MPMediaItemPropertyTitle];

    if (albumArtwork)
        [self.albumImageView setImage:[albumArtwork
➥ imageWithSize:self.albumImageView.bounds.size]];

    if (artistName)
        self.artistLabel.text = artistName;

    if (songName)
        self.songLabel.text = songName;
}

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

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

@end
```

You initialize this view by setting the `windowFrame` and `songQueue` properties ❶. The `windowFrame` property is used when you build your view. The `songQueue` property is used when you populate your view ❸, just like in `IAViewController`. The part that stands out the most is the `loadView` ❷ method, where you programmatically create your view. You calculate the frame of each view depending on the size of the window, which is dependent on the size of the external screen. All that you're doing here is adding one image view and two labels.

Next, you'll see how to attach this view controller to an external screen.

12.3.2 Displaying content on an external screen

To know when an external screen is available you need to add an observer for the `UIScreenDidConnectNotification` on the `NSNotificationCenter`. This is triggered whenever a new screen is connected via AirPlay or even through HDMI/VGA/DVI. Also, using the `UIScreenDidDisconnectNotification` lets you know when a screen is disconnected.

What would happen if an external screen were already connected *before* you registered for these notifications? In this case, you'd only receive the notification for when it is disconnected. This is why you should also check to see if another screen is already available when your application first becomes visible.

Add a call for a method that you'll soon implement on the bottom of `viewDidLoad` that will check for this:

```
[self checkForAndSetupExternalScreen];
```

Now add two observers to IAViewController.m to know when a new screen connects and disconnects. Add the following toward the bottom of `viewDidLoad`:

```
[[NSNotificationCenter defaultCenter] addObserver:self
➥ selector:@selector(screenDidConnect:)
➥ name:UIScreenDidConnectNotification
➥ object:nil];

[[NSNotificationCenter defaultCenter] addObserver:self
➥ selector:@selector(screenDidDisconnect:)
➥ name:UIScreenDidDisconnectNotification
➥ object:nil];
```

Also, add a call to remove yourself as an observer when your view disappears by overriding the `viewDidDisappear:` method:

```
- (void)viewDidDisappear:(BOOL)animated
{
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}
```

These two observers call for two new methods, `screenDidConnect:` and `screenDidDisconnect:`. These methods will be triggered when one of the notifications is observed. Add them to IAViewController.m using the following code:

```
- (void)screenDidConnect:(NSNotification *) notification
{
    [self checkForAndSetupExternalScreen];
}

- (void)screenDidDisconnect:(NSNotification *) notification
{
    if (self.externalWindow)
    {
        [self.externalWindow setHidden:YES];
        self.externalWindow = nil;
    }
}
```

❶ **Set up the external screen.**

❷ **Check if `externalWindow` property is not `nil`.**

❸ **Set `externalWindow` to `hidden`.**

❹ **Set `externalWindow` to `nil`.**

When a screen does connect, you call `checkForAndSetupExternalScreen` ❶. You're going to create this method to attach `IAExternalViewController` to the external screen. On disconnect you check to see if a property named `externalWindow` is not `nil` ❷. If it's not, you set it to `hidden` ❸ and then set it to `nil` ❹. This property is for a `UIWindow` attached to the external screen that houses an instance of `IAExternalViewController`.

Add these two properties to IAViewController.h now. First, add the following `import`:

```
#import "IAExternalViewController.h"
```

Now add the two properties to represent the external view controller and the external window:

```
@property (strong, nonatomic) UIWindow *externalWindow;
@property (strong, nonatomic) IAExternalViewController *externalView;
```

Once again, jump back into IAViewController.m to add the `checkForAndSetupExternalScreen` method shown in the following listing.

Listing 12.6 Checking for and then setting up an external screen

```
- (void)checkForAndSetupExternalScreen
{
    if ([[UIScreen screens] count] > 1)
    {
        UIScreen *screen = [UIScreen screens][1];
        self.externalWindow = [[UIWindow alloc]
➥ initWithFrame:screen.bounds];
        [self.externalWindow setScreen:screen];

        if (!self.externalView)
        {
            self.externalView = [[IAExternalViewController alloc]
➥ initWithWindowFrame:self.externalWindow.frame
➥ withSongQueue:self.songQueue];
            [self.externalWindow
➥ setRootViewController:self.externalView];
        }
        else
        {
            self.externalView.songQueue = self.songQueue;
            [self.externalView populateViewsWithSongQueue];
        }

        [self.externalWindow setHidden:NO];
    }
}
```

1. Check if more than one screen is available.
2. Retrieve second screen.
3. Initialize `externalWindow` using the second screen's bounds.
4. Set `externalWindow`'s screen to the second screen.
5. Check if `externalView` is not yet initialized.
6. Initialize `externalView` with window frame and song queue.
7. Set `externalView` as root view controller for `externalWindow`.
8. Set song queue if `externalView` is already initialized.
9. Manually populate views if `externalView` is already initialized.
10. Set window as not hidden.

Within this method you first check to see if there is more than one screen available ❶. If there's only one screen, then no external screen is attached. If an external screen is available, you retrieve the screen ❷ and initialize a new `UIWindow` using the screen bounds and set it to the `externalWindow` property ❸. You then set its `screen` property to be that of the new external screen ❹. If the `externalView` property is `nil` ❺, you create a new instance, passing in the window frame and the song queue ❻. You then set the external view as the root view controller for the external window ❼. If `externalView` is not `nil`, you just update the song queue ❽ and update its views to reflect the latest song in the queue ❾. Lastly, you set the window as not hidden ❿.

One final thing you should do is update your external view whenever a new song is played. Add the following to the bottom of the `playMusic:` action to manually update the song queue and the views on the external view controller:

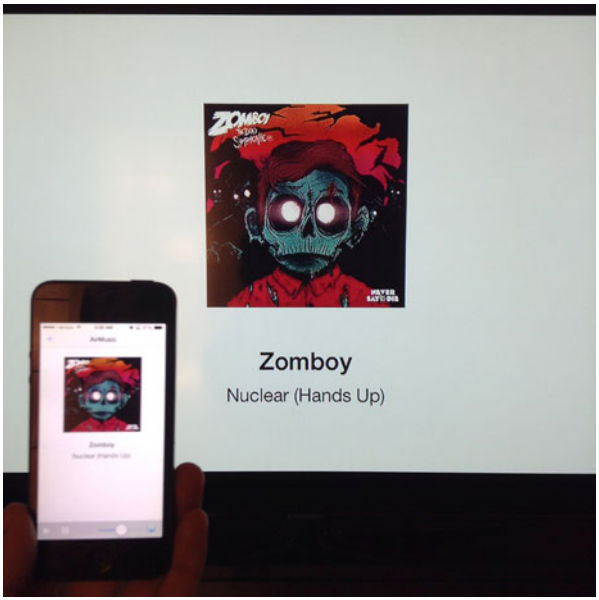


Figure 12.12 With AirPlay mirroring enabled, our app now displays custom content on an external screen.

```
if (self.externalView)
{
    self.externalView.songQueue = self.songQueue;
    [self.externalView populateViewsWithSongQueue];
}
```

Great job! You can now try it out by running the app on your device and then turning on AirPlay mirroring mode within the control center. Once mirroring is enabled, you should see your external screen updated with your custom view in `IAExternalViewController`, as shown in figure 12.12.

Also, the view should update whenever you play a new song. If you close the app or turn off AirPlay, the external screen should also reset. Great job—you've created your own mini music player that can also stream audio and display custom content on an external screen using AirPlay.

12.4 Summary

AirPlay provides an extremely simple way to stream audio, photos, and videos and even create an external display using AirPlay-supported devices, such as the Apple TV. You learned that the AV Foundation and MediaPlayer frameworks provide an easy way to stream audio and video using AirPlay. To see just how easy it is to start adding AirPlay support to your apps, you created your own simple AirPlay-powered music player. Finally, you learned about using AirPlay to display views on an external display.

- AirPlay allows you to stream audio wirelessly to any AirPlay-enabled device.
- Streaming video is currently possible with only a few AirPlay devices, such as the Apple TV.

- You can test AirPlay only on real devices and not on the Simulator because the iOS Simulator doesn't support it.
- The `AVPlayer` and `MPMoviePlayerViewController` have support for enabling or disabling AirPlay output.
- Adding an `MPVolumeView` allows you to control the system volume and route the audio to AirPlay-supported devices.
- External screens over AirPlay are available only when AirPlay is in mirroring mode.
- Utilizing external screens over AirPlay can enrich your user's experience.
- It's best to create views for external screens programmatically because of unknown sizes and Xcode limitations.
- By adding observers to the notification center, you can watch for new screens.

13 Integrating push notifications

This chapter covers

- Apple's Push Notification service
- Configuring your app to send and receive push notifications
- Sending remote push notifications
- Registering and scheduling local notifications

In iOS, applications are not allowed to perform continuous operations in the background. But what if something interesting does happen and you want to let your users know about that even if the app is closed? Say you create an application for sending short messages to other people. Chances are the user is not using your application when a friend asks him to have some margaritas by the beach.

Push notifications are the solution for these issues; they're a great way for apps to interact with users at any time. When an event of interest occurs, you send a specially crafted message to a user's device, and they'll see a message and hear an alert. This is similar to what happens when you get an SMS or a new email on your iPhone.

In this chapter you'll learn how push notifications take place. You'll learn how to configure your app to accept push notifications and how to send them. At the end of the chapter you'll have the tools to interact with your users, even when they

aren't using your application. You'll create a simple application called SaleAlerts, which alerts the user to new offers using push notifications, as shown in figure 13.1.

Figure 13.1 Push notification coming to our SaleAlerts application while the app is closed

13.1 Apple's Push Notification service

To understand Apple's Push Notification service (APNs), you first need to understand the basic concepts of how push notifications work.

In chapter 9 you learned how to establish communication with remote computers. Those communications were always in the form of making a request and receiving a response. In that kind of communication, there's always a one-to-one relationship between requests and responses. In other words, you get only what you ask for, when you ask for it. When doing push notifications, the app doesn't make requests. It subscribes to certain events, and those events are pushed to the app whenever they occur. A neat analogy to help you understand this mechanism is the traditional mail system. Letters arrive at your house whenever someone sends you something; you don't have to go to the post office every hour asking if there is something new. Letters will arrive even if you are not at home, and you'll eventually get them once you're there. Push notifications work the same way. Instead of doing a request asking for new messages, emails, offers, or whatever your app does, the app subscribes to a push notification service, and when something new is available, it'll get a notification. Figure 13.2 shows the difference between a traditional request/response communication and push notifications.

The Apple Push Notification service is the centerpiece of push notifications. It's a service provided by Apple that lets you send messages to users. But how does it

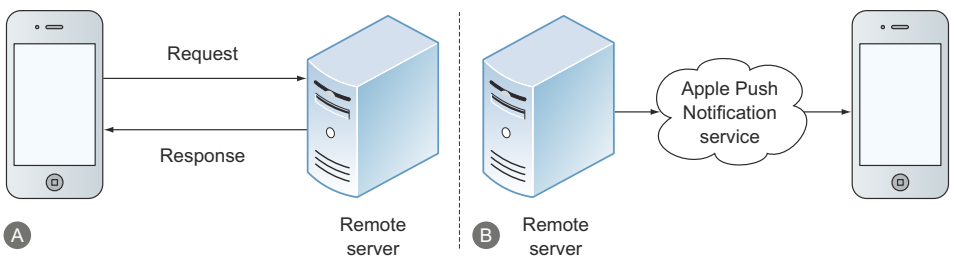


Figure 13.2 (A) Shows a traditional request/response communication between our app and a remote server; (B) shows how push notifications are originated from the server at any time.

work? When you turn on your iPhone, as soon as your device gets connected to the internet, iOS creates a persistent connection to the APNs, and this service assigns a unique identifier to the device. That's done automatically in the background by the operating system.

To be able to send push notifications from your application, you need to ask for push permission from the user. Once the permission is granted, the app is provided with a device token, which is an alphanumeric string that uniquely identifies the device. The device token is your way to identify users' devices and be able to send notifications. You can think of a device token as a street address. It's the address to which the notification will be directed.

To be able to send notifications you need to create some logic outside your iOS app. In short, you'll need to set up a computer with internet connectivity that will store device tokens of your users and, when needed, connect to the APNs and send push notifications. Later in the chapter we'll show some examples for the server side of the app. Those examples will be written in the Python language. We chose Python because its simple, unintimidating syntax will allows you to understand the concept even if you've never seen Python before. Figure 13.3 shows the device token flow. Steps 1 and 2 are done by the operating system, and steps 3 and 4 are controlled by your application.

Once the device token is provided and the user has granted you permission to receive push notifications, you're ready to send notifications. The push notification flow starts from your program running on the server side. It sends a text message along with a sound, an icon, and one or more destinations to the APNs (you'll see how to create these messages later on). The APNs validates that your app is allowed to send push notifications, and it redirects the message to the user device, which will show the text message on the screen and emit a sound. When the user touches the alert, your application opens. Figure 13.4 shows the push notification flow.

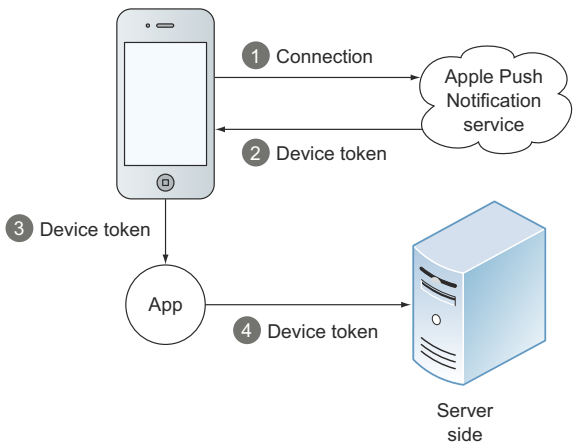


Figure 13.3 Device token flow: (1) A connection to the APNs is established by the operating system; (2) a unique device token is assigned; (3) when the app is granted push notification permission by the user, the device token is provided to the app; (4) you send the device token to your servers in order to be able to send notifications to that user later on.

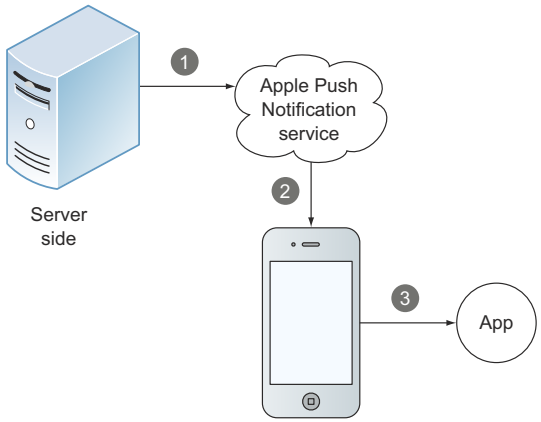


Figure 13.4 Push notification flow: (1) Our program on the server sends a request to the APNs; (2) if we have permission to send notifications, the APNs sends a push notification to the device; (3) when the notification is touched, the app opens.

13.2 Configuring your app to send and receive push notifications

Let's get down to work. First, you're going to create the application you'll use in this chapter. For that, you'll open Xcode and create a new, single-view application named SaleAlerts. As you did in the previous chapters, you'll set the class prefix to `IA`.

The interface of your application will be very simple so you can focus on the push notification side. It will consist of a `UILabel` in the middle of the main view holding a sale offer. Create your very simple interface by opening the Main.storyboard file in Interface Builder and dragging and dropping a `UILabel` object from the Object Library into the view controller, as shown in figure 13.5.

In order to support push notifications you must configure your application in the iOS Dev Center. To do that, open the following web page in your preferred browser: https://developer.apple.com/devcenter/ios/index.action. Now sign in to your account and click Certificates, Identifiers & Profiles.

The first thing you need to add from the Dev Center is an application ID. App IDs are strings used to uniquely identify applications. The format for app IDs is

ABCDEF123.com.manning.ios.SaleAlerts

The first part (ABCDEF123) is your team ID and the rest is the bundle ID. Note that the bundle ID part of your app ID must match the bundle identifier of your app. Bundle identifiers are automatically added when you create a project. Each time you created a project previously, you set the Company Identifier field to com.manning.ios. Then Xcode set a bundle identifier for you in the form of com.manning.ios.<Project Name>; for example, the application we're creating in this chapter will have com.manning.ios .SaleAlerts as its bundle identifier.

Go ahead and create the app ID by clicking Identifiers, selecting App IDs on the left panel of the iOS Dev Center, and then clicking the + button, as shown in figure 13.6.

Now you'll see a form for registering app IDs. Set the Name field of your app to SaleAlerts and choose Explicit App ID. Then set the bundle ID to any string you want;

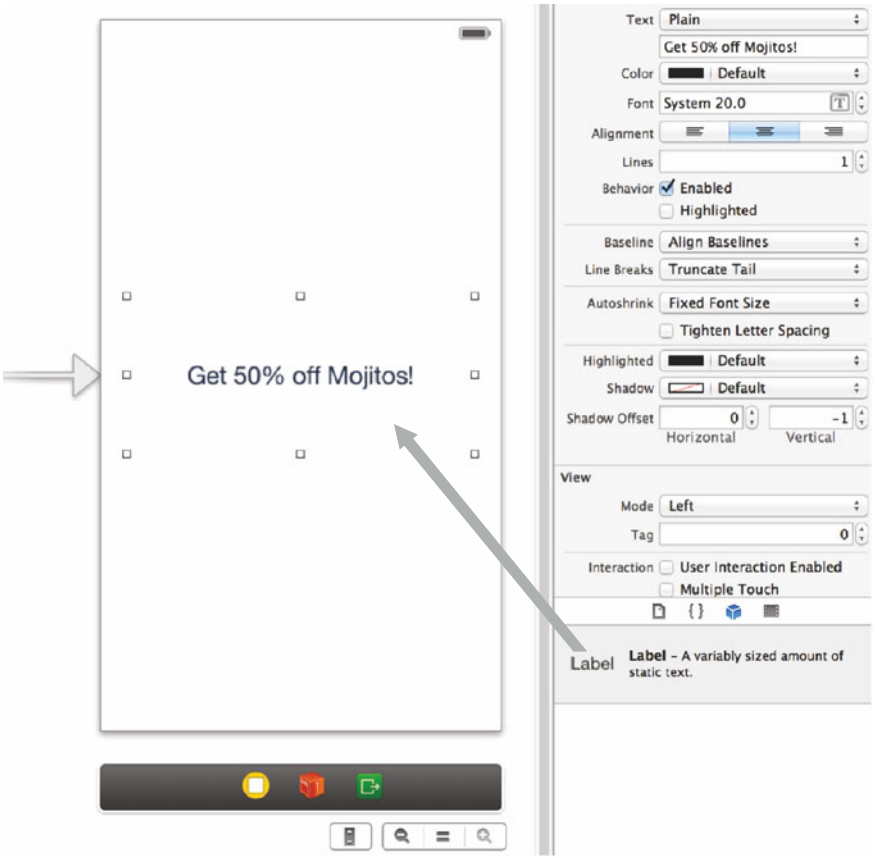


Figure 13.5 Our interface consists of only one `UILabel` in the middle.

we're setting ours to com.manning.SaleAlerts. Make sure to choose Push Notifications in the App Services section, as shown in figure 13.7. That will allow you to send (and receive) push notifications.

If you click App IDs in the Identifiers section again and select SaleAlerts App ID (the one you just created), you'll see that push notifications are still not enabled. In order to enable them you must click Edit and then create an SSL certificate. SSL certificates are used to establish remote secure communications. In the case of push notifications, the communication between your server and the APNs will be encrypted using this SSL certificate. Go ahead and create a development SSL certificate by clicking Create Certificate and following the step-by-step guide in the portal. Remember where you save the certificate files because you're going to use them later on.

Now you need to create a provisioning profile. Provisioning profiles are files that contain code-signing information as well as the distribution mechanism. They also contain information about what devices the app can be deployed to (for testing). To create a provisioning profile, choose the Provisioning Profiles option on the left of the

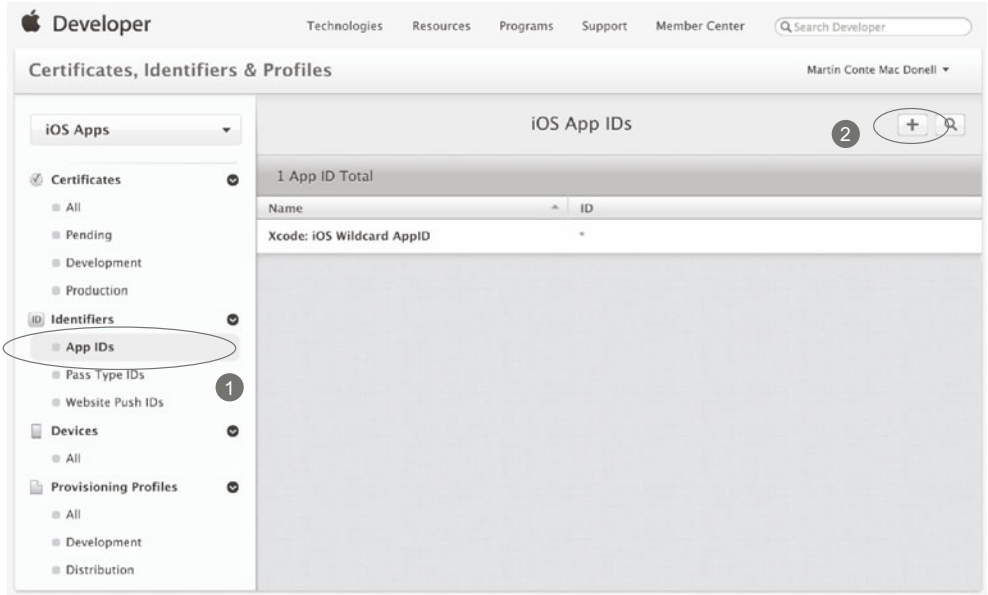


Figure 13.6 Create an app ID by (1) selecting App IDs from the menu and then (2) clicking the + button.

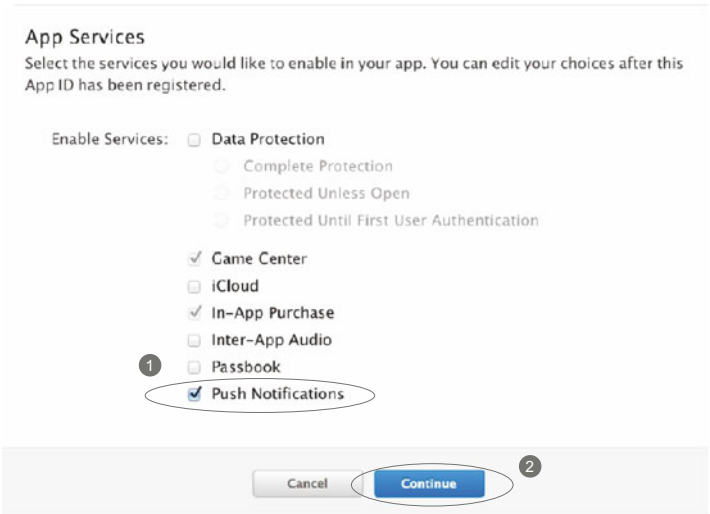


Figure 13.7 Enable push notifications for your app by (1) selecting Push Notifications and (2) clicking Continue.

iOS Dev Center and then click the + button. From the Add iOS Provisioning Profile screen, choose iOS App Development and then click Continue, as shown in figure 13.8. Then click Submit to confirm the profile creation. Next, you're going to choose your App ID, developer certificate, and device. Once that's done, you'll be able to download the provisioning profile and install it by dragging and dropping it onto the Xcode icon on the OS X dock bar.

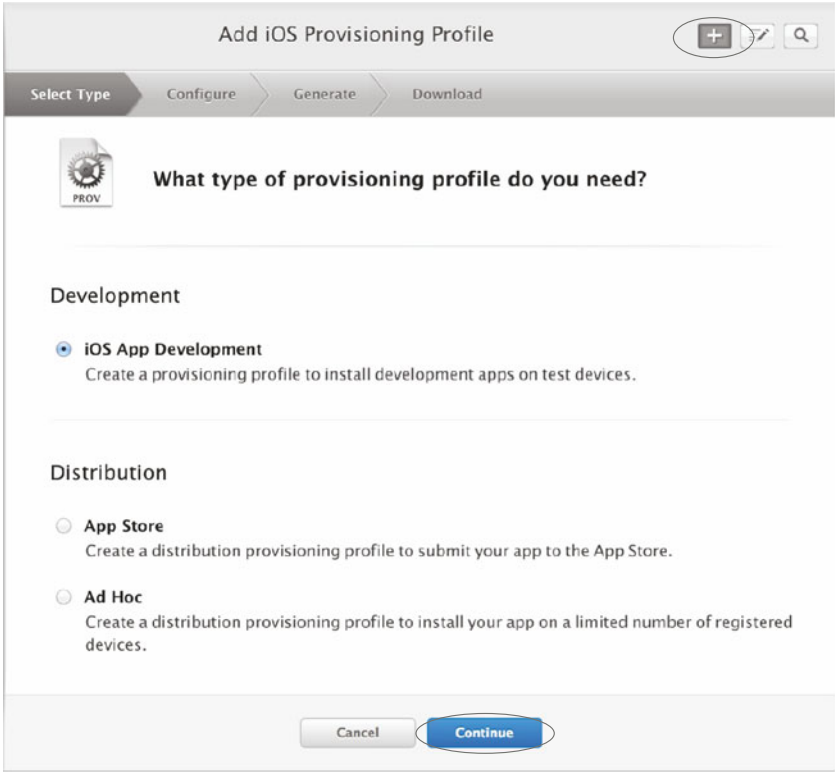


Figure 13.8 Create a new provisioning profile in the iOS Dev Center by clicking the + button and choosing the iOS App Development option.

With your App ID created and the provisioning profile associated with it, you've completed your application configuration. In the next section you'll write the logic to send push notifications from the server.

13.3 Sending push notifications

Before being able to receive notifications, you need to ask the user for permission. Once the user grants you the permission to receive push notifications on the device, iOS will provide you the token id of the device. When asking for permission you need to specify what type of push notifications you're going to implement. Notification types are badge, sound, and alert. These are `UIRemoteNotificationTypeBadge`, `UIRemoteNotificationTypeSound`, and `UIRemoteNotificationTypeAlert`, respectively:

- `UIRemoteNotificationTypeBadge`—This kind of notification adds a red circle to your application icon along with a number. You've probably seen this notification when, for example, receiving an email.
- `UIRemoteNotificationTypeSound`—This notification type plays a sound file when the user receives a push notification.

- `UIRemoteNotificationTypeAlert`—This notification type displays a text alert box with a custom message. This is the most common type of notification.

Notification types can be combined, meaning that you can send push notifications including a badge change, a sound, and a message.

Asking the user for permission is very straightforward; for that you'll open the IAAppDelegate file and create the method shown in the following listing.

Listing 13.1 Asking permission for receiving push notifications

```
- (void)applicationDidFinishLaunching:(UIApplication *)application
{
    [[UIApplication sharedApplication] registerForRemoteNotificationTypes:
➥   (UIRemoteNotificationTypeSound | UIRemoteNotificationTypeAlert)];
}
```

Asking the user for permission for receiving notifications including sounds and alerts

Do you remember this method? It's the one that iOS calls right after your application loads. There you're asking the user for permission to receive push notifications. Right after the app is launched, the user will see an alert similar to the one in figure 13.9.

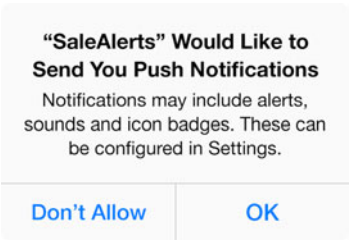


Figure 13.9 Users are asked if they would like to receive push notifications.

When the user allows you to send push notifications, iOS will call the following method: `application:didRegisterForRemoteNotificationsWithDeviceToken:`. As you can see by the name of the method, one of the parameters is the token ID of the device. We said before that this token is comparable to addresses and as such you'll need to store them on the server side for sending notifications in the future. This chapter won't cover how to persist the tokens on the server. Such operations are beyond the scope of the book, but we'll show some examples of how to send push notifications even when it's not done using Objective-C.

We're going to show how to send a push notification using the Python language. We'll cover the entire process step by step, so don't worry if there's something in the code that you don't immediately understand. The first step is shown in the following listing.

Listing 13.2 Step 1: Preparing the payload

```
import socket, ssl, struct

token = 'a4b8e18bd065c6ea7def614631ee4abaf26dbd56cde307e596fbf838fbb03296'
payload = '{"aps": {"alert": {"body": "Get 50% off Mojitos!"}}}'
```

Importing the packages to use in the push example

The push payload represented as JSON

Token example (that's exactly what iOS gives you when the user grants permission)

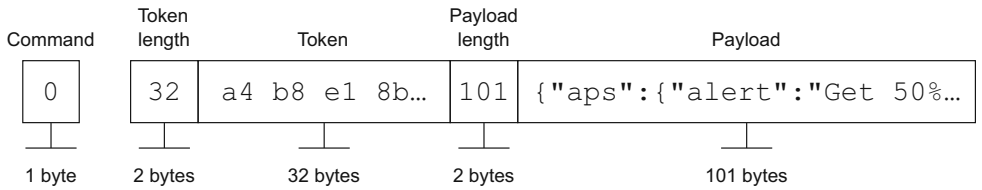


Figure 13.10 Binary format of the push notification binary package

Listing 13.2 shows the first step of the push notification script. This step only defines two variables. One is `token`, which contains an example token ID, and the other one is `payload`, containing your push notification payload. Did you find the payload's format familiar? It's JSON, a normalization format you learned in chapter 6. The JSON payload is defined as follows:

```
{
  "aps": {
    "alert": {
      "body": <Alert message>,
      "launch-image": <image filename used as the launch image>
    },
    "badge": <Badge number>,
    "sound": <Sound filename, played when the notification arrives>
  }
}
```

Note that not all keys are mandatory; as a matter of fact, in listing 13.2 you're using only the alert key.

In order to send push notifications to the APNs you need to use a specific format. The payload will be formatted as JSON, but the payload is only one part. The other part is the destination of the push notification, in other words, the token ID. Apple expects you to send push notifications through the APNs by sending the token ID and the payload, as shown in figure 13.10.

You need to create a binary package in which the bytes are arranged in a specific way. The first byte is the command (for simple push notifications you'll use 0) that defines the format of the package. The next two bytes are the token ID length (this will always be 32) followed by the token ID itself. Then you include the payload length followed by the payload itself formatted as JSON.

Now create the package in your script using the variables you prepared in listing 13.2, as shown in the next listing.

Listing 13.3 Step 2: Creating the binary package

```
byteToken = token.decode('hex')
theNotification = struct.pack('!BH32sH', 0, 32, byteToken, len(payload))
theNotification += payload
```

Decoding your token id string to binary (assuming that the token variable is a hex string)

Creating the binary package according to the previously mentioned format

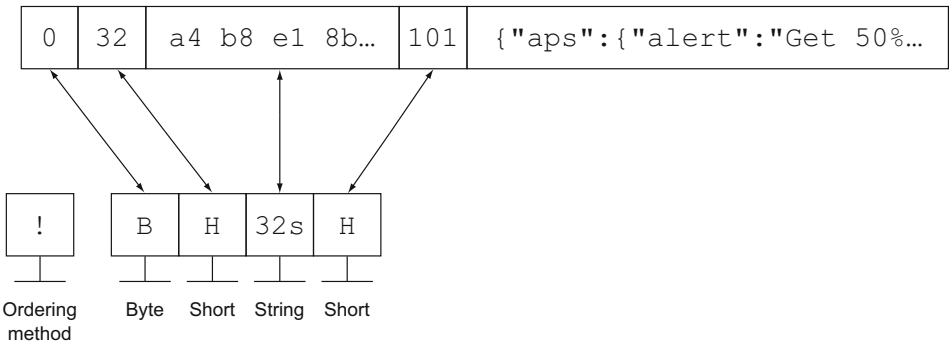


Figure 13.11 Link between a push notification package and the first argument of the `struct.pack` function

Here you create the binary package using a function called `pack`, which exists in most programming languages. This function is used to create binary packages according to a given format, which is the first argument of the function. The format determines how much space each element of your binary package will use. Figure 13.11 shows how !BH32sH is related to the binary structure shown previously.

The ! symbol at the very beginning is our way of telling the `struct.pack` function how to order bytes. Each letter after the symbol represents the space that each part of our package will use. Byte (B) is 1 byte and it will hold the command; short (H) is 2 bytes that will contain the token length. Then 32s is a string of 32 characters holding the token. Using the format and the arguments you pass to the function, you'll get a binary package that you can send to the APNs. In order to do that, you first need to create a connection to the APNs, which is what you're going to do in the next listing.

Listing 13.4 Step 3: Sending the package to the APNs

```
connection = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ssl_sock = ssl.wrap_socket(connection, certfile='yourCertificate.pem')
ssl_sock.connect(('gateway.sandbox.push.apple.com', 2195))
ssl_sock.write(theNotification)
ssl_sock.close()
```

Connecting the socket to the testing APNs ❷

Creating a secure socket by using your certificate provided by Apple and your private key ❶

Sending your previously crafted binary package ❸

Closing the connection once you're finished ❹

Using the certificate that you downloaded from the iOS Dev Center and your private key, you're going to create a secure communication to the APNs using SSL as ❶ and ❷ show. Once the connection is established, you can send as many push notifications as you want. In this case we're sending only one package that we previously crafted using the correct formatting. After you've finished sending push notifications ❸, you end the communication by closing the socket ❹.

If everything goes right, once you run the script you'll get a push notification in your device, similar to figure 13.1. When the user touches the notification (or slides it if they're on the lock screen), your app will open, and you'll have access to the entire payload by implementing the method `application:didReceiveRemoteNotification:` on the app delegate, as you'll do in the following listing.

Listing 13.5 Receiving a push notification from your app delegate

Getting the main view controller from the app delegate

```
- (void)application:(UIApplication *)application
     didReceiveRemoteNotification:(NSDictionary *)userInfo
{
    IAViewController *viewController = (IAViewController *)[[self window]
➥   rootViewController];
    NSDictionary *alert = userInfo[@"aps"][@"alert"];
    [[viewController offerLabel] setText:alert[@"body"]];
}
```

Getting the message from the push notification payload and showing it on the screen ❶

The JSON string you sent from your server is deserialized the same way you learned in chapter 6, meaning that instead of receiving a JSON string, you'll receive an `NSDictionary` with the same keys you included before. Note that now that the object is deserialized, you can access the values very easily ❶.

The last line sets the text of your main view controller label using the push notification message you just received. To be able to access the view controller you need to include `IAViewController` in your `IAAppDelegate`. For that, you'll go to the very top of your IAAppDelegate.m file and add this line:

```
#import "IAViewController.h"
```

Next, you'll link your label from Interface Builder to the `IAViewController` instance. This time, instead of linking it to the .m file as you did in the previous chapters, you'll link it to the .h file. The reason you're doing this is that you need this variable to be public to be able to access it from the app delegate. First, open the assistant editor from the Interface Builder by choosing View > Assistant Editor > Show Assistant Editor in the application menu bar. With the offer label selected in your interface, hold down the Control key while clicking and dragging from the label to your `IAViewController`'s class definition in the assistant editor, as shown in figure 13.12. Once you've finished dragging and let go, a modal will appear asking you to name the outlet you're setting on your class for this label. Name it `offerLabel` and then click Connect.

That's it! The next time you click a push notification, you'll see the offer on the screen.

13.4 Registering and scheduling local notifications

So far you've learned how to send and receive remote notifications when some new content is available, but remote notifications aren't the only type. There's another kind of notification called local notifications. Local notifications are visually the same as push notifications; they both indicate to the user that something interesting happened

Figure 13.12 Linking the offer label to the `IAViewController` public definition

in your app by showing a message, a sound, and/or an icon. But local notifications serve different needs: these kinds of notifications are strictly related to a date and a time and are scheduled beforehand. They're used for reminders, schedules, expirations, and the like. You don't need any interaction server-side in order to schedule local notifications.

Now you'll implement a reminder in your SaleAlerts application that will help the user remember that a new offer is available by showing an alert 10 minutes after a push notification is received. Add the following code into the `application:didReceiveRemoteNotification:` method you created in listing 13.5.

Listing 13.6 Scheduling a local notification

```
UILocalNotification *notification = [[UILocalNotification alloc] init];
notification.fireDate = [NSDate dateWithTimeIntervalSinceNow:600];
notification.alertBody = alert[@"body"];
[[UIApplication sharedApplication] scheduleLocalNotification:
➥ notification];
```

❶ **The local notification will be fired after 10 minutes (600 seconds).**

❷ **Setting the text that the user will see using the previously received push notification message**

❸ **Scheduling the notification you just created**

You first create a `UILocalNotification` instance, and then you set the message ❷ and the fire date ❶. Note that the fire date is expressed in seconds. Line ❸ schedules the notification you just created.

Similar to remote push notifications, when a local notification is touched and your app opens, iOS calls a function including the notification payload. The method called when a local notification arrives is `application:didReceiveLocalNotification:`, and the parameters are the same as in the remote notifications scenario you implemented previously.

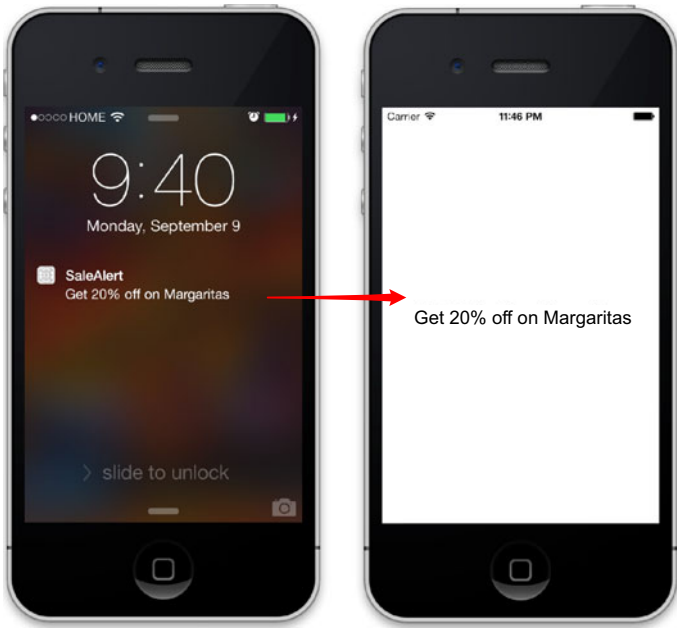


Figure 13.13 Touching (or sliding) a push notification opens your app, which will show the offer.

To summarize, you send a push notification from the server to the device including a message regarding an offer. The device receives the notification, and when the user opens your app by touching (or sliding) the notification, you show the offer on the screen and at the same time you schedule a local notification that will be shown after 10 minutes reminding the user about the offer, as shown in figure 13.13.

13.5 Summary

Throughout this chapter, you learned how push notifications are sent and received, helping you to create a communication channel with your users regardless of what application they're using at the moment the notification arrives. You learned the complete route of push notifications, starting from a message sent from a remote server, going through the Apple Push Notification service, and finally reaching the user's device. Here are some key topics we covered:

- When you enable push notifications, events are pushed to the app whenever they occur through APNs.
- In order to be able to send push notifications to a specific user, you have to ask the user for permission.
- Push notifications are sent from a server to your iPhone application.
- The format you use to create push notifications is JSON.
- To send and receive push notifications, you need to configure your certificates and application's parameters in the iOS Dev Center.

14 Applying motion effects and dynamics

This chapter covers

- Motion effects using `UIMotionEffects`
- Adding the parallax effect
- Realistic animations with UIKit Dynamics
- Simulating gravity, collisions, and elasticity
- Creating custom behaviors

With iOS 7 came flat textures devoid of gradients and out went skeuomorphic design that mimicked real-life physical objects. There was also the addition of parallax, which made interface objects appear to be three-dimensional by altering their position ever so slightly depending on the angle at which you're holding your device. This parallax effect and many others can be achieved by using the new motion APIs in UIKit. Also, before iOS 7 you needed to dive into complex math and physics if you wanted to create realistic physics effects in your views. Now there's also a whole new slew of APIs in UIKit Dynamics that you can use to create these realistic effects without having to be a mathematician. You'll learn about both motion and dynamics in this chapter. Together we'll build a fun little app that will serve as a catalog to showcase a few of the great things you can now do.

14.1 Creating your application

The app you'll prepare for this chapter will be rather quick and simple. It will involve two images, a basketball court floor for the background and an image of a basketball, which we'll use to demonstrate motion and dynamics. Open Xcode and create a new single-view application called Motion Ball.

Next, you'll need to download an archive that contains four images for the background and the basketball you'll use in the app. There are two versions of each image, a retina and non-retina version. Open your web browser and go to http://blim.co/HEbROX to download the image archive. Once it's downloaded, open the archive and you'll see the following files:

- basketball.png
- basketball@2x.png
- background.png
- background@2x.png

Jump back into Xcode and select your image assets in the project navigator. Next, drag all four files into the window that displays all of your image assets. You should see them added to your project, as shown in figure 14.1.

Next, open your storyboard and add a `UIImageView` that fills the entire view, and set the image to background, as shown in figure 14.2.

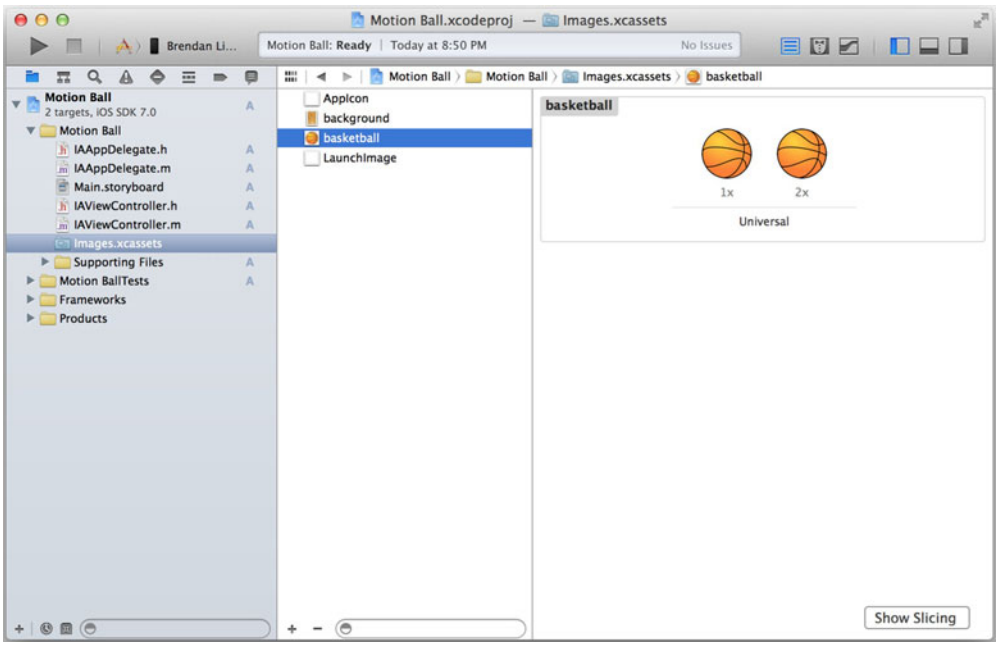


Figure 14.1 Add the images to your image assets in your Xcode project.

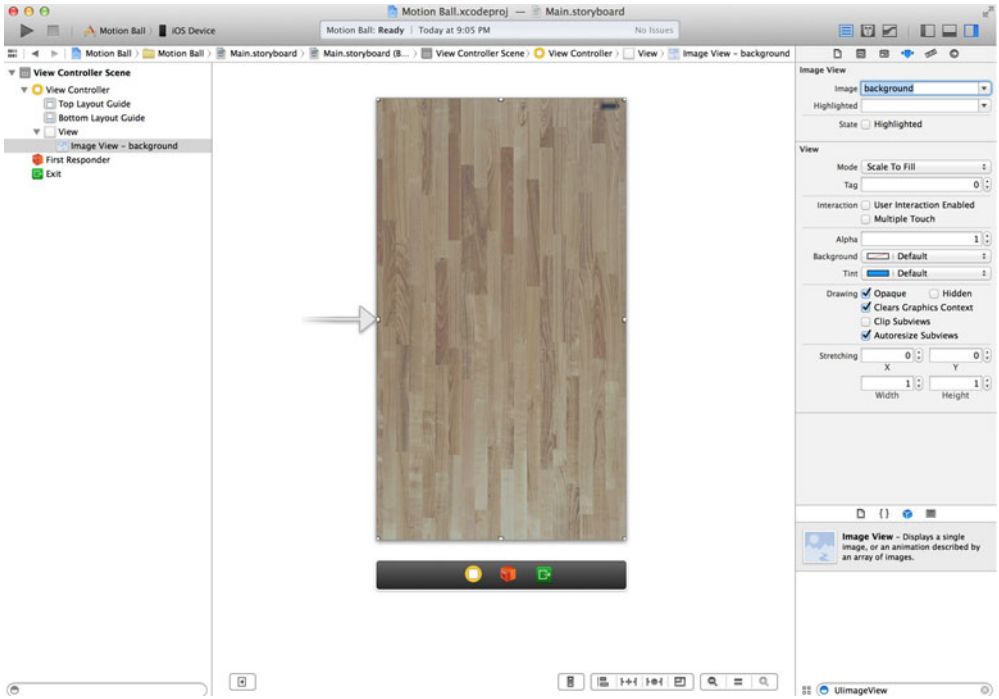


Figure 14.2 Add a `UIImageView` to show the background image you added to the image assets.

The last view you'll need to add is another `UIImageView` that's positioned in the center of the screen. Set its size to 150 x 150 and the image to basketball, as shown in figure 14.3.

Once the basketball's been added, create an outlet for it in IAViewController.h called basketball, as shown in figure 14.4. That's all the setup you'll need to do for your app. Let's now take a look at motion effects.

14.2 Using motion effects

Motion effects are used to add effects to views in your application based on the motion of the device running the application. This is accomplished by applying a motion effect to a specific view. These effects can be applied for horizontal and vertical movement when an iOS device is tilted. This also means that you can only see these results on a real device because you can't simulate this in the iOS Simulator. You'll see how to add a parallax effect to your application.

14.2.1 Adding the parallax effect

The parallax effect utilizes the accelerometer and gyroscope data to determine how a single axis on a view should be adjusted when a device is tilted horizontally or vertically. It'd be easy to show you what the parallax effect looks like if we were able to show you an animated GIF within this book. Instead, look at figure 14.5, which shows you what the parallax effect looks like when you tilt a device horizontally or vertically.

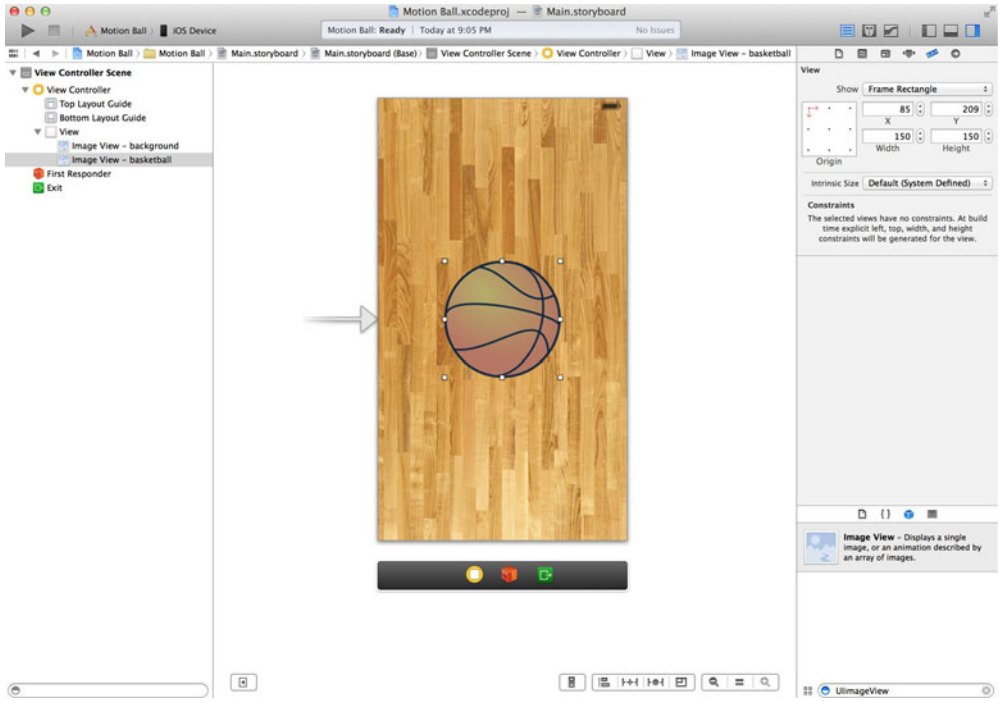


Figure 14.3 Add another image view to the center of the screen to represent the basketball.

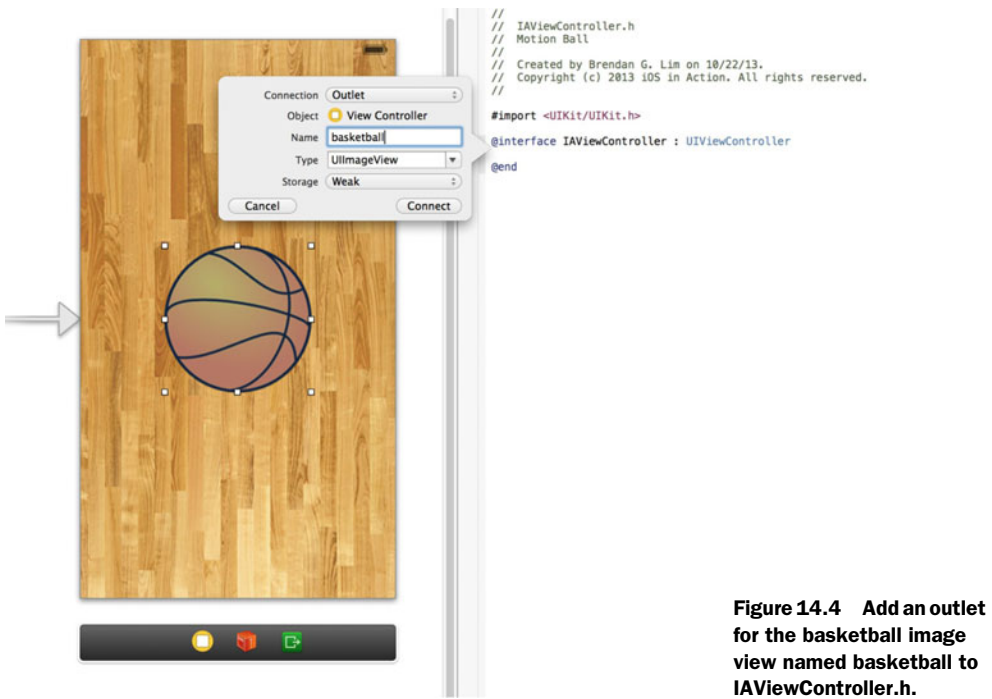


Figure 14.4 Add an outlet for the basketball image view named basketball to IAViewController.h.

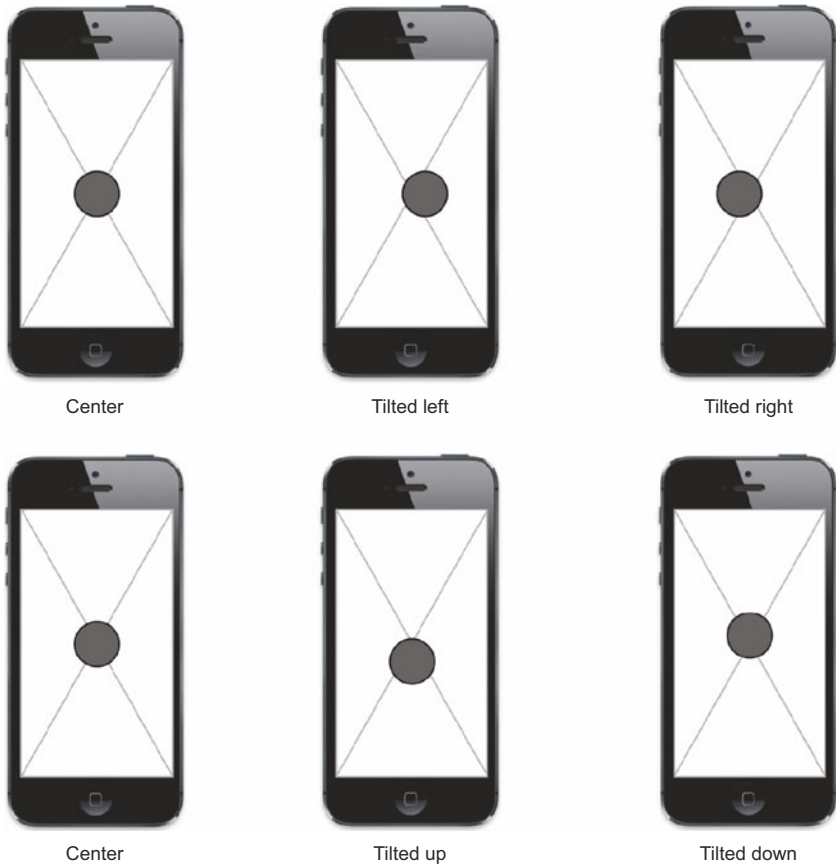


Figure 14.5 Demonstrating the parallax effect on a circle, which changes location when the device is tilted horizontally or vertically

In this figure the motion effect is applied to the dark gray circle. The background is used as a reference point to demonstrate the way the ball is positioned as you move the device to the left/right or up/down.

To create these effects there's a new class in the UIKit framework called `UIMotionEffect`, which serves as an abstract class. This means you can't use this class directly to create your own motion effects, but you can subclass it. Luckily there's an out-of-the-box class called `UIInterpolatingMotionEffect` that you can use to create the parallax effect.

To create a `UIInterpolatingMotionEffect` you can use the `initWithKeyPath:type:` method. The first parameter, the key path, is used to specify on which axis on the view you'd like to apply the effect. Normally this is done on the center point of the view. For instance, if you were to apply a horizontal motion effect, you'd apply it to the center.x key path. For a vertical effect you'd use center.y. The second parameter in the `initWithKeyPath:type:` method specifies the type of motion to track,

which is represented by two constants, `UIInterpolatingMotionEffectTypeTiltAlongHorizontalAxis` for horizontal tracking and `UIInterpolatingMotionEffectTypeTiltVerticalAxis` for vertical.

Once a `UIInterpolatingMotionEffect` instance is created, you can add a maximum value and a minimum value that are used to specify the amount a view should move when a device is tilted horizontally all the way to the left or to the right. This is accomplished by specifying an `NSNumber` for the `maximumRelativeValue` and `minimumRelativeValue` properties.

Add the parallax effect to your app so that the basketball's position changes when the device is tilted horizontally or vertically. Jump into IAViewController.m and add the following method.

Listing 14.1 Add parallax effect to basketball

```
- (void) addParallaxEffect
{
    UIInterpolatingMotionEffect *horizontalEffect =
➥ [[UIInterpolatingMotionEffect alloc] initWithKeyPath:@"center.x"
➥ type:UIInterpolatingMotionEffectTypeTiltAlongHorizontalAxis];

    UIInterpolatingMotionEffect *verticalEffect =
➥ [[UIInterpolatingMotionEffect alloc] initWithKeyPath:@"center.y"
➥ type:UIInterpolatingMotionEffectTypeTiltAlongVerticalAxis];

    verticalEffect.maximumRelativeValue = @(20);
    verticalEffect.minimumRelativeValue = @(-20);

    horizontalEffect.maximumRelativeValue = @(20);
    horizontalEffect.minimumRelativeValue = @(-20);

    [self.basketball addMotionEffect:verticalEffect];
    [self.basketball addMotionEffect:horizontalEffect];
}
```

1 Create vertical motion effect.
2 Create horizontal motion effect.
3 Set vertical maximum relative value.
4 Set vertical minimum relative value.
5 Set horizontal maximum relative value.
6 Set horizontal minimum relative value.
7 Add vertical motion effect to basketball.
8 Add horizontal motion effect to basketball.

Here you first create a vertical effect using the key path center.y so that the vertical effect is applied to the y-axis ❶. You then create a horizontal effect on the x-axis ❷. Next, you set the maximum ❸ and minimum ❹ values to 20 points for the vertical effect. Then you set the same for the maximum ❺ and minimum ❻ on the horizontal effect. Lastly you add the motion effect for vertical ❼ and horizontal ❽ to the `UIImageView` that represents your basketball.

The last thing to do is to make a call to this method within the `viewDidLoad` method:

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

You can try this out only if you have a paid developer account because you'll need to run this application on your device to see it in action. When you do run this, you

should see the basketball showcasing the parallax effect when the device is moved vertically or horizontally. Next, you'll be learning about UIKit Dynamics so that you can apply realistic animations to your basketball.

14.3 Using UIKit Dynamics

UIKit Dynamics gives you a way to animate views to produce realistic effects. Previously in iOS 6, in order to create these types of animations you'd need a deep understanding of math, physics, and Core Animation. UIKit Dynamics now allows you to add these types of effects to your apps.

14.3.1 Introduction to UIKit Dynamics

As is probably apparent from its name, UIKit Dynamics is part of the UIKit framework. The top-level class with the physics engine that's responsible for generating the effects you'll be using is the `UIDynamicAnimator` class. This class will accept *behaviors* that are applied to a specific view. Behaviors provide instructions to the animator's physics engine, which in turn applies the effects. These behaviors are represented by the `UIDynamicBehavior` class. Each `UIDynamicBehavior` can be applied to multiple `UIDynamicItems`. What's a `UIDynamicItem`? The `UIDynamicItem` is a protocol that defines a center, bounds, and a two-dimensional transform. Luckily the `UIView` class conforms to the `UIDynamicItem` protocol.

Take a look at figure 14.6 to see how these all relate to each other.

Also, the `UICollectionViewLayoutAttributes` class (you learned about this when reading about collection views) can be used with dynamic behaviors. In this chapter we'll be sticking to applying effects to `UIViews`.

When creating a new `UIDynamicAnimator` instance you'll have to pass in a reference view using the `initWithReferenceView:` method. This reference view will be the top-level view in your view controller, as shown here:

```
UIDynamicAnimator *animator = [[UIDynamicAnimator alloc]
➥ initWithReferenceView:self.view];
```

When using a `UIDynamicBehavior` you can use one of the out-of-the-box subclasses or create your own. The behaviors already created for us will solve almost all of our

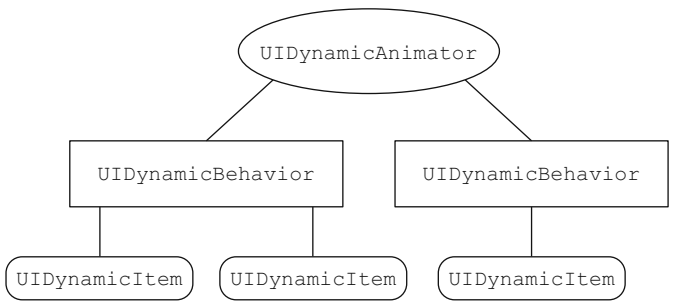


Figure 14.6 The `UIDynamicAnimator` takes in `UIDynamicBehaviors`, which can be applied to multiple objects that conform to the `UIDynamicItem` protocol.

needs, which is one of the reasons why we don't need to be rocket scientists to work with them. Each of them has its own specific behavior, as shown in table 14.1.

Table 14.1 Different types of `UIDynamicBehaviors`

Behavior	Description
`UIAttachmentBehavior`	Connection between two dynamic items
`UICollisionBehavior`	Collision between dynamic items
`UIGravityBehavior`	Gravity effect applied to dynamic items
`UIDynamicItemBehavior`	Effect to match ending velocity of a user gesture
`UIPushBehavior`	Apply force to a dynamic item from a push
`UISnapBehavior`	Snap a dynamic item to a specific point

How about you get started by adding some dynamic behaviors to your basketball? First, open Xcode and add a property to IAViewController.h called `animator`, as shown here:

```
@property (strong, nonatomic) UIDynamicAnimator *animator;
```

Now go to IAViewController.m and set the `animator` property in the `viewDidLoad` method using the `view` property as the reference view:

```
self.animator = [[UIDynamicAnimator alloc]
➥ initWithReferenceView:self.view];
```

Next, you should set the basketball to be able to interact with touch actions, because that's how you're going to be triggering these effects. Add the following to the bottom of `viewDidLoad` as well:

```
[self.basketball setUserInteractionEnabled:YES];
```

Great—you're ready to start adding some awesome behaviors.

14.3.2 Applying the gravity behavior

The gravity behavior is one of the simplest behaviors to use. The goal is to have the dynamic item, the basketball, fall with simulated gravity after it's tapped. To do this, you'll create two new methods, `setupGravity` and `dropBall:` in IAViewController.m. The `setupGravity` method is only to add a gesture recognizer to execute `dropBall:` when the ball is tapped. Add the following code:

```
- (void) setupGravity
{
    UITapGestureRecognizer *tapGesture = [[UITapGestureRecognizer alloc]
➥ initWithTarget:self
➥ action:@selector(dropBall:)];
    [self.basketball addGestureRecognizer:tapGesture];
}
```

Now add the following to the bottom of `viewDidLoad`:

```
[self setupGravity];
```

Finally, you can add the gravity behavior, which is executed when the basketball is tapped. This code is shown in the following listing.

Listing 14.2 Applying gravity to a view on tap

```
- (void) dropBall:(UITapGestureRecognizer *)recognizer
{
    UIGravityBehavior *gravity = [[UIGravityBehavior alloc]
➥ initWithItems:@[self.basketball]];
    [self.animator addBehavior:gravity];
}
```

❶ Create gravity behavior for basketball.

❷ Apply behavior to animator.

All of the magic is done in the two lines found in this method. You first create a new instance of `UIGravityBehavior` as the dynamic item ❶. Then you add the behavior to the animator ❷.

If you run the application now, you'll see the ball fall off the screen as soon as it's tapped. The start and end results are shown in figure 14.7.

It's amazing how this was accomplished with such a small amount of code. It actually took you more lines to set up the tap gesture than it did to simulate the gravity

Figure 14.7 After tapping, the ball will animate and fall off the screen as if gravity was pulling it down. After the animation, the ball will be off the screen.

effect. What if you wanted the ball to fall and come to a stop at the bottom of your view? You can do this by adding a collision behavior.

14.3.3 Applying a collision behavior

Collision behaviors allow you to define how dynamic items should react when they come in contact with one another. This is accomplished by using the `UICollisionBehavior` class. In the case of your basketball, when the gravity behavior is applied, it falls but never stops falling. You can get it to stop at the bottom of the screen by adding a collision behavior. Because the animator was created with knowledge of a reference view—the main view of your view controller—it knows where the imposed boundaries are. You can use this to your advantage in this situation.

The `UICollisionBehavior` class's `translatesReferenceBoundsIntoBoundary` property allows you to tell it to set the bounds of the reference view as the boundary of the behavior by setting it to `YES`. You'll add the collision behavior within the `dropBall:` method, as shown in the next listing.

Listing 14.3 Adding the collision behavior

```
- (void) dropBall:(UITapGestureRecognizer *)recognizer
{
    UIGravityBehavior *gravity = [[UIGravityBehavior alloc]
➥ initWithItems:@[self.basketball]];
    [self.animator addBehavior:gravity];

    UICollisionBehavior *collision = [[UICollisionBehavior alloc]
➥ initWithItems:@[self.basketball]];
    collision.translatesReferenceBoundsIntoBoundary = YES;
    [self.animator addBehavior:collision];
}
```

1 Create the collision behavior.

2 Set reference view bounds as boundary.

3 Apply behavior to animator.

Here you're adding three lines to add the collision behavior. First, you create a new `UICollisionBehavior` with the basketball view as the dynamic item ❶. Next, you set the bounds of the reference view as the boundary for the behavior ❷. Lastly, you add the behavior to the animator ❸.

Run the application and tap the basketball. You should see that the ball lightly bounces as soon as it hits the bottom boundary of the reference view. The start and finish positions are shown in figure 14.8.

One thing you also see in this example is that you can add multiple behaviors to an animator to make a unique effect. Next, you're going to learn how to make this image come to life by making it bounce like a real basketball.

14.3.4 Adding dynamic behavior

You're going to use the `UIDynamicBehavior` class to help with the bounce effect. The `UIDynamicBehavior` has a property that specifies the elasticity of an object that helps makes this possible. This property accepts a `CGFloat` that ranges from 0.0 (no bounce) to 1.0 (continuous elasticity).

Figure 14.8 After tapping it, the ball will fall with the same gravity behavior but will come to a light stop at the bottom of the reference view due to the collision behavior you added.

Also, a basketball never falls straight down without rotating. The `UIDynamicBehavior` class lets you specify angular and linear velocities for a specific dynamic item. You'll be adding an angular velocity by using the `addAngularVelocity:forItem:` method. This will be used to give the ball a little rotation when it collides with the boundaries of the reference view.

To simulate friction there's a `friction` property that takes values from 0.0 (no friction) to 1.0 (strong friction). Also, to give the object realistic density to simulate mass, there's even a `density` property that you can specify. The density specified is directly related to the pixel size of the dynamic item that it's applied to. For example, an object that is 100 x 100 with a 1.0 density value will accelerate to 100 points per second^2.

Update the `dropBall:` method as shown in the following listing.

Listing 14.4 Adding dynamic behavior

```
- (void) dropBall:(UITapGestureRecognizer *)recognizer
{
    UIGravityBehavior *gravity = [[UIGravityBehavior alloc]
➥ initWithItems:@[self.basketball]];
    [self.animator addBehavior:gravity];
```

```
    UICollisionBehavior *collision = [[UICollisionBehavior alloc]
➥ initWithItems:@[self.basketball]];
    collision.translatesReferenceBoundsIntoBoundary = YES;
    [self.animator addBehavior:collision];

    UIDynamicItemBehavior *bounce = [[UIDynamicItemBehavior alloc]
➥ initWithItems:@[self.basketball]];
    [bounce addAngularVelocity:0.2f forItem:self.basketball];
    bounce.elasticity = 0.8f;
    bounce.friction = 0.2f;
    bounce.density = 0.4f;
    [self.animator addBehavior:bounce];
}
```

❶ Create dynamic item behavior.

❷ Add angular velocity to basketball.

❸ Specify elasticity amount.

❹ Specify friction amount.

❺ Specify density of basketball.

❻ Add behavior to animator.

Here you're creating a new dynamic item behavior for the basketball as the dynamic item ❶. Then you're adding an angular velocity of 0.2 ❷, elasticity of 0.8 ❸, friction of 0.2 ❹, and density of 0.4 ❺. Lastly you're adding the behavior to the animator ❻.

When you run the app and tap the basketball, you'll notice that it rotates as it bounces and returns to its normal position as it comes to a stop at the bottom right of the screen. Figure 14.9 shows you how the ball ends up after the animator has finished.

You're encouraged to play with the values for angular velocity, elasticity, friction, and density to see just how they affect the animation.

Figure 14.9 After you tap the ball, the dynamic item behavior will cause the ball to rotate and fall based on the angular velocity, elasticity, friction, and density that you added.

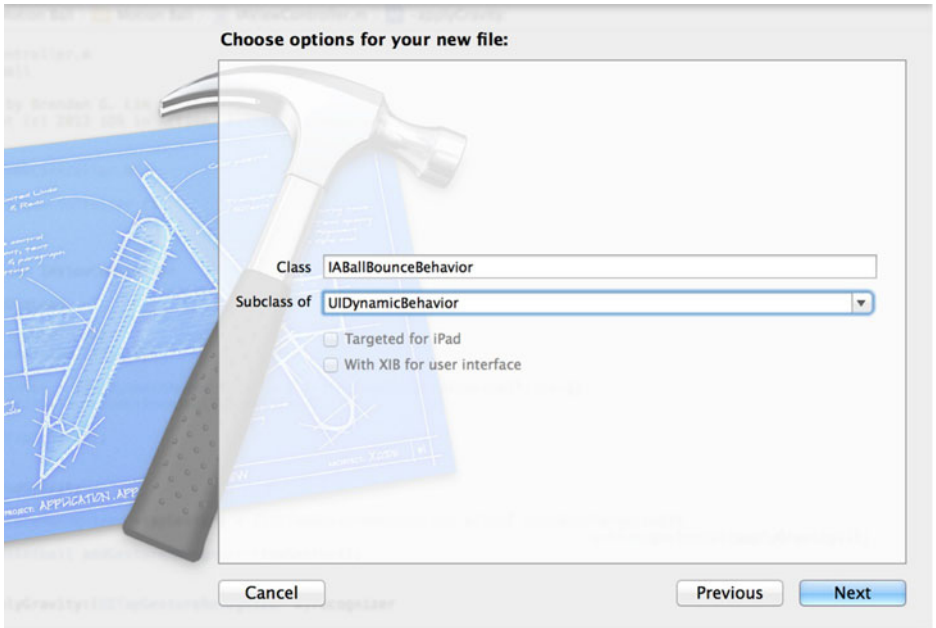


Figure 14.10 Create a subclass of `UIDynamicBehavior` called `IABallBounceBehavior`.

14.3.5 Creating a custom UIDynamicBehavior subclass

If you want to package all of these behaviors into one class, it's easy to do by creating a subclass of `UIDynamicBehavior`. First, go to the project navigator and create a new file in the Motion Ball group called IABallBounceBehavior that's a subclass of `UIDynamicBehavior`, as shown in figure 14.10.

After the new file's been created, open IABallBounceBehavior.h and add the following method declaration:

```
-(id) initWithItems:(NSArray *)items;
```

Next, open IABallBounceBehavior.m. Here you'll be adding all of the behaviors that you've just created, but instead of adding them to an animator, you'll be adding them as child behaviors. Add the code shown in the following listing.

Listing 14.5 Custom ball bounce behavior

```
-(id) initWithItems:(NSArray *)items
{
    if (self = [super init]) {
        UIGravityBehavior *gravity = [[UIGravityBehavior alloc]
➥ initWithItems:items];
        [self addChildBehavior:gravity];

        UICollisionBehavior *collision = [[UICollisionBehavior alloc]
➥ initWithItems:items];
```

1 Create gravity behavior.

2 Add gravity as child behavior.

3 Create collision behavior.

```
        collision.translatesReferenceBoundsIntoBoundary = YES;
        [self addChildBehavior:collision];

        UIDynamicItemBehavior *dynamic = [[UIDynamicItemBehavior alloc]
➥ initWithItems:items];
        for (UIView *item in items)
            [dynamic addAngularVelocity:0.2f forItem:item];

        dynamic.elasticity = 0.8f;
        dynamic.friction = 0.2f;
        dynamic.density = 0.4f;
        [self addChildBehavior:dynamic];
    }

    return self;
}
```

Add collision as child behavior. ❹

❺ **Create dynamic item behavior.**

❻ **Apply angular velocity to all items passed in.**

❼ **Add dynamic item behavior as child behavior.**

This should look extremely familiar. All you're doing is adding the custom behaviors you've already created into an initializer for the `IABallBounceBehavior` class. You first create the gravity behavior ❶ and add it as a child behavior ❷. Then you create the collision behavior ❸ and add that as a child behavior ❹. Lastly you create the dynamic item behavior ❺, apply velocity to all items in the items array ❻, and then add that as a child behavior ❼.

Next, open IAViewController.m. You can replace all of the dynamic behaviors with your new `IABallBounceBehavior`. First, you'll need to import the new class by adding the following to the top of the `IAViewController` class:

```
#import "IABallBounceBehavior.h"
```

Finally, you can replace all of the code within `dropBall:` with the two lines shown here:

```
IABallBounceBehavior *ballBounce = [[IABallBounceBehavior alloc]
➥ initWithItems:@[self.basketball]];
[self.animator addBehavior:ballBounce];
```

This shows how simple it is to wrap different dynamic behaviors to achieve a desired effect in one single behavior.

14.4 Summary

There are so many things you can do with motion effects and dynamics, and you've only just skimmed the surface. There literally are endless possibilities, even with dynamics alone. On top of using both of these, you can also combine Core Animation and gestures to make something truly unique. All of these together, if used wisely, can simulate realistic and fun animations that can help provide a level of delight for your users as they use your applications.

- You can use motion effects to provide a parallax effect to your views.
- The parallax effect works by using data from the accelerometer and gyroscope when a device is tilted horizontally or vertically.
- To be able to test motion effects, you need to run the applications on a physical device.

- UIKit Dynamics provides realistic animations without the need of a complex understanding of math and physics.
- Many dynamic behaviors can be used out of the box, such as gravity, collisions, and more.
- You can add multiple dynamic behaviors as child behaviors into a single `UIKitDynamicBehavior` subclass.

Appendix

The appendix covers

- Introduction to Objective-C
- Using blocks
- Optimizing applications with Grand Central Dispatch
- Understanding automatic reference counting

When you write applications, all you're doing is providing a set of instructions for the computer to follow. These instructions are specifically crafted in a language that both the compiler and humans can understand. In iOS, the language you write in is Objective-C, and we used it extensively throughout the book. This appendix includes information about the language that will provide you with a better understanding of how iOS applications work.

A.1 Introduction to Objective-C

Objective-C is the primary language you use when writing iOS applications. It's based on the C programming language with a couple of additions that make it a complete object-oriented language. We said "primary" because given that Objective-C is a superset of C, writing C code is perfectly valid. In fact, the compiler that Xcode uses (LLVM since Xcode 3.2.3, GCC on earlier versions) is a C compiler modified to understand all of Objective-C's additions.

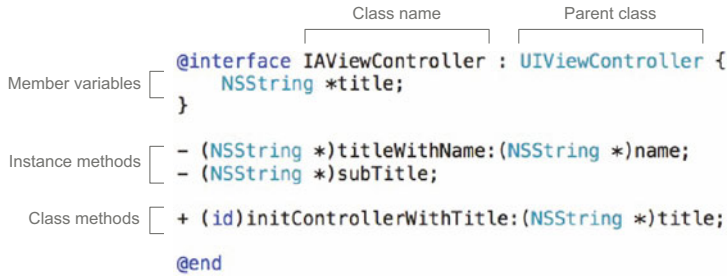


Figure A.1 Example of class declaration containing public variables, instance method, and class methods. This class is named `IAViewController` and inherits from `UIViewController`.

A.1.1 Class syntax

In a way, an application is an ecosystem of interconnected objects interacting with one another. In object-oriented programming (OOP) terms, an object is an instance of a class, and in Objective-C, classes are declared in two files:

1. The .h file, which contains the declarations of the class members (you call the declaration the *interface*) such as variables, methods, and properties
2. The .m file, which contains the implementation of the class

In figure A.1 you can see a class interface (.h file) and each part of the declaration. Member variables are set to protected by default, so if you want to access those variables from another class, you'll have to write accessor methods (see section A.1.3, "Properties," for more information about accessors). Figure A.2 shows the implementation side (.m file) of the class.

A.1.2 Message passing

When talking about Objective-C we usually refer to *message passing* instead of *calling functions*. Message passing seems very strange to developers coming from other languages, but its oddness fades quickly when you start working in the language. As you learned in the book, sending or passing messages to objects looks like this:

```
[object methodWithArgument:@"hey" andArgument:@"ho"];
```

Figure A.2 Class implementation (.m file). Inside the `@implementation - @end` block you'll write all your class logic.

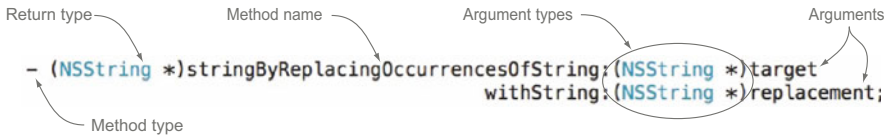


Figure A.3 Example of what an Objective-C method looks like. Note that the method's name and arguments are expressive and concise.

The square brackets introduce the fact that this statement is passing a message. We call the sentence on the left the *receiver*. In this case the receiver is `object`. Next, you see the arguments (if any). Objective-C is very explicit when naming arguments. You always send your arguments after a very explicit sentence. In the previous example, the first argument comes after `methodWithArgument:` and the second after `andArgument:`; this convention is useful for parsing the method purpose at first glance. Methods are perhaps the most common element of your programming interface, and as such you should take particular care in how you name them. The first portion of the method name should indicate the primary intent or result of calling the method, and the parameters should be named accordingly. We'll illustrate this with an example of a method you can find on NSString object instances; see figure A.3.

The name of the method is expressive and concise. You can easily see that it takes two strings as parameters, and just by reading the method you can tell what it does: It creates a new string by replacing all the occurrences of the string you're passing as the first argument with the string you're passing as the second argument.

There are two types of methods in Objective-C: class methods (represented by `+`) and instance methods (represented by `-`). A class method has no knowledge of the instance variables because it's not tied to an object instance. Class methods are mainly used for creating a new object, utility, or shared instance for a singleton class.

A.1.3 Properties

Data encapsulation is the mechanism whereby the implementation details of a class are kept hidden from outside the class. This is usually accomplished by creating getter and setter methods that take responsibility for these variables. In Objective-C you can use properties for that. The property declaration automatically declares getter/setter methods for us. It's a convenience notation used to replace the declaration and, optionally, implementation of accessor methods. This is something extensively used and very much recommended. Given that properties are always backed up by an instance variable (`ivar`) that's also created automatically, there are some memory management details of those variables that you need to understand. We'll talk about that in the next section, "Understanding automatic reference counting." Figure A.4 shows the properties declaration style.

As you can see, there are a couple of attributes you can set on the properties declaration. These attributes are defined as follows.

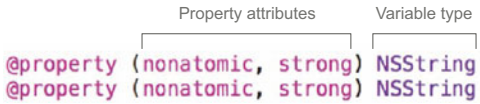


Figure A.4 Properties declaration style

WRITABILITY

You can define the property as read-only or read-write by using the keywords `readonly` and `readwrite`. Those keywords are mutually exclusive and allow you to control the availability of a setter. If you don't define this argument, the property is `readwrite` by default. Note that there's no way to make a write-only property. `Readonly` means that only the getter is created.

SETTER ATTRIBUTE

Properties allow you to specify an attribute that defines how memory is managed on the setter. The attributes that you can use are these:

- `strong`—Setting the property as `strong` tells the memory manager that the `ivar` that backs this property holds a strong reference to the object in question. In other words, the setter will make sure to hold the reference in memory until the instance is destroyed.
- `weak`—`Weak` means that the property doesn't become an owner of that object but just holds a reference to it. If the object's reference count drops to 0, even though you may still be pointing to it, it will be removed from memory.
- `copy`—`Copy` is required when the object is mutable. You use this when you need the value of the object as it is at this moment, and you don't want that value to reflect any changes made by other owners of the object.
- `assign`—`Assign` is the opposite of `copy`. When calling the getter of an assign property, it returns a reference to the actual data. Typically you use this attribute when you have a property of primitive type, for example, integers, floats, booleans, and the like.

CUSTOM ACCESSORS

With the keywords `setter` and `getter` you can control the name of the methods that are used for your getter and/or setter. These are specified with `getter=getterMethod` and `setter=setterMethod`.

ATOMICITY

Objective-C accessors sometimes have quite a bit of work to do, and this work isn't inherently thread-safe. When you declare a property as `atomic`, the setters and getters that are automatically created provide robust access to instance variables in a multithreaded environment. This means that before returning a value from the getter or setting a value via the setter, the compiler will make sure not to access the same variable at the same time from different threads. Setting the property as `nonatomic` tells the property to just set and get the value directly without any other consideration. Given that atomicity is not guaranteed, `nonatomic` is considerably faster than `atomic`. Note that even if `atomic` is the default value, it's more common to use `nonatomic` properties.

A.2 Using blocks

Blocks are an incredibly powerful addition to Objective-C, introduced in iOS 4. They're really nothing more than a chunk of code. What makes them unique is that a block can be executed inline as well as passed as an argument into methods and/or functions. They're also called *closures*, because they close around variables in a specific scope. Blocks look and operate much like C functions, and as such, they carry with them a rather obscure aspect of Standard C: function pointer declaration and casting syntax.

A.2.1 Block literals

When defining a block, you're actually creating a *block literal*. A literal is a value that can be built at compile time. For example, an integer literal is `3`, a string literal is `@"foobar"`. In other words, a literal is some data that is presented directly in the code, rather than indirectly through a variable. The term *literal* comes from the fact that you're writing data "literally" into your code (that is, exactly as written). You'll see why the fact that block declarations are literals is important later when we get into memory management.

Let's first see what a block literal looks like:

```
^(int a, int b)
{
    int powres = a ** b;
    return powres;
}
```

As you can see, it looks very similar to a C function body, except for these differences:

- There is a caret symbol preceding the function body.
- Return types are automatically inferred when not defined.
- There is no function name. We say that block literals are anonymous.

As with function and method definitions, the braces indicate the start and end of the block. Blocks can also take arguments and return values just like methods and functions. In a nutshell, block literals encapsulate a bunch of code the same way C functions do, but they also hold some really useful features, as you'll see in this section.

A.2.2 Block pointers

A block pointer isn't any different from an object pointer in the way that you can pass it to functions or create functions that return blocks. We'll illustrate how the assignment works by the following example:

```
int(^pow)(int, int) = ^(int a, int b)
{
    int powres = a ** b;
    return powres;
}
```

As you can see, the first word found is `int`, which is the return type of the block, followed by `(^pow)`. The caret symbol `^` replaces the star `*` that you usually use for declaring variable pointers. The meaning is the same, but by using the caret symbol

you're telling the compiler that instead of pointing to a value, you're pointing to a block of code. Immediately after the name of the block pointer you find `(int, int)`, which is the declaration of the block arguments. In this case, `(int, int)` means that the block is taking two arguments of type `int`. The right-hand side is the block literal that you saw a moment ago.

> **NOTE** The fact that blocks return something is confusing sometimes. As you'll see later on, the block returns a value that can be used by the caller of the block.

Another key difference between ordinary functions and blocks is that functions are defined in the global scope and blocks are defined in a local scope. In other words, blocks can be defined anywhere a variable can. The scope is very important when defining blocks because you can access variables from the same enclosing scope where the block is defined. Here's an example:

```
- (void)exampleMethod
{
    int variableInsideMethod = 10;

    void(^exampleBlock)(void) = ^(void){
        NSLog(@"We can access: %d", variableInsideMethod);
    };
}
```

As you can see, you can access the `variableInsideMethod` variable within your block.

A.2.3 Block invocation

So far you've learned how to declare blocks and how their pointers are assigned. It's important to understand that, so far, the code inside your blocks hasn't executed at all. You've created them, but you haven't used them. You'll see next how blocks are invoked:

```
int pow_result = pow(3, 4);
```

As you can see, block invocation has the exact same syntax as C function calls. Blocks return a specific type (or none) and take arguments the same way a C function does. In the previous example, `pow_result` contains an integer that is the resulting value of calculating 3 to the power of 4.

A.2.4 Common usage

A block represents a unit of executable code that can capture variables of the surrounding scope. This makes blocks ideal for asynchronous invocation; in fact, blocks are used extensively when writing asynchronous tasks. You'll see later that blocks are extremely handy when it comes to multithreading operations.

Other typical uses for blocks include the following:

- Running code when an asynchronous task is completed (for example, an HTTP request)
- Handling errors on asynchronous calls

- Handling asynchronous notifications
- As lambda functions, for iterations (sorting, enumeration, and the like)
- UIView animation and transitions

A.3 Optimizing applications with Grand Central Dispatch

Multithreading is treated as a black art by many programmers and for good reason. Even for an experienced programmer, writing and maintaining multithreaded code is hard and unpleasant. It's so difficult to do it right that it's usually recommended to avoid multithreading as much as possible.

The goal behind multithreading is very simple: perform more than one task at the same time. Take, for example, the iOS user interface operations: the tasks used to (re)draw the screen are all done in the main thread, that is, the same thread on which your program runs. While your logic is running, the user interface is not able to update, leading to a frozen screen. But that's not the only problem: touches also are dispatched on the main thread, and when you perform heavy operations on the main thread, such as I/O operations, complex computations, or synchronous network requests, you're preventing the operating system from catching these events. Alas, sometimes you don't have any other choice than using multithreading, but fortunately, Objective-C abstractions are way easier to use and understand.

Grand Central Dispatch

Grand Central Dispatch (GCD) is a technology added by Apple that allows concurrent executions just like multithreading does but without the complexity involved in writing multithreading code. It works by allowing specific tasks in a program to be queued for execution in parallel. You submit these tasks in the form of block objects. Blocks are queued in specific queues controlled by the operating system and then executed in parallel or in series. Let's look at an example of a queue invocation:

```
dispatch_async(
    dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
    ^{
        NSLog(@"This is executed in a different thread!");
    }
);
```

1 The first argument is the queue where you want to execute the asynchronous operation (see "Queue Priorities" for more information).

2 The second argument is the block of code that will be executed. Here you're just printing a message to standard output.

In order to execute asynchronous tasks you'll call `dispatch_async`, a function that takes two arguments: the queue in which you want to execute the task (these queues are automatically created for you) and the task itself.

Dispatch queues

iOS creates a series of dispatch queues for you to be used as replacements for multithreading operations, and as such, all low-level operations and multithreading handling are done by the operating system, so there is no guarantee as to which thread is going to execute your tasks. There are two types of queues:

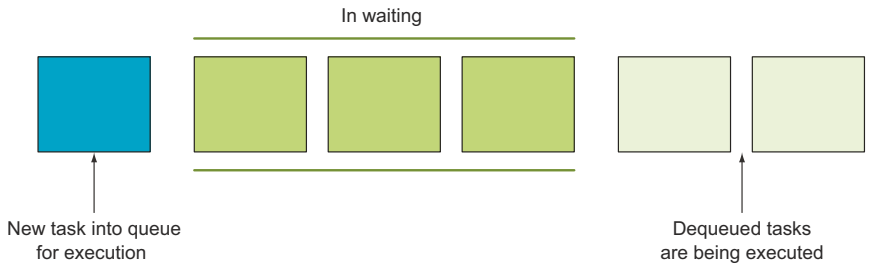


Figure A.5 An overview of how operation queues work on GCD. It's a traditional FIFO queue in which tasks are dequeued and executed, sometimes in parallel and sometimes in series according to execution type.

- *Concurrent*—Tasks are dequeued in First In, First Out (FIFO) order. As the name indicates, you use this kind of queue to execute tasks concurrently. The operating system automatically creates four concurrent dispatch queues for you with different priorities. Given the fact that tasks are executed concurrently, there's no guarantee as to the order in which tasks are finished.
- *Serial*—Tasks are dequeued in FIFO order, one at a time. The next task is dequeued only when the running one is finished. Serial queues guarantee you run only one task at a time. You use serial queues to ensure that tasks are executed in a predictable order. This is very useful when all tasks are related in a way and you need to be sure that each task is executed one after another.

Figure A.5 illustrates an abstraction of how dispatch queues work.

All global dispatch queues created by iOS that you'll see in the next section are concurrent. In order to use serial queues you'll have to create a new queue by using the function `dispatch_queue_create("queue name", 0)` and use this newly created queue as the first argument of each task you want to execute in serial mode.

QUEUE PRIORITIES

Queue priorities define the urgency of the tasks. Higher priorities will (hopefully) be executed sooner than lower priorities:

- `DISPATCH_QUEUE_PRIORITY_HIGH`—Tasks dispatched to the high-priority queue run at the highest priority. This means that the queue is scheduled for execution before any default-priority or low-priority queue.
- `DISPATCH_QUEUE_PRIORITY_DEFAULT`—Tasks dispatched to this queue run at the default priority; the queue is scheduled for execution after all high-priority queues have been scheduled but before any low priority queues have been scheduled.
- `DISPATCH_QUEUE_PRIORITY_LOW`—Tasks dispatched to this queue run at low priority; the queue is scheduled for execution after all high-priority and default-priority queues have been scheduled.
- `DISPATCH_QUEUE_PRIORITY_BACKGROUND`—Tasks dispatched to this queue run at background priority; the queue is scheduled for execution after all high-priority

queues have been scheduled, and the system runs items on a thread whose priority is set for background status. Such a thread has the lowest priority, and any disk I/O is throttled to minimize the impact on the system.

You can retrieve these queues by using `dispatch_get_global_queue(<queue name>, 0)`. One important queue that wasn't mentioned in this list is the main queue. The main queue is very useful when you need to run some specific tasks on the main thread, such as user interface operations. For example, if you want to update a label on screen, show a message, or update any view, you're forced to do it from the main thread because UI operations are not thread safe. For that, you'll use the exact same `queue_async()` function, and in order to retrieve the queue for its use, you'll call the function `dispatch_get_main_queue()`. This means that this code should run on the main thread:

```
dispatch_async(dispatch_get_main_queue(), ^{ NSLog(@"Main thread!"); });
```

A.4 Understanding automatic reference counting

Every program needs a way to write and read things from memory. When you create an instance of a class, or when you define a variable or mutate an object, you're allocating space in your device memory for later use. But because memory isn't infinite, you also need to release the memory you used when you don't need it anymore. The process of allocating and releasing memory is called memory management, and programming languages use different techniques to do it. Cocoa's memory management system is called *reference counting*. Each object keeps track of how many times it is being referenced and released. When the counter reaches zero, the object is marked as finished, and it's eventually removed. Figure A.6 shows how reference counting works with multiple objects. The rules are fairly simple. In short,

1. If you `alloc`, `new`, `copy`, or `retain` an object, you must balance that with `release` or `autorelease`.
2. If you obtain an object in some other way, and you need it to stay alive long-term, you must `retain` or `copy` it. This must, of course, be balanced later.

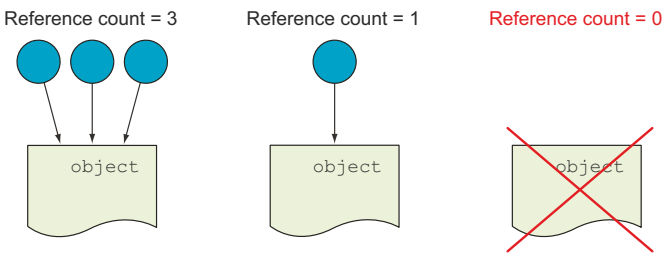


Figure A.6 This is an object with a reference count of 3, meaning that 3 classes asked for ownership of this object, then two classes released it, leaving a count of 1, and finally the last class released the object, subtracting the count by 1 and leaving the count as 0. At this point the object is removed from memory.

Memory management systems that delegate all the responsibility to developers usually create very complicated code and potential memory leaks (a memory leak is the repetitive allocation of memory without consequential release of it when no longer used). Before Xcode 4.2 that was the case with Cocoa's memory management system; developers were in charge of allocating memory as they needed it and releasing it when not used. Later on, with the introduction of LLVM and the Clang static analyzer, there was a way to catch all potential problems and see some tips on screen. But, you might ask, if Clang could spot these errors, why can't Clang just fix them for us? That's what Automatic Reference Counter (ARC) does. It's a layer on top of the compiler that inserts the needed calls to `retain` and `release` methods for you. Nothing really changed about how Objective-C manages your memory; ARC is a compiler-level feature that adds all those `retain/release` calls for you. Not only does ARC free you from writing the calls to `release/retain`, but it also *strictly prohibits* the use of these methods. In other words, ARC frees you from dealing with *most* memory management issues. But even though ARC makes your life easier, you can't ignore memory management completely. There are some cases where you still need to make some decisions about how memory is managed. We'll look at the most important ones now.

PROPERTIES AND ATTRIBUTES

You learned earlier about how properties are declared in Objective-C, as well as how to set properties through the `setter` attribute. These attributes define how memory is managed on the setter:

- `strong`
- `weak`
- `copy`
- `assign`

You can look back at section A.1.3, "Properties," for details about these attributes. It's important to understand that you must decide how your properties relate to other objects. You use these attributes to notify the compiler of these relationships. Strong references are a way of owning the variable, meaning that as long as the instance holding the property is alive, the variable will be as well. Weak references can be removed from memory at any time, and the instance has no control over that.

BLOCKS

If there's one thing to be careful about in ARC, it's the retain cycle. A retain cycle is a situation in which object A retains object B, and object B retains object A at the same time. This is a problem because object A won't be released from memory because it's still owned by object B, but object B won't ever be released either because it's still owned by object A.

Retain cycles are somewhat dangerous with blocks because all the variables from the surrounding scope you use inside the block will be retained; for example:

```
[self someMethodWithBlock:^{
    [self someOtherMethod];
}];
```

At this point, the block is held by the instance and the block itself is strongly holding `self`. This is a case of a retain cycle that will lead to a memory leak. The way to break the cycle in this case is to force `self` to be weak inside the block. In order to do that you must define a weak version of `self` before calling the block:

```
__weak id wself = self;
[self someMethodWithBlock:^{
    [wself someOtherMethod];
}];
```

You first create a variable with the `__weak` modifier called `wself`. This variable points to the actual instance, but thanks to the modifier you're telling the block not to strongly hold `self`.

A.5 Summary

This appendix has provided an introduction to Objective-C, which is the main programming language used to write apps for iOS. You learned how incredibly flexible and powerful Objective-C can be, and although an in-depth analysis of all the characteristics of Objective-C is beyond the scope of this book, we covered some key features that will make your life easier as an iOS developer. Some key topics we covered are these:

- When writing Objective-C you don't call functions; you send messages to objects.
- Properties create setters and getters for you, cutting down on the repetitive code you need to write.
- Blocks are pieces of code that you can save, execute, copy, or pass to other methods at runtime.
- Grand Central Dispatch can simplify concurrent operations, and it's a full replacement for traditional multithreading.
- There is a layer on the compiler that creates the memory management code you need. This mechanism is called Automatic Reference Counter.

index

Symbols

A

B

C

N

O

P

Q

R

S

T

U

V

W

X

iOS DEVELOPMENT

iOS 7 IN ACTION

Lim • Mac Donell

To develop great apps you need a deep knowledge of iOS. You also need a finely tuned sense of what motivates 500 million loyal iPhone and iPad users. iOS 7 introduces many new visual changes, as well as better multitasking, dynamic motion effects, and much more. This book helps you use those features in apps that will delight your users.

iOS 7 in Action is a hands-on guide that teaches you to create amazing native iOS apps. In it, you'll explore thoroughly explained examples that you can expand and reuse. If this is your first foray into mobile development, you'll get the skills you need to go from idea to app store. If you're already creating iOS apps, you'll pick up new techniques to hone your craft, and learn how to capitalize on new iOS 7 features.

What's Inside

- Native iOS 7 design and development
- Learn Core Data, AirPlay, Motion Effects, and more
- Create real-world apps using each core topic
- Use and create your own custom views
- Introduction and overview of Objective-C

This book assumes you're familiar with a language like C, C++, or Java. Prior experience with Objective-C and iOS is helpful.

Brendan Lim is a Y Combinator alum, the cofounder of Kicksend, and the author of *MacRuby in Action*.
Martin Conte Mac Donell, aka fz, is a veteran of several startups and an avid open source contributor.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/iOS7inAction

Free eBook
SEE INSERT

"A practical journey through the iOS 7 SDK."
—Stephen Wakely
Thomson Reuters

"A kickstart for newbs and a deft guide for experts."
—Mayur S. Patil
Clearlogy Solutions

"Mobile developer: don't you dare not read this book!"
—Ecil Teodoro, IBM

"The code examples are excellent and the methodology used is clear and concise."
—Gavin Whyte
Verify Data Pty Ltd

"Everything you need to know to ship an app, and more."
—Daniel Zajork
API Healthcare Corporation

ISBN 13: 978-1-617291-42-5
ISBN 10: 1-617291-42-0
54499
9 781617 291425

MANNING $44.99 / Can $47.99 [INCLUDING eBOOK]